```
/* Econ1.mac
    Maxima software for Economic Analysis
    Ted Woollett, Dec. 22, 2022
    https://home.csulb.edu/~woollett/
    http://home.csulb.edu/~woollett/eam.html

    Copyright (C)2021, 2022 Edwin L. Woollett  <woollett@charter.net>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU GENERAL PUBLIC LICENSE, Version 2, June 1991,
    as published by the Free Software Foundation.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details. You should have received a
    copy of the GNU General Public License along with this program.
    If not, see http://www.fsf.org/licensing/.
*/


/* functions defined in Econ1.mac:
     not included in this list are functions developed for
     LPsimplex.wxmx, LPduality.wxmx, and LPmatrix which appear at the bottom of this file.

   These functions have only been lightly tested and may not always
    be reliable.

1. CP(objective, [list_args] ) computes the first derivatives
   of f(x,y) and passes them to solve to look for critical points
   where df/dx and df/dy both vanish. If there are constraints on the
   problem of finding the desired critical points, then the program
   introduces Lagrange multipliers and passes the derivatives of f wrt
   those multipliers in addition.

   This function was created by David Stoutemyer, Prof. of Electrical
   Engineering, Univ. of Hawaii, in 1974, and is available in the Maxima
   share/calculus folder under the name optmiz.mac, load("optmiz"), and
   instead of CP (objective,...) use stapoints (objective,...),
   and the list of critical points is called stpts instead of cp.

   Most general invocation:
     CP (objective, lezeros, eqzeros, decisionvars)

   objective is an expression denoting the objective function or the
     label of such an expression.

   lezeros is a list of expressions which are constrained to be less
      than or equal to zero.  Use [] if no such constraints.

   eqzeros is a list of expressions which are constrained to equal zero, or
      the label of such a list.  Use [] if there are no such constraints.

   decisionvars is a list of the decision variables or the label of
       such a list.  May use [] if all variables in objective and
             constraints are decision variables.

   For convenience, brackets may be omitted from one-expression lists, and
         trailing [] arguments may be omitted.

    ---------------------------------------------
   CP (objective)  looks for critical points of objective (with
         no constraints) using solve.

   If there are "less than or equal to zero constraints"
     use CP (objective, lezeros).
```

If there are only "equal to zero constraints" use
CP (objective, [], eqzeros)

If there are both inequality and equality constraints, use
CP (objective, lezeros, eqzeros)

The code calls variables in objective, lezeros, and eqzeros
used to compute first and second derivatives of the
lagrangian function "decision variables".

If there are other symbols in objective which are not
"decision variables", then you need to tell CP
which symbols are to be considered decision variables
by including a fourth list as in
CP (objective, lezeros, eqzeros, decisionvars).

In this last case, if there are no constraints, use empty
lists as placeholders, as in CP(objective,[],[], decisionvars).

2. Extr (expr, critPt), or Extr (expr, var1, val1, var2, val2),
where critPt has the form (for example) [x = x1, y = y1],
examines the second derivatives of expr with respect to var1
and var2 evaluated at the given critical point, and returns
the list: [nature-of-critical-point, value-of-expr-at-critical-point]

3. Analyze (expr, critPts)
expr depends on two variables, which do not have to be x and y.
critPts is either one critical point or a list of several
critical points of the two variable expression expr.

acceptable forms for critPts (assuming expr depends on x and y):

[x = 3, y = 7], or [[x = 3, y = 7]]
or [[x = 3, y = 7], [x = -3, y = 7]]
or [[x = 3, y = 7], [x = -3, y = 7], [x = 0, y = 0]], etc.

1. Analyze calls Extr to determine the nature of the
critical point(s) provided, and
2. evaluates a list of second derivatives [Fxx, Fyy, Fxy]
at the critical points provided, returning that list.

4. plotCP (expr, [v1 = v1val, v2 = v2val])
or plotCP (expr, v1, v1val,dv1, v2, v2val, dv2)

plotCP calls wxdraw3d to make a 3d plot of the surface expr
as a function of variables v1 and v2 near the critical point
(v1val, v2val).
Orthogonal curves in a contrasting color which pass through
the critical point are added.
If the first syntax is used, the program selects the range in v1 and v2.
If the second syntax is used, the range of the plot is (v1 - dv1, v1 + dv1)
for variable v1 and (v2 - dv2, v2 + dv2) for variable v2.

5. colorMap (expr, [x, x1, x2, dx], [y, y1, y2, dy])
calls wxdraw2d to make a color map (density plot) of expr
with high values (max) in yellow, low values (min) in red, intermediate
values in blue.

6. contourPlot (expr, [x, x1, x2], [y, y1, y2], num)
calls wxcontour_plot to draw num curves of expr = constant value
over the 2d region specified.

7. contours3d (expr, [x,x1,x2], [y,y1,y2], numLevels)

The syntax shown assumes expr depends on x and y.

Draws both 3d contours and a map onto the x-y plane
using wxdraw3d.

this function automates the process:
(a specific example here):

```
wxdraw3d(color = light_gray, line_width = 3, xlabel = "x",
   ylabel = "y", title = sconcat ("contours of   ", zs),
   explicit(zs,x,-2,0,y,0, 4),
   contour_levels = 10, contour  = both,
   surface_hide   = true), wxplot_size = [680,680]$
```

8. LPM (amatrix, minorNum)
   returns the minorNum'th leading principal minor of the square amatrix.
   minorNum can be 1, 2, 3, ... , length(amatrix).

9. Minor (amatrix, arow, acol) returns the scalar produced by taking the
   determinant of the submatrix produced by deleting the row: arow and the
   column acol of the given matrix amatrix.

10. Qtest(amatrix)tests a symmetric numerical
matrix, assumed to be the matrix of coefficients of a quadratic form,
for sign definiteness, prints a conclusion, and returns a list
of the form
[["LPM1", LPM1], ["LPM2", LPM2],...,["LPMn", LPMn]].

11. CPtest (objective-func, critPts), used only for unconstrained
optimization problems, examines the nature of the candidate
extremum point (or list of points) of an objective expression
or function. CPtest internally uses the Hessian matrix method,
looking at the leading principal minors. If there are only two
variables in the expression expr, CPtest looks at the signs of
both fxx and fyy, and is able to distinguish an inflection point
from a saddle point. critPts can be [x = x0, y = y0] for one
critical point (if the objective func depends on x and y for
example), or a list of lists for multiple critical points.

12. cvec (alist) returns a matrix column vector

13. InputOutput (M, B)
M = Leontief matrix = (I - A),
(A = matrix of technical coefficients),
and B = the matrix column vector whose elements are the final demands,
computes the n leading principle minors of M, and if they are all
positive numbers, returns the unique solution matrix column vector
of total demands with all nonnegative elements, otherwise returns
"not viable"

14. Leontief (A) returns the Leontief matrix (I - A)
and checks that each element of A is positive.

15. Pinput (A, j) subtracts the sum of the jth column elements of the matrix A
from 1, returning the primary input value (labor, capital, etc) required
for the jth industry. A is the matrix of technical coefficients.

16. PItot (A, Xs) returns the total primary input (labor, capital, etc) required,
given the matrix of technical coefficients A and the total demands solution
vector Xs of the equation (I - A) . Xs = B, in which B is the matrix
column vector of final demands.

17. Bhessian (expr, gList, varList) returns the bordered Hessian matrix,
given the Lagrangian expression expr, a list of equality constraints gList,
and a list of variables varList.

18. BLPM(BH, m, k) -> LPM (BH, m + k) returns

the "kth bordered leading principal minor"
of a bordered Hessian matrix BH constructed from
m equality constraints. n is the number of variables
the Lagrange function depends on (excluding Lagrange multipliers).
BH has dimensions (m + n) x (m + n).
With n = length(BH) - m, we need m + 1 <= k <= n.

19. BHtest (BH, m, n), where BH is the bordered Hessian matrix,
m is the number of equality constraints,
n is the number of variables.
Normal return is a list of the values of the (n - m)
relevant leading principal minors, and one of three messages:
relative maximum, relative minimum, or indefinite.

20. Statics(FUNCLIST, EXOGENOUSVARLIST, PARAM)
Comparative Statics Analysis for set of functions
for which equilibrium is described by f(j) = 0,
the number of exogenous variables should equal the
number of functions, param is a single exogenous parameter.
*** This function only works if the functions f(j) contain
no implicit functions!

21. Msolve (J, X, Z) calls solve for solutions to the matrix
equation J . X = Z. The elements of the matrix column vector X
contain symbols which stand for the unknowns. J is a square n x n matrix
and X and Z are each n dimensional matrix column vectors.

22. Hessian (F, varList) is an alternative to the Maxima function hessian.

23,24,25. multiple, Dsingle, and single help distinuish different types of lists
returned by solve

26. gridSearch with one or two constraints

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],max,g(x,y))
returns [fmax, [xmax,ymax]] , and only looks at points for which g(x,y) >= 0.

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],min, g(x,y))
returns [fmin, [xmin,ymin]] , and only looks at points for which g(x,y) <= 0.

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],max, g1(x,y), g2(x,y))
returns [fmax, [xmax,ymax]] , and only looks at points for which both
g1(x,y) >= 0 and g2(x,y) >= 0.

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],min, g1(x,y), g2(x,y))
returns [fmin, [xmin,ymin]] , and only looks at points for which
both g1(x,y) <= 0 and g2 (x,y) <= 0.

