```
/* Econ2.mac
    Maxima software for Economic Analysis
    Ted Woollett, April 7, 2022
    http://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/.
*/

/* DE1 (a,b,t,y0) or DE1 (a,b,t) returns the solution of
  the first order difference equation
    y[t] = b*y[t-1] + a.
   The optional fourth argument y0 is taken as a given y[0].
   If y0 is not specified, the solution is returned in a
   form which includes an undefined constant %A.
   This function is used in Dowling17.wxmx.
*/


DE1 ( [argL] ) :=
  block ([ %r ],
    if length (argL) = 4 then (
         if argL[2] = 1 then argL[4] + argL[1]*argL[3]
         else (
               %r : argL[1]/(1 - argL[2]),
               (argL[4] - %r)*argL[2]^argL[3] + %r))
   else if length (argL) = 3 then (
          kill (%A),
          if argL[2] = 1 then
                %A + argL[1]*argL[3]
          else (
               %r : argL[1]/(1 - argL[2]),
               %A*argL[2]^argL[3] + %r))
   else (
          print ("syntax error"),
          done))$

/* linear second order difference equations with constant coefficients */

/* Given the 2nd order linear difference equation
    y[t] + b1*y[t-1] + b2*y[t-2] = a,
with constants b1,b2, a, yp[t] is returned by the Maxima function
ypart (b1,b2,a,t).
*/

ypart (_b1,_b2,_a, _t) :=
 (if _a = 0 then 0
    else if (_b1 + _b2 = -1 and _b1 = -2) then (_a/2)*_t^2
    else if (_b1 + _b2 = -1) then (_a*_t)/(2 + _b1)
else _a/(1 + _b1 + _b2))$

/* and yc[t] is given by the Maxima function ycompl (b1, b2, t).  */

ycompl (_b1,_b2,_t) :=
   block ( [%r1,%r2,%r,sarg,%g,%h,%k,%th],
```

```
     kill(%A1,%A2),
    sarg : _b1^2  - 4*_b2,
    /* assume coefficients are numerical here */
    if sarg = 0 or float (sarg) = 0.0 then (
        /* case repeated real roots */
        %r : -_b1/2,
        %A1*%r^_t + %A2*_t*%r^_t)
    else if sarg > 0 then (
        /* case distinct real roots */
        %r1 : (-_b1 + sqrt(sarg))/2,
        %r2 : (-_b1 - sqrt(sarg))/2,
        %A1*%r1^_t + %A2*%r2^_t)
    else (
        /* case sarg < 0, complex roots */
        %g : -_b1/2,
        %h : sqrt(-sarg)/2,
        %k : sqrt (%g^2 + %h^2),
        if %g = 0 then %th : asin(%h/%k) else %th : acos (%g/%k),
        (%k^_t)*(%A1*cos (%th*_t) + %A2*sin (%th*_t))))$

/* indefinite solution ysoln (b1,b2,a,t)
     given the 2nd order linear difference equation
         y[t] + b1*y[t-1] + b2*y[t-2] = a,
 */

ysoln (bb1,bb2,aa,tt) := (ypart (bb1,bb2,aa,tt) + ycompl (bb1,bb2,tt))$



/* do loop code to produce a list of values y[t]:
    ytList(b1,b2,a,y0,y1,tmax),
    given the 2nd order linear difference equation
    y[t] + b1*y[t-1] + b2*y[t-2] = a
*/


ytlist(bb1,bb2,aa,yy0,yy1, ttmax) :=
block ([ ytm1 : yy1, ytm2 : yy0, yyt,yyp, LL ],
    if freeof(t, aa) then (
      yyp : ypart (bb1,bb2,aa,t),
      print (" ye = ",yyp)),
    LL : [yy1, yy0],
    ytm1 : yy1,
    ytm2 : yy0,
     for tt:2 thru ttmax do(
        yyt : -bb1*ytm1 - bb2*ytm2 + subst (tt,t,aa),
        LL : cons (yyt, LL),
        ytm2 : ytm1,
        ytm1 : yyt),
    reverse (LL))$

/*  DE1S (A, B, t, Y0) or DE1S (A, B, t)
     returns Y[t], the matrix column vector solution
      of a set of n first order difference equations
      expressed in matrix form as
        Y[t] = A . Y[t-1] + B.
     Y0 is the optional matrix column vector of initial
       values.
      Y[t] has with n components if A is a
     square n x n matrix, and is returned in definite form if
     the fourth arg Y0 is included, else in terms
     of n constants _k[j].
