An Algorithm for the Newton Resultant

John Canny*
Paul Pedersen**

TR 93-1394
October 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Supported by a David and Lucille Packard Foundation Fellowship and by NSF PYI Grant IRI-8958577.
** Partially supported by NYI (NSF) CCR-9258533.
An Algorithm for the Newton Resultant

JOHN CANNY *        PAUL PEDERSEN **
U.C. BERKELEY       CORNELL UNIVERSITY

1 Introduction

Given a system of $n + 1$ generic Laurent polynomials, for $i = 1, \ldots, n + 1$,

$$f_i(x) = \sum_{q \in \mathcal{A}_i} c_{iq} x^q; \quad q = (q_1, \ldots, q_n); \quad x^q = x_1^{q_1} x_2^{q_2} \cdots x_n^{q_n}; \quad (1.1)$$

with (finite) support sets $\mathcal{A}_i \subseteq L$, where $L$ is some affine lattice isomorphic to $\mathbb{Z}^n$; we consider the Newton resultant $R(f_1, f_2, \ldots, f_{n+1})$. This is the unique (up to sign) irreducible polynomial with coefficients in $\mathbb{Z}$ and monomials in the $c_{iq}$ which determines whether or not system (1.1) has common roots in the algebraic torus $(\mathbb{C} - \{0\})^n$. The resultant depends only on the Newton polytopes $N_i := conv(\mathcal{A}_i) \subseteq \mathbb{R}^n$ of the sets $\mathcal{A}_i$. If the system could not have common roots in the torus for any choice of the $c_{iq}$, then the resultant is defined to equal 1, by convention. What we call the Newton resultant appears as the “sparse mixed resultant” in [CE], [PS], and the “$\mathcal{A}$–resultant” in [GKZ].

Our terminology emphasizes the dependence on the combinatorics of the Newton polytopes, and removes the misleading reference to sparsity in the sense of having few monomials. The algebraic torus is the natural setting for us, since we are interested in the properties of systems of polynomials which are invariant under symmetries of the affine lattice $L$. Translation by $q \in L$ corresponds to multiplication by $x^q$ at the level of polynomials, and since $x^q$ may have negative exponents, we should restrict to points none of whose coordinates are zero, i.e. the algebraic torus. We refer to roots in the algebraic torus as toric roots.

2 Basics about mixed volumes

The Minkowski sum $A \oplus B$ of polytopes $A, B \subseteq \mathbb{R}^n$ is their pointwise sum, $A \oplus B := \{ u + v : u \in A, v \in B \}$. We shall consider the iterated Minkowski

---

* Supported by a David and Lucile Packard Foundation Fellowship and by NSF PYI Grant IRI-8958377

** Partially supported by NYI (NSF) CCR-9258533
sum $N := N_1 \oplus \ldots \oplus N_{n+1}$.

A basic combinatorial-geometric invariant of any $n$ lattice polytopes $P_1, \ldots, P_n$ is their mixed volume $\mathcal{MV}(P_1, \ldots, P_n)$. It is defined as the coefficient of the monomial $\lambda_1 \lambda_2 \cdots \lambda_n$ in the polynomial $\text{Vol}(\lambda_1 P_1 \oplus \lambda_2 P_2 \oplus \ldots \oplus \lambda_n P_n)$. Classic expositions on mixed volumes may be found in [BF], and [E].

The mixed volume is determined (and may be computed) using mixed subdivisions of the Minkowski sum $P := P_1 \oplus \ldots \oplus P_n$. Mixed subdivisions arise by choosing (randomly) lifting functionals $\omega_i : \mathbb{R}^n \to \mathbb{R}$, and then considering “lifted polytopes” $\widehat{P}_i \subset \mathbb{R}^{n+1}$ = the graph of $P_i$ under $\omega_i$:

$$\widehat{P}_i := \{ (q, \omega_i(q)) : q \in P_i \}.$$ 

Let $\widehat{P} = \widehat{P}_1 \oplus \ldots \oplus \widehat{P}_n$ denote the Minkowski sum of the lifted polytopes. The lower convex hull $\Delta$ of $\widehat{P}$ is the collection of facets $\widehat{F}$ of $\widehat{P}$ whose inner normal has positive last component. Each such facet has the form $\widehat{F} = \widehat{F}_1 \oplus \ldots \oplus \widehat{F}_n$, where $\widehat{F}_i$ is a face of $\widehat{P}_i$. We say that $\widehat{F}$ is a mixed facet if $\dim(\widehat{F}_i) = 1$ for $i = 1, \ldots, n$. More generally, any facet of $\Delta$ has a type $(d_1, \ldots, d_n)$, where $d_i = \dim(\widehat{F}_i)$. Facets of $\Delta$ may be found using linear programming.