27. one constraint grid search function.

gridSearch11 (f(x,y),[x,x1,x2,dx], [y,y1,y2,dy],maxmin,g(x,y))
returns [fmax,[xmax,ymax]] if maxmin = max and g(x,y) >=0
or [fmin, [xmin,ymin]] if maxmin = min and g(x,y) <= 0

28. two constraint grid search function

gridSearch22 (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],maxmin,[g1,g2])
in which if maxmin = max, then g1(x,y) >= 0 and g2(x,y) >= 0
and if maxmin = min, then g1(x,y) <= 0 and g2(x,y) <= 0.

29. RealSoln(alist)

30. NonNegative (alist,vvL)

31. optimum (func, varL, constraints)
handles one or more equality constraints
syntax

```
       optimum (f,varL) case no equality constraints, calls CPtest
       optimum  (f,varL, g)  case one equality constraint g = 0, calls BHtest
       optimum (f, var, [g])
       optimum (f, varL, [g1, g2]) case both g1 = 0 and g2 = 0.
        etc.

      defines global cp
      screens solutions to only deal with real non-negative values
      of the search variables %vL.

32. optimumAll (func, varL, constraints)

    accepts all solutions returned by solve.
     defines global cp
     calls CPtest or BHtest

/*  Simple compounding and discounting functions  */

33. nPeriod (rate, PV, FV)

34. interest(PV, FV, nPeriod)

35.  fv(rate,PV, nPeriod)

36.  fvm(rate,PV, m, nPeriod)

37.  pv(rate, FV, nPeriod)

38. pvm (i, FV, m, t)

39. rate (i,m)

40. details is a global parameter set to false

41. killAB(),  kills all but the definitions of CP, Extr, Analyze, plotCP,
                    colorMap,contourPlot, contours3d,LPM, Minor,Qtest,
                    CPtest, gradient,cvec,InputOutput,Leontief, Pinput,
                    PItot,Bhessian, BLPM, BHtest,Statics, Hessian, Msolve,
                    multiple,Dsingle,single,gridSearch, gridSearch11,
                    gridSearch22,KKT,KKTmax,KKTmax2,KKTmin,KKTmin2,
                    optimumAll,RealSoln,NonNegative,optimum,nPeriod,
                    interest,fv,fvm,pv,details,and killAB.

Another  function, gradient(decslkmults), is defined and used by CP
to create a list of the first derivatives of the lagrangian needed
to pass to solve.

 */

/*  print (" end of comments on funcs defined")$  */

ttyoff:true $
/* slight modification of package optmiz.mac, part of share/calculus/
      folder in Maxima, written by David Stoutemyer in 1974.


    We have replaced the name of the package optmiz.mac by CP.mac and
      replaced function name stapoints by CP and replaced list name
          stapts by cp. We have then added a number of extra analysis
          and plotting functions at the end of the file.

      Ted Woollett, April, 2021  */

/* Functions and options for optimization using the algebraic
manipulation language MACSYMA.  Programmed by STOUTE (David
Stoutemyer), Electrical Engineering Department, University of Hawaii,
4/3/74.  For a description of its usage, see text file OPTMIZ USAGE. */
```

```
/* First, we set some options that respectively cause automatic
   printing of cpu time in milliseconds, force attempted equation
   solution even when there are more variables than unknowns,
   enables some (time consuming) techniques for solving equations that
   contain logs and exponentials, and enables the solution
   of consistent singular linear equations: */

/* Next, we set a switch to suppress the message that ordinarily
   occurs whenever a floating-point number is replaced with a rational
   number, and to prevent 1-by-1 matrices from being converted
   to scalars: */

/* The following pattern-matching statements permit trailing
[] arguments to be omitted: */

/* updated by function stapoints(objective,[list_args])
matchdeclare([a1,a2,a3,a4], true) $
tellsimp (stap(a1), stapoints(a1,[],[],[])) $
tellsimp (stap(a1,a2), stapoints(a1,a2,[],[])) $
tellsimp (stap(a1,a2,a3), stapoints(a1,a2,a3,[])) $
tellsimp (stap(a1,a2,a3,a4), stapoints(a1,a2,a3,a4)) $
*/
eval_when([translate,batch,demo,load,loadfile],rectform(sinh(1.2-.4*%i)),
dv(a)::=buildq([a],define_variable(a,'a,any)))$

dv(lagrangian)$ dv(eigen)$ dv(modhessian)$
dv(grad)$ dv(gradsub)$ dv(cp)$ dv(mhess)$ dv(objsub)$
dv(mhesssub)$ dv(cpolysub)$ dv(eigen)$ dv(decslkmults)$
dv(ndec)$ dv(neqz)$ dv(nlez)$ dv(ntot)$ dv(eigens)$

gradient(decslkmults) := /* This function recursively defines
      the gradient of the Lagrangian, with respect to the decision
      variables, rtslacks, and Lagrange multipliers. */
   if decslkmults = [] then []
   else cons(diff(lagrangian, first(decslkmults)),
      gradient(rest(decslkmults))) $

eval_when([translate,batch,demo,load,loadfile],
modh(g,d)::=buildq([g,d],
apply('define,[arrayapply('modhessian,[i,j]),
 /*internal array function for MODHESSIAN*/
  '('(if j>i then modhessian[j,i] /*(symmetric)*/
   else diff(g[i],d[j]) /*minus EIGEN from up-left diag*/
      - (if i=j and j<=ndec+nlez then eigen  else 0)))])))$


/* old definition
stapoints(objective, lezeros, eqzeros, decisionvars) := block( */

/* new definition */
CP(objective,[list_args]):=
block([lezeros, eqzeros, decisionvars],
(if list_args=[] then  lezeros: eqzeros: decisionvars:[]
else if length(list_args)=1 then (lezeros:first(list_args),
eqzeros: decisionvars:[]) else if length(list_args)=2 then
(lezeros:first(list_args),eqzeros:last(list_args),decisionvars:[])
else if length(list_args)=3 then (lezeros:first(list_args),
eqzeros:first(rest(list_args)), decisionvars:last(list_args)) else
error("wrong number of args to CP"), block(
/* This is the major function, which prints information about any
stationary points, then returns the value DONE. */

   [grindswitch, solveradcan, singsolve, ratprint, scalarmatrixp,
    dispflag,eqmult,rtslack,lemult,i,j], /* declare local variables */
   grindswitch: solveradcan: singsolve: true,
   ratprint: scalarmatrixp: false,
```

```
if member('modhessian,arrays) then apply('remarray,['modhessian]),

modh(grad,decslkmults) /*end MODHESSIAN*/,

   if not listp(lezeros) then lezeros: [lezeros],/* ensure list args*/
   if not listp(eqzeros) then eqzeros: [eqzeros],
   if decisionvars = [] /*default to all decision variables*/
      then decisionvars: listofvars([objective, lezeros, eqzeros])
   else if not listp(decisionvars) then decisionvars: [decisionvars],

   ndec: length(decisionvars), /*determine number of decision vars. */
   nlez: length(lezeros),/*determine number of inequality constraints*/
   neqz: length(eqzeros), /*determine number of equality constraints*/
   lagrangian: objective + sum(eqzeros[i]*eqmult[i],i,1,neqz)
      + sum((lezeros[i]+rtslack[i]**2)*lemult[i],i,1,nlez),

   decslkmults: [], /*form list of dec.vars., rtslacks & multipliers*/
   for i:neqz step -1 thru 1 do decslkmults: cons(eqmult[i],
      decslkmults),
   for i:nlez step -1 thru 1 do
      decslkmults: cons(lemult[i], decslkmults),
   for i:nlez step -1 thru 1 do
      decslkmults: cons(rtslack[i], decslkmults),
   decslkmults: append(decisionvars, decslkmults),
   grad: gradient(decslkmults),  /* form gradient  */
   dispflag: false, /* suppress automatic output from solve */
   cp: solve(grad,decslkmults),/* solve GRAD=0*/

   if cp = [] then apply('disp,["no stationary points found"])
   else( ntot: ndec + nlez + nlez + neqz,
      mhess:'mhess, mhesssub:'mhesssub,  /* unbind global matrices from
         previous case to permit different sizes. */
      mhess: genmatrix(modhessian, ntot, ntot), /*form HESS*/
 apply('remarray,['modhessian]),
 modhessian:'modhessian, /*destroy array to permit new
         definition for next use of analyze. */
      dispflag: true, /* permit automatic output from SOLVE */
      for i thru length(cp) do (
         objsub: apply('ev,[objective,cp[i],'rectform]),
 /*evaluate objective.*/
         gradsub:apply('ev,[grad,cp[i],'rectform]),
 /*eval. gradient at point*/
         apply('ldisplay,[arrayapply('cp,[i]), 'objsub, 'gradsub]),
       /* output */
         mhesssub:apply('ev,[mhess,cp[i],'rectform]),
 /*eval. modified Hessian */
         cpolysub:rectform(newdet(mhesssub)),/*form poly in EIGEN*/
            /* if CPOLYSUB is univariate use REALROOTS, otherwise
            use the slower SOLVE function: */
         if listofvars(cpolysub) = [eigen] and freeof('%i,cpolysub)
            then eigens: apply('ev,[realroots(cpolysub,rootsepsilon),'numer])
         else eigens: solve(cpolysub, eigen) )))))
  /* end of function CP. */ $

ttyoff:false$

/*  print (" end of def of CP")$  */

/*
Extr (expr, critPt), or Extr (expr, var1, val1, var2, val2),

