STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

MR 53

General Purpose Vector Epsilon Algorithm
ALGOL Procedures

P. Wynn

1964

Numerische Mathematik 6, 22—36 (1964)

General Purpose Vector Epsilon Algorithm ALGOL Procedures*

By
P. WYNN

---

* Communication MR 53 of the Computation Department of the Mathematical Centre, Amsterdam.

1.

At the 1962 IFIP Congress in Munich the author gave an invited expository talk on acceleration techniques in Numerical Analysis. It had been his intention to include in the proceedings of this Congress two general purpose ALGOL procedures together with a number of short programmes illustrating their use. In this way work in what is at this time a critical domain of inquiry in Numerical Analysis would have been thrown open to as large as possible a forum of experimentation. Due to restrictions which were imposed upon space it was not possible to publish these procedures in the Congress proceedings; it is the purpose of this note to cause them to be published here. Before giving these procedures a short explanation is embarked upon.

2.

The ε-algorithm is a computational device for accelerating the convergence of a slowly convergent sequence S_m $(m=0, 1, \ldots)$. From the initial conditions

$$\varepsilon_{-1}^{(m)}=0 \qquad (m=1, 2, \ldots), \qquad \varepsilon_0^{(m)}=S_m \qquad (m=0, 1, \ldots)$$

further quantities $\varepsilon_s^{(m)}$ are constructed by means of the relationship

$$\varepsilon_{s+1}^{(m)}=\varepsilon_{s-1}^{(m+1)}+\{\varepsilon_s^{(m+1)}-\varepsilon_s^{(m)}\}^{-1} \qquad (m, s=0, 1, \ldots). \tag{1}$$

It transpires that in certain cases the various sequences $\varepsilon_{2s}^{(m)}$ $(s=0, 1, \ldots)$ for fixed m, converge far more rapidly than the original sequence S_m $(m=0, 1, \ldots)$. (For the theory of this algorithm see [1] and its references). The quantities $\varepsilon_s^{(m)}$ may be arranged in the following array:

$$\begin{array}{ccccccc}
 & & \varepsilon_0^{(0)} & & & & \\
 & \varepsilon_{-1}^{(1)} & & \varepsilon_1^{(0)} & & & \\
 & & \varepsilon_0^{(1)} & & \varepsilon_2^{(0)} & & \\
 & \varepsilon_{-1}^{(2)} & & \varepsilon_1^{(1)} & & \varepsilon_3^{(0)} & \\
 & & \varepsilon_0^{(2)} & & \varepsilon_2^{(1)} & \cdot & \\
 & \varepsilon_{-1}^{(3)} & & \varepsilon_1^{(2)} & \cdot & \cdot & \\
 & \cdot & \varepsilon_0^{(3)} & \cdot & \cdot & & \\
 & \cdot & & \cdot & & & \\
 & \cdot & & & & &
\end{array}$$

in which the quantities in (1) occur at the vertices of a lozenge:

$$\begin{array}{ccc} & \varepsilon_s^{(m)} & \\ \varepsilon_{s-1}^{(m+1)} & & \varepsilon_{s+1}^{(m)} \\ & \varepsilon_s^{(m+1)} & \end{array}$$

3.

The whole of the ε-array may be built up by use of a vector l of quantities which, at some stage, stretches from $\varepsilon_0^{(m)}$ to $\varepsilon_m^{(0)}$. A new S_{m+1} ($\equiv \varepsilon_0^{(m+1)}$) is computed and the vector l is displaced one space downwards in the ε-array, so as to stretch from $\varepsilon_0^{(m+1)}$ to $\varepsilon_{m+1}^{(0)}$. The process requires two auxiliary storage boxes *aux2* and *aux1*, and a working space *aux0*; in Fig. 1 the process is incomplete, the vector l contains quantities lying along the heavy line.