Suppose $\pi : \mathbb{R}^{n+1} \to \mathbb{R}^n$ projects to the first $n$ coordinates, then $\Delta := \{ \pi(\widehat{F}) : \widehat{F} \in \Delta \}$ is a subdivision of $P$. Each cell $C$ of $\Delta$ has the form $\pi(\widehat{F}) = F_1 \oplus \ldots \oplus F_n$ where $\pi(\widehat{F}_i) = F_i$ is a face of $P_i$. We assume that the weights $\omega$ are sufficiently generic, meaning that $\dim(C) = \dim(F_1) + \ldots + \dim(F_n)$, for every $C \in \Delta$. In this case $\Delta$ is called a mixed subdivision of $P$. The projection of mixed facets of $\Delta$ are mixed cells of $\Delta$. The sum of the volumes of the mixed cells of any mixed subdivision $\Delta$ equals the mixed volume $\mathcal{MV}(P_1, \ldots, P_n)$. This is an integer which does not depend on the choice of subdivision. For details on mixed subdivisions and their relation to mixed volumes see [Bet].

Mixed cells are parallelotopes. If one considers them “half-open”, then their volume equals their number of lattice points. We may determine a consistent sense of “half open” by fixing some random direction $\delta$, and only counting lattice points lying on facets where $\delta$ is inward pointing, for instance.

In our application we have $n+1$ polytopes. Their Minkowski sum also has a mixed decomposition obtained by means identical to the above, i.e. as the projection of the lower envelope of a lifted sum, with the difference that now every cell, even a mixed cell, is the sum of at least one vertex from one of the $n+1$ inputs. Mixed cells in this setting are sums of $n$ edges and one vertex, so that lattice points lying in mixed cells of a mixed decomposition of $N = N_1 \oplus \ldots \oplus N_{n+1}$ have associated to them a unique index which contributes the vertex. This index is crucial to the construction we are going to describe.
3 Basics about Newton Resultants

The mathematical foundations for the theory of Newton resultants can be found in [GKZ], [PS], and [CE], where it is shown that:

\[ \deg f_j(R) = MV(N_1, \ldots, \widehat{N_j}, \ldots, N_{n+1}). \]

\[ R(\ldots, f_j \cdot f'_j, \ldots) = R(\ldots, f_j, \ldots) \cdot R(\ldots, f'_j, \ldots). \]

\[ R(f_1, f_2, \ldots, f_{n+1}) = R' \cdot \prod_{\gamma} f_1(\gamma), \]

where the product in item (3) is over toric roots of \( f_2 = \ldots = f_{n+1} = 0 \), and \( R' \) is a product of lower dimensional resultants not involving \( f_1 \).

In [CE] an algorithm is proposed for computing the Newton resultant using analogues of Macaulay matrices. We shall describe an explicit implementation of this algorithm with complexity estimates. Maple code for the implementation is included in the Appendix.

The basic principle underlying the construction is simple: regard each equation \( f_j(x) = 0 \) as a linear relation on the power-products \( x^q : q \in A_j; \)

\[ f_i(x) = (c_1, \ldots, c_n) \cdot (x^{q_1}, \ldots, x^{q_n}) = 0. \]  

Any common solution \( x_0 \) determines a null vector of this linear system by evaluation of the monomials \( x^q \) at \( x_0 \). However, one typically has more variables in \( \cup_j A_j \) than equations \( (n + 1) \). New equations can be generated by taking monomial multiples \( x^u f_j = 0 \) of the old equations. This generates new power-products \( x^{u+q} \), and the game is to pick multiples which generate as few new power-products as possible, so that one eventually ends up with as many equations as variables. Suppose we write this linear system as \( M \cdot (x^u) = 0 \), where \( M \) has components \( c_{ij} \) as in (3.1).

The vanishing of the determinant of this linear system is a necessary condition for the existence of a common root. Therefore the resultant, which is irreducible ([PS]) must divide \( \det(M) \). Multivariate factorization of the determinant, and extraction of the factor of appropriate degree (according to item (1) above) gives you the resultant.

