

A Mathematica Package for Well Formed Feasible Regions

Ronald D. Notestine
Faculty of Management
Chukyo University

Key word : Mathematica, Linear Programming, Feasible Region, Visualization

Following is a Mathematica package for fine-tuning systems of equations in the plane to produce a desired feasible region. While it bears many similarities to preceeding packages, it has been designed first of all with the aim of allowing the user to test different sets of corner points for a feasible region, and choose the most suitable for his or her purposes. It includes many options for varying the appearance of plots. Either a set of corner points, or a set of equalities/inequalities can be used to define the feasible region. If the latter, redundant constraints can be included, and they will be plotted accordingly.

Functions are included for writing data tables to files suitable for use in spreadsheet or word processor applications. Plots themselves can either be saved in encapsulated Postscript format, or pasted directly into most word processors.

The plotting functions take a large number of special options defined for this package. In addition, they can take, and will use, any options suitable for the built-in Mathematica Plot or Show functions. Unbounded feasible regions are handled with some extra user input. When plotted, constraint boundaries and feasible region corner points are clearly delineated. The manner of delineation (or no delineation) is in the control of the user. The user choose to plot the objective function at any level or levels desired, including an automatically determined maximum or minimum level.

One caveat: While the user can specify any names desired for the variables, the names will always be sorted into lexical order.

The listing of the package follows.

```

(* :Title: WFFR (standing for: Well Formed Feasible Region) *)

(* :Author: Ronald D. Notestine *)

(* :Summary:
This package contains functions for finding and plotting a
system of equations bounding a region, given the set of corner
points of that region. The region can also be plotted from a
given set of equations or inequalities. Informational tables
can be written to text files suitable for use in spreadsheet
programs
*)

(* ***** Well Formed Feasible Region ***** *)
(* ***** Well Formed Feasible Region ***** *)
(* ***** Well Formed Feasible Region ***** *)
(* *)
(* *)

BeginPackage["WFFR`"];

Needs["Graphics`ImplicitPlot`"];
Needs["Graphics`Colors`"];
Needs["Utilities`FilterOptions`"];
Needs["DiscreteMath`ComputationalGeometry`"];
Needs["DiscreteMath`Combinatorica`"];

Off[General::spell];
Off[General::spell1];

PlusAttributeDefault = Attributes[Plus];
Unprotect[Plus];
Attributes[Plus] = {Flat, Listable, NumericFunction, OneIdentity};
Protect[Plus];
(* ***** Unprotect and Clear *** Public Names *** Unprotect and Clear *)
(* ***** Unprotect and Clear *** Public Names *** Unprotect and Clear *)
Unprotect[LineSystem, WriteToTextFile, InterceptTable, BasicSolutions,
  PlotFeasibleRegion, ColorCycleAmount, ColorForLine, ColorForPoint,
  ColorForLabel, ColorForObjective, ColorForRegion, ConstraintLabels, DefaultFont,
  NonConstraints, ObjectiveFunction, ObjectiveValue, PlotStyleForObjective,
  ShadeFR, ShowBoundaryLines, ShowCompleteLines, ShowExtremePoints,
  ShowLineLabels, LineLabelsOffset, ThicknessForLine, ThicknessForObjective,
  SizeForLabel, SizeForPoint, SmallLineReductionFactor, RedundantConstraints,
  ReorderPoints, Title, Ketasuu, NonBasicsToZero];

Clear[LineSystem, WriteToTextFile, InterceptTable, BasicSolutions,
  PlotFeasibleRegion, ColorCycleAmount, ColorForLine, ColorForPoint,
  ColorForLabel, ColorForObjective, ColorForRegion, ConstraintLabels, DefaultFont,
  NonConstraints, ObjectiveFunction, ObjectiveValue, PlotStyleForObjective,
  ShadeFR, ShowBoundaryLines, ShowCompleteLines, ShowExtremePoints,
  ShowLineLabels, LineLabelsOffset, ThicknessForLine, ThicknessForObjective,
  SizeForLabel, SizeForPoint, SmallLineReductionFactor, RedundantConstraints,
  ReorderPoints, Title, Ketasuu, NonBasicsToZero];
(* *** end end ***** end end ***** end end ***** end end ***** *)
(* *** end end ***** end end ***** end end ***** end end ***** *)

(* ** Attributes BEGIN Attributes BEGIN Attributes BEGIN Attributes *** *)
(**) (*for names in context Global`)
(*private Attributes are inside Private section*)
Attributes[WriteToTextFile] = {Orderless};
(****)
Attributes[MakeBoundarySystem] = {Orderless};
(* ** Attributes END Attributes END Attributes END Attributes *** *)

```

```

(* ***** Messages ***** BEGIN ***** Messages BEGIN Messages ***** *)
(* ***** Messages ***** BEGIN ***** Messages BEGIN Messages ***** *)
(**)
WriteToTextFile::usage =
  "WriteToTextFile[data_List, filename_String] writes the first argument to
  the file named by the second argument. The file is written to the current
  directory. (See the Mathematica functions Directory[] and SetDirectory[.])
  The arguments can actually be in any order, so long as the data is in a list,
  and the file name is a string.";
(**)
InterceptTable::usage =
  "InterceptTable[comparatorlist, options___] constructs a table of the
  intersection points of the equations/inequalities in comparatorlist. The first
  argument (which is the only required argument) must be a list of equations,
  inequalities, or both: In the resulting table, all are listed as equations.
  Row and column titles are constructed.
  Options are: Title (any string,
  default is "Intersections") and Ketasuu (any positive integer, default is
  3).";
(**)
BasicSolutions::usage =
  "BasicSolutions[eqns, vars, options] returns a list of the basic solutions,
  given a list of equations and variables. It needs the standard add-on package
  DiscreteMath`Combinatorica`. The only option is NonBasicsToZero, which is
  False by default.";

(****)
MakeBoundarySystem::usage=
  "MakeBoundarySystem[{p1,p2},{p2,p3},...,{pn,p1}] (,options)]
  returns a list wrapped in the head 'LineSystem'.
  Each element is a list with the following structure:
  {pt1,pt2,number, equation, {pt1,pt2}, {x-axis-int,y-axis-int}}
  Where 'equation' is the equation of the line through p1 and p2.
  The option 'NonConstraints' is a list of position numbers of equations that
  are not actual boundaries (in unbounded FR case).";
(****)
(*duplicated as PlotFeasibleRegion msg 000809*)
MakeBoundarySystem::overlap=
  "The position of one or more non-constraints as specified by the option
  NonConstraints overlap(s) with the position(s) of one or both axes. The
  Specifications are: {n1, n2, ...}";
(****)
(*duplicated as PlotFeasibleRegion msg 000809*)
MakeBoundarySystem::badfr =
  "The center point of the implied Feasible Region (FR) does not satisfy one
  or more inequalities. It would appear that either the direction of the
  offending inequality(s), or the order of the constraints as given, is
  incorrect. {n1, n2, ...} The Violated Inequality(s): {1, 2, ...} The Center Point
  (arithmetic mean of the corner pts): {x, y}";
(****)
(* BEGIN PlotFeasibleRegion family of messages BEGIN *)
(**)
PlotFeasibleRegion::usage=
  "PlotFeasibleRegion plots the feasible region, given either a set of
  bounding corner points, or the set of bounding constraint equations. (The
  equations can be inequalities, see below.) If corner points, variable names
  must also be given. There are a large number of options. An objective
  function can also be plotted at any level specified, or at an optimum to be
  calculated by the function. {n1, n2, ...} Unbounded regions cannot, of course, be plotted
  in their entirety. The user must provide points or equations that specify
  horizontal and vertical upper plotting bounds, which are not actual
  constraints. These points or equations would appear in the argument list

```

exactly as if they were actual corner points or bounding sides of the FR. The option 'NonConstraints' is then used to specify which are not constraints. (The difference is that constraints have a special line drawn, and their intersections have special points drawn.) A table is printed showing the constraint equations, coordinates of corner points of the FR (pairwise intersections of the constraints), and the axis-intercepts of the constraint equations. If corner points are given as an argument, the option 'ReorderPoints->True' can be used to reorder the points into the correct order to form a convex region. (If such is possible!) If equations (or inequalities) are given, the user must provide them in a correct order. Labels for the boundaries of the FR are shown, unless the option 'ShowLineLabels->False' is given. The position of the labels can be adjusted using the option 'LineLabelsOffset'. Constraints that are redundant (do not intersect the interior of the feasible region) should be specified using the option RedundantConstraints. Inequalities can be given rather than equations. But, they are converted to equations as if the user had provided only equations. If an inequality given by the user appears to be reversed, however, the user is notified. The argument formats are:

```

PlotFeasibleRegion[pointlist,variablenames,opt1,opt2,...]
PlotFeasibleRegion[equationlist,opt1,opt2,...]
PlotFeasibleRegion[equationlist,variablenames,opt1,opt2,...]