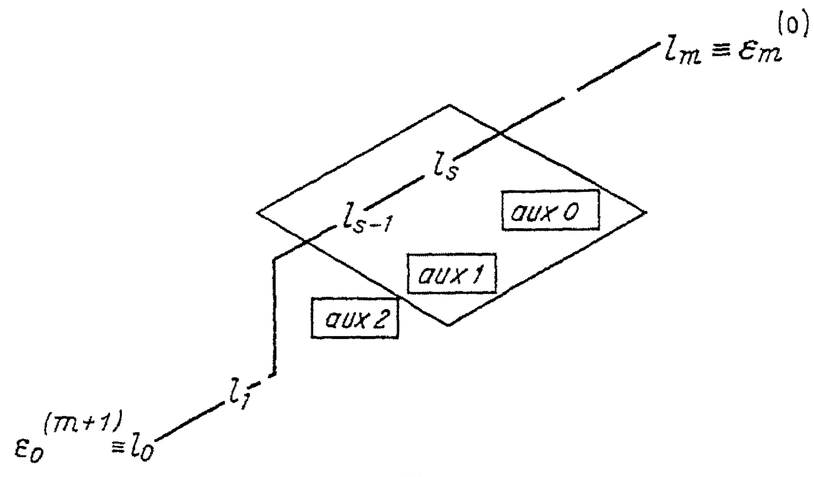

Fig. 1

The contents of l_{s-1}, l_s, *aux0* and *aux1* form a lozenge in the ε-array. The contents of *aux0* are computed from those of l_{s-1}, l_s and *aux1*; the contents of *aux2* are transferred to l_{s-1}, those of *aux1* to *aux2*, and those of *aux0* to *aux1*. The value of s is increased by unity and the process is repeated. If the quantities involved are non-scalar (i.e. vectors or matrices) it is quicker to change the labels on the boxes than move their contents around. In the procedures to be given there are three integer labels called *nought*, *one* and *two* and these take cyclically the values 0, 1 and 2.

4.

The ε-algorithm may be applied to slowly convergent vector sequences by using as a definition of the inverse of a vector a suggestion due to K. SAMELSON:

$$(y_1, y_2, \ldots, y_n)^{-1} = \left(\sum_{r=1}^{n} y_r \bar{y}_r\right)^{-1} (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_n).$$

The bars in this expression denote complex conjugate quantities: if the components of the vector are all real, the Samelson inverse is formed merely by dividing the vector throughout by the sum of the squares of its components. (This is done in the procedures by the procedure *real Saminv*.)

5.

Iterated vector sequences occur most naturally in Numerical Analysis in the following way. We are concerned with the function $\varphi(x)$ which is defined for $a \leqq x \leqq b$ and satisfies a functional equation of some sort (an integral equation, for example). This equation is solved numerically by finite difference methods, sub-dividing the range into N parts, we are concerned with a vector of solution values. The equation is solved iteratively, we obtain a sequence of vectors.

When computing such vector sequences attention must be focussed upon two quantities. Firstly the *distance* between the current vector and the next must be measured; in the procedures to be given the distance between two vectors $(une_1, une_2, \ldots, une_N)$ and $(autre_1, autre_2, \ldots, autre_N)$ is taken to be $\max_i \{abs\,(point\ distance_i)\}$, where $point\ distance_i = une_i - autre_i$. If the original iteration scheme converges this distance may be made as small as we please. Secondly some estimation of the *truncation error* must be made at each stage. Only if the distance between one iterate and its successor, and the estimate of the truncation error are both sufficiently small, can an iterated vector be accepted as a numerical solution to the problem in hand.

6.

Two general purpose vector ε-algorithm procedures are given.

The purpose of the first is to display the application of the ε-algorithm to iterated vectors produced by means of a functional equation as described above. The course of action adopted is as follows: Iterated vectors $\vartheta^{(m)}$ $(m=0, 1, \ldots, m\ \max)$ are produced by means of the finite difference treatment of a functional equation, and as they are produced the ε-algorithm is applied, using the vector l as described earlier. To each of the vectors in the even order columns of the ε-array one cycle of the original functional equation is applied, the corresponding distances and estimates of the truncation error are printed out in two separate triangular arrays.

It must be emphasized that the application of the functional equation to each of the entries in the even order ε-array should only be done for the purpose of display, with a relatively small value of N and for a few steps only, in order to see if application of the ε-algorithm has the desired effect.