    where critPt has the form (for example) [x = x1, y = y1],
    examines the second derivatives of expr with respect to var1
    and var2 evaluated at the given critical point, and returns
    the list: [nature-of-critical-point, value-of-expr-at-critical-point]

*/
```

```
Extr (func, [%subL]) :=
block ([varL, var1, var2, valL, subL, DCP,
        funcv, func11,func22, funcv1,funcv2],

    if length (%subL) = 1 then (
        subL : %subL[1],
        if length (subL) = 1 then error ("invalid syntax"),
        varL : map ('lhs, subL) )

    else  if length (%subL) = 4 then (
        varL :  [%subL[1], %subL[3] ],
        valL : [%subL[2], %subL[4] ],
        subL : map ("=", varL, valL )  )

    else return (" invalid syntax"),

    var1 : varL[1],
    var2 : varL[2],
    /*  display (subL, var1, var2), */

    funcv1 : subst (subL,  diff (func, var1)),
    funcv2 : subst (subL,  diff (func, var2)),
    if abs(float (funcv1)) > 1e-8 or abs(float (funcv2)) > 1e-8 then
       (print ("values are not a critical point for provided func"),
         return (done) ),

    funcv : float (subst (subL, func)),
    func11 : float (subst (subL, diff (func, var1, 2) )),
    func22 : float (subst (subL, diff (func, var2, 2) )),
    DCP : float (subst (subL, determinant (hessian (func,varL)))),

    if DCP > 0 and func11 > 0 then return ( [ "relative minimum", sconcat( "value = ", funcv) ] )
      else if DCP > 0  and  func11 < 0 then return ( [ "relative maximum", sconcat( "value = ",
      funcv) ] )
      else if DCP < 0 then (
           if func11*func22 < 0 then return ( [ "saddle point", sconcat( "value = ", funcv) ]  )
            else if func11*func22 >= 0 then return ( [ "inflection point", sconcat( "value = ",
            funcv) ] )   )
    else return ("undetermined") )$

/*
   Analyze (expr, critPts)
      expr depends on two variables, which do not have to be x and y.
     critPts is either one critical point or a list of several
        critical points of the two variable expression expr.

     acceptable forms for critPts (assuming expr depends on x and y):
     [x = 3, y = 7], or [[x = 3, y = 7]]
       or [[x = 3, y = 7], [x = -3, y = 7]]
       or  [[x = 3, y = 7], [x = -3, y = 7], [x = 0, y = 0]], etc.
     1. Analyze calls Extr to determine the nature of the
          critical point(s) provided, and
     2. evaluates a list of second derivatives [Fxx, Fyy, Fxy]
       at the critical points provided, returning that list.

*/

Analyze (expr, critPts) :=  block ( [u1, u2, %critPts],
     if not listp (part (critPts,1)) then %critPts : [critPts]
        else %critPts : critPts,
    [u1, u2] : map ('lhs, %critPts[1]),
    for j thru length (%critPts) do
    (print (j, "cp = ", %critPts[j],  Extr (expr, %critPts[j] ) ),
     print ( "              secondDeriv = ",
     subst (%critPts[j], [diff (expr, u1,2), diff (expr, u2, 2),
             diff (expr, u1, 1, u2, 1)])) ) )$
```

```
/* plotCP (expr, [v1 = v1val, v2 = v2val])
           or plotCP (expr, v1, v1val,dv1, v2, v2val, dv2)

  calls wxdraw3d to make a 3d plot of the surface of expr as a function of variables
   v1 and v2 near the critical point (v1val, v2val).Orthogonal curves in a contrasting
    color which pass through the critical point are added.

    If the first syntax is used, the program selects the range in v1 and v2.
If the second syntax is used, the range of the plot is (v1 - dv1, v1 + dv1) for
variable v1 and (v2 - dv2, v2 + dv2) for variable v2.
*/

plotCP (expr, [%subL] ) :=
      block ( [var1, var2, val1,val2, dv1, dv2,subL,
                     val1m, val1p, val2m, val2p, fac:0.5,ln,p10,p110 ],
    if length (%subL) = 6   then
              ( [var1, var2] :  [%subL[1], %subL[4] ],
                [val1, val2] : [%subL[2], %subL[5] ],
              [dv1, dv2] : [abs (%subL[3] ), abs (%subL[6] ) ],
              [val1m, val1p] : [val1 - dv1, val1 + dv1 ],
              [val2m, val2p] : [val2 - dv2, val2 + dv2 ],
                subL :  [ var1 = val1, var2 = val2] )

      else if length (%subL) = 1 then
              ( subL : %subL[1],
                ln : length (subL),
                p10 : part (subL,1,0),
                if ln = 2 then
                      (if (p10 = "[" ) then (print (" plotting only first critical point"),
                                                subL : subL[1])
                      else if (p10 # "=" ) then return ("invalid syntax")
                      else done)

                else if  ln = 1 then
                      (p110 : part(subL, 1, 1, 0),
                      if (p110 = "=") then subL : subL[1]
                      else if (p110 = "[") then
                                  (if (part (subL,1,1,1,0) = "=") then subL : subL[1][1])
                      else return ("invalid syntax") )

                else return ("invalid syntax"),

               [var1, var2] : map ('lhs, subL),
               [val1, val2] : map ('rhs, subL),

              if abs (val1) < 0.1 then
                  (val1p : 1, val1m : -1)
              else
                  (val1p : val1  + fac*abs(val1),
                  val1m : val1 - fac*abs(val1)),
            if abs (val2) < 0.1 then
                 (val2p : 1, val2m : -1)
             else
                (val2p : val2 + fac*abs(val2),
                 val2m : val2 - fac*abs(val2) ) )

        else return (" invalid syntax"),

      print ( "surface of ", expr),
        print ("near critical point = ", subL),
        wxdraw3d (xlabel = string (var1), ylabel = string (var2), color = green,
            explicit (expr, var1,val1m, val1p, var2, val2m, val2p),
            color = black,
            explicit ( subst (var1 = val1, expr), var1, val1, val1, var2, val2m, val2p),
```

```
            explicit ( subst (var2 = val2, expr), var1, val1m, val1p, var2, val2, val2) ))$

/*
colorMap (expr, [x, x1, x2, dx], [y, y1, y2, dy] )
   calls wxdraw2d to make a color map (or density plot) of expr
    with high values in yellow, low values in red, intermediate
    values in blue. A more continuous density plot occurs with
    smaller values of dx and dy.
*/

colorMap(EXPR, v1List, v2List) :=
 block ([v1, v1a, v1b, dv1,v2, v2a,v2b, dv2,
              n1, n2, v1L, v2L, v1v2vals, imatrix],
    [v1, v1a, v1b, dv1] : v1List,
  /*  display (v1, v1a, v1b, dv1),  */
    [v2, v2a, v2b, dv2] : v2List,
    n1 : floor( (v1b - v1a)/dv1 ),
    n2 : floor( (v2b - v2a)/dv2 ),
    v1L : float( makelist( v1a + dv1*ii, ii,1, n1 ) ),
    v2L : float( makelist( v2a + dv2*ii, ii,1, n2 ) ),
    v1v2vals : makelist( subst( v2 = v2L[jj],
                makelist( subst( v1 = v1L[ii], EXPR), ii,1,n1)), jj,1, n2),
    imatrix : apply ('matrix, v1v2vals),
    print (" color map for ", EXPR),
    wxdraw2d( palette = [red, blue, yellow], xlabel = string(v1), ylabel = string(v2),
         colorbox = "value", image (imatrix, v1a, v2a, v1b - v1a, v2b - v2a) ) )$

/*
  contourPlot (expr, [x, x1, x2], [y, y1, y2], num)
    calls wxcontour_plot to draw num curves of expr = constant value
    over the 2d region specified.
*/

contourPlot (EXPR, v1List, v2List, numLevels) :=
 block ([pstring],
    pstring : sconcat ("set cntrparam levels ", numLevels),
   /* display (pstring), */
    wxcontour_plot (EXPR, v1List, v2List,
        [gnuplot_preamble, pstring] ) )$

/* contours3d (expr, [x,x1,x2], [y,y1,y2], numLevels) */

/*  this function automates the process:
 (a specific example here):

wxdraw3d(color = light_gray, line_width = 3, xlabel = "x", ylabel = "y",
                title = sconcat ("contours of   ", zs),
            explicit(zs,x,-2,0,y,0, 4),
            contour_levels = 10, contour  = both,
            surface_hide   = true), wxplot_size = [680,680] $
*/


contours3d (EXPR, v1List, v2List, numLevels) :=
  block ([EXPLICIT],
   EXPLICIT : apply ('explicit, flatten ([ [EXPR],v1List,v2List])),
   ev (apply ('wxdraw3d, flatten ([ [color = light_gray, line_width = 3,
            xlabel = string(v1List[1]), ylabel = string (v2List[1]),
            title = sconcat (" contours of   ", EXPR) ],
            [EXPLICIT],[contour_levels = numLevels, contour = both,
             surface_hide = true] ])), wxplot_size = [680,680] ))$


/*
LPM (M, k)
   returns the k'th leading principal minor of the square matrix M.
   k can be 1, 2, 3, ... , length(M).
```

```
   LPM (M,k) is a determinant. LPM(M,1) = M[1,1] = 1x1 determinant.
   LPM(M,2) = M[1,1]*M[2,2] - M[1,2]*M[2,1] = 2x2 determinant.
