

Algorithms

J. G. HERRIOT, Editor

ALGORITHM 348 MATRIX SCALING BY INTEGER PROGRAMMING [F1]

R. R. KLIMPEL (Recd. 4 Mar. 1968, 13 June 1968, 16 Oct. 1968 and 21 Nov. 1968)

Computation Research Laboratory, The Dow Chemical Co., Midland, MI 48640

KEY WORDS AND PHRASES: integer programming, linear algebra, mathematical programming, matrix condition, matrix scaling

CR CATEGORIES: 5.14, 5.41

procedure *scale* (*a*, *m*, *n*, *g*, *u*, *v*);
 value *m*, *n*, *g*; **integer** *m*, *n*; **real** *g*;
 real array *a*; **integer array** *u*, *v*;
comment The use of scaling to precondition matrices so as to improve subsequent computational characteristics is of considerable importance. To measure the scaling condition of a matrix, a_{ij} ($i=1, \dots, m$ and $j=1, \dots, n$), Fulkerson and Wolfe [1] suggested the ratio of the matrix entry of largest absolute value to that of the smallest nonzero absolute value. This procedure implements the method of [1], i.e. finding multiplicative row factors, r_i , and column factors, s_j , which, when applied, minimize the above condition number. The minimization problem can be expressed as an equivalent additive discrete problem by taking logarithms and defining:

$$r_i = g^{u_i}, \quad s_j = g^{v_j}, \quad b_{ij} = \log_g(\text{abs}(a_{ij}))$$

and taking c_{ij} to be the least integer greater than or equal to b_{ij} . Thus the formulation becomes: minimize an integer w subject to the constraints $0 \leq u_i + v_j + c_{ij} \leq w$ where u_i and v_j are unrestricted and integral in value. The effect of decreasing the value of the base g would be to more accurately approximate the continuous scaling problem by the discrete form.

REFERENCE:

1. FULKERSON, D. R., AND WOLFE, P. An algorithm for scaling matrices. *SIAM Rev.* 4 (1962), 142-146;

begin
 integer array *c*[1:*m*, 1:*n*], *ri*[1:*m*], *si*[1:*n*];
 real *val*;
 integer *max*, *store*, *markr*, *markc*, *num*, *nopt*, *i*, *j*;
 nopt := 0;
 comment Create initial integer matrix *c*. Due to machine round-off errors, it may be desirable for some problems to insert a tolerance when checking for zero values of the input matrix and for matrix entries which are exact integral powers of the base *g*;
 for *i* := 1 **step** 1 **until** *m* **do**
 for *j* := 1 **step** 1 **until** *n* **do**
 begin
 if (*a*[*i*, *j*]=0) **then**
 begin
 c[*i*, *j*] := 0;
 go to *intf*
 end;
 end;