```

In all cases, the options (opt1 etc) are optional arguments (they may or may not be present). The other arguments must be present as given. If a list of equations (or, inequalities) is given along with a list of variable names, the only variable names appearing in the equations or inequalities should be precisely those in the variablename list. (And only those!) If the names in the equation list do not match those in the variable list, the results are unpredictable. It is strongly recommended that, if an equation or inequality list is given, no variable name list be given. The function will automatically extract the variable names. To see a list of all options, execute the expression: Options[PlotFeasibleRegion]. To see them in a more easily read form, execute the expression:

```

ColumnForm[Options[PlotFeasibleRegion]].";
(**)

```

PlotFeasibleRegion::badlbls=

"The number of constraint labels, '1', does not match the number of constraints, '2'. Resetting ConstraintLabels option to Automatic";

```

(**)

```

PlotFeasibleRegion::badOF=

"The Objective Function must not be an equation. That is, it should not be of the form $z=2x+3y$, but rather just $2x+3y$. The value of the OF is given with the option ObjectiveValue->{7}, or ObjectiveValue->{7,9,15}, for example. The Objective Function as given here is: `` This form cannot be processed, and the function 'PlotFeasibleRegion' has terminated.";

```

(**)

```

PlotFeasibleRegion::badfr =

"The center point of the implied Feasible Region (FR) does not satisfy one or more inequalities. It would appear that either the direction of the offending inequality(s), or the order of the constraints as given, is incorrect. The Violated Inequality(s): '1' The Center Point (arithmetic mean of the corner pts): '2'";

```

(**)

```

PlotFeasibleRegion::overlap=

"The position of one or more non-constraints as specified by the option NonConstraints overlap(s) with the position(s) of one or both axes. The Specifications are: ``";

```

(**)
(*used in utility function PerimeterOrder*)

```

PlotFeasibleRegion::inhull =

"One or more of the points given as corner points appear(s) to fall in the interior of the convex hull of the set of all points. The set of all points given: '1' The point(s) in the interior of the feasible region: '2'";

```

(**)

```

```

PlotFeasibleRegion::nonsym =
  "Expected symbols as variable names, but received `1` and `2` instead.";
(* BEGIN PlotFeasibleRegion Option USAGE messages BEGIN *)
ColorCycleAmount::usage =
  "ColorCycleAmount is an option to the PlotFeasibleRegion function.
Controls the difference in shading between the thin lines showing the
constraints, and the thicker lines showing the boundaries of the feasible
region. Should be a number between 0 and 1. Default is 1/2. See also
ColorForLine, ColorForPoint, ColorForRegion, ColorForObjective, and
ColorForLabel";
(**)
ColorForLine::usage =
  "ColorForLine is an option to the PlotFeasibleRegion function. The color
used for the thicker lines showing the boundaries of the feasible
region. Default is
  RGBColor[0.282403,0.2392040,0.5451060] (*
    DarkSlateBlue in the Graphics`Colors` package*). See also
  ColorCycleAmount, ColorForPoint, ColorForRegion, ColorForObjective,
  ColorForLabel, and SmallLineReductionFactor";
(**)
ColorForPoint::usage =
  "ColorForPoint is an option to the PlotFeasibleRegion function. The color
used for the thick points showing the corners of the feasible
region. Default
  is RGBColor[0.1843001,0.309793,0.309793] (*DarkSlateGray*). See also
  ColorForLine, ColorCycleAmount, ColorForRegion, ColorForObjective,
  and ColorForLabel";
(**)
ColorForLabel::usage =
  "ColorForLabel is an option to the PlotFeasibleRegion function. Color of
the typeface used for the labels of the sides bounding the feasible
region. Default is
  RGBColor[0.,0.,0.] (*Black in the Graphics`Colors` package*). See also
  ColorForLine, ColorForPoint, ColorForRegion, ColorForObjective,
  ColorCycleAmount, ShowLineLabels, and SizeForLabel";
(**)
ColorForObjective::usage =
  "ColorForObjective is an option to the PlotFeasibleRegion function. Color
used for the line representing the objective function, if an objective
function is plotted. Default is
  RGBColor[0.,0.,0.] (*Black in the Graphics`Colors` package*). See also
  ColorForLine, ColorForPoint, ColorForRegion, ColorForLabel, ColorForLabel,
  ObjectiveFunction, and ObjectiveValue";
(**)
ColorForRegion::usage =
  "ColorForRegion is an option to the PlotFeasibleRegion function. Color used
to shade the feasible region in the plot. Default is GrayLevel[0.9]. See
also ColorForLine, ColorForPoint, ColorForObjective, ColorForLabel,
ColorForLabel, and ShadeFR.";
(**)
ConstraintLabels::usage =
  "ConstraintLabels is an option to the PlotFeasibleRegion function. A list
giving the labels to be used for each bounding side of the feasible region.
Default is Automatic, in which case the sides are numbered from 1. (The
corresponding equations are shown in the table printed below the plot.) See
also ShowLineLabels, SizeForLabel, and LineLabelsOffset.";
(**)
DefaultFont::usage =
  "DefaultFont is an option to the PlotFeasibleRegion function. The font,
and fontsize, used to print the constraint labels. Must be a list consisting
of a string giving the name of the font, and an integer giving its pointsize.
Default is {¥"Courier¥",12}.";
(**)

```

NonConstraints::usage =

"NonConstraints is an option to the PlotFeasibleRegion function. In order to plot unbounded feasible regions, the user must include dummy constraints (e.g. $x_1 \leq \max X_1$, $x_2 \leq \max X_2$) to indicate the extent of the feasible region that the user wishes to plot. When doing this, the user should also indicate the positions of the dummy constraints in the constraint list. If this is not done, thick boundary lines, and large corner points, will be plotted to make these plotting limits look as if they were actually boundaries of the feasible region. For example: Suppose the entire non-negative quadrant is the feasible region ($x \geq 0$ & $y \geq 0$). But, we wish to plot it only up to 12 in the x-direction, and 10 in the y-direction. Then, our list of constraints might be: $\{y \geq 0, x \leq 12, y \leq 10, x \geq 0\}$. In which case, we would also use NonConstraints-> $\{2,3\}$. This is because the second and third items in the constraint list ($x \leq 12, y \leq 10$) are not part of the problem, but only plotting limits. The resulting plot will have heavy boundary lines only along the x-axis ($y \geq 0$) and the y-axis ($x \geq 0$). Further, the only corner point will be at the intersection of the two constraints, (0,0). If a list of corner points is given instead of a list of constraints, the implied positions of the nonconstraints should be given. In the example above, if the points are $\{(0,0), \{0,12\}, \{12,10\}, \{0,10\}\}$, then the option should again be NonConstraints-> $\{2,3\}$. Default is the empty list, {}.";

(**)

ObjectiveFunction::usage =

"ObjectiveFunction is an option to the PlotFeasibleRegion function. It should be a linear equation with all variable terms collected to the left of the equal sign, and all constants to the right. Examples are $2x + 3y == 9$ and $x[1] - 5x[2] == 5$. If objective function lines are to be plotted for more than one value of the objective function, they must be specified using the option ObjectiveValue-> $\{val1, val2, \dots\}$, in which case the rhs of the ObjectiveFunction equation is ignored. Default is None. See also ObjectiveValue and PlotStyleForObjective.";

(**)

ObjectiveValue::usage =

"ObjectiveValue is an option to the PlotFeasibleRegion function. A list giving the value or values at which the objective line should be plotted. Must be a list of numerical values, or expressions which evaluate to numerical values. If Maximize or Minimize is given, the appropriate optimum line is plotted. If Automatic, the rhs of the objective function given by the option ObjectiveValue is used. Unless the value of that option is None, in which case no line is plotted. Default is Automatic. See also ObjectiveFunction and PlotStyleForObjective.";

(**)

PlotStyleForObjective::usage =

"PlotStyleForObjective is an option to the PlotFeasibleRegion function. The style with which the objective line should be plotted. Anything suitable for the PlotStyle option to the Plot command can be used. Default is Automatic (which at this writing produces a thick, dashed line.)";

(**)

ShadeFR::usage =

"ShadeFR is an option to the PlotFeasibleRegion function. Whether or not to shade the interior of the feasible region. Boolean: True or False. Default is True.";

(**)

ShowBoundaryLines::usage =

"ShowBoundaryLines is an option to the PlotFeasibleRegion function. Whether or not to plot the (normally thick) boundary lines of the feasible region. Boolean: True or False. Default is True. See also ShadeFR, ShowCompleteLines, ShowExtremePoints, ShowLineLabels, ColorForLine, ColorCycleAmount, and SmallLineReductionFactor.";

(**)

