CHAPTER 4

Computer Codes

Our purpose in this chapter is to offer two complete programs which will help considerably in exploring the methodology of our work. In sec. 4.1.1 we have given the pseudo code of the computer program for the weighted unconstrained case, whose Pascal program appears in sec. 4.1.2 whereas sec. 4.1.3 describes the program in Pascal language in the constrained equiweighted case.

4.1.1 Pseudo code of the weighted minimax problem

We now present the pseudo code of our algorithm of the MinimumSpanningDiamond in the unconstrained case. We assume the availability of a subroutine InitialStep having as its input the coordinates of the demand points belonging to the set S together with the weights associated with them. This procedure outputs a point P whose weighted rectilinear distances from A, B $\in$ S are equal. We further assume the existence of another subroutine Stretch which calculates the set of alternative optimal solutions constituting a portion of an edge of EP(A,B). ALGORITHM MSD processes the set S of demand points and yields the minimum spanning diamond of S. UNDONE, a Boolean variable, being initially set to TRUE, becomes FALSE with the attainment of the solution in which case the algorithm terminates.

Algorithm MSD

Let Γ be the portion of $EP(A,B)$ extending from P to $\partial R(A,B)$

Call InitialStep

UNDONE = TRUE

WHILE UNDONE DO

  IF $P \in R(A,B)$ THEN UNDONE = FALSE

  ELSE

   $S1 = S \setminus \{A,B\}$

   $S2 = \{C \in S1 |$ the weighted distance from $P1 \in \Gamma$, different from P, is the same as that of A or B where PP1 is least$\}$