   etc.
*/


LPM (%M, num) :=
block ([lm, fl : [ ] ], local (rl),
    lm : length ( %M),
    if length ( transpose (%M) ) # lm then return ("not square matrix"),
    if num > lm then return (" num invalid "),
    if num = 1 then %M[1,1]
    else if num = lm then determinant (%M)
    else
      ( for j thru num do rl[j] : part (row (%M, j), 1),
        for j : num thru 1 step -1 do fl : cons ( rest (rl[j], - (lm - num))  , fl),
        determinant ( apply ('matrix, fl))))$


/* Minor (amatrix, arow, acol) returns the scalar produced by taking the
    determinant of the submatrix produced by deleting the row: arow and the
    column acol of the given matrix amatrix.
*/

Minor (MM, mm, nn) := determinant (minor (MM, mm, nn))$



/*  The Maxima function Qtest(amatrix)tests a symmetric numerical
     matrix, assumed to be the matrix of coefficients of a quadratic form,
     for sign definiteness, prints a conclusion, and returns a list
     of the form

     [ ["LPM1", LPM1], ["LPM2", LPM2],...,["LPMn", LPMn] ].
*/


Qtest (%M) :=
block ([num, lpm,lpmp, indef : false,
              alt : false,%pos : false, signL : []],
   num : length (%M),

   if length (transpose (%M)) # num then return (" not a square matrix"),
   if not is (equal ( float (transpose (%M) - %M), zeromatrix (num,num))) then
              return (" not a symmetric matrix"),

   for i thru num do
     for j thru num do
        if not numberp (%M[i,j]) then error(" not a purely numerical matrix"),

    /* look at %M[1,1]  element  */
   lpm : %M[1,1],
   signL : cons ([sconcat("LPM",1),lpm], signL),
  /* print (" look at M[1,1]"),  */
  /* display (lpm), */

   if lpm = 0 then (
       indef : true,
       print (" LPM1 equals zero"))
    else if abs (float (lpm)) < 1e-7 then (
       indef : true,
       print (" abs(float(LPM1)) close to zero"))
    else (
       lpmp : lpm,
       if lpm < 0 then (
           alt : true  /*,    print ("alt set true") */ )
        else if lpm > 0 then (
```

```
          %pos : true  /* , print ("%pos set true") */ )),

     /* print (" start i = 2,..."),   */

   for i : 2 thru num do (
       lpm : LPM (%M, i),
       signL : cons ([sconcat("LPM",i),lpm], signL),
/*     display (i,lpm), */

       if (lpm = 0 or abs (float (lpm)) < 1e-7) then (
          indef : true,
          alt : false,
          %pos : false,
          print (" for i = ",i, "LPM zero or close to zero")),

       if not indef then  /* check for min or max pattern  */
          if alt then if lpmp*lpm > 0 then (
                   alt : false,
                   indef : true )
          else if %pos then if lpmp*lpm < 0 then (
                       %pos : false,
                       indef : true),

       lpmp : lpm),

   if indef then print ("indefinite ")
    else if alt then print ("negative definite ")
    else if %pos then print ("positive definite "),

   signL : reverse (signL),
    print (signL),
    float ( map ('lambda ([LL], LL[2]), signL)) )$

/* CPtest (objective-func, critPts), used only for unconstrained
    optimization problems, examines the nature of the candidate
    extremum point (or list of points) of an objective expression
    or function. CPtest internally uses the Hessian matrix method,
    looking at the leading principal minors. If there are only two
    variables in the expression expr, CPtest looks at the signs of
    both zxx and  zyy, and is able to distinguish an inflection point
    from a saddle point. critPts can be [x = x0, y = y0] for one
    critical point (if the objective func depends on x and y for
    example), or a list of lists for multiple critical points.


 */

CPtest (expr, critPts) :=
 block ( [vL, nv, nCP, cPts, d1, anerr, %H, Hcp,
                 lpm, signL, lpmp, %pos, %neg, %indef,
                 Fxx, Fyy],
   /* display (critPts), */
    if not listp (part (critPts,1)) then cPts : [critPts]
        else cPts : critPts,
    /* display (cPts, cPts[1]), */
    nCP : length (cPts),
    vL : map ('lhs, cPts[1]),
    nv : length(vL),
    /* display (nCP, vL,nv), */
    /* Hessian matrix */
    %H : hessian (expr, vL),
    /* display (%H),       */

    for j thru nCP do (
         %indef : false,
         %neg : false,
         %pos : false,
```

```
    signL : [],
    anerr : false,
   /* print (sconcat ("cp",j)), */



    /* check first derivatives at critical point */
 /*  print (j, "check first derivatives at critical point"), */
     for k thru nv do (
            /* display (k), */
            d1 : subst (cPts[j], diff (expr, vL[k])),
            /* display (d1),*/
            if abs (float (d1)) > 1e-10 then (
                   /* display (k, d1), */
                   print(sconcat("cp",j),"   first derivative of expr with respect
                   to",vL[k],
                            "not equal to zero at point", cPts[j] ),
                    anerr : true,
                   return())),
    if anerr then return( ),
    Hcp : subst (cPts[j], %H),
   /* print (j,cPts[j],"  ",Hcp), */
     /* calculate leading principal minors */
     /* look at Hcp[1,1] element */
    lpm : Hcp[1,1],
   /* print (" i = 1, lpm = ", lpm), */
    signL : cons (["LPM1", lpm], signL),
   /* display (signL), */
    if lpm = 0 then (
             %indef : true,
             print (" for ",sconcat("cp",j),cPts[j]," LPM1 equals zero"))
    else if abs (float (lpm)) < 1e-7 then (
            %indef : true,
            print (" for ", sconcat("cp",j),cPts[j],",  abs(float(LPM1)) close to
            zero"))
    else (lpmp : lpm,
            if lpm < 0 then (
                     %neg : true  /*,    print ("%neg set true") */ )
             else if lpm > 0 then (
                      %pos : true  /* , print ("%pos set true") */ )),
for i : 2 thru nv do (
            lpm : LPM (Hcp, i),
            signL : cons ([sconcat("LPM",i), lpm], signL),
           /* display (i,lpm),  */

            if (lpm = 0 or abs (float (lpm)) < 1e-7) then (
                    %indef : true,
                     %neg : false,
                     %pos : false,
                     print ("for ",sconcat("cp",j),cPts[j],
                             sconcat ("LPM",i),"is zero or close to zero") ),

           if not %indef then  /* check for min or max pattern  */
                    if (%neg and lpmp*lpm > 0) then (
                            %neg : false,
                            %indef : true )
                    else if (%pos and lpmp*lpm < 0) then (
                           %pos : false,
                           %indef : true),

          lpmp : lpm),



if (%indef and nv = 2) then (
     Fxx : Hcp[1,1],
     Fyy : Hcp[2,2],
```

```
          if Fxx*Fyy < 0 then print (sconcat("cp",j),cPts[j],", saddle point" )
                  else if Fxx*Fyy > 0 then  print (sconcat("cp",j),cPts[j], ", inflection point")
                  else print ( sconcat("cp",j),cPts[j], ", not determined"))

     else if %indef then print (sconcat("cp",j),cPts[j], ", not determined ")
     else if %neg then print (sconcat("cp",j),cPts[j], ", relative maximum, value = ",
            float (subst(cPts[j],expr)))
     else if %pos then print (sconcat("cp",j),cPts[j], ", relative minimum, value = ",
            float (subst(cPts[j],expr)))
     else print (sconcat("cp",j),cPts[j], ", not determined"),

     signL : reverse (signL),
     if details then (
          print (signL),
          print (float ( map ('lambda ([LL], LL[2]), signL)))),
     done )$

/* cvec (alist) returns a matrix column vector */


cvec(zzL) :=
    ( if not listp(zzL) then (
        print(" arg of cvec should be a list"),
        return()),
    transpose (matrix (zzL) ))$

/*  InputOutput (M, B)
         M = Leontief matrix = (I - A),
         (A  = matrix of technical coefficients),
         and B = the matrix column vector whose elements are the final demands,
   computes the n leading principle minors of M, and if they are all
   positive numbers, returns  the unique solution matrix column vector
   of total demands with all nonnegative elements, otherwise returns
   "not viable"
*/


InputOutput (leontief, finalDemand) :=
  block ( [num, mm, signerr : false, lpm,  lpmerr : false   ],
    num : length (leontief),
    if length (finalDemand) # num then
         return (" length of finalDemand # length of leontief"),
    /* check off-diag signs of leontief */
    for i thru num do (
       for j thru num do if i # j then (
           mm : leontief [i, j],