ShowCompleteLines::usage =

"ShowCompleteLines is an option to the PlotFeasibleRegion function. Whether or not to plot the (normally thin) constraint lines. Note, these

```

lines extend from edge to edge of the plotted area, and are not the boundary
lines for the feasible region. (Though, they normally lie on that boundary
for part of their length.)\n Boolean: True or False. Default is True.\nSee
also ShadeFR, ShowBoundaryLines, ShowExtremePoints, ShowLineLabels,
ColorForLine, ColorCycleAmount, and SmallLineReductionFactor.";
(**)
ShowExtremePoints::usage =
  "ShowExtremePoints is an option to the PlotFeasibleRegion function.\n
Whether or not to plot the corner points of the feasible region.\n Boolean:
True or False. Default is True.\nSee also ShadeFR, ShowBoundaryLines,
ShowCompleteLines, ShowLineLabels, ColorForPoint, and ColorCycleAmount.";
(**)
ShowCornerPoints::usage =
  "NOT an option to the PlotFeasibleRegion function.\n\t\t\tUse the option
--> ShowExtremePoints";
(**)
ShowVertices::usage =
  "NOT an option to the PlotFeasibleRegion function.\n\t\t\tUse the option
--> ShowExtremePoints";
(**)
ShowLineLabels::usage =
  "ShowLineLabels is an option to the PlotFeasibleRegion function.\nWhether
or not to show labels for the boundaries of the feasible region.\n Boolean:
True or False. Default is True.\nSee also LineLabelsOffset, and also
ShadeFR, ShowBoundaryLines, ShowCompleteLines,, ColorForLine, and
ColorCycleAmount.";
(**)
LineLabelsOffset::usage =
  "LineLabelsOffset is an option to the PlotFeasibleRegion function.\nIt is
used to adujust the position of the labels for the lines bounding the
feasible region. It is given as a fraction of the length of the line. For a
given number, e.g. 0.20, the amount of offset will be greater for longer
lines than for shorter ones. It is normally given as a list on numbers, one
for each label.\nSee also ShowLineLabels, and also ShadeFR,
ShowBoundaryLines, ShowCompleteLines, x, ColorForLine, and ColorCycleAmount.";
(**)
ThicknessForLine::usage =
  "ThicknessForLine is an option to the PlotFeasibleRegion function.\nThe
thickness of the thicker lines used to bound the feasible region. The width
of the thinner constraint lines will be a fixed fraction of this. The
fraction is given by value of the option SmallLineReductionFactor, which at
this writing has a default of 3, meaning the thin lines will be 1/3 the width
of the thick lines.\nDefault is 0.015\nSee also ShowBoundaryLines,
ShowCompleteLines, ColorForLine, ColorCycleAmount, and
SmallLineReductionFactor.";
(**)
ThicknessForObjective::usage =
  "ThicknessForObjective is an option to the PlotFeasibleRegion function.\n
Gives the thickness for the plot of the objective line. Units are fraction of
the size of the whole plot.\nDefault is 0.015";
(**)
SizeForPoint::usage =
  "SizeForPoint is an option to the PlotFeasibleRegion function.\nGives the
size for the points at the corners of the feasible region plot. Units are
fraction of the size of the whole plot.\nDefault is 0.05";
(**)
SmallLineReductionFactor::usage =
  "SmallLineReductionFactor is an option to the PlotFeasibleRegion function.\n
The constraint lines are always a fixed fraction of the thickness of the
feasible region boundary lines. This option gives the reduction factor.\nA
factor of 3 means that the constraint lines are one-third the thicness of the
boundary lines. (Note, the reciprocal action!)\nDefault is 3.\nSee also

```

```

ShowCompleteLines, ColorForLine, ColorCycleAmount, and ThicknessForLine.";
(**)
RedundantConstraints::usage =
  "RedundantConstraints is an option to the PlotFeasibleRegion function.¥n
  PlotFeasibleRegion will always try to fit all constraints to the boundary of
  the feasible region, unless you tell it otherwise. This option specifies all
  constraints that lie outside the feasible region, and are thus redundant.
  (More accurately, all constraints that do not intersect the interior of the
  feasible region.)¥n
  The value given should be a list giving the positions of
  all redundant constraints in the constraint list. If a list of corner points
  is given in place of a list of constraints, give the implied position of the
  redundant constraints. (See the explanation for the option NonConstraints for
  an example.)¥n
  If there are no redundant constraints, the option should either
  not be specified, or given the value of the empty list.¥n
  Default is {}, the empty list.¥n
  See also ShowCompleteLines.";
(**)
ReorderPoints::usage =
  "ReorderPoints is an option to the PlotFeasibleRegion function.¥n
  When specifying the feasible region through a set of corner points, the points
  neednot be in proper order if this option is set to True. PlotFeasibleRegion
  will attempt to reorder the points sequential order around the convex hull
  that they define.";
(* END PlotFeasibleRegion Option USAGE messages END *);
(* END PlotFeasibleRegion family of messages END *)
(* ***** Messages END Messages END Messages ***** *)
(* ***** Messages END Messages END Messages ***** *)

(* ***** Attributes BEGIN Attributes BEGIN Attributes ***** *)
(* ***** Attributes BEGIN Attributes BEGIN Attributes ***** *)
(**)
Attributes[WriteToTextFile] = {Orderless};

(* ***** Attributes END Attributes END Attributes ***** *)
(* ***** Attributes END Attributes END Attributes ***** *)

(* ***** Options BEGIN Options BEGIN Options ***** *)
(* ***** Options BEGIN Options BEGIN Options ***** *)
(**)
Options[InterceptTable] = { Title->"Intersections.", Ketasu->3};
Options[BasicSolutions] = {NonBasicsToZero->False};
(****)
Options[MakeBoundarySystem] = {NonConstraints->{}};

(* ***** Options END Options END Options ***** *)
(* ***** Options END Options END Options ***** *)

(* ** Private BEGIN Private Private BEGIN Private Private *** *)
(* ** Private BEGIN Private Private BEGIN Private Private *** *)
(* ** Private BEGIN Private Private BEGIN Private Private *** *)
(*)
Begin["`Private`"]

(* ***** BEGIN BEGIN BEGIN BEGIN BEGIN BEGIN ***** *)
(* ***** BEGIN BEGIN BEGIN BEGIN BEGIN BEGIN ***** *)
(* **** Utility Functions: only in context Private ***** *)
(* **** Utility Functions: only in context Private ***** *)

(* *****BEGIN***** BEGIN ** UNProtect and Clear ** BEGIN ** BEGIN***** *)
Unprotect[PointQ, PointsQ, Points2Q, ComparatorQ, ComparatorListQ,
  COMPARATORS, VarNamesQ, SolveEqns, Equations, Eqns, ReduceToLowest,
  EqnThru2Pts, Angle, PerimeterOrder, CleanUpper, CleanLower,
  ConsecutiveConstraints, ExtremePoints, MaxXY, VerticalImplicitPlotLine,
  CorrectObjectiveValue, IntersectionPoints, MakeBoundarySystem];

```



```

Clear[PointQ, PointsQ, Points2Q, ComparatorQ, ComparatorListQ,
  COMPARATORS, VarNamesQ, SolveEqns, Equations, Eqns, ReduceToLowest,
  EqnThru2Pts, Angle, PerimeterOrder, CleanUpper, CleanLower,
  ConsecutiveConstraints, ExtremePoints, MaxXY, VerticalImplicitPlotLine,
  CorrectObjectiveValue, IntersectionPoints, MakeBoundarySystem];
(* ***** END ***** END ** UNProtect and Clear ** END END ***** *)

(* ** Attributes BEGIN Attributes BEGIN Attributes BEGIN Attributes *** *)
(**)
Attributes[EqnThru2Pts] = {Orderless};
Attributes[WriteToFile] = {Orderless};
(* ** Attributes END Attributes END Attributes END Attributes *** *)

(* ** Definitions BEGIN Definitions BEGIN Definitions *** *)
(* ** Definitions BEGIN Definitions BEGIN Definitions *** *)

(* ** BEGIN ** Argument Checks ** BEGIN ** *)
(****)
COMPARATORS = {Equal, Greater, GreaterEqual, Less, LessEqual, Unequal};
(****)
(*Use Head twice so that x[1] counts as symbol*)
VarNamesQ[vars_List, numvars_Integer:2] :=
  And[VectorQ[vars, Head@Head[#] == Symbol&], Length[vars] == numvars];
(****)
PointQ[pts_List] := And[VectorQ[pts], Length[pts] == 2];
(****)
PointsQ[pts_List] :=
  And[MatrixQ[pts], Dimensions[pts][[2]] == 2];
(****)
Points2Q[pts_List] := And[MatrixQ[pts], Dimensions[pts] == {2, 2}];
(****)
ComparatorQ[eqn_] := If[MemberQ[COMPARATORS, Head[eqn]], True, False];
(****)
ComparatorQ[eqnlis_?VectorQ] := And@@(ComparatorQ /@ eqnlis);
(****)
ComparatorListQ[eqnlis_?VectorQ] := (*All Comparators; Exactly 2 vars*)
  And[
    And@@(ComparatorQ /@ eqnlis),
    Length[Variables[#][[1]]] & /@ eqnlis // Flatten // Union] == 2
];
(* ** END ** Argument Checks ** END ** *)

(* ** BEGIN ** Solving Equation Systems ** BEGIN ** *)
(****)
Equations[sys_LineSystem] := Flatten[Cases[#, _Equal] & /@ sys][[1]];
Equations[sys_List] := Flatten[Cases[#, _Equal] & /@ sys];
(****)
Eqns[sys_LineSystem] := Equations[sys];
Eqns[sys_List] := Equations[sys];
(****)
SolveEqns[eqns_?ComparatorListQ, opts___?OptionsQ] :=
  Module[{vars, sol},
    Needs["Utilities`FilterOptions`"];
    vars = Union[Flatten[Variables[#][[1]]] & /@ eqns];
    Off[Solve::svars];
    sol = Flatten[Solve[eqns, vars, FilterOptions[Solve, opts]]];
    On[Solve::svars]; (*turn back on before returning soln*)
    sol (*return solution to equations*)
  ];
(****)
(* if variables given, USE them *)
SolveEqns[eqns_?ComparatorListQ, vars_?VarNamesQ, opts___?OptionsQ] :=
  Module[{sol, sortvars},