   IF $|S2| = 0$ THEN

     $P = EP(A,B) \cap \partial R(A,B)$

     UNDONE = FALSE

   ELSE

    IF $|S2| = 1$ THEN

      IF $P1 \in R(A,B)$ or $R(B,C)$ or $R(A,C)$ THEN UNDONE = FALSE

      ELSE drop A or B and assign C to the point dropped

    ENDIF

   ELSE

    select two points from $S2 \cup \{A,B\}$, rename them A,B

    and $P \longleftarrow P1$

  ENDIF

ENDWHILE

Call Stretch

output the minimum spanning diamond

END

4.1.2 Pascal program for the weighted unconstrained location problem

```
Program Minimum_spanning_diamond;
uses  crt,dos;
type list=array[1..n] of real;
 var infile,outfile:text;
     x,y,w:list;
     i,i1,i2,i3,l1,m1,l2,m2,ic:integer;
     d,ds,x1,x2,y1,y2,h,k,u,v,dif,lambda,w1:real;
     flag,aflag,bflag:boolean;
     hh,mm,ss,hs:word;
 Procedure Maxdist(var ii:integer);
 {Proc to find point i1 at a max wtd rect dist from (x2,y1)}
  var d,maxd:real;
  begin
   maxd:=-maxint;
   for i:=1 to n do
    begin
     d:=w[i]*(x2-x[i]+y[i]-y1);
     if maxd<d then
      begin
      maxd:=d;
       ii:=i
      end
    end
  end;                              {End Maxdist}
 Procedure Diff(var df:real; j1,j:integer; c,d:real);
 {This proc calculates the difference of rect distances of}
 {2 points j1,j from a given point (c,d)}
 var f1,f2:real;
 begin
  f1:=w[j1]*(abs(x[j1]-c)+abs(y[j1]-d));
  f2:=w[j]*(abs(x[j]-c)+abs(y[j]-d));
  df:=f1-f2
 end;                               {End Diff}
 Procedure Product(a,b,c,d:real);
 {This proc finds the sign of the product of rect dist of}
 {points i1,i2 from the points (a,b) and (c,d)}
  var dif1,dif2:real;
  begin
   bflag:=true;
   diff(dif1,i1,i2,a,b);
   diff(dif2,i1,i2,c,d);
   if dif1*dif2 <0 then bflag:=false
  end;                              {End Product}
 Procedure Quadrant(var p,q:integer; a,b,c,d:real);
 {Proc to find the quadrant in which (b,d) lies w.r.t.(a,c)}
  begin
   p:=0;q:=0;
   if a>b then p:=-1;
   if a<b then p:=1;
   if c>d then q:=-1;
   if c<d then q:=1
  end;                              {End Quadrant}
```

```
Procedure Min_max(var min,max:real; r,s:real);
{Proc to find the max and min of 2 real numbers r and s}
 begin
  min:=r;
  if min>s then
   begin
     max:=min;
     min:=s
   end
  else max:=s
 end;                              {End Min_max}
Procedure Rectangle(a,b,f1,g1,f2,g2:real; var fl:boolean);
{Proc to test if (a,b) is within or on the rectangle with}
{(f1,g1),(f2,g2) as opposite vertices}
 begin
  fl:=true;
  min_max(x1,x2,f1,f2);
  min_max(y1,y2,g1,g2);
  if ((a)=x1) and (a(=x2)) and ((b)=y1) and (b(=y2))
   then fl:=false
 end;                              {End Rectangle}
Procedure Select;
{This proc selects 2 points needed for the next iteration}
 begin
  Quadrant(l1,m1,x[i1],h,y[i1],k);
  Quadrant(l2,m2,x[i2],h,y[i2],k);
  if (l1=l2) and (m1=m2) then    {I,III,VII,IX w.r.t. i1,i2}
   if w[i1]<w[i2] then i2:=i3 else i1:=i3
  else                            {II,IV,VI,VIII w.r.t. i1,i2}
   begin
    Quadrant(l2,m2,x[i3],h,y[i3],k);
    if (l1=l2) and (m1=m2)       {I,III,VII,IX w.r.t. i1,i3}
     then i1:=i3 else i2:=i3
   end
 end;                              {End select}
Procedure Interchange(var j1,j2:integer);
{This is a proc for swapping 2 integers}
 var t:integer;
 begin
  t:=j1; j1:=j2; j2:=t
 end;                              {End Interchange}
Procedure Point(var k1:real;k2:real;p1,p2,q1,q2:integer);
{Proc to find the point of intersection of the equipolygon}
{and the boundary of the rectangle}
 var a:real;
 begin
  a:=w[i1]*(l1*x[i1]+m1*y[i1])-w[i2]*(l2*x[i2]+m2*y[i2]);
  k1:=(a-k2*(w[i1]*p1-w[i2]*p2))/(w[i1]*q1-w[i2]*q2);
 end;                              {End Point}
Procedure Boundary;
var l,m:integer;
{Proc to obtain the point of intersection of the edge of }
{the equipolygon and the boundary of the rectangle moving}
{along the direction of descent}
 begin
```

```
min_max(x1,x2,x[i1],x[i2]);
min_max(y1,y2,y[i1],y[i2]);
Quadrant(l1,m1,x[i1],h,y[i1],k);
Quadrant(l2,m2,x[i2],h,y[i2],k);
l:=l1+l2; m:=m1+m2; u:=maxint; v:=maxint;
if (l=0) or (m=0) then  {zone IV,VI or VIII,II}
 if (l=0) then      {zone IV or VI}
  begin
