
RC9000-10/RC8000

SW8585 Compiler Collection

Math./Stat. Library

Reference Manual

Keywords:

RC8000, RC9000-10, Mathematical, Statistical

Abstract:

This manual contains procedures and programs concerning the general problems in numerical and statistical analysis.

Date:

January 1989

PN: 991 11 288

Copyright © 1988, Regnecentralen a · s/RC Computer a · s
Printed by Regnecentralen a · s, Copenhagen

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

Table of Contents

Introduction.....	1
adapint.....	3
Bessel Functions - $I_n(x)$ and $K_n(x)$	9
Bessel Functions - $J_n(x)$ and $Y_n(x)$	15
beta.....	21
decompose; solve.....	29
eberlein.....	39
fft.....	65
fit.....	71
gamma.....	83
householder.....	89
invertsyl.....	109
jacobi.....	121
minimum.....	131
pzero.....	149
runge_kutta.....	165
solineq.....	187
solvesyl.....	197
zero1(x, F, a, b, eps).....	205
data survey, Part 1.....	215
recordinput, Part 1.....	241
polyrootc.....	273



Introduction

The purpose of the package is to supply the users with reliable and efficient procedures and programs concerning the general problems in numerical and statistical analysis.

The algorithms are available as external procedures or complete programs. The detailed formats are described below.

Format Of Procedure Description

The complete description of a procedure has the following sections:

1. Functions And Parameters

A short description of the type of problem the procedure solves; the procedure heading with complete specifications; a concise description of the parameters classified as *Call parameters*, *Return parameters*, *Call and Return Parameters*, or *Other parameters* (e.g., parameters used with Jensens's device).

2. Method

A detailed description of the mathematical or statistical method and of algorithmic subtleties with suitable references to literature.

3. Accuracy, Time and Storage Requirements

A summary of the available information on the numerical accuracy, the execution time, the core store and backing store requirement for the translated program, and the number of lines of the procedure text, including the last comment.

4. Test and Discussion

A comparison with other, similar procedures; a survey of the performed test runs and a few characteristic test results; a simple ALGOL program showing a typical application of the procedure; a few results from runs with this program. Suggestions for changes in the algorithm to meet special demands.

5. References

References to relevant literature, if any.

6. Algorithm

The complete text of the procedure, including the FP-commands, the ALGOL text, and the comments. This section may be omitted in case of very long procedures or procedures programmed in assembler.

The description may be incomplete, but it will at least contain section 1, *Function And Parameters*.

Classification

The programs are classified by means of a set of descriptors. Each algorithm has at least one descriptor from each of the three columns:

Mathematical	Complete descr	Algol procedure
Statistical	Incomplete -	Algol program
	In standard user's	Slang subroutine
	package	Slang program
		Fortran subprogram
		Fortran program

and one or more subject descriptors, like

Complex arithmetic	Matrix	Integration
Db. prec. arithmetic	Inversion	Multi
Special functions	Eigenvalues	Diff. equation
Bessel	Eigenvectors	---
Gamma	Linear equations	

adapint

real procedure adapint(a, b, f, x, delta, order)

1. Function and parameters.

Call parameters:

a, b	real values. The endpoints of the interval over which the integration is carried out. It is allowed to have $b < a$.
delta	real value. The permitted error relative to I_a .

Return parameter:

adapint	real procedure. The approximation of the integral of $f(x)$ obtained by the procedure.
---------	--

Other parameters:

f	real. The function $f(x)$ given as an expression in x . f and x are used as 'Jensen parameters'.
x	real. The independent variable used in the expression $f(x)$. x need not be initialized. Upon exit $x = \text{sign}(e - I_a \times \text{delta}) \times e$, where e is an estimate of the abs error. So $x > 0$ indicates a failure and $x \leq 0$ a success.

2. Method.

The real procedure adapint calculates the integral of a function $f(x)$ from a to b within a prescribed accuracy given by the parameter: delta. This is achieved by making further subdivisions of those subintervals, where the error is too large - and only of those. These subdivisions are stopped when the desired accuracy is obtained or when the number of subdivisions reaches its permitted upper bound. In all cases the procedure delivers on exit an approximation to the integral and an indication of success or failure.

The procedure is particularly useful when the function $f(x)$ exhibits an almost singular behaviour within the interval (a, b) like $1/\sqrt{x+1}$ over $(0, 1)$. etc. In such cases it is almost always possible to get through by a proper choice of the governing parameter: delta.

The method uses a 7-point formula, and all subdivisions are nondestructive (i.e. all function evaluations are used). The successive subdivisions are carried out so that the squares of the estimates of the absolute error in all subintervals finally obtained are uniformly distributed. The calculation is considered successful if $e \leq I_a \times \delta$, where $e = \text{abs}(\text{the estimate of the absolute error})$ and I_a is the integral of $\text{abs } f$ over (a, b) . I_a is computed by the procedure but not delivered as return value. Upon exit x is assigned the value $\text{sign}(e - I_a \times \delta) \times e$. So a success is indicated by $x = 0$ or $x = -e < 0$, and a failure by $x = e > 0$. Since I_a usually is not known in advance, it is necessary to have a realistic estimate of I_a before running the procedure, so that δ can be properly fixed. This estimate may be obtained either by an honest guess or by means of the procedure itself. It must be noticed that the estimate of the error ($=e$) made by the procedure usually is 10 - 100 times as large as the actual error.

Ex. The call `adapint(0,5,exp(x),x,n-4)` gives `adapint = 147.4131649` (true value = 147.413159...) and $x = -1.2 \times 10^{-3}$, success (actual abs error = 5.8×10^{-6}).

3. References.

The present algorithm is the result of many experiments made at Regnecentralen during the last years and is not described in the literature. For a similar algorithm, see:

- [1] H. O'Hara, and Francis J. Smith: The evaluation of definite integrals by interval subdivision. The Computer Journal, Vol 12.2 (May 1969), p. 179-182.

4. Algol procedure

```
adapint = set 4
adapint = algol index.no message.yes
```

external

```
real procedure adapint(a,b,f,x,delta); message adapint version 1.10.69;
value a,b,delta; real a,b,f,x,delta;
```

```
begin array A(1:60); boolean selection,ex; integer p,Sign;
  real e,fe,fa,fb,h,x1,x2,x4,x6,x7,f1,f2,f4,f6,f7,base,r,s,t,
  sa,sb,hmin,sum,eps,dev,dd;
  Sign:=sign(b-a); h:=abs(b-a); hmin:=h/18200; selection:=true;
  eps:=(6615/192Xdelta)X2/(if h=0 then 1 else h); sum:=dd:=0;
  x:=x4:=(a+b)/2;f4:=f; x:=a;fa:=f;x:=e:=b;fe:=fb:=f;
  x:=x2:=(a+x4)/2;f2:=f;x:=x6:=(b+x4)/2;f6:=f;
  s:=abs(4Xf4+fa+fb)X2; base:=sXh; p:=-3; goto TEST;
STORE:
  if abs(3Xfb-8Xf7+6Xf6-f4)<abs(3Xfa-8Xf1+6Xf2-f4) then
  begin
    A(p):=b;A(p+1):=fb;A(p+2):=f7;A(p+3):=f6;
    b:=e:=a;fb:=fe:=fa;a:=x4;fa:=f4;
    x6:=x1;f6:=f1;x4:=x2;f4:=f2;s:=2Xsa
  end
  else
  STORE2:
  begin
    A(p):=a;A(p+1):=fa;A(p+2):=f1;A(p+3):=f2;
    a:=x4;fa:=f4;x4:=x6;f4:=f6;x6:=x7;f6:=f7;s:=2Xsb
  end;
  x:=x2:=(a+x4)/2;f2:=f;
TEST:
  x:=x1:=(a+x2)/2;f1:=f;x:=x7:=(b+x6)/2;f7:=f;h:=abs(b-a);
  sa:=abs(4Xf2+fa+f4);sb:=abs(4Xf6+fb+f4);base:=(sa-s+sb)Xh+base;
  r:=f2+f6;s:=f1+f7;t:=fa+fb;
  dev:=(84Xr-64Xs+15Xt-70Xf4)X2Xh; ex:=dev>baseX2Xeps;
  if (h-hmin)X9>abs x4\p<56\ex then
  begin
    p:=p+4; goto if selection then STORE else STORE2
  end;
  sum:=sum+(2016Xr+2048Xs+549Xt+4004Xf4)Xh;
  dd:=devXh+dd; if ex then eps:=(16Xeps+dev/baseX2)/4;
  if p>0 then
  begin
    selection:=false;
    t:=A(p); if (t-a)X(a-e)<0 then
```

```
begin
  b:=e; fb:=fe; e:=a; fe:=fa
end
  else
begin
  b:=a;fb:=fa
end;
a:=t; fa:=A(p+1); f2:=A(p+2); f4:=A(p+3);
x4:=(a+b)/2; x:=x6:=(x4+b)/2; f6:=f; x2:=(a+x4)/2;
s:=abs(4xf4+fa+fb)x2; p:=p-4; goto TEST
end from STORE;
adapint:=Signxsum/13230; dd:=16/6615xsqrt(dd);
x:=if dd>deltaXbase/12 then dd else -dd
end adapint;

end
```

Bessel Functions - $I_n(x)$ and $K_n(x)$

1. Function and Parameters.

besselik calculates the modified Bessel functions:
 $I_0(x), \dots, I_n(x)$ and $K_0(x), \dots, K_n(x)$.

Procedure heading:

```
procedure besselik (n,x,I,K);
value n,x; real x; integer n;
array I,K;
```

Call parameters:

```
n      : (real or integer) (real is rounded to nearest integer)
          maximum order of the Bessel functions.
          n must be  $\geq 0$ .
x      : (real or integer) the argument, must be  $> 0$ .
```

Return parameters:

```
I      : (real array I(0:n) )
          the values of the calculated functions:
           $I(0)=I_0(x), \dots, I(n)=I_n(x)$ .
K      : (real array K(0:n) )
          the values of the calculated functions:
           $K(0)=K_0(x), \dots, K(n)=K_n(x)$ .
```

2. Method.

First besselik calculates all the values of $I_j(x)$, see [1].

The recurrence is performed from an upper bound nb. If $x+14 \leq n+3$ this integer is set equal $n+4$ otherwise $x+14$.

Then $i(nb, x)$ is assigned the value of $(x/2)^{nb} / (1 \times 2 \times \dots \times nb)$,
 while $i(nb+1, x), \dots, i(n, x)$ is set to 0.
 $i(nb-1, x), \dots, i(0, x)$ are then computed from the recurrence formula

$$i(j-1, x) = 2xj/x \cdot i(j, x) + i(j+1, x).$$

But since $I_j(x)/i(j, x)$ is the same number for all $j \leq nb$, $I_j(x)$ can be calculated from the formula

$$\exp(x) = I_0(x) + 2x \sum_{k=1}^{nb} I_k(x)$$

by replacing x by $\text{abs } x$.

Then $K_0(x)$ and $K_1(x)$ are calculated by polynomial approximation see [2]:

```
if  $0 < x < 2$  then
   $K_0(x) := P_1((x/2) \times x^2) - \ln(x/2) \times I_0(x)$ ;
   $K_1(x) := P_2((x/2) \times x^2) / x + \ln(x/2) \times I_1(x)$ ;
else
   $K_0(x) := P_3(2/x) / \exp(x) / \sqrt{x}$ ;
   $K_1(x) := P_4(2/x) / \exp(x) / \sqrt{x}$ ;
where  $P_i(x)$  is a polynomial of 6. degree.
```

Further values of $K_j(x)$ are calculated by the recurrence formula:

$$K[i+1] = 2 \times i / x \times K(i) + K(i-1).$$

3. Accuracy, Time and Storage Requirement.

Accuracy: relative error $< 10^{-7}$
 Time: approx. $10 + 0.5 \times n$ ms
 e.g. $x=2$, $n=10$: 11 ms
 $x=10$, $n=40$: 29 ms
 Storage requirement: 3 segments of program and 8 local real variables.
 Typographical length: 87 lines incl. last comment.

4. Test and Discussion.

The algorithm is similar to the GIER procedure O.No. 179. [3].

Below program gives the following output:

```
begin
comment here the procedure is copied unless it is already
translated as an external;

integer i,n; real x;
write(out,<:<12<10> n x:>,false add 32,13,<:I(n):>,
false add 32,18,<:K(n)<10>:>);
AGAIN:
read(in,n); if n=-1 then goto END;
begin array K,I(0:n);
read(in,x);
besselik(n,x,I,K);
write(out,<:<10>:>,<<dd>,n,<<dd.dd>,x,
<< -d.ddddd ddddd10-dd>,I(n),K(n) );
goto AGAIN;
end inner block;
END:
end

data:
0, 0.01
0, 0.5
0, 5
1, 5
10, 5
20, 5
-1,
```

n	x	I(n)	K(n)
0	0.01	1.00002 50003 ₁₀ 0	4.72124 47360 ₁₀ 0
0	0.50	1.06348 33708 ₁₀ 0	9.24419 07256 ₁₀ -1
0	5.00	2.72398 71829 ₁₀ 1	3.69109 83816 ₁₀ -3
1	5.00	2.43356 42146 ₁₀ 1	4.04461 33826 ₁₀ -3
10	5.00	4.58004 44196 ₁₀ -3	9.75856 28020 ₁₀ 0
20	5.00	5.02423 93598 ₁₀ -11	4.82700 05078 ₁₀ 8

5. References.

- [1] Goldstein and Thaler: Recurrence Techniques for the Calculation of Bessel Functions, MTAC 13 (1959), p. 102.
- [2] Allen, E.E.: Polynomial Approximations to some modified Bessel Functions, MTAC Vol. 10, 1956, p. 162-164.
- [3] Zachariassen, J.: Bessel I and K, Algol procedure, Regnecentralen, April 1964, GEL O.No. 179.
- [4] British Association Mathematical Tables, Vol. VI, Bessel Functions, zero and unity, Cambridge University Press, 1958.
- [5] British Association Mathematical Tables, Vol. X Bessel Functions, order 2 to 20, Cambridge University Press, 1952.

6. Algorithm.

```

besselik=set 3
besselik=algol
external

procedure besselik (n,x,I,K);
value n,x; real x; integer n;
array I,K;

begin integer i,nb,m;
real a,j0,j1,j2,sum,xhalf;
  m:=n;
  nb:=x+14;
  if nb<=n+3 then nb:=n+4;
  xhalf:=x/2;
  if x<=10-150 then nb:=0
  else
  begin
    j1:=1;
    i:=0;
    for i:=i+1 while j1>10-150/i<=nb do j1:=j1*xxhalf/i;
    nb:=i-1
  end;
  comment nb is the upper bound for recurrence;
  if nb<=n then
  begin
    for i:=nb+1 step 1 until n do I(i):=0;
    m:=nb
  end;
  sum:=j2:=0;
  for i:=nb step -1 until 1 do
  begin
    if i<=m then I(i):=j1;
    j0:=i/xhalf*j1 + j2;
    sum:=sum+j0;
    j2:=j1; j1:=j0
  end recurrence loop;

```

```

sum:=exp(x)/(2*sum-j0);
a:=I(0):=j0*sum;
j2:=j2*sum; if n>0 then I(1):=j2;
for i:=n step -1 until 2 do I(i):=I(i)*sum;
comment all values of I0(x),...,In(x) are calculated;
if xhalf<1 then
begin
  j0:=xhalf*x2;
  j1:=ln(x)-.693147181;
  a:=K(0):=((((( .00000740 *j0 + .00010750)*j0
                + .00262698)*j0 + .03488590)*j0
                + .23069756)*j0 + .42278420)*j0
                -.57721566 - j1*a;
  if n>0 then
    j2:=K(1):(((((-.00004686 *j0 - .00110404)*j0
                  -.01919402)*j0 - .18156897)*j0
                  -.67278579)*j0 + .15443144)*j0
                  + 1 )/x + j1*j2
end
else
begin
  j0:=1/xhalf;
  j1:=sqrt(x)*exp(x);
  a:=K(0):((((((( .00053208 *j0 - .00251540)*j0
                  +.00587872)*j0 - .01062446)*j0
                  +.02189568)*j0 - .07832358)*j0
                  +1.25331414)/j1;
  if n>0 then
    j2:=K(1):((((((- .00068245 *j0 + .00325614)*j0
                  -.00780353)*j0 + .01504268)*j0
                  -.03655620)*j0 + .23498619)*j0
                  +1.25331414)/j1
end calculating K0(x) and K1(x) by
  polynomial approximation;
for i:=2 step 1 until n do
begin
  sum:=K(i):=a+(i-1)/xhalf*j2;
  a:=j2; j2:=sum
end recurrence loop
end besselik;

comment
  besselik calculates the modified bessel functions:
  I0(x),...,In(x) and K0(x),...,Kn(x).

Call parameters:
  n      : (real or integer)
           maximum order of the Bessel functions.
           n must be >=0.
  x      : (real or integer) the argument, must be > 0.

Return parameters:
  I      : (real array I(0:n) )
           the values of the calculated functions:
           I(0)=I0(x),...,I(n)=In(x).
  K      : (real array K(0:n) )
           the values of the calculated functions:
           K(0)=K0(x),...,K(n)=Kn(x);

```

Bessel Functions - $J_n(x)$ and $Y_n(x)$

1. Function and Parameters.

besseljy calculates the Bessel functions of first kind $J_0(x), J_1(x), \dots, J_n(x)$ and the Bessel functions of second kind $Y_0(x), Y_1(x), \dots, Y_n(x)$ of integer order and with real argument.

Procedure heading:

```
procedure besseljy (n,x,J,Y);
value n,x; integer n; real x; array J,Y;
```

Call Parameters:

n (integer or real) (real is rounded to nearest integer)
maximum order of the Bessel functions.
n must be ≥ 0 .
x (integer or real) the argument, must be < 0 .

Return Parameters:

J (real array J(0:n))
the calculated values of the functions
 $J(0)=J_0(x), \dots, J(n)=J_n(x)$.
Y (real array Y(0:n))
the calculated values of the functions
 $Y(0)=Y_0(x), \dots, Y(n)=Y_n(x)$.

2. Method.

First the functions $J_i(x)$ are calculated.

For $\text{abs } x \leq 10^{-5}$ the procedure uses a truncated power expansion

$$J(i)(x) = (x/2)^i \times i! / i!$$

for $i = 0, 1, \dots, n$.

For $\text{abs } x > 10^{-5}$ the values are found by recurrence, see [1]

$$j(i-1)(x) = 2xi/x \times j(i)(x) - j(i+1)(x)$$

for $i = nb-2, nb-3, \dots, 1$, where $j(nb)(x)=0$ and $j(nb-1)(x)=10^{-150}$.
nb is an even integer found as a function of x and n, within the limits given in [2].

After recurrence $J(i)(x)$ is found as

$$J(i) = j(i)(x) / (j(0)(x) + 2 \times \sum_{m=1}^{nb/2} j(2m)(x))$$

for $i = 0, 1, \dots, n$.

Then $Y_0(x)$ and $Y_1(x)$ are found from the summation theorems, see [1]:

$$Y_0(x) = 2/\pi \times ((\gamma + \ln(\text{abs}(x/2))) \times J_0(x) - 4 \times \sum_{p=1}^{nb/2} ((-1)^p \times p / (2^p) \times J(2^p)(x)))$$

$$Y_1(x) = 2/\pi \times (-1/x \times J_0(x) + (\ln(\text{abs}(x/2)) + \gamma - 1) \times J_1(x) + \sum_{o=3}^{nb/2} ((-1)^o \times ((a+1)/2) \times 4^o \times J_o((o \times 2 - 1) \times J_0(x)))$$

for odd values of o . The upper bound $nb/2$ is a substitute for infinity, see [2].

3. Accuracy, Time and Storage Requirements.

Accuracy: relative error $< 10^{-7}$
 Time: if $\text{abs } x \leq 10^{-5}$: approx. 5 ms
 else approx. $10 + 0.7 \times n$ ms.
 e.g. $x=2$, $n=20$: 17 ms
 $x=10$, $n=40$: 28 ms
 Storage requirements: 2 segments of program and 8 local real variables.
 (during translation 3 segments).
 Typographical length: 75 lines incl. last comment.

4. Test and discussion.

The algorithm is similar to the GIER procedure O.No. 208, [3].

Below program gives the following output:

```
begin

comment here the procedure is copied unless it is already
translated as an external;

integer i,n; real x;
write(out,<:12<10> n    x :>,false add 32,13,<:J(n):>,
      false add 32,18,<:Y(n)<10>:>);
AGAIN:
read(in,n); if n=-1 then goto END;
begin array J,Y(0:n);
read(in,x);
besseljy(n,x,J,Y);
write(out,<:10>:>,<<dd>,n,<<dd.ddd>,x,
      << -d.ddd dd-dd>,J(n),Y(n) );
goto AGAIN;
end inner block;
END:
end

data:
0, 0.001
0, 0.5
0, 5
1, 5
10, 5
20, 5
-1,
```

besseljy - 3 -

n	x	J(n)		Y(n)	
0	0.001	9.99999	75004 ₁₀ -1	-4.47141	66116 ₁₀ 0
0	0.500	9.38469	80724 ₁₀ -1	-4.44518	73352 ₁₀ -1
0	5.000	-1.77596	77133 ₁₀ -1	-3.08517	62526 ₁₀ -1
1	5.000	-3.27579	13760 ₁₀ -1	1.47863	14342 ₁₀ -1
10	5.000	1.46780	26472 ₁₀ -3	-2.51291	10098 ₁₀ 1
20	5.000	2.77033	00515 ₁₀ -11	-5.93396	52968 ₁₀ 8

5. References.

- [1] Goldstein and Thaler: Recurrence Techniques for the Calculation of Bessel Functions, MTAC 13 (1959), p.102.
- [2] Randels and Reeves: Notes on Empirical bounds for Generating Bessel Functions. Comm. ACM, v.1, No. 5, 3.
- [3] Tove Asmussen: Bessel J and Y, Gier System Library, O.No. 208, Regnecentralen, April 1964.
- [4] British Association Mathematical Tables, Vol. VI, Bessel Functions, zero and unity, Cambridge University Press, 1958.
- [5] British Association Mathematical Tables, Vol. X Bessel Functions, order 2 to 20, Cambridge University Press, 1952.

6. Algorithm.

```
besseljy:=set 3
besseljy:=algol
external
```

```
procedure besseljy (n,x,J,Y);
value n,x; integer n; real x; array J,Y;
begin integer a,nb,N;
real j0,j1,sum,y0,y1,y2;
boolean even;
sum:=abs x;
x:=j1:=x/2;
y0:=y1:=0;
y2:=ln(sum)-0.1159315156584;
if sum<=10-5 then
begin
J(0):=sum:=j0:=1;
for nb:=1 step 1 until n do J(nb):=J(nb-1)*x/nb
end
else
```

```

begin
  N:=n+1;
  if n>10 then
    begin
      for N:=N-1 while (sum/N)×N<10-100 do J(N):=0;
      N:=N+1
    end;
  nb:=0.525×sum+15;
  nb:=2×(if nb <=N//2 then N//2+1 else nb);
  j1:=10-150;
  j0:=sum:=0;
  even:=false;
  a:=(-1)×((nb-2)//2);
  for nb:=nb-1 step -1 until 2 do
    begin
      if nb<N then J(nb):=if even then j0 else j1;
      if even then
        begin j1:=nb/x×j0-j1; y0:=a/nb×j0+y0 end
      else
        begin
          j0:=nb/x×j1-j0;
          y1:=a×nb/(nb×2-1)×j1+y1;
          a:=-a;
          sum:=sum+j0
        end;
      even:=¬even
    end;
  j0:=j1/x-j0;
  sum:=2×sum+j0;
  J(0):=j0/sum;
  if n>0 then J(1):=j1/sum;
  for nb:=N-1 step -1 until 2 do J(nb):=J(nb)/sum
end;
Y(0):=y0:=0.63661977236758×(4×y0+y2×j0)/sum;
if n>0 then Y(1):=y1:=0.63661977236758×
  (-j0/x/2+(y2-1)×j1-y1×4)/sum;
for nb:=2 step 1 until n do
  begin
    Y(nb):=y2:=(nb-1)/x×y1-y0;
    y0:=y1; y1:=y2
  end
end
end besseljy;

```

comment

besseljy calculates the Bessel functions of first kind $J_0(x), J_1(x), \dots, J_n(x)$ and the Bessel functions of second kind $Y_0(x), Y_1(x), \dots, Y_n(x)$ of integer order and with real argument.

Call Parameters:

n (integer or real)
maximum order of the Bessel functions.
n must be ≥ 0 .
x (integer or real) the argument, must be > 0 .

Return Parameters:

J (real array J(0:n))
the calculated values of the functions
 $J(0)=J_0(x), \dots, J(n)=J_n(x)$.
Y (real array Y(0:n))
the calculated values of the functions
 $Y(0)=Y_0(x), \dots, Y(n)=Y_n(x)$;

beta

beta - 1 -

Beta function, beta(x,y)

1. Function and parameters.

beta(x,y) approximates the beta function.

beta(x,y) = integral from 0 till 1 of $(1-t)^{x-1} t^{y-1} dt$

procedure heading:

```
real procedure beta(x,y);  
value x,y; real x,y;
```

procedure identifier:

```
beta      : (real)  
           approximate function of arguments not resulting  
           in under - or overflow, in which case beta is  
           undefined.
```

call parameters:

```
x,y      : (real or integer)  
           arguments.
```

2. Method.

The value of beta(x,y) is calculated in the range $1 \leq x \leq 2$, $1 \leq y \leq 2$ by means of the formula

beta(x,y) = $\frac{\Gamma(x+1) \times \Gamma(y+1)}{x/y \times \Gamma(x+y)}$ or
beta(x,y) = $\frac{\Gamma(x+1) \times \Gamma(y+1)}{x/y \times (x+y-1) \times \Gamma(x+y-1)}$

according to whether $x+y \leq 3$ or $x+y > 3$.

The value of gamma(z) is approximated in the range $2 \leq z \leq 3$ by a rational function of $z-2$, which is given as approximation 5231 in reference (1).

For arguments outside the range $1 \leq x \leq 2$, $1 \leq y \leq 2$, reductions are performed according to the formula:

beta(x+1,y) = $x/(x+y) \times \text{beta}(x,y)$

3. Accuracy and time requirement.

The maximum relative error will be about

$$\max(1, (\text{abs}(x) + \text{abs}(y)) \times 10^{-10})$$

The c.p.u.-time used for a call of beta is crudely

$$5 + 0.1 \times (\text{abs}(x) + \text{abs}(y)) \text{ milliseconds.}$$

4. Test.

testprogram and output:

```
begin
  real b,x,y;
  for overflows:=0 while read(in,x,y)=2
  do
    begin
      b:=beta(x,y);
      write(out,<:<10>x=:>,<<-ddd.d>,x,<: y=:>,
            y,<: beta(x,y)=:>,<<dddddddddd0000-ddd>,b,
            << dd-dd>,abs(b-gamma(x)*gamma(y)/gamma(x+y))
            );
      setposition(out,0,0);
    end;
  end;
```

x=	0.5	y=	0.5	beta(x,y)=	3141592653760 ₁₀	-12	0 ₁₀	0
x=	1.0	y=	1.0	beta(x,y)=	1000000000000 ₁₀	-12	58 ₁₀	-12
x=	100.0	y=	1.0	beta(x,y)=	100000000004000 ₁₀	-16	45 ₁₀	-14
x=	10.0	y=	-0.5	beta(x,y)=	-10783381324400 ₁₀	-12	0 ₁₀	0

beta-3-

5. Algol procedure.

```
beta=set 2
beta=algol
external
real procedure beta(x,y);
value x,y;
real x,y;

begin
  real h,w;
  for w:=0,x
  do
    begin
      if w=0
      then h:=1
      else
      begin
        x:=y;
        y:=w
      end
    ;
    if x>2
    then
      begin
        for x:=x-1 step -1 until 1
        do h:=h*x/(x+y);
        x:=x+1
      end
    else
      if x<1
      then
        for x:=x step 1 until 1
        do h:=h*x(x+y)/x
      end
    ;
    w:=x+y-1;
    if w>2
    then
```

```

begin
  h:=h/(wxxxxy);
  w:=w-2
end
else
begin
  h:=h/(xxy);
  w:=w-1
end
;
for w:=((((
  .039301346419 xw
  +.142928007949)xw
  +1.09850630453 )xw
  +3.36954359131 )xw
  +12.8021698112 )xw
  +22.9680800836 )xw
  +43.9410209189 )
  /
  (((      w
  -7.15075063299)xw
  +4.39050474596)xw
  +43.9410209191 )
  while y>0
do
begin
  if x>0
  then
  begin
    h:=h/w;
    w:=x-1;
    x:=0
  end
  else
  begin
    h:=hXw;
    w:=y-1;
    y:=0
  end
end
end

```

beta-5-

```
;
beta:=hXw
end beta;
```

6. Reference.

- (1) J.F. Hart and oth.:
Computer Approximations,
John Wiley and Sons, 1968, p. 130-136

**decompose
solve**

1. Function and Parameters.

1.1 Decompose:

Decompose calculates upper and lower triangular matrices u and l such that $l*u=a$, where a is a given $n*n$ square matrix. With the additional requirement $u(i,i)=1$, the decomposition is unique if a is non-singular. In order to ensure numerical stability, row-exchanges are performed (explicitly) and information about these exchanges is stored for further use in subsequent procedures handling the decomposed matrix.

Implied procedure head:

```
boolean procedure decompose(a,p,mode);  
value mode;  
array a;  
integer array p;  
integer mode;
```

Call parameter:

mode : (integer or real). This parameter governs the precision in the calculation of the inner-products in the algorithm:
mode=0 : The inner-products are calculated in normal floating point mode.
mode=1 : The inner-products are calculated by means of intermediate variables of 45 bits mantissa and 24 bits exponent.

Call and Return Parameter:

a : (real array or zone record with $n*n$ elements). Contains at entry the square matrix to be decomposed. On exit, each element of a is replaced by the corresponding element of u or l . (The diagonal of u is not stored). In case of a one-dimensional array or a record, the elements of a must be stored row-wise.

Return Parameters:

decompose : (boolean). True if the matrix a is non-singular, otherwise false.
p : (integer array with n elements). Contains information about the row-exchanges. (see section 2.Method).

1.2 Solve:

Solve calculates the solution-vector x to the system of equations $a \cdot x = b$, where a is a $n \times n$ square matrix, decomposed by a previous call of `decompose`, and where b is a column vector containing the given righthand side. Thus, the solution of several systems of equations with the same matrix of coefficients requires one call of `decompose` followed by a number of calls of `solve`.

Implied procedure head:

```
procedure solve(a,p,mode,b);  
value mode;  
array a,b;  
integer array p;  
integer mode;
```

Call Parameters:

mode	: (real or integer). cf. <code>decompose</code> .
a	: (real array or zone record with $n \times n$ elements). Contains the decomposed coefficient-matrix as produced by <code>decompose</code> .
p	: (integer array with n elements). Contains information on the rowexchanges of the matrices held in a .

Call and Return Parameter:

b	: (real array or zone record with n elements). Contains on entry the given right-hand side. On exit, the corresponding solutions are stored in b .
---	--

1.3 Parameter-check.

In case of wrong parameters the run is terminated with an error message on current output consisting of the procedure name (`decomp` or `solve`) and a number, indicating the wrong parameter as follows:

- 1: The number of elements of a is different from $n \times 2$ (n being the number of elements of p).
- 2: Wrong content of p (`solve` only). Indicates an impossible row-exchange or an attempt to solve a singular system of equations.
- 3: $mode < 0$ or $mode > 1$.
- 4: The number of elements of b is different from n (`solve` only).

2. Method.

Decompose produces the triangular matrices l and u in n steps, in the k -th of which the k -th column of l and the k -th row of u ($0 \leq k \leq n-1$) are calculated by

$$(2.1) \quad l: \quad a(j,k) := a(j,k) - \sum_{i=0}^{k-1} a(j,i) * a(i,k) \quad ; \quad j := k, k+1, \dots, n-1$$

$$(2.2) \quad u: \quad a(k,j) := (a(k,j) - \sum_{i=0}^{k-1} a(i,j) * a(k,i)) / a(k,k); \quad j := k+1, k+2, \dots, n-1$$

During the calculation of the elements of l , the k -th pivotal index, piv , is found using the criterion

$$\text{abs } a(j,k) / 2^{**\text{ex}(j)} = \text{maximum with respect to } j$$

where $\text{ex}(j)$ is the initial maximum exponent of the numbers constituting the j -th row.

This pivotal strategy is chosen on two counts: It is simple, and none is known to be universally better (cf. [1]).

If all elements of the column of l turns out to be (exactly) zero, $p(1)$ is set equal to 2048, and the procedure exits with the value false.

Otherwise, $p(k)$ is set to the pivotal index, piv , and if piv is greater than k , $\text{ex}(piv)$ is set to $\text{ex}(k)$ and the k -th and the piv -th row of a are exchanged before the elements of u are calculated.

Solve proceeds in two steps: First, the equations

$$l * y = b$$

is solved for y , exchanging the elements of b as described by p , after which the final solution x is found by solving

$$u * x = y$$

Here, b is successively replaced by y and x . The formulae used are analogous to those of (2.1)-(2.2):

$$(2.3) \quad l: \quad b(k) := (b(k) - \sum_{i=0}^{k-1} b(i) * a(k,i)) / a(k,k) \quad ; \quad k := 0, 1, \dots, n-1$$

$$(2.4) \quad u: \quad b(k) := b(k) - \sum_{i=k+1}^{n-1} b(i) * a(k,i) \quad ; \quad k := n-2, n-3, \dots, 1, 0$$

During the first step, it is checked that $n > p(k) \geq k$. If this check fails, the run is terminated as described in section 1.

If the value of the parameter mode is 1, the inner-products of (2.1)-(2.4), i.e. expressions of the form

$$-(\text{sum } r(i) * s(i) + r(k) * (-1))$$

are calculated by retaining 45 bits of each product and adding this to a sum of 45 significant bits. (The exponent is kept in 24 bits).

Thus, instead of the rounding errors in each multiplication and addition, introduced by the normal floating-point operations, an error is introduced only in the final rounding of the sum to a floating-point number. However, it should be noted that only to a certain extent this procedure can cope with a severe cancellation of significant bits that may arise when a product is added to the sum.

The following peculiarities, due to the fact that the procedures are written in the assembler language SLANG 3, should be mentioned:

- a) The error message constituents lin.eq.1 and lin.eq.2 occur in these messages instead of ext<line number>. The possibilities are:

lin.eq.1 : Overflow/underflow in calculations outside the inner-product procedure.

lin.eq.2 : a) Overflow/underflow in the inner-product procedure.
(If mode=1, this can happen only in the final rounding to a floating-point number).

- b) The parameter errors as described in section 1.3

Some examples are shown in section 4.

- b) The formal parameter p contains as explained the pivotal indices; however, the k-th index is not found in p(k) (i.e. the word number k of p), but in the k-th byte of p. A possible way of unpacking these indices is shown in the program in section 4.

The remaining bytes of p are used for the exponents ex(k).

- c) As stated implicitly in section 1, the index bounds and the number of indices of the actual array-parameters are irrelevant. Only the number of elements in the declaration is taken into consideration.

3. Accuracy, Time and Storage Requirements.

Accuracy: Depends on the problem and on the choice of the parameter mode.

The table below shows the median-error (in units of 10^{-10}) of 11 sets of equations, consisting of equally distributed random numbers $(-10^{20} | 10^{20})$. The error is expressed as the residual norm relative to the norm of the right-hand side. (The Euclidian norm is used).

order	error mode=0	error mode=1
10	2.6	1.9
20	3.9	2.3
30	7.4	4.0
40	9.5	5.0
50	27	8.3
60	21	7.6
70	33	8.7

Time: Based on recorded solution-times for the systems mentioned above, the following execution-times in msec., expressed as functions of the order, holds within +10 pct. for orders between 50 and 100:

	mode=0	mode=1
decompose	$0.02 * (1 + 10/n) * n^{**3}$	$0.08 * (1 + 5/n) * n^{**3}$
solve	$0.07 * n^{**2}$	$0.3 * n^{**2}$

Storage Requirements:

2 segments of program
0 variables.

4. Test and Discussion.

As may be expected, the results obtained for mode=1 are significantly better than those for mode=0 only if n is sufficiently large. On the other hand, if the system is ill-conditioned, the results can be widely different even for small n. As an example, the system

	10	7	8	7		32
	7	5	6	5		23
a:	8	6	10	9	b:	33
	7	5	9	10		31

the exact solution of which is (1,1,1,1), yields the results:

mode=0:

decomposed matrix:

```
7.0000000000! 0 7.1428571432! -1 8.5714285716! -1 7.1428571432! -1
8.0000000000! 0 2.8571428570! -1 1.1000000002! 1 1.1500000002! 1
7.0000000000! 0 0.0000000000! 0 3.0000000001! 0 1.6666666667! 0
1.0000000000! 1 -1.4285714296! -1 1.0000000014! 0 -1.6666666791! -1
```

piv	ex	solutions
1	4	1.0000000459! 0
2	4	9.9999992456! -1
3	4	1.0000000186! 0
3	4	9.9999998884! -1

mode=1:

decomposed matrix:

```
7.0000000000! 0 7.1428571432! -1 8.5714285716! -1 7.1428571432! -1
8.0000000000! 0 2.8571428570! -1 1.1000000002! 1 1.1500000002! 1
7.0000000000! 0 -2.9103830457! -11 3.0000000002! 0 1.6666666665! 0
1.0000000000! 1 -1.4285714290! -1 1.0000000009! 0 -1.6666666733! -1
```

piv	ex	solutions
1	4	1.0000000082! 0
2	4	9.9999998644! -1
3	4	1.0000000034! 0
3	4	9.9999999800! -1

The Euclidian error-norm is $9.1 \cdot 10^{-8}$, $1.6 \cdot 10^{-8}$ respectively.

The following program was used

```
lin.eq. test parameter error etc.
begin integer d1,d2,d3,d4,mode;
  underflows:=-1;
  read(in,d1,d2,d3,d4,mode);
  begin array a(1:d1), b(d2:d3);
    integer array p(1:d4). piv(1:2*d4);
    integer i,j,k;
    read(in,a,b);
    if -,decompose(a,p,mode) then write(out,<:<10>sing:>);
    write(out,<:<10>decomposed matrix:>);
    k:=1;
    for i:=1 step 1 until d4 do
      begin write(out,<:<10>:>);
        for j:=1 step 1 until d4 do
          begin write(out,<< -d.ddddddddd!-dd>,a(k));
            k:=k+1
          end;
        end;
```

```

      j:=p(i);
      piv(2*i-1):=j shift (-12) extract 12;
      piv(2*i):=j extract 12;
    end;
    solve(a,p,mode,b);
    write(out,<:<10><10>      piv    ex    solutions:>);
    for i:=1 step 1 until d4 do
      write(out,<:<10>>,<< dddd>,piv(i),piv(i+d4),
        << -d.ddddddddddd'-dd>,b(i+d2-1))
    end block
  end

```

This program produces the error-messages shown below when the input is

4, 1, 2, 2, 0, a, 1, 1,

where a means the four elements of a 2*2 matrix:

I) a: 1, 2, 1, 2

```

      solve      2  lin. eq. 2
      called from    line 21-22

```

II) a: '400, '-400, '400, '-400

```

      real      lin. eq. 1
      called from    line 8-8

```

III) a: 1, 8'615, 0.5, -8'615

```

      real      lin. eq. 1
      called from    line 21-22

```

IV) a: 1, 12'615, 0.5, -12'615

```

      real      lin. eq. 2
      called from    line 8-8

```

5. References

- [1] Forsythe, G and Moler, C.B.: Computer Solution of Linear Algebraic Systems. Prentice-Hall. 1967.

6. Algorithms.

Since the procedures are written in SLANG, the algorithms will not be given.

eberlein

Solution of the eigenproblem for real matriceseberlein(n, a, t, tmx, first, result).1. Function and parameters.

It is possible to chose one of several forms of solution according to the rules given in the parameter list.

If the iteration process does not converge within the given number of iterations or converges to a matrix that is not of block diagonal form, no solution is found. This situation is indicated by the boolean parameter, first.

Input parameters:

n : the order of matrix a.

result : if result is true then in case of convergence the eigenvalues will be placed in the two first columns of matrix a.

Input/Output parameters:

a[1:n,1:n] : at entry the matrix for with the eigenproblem is to be solved.

At exit one of the following three situations can occur:

1) if convergence occurs and result is false :

the real eigenvalues occupy diagonal elements while real and imaginary parts of complex conjugate eigenvalues occupy diagonal and off diagonal corners of 2×2 blocks on the main diagonal.

2) if convergence occurs and result is true :

the eigenvalues will be placed in the two first columns according to the following rules

a real eigenvalue $x = a[j,j]$ makes

$$\begin{aligned} a[j,1] &= x \\ \text{and} \quad a[j,2] &= 0 \end{aligned}$$

a complex conjugate pair of eigenvalues

$$x + i y = a[j,j] + i a[j,j+1]$$

and $x - i y$ makes

$$\begin{aligned} a[j,1] &= x \\ a[j,2] &= y \\ a[j+1,1] &= x \\ \text{and} \quad a[j+1,2] &= -y \end{aligned}$$

3) if convergence fails no eigenvalues can be calculated as a result of the procedure call. The matrix, a , is equal to the transformed matrix. During a new call of ,eberlein, it is possible to try whether more iterations will result in convergence or not. (first is set to false).

$t[1:n,1:n]$: if first is false at entry and $tmx > 0$ then t given at entry is multiplied by the transformation matrix calculated in the procedure.

Eigenvectors of real eigenvalues occupy columns of the transformation matrix. Eigenvectors corresponding to complex conjugate eigenvalues given by

$$\begin{aligned} & a[j,j] + i a[j,j+1] \\ \text{and} \quad & a[j,j] - i a[j,j+1] \end{aligned}$$

are formed as

$$\begin{aligned} & t[k,j] + i t[k,j+1] \\ \text{and} \quad & t[k,j] - i t[k,j+1] \end{aligned}$$

where $k = 1, 2, \dots, n$.

`tmx` : at entry:

the maximum number of transformations performed is `abs(tmx)`. If `tmx < 0` then `t` is unaltered.

at exit `tmx` records the number of transformations performed.

`first` : at entry tells whether `t` is a result of a foregoing transformation or not. (see under `t[1:n,1:n]`).

at exit `first` is true if convergence occurs in less than `tmx` iterations otherwise `first` is false.

2. Method.

The procedure is based on a modification of a generalized Jacobi-method [1]. There exists no proof of convergence for this special modification, but numerical experiments have shown the worth of the method.

A transformation matrix T transforms the matrix A into a matrix of block diagonal form $A' = T^{-1}AT$.

The transformation matrix T is generated from a sequence of two-dimensional transformations $T_1(k, m)$, where (k, m) is the pivot pair.

Each T_i is of the form RS where R is a rotation and S a shear.