This is exactly the type of algorithm defined in [CE], where a matrix \( M \) is constructed whose columns are indexed by the monomials of the Minkowski sum \( N = N_1 \oplus \ldots \oplus N_{n+1} \), and whose rows are indexed by monomial multiples
of the $f_j$'s: If $p \in C \subset N = N_1 \oplus \ldots \oplus N_{n+1}$, and $C = F_1 \oplus \ldots \oplus F_{n+1}$, and
if $i$ is the largest index for which $F_i$ is a vertex — say $q_j$, corresponding to the $j$-th monomial $c_{ij} x^{q_i}$ of $f_i$ — then let $RC(p) := (i, j)$. This is the so-called "row content" of $p$. Define $M$ by

$$
M_{p,q} = \begin{cases} c_{ik}, & \text{if } q - p + q_{ij} = q_{ik} \text{ for some } k, \text{ where } (i, j) = RC(p) \\
0, & \text{if } q - p + q_{ij} \in A_i
\end{cases}
$$

We refer to [CE] for details of the proof that the matrix so constructed is square and has non-identically-vanishing determinant. We refer to this matrix $M$ as a Newton matrix for the system (1.1).

Our objective here is to describe a concrete implementation of this algorithm. Optimally there would be exactly $\mathcal{MV}(f_1, \ldots, f_j, \ldots, f_n)$ rows indexed by multiples of $f_j$, so that the degree of the determinant $\det(M)$ would coincide with the degree predicted in item (1) above. Generally speaking, this will not happen. We can hope that by adding rows "greedily", i.e. only according as we need them to cover the monomials determined by columns of $M$ which we have already occupied, then we might find a sub-determinant of $M$ which is already good enough.

4 Description of the Algorithm

Input: A system of generic polynomials (1.1)

Output: A matrix $M$, whose determinant has the Newton resultant as a factor.

Parameters: $n =$ dimension; $m$ bounds $|A_i|$; $\overline{m}$ bounds the number of extreme points of $N_i = \text{conv}(A_i)$; $h =$ the number points of the Minkowski sum $N = N_1 \oplus \ldots \oplus N_{n+1}$, which is $O(m^{n+1})$.

Step 1.

Compute the convex hulls $N_i = \text{conv}(A_i)$ of the exponent vectors of the inputs (1.1). Assemble the vertices of $N_i$ as the columns of a matrix,

$$
A^{(i)} = \begin{bmatrix}
q_{i11} & \cdots & q_{i1m} \\
\vdots & & \vdots \\
q_{in1} & \cdots & q_{inm}
\end{bmatrix}
$$

We approximate the convex hull by sorting the given lattice points in a linear number of pseudo-random directions, and then run LP's to determine which of the remaining points are not hull vertices. Any such point $p$ will be feasible for
the LP with convexity constraints

\[ \sum_{j \in C} \lambda_j v_j = p, \]

\[ \sum_{j \in C} \lambda_j = 1, \]

\[ \lambda_j \geq 0, \quad j \in C. \]

where \( C \) is the current set of candidate hull vertices.

Suppose we use \( k \) random directions. The combinatorial complexity of approximating the hull is \( O(knm \log_2(m)) \). The remaining LP’s run in \( O(md^{3.5}) \) steps using interior point methods. The initial approximation appears as a heuristic improvement.

**Step 2.**

Compute a collection of pseudo-random vectors which will be used to lift the points of each polytope, \( L = [L_1, \ldots, L_{n+1}] \). The cost here is \( O(n^2) \).

**Step 3.**

Set up LP equations defining the lower envelope of the Minkowski sum of the lifted points as follows. A general point of of \( N_i \) has the form

\[ v^{(i)} = A^{(i)} \cdot [\lambda_{i1}, \ldots, \lambda_{im}]^T, \]

where the \( \lambda_i \) satisfy a convexity constraint

\[ \lambda_i = \lambda_{i1} + \ldots + \lambda_{im} = 1. \]

A general point of \( N = N_1 \oplus \ldots \oplus N_{n+1} \) has the form

\[ v = v^{(1)} + \ldots + v^{(n+1)}. \]

The objective function is determined by the random lifting vectors. The lower envelope \( \hat{\Lambda} \) consists of those points \( \hat{v} = (v, v_{n+1}) \) whose height \( v_{n+1} \) over the point \( v \in N \) is minimal. Therefore, the objective function is defined to be

\[ B = L_1 \cdot v^{(1)} + \ldots + L_{n+1} \cdot v^{(n+1)}. \]

The cost here is \( O(nm^2) \).