7.

Numerical experience indicates that of all the sequences which may be derived by applying the ε-algorithm to a fixed number of iterated vectors, $\varepsilon_{2s}^{(0)}$ and $\varepsilon_{2s}^{(1)}$ $(s=0, 1, \ldots)$ converge the most rapidly.

Accordingly in the case of the second vector ε-algorithm procedure the course of action is as follows: The functional equation is iterated and the ε-algorithm applied; the distance between either $\varepsilon_m^{(0)}$ and $\varepsilon_{m-2}^{(2)}$ if m is even or $\varepsilon_{m-1}^{(1)}$ and $\varepsilon_{m-3}^{(3)}$ if m is odd is then examined. If this distance is less than some stipulated small quantity then the iteration-acceleration process may be at an end. The vector $\varepsilon_m^{(0)}$ (or $\varepsilon_{m-1}^{(1)}$) is then submitted to one iteration cycle of the original functional equation; if the distance between $\varepsilon_m^{(0)}$ (or $\varepsilon_{m-1}^{(1)}$) and its successor from the functional equation is less than the given stipulated agreement then for better or for worse the iteration is at an end. The corresponding estimate of the truncation error is then examined and if this is less than the given small quantity, the iterate resulting from $\varepsilon_m^{(0)}$ (or $\varepsilon_{m-1}^{(1)}$) is accepted as the required numerical solution. (If the estimate of the truncation error is too large, then the calculations can always be repeated with a smaller interval.)

8.

Now that the general mathematical and organisational considerations have been dealt with, we can proceed to the details of programming. We first give three auxilary procedures of which considerable use is made.

It will be recalled that it is quite important to know whether the suffix of the quantity $\varepsilon_s^{(m)}$ is even or odd. For this reason use is made of the following

```
boolean procedure even(integer);
value integer; integer integer;
even := (integer = (integer ÷ 2) ×2);
```

This has as input the integer *integer*; the value of the boolean procedure *even* is **true** if *integer* is even and **false** if *integer* is odd.

Furthermore it is frequently necessary to assign values to vectors. To assist in this and further arithmetic operations upon vectors, we adopt the convention that the suffix of each vector component is called i: the integer i is not used for any other purpose. A vector assignment of the sort indicated by

```
for i := 0 step 1 until length do une_i := autre_i;
```

is carried out by the following

```
procedure vecteq (une, autre, length);
value length; integer length;
real une, autre;
begin i := 0;
      EQUATE: une := autre;
      i := i + 1; if i ≦ length then goto EQUATE
end vecteq;
```

It can be seen from thus that the suffix of the first element of every vector is assumed to be zero; the programmes however do allow for vectors of variable length to be dealt with.

The Samelson inverse *res* of the real vector *it* is constructed by means of the following

```
procedure real Saminv (res, it, length);
value length; integer length;
real res, it;
begin real denom, compt;
      i := 0; denom := 0;
      DENOMINATOR: compt := it;
      denom := denom + compt × compt;
      i := i + 1; if i ≦ length then goto DENOMINATOR;
      vecteq (res, it/denom, length)
end real Saminv;
```

9.

We now give the vector ε-algorithm procedure intended for display, as described in § 6. The following variables are assumed to have been declared non-locally:

i: the component suffix as described in § 8.
aux[0:2, 0:*order*]: the boxes as described in § 3.
nought, one, two, spare label: the labels as mentioned in § 3.
distance: which indicates the distance between two successive iterates as described in § 5.
truncation error: as mentioned in § 7 (the value of this and the preceding variable we shall assume to be computed by means of the procedure *functional equation*).

When being used in conjunction with the procedure *display vectepsalg* it is assumed that the procedure *functional equation* takes its input vector from the box *aux2* and places its output vector into the box *aux0*.