val := $\ln(\text{abs}(a[i, j]))/\ln(g)$;
 c[*i*, *j*] := *entier*(*val*) + 1;
 if ((*c*[*i*, *j*]-1)=*val*) **then** *c*[*i*, *j*] := *c*[*i*, *j*] - 1;
intf:
 end;
 comment Select initial values of u_i and v_j that satisfy constraints of discrete formulation;
 for *i* := 1 **step** 1 **until** *m* **do**
 begin
 u[*i*] := *c*[*i*, 1];
 for *j* := 2 **step** 1 **until** *n* **do**
 if (*c*[*i*, *j*] < *u*[*i*]) **then** *u*[*i*] := *c*[*i*, *j*];
 u[*i*] := -*u*[*i*];
 end;
 for *j* := 1 **step** 1 **until** *n* **do**
 begin
 v[*j*] := *c*[1, *j*] + *u*[1];
 for *i* := 2 **step** 1 **until** *m* **do**
 begin
 store := *c*[*i*, *j*] + *u*[*i*];
 if (*store* < *v*[*j*]) **then** *v*[*j*] := *store*;
 end;
 v[*j*] := -*v*[*j*];
 end;
 comment Step one. Initialize row and column markers with unmarked rows and columns denoted by a 1 in *ri*[*i*] and *si*[*j*], respectively. Locate and mark maximum entry of current working array;
rcmax: *max* := 0;
 for *i* := 1 **step** 1 **until** *m* **do**
 begin
 ri[*i*] := 1;
 for *j* := 1 **step** 1 **until** *n* **do**
 begin
 if (*i* = 1) **then** *si*[*j*] := 1;
 if (*nopt*=0) **then** *c*[*i*, *j*] := *u*[*i*] + *v*[*j*] + *c*[*i*, *j*];
 if (*c*[*i*, *j*] ≥ *max*) **then**
 begin
 markr := *i*;
 markc := *j*;
 max := *c*[*i*, *j*];
 end
 end
 end;
 nopt := 1;
 ri[*markr*] := -1;
 comment Repeat steps two and three in succession until there are either no freshly marked rows or no freshly marked columns. Any row or column marked in the immediately preceding application of step one, two, or three is called freshly marked and denoted by -1 in the appropriate indicator vector. Previously marked rows and columns that are not freshly marked are denoted by zero values;
 comment Step two;
rmarks: *num* := 0;
 for *i* := 1 **step** 1 **until** *m* **do**
 begin
 if (*ri*[*i*] > -1) **then go to** *rmarkf*;

```

    ri[i] := 0;
    num := num + 1;
    for j := 1 step 1 until n do
        if (si[j]=1) ∧ (c[i, j]=0) then si[j] := -1;
rmarkf:
    end;
    if (num=0) then go to change;
    comment Step three;
    num := 0;
    for j := 1 step 1 until n do
    begin
        if (si[j]>-1) then go to cmarkf;
        si[j] := 0;
        num := num + 1;
        for i := 1 step 1 until m do
            if (ri[i]=1) ∧
                ((c[i, j]=max) ∨ (c[i, j]=(max-1))) then
                ri[i] := -1;
        cmarkf:
    end;
    if (num≠0) then go to rmarks;
    comment Step four. Modify integer scaling factors u and v
    and adjust current working matrix (cij+ui+vj);
    change: if (si[markc]<1) then go to finis;
    for i := 1 step 1 until m do
    if (ri[i]<1) then
    begin
        u[i] := u[i] - 1;
        for j := 1 step 1 until n do
            c[i, j] := c[i, j] - 1
    end;
    for j := 1 step 1 until n do
    if (si[j]<1) then
    begin
        v[j] := v[j] + 1;
        for i := 1 step 1 until m do
            c[i, j] := c[i, j] + 1
    end;
    go to rmax;
    finis:
end

```

ALGORITHM 349
 POLYGAMMA FUNCTIONS WITH ARBITRARY
 PRECISION* [S14]

ADILSON TADEU DE MEDEIROS AND
 GEORGES SCHWACHHEIM (Recd. 15 Mar. 1968, 1 July
 1968, 28 Oct. 1968 and 3 Dec. 1968)
 Centro Brasileiro de Pesquisas Físicas, Rio de Janeiro,
 ZC 82, Brasil

* This work was supported by the Conselho Nacional de Pesquisas
 and the Banco Nacional do Desenvolvimento Economico of Brasil.

KEY WORDS AND PHRASES: polygamma function, psi
 function, digamma function, trigamma function, tetragamma
 function, pentagamma function, special functions
 CR CATEGORIES: 5.12

```

procedure polygamma (n, z, nd, polygam, error);
    value n, z, nd; real z, polygam; integer n, nd; label error;
    comment This procedure assigns to polygam the value of the
    polygamma function of order n for any real argument z. For
    n = 0, we have the psi or digamma function, for n = 1 the tri-

```

gamma function, for n = 2 the tetragamma function, and so on.
 For arguments that are poles of the function (nonpositive
 integer values), an exit is made through the label *error*. The
 parameter *nd* gives the requested relative precision expressed
 in number of decimal digits.

