Programming the Graphics of Two-Dimensional LP Problems in Management Science

Ronald D. Notestine

Introduction

The importance of so-called "Computer Algebra Systems" for the visualization of mathematical concepts has been much discussed in recent years. An important aspect of the software package Mathematica is that it is also a complete programming environment, as well as a system for doing mathematics. In a class in which both mathematical and computer programming skills are of concern, this is an important issue.

In fact, the reason for its great growth in popularity among research institutions, engineers, architectual designers, financial analysts, and others is that the system for doing mathematics is itself the programming environment. Further with its extensive pattern matching capability, powerful techniques are immediately available which would require painstaking programming to duplicate in a traditional procedural language.

GraphLP2D is a package written in the Mathematica programming language. It can be used to make graphs for 2 dimensional linear programming problems, and was made for instructional purposes. The package is used with a dual purpose in class. One purpose is mathematical: to help users to visualize simple Linear Programming problems, and the relation of the feasible region to optimal solutions. The second purpose is in programming: The users write Mathematica functions that utilize the existing package. Then, they rewrite the package to give it more functionality.

As the user receives the package, knowledge of Mathematica programming is needed to make use of it. For example, there is no function to draw a graph showing the constraint lines, or their intersections. But, there is a function that finds the coordinates of all intersections of constraint lines. There are then other functions that construct Mathematica Graphics Objects for the points and lines.

A Simple Maximum Problem

Here is a standard, 2-variable LP problem:

```
Maximize:    z = 4x + 3y
Subject to:   x + 2y  <=  8
             2x    y  <=  7
             x >=  0,   y  >=  0
```

To use the package, we must define vectors for the objective function, 'c', the rhs of the constraints, 'b', and the coefficient matrix, 'mm', of the lhs of the constraints. Further, we need a vector of the operations relating the rows of mm with the elements of b. Both vectors and matrices are represented in Mathematica as lists.

So, we enter the following:

```
c      = {4,3};
mm     = {{1,2},{2,1},{1,0},{0,1}};
b      = {8,7,0,0};
ops    = {LessEqual,LessEqual,
          GreaterEqual,GreaterEqual};
```

Notice that the user is required to explictly enter all constraints, including the non-negativity of x and y.

A Rudimentary Data Structure

To find the set of intersections of the four constraints (two normal, plus the two non-negativity constraints), the user uses the package function 'IntersectionsIL'

```
intsIL = IntersectionsIL[mm,b]

{{{1, 2}, {2, 3}}, {{1, 3}, {0, 4}}, {{1, 4}, {8, 0}},

                                  7
  {{2, 3}, {0, 7}}, {{2, 4}, {-, 0}}, {{3, 4}, {0, 0}}}
                                  2
```

Here, the user encounters a rudimentary data structure. The output is what is called an "Indexed List" in the parlance of the package. In this case, it is a list of intersections of constraint equations, indexed by the numbers of the two equations intersecting at that point.

The 'ColumnForm' function strips off the outermost parentheses and displays the elements of the list one per row, making it easier to read.

```
intsIL//ColumnForm
   {{1, 2}, {2, 3}}
   {{1, 3}, {0, 4}}
   {{1, 4}, {8, 0}}
   {{2, 3}, {0, 7}}
             7
   {{2, 4}, {-, 0}}
             2
   {{3, 4}, {0, 0}}
```

Graphics Primitives

In order to make graphs, we must have several quantities available. One basic item is the indexed list of all constraint intersections, shown above. But there are others.

In order to show the points of intersection, we must put the x-y coordinates (numbers) inside the head 'Point'. For example, the coordinates of the origin are {0,0}, but if we want to show that point in a Mathematica graphic, we must use:

```
Point[{0,0}]
```

The head 'Point' is sometimes called an "inert" head. It takes no action on its argument. Its only purpose is to label the list arguments as the coordinates of a point. (We will see later that we will need another layer of an inert head, "Show", before we can actually display the point on screen.)

To make a list of the 'Point' graphics primitives for the intersections, we strip out the second (last) sub-element of each element and wrap the head 'Point' around each of the resulting elements. The Mathematica function 'Map' does this for us. (For which we can also use the notation '/@'.)

Here, we extract the list of x-y coordinates from the indexed list.

```
Last /@ intsIL
                                          7
   {{2, 3}, {0, 4}, {8, 0}, {0, 7}, {-, 0}, {0, 0}}
                                          2
```

The function ConstraintLines gives another useful definition for a Graphics Primitive: the segments of the constraint lines that connect the points of intersection.

```
conLinesGP = ConstraintLines[intsIL]

   {{Line[{{2, 3}, {0, 4}}], Line[{{0, 4}, {8, 0}}]},

                                                 7
    {Line[{{2, 3}, {0, 7}}], Line[{{0, 7}, {-, 0}}]},
                                                 2
```

```
{Line[{{0, 4}, {0, 7}}], Line[{{0, 7}, {0, 0}}]},

                  7                 7
{Line[{{8, 0}, {-, 0}}], Line[{{-, 0}, {0, 0}}]}}
                  2                 2
```

Other graphics primitives we will use are easily concocted on the spot. However, this one requires a little programming, and so is better built into the package.

Corners of the Feasible Region

The package function FeasibleCornersIL extracts only those constraint intersections that are feasible (satisfy all constraints), and puts them in order, which may be either clockwise or counter-clockwise. (The corners are put in order so that the feasible region will display properly using the 'Polygon' function.)