It is frequently an advantage to know when the procedure *functional equation* is being used for the first time (an example of this will be given later). For this reason use is made of the further non-locally declared (boolean) variable *first time* to which the value **true** is assigned immediately prior to the first call of the procedure functional equation.

```
comment The following procedure makes use of the non-local functional procedures NLCR, which operates the New Line Carriage Return mechanism of the output printing apparatus, and print (...) which causes the value of the bracketed variable to be printed;
procedure display vectepsalg (order, mmax, functional equation, col);
comment The values of the indices of the vectors being treated run from 0 to order. mmax is the number of times the functional equation is iterated;
value order, mmax, col;
integer order, mmax, col;
procedure functional equation;
begin integer m, s, spare label, sanfang, two mmax;
    boolean printing distances;
    array l[0:mmax, 0:order],
        display[0:(mmax×(mmax+4))÷4, 0:1],
        theta M plus1[0:order];
    procedure take sample (ms);
    value ms; integer ms;
    begin first time := (m = 0);
        functional equation;
        comment After the values of distance and truncations error have
                been computed by means of functional equation, they
                are mapped onto the display vector;
        display[ms, 0] := distance;
        display[ms, 1] := truncation error;
comment If the samples are taken from the first column of the
        epsilon array, then the next iterate produced by the
        functional equation is clearly the next member of the
        original sequence. It is stored for future use;
```

```
        if ms ≦ mmax then
        vecteq (theta M plus 1[i], aux[nought, i], order)
end take sample;
two mmax := 2 × mmax;
comment First iterate put equal to zero;
vecteq (theta M plus 1[i], 0, order);
for m := 0 step 1 until mmax do
begin comment Labels on boxes given initial values;
        nought := 0; one := 1; two := 2;
        comment First member of new backward diagonal put in aux1;
        vecteq (aux[one, i], theta M plus 1[i], order);
        for s := 0 step 1 until m — 1 do
        begin comment epsilon algorithm process, see section 3;
                real Saminv (aux[nought, i], aux[one, i] — l[s, i], order);
                if s ≠ 0 then
                begin vecteq (aux[nought, i], aux[nought, i] + l[s — 1, i],
                                                                order);
                        vecteq (l[s — 1, i], aux[two, i], order)
                end non zero s;
                comment The labels on the boxes are now changed;
                spare label := nought;
                nought := two; two := one;
                one := spare label;
                if even (s) then
                take sample ((s × (two mmax — s)) ÷ 4 + m)
        end s;
        comment End of backward diagonal reached;
        if m > 0 then vecteq (l[m — 1, i], aux[two, i], order);
        vecteq (l[m, i], aux[one, i], order);
        if even (m) then
        begin two := one;
                take sample ((m × (two mmax — m + 4)) ÷ 4)
        end taking sample at end of l
end m;
printing distances := true;
TRIANGULAR ARRAY: NLCR; NLCR;
comment Procedure now prints out vertical strips, col columns in width,
                of the even order epsilon arrays. sanfang is the suffix of the
                column at the beginning of each strip;
for sanfang := 0 step 2 × col until mmax do
begin NLCR;
for m := 0 step 1 until mmax — sanfang ÷ 2 do
begin NLCR;
        for s := sanfang step 2 until sanfang + 2 × (col — 1) do
        begin comment Member of even order epsilon array is now
                        fished out from the display vector;
```

```
                    if (s ÷ 2 ≦ m) ∧ (m ≦ mmax − (s ÷ 2))
                    then print (display [(s × (two mmax + 2 − s)) ÷ 4 + m,
                        if printing distances then 0 else 1])
                end s
            end sanfang
        end m;
        if printing distances then
            begin printing distances := false;
                goto TRIANGULAR ARRAY
            end returning to print out truncation errors
    end display vectepsalg;
```

10.