Let a_{ij} , r_{ij} and s_{ij} , $i = 1, 2, \dots, n$, $j = 1, 2, \dots, n$ be the elements of A, R and S respectively.

Then the rotation is determined as

$$\begin{aligned} r_{kk} &= r_{mm} = \cos x \\ r_{km} &= -r_{mk} = -\sin x \\ r_{ij} &= i_j \quad (\text{kroncker-delta}) \quad i \neq k, m \text{ and } j \neq k, m \end{aligned}$$

where x are given by

$$\tan 2x = (a_{km} + a_{mk}) / (a_{kk} - a_{mm})$$

x being chosen so that after the transformation the norm of the k-th column is greather than or equal to the norm of the m-th column.

The shear is determined by

$$\begin{aligned} s_{kk} &= s_{mm} = \cosh y \\ s_{km} &= s_{mk} = -\sinh y \\ s_{ij} &= i_j \quad \text{otherwise} \end{aligned}$$

y is chosen to reduce the Euclidean norm of

$$A_{i+1} = (T_1 T_2 \dots T_i)^{-1} A_i (T_1 T_2 \dots T_i).$$

In particular

$$\tanh y = (ED - H/2) / (G + 2(E^2 + D^2))$$

where

$$E = a_{km} - a_{mk}$$

$$D = \cos 2x (a_{kk} - a_{mm}) + \sin 2x (a_{km} + a_{mk})$$

$$G = \sum_{i \neq k, m} (a_{ki}^2 + a_{ik}^2 + a_{im}^2 + a_{mi}^2)$$

$$H = \cos 2x (2 \sum_{i \neq k, m} (a_{ki} a_{mi} - a_{ik} a_{im}))$$

$$- \sin 2x (\sum_{i \neq k, m} (a_{ki}^2 + a_{im}^2 - a_{ik}^2 - a_{mi}^2))$$

The process will normally result in a matrix A' with real eigenvalues on the diagonal and complex conjugate eigenvalues in 2×2 blocks on the main diagonal (eigenvalues $a_{jj} \pm ia_{jj+1}$). 2×2 blocks because the process theoretically results in

$$||a^1|| \geq ||a^2|| \geq \dots \geq ||a^n||$$

where $||a^1||$ is the norm of the 1-th column of A' .

If however more than two eigenvalues are of the same norm the above picture does not hold. This also happens if the matrix A_i at some step is near a more general solution of block diagonal form, because the convergence criteria then may stop the process.

The procedure however results in a form with 2×2 blocks obtained by interchanging rows and columns if necessary.

A matrix of the form $aI + S$ where I is the identity matrix and S is skew symmetric, and cases with blocks of this form cannot be handled by the procedure. (If the form not it self is a solution). There is no guarantee that the transformation method used will not result in convergence to a form embedding blocks from which the procedure cannot calculate the eigenvalues. In testexamples this did happen, but only in special cases chosen to examine the stability of such solutions. Notice that if the number of iterations performed is less than the maximum number of iterations allowed and ,first, is false then the resulting matrix is of the above mentioned form.

The eigenvectors are calculated from the rows of the transformation matrix, as described in the data list.

Numerical examples have shown that eigenvectors corresponding to multiple eigenvalues are normally linear dependent. (Except for numerical errors).

The algol procedure is based on an algorithm developed by Eberlein and Boothroyd [2]. The following changes are however made:

1. A program part ensures that the eigenvalues are placed in 1×1 and 2×2 blocks on the main diagonal. Matrices obtained as a result of convergence for which 1×1 and 2×2 blocks cannot be constructed results in alarm message through the parameters first and tmx.

2. the parameter list is changed.
3. a new dynamical form of convergence criterion is introduced.

The convergence criterion is based on the four reals ep , eps , $eps1$, and $eps2$.

At the start of the procedure ep , eps and $eps1$ are calculated as

$$\begin{aligned} ep &= \max \times 10^{-14} \\ eps &= \max \times 1.001 \times 10^{-9} \\ eps1 &= \max \times 10^{-3} \end{aligned}$$

where

$$\max = \text{maximum}(|a_{ij}|) \quad 1, j = 1, 2, \dots, n.$$

If for a single value of $eps1$

$$|a_{ij} - a_{ji}| \leq eps1 \text{ or } (|a_{ij} + a_{ji}| \leq eps1 \text{ and } |a_{ii} - a_{jj}| \leq eps1)$$

no more transformation are carried out before after a change of $eps1$. If $eps1 < eps$ convergence has occurred, otherwise a new value of $eps1$ is calculated as $eps1 = eps1/10$. The above value of eps makes sure that the resulting $eps1$ is near to and less than eps . $eps = \max \times 10^{-9}$ could cause an extra serie of iterations with $eps1 = \max \times 10^{-10}$ because of rounding errors.

As pivot pairs are only chosen pairs of elements for which

$$\begin{aligned} &(|a_{ij} - a_{ji}| > eps2 \text{ and } |a_{ii} - a_{jj}| > eps2) \\ &\text{or } |a_{ij} + a_{ji}| > eps2 \end{aligned}$$

If only identity transformations occurs as a result of this rule then a new value of $eps2$ is calculated as $eps2 = eps2/10$. Numerical experiments have shown that this extra mechanism is necessary to ensure convergence of some ill conditioned numerical examples.

The starting value of $eps2$ for every new value of $eps1$ is $eps2 = eps1/10$.

If $eps2$ gets less than ep convergence is not obtained by this algorithm and the process is stopped.

The values of ep , eps , $eps1$, and $eps2$ are results of experiments reducing the computation time about 30 per cent compared with a program making transformations for all pairs of elements cyclically.

- 7 -

3. Accuracy and storage Requirements.

Accuracy

In case of convergence the following inequalities holds for the elements of A'

$$\begin{aligned} &|a_{ij} - a_{ji}| \leq \text{eps or} \\ &|a_{ij} + a_{ji}| \leq \text{eps and } |a_{ii} - a_{jj}| \leq \text{eps} \end{aligned}$$

for $i = 1, 2, \dots, n-1$ and $j = i+1, i+2, \dots, n$ where

$$\text{eps} = \max(|\text{elements of original matrix, } a|) \times 10^{-9}$$

Storage requirements

- 1) ALGOL 5, index check: 7 tracks of program and 52 local variables
- 2) ALGOL 5, no index check: 6 tracks of program and 52 local variables.

Typographical length 167 lines of program exclusive the comment after the last end.

4. Test and discussion

The procedure has been tested on the ALGOL 5 system for matrices of order ≤ 12 .

The testprogram makes besides call of - eberlein - a calculation of

$$\text{testnorm} = ||A \times \underline{x} - \lambda \underline{x}|| / ||\underline{x}||$$

where A is the matrix for which the eigenproblem is solved, $\underline{x}$ is a calculated eigenvector and λ the corresponding eigenvalue.

Calculation of - testnorm - is made by a real procedure testnorm(n, A, t, k, complex, x1, x2) in the testprogram.

A list of input parameters, results, and calculated values of - testnorm - is delivered by the testprogram.

The starting values and following calculation of eps1 and eps2 are obtained as results of experiments resulting in 30 per cent decrease in execution time in solving testexamples.

- 9 -

Example no 1

Time:

ALGOL 5, core storage, no index check 0.36 sec.

Input parameters:

n = 3
tmx = 50

matrix a:

1.000	0.000	0.010
0.100	1.000	0.000
0.000	1.000	1.000

Results:

tmx = 15
first = true

Eigenvalues after 15 iterations

1	1.1000000000	
2	0.9499999999	+0.0866025403x1
3	0.9499999999	-0.0866025403x1

Eigenvectors:

1	-0.2745741273	
	-0.2745741274	
	-2.7457412704	
2	0.3262845267	+0.1836278766x1
	-0.0041158575	-0.3743846272x1
	-3.2216866940	+1.9075675104x1
3	0.3262845267	-0.1836278766x1
	-0.0041158575	+0.3743846272x1
	-3.2216866940	-1.9075675104x1

Testnorm for corresponding eigenvalues and eigenvectors

no. of eigenvalue	testnorm
-------------------	----------

1	1.5 _n -10
2 and 3	1.8 _n -10

Example no 4

Time:

ALGOL 5, core storage, no index check 10.1 sec.

Input parameters:

n = 7
tmx = 100

matrix a:

-1	1	0	0	0	0	0
-1	0	1	0	0	0	0
-1	0	0	1	0	0	0
-1	0	0	0	1	0	0
-1	0	0	0	0	1	0
-1	0	0	0	0	0	1
-1	0	0	0	0	0	0

Results:

tmx = 64
first = true

Eigenvalues after 64 iterations

1	-0.9999999982	
2	0.7071067794	+0.7071067790x1
3	0.7071067794	-0.7071067790x1
4	-0.7071067788	+0.7071067790x1
5	-0.7071067788	-0.7071067790x1
6	0.0000000001	-0.9999999956x1
7	0.0000000001	+0.9999999956x1

Eigenvectors:

1	-0.5585067124	
	0.0000000001	
	-0.5585067124	
	0.0000000001	
	-0.5585067123	
	0.0000000000	
	-0.5585067125	
2	-0.1107710832	+0.3310964522x1
	-0.4232186137	+0.4868900143x1
	-0.7543150660	+0.3761189310x1
	-0.9101086283	+0.0636714003x1
	-0.7993375446	-0.2674250515x1
	-0.4868900143	-0.4232186137x1
	-0.1557935621	-0.3124475306x1

3	-0.1107710832	-0.3310964521x1
	-0.4232186137	-0.4868900143x1
	-0.7543150660	-0.3761189310x1
	-0.9101086283	-0.0636714003x1
	-0.7993375446	+0.2674250515x1
	-0.4868900143	+0.4232186137x1
	-0.1557935621	+0.3124475306x1
4	0.4530114436	-0.6611576004x1
	0.6001930025	+0.1266788858x1
	-0.0609645978	-0.3263325577x1
	0.7268718886	-0.4735141167x1
	0.2738604449	+0.1876434837x1
	0.1266788860	-0.6001930026x1
	0.7878364866	-0.1471815591x1
5	0.4530114436	+0.6611576004x1
	0.6001930025	-0.1266788858x1
	-0.0609645978	+0.3263325577x1
	0.7268718886	+0.4735141167x1
	0.2738604449	-0.1876434837x1
	0.1266788860	+0.6001930026x1
	0.7878364866	+0.1471815591x1
6	0.5486381922	+0.2372706779x1
	0.7859088684	-0.3113675134x1
	0.2372706765	-0.5486381898x1
	-0.0000000002	+0.0000000002x1
	0.5486381923	+0.2372706781x1
	0.7859088687	-0.3113675132x1
	0.2372706767	-0.5486381901x1
7	0.5486381922	-0.2372706779x1
	0.7859088684	+0.3113675134x1
	0.2372706765	+0.5486381898x1
	-0.0000000002	-0.0000000002x1
	0.5486381923	-0.2372706781x1
	0.7859088687	+0.3113675132x1
	0.2372706767	+0.5486381901x1

Testnorm for corresponding eigenvalues and eigenvectors

no. of eigenvalue	testnorm
1	2.0 _n -9
2 and 3	3.1 _n -9
4 and 5	3.0 _n -9
6 and 7	5.9 _n -9

Example no 11

Time:

ALGOL 5, core storage, no index check 2.43 sec.

Input parameters:

n = 4
tmx = 100

matrix a:

1	0	0	0
1	1	0	0
0	1	1	0
0	0	1	1

Results:

tmx = 45
first = true

Eigenvalues after 45 iterations

1	1.0014985857	+0.0014992361x1
2	1.0014985857	-0.0014992361x1
3	0.9985014113	+0.0014978791x1
4	0.9985014113	-0.0014978791x1

Eigenvectors:

1

-0.0000637318	+0.0000264031x1
-0.0124657185	+0.0300779129x1
5.8782571872	+14.1901547856x1
6694.9278156800	+2771.1941409600x1

2

-0.0000637318	-0.0000264031x1
-0.0124657185	-0.0300779129x1
5.8782571872	-14.1901547856x1
6694.9278156800	-2771.1941409600x1

3

0.0000637404	+0.0000264010x1
-0.0124478455	-0.0300712114x1
-5.8782228542	+14.1910426896x1
6696.9885651200	-2775.7892716800x1

4

0.0000637404	-0.0000264010x1
-0.0124478455	+0.0300712114x1
-5.8782228542	-14.1910426896x1
6696.9885651200	+2775.7892716800x1

- 13 -

Testnorm for corresponding eigenvalues and eigenvectors

no. of eigenvalue	testnorm
1 and 2	1.5×10^{-9}
3 and 4	1.8×10^{-9}

Example no 12

Input parameters:

n = 3
tmx = 50

matrix a:

1.000	1.000	1.001
-1.000	1.000	0.000
-1.000	0.000	1.000

Results:

tmx = 28
first = false

Limiting matrix after 28 iterations

1.00×10^0	-1.17×10^0	7.60×10^{-1}
1.17×10^0	1.00×10^0	2.22×10^{-1}
-7.60×10^{-1}	-2.22×10^{-1}	1.00×10^0

5. References.

- [1] P.I. Eberlein: A Jacobi-like Method for the Automatic Computation of Eigenvalues and Eigenvectors of an Arbitrary Matrix.
I. SIAM, vol. 10, No. 1, 1962.
- [2] P.I. Eberlein and John Boothroyd: Solution to the Eigenproblem by a Norm Reducing Jacobi Type Method
Numerische Mathematik 11, 1-12 (1968).

6. Algorithm

```
eberlein=set 7
eberlein=algol
external
```

```
procedure eberlein(n,a,t,tmx,first,result);
value n;
boolean first,result;
integer n,tmx;
array a,t;
comment 1 ;
begin
  real eps,ep,a11,a1j,a1i,h,g,hj,aik,aki,aim,ami,tep,tem,d,c,e,akm,amk,cx,sx,
    cot2x,sig,cotx,cos2x,sin2x,te,tee,yh,den,tanhy,chy,shy,c1,c2,s1,s2,
    tk1,tmi,tk,tim,eps1,eps2;
  integer i,j,k,m,it,nless1;
  boolean mark,left,right;
  mark := right := false;
  if tmx > 0 then
    begin
      right := true;
      if first then
        for i := 1 step 1 until n do
          begin
            comment identity matrix is formed in t;
            t(i,i) := 1;
            for j := i+1 step 1 until n do t(i,j) := t(j,i) := 0;
          end
        end;
      tmx := abs(tmx);
      comment computation of the maximum absolute element of a;
      ep := 0;
      for i := 1 step 1 until n do
        for j := 1 step 1 until n do
          if abs(a(i,j)) > ep then ep := abs(a(i,j));
        end
      comment 2 ;
      eps := ep*1.001*10-9;
      eps1 := ep*10-3;
      ep := ep*10-14;
      first := true;
      nless1 := n-1;
      comment main loop , tmx iterations;
      for it := 1 step 1 until tmx do
        begin
          eps2 := eps1/10;
          comment compute convergence criteria;
          for i := 1 step 1 until n do
            begin
              a11 := a(i,i);
              for j := i+1 step 1 until n do
                begin
                  a1j := a(i,j);
                  aji := a(j,i);
                  if (abs(a1j-aji) > eps1 and abs(a11-a(j,j)) > eps1)
                    or abs(a1j+aji) > eps1 then goto cont
                end
              end convergence test, all i,j;
              goto next_eps1;
            end
          end
        end
      end
```

```

cont:      comment next transformation begins;
mark := true;
for k := 1 step 1 until nless1 do
for m := k + 1 step 1 until n do
begin
h := g := hj := yh := 0;
d := a(k,k) - a(m,m);
akm := a(k,m);
amk := a(m,k);
c := akm + amk;
e := akm - amk;
if (abs(e) <= eps2 or abs(d) <= eps2)
and abs(c) <= eps2 then goto skip;
for i := 1 step 1 until n do
begin
aik := a(i,k);
aim := a(i,m);
te := aik × aik;
tee := aim × aim;
yh := yh + te - tee;
if i > k and i > m then
begin
aki := a(k,i);
ami := a(m,i);
h := h + aki×ami - aik×aim;
tep := te + ami×ami;
tem := tee + aki×aki;
g := g + tep + tem;
hj := hj - tep + tem;
end
end i;
h := h + h;
if abs(c) <= ep then
begin
comment take R as identity matrix;
cx := 1;
sx := 0;
end else
begin
comment compute elements of R;
cot2x := d/c;
sig := if cot2x < 0 then -1 else 1;
cotx := cot2x + (sig×sqrt(1+cot2x××2));
sx := sig/sqrt(1+cotx××2);
cx := sx × cotx;
end;
if yh < 0 then
begin
tem := cx;
cx := sx;
sx := -tem;
end;
cos2x := cx××2 - sx××2;
sin2x := 2×sx×cx;
d := d×cos2x + c×sin2x;
h := h×cos2x - hj×sin2x;
den := g + 2×(exe + d×d);
tanhy := (exd - h/2)/den;

```

```

comment compute elements of S;
chy := 1/sqrt(1 - tanhy*tanhy);
shy := chy*tanhy;
comment elements of R×S = T;
c1 := chy*cx - shy*sx;
c2 := chy*cx + shy*sx;
s1 := chy*sx + shy*cx;
s2 := shy*cx - chy*sx;
comment decide whether to apply this transformation;
if abs(s1) > ep or abs(s2) > ep then
  begin
    comment at least one transformation is made so;
    mark := false;
    comment transformation on the left;
    for i := 1 step 1 until n do
      begin
        aki := a(k,i);
        ami := a(m,i);
        a(k,i) := c1*aki + s1*ami;
        a(m,i) := s2*aki + c2*ami;
      end left transformation;
      comment transformation on the right;
      for i := 1 step 1 until n do
        begin
          aik := a(i,k);
          aim := a(i,m);
          a(i,k) := c2*aik - s2*aim;
          a(i,m) := c1*aim - s1*aik;
          if right then
            begin
              comment form right vectors;
              tik := t(i,k);
              tim := t(i,m);
              t(i,k) := c2*tik - s2*tim;
              t(i,m) := c1*tim - s1*tik;
            end
          end right transformation
        end;
      end;
    end k,m loops;
  if mark then
    begin
      comment 3 ;
      if eps2 < ep then goto stop;
      eps2 := eps2/10;
      goto cont;
    end else goto new_loop;
  next_eps1:
    eps1 := eps1/10;
    comment 4 ;
    if eps1 < eps/2 then
      begin
        tmx := it - 1;
        goto done;
      end;
    new_loop:
      end it loop;

```

```

stop:
  first := false;
  tmx   := it-1;
done:
  if first then
    begin
      comment 5 ;
      for i := 1 step 1 until n-2 do
        begin
          mark := false;
          aii  := a(i,i);
          for j := i+1 step 1 until n do
            if abs(aii-a(j,j)) <= eps ^ abs(a(i,j)-a(j,i)) > eps then
              begin
                if mark then goto stop;
                mark := true;
                if j = i+1 then goto next;
                for k := 1 step 1 until n do
                  begin
                    aik      := a(i+1,k);
                    a(i+1,k) := a(j,k);
                    a(j,k)   := aik;
                  end;
                for k := 1 step 1 until n do
                  begin
                    aki      := a(k,i+1);
                    a(k,i+1) := a(k,j);
                    a(k,j)   := aki;
                    if right then
                      begin
                        tki      := t(k,i+1);
                        t(k,i+1) := t(k,j);
                        t(k,j)   := tki;
                      end;
                  end;
                end;
              end;
          end;
        next:
          end;
        end;
      comment the eigenvalues are placed in the first two columns;
      left := right := false;
      if result then
        for i := 1 step 1 until n do
          begin
            if -,right and i < n then
              left := abs(a(i,i+1)-a(i+1,i)) > eps and
                abs(a(i,i)-a(i+1,i+1)) <= eps;
            a(i,1) := if right then a(i-1,1) else a(i,i);
            a(i,2) := if left then a(i,i+1) else
              if right then -a(i-1,2) else 0;
            right := left;
            left  := false;
          end;
        end;
      end;
    end eberlein;

```

comment

1. eberlein solves the eigenproblem for a real matrix by means of a sequence of Jacobi-like transformations.

Input parameters:

n : the order of matrix a.

result : if result is true then in case of convergence the eigenvalues will be placed in the two first columns of matrix a.

Input/Output parameters:

a[1:n,1:n] : at entry the matrix for with the eigenproblem is to be solved.

At exit one of the following three situations can occur:

- 1) if convergence occurs and result is false :

the real eigenvalues occupy diagonal elements while real and imaginary parts of complex conjugate eigenvalues occupy diagonal and off diagonal corners of 2x2 blocks on the main diagonal.

- 2) if convergence occurs and result is true :

the eigenvalues will be placed in the two first columns according to the following rules

a real eigenvalue $x = a[j,j]$ makes

$a[j,1] = x$
and $a[j,2] = 0$

a complex conjugate pair of eigenvalues

$$x + i y = a[j,j] + i a[j,j+1]$$

and $x - i y$ makes

$a[j,1] = x$
 $a[j,2] = y$
 $a[j+1,1] = x$
and $a[j+1,2] = -y$

- 3) if convergence fails no eigenvalues can be calculated as a result of the procedure call. The matrix, a, is equal to the transformed matrix. During a new call of ,eberlein, it is possible to try whether more iterations will result in convergence or not. (first is set to false).

$t[1:n,1:n]$: if first is false at entry and $tmx > 0$ then t given at entry is multiplied by the transformation matrix calculated in the procedure.

Eigenvectors of real eigenvalues occupy columns of the transformation matrix. Eigenvectors corresponding to complex conjugate eigenvalues given by

$$\begin{aligned} & a[j,j] + i \times a[j,j+1] \\ \text{and} \quad & a[j,j] - i \times a[j,j+1] \end{aligned}$$

are formed as

$$\begin{aligned} & t[k,j] + i \times t[k,j+1] \\ \text{and} \quad & t[k,j] - i \times t[k,j+1] \end{aligned}$$

where $k = 1, 2, \dots, n$.

tmx : at entry:

the maximum number of transformations performed is $abs(tmx)$. If $tmx < 0$ then t is unaltered.

at exit tmx records the number of transformations performed.

first : at entry tells whether t is a result of a foregoing transformation or not. (see under $t[1:n,1:n]$).

at exit first is true if convergence occurs in less than tmx iterations otherwise first is false.

- A dynamical form of the convergence criterion is introduced, which are based on the four reals ep , eps , $eps1$, and $eps2$. In case of convergence of the iteration process the resulting matrix, a satisfies

$$\begin{aligned} & (abs(a(1, j) - a(j, 1)) \leq eps1 \\ & (\vee abs(a(1, 1) - a(j, j)) \leq eps1) \\ & \wedge abs(a(1, j) + a(j, 1)) \leq eps1 \end{aligned}$$

where $eps1 < eps/2$

3. If convergence is not obtained and the resulting transformation matrix is the identify matrix then if $eps2 = eps2/10 < ep$ the process is stopped (no solution) otherwise a new transformation is made with $eps2 = eps2/10$.
4. If $eps1 < eps/2$ the convergence criterion is fulfilled and the iteration process is stopped. If $eps1 \geq eps$ the calculation is continued with the new value of $eps1$.
5. A look up for the eigenvalues is made and at the same time it is controlled whether the resulting matrix 1×1 and 2×2 is on block diagonal form or not.

If the matrix does not consist of 1×1 and 2×2 blocks this is (if possibly) obtained by interchange of rows and columns on the a and t matrices.

Special forms of matrices that fulfil the convergence criterion are not of block diagonal form (with at most two -valid- elements in a row or column) and the procedure eberlein can not solve the eigenproblem for these special matrices;

7. Testprogram.

```

begin
real procedure testnorm(n,A,t,k,complex,x1,x2);
value n,k,complex,x1,x2;
array A,t;
boolean complex;
integer n,k;
real x1,x2;
comment The procedure performs a test of eigenvalues and corresponding
        eigenvectors calculated by procedure eberlein;
begin
integer i,j;
real sum,sum1,sum2,norm;
sum := norm := 0;
for i := 1 step 1 until n do
begin
sum1 := sum2 := 0;
for j := 1 step 1 until n do sum1 := sum1+A(i,j)*t(j,k);
if complex then
for j := 1 step 1 until n do sum2 := sum2+A(i,j)*t(j,k+1);
sum := (if complex then
        (sum1-x1*t(i,k)+x2*t(i,k+1))*x2
        +(sum2-x2*t(i,k)-x1*t(i,k+1))*x2
        else
        (sum1-x1*t(i,k))*x2 ) + sum;
end;
for i := 1 step 1 until n do norm := norm+t(i,k)*x2;
if complex then
for i := 1 step 1 until n do norm := norm + t(i,k+1)*x2;
x1 := if complex then sqrt(x1*x2+x2*x2) else abs x1;
testnorm :=sqrt(sum/norm)/x1;
end procedure testnorm;

integer i,j,n,no,m,layoutno,tmx,total,res;
boolean b1,b2,complex,first,result;
real im,x1,x2,layout;
read data:
total := 0;
i:=read (in,no,n,m,res);
comment
no = example no,
n = order of matrix,
m = value of tmx,
res = integer code for result;
if i <4 then goto stop;
tmx:=m;
result := res = 1;
write (out,<:10>Example_no:>,<<ddd>,no,
        <:10><10>Input_parameters:<10><10>n____=:>,<<dddd>,n,
        <:10>tmx____=:>,tmx,<:10><10>matrix_a:>);

```

- 23 -

```

begin
array a,t,A(1:n,1:n);
read (in,layoutno);
comment layout no serves a choice between three layouts in
      output of the unaltered matrix a;
layout := case layoutno of (real<< -ddd>,
real<< -d.ddd>,real<< dd>,real<< -d.d>);
write (out,<:<10>:>);
for j := 1 step 1 until n do
begin
write (out,<:<10>:>);
for i := 1 step 1 until n do
begin
comment input and output of matrix a;
read (in,a(j,i));
A(j,i) := a(j,i);
write (out,string layout,a(j,i));
end
end;
first := true;
new_eber:
eberlein(n,a,t,tmx,first,result);
write (out,<:<10><10><10>Results:<10><10>tmx ___=:>,<<-ddd>,tmx);
if first then
write (out,<:<10>first = true:>) else
write (out,<:<10>first = false:>);
if -,result or -,first then
begin
write (out,<:<10><10><10>Limiting matrix after:>,
<<ddd>,total+tmx,<:_iterations<10>:>);
for j := 1 step 1 until n do
begin
write (out,<:<10>:>);
for i := 1 step 1 until n do
write (out,<< -d.dd10dd>,a(j,i));
end
end else
begin
write (out,<:<10><10><10>Eigenvalues after:>,
<<ddd>,total+tmx,<:_iterations<10>:>);
for i := 1 step 1 until n do
begin
write (out,<:<10>:>,<<dd>,i,
<< -ddd.ddddddddd>,a(i,1));
if a(i,2) <> 0 then
write (out,<< +ddd.ddddddddd>,a(i,2),<:xi:>);
end;
if m = 0 then goto next;
write (out,<:<10><10><10>Eigenvectors:<10>:>);
b1 := b2 := false;
for i := 1 step 1 until n do
begin
write(out,<:<10>:>,<<dd>,i,<:<10>:>);
b1 := -,b2 and a(i,2) <> 0;

```

```

for j := 1 step 1 until n do
  begin
    im := if b2 and m>0 then t(j,i-1) else
           if b2 and m<0 then t(i-1,j) else
           if m=0 then t(j,i) else t(i,j);
    write (out,<< -dddd.iiiiiiiiidd>,im);
    im := if b2 and m>0 then -t(j,i) else
           if b2 and m<0 then -t(i,j) else
           if b1 and m>0 then t(j,i+1) else
           if b1 and m<0 then t(i+1,j) else 0;
    if b1 or b2 then
      write (out,<< +dddd.iiiiiiiiidd>,im,<:x:>);
      write (out,<:70>:>);
    end;
    b2 := b1;
    b1 := false;
  end;
write (out,<:10><10><10><10>Testnorm for corresponding>,
      <: eigenvalues and eigenvectors<10><10><10>:>,
      <:no. of eigenvalue_____testnorm<10>:>,
      <:10>:>);
b1 := false;
for i := 1 step 1 until n do
  begin
    x1 := a(i,1);
    x2 := a(i,2);
    complex := x2 <> 0;
    b1 := -,b1 and complex;
    if b1 or -,complex then
      im := testnorm(n,A,t,i,complex,x1,x2);
    if b1 then
      begin
        write(out,<:10>:>,<< __-dd>,1,<: and:>,i+1,
              <:_____>:>,<<d.d_10+dd>,im);
        end else
      if -, complex then
        begin
          write(out,<:10>:>,<< __-dd>,1,
                <:_____>:>,
                <<d.d_10+dd>,im);
          end;
        end;
      end;
    end;
  end;
next:
  if-, first and tmx = m then
    begin
      total := total+tmx;
      tmx := m;
      if total <= 3xm then goto new_eber;
    end;
  write(out,<:12>:>);
  end;
  goto read_data;
stop:
end testprogram

```

fft

fft


```
procedure fft(A, B, m, analysis);
```

1. Function and parameters:

Call parameters:

m integer value. m determines the dimension $N = 2 \times m$ of A and B . m must be $< 2^4$ to be meaningful, but the largest possible value is $m = 13$ in a computer with 128 k bytes storage capacity (corresponding to 32 k reals).

analysis integer value. $\text{analysis} = +1$ gives + sign in the sum, i.e. a Fourier-synthesis is carried out.
 $\text{analysis} = -1$ gives - sign in the sum, i.e. a Fourier-analysis is carried out.

Call and Return parameters:

$A, B(0:N-1)$ real arrays. They must on entry contain the real and imaginary part of $X(t)$ in normal order. Upon exit they contain the real and imaginary part of the Fourier sum $Y(s)$, also in normal order.

2. Method.

The procedure `fft` calculates the Fourier sum

$$Y(s) = \sum_t X(t) \exp(+2\pi i t s / N), \quad N = 2 \times m, \quad t = 0, 1, \dots, N-1$$

for $s = 0, 1, \dots, N-1$ by means of the Cooley-Tukey algorithm (the 'Fast Fourier Transform').

The first part of the procedure (`p:= 0; --- shift (m-j-24) end;`) delivers the data A, B in reverse binary order. The second part (`p:= 1; ... p:= p1 end`) performs the summation in place. So it is possible to carry out the summation with the data A, B in reverse binary order simply by omitting the first part.

The loss of accuracy is almost proportional to m and is for $m = 8$ about 1 significant decimal.

The running time is proportional to $N \times m$ and is for $m = 13$ about 45 seconds.

3. References.

About the Cooley-Tukey algorithm and its implementation, see:

- [1] Cooley, J.W., and Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comp. 19, 90 (April 1965), p. 297-301.
- [2] R.C. Singleton: On computing the fast Fourier Transform. Communications of the ACM, Vol. 10, N. 10, (October 1967), p. 647-654.

4. Algol procedure.

```
(  
fft=set 2  
fft=algol message.yes index.no  
end)  
  
external  
message fft version 01.07.71., RCSL NO 31-D5, correction by dh;  
procedure fft(A,B,m,analysis);  
value m,analysis; integer m, analysis; array A,B;  
begin integer i,j,k,n,p,p1,q,q2,j0,j1,j2;  
  real v,x0,x1,y0,y1,c,c1,cc,c2,s,ss,ss1,s2;  
  n:=1 shift m-1; v:=3.14159265359; p:=0;  
  for i:=0 step 1 until n do  
  begin  
    if i<p then  
    begin  
      c:=A(i);A(i):=A(p);A(p):=c;  
      c:=B(i);B(i):=B(p);B(p):=c  
    end;  
    k:=p shift (24-m); j:=-1;  
    for j:=j+1 while k<0 do k:=k shift 1;  
    p:=(-8388607-1+k) shift (m-j-24)  
  end;  
  q2:=p:=1;ss1:=0.0;  
  for i:=1 step 1 until m do  
  begin  
    ss:=ss1;v:=v/2.0;
```

```

ss1:=sin(v);cc:=2.0Xss1Xss1; q:=p-1;p1:=p+p; c1:=s:=0.0;
for j:=0 step 1 until q do
begin
  if j=q2 then c1:=s:=0.0; c:=1.0-c1;
  if j<q2 then
  begin
    c2:=c;s2:=s
  end
  else
  begin
    c2:=-s;s2:=c
  end;
  if analysis<0 then s2:=-s2;
  for k:=0 step p1 until n do
  begin
    j1:=j+k; j2:=j1+p;
    x1:=A(j2)Xc2-B(j2)Xs2; y1:=A(j2)Xs2+B(j2)Xc2;
    A(j2):=A(j1)-x1;B(j2):=B(j1)-y1;
    A(j1):=A(j1)+x1;B(j1):=B(j1)+y1
  end;
  c1:=c1+(ccXc+ssXs); s:=s+(ssXc-ccXs)
end;
q2:=p;p:=p1
end
end;

comment call parameters:

m      (integer value). m determines the dimension  $N=2 \times m$ 
      of A and B. m must be  $\geq 0$  and  $\leq 4$  to be meaningful
      but is in practice more limited since the core store
      has to contain the two arrays A and B plus some few
      other variables.

analysis (integer value). For analysis<0 a Fourier analysis
      is carried out:  $Y(s):=\text{SUM}(X(t)\exp(-2\pi i \times t \times s/N))$ .
      For analysis $\geq 0$  a Fourier synthesis is carried out:
       $Y(s):=\text{SUM}(X(t)\exp(+2\pi i \times t \times s/N))$ .

```

call and return parameters:

A,B(0:N-1) (real arrays). They must on entry contain the real
and imaginary part of the given data $X(t)=A(t)+iB(t)$
in normal order. Upon exit they will contain the
real and imaginary part of the Fourier sum
 $Y(s)=A(s)+iB(s)$, also in normal order;

end;

fit

fit

fit

RCSL 31-D129

Page 1/10

1. Function and Parameters

Given n data points (x_i, y_i) with weights p_i , where $i = 1, 2, \dots, n$. The procedure `fit` computes the coefficients of a polynomial $P(k, x)$, of degree k , such that the quantity

$$\begin{aligned} & i = n \\ & \text{SUM}(p_i \times (y_i - P(k, x_i))^2) \\ & i = 1 \end{aligned} \tag{1}$$

is minimum.

Bounds on the permissible order of $P(k, x)$ may be specified in the procedure call so that

$$\text{lower bound} \leq \text{order} \leq \text{upper bound}. \tag{2}$$

Moreover, the procedure will within these bounds select an order which is best in the following sense: Set the minimum value of (1), which is a function of the order, be denoted by $v(\text{order})$ and let

$$d(\text{order}) = v(\text{order}) / (n - \text{order} - 1),$$

which is an estimate of the restvariance. The procedure will then select the smallest order in the interval (2) such that $d(\text{order}) \leq d(\text{order}+1)$ or if this does not occur then $\text{order} = \text{upper bound}$. This means that the procedure increases the order of $P(k, x)$ only if the accuracy can be improved by doing it.

The procedure head is:

```
procedure fit(i, pi, xi, yi, C, l, u);  
  value                l, u ;  
  integer      i,      l, u ;  
  real          pi, xi, yi      ;  
  array                C      ;
```

Call parameters:

l : (integer) The lowest permissible order of $P(k, x)$
 u : (integer) The highest permissible order of $P(k, x)$

Return parameters:

$C(0:u)$: (array) Contains the coefficients of the fitted polynomial

$$P(i, x) = C(0) + C(1) \times x + C(2) \times x \times x + \dots + C(i) \times x \times x \times \dots \times x$$

where i is the order of P as determined by the procedure. If $i < u$ the remaining array elements are undefined.

Call and Return parameters:

- i : (integer) In the call i gives the number of data points (x_i , y_i) in the input. Upon exit i is the order of the fitted polynomial P .
Moreover i is used as index parameter in the expressions for p_i , x_i , y_i .
- p_i : (real) An expression (using the parameter i) giving the weight of point no. i .
- x_i : (real) An expression (using the parameter i) giving the x -coordinate of point no. i .
- y_i : (real) An expression (using the parameter i) giving the y -coordinate of the point no. i .

2. Method

The following description is based upon [1] where details and proofs are given.

Instead of expressing the approximating polynomial $P(k, x)$ directly as a sum of powers of x , we write $P(k, x)$ as a linear combination of polynomials $f(0, x)$, $f(1, x)$, $f(2, x)$, The polynomial $f(j, x)$ is a polynomial in x of proper degree j , which means that it effectively contains a term x^j (and the highest degree of x is not $j-1$):

$$P(k, x) = \sum_{j=1}^{j=k} (c(j) \times f(j, x)), \quad (3)$$

fit

RCSL 31-D129

Page 3/10

where k is the degree of the approximating polynomial P . The coefficients $c(j)$ are to be determined. The polynomials $f(j, x)$ are orthogonal with respect to the datapoints (x_i, y_i) and the weights p_i , $i = 1, 2, \dots, n$:

$$\begin{aligned} i &= n \\ \text{SUM}(p_i x_i f(j_1, x_i) x_i f(j_2, x_i)) &= 0, \text{ for } j_1 \neq j_2. \\ i &= 1 \end{aligned}$$

When $j_1 = j_2$ we define

$$\begin{aligned} i &= n \\ \text{SUM}(p_i x_i f(j, x_i) x_i f(j, x_i)) &= w(j); j = 0, 1, 2, \dots \\ i &= 1 \end{aligned}$$

The orthogonal polynomials can be computed recursively by means of a three-term recurrence relation

$$f(j+1, x) = (x - a(j))x f(j, x) - b(j)x f(j-1, x), j = 0, 1, 2, \dots \quad (4)$$

The recursion begins with $f(0, x) = 1$, and $b(0) = 0$. The coefficients $a(j)$ and $b(j)$ are determined by

$$\begin{aligned} i &= n \\ a(j) &= \text{SUM}(p_i x_i x_i f(j, x_i) x_i^2) / w(j); j = 0, 1, 2, \dots \quad (5a) \\ i &= 1 \end{aligned}$$

$$b(j) = w(j) / w(j-1); j = 1, 2, 3, \dots \quad (5b)$$

The coefficients $c(j)$ in (3) can be computed as

$$\begin{aligned} i &= n \\ c(j) &= \text{SUM}(p_i x_i f(j, x_i) x_i y_i) / w(j) \\ i &= 1 \end{aligned}$$

or - because the polynomial $f(j, x)$ is orthogonal to $P(j-1, x)$ - one also have, except for $j = 0$:

$$\begin{aligned} i &= n \\ c(j) &= \text{SUM}(p_i x_i f(j, x_i) x_i (y_i - P(j-1, x_i))) / w(j). \\ i &= 1 \end{aligned}$$

fit

RCSL 31-D129

Page 4/10

This is numerically more convenient, because

$$R(j+1, i) = y_i - P(j, x_i)$$

can be computed having the previous value:

$$R(j+1, i) = R(j, i) - c(j) \times f(j, x_i).$$

This holds because the polynomials f are orthogonal such that:

$$P(j, x) = P(j-1, x) + c(j) \times f(j, x).$$

During the computations the procedure performs the following steps: first it is tried to fit the given data points by means of a polynomial of order = lower bound, and then order = lower bound + 1 and so on. Each time the order is increased by one a new polynomial is generated, but this is easily done because the coefficients $c(j)$ already found are unchanged (This would not have been the case if an ordinary power expansion was used.) The sum of squares $v(j)$ (see (1)) is computed (also recursively) from $v(j-1)$:

$$v(j) = v(j-1) - w(j) \times c(j) \times 2.$$

From $v(j)$ the rest variance $d(j)$ is determined

$$d(j) = v(j)/(n-j-1).$$

When the rest variance decreases, i.e. when

$$d(m-1) > d(m),$$

the degree of the approximating polynomial is increased by one and (unless $m = \text{upper bound}$) a polynomial of degree $m + 1$ is tried, and so on until

$$d(j-1) \leq d(j)$$

in which case $j - 1$ is chosen as the degree (provided that $j - 1 \leq \text{upper}$

fit

RCSL 31-D129

Page 5/10

bound). This corresponds with

$$(n-j) \times w(j) \times c(j) \times x^2 \leq v(j-1),$$

which is used as stop-criterion.

When now a degree k has been chosen the polynomial (3) has to be transferred into a polynomial (a sum of powers):

$$P(k, x) = \sum_{j=1}^k C(j) \times x^j.$$

This is done recursively by means of $e(i, j)$ where $e(i, 0) = c(i)$ and $C(i) = e(i, i+1)$. The transformation is performed when $e(i, j+1)$ is expressed by means of $e(i, j)$. At a certain stage of the process we have

$$P(k, x) = \sum_{i=0}^{j-1} e(i, i+1) \times x^i + \sum_{i=j}^k e(i, j) \times f(i-j, x) \times x^j. \quad (6)$$

When we put $j = 0$ and $j = k + 1$ in (6) we get the above-mentioned connections with $c(i)$ and $C(i)$. Writing (6) for a certain j and for $j + 1$ we get a relation containing $f(i-j-1, x) \times x$. By means of the recurrence relation (4) we get

$$f(j, x) \times x = f(j+1, x) + a(j) \times f(j, x) + b(j) \times f(j-1, x)$$

which is introduced in the relation determined above. When we equate terms with a common factor $f(i-j, x)$ we get the recurrence relation

$$e(i, j+1) := e(i, j) - e(i+1, j+1) \times a(i-j) - e(i+2, j+1) \times b(i-j+1)$$

where $i \geq j \geq 0$; $e(k+1, j+1) = e(k+2, j+1) = 0$. From this relation the coefficients $e(i, i+1) = C(i)$, $i = 0, 1, \dots, k$, are obtained, and they are the desired coefficients in the power expansion of $P(k, x)$.

fit

RCSL 31-D129

Page 6/10

3. Time, and Storage Requirements

Time: $(2+k) \times n$, mS, where k = order of the polynomial obtained, n = the number of data points.

Storage Requirements: 3 segments + $25 + 4 \times k + 8 \times n$ words, where k = order of the polynomial, n = the number of data points.

Program text: 29 lines on 2 segments.

4. Test and Discussion

The procedure has been run with some examples.

Example.