   if (m=-2) then v:=y1 else v:=y2;
   product(x1,v,x2,v);
   if bflag=true then
    begin
     v:=maxint;
     if w[i1]>w[i2] then u:=x[i2] else u:=x[i1]
    end
  end
 else  {m=0}
  begin             {zone VIII or II}
   if (l=2) then u:=x2 else u:=x1;
   product(u,y1,u,y2);
   if bflag=true then
    begin
     u:=maxint;
     if w[i1]>w[i2] then v:=y[i2] else v:=y[i1]
    end
  end;
if (l1=l2) and (m1=m2) then {zone VII or I or III or IX}
 if (m=-2) then                {zone VII or I}
  begin
   v:=y1; product(x1,v,x2,v);
   if bflag=false then
    begin
     v:=maxint;
     if (l=2) then u:=x2 else u:=x1
    end
  end
 else          {m=2}
  begin        {zone III or IX}
   v:=y2; product(x1,v,x2,v);
   if bflag=false then
    begin
     v:=maxint;
     if (l=2) then u:=x2 else u:=x1
    end
  end;
if (l1=0) or (m1=0) or (l2=0) or (m2=0) then
{Lines of separation of all zones in pairs except V}
 begin
  if (m=2) or (m=-2) then
   begin
    if (m=2) then v:=y2 else v:=y1;
    product(x1,v,x2,v);
    if bflag=false then
     if (l1<>0) then l2:=-l1 else l1:=-l2
    else
```

```
        if (l1<>0) then l2:=l1 else l1:=l2
       end;
      if (l=2) or (l=-2) then
       begin
        if (l=2) then u:=x2 else u:=x1;
        product(u,y1,u,y2);
        if bflag=false then
         if (m1<>0) then m2:=-m1 else m1:=-m2
        else
         if (m1<>0) then m2:=m1 else m1:=m2
       end
     end;
   if (u=maxint) then point(u,v,m1,m2,l1,l2)
                 else point(v,u,l1,l2,m1,m2)
 end;                          {End Boundary}
Procedure Test(a1,b1,a2,b2:real);
{Proc to find euclidean dist of (h,k) lying on a given }
{equipolygon from the point of intersection of this and}
{another equipolygon}
 var xm,ym,z:real;
 begin
  xm:=(a1+a2)/2; ym:=(b1+b2)/2;
  Quadrant(l2,m2,x[i],xm,y[i],ym);
  z:=w[i1]*(l1*(x[i1]-h)+m1*(y[i1]-k))-w[i]*(l2*(x[i]-h)+
     m2*(y[i]-k));
  z:=z/(w[i1]*(l1*(u-h)+m1*(v-k))-w[i]*(l2*(u-h)+m2*(v-k)));
  if (z>0) then
   begin
    lambda:=z;
    i3:=i
   end
 end;                          {End Test}
Procedure Convex(var lambda1:real; a,b,c:real);
{Proc to test if a real number lies in a given interval}
 begin
  if (b<>a) then
   begin
    lambda1:=(c-a)/(b-a);
    if (lambda1<0) or (lambda1>1) then lambda1:=maxint
   end
  else lambda1:=maxint
 end;                          {End Convex}
Procedure Update;
{Proc to update a given point (h,k) for the next iteration}
 var xm,ym,r,s:real;
 Procedure value(var f:real; lamda,a,b:real);
  begin
   f:=lamda*b+(1-lamda)*a
  end;
 begin
  lambda:=maxint; i3:=maxint;
  xm:=(u+h)/2;ym:=(v+k)/2;
  quadrant(l1,m1,x[i1],xm,y[i1],ym);
  for i:=1 to n do
   begin
```

```
if (i<>i1) and (i<>i2) then
 begin
  diff(dif,i1,i,u,v);
  if (dif<=0 ) then
   begin
    convex(r,h,u,x[i]);
    convex(s,k,v,y[i]);
    if (r<>maxint) and (s<>maxint) then
     begin
      if (r<s) then
       begin
        value(ym,r,k,v);
        diff(dif,i1,i,x[i],ym);
        if (dif<=0) then test(h,k,x[i],ym)
        else          {dif>0}
         begin
          value(xm,s,h,u);
          diff(dif,i1,i,xm,y[i]);
          if (dif<=0) then test(x[i],ym,xm,y[i])
          else       {dif>0}
           begin
            diff(dif,i1,i,u,v);
            if (dif<=0) then test(xm,y[i],u,v)
           end
         end
       end
      else            {r>=s}
       begin
        value(xm,s,h,u);
        diff(dif,i1,i,xm,y[i]);
        if (dif<=0) then test(h,k,xm,y[i])
        else  {dif>0}
         begin
          value(ym,r,k,v);
          diff(dif,i1,i,x[i],ym);
          if (dif<=0) then test(xm,y[i],x[i],ym)
          else       {dif>0}
           begin
            diff(dif,i1,i,u,v);
            if (dif<=0) then test(x[i],ym,u,v)
           end
         end
       end
     end
    else
     if (r<>maxint) then
      begin
       value(ym,r,k,v);
       diff(dif,i1,i,x[i],ym);
       if (dif<=0) then test(h,k,x[i],ym)
       else          {dif>0}
        begin
         diff(dif,i1,i,u,v);
         if (dif<=0) then test(x[i],ym,u,v)
        end
```

```
            end
          else
           if (s<>maxint) then
            begin