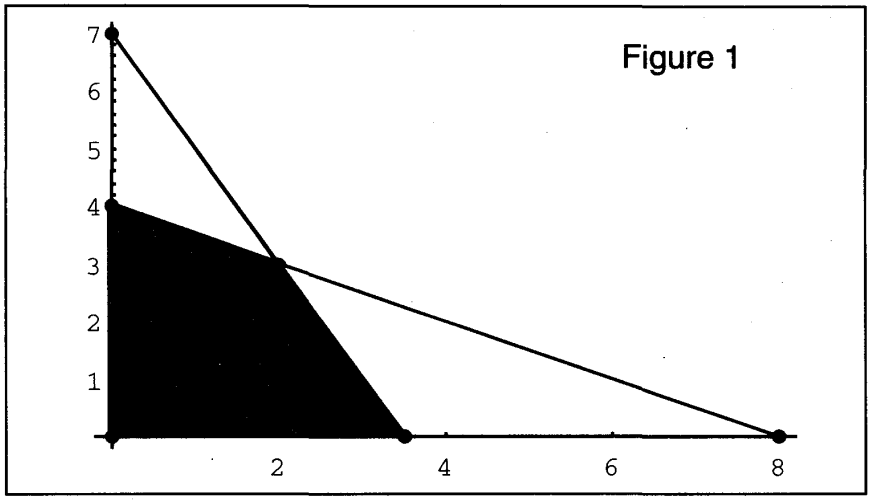
```
feasCornsIL = FeasibleCornersIL[mm,ops,b];
```

At first sight, it would seem better to leave extraction of the feasible intersections to the user. If it were just a matter of making a list of feasible intersections in any order, this would be so. However, in order to use these points in the Polygon function, with the desired result, the points must be in either clockwise or counter-clockwise order around the perimeter. This provides a good programming example for the user.

Graphing the Constraints

We use the show function with the graphics primitives made so far. The Polygon function is applied to the list of xy coordinates of intersections. The list of coordinates is the list of the the last subelements of each element of the list cornersIL.

```
Show[
      Graphics[{GrayLevel[.5],
            Polygon @ (Last /@ feasCornsIL) }],
      Graphics[{PointSize[.02],
            Point /@ (Last /@ intsIL)}],
      Graphics[ conLinesGP ],
      Graphics[{Thickness[.0075],
             ConstraintLines @ feasCornsIL}],
      Axes->True
];
```



The Function 'ObjectiveLine'

The user can make a Graphics Primitive for the objective line by using the package function: 'ObjectiveLine'.

```
ObjectiveLine[{4,3}, 10]
                10    5
   Line[{{0, --}, {-, 0}}]
                 3    2
```

This can be used to make a movie of the feasible region and the line for the objective function, $z == c1\ x + c2\ y$, corresponding to different values of z.
It is convenient to make a single Graphics Object for the feasible region

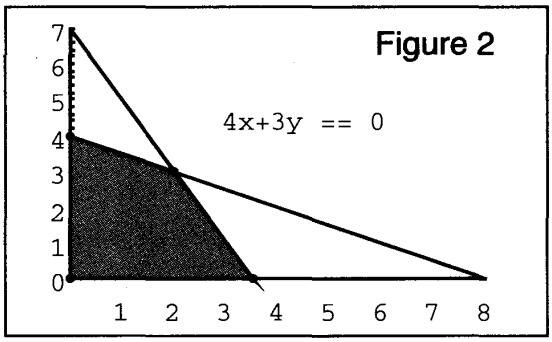
```
feasRegionGO =
      Graphics[{
        GrayLevel[.5],
            Polygon[ Last /@ feasCornsIL ],
        PointSize[.02],GrayLevel[0],
            Point /@ (Last /@ feasCornsIL),
        conLinesGP,
        Thickness[.0075],
            ConstraintLines[feasCornsIL]
      }];
```

The movie is made using a Do loop:

```
Do[
      Show[
        feasRegionGO,
        Graphics[{
              Thickness[.01],GrayLevel[.25],
              ObjectiveLine[c,z]
```

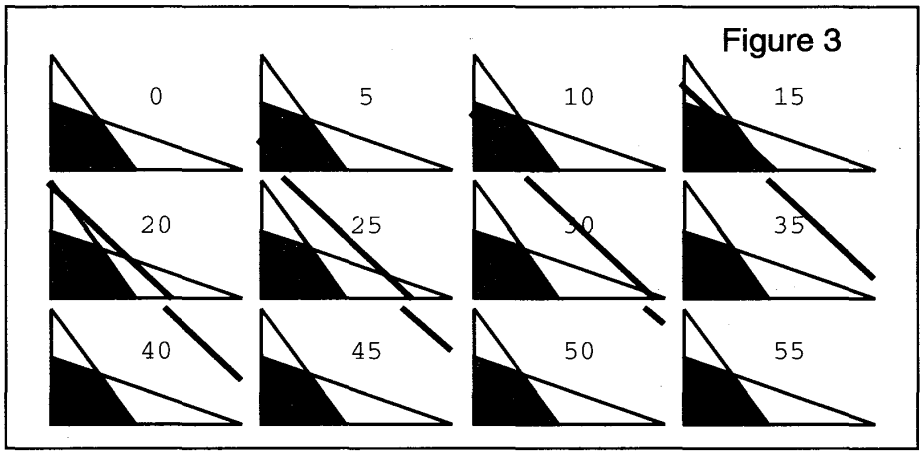
```
        }],
        Graphics[
          Text[
            "4x+3y == "<>ToString[z],
            {4.5,4.5}
          ]
        ],
        PlotRange->{{0,8},{0,7}},
        Axes->True
        ],
        { (*for*) z, (*from*) 0, (*to*) 55, (*step*) 5 }
]
```

The first cell generated (z = 0) is shown at right. When the user views this electronically, on the computer screen, he can play the whole sequence of graphics cells as a movie. Controls at the bottom of the screen allow him to adjust the speed and play in stop-motion etc.