A table of $y = xxx9 - xxx5$ with weights $p = xxx2 + 1$. The variance is $\sum(p \times (y - P(x))^2) / (n - o - 1)$

10 points, $l=6$, $u=8$, order = 8 103.000 ms
coefficients:

-2.61591₁₀-3
1.63066₁₀-1
2.88315₁₀-1
-1.94733₁₀ 0
-6.81127₁₀-1
5.13975₁₀ 0
-2.37534₁₀ 0
-4.90519₁₀ 0
4.36197₁₀ 0

p	x	v	P(x)	v-P(x)
1.00490 ₁₀ 0	-7.00000 ₁₀ -2	1.68066 ₁₀ -6	-1.19751 ₁₀ -2	1.2 ₁₀ -2
1.73960 ₁₀ 0	8.60000 ₁₀ -1	-2.13100 ₁₀ -1	-2.04917 ₁₀ -1	-8.2 ₁₀ -3
4.20410 ₁₀ 0	1.79000 ₁₀ 0	1.70282 ₁₀ 2	1.70282 ₁₀ 2	6.8 ₁₀ -5
1.09610 ₁₀ 0	-3.10000 ₁₀ -1	2.83648 ₁₀ -3	1.11621 ₁₀ -2	-8.3 ₁₀ -3
1.38440 ₁₀ 0	6.20000 ₁₀ -1	-7.80762 ₁₀ -2	-9.69892 ₁₀ -2	1.9 ₁₀ -2
3.40250 ₁₀ 0	1.55000 ₁₀ 0	4.26933 ₁₀ 1	4.26938 ₁₀ 1	-5.5 ₁₀ -4
1.30250 ₁₀ 0	-5.50000 ₁₀ -1	4.57231 ₁₀ -2	4.33494 ₁₀ -2	2.4 ₁₀ -3
1.14440 ₁₀ 0	3.80000 ₁₀ -1	-7.75830 ₁₀ -3	9.78251 ₁₀ -3	-1.8 ₁₀ -2
2.71610 ₁₀ 0	1.31000 ₁₀ 0	7.50371 ₁₀ 0	7.50219 ₁₀ 0	1.5 ₁₀ -3
1.62410 ₁₀ 0	-7.90000 ₁₀ -1	1.87854 ₁₀ -1	1.88112 ₁₀ -1	-2.6 ₁₀ -4

variance = 1.20₁₀-3

10 points, $l = 7$, $u = 9$, order = 8 114.000 ms variance = 1.20₁₀-3

10 points, $l = 8$, $u = 9$, order = 8 112.000 ms variance = 1.20₁₀-3

10 points, $l = 8$, $u = 9$, order = 9 121.000 ms

fit

RCSL 31-D129

Page 7/10

coefficients:

2.79397_{10}^{-9}
 2.79397_{10}^{-9}
 -6.51926_{10}^{-9}
 -5.82077_{10}^{-9}
 1.30385_{10}^{-8}
 -1.00000_{10}^0
 -1.73750_{10}^{-8}
 1.68802_{10}^{-9}
 -0.29104_{10}^{-9}
 1.00000_{10}^0

P	x	v	P(x)	v-P(x)
1.00490_{10}^0	0	-7.00000_{10}^{-2}	1.68066_{10}^{-6}	1.68323_{10}^{-6}
1.73960_{10}^0	0	8.60000_{10}^{-1}	-2.13100_{10}^{-1}	-2.13100_{10}^{-1}
4.20410_{10}^0	0	1.79000_{10}^0	1.70282_{10}^2	1.70282_{10}^2
1.09610_{10}^0	0	-3.10000_{10}^{-1}	2.83648_{10}^{-3}	2.83648_{10}^{-3}
1.38440_{10}^0	0	6.20000_{10}^{-1}	-7.80762_{10}^{-2}	-7.80642_{10}^{-2}
3.40250_{10}^0	0	1.55000_{10}^0	4.26933_{10}^1	4.26933_{10}^1
1.30250_{10}^0	0	-5.50000_{10}^{-1}	4.57231_{10}^{-2}	4.57231_{10}^{-2}
1.14440_{10}^0	0	3.80000_{10}^{-1}	-7.75830_{10}^{-3}	-7.75830_{10}^{-3}
2.71610_{10}^0	0	1.31000_{10}^0	7.50371_{10}^0	7.50371_{10}^0
1.62410_{10}^0	0	-7.90000_{10}^{-1}	1.87854_{10}^{-1}	1.87584_{10}^{-1}

variance = 9.00_{10}^9

15 points, 1 = 6, u = 8, order = 8 145.000 ms

coefficients:

-5.63362_{10}^{-2}
 2.88497_{10}^{-2}
 1.92269_{10}^0
 -1.79384_{10}^0
 -7.03926_{10}^0
 8.64038_{10}^0
 2.95869_{10}^0
 -1.04081_{10}^1
 5.72527_{10}^0

P	x	v	P(x)	v-P(x)
1.00490_{10}^0	0	-7.00000_{10}^{-2}	1.68066_{10}^{-6}	-4.85023_{10}^{-2}
1.73960_{10}^0	0	8.60000_{10}^{-1}	-2.13100_{10}^{-1}	3.4_{10}^{-2}
4.20410_{10}^0	0	1.79000_{10}^0	1.79282_{10}^2	1.70294_{10}^2
1.09610_{10}^0	0	-3.10000_{10}^{-1}	2.83648_{10}^{-3}	8.91634_{10}^{-2}
1.38440_{10}^0	0	6.20000_{10}^{-1}	-7.80762_{10}^{-2}	-4.89320_{10}^{-2}
3.40250_{10}^0	0	1.55000_{10}^0	4.26933_{10}^1	4.26580_{10}^1
1.30250_{10}^0	0	-5.50000_{10}^{-1}	4.57231_{10}^{-2}	1.71621_{10}^{-2}
1.14440_{10}^0	0	3.80000_{10}^{-1}	-7.75830_{10}^{-3}	5.50052_{10}^{-2}
2.71610_{10}^0	0	1.31000_{10}^0	7.50371_{10}^0	7.55223_{10}^0
1.62410_{10}^0	0	-7.90000_{10}^{-1}	1.87854_{10}^{-1}	1.91326_{10}^{-1}
1.01960_{10}^0	0	1.40000_{10}^{-1}	-5.37617_{10}^{-5}	-2.17621_{10}^{-2}
2.14490_{10}^0	0	1.07000_{10}^0	4.35907_{10}^{-1}	4.34033_{10}^{-1}
5.00000_{10}^0	0	2.00000_{10}^0	4.80000_{10}^2	4.79998_{10}^2
1.01000_{10}^0	0	-1.00000_{10}^{-1}	9.99900_{10}^{-6}	-3.89867_{10}^{-2}
1.68890_{10}^0	0	8.30000_{10}^{-1}	-2.06964_{10}^{-1}	-2.38302_{10}^{-1}

variance = 5.71_{10}^{-3}

fit

RCSL 31-D129

Page 8/10

15 points, l = 7, u = 9, order = 9 163.000 ms

coefficients:

$3.72529_{10^{-9}}$
 $-1.95578_{10^{-8}}$
 $3.02680_{10^{-8}}$
 $-7.89296_{10^{-8}}$
 $-6.37956_{10^{-8}}$
 -1.00000_{10^0}
 $-1.64146_{10^{-7}}$
 $-2.26632_{10^{-7}}$
 $2.11265_{10^{-7}}$
 1.00000_{10^0}

p	x	v	P(x)	v-P(x)	
1.00490 _{10^0}	0	-7.00000 _{10^{-2}}	1.68066 _{10^{-6}}	1.68593 _{10^{-6}}	-5.3 _{10^{-9}}
1.73960 _{10^0}	0	8.60000 _{10^{-1}}	-2.13100 _{10^{-1}}	-2.13100 _{10^{-1}}	4.9 _{10^{-9}}
4.20410 _{10^0}	0	1.79000 _{10^0}	1.70282 _{10^2}	1.70282 _{10^2}	3.0 _{10^{-8}}
1.09610 _{10^0}	0	-3.10000 _{10^{-1}}	2.83648 _{10^{-3}}	2.83649 _{10^{-3}}	-1.3 _{10^{-8}}
1.38440 _{10^0}	0	6.20000 _{10^{-1}}	-7.80762 _{10^{-2}}	-7.80762 _{10^{-2}}	6.2 _{10^{-9}}
3.40250 _{10^0}	0	1.55000 _{10^0}	4.26933 _{10^1}	4.26933 _{10^1}	3.0 _{10^{-8}}
1.30250 _{10^0}	0	-5.50000 _{10^{-1}}	4.57231 _{10^{-2}}	4.57231 _{10^{-2}}	-1.4 _{10^{-8}}
1.14440 _{10^0}	0	3.80000 _{10^{-1}}	-7.75830 _{10^{-3}}	-7.75830 _{10^{-3}}	2.9 _{10^{-9}}
2.71610 _{10^0}	0	1.31000 _{10^0}	7.50371 _{10^0}	7.50371 _{10^0}	1.8 _{10^{-8}}
1.62410 _{10^0}	0	-7.90000 _{10^{-1}}	1.87854 _{10^{-1}}	1.87854 _{10^{-1}}	1.5 _{10^{-8}}
1.01960 _{10^0}	0	1.40000 _{10^{-1}}	-5.37617 _{10^{-5}}	-5.37604 _{10^{-5}}	-1.4 _{10^{-9}}
2.14490 _{10^0}	0	1.07000 _{10^0}	4.35907 _{10^{-1}}	4.35907 _{10^{-1}}	4.9 _{10^{-9}}
5.00000 _{10^0}	0	2.00000 _{10^0}	4.80000 _{10^2}	4.80000 _{10^2}	6.0 _{10^{-8}}
1.01000 _{10^0}	0	-1.00000 _{10^{-1}}	9.99900 _{10^{-6}}	1.00051 _{10^{-5}}	-6.1 _{10^{-9}}
1.68890 _{10^0}	0	8.30000 _{10^{-1}}	-2.06964 _{10^{-1}}	-2.06964 _{10^{-1}}	5.2 _{10^{-9}}

variance 0.00_{10^0}

15 points, l = 8, u = 10, order = 9 176.000 ms variance = 0.00_{10^0}

15 points, l = 4, u = 11, order = 9 171.000 ms variance = 0.00_{10^0}

end

The employed program:

```

begin integer i, l1, n, k, j, h; real x, v, t, t0, q, s, a; array C(0:12);
  real procedure p(dum); integer dum;
  begin
    x:= (31x1) mod 101x(3/100)-1;
    y:= xxx9-xxx5;
    p:= 1+xxx2
  end p;

  for n:= 10,15 do begin
    l1:= -1;
    for k:= 6 step 1 until 9 do begin
      t0:= time+25600;

```

```

for j:= 0, j+10 while t < t0 do begin
  i:= n; fit(i,p(i),x,v,C,k,k+2); t:= time
end;
t:= (t-t0+25600)/j;
write(out,<:<10><10>:>,n,<: points, order:=>, i, << ddd.000>,
  t,<: ms:>);
if i > i1 then begin
  write(out, <:<10>coefficients:>);
  for j:= 0 step 1 until i do
    write(out,<:<10>:>,<<-d.ddddd10-d>, C(j));
  write(out,<:
p          x          v          P(x)          v-P(x):>);
  end i > i1;
  s:= 0; j:= 1;
  for i:= 1 step 1 until n do begin
    q:= p(i) ; a:= C(i);
    for h:= j - 1 step -1 until 0 do a:= a*x+C(h);
    if j > i1 then write(out,<:<10>:>,<<-d.ddddd10-d>,q,x,v,a,
      <<-d.d10-d>,v-a);
    s:= s + (y-a)*x*x*q
  end i;
  write(out,<<-d.dd10-d>,<: variance:=>,
    if n =j+1 then 9109 else s/(n-j-1));
  i1:= j
end k;
end n
end

```

5. References

- [1] Forsythe, George E.: Generation and Use of Orthogonal Polynomials for Data-Fitting with a Digital Computer. Jour. Soc. Indust. Appl. Math. 5 (1957) pp. 74-88.

6. Algorithm

```

external procedure fit(i,pi,xi,yi,C,l,u); value l,u;
    integer i,l,u; real pi,xi,yi; array C;
begin integer j,k,n;
    real fj,r,rf,f,fx,f1,a,b,c;
    array F,F1,X,R(1:i),A,B(0:u);
    n:=i; r:=rf:=f:=fx:=b:=0;
    for i:=1 step 1 until n do begin
        fj:=F(i):=sqrt(pi); F1(i):=0;
        X(i):=xi;
        R(i):=yi*fj; r:=r+R(i)×2; rf:=rf+R(i)×fj;
        f:=f+fj×fj; fx:=fx+X(i)×fj×fj
    end i;
    for i:=0,k+1 while k<1 | (i<u ∧ fx<(n-i)×rf×rf) do begin
        k:=i; a:=A(k):=fx/f; c:=C(k):=rf/f;
        f1:=f; r:=rf:=f:=fx:=0;
        for j:=1 step 1 until n do begin
            R(j):=R(j)-F(j)×c; r:=r+R(j)×2;
            fj:=(X(j)-a)×F(j)-b×F1(j); F1(j):=F(j); F(j):=fj;
            rf:=rf+R(j)×fj; f:=f+fj×fj; fx:=fx+X(j)×fj×fj
        end j;
        b:=B(k):=f/f1
    end i;
    if fx<(n-k-1)×rf×rf then C(k+1):=rf/f else k:=k-1;
    i:=k+1;
    for l:=0 step 1 until k do begin
        C(k):=C(k)-A(k-1)×C(k+1);
        for j:=k-1 step -1 until l do C(j):=C(j)-A(j-1)×C(j+1)-B(j-1)×C(j+2)
    end l
end fit; end

```

gamma

gamma

Gamma function, gamma(z)

1. Function and parameters.

gamma(z) approximates the gamma function in the range
 $-301 < z < 301$.

Procedure heading:

```
real procedure gamma(z);  
value z; real z;
```

Procedure identifier:

```
gamma      : (real)  
            approximated function of an argument not  
            resulting in under- or overflow, in which  
            case gamma is undefined.
```

Call parameter:

```
z          : (real or integer)  
            argument, values equal to nonpositive  
            integers and values exceeding the range  
            above will give floating point under-  
            or overflow.
```

2. Method.

The value of gamma(2+x) is approximated in the range
 $0 \leq x \leq 1$ by a rational function given as approximation 5231
in (1) with the numerator degree 6 and denominator degree 3.

For arguments outside the basic range $2 \leq z \leq 3$, successive
multiplications or divisions are performed according to
the recurrence formula:

$$\text{gamma}(z+1) = z \times \text{gamma}(z)$$

The value of the argument is not controlled in any way.

3. Accuracy, Time- and Storage Requirement.

3.1 Accuracy.

The error estimates given below assume the argument to be exactly represented.

Outside the range $-2 < z < 7$ the increment in $\text{gamma}(z)$ caused by an increment of one unit in the last binary place of z will be greater than the computational error of the procedure in any case.

The error estimates are given as functions of:

$u := \text{abs}(\text{entier}(z-2)) + 6$

max rel error : $u \times 2.9 \times 10^{-11}$

safe upper bound for the relative error
of $\text{gamma}(z)$.

rel mean error: $\text{sqrt}(u) \times 1.2 \times 10^{-11}$

relative mean error (standard error) of
 $\text{gamma}(z)$ assuming a random distribution
of rounding errors and mantissas of
floating point numbers.

The probability of a relative error greater
than $3 \times (\text{rel mean error})$ is less than 0.01.

3.2 Time Requirement.

Approximate cpu-time: $z \geq 2$: $720 + \text{entier}(z-2) \times 81$ usec
 $z < 2$: $720 + \text{entier}(z-2) \times 72$ usec

3.3 Storage Requirement:

Codelength: 1 segment.

Typographical length: 47 lines incl. last comment.

4. Test and discussion.

The procedure has been compared with a double precision procedure and with values of the gamma function given in (2). The results are in accordance with the theoretical estimate of the mean error.

gamma - 3 -

Simple testprogram with data and output:

begin

comment: here the procedure is copied (without the first
3 lines) unless it is already translated as an
external;

```

real z, g;
write(out,<:<12>
      z                      gamma(z)<10>:>);
AGAIN:
  overflows:=underflows:=0;
  read(in,z);
  write(out,<:<10>:>,<<-d.dd ddd ddd dddn-ddd>, z);
  if z>1000 then goto FINISH;
  g:=gamma(z);
  if overflows>0 then
    write(out, false add 32, 15, <:overflow:>)
  else if underflows>0 then
    write(out, false add 32, 15, <:underflow:>)
  else write(out,
            <<          -d.dd ddd ddd dddn-ddd>, g);
    goto AGAIN;
FINISH:
end;
```

data: 0.5, 1, 10, 301, 302,-0.5,-300.9,-301.9,1001

```

output:
      z                      gamma(z)
5.00 000 000 000n -1      1.77 245 385 088n 0
1.00 000 000 000n 0      1.00 000 000 000n 0
1.00 000 000 000n 1      3.62 880 000 000n 5
3.01 000 000 000n 2      3.06 057 512 208n 614
3.02 000 000 000n 2      overflow
-5.00 000 000 000n -1     -3.54 490 770 176n 0
-3.00 900 000 016n 2      -1.95 307 772 968n-616
-3.01 900 000 016n 2      underflow
1.00 100 000 000n 3
end
```

5. References.

- (1) J.F.Hart and oth.:
Computer Approximations,
John Wiley and Sons, 1968, p.130-136
- (2) M.Abramowitz and I.H.Stegun:
Handbook of Math. Functions,
National Bureau of Standards, 1965, p.253-275.

6. Algorithm.

```
gamma = set 1
gamma = algol
external
```

```
real procedure gamma(z);
value z; real z;
begin
  real h;
  h:=1.0;
  if z>2.0 then
    begin
      for z:=z-1.0 step -1.0 until 2.0 do h:=h*xz;
      z:=z-1.0
    end
  else if z<1.0 then
    begin
      for z:=z step 1.0 until 0.0 do h:=h/z;
      h:=h/z/(z+1.0)
    end
  else begin h:=h/z; z:=z-1.0 end;
  gamma:=(((((((+.039 301 346 419)xz+.142 928 007 949)xz
    +1.09 850 630 453)xz+3.36 954 359 131)xz
    +12.8 021 698 112)xz+22.9 680 800 836)xz
    +43.9 410 209 189)/
    (((+1.00 000 000 000)xz-7.15 075 063 299)xz
    +4.39 050 474 596)xz+43.9 410 209 191)xh
end gamma;
```

comment:

gamma(z) approximates the gamma function in the range
-301<z<301.

Procedure identifier:

gamma : (real)
approximated function of an argument not
resulting in under- or overflow, in which
case gamma is undefined.

Call parameter:

z : (real or integer)
argument, values equal to nonpositive
integers and values exceeding the range
above will give floating point under-
or overflow.

householder

householder

1. Function and parameters

Let a denote a real symmetric matrix of order n , and let $ev(1)$, $ev(2)$, ..., $ev(n)$ denote the eigenvalues for this matrix arranged in an increasing sequence, that is $ev(i) \leq ev(j)$ whenever $i \leq j$. Let $m1$ and $m2$ be prescribed integers so that $1 \leq m1 \leq m2 \leq n$. The procedure householder calculates the eigenvalues $ev(m1)$, $ev(m1+1)$, ..., $ev(m2)$ and, if wanted the corresponding eigenvectors.

Procedure head:

```
householder(n, m1, m2, a, ev, x, eps1);  
value n, m1, m2, eps1;  
real eps1;  
array a, ev, x;  
integer m1, m2, n;
```

Call parameters:

n : the order of the given matrix;
 $m1$: an integer, $1 \leq m1 \leq n$, denoting the number of the smallest eigenvalue to be calculated.
 $m2$: an integer, $m1 \leq m2 \leq n$, denoting the number of the greatest eigenvalue to be calculated.
 a : a real array $a(1:n \times (n+1)/2)$;
a must contain the lower triangular part of the given symmetric matrix in the following way:
the diagonal element number i is stored in $a(i \times (i+1)/2)$
 $i = 1, 2, \dots, n$;
the element in the i 'th row and j 'th column where $j < i$ is stored in $a((i-1) \times i/2 + j)$.

Call/Return parameters:

$eps1$: at entry $eps1$ is positive or negative.
if $eps1$ is positive the eigenvectors are calculated. The absolute value of $eps1$ is a quantity affecting the precision to which the eigenvectors are computed (See part 2.2);
at exit $eps1$ denotes an upper bound for the error in any of the calculated eigenvalues.

Return parameters:

ev : a real array ev(m1:m2) containing the calculated eigenvalues.
x : a real array x(m1:m2, 1:n+2); if the eigenvectors are calculated, they are stored in x in such a way that x(k,1), ... x(k,n) denotes the eigenvector corresponding to ev(k);
(for each k x(k,n+1) = x(k,n+2) = 0; these quantities are only introduced for ease of programming).

2. Method

The method consists of four parts, tridiagonalisation, calculation of eigenvalues, calculation of eigenvectors, and backtransformation.

2.1. Tridiagonalisation

A matrix is said to be on tridiagonal form, if all elements that are not in the diagonal or just over or under the diagonal, are zero.

Let A_1 be the given symmetric matrix of order n.

A_1 is transformed - by n-2 orthogonal transformations - to a matrix on triangular form.

Each transformation P_i ($i = 1, 2, \dots, n-2$) is of the form

$$P_i = I - 2w_i w_i^T$$

where I is the identity-matrix and w_i^T is the row:

$$w_i^T = (w_{i,1}, w_{i,2}, \dots, w_{i,n-1}, 0, \dots, 0).$$

and w_i the corresponding column.

Let $A_{i+1} = P_i A_i P_i$ $i = 1, 2, \dots, n-2$

For each i the terms $w_{i,1}, w_{i,2}, \dots, w_{i,n-1}$ are chosen in such a way that

$$1^0. w_i^T w_i = 1$$

2⁰. In A_{i+1} the elements in the rows number n, n-1, ..., n-i+2 are the same as in A_i . The row number n-i+1 is put on 'triangular' form.

- 3 -

Let the elements of A_i be denoted a_{ij} . Put

$$t = n - i.$$

$$\sigma = a_{t+1,1}^2 + a_{t+2,2}^2 + \dots + a_{t+1,t}^2.$$

$$h_i = \sigma \pm a_{t+1,t} \sqrt{\sigma}.$$

(+ is used if $a_{t+1,t} \geq 0$ else - is used.)

It comes out, that $w_{i,1}, w_{i,2}, \dots, w_{i,t}$ must be chosen as follows

$$w_{i,t} = (a_{t+1,t} \pm \sqrt{\sigma}) / \sqrt{2h_i}.$$

$$w_{i,j} = a_{t+1,j} / \sqrt{2h_i} \quad j = 1, 2, \dots, t-1.$$

By introducing

$$u_i^T = (a_{t+1,1}, a_{t+2,2}, \dots, a_{t+1,t-1}, a_{t+1,t} \pm \sqrt{\sigma}, 0, \dots, 0).$$

one will obtain

$$P_i = I - \frac{u_i u_i^T}{h_i}$$

and by introducing the vectors p_i, q_i and the scalar k_i as follows

$$p_i = A_i u_i / h_i$$

$$k_i = u_i^T p_i / (2h_i)$$

$$q_i = p_i - k_i u_i$$

a rather simple calculation will show that

$$A_{i+1} = A_i - u_i q_i^T - q_i u_i^T$$

since A_{i+1} is symmetric one is only calculating the lower triangular part of the matrix.

The above equation is used for the calculation of the first t rows ($t = n-i$) in A_{i+1} . The row number $t+1$ is on triangular form with the diagonal element unchanged from A_i and the element $(t+1, t) = a_{t+1,t} \pm \sqrt{\sigma}$. The rows number $t+1, \dots, n$ are according to 2^0 - unchanged from A_i .

At entry the lower triangular part of the given matrix is stored in the array a . For each i the array a is used only to store the lower triangular part of the first t rows of A_{i+1} . The other rows are on triangular form, and the diagonal and subdiagonal elements from these rows are stored in to arrays c and b .

The row number $t+1$ of the array a is used to store information enough to determine the transformation P_i . Now, P_i is determined by the vector u_i^T and the scalar h_i . By replacing in the array a the element $a_{t+1,t}$ by $a_{t+1,t} \pm \sqrt{\text{sigma}}$ one obtain that the non-zero elements of the $t+1$ row in a are exactly the vector u_i^T . Furthermore from these elements h_i can be determined. Recalling that

$$h_i = \text{sigma} \pm a_{t+1,t} \sqrt{\text{sigma}}.$$

and denoting by sigma_i the square-sum of the elements in u_i^T one will obtain

$$\begin{aligned} \text{sigma}_i &= a_{t+1,1}^2 + \dots + a_{t+1,t-1}^2 + (a_{t+1,t} \pm \sqrt{\text{sigma}})^2 = \\ &= 2\text{sigma} \pm 2a_{t+1,t} \sqrt{\text{sigma}} = 2h_i. \end{aligned}$$

So $h_i = \text{sigma}_i/2$.

For further information about this part see [4], [6].

2.2. Calculation of eigenvalues

This is based on the following theorem:

let $c_1, \dots, c_n$ denote the diagonal element and $b_2, \dots, b_n$ the sub-diagonal elements of a symmetric triangulator matrix. For each real number x_0 let the sequence $t_1(x_0), t_2(x_0), \dots, t_n(x_0)$ be defined - if possible - as follows

$$t_1(x_0) = c_1 - x_0$$

$$t_i(x_0) = (c_i - x_0) - b_i^2/t_{i-1}(x_0). \quad i = 2, \dots, n.$$

Let $h(x_0)$ denote the number of negative $t_i(x_0)$.

Then $h(x_0)$ is equal to the number of eigenvalues less than or equal to x_0 .

Assume that the eigenvalues are arranged in an increasing sequence and that the k 'th eigenvalue, $\text{ev}(k)$, is to be calculated. Let x_1 and x_2 be real numbers satisfying $x_1 \leq \text{ev}(k) < x_2$. Such numbers exist, e.g. if norm is denoting the infinity norm of the matrix then $x_1 = -\text{norm}$ and $x_2 = \text{norm}$ will do.

Let $x_0 = (x_1 + x_2)/2$.

$h(x_0)$ is calculated by using the above mentioned formular for $t_i(x_0)$

$i = 1, 2, \dots, n$.

A new pair (x_1, x_2) is defined in the following way:

if $h(x_0) \geq k$ then $x_1 := x_1$ and $x_2 := x_0$ else $x_1 := x_0, x_2 := x_2$.

For the new pair the procedure is repeated. This is done as long as $x_2 - x_1 > 2 \times 10^{-10} \times (\text{abs}(x_1) + \text{abs}(x_2)) + \text{eps1}$ where eps1 is a prescribed quantity.

At the end one puts $\text{ev}(k) := (x_1 + x_2)/2$.

Since $\text{abs}(x_1)$ and $\text{abs}(x_2)$ always are bounded by norm, it follows that the error in any eigenvalue is bounded by $4 \times 10^{-10} \times \text{norm} + \text{eps1}$. This number is calculated and stored in eps1 .

When calculating the k 'th eigenvalue, $h(x_0)$ is determined for some x_0 . The value of $h(x_0)$ gives information not only about the k 'th eigenvalue, but in general about the eigenvalues of the matrix. By introducing an array $p(i)$ satisfying for each i $p(i) \leq \text{ev}(i)$ this information is stored as follows:

if $p(h(x_0) + 1) < x_0$ then $p(h(x_0) + 1) := x_0$;

when calculating the k 'th eigenvalue one is at the start putting

$x_1 := \max p(1), \dots, p(k) ; x_2 := \text{ev}(k+1);$

For further information about this part see [2], [5], [6].

2.3. Calculation of eigenvectors

The matrix is as in 2.2 a symmetric matrix on triangular form with diagonal elements $c_1, c_2, \dots, c_n$ and subdiagonal elements $b_2, \dots, b_n$. Let ev denote a calculated eigenvalue.

Finding an eigenvector corresponding to ev is equivalent to solve the system

$$\begin{aligned} (c_1 - \text{ev})x_1 + b_2x_2 &= 0 \\ b_2x_1 + (c_1 - \text{ev})x_2 + b_3x_3 &= 0 \\ \vdots & \\ b_{n-1}x_{n-2} + (c_{n-1} - \text{ev})x_{n-1} + b_nx_n &= 0 \\ b_nx_{n-1} + (c_n - \text{ev})x_n &= 0 \end{aligned} \tag{I}$$

where $(x_1, \dots, x_n)$ denote the wanted eigenvector.

A natural way to solve this system would consist in putting $x_1 = 1$ finding x_2 from the first equation, x_3 from the next and so on; but, as shown in [4], a method like this will often - for several reasons - give hopeless, inaccurate results.

Using a method developed by J.H. Wilkenson ([4]), one is instead solving a system derived from (I) by replacing the zeros on the right side by suitable quantities $d_1, \dots, d_n$.

These equations are solved by successive elimination of the variables $x_1, x_2, \dots, x_{n-1}$, but some kind of pivoting is necessary; for each i , x_i is eliminated from the equation, which has the numerical largest coefficient in x_i ; more precisely, at the first step we are considering the two first equations

$$(c_1 - cv)x_1 + b_2x_2 = d_1$$

$$b_2x_2 + (c_2 - ev)x_2 + b_3x_3 = d_2.$$

The equation which has the numerical largest coefficient is x_1 is denoted

$$p_1x_1 + q_1x_2 + r_1x_3 = d_1''$$

from this equation x_1 is calculated and the expression inserted in the other equation. The so obtained equation in x_2 and x_3 is denoted

$$u_2x_2 + v_2x_3 = d'$$

At the i 'th step we are considering the two equations

$$u_ix_i + v_ix_{i+1} = d_i'$$

$$b_{i+1}x_i + (c_{i+1} - cv)x_{i+1} + b_{i+1}x_{i+2} = d_{i+1}.$$

again the equation which has the numerical largest coefficient in x_i is denoted

$$p_ix_i + q_ix_{i+1} + r_ix_{i+2} = d_i''$$

from this equation x_i is calculated and the expression inserted in the other equation.

In this way we obtain the following system:

$$p_1x_1 + q_1x_2 + r_1x_3 = d_1''$$

$$p_2x_2 + q_2x_3 + r_2x_4 = d_2''$$

:

$$p_{i-2}x_{n-2} + q_{n-2}x_{n-1} + v_{n-2}x_n = d_{n-2}''$$

$$p_{n-1}x_{n-1} + q_{n-1}x_n = d_{n-1}''$$

$$p_nx_n = d_n''.$$

We now assume, that $d_1, d_2, \dots, d_n$ were chosen in such a way, that $d_1'', d_2'', \dots, d_n''$ are all equal to one.

This system is solved in the natural way and the obtained vector normed. (and again denoted $x_1, \dots, x_n$). It can be proved ([4]) that this vector will usually be a good approximation, at least it will never be hopeless inaccurate.

A vector with sufficient accuracy is obtained by solving the above system once again, but replacing the terms $d_1'', \dots, d_n''$ by the coordinates in the first approximation $x_1, \dots, x_n$.

For further information about this part see [3], [4], [6].

2.4. Backtransformation

The problem is to transform the calculated eigenvectors (for the triangular matrix) to eigenvectors corresponding to the original matrix. Recalling that the original matrix was transformed to a matrix on tri-diagonal form by $n-2$ orthogonal transformations $P_1, P_2, \dots, P_{n-2}$, it easily follows, that if z_{n-1} is an eigenvector for the triangular matrix then

$P_1 P_2 \dots P_{n-2} z_{n-1}$ is an eigenvector for the original matrix.

Putting $P_i P_{i+1} \dots P_{n-2} z_{n-1} = z_i$

one will obtain $P_i z_{i+1} = z_i$

and the wanted vector z_1 , is calculated in $n-2$ steps. Using the notation from 2, 1 (tridiagonalisation) one will get

$$z_i = z_{i+1} - \frac{u_i u_i^T}{h_i} z_{i+1} \text{ (because } P = I - \frac{u_i u_i^T}{h_i} \text{)}.$$

The non-zero elements of u_i are stored in the $t+1$ row ($t = n-i$) of the array a and $h_i = \text{sigma}/2$, where sigma denotes the square-sum of the elements in u_i (see 2.1).

3. Accuracy, Time and Storage Requirements

Accuracy: The accuracy in the eigenvalues depends on the value of the call parameter eps1 .

It easily follows from the description of the method part 2.2, that the error in any eigenvalue is bounded by $4 \times 10^{-10} \times \text{norm} + \text{eps1}$ where norm denotes the infinity norm of the triangular matrix.

For further information on this part see 4. Test and Discussion.

Time : This depends on the wanted accuracy, that is the term eps1, and first of all on the order n of the matrix equation. Generally the execution time will be proportional to $n \times n^2$. Using $\text{eps1} = 10^{-10}$ and denoted by

I : The execution time when all eigenvalues and all eigenvectors are calculated

II : The execution time when all eigenvalues but no eigenvectors are calculated.

III: The execution time when only the greatest eigenvalue and the corresponding eigenvector are calculated.

the greatest execution times (in sec.) obtained were as follows:

Order of the matrix	I	II	III
5	0.32	0.25	0.09
10	1.32	0.89	0.28
15	3.22	1.99	0.68
20	6.30	3.63	1.39
25	10.75	5.91	2.46

The following example illustrates the connection between the execution time and the value of eps1, where all eigenvalues and eigenvectors for a matrix of order 20 are calculated:

eps1 =	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}
Time =	4.88	5.15	5.44	5.74	5.94	6.16	6.30

Storage requirements: 9 segments of program

Typographical length : 149 lines

4. Test and Discussion

The procedure has been tested by several matrices, essentially the following four types (denoting by $a(i,j)$ the element in the i 'th row and the j 'th column and by n the order of the matrix in question):

Type I : $a(i,j) = a(j,i) = n - i + 1$. This matrix has well-separated eigenvalues given by

$$ev(i) = \frac{1}{2(1 - \cos(\frac{2i-1}{2n+1} \pi))} \quad i = 1, 2, \dots, n$$

Type II : $a(i,j) = a(j,i) = 1$ for all i, j .

All eigenvalues are 0 except one which is n

Type III : $a(i,j) = a(j,i) = 0$ for $i \neq j$ else 1.

All eigenvalues are -1 except one which is $n-1$.

Type IV : $a(i,j) = 0$ for $j < i-1$ and $j > i+1$.

$$a(i,i-1) = a(i,i+1) = 1.$$

$$a(i,i) = \left| \frac{n+1}{2} - i \right| \quad i = 1, 2, \dots, n.$$

The matrix has a number of extremely close, but not coincident eigenvalues.

When all eigenvalues and all eigenvectors are calculated, a measure for the error for the whole procedure is obtained by checking the identity $Ax_k = ev(k)x_k$ for each k .

Finding the largest deviation in any coordinate and using as testnorm the mean of these k numbers, the following results are obtained:

Matrix	Value of eps1			
	10^{-4}	10^{-6}	10^{-8}	10^{-10}
Type I order 10	3.1×10^{-5}	9.0×10^{-7}	1.5×10^{-8}	3.2×10^{-9}
Type I order 20	1.4×10^{-5}	1.5×10^{-6}	1.6×10^{-8}	3.9×10^{-8}
Type I order 25	2.0×10^{-4}	2.1×10^{-6}	3.2×10^{-8}	2.1×10^{-8}
Type II order 10	1.2×10^{-5}	3.3×10^{-7}	1.9×10^{-9}	3.8×10^{-10}
Type II order 20	3.5×10^{-5}	7.6×10^{-8}	3.1×10^{-9}	2.0×10^{-10}
Type II order 25	4.5×10^{-5}	3.5×10^{-7}	4.6×10^{-10}	6.4×10^{-10}
Type III order 10	1.2×10^{-5}	9.3×10^{-8}	1.4×10^{-9}	1.7×10^{-10}
Type III order 20	1.2×10^{-5}	6.7×10^{-8}	6.3×10^{-10}	4.3×10^{-10}
Type III order 25	5.4×10^{-6}	7.0×10^{-8}	1.5×10^{-9}	6.0×10^{-10}
Type IV order 11	2.1×10^{-3}	9.8×10^{-6}	2.1×10^{-7}	9.1×10^{-9}
Type IV order 15	6.7×10^{-3}	6.8×10^{-5}	6.0×10^{-7}	1.7×10^{-7}
Type IV order 21	1.5×10^{-2}	1.9×10^{-3}	6.3×10^{-4}	4.1×10^{-6}

The jacobi algorithm solves almost the same problem as householder; The only difference is, that the jacobi procedure necessarily calculates all the eigenvalues (and eigenvectors), while it is possible with the householder procedure only to calculate some of the eigenvalues (and eigenvectors). Calculating all eigenvalues and all eigenvectors and using in householder $\text{eps1} = 10^{-10}$ a comparison between the two procedures gave the following results:

Matrix	Testnorm	Testnorm	Time	Time
	for householder	for jacobi	for householder	for jacobi
Type I order 5	$1.4_{10}-9$	$0.8_{10}-9$	0.35	0.27
Type I order 10	$3.2_{10}-9$	$4.0_{10}-9$	1.33	2.02
Type I order 15	$2.2_{10}-8$	$1.0_{10}-8$	3.29	6.61
Type I order 20	$3.5_{10}-8$	$2.2_{10}-8$	6.30	14.92
Type I order 25	$2.1_{10}-8$	$3.2_{10}-8$	11.12	29.08
Type II order 5	$7.0_{10}-10$	$5.0_{10}-10$	0.20	0.07
Type II order 10	$1.6_{10}-10$	$0.1_{10}-10$	0.53	0.22
Type II order 15	$4.0_{10}-10$	$1.0_{10}-10$	1.13	0.55
Type II order 20	$4.5_{10}-10$	$1.6_{10}-10$	1.98	0.97
Type II order 25	$7.4_{10}-10$	$1.2_{10}-10$	3.46	1.38

Remembering that the matrices of type I have well-separated eigenvalues, and that the matrices of type II have all but one eigenvalue equal to zero, one might draw the following conclusion:

The procedure householder is to be preferred in case of matrices with separated eigenvalues, because of higher speed, or in cases, where only one or a few eigenvalues are wanted.

The procedure jacobi is to be preferred in case of matrices with coincident eigenvalues.

Example

We consider a symmetric matrix of order n . The term $m1$ denotes the number of the smallest, $m2$ the number of the greatest eigenvalue to be calculated. The eigenvectors are calculated only if the term $eps1$ is positive. Input is the value of the quantities n , $m1$, $m2$, $eps1$ and the lower triangular part of the matrix.

Testprogram

```
begin
  integer n, m1, m2, i, k;
  real eps1;
  boolean vect;
  read(in, n,m1,m2,eps1); vect:= eps > 0;
  begin
    array a(1:n*(n+1)/2), x(m1:m2, 1:n+2), ev(m1:m2);
    for i:= 1 step 1 until n*(n+1)/2 do read(in, a(i));
    householder(n, m1,m2,a,ev,x,eps1);
    write(out, <:Eigenvalues <10><10>:>);
    for i:= m1 step 1 until m2 do
      write(out, <<dd>, i, << -dddd.dddddddddd>, ev(i), <:<10>:>);
      if vect then
        begin
          write(out, <:<10> Eigenvectors<10>:>);
          for k:= m1 step 1 until m2 do
            begin
              write(out, <:<10>:>, <<dd>, k, <:<10>:>);
              for i:= 1 step 1 until n do
                write(out, << -dddd.dddddddddd>, x(k, i), <:<10>:>);
            end k;
          end vect;
        end;
      end;
    end;
```

For the matrix of order 5:

5	4	3	2	1
4	6	0	4	3
3	0	7	6	5
2	4	6	8	7
1	3	5	7	9

using m1 = 3, m2 = 5 and eps1 = 10^{-8} the complete output is:

Eigenvalue

3	4.848950119
4	7.513724158
5	22.406875316

Eigenvectors

```
3
  -0.547172796
    0.312569920
  -0.618112076
    0.115606593
    0.455493746
```

```
4
  -0.550961958
  -0.709440337
    0.340179132
    0.083410953
    0.265435679
```

```
5
    0.245877938
    0.302396039
    0.453214523
    0.577177152
    0.556384584
```

end

For the matrix of order 10:

10	9	8	7	6	5	4	3	2	1
9	9	8	7	6	5	4	3	2	1
8	8	8	7	6	5	4	3	2	1
7	7	7	7	6	5	4	3	2	1
6	6	6	6	6	5	4	3	2	1
5	5	5	5	5	5	4	3	2	1
4	4	4	4	4	4	4	3	2	1
3	3	3	3	3	3	3	3	2	1
2	2	2	2	2	2	2	2	2	1
1	1	1	1	1	1	1	1	1	1

using m1 = 1, m2 = 10 and eps1 = - π -10 the complete output is:

Eigenvalues

```
1      0.255679563
2      0.273786762
3      0.307978528
4      0.366208875
5      0.465233088
6      0.643104132
7      1.000000000
8      1.873023068
9      5.048917339
10     44.766068656
```

end

5. References

- (1) J.H. Wilkinson: Householders method for symmetric matrices. Numerische Mathematik 4, p. 354-361 (1962)
- (2) J.H. Wilkinson: Calculation of the eigenvalues of a symmetric diagonal matrix by the method of bisection. Numerische Mathematik 4, p. 362-367 (1962).
- (3) J.H. Wilkinson: Calculation of the eigenvectors of a symmetric tridiagonal matrix by inverse iteration. Numerische Mathematik 4, p. 368-376 (1962).
- (4) J.H. Wilkinson: Calculation of the eigenvectors of co-diagonal matrices, Computer Journal 1, p. 90-96 (1958).
- (5) J.H. Wilkinson, W. Bath, R.S. Martin: Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection Numerisch Mathematik 9, p. 386-393 (1967).
- (6) P. Naur: Eigenvalues and eigenvectors of real symmetric matrices, BIT 4, p. 120-130 (1964).