```

```

Needs["Utilities`FilterOptions`"];
Off[Solve::svars];
sortvars = Sort[vars];
sol = Flatten[Solve[eqns,sortvars ,FilterOptions[Solve,opts]]];
On[Solve::svars]; (*turn back on before returning soln*)
Sol (*return solutioun to equations*)
];
(* ** END ** Solving Equation Systems ** END ** *)

(* ** BEGIN ** Equations Thru 2 Pts & Reducing to Lowest Terms ** BEGIN ** *)
(***)
EqnThru2Pts::nonsym =
  "Expected symbols as variable names, but received `1` and `2` instead.";
(***)
ReduceToLowest[eqn_?ComparatorQ,vars_?VarNamesQ] :=
  Module[{gcd,x1,x2,a1,a2,num,c,s,eqnout,x,rhs,coeff},
    {x1,x2} = Sort[vars];
    gcd =
      GCD@@Flatten@{Coefficient[eqn[[1]],{x1,x2}],eqn[[2]]};
    {a1,a2}={
      Last/@(CoefficientList[eqn[[1]],#]&/@{x1,x2})/.
        {num_?NumericQ,c_?NumericQ * s_Symbol}->{num,0}/.
        {c_?NumericQ * s_Symbol, num_?NumericQ}->{0,num}/.
        { s_Symbol, num_?NumericQ}->{0,num}/.
        { num_?NumericQ, s_Symbol}->{num,0}
    };
    eqnout = ({a1,a2}/gcd).{x1,x2} == eqn[[2]]/gcd;
    (* convert '-x==0' to 'x==0' *)
    eqnout/.Equal[Times[coeff_,xxx_Symbol],rhs_]->Equal[xxx,rhs/coeff]
  ];
(***)
ReduceToLowest[eqn_?ComparatorQ] :=
  With[{vars = Union[Flatten[Variables[eqn[[1]]]]]},
    ReduceToLowest[eqn,vars]
  ];
(***)
EqnThru2Pts[pts_?Points2Q,vars_?VarNamesQ] :=
  Module[{x,y,xx1,xx2,yy1,yy2,xxx,coeff,rhs},
    {x,y} = Sort[vars];
    {{xx1,yy1},{xx2,yy2}}=pts;
    If[yy2>yy1,
      (*then return equation:*)
      (yy2-yy1)*x-(xx2-xx1)*y==(yy2-yy1)*xx2-(xx2-xx1)*yy2,
      (*else return equation:*)
      -((yy2-yy1)*x-(xx2-xx1)*y)==-((yy2-yy1)*xx2-(xx2-xx1)*yy2)
    ]/.Equal[Times[coeff_, xxx_Symbol], rhs_]->Equal[xxx,rhs /coeff]
  ];
(***)
(*For use with MapIndexed, 'pos' is position of the point-pair*)
(* Used in function 'MakeBoundarySystem' *)
EqnThru2Pts[pts_?Points2Q,{pos_Integer},vars_?VarNamesQ] :=
  {pos,EqnThru2Pts[pts,vars]};
(* ** END ** Equations Thru 2 Pts & Reducing to Lowest Terms ** END ** *)

(* ** BEGIN ** Convex Hull & Perimeter Ordering ** BEGIN ** *)
(***)
Angle[a_?PointQ,b_?PointQ] := (* avoids indeterminate forms *)
  Module[{Embed3D,anglevalue,angledirection},
    Embed3D[{x_?AtomQ,y_?AtomQ}] := {x,y,0};
    (* absolute value of angle *)
    anglevalue = ArcCos[(a.b)/Sqrt[Cross[a.a,b.b]]];
    angledirection = (* sign of angle direction ccw=plus *)
      Module[{aa=Embed3D[a], bb=Embed3D[b]},

```

```

      Sign@ArcSin[Last[ Cross[aa,bb]/(aa.aa*bb.bb)] ]
];
(* return signed angle... plus is CounterClockWise*)
angledirection * anglevalue
];
(****)
PerimeterOrder[ptlis_?MatrixQ] :=
Module[{conhull},
  Off[Arg::indet];
  conhull = ptlis[[ConvexHull[ptlis]]];
  (*needs DiscreteMath`ComputationalGeometry` package*)
  On[Arg::indet];
  If[NotEqual[Length[conhull],Length[ptlis]],(*interior `vertex`!*)
    Message[PlotFeasibleRegion::inhull,ptlis,Complement[ptlis,conhull]];
  ];
  (*return the convex hull in counterclockwise, perimeter order*)
  conhull
];
(* ** END ** Convex Hull & Perimeter Ordering ** END ** *)

(* ** BEGIN ** CleanUpper & CleanLower ** BEGIN ** *)
(****)
CleanUpper[x_?NumericQ] :=
Module[{rdigs,d1,d2,d3,clog,alpha},
  rdigs = RealDigits[x//N];(*N guarantees 6 real digits*)
  {d1,d2,d3}=Take[rdigs[[1]],3];
  (*leading 3 digits*)
  clog=rdigs[[2]]-1;
  (*ceiling of Log x, base10*)
  alpha = 10^(clog-2);
  Which[
    10d2+d3 <=48,      (* go to 15 + alpha *)
      d1*10^clog + 5*10^(clog-1)+ alpha,
    10d2+d3 <=98,      (* go to 20 + alpha *)
      2*d1*10^clog+ alpha,
    True,               (* go to 25 + alpha *)
      2*d1*10^clog + 5*10^(clog-1)+ alpha
  ]//N
];
(****)
CleanLower[x_?NumericQ] :=
Module[{rdigs,d1,d2,d3,clog},
  rdigs = RealDigits[x//N];(*N guarantees 6 real digits*)
  {d1,d2,d3}=Take[rdigs[[1]],3];(*
  leading 3 digits*)
  clog=rdigs[[2]]-1;
  (*ceiling of Log x, base10*)
  Which[
    10d2+d3 >=52,      (* go to 1.5 times d1 *)
      d1*10^clog + 5*10^(clog-1),
    10d2+d3 >= 2,      (* go to d1 *)
      d1*10^clog,
    True,              (* go to 1/2 of d1 *)
      0.5*d1*10^clog
  ]//N
];
(* ** END ** CleanUpper & CleanLower ** END ** *)

(* ** BEGIN ** ConseqConstr, XtremePts, MaxXY, VertLines ** BEGIN ** *)
(*ConsecutiveConstraints Function*)
(*We must remove the eqns for those FR sides which are not constraints.*)
(*But, rather are plot limits for unbounded sides of an unbounded FR. *)
(*We do not wish to draw lines or intersection points for these. *)