             value(xm,s,h,u);
             diff(dif,i1,i,xm,y[i]);
             if (dif<=0) then test(h,k,xm,y[i])
             else           {dif>0}
              begin
               diff(dif,i1,i,u,v);
               if (dif<=0) then test(xm,y[i],u,v)
              end
            end
          else test(h,k,u,v);
          u:=h*(1-lambda)+u*lambda;
          v:=k*(1-lambda)+v*lambda
         end    {dif<=0}
       end      {i<>i1,i2}
    end;                        {end of for}
   h:=u; k:=v
  end;                           {End Update}
Procedure Initial_step;
{Proc to find (h,k) equidistant from at least 2 points}
 var d1,d11:real;
 begin
  x2:=x[1]; y1:=y[1];
  for i:=2 to n do
   begin
    if x2<x[i] then x2:=x[i];
    if y1>y[i] then y1:=y[i]
   end;
  Maxdist(i1);
  i2:=0;k:=y1;h:=x2;u:=x[i1];v:=y1;
  update;
  if i3=maxint then
   begin
    h:=x[i1];u:=x[i1];v:=y[i1];
    update;
    k:=v
   end
  else h:=u;
  i2:=i3
 end;                           {End Initial_step}
Procedure Stretch;
{Proc to find the line segment giving the optimal soln set}
 var l,m:integer;
     f1,f2,f3,f4:real;
 Procedure One(b,c:real;p1,p2,q1,q2,s:integer);
  begin
   point(b,c,p1,p2,q1,q2);
   if s=2 then diff(dif,i1,i3,c,b)
          else diff(dif,i1,i3,b,c);
   if dif>0 then
    begin
     if s=2 then begin u:=c; v:=b end
```

```
            else begin u:=b; v:=c end;
     update
    end
   else u:=maxint
  end;
begin
 Min_max(x1,x2,x[i1],x[i2]);
 Min_max(y1,y2,y[i1],y[i2]);
 Quadrant(l1,m1,x[i1],h,y[i1],k);
 Quadrant(l2,m2,x[i2],h,y[i2],k);
 l:=l1+l2;m:=m1+m2;u:=maxint;
 diff(f1,i1,i2,x1,y1); diff(f2,i1,i2,x2,y1);
 diff(f3,i1,i2,x2,y2); diff(f4,i1,i2,x1,y2);
 if (l=0) and (m=0) then      {(h,k) inside the rectangle}
  begin
   if (f1*f2<0) then one(u,y1,m1,m2,l1,l2,1);
   if (f2*f3<0) and (u=maxint) then
            one(v,x2,l1,l2,m1,m2,2);
   if (f3*f4<0) and (u=maxint) then
            one(u,y2,m1,m2,l1,l2,3);
   if u=maxint then
         begin
          u:=x1; point(v,u,l1,l2,m1,m2)
         end
  end                          {(h,k) on boundary of rectangle}
 else
  begin
   if (m=-1) then
    begin
     if (f2*f3<0) then u:=x2
     else
      if (f3*f4<0) then v:=y2 else u:=x1
    end
   else
    if (l=1) then
     begin
     if (f1*f2<0) then v:=y1
     else
      if (f3*f4<0) then v:=y2 else u:=x1
    end
   else
    if (m=1) then
     begin
     if (f1*f2<0) then v:=y1
     else
      if (f2*f3<0) then u:=x2 else u:=x1
    end
   else
    begin
     if (f1*f2<0) then v:=y1
     else
      if (f2*f3<0) then u:=x2 else v:=y2
    end;
   if u=maxint then point(u,v,m1,m2,l1,l2)
               else point(v,u,l1,l2,m1,m2);
```

```
      update
     end
   end;                              {End Stretch}
begin                               {main action statements}
  clrscr;
  flag:=true;
  assign(infile,'file.dat');
  reset(infile);
  assign(outfile,'out.put');
  append(outfile);
  writeln('supply the number of data points');
  readln(n);
  for i:=1 to n do
   readln(infile,x[i],y[i],w[i]);
  gettime(hh,mm,ss,hs);
  write(outfile,hh,':',mm,':',ss,'.',hs:2);
  initial_step;
  rectangle(h,k,x[i1],y[i1],x[i2],y[i2],aflag);
  if aflag=false then flag:=false else Boundary;
  ic:=0;
  while flag do
   begin
    update;
    if (i3=maxint) then flag:=false
    else
     begin
      rectangle(h,k,x[i1],y[i1],x[i3],y[i3],aflag);
      if aflag=false then
       begin
        flag:=false;
        interchange(i2,i3)
       end
      else
       begin
        rectangle(h,k,x[i2],y[i2],x[i3],y[i3],aflag);
        if aflag=false then
         begin
          flag:=false;
          interchange(i1,i3)
         end
        else select
       end
     end;                          {end i3<>maxint}
    Boundary;
    ic:=ic+1
   end;                            {end while}
  writeln('total no. of iterations=',ic);
  write('The stretch extends from (',h:5:2,',',k:5:2);
  Stretch;
  writeln(') to (',u:5:2,',',v:5:2,')');
  gettime(hh,mm,ss,hs);
  write(outfile,'       ',hh,':',mm,':',ss,'.',hs:2);
  writeln(outfile,'              ',ic);
  close(outfile);
  close(infile);
end.
```