It computes the polygamma function through the asymptotic
 series

$$\psi^{(n)}(z) \sim (-1)^{n-1} \left[\frac{(n-1)!}{z^n} + \frac{n!}{2z^{n+1}} + \sum_{k=1}^{\infty} B_{2k} \frac{(2k+n-1)!}{(2k)! z^{2k+n}} \right]$$

except for n = 0, when the first term is -ln(z).

If the simple empirical relationship

$$2z > n + nd$$

is true, as well as $z > n$, one enters directly into the asymptotic
 series with the original argument. Otherwise, the computation
 of small arguments is reduced to that of sufficiently large argu-
 ments, applying repeatedly the recurrence relation:

$$\psi^{(n)}(z+1) = \psi^{(n)}(z) + (-1)^n n! z^{-n-1}$$

To save computation time, the argument, once larger than n,
 is increased just to the point when the minimum term of the
 asymptotic expansion is sufficiently small so as not to alter the
 value of the result within the chosen precision.

The order of the minimum term is estimated by the first order
 approximation

$$\pi z - n/2,$$

and the corresponding absolute value by the approximation
 formula

$$(2\pi)^n \exp(-2\pi z).$$

Negative arguments are related to positive ones through the
 reflection formula:

$$(-1)^n \psi^{(n)}(1-z) = \psi^{(n)}(z) + \pi \frac{d^n}{dz^n} \cot \pi z$$

The *n*th-order derivative of the cotangent is computed by
 term by term differentiation of the tangent or cotangent series
 after the convenient trigonometric reductions of the argument's
 value.

This procedure is not recursive and uses no own variable;
begin

```

real pi, pf, soma, zq, tl, fac, prec, w, sab, pv;
integer pr, nl, kl, ml;
real procedure fat (n);
    value n; integer n;

```

```

begin
    real f; integer i;
    f := 1;
    for i := n step -1 until 2 do f := f × i;
    fat := f
end of fat;

```

```

procedure inc (s, x1, L);
    real s, x1; label L;

```

```

begin
    real sant;
    sant := s; s := s + x1;
    if abs (s-sant) ≤ abs (prec × s) then go to L
end of inc;

```

comment The procedure *polygamma* uses a table of coeffi-
 cients *sb* for its series with the value

$$sb(i) = \frac{|B_{2i}|}{(2i)!} = \frac{\sum_{k=1}^{\infty} (-1)^{k-1}/k^{2i}}{\pi^{2i}(2^{2i-1}-1)} \cong \frac{2}{(2\pi)^{2i}},$$

the last being an asymptotic value for large i . The computation of these coefficients need not to be repeated at each procedure call; so it is convenient to transfer the declaration and block below to the main program and execute it just once.