Here is an array of all of the frames of the movie. (Modified slightly for clarity: tickmarks have been eliminated, label abbreviated, and objective line darkened.)

The movie shows clearly that the maximum, feasible value of the objective function is at x=2, and y=3. And, That this happens for z between 15 and 20. It is extremely easy for the user to zoom in and repeat the animation for values of z between 15 and 20. we will not do that here. Instead, we will pass on to the next stage.



The Objective Function at the Corners of the Feasible Region

It is well known that all extreme solutions of a linear objective function over a convex feasible region must occur at a vertex of the feasible region. So, another way to solve our linear programming problem is to evaluate the objective function at the corners of the feasible region, and choose a maximal (minimal) corner.

To find the value of the objective function at a point { x, y }, we take the vector dot product of { x, y } with the vector of coefficients of the objective function. The vector of objective coefficients is c = { 4, 3 }. So the value of the objective function at the point { 2, 3} is:

```
Dot[c,{2,3}]
    17
```

To find the list of all objective function values at corner points, we map the Dot function onto the list of all xy coordinates for the corner points.

```
Dot[c,#]& /@ (Last /@ feasCornsIL)
   {17, 12, 0, 14}
```

At this point, we might ask the user to construct a function to find a corner that maximizes the objective function. Here is one possible definition:

```
maxZXY[vecC_, cornerListIL_ ] :=
      Module[{cornerZs},
        cornerZs = Dot[vecC,#]& /@ (Last /@ cornerListIL);
        Select[
           { Dot[vecC,#], # }& /@ (Last /@ cornerListIL),
           #[[1]] == Max[cornerZs ]&
        ]
      ]//First

maxZXY[c,feasCornsIL]
   {17, {2, 3}}
```

The result of this function is a list giving both the maximum feasible value of the objective function, and the coordinates of the corner where this maximum value is attained. It is easily used to draw a picture:

```
Show[
      feasRegionGO,          (*  Feasible Region *)
      Graphics[{                    (*  Line for max z  *)
        Thickness[.01],GrayLevel[.25],
        ObjectiveLine[
          c,
          First[ maxZXY[c,feasCornsIL] ]
        ]
      }],
      Graphics[{      (*  Circle point with max z *)
        PointSize[.025],GrayLevel[0],
        Circle[#, .5]& @ Last[maxZXY[c,feasCornsIL]]
      }],
      Graphics[                     (*  Text Label *)
        Text[
          "4x+3y == "<>
          ToString[First[maxZXY[c,feasCornsIL]]],
          {4.5,4.5}
        ]
      ],
      PlotRange->{{0,8},{0,7}},
      Axes->True
]
```

The displayed graph resulting from the 'Show' command above.

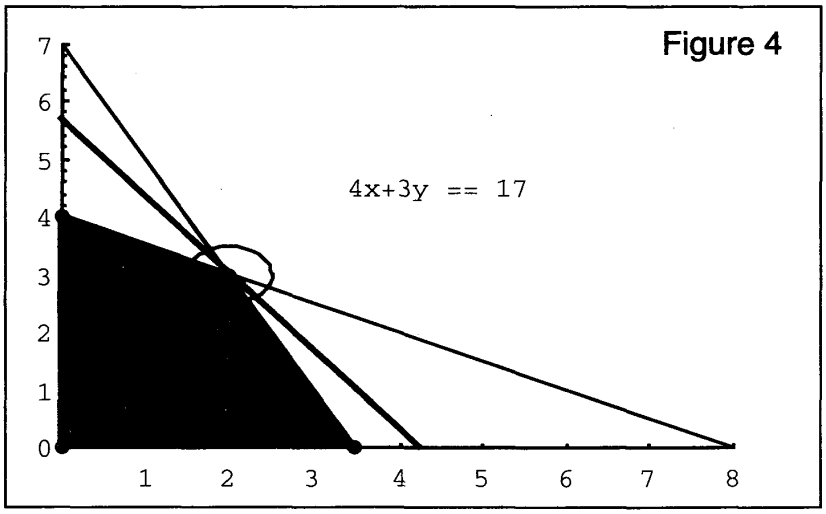


Figure 4

A Small Feasible Region, Away from the Origin

Here, we have a small feasible region, narrowly bound between two parallel lines. For reasons of space, we omit most of the Mathematica code. It is mostly similar or identical to that above.