           if (mm > 0 or float (mm) > 1e-10) then (
                signerr : true,
                print (" i = ",i,"  j = ",j," mm = ",mm),
                print (" off-diagonal elements of leontief should be nonpositive"),
                return ( ) ) ),
            if signerr then return () ),
    if signerr then return (done),
    /* check leading principal minors */
    for i thru num do (
         lpm : LPM (leontief, i),
         if ( lpm <= 0 or float (lpm) <= 1e-10) then (
            lpmerr : true,
            return () )),
    if lpmerr then "not viable"
    else invert (leontief) . finalDemand )$

/*  Leontief (A) returns the Leontief matrix (I - A)
     and checks that each element of A is positive.
*/

Leontief (techcoeff) :=
```

```
   block ( [num, tc, signerr : false],
     num : length (techcoeff),
    /* check signs of techcoeff matrix */
     for i thru num do (
         for j thru num do (
             tc : techcoeff [i, j],
             if ( tc < 0 or float (tc) < -1e-10) then (
                 print (" for i = ",i,"  j = ",j, "   tc = ", tc),
                 signerr : true,
                 return () ),
        if signerr then return () ),
    if signerr then return ( ) ),
    if signerr then (
        print (" all elements of techcoeff matrix should be nonnegative"),
        done)
      else  (diagmatrix (length (techcoeff),1)  - techcoeff) )$

/*  Pinput (A, j) subtracts the sum of the jth column elements of the matrix A
     from 1, returning the primary input value (labor, capital, etc) required
    for the jth industry. A is the matrix of technical coefficients.
*/

Pinput (techcoeff, nn) := (1 - sum (techcoeff [k, nn], k, 1, length (techcoeff)))$

/*  PItot (A, Xs) returns the total primary input (labor, capital, etc) required,
    given the matrix of technical coefficients A and the total demands solution
    vector Xs of the equation (I - A) . Xs = B, in which B is the matrix
    column vector of final demands.
*/


PItot (techcoef, solnvec) :=
  (sum (Pinput (techcoef, kk)*solnvec[kk,1], kk, 1, length (techcoef) ) )$

/* Bhessian (expr, gList, varList)

   Bhessian returns the bordered Hessian matrix,
   given the Lagrangian expression expr, a list
   of equality constraints gList, and a list of variables
   varList.
*/



Bhessian (func, ggList, varList) :=
  block ([num, meq, hM, gL, MM], local(aa),
    if not listp (ggList) then ggList : [ggList],
    num : length(varList),
    meq : length (ggList),

    hM : hessian (func, varList),

    MM : genmatrix (aa, meq + num, meq + num),

    for i thru meq do
        for j thru meq do MM[i,j] : 0,

    for i thru meq do (
        gL : jacobian ([ggList[i]], varList)[1],
        for j : meq + 1 thru meq + num do (
           MM[i,j] : gL[j - meq],
           MM[j,i] : gL[j - meq])),

    for i : meq + 1 thru meq + num do
          for j : meq + 1 thru meq + num do
               MM[i,j] : hM[i-meq,j-meq],
    MM)$
```

```
/* BLPM( BH, m, k) -> LPM (BH, m + k),
       the kth bordered leading principal minor
      of a bordered Hessian matrix BH constructed from
      m equality constraints. n is the number
       of variables the Lagrange function depends on
        (excluding Lagrange multipliers). BH has dimensions
        (m + n) x (m + n).
        With n = length(BH) - m,
        we need m + 1 <= k <= n.      */

    BLPM (%bh, %m, %k) :=
     block ( [lbh,nvar],
        lbh : length (%bh),
        nvar : lbh - %m,
        /* display (%bh, lbh, %m, nvar, %k), */
       if %k > nvar then (
           print ("max value of k is", nvar),
           return(done))
       else if %k < %m + 1 then (
           print (" min value of k is ",%m + 1 ),
           return (done))
       else  LPM (%bh, %m + %k))$

/* BHtest (BH, m, n)

BH is the bordered Hessian matrix, m is the number of
equality constraints, n is the number of variables.
Normal return is a list of the values of the (n - m) relevant leading principal
minors, and one of three messages: relative maximum, relative minimum,
or indefinite.

*/

BHtest (%BH, %m, %n) :=
  block ([lpmList : [ ], lpmlabels : [ ],%len : %n - %m,
           %max : false, %min : false, %zero : false, %lpm  ],
    if %len < 1  then return (" n must be greater than m "),
    for j : 2*%m + 1 thru %m + %n do
       lpmList : cons (LPM (%BH, j), lpmList),
    lpmList : reverse (lpmList),
    for j thru %len do (
       %lpm : lpmList[j],
       if abs (float (%lpm)) < 1e-8 then (
          if details then print ("lpmList[",j,"] < 1e-8 "),
           %zero : true)),

    /* look at sign of last lpm in list */
    %lpm : lpmList [%len],

    if (evenp(%n) and %lpm > 0) or (oddp (%n) and %lpm < 0) then (
       %max : true,
       if %len > 1 then
         for j thru %len - 1 do
            if lpmList[j]*lpmList[j+1] > 0 then %max : false),

    if (evenp(%m) and %lpm > 0) or (oddp (%m) and %lpm < 0) then (
       %min : true,
       if %len > 1 then
         for j thru %len - 1 do
            if lpmList[j]*lpmList[j+1] < 0 then %min : false),

    if (%zero or (%max and %min)) then print ("indefinite")
     else if %max then print ("relative maximum")
     else if %min then print ("relative minimum")
     else print ("indefinite"),
```

```
    for j : 2*%m + 1 thru %m + %n do
       lpmlabels : cons (sconcat ("LPM",j),lpmlabels),
    lpmlabels : reverse (lpmlabels),
    map ("=", lpmlabels, lpmList))$

/* optimumAll (func, varL, constraints)
  func depends on two or more variables.
  accepts all solutions returned by solve.
  defines global cp
  calls CPtest or BHtest

 */



optimumAll ([%v]) :=
 block ([%f, %vL, %cL,nct,nV, %Lagr,Lgrad, %lamL : [],
           %xlamL,%solns,asoln,%BH,%LPM], local( lam),
   /* display (%v), */
   %f : first (%v),
   %vL : second (%v),
   if not listp (%vL) then %vL : [%vL],
   nV : length (%vL),
   if nV = 1 then return ("need two or more variables as list")
    else if length (%v) = 2 then %cL : []
    else if listp (third (%v)) then %cL : third (%v)
    else %cL : rest (%v, 2),
   /* display (%f, %vL, %cL), */
   nct : length(%cL),
   if nct = 0 then %xlamL : %vL
    else (
        for i thru nct do push(lam[i],%lamL),
        %lamL : reverse (%lamL)),
   /* display (%lamL),  */
   if nct = 0 then %Lagr : %f
    else %Lagr : ratsimp (%f + apply ("+", %lamL*%cL)),
    print (" lagrangian = ",%Lagr),
    %xlamL : flatten ([%vL,%lamL]),
    if details then display (%xlamL),

   Lgrad : jacobian ([%Lagr], %xlamL)[1],
    /* display (Lgrad), */
    %solns : solve (Lgrad, %xlamL),
    if details then display (%solns),
    cp : factor (%solns),  /* defines global list of solutions */

    if nct = 0 then (  /* case no constraints, use CPtest  */
        if multiple (%solns) then for i thru length (%solns) do (
                  if details then print (" i = ",i,"  %solns[i]  = ",%solns[i]),
                  CPtest (%f, %solns[i]))
        else if Dsingle (%solns) or single (%solns) then (
             if Dsingle (%solns) then asoln : %solns[1] else asoln : %solns,
             if details then display (asoln),
             CPtest (%f, asoln))
        else return ("problem with %solns form") )

    else /* case one or more constraints (g = 0) , use bordered hessian matrix test*/
         ( %BH : Bhessian (%Lagr,%cL,%vL),
           /* display (%BH), */
           if multiple (%solns) then for i thru length (%solns) do (
                 print ("---------------------------"),
                 print ("i = ",i," soln = ",%solns[i],"  objsub = ",at (%f,%solns[i])),
                 print (" soln = ",float(%solns[i]),"  objsub = ", float(at (%f,%solns[i]))),
                %LPM : BHtest ( at(%BH,%solns[i]), nct, nV),
                 print (" LPM's = ", float(%LPM)))
           else if Dsingle (%solns) or single (%solns) then (
                if Dsingle (%solns) then asoln : %solns[1] else asoln : %solns,
```

```
               print (" soln = ", asoln,"  objsub = ",at(%f,asoln)),
               print (" soln = ", float(asoln),"  objsub = ",float (at(%f,asoln))),
               %LPM : BHtest (at(%BH,asoln), nct, nV),
                print (" LPM's = ", float(%LPM)) )
          else return("problem with %solns form")),
   done)$



RealSoln(alist) :=
 block([rL, notreal : false],
    rL : map ('rhs, alist),
    for i thru length (alist) do
        if imagpart (rL[i]) # 0 then (
        notreal : true,
        return()),
    if notreal then false else true)$