```

```

(*But, rather to find intersection points for actual constraints, the *)
(*constraints must be placed in a list in the correct, consecutive *)
(*order. If the unbounded sides come in the middle, we make our new *)
(*reordered) list by starting with the first constraint eqn after the *)
(*non-constraint eqns (the 'gap'), and continuing from the front of *)
(*the list, until we have included all actual constraint equations. *)

(*BEGIN ConsecutiveConstraints definitions*)
(*Case of bounded FR... Nothing to do here, return the eqn list as is *)
(****)
(*leave as-is, all sides are constraining*)
ConsecutiveConstraints[allEqns_?(VectorQ[#,Head[#]==Equal&]&),{}]:= AllEqns;

(*Case of unbounded FR. Remove non-constraints, put rest in conseq order *)
(*Evaluated only when 3rd arg not empty list: at least 1 non-constraint *)
(****)
ConsecutiveConstraints[
  allEqns_?(VectorQ[#,Head[#]==Equal&]&),
  nonConstraintPos_?(VectorQ[#,NumericQ]&)
] :=
Module[{labels,pre,post},
  (*get indices of constraints before and after the gap*)
  labels = Range[Length[allEqns]];
  pre = Take[labels,nonConstraintPos[[1]]-1];
  post = Take[labels,-(Length[allEqns]-nonConstraintPos[[-1]])];
  (*make a new list of constraints only, in consecutive order*)
  If[And[pre=={},post=={}],
    allEqns,
    allEqns[[Join[post,pre]]],
    allEqns[[Join[post,pre]]]
  ]
];
(*END of ConsecutiveConstraints definitions*)
(*BEGIN ExtremePoints definitions*)
(****)
(*if 2nd arg is not empty list, then at least one non-constraint side*)
(*Case 1: all sides constraints, close the circle of equations*)
ExtremePoints[conseqconstraints_,{x_,y_},{}]:=
Module[{result,eqnlis=Join[conseqconstraints,{conseqconstraints[[1]]}],
  Off[Solve::"svars"];
  result=Point/@({x,y}/.Flatten[Solve[#]]&/@ Partition[eqnlis,2,1]);
  Off[Solve::"svars"];
  result
];
(****)
(*Case 2: at least 1 side not constraint, do not close the circle*)
ExtremePoints[conseqconstr_,{x_,y_},nonconstraintpos_]:=
Module[{result},
  Off[Solve::"svars"];
  result =Point/@({x,y}/.Flatten[Solve[#]]&/@Partition[conseqconstr,2,1]);
  Off[Solve::"svars"];
  result
];
(*END of ExtremePoints definitions*)

(*BEGIN MaxXY definitions*)
(*Normally, we calculate upper plot limits for the user *)
(*However, if the user uses the option PlotRange to designate *)
(*upper limits, we want all graphics to use those *)
(****)
MaxXY[pts_,plotrange_] :=
Module[{maxx,maxy,bigx,bigy},
  {maxx,maxy} = {Max@#[[1]],Max@#[[2]]}&@Transpose[pts];

```

```

{bigx,bigy} = CleanUpper/@{maxx,maxy};
If[plotrange==Automatic,(*default value*)
  (*then both to cleanupper*)
  {bigx,bigy} = CleanUpper/@{maxx,maxy},
  (*else, check x, if Automatic, use default *)
  If[plotrange[[1]]==Automatic,
    bigx = CleanUpper@maxx,
    bigx = plotrange[[1,-1]],
    bigx = plotrange[[1,-1]]
  ];
(*AND, check y (still in case else)*)
If[plotrange[[2]]==Automatic,
  bigy = CleanUpper@maxy,
  bigy = plotrange[[2,-1]],
  bigy = plotrange[[2,-1]]
],
(*default, do it all again..check x *)
If[plotrange[[1]]==Automatic,
  bigx = CleanUpper@maxx,
  bigx = plotrange[[1,-1]],
  bigx = plotrange[[1,-1]]
];
(*AND, check y (still in case default)*)
If[plotrange[[2]]==Automatic,
  bigy = CleanUpper@maxy,
  bigy = plotrange[[2,-1]],
  bigy = plotrange[[2,-1]]
]
];
{bigx,bigy}
];
(*END of MaxXY definitions*)
(*BEGIN VerticalImplicitPlotLine definitions*)
(* A vertical constraint line is not handled by ImplicitPlo*)
(* It only produces an error message, suppressed in this notebook*)
(* The FR boundary lines are produced, since they are graphics running*)
(* between intersection points. But, in order to produce the constraint*)
(*line outside the FR, we need a special procedure.*)
(*The procedure plots x==const as y==const, then exchanges vars again*)
(****)
VerticalImplicitPlotLine[Equal[x1_,c1_],{x2_,from_,to_}]:=
  With[{plt = ImplicitPlot[x1==c1,{x2,from,to},DisplayFunction->Identity]},
    Print[StringForm["plt[[1,1,1,-1]] = ``",plt[[1,1,1,-1]]//InputForm]];
    Graphics[{
      (*need only first & last points of line: reverse x1 & x2*)
      Line[Reverse /@ {#[[1]],# [[-1]]}&[ plt[[1,1,1,-1,1]]]]
    }]
];
(****)
VerticalImplicitPlotLine[Equal[x1_,c1_],{x2_,from_,to_},opts_?OptionQ]:=
  With[{plt=ImplicitPlot[x1==c1,{x2,from,to},opts,DisplayFunction->Identity]},
    Graphics[{
      (*need only first & last points of line: reverse x1 & x2*)
      Sequence@@(plt[[1,1,1]]//Reverse//Rest),
      Line[Reverse /@ {#[[1]],# [[-1]]}&[plt[[1,1,1,-1,1]]]]
    }]
];
(****)
VerticalImplicitPlotLine[{Equal[x1_,c1_]},{x2_,from_,to_},opts_?OptionQ]:=
  VerticalImplicitPlotLine[Equal[x1,c1],{x2,from,to},opts];
(****)
VerticalImplicitPlotLine[allelse_] := Graphics[{}];
(* ** END ** ConseqConstr, XtremePts, MaxXY, VertLines ** END ** *)

```

```

(* ** BEGIN ** CorrectObjectiveValue ** BEGIN ** *)
(*About the function CorrectObjectiveValue: *)
(* The PlotFeasibleRegion function allows *)
(*the user to also plot one or more levels of the objective*)
(*function as an option. *)
(*The user must specify the objective function as a standard line.*)
(*the form is: ObjectiveFunction->{12x1 + 9 x2 == 9} *)
(*If multiple levels of the objective (more than 1 line) are desired, *)
(*another option is specified. For example *)
(* ObjectiveValue -> {9,12,15,18,21} *)
(*will print five different objective function lines, *)
(*The CorrectObjectiveValue function processes these options*)
(*The arguments are ObjectiveValue and ObjectiveFunction, in that order *)
(****)
CorrectObjectiveValue[Automatic, None] := Automatic;
(****)
CorrectObjectiveValue[Automatic, Equal[lhs_, rhs_] ] := {rhs} // Flatten;
(****)
CorrectObjectiveValue[objectivevalue_, _] := {objectivevalue} // Flatten;
(* ** END ** CorrectObjectiveValue ** END ** *)

(* Utility Functions, defined and used only in context Private *)
(* Utility Functions, defined and used only in context Private *)
(* ***** END END END END END END END ***** *)
(* ***** END END END END END END END ***** *)

(* ***** BEGIN BEGIN BEGIN BEGIN BEGIN BEGIN ***** *)
(* ***** BEGIN BEGIN BEGIN BEGIN BEGIN BEGIN ***** *)
(* **** Functions for USER, available in context Global` ** ** *)
(* **** Functions for USER, available in context Global` ** ** *)

(* ** BEGIN ** MakeBoundarySystem & IntersectionPoints ** BEGIN ** *)
(****)
IntersectionPoints[sys_LineSystem] :=
  Flatten[#[[3]] & /@ sys[[1]], 1] // Union // PerimeterOrder;
(****)
IntersectionPoints[sys_List] :=
  Flatten[#[[3]] & /@ sys, 1] // Union // PerimeterOrder;
(* BEGIN Fundamental Definition *)
MakeBoundarySystem[
  boundary:{pt__?Points2Q}, vars_?VarNamesQ,
  opts___?OptionQ
]:=
Module[{x,y,equationsThruPointPairs,xxx,curline,linenum,insertblank,
  axeslocs,constraintlocs,bdrylabels},
  {nonconstraints} = {NonConstraints} /.
    {opts} /. Options[PlotFeasibleRegion];
  {x,y} = Sort[vars];
  equationsThruPointPairs =
  Map[{EqnThru2Pts[#, {x,y}], #] &, boundary];
  (*make the boundary labels*)
  (*Get the positions of three catagories: *)
  (* axes, nonconstraints, nonaxes-constraints*)
  (*the nonconstraint locs given by user in option are *)
  (*bounds for plotting unbounded FR *)
  bdrylabels=Range@Length[boundary];
  axeslocs = Position[First/@equationsThruPointPairs, _Symbol==0] // Flatten;
  If[Or[nonconstraints==None, nonconstraints==Automatic],
    nonconstraints={},
    nonconstraints=nonconstraints, (*leave as-is*)
    nonconstraints=nonconstraints
  ];
];