*/



DE1S ( [argL]  ) :=
```

```
block ([ _A,  _B,  _t,  _n, _Ye, _eval, _evec, _Yindef], local(_r,_V,_k),
      _A : argL[1],
      _B : argL[2],
       _t : argL[3],
      _n : length (_A),
     _Ye : invert (ident(_n) - _A) . _B,
   /* display (_Ye), */
    [_eval, _evec] : eigenvectors (_A),
    if length (_eval[2]) < _n then return ("DE1S is not able to handle repeated roots"),
   /* display (_eval, _evec), */
    for j thru _n do (
        _r[j] : _eval[1][j],
        _V[j] : cvec (_evec[j][1])),
    _Yindef : _Ye + sum (_k[j]*_r[j]^_t*_V[j], j, 1, _n),
/*     display (_Yindef),      */
    if length (argL) = 3 then _Yindef
     else   at (_Yindef, colVecSolve (at (_Yindef, _t = 0), argL[4])))$

/* DE2S (A1, A2, B, t, Y0)

The Maxima function DE2S (A1, A2, B, t, Y0), uses matrix methods to solve
for the solution of A1 . Y[t] = A2 . Y[t - 1] + B, in which Y[t] is a matrix column vector with
n elements
depending on t, A1 and A2 are square n x n matrices of numerical elements, and B is either a
given n element
matrix column vector (with constant numerical values), or the number 0. Finally Y0 is an n
element
numerical matrix column vector giving the desired initial values Y(0).

For a problem in which there is no B term, and we are solving A1 . Y[t] = A2 . Y[t-1], with
Y[0] = Y0,
we can either use DE2S (A1,A2, 0, Y0) or  DE2S (A1,A2, zeromatrix (n, 1), Y0), in which
n = length(A) = length (Y0).

The solution method is to multiply every term by invert(A1) (provided det(A1) # 0), which
reduces the set of equations to the form Y[t] = D . Y[t-1] + E, and then call DE1S.

  */

DE2S (_A1, _A2, _B, _t, _Y0) :=
block (
   if det (_A1) = 0 then return (" DE2S method depends on det (A1) not being zero"),
   DE1S (invert (_A1) . _A2, invert (_A1) . _B, _t, _Y0))$




/* Ytlist (A, B, Y0, tmax) produces a column vector
    whose jth component is the list of values
    [yj[0], yj[1], yj[2],...,yj[tmax] ]
    using directly the first order difference equation

      Y[t] = A . Y[t-1] + B

    repeatedly, starting with Y[1] = A . Y0 + B,
    without using the eigenvalues or eigenvectors of A.
*/



Ytlist (AA, BB, YY0, ttmax) :=
block ([nn,YYp,YYt, CCL : [] ], local (yyL),
    nn : length (AA),
    for j thru nn do yyL[j] : [YY0[j,1]],
    YYp : YY0,
    for tv thru ttmax do (
        YYt : AA . YYp + BB,
```

```
        for j thru nn do yyL[j] : cons (YYt[j,1], yyL[j] ),
        YYp : YYt),
    for j thru nn do yyL[j] : [reverse ( yyL[j] )],
    for j thru nn do CCL : cons (yyL[j], CCL),
    apply ('matrix, reverse (CCL) ) )$







/*
  pIUlist (g, j, βk, m, y0, y1, tmax), produces a list of
   values of the inflation rate p[t],
       p[0] = y0, p[1] = y1, p[2], ..., p[tmax],
     corresponding to given numerical values of g, j, β*k, m, y0, y1, and tmax.
     This is based on the model of inflation and unemployment in discrete time
    (Chiang and Wainwright, Sec 18.3) discussed in the worksheet Dowling18C.wxmx.

After calculating b1, b2, and a, corresponding to
     the 2nd order linear difference equation
      p[t] + b1*p[t-1] + b2*p[t-2] = a  or equivalently
       p[t+2] + b1*p[t+1] + b2*p[t] = a,
   this function calls ytlist (b1,b2,a,y0,y1,tmax).