The Graph

```
Show[ plotViewGO, Axes->True];
```

The feasible region appears to be the narrow strip between the points. The Maximum Objective Function is the thicker gray line passing through the upper of the two points. (We omit the code that produces the Graphic Object 'plotViewGO')

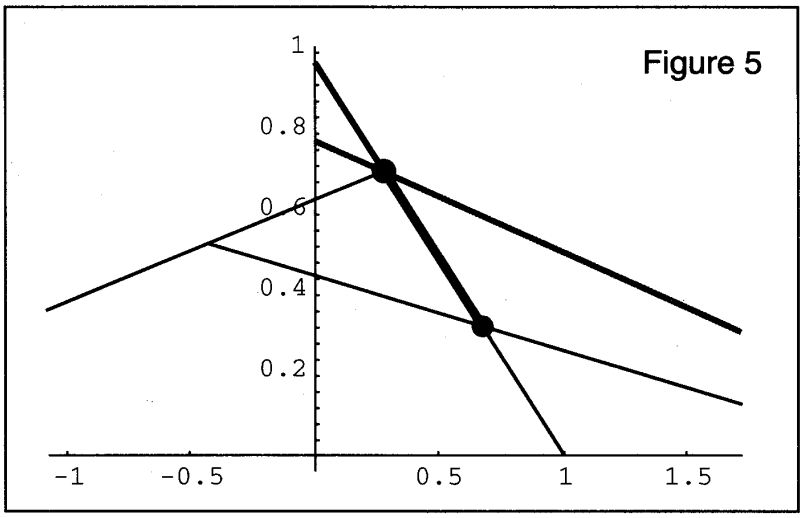


Figure 5

We can zoom in. Below is a closeup of the top of the feasible region. The objective function line is shown passing through the maximum feasible corner point. A copy of the overall picture is in the upper right corner.

```
Show[
      plotViewGO,
      Graphics[
        Rectangle[
           {0.29, 0.6950},
           {0.30, 0.7095},
           plotViewGO
        ]
      ],
      Axes->True,
      AxesOrigin->{0.275, 0.710},
      PlotRange->{{0.275,0.300},{0.690,0.710}}
];
```

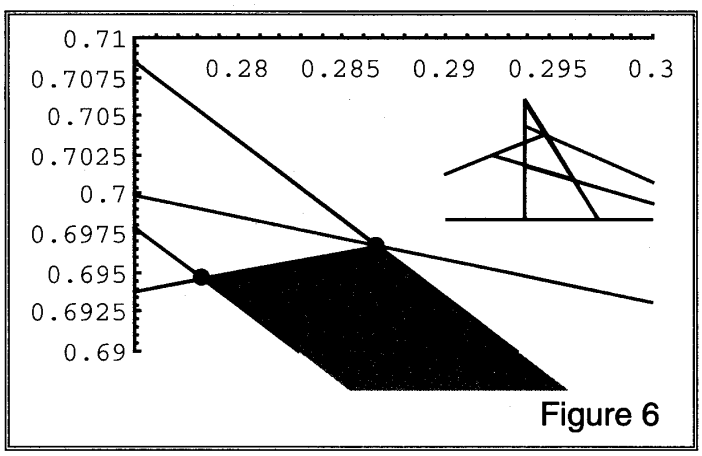


Figure 6

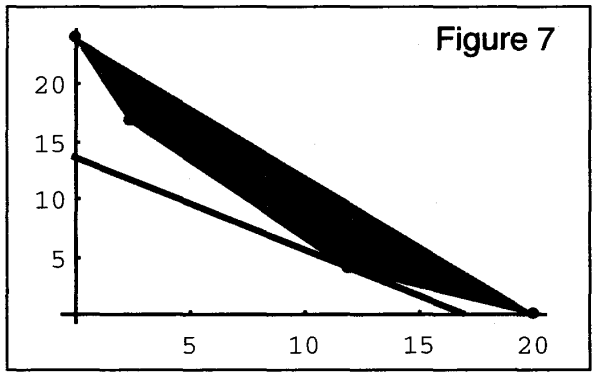
The main figure clearly shows the relation of the objective function to the feasible region. While, at the same time, the relation of the feasible region to the overall constraint set is clearly shown in the inset graphic.

A Minimization Problem, with Variation of Two Parameters

We can also display unbounded feasible regions. Here we have the feasible region for a minimization problem. With the objective function for the minimum feasible value shown in gray.

If the user wishes to investigate the effect of varying one more of the parameters, he may. Here we illustrate the result of animating simultaneous changes in the first coefficient of the objective function and the availablity of the resource for the first constraint.