6. Algol text

```
householder = set 9  
householder = algol  
external
```

```
procedure householder(n,m1,m2,a,ev,x,eps1);  
value n,m1,m2,eps1;  
real eps1;  
array a,ev,x;  
integer m1,m2,n;
```

```
begin  
integer i,j,k,i0,j0,i1,t,t0,t1;  
real h,s,k1,sigma,at,bt,eps,b1,b11,norm,x1,x2,x0,u,v;  
array c(1:n),r(0:n),p,b,q(1:n+1),m(1:n+2);  
boolean vect;
```

```

eps:=0; j:=n*(n+1)/2;
for i:=1 step 1 until j do eps:=eps + abs(a(i));
eps:=(310-11)*eps/j;
for i:=1 step 1 until n-2 do
begin
  t:=n-i; t0:=t*(t+1)/2; t1:=t0 + t;
  sigma:=0;
  for k:=t0+1 step 1 until t1 do sigma:=sigma+a(k)*2;
  at:=a(t1);
  b(t+1):=bt:= if at>0 then-sqrt(sigma) else sqrt(sigma);
  if abs(bt)>eps then
  begin
    h:=sigma-at*bt; a(t1):=at-bt;
    for j:=1 step 1 until t do
    begin
      comment computation of pi;
      s:=0; j0:=(j-1)*j/2;
      for k:=1 step 1 until j do s:=s+a(j0+k)*a(t0+k);
      for k:=j+1 step 1 until t do s:=s+a(k*(k-1)/2+j)*a(t0+k);
      q(j):=s/h;
    end j;
    k1:=0;
    comment computation of ki;
    for j:=1 step 1 until t do k1:=k1+a(t0+j)*q(j);
    k1:=k1/2/h;
    comment computation of qi;
    for j:=1 step 1 until t do q(j):=q(j)-k1*a(t0+j);
    for j:=1 step 1 until t do
    begin
      comment computation of the i+1 matrix;
      j0:=(j-1)*j/2;
      for k:=1 step 1 until j do
        a(j0+k):=a(j0+k)-a(t0+j)*q(k)-a(t0+k)*q(j);
      end j;
    end abs(bt)>eps;
  end i;
  for i:=1 step 1 until n do c(i):=a(i*(i+1)/2);
  b(2):=a(2); b(1):=(n+1):=0;

  comment the eigenvalues ev(m1),ev(m1+1), . . . ,ev(m2)
  are now calculated;

  vect:=(if eps1<0 then false else true);
  eps1:=abs(eps1);
  norm:=0;
  for i:=1 step 1 until n do
  begin
    h:=abs(b(i))+abs(c(i))+abs(b(i+1));
    if norm<h then norm:=h;
    q(i):=b(i)*2;
  end i;
  for i:=m1 step 1 until m2 do p(i):= -norm;
  for k :=m2 step -1 until m1 do
  begin
    comment computation of the k eigenvalue;

```

```

for i:=m1 step 1 until k-1 do if p(i)>p(k) then p(k):= p(i);
x1:=p(k); x2:= ( if k<n then ev(k+1) else norm);
for x0:=(x1+x2)/2 while x2-x1>2×10-10×(abs(x1)+abs(x2))+eps1 do
begin
  h:=0; s:=1;
  for i:=1 step 1 until n do
  begin
    s:=c(i)-x0-(if s<0 then q(i)/s else abs(b(i))×1010);
    if s<0 then h:=h+1;
  end i;
  if h>=k then x2:=x0 else x1:=x0;
  if p(h+1)<0 then p(h+1):=x0;
  end x0;
  ev(k):=x0;
end k;
eps1:=1/2×eps1+4×10-10×norm;

```

```

if vect then
begin
  comment computation of the eigenvectors corresponding
  to the calculated eigenvalues;
  eps:= (3×10-11)×norm;
  for k:=m2 step -1 until m1 do
  begin
    comment the pivotal equations are calculated;
    u:=c(1)-ev(k); v:=b(2);
    if abs(v)<eps then v:=eps;
    for i:=1 step 1 until n-1 do
    begin
      bi:=b(i+1); if abs(bi)<eps then bi:=eps;
      bi1:=b(i+2); if abs(bi1)<eps then bi1:=eps;
      if abs(u)>abs(bi) then
      begin
        p(i):=u; q(i):=v; r(i):=0;
        m(i+1):=bi/u;
        u:=c(i+1)-ev(k)-m(i+1)×v; v:=bi1;
      end
      else
      begin
        p(i):=bi; q(i):=c(i+1)-ev(k);
        r(i):=bi1; m(i+1):=u/bi;
        u:=v-m(i+1)×(c(i+1)-ev(k));
        v:=-m(i+1)×bi1;
      end;
    end i;
    q(n+1):=q(n):=r(n):=x(k,n+1):=x(k,n+2):=h:=0;
    p(n):=if abs(u)>eps then u else eps;
    for i:=n step -1 until 1 do
    begin
      comment the first approximation;
      x(k,i):=(1-q(i)×x(k,i+1)-r(i)×x(k,i+2))/p(i);
      h:=h+x(k,i)×2;
    end;
  end;

```

```
h:=sqrt(h);
for i:=1 step 1 until n do x(k,i):=x(k,i)/h;
h:=0;
for i:=n step -1 until 1 do
begin
  comment the second approximation;
  x(k,i):=(x(k,i)-q(i)x(k,i+1)-r(i)x(k,i+2))/p(i);
  h:=h+x(k,i)xx2;
end;
h:=sqrt(h);
for i:=1 step 1 until n do x(k,i):=x(k,i)/h;
end k;

comment the calculated eigenvectors are now transformed
to eigenvectors corresponding to the original matrix;

for k:=m1 step 1 until m2 do
begin
  for j:=n-2 step -1 until 1 do
  begin
    t:=n-j; t0:=tX(t+1)/2; sigma:=0;
    for i:=1 step 1 until t do sigma:=sigma+a(t0+i)xx2;
    if sigma<>0 then
    begin
      s:=0;
      for i:=1 step 1 until t do s:=s+a(t0+i)x(k,i);
      s:=-2xs/sigma;
      for i:=1 step 1 until t do
        x(k,i):=x(k,i)+sxa(t0+i);
      end sigma<>0;
    end j;
  end k;
end vect;
end;
end;
```


invertsymb


```
boolean procedure invertsym(n, A);
```

1. Function and parameters.

```
boolean procedure invertsym(n, A);  
value n integer n; array A;
```

Function

The procedure inverts a symmetrical $n \times n$ matrix $M(1:n, 1:n)$ of which the lower part is stored as a one-dimensional array $A(1:n \times (n+1) // 2)$ so that

$$M(r, s) = M(s, r) = A(r \times (r-1) // 2 + s) \quad \text{for } 1 \leq s \leq r \leq n.$$

On return the inverse of M is found stored in A and the procedure is given the value true. This is only in case the call of the procedure has been a success. If it is a failure (i.e. if M is singular) the procedure has the value false, but even in this case the result M' found in A is with meaning, since M' will have the property that $M' \times B$ is a solution of the matrix equation $M \times X = B$ whenever this equation has a solution. Moreover, the degenerate elements may be found as those diagonal elements for which the corresponding rows and columns are identically zero.

Parameters

call parameter:

n integer. The order of M

call and return parameter:

$A(1:n \times (n+1) // 2)$ array. Must on entry contain the lower half of M , so that $M(r, s) = M(s, r) = A(r \times (r+1) // 2 + s)$. At return A will contain the inverse of M stored in the same way

return parameter:

invertsym boolean procedure. It is false if M is singular else true.

2. Mathematical Method.

The method is by Gauss-Jordan elimination using pivoting n times. In each step there are 3 cases.

Case 1: There is an index r , which has not been used as pivot index in an earlier step and for which the diagonal element $M(r, r)$ is $\neq 0$. Let E be the set of all such indices. A new pivot index is selected from E in the following way: For each r in E the quantity

$$m(r) = \max_{s \in E, s \neq r} \text{abs } M(r, s) / \text{abs } M(r, r) \quad r \in E$$

(maximum over s in E , $s \neq r$)

is computed, and the pivot index r is chosen arbitrarily among those indices which make $m(r)$ attain its minimum. A pivoting is carried out with $M(r, r)$ as pivot element, and in a boolean array $B(1:n)$ the r 'th element is set to false to indicate that this index cannot be used in later steps.

The pivoting means that the elements $M(i, k)$ are replaced by

$$\begin{aligned} M(i, k) &= M(i, r) \times M(r, i) / M(r, r) & \text{for } i \neq r \wedge k \neq r \\ M(r, k) &= M(r, k) / M(r, r) & \text{for } i \neq r \wedge k = r \\ - M(i, r) / M(r, r) & & \text{for } i = r \wedge k \neq r \\ 1 / M(r, r) & & \text{for } i = r \wedge k = r \end{aligned}$$

The result of this transformation is not a symmetrical matrix but

$$M(r, s) = \begin{aligned} &-M(s, r) \text{ if } r \text{ has been pivot index, and } s \text{ has not} \\ &\text{(i.e. } B(r) = \text{false, } B(s) = \text{true)} \end{aligned}$$

$$M(s, r) \text{ in all other cases.}$$

Only the lower part of M is stored in A , since the upper part may be reestablished by means of B .

Case 2: $M(r, r) = 0$ for all r not used as pivot indices before, but there are elements $M(r, s) \neq 0$ outside the diagonal (i.e. for $r \neq s$) for some r and s not used as pivot indices before. In this case two new pivot indices r and s have to be chosen. First s is chosen arbitrarily among such possible indices. Next to choose r , let E be the set of the indices $r \neq s$ not used before as pivot indices and for which $M(r, s) \neq 0$. For each r in E the quantity

$$m(r) = \max_{k \neq r, s} \text{abs } M(r, k) / \text{abs } M(r, s)$$

where k runs over all indices $\neq r$ and $\neq s$ not used as pivot indices. Now r is chosen such that $m(r)$ attains its minimum (which possibly is zero). In the boolean array B the r 'th and s 'th element are set to false to indicate that these indices may not be used in the following steps. Now a pivoting is carried out with $M(r, s)$ and $M(s, r)$ as pivot elements. This means that the matrix elements $M(i, k)$ are replaced by

$M(i, k) - M(i, r) \times M(s, k) / M(r, s) - M(i, s) \times M(r, k) / M(r, s)$	for $i \neq r \wedge i \neq s \wedge k \neq r \wedge k \neq s$
$M(i, r) / M(r, s)$	for $i \neq r \wedge k = s$
$-M(r, k) / M(r, s)$	for $i = s \wedge k \neq r$
$M(i, s) / M(r, s)$	for $k = r \wedge i \neq s$
$-M(s, k) / M(r, s)$	for $i = r \wedge k \neq s$
$1 / M(r, r)$	for $i = r \wedge k = s$

As in case 1 the result is not a symmetrical matrix, but the upper part may be reestablished in the same manner from the lower part.

Case 3: There are no matrix elements $M(r, s) \neq 0$, where r and s have not been pivot indices. In this case the submatrix of M obtained by taking only the indices not used as pivot indices is identical zero. This means that M is singular. The value of the procedure is then set to false and the remaining rows and columns are set to zero, so that the result delivered in A may have the property mentioned in the section above.

If it is possible to do the pivoting n times without ever entering case 3 then M is nonsingular. So the value of the procedure is set to true, and the result of the algorithm delivered in A is the inverse of M .

3. Accuracy, time and storage requirement

Accuracy

In practice the relative error measured as $||AX - B|| / ||X||$ has been found to be about 10^{-10} . This is not an exact error bound. Theoretical error bounds are discussed in detail in literature, see e.g. Forsythe and Moler (ref).

Time: $.14 \times (n+1) \times 3$ mS

Storage requirement

Program length: 6 segments

variables: $23 + 2.5 \times n$ words in stack.

Typographical length: 145 lines, 6 segments.

4. Test and discussion

The procedure is intended for use in such cases where the total matrix M is too big for the available store. A program using `decompose` and `solve` will be faster than a program using `invert_sym` even if the program must generate the matrix M from the half matrix A .

The procedure has been tested by some random matrices and by a representative set of singular matrices.

The following program will read n , A and write out the inverse of A :

Program to read a symmetrical matrix and output its inverse.

```
begin integer n, i, j, k, l;
  read(in, n);
  begin array A(1:(n*(n+1)) shift (-1));
    read(in, A);
    if -, invertsym(n, A) then write(out, <:<10> A is singular:>);
    write(out, <:<10>:>);
    for i:= 1 step 5 until n do
      begin
        j:= if i + 4 < n then i + 4 else n;
        for k:= i step 1 until j do write(out, <<_____ddd>, k);
        for k:= i step 1 until n do
          begin
            write(out, <:<10>:>, <<ddd>, k);
            j:= if i + 4 < k then i + 4 else k;
            for l:= i step 1 until j do
              write(out, <<_d.ddd_ddd_ddd>, A((k*(k-1)) shift (-1) + 1));
            end k;
            write(out, <:<12<10>:>)
          end i
        end A
      end program;
```

5. Reference

Georg Forsythe and Cleve B. Moler: Computer solution of Linear Algebraic Systems, Prentice-Hall, Inc. (1967).

6. Algorithm

```
invertsym = set 6
invertsym = algol
external
```

```
boolean procedure invert_sym(n,A);
message invert sym, 13 11 69, RCSL 53-M5;
    value n; integer n; array A;
begin integer i,j,k,r,s,t,r1,s1,p;
    real m, aj,ak,ar,aj1,mp;
    boolean bj,mf;
    array M(1:n); boolean array B(1:n);

    i:=0;
    for p:= 1 step 1 until n do
    begin
        m:=0;
        for k:=p-1 step -1 until 1 do
        begin
            if abs A(i+k)> m then m:= abs A(i+k);
            if abs A(i+k)>M(k) then M(k):=abs A(i+k)
        end k;
        M(p):= m; B(p):= true; i:=i+p
    end p;
    t:=n; mp:=-1; mf:=true;
    for j:=n step -1 until 1 do
    begin
        if mf then
        begin
```

```

    if abs A(i)>M(j)>mp then
    begin
        if M(j)=0 then mf:=false else mp:=abs A(i)/M(j); p:=j
    end abs A(i)>M(j)>mp
    end mf;
    M(j):=0; i:=i-j
end j;
next_pivot:
s:=p; r:=(s*(s-1))shift(-1);
if mp>0 | -,mf then
begin comment this is the normal case where
    there has been found a pivot-element
    in the diagonal;
B(s):=false; t:=t-1; ar:=A(r+s):=1/A(r+s); mp:=-1; mf:=true;
for j:=n step -1 until 1 do if j<>s then
begin
    i:=(j*(j-1))shift(-1); bj:=B(j); m:=M(j);
    aj:=if s<j then A(i+s)*ar else
        (if bj then ar else -ar)*A(r+j);
    for k:= 1 step 1 until j do if k<>s then
    begin
        ak:=A(k+i):=A(k+i)-(if k<s then A(k+r)*aj else
            (if B(k) then aj else -aj)*A((k*(k-1))shift(-1)+s));
        if bj then begin if mf then begin if k<j then
            begin
                if abs ak>M(k) then M(k):= abs ak;
                if abs ak>m then begin if B(k) then m:=abs ak end;
            end end end bj
        end k;
        if s<j then A(i+s):=aj else A(r+j):=if bj then -aj else aj;
        if bj then
        begin
            if mf then
            begin
                if abs ak>m*mp then
                begin

```

- 7 -

```

        if m=0 then mf:=false else mp:=abs ak/m; p:=j
        end abs ak>m*mp
    end mf;
    M(j):=0
end bj
end j;
goto next_pivot
end mp>0 | -,mf;
if mp=0 then
begin comment this is the exceptional case where
    all diagonal-elements are zero;
    B(s):=false; m:=0;
    for j:=s-1 step -1 until 1 do if B(j) then
begin
    i:=(j*(j-1))shift(-1); ak:=0;
    for k:= s-1 step -1 until 1 do if B(k) then
begin
        if abs A(if k<j then k+i else j+(k*(k-1))shift(-1))>ak then
            ak:=abs A(if k<j then k+i else j+(k*(k-1))shift(-1))
        end k;
        if abs A(r+j)>m*ak then
begin
            s1:=j;
            if ak=0 then goto L;
            m:=abs A(r+j)/ak
        end
    end j;
L: t:=t-2; r1:=(s1*(s1-1))shift(-1);
    ar:=A(r+s1):=1/A(r+s1); B(s1):=false; mp:=-1; mf:=true;
    for j:=n step -1 until 1 do if j<s\j<s1 then
begin
        i:=(j*(j-1))shift(-1); bj:=B(j); m:=M(j);
        aj:=if s<j then A(i+s)*ar else
            (if bj then ar else -ar)*A(r+j);
        aj1:=if s1<j then A(i+s1)*ar else
            (if bj then ar else -ar)*A(r1+j);

```

```

for k:=1 step 1 until j do if k<=s & k<=s1 then
begin
  ak:=A(i+k):=A(i+k)-(if k<=s then A(r+k)*aj1 else
    (if B(k) then aj1 else -aj1)*A((k*(k-1))shift(-1)+s))
    -(if k<=s1 then A(r1+k)*aj else
      (if B(k) then aj else -aj)*A((k*(k-1))shift(-1)+s1));
  if bj then begin if mf then begin if k<j then
begin
  if abs ak>m then begin if B(k) then m:=abs ak end;
  if abs ak>M(k) then M(k):= abs ak
end end end bj
end k;
if s<j then A(i+s):=aj1 else
  A(r+j):=if bj then -aj1 else aj1;
if s1<j then A(i+s1):=aj else
  A(r1+j):=if bj then -aj else aj;
if bj then
begin
  if mf then
begin
  if abs ak>m*mp then begin
    if m=0 then mf:=false else mp:=abs ak/m; p:=j
end abs ak>mp
end mf;
M(j):=0
end bj
end j;
goto next_pivot
end m=0;
invert_sym:= t=0;
if t<=0 then
begin
  i:=0;
  for j:=1 step 1 until n do
begin
  for k:=1 step 1 until j do if B(j) | B(k) then A(i+k):=0;
  i:=i+j
end j
end t<=0
end invert_sym;

```

comment

Parameters

call parameter:

n integer. The order of M

call and return parameter:

A(1:n×(n+1)//2) array. Must on entry contain the lower half of M,
so that $M(r, s) = M(s, r) = A(r \times (r+1) // 2 + s)$.
At return A will contain the inverse of M
stored in the same way

return parameter:

invertsym boolean procedure. It is false if M is singular
else true;

jacobi

jacobi

Jacobi calculates all the eigenvalues and, if desired, the corresponding eigenvectors of a symmetric matrix by the method of Jacobi.

Procedure head:

```
real procedure jacobi(a,lambda,x,vect,maxscan);  
value vect,maxscan;  
array a,lambda,x;  
boolean vect;  
integer maxscan;
```

Call parameters:

a : a real array containing the given matrix.

vect : (boolean) . If vect is true, the eigenvectors will be calculated.

maxscan : (integer or real). If the value of maxscan is > 0, at most this number of scans are performed in the procedure. If the value is 0, no limitation is imposed on the number of scans. (see section 2. Method).

Return parameters:

jacobi : (real). Contains the number of rotations in the last two bytes and the number of scans in the first two bytes. If the procedure exits because the maximum number of scans is reached, the negative number of scans is stored. Hence the sign of the procedure value reveals its success.

lambda : (real array). Contains on exit the calculated eigenvalues.

x : (real array). If the eigenvectors are wanted, they are stored as column-vectors in x. The eigenvector associated with the eigenvalue lambda(i) is stored in x(.,i).

Parameter check:

The orders of the matrix a and the vectors x and lambda are not given as parameters, but are checked by the procedure in this way:

First, the upper subscript bound of the array lambda is assigned to the order. Then it is checked that the arrays are declared

a(1:order,1:order)	(1)
lambda(1:order)	(2)
x(1:order,1:order)	(4)

(this last check is performed only if the eigenvectors are wanted; in fact, if vect=false, x can be any real array and is never touched).

In case of error, the execution is terminated by the error-message on current output

jacobi <error-number>

where <error-number> is the sum of the numbers attached to each wrong array as stated above. Thus the declaration

```
a(1:order,0:order)
lambda(1:order)
x(1:order)
```

yields the error-number 5 if vect=true, otherwise 1.

This initial check of the parameters implies that there is no need for index-check in the procedure.

2. Method.

The method consists of a number of scans of all the super-diagonal elements of the matrix. If the element in question is greater in absolute value than a certain threshold (approximately the current root-mean-square of all super-diagonal elements), a rotation is performed, so designed that this element becomes zero.

The exit condition is that the current threshold is less than

$5 \times 10^{-13} \times \text{initial threshold}$ (or that the maximum number of scans is reached).

Since the procedure converges at least quadratically, little time is saved by reducing the accuracy.

Just before exit, the super- and main-diagonal elements are reestablished so that the matrix is unchanged on exit.

For further details, see [1].

3. Accuracy, Time and Storage Requirement.

Accuracy: The relative error of the eigenvalues and, if the eigenvectors are calculated, the greatest element of $(x^t x - I)$ is unlikely to exceed $n \times$ (the relative machine accuracy, appr. 3×10^{-11}). This applies also to the greatest element of $(a x x - x x L) / \max(\lambda)$. However, if the magnitudes of the eigenvalues are highly different it may happen that the eigenvalues of low magnitude are determined less accurate. It can be shown (see [1]) that the absolute error of the eigenvalues is bounded by

$$2 \times \|L\| / \sqrt{1-nI} \times (nI / (1-\sqrt{1-nI}) + nA),$$

where

$$\begin{aligned} L &= \text{diag}(\lambda) \\ nI &= \|x^t x - I\| ; x^t \text{ is } x \text{ transposed.} \\ nA &= \|a x x - x x L\| / \|L\|. \end{aligned}$$

Time : Generally proportional to $n \times 3$ when n is large (see section 4. Test and Discussion).

Storage requirement: 5 segments of program
18 local real variables.

4. Test and Discussion.

Several matrices have been tried by the test program (or slightly modified versions) at the end of this section.

The table below shows the type and order of the matrix in question, the number of scans, the number of rotations, the time consumed by the procedure (in sec.), and the norms nI and nA as defined in section 3. (the infinity-norm is used.)

type	n	scan	rotation	time	nI	nA
a	10	14	180	1.9	1.7×10^{-9}	8.3×10^{-10}
a	20	17	796	14.6	7.1×10^{-9}	2.1×10^{-9}
b	15	12	327	4.8	1.2×10^{-9}	9.4×10^{-10}
c	9	4	12	.15	2.5×10^{-10}	9.2×10^{-11}
d	8	11	69	.68	7.6×10^{-10}	6.1×10^{-10}

The types represent the following matrices:

a) HBH-matrices. The general element of a n-th order matrix is

given by $a(i,j)=a(j,i)=n-i+1$. The eigenvalues are

$$l(i)=0.25/\sin((2i-1)\pi/(4n+2))\times 2$$

b) The matrix

$$a(i,j)=a(j,i)=\text{if } i < j \text{ then } 1 \text{ else } 10 \times (i-1)$$

c) The matrix

$$a(i,j)=a(j,i)=\text{if } i=j \text{ then } 0 \text{ else } 1$$

All eigenvalues are -1 except for one which is n-1.

d) The matrix and the complete output of the testprogram for this case are:

matrix:

```

611
196 899
-192 113 899
407 -192 196 611
-8 -71 61 8 411
-52 -43 49 44 -599 411
-49 -8 8 59 208 208 99
29 -44 52 -23 208 208 -911 99

```

```

scans=    11    rotations=    69    time=    0.68 sec.
nI = 7.6e-10    nA = 6.1e-10

```

eigenvalues:

```

1.0200490189e 3
1.00000000002e 3
1.00000000003e 3
9.8048646596e -2
2.5609435927e -9
1.02000000004e 3
1.0199019515e 3
-1.0200490187e 3

```

end

Test program:

```

test1 jacobi
begin integer n;
  for underflows:=-1 while read(in,n)>0 do
    begin real x,eli,ela,maxl,normi,norma,la,layout,t0,t1,si,sa;
      array a,t(1:n,1:n),l(1:n);
      integer i,j,k,layno;

      read(in,layno);
      for i:=1 step 1 until n do
        for j:=1 step 1 until i do
          begin read(in,x);
            a(i,j):=a(j,i):=x
          end;

```

```

t0:=systime(1,0,1a);
for i:=0,i+1 while t1<t0+2.56 do
begin x:=jacobi(a,1,t,true,0);
  t1:=systime(1,0,1a)
end;
t0:=(t1-t0)/i;

maxl:=normi:=norma:=0;
for i:=1 step 1 until n do
begin si:=sa:=0;
  for j:=1 step 1 until n do
  begin eli:=ela:=0;
    la:=l(j);
    for k:=1 step 1 until n do
    begin eli:=eli+t(k,i)*t(k,j);
      ela:=ela+a(i,k)*t(k,j)
    end k;
    if i=j then eli:=eli-1;
    ela:=ela-t(i,j)*la;
    si:=si+abs eli;
    sa:=sa+abs ela
  end j;
  if si>normi then normi:=si;
  if sa>norma then norma:=sa;
  if abs l(i)>maxl then maxl:=abs l(i)
end i;

layout:=real(case layno of(<<d>,<<dd>,<<-dd>,<<-ddd>,<<-dddd>,
  <<d>,<<d>,<<-d.ddd>,<<-d.ddddd>));
write(out,<:<12>matrix:>);
for i:=1 step 1 until n do
begin k:=0;
  write(out,<:<10>:>);
  for j:=1 step 1 until i do
  begin write(out,string layout,a(i,j));
    k:=k+layno;
    if k>70 and j<i then
    begin write(out,<:<10>:>,false add 32,layno);
      k:=layno
    end
  end
end write matrix;
write(out,<:<10><10>scans=>,<<-dddd>,x shift(-24) extract 24,
  <: rotations=>,x extract 24,
  <: time=>,<< ddd.00>,t0,<: sec:>,
  <:<10>nI =>,<< d.dn-dd>,normi,
  <: nA =>,norma/maxl,
  <:<10><10>eigenvalues:<10>:>);
for i:=1 step 1 until n do
  write(out,<<-d.ddddddddn-dd>,l(i),<:<10>:>)
end read n
end

```

5. References

- [1] Kahan, W. and Green, D. : Eigenvalues and Eigenvectors of a Real Symmetric Matrix. (Unpublished but copies of the paper are achievable on demand)

6. Algorithm

```
jacobi=set 5
jacobi=algol index.no

external
real procedure jacobi(a,lambda,x,vect,maxscan);
message jacobi, version 20.11.69, RCSL NO: 55-D61;
value vect,maxscan;
integer maxscan;
boolean vect;
array a,lambda,x;
begin real eps,t,ave,s,u,thresh,dlow,d,c,aij,ajj;
  integer i,j,ii,jj,jl,n,nrscan,nrrot;
  boolean again;

  i:=if system(3,n,lambda)<>1 then 2 else 0;
  j:=system(3,ii,a);
  if j<>n+1 or ii<>n*(n+1) then i:=i+1;
  j:=system(3,ii,x);
  if (j<>n+1 or ii<>n*(n+1)) and vect then i:=i+4;
  if i>0 then system(9,i,<:<10>jacobi :>);

  if vect then
    for i:=1 step 1 until n do
      begin x(i,i):=1;
        for j:=i+1 step 1 until n do x(i,j):=x(j,i):=0
      end x:=identity;

  d:=0;
  for i:=1 step 1 until n do
    begin lambda(i):=a(i,i);
      for j:=i+1 step 1 until n do d:=d+a(i,j)*x2
    end i;

  nrscan:=nrrot:=0;
  if d>0 then
    begin dlow:=p-7*d;
      ave:=(n-1)*n*0.55;
      thresh:=sqrt(d/ave);
      eps:=5*p-13*thresh;

scan:again:=false;
  nrscan:=nrscan+1;
  for i:=n-1 step -1 until 1 do
    for j:=i+1 step 1 until n do
      begin comment scan;
        aij:=a(i,j);
        if abs aij >= thresh then
          begin ajj:=a(j,j);
            s:=ajj-a(i,i);
            t:=abs aij;
            if s+t<>s then
              begin comment rot<>0;
                again:=true;
                nrrot:=nrrot+1;
                if abs s<=p-6*t then s:=c:=0.70710678118 else
                  begin t:=aij/s;
                    s:=0.25/sqrt(t*x2+0.25);
                    c:=sqrt(s+0.5);
                    s:=2*t*xs/c
                  end rot<>p1/4;
```

```

for ii:=1 step 1 until i do
begin t:=a(ii,i); u:=a(ii,j);
  a(ii,i):=cxt-sxu;
  a(ii,j):=sxt+cxu
end;
jl:=j-1;
for ii:=i+1 step 1 until jl do
begin t:=a(i,ii); u:=a(ii,j);
  a(i,ii):=cxt-sxu;
  a(ii,j):=sxt+cxu
end;
a(j,j):=sxaij+cxajj;
a(i,i):=cxa(i,i)-sx(cxaij-sxajj);
for ii:=j step 1 until n do
begin t:=a(i,ii); u:=a(j,ii);
  a(i,ii):=cxt-sxu;
  a(j,ii):=sxt+cxu
end;

if vect then
for ii:= 1 step 1 until n do
begin t:=x(ii,i); u:=x(ii,j);
  x(ii,i):=cxt-sxu;
  x(ii,j):=sxt+cxu
end;

d:=d-aijxx2;
if d<dlow then
begin d:=0;
  for ii:=n-1 step -1 until 1 do
  for jj:=ii+1 step 1 until n do
    d:=d+a(ii,jj)xx2;
  dlow:=n-7xd
end;

thresh:=sqrt(d/ave);
if thresh<eps then goto quit
end rotation
end aij
end scan;
if again and (maxscan>0=>maxscan>nrscan) then goto scan;

if again then nrscan:=-nrscan;

quit:for i:=1 step 1 until n do
begin t:=a(i,i);
  a(i,i):=lambda(i);
  lambda(i):=t;
  for j:=i+1 step 1 until n do a(i,j):=a(j,i)
end i
end d>0;
jacobi:=0.5 add nrscan shift 24 add nrrot
end jacobi;

```

comment

Call parameters:

a : a real array containing the given matrix.
vect : (boolean) . If vect is true, the eigenvectors
will be calculated.
maxscan : (integer or real). If the value of maxscan is > 0,
at most this number of scans are performed in the
procedure. If the value is 0, no limitation is imposed
on the number of scans.

Return parameters:

jacobi : (real). Contains the number of rotations in the last two bytes and the number of scans in the first two bytes. If the procedure exits because the maximum number of scans is reached, the negative number of scans is stored. Hence the sign of the procedure value reveals its success.

lambda : (real array). Contains on exit the calculated eigenvalues.

x : (real array). If the eigenvectors are wanted, they are stored as column-vectors in x. The eigenvector associated with the eigenvalue lambda(i) is stored in x(.,i);

minimum

1. Function and parameters

Let F denote a real, twice differentiable function in n variables, and suppose that the first order derivatives of F are given analytically (that is, as expressions depending upon the n variables). Suppose that in a given area the function is bounded below and has a minimum. From a reasonable good starting point the procedure finds this minimum by finding a point at which all the first order derivatives are zero (that is, smaller than a prescribed quantity).

Procedure head:

```
minimum(n, i, x, F, delta, eps, point);  
value n;  
integer i, n;  
real eps, F, delta;  
array x, point;
```

Call parameters:

n: the number of variables for the given function.

Call/Return parameters:

```
point:       a real array point(1:n);  
             at entry point contains the starting point for the  
             procedure;  
             at exit point contains the coordinates of the point  
             at which the minimum is obtained;
```

eps: a real quantity affecting the precision to which the
 minimum is calculated. Consider the norm of the vector
 consisting of the first order derivatives. If this
 norm is smaller than eps, then the procedure will stop;
 at exit eps contains the norm of the vector described
 above.

Return parameters:

minimum: the value of the obtained minimum;

Other parameters:

F: a real procedure denoting the given function. In a program in which the procedure minimum is called, F must be declared in the following way:

```
real procedure F(x);  
array x;  
F:= the given expression;
```

delta: a real procedure delta(i, x) denoting for each i the partial derivative of F with respect to the variable x(i);
In a program in which the procedure minimum is called, delta must be declared in the following way:

```
real procedure delta(i, x);  
integer i;  
array x;  
delta:= case i of (... , ... , ...);
```

In the parenthesis there must be n expressions, where the i-th expression denotes the partial derivative of F with respect to the variable x(i);

2. The method

Let F denote a function in n variables, and let x denote the n-dimensional point with coordinates (x(1), x(2), ..., x(n)). F is said to have a minimum at a point x0, if there exist a small area including x0, in which the value of F at each point is greater than F(x0). Most of the various methods for finding a minimum for a function in n variables has one idea in common: They are all iterative processes based upon a rule, which for each point specifies a certain direction in which the next point of the process is to be found, and for each such direction specifies how to find the next point. Now, suppose that

the function is differentiable. By the gradient of F at the point x - denoted $\text{gradient}(x)$ - we understand the n -dimensional vector, which as the i -th coordinate has the partial derivative of F with respect to $x(i)$ at the point x .

The method used in the following program is essentially based upon to papers of A.A. Goldstein ((2), (3)). We suppose, that the function is twice differentiable and that the gradient is given analytically. It is well known, that the gradient will vanish at a minimum point.

Let the points of the iterative process be denoted $x_1, x_2, x_3, \dots, x_k, \dots$, where x_1 is given by the input array point.

For each k $f_i(x_k)$ denotes the n -dimensional vector which terminates the new direction.

We choose $f_i(x_1) = \text{gradient}(x_1)$.

For each k the number $h(k)$ is defined as:

$$h(k) = r \times \text{norm}(n, f_i(x_k)).$$

r is calculated at the beginning of the program in such a way that $h(1) < 1/5$.

norm is denoting the ordinary n -dimensional Euklidian norm.

Then the algorithm, at each point x_k , consists of the following two steps:

1. DIRECTION:

We compute an $n \times n$ matrix, which is an approximation to the matrix consisting of the second order derivatives of F .

For each j let $F(j)$ denote the vector, which has the j -th coordinate equal to 1 and the others equal to zero.

We then compute the matrix $Q(x_k)$ which has the j -th column equal to $(\text{gradient}(x_k + h(k) \times F(j)) - \text{gradient}(x_k))/h(k)$.

If the matrix $Q(x_k)$ is singular (it has no inverse) then we define the new direction $f_i(x_k)$ by

$$f_i(x_k) = \text{gradient}(x_k).$$

Suppose now, that $Q(x_k)$ has an inverse, which we denote $P(x_k)$.

If $(\text{gradient}(x_k), P(x_k) \times \text{gradient}(x_k)) > 0$

(where $(\quad, \quad)$ denotes the ordinary innerproduct) then we define $f_i(x_k)$ by

$$f_i(x_k) = P(x_k) \times \text{gradient}(x_k).$$

If $(\text{gradient}(x_k), P(x_k) \times \text{gradient}(x_k)) \leq 0$
then we define $f_i(x_k)$ by
 $f_i(x_k) = \text{gradient}(x_k)$.

2. KONSTANT:

The next point in the process is now obtained on the form

$$x_k - g_k \times f_i(x_k)$$

where g_k is a constant calculated as follows:

Let $\text{product} = (\text{gradient}(x_k), f_i(x_k))$.

Let $f_1 = F(x_k)$.

Let $f_2 = F(x_k - g_k \times f_i(x_k))$.

Then g_k is calculated such that

$f_2 < f_1$ and $(f_1 - f_2) < g_k \times \text{product}$.

It can be proved, by using the Taylor formula, that such a g_k always exists, and that x_k calculated in this way will converge to a minimum-point for $F((2), (3))$. From a numerical point of view however, g_k might fail to exist, and in this case the procedure will stop.

3. Accuracy, Time and Storage Requirements

Accuracy: As measure of accuracy we use the norm of the gradient. If the procedure succeeds, then at the end this norm is smaller than the call parameter eps .

Time: This depends on the wanted accuracy and first of all on the problem in question, so it is not possible to give general rules for this. (See 4. Test and Discussion).

Storage requirements: 10 segments of program

Typographical length: 248 lines.

4. Test and Discussion

The procedure have been tested on several functiones among which we describe the two most difficult problems:

1. Minimising the function in two variables

$$F = 100 \times (x(2) - x(1) \times x(2)) \times x(2) + (1 - x(1)) \times x(2)$$

2. Finding a solution to the following three non-linear equations:

$$\sin(x(1) \times x(2)) + \exp(x(2)) \times x(3) - 4 = 0$$

$$x(1) + x(2) + x(3) - 3 = 0$$

$$x(1) + x(2) \times x(2) + x(3) \times x(3) - 14 = 0$$

This is done by minimising the square-sum of the three equations.

First we consider the problem 1:

The function F has minimum at the point (1, 1) with functionvalue 0.

Starting at the point $x(1) = -1.2$ and $x(2) = 1$ and using different values of the term eps, the following results were obtained:

	Value of eps			
	10^{-4}	10^{-6}	10^{-8}	10^{-10}
Minimum	0.999999592 0.999999183	0.999999592 0.999999183	1.000000000 1.000000000	1.000000000 1.000000000
Fc.-value	0.000000000	0.000000000	0.000000000	0.000000000
Gr.-norm	$7.3 \cdot 10^{-5}$	$4.2 \cdot 10^{-7}$	$5.1 \cdot 10^{-8}$	$1.7 \cdot 10^{-10}$
Ex.-time	0.76	0.75	0.75	0.75

(the execution time is in seconds).

It follows, that the procedure succeeds in all 4 situations, and that smaller values of eps does not affect the execution time. This last observation however can not be stated in general, (see below under problem 2).

Using $\text{eps} = 10^{-8}$ and using different starting points the following results were obtained:

Starting-	-1.200000000	0.000000000	-0.500000000	2.000000000
point	1.000000000	1.000000000	-0.500000000	0.250000000
Execution-				
time	0.75	0.49	0.71	0.80

In all 4 situations the minimum was obtained at the point:

1.000000000
1.000000000

with the functionvalue 0.000000000 and gradient norm $5.1 \cdot 10^{-8}$.

Again the procedure succeeds in all 4 situations.

Next, consider the problem 2 in three variables. Starting at the point $x(1) = 0$, $x(2) = 0$, $x(3) = 2.5$ and using different values of the term ϵ , the following results were obtained:

	Value of ϵ			
	10^{-4}	10^{-6}	10^{-8}	10^{-10}
Minimum	0.097831561 0.512917627 2.389250732	0.097830233 0.512919004 2.389250762	0.097830224 0.512919014 2.389250762	0.097830224 0.512919014 2.389250762
Fc.-value	0.000000000	0.000000000	0.000000000	0.000000000
Gr.-norm	$5.8 \cdot 10^{-5}$	$3.9 \cdot 10^{-7}$	$3.2 \cdot 10^{-8}$	$3.2 \cdot 10^{-8}$
Ex.-time	1.96	2.55	3.81	3.97

It follows, that the procedure succeeds in the first three situations, but that it is not possible to make the gradient norm smaller than $3.2 \cdot 10^{-8}$, so in this sense the procedure does not succeed in the last situation. In this case smaller values of ϵ gives greater execution time, even if the obtained minimumpoints are practically the same in the last three cases.

Using $\epsilon = 10^{-8}$ and using different starting points the following results were obtained:

Starting-	0.000000000	0.000000000	0.500000000	1.000000000
point	0.000000000	0.000000000	1.000000000	1.000000000
	2.500000000	1.000000000	2.000000000	1.000000000
Execution-				
time	3.81	1.97	3.09	2.45

In all 4 situations the minimum was obtained at the point:

0.097830223
0.512919014
2.389250762

with the functionvalue 0.000000000 and gradient norm $3.2 \cdot 10^{-8}$

It follows, that the procedure succeeds in all 4 situations.

Example

Consider the function

$$F = 100 \times (x(2) - x(1) \times x(2)) \times x(2) + (1 - x(1)) \times x(2)$$

Starting at the point $x(1) = -1.2$ and $x(2) = 1$ the following program might be used to find the minimum of F:

Testprogram

```
begin
  integer i, j;
  real a, eps;
  array x, point(1:2);

  real procedure F(x);
  array x;
  F:= 100 × (x(2) - x(1)×x(2)) × x(2) + (1 - x(1)) × x(2);

  real procedure delta(i, x);
  integer i;
  array x;
  delta:= case 1 of (-400×x(1) × (x(2) - x(1)×x(2)) - 2 × (1 - x(1)),
                    200 × (x(2) - x(1)×x(2)));

  point(1):= -1.2; point(2):= 1; eps := 10-8;
  a:= minimum(2, 1, x, F(x), delta(1, x), eps, point);
  write(out, <:Minimum obtained at the point <10>:>);
  for j:= 1 step 1 until 2 do
    write(out, <:<10>:>, <<-dddd.dddddddddd>, point(j));
    write(out, <:<10×10> Minimumvalue =:>, <<-dddd.dddddddddd>, a);
    write(out, <:<10×10> Gradient norm=:>, <<-d.d10-dd>, eps);
end;
```

This will give the following output:

Minimum obtained at the point

1.000000000

1.000000000

Minimumvalue = 0.000000000

Gradient norm = 1.0' -8

end

In the program we use a boolean procedure inverse to find the inverse (if it exist) of an $n \times n$ matrix.

The procedure is based upon Simpel Gaussian illimination and is only introduced in order to make the program complete. One could use any other procedure of this sort, for ex. decompose-solve from RC mathematical procedure library.

Since a minimum of the function F is a maximum of the function $-F$, the procedure will of course be able to find maximum as well as minimum.