*/

pIUlist (gg,jj,betak,mm,%y0,%y1,%tmax) :=
block ([bb1,bb2,aa],
    /* check params */
    if betak <= 0 then return("need β*k > 0"),
    if gg < 0 or gg > 1 then return("need 0 < g <= 1"),
    if jj < 0 or jj > 1 then return("need 0 < j <= 1"),
    if not numberp (mm) then return (" m must be a number"),
    aa : jj*mm*betak/(1 + betak),
    bb1 : -(1 + gg*jj + (1 - jj)*(1 + betak))/(1 + betak),
    bb2 : (1 - jj*(1 - gg))/(1 + betak),
    ytlist (bb1,bb2,aa, %y0, %y1,%tmax))$






/*

 Multiplier-Accelerator model do loop code
     MAlist (, b, IaG0, Y0, Y1, tmax)
     produces a list of lists of the form
     [t, Ct, Idt, IaG0, Yt]
*/




MAlist (aa,bb,iag0,yy0,yy1, ttmax) :=
block ([ ytm1 : yy1, ytm2 : yy0, Ct, Idt, Yt, LL ],
    Ye : iag0/(1 - bb),
    display (Ye),
    LL : [ ["t", "Ct","Idt","Ia + G0","Yt"] ],
    LL : cons ([0, 0, 0, iag0, yy0], LL),
    LL : cons ([1, bb*yy0, 0, iag0, yy1], LL),
    ytm1 : yy1,
    ytm2 : yy0,
     for tt:2 thru ttmax do(
        Ct : bb*ytm1,
        Idt : aa*bb*(ytm1 - ytm2),
        Yt : Ct + Idt + iag0,
```

```
        LL : cons ([tt, Ct, Idt,iag0, Yt], LL),
        ytm2 : ytm1,
        ytm1 : Yt),
    reverse (LL))$

/*  if maL is the list produced by MAlist,  apply ('matrix, maL)
       will turn the output of MAlist into a formatted table.
*/



/* makeLists(maL) produces sublists tL, CtL, IdtL, IaG0L, YtL
   from the output of MAlist.
*/

aL(LL, num) := makelist (LL[j][num], j, 2, length (LL))$

makeLists (alist) :=
  (tL : aL(alist,1),
    CtL : aL(alist,2),
    IdtL : aL(alist,3),
    IaG0L : aL(alist,4),
    YtL : aL(alist,5),
    done)$




/*
For 2nd order ODE's with constant coefficients and terms with the general
form  y'' + A*y' + B*y(t) = C,an alternative approach to using ode2 is to
design a Maxima function which requires us to decide (ahead of time) on the
relative sizes of A,B,C (or express them symbolically) and ask for the
specific cases: complex roots (sines and cosines) if A^2 < 4*B,
distinct real roots (if A^2 > 4*B),
or double real roots (if A^2 = 4*B).
For the arg 'type' in Lode2 then use either complex, real, or double.
This function is used in Dowling18A.wxmx.
*/

/* linear ode2 with constant coefficients
   y'' + A y' + B y(t) = C

   Lode2(y,t,type,A,B,C)

   where type = either real, complex, or double.
*/



Lode2(%y,%t,%type,%A,%B,%C) :=
block ([_omega, _r,_term, _arg  ],
    if %A = 0 or %B = 0 then
        ode2('diff(%y,%t,2) + %A*'diff(%y,%t) + %B = %C, %y, %t)
     else (
    kill(%k1,%k2),
    _r : %A/2,
    if %C = 0 then _term : 0
      else (
           _term : %C/%B,
            _term : (-num(_term))/(-denom(_term))),

    if %type = real then (
        print(" this assumes that ", %A^2, " > ",4*%B ),
         _arg : %A^2 - 4*%B,
         if numberp (_arg) then
            if _arg < 0 then return ("assumption incorrect"),
      /*  print ("_arg = ", _arg), */
       if dofactor then _arg : factorsum (_arg),
```

```
      /*  print ("_arg = ", _arg), */
        _omega : sqrt(_arg)/2,
       /*  print ("_omega = ", _omega), */
     if doratsimp then   %y = %k1*exp(- ratsimp((_r - _omega))*%t)
                + %k2*exp(- ratsimp((_r + _omega))*%t)  + _term
        else
          %y = %k1*exp(- (_r - _omega)*%t)
                + %k2*exp(- (_r + _omega)*%t)  + _term)
    else if %type = complex  then (
        print (" this assumes that ", 4*%B, " > ", %A^2),
          _arg : 4*%B - %A^2,
          if numberp (_arg) then
            if _arg < 0 then return ("assumption incorrect"),
             /*   print ("_arg = ", _arg), */
         if dofactor then _arg : factorsum (_arg),
               /*  print ("_arg = ", _arg), */
        _omega : sqrt (_arg)/2,
       /*  print ("_omega = ", _omega), */
        %y = exp(-_r*%t)*(%k1*sin(_omega*%t) + %k2*cos(_omega*%t))
               +  _term)
    else if %type = double then (
        print (" this assumes that ", %A^2, " = ", 4*%B),
        if %C = 0 then _term : 0
         else (
               _term : 4*%C/%A^2,
                _term : (-num(_term))/(-denom(_term))),
        %y = (%k1 + %k2*%t)*exp (- _r*%t) + _term )
    else print ("type should be real, complex, or double")))$

/*
  croots (A,B)
  1. returns the solutions of the equation r^2 + A*r + B = 0.