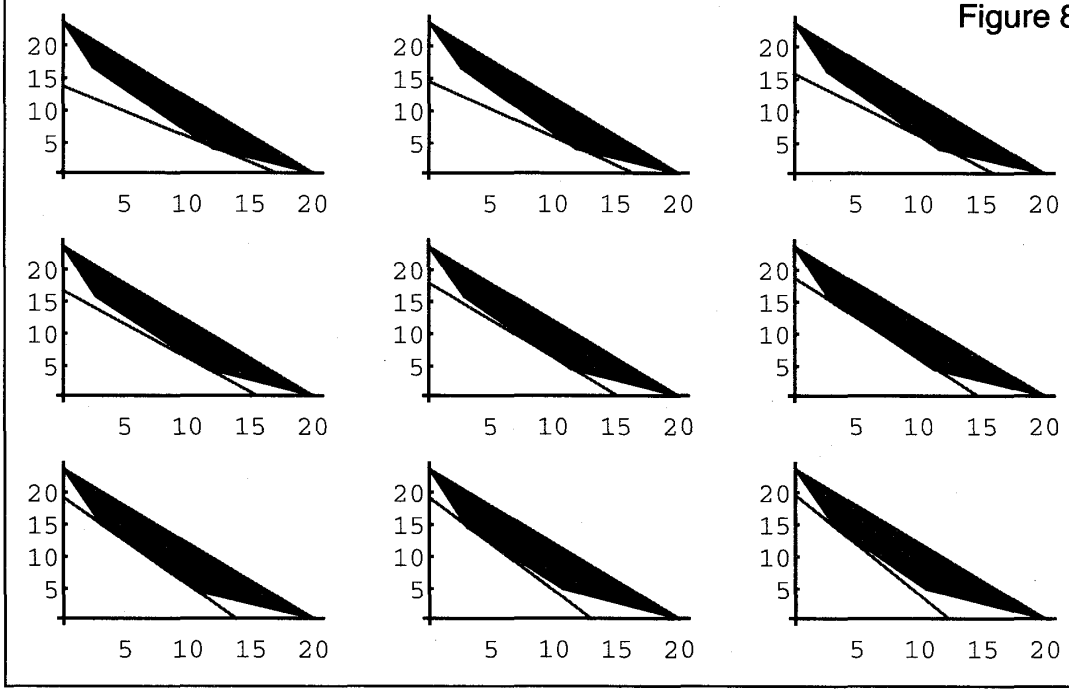
Here is the Do loop to produce the graphics.




```
Do[
      Module[{minz,newc,newb,frCornsIL},
        newb = b;
        newb[[1]] -= dx/2; (* Note the C-like '-=' *)
        frCornsIL = FeasibleCornersIL[mm,ops,newb];
        newc = c + {dx,0}; (* change c1 only *)
        minz = First @ minZXY[newc,frCornsIL];
        Show[
          Graphics[{
            Thickness[.01],GrayLevel[.5],
            ObjectiveLine[newc,minz]
          }],
          feasRegGO[frCornsIL],
          Axes->True
        ]
      ],
      {dx, 0, 10, 0.5}
]
```

The effect is to increase the negativity of the slope of the objective function, and pull the center constraint in the figure toward the origin. This is probably better seen than described. Here is some of the sequence of graphics generated:

Figure 8



Again, we should emphasize that the user actually sees a sequence of graphics that he can animate through a simple double click. The Array shown above has been made for publication on the printed page only. (Although the user could easily make it as well.)

A Bit of Detail

Here is an example of a function from within the package that the user is expected to study and understand. It should be empasized that this was not written as an example of the most elegant programming possible in Mathematica. Rather it is meant to be accessible to a relatively inexperienced undergraduate.

The purpose of this function is to prepare the list of coordinates of the corner points of the feasible region for display using the Polygon function. If the corners are not arranged in either clockwise or counter-clockwise order, the result will be an hour-glass sort of shape, instead of a polygon.

```
wrongShape = Graphics[ Polygon[{{0,0},{1,0},{0,1},{1,1}}] ];
rightShape = Graphics[ Polygon[{{0,0},{1,0},{1,1},{0,1}}] ];
Show[GraphicsArray[{wrongShape,rightShape}]]
```

At the right are the results of the two different orderings of the same four points:

Figure 9

The Function in Question

Here is the function to protect against unintended hour-glasses by ordering the indexed list of feasible corners

```
PerimeterOrderIL[cornersIL_List] :=

    Module[{cornlist,openC,oldlist,newlist,corner},

      cornlist = cornersIL;

      (* Bounded? If not, put in dummy point *)
      If[Not[BoundedQ[cornlist]], (
         openC = OpenConstraints[cornlist];
         Print["Feasible region appears unbounded."];
         Print["Open constraints are ",
             openC[[1]]," and ",openC[[2]] ];
         (* Insert Dummy point *)
         cornlist = Append[
                  cornlist,
                  {openC,Infinity}
                  ]
      )];

      oldlist     =  Rest[cornlist];
      newlist     = {First[cornlist]};
      corner      =  First[cornlist];

      While[oldlist != {},
         corner = Select[oldlist,
                  (Intersection[#[[1]],
                           corner[[1]] ] != {} )&
             ]//First;
         newlist = Append[newlist, corner];
         oldlist = Complement[oldlist,{corner}]
      ];

      (* Remove dummy point *)
      Select[newlist, (#[[2]] =!= Infinity)&]
    ]
```

Explanation

The first thing done is to define a module. This provides a protected local envirionment, in which local variables can be defined and used. The effect is equivalent to the definition of a procedure in a language such as Pascal or C.

Next, the feasible region is checked for boundedness. The method used to sort on the corners requires that each constraint be associated with exactly two corner points. This is not true if the

feasible region is unbounded. In this case, a "corner at infinity" is temporarily added.

The boundedness is checked using another function defined in the package "BoundedQ", but not reproduced here. If the answer is that the feasible region is unbounded, then another function, also not reproduced here, provides the list of exactly which two constraints do not meet. An element consisting of the numbers of these two constraints, "openC", and the marker "Infinity" is then appended to the list of corner points

```
cornlist = Append[ cornlist, {openC,Infinity} ]
```

Next, is the heart of the function. A 'While' loop is used. Corners are taken one at a time and added to the new, ordered, list of corners. At the same time, that corner is removed from the old list of corners. This process continues until there are no more corners left in the old list.