```

```

If[(*if axes list and nonconstraint list overlap, warning message!*)
  NotEqual[Intersection[axeslocs, nonconstraints ],{}],
  Message[
    PlotFeasibleRegion::overlap,
    TableForm[{
      Join[{isAxis}, axeslocs],
      Join[{isNotConstraint}, nonconstraints]
    }]
  ](* no else clause*)
];(*end if*)
bdrylabels=
  bdrylabels/.
    ( #->isNotConstraint& /@ nonconstraints)/.
    ( #->isAxis& /@ axeslocs)/.
    comp = Complement[bdrylabels, Union[nonconstraints, axeslocs]];
    Thread[(*number anything not tagged (restart from 1) *)
      Rule[ (*existing positions go to 1,2,3...*)
        comp,
        Range[Length[comp]] ](*end Rule*) ](*end Thread*) ];
(*thread the labels in as first element of line system elements*)
equationsThruPointPairs=
  Thread@Join[{bdrylabels, equationsThruPointPairs}]/.
  Flatten[#,2]& // Partition[#,3]&;
(*reduce to lowest, put axis intercepts, wrap in head LineSystem*)
LineSystem@
  Map[(*the following function (a 4-element list):*)
    #[[1]], (* Sequence Number in this list *)
    ReduceToLowest[#[[2]],{x,y}], (* Equation *)
    #[[3]], (* Intersect w/ Adjacent Eqns *)
    {
      (* Axis Intercepts of Equation *)
      {x,0}/.Solve[{#[[2]],y==0},x],
      {0,y}/.Solve[{#[[2]],
        x==0},y]
    }//Flatten[#,1]&
  ]&, (* end of 4-element list function*)
  equationsThruPointPairs (*map function onto this*)
] (*end of Map*)
];
(* END Fundamental Definition *)
(****)
MakeBoundarySystem[eqIneqLis_?ComparatorListQ,opts___?OptionQ] :=
  Module[{eqns,vars,pts,ptprs,midpt,ineqs},
    eqns = Equal@@@#& /@ eqIneqLis;
    vars = Union[Flatten[Variables[#[[1]]]&/@eqns]];
    pts = vars/.SolveEqns[#]&/@Partition[Join[{eqns[[-1]]},eqns],2,1];
    ptptrs = Partition[Join[pts,{pts[[1]]}],2,1];
    (*Are any inequalities violated? if so, warn user (ignore equations)*)
    (*Find midpt of corners. test against any inequalities in the list *)
    (* (cannot be sure equalities are meant to be such, so ignore them)*)
    midpt = Plus@@pts/Length[pts]; (*the midpoint of the line intersections*)
    If[(*Did user give an inequality that is violated by the midpoint?*)
      Not[ (*any cases of False? (not all True?)*
        And@@(*T/F for midpt against each inequality*)
          #/.Thread[{x,y}->midpt]&/@(ineqs>DeleteCases[eqIneqLis,_Equal]])
    ],
    (*if so, warn user at least one inequality is violated*)
    Message[PlotFeasibleRegion::badfr,
      Select[ineqs,Not[#/.Thread[{x,y}->{19/2,5}]]&],midpt
    ]
  ]; (*end if*)
  (*pass point pairs defining boundary line segments, *)
  (* along with vars & opts*)
  MakeBoundarySystem[ptprs,vars,opts]

```

```

];
(****)
MakeBoundarySystem[eqIneqLis_?ComparatorListQ,vars_?VarNamesQ,
                    opts___?OptionQ] :=
Module[{eqns,sortvars,pts,ptprs,midpt},
  eqns = Equal@@#& /@ eqIneqLis;
  sortvars = Sort[vars];
  pts = sortvars/.SolveEqns[#]&/@Partition[Join[{eqns[[-1]]},eqns],2,1];
  ptptrs = Partition[Join[pts,{pts[[1]]}],2,1];
  (*Are any inequalities violated? if so, warn user (ignore equations)*)
  (*Find midpt of corners. test against any inequalities in the list *)
  (* (cannot be sure equalities are meant to be such, so ignore them)*)
  midpt = Plus@@pts/Length[pts]; (*the midpoint of the line intersections*)
  If[(*Did user give an inequality that is violated by the midpoint?*)
    Not[ (*any cases of False? (not all True?)*
      And@@( (*T/F for midpt against each inequality*)
        #/.Thread[{x,y}->midpt]&/@(ineqs>DeleteCases[eqIneqLis,_Equal]))
    ],
    (*if so, warn user at least one inequality is violated*)
    Message[MakeBoundarySystem::badfr,
      Select[ineqs,Not[#/.Thread[{x,y}->{19/2,5}]]&],
      midpt
    ]
  ]; (*end if*)
  (*pass point pairs defining boundary line segments, *)
  (* along with vars & opts*)
  MakeBoundarySystem[ptprs,sortvars,opts]
];
(****)
MakeBoundarySystem[ptlis_?(VectorQ[#,PointQ]&),vars_?VarNamesQ,
                    opts___?OptionQ] :=
Module[{sortvars,ptprs,rearrange},
  sortvars = Sort[vars];
  ptptrs=Partition[Join[ptlis,{ptlis[[1]]}],2,1];
  MakeBoundarySystem[ptprs,sortvars,opts]
];
(* ** END ** MakeBoundarySystem & IntersectionPoints ** END ** *)

(* ** BEGIN ** WriteToTextFile ** BEGIN ** *)
(****)
WriteToTextFile[data_List, filename_String] :=
Module[{fil,dir},
  fil = OpenWrite[filename];
  dir = Directory[];
  Print[StringForm[
    "Writing data to text file with pathname¥n¥t `\"",
    StringJoin[dir,":",filename]
  ]];
];
Scan[
  ( WriteString[fil,First[#]];
    Scan[
      WriteString[fil,"¥t",#]&,
      Rest[#]
    ];
    WriteString[fil,"¥n"]
  )&,
  data
];
Close[fil]
];
(****)
WriteToTextFile[linesys_LineSystem, fil_String] :=
With[(*local quantities:*)

```



```

data = MapAt[ (*fractions mess up the write to disk*)
  If[MemberQ[{Integer}, Head[#]], #, N[#/3]] &,
  #,
  {{3}, {4}}
] & /@ linesys[[1]],
headings = {"No.", "Eqn", "x1", "y1", "x2", "y2", "x-int", "-", "-", "y-int"}
],
WriteToTextFile[Join[{headings}, Flatten/@data], fil]
];
(* ** END ** WriteToTextFile ** END ** *)

(* ** BEGIN ** InterceptTable ** BEGIN ** *)
(* InterceptTable: Output is a table meant to be exported*)
(* to an Excel spreadsheet. First line is a title,*)
(****)
(* Version to use with LineSystem object *)
InterceptTable[linesys_LineSystem, vars_?VarNamesQ, opts___?OptionQ] :=
  With[{sortvars = Sort[vars], eqlis = #[[2]] & /@ linesys[[1]]},
    InterceptTable[eqlis, sortvars, opts]
  ];
(****)
InterceptTable[linesys_LineSystem, opts___?OptionQ] :=
  Module[{eqlis, vars},
    eqlis = #[[2]] & /@ linesys[[1]],
    vars = Union[Flatten[Variables[#[[1]]] & /@ eqlis]],
    InterceptTable[eqlis, vars, opts]
  ];
(****)
(* If vars not given as arg, extract and pass *)
InterceptTable[eqlis_?ComparatorListQ, opts___?OptionQ] :=
  With[{vars = Union[Flatten[Variables[#[[1]]] & /@ eqlis]]},
    InterceptTable[eqlis, vars, opts]
  ];
(****)
InterceptTable[boundpts: {pt__?PointQ},
  vars_?VarNamesQ,
  opts___?OptionQ ] :=
  Module[{sortvars, mbsopts, itopts},
    sortvars = Sort[vars];
    mbsopts = FilterOptions[MakeBoundarySystem, opts];
    itopts = FilterOptions[InterceptTable, opts];
    InterceptTable[
      MakeBoundarySystem[boundpts, sortvars, mbsopts],
      sortvars,
      itopts
    ]
  ];
(****)
(* Version to use with list of Equations and Variables *)
InterceptTable[complis_?ComparatorListQ, vars_?VarNamesQ, opts___?OptionQ] :=
  Module[{eqlis, sortvars, n, intstable, colhead1, colhead2, title, ketasuu},
    {title, ketasuu} = {Title, Ketasuu} /. {opts} /. Options[EqnInterceptTable];
    (* check ketasuu, it must be a positive integer*)
    ketasuu = If[MemberQ[{Integer}, And[Head[ketasuu]], ketasuu > 0],
      ketasuu, (*is okay, else use default:*)
      ketasuu /. Options[InterceptTable], (*default*)
      ketasuu /. Options[InterceptTable] (*default, by gum*)
    ];
    (*ComparatorListQ insures exactly 2 variables*)
    eqlis = Equal@@# & /@ complis; (*convert all to equatons*)
    sortvars = Sort[vars];
    n = Length[eqlis];
    intstable = Array[{"-", "-"} &, {n, n}];