4.1.3 Pascal program of the equiweighted constrained minimax location problem

```
program Constrained;
uses   crt,dos;
const inf = 1.0E+35;
type   list  =array[1..100] of real;
       list1 =array[1..500] of real;
       row    =array[1..4] of real;
       col    =array[1..2] of real;
       index =set of 1..100;
var    infil,outfil:text;
       a,b,c,s,a1,b1,c1,x1,y1:list;
       x,y:list1;
       ax,ay,cc,dd,m1:row;
       z:col;
       s1,s2:index;
       m,n,i,i1,i2,i3,j,jj,l1,l2,p,choice,count:integer;
       u1,u2,v1,v2,h1,k1,h2,k2,h,k,x0,y0,c11,m11:real;
       m12,alpha:real;
       flag,done:boolean;
 procedure interchange(var d1,e1,f1,d2,e2,f2:real);
 { Swaps two sets of variables}
 var t1,t2,t3:real;
 begin
   t1:=d1;d1:=d2;d2:=t1;
   t2:=e1;e1:=e2;e2:=t2;
   t3:=f1;f1:=f2;f2:=t3
 end;                                  { End interchange }
 procedure Diamond(var x1,x2,y1,y2:real);
 {Finds the stretch KQ in the unconstrained case}
 var u,v:real;
  function FindMax(t1,t2:list1;p,q:integer):real;
  {Constructs the smallest rectangle containing all demand
   points having sides parallel to x+y=0, -x+y=0}
  var  z,max:real;
  begin
   for i:=1 to n do
    begin
     z:= p*t1[i] + q*t2[i];
       if (i=1) or (z>max) then max:=z;
    end;
   FindMax := max
  end;
 begin
  cc[2]:= FindMax(x,y,1,1);
  cc[1]:= - FindMax(x,y,-1,-1);
  cc[4]:= FindMax(x,y,-1,1);
  cc[3]:= - FindMax(x,y,1,-1);
  {Finding coordinates of K(x1,y1) and Q(x2,y2)}
  u:=(cc[2]-cc[1]);  v:=(cc[4]-cc[3]);
  x1:= (cc[1]-cc[3])/2;  x2:= (cc[2]-cc[4])/2;
  if u >= v then
   begin
```

```
    y1 := (cc[2]+cc[3])/2; y2 := (cc[1]+cc[4])/2
   end
  else
   begin
    y1 := (cc[1]+cc[4])/2; y2 := (cc[2]+cc[3])/2
   end
end;                                  { End Diamond }
Procedure FindSegment(var f1,g1,f2,g2:real;
                          var found:boolean);
{Finds the optimal solution set in case KQ has a non-void
 intersection with the constrained region R}
var  d,e,ub,lb,vmax,vmin:real;
begin
 found := true;
 ub:=1; lb:=0; i :=1;
 while (i<=m) and found do
  begin
   d:=a1[i]*(u1-u2) + b1[i]*(v1-v2);
   e:=c1[i] - a1[i]*u2 - b1[i]*v2;
   if d>0 then
    begin
     vmax:= e/d;
     if ub > vmax then ub := vmax
    end
   else
    if d<0 then
     begin
      vmin:= e/d;
      if lb < vmin then lb := vmin
     end
   else
    if e<0 then found := false;
   i := i + 1
  end;
 if (ub < lb) then found := false
 else
  begin
   f1 := ub*u1 + (1-ub)*u2;
   g1 := ub*v1 + (1-ub)*v2;
   f2 := lb*u1 + (1-lb)*u2;
   g2 := lb*v1 + (1-lb)*v2
  end
end;                                  { End FindSegment }
procedure FindDominate(var u,v:real; var j1,j2:integer);
{Finds dominating side(s) at a point}
var  d:row; max:real;
begin
 j1 := 0; j2 := 0;
 d[1]:=abs(u+v-cc[1]);  d[2]:=abs(u+v-cc[2]);
 d[3]:=abs(-u+v-cc[3]); d[4]:=abs(-u+v-cc[4]);
 max := 0;
 for i:= 1 to 4 do
  begin
   if d[i]>max then
```

```
     begin
      max := d[i];
      j1 := i
     end
    else
     if d[i]=max then j2 := i
   end
end;                                  { End FindDominate }
procedure Gauss(var a1,b1,c1,a2,b2,c2,u,v:real);
{Finds the point of intersection of two linear equations}
var k1:real;
begin
 if a1<>0 then
  begin
   k1:=a2/a1;
   if (b2-k1*b1 <>0) then
    begin
     v:=(c2-k1*c1)/(b2-k1*b1);
     u:=(c1-v*b1)/a1
    end
   else writeln('the solution does not exist')
  end
 else
  begin
   v:=c1/b1;
   if a2<>0 then u:=(c2-v*b2)/a2 else writeln('parallel')
  end
end;                                  { End Gauss }