The manner in which this is done is, first, to look at any corner and ask which two constraints intersect at that corner. Then, we look along one of the constraints to find the other corner point involving that constraint. From that point, we look along the other constraint to find the next corner point. We continue walking along the constraints from corner to corner, until there are no more corners.

The manner in which we program this in Mathematica is straghtforward: We find the element of the indexed list of corners where the set theoretic intersection of the costraint numbers with the current constraint number is not empty

```
corner = Select[oldlist,
         (Intersection[#[[1]],
            corner[[1]] ] != {} )&
        ]//First
```

This corner is then appended to the new list, and removed from the old list using set complementation.

When the list is finished, we need only remove the "point at infinity".

```
Select[newlist, (#[[2]] =!= Infinity)&]
```

We select only those elements whose second element is not Infinity. Since this is the last line of the function, and is not followed by a semi-colon, it is the value returned by the function.

Summary

GraphicsLP2D is really a fairly simple-minded package. It does have a bit more sophistication than one might guess from just the summary above. For example, it performs some checking on function arguments: It will respond properly if some arguments are given in a different order, and it will not respond at all in many cases of incorrect arguments. It is far from unbreakable, but study of the manner in which it performs argument checking, and extending it to more cases provides good practice for the user.

Extension of this package to cover three dimensions would truly be a truly worthwhile, and ambitious, user project.

References

Blackman, N. (1992). Mathematica: A Practical Approach. Englewood Cliffs, NJ: Prentice Hall.

Gaylord, R. J.,Kamin, S. N., & Wellin, P. R. (1993). Introduction to Programming with Mathematica. Santa Clara, CA: Springer-Verlag.

Gray, J. W. (1994). Mastering Mathematica: Programming Methods and Applications. AP Professional.

Hillier, F. S., & Lieberman, G. J. (1992). Introduction to Mathematical Programming. McGraw Hill.

Maeder, R. (1991). Programming in Mathematica. Addison-Wesley.

Maeder, R. (1994). The Mathematica Programmer. AP Professional.

Wolfram, S. (1991). Mathematica (2nd ed.). Addison-Wesley.