```

```

Do[ intstable[[i,j]] =sortvars /.
  SolveEqns[{eqlis[[i]],eqlis[[j]]},sortvars,FilterOptions[Solve,opts]],
  {i,1,n},{j,i+1,n}
]/.{$Coincident->"same",$NoIntersection->"none",Fail->"fail"};
(* remove col for eqn 1, flatten rows, fractions->numbers *)
intstable = Map[If[MemberQ[{Integer}, Head[#]],#,N[#,ketasuu]]&,
  Flatten/@Rest/@intstable,
  {2 }
];
(* Next, insert eqn nbr and eqn at start of each row *)
intstable = MapIndexed[Join[#2,{eqlis[[Sequence@@#2]]},#1]&,intstable];
(* Make headings for the columns. *)
(* Repeat each eqn number, once each for x and y coords *)
colhead1=Join[{"","Eqn No.->"},Thread[{Range[2,n],Range[2,n]}]]//Flatten];
colhead2=Join[{"No.", "Equation"},Table[Sort[vars],{n-1}]]//Flatten];
Join[{{title}},{colhead1},{colhead2},intstable]
];
(* ** END ** InterceptTable ** END ** *)

(* ** BEGIN ** BasicSolutions ** BEGIN ** *)
(****)
BasicSolutions[eqns_List,vars_List,opts___?OptionQ] :=
Module[{iBasics,iNonBasics,iSetPairs,nonbasicstozero},
  {nonbasicstozero}={NonBasicsToZero}/.{opts}/.Options[BasicSolutions];
  iBasics=
  Union[Sort/@(
    Take[#,Length[eqns]]&/@ Permutations[Range[Length[vars]]]]];
  iNonBasics=Complement[Range[Length[vars]], # ]& /@ iBasics;
  zeroVec = Array[0&,Length[vars]-Length[eqns]];
  iSetPairs= Thread[{iBasics,iNonBasics}];
  If[nonbasicstozero == False,
    Flatten[Solve[eqns,vars[[#]]]&/@iBasics,
      Flatten[Solve[eqns,vars[[#[[1]]]]]]]/.
      Thread[vars[[#[[2]]]]->zeroVec] & /@ iSetprs,
    Flatten[Solve[eqns,vars[[#[[1]]]]]]]/.
      Thread[vars[[#[[2]]]]->zeroVec] & /@ iSetprs,
  ]
];
(* ** END ** BasicSolutions ** END ** *)

(* ** BEGIN ** PlotFeasibleRegion ** BEGIN ** *)
(* ** BEGIN ** PlotFeasibleRegion ** BEGIN ** *)
(*Auxilliary definitions:*)
(****)
(*if already have a fully defined LineSystem object*)
PlotFeasibleRegion[sys_LineSystem, opts___?OptionQ] :=
  With[{vars = Union[Flatten[Variables[#[[1]]]&/@Eqns[sys]]]},
    PlotFeasibleRegion[sys,vars,opts]
  ];
(****)
(*next, for lists of (in)equailities (sorted out in MakeBoundarySystem)*)
PlotFeasibleRegion[eqIneqLis_?ComparatorListQ,vars_?VarNamesQ,
  opts___?OptionQ] :=
  PlotFeasibleRegion[MakeBoundarySystem[eqIneqLis,Sort@vars],Sort@vars,opts];
(****)
PlotFeasibleRegion[eqIneqLis_?ComparatorListQ,opts___?OptionQ] :=
  PlotFeasibleRegion[MakeBoundarySystem[eqIneqLis,opts],opts];
(****)
(*for sets of coordinates of corner points*)
PlotFeasibleRegion[boundpts:{pt___?PointQ},vars_?VarNamesQ,opts___?OptionQ]:=
  Module[{reorderpoints,ptlis},
    reorderpoints=ReorderPoints/.{opts}/.Options[PlotFeasibleRegion];

```

```

If[reorderpoints==True,
  ptlis = PerimeterOrder[boundpts],
  ptlis = boundpts,
  ptlis = boundpts
];
PlotFeasibleRegion[MakeBoundarySystem[ptlis,Sort@vars],opts]
];
(****)
PlotFeasibleRegion[boundptprs:{pt__?Points2Q},vars_?VarNamesQ,
  opts___?OptionQ]:=
  PlotFeasibleRegion[MakeBoundarySystem[boundptprs,Sort@vars],opts];
(****)
(*Primary definition:*)
PlotFeasibleRegion[sys_LineSystem,vars_?VarNamesQ, opts___?OptionQ] :=
  Module[{x,y,syslis,allEqns,offsetright, bigx,bigy,
    maxx,maxy,labelNumsLocsOffsets,
    lblsize,lblcolor,linecolor,regioncolor,
    pointcolor,linethickness,pointsize,labeloffset,
    boundaryLines,extremePts,interiorPoly,axisPts,
    boundaryLinesPlot,completeLinesPlot,graffik,
    emptyplot,showlines,nonconstraints,bdryEqns,
    bdrylabels,nonboundpos,axispos,boundpos,constraintpos,
    post,pre,conseqconstraints,
    objectivefunction,objectivevalue,objectivePlot,
    colorforobjective,plotstyleforobjective,thicknessforobjective,
    constraintlabels,labelsToPrint,extremePointsPlot,
    showboundarylines,showextremepoints,shadefr,
    interiorPolyPlot,lineLabelsPlot,completeLinesBoundaryOverlayPlot,
    vertLineEqn,vertLinePlot,thinLineStyle,
    redundantconstraints,tableToPrint
  },
  Needs["Graphics`ImplicitPlot`"];
  Needs["Utilities`FilterOptions`"];
  Needs["Graphics`Colors`"];
  {lblsize,lblcolor,linecolor,regioncolor, pointcolor,
    linethickness, pointsize,linereduce, colorcycle,
    labeloffset, reorderpoints,showlines,
    showlinelabels,constraintlabels,
    objectivefunction,objectivevalue,colorforobjective,
    plotstyleforobjective, thicknessforobjective,
    showboundarylines,showextremepoints,shadefr,
    redundantconstraints, nonconstraints, plotrange} =
  {SizeForLabel,ColorForLabel,ColorForLine,ColorForRegion,
    ColorForPoint,ThicknessForLine,SizeForPoint,
    SmallLineReductionFactor, ColorCycleAmount,
    LineLabelsOffset,ReorderPoints, ShowCompleteLines,
    ShowLineLabels,ConstraintLabels,
    ObjectiveFunction,ObjectiveValue,ColorForObjective,
    PlotStyleForObjective,ThicknessForObjective,
    ShowBoundaryLines,ShowExtremePoints,ShadeFR,
    RedundantConstraints,NonConstraints,PlotRange}/.
    {opts}/.Options[PlotFeasibleRegion]/.
    Options[Plot];

  {x,y} = Sort[vars];
  syslis=sys//First;
  (*if objective function has been given, strip any list brackets from it*)
  objectivefunction={objectivefunction}//Flatten//First;
  (*if not set by user, set 'objectivevalue' to rhs of the obj funct*)
  objectivevalue=CorrectObjectiveValue[objectivevalue,objectivefunction];
  (*note we select only real boundaries.. not unbounded sides*)
  (*syslis may include lines meant only as plot limits for unbounded FR*)
  (*user specifies these in option NonConstraints, their labels are ""*)
  (*the real bounding sides have number labels*)

```

```

bdrylabels = #[[1]]& /@ syslis;
(*tag for Axis is 'isAxis'; Tag for non-constraint is 'isNotConstraint'*)
nonboundpos = Position[bdrylabels,isNotConstraint]//Flatten;
axespos = Position[bdrylabels,isAxis]//Flatten;
constraintpos = Position[
    bdrylabels, (*must also exclude Head 'List'*)
    ?(Not@MemberQ[{isNotConstraint,isAxis,List},#]&)
]//Flatten;
boundpos = Union[axespos,constraintpos];
(*pick out FR bdrys: constraints and axes; relative order is unchanged*)
boundaryLines = Line/@Union#[[3]]&/@syslis[[boundpos]];
allEqns = Equal@@#&/@([2]]&/@syslis);(*insure all are equalities.*)
bdryEqns = Equal@@#&/@([2]]&/@syslis[[boundpos]]);(*only actual bdrys*)
vertLineEqn =
    Cases[bdryEqns,Equal[x,const_?(NumericQ[#]_#_0&)];(*vert, not axis*)
    n = Length[allEqns]; (*number eqns given, includes nonbounding sides*)
(*offsets for positioning the boundary labels*)
offsetright[{xx_,yy_},frac_?NumericQ] :=frac{yy,-xx};
offsetright[{xx_,yy_},{frac_?NumericQ}] :=frac{yy,-xx};
labeloffset = If[MemberQ[{List},Head[labeloffset]],
    labeloffset,
    Table[labeloffset,{Length[bdryEqns]}],
    Table[labeloffset,{Length[bdryEqns]}]
];
(*shading the FR: include any non-constraint lines in unbounded FR*)
interiorPoly =
    Polygon@PerimeterOrder[Union#[[3]]&/@syslis]//Flatten[#,1]&;
(*extremePts are extreme points of FR: only constraint intersections*)
(*order the eqns so no unbounded gaps intervene, then find intersect'ns*)
conseqconstraints = ConsecutiveConstraints[allEqns,nonboundpos];
extremePts= ExtremePoints[conseqconstraints,{x,y},nonboundpos];
(*we also need the intersections with the axes*)
axisPts = Cases[Union#[[4]]& /@syslis]//Flatten[#,1]&,{_,_}];
(*find largest values to plot, calc best upper plot range limits*)
{bigx,bigy} = MaxXY#[[3]]&/@syslis,plorange];(*int'n pts of eqns*)
(*Do number of labels matches number of constraints? *)
(*If not, reset labels to Automatic*)
If[(*bad label list*)
    And[Head[constraintlabels]==List,
    NotEqual[Length[constraintlabels],Length[constraintpos]]],
    (*then*)
    Message[PlotFeasibleRegion::badlbls,constraintlabels,
        allEqns[[constraintpos]]];
    constraintlabels = Automatic
    (*else nothing*)
];
(*no user spec for constraint labels? Then, use numbers made above*)
labelsToPrint=
    Switch[constraintlabels,
        Automatic, bdrylabels[[constraintpos]],
        None,      Table["",{Length[constraintpos]}],
        False,     Table["",{Length[constraintpos]}],
        True,       bdrylabels[[constraintpos]],
        Constraints|Equations|ConstraintEquations,
            StringReplace[#, " ">""]&/@
                ToString/@allEqns[[constraintpos]],
            constraintlabels
    ];
tableToPrint=
    Switch[constraintlabels,
        (*if constraints are their own labels, just number constraints*)
        Constraints|Equations|ConstraintEquations,
            MapThread[{#1,Sequence@@Rest[#2]}&,