The vector ε-algorithm procedure intended for serious application (as described in § 7) is now given. The non-locally declared variables of which this procedure makes use are as before. This time, however, it is assumed that the procedure *functional equation* takes its input vector from the box *aux2* and places its output vector into the box *aux1*.

```
procedure vectepsalg (result, order, stipulated agreement, functional equation,
                      available storage, storage not exceeded,
                      small enough truncation error);
value order, stipulated agreement, available storage;
real result, stipulated agreement;
integer order, available storage;
boolean storage not exceeded, small enough truncation error;
procedure functional equation;
begin integer s, spare label, m;
    array l [0: available storage ÷ (order + 1) − 1, 0: order],
        theta M plus 1 [0: order];
    real procedure test distance (une, autre);
    real une, autre;
    begin real abstand, point distance;
        abstand := 0; i := 0;
        MEASURE: point distance := une − autre;
        if abs (point distance) > abs (abstand) then
        abstand := point distance;
        i := i + 1; if i ≦ order then goto MEASURE;
        test distance := abstand
    end obtaining test distance;
    comment Epsilon process prepared, first iterate put equal to zero;
    m := 1; vecteq (theta M plus 1 [i], 0, order);
    vecteq (l [0, i], 0, order);
    EPSALG: comment Labels on boxes given initial values;
    s := nought := 0; one := 1; two := 2;
    first time := (m = 1);
    comment Last iterate put into aux2 in preparation for functional equation;
    vecteq (aux [two, i], theta M plus 1 [i], order);
    functional equation;
```

```
comment Next iterate stored array;
vecteq(theta M plus 1[i], aux[one, i], order);
EPSLOOP: comment epsilon algorithm process, see section 3;
real Saminv(aux[nought, i], aux[one, i] − l[s, i], order);
if s ≠ 0 then
begin vecteq(aux[nought, i], aux[nought, i] + l[s − 1, i], order);
      vecteq(l[s − 1, i], aux[two, i], order)
end;
comment The labels on the boxes are now changed;
spare label := nought;
nought := two; two := one;
one := spare label;
s := s + 1; if s < m then goto EPSLOOP;
comment End of backward diagonal now reached;
vecteq(l[m − 1, i], aux[two, i], order);
vecteq(l[m, i], aux[one, i], order);
if m ≧ 2 then
begin comment Examine distance and truncation error, see section 7;
      nought := (if even(m) then 1 else 0);
      if abs(test distance(l[m − 3 + nought, i], l[m − 1 + nought, i]))
                  < stipulated agreement then
      begin vecteq(aux[two, i], l[m − 1 + nought, i], order);
            functional equation;
            if abs(distance) ≦ stipulated agreement then
            begin storage not exceeded := true;
                  if truncation error ≦ stipulated agreement then
                  begin small enough truncation error := true;
                        vecteq(result, aux[one, i], order)
                  end complete success
                  else small enough truncation error := false;
                  goto END
            end examining truncation error
      end examining distance
end examining distance and truncation error;
m := m + 1;
if m × (order + 1) < available storage
then goto EPSALG
else storage not exceeded := false;
END:
end vectepsalg;
```

Note: It is remarked in passing that the preceding two procedures may be made to produce the results of the ϱ-algorithm merely by causing the integer s to be declared non-locally and changing the assignment

vecteq(*res*, *it*/*denom*, *length*);

of the procedure *real Saminv* to

vecteq(*res*, $(s+1) \times$ *it*/*denom*, *length*);

11.

The two vector ε-algorithm procedures referred to in the title of this paper have now been given. In order to illustrate their use we give an example which is of considerable interest in itself. It concerns the iterative solution of the Lichtenstein-Gershgorin equation:

$$\vartheta(s) = \frac{k}{\pi}\int_0^{\pi}\left\{\frac{k_1\vartheta(t)}{1-k_2\cos(t+s)} - \frac{k_1\vartheta(\pi-t)}{1+k_2\cos(t+s)}\right\}dt + \left.\begin{array}{r} \\ + 2\arctan\left\{\dfrac{k^{-1}\sin(s)}{(1-\cos(s))\,[k_3\cos(s)-k^{-2}]}\right\}\end{array}\right\} \quad (1)$$

where

$$k_1 = (k^2+1)^{-1}, \qquad k_2 = k_1(k^2-1), \qquad k_3 = 1-k^{-2}.$$

The equation is solved iteratively so as to produce the iterated vectors $\vartheta^{(m)}$ $(m=0, 1, \ldots)$ by use of the scheme:

$$\vartheta^{(0)} = 0, \qquad \vartheta^{(m+1)} = K\vartheta^{(m)} + \beta \qquad (m=0, 1, \ldots),$$