福田治郎，児玉正憲，中道博 (1993) ＯＲ入門. 東京，多賀出版

Appendix: The Complete Package Listing

```
(*************************************************************)
(*                                                           *)
(*                        Package                            *)
(*                                                           *)
(*                      GraphLP2D                            *)
(*                                                           *)
(*                 Author: Ronald D. Notestine               *)
(*                   Date: Mar 10, 1994                      *)
(*                                                           *)
(*************************************************************)

BeginPackage["graphLP2D`"]

ConvertIneq::usage = "
ConvertIneq[ lhs <= rhs (or lhs >= rhs) ] puts the\n
       \tinequality ( lhs <= rhs, or lhs >= rhs) into\n
       \tstandard form e.g:\n
       \t\t\t1+2x<=5-3y   ==>   2x+3y<=4.\n
       \tThe rhs side is made positive:\n
       \t\t\t 2x-3y<=-2   ==>  -2x+3y >= 2"

IntersectionsIL::usage = "
IntersectionsIL[m,b] produces an indexed list of the\n
       \tpairwise solutions of the system of equations\n
       \t\t\tm.{x,y} == b.\n
       \tWhere { {i,j}, {x,y} } represents the\n
       \tintersection, {x,y}, of equations number i and j\n
       \t, the ith and jth rows of matrix m.\n
IntersectionsIL[m, b, k] produces the subset of pairwise\n
       \tsolutions that satisfy row k of the system m.{x,y} == b.\n
       \t(Only intersections involving equation k.)\n
IntersectionsIL[indexedList, k] produces the subset of\n
       \tindexedList with integer k as one element of the index\n
       \telement. (Only intersections involving equation k.)"

FeasibleCornersIL::usage = "
FeasibleCornersIL[m,ops,b] produces the indexed list of the\n
       \tx-y coordinates of the corners of the feasible region,\n
       \tin counter-clock-wise order. Where, ops is the list of\n
       \toperations (<=, >=, or ==) relating the rows of m with\n
       \tthe elements of b. The format of the output is\n
       \t\t {{i,j},{x,y}},\n
       \tindexed on rows of m."
```

```
ConstraintLines::usage = "
ConstraintLines[indexedList] constructs a list of Line graphic\n
       \tprimitives from  an indexed list of coordinates. The\n
       \tlines are through each pair of x-y coordinates that\n
       \tshare a common index.\n
ConstraintLines[indexedList,k] constructs a list of the Line\n
       \tgraphic primitives, pertaining to index k, from  an\n
       \tindexed list of coordinates."

ConstraintsIL::usage = "
ConstraintsIL[m, ops, b,{x_Symbol, y_Symbol}] produces an\n
       \tindexed list of the constraints. Where m is the\n
       \tconstraint coefficient matrix, b the list of constraint\n
       \tlimits, and ops the operations relating them. (LessEqual,\n
       \tGreaterEqual, etc.) The last argument must be a list of\n
       \ttwo symbols (used in the equations/inequalities)."

FeasibleQ::usage = "
FeasibleQ[m,ops,b,{x0,y0}] returns True if {x0,y0} is located\n
       \twithin the feasible region for m.{x,y}<=b,\n
       \tand false otherwise."

BoundedQ::usage = "
BoundedQ[FeasibleCornersIL] returns True if the feasible\n
       \tis bounded, and False otherwise."

ObjectiveLine::usage = "
ObjectiveLine[{c1,c2},val] makes a Line graphics primitive\n
       \tthat goes from from axis to axis.\n
       \tIf one of c1 and c2 is 0, the line goes\n
       \t from one axis to the line x=y.\n
       \tMust have val>0, c1>=0, and c2>=0, (not both zero)."

Begin["`Private`"]

DIM = 2;       (* This package is for 2 Dimensions        *)
                     (* DIM is used to screen some fctn args     *)
OPERATORS = {GreaterEqual,LessEqual,Equal};

(* * * Begin      ConvertIneq Begin        * * *)
(* ConvertIneq is from an original by Ed Greaves *)
ConvertIneq[ ineq_[lhs_,rhs_] ] :=
       Module[{temp,const,result},
             temp = Expand[ lhs - rhs ];
             const = If[Head[temp] === Plus,
                       Select[temp,NumberQ],
                       0 ];
             result = ineq[temp - const, -const];
             result[[1]] = (*get rid of stubborn real zeroes*)
               If[result[[1,1]]//NumberQ,
                  result[[1]]//Rest,
```

```
                result[[1]]        ];
            If[ -const<0,
                result = Minus /@ result;
                result = result /.
                            {GreaterEqual -> LessEqual,
                             LessEqual -> GreaterEqual }
            ];
            result
        ]
(* * *  END        ConvertIneq  END    * * *)



(* * *  Begin Intersections definitions Begin                    * * *)
(*                                                                      *)
(* Contains: LinSolve, IntersectionsIL                                  *)

(**)
LinSolve[{m1_,m2_},{b1_,b2_}] :=
        Module[{soln},
            Off[LinearSolve::nosol]; (* message off      *)
            soln = LinearSolve[{ m1, m2 }, { b1, b2 } ];
            Which[
              Head[soln] === List, (
                 On[LinearSolve::nosol];
                 Return[soln]       (* solution *)
              ),
              Head[soln] == LinearSolve, (
                 On[LinearSolve::nosol];
                 Return[None]
              ),
              True, (
                 On[LinearSolve::nosol];
                 Print["LinSolve: Failed"];
                 Return[$Failed]
              )
            ]
        ]
(**)
IntersectionsIL[ m_List, b_List ] :=
        Module[{ nR, sols },
            nR = Length[m];
            sols =
              Table[
                 { {i,j},
                   LinSolve[{ m[[i]], m[[j]] },
                            { b[[i]], b[[j]] }
                   ]
                 },
                 {i,1,nR-1},
                 {j,i+1,nR}
              ]//Flatten[#,1]&;
```

```
            (* Select only cases where intersection found *)
            Select[sols, (Head[#[[2]]]===List)& ]

       ](* args shape OK? *) /; (Length[m] == Length[b]  &&
                                 Length[Dimensions[m]] == 2 &&
                                 Length[Dimensions[b]] == 1  )
(**)
IntersectionsIL[listIL_List, k_Integer ] :=
                      Select[
                          listIL,
                          MemberQ[ #[[1]], k ]&
                      ]
(**)
IntersectionsIL[ m_List,  b_List, k_Integer ] :=
       IntersectionsIL[
            IntersectionsIL[m,b],
            k
       ]
(**)   (* Mistaken inclusion of ops  *)
IntersectionsIL[ m_List,  b_List, ops_List ] :=