5. References

- (1) D. Fletcher and M.J.D. Powell:
A rapidly convergent descent method for minimisation.
Comput. Journal 6 p. 163-168 (1963)
- (2) A.A. Goldstein: On steepest descent.
Journal Siam Control Vol. 3 No 1 p 147-151 (1965)
- (3) A.A. Goldstein and J.F. Puce:
An effective algorithm for minimisation
Numerische Mathematik 10 p. 184-189 (1967)
- (4) E. Isaacson and H.B. Keller:
Analyses of numerical Methods
John Wiley and Sons, Inc. (1966)

6. Algol text

```
minimum = set 10
```

```
minimum = algol
```

```
external
```

```
real procedure minimum(n,i,x,F,delta,eps,point);
```

```
value n;
```

```
integer i,n;
```

```
real eps,F,delta;
```

```
array x,point;
```

```
begin
```

```
integer j;
```

```
real h,g,g1,gamma,r,f1,f2,f3,product,k,s;
```

```
array psi,y,z,b(1:n),p,q(1:n,1:n);
```

```
real procedure norm(n,a);
```

```
value n;
```

```
integer n;
```

```
array a;
```

```
begin
```

```
comment this is the ordinary norm in the n-dimensional Euklidian  
space;
```

```
real h;
```

```
h:=0;for i:=1 step 1 until n do h:=h+a(i)××2;
```

```
norm:=sqrt(h);
```

```
end;
```

```
real procedure innerproduct(n,a,b);
```

```
value n;
```

```
integer n;
```

```
array a,b;
```

```
begin
```

```
comment this is the ordinary innerproduct in the n-dimensional
```

```
Euklidian space;
real h;
h:=0; for i:=1 step 1 until n do h:=h+a(i)*b(i);
innerproduct:=h;
end;
```

```
procedure equal(n,a,b);
value n;
integer n;
array a,b;
begin
  comment the procedure identifies two arrays;
  for i:=1 step 1 until n do b(i):=a(i);
end;
```

```
boolean procedure inverse(n,a,b);
value n;
integer n;
array a,b;
  comment the procedure finds the inverse ( if it exists ) of the
  matrix a by Gaussian illimination.If the inverse exist,it is
  stored in b.If the inverse does not exist,inverse is false;
begin
  integer i,j,k,m,pivotnr;
  real pivot,s;
  array c(1:n,1:n),x(1:n),d(1:n);

  inverse:=true;
  for m:=1 step 1 until n do
  begin
    comment for each m one is solving the linear system,which on the
    wright side has the m-th column in the unit-matrix,and on the left
    side the given matix as coefficientmatrix and the m-th column in
    the wanted inverse as unknown;
```

```
for j:=1 step 1 until n do
for i:=1 step 1 until n do c(i,j):=a(i,j);
for i:=1 step 1 until n do d(i):=(if i=m then 1 else 0);
for k:=1 step 1 until n-1 do
begin
  comment among the last n-k+1 equations one is finding the equation,
  which has the numerical largest coefficient in x(k);
  pivot:=0; pivotnr:=0;
  for i:=k step 1 until n do if abs(c(i,k))>pivot then
  begin pivot:=c(i,k); pivotnr:=i; end;
  if pivot=0 then begin inverse:=false; goto END;end;
  comment if pivot=0 then the given matrix has determinant 0 and
  consequently no inverse;
  if pivotnr<k then
  begin
    comment equation number k is replaced by equation number pivotnr
    and vica versa;
    s:=d(k); d(k):=d(pivotnr); d(pivotnr):=s;
    for j:=k step 1 until n do
    begin
      x(j):=c(k,j); c(k,j):=c(pivotnr,j); c(pivotnr,j):=x(j);
    end;
  end if pivotnr<k;
  for i:=k+1 step 1 until n do
  begin
    comment x(k) is calculated from the k-th equation, and the
    expression inserted in the following n-k equations;
    d(i):=d(i)-d(k)*c(i,k)/c(k,k);
    for j:=k+1 step 1 until n do
      c(i,j):= c(i,j)-c(i,k)*c(k,j)/c(k,k);
    end;
  end k;
  if c(n,n)=0 then begin inverse:=false; goto END;end else
  x(n):=d(n)/c(n,n);
  for i:=n-1 step -1 until 1 do
  begin
    comment for each i x(i) is calculated from the equation
     $c(i,i)x(i) + c(i,i+1)x(i+1) + \dots + c(i,n)x(n) = d(i),$ 
```

```

where x(i+1),. . . x(n) are known;
s:=0; for j:=n step -1 until i+1 do s:=s+c(i,j)*x(j);
x(i):=(d(i)-s)/c(i,i);
end;
for i:=1 step 1 until n do b(i,m):=x(i);
end m;
END: end;

```

```

procedure search(n,g,y,psi,f2);
value n,g;
integer n;
real g,f2;
array y,psi;
begin
  comment the procedure finds the value of the function to be
  minimised,that is  $k \times F$ ,at the point obtained from y by going the
  distance g in the direction -psi;
  for i:=1 step 1 until n do x(i):=y(i)-g*psi(i);
  f2:= $k \times F$ ;
end;

```

```

equal(n,point,x);equal(n,x,y); k:=1;
for i:=1 step 1 until n do psi(i):=delta;
comment psi is the gradient of F at the starting point;
equal(n,psi,b); h:=product:=norm(n,psi);
if h<1 then r:=1/5 else r:=1/(5*h);

```

KONSTANT:

```

; comment at each step of the iterativ process the procedure will
goto KONSTANT and run through the following.A point y and a
direction psi is given,and the problem is to find a konstant g
such that the point y-g*psi can be used as the next point;
h:=norm(n,psi); equal(n,y,x);
if h/product<1/10 then

```

```
begin
  comment psi is too small relativ to the gradient which implies,
  that the greatest possible progress is too small. We therefore
  consider the function  $k \times F$ , where  $k$  is defined below;
   $k := (h / \text{product}) \times (1/n)$ ;
  for i:=1 step 1 until n do  $\text{psi}(i) := (1/k) \times n \times \text{psi}(i)$ ;
  for i:=1 step 1 until n do  $b(i) := k \times \text{delta}$ ;
   $h := \text{norm}(n, \text{psi})$ ;
  if  $h < 1$  then  $r := 1/5$  else  $r := 1/(5 \times h)$ ;
end;
 $h := r \times h$ ;  $f1 := k \times F$ ;
comment h is used below as the small quantity in the approximation
of the second order derivatives of  $F$ ,  $r$  is introduced in order to
insure, that this quantity is not too big at the beginning;
 $\text{product} := \text{innerproduct}(n, b, \text{psi})$ ;
 $g := 1$ ;  $g1 := 0$ ;
search(n, 1, y, psi, f2);
if  $f1 - f2 \geq 1/4 \times \text{product}$  then
begin
   $f1 := f2$ ; equal(n, x, y); goto DIRECTION;
end;
comment in this case we use  $g = 1/4$  and the next point is
therefore obtained as  $y - 1/4 \times \text{psi}$ ;
 $s := (\text{if } s < 1 \text{ then } 10^{-10} \text{ else } 1/s \times 10^{-10})$ ;
for  $g := g/2$  while  $f1 \leq f2$  do
begin
  search(n, g, y, psi, f2);
  if  $g < s$  then begin equal(n, y, x); goto END; end;
end;
comment if  $g$  is smaller than  $s$  (see the definition of this term)
then the next point of the process will be practically equal to
the present, and we must therefore conclude, that the procedure is
unable to make further progress;
 $g := 2 \times g$ ; equal(n, x, z);
if  $(f1 - f2) < g \times \text{product}$  then goto SECOND else
begin
  comment in this case the functionvalue at  $y - g \times \text{psi}$  is smaller
  than  $f1$ , but the condition  $f1 - f2 < g \times \text{product}$  is not satisfied and
```

therefore g is too small;

$g1:=g$; $g:=2 \times g$;

FIRST:

$g:=(g1+g)/2$;

search(n,g,y,psi,f2);

if $f1 < f2$ then goto FIRST else

begin

if $(f1-f2) < g \times \text{product}$ then

begin equal(n,x,y); $f1:=f2$; goto DIRECTION; end else

begin $g:=2 \times g - g1$; $g1:=(g+g1)/2$; goto FIRST; end;

end;

end;

SECOND:

; comment in this case the functionvalue at $y-g \times \text{psi}$ is smaller than $f1$ and the condition $f1-f2 < g \times \text{product}$ is satisfied. We therefore look for a smaller g for which this condition is satisfied and with a smaller functionvalue than before;

$g:=(g1+g)/2$; search(n,g,y,psi,f3);

if $f2 <= f3$ then

begin equal(n,z,x); equal(n,x,y); $f1:=f2$; goto DIRECTION;

end else

begin

if $(f1-f3) < g \times \text{product}$ then

begin $f2:=f3$; equal(n,x,z); goto SECOND; end

else goto THIRD;

end;

THIRD:

; comment in this case the functionvalue is smaller than before, but the condition mentioned before is not satisfied, so g is too small;

$g:=2 \times g - g1$; $g1:=(g+g1)/2$; $g:=(g+g1)/2$;

search(n,g,y,psi,f3);

if $(f1-f3) >= g \times \text{product}$ then goto THIRD else

begin

if $f3 >= f1$ then goto THIRD else

begin equal(n,x,y); $f1:=f3$; goto DIRECTION; end;

end;

DIRECTION:

```
; comment at each step of the iterativ process the procedure will
goto DIRECTION and run through the following. A point x is given
and the problem is to determine the direction in which the next
point is to be found;
for i:=1 step 1 until n do b(i):=k*delta;
product:=norm(n,b);
if product<k*eps or product<10-10 then goto END;
comment if product<k*eps then the wanted accuracy is obtained.
if product<10-10 then in most situations it will be meaningless
to look for further progress;
for j:=1 step 1 until n do
begin
  comment an approximation to the matrix consisting of the second
  order derivatives of k*F is calculated and the result stored in q;
  for i:=1 step 1 until n do x(i):=(if i=j then y(i)+h else y(i));
  for i:=1 step 1 until n do p(i,1):=k*delta;
  for i:=1 step 1 until n do q(i,j):=(p(i,1)-b(i))/h;
end j;
if -,inverse(n,q,p) then goto STEEPEST else
begin
  comment if the inverse of q exist, then the vector psi is
  obtained by multiplying the inverse matrix with the gradient;
  for i:=1 step 1 until n do
  begin
    psi(i):=0; for j:=1 step 1 until n do
      psi(i):=psi(i)+p(i,j)*b(j);
    end;
  end;
  if innerproduct(n,psi,b)<=0 then goto STEEPEST else
  goto KONSTANT;
  comment if innerproduct(n,psi,b)<=0 then we can not be sure to
  find a point with smaller functionvalue in the direction psi,
  and therefore psi can not be used. If the innerproduct is >0
  then psi is the new direction;

STEEPEST:
  equal(n,b,psi); goto KONSTANT;
```

```
comment the gradient is used as the new direction;

END:
; comment the present value of the relevant quantities are stored
in the return parameters;
for i:=1 step 1 until n do b(i):=delta;
minimum:=F; eps:=norm(n,b);
for i:=1 step 1 until n do point(i):=x(i);
end;
end;
```

pzero

pzero(order, coef, root)1. Function and parameters

pzero calculates all roots (complex or real) of 2nd, 3rd and 4th order polynomials with real coefficients.

Call: pzero(order, coef, root)

pzero is a boolean procedure which is false if $\text{order} > 4$ or $\text{order} < 2$ or $\text{coef}(\text{order}) = 0$. In this case no computations are made, otherwise pzero is true.

order (call value, integer)
 specifies the order of the polynomial.

coef (call value, array) minimum bounds(o:order)
 specifies the coefficients of the polynomial
 $p(z) := \text{SUM}(\text{coef}(i) \times z^{\times i}), i := 0, 1, \dots, \text{order}.$

root (return value, array) minimum bounds(1:order, 1:2)
 If pzero is true then root specifies the calculated roots
 $z_i, i := 1, 2, \dots, \text{order}$ so that
 $\text{Re } z_i = \text{root}(i, 1)$ and
 $\text{Im } z_i = \text{root}(i, 2).$

2. Method2.1. General

Extremely large or small roots are detected at the first stage of computation, and further computations are executed on the quotient polynomial, where quotient polynomial everywhere in this description

means $p(z) / \text{PRODUCT}(z - z_i)$

where z_i are the roots already found by pzero.

The (quotient) polynomial is now normalized so that $\text{coef}(\text{order})$ equals 1.

2.2. Second order polynomials: $p(z) = z^2 + bz + c$

The quantity $d := b^2 - 4c$ determines whether the roots are complex ($d < 0$) or real ($d \geq 0$).

If the roots are real then the numerically largest root is calculated from

$$z_1 := (-b - \sqrt{b^2 - 4c})/2$$

and the remaining root from

$$z_2 := \text{if } z_1 = 0 \text{ then } 0 \text{ else } c/z_1$$

otherwise the roots are calculated from

$$z_1 := (-b + i\sqrt{4c - b^2})/2$$

and

$$z_2 := (-b - i\sqrt{4c - b^2})/2$$

where i is the imaginary unit.

2.3. Third order polynomials: $p(z) = z^3 + az^2 + bz + c$

If there are multiple roots then all of the roots are calculated directly from the coefficients a, b and c , otherwise one real root is determined by a Newton Iteration whereafter the remaining two roots are calculated from the quotient second order polynomial (see 2.2).

Analysis and iteration starting point:

The transformation $w = z + a/3$ yields

$$p(z) = 0 \Leftrightarrow q(w) = w^3 + dw + e = 0.$$

Define

$$r := 27e^2 + 4d^3.$$

a) 1 real and 2 complex conjugate roots:

are called $2k$, $-k + im$ and $-k - im$

where i is the imaginary unit.

$$q(w) = w^3 + (m^2 - 3k^2)w - 2k(k^2 + m^2)$$

implies

$$r = [2m(m^2 + 9k^2)]^2 \geq 0$$

and defining

$$f(m) = 4 \times |e| = 8|k(k^2 + m^2)| \geq 8|k|^3$$

yields

$$(4|e|)^{1/3} \geq 2|k|$$

b) 3 real roots:

are called k , m and $-k - m$ where k is the numerically largest root.

$$q(w) = w^3 - (k^2 + m^2 + km)w + km(k + m)$$

implies

$$r = -[(k - m)(2k + m)(k + 2m)]^2 \leq 0 \quad (xx)$$

and defining

$$f(m) = 4|d|/3 = 4|k^2 + m^2 + km|/3$$

yields

$$\min f(m) = f(-d/2) = k^2$$

so

$$2 \times \sqrt{|d|/3} \geq |k|$$

From (x) and (xx) we see that

$r < 0 \Rightarrow$ real roots

$r = 0 \Rightarrow$ multiple roots

$r > 0 \Rightarrow$ complex roots

and iteration starting point s is chosen to be

$$s := 2\sqrt{|d|/3} \text{ if } r < 0 \\ (4|e|)(1/3) \text{ if } r > 0.$$

The quotient second order polynomial:

is calculated from

$$(z^2 + pz + q)(z - z_1) = z^3 + az^2 + bz + c$$

where z_1 is the real root obtained by iteration.

If $|a + z_1| > |a|/8$ then p is calculated from

$$p := a + z_1$$

and

$$q := b + pz_1 \quad \text{if } |b + pz_1| \geq |b|/8 \\ - c/z_1 \quad \text{if } |b + pz_1| < |b|/8$$

otherwise

$$q := -c/z_1$$

and

$$p := (q - b)/z_1.$$

2.4. Fourth order polynomials: $p(z) = z^4 + az^3 + bz^2 + cz + d$

I: A linear transformation $w = z + a/4$ yields

$$p(z) = 0 \Leftrightarrow q(w) = w^4 + 1w^2 + mw + n = 0.$$

now the sum of the transformed roots equals zero.

II: The transformed roots are calculated using the method of Descartes.
This method involves the solution of a third order equation, and
this is performed as described in 2.3.

III: The roots are now accepted if $|\operatorname{Re} z_i| > |a|/32$ so at least one root
must be accepted unless they all equal zero.

IV a) if one root is accepted by III

then the reciprocal roots are calculated from

$$dz^4 + cz^3 + bz^2 + az + 1 = 0$$

as described in 2.4-I, II and III.

If one of the reciprocal roots is accepted then we have two accepted roots, so the remaining two roots are calculated as described in IV b, otherwise the former accepted root is not used. The number of accepted reciprocal roots then determines whether further calculations are performed as described in IV, b, c or d.

b) if two roots are accepted by III (or IV)

then the quotient second order polynomial is calculated and solved
as described in 2.2.

c) if three roots are accepted by III (or IV)

then the remaining real root is calculated using the fact that the
product of the roots equals d.

d) if four roots are accepted by III (or IV)

then no further calculations are performed by pzero.

3. Accuracy, time- and storage requirements

3.1. Accuracy

If an actual equation is ill-conditioned and you want the roots to a specified degree of accuracy a much greater accuracy may be necessary in the intermediate calculations. On the other hand a user is not supposed to know anything about the conditioning of the actual equation, so standard input to RC4000 of 48-bits reals is used.

3.2. Time- and storage requirements

Approximate cpu-time used by pzero:	(order - 1) × 0.02 sec.
Codelength:	12 segments
Typographical length:	223 lines incl. last comment.

4. Test and discussion:

pzero has been tested on the RC4000 computer with a testprogram which performs

- 1) generation of order and coefficients
- 2) call of pzero
- 3) calculation of root generated coefficients of the polynomial
$$p(z) := \text{PRODUCT}(z - z_i), i := 1, 2, \dots, \text{order}$$
- 4) calculation of relative differences between the original and the root generated coefficients.

Now the smallness of the differences is chosen as a measure of the goodness of pzero.

pzero has been tested with a large number of both prepared ill-conditioned coefficients and random coefficients input and in both cases with satisfying results.

Some test examples (the check column describes the relative differences):

example number 1

given equation	check
coef(4) = 1.0000000000 ₁₀ 0	
coef(3) = 1.0000000000 ₁₀ 0	-5.82 ₁₀ -11
coef(2) = 1.0000000000 ₁₀ 0	-5.82 ₁₀ -11
coef(1) = 1.0000000000 ₁₀ -7	-5.90 ₁₀ -4
coef(0) = 1.0000000000 ₁₀ 0	0.00 ₁₀ 0

calculated roots

3.5180794434 ₁₀	-1+7.2034175772 ₁₀	-1	x1
3.5180794434 ₁₀	-1-7.2034175772 ₁₀	-1	x1
-8.5180794432 ₁₀	-1+9.1129213536 ₁₀	-1	x1
-8.5180794432 ₁₀	-1-9.1129213536 ₁₀	-1	x1

example number 2

given equation	check
coef(4) = 1.0000000000 ₁₀ 0	
coef(3) = -6.8619274672 ₁₀ -1	0.00 ₁₀ 0
coef(2) = -8.8228487860 ₁₀ -1	-6.60 ₁₀ -11
coef(1) = 6.8619274672 ₁₀ -1	0.00 ₁₀ 0
coef(0) = -1.1771512141 ₁₀ -1	0.00 ₁₀ 0

calculated roots

3.4309637339 ₁₀	-1
1.0000000000 ₁₀	0
3.4309637335 ₁₀	-1
-1.0000000000 ₁₀	0

example number 3

given equation	check
coef(4) = 1.0000000000 ₁₀ 0	
coef(3) = -1.4215286873 ₁₀ 1	0.00 ₁₀ 0
coef(2) = 7.1429889252 ₁₀ 1	1.04 ₁₀ -10
coef(1) = -1.4369489911 ₁₀ 2	3.63 ₁₀ -10
coef(0) = 8.5480296736 ₁₀ 1	6.97 ₁₀ -10

calculated roots

4.4050104852 ₁₀	0
4.4051469640 ₁₀	0
4.4051294242 ₁₀	0
9.9999999956 ₁₀	-1

example number 4

given equation

check

coef(4)= 1.0000000000 ₁₀	0		
coef(3)= -2.4628394422 ₁₀	0	4.32 ₁₀	-11
coef(2)= 2.7192690981 ₁₀	0	0.00 ₁₀	0
coef(1)= -1.2204258468 ₁₀	0	4.77 ₁₀	-11
coef(0)= 2.0540067855 ₁₀	-1	1.42 ₁₀	-10

calculated roots

6.7204851918 ₁₀	-1+1.1559340115 ₁₀	-2	x1
6.7204851918 ₁₀	-1-1.1559340115 ₁₀	-3	x1
6.7437120188 ₁₀	-1+1.1667236046 ₁₀	-3	x1
6.7437120188 ₁₀	-1-1.1667236046 ₁₀	-3	x1

example number 5

given equation

check

coef(4)= 1.0000000000 ₁₀	0		
coef(3)= -5.9418329952 ₁₀	0	0.00 ₁₀	0
coef(2)= 1.3239517255 ₁₀	1	0.00 ₁₀	0
coef(1)= -1.3111166744 ₁₀	1	7.10 ₁₀	-11
coef(0)= 4.8690226978 ₁₀	0	2.39 ₁₀	-10

calculated roots

1.4854582489 ₁₀	0		
1.4844816864 ₁₀	0		
1.4859465301 ₁₀	0+8.4572793336 ₁₀	-4	x1
1.4859465301 ₁₀	0-8.4572793336 ₁₀	-4	x1

6. Complete algol text:

pzero=set 12

pzero=algol

external

message pzero, version 22/5-70, RCSL 53-M4;

boolean procedure pzero(order, coef, root);

value order;

integer order;

array coef, root;

```

begin
array arr(0:4);
integer accept,i;
real x,push,a,b,c,d;
boolean ok;

procedure order4;
begin
real a,b,c,d,x,push;
integer i;
  push:=arr(3)/4;
  c:=((-3*push**2+arr(2))*push-arr(1))*push+arr(0);
  b:=(push*arr(3)-arr(2))*2*push+arr(1);
  a:=-3*arr(3)**2/8+arr(2);
  if b<0 then
  begin
    order3(2*a,a**2-4*c,-b**2);
    for i:=0,1+i while root(i,2)<0 or root(i,1)<0 do;
      x:=root(i,1);
      d:=b;
      b:=a+x;
      a:=sqrt(x);
      x:=d/a;
      if abs(b-x)>abs(b+x) then b:=b-x else
      begin
        b:=b+x;
        a:=-a
      end;
      b:=b/2
    end
  else
    if a**2<4*c then
    begin
      b:=sqrt(c);
      a:=sqrt(2*b-a)
    end
    else
    begin
      b:=a+sgn(a)*sqrt(a**2-4*c)/2;
      a:=0
    end;
  end;
end;

```

```

order2(a,b,1);
order2(-a,if b=0 then 0 else c/b,3);
x:=abs push/8;
for i:=1,2,3,4 do
if abs(root(i,1)-push)>x then
begin
    accept:=1+accept;
    root(accept,1):=root(i,1)-push;
    root(accept,2):=root(i,2)
end;
exit:
end order4;

```

```

procedure order3(a,b,c);
value          a,b,c;
real           a,b,c;
begin
real push,p,q,r;
    push:=-a/3;
    p:=a**2-3*b;
    q:=(-2*push**2+b)*push+c;
    r:=(27*c-a*(18*b-4*a**2))*c+b**2*(4*b-a**2);
    if abs r<=((27*abs c+abs a*(18*abs b+4*a**2))*abs c
                +b**2*(4*abs b+a**2))*3**11
    then
    begin
        d:=(a**2+3*abs b)*3**11;
        if p+d<0 then goto newton;
        q:=if p-d<0 then 0 else sgn(q)*sqrt(p)/3;
        root(1,1):=root(2,1):=push+q;
        root(3,1):=push-2*q;
        root(1,2):=root(2,2):=root(3,2):=0;
        goto exit
    end;
end;

```

newton:

```

r:=push-sgn(q)*(if r<0 and p>=0 then 2*sqrt(p)/3 else(4*abs q)**(1/3));
for p:=((2*r+a)*r**2-c)/((3*r+2*a)*r+b),
      ((2*r+a)*r**2-c)/((3*r+2*a)*r+b)
      while abs(p-push)<abs(r-push) do r:=p;
root(1,1):=r;
root(1,2):=0;
p:=a+r;
q:=b+p*r;
q:=if abs p<abs a/8 or abs q<abs b/8 then -c/r else q;
p:=if abs p<abs a/8 then (q-b)/r else p;
order2(p,q,2);

```

exit:

end order3;

procedure order2(b,c,first);

value b,c,first;

real b,c;

integer first;

begin

real d;

d:=b**2-4*c;

d:=sgn(d)*sqrt(abs d);

if d<0 then

begin

root(first,1):=root(1+first,1):=-b/2;

root(first,2):=d/2;

root(1+first,2):=-d/2

end else

begin

d:=root(first,1):=(-b-sgn(b)*d)/2;

root(1+first,1):=if d=0 then 0 else c/d;

root(first,2):=root(1+first,2):=0

end

end order2;

accept:=0;

ok:=pzzero:=order>1 and order<5 and coef(order)<0;

- 11 -

```
if -,ok then goto finis;
for i:=order step -1 until 0 do arr(i):=coef(i);
low:
x:=if arr(1)=0 then arr(0) else -arr(0)/arr(1);
for i:=0,1+i while arr(i)-arr(1+i)*x=arr(i) do
if i=order-1 then
begin
for i:=0 step 1 until order-1 do arr(i):=arr(1+i);
goto comb;
end;
x:=-arr(order-1)/arr(order);
for i:=0,1+i while arr(i)*x-arr(i-1)=arr(i)*x do
if i=order-1 then goto comb;
goto normal;
comb:
root(order,1):=x;
root(order,2):=0;
order:=order-1;
if order>1 then goto low;
root(1,1):=-arr(0)/arr(1);
root(1,2):=0;
goto finis;
normal:
x:=arr(order);
for i:=order step -1 until 0 do arr(i):=arr(i)/x;
case order-1 of
begin
order2(arr(1),arr(0),1);
order3(arr(2),arr(1),arr(0));
begin
order4;
select: case accept of
begin
begin
arr(4):=root(1,1);
x:=coef(0);
for i:=0,1,2,3 do arr(i):=coef(4-i)/x;
accept:=0;
```

```

order4;
if accept>1 then
begin
  for i:=1,1+i while i<=accept and i<5 do
    if root(i,2)=0 then root(i,1):=1/root(i,1)
    else
      begin
        x:=root(i,1)××2+root(i,2)××2;
        root(i,1):=root(1+i,1):=root(i,1)/x;
        root(i,2):=root(i,2)/x;
        root(1+i,2):=-root(i,2);
        i:=1+i
      end;
    end
  else
    begin
      root(2,1):=1/root(1,1);
      root(1,1):=arr(4);
      accept:=2
    end;
  x:=coef(4);
  for i:=0,1,2,3 do arr(i):=coef(i)/x;
  goto select
end;

begin
  d:=-root(1,1)-root(2,1);
  c:=root(1,1)×root(2,1)-root(1,2)×root(2,2);
  b:=arr(0)/c;
  a:=if abs (arr(1)/b-d)<abs (arr(3)-d)
    then arr(3)-d
    else (arr(1)-b×d)/c;
  order2(a,b,3)
end;

```

```

begin
  a:=if root(1,2)=0
    then root(1,1)×(root(2,1)×root(3,1)-root(2,2)×root(3,2))
    else root(3,1)×(root(1,1)×x2+root(1,2)×x2);
  root(4,1):=arr(0)/a;
  root(4,2):=0
end;;;
end
end
end;
finis:
end pzero;

```

comment:

pzero(order,coef,root) calculates real and complex roots
of 2nd, 3rd and 4th order polynomials with real coefficients:

$$p(z) = \text{coef}(\text{order}) \times z^{\text{order}} + \dots + \text{coef}(1) \times z + \text{coef}(0).$$

pzero is false if $\text{order} > 4$ or $\text{order} < 2$ or $\text{coef}(\text{order}) = 0$,
otherwise pzero is true.

order (call value, integer) specifies the order of the
polynomial.

coef (call value, array) specifies the coefficients of
the polynomial.

root (return value, array).

If pzero is true then root specifies the roots of
the polynomial: z_i , $i=1,2,\dots,\text{order}$ so that

$$\text{Re } z_i = \text{root}(i,1)$$

$$\text{Im } z_i = \text{root}(i,2);$$

end;

runge_kutta

1. FUNCTION and PARAMETERS.

Runge_kutta solves a system of first order ordinary differential equations on the form

$$dx(j)/dt = f_j(t, x(1), x(2), \dots, x(n)); j = 1, 2, \dots, n$$

with given initial values using the parameter t as integration variable.

The procedure heading is:

```
boolean procedure runge_kutta(f, x, t, eps, dts, max, fstop);  
value          eps;  
integer        max;  
real           t, eps, dts, fstop;  
procedure      f;  
array          x;
```

Call parameter:

eps: A real variable.
The tolerance for the relative error. The procedure tries to control the steplength such that the accumulated relative error does not exceed $\text{eps} \cdot (b-a)$ where $b-a$ is the length of the integration interval. eps must be positive and should be chosen between 10^{-1} and 10^{-11} depending on the accuracy wanted. It is recommended to choose $\text{eps} \geq 10^{-8}$ because with smaller eps the procedure may often use an excessive amount of work for only a slight improvements of the results.

Call/Return parameters:

x: a real array, declared as: array x(1:n). The index bounds must be 1 and n = the number of equations, respectively.
On entry, x contains the initial values of the dependent variables.
On exit, x contains the result, i.e. the values of these variables at the point where the stop condition is fulfilled.
If the lower index bound of x is not 1, the run is terminated with the alarm message <:rungekut:>.

t: a real variable.
On entry, t contains the initial value of the independent variable.
On exit, t contains the final value at the point where the stop condition is fulfilled.

max: an integer variable.
On entry, max denotes the maximum number of integration steps to be performed by the procedure.
On exit, max contains the number of steps actually performed counting accepted as well as rejected steps.

dts: a real variable.
On entry, dts contains the initial step size. The procedure integrates in the t-direction given by the sign of dts. If dts = 0, the entire length of the interval, fstop-t, is used as first guess on the step size. On exit, dts contains the estimated size of the next step. This is useful if integration is continued by repeated calls of the procedure.

Return parameter:

runge_kutta: On exit, the boolean procedure is true when the integration was successful, i.e., the max number of steps was sufficient. If the max number of steps was used before reaching the stop criterion the procedure returns with the value false, the parameters t and x containing the current values.

Other parameters:

f: A procedure with 3 parameters, declared with the heading:

```
procedure f(x, t, dxdt);  
  real t;  
  array x, dxdt;
```

The call

```
  f(x, t, k)
```

where the array x contains the values of the dependent variables and t the value of the integration variable,

must assign to k the function values, i.e.

$$k(j) = f_j(t, x(1), \dots, x(n)); \quad j = 1, \dots, n.$$

Neither x nor t may be changed by the procedure.

fstop:

A real expression used as stop criterion:

If **fstop** is constant, integration continues until

$t = \text{fstop}$, i.e., **fstop** is simply the final t -value.

If the value of **fstop** changes during the integration, the procedure terminates when **fstop** = 0.

The parameter **fstop** is called once initially and once per accepted step; the initial value may be zero without terminating the integration.

2. METHOD.

2.1. Mathematical formulae

The fifth order Runge Kutta formulae used are derived by Zonneveld [1]. They use 6 intermediate points in each interval and one additional point for the error and step control, thus requiring 7 calls per step of the procedure f .

The formulae are exact up to and including the fifth order term of the Taylor expansion and gives an estimate of this last term, which is used to determine whether the step should be accepted or not; at the same time it is used to estimate the size of the next step as explained below.

2.2. Termination.

The procedure may be terminated in four ways:

- a) When $t = \text{fstop}$. This is used when integrating over a fixed interval, say a to b ; the procedure should be called with b as the last parameter, and with max sufficiently large.
- b) When **fstop** = 0. If the value of **fstop** is not constant, integration continues until **fstop** changes sign; then a zero-finding algorithm is entered to find the point where **fstop** = 0. The algorithm is an adaptively adjusted weighting between regula falsi and bisection; it iterates until the length of the root enclosing interval is smaller than

$10^{-8*\mu}$ where μ is the smaller of the 2 last accepted regular steps. For details see comment 8 and 9 to the algorithm.

- c) If the number of steps permitted (max) is exceeded. The return parameters of the procedure allow continued integration and this feature may be used to monitor the integration of tricky functions: If you call the procedure repeatedly with a small value of max, the calling program gets a chance to react on intermediary values, if necessary.

This exit is also used to prevent the procedure from cycling: There is no lower limit on the adaptive step length, hence you may - unintentionally - call the procedure with parameters causing a very lengthy integration with extremely small steps. The value of the procedure being false shows this to be the exit cause, and the parameters t and x always contain the actually achieved values.

- d) If the procedure is called with a second parameter X which is not an array with lower index bound 1, the procedure immediately terminates the run with the alarm message 'rungekut'.

2.3. Error Control.

Accumulated error estimate.

The error control algorithm tries to distribute the total error in proportion to the total variation of the sought function x. It may be shown that a suitable way is to adapt the step length dt such that

$$(2.1) \quad |\text{local error}| \approx \text{eps} * dt * (\text{variation}(x) + \text{eps} * |x|);$$

Under certain hypotheses on the function x(t) this leads to an accumulated error over the interval (a, b) which approximately satisfies

$$(2.2) \quad \left| \frac{\text{acc. error}}{\text{variation}(x)} \right| \approx \text{eps} * |b - a|$$

If the length of the integration interval |b-a| always were known at call time, this factor might be included in the step length algorithm.

- 5 -

But $|b-a|$ is not known when integrating with a variable $fstop$ parameter, and as a consequence the procedure always works in accordance with (2.1) and (2.2).

When integrating a system of equations, say n equations, expressions (2.1) and (2.2) are substituted by

$$(2.3) \quad \max_n \left| \frac{\text{local error}}{\text{variation}(x) + \text{eps} * |x|} \right| \approx \text{eps} * dt$$

and

$$(2.4) \quad \max_n \left| \frac{\text{acc. error}}{\text{variation}(x)} \right| \approx \text{eps} * |b-a|$$

The formula in [1] doesn't give the local error, but estimates the 5th order term in the Taylor series by

$$\text{5th order term} = (k_0 * 21 - k_2 * 162 + k_3 * 224 - k_4 * 125 + k_5 * 42) / 14$$

but since the formula for y is exact up to and including this 5th term it is reasonable to use the estimate

$$\text{local error} = \text{5th order term} * dt$$

and hence (2.3) becomes

$$(2.5) \quad \max_n \left| \frac{\text{5th order term}}{(\text{variation}(x) + \text{eps} * |x|) * \text{eps}} \right| \approx 1 \quad .$$

Accept criterion

For each equation the procedure calculates, in every step,

$$s = (\text{variation}(x) + \text{eps} * \text{abs}(x)) * \text{eps}$$

$$f = \text{abs}(\text{5th_order_term}) / s$$

and

$$\text{sft} = \max(f), \quad \text{max over all equations.}$$

In accordance with (2.5) the step is accepted if $\text{sft} \leq 1$, and rejected otherwise. In the extrapolation algorithm for the length of the next step the procedure tries continuously to keep sft slightly smaller than 1, thus safely fulfilling (2.5).

Step estimation after reject ($sft > 1$).

In sft the denominator s may be considered locally constant and hence we have $appr.$

$$sft = \text{some_constant} * h^{**5} .$$

Therefore the new, optimal step length should be

$$(2.6) \quad h_{new} = h * \sqrt[5]{1/sft}$$

but because of slightly easier calculation and in order to introduce a safety margin the following formula is used instead:

$$h_{new} = 0.95 * h * \sqrt[4]{1/sft} .$$

Remark: Because the procedure is especially suited to integrate tricky equations the formula is deliberately chosen so that the step length may become arbitrarily small.

Step extrapolation after accept ($sft \leq 1$).

The optimal new step length is again given by (2.6), but since sft may become arbitrarily small - or even equal to zero - this extrapolation is replaced by a formula giving a reasonable limited maximum growth of h and behaving like (2.6) in the neighbourhood of $sft = 1$. Following Zonneveld [1] we approximate $\sqrt[5]{1/sft}$ by

$$(2.7) \quad \mu = 1/(1 + sft) + 0.45$$

with the range 0.95 to 1.45 instead of 1 to infinity, and modify this by using a one step memory in the algorithm: the last accepted values of μ and h are kept and used in the final extrapolation

$$(2.8) \quad h_{new} = h * (h/h_{old} * \mu + \mu - \mu_{old}) .$$

The effect of equation (2.8) is to introduce an 'overrelaxation' based on the development of h and μ over the last two steps: If $h_{old} < h$ and $\mu_{old} < \mu$, then h_{new} will become larger than estimated by (2.6) or (2.7): if e.g. $sft = 0$ over several steps, then h will grow approximately as 1, 2, 5, 18, 97, ... If, on the other hand, $sft = 1$ over several steps, equation (2.8) cautiously makes h smaller slowly: μ is 0.95 steadily and h will diminish approximately as 1, 0.8, 0.6, 0.45, ...

2.4. Round-off errors.

It may be shown that even when all arithmetic operations are performed with correct rounding, as in RC4000, the accumulated round-off errors in the summation of x-values are reduced considerably by using quasi-double precision; see Möller [2]. Therefore the summation of t and x is done using quasi-double precision.

Since this works equally well in 36-bits and in 33-bits arithmetic, the procedure gives almost identical results when working in the low and in the high precision mode of RC4000: When working with large values of eps, $\text{eps} \geq 10^{-7}$, the results from the tests are identical in low and in high precision; with $\text{eps} < 10^{-7}$, the procedure often uses more steps in the low precision mode, but the resulting errors are in many cases the same as in the high precision mode.

3. TIME and STORAGE REQUIREMENT

The procedure uses $60 + 12 * N$ local variables (reals) and 1 local procedure with no parameters. The translated procedure has a length of 6 segments.

The execution time for the procedure itself is approximately $4 + 7N$ msec per step where N is the number of equations. The figure includes call of the f-procedure but the time for executing the body of this procedure must be added (it is called 7 times per Runge-Kutta step).

4. EXAMPLES of USE.

Problem 1: Solve two differential equations, say,

$$y' = t + \sin(y-z)$$

$$z' = t/y$$

over the interval $0.5 \leq t \leq 3.5$ with initial values $y(0.5) = 1$, $z(0.5) = 2$, and with a relative error smaller than 1 promille. The solution is wanted for $t = 4.5$ only.

Solution 1: The program structure of the solution is as follows:

1) In the program block head is declared:

```
integer Max;
real t, dt;
array YZ(1:2);
procedure F1(x, t, dx);
array x, dx; real t;
begin
    dx(1) := t + sin(x(1) - x(2));
    dx(2) := t/x(1);
end;
---
```

2) The procedure is then called:

```
---
t:= 0.5; dt:= 10-2; Max:= 600;
YZ(1):= 1; YZ(2):= 2;
if -,rungekutta(F1, YZ, t, 10-3, dt, Max, 3.5)
    then begin
        comment: error action;
    end;
---
```

After this call YZ(1:2) contain the solutions y(3.5) and z(3.5), Max contains the number of steps used, and dt the last estimated step size.

Problem 2: The solution to problem 1 is wanted printed out for t = 1, 1.5, 2, ... , 3.5.

Solution 2: With the same declarations as above the procedure is now called inside a loop:

```
---
t:= 0.5; dt:= 10-2;
YZ(1):= 1; YZ(2):= 2;
for tslut:= 1, 1.5, 2, 2.5, 3, 3.5 do
begin
    Max:= 100;
    if -,rungekutta(F1, YZ, t, 10-3, dt, Max, tslut)
        then begin
            comment: error action;
        end;
    write(..., YZ(1), YZ(2), ...);
end;
---
```

Problem 3: The solution to problem 1 is wanted at the point where y has a minimum, i.e., where $y' = 0$.

Solution 3: With the same declarations and initialization as in solution 1, the procedure is called with the same parameters except for the last one:

```
if -, rungekutta(F1, ..., t + sin(YZ(1) - YZ(2))) then --- ;
```

Problem 4: In order to analyze the behaviour of the procedure on the differential equations of problem 1, a printout is wanted for every 10 steps of integration.

Solution 4: With the same declarations as above the procedure is called as follows:

```
t:= 0.5; dt:= 10-2;
YZ(1):= 1; YZ(2):= 2;
for t:= t while t < 3.5 do
begin
  Max:= 10;
  rungekutta(F1, YZ, t, 10-3, dt, Max, 3.5);
  write(out, ..., t, YZ(1), YZ(2), ...);
end;
```

5. TEST

The procedure was tested on several examples among which were:

Test 1: The three equations

$$y_1' = y_2, \quad y_2' = y_3,$$

$$y_3' = 2 * (y_1 * y_3 + y_2 ** 2)$$

over the interval $0 \leq t \leq 1.5$ with the initial values

$$y_1 = 0, \quad y_2 = 1, \quad y_3 = 0.$$

The solution is $y_1(t) = tg(t)$, which tends to infinity. At $t = 1.5$ the values are apprx. $y_1 = 14$, $y_2 = 184$, $y_3 = 5000$.

Test 6: The two equations

$$y_1' = y_2 * y_1 ** 2,$$

$$y_2' = -1/y_1$$

over the interval $0 \leq t \leq 4$ with initial values $y_1 = y_2 = 1$. The solution is

$$y_2(t) = \cosh(t) - \sinh(t) = \exp(-t)$$

$$y_1(t) = 1/y_2(t) = \exp(t)$$

As $\cosh(t)$ and $\sinh(t)$ approach each other, y_2 grows rapidly and y_1 disappears. At $t = 4$ their values are $y_1 = 55$ and $y_2 = 0.02$.

Test 3: One equation

$$y' = t ** p * (1-t) ** q$$

over the interval $0 \leq t \leq 1$, where p and q are given integers, and with the initial value $y = 0$. The solution is the betafunction with $y(1) = B(p+1, q+1)$. The test was run with $p = q = 4$ and with $p = 24$, $q = 49$. The solution $y(t)$ has extremely small variation, in the first case apprx. 10^{-3} and in the second case apprx. 10^{-21} , but the higher derivatives vary much more.

Test 5: The Volterra equations

$$y_1' = a*y_1 - b*y_1*y_2$$

$$y_2' = c*y_1*y_2 - d*y_2$$

over some interval $0 \leq t \leq T$, where a , b , c , and d are given constants, and with given initial values. The solution is periodically oscillating in both variables representing the growth of two conflicting populations.

The tests were mainly concentrated on the following points of interest:

Step-correction algorithm: Several algorithms were tried, including the one proposed by Zonneveld [ref. 1] and the one used in the Gier Algol procedure [ref. 3]. Finally the present one was selected as the best in the sense of minimizing the number of steps required to obtain a certain accuracy.

Eps and the resulting error: As developped in chapter 2 above it is expected that the parameter eps and the resulting error are related appr. as follows

$$\text{Error} = C * \text{eps} * \text{variation}(y) * (b-a);$$

In all the tests except Test 3 this linear relationship was confirmed, and with a constant C between 0.1 and 0.02. Test 6 was the only case with $C > 1$, namely $C \approx 100$. Test 3 with $p = 24$ and $q = 49$ showed a more irregular relationship between Error and eps but still the accumulated error fell below the expected value with $C = 1$.

Precision of the arithmetic: Several tests were carried out both with the normal 36-bits precision floating-point arithmetic and with 33 bits precision. The results were almost identical but for small values of eps, 10^{-8} and 10^{-9} , the procedure used more steps in the 33 bits mode. This may be explained as follows: The results are almost the same because the quasi-double precision works equally well in 36 bits and in 33 bits mode and, in fact, makes both of them look like a (40-50)-bits mode. But in the error control the calculation of the local error estimate is disturbed considerably by the rounding to 33 bits mantissa.

Comparison with other methods: The criterion used for comparison is the number of function evaluations (= calls of the f-procedure) plotted against the accumulated error (relative or absolute).

For nice, smooth solutions it uses almost twice as many function evaluations as a good version of the Hamming predictor-corrector procedure. In more 'difficult' cases (e.g., large higher derivatives) they perform equally well, by and large, but the predictor-corrector is more dependant on a judicious choice of initial step length; the Runge-Kutta procedure gives almost identical results for a wide range of initial step lengths.

In [4] a number of methods were compared and one of the test problems were Test 6. For large values of the error ($> 5_{10}^{-3}$) the present procedure performs very much like the extrapolated Runge-Kutta Arronx of [4]. For smaller errors it works considerably much better than any of the cited procedures but this may be due to a better floating-point arithmetic: 36-bits and quasi-double mode compared with the 28-bits precision of [4].

6. References.

- [1] J.A. Zonneveld: Automatic Numerical Integration.
Mathematical Centre Tracts 8, Amsterdam 1964.
- [2] O. Møller: Quasi Double Precision in Floating Addition.
BIT 5 (1965), 37-50.
- [3] A. Jessen: Simultaneous First Order Differential Equations:
runge kutta general procedure.
Gier System Library No. 522. A/S Regnecentralen 1969.
- [4] Phyllis Fox: A Comparative Study of Computer Programs for integrating Differential Equations.
Comm. ACM 15 (Nov. 1972), 941-948.