where the symbols K and β have a meaning made obvious by inspection of (1). The integrals are approximated by means of the operational formulae

$$\int_a^{a+nh} f(t)\,dt \equiv h\{\tfrac{1}{2}f_0 + f_1 + \cdots + f_{n-1} + \tfrac{1}{2}f_n + C\} \quad (2)$$

where

$$C = \tfrac{1}{2}(\Delta f_0 - \nabla f_n) - \tfrac{1}{24}(\Delta^2 f_0 + \nabla^2 f_n) + \tfrac{19}{720}(\Delta^3 f_0 - \nabla^3 f_n) - \tfrac{3}{160}(\Delta^4 f_0 + \nabla^4 f_n).$$

It is assumed that the magnitude of the truncation error during each iteration cycle is indicated by $\max\left(\left|\tfrac{3}{160}\Delta^4 f_0\right|, \left|\tfrac{3}{160}\nabla^4 f_n\right|\right)$ taken over the range of s.

A procedure for the iteration of the integral equation is now given. As before the index i of the component ϑ_i is a non-locally declared integer. The estimates of the *distance* between two successive iterates, and of the *truncation error*, are **real** variables declared non-locally. Use is also made of the non-local Boolean variable (*first time*) mentioned earlier; when this has the value **true** the iteration cycle is prepared. The procedure has as input a *parameter* (k), the *number of intervals* (N), and the *current* vector of real values. It produces the *next* iterate:

```
procedure Lichtenstein Gershgorin
          (next, current, parameter, number of intervals);
value parameter;
real next, current, parameter;
integer number of intervals;
begin integer N, two N, N durch 2;
      real pi;
own real array kernel, beta [0:number of intervals];
N := number of intervals;
two N := 2×N; N durch 2 := N ÷ 2;
pi := 3.14159 26535 89794;
if first time then
begin comment Prepare the integral equation;
```

```
real pi over N, au, ks, k1, k2, k3, k4,
     au1, au2, au3, au4, k times k1 over N;
comment The following procedure ensures that the correct value
        of the arctangent is taken;
real procedure correct side of cut;
correct side of cut:=(if au2≦0 then au2 else au2-2×pi);
ks:=parameter×parameter;
k1:=1/(1+ks); k2:=(ks-1)/(ks+1);
k4:=1/ks; k3:=1-k4;
pi over N:=pi/N;
k times k1 over N:=parameter×k1/N;
for i:=0 step 1 until N durch 2 do
begin comment The evaluation of the kernel and of beta is done
                for s increasing from 0 and decreasing from pi
                simultaneously;
    au:=cos(i×pi over N); au1:=k2×au;
    kernel[i]:=k times k1 over N/(1-au1);
    kernel[N-i]:=k times k1 over N/(1+au1);
    if i≠0 then
    begin au1:=sin(i×pi over N);
          au3:=k3×au;
          au4:=au1/parameter;
          au2:=2×arctan(au4/((1-au)×(au3-au4)));
          beta[i]:=correct side of cut;
          au2:=2×arctan(au4/((1+au)×(-au3-au4)));
          beta[N-i]:=correct side of cut
    end non zero i
    else
    begin beta[0]:=-pi; beta[N]:=0
    end zero i
end i
end preparing integral equation;
comment Cycling process: compute the integrand;
begin integer i1, j, ij1, ij2;
    real sum, theta t, theta pi minus t,
         forth integrand, back integrand,
         minus1n, forward diff, backward diff,
         last term, point distance;
    boolean Neven;
    array end, beginning [0:4, 0:4], c[1:4];
    c[1]:=1/12; c[2]:=-1/24;
    c[3]:=19/720; c[4]:=-3/160;
    Neven:=(N=2×Ndurch2);
    distance:=truncation error:=0; j:=0;
    comment j indicates the value of s;
    ITERATION: sum:=0; i1:=0;
```

```
comment i1 indicates the value of t;
INTEGRATION: if¬ first time then
begin i:=i1; theta t:=current;
      i:=N-i1; theta pi minus t:=current;
      ij1:=i1+j;
      comment ij1 indicates the value of t+s;
      if ij1>N then ij1:=two N-ij1;
ij2:=(if i1>j then N+j-i1 else N-j+i1);
comment ij2 indicates the value of s+pi-t;
forth integrand:=kernel[ij1]×theta t
                 -kernel[N-ij1]×theta pi minus t;
back integrand:=kernel[ij2]×theta pi minus t
                -kernel[N-ij2]×theta t;
comment The evaluation of the sum in (2) is being con-
        ducted in two directions simultaneously. (The
        symmetry of the kernel is being exploited);
sum:=sum+
(if i1=0 then (forth integrand+back integrand)×0.5
else if¬Neven∨(Neven∧i1≠Ndurch2)
     then forth integrand+back integrand
     else forth integrand);
if i1<4 then
begin comment Store function values in preparation for
                evaluation of C in (2);
      beginning[i1,0]:=forth integrand;
      end[4-i1,0]:=back integrand
end storing;
i1:=i1+1;
if i1≦Ndurch2 then goto INTEGRATION;
comment Evaluate C in (2);
minus1n:=-1;
for ij1:=1,2,3,4 do
begin for i1:=ij1 step 1 until 4 do
      begin beginning[i1,ij1]:=
            beginning[i1,ij1-1]
                -beginning[i1-1,ij1-1];
            end[i1,ij1]:=
                end[i1,ij1-1]-end[i1-1,ij1-1]
      end differencing;
      forward diff:=c[ij1]×beginning[ij1,ij1];
      backward diff:=minus1n×c[ij1]×end[4,ij1];
      sum:=sum+forward diff+backward diff;
      minus1n:=-minus1n
end computing C;
comment Record the maximum truncation error;
for last term:=abs(forward diff),
               abs(backward diff) do
```

```
            begin if last term > truncation error then
                    truncation error := last term
            end estimating truncation error
    end passing through t values;
    sum := sum + beta[j];
    i := j; next := sum;
    comment Estimate the distance between the current vector and
            the next;
    point distance := sum - current;
    if abs(point distance) > abs(distance) then
        distance := point distance;
    j := j + 1; if j < N then goto ITERATION
        end cycling process
end Lichtenstein Gershgorin;
```