procedure counterclock;
{Arranges the constraints to form the convex polyhedron}
var j1,j2:integer;
 procedure partition;
 {Partitions the constraints in sets s1,s2
  according as b[i] < or > 0}
 begin
  s1 := []; s2 := [];
  p := 0;
  for i:= 1 to m do
   begin
    if (b[i] < 0) or ((b[i] = 0) and (a[i] < 0)) then
     begin
      p := p + 1;
      s1 := s1 + [i]
     end
    else s2 := s2 + [i]
   end
 end;
 procedure sort(i1:integer);
 {Sorts the constraints in ascending order of slopes}
 var t:real;
 begin
  for i := 1 to i1-1 do
   for j := i+1 to i1 do
    if s[i] > s[j] then
```

```
      begin
       interchange(a1[i],b1[i],c1[i],a1[j],b1[j],c1[j]);
       t := s[i]; s[i] := s[j]; s[j] := t
      end
 end;
begin
 partition;
 j1 := 1; j2 := p + 1;
 for i := 1 to m do
  if i in s1 then
   begin
    a1[j1] := a[i];
    b1[j1] := b[i];
    c1[j1] := c[i];
    j1 := j1 + 1
   end
  else
   begin
    a1[j2] := a[i];
    b1[j2] := b[i];
    c1[j2] := c[i];
    j2 := j2 + 1
   end;
 for i := 1 to m do
  if (b1[i]<>0) then s[i] := (-a1[i]/b1[i])
  else
   if (a1[i] < 0) then s[i] := -inf  else s[i] := inf;
 sort(p);
 p := m-p;
 sort(p);
end;                                { End Counterclock }
procedure Vertices(r,s,t:list);
{Obtains the vertices of the polyhedron}
begin
 for i := 1 to m do
  if (i<>m) then Gauss(r[i],s[i],t[i],r[i+1],s[i+1],t[i+1],
       x1[i+1],y1[i+1])
  else Gauss(r[m],s[m],t[m],r[1],s[1],t[1],x1[1],y1[1])
end;                                { End Vertices }
procedure InitialChoice(var f,g:real);
{Assigns starting values to (h,k) }
begin
 f := (x1[1]+x1[2]+x1[3])/3;
 g := (y1[1]+y1[2]+y1[3])/3
end;                                { End InitialChoice }
procedure Convex(c,d:real; var u,lambda:real;
                         var found:boolean);
{Finds if a real number u lies between 2 real numbers c,d}
begin
 found := false;
 if (c<>d) then lambda := (u-d)/(c-d) else lambda := 2;
 if (lambda>=0) and (lambda<=1) then found := true
end;                                { End Convex }
```

```
procedure NextMovement(v:real; l,s:list);
{Finds the direction of movement at a nonoptimal point}
var lamda,zz:real; aflag:boolean;
begin
 i1 := 0; i2 := 0; i3 := 0;
 for i := 1 to m do
  begin
   convex(l[i],l[i+1],v,lamda,aflag);
   if (aflag=true) then
    if (i1=0) then
     begin
      i1 := i; z[1] := lamda*s[i] + (1 - lamda)*s[i+1]
     end
    else              { i1 <> 0 }
     begin
      if (i2=0) then
       begin
        i2 := i; z[2] := lamda*s[i] + (1 - lamda)*s[i+1]
       end
      else            { i1,i2 <> 0 }
       if (i3=0) then
        begin
         i3 := i; zz := lamda*s[i] + (1 - lamda)*s[i+1]
        end
     end;
   if (i2=i1+1) then
    begin
     z[2] := zz; i2 := i3
    end
  end
end;                                { End NextMovement }
procedure DominatingPair;
{Assigns coordinates of a vertex to (x0,y0) when two
 two dominating sides exist}
begin
 if (l1=1) and (l2=3) then choice := 1;
 if (l1=1) and (l2=4) then choice := 2;
 if (l1=2) and (l2=4) then choice := 3;
 if (l1=2) and (l2=3) then choice := 4;
 case choice of
 1 : begin x0:=ax[1]; y0:=ay[1]; count:=1 end;
 2 : begin y0:=ay[2]; x0:=ax[2]; count:=2 end;
 3 : begin x0:=ax[3]; y0:=ay[3]; count:=1 end;
 4 : begin y0:=ay[4]; x0:=ax[4]; count:=2 end
 end
end;                                { End DominatingPair }
procedure SingleSide;
{Assigns coordinates of a vertex to (x0,y0) when two
 one dominating sides exists}
begin
 if (l1=1) or (l1=3) then
  begin
   x0 := ax[1];