**Step 4**

Compute the Minkowski sum \( N \), and the center of gravity \( g \) of \( N \); \( g \) is used as an initial monomial for the construction of \( M \). Then \( enqueue(V, g) \), i.e. enqueue \( g \) on the work queue \( V \).

The cost here is \( O(h) \).
Step 5.

Loop:
If V is empty then break (goto 7).
Set $v_0 := dequeue(V)$. 
Check which polytope points contribute to the optimum sum for $v_0$. This is done by running the LP:

\[
\begin{align*}
v &= v_0, \\
\lambda_i &= 1, \quad 1 \leq i \leq n + 1 \\
\lambda_{ij} &\geq 0 \\
\min(B)
\end{align*}
\]

The vertices appearing in the representation of $v$ as a sum points from the $N_j$ can be read off from the coefficients which equal 1 in the representation $v = v_0$. The cost here is $O((mn)^{3.5})$ since there are $mn$ variables $\lambda_{ij}$.

Step 6.

Compute the next row of the Newton matrix $M$:

step 6.1 Find the largest index $i$ of a vertex $q_{ij}$ contributing to the representation of $v_0$ from step 5.

step 6.2 Compute the displaced lattice vector $v' := v - q_{ij}$.

step 6.3 Compute the product $x^{v'} \cdot f_i$, placing the indeterminate coefficients in the appropriate columns of $M$.

step 6.4 For each column $j$ of $M$ init by a monomial from step 6.3, if the lattice point $w_j$ labeling column $j$ is not already processed or enqueued, then enqueue($V, w_j$).

step 6.5 GOTO 5 (Loop)

The cost here is linear in $n$ and $m$.

Step 7. Compute $\det(M)$.

The cost here is $O(h^{\alpha+1+\epsilon} \log_2(h))$, using Berkowitz's algorithm [B], where $\alpha$ is the current best exponent for matrix multiplication, and $\epsilon > 0$.

The overall cost is dominated by the cost of the determinant. Apart from this, the dominant component of the worst case complexity comes from the LP's occurring in the loop at step 5. The loop may execute as many as $h$ times, with each loop costing as much as $O((mn)^{3.5})$ steps for the LP.
5 Appendix A

Maple code for the Newton matrix algorithm. (Warning: there are forward references.)

```maple
with(linalg):
with(simplex):

# verts -- calculates array of exponents of monomials in poly
verts := proc(poly, vars)

local d, n, vert;

d := nops(vars);
n := nops(poly);
vert := array(1..d, 1..n);
for i to n do
    term := op(i, poly);
    for j to d do
        vert[j, i] := degree(term, vars[j])
    od;
    od;
    vert
end: # verts

# randlin -- random linear functionals
randlin := proc(d)
```

local i,j,l,die;

    die := rand(1..1000);
    l := array(1..d);
    for i to d do
        l[i] := array(1..d-1);
        for j to d-1 do l[i][j] := die() od
    od;
    l
end: # randlin

# randvect -- random displacement vector of length n

randvect := proc(n)
local rproc;

    rproc := rand(1..1000);
    r := array(1..n);
    for i to n do r[i] := convert(rproc()/10000, float) od;
    r
end: # randvect

# make_eqns -- sets up LP for lower envelope

make_eqns := proc(A, l, d)
# A = array of matrices, each contains vertices as columns.
# l = random linear functional
# d = length of A = dimension - 1;
#
# Constructs the following input equations for LP package:
#  objective = \sum_i l_i (\sum_j \lambda_{i,j} A_{i,j}),
#  vertex eqns lhs = \sum_{ij} \lambda_{ij} A_{ij},
#  conv. constr. = \sum \lambda_{i,j} = 1;
# where
#  l_i = random linear functional,
#  \lambda_{i,j} = LP variables,

local C, cd, conveq, convsum, i, ipoly, j;
local lambda, nverts, objective, vertlhs;

objective := 0; # objective function
vertlhs := vector(d-1, 0); # lhs vertex equations
conveq := {}; # convexity constraints
nverts := dotp(vector(d,1), map(coldim, A)); # tot. no. vert.

for i to d do
  ipoly := A[i];
  convsum := 0;
  cd := coldim(ipoly);
  lambda := array(1..cd); # array for new variables

  for j to cd do
    lambda[j] := 'lam'.i.'x'.j;
    convsum := convsum + lambda[j];
  od;
  conveq := conveq union {convsum = 1};
C := multiply(ipoly, lambda);
vertlhs := add(vertlhs, C);
objective := objective + multiply(transpose(l[i]), C);
od;
[objective, conveq, vertlhs, nverts]
end: # make_eqns