7. THE ALGORITHM

```
runge kutta=set 6
runge kutta=algol index.no
external
```

```
boolean procedure runge_kutta(fx,x,t,eps,dts,max,fstop);
value eps;
integer max;
real t,dts,fstop,eps;
procedure fx;
array x;

begin integer n;
if system(3,n,x) <> 1 then system(9,0,<:(10>rungekut:>);

begin integer i,nt;
  boolean tstop,first;
  real c0,c1,e0,e2,e3,e4,e5,dt0,dt,h0,
    h,mu0,mu,b,sft,t1,qt,d,f,q,w,s;
  array a(0:27),dv(0:6),qx,x1,dx,ll,ul(1:n),k(0:6,1:n);

  procedure evaluate;
  begin integer ic,it,i,to;
    real v;
    boolean last;
    it:=0;
    v:=t1;
    for to:=0 step 1 until 6 do
      begin fx(x,v,dx);
        last:=to>5;
        it:=it+to;
        for i:=n step -1 until 1 do
          begin k(to,i):=dx(i)*dt;
            v:=0;
            for ic:=to step -1 until 0 do
              v:=k(ic,i)*a(it+ic)+v;
            if last then dx(i):=v:=v+qx(i);
            x(i):=x1(i)+v
          end;
        v:=dv(to)*dt+qt+t1
      end;
    t:=v
  end evaluate;
```

```

comment 1: initialize;

for i:=0 step 1 until 27 do
a(i):=case i+1 of (2/9,
  1/12 , 1/4 ,
  1/8 , 0 , 3/8 ,
  53/125 , -27/25 , 126/125 , 56/125 ,
  19/24 , -9/4 , 23/14 , 2/3 , 25/168 ,
  -9/4 , 27/4 , -9/7 , -4 , 25/14 , 0 ,
  35/336 , 0 , 81/168 , 0 , 125/336 , 0 , 1/24);
for i:=0 step 1 until 6 do
  dv(i):=case i+1 of (2/9, 1/3, 1/2, 4/5, 1, 1, 1);
e0:=3/2; e2:=-81/7; e3:=16; e4:=-125/14; e5:=3;
for i:=n step -1 until 1 do
begin qx(i):=0;
  ll(i):=ul(i):=x1(i):=x(i)
end;
nt:=max;
c1:=c0:=fstop;
t1:=t;
dt0:=dt:=if dts=0 then (c0-t1) else dts;
tstop:=true;
runge kutta:=true;
mu:=1.05;
qt:=0;

for nt:=nt-1 while nt>0 do
begin comment 2: main loop;
  evaluate;

  comment 3: error estimate;

  sft:=10-10;
  for i:=n step -1 until 1 do
  begin s:=x(i);
    if s>ul(i) then ul(i):=s else
    if s<ll(i) then ll(i):=s;
    s:=(ul(i)-ll(i)+eps*abs s+10-500)*eps;
    f:=(abs(k(0,i)*e0+k(2,i)*e2+k(3,i)*e3+k(4,i)*e4+k(5,i)*e5))/s ;
    if f>sft then sft:=f;
  end;
  if sft<=1 then

begin comment 4: accept;

  c1:=fstop;
  if tstop then
  begin tstop:=c0=c1;
    if tstop and (c1=t or dt*dt0<=0) then goto slut
  end
  else if c0*c1<0 then goto slut;
  c0:= c1;

```

```
comment 5: new dt;

mu0:=mu;
mu:=1/(1+sft)+0.45;
b:=dt/dt0;
s:=dt0;
dt0:=dt;
dt:=(b*mu+mu-mu0)*dt;

comment 6;

qt:=dt0+qt-(t-t1);
t1:=t;
if tstop and (c1-t-qt)/dt<1 then dt:=c1-t-qt;
for i:=n step -1 until 1 do
begin
  qx(i):=dx(i)-(x(i)-x1(i));
  x1(i):=x(i)
end
end accept
else

  begin comment 7: reject;

    for i:=n step -1 until 1 do x(i):=x1(i);
    q:=sqrt(sqrt(sft));
    first:=dt0=dt;
    dt:=dt*0.95/q;
    mu:=mu*2*q/(q+1);
    if first then dt0:=dt;
  end reject
end main loop;
runge_kutta:=false;

slut:
max:=max-nt;
dts:=dt0;
if c0*c1<0 then

begin comment 8: terminating step;

d:=q:=w:=b:=0.99;
f:=h0:=0;
h:=dt:=dt0;
mu:=abs(if s/h<1 then s else h);
for s:=h-h0 while abs s>(abs dt+mu)*10-8 do

begin comment 9: zero determination;

f:=b;
b:=c0+c1;
for i:=n step -1 until 1 do x(i):=x1(i);
if f*b<0 then begin w:=q; q:=d end;
```

```
d:=b/(c0-c1);
dt:=(d*w-1)*s/2+h;
evaluate;
f:=fstop;
if f=0 then h:=dt;
if c0*f<0 then
begin c1:=f; h:=dt end else
begin c0:=f; h0:=dt end;
d:=w:=((if b*f>0 then 1-w else -0.8)*0.9*abs d+1)*w
end zero determination;
if c0*f>0 then
begin dt:=h;
  for i:=n step-1 until 1 do x(i):=x1(i);
  evaluate;
end
end terminating step
end inner block
end rungekutta;
```

8. COMMENTS

The main variables of the procedure are:

A(0:27), DV(0:6), e0, e2, e3, e4, e5: Coefficients from the Runge-Kutta formulae;

x1(1:n): Old values of x;

dx(1:n): Actual increments of x, such that $x(i) = x1(i) + dx(i)$;

qx(1:n): Accumulated rounding errors of x;

ll, ul(1:n): Lower and upper limits of x since entry, hence $ul(i) - ll(i)$ is the variation of $x(i)$;

dt: New step length estimate;

dt0: Last accepted step size;

t1: Last value of t, the integration variable; i.e., t1 is the endpoint of the last accepted step and t is the endpoint of the step being tested;

qt: Accumulated rounding error of t;

sft: Step accept criterion;

mu, mu0: New and old value of step size relaxation factor;

c1, c0: New and old value of the stop criterion fstop;

tstop: true if fstop = constant, false otherwise.

comment 1: The array A is initialized with the coefficients of the Runge-Kutta formulae for k(1), k(2), ... of [1]. (Note that coefficients for k(6) precede those for k(5).) DV is initialized with the factors on h corresponding to the k-s. e0 to e5 contain the coefficients of the error estimate formula. The accumulators qx(i) and qt for the quasi double precision errors are reset. The lower and upper limits of variation ll(i) and ul(i) and the 'old' value of x x1(i) are all reset to the initial values x(i). The present and the old step length estimate, dt and dt0, are set equal the entry value of the parameter dts; if this is zero the total length of the integration interval is used as first guess.

comment 2, Main loop: Here starts the main loop with one cycle per step.

The loop consists of

- function evaluation,
- local error estimate,
- actions when a step is accepted or actions when a step is rejected.

comment 3, Error estimate: For each component $x(i)$ the lower and upper limit is updated, the Runge-Kutta error estimate f is calculated in accordance with (2.5). The maximum over all components $x(i)$ is denoted sft .

comment 4, Accept: The new step dt is accepted, $x(i)$ contain the corresponding new x -values. The first action is to check the stop criterion: If $fstop = constant$, $tstop$ is true and integration continues until $t = fstop$; the double if-statement ensures the correct setting of $tstop$ in the first step; the condition $dt*dt0 \leq 0$ ensures exit if the procedure is called with a dts much larger than the integration interval or if the step length extrapolation yields a crazy result. If $fstop$ is not constant, exit is made when its sign changes ($c0*c1 < 0$).

comment 5, New dt: The size of the step just accepted is stored as $dt0$, and a guess for the next step is calculated according to (2.8). While the very last step dt may be very small in order to 'hit' exactly $t = fstop$, $dt0$ always contains the last, normally calculated and accepted step length. Therefore $dt0$ is used to set the return value of dts after the last step.

comment 6: The following statements perform the quasi double precision arithmetic on t , using qt as error accumulator, and on $x(i)$ using $qx(i)$ as error accumulators. If the estimated new step length dt overshoots the goal $t = c1$, dt is regulated accordingly. Finally $t1$ and $x1(i)$ are updated.

comment 7: The step is rejected, $x(i)$ are reset to the old values $x1(i)$ and a smaller step length dt is calculated according to section 2.3. Furthermore mu , which in the next accepted step will be used as 'old value' $mu0$, is increased a little; this will make the dt -extrapolation after the first accepted step a little more cautious. In case the very first step is much too large and hence rejected, $dt0$ has to be re-initialized; this is done in the last statement of the loop, where $dt0 = dt$ is used to indicate the first step.

comment 8, Terminating step: This section is entered only for variable fstop, in order to interpolate over the last accepted step to find the t-value and corresponding x(i)-s for which fstop = 0. The variables have the following contents (snapshot values) just before entering the interpolation itself (the statement: for s:= ...)

t1, x1(i): Old accepted values where fstop still had the original sign, say fstop = c0 > 0.
t, x(i): Newest calculated point, where fstop = c1 < 0.
h = dt0 = distance from t1 to t.
d, q, w, b: Contain initial values for different weights and coefficients used in the interpolation.

comment 9, Zero determination: t1 is used as the fixed base point. The end points of the 'newest' root-containing interval are t1 + h0 and t1 + h where fstop has the values c0 and c1; s = h - h0 is the length of this interval. The next point tested for fstop = 0 is t1 + dt, where dt is a weighted mixture of a bisection $dt = h - s/2$ and a slightly modified regula falsi $dt = h + (c0 + c1)/(c0 - s1) * s/2$.

solineq

input:

The procedures decompose and solve, R0SL No. 53-MF13 must be present in R04000 before the solineq tape is input by the command i tre.

Program call:

The program is called by the command solineq datascource , e.g. solineq tre. Output will appear on current out. Data is tested by the program.

Data:

n : number of equations
 m : number of right-hand sides
 a(1:n,1:n) : coefficients of the equations
 b(1:n,1:m) : right-hand sides

output trim : - (minus) => standard trim is used,
 which means output of input data
 and solutions with max 70 lines
 per page, max 80 characters per
 line and 7 digits per number
 or
 4 numbers: inputout
 lines per page
 characters per line
 digits printed in output.
 inputout=0: input data is not printed
 inputout=1: input data is printed
 20<=lines per page<=100
 40<=characters per line<=130
 3<=digits in output<=11.
 Standard trim is 1,70,80,7

Examples:

data set:

2 1
 7.1 4
 1.8 2.3

8.1 17

-

program call:

o tpf
 solinq trf

punch output:

Data:

2x2 matrix

a 1,1= 7.100000₁₀ 0 a 1,2= 4.000000₁₀ 0
 a 2,1= 1.800000₁₀ 0 a 2,2= 2.500000₁₀ 0

right-hand sides:

b1= 8.100000₁₀ 0 b2= 1.700000₁₀ 1

S O L U T I O N S

$x1 = -5.407448_{10} \ 0$ $x2 = 1.162522_{10} \ 1$

data set:

$n=6$

$m=3$

a:

1 4 6 7 8 5

7 4 5 9 5 5

8 4 7 5 9 2

4 5 6 2 3 9

3 4 1 7 1 8

8 5 6 4 9 9

b:

1 2 3 4 5 6

2 3 4 5 6 7

9 8 7 6 5 4

0 40 60 3

program call:

soldata=set 5

soldata=edit trf

f

o lp

solineq soldata

printer output:

S O L U T I O N S

set no 1

$x1 = 2.95_{10} -1$

$x2 = 2.77_{10} \ 0$

$x3 = -1.54_{10} \ 0$

$x4 = -1.34_{10} -1$

$x5 = 1.46_{10} -2$

$x6 = -6.39_{10} -2$

set no 2

$x1 = 2.17_{10} -1$

$x2 = 2.51_{10} \ 0$

$x3 = -7.94_{10} -1$

$x4 = -1.05_{10} -2$

$x5 = -3.08_{10} -1$

$x6 = -1.87_{10} -1$

set no 3

$x1 = 1.39_{10} -1$

$x2 = 2.24_{10} \ 0$

$x3 = -4.80_{10} -2$

$x4 = 1.13_{10} -1$

$x5 = -6.30_{10} -1$

$x6 = -3.10_{10} -1$

Program listing:

```
(clear solineq
solineq=set 36
solineq=algol
end)

begin
comment  program for solving a system of linear equations by
means of two standard procedures : decompose + solve.;

integer n,i,j,m,M,k,k1,k2,k3,digits,sp1,
linesperpage,charperline,linespergroup,nosperline,linesleft,blines;
real r,rm,rm,rr,lay,lay1,lay2;
boolean first,consolinput,inputout,error,b1,biga,bigb,sp;
array arr(1:2);

procedure changepage(left,wanted); integer left,wanted;
begin
  if left<wanted then
  begin
    linesleft:=linesperpage;
    write(out,<:<12>:>);
  end
end changepage;

procedure alarm(r,s); real r; string s;
begin integer i;
  i:=r;
  write(out,<:10>:>);
  if i=r then write(out,i) else write(out,r);
  write(out,s,<:<10>:>);
  error:=true
end alarm;

procedure notused;
begin boolean first;
integer i,j;
  first:=true;
  repeatchar(in);
rep:
  i:=readchar(in,j);
  if -,consolinput and j<25 or consolinput and j<10 then
  begin
    if first then
    begin
      if j=10 or j=32 then goto rep
      else write(out,<:<10>Following was not input: >)
    end;
    first:=false;
    write(out,false add j,1);goto rep
  end
end notused;
```

```
procedure testnandm;
begin
  n:=rn;
  if rn<n or n<1 or n>200 then
    alarm(rn,<: is not an acceptable value of n:>);
  M:=rm;
  if M<1 or M>100 or M<>rm then
    alarm(rm,<: is not an acceptable value of m:>);
  if error then goto STOP;
  system(2,i,arr);
  j:=6*n+4*r*M+4*n*n+200;
  if j>i then
    begin
      write(out,<:<10>A dataset of n= :>,<<d>,n);
      write(out,<: and m= :>,<<d>,M);
      write(out,<:
is too big for this process and will result in a stack message.
In this case you should increase your process by approx.<10> :>,
      <<d>,100*round((j-i)/100),
      <: bytes or preferable more<10>:>);
    end;
end testnandm;

procedure testmatrix(c,n,m,k,name,big);
array c; integer n,m,k; string name; boolean big;
begin integer i,j; boolean first;
  first:=true;
  big:=false;
  for i:=1 step 1 until n do
    for j:=1 step 1 until m do
      if c(i,j)>_616 and i*j<=k then
        begin
          if first then write(out,<:<10>Dataerror in matrix:>,
                                name,<:<10>Illegal number in element::>);
          write(out,<<dd>,<:<10>:>,i,<:, :>,j);
          error:=true;
          first:=false;
        end
      else if c(i,j)>_9.99_9 then big:=true;
      if -,first then
        begin
          write(out,<:<10><10>:>); setposition(out,0,0);
        end;
    end testmatrix;

begin integer array ia(1:20);
  getzone(in,ia);
  r:=r shift 24 add ia(2) shift 24 add ia(3);
  consolinput:=r=real<:conso:> add 108 or
              r=real<:termi:> add 110;
end;

error:=false;
i:=read(in,rn,rm);
if i<>2 then
  begin
    write(out,<:<10>empty reader<10>:>);
```

```

    if i=1 then
    begin
        n:=rn;
        write(out,<:n=:>);
        if n=rn then write(out,n) else write(out,m);
        write(out,<:<10>no input to m=:>);
    end;
    goto STOP
end;

testnandm;

begin array b(1:n),a(1:n,1:n),bm(1:n,1:M);
integer array p(1:n);

inputout:=true;
linesperpage:=70;
charperline:=80;
digits:=7;

sp:=false add 32;
k:=read(in,a);
k1:=read(in,bm);
r:=100;
k2:=read(in,r);
if k2=1 and r<616 then k3:=read(in,rn,rm,rr) else k3:=0;
notused;

if k<n then
begin
    write(out,<:<10>:,<<dd>,k,<: elements input to matrix,
should be n=:>,n);
    error:=true;
end;

testmatrix(a,n,n,k,<: a:,>,biga);

if k1<n then
begin
    write(out,<:<10>:,<<dd>,k1,
        <: elements input to right-hand sides,
should be n=:>,n);
    error:=true;
end;

testmatrix(bm,n,M,k1,<: b:,>,bigb);

if k3=1 or k3=2 then write(out,<:
output trim should be 0 or 4 and not:,<<-d>,k3+1,<: numbers:);
if k2=1 and r<616 then
begin
    if r<0 and r>1 then
    begin
        i:=r;
        write(out,<:<10>inputout must be either 0 or 1 and not:);
        if i=r then write(out,i) else write(out,r);
    end
    else inputout:=r=1;
end;

```

```

if k3>0 then
begin
  i:=rn;
  if i<rm or i<20 or i>100 then
  begin
    write(out,<:<10>:>);
    if i=rm then write(out,i) else write(out,rm);
    write(out,<: is not acceptable as lines per page:>);
  end
  else linesperpage:=i;
end;
if k3>1 then
begin
  i:=rm;
  if i<rm or i<40 or i>130 then
  begin
    write(out,<:<10>:>);
    if i=rm then write(out,i) else write(out,rm);
    write(out,<: is not acceptable as characters per line:>);
  end
  else charperline:=i;
end;
if k3=3 then
begin
  i:=rr;
  if i<rr or i<3 or i>11 then
  begin
    write(out,<:<10>:>);
    if i=rr then write(out,i) else write(out,rr);
    write(out,<: is not acceptable as digits in output:>);
  end
  else digits:=i;
end;

if error then goto STOP;

k1:=if n>9 then 1 else 0;
i:=digits+k1+(if biga then 12 else 11);
charperline:=charperline-i;
nosperline:=charperline/(i+1)+1;
if nosperline>n then nosperline:=n;
sp1:=(charperline+i-nosperline*i)/(nosperline-1)-1;
if sp1<0 then sp1:=0 else
if sp1>4 then sp1:=4;
linespergroup:=(n/nosperline)+
  (if n mod nosperline =0 then 1 else 2);
blines:=3+M*(linespergroup+(if M>1 then 4 else 0));
linesleft:=linesperpage;

lay1:=real( case digits-2 of(
  <<-d.dd10-d>, <<-d.ddd10-d>, <<-d.dddd10-d>,
  <<-d.ddddd10-d>, <<-d.dddddd10-d>, <<-d.dddddddd10-d>,
  <<-d.ddddddddd10-d>, <<-d.ddddddddd10-d>, <<-d.ddddddddd10-d>));
lay2:=real(case digits-2 of(
  <<-d.dd10-dd>, <<-d.ddd10-dd>, <<-d.dddd10-dd>,
  <<-d.ddddd10-dd>, <<-d.dddddd10-dd>, <<-d.dddddddd10-dd>,
  <<-d.ddddddddd10-dd>, <<-d.ddddddddd10-dd>,
  <<-d.ddddddddd10-dd>));

```

```

if inputout then
begin
  begin
    write(out,<:<12<10>
Data:<10<10>:>,<<d>,n,<:x:>,n,<: matrix:>);
    linesleft:=linesleft-4;
    lay:=if biga then lay2 else lay1;
    for i:=1 step 1 until n do
    begin
      write(out,<:<10>:>);
      for j:=1 step 1 until n do
        write(out,if j mod nosperline=1 then <:<10>:> else <:;>,
          sp,if j mod nosperline=1 then 0 else sp1+1,<:a:>,
          <<dd>,i,<:,>,<<d>,j,if n>9 and j<10 then <: =:> else <:=:>,
          string lay,a(i,j));
        linesleft:=linesleft-linespergroup;
        changepage(linesleft,linespergroup);
      end;
      if blines<=linesperpage then changepage(linesleft,blines);
      changepage(linesleft,linespergroup);
      if M>1 then linespergroup:=linespergroup+4;
      write(out,<:<10<10<10>right-hand sides:>);
      linesleft:=linesleft-5;
      lay:=if bigb then lay2 else lay1;
      for j:=1 step 1 until M do
      begin
        if M>1 then write(out,<:<10<10<10>set no:>,j,<:<10>:>);
        write(out,<:<10>:>);
        for i:=1 step 1 until n do
          write(out,if i mod nosperline=1 then <:<10>:> else <:;>,
            sp,if i mod nosperline=1 then 3 else sp1-1+(if -,biga and bigb
            then 3 else if biga and -,bigb then 4 else 5),
            <:b:>,<<d>,i,if n>9 and i<10 then <: =:> else <:=:>,
            string lay,bm(i,j));
          linesleft:=linesleft-linespergroup;
          changepage(linesleft,linespergroup);
        end;
        write(out,<:<10>:>);
        linesleft:=linesleft-1;
        if M>1 then linespergroup:=linespergroup-4;
      end;
    end;
  end;

  b1:=true;

  if -,decompose(a,p,1) then
  begin
    write(out,<:<10>The given matrix is singular<10>:>);
    goto STOP
  end;
  if M>1 then linespergroup:=linespergroup+4;
  for m:=1 step 1 until M do
  begin
    for i:=1 step 1 until n do b(i):=bm(i,m);
    solve(a,p,1,b);
    bigb:=false; for i:=1 step 1 until n do if b(i)>9.9x9 then bigb:=true;
    lay:=if bigb then lay2 else lay1;

```

```

if b1 then
begin
  if -,inputout then write(out,<:<12>:>);
  if blines<=linesperpage then changepage(linesleft,blines);
  changepage(linesleft,linespergroup);
  write(out,<:<10×10×10>:>,false add 32,
        nosperline×(12+sp1+digits)//2-8,<:S O L U T I O N S:>);
  linesleft:=linesleft-5;
end;
if M<>1 then write(out,<:<10×10×10>set no:>,m,
                  <:<10>:>) else write(out,<:<10>:>);
for i:=1 step 1 until n do
  write(out,if i mod nosperline=1 then <:<10>:> else <::>,
        sp,if i mod nosperline=1 then 3 else sp1-1+(if -,biga
        and bigb then 3 else if biga and -,bigb then 4 else 5),
        <:x:>,<<d>,i,
        if i>9 and i<10 then <: =:> else <: =:>,
        string lay,b(i));
  linesleft:=linesleft-linespergroup;
  changepage(linesleft,linespergroup);
  b1:=false;
end m;

write(out,<:<10×10×12>:>);
end of block;
STOP:
end

```

solvesym

solvesym

boolean procedure solvesym(n, m, A, X);

1. Function and parameters.

The procedure solves the generalized linear algebraic equation

$$M \times X = B$$

where M is a symmetrical $n \times n$ matrix of coefficients, B is a given $n \times m$ matrix and X is the unknown $n \times m$ matrix.

In this procedure X and B are stored in the same array X, B on entry and X at return.

The lower half of $M(1:n, 1:n)$ is stored in an array A such that $M(s, r) = M(r, s) = A(r \times (r-1) // 2 + s)$. $s \leq r$.

If M is singular then the procedure will come out with the value false. For each degree of degeneration one of the diagonal elements in M, say $A(s \times (s+1) // 2)$, is zero, and the corresponding elements of X, $X(s, k)$, $k = 1, 2, \dots, m$, must be zero or very small if the given equation $M \times X = B$ has a solution.

Procedure heading:

boolean procedure solvesym(n, m, A, X);

value n, m;

integer n, m;

array A, X;

Call parameters:

integer n The number of equations.

integer m the number of right sides.

real array $A(1:m \times (n+1) // 2)$ the lower half of the coefficient matrix
 $M(r, s) = M(s, r) = A(r \times (r-1) // 2 + s)$, $s \leq r$.

Call and Return parameter:

real array $X(1:n, 1:m)$ is the right sides at call and the solutions at return.

Return parameter:

boolean solvesym. false if A is singular, true if A is nonsingular.

2. Mathematical method.

The method is the usual Gauss reduction with diagonal pivoting. The pivoting criterion is the following:

In each step a new pivot index r is selected among the not used indices so that

$$\text{abs } M(r, r) / \max \text{ abs } M(r, s)$$

attains its maximum; and the reduction is carried out in the usual way by making the r 'th column = 0 under the diagonal. However, if all possible diagonal elements are zero this can not be done. In that case an index r is found so that

$$\begin{aligned} &\max \text{ abs } M(r, s) \\ &s \neq r \end{aligned}$$

attains its minimum.

If this minimum is zero then the whole row is zero and the matrix is singular. In this case the procedure value is set to false and the corresponding r is set to 'has been pivot element', and the search for another r is continued. However, if the minimum is > 0 then row k is replaced by $(\text{row } k) + (\text{row } r) \times M(k, r)$ for all k which have not been pivot index. This will make at least one diagonal element $\neq 0$ and the pivot index may be selected as above. The process can now go on until there are zeros under the whole diagonal of M and the solution obtained by simple backward elimination.

If M is singular some of the diagonal elements $M(r, r)$ are zero. During the backward reduction the division by such a diagonal element is skipped. Moreover, the corresponding elements in the r 'th row of $X(r, k) = B(r, k)$ will have to be zero (or very small compared to the original values) in case the given equation has a solution.

3. Accuracy, time and storage Requirement.

Accuracy.

In practice the relative error measured as $||AX - B|| / ||X||$ has been found to be about 10^{-10} . This is not an errorbound, the errorbound has been discussed in detail in literature see e.g. Forsythe og Moler. (ref).

Time.

For $m = 1$ the time is $.2 \times (n+1) \times 3$ ms

Storage requirement.

The procedure is 4 segments long on backing-store. It uses $70 + 3.5 \times n$ words in stack.

Typographical length: 103 lines, 4 segments.

4. Test and discussion

The procedure is intended for use in such cases where the total matrix M is too big for the available store. A program using decompose and solve will be faster than a program using solve_sym, even if the program must generate the matrix M from the half matrix A .

The procedure has been tested by some equations with coefficients chosen at random and by a representative set of singular equations.

The following program will read n , m , A , B , solve the linear algebraic equation $A \times X = B$ and write out the X :

Input, solution and output of a symmetrical set of linear algebraic equations

```
begin integer n, m, i, j, k, l; boolean s;
  read(in, n, m);
  begin array A(1:(n*(n+1)) shift (-1)), B(1:n, 1:m);
    read (in, A, B);
    s:= -, solvesym(n, m, A, B);
    if s then write(out, <:<10> A is singular:>);
    write(out, <:<10>:>);
    for i:= 1 step 5 until m do
      begin
        j:= if i+4 < m then i + 4 else m;
        for k:= i step 1 until j do write(out, <<_____ddd>, k);
        for k:= 1 step 1 until n do
          begin
            write(out, <:<10>:>, <<ddd>, k, if s then (if A((k*(k+1)) shift (-1))=0
              then <:x_:> else <: _:>) else <: _:>);
            for l := i step 1 until j do
              write(out, <<_-d.ddddddd_-dd>, B(k, l))
            end k;
            write(out, <:<12><10>:>)
          end l
        end i
      end A
    end program;
```

5. Reference

George Forsythe and Cleve B. Moler: Computer Solution of Linear Algebraic Systems. Prentice-Hall, Inc. (1967).

6. Procedure text.

```
solvesym = set 4  
solvesym = algol  
external
```

```
boolean procedure solve_sym(n,m,A,X);  
message solve sym, version 18 11 69, RCSL 53-M6;  
    value n,m; integer n,m; array A,X;  
begin integer i,j,k,r,s,t;  
    real ai,ak,ar,mi;  
    array M(1:n); integer array R(1:n); boolean array B(1:n);  
  
    j:=0; solve_sym:=true;  
    for i:= 1 step 1 until n do  
    begin  
        mi:=0;  
        for k:=i-1 step -1 until 1 do  
        begin  
            if abs A(k+j)> mi then mi:=abs A(k+j);  
            if abs A(k+j)>M(k) then M(k):=abs A(k+j)  
        end k;  
        M(i):=mi; B(i):=true; j:=j+i;  
    end i;  
    s:=1;  
    for t:= 1 step 1 until n do  
    begin  
        mi:=ak:=-1;  
        for i:= 1 step 1 until n do if B(i) then
```

```

begin
  ai:=abs A((ix(i+1))shift(-1));
  if M(i)>0 then
    begin
      if mi×M(i)<ai then
        begin
          if ai<0 then
            begin
              mi:=ai/M(i); s:=i
            end else if M(i)×ak<1 then
              begin
                ak:=1/M(i); s:=i
              end
            end
          end
        end (i) > 0 else
        begin
          R(t):=i; B(i):=false; t:=t+1;
          if ai=0 then solve_sym := false
        end M(i)<0
      end i;
    if B(s) then
      begin
        r:=(s×(s-1))shift(-1); ar:= A(r+s);
        if ar=0 then begin ar:=-1; t:=t-1 end else R(t):=s;
        B(s):= false;
        for i:= 1 step 1 until n do if B(i) then
          begin
            j:=(ix(i-1))shift(-1);
            ai:=A(if i<s then r+i else j+s)/ar; mi:=-1;
            for k:= i step -1 until 1 do if B(k) then
              begin
                ak:= A(j+k):=A(j+k)
                  -ai×A(if k<s then r+k else (k×(k-1))shift(-1)+s);
                if abs ak>mi then
                  begin
                    if i=k then goto L1;
                    mi:=abs ak
                  end;
                if abs ak>M(k) then M(k):=abs ak;
              end
            end k;
            M(i):=mi;
            for k:=1 step 1 until m do X(i,k):=X(i,k)-ai×X(s,k)
          end i;
        L1:

```

```

    if A(r+s)=0 then
    begin
        mi:=0;
        for k:=1 step 1 until n do if B(k) then
        begin
            ak:=abs A(if k<s then r+k else (k*(k-1))shift(-1)+s);
            if ak>mi then mi:=ak;
            if ak>M(k) then M(k):=ak;
        end k;
        B(s):= true
    end A(r+s)=0
end B(s);
end t;
for t:= n step -1 until 1 do
begin
    s:=R(t); r:=(s*(s-1))shift(-1);
    for i := t+1 step 1 until n do
    begin
        j:=R(i); ai:=A(if j<s then r+j else (j*(j-1))shift(-1)+s);
        for k:=1 step 1 until m do X(s,k):= X(s,k)-ai*X(j,k)
    end i;
    ai:= A(r+s);
    if ai<> 0 then for k:= 1 step 1 until m do X(s,k):=X(s,k)/ai
end t
end solve sym;

```

comment

Call parameters:

integer n the number of equations.
integer m the number of right sides.
real array A(1:m*(n+1)//2) the lower half of the coefficient matrix.

$$M(r, s) = M(s, r) = A(r*(r-1)//2 + s)$$

Call and Return parameter:

real array X(1:n, 1:m) is the right sides at call and the solutions at
return.

Return parameter:

boolean solvesym. false is A is singular, true if A is nonsingular;

zerol(x, F, a, b, eps)

zerol(x, F, a, b, eps)

1. Function and Parameters.

1: Function:

The boolean procedure evaluates one zero of the function $F(x)$ within the interval $a \leq x \leq b$. The method is based on regula falsi and bisection combined with an adaptive parameter giving the weights of regula falsi and bisection.

Call parameters:

a, b: real value parameters specifying the end points of the interval within which the zero is calculated. This interval is $a \leq x \leq b$ if $a < b$, otherwise $a \leq x \leq b$.

eps: A real name parameter giving the accuracy with which the zero is determined.

Relative accuracy may be specified by substituting an expression like $\delta \times x$ for eps.

If eps specifies an accuracy that is not obtainable calculations are stopped with the obtainable accuracy.

Return parameters:

x: a real name parameter being the independent variable in the expression giving F .

On exit the `z e r o` determined by `zero1`.

Need not be initialized.

zero1: The boolean procedure name is set to false if
 $F(a) > 0$ and $F(b) > 0$ or $F(a) < 0$ and
 $F(b) < 0$, otherwise zero1 is true.

Other parameters:

F: a real name parameter specifying the function for
 which the zero is to be evaluated.
F m u s t be supplied as an expression depending
 on x.

2. Method

The procedure calculates for each iteration a new value as a weighted mean between a regula falsi and a bisection value:

$a \leq x \leq b$ being the interval in which the zero is to be evaluated, with $fa = F(a) > 0$ and $fb = F(b) < 0$, the following algorithm is used:

$xr = a - fa \times (b - a) / (fb - fa)$
(i.e. x value obtained by regula falsi)

$xb = (b + a) / 2$
(i.e. x value obtained by bisection).

The new value of x is now calculated as

$x = xr + (xb - xr) \times vb$

where the weight factor, vb satisfies $0 \leq vb \leq 1$.

And the value of vb is calculated as

$vb := \text{if } a < xr \text{ and } xr < b \text{ then } vb \times vb / 2 \text{ else } 1;$

i.e. if x_r , the x value calculated by regula falsi method, is inside the new interval then regula falsi might be better than the x just calculated and more weight are given to regula falsi in the next iteration (i.e. smaller vb), otherwise the next iteration is pure bisection ($vb = 1$).

$f = sg \times F(x)$ is evaluated for the new x value and a new interval ($a|b$) is determined as:

```
if  $f > 0$  then begin  $b := x$ ;  $fb := f$  end  
else begin  $b := x$ ;  $fa := f$  end;
```

The factor sg is $f = sg \times F(x)$ is introduced in order to give a simple algorithm inside the iteration loop.

Before starting iteration sg is initialized as

```
 $sg :=$  if  $fa > 0$  then 1 else -1;
```

and all values of F are multiplied by sg , (i.e. $fa > 0$ and $fb < 0$).

If the parameters specifying a and b gives $b < a$ then a interchange of these two parameters are made in the start of the program.

However if $F(b)$ and $F(a)$ are both either greater than or less than 0 then the method does not work and the boolean name `zero1` is set to false indicating that no zero is evaluated, otherwise `zero1` is true.

3. Accuracy and storage requirement.

3.1. The accuracy is determined by the input parameter `eps` giving the absolute precision of the zero. If however an expression giving `eps` includes the factor x (the independent variable) then relative precision is automatically used.

If an accuracy higher, than the one obtainable in RC 4000, is specified then a result with the highest obtainable precision is delivered.

3.2. Storage requirements:

1 segment + 9 real variables

4. Test and discussion.

zero1 is tested by use of the 6 functions used in ref. 1 for test of Gier procedures.

Results of this test using testprogram as given in section 7 are:

Textexamples for : external boolean procedure zero1(x,F,a,b,eps)

F(x)	a	b	eps	x	iter
5.33+2.6xx	-9.9	2.1	'-6	-2.05' +0	8
ln(x/0.7)	0.1	2	'-8	7.00' -1	12
exp(x)-0.4	-5	1	xx'-7	-9.16' -1	12
sin(x)-sin(1.55)	-3	1.59	'-5	1.59' +0	11
xxx3 + x	-0.5	2	'-8+abs(x)x'-6	1.93' -15	9
xxx5	-1	2	'-6	6.96' -7	24

x = the zero calculated by zero1

iter = the number of references to F

These result may be compared with results from ref. 1 showing that although using a very simple strategy zero1 is very fast.

5. References.

Bo Munch-Andersen: Zero, Algol procedure, Regnecentralen October 1965,
Gier System Library, Order No. 409.

6. Algol program

```
zero1:=set 1
zero1:=algol
external
```

```
boolean procedure zero1(x,F,a,b,eps);
value a,b; real x,F,a,b,eps;
begin
  real fa,fb,f,vb,sg,v,xr;
  comment 1;

  zero1:= true;
  if a > b then begin f := a; a := b; b := f end;
  x := a;
  f := F;
  sg := if f > 0 then 1 else -1;
  fa := sg*x;
  if fa = 0 then goto out;
  x := b;
  fb := sg*F;
  if fb = 0 then goto out;
  if fb > 0 then begin zero1 := false; goto out end;
  vb := 1;
```

```
next:
  v := b-a;
  x := (b+a)/2;
  if v < 2*abs(eps) or v < 1.2e-10*abs(x) then goto out;
  comment 2;
  xr := a-fa*v/(fb-fa);
  x := xr+(x-xr)*vb;
  f := sg*x;
  if f = 0 then goto out else
  if f > 0 then begin a := x; fa := f end
  else begin b := x; fb := f end;
  comment 3;
  vb := if a < xr and xr < b then vb*v/2 else 1;
  goto next;
```

```
out:
end;
```

```
comment
```

```
1:
```

Reference:

RC4000 System Library
Order No. 55-D44
A/S Regnecentralen, July 1969
N. Schreiner Andersen

Function:

The boolean procedure evaluates one zero of the function $F(x)$ within the interval $a \leq x \leq b$. The method is based on regula falsi and bisection combined with an adaptive parameter giving the weights of regula falsi and bisection.

Call parameters:

a, b : real value parameters specifying the end points of the interval within which the zero is calculated. This interval is $a \leq x \leq b$ if $a < b$ otherwise $b \leq x \leq a$.

eps : A real name parameter giving the accuracy for which the zero is determined.

Relative accuracy is specified through an expression with factor x , i.e. $x \times 10^{-7}$ gives a relative accuracy of 10^{-7} .

If eps specifies an accuracy that is not obtainable within RC4000 calculations are stopped with the obtainable accuracy.

Return parameters:

x : a real name parameter being the independent variable in the expression giving F .

On exit the zero determined by zero1 .

zero1 : The boolean procedure name is set to false if $F(a) > 0$ and $F(b) > 0$ or $F(a) < 0$ and $F(b) < 0$, otherwise zero1 is true.

Other parameters:

F : a real name parameter specifying the function for which the zero is to be evaluated.
 F must be supplied with an expression depending on x .

2: In order to avoid that calculations can not stop because of too small eps (below the precision obtainable on RC4000) a security is put in here causing stop on $v < 1.2 \times 10^{-10} \times \text{abs}(x)$.

3: A new weight, vb is calculated before next iteration;

end zero1 ;

7. Testprogram

A/S Regnecentralen
Testprogram for procedure zero1
NSA, 1.09.69.

```

begin
real procedure F(n);
integer n;
begin
  i := i + 1;
  F := case n of (5.33+2.6*x, ln(x/0.7), exp(x)-0.4, sin(x)-sin(1.55),
                  x**3 + x, x**5);
end F;

real x; integer i;

write(out,<:
Testexamples for : external boolean procedure zero1(x,F,a,b,eps)

:>);
write(out,<:
      F(x)          a      b      eps          x          iter
:>);

i := 0; zero1(x,F(1),-9.9,2.1,n-6);
write(out,<:
5.33+2.6*x          -9.9  2.1    n-6          :>,<<  -d.ddn+dd>,x,<<  -dd>,i);

i := 0; zero1(x,F(2),0.1,2,n-8);
write(out,<:
ln(x/0.7)           0.1  2      n-8          :>,<<  -d.ddn+dd>,x,<<  -dd>,i);

i := 0; zero1(x,F(3),-5,1,x**n-7);
write(out,<:
exp(x)-0.4          -5    1      x**n-7      :>,<<  -d.ddn+dd>,x,<<  -dd>,i);

i := 0; zero1(x,F(4),3,1.59,n-5);
write(out,<:
sin(x)-sin(1.55)    -3    1.59  n-5          :>,<<  -d.ddn+dd>,x,<<  -dd>,i);

i := 0; zero1(x,F(5),-0.5,2,n-8+abs(x)*n-6);
write(out,<:
x**3 + x            -0.5  2      n-8+abs(x)*n-6 :>,<<  -d.ddn+dd>,x,<<  -dd>,i);

i := 0; zero1(x,F(6),-1,2,n-6);
write(out,<:
x**5                -1    2      n-6          :>,<<  -d.ddn+dd>,x,<<  -dd>,i);

write(out,<:<10x10x10>
      x      = the zero calculated by zero1
      iter   = the number of references to F
:>);
end testprogram;

```


data survey, Part 1

CONTENTS:

	Page
1. Statistical theory	2
2.1. Input/output system	3
2.2. Syntax of input data	3
2.3. Semantics of input	4
2.4. Error messages	5
3. Program tape, storage requirements, capacity, and running time	7
4.1. Program structure	7
4.2. Method	8
5. References	9
6. User's example	10
6.1. Preparation of input tape	10
6.2. Output, interpretation and running time	11

APPENDIX:

7. Program manuscript in ALGOL 5	16
----------------------------------	----

1. STATISTICAL THEORY.

It is assumed, that the observations (of the same variable) are stochastically independent and identically distributed with a distribution having central moments of order ≤ 4 ,
(e.g. $EX < \infty$ and $E(X - EX)^{\times j} < \infty$ for $j = 2, 3, 4$).

If we put $\text{mean} = m1 = \text{sum}(1 \leq i \leq n) (X(i)) / n$ and $S(j) = \text{sum}(1 \leq i \leq n) ((X(i) - m1)^{\times j})$ for $j = 2, 3, 4$, the program computes:

```
m2 = variance = S(2) / (n - 1)
    std.dev. = sqrt(m2)
m3 = skewness = S(3) / (S(2)^{\times 1.5})
m4 = kurtosis = S(4) / (S(2)^{\times 2})
Students' t = m1 / (std.dev. / sqrt(n)).
```

The 95 pct. confidence interval for mean is defined as:

$m1 - 1.96 \times \text{std.dev.} / \sqrt{n} < \text{mean} < m1 + 1.96 \times \text{std.dev.} / \sqrt{n}$.

The histograms are made in such a way, that the column height (and not as usual the column area) is proportional to the group density.

The chi square distribution test is based on

```
chi square = sum(1 \leq i \leq k) ((obs(i) - nxp(i))^{\times 2} / (nxp(i)))
            = -n + sum(1 \leq i \leq k) (obs(i)^{\times 2} / (nxp(i))),
```

with n = number of observations, k = number of groups, $hxp(i)$ = the expected and $obs(i)$ = the observed number of observations in the i -th group. For $nxp(i) \rightarrow \infty$ chi square is approximately chi square distributed with $k-3$ degrees of freedom, but the adaption is only acceptable when $\min(1 \leq i \leq k) (nxp(i)) > 5.0$, and the program therefore unites groups (by deleting some of the grouplimits) in order to let this condition be true.

The program use the following approximations of the normal distribution cumulative function $\phi(y)$ (see ref (4)):

(1) $\phi(y) \text{ eq. } \exp(-y^{\times 2} / 2) \times .39894 \times ((.9372980xp - .1201676)xp + .4361836)xp$
with $p = 1 / (\text{abs}(y) \times .33267 + 1.0)$ for $y \leq 0$.

(2) $\phi^{-1}(y)$ eq. $-p+(2.30753+.27061xp)/((.04481xp+.99229)xp+1.0)$
 with $p = \sqrt{\ln(y) \times (-2.0)}$ for $0.0 \leq y \leq 0.5$

which combined with $\phi(-y) = 1 - \phi(y)$ for x real

and $\phi^{-1}(y) = -\phi^{-1}(1-y)$ for $0 \leq y \leq 1$ gives total approximations
 of ϕ and ϕ^{-1}

2.1. INPUT/OUTPUT SYSTEM.

For facilitating the problems about input/output, the two standard zones in/out (connected to current input/output) are used for input/output on character level. Output is made for format A4 vertical and produced with 8 leading spaces on each line.

Besides the computed output, the output will contain small notes about the different parts of input, and possibly also some error messages.

2.2. SYNTAX OF INPUT DATA.

The input file (= current input) must contain data in accordance with the following syntax:

```

<input file> ::= <identification><data set>* <end>01 EM
<identification> ::= < <legal character>0110 <
<legal character> ::= <letter>|<digit>|+|-|×|/|=|;|(|)|'|:|.|,|NL|SP
<end> ::= e
<data set> ::= <variables>|<variables><specification>
<specification> ::= <groups>|<execute mark>|<transgenerations>|
                  <compute>|<subset>
<variables> ::= <experiment number>,<variable number>,<number of cases><
                  <name> < <observation>23000 <
<name> ::= <empty>|<letter><legal character>040 <
<observation> ::= <real>,<|<observation><checksum> s
<groups> ::= g <grouplimit>248 <
<grouplimit> ::= <real bigger than preceding (if one)>,

```

```
<execute mark> ::= <
<transgenerations> ::= t <trans. spec.>18 <
<trans. spec.> ::= <trans. type>, <constant 1>, <constant 2>,
<trans. type> ::= 1|2|3
<compute> ::= c <indicator normal>, <indicator exponential> <
<subset> ::= s <first case>, <last case> <
<constant> ::= <real>
<experiment number> ::= <integer>
<variable number> ::= <integer>
<number of cases> ::= <integer>
<indicator> ::= <integer>
<first case> ::= <integer>
<last case> ::= <integer>
```

Extra commas, spaces, and new line are allowed everywhere, and are blind outside texts. Tape feed, all holes, FF, HT, and VT are blind everywhere in <input file>.