   if (l1=1) then y0 := ay[2] else y0 := ay[4];
  end
```

```
 else          { (l1=2) or (l1=4) }
  begin
   x0 := ax[3];
   if (l1=2) then y0 := ay[4] else y0 := ay[2];
  end
end;                                  { End SingleSide }
Procedure Update(var f,g:real);
{Selects one of the two points of intersection of x=h or
 y=k with the boundary}
begin
if abs(f-z[1]) < abs(f-z[2]) then
 begin
  g := z[1];
  i := i1
 end
else
 begin
  g := z[2];
  i := i2
 end
end;                                  { End Update }
procedure InitialStep;
{Reaches an active boundary from an interior point
 maintaining primal feasibility}
var alpha:real; aflag:boolean;
begin
 aflag := true;
 vertices(a1,b1,c1);
 x1[m+1] := x1[1];
 y1[m+1] := y1[1];
 InitialChoice(h,k);
 FindDominate(h,k,l1,l2);
 if (l2=0) then
  begin
   SingleSide;
   NextMovement(k,y1,x1);
   Convex(z[1],z[2],x0,alpha,aflag);
   if (aflag=true) then       {y=k meets x=x0 inside R}
    begin
     h := x0;
     FindDominate(h,k,l1,l2);
    end
   else Update(x0,h)          {y=k meets the boundary}
  end;
 if (l2<>0) then
  begin
   DominatingPair;
   if (count=1) then          {(choice=1) or (choice=3)}
    begin
     NextMovement(h,x1,y1);
     Update(y0,k);
    end
   else                       {(choice=2) or (choice=4)}
```

```
    begin
     NextMovement(k,y1,x1);
     Update(x0,h);
    end
  end
end;                                { End InitialStep }
procedure Replace(t,u:list; f:real);
{Updates boundary if SC is not satisfied at extreme point}
begin
 m11 := s[i];
 if abs(t[i+1]-f) < abs(t[i]-f) then
  begin
   i := i + 1;
   h := t[i]; k := u[i]
  end
 else
  begin
   i := i - 1;
   h := t[i]; k := u[i]
  end;
 m12 := s[i];
 if (m1[1]=1) then
  begin
   if (m1[1]*m11<0) then
    begin if (m11*m12<0) and (m12<1) then done := true end
   else
    if (((m11>1) and (m12<1)) or ((m11<1) and (m12>1)))
     then done := true
  end
 else                               { m1[1]=-1}
  begin
   if (m1[1]*m11<0) then
    begin if (m11*m12<0) and (m12>-1) then done:=true end
   else
    if (((m11<-1) and (m12>-1)) or
    ((m11>-1) and ((m12<-1) or (m12>0)))) then done:=true
  end
end;                                { End Replace }
procedure One(var f,g:real; u:real; t,w:list);
{Course to be taken when s[i]*m1[1]>0 }
var lamda:real; aflag:boolean;
begin
 Convex(t[i],t[i+1],u,lamda,aflag);
 if (aflag=true) then
  begin
   f := u; g := lamda*w[i] + (1-lamda)*w[i+1]; done := true
  end
 else replace(x1,y1,u)
end;                                { End One }
procedure Two;
{Course of action when (h,k) is on x=x0 or y=y0 }
var aflag:boolean;
begin
 if (count=1) then
```

```
  begin
   if (abs(s[i])<1) then done := true  { M is optimal }
   else                                { abs(s[i])>=1 }
    begin
     if (s[i]<0) then y0 := ay[2] else y0 := ay[4];
     Convex(y1[i],y1[i+1],y0,alpha,aflag);
     if (aflag=true) then
      begin
       done := true;
       k := y0; h := alpha*x1[i] + (1-alpha)*x1[i+1]
      end
     else Replace(y1,x1,y0)
    end
  end
 else                                 { (count=2) }
  begin
   if (abs(s[i])>1) then done := true  { N is optimal }
   else                                { abs(s[i])<=1 }
   begin
    if (s[i]<0) then x0 := ax[1] else x0 := ax[3];
    Convex(x1[i],x1[i+1],x0,alpha,aflag);
    if (aflag=true) then
     begin
      done := true;
      h := x0; k := alpha*y1[i] + (1-alpha)*y1[i+1]
     end
    else Replace(x1,y1,x0)
   end
  end
end;                                  { End Two }