# cog -- computes center of gravity


cog := proc(ipoly, d)
# ipoly = array of exponent columns.

local cd, i, v;

    cd := coldim(ipoly);
    v := multiply(ipoly, convert(vector(cd, 1), matrix));
    scalarmul(v, 1/cd)
end: # cog

# Table of digit to num conversions

chars := 0:
chars[‘0’] := 0:
chars[‘1’] := 1:
chars[‘2’] := 2:
chars[‘3’] := 3:
chars[‘4’] := 4:
chars[‘5’] := 5:
chars[‘6’] := 6:
chars[‘7’] := 7:
chars[‘8’] := 8:
chars[‘9’] := 9:

# recode -- converts string ‘lamMxN’ into vector [M, N]

recode := proc(z)
# converts string ‘lamMxN’ into vector [M, N]
local i, L, M, N, s, state;

L := length(z);
M := 0;
N := 0;
state := 1;
for i from 4 to L do
    s := substring(z,i..i);
    if s = ‘x‘ then state := 2 fi;
    if state = 1 then M := 10*M + chars[s]
    elif state = 2 then state := 3
    elif state = 3 then N := 10*N + chars[s] fi
od;
[M, N]
end:

# opt_verts -- checks which polytope points contribute
# to optimum sum for vert.
opt_verts := proc(edata, vert, d)
  # edata = equations from make_eqns(),
  # vert  = vertex in Minkowski sum,
  # d     = dimension + 1.

  local assigns, conveq, eps, eqns, ivert, j, nverts, objective,
         optverts, verteql, vertlhs;

  # Pick apart input from make_eqns()
  objective := op(1, edata); # objective function
  conveq := op(2, edata); # convexity constraints
  vertlhs := op(3, edata); # vertex eqns left hand sides
  nverts := op(4, edata); # total number of input vertices

  # Run LP to find optimal sum (on lower envelope) for vert
  eps := 1.0 * 10^(-6);
  eqns := conveq;
  for j to d-1 do
    eqns := eqns union {vertlhs[j] = vert[j,1]} od;
  assigns := minimize(objective, eqns, NONNEGATIVE);

  # find vertices appearing in the representation
  optverts := {};
  for a in assigns do
    if (abs(rhs(a)-1.0) < eps) then
      optverts := optverts union {recode(lhs(a))};
    fi
  od;
  optverts
end: # opt_verts
# compute_rows -- greedy algorithm for rows of Newton matrix

```plaintext
compute_rows := proc(A, B, edata, v, delta, d)
  local cols, guide, i, imax, ipoly, jmax, overt, rows, vt1, vt2;

  rows := {}; guide := {}; cols := {eval(v)};
  while not (eval(v) = {}) do  # i.e. cont. while v is defined
    print(v);
    rows := rows union {eval(v)};
    # lower hull vertices:
    overt := opt_verts(edata, add(v,delta), d);
    imax := 0;
    jmax := 0;
    for vt1 in overt do    # find index of last polytope
      if vt1[1] > imax then  # ..contrib. vert. to opt. sum
        imax := vt1[1];
        jmax := vt1[2]
      fi
    od;
  od;
```

guide := guide union {eval(v), imax, jmax};
v_{t2} := \text{add}(v, \text{col}(A[imax], jmax), 1, -1);
ipoly := B[imax];
for i to \text{coldim}(ipoly) do 
  \text{vt1} := \text{add}(\text{vt2}, \text{col}(ipoly, i)); \# \text{mult. of } B[imax]
  \text{if not } \text{vmem}(\text{vt1}, \text{cols}) \text{ then}
    \text{cols} := \text{cols union } \{\text{eval}(\text{vt1})\} \text{ fi}
  od;
v := \{}; \# \text{undefine } v, \text{ and then check}
for vt1 in cols do 
  \text{if not } \text{vmem}(\text{vt1}, \text{rows}) \text{ then } v := \text{eval}(\text{vt1}); \text{ break fi}
  od
od;
guide
end: \# \text{compute rows}

####################################################################################################################
# \text{newtonres} -- computes Newton resultant matrix
# \text{-----------------------------------------------}