12.

Three complete ALGOL programmes will now be given. The first serves to illustrate the use of the procedure just given, and carries out the straightforward iteration of the integral equation.

```
comment Iteration of Integral Equation;
begin integer m, i, M, M plus1, N, col;
        real distance, truncation error, k, delta;
        boolean first time;
        comment This comment must be replaced by the procedures vecteq and
                Lichtenstein Gershgorin;
        k := read; N := read; delta := read; col := read;
        comment read is a non-ALGOL procedure whose function is obvious;
        NLCR; print(k); print(N); print(delta);
        begin array theta[0:1, 0:N];
            m := M := 0; M plus1 := 1;
            vecteq(theta[M, i], 0);
            CYCLE: first time := (m = 0);
            Lichtenstein Gershgorin
                (theta[M plus1, i], theta[M, i], k, N);
            NLCR; print(m); TAB; print(distance);
            comment TAB is a non-ALGOL procedure which moves the type-
                    writer carriage to the next tabulator stop;
            if¬ first time then
            begin TAB; print(truncation error)
            end;
            if abs(distance) > delta then
            begin m := m + 1; M := M plus1; M plus1 := 1 - M;
                goto CYCLE
            end returning to iteration
```

```
else
begin NLCR; if truncation error ≦ delta then
        for i := 0 step 1 until N do
        begin if (i ÷ col) × col = i then NLCR;
                print(theta[M plus1, i])
        end printing component
    end printing whole solution
end block in which theta is declared
end
```

13.