NonNegative (alist,vvL) :=
block ( [lL, rL,  %nn: true],
    lL : map ('lhs, alist),
    rL : map ('rhs, alist),
    for i thru length(rL) do (
         if member (lL[i],vvL) and rL[i] < 0 then (
              %nn : false,
              return( ))),
    %nn)$


/*  optimum (func, varL, constraints)
   func depends on two or more variables
   handles one or more equality constraints, if such are imposed,
   syntax
   optimum (f,varL) case no equality constraints, calls CPtest,

   optimum  (f,varL, g)  case one equality constraint g = 0, calls BHtest,
    can also use easier syntax for one constraint: optimum (f, var, [g]),
   optimum (f, varL, [g1, g2]) case both g1 = 0 and g2 = 0, calls BHtest,
   etc.

  defines global list cp,
  screens solutions to only deal with real non-negative values
     of the search variables %vL.


 */

optimum ([%v]) :=
 block ([%f, %vL, %cL,nct,nV, %Lagr,Lgrad, %lamL : [],
           %xlamL,%solns,asoln,%BH,%LPM], local( lam),
   /* display (%v), */
   %f : first (%v),
   %vL : second (%v),
   if not listp (%vL) then %vL : [%vL],
   nV : length (%vL),
   if nV = 1 then return ("need two or more variables as list")
    else if length (%v) = 2 then %cL : []
    else if listp (third (%v)) then %cL : third (%v)
    else %cL : rest (%v, 2),
   /* display (%f, %vL, %cL), */
   nct : length(%cL),
   if nct = 0 then %xlamL : %vL
    else (
        for i thru nct do push(lam[i],%lamL),
        %lamL : reverse (%lamL)),
   /* display (%lamL), */
```

```
   if nct = 0 then %Lagr : %f
    else %Lagr : ratsimp (%f + apply ("+", %lamL*%cL)),
    print (" lagrangian = ",%Lagr),
    %xlamL : flatten ([%vL,%lamL]),
    if details then display (%xlamL),

   Lgrad : jacobian ([%Lagr], %xlamL)[1],
    if details then display (Lgrad),
    %solns : solve (Lgrad, %xlamL),
    print ("solve returns ", %solns),
    cp : [ ],  /* defines global list of solutions */
    /* screen %solns and retain only real and non-negative
       solutions */
       print ("optimum only evaluates real non-negative solutions"),
    if multiple(%solns) then for i thru length(%solns) do (
        asoln : %solns[i],
        if details then print ("i = ",i," asoln = ",asoln),
        if RealSoln (asoln) and NonNegative (asoln,%vL) then push (asoln, cp))
     else if Dsingle (%solns) or single (%solns) then (
          if Dsingle (%solns) then asoln : %solns[1] else asoln : %solns,
          if details then print (" asoln = ",asoln),
         if RealSoln (asoln) and NonNegative (asoln,%vL) then push (asoln, cp))
     else return(" problem getting %solns "),

     cp : reverse (cp),
      if details then print (" cp = ", cp),
     if length (cp) = 0 then return ("no real non-negative solutions"),

    if nct = 0 then (  /* case no constraints, use CPtest  */
        if multiple (cp) then for i thru length (cp) do
                  CPtest (%f, cp[i])
        else if Dsingle (cp) or single (cp) then (
             if Dsingle (cp) then asoln : cp[1] else asoln : cp,
             CPtest (%f, asoln))
        else return ("problem with cp form") )

    else /* case one or more equality constraints (h = 0) , use bordered hessian matrix test*/
         ( %BH : Bhessian (%Lagr,%cL,%vL),
           /* display (%BH), */
           if multiple (cp) then for i thru length (cp) do (
                 print ("----------------------------"),
                 print ("i = ",i," soln = ",cp[i],"  objsub = ",at (%f,cp[i])),
                 print (" soln = ",float(cp[i])," objsub = ", float(at (%f,cp[i]))),
                %LPM : BHtest ( at(%BH,cp[i]), nct, nV),
                 print (" LPM's = ", float(%LPM)))
           else if Dsingle (cp) or single (cp) then (
                if Dsingle (cp) then asoln : cp[1] else asoln : cp,
                print (" soln = ", asoln,"  objsub = ",at(%f,asoln)),
                print (" soln = ", float(asoln),"  objsub = ",float (at(%f,asoln))),
                %LPM : BHtest (at(%BH,asoln), nct, nV),
                 print (" LPM's = ", float(%LPM)) )
           else return("problem with cp form")),
   done)$

/* Statics(FUNCLIST, EXOGENOUSVARLIST, PARAM)
Comparative Statics Analysis for set of functions
 for which equilibrium is described by f(j) = 0,
 the number of exogenous variables should equal the
 number of functions, param is a single exogenous parameter.
 *** This function only works if the functions f(j) contain
       no implicit functions!

*/

 Statics (funcL, endogL, param) :=
   block ( [%labels, %labelL : [ ], %XS],
       %XS :  invert ( jacobian(funcL, endogL)) . (- jacobian (funcL, [param])),
```

```
       for j thru length(endogL) do
            %labelL : cons ( [sconcat (endogL[j], param)], %labelL),
       %labels : apply ('matrix, reverse(%labelL)),
      addcol(%labels, %XS))$

/* Msolve (J, X, Z) calls solve for solutions to the matrix
   equation J . X = Z. The elements of the matrix column vector X
   contain symbols which stand for the unknowns. J is a square n x n matrix
   and X and Z are each n dimensional matrix column vectors.
*/


Msolve (%J, %X, %Z) :=
  (solve (flatten (args ( %J . %X - %Z )), args (transpose (%X))[1] ))$

/* Hessian (F, varList) is an alternative to the Maxima function hessian */

Hessian (%func, %vL)  :=
block ([ %dL : [] ],
    for j thru length (%vL) do
         %dL : cons (diff (%func, %vL[j]), %dL),
    %dL : reverse (%dL),
    jacobian (%dL, %vL))$

/* multiple, Dsingle, and single help distinuish different types of lists
   returned by solve */

multiple(%X) :=
 (if length(%X) > 1 and part(%X,1,0) = "[" then true else false)$

Dsingle(%X) := (if length(%X) = 1 and part(%X, 1, 0) = "[" then true else false)$

single (%X) :=
 ( if length(%X) > 1 and part(%X, 1,0) = "=" then true else false)$

/*  examples of use:
   <if multiple (A) then for j thru length(A) do (...),>
    <if Dsingle(B) then B : B[1]>,  removes one layer of brackets.

(%i23)  s1 : [x = 3,y = 4, z = 5];
    ds1 : [s1];
    m2 : [s1,s1];
    m3 : [s1,s1,s1];
    m4 : [s1,s1,s1,s1];
(s1)    [x=3,y=4,z=5]

(ds1)   [[x=3,y=4,z=5]]

(m2)    [[x=3,y=4,z=5],[x=3,y=4,z=5]]

(m3)    [[x=3,y=4,z=5],[x=3,y=4,z=5],[x=3,y=4,z=5]]

(m4)    [[x=3,y=4,z=5],[x=3,y=4,z=5],[x=3,y=4,z=5],[x=3,y=4,z=5]]

(%i24)  map ('length, [s1,ds1,m2,m3,m4]);
(%o24)  [3,1,2,3,4]
(%i25)  map (lambda ([xx], part (xx, 1,0)), [s1,ds1,m2,m3,m4]);
(%o25)  ["=","[","[","[","["]

(%i27)  map ('single, [s1,ds1,m2,m3,m4]);
(%o27)  [true,false,false,false,false]
(%i28)  map ('Dsingle, [s1,ds1,m2,m3,m4]);
(%o28)  [false,true,false,false,false]
(%i29)  map ('multiple, [s1,ds1,m2,m3,m4]);
(%o29)  [false,false,true,true,true]

 */
```

```
/*    gridSearch with one or two constraints

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],max,g(x,y))
     returns [fmax, [xmax,ymax]] , and only looks at points for which g(x,y) >= 0.

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],min, g(x,y))
     returns [fmin, [xmin,ymin]] , and only looks at points for which g(x,y) <= 0.

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],max, g1(x,y), g2(x,y) )
     returns [fmax, [xmax,ymax]] , and only looks at points for which both
             g1(x,y) >= 0  and g2(x,y) >= 0.

gridSearch (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],min, g1(x,y), g2(x,y) )
     returns [fmin, [xmin,ymin]] , and only looks at points for which
        both g1(x,y) <= 0  and g2 (x,y) <= 0.

*/


gridSearch ([%vL]) :=
(if length (%vL) = 5 and length (%vL[5]) = 2 then apply ('gridSearch22, %vL)
    else if length (%vL) = 6 then
      apply  ('gridSearch22, append (rest (%vL,-2), [ rest (%vL,4)]))
    else if length (%vL) = 5 then  apply ('gridSearch11, %vL)
    else print ("incorrect syntax")  )$

/*

(%i27)  Pr : 64*x - 2*x^2 + 96*y - 4*y^2 - 13$
        g : 20 - x - y$

(%i28)  gridSearch(Pr, [x,10,11,0.1],[y, 9,10,0.1], max, g);
(%o28)  [989.66,[x=10.7,y=9.3]]
(%i29)  gridSearch(Pr, [x,10,11,0.1],[y, 9,10,0.1], max, [g]);
(%o29)  [989.66,[x=10.7,y=9.3]]
*/

/*
(%i30)  u : x*y$
    g1 : 144 - 3*x - 4*y$
    g2 : 120 - 5*x - 2*y$
(%i31)  gridSearch (u, [x,13.5,13.9,0.01], [y, 25.4,30,0.01], max, [g1,g2]);
(%o31)  [352.48,[x=13.71,y=25.71]]
(%i32)  gridSearch (u, [x,13.5,13.9,0.01], [y, 25.4,30,0.01], max, g1,g2 );
(%o32)  [352.48,[x=13.71,y=25.71]]

*/


/*   one constraint grid search function.

gridSearch11 (f(x,y),[x,x1,x2,dx], [y,y1,y2,dy],maxmin,g(x,y) )
   returns [fmax,[xmax,ymax] ] if maxmin = max and g(x,y) >=0