```

```

        {Range[Length[constraintpos]], syslis[[constraintpos]]}
    ],
    (*otherwise, labels are indeed labels, include them in table*)
    MapThread[{#1, Sequence@@Rest[#2]} &,
        {labelsToPrint, syslis[[constraintpos]]}
    ]
];
(*how much to offset each constraint label from its line*)
labelNumsLocsOffsets=
    MapIndexed[
        { #1[[1]], (*constraint number*)
          ((#1[[1]]+#1[[2]])/2&[#1[[3]]]), (*loc coords*)
          offsetright[#1[[3,2]]-#1[[3,1]],
            RotateLeft[labeloffset, (#2)-1][[1]]] (*offset*)
        } &,
        (*replace auto generated nbrs with users labels, if needed*)
        MapThread[{#1, Sequence@@Rest[#2]} &,
            {labelsToPrint, syslis[[constraintpos]]}
        ]
    ];
(*This section constructs Graphic objects needed for final display*)
(*Graphics object of the constraint line segments bounding the FR*)
(*these are thicker, and will lie behind the completeLinesPlot, below*)
interiorPolyPlot =
    If[shadefr==True,
        Graphics[{{regioncolor, interiorPoly}}],
        Graphics[{}],
        Graphics[{}]
    ];
boundaryLinesPlot =
    If[showboundarylines==True,
        Graphics[{Thickness[linethickness], linecolor, boundaryLines}],
        Graphics[{}],
        Graphics[{}]
    ];
extremePointsPlot =
    If[showextremepoints==True,
        Graphics[{PointSize[pointsize], pointcolor, extremePts}],
        Graphics[{}],
        Graphics[{}]
    ];
lineLabelsPlot =
    If[showlinelabels==True,
        Graphics[{
            Text[#[[1]],#[[2]],#[[3]],
                TextStyle->{FontSize->lblsize, FontColor->lblcolor}
            ] & /@ labelNumsLocsOffsets
        ]],
        Graphics[{}],
        Graphics[{}]
    ];
(*Turn off messages and make empty plot used as a background *)
Off[Plot::plnr];
emptyplot =
    Plot[-Infinity*x, {x, 0, bigx}, AxesLabel->{x, y},
        FilterOptions[Plot, opts]//Evaluate,
        FilterOptions[Plot, Options[PlotFeasibleRegion]]//Evaluate,
        PlotRange->{{0, bigx}, {0, bigy}},
        AxesLabel->{x, y},
        DisplayFunction->Identity
    ];
On[Plot::plnr];
(*make the objective function plot *)
Off[ImplicitPlot::var];

```

```

objectivePlot =
  If[Head[objectivefunction]==Equal,
    (*then*)
    ImplicitPlot[(*as many as elems in objectivevalue *)
      FoldList[#1[[1]]==#2&,objectivefunction,objectivevalue]//Rest,
      {x,0,bigx},
      FilterOptions[ImplicitPlot, opts]//Evaluate,
      FilterOptions[ImplicitPlot, Options[PlotFeasibleRegion]]//Evaluate,
      PlotStyle->{{Thickness[linethickness], colorforobjective,
        Dashing[{0.03,0.03}]}}},
      AxesLabel->{x,y},DisplayFunction->Identity
    ],(*end then*)
    (*else and default, both empty plot*)
    Graphics[{}],
    Graphics[{}]
  ];
On[ImplicitPlot::var];
(*plot the complete constraint lines in thinner & brighter color*)
(*two steps, non-vertical, then vertical lines *)
thinLinePlotStyle =
  {{Thickness[linethickness/2],Mod[#+colorcycle,1]&/@linecolor}};
Off[ImplicitPlot::var];
completeLinesPlot = (*all but the vertical lines*)
  If[showlines==True,
    (*then*)
    ImplicitPlot[
      allEqns[[Position[bdrylabels,_?NumericQ]//Flatten]],
      {x,0,bigx},
      FilterOptions[ImplicitPlot, opts]//Evaluate,
      FilterOptions[ImplicitPlot,Options[PlotFeasibleRegion]]//Evaluate,
      PlotStyle->thinLinePlotStyle, AxesLabel->{x,y},
      DisplayFunction->Identity
    ],
    (*else*) emptyplot,
    (*default*) emptyplot
  ];
vertLinePlot =
  VerticalImplicitPlotLine[vertLineEqn,{y,0,bigy},
    PlotStyle->thinLinePlotStyle,DisplayFunction->Identity
  ];
completeLinesBoundaryOverlayPlot =
  If[showlines==True,
    (*then make thin lines to overlay thick boundary lines*)
    Graphics[{Mod[#+colorcycle,1]&/@linecolor,
      Thickness[linethickness/linereduce],
      boundaryLines
    }],
    (*else and default, both empty plot*)
    Graphics[{}],
    Graphics[{}]
  ];
Off[Graphics::gptn];
graffik=Show[
  (*lay on thin lines foe each constraint equation*)
  completeLinesPlot,vertLinePlot,
  (*color the feasible region interior*)
  interiorPolyPlot,
  (*put on the thick lines bounding the feasible region*)
  boundaryLinesPlot,
  (*the thin lines have been covered by FR interior and boundary*)
  (*redraw thin lines over thick boundary lines*)
  completeLinesBoundaryOverlayPlot,

```

```

(*show the labels as requested by the user*)
  lineLabelsPlot,
(*show the extreme points of the feasible region*)
  extremePointsPlot,
(*overlay the graphic of the OF last (may be blank)*)
  objectivePlot,
(*options follow:*)
  FilterOptions[Plot,opts],
  DisplayFunction->$DisplayFunction,
  PlotRange->{{-1,bigx},{-1,bigy}}
];
On[Graphics::gptn];
(*Print[
  MatrixForm[
    MapThread[{#1,Sequence@@Rest[#2]]&,{labelsToPrint,
      syslis[[constraintpos
        ]]]];*)
Print[MatrixForm[tableToPrint]];
graffik (**)
];
(* ** END ** PlotFeasibleRegion ** END ** *)
(* ** END ** PlotFeasibleRegion ** END ** *)

(* **** Functions for USER, available in context Global` ** *** *)
(* **** Functions for USER, available in context Global` ** *** *)
(* **** END END END END END END END END ***** *)
(* **** END END END END END END END END ***** *)

(* ** END Definitions END Definitions END Definitions END *** *)
(* ** END Definitions END Definitions END Definitions END *** *)
(* ** END Definitions END Definitions END Definitions END *** *)
(* ** END Definitions END Definitions END Definitions END *** *)

(* **** Protect names defined and used only in context Private *** *)
Protect[PointQ,PointsQ,Points2Q,ComparatorQ,ComparatorListQ,
  COMPARATORS,VarNamesQ, SolveEqns,Equations,Eqns, ReduceToLowest,
  EqnThru2Pts,Angle,PerimeterOrder,CleanUpper,CleanLower,
  ConsecutiveConstraints, ExtremePoints,MaxXY,VerticalImplicitPlotLine,
  CorrectObjectiveValue,IntersectionPoints,MakeBoundarySystem];
(* END* Protect names defined and used only in context Private END *)

End[]
(* ** Private END Private END Private Private END Private *** *)
(* ** Private END Private END Private Private END Private *** *)
(* ** Private END Private END Private Private END Private *** *)

Protect[LineSystem,WriteToTextFile,InterceptTable,BasicSolutions,
  PlotFeasibleRegion,ColorCycleAmount,ColorForLine,ColorForPoint,
  ColorForLabel,ColorForObjective,ColorForRegion,ConstraintLabels,DefaultFont,
  NonConstraints,ObjectiveFunction,ObjectiveValue,PlotStyleForObjective,
  ShadeFR,ShowBoundaryLines,ShowCompleteLines,ShowExtremePoints,
  ShowLineLabels,LineLabelsOffset,ThicknessForLine,ThicknessForObjective,
  SizeForLabel,SizeForPoint,SmallLineReductionFactor,RedundantConstraints,
  ReorderPoints,Title,Ketasuu,NonBasicsToZero];

On[General::spell];
On[General::spell1];

EndPackage[]

```