One should replace *flund* by the smallest positive real number within the machine representation, and *ms* by the number of decimal digits of the mantissa;

```

array sb [1 : entier (.272 × ln(2/flund))];
begin
  real piq, sm, pipo, ptwo, dpi, sa;
  integer sg, in, k2, imax;
  array tr, q[2 : entier (10 ↑ (ms/22))+1];
  imax := entier (.272 × ln(2/flund));
  piq := 9.86960440108935861883449099987615113531369940724079;
  pipo := piq ↑ 11; ptwo := 2097152; dpi := 4 × piq;
  sb [1] := 1/12;
  sb [2] := 1/720;
  sb [3] := 1/30240;
  sb [4] := 1/1209600;
  sb [5] := 1/47900160;
  sb [6] := 691/1307674368103;
  sb [7] := 1/74724249600;
  sb [8] := 3617/1067062284288104;
  sb [9] := 43867/5109094217170944103;
  sb [10] := 174611/8028576626982912105;
  sm := 1; sg := -1;
  for in := 2, in + 1 while sm ≠ sa do
  begin
    q[in] := 1/(in × in);
    tr[in] := sg × q[in] ↑ 11; sa := sm;
    sm := sm + tr[in]; sg := -sg
  end;
  sb[11] := sm/(pipo × (ptwo-1));
  for k2 := 12 step 1 until imax do
  begin
    sm := 1; in := 1;
  B: in := in + 1; tr[in] := tr[in] × q[in]; sa := sm;
    sm := sm + tr[in]; if sa ≠ sm then go to B;
    pipo := pipo × piq; ptwo := ptwo × 4;
    sb[k2] := sm/(pipo × (ptwo-1));
    if in = 2 then go to L
  end;
  go to A;
  L: for k2 := k2 + 1 step 1 until imax do
    sb[k2] := sb[k2-1]/dpi;
  A: end of sb coefficients computation;
  pi := 3.14159265358979323846264338327950288419716939937510;
  prec := 10 ↑ (-nd); fac := fat (n);
  pr := if n ÷ 2 × 2 = n then 1 else - 1;
  pf := pr × fac; n1 := n + 1;
  if z ≤ 0 then
  begin
    if z = entier(z) then go to error
  else
  begin
    real x, y; integer d, l; Boolean C;
    k1 := pr; d := z; x := d - z;
    if x > 0 then l := 1
  else
    begin x := -x; l := -pr end;
    C := x > .25; y := pi × ( if C then (.5-x) else x);
    if n = 0 then
      soma := l × pi × ( if C then sin(y)/cos(y) else cos(y)/sin(y) )
    else
  begin
    integer m, np, j, i; integer array ft [1:4];

```

```

    real y2, p, f, t, s, v;
    m := n ÷ 2; np := m × 2;
    ft[1] := np + 1; ft[2] := np; ft[3] := pr;
    ft[4] := 0; y2 := y × y; j := m + 1;
    f := fat(np+1); p := 4 ↑ (m+1);
    t := if pr = -1 then 1 else y;
    s := if C then 0 else pf/y ↑ n1;
  E: v := if C then p × (1-p) else p;
    inc(s, -sb[j] × f × t × v, D);
    for i := 1 step 1 until 4 do
      ft[i] := ft[i] + 2;
    f := f × ft[1] × ft[2] × y2/(ft[3] × ft[4]);
    p := 4 × p; j := j + 1;
    go to E;
  D: soma := l × pi ↑ n1 × ( if C then s × pr else s )
  end
end;
z := 1 - z; w := z ↑ n;
pv := if n = 0 then ln(z) else fac/(n × w);
sab := abs(soma);
if pv < sab then nd := nd - .434 × ln(sab/pv)
end
else
begin soma := 0; k1 := 1; w := z ↑ n end;
if nd ≤ 0 then go to L;
if 2 × z < n + nd ∨ z < n then
begin
  real term, cond;
  term := -pf/(z × w);
  inc(soma, term, L);
  cond := (n × 1.8378 - ln(abs(term)) + 2.3025 × nd) × .1591;
  if cond < n then cond := n;
  if cond ≤ z then z := z + 1
  else
  begin
    integer ip, k;
    ip := cond - z + 1;
    if ip < 1 then go to L;
    for k := 1 step 1 until ip do
      inc(soma, -pf/(z+k) ↑ n1, L);
      z := z + ip + 1
  end
  w := z ↑ n
end;
  inc(soma, if n=0 then ln(z) else -pf/(n × w), L);
  inc(soma, -pf × .5/(z × w), L);
  zq := z × z; t1 := pf × n1/(w × zq);
  for m1 := 2 step 2 until 6.283 × z + n do
  begin
    inc(soma, -t1 × sb[m1÷2], L);
    t1 := -t1 × (n1+m1) × (n+m1)/zq
  end;
  L: polygam := soma × k1
end of polygamma

```

The policy concerning the contributions of algorithms to *Communications of the ACM* appears, most recently, in the January 1969 issue, page 39. A contribution should be in the form of an algorithm, a certification, or a remark. An algorithm must normally be written in the ALGOL 60 Reference Language or in USASI Standard FORTRAN or Basic FORTRAN.