   or  [fmin, [xmin,ymin] ] if maxmin = min and g(x,y) <= 0
*/

gridSearch11(func,xR,yR,maxmin,constraint) :=
block ([%x,%y,%x1,%x2,%dx,%y1,%y2,%dy,fL:[],xyL:[], gval,
        nx,ny,pxy,fopt,iopt,xyopt ], local(%F,%g),
    if not member(maxmin,[max,min]) then return("maxmin = max or min"),
    if listp (constraint) then constraint : constraint[1],
    [%x,%x1,%x2,%dx] : xR,
    [%y,%y1,%y2,%dy] : yR,
    nx : floor((%x2 - %x1)/%dx),
    ny : floor((%y2 - %y1)/%dy),
   define( %F(%x,%y), func),
```

```
    define (%g(%x,%y), constraint),
    for i:0 thru nx do
          for j:0  thru ny do (
               pxy : [%x1 +i*%dx, %y1  + j*%dy],
                gval : apply (%g, pxy),
            /*  display (i,j,gval),  */
               if (maxmin = max and gval >= 0) or (maxmin = min and gval <= 0)  then (
                     push( pxy, xyL),
                     push ( apply (%F,pxy), fL))),
    xyL : reverse (xyL),
    fL : reverse (fL),
 /*   if details then display (xyL,fL), */
    if maxmin = max then fopt : lmax(fL)  else  fopt : lmin(fL),
    for i thru length(fL) do
         if fL[i] = fopt then iopt : i,
    xyopt : [ %x = first( xyL[iopt]), %y = second ( xyL[iopt]) ],
    [fopt, xyopt]  )$



/*  two constraint grid search function

gridSearch22 (f(x,y),[x,x1,x2,dx],[y,y1,y2,dy],maxmin,[g1,g2])
in which if maxmin = max, then g1(x,y) >= 0 and g2(x,y) >= 0
and if maxmin = min, then g1(x,y) <= 0 and g2(x,y) <= 0.
*/


gridSearch22(func,xR,yR,maxmin,gL) :=
block ([%x,%y,%x1,%x2,%dx,%y1,%y2,%dy,fL:[],xyL:[],
        g1val, g2val,  nx,ny,pxy,fopt,iopt,xyopt ], local(%F,%g1,%g2),
    if not member(maxmin,[max,min]) then return("maxmin = max or min"),
    if not listp (gL) then ( print (" gL = list of two constraint functions"),
           return(done)),
    [%x,%x1,%x2,%dx] : xR,
    [%y,%y1,%y2,%dy] : yR,
    nx : floor((%x2 - %x1)/%dx),
    ny : floor((%y2 - %y1)/%dy),
   define( %F(%x,%y), func),
    define (%g1(%x,%y), gL[1]),
    define (%g2(%x,%y), gL[2]),
    for i:0 thru nx do
          for j:0  thru ny do (
               pxy : [%x1 +i*%dx, %y1  + j*%dy],
               g1val : apply (%g1, pxy),
               g2val : apply (%g2, pxy),
            /*  display (i,j,g1val,g2val),  */
               if (maxmin = max and g1val >= 0 and g2val >= 0)
                     or (maxmin = min and g1val <= 0  and g2val <= 0)  then (
                         push( pxy, xyL),
                         push ( apply (%F,pxy), fL))),
    xyL : reverse (xyL),
    fL : reverse (fL),
 /*   if details then display (xyL,fL), */
    if maxmin = max then fopt : lmax(fL)  else  fopt : lmin(fL),
    for i thru length(fL) do
         if fL[i] = fopt then iopt : i,
    xyopt : [ %x = first( xyL[iopt]), %y = second ( xyL[iopt]) ],
    [fopt, xyopt]  )$

/****************** Simple Compounding Functions ***********/
/*
   nperiod (i, PV, FV) returns the
   number of compounding periods nPeriod needed to
   achieve the final value FV, given the initial value PV,
   and the interest rate per compounding period 'rate'.
*/
```

```
nPeriod (rate, %PV,%FV) :=
block ([numer:true],
    log (%FV/%PV) / log (1 + rate))$



/* interest (PV, FV, nperiods)  returns the ratio (i/m)
   in which nperiods = m*t and m = number of compoundings per year
   and t = number of years and i = decimal interest rate per year.
*/

interest(%P,%F,%n) :=
block ([numer:true],
    (%F/%P)^(1/%n) -1 )$

/*
   fv(i,PV,n)

The future value fv  of a **single** payment made n compounding periods in the past in a
financial environment in which investment oportunities are available which  yield a rate of
interest "rate" is gotten by compounding the past payment by n factors of the compounding
factor (1+rate).

*/

fv(rate, past_payment, num) := past_payment*(1+rate)^num$

/* fvm(rate, PV, m, n)
   calculates future value for compounding m times each of n periods,
   given present value and rate per period.
   for example: fvm(i,P,m,n) uses decimal interest rate i per year, with
   m compoundings per year, for n years, to calculate the future value
   of PV = P.
*/

fvm(rate,%P,%m,%n) :=
block ([numer:true],
    %P*(1 + rate/%m)^(%m*%n))$

/*

  pv(i,FV,n)

The present value pv  of a **single** payment made num compounding periods in the future in a
financial environment in which investment oportunities are available which  yield a rate of
interest "rate" is gotten by "discounting" the future payment by num factors of the "discount
factor" v = 1/(1+rate).
*/

pv(rate,future_payment,num):=future_payment/(1+rate)^num$

/*   pvm (i,FV,m,n)

present value of a future amount FV received n years in future with possible returns
   based on i interest rate per year compounded m times per year
*/


pvm (%ii,%FV, %m, %n) := %FV/(1 + %ii/%m)^(%m*%n)$

/*
  r =  rate(i,m) converts  FV = PV*(1 + i/m)^(m*t) = PV*exp(r*t)

*/

rate (%ii, %m) := %m*log(1 + %ii/%m)$
```

```
/* functions used in LPsimplex01.wxmx */

/* remove one item whose place in the list is
   the positive integer mm% from list aL%
   and return depleted list  */

remL1 (_aL% ,_mm%) :=
    block ([ln],
     ln : length (_aL%),
     if _mm% > ln then (
       disp ("remL1: n > length (list)"),
       return () ),

     if _mm% = 1 then rest ( _aL%, 1 )
     else if _mm% = ln then rest ( _aL%, -1 )
     else flatten ( cons (rest ( _aL%, -(ln - _mm% + 1 ) ),
                  rest ( _aL%, _mm% ) ) ) )$


/* remove items whose position in the list aaL are in the list of positive integers vvL
    and return the depleted list.
    Example: remL([x1,x2,x3,x4,x5],[2, 5]) returns [x1,x3, x4]

(%i342) remL([x1,x2,x3,x4,x5],[5, 2, 3]);
(%o342) [x1,x4]
(%i343) remL([x1,x2,x3,x4,x5],[ 2, 3, 5]);
(%o343) [x1,x4]

    */

remL(aaL,vvL) :=
  block ([ttL, vsL, nc:0, ve],
   ttL : copy (aaL),
    vsL : sort (vvL),
   for vv in vsL do (
        ve : vv - nc,
        ttL : remL1 (ttL, ve),
        nc : nc + 1),
   ttL)$

/* maxlp (obj, condL) is a "wrapper" for maximize_lp which simplifies the input and output.
The function maxlp(objective, condition-list) assumes all input (decision) variables are
to be assumed nonnegative. maxlp calls maximize_lp from the simplex package.  */

maxlp(obj, condL) :=
block( [mlpr, zmax, varmax  ],
    mlpr : maximize_lp (obj,condL,listofvars (condL)),
    if not listp (mlpr) then return (mlpr),
    zmax : mlpr[1],
    varmax : sort (mlpr[2]),
    print ("for z = ",obj, ","),
    print ( "such that, "),
    for j thru length (condL) do print (condL[j],","),
    print (" z*  = ",zmax," with ", simplode( join (varmax,
                     makelist (", ", j, 1, length (varmax))))),
    [zmax, varmax])$




/* lists vL and bL are global lists used by both
         tableau and pivot1, bL is used by tratio.  */

tableau (tRows) :=
 block ([mm, dtRows], local (dtR),
```

```
    mm : length(tRows),  /* mm = num rows */
    for j thru mm do
       dtR[j] : append (tRows[j], [bL[j]]),
    dtRows : [vL],
    for j thru mm do dtRows : cons (dtR[j], dtRows),
    apply (matrix, reverse (dtRows)))$

/* tratio (RL, ncol) assumes the z-row is the first row,
  and the remaining rows come from A.X = b. This agrees
  with the convention used by Butenko in his YouTube videos.
  */

tratio(RRL, ncol) :=
  block ([nrows, nel, ratioL : [] ,rvec ], local (ar),
    nrows : length(RRL),
    for j thru nrows do ar[j] : RRL[j],
    nel : length (RRL[1]),
    for j: 2 thru nrows do
        if ar[j][ncol]  <= 0 then ratioL : cons ("-", ratioL)
         else ratioL : cons ( ar[j][nel] / ar[j][ncol], ratioL),
    rvec : cvec ( reverse (ratioL)),
    float (addcol (rvec, cvec (rest (bL)))))$

/* bratio(RL, ncol) assumes the z-row is the last row,
   all but the last row are from A.X = B, and
   the list bL has the form [x4,x5,...,z]. This allows us to
   use the conventions used by B/N.
*/


bratio(RRL, ncol) :=
  block ([nrows, nel, ratioL : [] ,rvec ], local (ar),
    nrows : length(RRL),
    for j thru nrows do ar[j] : RRL[j],
    nel : length (RRL[1]),
    for j thru nrows - 1 do
        if ar[j][ncol]  <= 0 then ratioL : cons ("-", ratioL)
         else ratioL : cons ( ar[j][nel] / ar[j][ncol], ratioL),
    rvec : cvec ( reverse (ratioL)),
    float (addcol (rvec, cvec (rest (bL, -1)))))$

/* b2ratio(RL, ncol) assumes the last two rows are z-rows,
   all but the last two rows are from A.X = B, and
   the list bL has the form [x4,x5,...,z1,z2]. This allows us to
   use the conventions used by B/N in the "two phase simplex method".