2.3. SEMANTICS OF INPUT.

It appears from the definition of <data set>, that one set of observations might be object for several examinations with different <specification>'s because every <execute mark> causes an examination in accordance with the <specification>'s before the <execute mark>. Every part of <specification> will be valid as long as it can be interpreted correctly, or until it is actively altered by writing a new part of <specification> of the same kind. The <specification>'s are always attached to the preceding variable, and if several <specification>'s of the same kind are specified without an <execute mark> between, the last <specification> of each kind will be valid at <execute mark>.

Check sums among observations may appear everywhere (and may be omitted totally). The notion <indicator> has the following explanation: if indicator > 0 the corresponding fractile diagram will appear in output, otherwise it will be suppressed. The program is initialized by c 1, -1 <, and the histograms can never be suppressed. If the <specification>'s do not contain any <groups> (before a certain <execute mark>), the corre-

sponding examination is performed with grouplimits computed by the program (these grouplimits will be reasonable in the most cases).

If no <subset> has been specified, the corresponding examination will of course cover all the observations in the actual <data set>, and it is evident that the <data set>'s not need to have the same number of observations (cases).

By using <groups> it is important to notice, that the grouplimits must be given in increasing order, whereas the groups not need to be equal in length. The grouplimits and the groupdensities are printed in output in order to obtain correct interpretation in case of groups not equal in length.

By using <transgenerations> it is very important to notice, that the very observations not are stored separately, but are changed by transgenerations, and that all transgenerations are made successively. The program gives possibilities of three kinds of transgenerations, and they are:

```
trans. type = 1 <=> y:= ln(y + constant1)×constant2
trans. type = 2 <=> y:= (y + constant1)××constant2
trans. type = 3 <=> y:= (y + constant1)×constant2.
```

If you want to examine $y := ((\ln(y + 3.5) + 4.2) \times \times 2.1)$ this can be done by writing: t 1, 3.5, 1, 2, 4.2, 2.1 < before the actual <execute mark>, which means $y := \ln(y + 3.5) \times 1$; followed by $y := (y + 4.2) \times \times 2.1$;

The syntax of input data is made in accordance with the syntax which was valid in 1969 for the GIER programs on regression analysis developed by Alex Jessen on A/S Regnecentralen.

2.4. ERROR MESSAGES.

The program can generate the following error messages:

- (a) checksum error: computed sum = dddd check = dddd
means that the sum made by adding observations in <input file> does not agree with the <checksum> given in <input file>.

- (b) cases on tape = ddd cases = ddd
means that the number of cases actual found in <input file> does not agree with the <number of cases> given in the variable head
- (c) but these are rejected
means that the limits in <groups> not has been given in increasing order (if necessary the program will compute some suitable limits).
- (d) a missing execute mark of end of data is generated
means that is met before the sequence <e and this causes an extra examination performed with the current <specification>'s
- (e) extra examination (not specified in input)
means that two sets of <variables> has been given without any <specification> between (not even an <execute mark>), and this causes an extra examination with the current <specification>'s

These error messages do not terminate the run, and as a general rule, the run is continued with the information actually found in <input file>.

The program can produce 7 other error messages, which all will terminate the run with a suitable short error message followed by a copy of 250 characters of <input file> from the place where the error has been detected.

The essence of these error messages are:

- (1) error in art of information
- (2) error in subsets
- (3) error in number of constants in transgeneration information
- (4) error in art of transgeneration
- (5) which has too many observations
- (6) error in number of grouplimits
- (7) identification not terminated by < .

3. PROGRAM TAPE, STORAGE REQUIREMENTS, CAPACITY AND RUNNING TIME.

The program data survey is written in ALGOL 5, and is available on 8 channel paper tape in the normal ISG form (with even parity).

Compilation of the program can be done with the following file processor commands (see ref (2))

```
datasurvey = set 44  
datasurvey = algol tre
```

Reading and compilation of the program takes approx. 27 seconds and the compiled program occupies 44 or 43 segments on backing storage corresponding to compilation with index.yes or index.no.

Compilation requires a process on 12 k bytes, whereas run requires 23 k bytes in order to obtain acceptable speed at running time. The capacity of the program is 3000 observations (exclusive checksums) in each <data set>, 48 grouplimits in each part of <grou> and 48 numbers in each part of <transgenerations> (corresponding to 8 successive transgenerations). These maximal limits can easily be altered by altering the declaration of the arrays intens, group, obs and trngen in the main program block.

Because of the structure of RC 4000 the run time cannot be given exactly, but as a rough guide it can be mentioned that:

```
reading and compilation takes approx. 27 seconds,  
computation of moments takes about 1000 observations/second,  
transgeneration takes 500-3000 observations/second,  
grouping takes about 800 observations/second, and  
that printer output occupies approx. 60-80 pct. of the total running time (excl. compilation).
```

4.1. PROGRAM STRUCTURE.

The program data survey has been made with extensive use of procedures, and the greater part of these have no parameters. This implies that the procedures often use global variables for storing results. In order to get a clearer coding almost all variables are called by names

which shows their use. To obtain that the names also could be pronounced, many of the identifiers are rather long.

The reading part of the program is placed between the labels `data:` and `execute:`, and is formed as a case statement.

In every case, the actual information is controlled as far as possible and a short message about the information is given in output. When a hard error occurs, the program calls procedure `error`, which uses the global integer `inftyp`, and returns to label `exit_program:`.

The errors mentioned in part 2.4 ((a) to (e)) do not terminate the run but gives reasonable error messages and -reactions.

The output is made for format A4 vertical with 8 leading spaces in each line. Pages and `<execute mark>`'s are in output counted 1, 2, and so on, and every page in putput will contain examination number, page number, the text given in `<identification>`, `<variable number>`, and `<name>` in the actual `<variables>`.

4.2. METHOD.

The two central procedures in the program are

(1) procedure grouping;

the procedure groups the observations stored in array `obs(first:last)` according to the `grouplimits` placed in array `group(1:groupnumber)` (with `group(groupnumber):= oo`) and place the result in the integer array `intens(1:groupnumber)`.

The automatical determination of `grouplimits` (controlled by the boolean `groups`) is made according to the following rules:

`grouplength` is based on standard deviation/ q with $q:=$ if number of observations < 100 then 2 else 3. This basic length is rounded according to a logarithmic scale in this way

basic length [1.25, 2.5[$\Rightarrow$ `grouplength:= 2.0`,

basic length [2.5, 6.0[$\Rightarrow$ `grouplength:= 5.0` and

else `grouplength:= 1.0` or 10 (according to the logarithmic scale).

The groups are placed in multipla of the `grouplength`, with at least one observation in the two outermost groups.

The basic grouping is performed in two steps:

- (a) a rough partitioning according to a division of the groups in three disjoint parts
- (b) the exact grouping is then performed in the classes defined by
 $\text{obs class}(i) \Leftrightarrow \text{obs} > \text{group}(i - 1) \text{ and } \text{obs} \leq \text{group}(i)$ (with
 $\text{group}(0) := -\infty$ and $\text{group}(\text{groupnumber}) := +\infty$)

- (2) procedure moments;

the computation of moments is done on the observations stored in $\text{obs}(\text{first}:\text{last})$. The results are placed in variables min , max , m1 , stdev , m2 , m3 , and m4 . The following numerical algorithm is used:

Let $n := \text{last} - \text{first} + 1$ and $S(j) := \sum(\text{first} \leq i \leq \text{last})(\text{obs}(i) - \text{obs}(\text{first})) \times j / n$ for $j = 1, 2, 3$ and 4 , then we obtain (m3 and m4 only for $\text{m2} > 0.0$)

$$\text{m4} := (S(4) - 4 \times S(3) \times S(1) + 6 \times S(2) \times S(1) \times 2 - 3 \times S(1) \times 4) / \text{m2} \times 2$$

$$\text{m3} := (S(3) - 3 \times S(2) \times S(1) + 2 \times S(1) \times 3) / \text{m2} \times 1.5$$

$$\text{m2} := S(2) - S(1) \times 2$$

$$\text{m1} := S(1) + \text{obs}(\text{first})$$

Finally we obtain

$$\text{variance} := \text{m2} \times n / (n - 1) \text{ and } \text{stdev} := \sqrt{\text{m2}}$$

If $\text{m2} = 0$ the examination is terminated with an error message, and the program returns to label `new`.

5. REFERENCES.

- (1) Søren Lauesen: ALGOL 5 Users Manual
A/S Regnecentralen Copenhagen July 1969.
- (2) Søren Lauesen: File Processor Users Manual
A/S Regnecentralen Copenhagen April 1968.
- (3) A. Hald: Statistical Theory with Engineering Applications,
Wiley, London 1952.

- (4) Handbook of Mathematical Functions,
National Bureau of Standards. Chap. 26.2.

6. USER'S EXAMPLE.

6.1. Preparation of input tape.

We have measured the height of 74 soldiers in centimeters and want to examine whether the observations can be described by a normal distribution.

Besides we want some different histograms according to different methods of grouping the observations.

We shall make 2 examinations:

- 1: The automatical determination of grouplimits has to be tried, and we want only fractile diagram in the normal distribution.
- 2: Only normal fractile diagram with grouplimits = 160, 163, 166, 169, 172, 175, and 181 is wanted.

Because the program gives 8 leading spaces in each line we may let every line in <identification> have 8 leading spaces (this will be nice, but is not necessary).

These claims will give an input tape (file) like this:

```
<      identification for testdata to  
      program: data survey  april 1970<
```

```
35,  1,  74<  
height of soldiers 67/68 in centimeters.<
```

```
171,170,176,170,172,168,169,167,167,165,165,179,  
161,160,183,175,178,174,174,173,169,176,168,172,  
172,172,170,175,170,170,171,171,171,170,169,168,  
168,165,162,173,178,174,173,172,169,175,177,172,  
177,173,173,172,171,170,171,173,171,168,167,167,  
164,164,173,166,177,180,174,172,171,168,169,176,  
172,178<<
```

```
c 1,0<  g 160,163,166,169,172,175,178,181<<  
end of <input file>.
```

6.2. USER'S EXAMPLEOutput, interpretation, and run time.

The test run gave this output (4 pages) on current output:

17 11 70 examination number 1 page 1
sph. data-survey

identification for testdata to
program: data survey april 1970

variable number 1 height of soldiers 67/78 in centimeters.

input of observations: total 74 cases without checksum control
execute mark

number of cases	minimum	maximum
74	160.0000	183.0000

mean	variance	stand.dev.	skewness	kurtosis
171.1622	1939.800 ₁₀ -2	4.404316	-0.046365	3.234582

t-test for mean=0 is t = 334.307
which has 73 degrees of freedom.

95 pct. confidence interval is 170.1587 < mean < 172.1657

histogram: every x represents 1 observation

number of cases	upper class-limit	
3	162.0000	XXX
2	164.0000	XX
4	166.0000	XXXX
10	168.0000	XXXXXXXXXX
12	170.0000	XXXXXXXXXXXX
17	172.0000	XXXXXXXXXXXXXXXXXXXX
11	174.0000	XXXXXXXXXXXX
6	176.0000	XXXXXX
6	178.0000	XXXXXX
2	180.0000	XX
0	182.0000	
1		X
total		
74		

17 11 70 examination number 1 page 2
søh. data-survey

identification for testdata to
program: data survey april 1970

variable number 1 height of soldiers 67/78 in centimeters.

fractile diagram in the normal distribution

fraction in pct.	upper class- limit	estimates of									
		position parameter =					scale parameter =				
		171.1622					4.4043				
		-2.0	-1.0	0.0	1.0	2.0					
		.	.	.	.	.	.	.	.	.	.
4.05	162.0000		X								
6.76	164.0000		X								
12.16	166.0000			X							
25.68	168.0000				X						
41.89	170.0000					X					
64.86	172.0000						X				
79.73	174.0000							X			
87.84	176.0000								X		
95.95	178.0000									X	
98.65	180.0000										X
98.65	182.0000										X
		.	.	.	.	.	.	.	.	.	.

chi square test for the normal distribution is chisq = 2.9918
which has 5 degrees of freedom.

17 11 70
sqh. data-survey

examination number 2

page 3

identification for testdata to
program: data survey april 1970

variable number 1 height of soldiers 67/78 in centimeters.

output speification: histogram, fractile normal

group specification: limits=

160.0000	163.0000	166.0000	169.0000	172.0000
175.0000	178.0000	181.0000		

execute mark

number of cases	minimum	maximum
74	160.0000	183.0000

mean	variance	stand.dev.	skewness	kurtosis
171.1622	1939.800 ₁₀ -2	4.404316	-0.046365	3.234582

t-test for mean=0 is t = 334.307
which has 73 degrees of freedom.

95 pct. confidence interval is 170.1587 < mean < 172.1657

histogram: every x represents 1 observation

number of cases	upper class-limit	
1	160.0000	X
2	163.0000	XX
6	166.0000	XXXXXX
15	169.0000	XXXXXXXXXXXXXXXXXX
24	172.0000	XXXXXXXXXXXXXXXXXXXXXXXXXXXX
14	175.0000	XXXXXXXXXXXXXXXXXX
9	178.0000	XXXXXXXXXX
2	181.0000	XX
1		X
total		
74		

17 11 70
sph. data-survey

examination number 2

page 4

identification for testdata to
program: data survey april 1970

variable number 1 height of soldiers 67/78 in centimeters.

fractile diagram in the normal distribution

fraction in pct.	upper class- limit	estimates of						position parameter =		scale parameter =	
		-2.0	-1.0	0.0	1.0	2.0		171.1622		4.4043	
		.	.	.	.	.	.	.	.	.	.
1.35	160.0000	X									
4.05	163.0000		X								
12.16	166.0000			X							
32.43	169.0000				X						
64.86	172.0000					X					
83.78	175.0000						X				
95.95	178.0000									X	
98.65	181.0000										X
		.	.	.	.	.	.	.	.	.	.

chi square test for the normal distribution is chisq = 2.2205
which has 3 degrees of freedom.

The interpretation of this may be:

The height of soldiers is very nicely described by a normal distribution with parameters $(\bar{x}, \sigma^2) = (171.16, (4.40) \times 2)$. The empirical distribution is nearly symmetric (skewness = -0.046), and the chi square distribution test is not significant even at 50 pct. level. It is noticed that the description seems a little better when using grouplimits at $160 + 3Xh$ instead of $160 + 2Xh$, but that might be because the group-length 2 is too small when only having 74 cases.

You might be interested in examining whether a log normal distribution can be used and this can be done by adding $t \ 1, 0.0, 1000 <$ just before end of <input file> (this means that y is substituted by $\ln(y) \times 1000.0$;) If you only want to examine the exponential distribution, you may write $c \ 0, 1 <$ and if you want to examine both distributions, you may write $c1, 1 <$.

Running time for this example was approx.:

Program reading and compilation	27 sec.
Reading, computing, and output to printer	8 sec.
Total	<u>35 sec.</u>

APPENDIX7. PROGRAM MANUSCRIPT IN ALGOL 5.

```
begin comment
  søren henckel. 20 04 70. data-survey (rc 4000-edition)
  an algol 5 translation of data-survey (gier-edition) of 301069;
  message data-survey, version 1, 200470, RCSL 53-S1;

  integer accno,cases,char,control,elem,em,first,last,
    limitnumber,groupnumber,inftyp,margin,maxnumber,
    page,pollex,ps,varnu;

  integer array intens(1:49), sub(1:2), table(0:127);

  real date,lay1,lay2,lay3,lay4,max,min,m1,m2,m3,m4,stdev;

  real array group(1:49), ident(1:19), name(1:8),
    obs(1:3000), output(1:2), trg(1:24);

  boolean cross,groups,head,means,no,ok,space,variab;

  comment declaration of the procedures
    error, expcum, expfrac, fracdiag, grouping, head new page,
    histogram, information, moments, nffrac, outtest, phi,
    pstep, skip, syntax_error, terminators, textline, and trngen;

  procedure error;
  begin
    information(<:error detected in (or after) variable number:>);
    write(out,string lay1,varnu);
    if inftyp=0 then inftyp:= 1;

    textline(2,margin,case inftyp of (
      <:error in art of information:>,
      <:error in subsets:>,
      <:error in number of constants in transgeneration information:>,
      <:error in art of transgeneration:>,
      <:which has too many observations:>,
      <:error in number of grouplimits:>,
      <:identification not terminated by <60>:>));

    if inftyp<4 and inftyp<7 then
      textline(1,margin,<:or some syntactical error:>);
      textline(2,margin,
        <:run on this data set is terminated. copy of input:<10X10>:>);
      table(60):= em+35; char:= ps:= 0; repeatchar(in);
      for ps:= pstep while ps<250 and char<25 do
        begin
          readchar(in,char); write(out,no add char,1)
        end copy max 250 characters or to end_of_medium;
        goto exit_program
      end error;
```

```

real procedure expcum(obs); value obs; real obs;
    expcum:= if obs>0.0 then 1.0-exp(-obs) else 0.0;

integer procedure expfrac(obs); value obs; real obs;
    expfrac:= -ln(1.0-obs)×10.0;

procedure fracdiag(fractile,cum,maxfrac,start,position,scale,text1,text2);
value
    maxfrac,start,position,scale;
integer procedure fractile;
real procedure cum;
real maxfrac,start,position,scale;
string text1,text2;
begin
    integer df,i,j,cumulative,relative;
    real expect,frac,limit,maxcum,mincum,test;

    procedure print_ave;
    begin
        write(out,<:<10>:>,space,23);
        for i:= 10 step -1 until 0 do write(out,<: .:>);
        write(out,<:<10>:>)
    end print_ave;

    head new page;
    textline(3,margin,<:fractile diagram in the :>);
    write(out,text1,<: distribution:>);
    textline(3,margin+30,<: estimates of:>);
    textline(1,margin,
    <:fraction upper class- position parameter =:>);
    write(out,string lay3,position);
    textline(1,margin,<:in pct. limit:>);
    write(out,space,16,<:scale parameter =:>,string lay3,scale,
    <:<10>×<10>:>,space,if start=0.0 then 19 else 24,<< -d.d>,
    start,start+1.0,start+2.0,start+3.0,start+4.0);
    print ave;
    cumulative:= relative:= 0; mincum:= 0.0;
    test:= -accno; df:= -3;

    for i:= 1 step 1 until limitnumber do
    begin
        j:= intens(i); cumulative:= cumulative+j;
        limit:= group(i); frac:= cumulative/accno;
        write(out,<:<10>:>,<< ddd.dd>,
        frac×100.0,string lay3,limit);
        if frac<maxfrac then
        write(out,space,if frac>0.0 then fractile(frac)+1 else 1,<:<88>:>);
        relative:= relative+j;
        limit:= (limit-position)/scale;
        maxcum:= cum(limit);
        expect:= (maxcum-mincum)×accno;
        if expect>5.0 then
        begin
            test:= relative××2/expect+test; df:= df+1;
            relative:= 0; mincum:= maxcum
        end expect>5.0;
        if groupnumber<21 then write(out,<:<10>:>)
    end i-loop;

```

```

print axe;
expect:= (1.0-mincum)×accno;
if expect=0.0 then expect:= 10-6;
relative:= intens(groupnumber)+relative;
if df>-1 then
  outtest(text2,relative××2/expect+test,df+1)
else
  textline(2,margin,<:chi square test omitted because df=0.<10>:>)
end fracdiag;

```

```

procedure grouping;
begin
  integer i,j,poi1,poi2;
  real exobs,group1,group2,length;
  if -, groups then
    begin
      length:= stdev/(if accno<100 then 2.0 else 3.0);
      length:= ln(length)×.434294482;
      poi1:= entier length; length:= length-poi1;
      length:= if length<.0969100 then 1.0 else
        if length<.3979400 then 2.0 else
          if length<.7781513 then 5.0 else 10.0;
      length:= 10.0××poi1×length;
      group1:= group(1):= (entier(min/length)+1.0)×length;
      limitnumber:= entier((max-group1)/length)+1.0;
      limitnumber:= if limitnumber>48 then 48 else
        if limitnumber<2 then 2 else limitnumber;
      for i:= 2 step 1 until limitnumber do
        group(i):= group1:= group1+length;
      groups:= ok
    end if -, groups, determining grouplimits;

    groupnumber:= limitnumber+1;
    poi1:= groupnumber//5;
    poi2:= (if groupnumber mod 3 = 1 then 1 else 0) + poi1 + poi1;
    group1:= group(poi1); group2:= group(poi2);
    group(groupnumber):= max+10.0;
    for i:= groupnumber step -1 until 1 do intens(i):= 0;
    for i:= first step 1 until last do
      begin
        exobs:= obs(i);
        j:= if exobs>group2 then poi2 else
          if exobs>group1 then poi1 else 0;
        for j:= j+1 while group(j)<exobs do;
          intens(j):= intens(j)+1
        end central grouping loop (i-loop)
      end grouping;
    end grouping;

```

```

procedure head_new_page;
begin
  page:= page+1; ps:= 0;
  write(out,<:12>:>,<< -dd dd dd>,date,space,18,
    <:examination number:>,string lay2,poiex,space,10,
    <:page:>,page,<:10>:>,space,margin,
    <:søh. data-survey<10×10>:>,string ident(pstep));

```

```

textline(2,margin,<:variable number:>); ps:= 0;
write(out,string lay1,varnu,string name(pstep),<:<10>:>);
head:= no
end head_new_page;

```

```

procedure histogram;
begin
  integer i,half,maxint,relative;
  if limitnumber+control>26 then head_new_page; maxint:= 0;
  for i:= groupnumber step -1 until 1 do
    if intens(i)>maxint then maxint:= intens(i);
    textline(2,margin,<:histogram: every x represents:>);
    maxint:= (maxint-1)//45+1; half:= maxint//2;
    write(out,string lay1,maxint,<: observation:>,
    if maxint>1 then <:s:> else <::>);
    textline(3,margin,<:number of upper class->);
    textline(1,10,<:cases limit<10><10>:>);
    for i:= 1 step 1 until limitnumber do
      begin
        relative:= intens(i); write(out,string lay4,relative,
        string lay3,group(i),<: :>,
        cross,relative//maxint,
        if relative mod maxint>half then <:x<10>:> else <:<10>:>)
      end i-loop;
    relative:= intens(groupnumber);
    write(out,string lay4,relative,space,15,cross,relative//maxint,
    if relative mod maxint>half then <:x<10>:> else <:<10>:>,
    space,11,<:total<10>:>,accno)
  end histogram;

```

```

procedure information(text); string text;
begin
  control:= control+1;
  if head then head_new_page; textline(1,margin,text)
end information;

```

```

procedure moments;
begin
  integer i;
  real delta1,delta2,exobs;
  m1:= m2:= m3:= m4:= min:= max:= 0.0;
  exobs:= obs(first);
  for i:= first+1 step 1 until last do
    begin
      delta1:= obs(i)-exobs; delta2:= delta1**2;
      m1:= delta1+m1; m2:= delta2+m2;
      m3:= delta2*delta1+m3; m4:= delta2**2+m4;
      if delta1<min then min:= delta1 else
      if delta1>max then max:= delta1
    end central summation-(i)-loop;
    accno:= last-first+1;
    min:= min+exobs; max:= max+exobs;
    m1:= m1/accno; m2:= m2/accno;
    m3:= m3/accno; m4:= m4/accno;
    m4:= -4.0*m1*m3+m1**2*6.0*m2-m1**4*3.0+m4;
    m3:= -m1*m2*3.0+m1**3*2.0+m3;

```

```

m2:= -m1×2+m2;   m1:= m1+exobs;
if m2<=0.0 then
begin
  textline(2,margin,
    <:examination terminated because variance=0.:>);
  goto new
end variance=<0.0;
m3:= m3/sqrt(m2)/m2;   m4:= m4/m2/m2;
m2:= m2×accno/(accno-1);   stdev:= sqrt(m2);   means:= ok
end procedure moments;

integer procedure  nffrac(obs);  value obs;  real obs;
begin
  real p;
  p:= if obs<.5 then obs else 1.0-obs;
  p:= sqrt(ln(p)×(-2.0));
  p:= -( .27061×p+2.30753)/((.04481×p+.99229)×p+1.0)+p;
  nffrac:= (2.5+(if obs<.5 then -p else p))×10.0
end nffrac;

procedure  outtest(text,test,df);  value test,df;
string text;  real test;  integer df;
begin
  textline(2,margin,text);
  write(out,<<-dddd.dd00>,test,<:<10>           which has:>,
    string lay1,df,<:  degree:>,
    if df>1 then <:s:> else <::>,<: of freedom.<10>:>)
end outtest;

real procedure  phi(obs);  value obs;  real obs;
begin
  real p;
  p:= 1.0/(abs obs×.33267+1.0);
  p:= exp(-obs×2/2.0)×((.9372980×p-.1201676)×p+.4361836)×p×.39894;
  phi:= if obs<0.0 then p else 1.0-p
end phi;

integer procedure  pstep;
                pstep:= ps:= ps+1;

procedure  skip;
begin
  integer class;
  repeatchar(in);  for class:= readchar(in,char) while class<8 do
end skip;

boolean procedure  syntax_error(arr);  real array  arr;
begin
  integer i;
  boolean fault;
  fault:= no;
  for i:= elem step -1 until 1 do  fault:= arr(i)>100 or fault;
  syntax_error:= fault
end proc syntax_error;

```

```

procedure terminators(new_class); value new_class; integer new_class;
begin
  integer i;
  for i:= 10,32,44 do table(i):= new_class shift 12 + i
end terminators;

```

```

procedure textline(lines,place,text); value lines,place;
integer lines,place; string text;
  write(out,false add 10,lines,space,place,text);

```

```

procedure trngen;
begin
  integer i,j,kind;
  real    const1,const2;
  infty:= 4;
  for j:= 1 step 3 until elem do
  begin
    kind:= trg(j); const1:= trg(j+1); const2:= trg(j+2);
    if kind<1 or kind>3 then error;
    textline(1,margin,<:y = :>); control:= control+1;
    case kind of
    begin
      begin
        for i:= maxnumber step -1 until 1 do
          obs(i):= ln(obs(i)+const1)xconst2;
          write(out,<:ln:>)
        end case 1;
        for i:= maxnumber step -1 until 1 do
          obs(i):= (obs(i)+const1)xconst2;
        for i:= maxnumber step -1 until 1 do
          obs(i):= (obs(i)+const1)xconst2
        end case;
        write(out,<:(y+(:>,string lay3, const1,
        case kind of (<:))x(:>,<:))x(:>,<:))x(:>),
        const2,<:).:>)
      end j-loop
    end procedure trngen;

```

```

comment date is found by calling systime.
  initializing part for variables;
systime(1,0,m1); date:= systime(2,m1,m2);
lay1:= real<< -d>; lay2:= real<<-dddd>;
lay3:= real<<-ddddddd.d000>;
lay4:= real<< -dddd>;
head:= ok:= variab:= true; groups:= means:= no:= false;
cross:= no add 88; em:= 8 shift 12 + 25; space:= no add 32;
name(1):= real<: :> add 32; name(2):= real<:no na:> add 109;
name(3):= real<:e yet:>;
elem:= first:= last:= maxnumber:= poiex:= 1;
control:= page:= varnu:= 0; margin:= 8;
output(1):= 1.0; output(2):= 0.0; infty:= 7;

```

```

comment choosing alphabet;
for char:= 127 step -1 until 1 do
table(char):= case char of (
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 8, 7, 0, 0, 0, 0, 0, 6, 6, 0, 0, 0, 0, 6, 5, 6,
6, 6, 3, 6, 3, 4, 6, 2, 2, 2, 2, 2, 2, 2, 2, 2, 6, 6, 8,
6, 6, 0, 0, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 0, 0, 0, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 0, 0) shift 12 + char; table(char):= char;
tableindex:= 0; intable(table);

```

```

write(out,<:<15>:>);

```

```

comment start of input of data; readchar(in,char);
skip; readstring(in,ident,elem); skip;
ident(19):= ident(19) shift (-margin) shift margin;
table(60):= em;

```

data:

```

repeatchar(in); readchar(in,char); if char<25 then error;
table(60):= em +35; terminators(0); inftyp:= readchar(in,char);
terminators(7); table(60):= em;
inftyp:= if inftyp<6 then 5 else if char=60 then 1 else
if char=103 then 6 else if char=116 then 3 else
if char=115 then 2 else if char=99 then 7 else
if char=101 or char=25 then 4 else 0;
if inftyp=0 then error;

```

input:

```

case inftyp of
begin
begin
variab:= ok; information(<:execute mark:>); goto execute
end case 1 (execute mark);
begin
elem:= read(in,sub);
if elem<2 or sub(1)<1 or sub(1)>=sub(2) or sub(2)>maxnumber then error;
means:= sub(1)=first and sub(2)=last and means;
first:= sub(1); last:= sub(2);
information(<:subset specification: from case:>);
write(out,string lay1,first,<: to case:>,last)
end case 2 (subsets);
begin
elem:= read(in,trg);
if syntax_error(trg) or elem mod 3 <> 0 or elem<3 then error;
information(<:transgenerations (successive):>);
trngen; groups:= means:= variab:= no
end case 3 (transgenerations);
if variab then goto exit_program
else
begin
information(<:a missing execute mark at end of data is generated:>);
goto execute
end missing execute;

```

```

begin
  if -, variab then
    begin
      information(<:extra examination (not specified by execute mark):>);
      goto execute
    end variables too early;
    repeatchar(in); read(in,varnu,varnu,cases); skip;
    terminators(0); readchar(in,char); repeatchar(in);
    terminators(6); read string(in,name,2); skip;
    name(8):= name(8) shift (-8) shift 8;
    terminators(7); head new page;
    group:= mean:= variab:= 10; m1:= 0.0; ps:= 0;

more:
  table(115):= em; elem:= read(in,group);
  table(115):= em+90; repeatchar(in); readchar(in,char);
  if char=115 then elem:= elem-1;
  for first:= elem step -1 until 1 do m1:= group(first)+m1;
  if ps+elem>5000 then error else
  for first:= 1 step 1 until elem do obs(pstep):= group(first);
  if char=115 then
  begin
    if abs(group(first)-m1)>10-2 then
      write(out,<:<10> checksum error: computed sum =:>,
        string lay3,m1,<: check =:>,group(first));
      means:= ok; m1:= 0.0; control:= control+1
    end checksum;
    if char<25 then goto more
    else
      table(115):= 6 shift 12 + 115;
      elem:= last:= maxnumber:= ps; first:= 1;
      if last<cases then
        write(out,<:<10> cases on tape =:>,
          string lay1,last,<: cases =:>,cases);
        if syntax error(obs) or elem<2 then error;
        information(<:input of observations: total:>);
        write(out,string lay1,last,<: cases:>,
          if means then <: with:> else <: without:>,
          <: checksum control:>); means:= no; control:= control+1
      end case 5 (variables);
      begin
        elem:= read(in,group);
        if syntax error(group) or elem>48 or elem<2 then error;
        limitnumber:= elem; groups:= ok;
        for elem:= elem-1 step -1 until 1 do
          groups:= group(elem+1)>group(elem) and groups;
        variab:= -, groups and variab;
        information(<:group specification: limits=:>);
        for elem:= 1 step 1 until limitnumber do
          write(out,if (elem - 1) mod 5 = 0 then <:<10> :> else <:,>,
            string lay3,group(elem)); control:= (limitnumber+4)//5+control;
          if -, groups then
            textline(1,margin,<:but these limits are rejected:>)
      end case 6 (given grouplimits);

```

```

begin
  read(in,output); skip;
  information(<:output speification: histogram:>);
  write(out,if output(1)>0.0 then <:, fractile normal:> else <::>,
        if output(2)>0.0 then <:, fractile exponential:> else <::>)
  end case 7 (output specification)
end case;
goto data;

execute:
  if first>1 or last<maxnumber then
  begin
    textline(2,margin,<:first case =:>);
    write(out,string lay1,first,<: and last case =:>,last)
  end printing of subset;

  if -, means then moments;

  textline(3, 15,<:number of cases      minimum      maximum<10>:>);
  write(out,space,19,string lay2,accno,space,9,string lay3,min,max);
  textline(3,13,
    <:mean      variance      stand.dev.      skewness      kurtosis<10>:>);
  write(out,space,6,string lay3,m1,<< -dddd.ddddn-d>,m2,
    <<-dddddd.d00000>,stdev,m3,m4,<:<10>:>);
  outtest(<:t-test for mean=0 is t =:>,m1/stdev*sqrt(accno),accno-1);
  m2:= stdev/sqrt(accno)*1.96;
  comment m2 is here used as a temporary result;
  textline(2,margin,<:95 pct. confidence interval is :>);
  write(out,string lay3,m1-m2,<: < mean <60>:>,m1+m2,<:<10>:>);
  m2:= stdev**2; comment now m2 again denotes the variance;

      grouping;      histogram;

  if output(1)>0.0 then
    fracdiag(nffrac,phi,.9969,-2.0,m1,stdev,<:normal:>,
    <:chi square test for the normal distribution is chisq =:>);

  if output(2)>0.0 then
    fracdiag(expfrac,expcum,.9947,0.0,0.0,m1,<:exponential:>,
    <:chi square test for the exponential distribution is chisq =:>);

new:
  poiex:= poiex+1; control:= 0; head:= ok;
  if variab then goto data;
  variab:= ok;
  goto input;
exit_program:
  write(out,<:<12>:>)
end program

```

recordinput, Part 1

CONTENTS:

	Page
1. Function and purpose	2
2.1. Input/output files	4
2.2. Input syntax	5
2.3. Format of output file	6
2.4. Format and explanation of error messages	7
3. Program tape, storage requirements and run time	8
4.1. Method	9
4.2. Program structure	10
5. References	12
6.1. User's example, program	13
6.2. User's example, input tape	14
6.3. User's example, output	14

APPENDIX:

7.1. Possible alterations	17
7.2. Program manuscript in ALGOL 5	17

1. FUNCTION AND PURPOSE.

The procedure recordinput attempts to solve the problems concerning input of control information to the ROST statistical package and has been made in order to fulfil the following general rules:

- 1: It should be possible to divide any input paper tape into a number of logical records.
- 2: You may delete any record containing anything.
- 3: In every possible error situation it should be possible to give a correct picture (list) of the erroneous record.
- 4: All not deleted records has to appear in the output file generated by the procedure, whereas all erroneous or deleted records has to be copied on current output mixed with error messages.
- 5: The procedure should only check for syntax errors, whereas semantic errors has to be detected in a later scan.
- 6: The entire data paper tape has to be read by calling the procedure once.

Procedure heading:

```
external boolean procedure
recordinput(maxchar,maxparam,realtext,realname,descriptor,tailcontent);
value      maxchar,maxparam,realtext,realname;
integer    maxchar,maxparam,realtext,realname;
real array                                descriptor;
integer array                                tailcontent;
```

Call parameters:

integer maxchar = maximum number of non blind characters in one input
record
(maxchar must be ≥ 5)

integer maxparam = maximum number of parameters in an output record
(maxparam must be ≥ 1)

integer realtext = maximum number of reals used for storing a text
parameter

integer realname = maximum number of reals used for storing a name
parameter
(realtext and realname must be ≥ 0)

real array descriptor(1:number_of_descriptors)
each array element must at call contain one descrip-
tor as a short string (e.g. either the entire de-
scriptor or, for long descriptors, exactly the 5
first characters and a null character).

integer array tailcontent(-1:number_of_descriptors)
the array elements numbered 1 until number_of_de-
scriptors must at call contain an integer number
code ($1 \leq \text{code} \leq 5$) showing what the tail of the
corresponding kind of record is intended to con-
tain. Explanation of the code is:

code = 1 $\Leftrightarrow$ a number of text parameters (at least one)

code = 2 $\Leftrightarrow$ exactly one non empty name parameter

code = 3 $\Leftrightarrow$ a number of number parameters
(mixed reals and integers)

code = 4 $\Leftrightarrow$ empty (e.g. only separators)

code = 5 $\Leftrightarrow$ end of input tape (a possible tail is ignored).

integer array elements

tailcontent(-1) = number of (different) descriptors

tailcontent(0) = number of segments reserved on backing storage for
output of records (must be ≥ 0)

Return parameters:

boolean recordinput true if no troubles with the output file has occurred, otherwise false.

If the procedure return value is false, none of the call parameters have been touched, otherwise we have the following return parameters:

real array descriptor(1:2) name (generated by the monitor) on the backing storage file used for output of records.

integer array elements

tailcontent(0) = number of accepted records in the <input file>
tailcontent(1) = total number of records read by the procedure

The rest of the call parameters are unaltered.

Check of call parameters:

The procedure checks that the call parameters are reasonable but a complete check is not performed.

2.1. INPUT/OUTPUT FILES.

In order to facilitate the use of procedure recordinput, the standard zones in/out (connected to current input/output) has been selected as input file for data on character level and output file (on character level) for error messages, whereas the records accepted or generated by the procedure are placed on a backing storage file with a name generated by the monitor (and delivered as return parameter).

The format of error messages is A4 vertical with approx. 60 lines on each page with each line containing maximally 71 characters starting in position 1.

Because the usually used printers only have one kind of letters and the File Processor requires small letters and in order to minimize the number of error situations, all letters in the <input file> are transformed to small letters.

The notion <record> may need further explanation: In order to have input partitioned in logical parts and to facilitate the preparation of control information, it is comfortable to let the kind of information be given in front of the information, and among many possibilities, a verbal description has been preferred. This means that every record begins with a text showing what the rest of the record is about. An example may be:

ident: output identification<

where ident shows, that the rest of the record (to the character <) has to be used as an identification text in output.

2.2. INPUT SYNTAX.

```

<input file> ::= <record>*1
<record> ::= <accept record> | <delete record> | <EM record>
<accept record> ::= <descriptor> <parameter>*0 <accept>
<delete record> ::= <any string without <terminator> > <delete>
<EM record> ::= <any string without <terminator> > <EM>
<terminator> ::= <accept> | <delete> | <EM>
<accept> ::= <separator>*0 < <between records>
<delete> ::= <separator>*0 > <between records>
<EM> ::= <end of medium character>
<between records> ::= <<separator> | < | >>*0
<separator> ::= , | ; | : | SP | NL | HT | VT | FF | <separator> <separator>
<descriptor> ::= <name>
<parameter> ::= <number parameter> | <text parameter> | <name parameter>
<number parameter> ::= <separator> <number>
<number> ::= <leading part>*0 <decimal part>*0 <exponential part>*0
<leading part> ::= <sign>*0 <digit>*0
<decimal part> ::= <decimal point> <digit>*1
<exponential part> ::= <exponent mark> <sign>*0 <digit>*1
<name parameter> ::= <separator> <name>

```

```

<name> ::= <letter><letter>|<digit>>*
<text parameter> ::= <text start><text>
<text start> ::= : <separator without <:>>*
<text> ::= <legal character without <separator> <:> and <terminator>>
          <legal character without <:> and <terminator>>*|empty
<legal character> ::= <character in the normal ISO set which is available
                      on printer without <question mark>>

```

Further the part of an <accept record> which is not the <descriptor> is in the following denoted as the tail.

2.3. FORMAT OF OUTPUT FILE

The output file will contain a number of records consisting of real elements in the basic format:

If the descriptor is of kind number k and the number of parameters in the record is p, the format of the output record is:

element index	content
1	p+2 as real (converted by assignment)
2	k as real (- - -)
3	parameter(1)
4	parameter(2)
	.
	.
	.
p+2	parameter(p).

The procedure adds three new kinds of descriptors to the kinds given in the call:

- 1: kind of descriptor = number of descriptors + 1
is used for indicating errors in records where the determination of the kind of descriptor has been impossible. These records have parameter number:= 0;

- 2: kind of descriptor = number of descriptors + 2
is used for indicating errors in the tail of a record with
known kind of descriptor.
These records have parameter number := 1; and
parameter(1) := found kind of descriptor;
- 3: kind of descriptor = number of descriptors + 3
indicates an end of data record (code in tailcontent = 5) or an
<EM record>. This record (exactly one from each successful
call of the procedure) has parameter number := 0;

The format of output records containing a number of texts (code in
tailcontent array = 1) needs a special explanation:

These records contain 2 real elements showing (as usual) the total
recordlength and the kind of record. Further they contain a number of
subrecords (exactly one subrecord for each textparameter in the input
record).

If the textparameter occupies 1 reals, the length of the subrecord
is 1 + 1 real elements. In the first of these elements the textlength
(=1) is placed as a real, and the rest of the subrecord contains the
textparameter packed with 6 characters in each real element (and a null
character placed after the last textcharacter).

The records are outrec'ed in such a way, that every zone buffer con-
tains an entire number of records.

2.4. FORMAT AND EXPLANATION OF ERROR MESSAGES.

If a record is not deleted (by >), the procedure first tries to de-
termine the kind of descriptor (e.g. kind of record). If this succeeds,
the syntactical check is continued according to general rules and the
number code in the corresponding element of the call parameter tailcon-
tent.

If an error is detected (or if the record has been deleted) the
procedure will produce an error message supplied with a character pic-
ture of the erroneous (or deleted) record.

The error message format is:

record number<<-d> <error message>
copy of record:<character list of record>

In the <character list of record> the total record will be printed with these modifications:

- (a) the characters NL, FF, HT and VT will in list be replaced by SP.
- (b) the list is made with 71 maximally characters on each line.

The records in <input file> are counted 1,2,3 and so on, and the error message output is partitioned in pages numbered 1,2,3 and so on (each page will contain approx. 60 lines = format A4 vertical, with output starting in printing position number 1).

The 30 different error messages are made in such a way, that I mean they need no further explanation. An example of an error message can be:

28 10 70 record input syntax errors in data page 3

record number 34 has some syntax error in a number parameter
copy of record: model 73 75-<

3. PROGRAM TAPE, STORAGE REQUIREMENTS AND RUNNING TIME.

The procedure recordinput is written in ALGOL 5 as an external boolean procedure (see ref. 1), and is available as algol text on 8 channel paper tape both in the normal ISO form (with even parity) and in flexo-writer mode (with odd parity); these tapes must be loaded by <i tre> and <i trf> respectively.

All letters in the program text are small letters because the normally used printers only have one kind of letters.

The structure on RC 4000 does not permit giving exactly times for compilation and running time, but some general rules can be mentioned:

The compiled external procedure occupies 19 segments on backing storage, and reading and compilation takes approx. 25 seconds.

Compilation and running may be done in a normal 10 k bytes process, but it will be much more comfortable to run the program in a 15 k bytes process. With $\text{maxparam} < 126$ and $\text{maxchar} < 1000$ it is possible to run the program in a 10 k bytes process whereas $\text{maxparam} = 250$ and $\text{maxchar} = 2500$ requires approx. 15 k bytes. These sizes are minimal, and it is possible to cut down running time with approx. 50 pct. by using respectively 15 k and 20 k bytes.