The second complete programme serves to show how the procedure *display vectepsalg* should be used.

```
comment Display Application of Epsilon Algorithm to Iteration of Lichtenstein
        Gershgorin Integral Equation;
begin integer i, N, nought, one, two, mmax, col;
    real distance, truncation error, k;
    boolean first time;
    comment This comment must be replaced by the procedures Lichtenstein
            Gershgorin, even, vecteq, real Saminv, and display vectepsalg;
    k := read; N := read; mmax := read; col := read;
    NLCR; print(k); print(N);
    begin array aux[0:2, 0:N];
        procedure disp vectepsalg L G;
        Lichtenstein Gershgorin (aux[nought, i], aux[two, i], k, N);
        display vectepsalg (N, mmax, disp vectepsalg L G, col);
    end block in which the size of the boxes is declared
end
```

14.

The third complete programme illustrates the use of the procedure *vectepsalg*

```
comment Application of the Epsilon Algorithm to the Iteration of the Lichten-
        stein Gershgorin Integral Equation;
begin integer i, N, nought, one, two, col, very end;
    real distance, truncation error, k, delta;
    boolean perhaps successful, indeed successful, first time;
    comment This comment must be replaced by the procedures Lichtenstein
            Gershgorin, even, vecteq, real Saminv, and vectepsalg;
    k := read; delta := read; N := read;
    very end := read; col := read;
    NLCR; print(k); print(N); print(delta); print(very end);
    begin array aux[0:2, 0:N], final answer[0:N];
        procedure vecteps LG;
        Lichtenstein Gershgorin (aux[one, i], aux[two, i], k, N);
        vectepsalg (final answer[i], N, delta, vecteps LG, very end,
            perhaps successful, indeed successful);
```

```
if perhaps successful ∧ indeed successful then
for i := 0 step 1 until N do
begin if (i ÷ col) × col = i then NLCR;
      print (final answer [i])
end printing component
end block in which size of boxes and final result is declared
end
```

15.

Two series of numerical results which may be produced by use of the second complete programme, but which may nevertheless be used to verify the working of the other two, are given in Tables 1 and 2. Here the range $0 - \pi$ has been divided into $(N=)$ 72 steps and the parameter k has been taken to be 7·5. The two triangular arrays described in § 6 are as follows:

Table 1. *Distances*

m \ s	0	2	4	6
0	−5·41101			
1	+3·85450	−0·50711		
2	−2·69179	+0·04957	−0·01263	
3	+1·89641	−0·01810	+0·00203	−0·00031
4	−1·36144	+0·00912	−0·00042	
5	+0·99621	−0·00508		
6	−0·73976			

Table 2. *Estimates of the truncation error*

m \ s	0	2	4	6
0	0·0			
1	0·00007	0·00005		
2	0·00004	0·00005	0·00005	
3	0·00005	0·00004	0·00005	0·00005
4	0·00004	0·00005	0·00005	
5	0·00005	0·00005		
6	0·00004			

Examination of Table 1 indicates that the convergence of the original iterative scheme (indicated by the successive distances $-5{\cdot}4$, $+3{\cdot}8$, $-2{\cdot}7$, ...) is rather slow but that that of the transformed sequence $\varepsilon_{2s}^{(0)}$ $(s=0, 1, \ldots)$ (indicated by the successive distances $-5{\cdot}4$, $-0{\cdot}51$, $-0{\cdot}013$, ...) is much more rapid.

16.

It will be realised that the given vector ε-algorithm procedures have quite general application in Numerical Analysis. In particular they may be applied to iterative techniques of linear algebra (the Jacobi relaxation scheme, the Gauss-Seidel relaxation scheme, and so on). (Of course, the estimation of the truncation error may be dispensed with here.) Furthermore, since any manifold

which is likely to occur in Numerical Analysis may easily be mapped onto a vector, these procedures may easily be applied to the iterative solution of partial differential equations and further related problems.

17.

The results displayed in Tables 1 and 2 were produced on the X1 computer in Amsterdam using an ALGOL translator constructed by J. A. ZONNEVELD and E. W. DIJKSTRA.

References

[1] WYNN, P.: Acceleration Techniques in Numerical Analysis, with Particular Reference to Problems in One Independent Variable. Proceedings of the IFIP Congress 1962. North Holland Publishing Co. pp. 149—156.

Mathematish Centrum
2e Boerhaavestraat 49, Amsterdam-O

(Received July 12, 1963)