*/



b2ratio(RRL, ncol) :=
  block ([nrows, nel, ratioL : [] ,rvec ], local (ar),
    nrows : length(RRL),
    for j thru nrows do ar[j] : RRL[j],
    nel : length (RRL[1]),
    for j thru nrows - 2 do
        if ar[j][ncol]  <= 0 then ratioL : cons ("-", ratioL)
         else ratioL : cons ( ar[j][nel] / ar[j][ncol], ratioL),
    rvec : cvec ( reverse (ratioL)),
    float (addcol (rvec, cvec (rest (bL, -2)))))$




pivot1 (Rows, pElement ) :=
block ([nR, nE, pR, pC,pRows ], local (rr),
    nR : length (Rows),
```

```
    nE : length (Rows[1]),
    pR : pElement[1],
    pC : pElement[2],
    print ("pivot row = ",pR," pivot col = ", pC, "value = ", Rows[pR][pC]),
    print ( vL[pC], "enters Basis,", bL[pR], "leaves Basis"),
    /* change global bL */
    bL[pR] : vL[pC],
    /* display (bL), */
    for j thru nR do rr[j] : Rows[j],

    /* set pivot element to value 1 */
    rr[pR] : ratsimp (expand ( rr[pR] / rr[pR][pC])),
    /* set other elements of pivot column to 0 */
    for j thru nR do
       if j # pR then rr[j] : ratsimp ( expand ( rr[j] - rr[j][pC]*rr[pR])),
    pRows : [],
    for j thru nR do pRows : cons (rr[j], pRows),
    pRows : reverse (pRows),
    print (tableau (pRows)),
    pRows)$


/* The following functions are used in the matrix simplex method.

In these functions we use the following global variables:
 c, X, A, b, NV, BV, N, B, Xn, Xb, cN, cB

 and start with solving the matrix equation
   A . X = b for X >=0, or with N a matrix of the columns of
   A corresponding to the non-basic variables (in order left to right)
   and B a matrix of the columns of A corresponding to the basic
   variables (in order left to right),
   N . Xn + B . Xb = b, which can be solved for Xb (the basic variables) in
   terms of the non-basic variables Xn:
   Xb = - invert(B) . N . Xn + invert(B) . b


*/

/*  MM is a matrix,
    colL is a list of column numbers of MM to be used for new matrix;
     thanks to Stavros Macrakis for this code. */

newM (MM, colL) := transpose (part ( transpose (MM), colL))$

zrow() := - (cN - cB . invert(B) . N)$
zrhs() := cB . invert(B) . b$
Ib() := ident (length (Xb))$


alias (lme, list_matrix_entries);

/* coefNV ()
   is the matrix of coefficients of Xn in the
   equation:  Xb + invert(B) . N . Xn = invert(B) . b
   which can be written as
        Xb + coefNV() . Xn = bnew()
   using our definitions here.  */

coefNV () := invert (B) . N$

colN(jj) := col ( coefNV(), jj)$

/* bnew()
    the matrix column of numbers labelled 'rhs'
   in the matrix tableau */
```

```
bnew() := invert (B) . b$

/* colNum (in the function Mratio) is the column number of the matrix of coefficients
        invert(B) . N and is the same as the column number of the row matrix zrow()
        which has the largest negative coefficient.
        The label of that column, given above by transpose(Xn),
         identifies the pivot variable which enters the Basis.
*/


Mratio(colNum) :=
  block ([bb, NC, RC : [ ] ],
    /*  bb : bnew (), */
    bb : invert(B) . b,
    /* NC : colN (colNum), */
     NC : col ( invert (B) . N, colNum),
     for j thru length (bb) do
        if bb[j, 1] = 0 or NC[j, 1] <= 0 then RC : cons ("-", RC)
          else RC : cons ( bb[j, 1]/NC[j, 1], RC),
    RC : reverse (RC),
    float (addcol ( cvec (RC), Xb)))$

BNV (Enter, Leave) :=
  (BV : sort (cons (Enter, delete (Leave, BV))),
    NV : sort (cons (Leave, delete (Enter, NV))))$

Mdisplay() := (display (NV, BV, N, B, Xn, Xb, cN, cB))$

Mdefine () :=
    (N : newM (A, NV),
    B : newM (A, BV),
    Xn : part (X, NV),
    Xb : part (X, BV),
    cN : transpose (part (c, NV)),
    cB : transpose (part (c, BV)),
    if details then Mdisplay ())$

Mtableau() :=
(matrix ( ["transpose(Xb)","transpose(Xn)"," ", " "],
        [transpose (Xb), transpose (Xn), "rhs", "Basis"],
    [0, zrow(), zrhs(), "z"],
    [Ib(), coefNV(), bnew(), Xb]))$

Mpivot(Enter, Leave) :=
(  print (X[Enter,1],"enters, ",X[Leave,1],"leaves Basis"),
    BNV (Enter, Leave),
    Mdefine(),
    Mtableau())$

/*   toMTab (rowsL)
uses global vL, bL, and the list of rows rowsL of tableau method to create equivalent
matrix tableau. Before calling toMTab for the first time in a given problem,
you must define the global matrix column vector c such that
 we start with max z = transpose(c) . X.

 Then toMTab(rowsL) defines global X, A, b, NV, BV,
 corresponding to the current set of rows in the tableau method
calling Mdefine() to define Xn, N, Xb, B, cN, cB which allows creation of the corresponding
matrix tableau  */



toMTab (RRows ) :=
  block ( [ nrr,  aL : [], srowsL : [], nsr, bbL : [], BVL : [ ],
        BVLn : [], NVLn : [] ], local (lrows),
    nrr : length (RRows),
    /* before calling Mdefine() we need to define
```

```
      X, c, A, b, NV, and BV;
    Before calling toMTab for the first time
    you must define the global matrix column
    vector c, such that we begin with
    max z = transpose(c) . X    */

    /* get X from global list vL  */
    X : cvec (rest (vL, -2)),

   /* display (X, c), */
    for j thru nrr - 1 do
      lrows[j] : append (RRows[j+1], [bL[j+1]]),
   /*  for j thru nrr - 1 do  print(j, lrows[j]), */
    /* sort tagged rows to same order as variables in X */
    for k thru length (X) do
         for j thru nrr-1 do
             if bL[j+1] = X[k, 1] then srowsL : cons (lrows[j], srowsL),
    srowsL : reverse(srowsL),
   /* for j thru nrr - 1 do print (j, srowsL[j]), */
    for j thru nrr - 1 do  aL : cons ( rest (srowsL[j], -2), aL),
    A : apply ('matrix, reverse (aL)),
    /*  display (A),  */
    nsr : length (srowsL[1]),
    for j thru nrr - 1 do bbL : cons (srowsL[j][nsr-1], bbL),
    b : cvec ( reverse (bbL)),
    /*  display (b),  */
    for j thru nrr-1 do BVL : cons (last (srowsL[j]), BVL),
    BVL : reverse( BVL),
    /*  display (BVL), */
    for k thru length(X) do
      for j thru length (BVL) do
       if BVL[j] = X[k,1] then BVLn : cons (k, BVLn),
    BVLn : reverse (BVLn),
    /* display (BVLn), */
    for k thru length(X) do
       if lfreeof(BVLn, k) then NVLn : cons (k, NVLn),
    NV : reverse (NVLn),
  /*    display (NV),  */
    BV : BVLn,
    Mdefine(),
    Mtableau())$



/* ========================== */

/* functions used in LPmatrix.wxmx which put the z-row(s) on
the bottom and make the displayed results easier to read and
interpret;

Mbtableau and Mbpivot are  matrix method functions which place the "z-row" on the bottom.  */

Mbtableau() :=
block ([ IIb, ccoefNV, bbnew, mrr : []], local (rr),
    IIb : copy (Ib()),
    ccoefNV  : copy (coefNV ()),
    bbnew : copy (bnew()),
    rr[0] : append (lme (transpose (Xb)),[|], lme (transpose (Xn)), [|, "rhs", "Basis"] ),
    for j thru length (BV) do
       rr[j] : append (IIb[j], [|], ccoefNV[j],[|], bbnew[j], Xb[j]),
    rr[1 + length(BV)] : append (makelist (0,k,1,length(Xb)), [|], lme (zrow()),[ |, zrhs(),
    "z"]),
    for j : 0 thru 1 +  length(BV) do
      mrr : cons (rr[j], mrr),
    apply ( 'matrix, reverse (mrr)))$
```

```
Mbpivot(Enter, Leave) :=
(  print (X[Enter,1],"enters, ",X[Leave,1],"leaves Basis"),
    BNV (Enter, Leave),
    Mdefine(),
    Mbtableau())$

/* DMratio (rowNum) is used to look for the minimum absolute value of the
  ratios of the z-row coefficients to the non-basic work row negative coefficients
  when using B/N's "Dual Simplex Method". See LPmatrix.wxmx  */

DMratio(rowNum) :=
  block ([zL, wL, RC : [ ] ],
     zL : lme (zrow()),
    wL : lme (row (coefNV(), rowNum)),
     for j thru length (zL) do
        if wL[j] >= 0 then RC : cons ("-", RC)
          else RC : cons ( abs (zL[j]/wL[j]), RC),
    RC : reverse (RC),
    transpose (float (addcol (Xn, cvec (RC) ))))$

/*
  killAB() kills all but the functions included in the
    internal parenthesis: kill (allbut (...))
*/

killAB() := kill (allbut (Extr, gradient, CP, Analyze,
                      plotCP, colorMap,contourPlot,contours3d,LPM,
                      Minor, Qtest, CPtest, cvec,InputOutput,
                     Leontief, Pinput, PItot, Bhessian,BLPM,
                     BHtest, Statics, Hessian, Msolve,multiple,
                     Dsingle, single,gridSearch, gridSearch11,gridSearch22,
                      KKT,KKTmax,KKTmax2,KKTmin,KKTmin2,optimumAll,
                     RealSoln,NonNegative, optimum,nPeriod, interest,
                     fv, fvm, pv,pvm,rate, details, killAB) )$

/* set global parameter details */

details : false$
```