\text{newtonres} := \text{proc}(A, B)
# A, B = array of arrays, ((d-1) X no. vert.), cols are vert.
# A = vertices only,
# B = all exponents in Newton polytopes

local d, delta, edata, i, ip, ipoly;
local ivert, j, k, M, msize, v, vt1, vt2;

d := \text{rowdim}(\text{convert}(A, \text{matrix})); \# no. polytopes = dim. - 1
delta := \text{randvect}(d-1); \# \text{rand. displacement vector}
l := \text{randlin}(d); \# \text{rand. linear functional}
print('Computing data common to all vertex equations');
edata := make_eqns(A, l, d);  # [obj, conveq, vertlhs]

print('greedy search for row indices');
v := cog(A[1], d);  # start at the ctr. grav.
for i from 2 to d do  # ..of Minkowski sum
  v := add(v, cog(A[i], d)) od;
v := map(round, v);
guide := compute_rows(A, B, edata, v, delta, d);

print('constructing the matrix');
msize := nops(guide);
guide := convert(guide, list);
M := array(1..msize, 1..msize);

for i to msize do
  ivert := guide[i];
  ip := ivert[2];
  vt1 := add(ivert[1], col(A[ip], ivert[3]), 1, -1);
  ipoly := B[ip];
  for j to msize do
    vt2 := add(guide[j][1], vt1, 1, -1);
    M[i, j] := 0;
    for k to coldim(ipoly) do
      if equal(vt2, submatrix(ipoly, 1..d-1, [k])) then
        M[i, j] := 'C'.ip.'x'.k
      fi
    od
  od
od;
M
end: # mixed

# randhull -- approx. convex hull, sorting in random directions

randhull := proc(v,limit)
# v = input vector of vertices
# limit = number of random directions to sort by

local d, i, j, k, max, maxindex, min, minindex, n, newc, rv, t;
local processed, unprocessed, vertlhs, conveq, convsum, lambda;
local work, work1, possible, sub, temp;

    d := rowdim(v); # v = input vector of vertices, d = dim.
n := coldim(v); # n = numbers of vertices

    processed := {};
    unprocessed := {};
    for i to n do unprocessed := unprocessed union {i} od;

    for i to limit do
        rv := randmatrix(1,d);
        rv := scalarmul(rv, 1/norm(rv));

        minindex := 1;
        maxindex := 1;
        min := multiply(rv, col(v, 1));
        min := min[1];
        max := eval(min);
for j from 2 to n do
     t := multiply(rv, col(v, j));
     t := t[1];
     if t < min then
         minindex := j;
         min := t
     elif t > max then
         maxindex := j;
         max := t
     fi
od;

processed := processed union {minindex, maxindex};
unprocessed := unprocessed minus {minindex, maxindex}

od;  # ends random iterations

# Now use LP to check other vertices
lambda := matrix(1, n);
for i to n do lambda[1, i] := 'lam'.i od;
work1 := unprocessed;
possible := processed;
work := {};

while (work minus work1 <> {}) or (work1 minus work <> {}) do
    work := work1;

    for i in work do
        temp := possible minus {i};
        sub := convert(temp, list);
        vertlhs := vector(d, 0); # vertex equation lhs
        conveq := {}; # convexity constraint
convsum := 0;
for j in temp do
    convsum := convsum + lambda[i, j]
od;
conveq := conveq union {convsum = 1};
vertlhs := multiply(submatrix(v, 1..d, sub),
    transpose(submatrix(lambda, 1..1, sub)));
for j to d do
    conveq := conveq union {vertlhs[j, 1] = v[j, i]}
od;
if feasible(conveq, NONNEGATIVE) then
    print(col(v,i), 'interior');
    work1 := work1 minus {i};
    possible := possible minus {i}
else
    possible := possible union {i}
fi
od
od;

possible;
end: # randhull

******************************************************************************
# dotp -- dot product of vectors
******************************************************************************
dotp := proc(x, y)
    multiply(transpose(x), y)
end: # dotp
### norm -- computes the 2 norm of a vector (as 1xn matrix)

```plaintext
norm := proc(x)
local i, r;

r := x[1,1]^2;
for i from 2 to coldim(x) do r := r + x[1,i]^2 od;
convert(sqrt(r), float)
end: # norm
```

### vmem -- checks if el lies in (set or list) vset, using equal

```plaintext
vmem := proc(el, vset)
local found, el2;

found := false;
for el2 in vset do
    if equal(el, el2) then found := true; break fi od;
found
end: # vmem
```

### set2array -- convert set of row vectors to array of col vectors

```plaintext
```
set2array := proc(s)
    n := nops(s);
    d := nops(convert(op(1, s), list));
    transpose(array(1..n, 1..d, convert(s, list)));
end: # set2array
6 References