procedure Four;
{Course to be adopted when s[i]*m1[l1]<0}
var aflag:boolean;
begin
 if (b1[i]<>0) then
  begin
   c11 := (c1[i]-a1[i]*x0)/b1[i];
   Convex(k,y0,c11,alpha,aflag);
   if (aflag=true) then
    begin
     Convex(x1[i],x1[i+1],x0,alpha,aflag);
     if (aflag=true) then
      begin
       FindDominate(h,k,l1,l2);
       DominatingPair;
       h := x0;
       NextMovement(h,x1,y1);
       Update(y0,k);
       if (abs(s[i])<1) then done := true  { M is optimal }
       else                          { abs(s[i])>=1 }
        begin
         Convex(y1[i],y1[i+1],y0,alpha,aflag);
         if (aflag=true) then
          begin
```

```
              done := true;
              k := y0; h := alpha*x1[i] + (1-alpha)*x1[i+1]
             end
            else Replace(y1,x1,y0);                    { aflag=false }
            aflag := true
           end
         end
        else Replace(x1,y1,x0);                        { aflag=false }
        aflag := true
       end
     end;
    if (b1[i]=0) or (aflag=false) then
     begin
      Convex(y1[i],y1[i+1],y0,alpha,aflag);
      if (aflag=true) then
       begin
        if (abs(s[i])>1) then
         begin
          done := true;                                { M is optimal }
          k := y0; h := alpha*x1[i] + (1-alpha)*x1[i+1]
         end
        else
         begin
          jj := l1;
          FindDominate(h,k,l1,l2);
          if (l1=jj) then l1 := l2;
          SingleSide;
          Convex(x1[i],x1[i+1],x0,alpha,aflag);
          if (aflag=true) then
           begin
            done := true;
            h := x0; k := alpha*x1[i] + (1-alpha)*x1[i+1]
           end
          else Replace(y1,x1,y0)
         end
       end
      else Replace(x1,y1,x0)
     end
  end;                                  { End Four }
begin                                   { main action statements }
  clrscr;
  assign(infil,'input file');
  {The coordinates of the demand points and the coefficients
   of the linear constraints to be read from the input file}
  reset(infil);
  assign(outfil,'output file');
  rewrite(outfil);
  writeln('supply no. of constraints m and no. of points n');
  readln(m,n);
  for i := 1 to n do readln(infil,x[i],y[i]);
  for i := 1 to m do readln(infil,a[i],b[i],c[i]);
  Diamond(u1,u2,v1,v2);
  ax[1]:=(cc[1]-cc[3])/2; ay[1]:=(cc[1]+cc[3])/2;
  ax[2]:=(cc[1]-cc[4])/2; ay[2]:=(cc[1]+cc[4])/2;
```

```
 ax[3]:=(cc[2]-cc[4])/2; ay[3]:=(cc[2]+cc[4])/2;
 ax[4]:=(cc[2]-cc[3])/2; ay[4]:=(cc[2]+cc[3])/2;
 for i := 1 to 4 do
  if (i=1) or (i=2) then m1[i] := -1 else m1[i] := +1;
 Counterclock;
 FindSegment(h1,k1,h2,k2,flag);
 if flag=true then
  begin
   write('The segment from (',h1,', ',k1,')','to (',h2);
   writeln(', ',k2,') constitutes the required solution')
  end
 else                                { flag=false }
  begin
   done := false;
   InitialStep;
   while not(flag) do
    begin
     if (l2<>0) then
      begin
       Two;
       if (done=true) then flag := true
      end
     else                           { l2=0 }
      begin
       if (s[i]*m1[l1]>0) then
        begin
         if (abs(s[i])<1) then One(h,k,x0,x1,y1)
                          else One(k,h,y0,y1,x1);
         if (done=true) then flag := true
        end
       else                         { (s[i]*m1[l1]<0) }
        begin
         Four;
         if (done=true) then flag := true
        end
      end
    end                             { end while }
  end;
 writeln(outfil,'h= ',h,'k= ',k);
 close(outfil);
 close(infil)
end.
```