Normally the control information is not very large and therefore the speed of the procedure is not very critical, but as a general rule you may say, that the procedure reads 1 page of control information in about 5-15 seconds (real time) dependent on the number of erroneous records (which has to be copied in the error list).

4.1. METHOD.

The general rule about character list of erroneous records in any situation (even in case of illegal numbers) combined with the claim on the possibility of deletion at any time and the claim on unique partitioning of any input tape in logical records are the reasons why input is performed on character level by means of the integer procedure `class_of_input`. This procedure reads one character from zone in and places it (if possible) in the boolean array `character(1:maxchar+1)` for later inspection. Further this procedure checks the basic syntax (illegal characters, EM character and long record).

If the record is an <accept record> the kind of descriptor is determined according to the following rule: if the character string expected to be the <descriptor> contains more than 5 characters, only the first 5 characters are used in textmatching for determining the kind of descriptor, otherwise the total descriptor character string is used (e.g. the three strings <variables>, <variavzig> and <variab> are all equivalent with the given descriptor string <varia>).

If the record is terminated by <delete> the procedure gives a deletion message and a character picture of the deleted record.

If the record is not deleted and the determination of the kind of descriptor has succeeded, the conversion of the tail is performed by scanning the characters stored in the boolean array `character(tailstart:position)`.

The number parameters are converted in four parts: <leading part>, <decimal part>, <exponential part>, and <sign> which later are combined by the statement

number := (leading part + decimal part) × exponential part × sign;

This way of conversion causes the following conditions to be true

$0 \leq \text{abs}(\text{leading part}) \leq 8388599$,

$0 \leq \text{decimal part} \times 10^{\text{number of decimals}} \leq 8388599$ and

$0 \leq \text{abs}(\text{exponent}) \leq 599$.

Numbers with syntax as +'-3 are converted correctly whereas 0.0'1, 0.0'1 and 0'1 all gives wrong conversion to 10. All other ALGOL 60 numbers are correctly converted.

The name- and text parameters are packed with 6 characters in each real element. After the last character in each parameter (possibly in the next real element) is placed a null character in order to facilitate the later use of these parameters.

4.2. PROGRAM STRUCTURE.

The procedure has been programmed in such a way that alterations are as easy as possible to make. All variables (except i, j and text) have names which shows their use.

The two main procedures are

1: integer procedure class_of_input

2: integer procedure unpack_character.

which are used for character input and conversion respectively.

The procedure error (error_type) is large and rather slow because of the 30 long error texts combined with the character list of the erroneous record, and the decision of the further error reactions such as creating parameters for possible error records.

The entire procedure body of procedure recordinput consists of one block containing several local blocks each one containing declarations of very local quantities.

After initializing and finding (with check) the date of run, the call parameters are checked in order to avoid errors not covered by the control in running system (without index check). Next a backing storage file, with name generated by the monitor (and delivered as return parameter) and size as given in the call parameter tailcontent(-1), is created (if possible).

Initializing of the alphabet stored in the integer array table(0:127) is done according to the following scheme:

input class	contents
0	DEL, NULL
1	empty
2	digits (0, 1, ... , 9)
3	sign (+ -)
4	decimal point (.)
5	exponent mark ('')
6	small and capital letters (a,..., $\overset{\circ}{a}$, A,..., $\overset{\circ}{A}$)
7	empty
8	graphics(= / exclamation mark 'and')(× _ and '')
9	illegal characters (the rest) shown as <question mark>
10	terminators (< > and EM)
11	normal separators (; , SP NL FF HT and VT)
12	text start (:)

The active part of the procedure is from label action: in line 220 to label finish_input: in line 585. This part is formed as a case statement controlled by the integer variable state. Each case refer to a state within reading or conversion of one record.

Case 1 and 2 is reading (and packing) descriptor and tail respectively.

Case 3, 4, and 5 corresponds to accept-, delete-, and end- (of data) record respectively,

Case 6 is conversion of the tail and contains 5 subcases corresponding to the 5 possible different number codes in the call parameter tailcontent. The third subcase (conversion of numbers) is a case statement containing 8 cases (controlled by the integer variable

state, which here means state within one number parameter).

Case 7 is outrec of the syntactically correct and converted record. The records are made in such a way that each zone buffer contains an entire number of complete records.

Case 8 is used when a record contains more than maxchar non blind characters. The list is continued until a <terminator> is met and the further action is decided.

Case 9 is used in case of error in call parameters, problems with create entry or by call of the block procedure error_in_doc.

The total action scheme is:

state	action	new state
1	read descriptor	2,3,4,5,8,9
2	read tail	3,4,5,8,9
3	determine descriptor in accept record	6,7
4	list delete record	1
5	end (of data) action	9, -
6	convert the tail	7,5
7	outrec of (correct) record	1, 9
8	list long record	4,5,7
9	unspecified problems	-

5. REFERENCES.

- (1) Søren Lauesen: ALGOL 5. User's manual
A/S Regnecentralen Copenhagen. July 1969.
- (2) Søren Lauesen: File Processor. User's manual
A/S Regnecentralen Copenhagen. April 1968
- (3) Per Brinch Hansen: Multiprogramming System
A/S Regnecentralen Copenhagen. April 1968.

6.1. USER'S EXAMPLE, PROGRAM.

The following program has been used for control of procedure re-
cordinput:

test program for recordinput of 28 10 70

```

begin
  zone z(256,2,stderr); integer array t(-1:9);
  real array d(1:12); integer i,j,k,m;
  message s0h test record input of 28 10 70;
  for i:= 9 step -1 until 1 do
  begin
    t(i):= case i of (5,3,2,3,3,1,3,1,4);
    d(i):= real ( case i of ( <:end:>,<:list:>,<:data:>,
      <:output:>,<:model:>,<:comme:>,<:varia:>,<:head:>,<:execu:>))
  end;
  t(-1):=15; t(0):=9; d(10):=real <:unknown kind:>;
  d(11):= real <:error in known kind:>; d(12):=d(1);
  if recordinput(250,35,12,2,d,t) then
  begin
    i:= j:= 1; open(z,4,string d(increase(i)),0);
    write(out,<:<12> list from area: :>,
      string d(increase(j))); m:= 0; d(2):= real <:list:>;
  new:
    i:= inrec(z,2); j:= z(1)-2; if j<0 then
    begin
      write(out,
        <:<10><10> segment change. rest=:>,<<-d>,i+2);
      inrec(z,i); goto new
    end;
    i:= z(2); inrec(z,j); m:= m+1;
    write(out,<:<10><10>record number=:>,<<-d>,m,
      <: of kind number=:>,i,<: = :>,string d(i),<:<10>:>);

    case case i of (0,3,4,3,3,2,3,2,1,1,5,6) of
    begin
      write(out,<:empty record. length =:>,<<-d>,j);
      begin
        write(out,<:text record. length =:>,<<-d>,j); k:= 1;
        for l:= k+1 while k<=j do
          write(out,<:<10>length =:>,<<dd>,z(k-1),<:::>,
            string z(increase(k)),<:::>)
        end;
      begin
        write(out,<:number record. length =:>,<<-d>,j);
        for i:= 1 step 1 until j do
          write(out,if i mod 6=1 then <:<10>:> else <:::>,
            <<-dddd.dd<sub>10</sub>-dd>,z(i))
        end;
      begin
        i:= 1; write(out,<:name record. length =:>,<<-d>,j,<:::>,
          string z(increase(i)),<:::>)
        end;
    end;
  end;

```

```

        write(out,<:found kind =:>,<<-d>,z(1),<: = :>,string d(z(1)));
    begin
        write(out,<:end of control file<12>:>); goto finish
    end
end case;
goto new;
finish:
    end if record input;
    close(z,true)
end program

```

6.2. USER'S EXAMPLE, INPUT TAPE.

The test run has been performed using this <input file>, punched on 8 channel paper tape in flexowriter mode:

```

Test data for recordinput of 28 10 70>
data testfile70<
head : 1970 experiments (regression of height/age):
;text number 2<
Variables 7, 4a<ouTput 1 -7 10-4,,1.03612<
variab ,7,, 4,,,;<
head:,a;B:
C::E:<
execution<
erroneous kind of descriptor (erroneous is illegal)<
10. record shows deletion of hard errors>
execu-tion<
Head experiment number 2< variables 86043780<
head : 080446/0519 = number code for output identification <
variables<
execu:<
varia 3,7. 4< data:version3<variables 3 7.0 4<
end of testdata for recordinput containing 20 records<

```

6.3. USER'S EXAMPLE, OUTPUT.

From this we obtain the following of output:

```

17 11 70      record input      syntax errors in data      page 1

```

record number 1 has been deleted in input (by >)
copy of record: test data for recordinput of 28 10 70>

record number 4 has some syntax error in a number parameter
copy of record: variables 7, 4a<

- 15 -

record number 9 has an illegal kind of descriptor
copy of record: erroneous kind of descriptor (erroneous is illegal)<

record number 10 has been deleted in input (by >)
copy of record: 10. record shows deletion of hard errors>

record number 11 contains an illegal character in the descriptor
copy of record: execu-tion<

record number 12 has no text start (:) before the first text
copy of record: head experiment number 2<

record number 13 has overflow in leading part of a number parameter
copy of record: variables 86043780<

record number 17 has empty digit part in the decimal part of a number
copy of record: varia 3,7. 4<

record number 20 is an end of input record. input finished.

survey from record_input:

total number of records in input was 20
and of these were 12 accepted.

list from area: wrk000015

record number 1 of kind number 3 = data
name record. length = 2:testfile70:

record number 2 of kind number 8 = head
text record. length = 13
length = 8:1970 experiments (regression of height/age):
length = 3:text number 2:

record number 3 of kind number 11 = error in known kind
found kind = 7 = varia

record number 4 of kind number 4 = output
number record. length = 4

1000.00₁₀ -3 -7000.00₁₀ -3 1000.00₁₀ -7 1036.00₁₀ 9

record number 5 of kind number 7 = varia
number record. length = 2
7000.00₁₀ -3 4000.00₁₀ -3

record number 6 of kind number 8 = head
text record. length = 12
length = 1:a:
length = 1:b:
length = 1:c:
length = 1::
length = 1:e:
length = 1::

record number 7 of kind number 9 = execu
empty record. length = 0

record number 8 of kind number 10 = unknown kind
empty record. length = 0

record number 9 of kind number 10 = unknown kind
empty record. length = 0

record number 10 of kind number 11 = error in known kind
found kind = 8 = head

record number 11 of kind number 11 = error in known kind
found kind = 7 = varia

record number 12 of kind number 8 = head
text record. length = 12
length = 9:080446/0519 = number code for output identification :

record number 13 of kind number 7 = varia
number record. length = 0

record number 14 of kind number 9 = execu
empty recrd. length = 0

record number 15 of kind number 11 = error in known kind
found kind = 7 = varia

record number 16 of kind number 3 = data
name record. length = 2:versionj:

record number 17 of kind number 7 = varia
number record. length = 3
3000.00₁₀ -3 7000.00₁₀ -3 4000.00₁₀ -3

record number 18 of kind number 12 = end
end of control file

APPENDIX.7.1. POSSIBLE ALTERINGS.

It might be wanted to introduce different action on end of data record (code = 5) and <EM record> in order to read several input tapes in one call of recordinput (else the output records from the different input paper tapes (separated by an end of medium character) will be placed on different backing storage files). It might also be wanted to introduce new kinds of number codes (new kinds of tail conversion). This can be done by adding new subcases in case 6 (tail conversion case) and by altering a little in procedure error for possible new types of errors.

7.2. PROGRAM MANUSCRIPT IN ALGOL 5.

```
(; procedure record_input has to be loaded by < i tre >
clear recordinput
recordinput:=set 19
recordinput:=algol index.no message.yes
end ;
)
external
boolean procedure
  record_input(maxchar,maxparam,realtext,realname,descriptor,tailcontent);
  value      maxchar,maxparam,realtext,realname;
  integer     maxchar,maxparam,realtext,realname;
  real array                                     descriptor;
  integer array                                tailcontent;
begin
  comment
  procedure record_input made october 1970
  on a/s regnecentralen københavn by søren henckel.

  call parameters:

  maxchar      = maximal number of characters in an input record>=0
  maxparam     = maximal number of parameters in an output record>=1
  realtext     = maximal number of reals used for one text>=0
  realname     = maximal number of reals used for one name>=0
  descriptor   = a real array declared descriptor(a:b) with
                  a<=1<=2<=b and number of descriptors<=b. each element
                  must contain one descriptor as a short string
                  (for long descriptors exactly the 5 first characters).
  tailcontent  = an integer array declared tailcontent(c:d) with
                  c<=-1<=1<=d and number of descriptors<=d. each
                  element must contain one of the numbers 1,2,3,4, and 5
                  showing what records of the corresponding kind is
                  supposed to contain as parameters
                  (see in head of case 6 for further details).
  tailcontent(-1):=number of segments for record output>=1
  tailcontent(0):= number of different descriptors.
```

return parameters:

if return value of the procedure is false, none of the parameters are altered, otherwise they contain the following quantities:

descriptor(1:2):= document name on the backing storage used
for record output (name generated by the monitor)
tailcontent(-1):= number of segments for record output (:=call value)
tailcontent(0):= number of accepted records
tailcontent(1):= number of records on input tape

3 new kinds of descriptors are added to the kinds introduced by the call:

kind of descriptor=number of descriptors+1
is used for indicating errors in records with unknown kind of descriptor, and these records have parameternumber:= 0,

kind of descriptor=number of descriptors+2
is used for indicating errors in the tail of a record with known kind of descriptor. these records have parameternumber:= 1, and parameter(1):= found kind_of_descriptor,

kind of descriptor=number of descriptors+3
is indicating an end of medium record or an end of data record, and has parameternumber:= 0.
;

message record_input, version 1, 28.10.70. RCSL 53-S7;

integer array table(0:127);
real array parameter(1:max_param);
boolean array character(1:max_char+1);
integer i,j,char,class,date,descriptor length,
descriptor start,kind of descriptor,lines written,
number of descriptors,ok_record,page,parameter_number,
position,record,state,tail_start;
real blank,text;
boolean error_in_name,illegal_char;
zone output_zone((max_param+129)//128*256,2,error_in_doc);

comment
declaration of the procedures
class of input, error in doc, error,
error_head_new_page, and unpack_character;

integer procedure class_of_input;
begin
class_of_input:= read_char(in,char);
if char<25 then
begin
position:= position+1;
character(position):= false add char;

```

        if char=63 then illegal_char:= true;
        if position>max char then error(17)
    end not_end_of_medium
    else
        error(22)
    end procedure class_of_input;

procedure error_in_doc (connected_zone,status,bytes);
zone                      connected_zone;
integer                   status,bytes;
begin
    boolean not first;
    not first:=false;
    if lines_written>54 then error_head_new_page;
    write(out,<:<10<10<10>:>,
    <:problems with the backing storage on :>,
    <<d>,tail_content(-1),<: segments<10>:>,
    <:which is used for record output.<10>status = :>);
    for i:= 23 step -1 until 1 do
        if false add (status shift (-i)) then
            begin
                write(out,if not first then <: + 1 shift :> else
                    <:1 shift :>,<<d>,i);
                not first:= true
            end printing bit 0 to 22;
            if false add status then write(out,<: + 1:>);
            write(out,<:<10>bytes transfered = :>,<<d>,bytes); error(24)
        end procedure error_in_doc;

```

```

procedure error (error_type);
value          error_type;
integer        error_type;
begin
    if (position-15)//71+lines_written>54 then error_head_new_page;
    comment error message cannot be printed on this page;
    lines_written:= lines_written+5;
    write(out,<:<10<10<10>record number:>,<<-d>,record,
    case error_type of (
        <: contains illegal characters in the tail (shown as <63>):>,
        <: has no text start (:) before the first text:>,
        <: cannot be output because of too many parameters:>,
        <: contains a name parameter not starting with a small letter:>,
        <: contains an illegal character in a name parameter:>,
        <: has overflow in leading part of a number parameter:>,
        <: has overflow in decimal part of a number parameter:>,
        <: has empty digit part in the decimal part of a number:>,
        <: has overflow in exponential part of a number:>,
        <: has an illegal sign in exponential part of a number:>,
        <: has empty digit part in the exponential part of a number:>,
        <: has illegal termination after an exponential part:>,
        <: has some syntax error in a number parameter:>,
        <: has not empty parameter part (it must be empty):>,
        <: contains an illegal character in the descriptor:>,
        <: has an illegal kind of descriptor:>,
        <: cannot be read because of typographical length:>,
        <: has been deleted in input (by >):>,

```

```

<: contains a long text which is cut to max characters:>,
<: contains a long name which is cut to max characters:>,
<: is an end of input record.  input finished.:>,
<: is an end of medium record.  input finished.:>,
<: has shown errors in call parameters (dimension or content):>,
<: caused the problems shown above.<10>run terminated.:>,
<: gave monitor trouble<10>catalog func. forbidden in call process:>,
<: gave monitor trouble<10>catalog input/output error:>,
<: gave monitor trouble<10>entry with same name already exists:>,
<: gave monitor trouble<10>the catalog is full:>,
<: gave monitor trouble<10>requested area size is not available:>,
<: gave monitor trouble<10>name format is illegal:>);
if position>0 then <:<10>copy of record:> else <::>);
j:= 15;  comment 15 characters written on the last line;
for i:= 1 step 1 until position do
begin
  write(out,character(i),1);
  j:= j+1;
  if j=71 then
  begin
    write(out,<:<10>:>);  j:= 0;
    lines_written:= lines_written+1
  end new_line in list
end list on character level on current output;
if error_type<17 then ok_record:= ok_record-1;
comment 'an error record is made, but this is not an ok_record;
if error_type<15 then
begin
  parameter_number:= 1;
  parameter(1):= kind of descriptor;
  kind of descriptor:= number of descriptors+2
end error in tail of a record with known kind of descriptor
else
if error_type<18 then
  kind of descriptor:= number of descriptors+1;
state:=if error_type<17 then 7 else
  if error_type<23 then
    (case error_type-16 of (8,1,7,7,5,5)) else 9;
goto action
end procedure error;

procedure error_head_new_page;
begin
  lines_written:= 1;  page:= page+1;
  write(out,<:<12>:>,<<dd dd dd>,date,
  <: record input syntax errors in data page:>,
  <<-d>,page,<:<10>:>)
end procedure error_head_new_page;

integer procedure unpack_character;
begin
  tail_start:= tail_start+1;
  char:= character(tail_start) extract 8;
  unpack_character:= table(char) shift (-12) extract 12
end procedure unpack_character;

```

```
write(out,<:<15>:>); state:= 1; lines_written:= 200;
ok_record:= page:= position:= record:= 0;
number_of_descriptors:= tail_content(0);
blank:= blank shift 48; record_input:= false;
```

```
comment date is found by calling systime. if the date
in the monitor is wrong, date is set to 28 10 70;
systime(1,0,text); date:= systime(2,text,text);
j:= date mod 100; comment j:= year;
if j<70 then date:= 281070;
```

```
if system(3,i,descriptor)>1 or i<number_of_descriptors or i<2
or system(3,i,tailcontent)>-1 or i<number_of_descriptors
or tail_content(-1)<1 or real_text<0 or real_name<0
then error(23);
comment check of dimension and content of call parameters;
```

```
comment creating area for record output by calling procedure monitor;
table(0):= tail_content(-1);
open (outputzone,4,<:>,0);
i:= monitor(40,outputzone,0,table);
if i > 0 then error(24+i); comment troubles with create entry;
```

```
comment initializing of table(0:127);
for i:= 31 step -1 until 1 do table(i):= 9 shift 12 + 63;
for i:= 15 step -1 until 1 do table(i+32):=
case i of (8,8,0,0,8,8,5,8,8,8,3,11,3,4,8) shift 12 + i + 32;
for i:= 57 step -1 until 48 do table(i):= 2 shift 12 + i;
for i:= 125 step -1 until 97 do
table(i):= table(i-32):= 6 shift 12 + i;
table(35):= table(36):= table(63):= table(64):=
table(94):= table(96):= table(126):= 9 shift 12 + 63;
table(9) := table(10):= table(11):=
table(12):= table(32):= 11 shift 12 + 32;
table(25):= 10 shift 12 + 25;
table(58):= 12 shift 12 + 58; table(59):= 11 shift 12 + 59;
table(60):= 10 shift 12 + 60; table(61):= 8 shift 12 + 61;
table(62):= 10 shift 12 + 62; table(95):= 8 shift 12 + 95;
table(0):= table(127):= table_index:= 0;
intable(table);
```

```
comment of typographical reasons
the program text is moved 3 positions to the left;
```

action:

case state of

begin

begin

```
comment case 1 initializing before new record.
reading and first check of the descriptor;
```

```
record:= record+1;
kind_of_descriptor:= parameter_number:= position:= 0;
illegal_char:= false;
for class:= class_of_input while class>9 do;
```

```

comment skip of leading separators and terminators before descriptor;
descriptor_start:= position;
error in name:= class<6;
for class:= class of input while class<10 do
if class<2 and class<6 then error in name:= true;
descriptor_length:= position-descriptor_start;
tail_start:= position;
state:= if class>10 then 2 else if char=60 then 3 else 4
end case 1 (reading and checking descriptor);

begin
comment case 2 reading the tail (which is not empty);

for class:= class of input while class<10 do;
comment reading characters in the tail;
state:= if char=60 then 3 else 4
end case 2 (reading tail);

begin
comment case 3 accept record e.g. record terminated by < .
check for hard errors and errors in kind of descriptor;

if descriptor_length>5 then descriptor_length:= 5;
j:= descriptor_start+descriptor_length-1;
comment the characters in descriptor are numbered
descriptor_start,descriptor_start+1,descriptor_start+2, and so on,
so j denotes the number of the last character in the checked
part of the descriptor. for long descriptors only the 5 first
characters are checked;
for i:= descriptor_start step 1 until j do
text:= text shift 8 add (character(i) extract 8);
text:= text shift ((6-descriptor_length)*8);
comment now text contains the part of the descriptor
which is used for determining kind of descriptor;
if -,error in name then
for i:= number of descriptors step -1 until 1 do
if descriptor(i)=text then
begin
kind of descriptor:= i;
i:= 1
end determining kind of descriptor;
if kind of descriptor=0 or error in name or illegal char
then error(if error in name then 15 else
if kind of descriptor=0 then 16 else 1);
comment hard errors in record;
state:= 6
end case 3 (accept record);

error(18); comment case 4 (delete record);

begin
comment case 5 end of medium record or end of input record;

i:= outrec(outputzone,0);
if i<2 then

```

```
begin
  outrec(outputzone,i);
  outputzone(1):= 0.0
end active segment change;
outrec(outputzone,2);
outputzone(1):= 2.0;
outputzone(2):= number of descriptors+3;
ok_record:= ok_record+1;
close(outputzone,false);
getzone(outputzone,table);
descriptor(1):= blank add table(1) shift 24 add table(2);
descriptor(2):= blank add table(3) shift 24 add table(4);
comment storing the generated name in descriptor(1:2);
tail_content(0):= ok_record;
tail_content(1):= record;
record_input:= true;
comment assigning all return parameters;

if lines written>54 then error head new page;
write(out,<:<10x10x10>survey from record<95>input:<10x10>:>,
<:total number of records in input was :>,<<d>,record,
<:<10>and of these were :>,ok_record,<: accepted.:>);
comment printing survey on current output;
goto finish input
end case 5 (end_of_medium record or end_of_input record);
```

```
begin
  comment case 6 conversion of the tail in all cases.
  the following case statement corresponds to the kind of tail
  e.g. what the tail is intended to contain
  (this is by call stored in tail_content(1:number of descriptors)
  kind of tail=1 a number of texts (at least one)
  kind of tail=2 exactly one (non empty) name
  kind of tail=3 a number of numbers (mixed integers and reals)
  kind of tail=4 must be empty (e.g. only separators are allowed)
  kind of tail=5 end of input record, a possible tail is ignored;

  tail_start:= tail_start-1;
  comment for conversion of the tail it is comfortable to let
  tail_start denote the character just before the first character
  in the tail (because of for-while statements);

  case tail_content(kind_of_descriptor) of
  begin
    comment kind of tail=1 e.g. tail is a number of texts;

    integer text_start,words_in_text;
    boolean long_texts;
    long_texts:= false;
    for class:= unpack character while class=11 do;
    comment skipping separators (not :) before first text;
    if class<12 then error(2);
    comment missing textstart before the first text;
```

```

for text_start:= parameter_number+1 while class<10 do
begin
  if text_start<max_param then
    parameter_number:= text_start else error(3);
  for class:= unpack_character while class=11 do;
  comment skipping separators (not : ) before text;
  words_in_text:= 1;
  if class<9 then
  begin
    comment text is not empty;
    text:= blank add char; i:= 1;
    for class:= unpack_character while class<9 or class=11 do
    begin
      text:= text shift 8 add char;
      i:= i+1;
      if i=6 then
      begin
        parameter_number:= parameter_number+1;
        if parameter_number=max_param
        then error(3);
        parameter(parameter_number):= text;
        text:= blank; i:= 0;
        words_in_text:= words_in_text+1
      end one real filled
    end packing legal char in text;
    if words_in_text>real_text then
    begin
      long_texts:= true;
      words_in_text:= real_text;
      parameter_number:= text_start+real_text;
      parameter(parameter_number):=
        parameter(parameter_number) shift (-8) shift 8
    end cutting long text
    else
    begin
      parameter_number:= parameter_number+1;
      parameter(parameter_number):=
        text shift ((6-i)*8)
    end text not too long
    end packing not empty text
  else
  begin
    parameter_number:= parameter_number+1;
    parameter(parameter_number):= blank
  end packing empty text;

  parameter(text_start):= words_in_text;
  comment words_in_text is placed as a real the
  preceding parameter;
end converting texts;

if long_texts then error(19);
state:= 7;
end kind of tail=1 (e.g. tail is a number of texts);

```

```
begin
  comment kind of tail=2 e.g. tail is a name;

  for class:= unpack character while class>10 do;
  comment skipping leading separators before name;
  if class=6 then text:= blank add char else error(4);
  comment first character in name must be a (small) letter
  and name must not be empty;
  i:= 1;
  for class:= unpack character while class<10 do
  if class=2 or class=6 then
  begin
    text:= text shift 8 add char;
    i:= i+1;
    if i=6 then
    begin
      parameter number:= parameter number+1;
      if parameter number=max_param
      then error(3);
      parameter(parameter number):= text;
      text:= blank; i:= 0
    end one real filled
  end character legal in name
  else
    error(5);

  parameter number:= parameter number+1;
  if parameter_number>real_name then
  begin
    parameter number:= real name;
    parameter(parameter number):=
    parameter(parameter number) shift (-8) shift 8;
    error(20)
  end cutting name parameter to real_name words (reals)
  with error message;
  state:= 7;
  parameter(parameter number):= text shift ((6-i)*8);
  comment preparation of the last parameter as text parameter;
  end kind of tail=2 (e.g. tail is a name);

begin
  comment kind of tail=3
  e.g. tail is a number of numbers (integers or reals);

  integer decimals, digit_number, exponent, leading part;
  real decimal part, exponential part, exponent sign, sign;
  state:= 1; comment state within one number parameter;

convert_numbers:
  case state of
  begin
    begin
      comment case 1 initializing and leading separators;

      leading part:= 0; decimal part:= 0.0;
      exponential part:= sign:= 1.0;
```

```

for class:= unpack_character while class>10 do;
comment skipping leading separators before leading part;
state:= case class of (0,3,2,4,5,8,8,8,8,7,0,0)
end case 1 (leading separators);

```

```

begin
comment case 2 sign before leading part;

if char=45 then sign:= -1.0;
comment sign was initialized with sign:= 1.0;
class:= unpack_character;
state:= case class of (0,3,8,4,5,8,8,8,8,8,8,8)
end case 2 (sign);

```

```

begin
comment case 3 leading part;

leading_part:= char-48;
for class:= unpack_character while class=2 do
if leading_part<838860 then
leading_part:= leading_part*10+char-48
else
error(6);
state:= case class of (0,0,8,4,5,8,8,8,8,6,6,6)
end case 3 (leading part);

```

```

begin
comment case 4 decimal part;

decimals:= digit_number:= 0;
for class:= unpack_character while class=2 do
if decimals<838860 then
begin
digit_number:= digit_number+1;
decimals:= decimals*10+char-48
end not overflow
else
error(7);
if digit_number=0 then error(8);
decimal_part:= 10.0*(-digit_number)*decimals;
comment scaling decimals to correct size;
state:= case class of (0,0,8,8,5,8,8,8,8,6,6,6)
end case 4 (decimal part);

```

```

begin
comment case 5 exponential part;

exponent:= digit_number:= 0;
exponent_sign:= 0.0;
for class:= unpack_character while class<4 do
if class=2 then
begin
if exponent_sign=0.0 then exponent_sign:= 1.0;
if exponent<60 then
begin
digit_number:= digit_number+1;

```

```
        exponent:= exponent*10+char-48
    end not exponent overflow
    else
        error(9)
    end class=2
    else
        if exponent sign=0.0 then
            exponent_sign:= if char=45 then -1.0 else 1.0
        else
            error(10);
        comment end class<4;
        if digit number=0 then error(11);
        if class<9 then error(12);
        exponential_part:= 10.0**(exponent*exponent sign);
        comment transforming exponential part to a factorial part;
        if leading part+decimal part=0.0 then leading part:= 1;
        comment numbers on the form +10-7 are accepted
        whereas 0.010-7, 010-7, and .010-7 all gives
        wrong conversion to 0.0000001 ;
        state:= 6
    end case 5 (exponential part);

begin
    comment case 6 final conversion of one number;

    parameter number:= parameter number+1;
    if parameter number>max_param
    then error(3);
    parameter(parameter number):=
    (leading part+decimal part)*exponential_part*sign;
    state:= if class>10 then 1 else 7
end case 6 (final conversion);

comment case 7 end of record and conversion.
case 7 in the big case (case action) is outrec
of correct converted record;
goto action;

comment case 8 syntax errors in numbers;
error(13)
end case state by converting numbers;

goto convert numbers
end kind of tail=3 (e.g. tail is a number of numbers);

begin
    comment kind of tail=4
    e.g. tail is empty (only separators are allowed);

    for class:= unpack character while class<>10 do
        if class<9 then error(14);
        state:= 7
    end kind of tail=4 (e.g. empty tail);
```

```

begin
    comment kind of tail=5 e.g. end of input record
    works as an end of medium record, but without listing;

    position:= 0;
    error(21)
    end case kind of tail=5 (e.g. end of input record)
    end case kind of tail statement at tail conversion
end case 6 (conversion of tail in all cases);

begin
    comment case 7 record reading, control, and conversion finished.
    outrec of the final parameters stored in parameter(1:parameter_number);

    i:= outrec(outputzone,0);
    comment i:= elements left in used share;
    parameter_number:= parameter_number+2;
    comment parameter_number:= elements in total output record;
    if i<parameter_number then
    begin
        outrec(outputzone,i);
        outputzone(1):= 0.0
    end active segment change;
    outrec(outputzone,parameter_number);
    outputzone(1):= parameter_number;
    outputzone(2):= kind of descriptor;
    for i:= parameter_number step -1 until 3 do
        outputzone(i):= parameter(i-2);
    ok_record:= ok_record+1;
    state:= 1
    end case 7 (outrec of correct record);

begin
    comment case 8 record contains more than max char
    characters. these have been listed by calling error(17)
    j denotes (from procedure error) number of characters
    written on the last line;

    for j:= j+1 while read_char(in,char) <>10 do
    begin
        write(out,false add char,1);
        if j=71 then
        begin
            write(out,<:<10>:>); j:= 0;
            lines_written:= lines_written+1
        end new line
    end listing rest of record;
    write(out,if char=60 then <:<60>:> else if char=62 then
    <:<62>:> else <:<69X77>:>);
    if char=62 then
    begin
        position:= 0; state:= 4
    end deletion of large record (+error message , -list)
    else
    if char=60 then
    begin
        state:= 7; ok_record:= ok_record-1
    end
    end
end

```

```
    end making error record (is not an ok_record)
  else
    state:= 5; comment large end of input record;
  end case 8 (record too large as text);

  comment case 9 problems with source for resulting records.
  this case is activated from the block procedure error_in_doc which
  calls error(24);
  goto finish_input
end case state_at reading records;

goto action;

finish_input:  intable(0)
end external procedure record_input;
end
```


polyrootc

1. Function and Parameters.

The procedure `polyrootc` determines the n complete roots of n -th degree polynomial with complex coefficients.

The procedure head is:

```
procedure polyrootc(n,a,z);
  value          n      ;
  array          a,z    ;
  integer        n      ;
```

Call parameters:

n : (integer) The degree of the polynomial ($n \geq 1$).
 $a(0:n,1:2)$: (array) Must on entry contain the coefficients of the polynomial:
 $a(j,1)$ = real part of a_j
 $a(j,2)$ = imaginary part of a_j ,
 where a_j is the complex coefficient which multiplies z^j .
 (a is unchanged after the call of the procedure.)

Return parameter:

$z(1:n,1:2)$: (array) Contains the n complex roots:
 $z(j,1)$ = real part of the j -th root,
 $z(j,2)$ = imaginary part of the j -th root.
 (Normally a larger value of j corresponds with a larger value of the modulus of the root.)

2. Method.

The procedure 'polyrootc' determines the n complex roots of the polynomial

$$P_n(z) = \sum_{j=0}^n a_j z^j, \quad a_n \neq 0$$

where the coefficients a_j are complex numbers.

The method is recursive: A root $d = d_r + i d_i$ of the polynomial $P_n(z)$ is found as shown below. Then by means of Horner's scheme a polynomial $P_{n-1}(z)$ of degree $n-1$ is determined:

$$P_{n-1}(z) \cdot (z-d) = P_n(z).$$

The whole process is repeated until the degree is reduced to one, and the root of a first degree polynomial is easily determined.

The determination of a root, d , of the equation $P_n(z) = 0$ is done in the following iterative manner:

From a known value $z = z^*$ a new value $z = \hat{z}$ is found, as shown below. At first we use $z^* = 0$. Because of the facts that $|P_n(z)| \geq 0$ and $|P_n(z)| = 0 \Leftrightarrow P_n(z) = 0$ we have as a convergence-criterion that $\hat{z}$ is a better approximation to a root (d) than z^* , when

$$|P_n(\hat{z})| < |P_n(z^*)|. \quad (1)$$

A number, d , is here considered as a root when

$$|P_n(d)|^2 \leq |P_n(0)|^2 \cdot 2.12_{10}^{-22}.$$

Below is given another root-criterion.

In determining $\hat{z}$ from a given z^* we first perform a 'normal - step' (step A, see below) and investigate whether the inequality (1) is satisfied. If this is the case the value $\hat{z}$ is used as a new value z^* and the process is repeated. If (1) is not satisfied another $\hat{z}$ is determined performing a 'special - step' (step B, see below). We again investigate whether (1) is satisfied or not, and it is decided whether step A or step B should be performed.

Step A, Normal step:

$P_n(z)$ is expanded from $z = z^*$

$$P_n(z) = \sum_{j=0}^n b_j (z-z^*)^j \quad (2)$$

where the coefficients b_j

$$b_j = \frac{P_n^{(j)}(z^*)}{j!}$$

are determined by a method which is similar to Horner's scheme. Here $b_0 \neq 0$ because $P_n(z^*) \neq 0$. When $b_1 \neq 0$ and the remaining coefficients are zero we have

$$P_n(z) = b_0 + b_1(z - z^*)$$

and the root is equal to

$$z = z^* - \frac{b_0}{b_1}.$$

When $b_1 = 0$, $b_2 \neq 0$ and the remaining coefficients are zero we have

$$P_n(z) = b_0 + b_2(z - z^*)^2$$

and the root is equal to

$$z = z^* + \sqrt[2]{-\frac{b_0}{b_2}}.$$

This is generalized to the case where $|b_s|$ is large, and the remaining coefficients (except b_0) are small, and we get

$$z \approx z^* + \sqrt[s]{-\frac{b_0}{b_s}}.$$

The value of s is determined from

$$\min_{j=1(1)n} \left| \frac{b_0}{b_j} \right|^{1/j}.$$

The value j for which this minimum is attained is equal to s . If the minimum is attained for more values of j the smallest one is used as s . The determination of the minimum value is carried out parallel with the computation of the coefficients b_j .

For use here (and in connection with step B) we compute

$$q := (|b_0|^2 / |b_s|^2)^{1/s}$$

$$r := q^{1/2}. \quad (3)$$

Then $\hat{z}$ is determined from z^* by means of the relation

$$\hat{z} := z^* + r(-b_0/b_s |b_s/b_0|)^{1/s} \quad (4)$$

or

$$\hat{z} := z^* + r \exp(ih)$$

where

$$h := \arg \left(-\frac{b_0}{b_s} \left| \frac{b_s}{b_0} \right| \right)^{1/s} = \frac{1}{s} \arg(-b_0 \bar{b}_s).$$

If

$$\hat{z} - z^* = r \cos(h) + i r \sin(h) \quad (5)$$

is equal to 'zero' in the sense of diskrete numbers, the iteration-process is discontinued, and $\hat{z}$ is used as the root d . Then $P_n(z)$ can be reduced to a polynomial $P_{n-1}(z)$.

If the difference (5) is not zero, $P_n(\hat{z})$ is computed using Horner's scheme, and when

$$|P_n(\hat{z})|^2 \leq |P_n(0)|^2 * 2.12_{10}^{-22} \quad (6)$$

the number $\hat{z}$ is considered as a root, d , and $P_{n-1}(z)$ can be determined.

If $\hat{z}$ is not a root according to (6) the values $|P_n(\hat{z})|$ and $|P_n(z^*)|$ are compared. If (1) is satisfied the process converges, and the value $\hat{z}$ just found is used as a new z^* , and Step A is repeated. If (1) is not satisfied the process does not converge using 'normal-steps' and a special emergency-step must be performed.

Step B, Special step:

Here a new value $\hat{z}$ is sought which is closer to z^* than the value $\hat{z}$ found by means of Step A. It is always possible to find a value $\hat{z}$ such that (1) is satisfied. Performing this step the value of r determined above (5) is halved: $r := r/2$.

The polynomial $P_n(z)$ is again written in the form (2) with unchanged coefficients. When $|z - z^*| = r$ (using the halved value) the term in the expansion (2) having the largest modulus is

$$\max_{j=1(1)n} r^j |b_j|.$$

The value j for which the maximum is attained is denoted by s ; when several values of j give maximum the smallest one is used. Notice that the determination of s here and in Step A are similar, but the derived values s need not to be the same.

Using the halved value of r and the derived value s one can compute a new $\hat{z}$ by means of the formula (4). Furthermore the investigation of the case $\hat{z} = z^*$ and whether the process converges or not according to (1) is carried out as described in connection with Step A. If the process does not converge, r is halved once more and Step B is repeated, and so on.

The method used here is derived from the works [2] and [3].

3. Time and Storage Requirements.

Time: Depends upon the number of roots ($= n$) and how they are located in the complex plane.

Storage Requirements: 6 segments, $4*n + 21$ words in stack where $n =$ degree of the polynomial, and 83 lines of program.

4. Test and Discussion.

A general discussion of the method is found in [2] and [3]. The procedure has been tested by applying it to several polynomials with known roots.

5. References.

- [1]: Dejon, Bruno and Peter Henrici (ed.): Constructive Aspects of the Fundamental Theorem of Algebra. Wiley - Interscience, London, 1969, 8 + 337 pp.
- [2]: Dejon, Bruno and Karl Nickel: A Never Failing, Fast Convergent Root-Finding Algorithm. [1] pp. 1-35.
- [3]: Nickel, Karl: Die numerische Berechnung der Wurzeln eines Polynoms. Numerische Mathematik, Vol. 9 (1966) pp. 80-98.

6. Algorithm.

```
polyrootc:=set 6
polyrootc:=algol message.yes
external

procedure polyrootc(n,a,z); value n;
message polyrootc, version 19 3 70, RCSL;
array a,z; integer n;

begin array br,bi(0:n-1),er,ei(1:n);
      integer i,j,k,s;
      real ar,ai,a2,cr,ci,c,fr,fi,f,zr,zi,p,q,dr,di,h,eps;

      ar:=a(n,1); ai:=a(n,2); a2:=ar**2+ai**2;

      for i:=n step -1 until 1 do
        begin z(i,1):=br(i-1):=a(i-1,1); z(i,2):=bi(i-1):=a(i-1,2)
        end i;

      for k:=n-1 step -1 until 1 do
        begin dr:=di:=0; f:=1'600;
          cr:=br(0); ci:=bi(0); er(k+1):=ar; ei(k+1):=ai;
          eps:=(cr**2+ci**2)*2.12'-22;

iteration:
          c:=cr**2+ci**2;
```

```
if c>eps then
begin

  if c<f then
  begin zr:=dr; zi:=di; fr:=cr; fi:=ci; f:=c;
    p:=if a2<c then 1.1/a2 else 1/c; q:=1;

    for j:=1 step 1 until k do
    begin dr:=ar; di:=ai;

      for i:=k step -1 until j do
      begin h:=dr;
        dr:=br(i):=br(i)+zr*dr-zi*di; di:=bi(i):=bi(i)+zr*di+zi*h
      end i;

      if (dr**2+di**2)*p>1 then
      begin p:=1/(dr**2+di**2); q:=(p*c)**(1/j); s:=j
      end;

      p:=p*q; er(j):=dr; ei(j):=di
    end j;

    if a2*p>1 then begin s:=k+1; q:=(c/a2)**(1/s) end

  end c<f else
  begin q:=q/4; p:=1/(er(1)**2+ei(1)**2); s:=1;

    for i:=2 step 1 until k do
    begin p:=p*q;
      if (er(i)**2+ei(i)**2)*p>1 then
      begin p:=1/(er(i)**2+ei(i)**2); s:=1
      end
    end i;

    if a2*p*q>1 then s:=k+1
  end c>f;
```

```

h:=arg( -fr*er(s)-fi*ei(s),fr*ei(s)-fi*er(s) )/s;
p:=sqrt(q);   dr:=zr+p*cos(h); di:=zi+p*sin(h);
write(out,<:<10>:>,<<-d.dddddddddd'-dd>,zr,zi,<< ddd>,s,<< -d.dddd'-dd>,
p,180/5.1415926536*h);
cr:=ar; ci:=ai;
for i:=k step -1 until 0 do
begin h:=cr;
  cr:=br(i):=z(i+1,1)+dr*cr-di*ci;
  ci:=bi(i):=z(i+1,2)+dr*ci+di*h
end i;

if zr<dr | zi<di then goto iteration
end c>eps;

for i:=1 step 1 until k do
begin z(i,1):=br(i-1):=br(i); z(i,2):=bi(i-1):=bi(i)
end i;

z(k+1,1):=dr; z(k+1,2):=di
end k;

z(1,1):=( -ar*br(0)-ai*bi(0) )/a2;
z(1,2):=( -ar*bi(0)+ai*br(0) )/a2
end polyrootc;

```