                                    IntersectionsIL[m,b]

(* * *       END Intersections definitions END            * * *)

(* * *       BEGIN Constraints definitions BEGIN  * * *)
(*                                                              *)
(* Contains: ConstraintLines, ConstraintsIL                     *)

(**)
ConstraintLines[listIL_List, k_Integer] :=
       Line /@
            Partition[
              Last /@ IntersectionsIL[listIL,k],
              2,1
            ]
(**)
ConstraintLines[listIL_List] :=
       Table[
            Line /@
              Partition[#,2,1]&[
                 Last /@ Select[ listIL, MemberQ[#[[1]], k]& ]
              ],
            {k, Length[Union[Flatten[First /@ listIL]]] }
       ]
(**)
ConstraintsIL[ m_List, ops_List, b_List,
                      {x_Symbol, y_Symbol} ] :=
       Module[{n},
            n = Length[m];
            Join[
              Table[
```

```
                {   i,
                    (m[[i]].{x,y}) ~ ops[[i]] ~ b[[i]]
                },
                {i,1,n}
              ],
              {
                { n+1, x >= 0 },
                { n+2, y >= 0 }
              }
            ]

      ] /; (* args OK? *)  (
              (Length[m] == Length[ops] == Length[b]) &&
               And @@ ( MemberQ[OPERATORS, #]& /@ ops )  )
(**)
ConstraintsIL[ m_List, b_List, ops_List,
                    {x_Symbol, y_Symbol} ] :=
      ConstraintsIL[ m, ops, b, {x,y} ]

(* * *       END Constraints definitions END              * * *)


(* * * BEGIN Feasible Region definitions BEGIN    * * *)
(*                                                                *)
(* Contains: BoundedQ, OpenConstraints,                          *)
(*              FeasibleCornersIL, FeasibleQ,                    *)
(*              PerimeterOrderIL                                 *)
(**)
BoundedQ[feasCornsIL_List] :=
      Module[{cNbrs, occursPerC},
            cNbrs = (First /@ feasCornsIL)//Flatten;
            occursPerC = Length /@ (Cases[cNbrs,#]& /@ cNbrs);
            FreeQ[occursPerC, 1] &&
                  FreeQ[feasCornsIL, Infinity]
      ]
(**)
OpenConstraints[feasCornsIL_List] :=
      Module[{cNbrs, allC, occursPerC, openC},
            allC = (First /@ feasCornsIL)//Flatten;
            cNbrs = allC//Union//Sort;
            occursPerC =
              Length /@ (Cases[allC,#]& /@ cNbrs);
            openC =
              Select[
                cNbrs,
                (
                  occursPerC[[
                        Position[cNbrs,#]//First
                            ]] == {1}
                )&
```

```
            ];
          If[Length[openC] > 2 ||
              Length[openC] == 1, (
            (* Then *)
            Print["OpenConstraints: Unexpected number of unpaired
constraints"];
            Print["Constraints no. list is: ",openC];
          ),(* Else *) (
            openC
          )]
      ]
(**)
FeasibleCornersIL[m_List, ops_List, b_List] :=
      Module[{x,y,constraints,ptsIL},

          constraints = (* automatically includes axes *)
              Last /@ ConstraintsIL[ m, ops, b, {x,y} ];

          ptsIL =  IntersectionsIL[m,b];

          ptsIL =  Select[ptsIL,
                   (And @@
                       (constraints /. {x->#[[2,1]],
                                        y->#[[2,2]]} )
                   )&
              ];

          If[ptsIL == {},
                          (* Then Not Feasible        *)
            Print["FeasibleCornersIL: No Feasible Region Exists"];
            Return[Null]                  (* **BREAK: RETURN[Null]**   *)
          ];

              (* Return Ordered Feasible Corners      *)
          ptsIL //PerimeterOrderIL

      ] /; (* args OK? *)  (
            (Length[m] == Length[ops] == Length[b]) &&
             And @@ ( MemberQ[OPERATORS, #]& /@ ops )  )
(**)
FeasibleCornersIL[m_, b_, ops_] :=
            FeasibleCornersIL[m,ops, b]
(**)
FeasibleQ[m_List, ops_List, b_List,
                        { x0_?NumberQ, y0_?NumberQ } ] :=

      Module[{x, y, constraints},

          constraints = (* inequalitiesIL will include axes *)
              Last /@ inequalitysIL[ m, ops, b, {x,y} ];

          (* Are ALL constraints satisfied? Return: *)
```

```
            And @@ ( constraints /. {x->x0,y->y0} )

      ] /; (* args OK? *)  (
              (Length[m] == Length[ops] == Length[b]) &&
               And @@ ( MemberQ[OPERATORS, #]& /@ ops )  )

(**)
PerimeterOrderIL[cornersIL_List] :=

      Module[{cornlist,openC,oldlist,newlist,corner},

            cornlist = cornersIL;

            (* Bounded? If not, put in dummy point *)
            If[Not[BoundedQ[cornlist]], (
              openC = OpenConstraints[cornlist];
              Print["Feasible region appears unbounded."];
              Print["Open constraints are ",
                     openC[[1]]," and ",openC[[2]] ];
              (* Insert Dummy point *)
              cornlist = Append[
                            cornlist,
                            {openC,Infinity}
                          ]
            )];

            oldlist  =  Rest[cornlist];
            newlist  = {First[cornlist]};
            corner       =  First[cornlist];

            While[oldlist != {},
              corner = Select[oldlist,
                         (Intersection[#[[1]],
                                       corner[[1]] ] != {} )&
                       ]//First;
              newlist = Append[newlist, corner];
              oldlist = Complement[oldlist,{corner}]
            ];

            (* Remove dummy point *)
            Select[newlist, (#[[2]] =!= Infinity)&]

      ]     /; Dimensions[First[cornersIL]] == {DIM,DIM}

(* * *  END Feasible Region definitions END                 * * *)

(* * *  BEGIN ObjectiveLine definitions BEGIN               * * *)
(*                                                                 *)
(* Contains: ObjectiveLine                                        *)

(* Usual case,  c1 and c2 > 0 *)
(**)
```

```
ObjectiveLine[{ c1_?(#>0&), c2_?(#>0&) }, val_?(#>=0&) ] :=
             Line[{ {0, val/c2}, {val/c1, 0} }]

(* c1 is 0    *)
(**)
ObjectiveLine[{ c1_?(#==0&), c2_?(#>0&) }, val_?(#>0&) ] :=
             Line[{ {0, val/c2}, {val/c2, val/c2} }]

(* c2 is 0    *)
(**)
ObjectiveLine[{ c1_?(#>0&), c2_?(#==0&) }, val_?(#>0&) ] :=
             Line[{ {val/c1, val/c1}, {val/c1, 0} }]

(* * *        END ObjectiveLine definitions END            * * *)

End[]

Protect[IntersectionsIL,   ConstraintLines,
             FeasibleCornersIL,      FeasibleQ,
             BoundedQ,               ObjectiveLine,
             ConvertIneq  ];

EndPackage[]
```