  2. returns the characteristic roots of the linear
    second order ordinary differential equation (ode) with constant
    coefficients: y'' + A*y' + B*y = C.

*/


croots(%A,%B) := reverse (map ('rhs, solve (rr^2 + %A*rr + %B = 0, rr)))$

/*
  with symbolic A,B:

(%i6)   croots (A,B);
(%o6)   [ (sqrt(A^2 - 4*B) -A)/2,  -(sqrt(A^2 - 4*B)+A)/2  ]



  croots (A,B), numer;
  can also be used with numerical values of A and B.

  [r1, r2] : croots(A,B) assigns the first element of the
  list returned to the symbol r1, the second to r2.

*/

squarep (MM) :=
 block (if length (MM) = length (transpose (MM)) then true else false)$

det (MM) := block(if squarep (MM) then determinant (MM)
    else return ("matrix must be square"))$

mtrace (MM) :=
block ( if squarep (MM) then sum (MM[j, j], j,1, length (MM))
    else return (" matrix must be a square matrix"))$

/* Mcroots(M) assumes length(M) = 2  */
```

```
Mcroots (MM) :=
block ([MSqrt, MTr],
    if squarep (MM) then (
    MTr : mtrace (MM),
    MSqrt : sqrt ( MTr^2 - 4* abs (det (MM))),
   [ (MTr + MSqrt)/2,  (MTr - MSqrt)/2])
   else return (" matrix must be a square matrix"))$


/* following functions also defined
   in mbe5.mac
*/

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

alias(lme, list_matrix_entries)$

/*
 to_solve(Ab,xL)
  generate list of n equations from augmented matrix Ab with n rows
   and n+1 columns, given n-element list xL

(%i26) xL;
(%o26) [x1,x2]
(%i27) Cd;
(%o27) matrix([1,1,2],[0,1,1])
(%i28) to_solve(Cd, xL);
(%o28) [x2+x1 = 2,x2 = 1]
*/

to_solve(Ab1, xL1) :=
block( [rL, eqn1, eqn_list : [], j],
     for j thru length(xL1) do (
         rL : lme (row (Ab1,j)),
         eqn1 : rest(rL,-1) . xL1 = last(rL),
         eqn_list : cons (eqn1, eqn_list)),
    reverse (eqn_list))$


/*

solve_aug (M,L)
  solves for the n unknowns in a list L, given
  the augmented matrix M for a system of n equations.
  Calls to_solve defined above and then solve.
*/


solve_aug(Bc1,zL1) :=
block([ssL],
    ssL : to_solve(Bc1,zL1),
    ssL : solve(ssL, zL1),
    if length(ssL) = 1 then first(ssL)
       else ssL)$

/*
(%i31) xL;
(%o31) [x1,x2]
(%i32) Cd;
(%o32) matrix([1,1,2],[0,1,1])
(%i33) solve_aug(Cd,xL);
(%o33) [[x1 = 1,x2 = 1]]
```

```
*/

/*
Solve for n unknown coefficients in n given equations implied
by the matrix equation B = C, in which B and C are each column vectors
of length n.
*/

colVecSolve(%B, %C) :=
block ([%eqnL : [], num : length (%B) ],
    if length (%C) # num  then return (" column vecs must be same length"),
    for j thru num do %eqnL : cons (%B[j,1] = %C[j,1], %eqnL),
    solve (%eqnL)[1])$


ODE1S_debug (%A, %B, %Y0) :=
block ([%ev,%mul,%evec,%n, %abort, %Yindef], local (%r, %V, %k),
    if not squarep (%A) then return (" A must be a square matrix"),
    %n : length (%A),
    if length (%Y0) # %n then return ("Y0 must be a matrix column vector of length ", %n),
    if numberp(%B) then if %B = 0 then %B : zeromatrix (%n, 1)
       else return ("B must be a zeromatrix of length ",%n," or the number 0"),
    if length (%B) # %n then return ("B must be a matrix column vector of length ",%n,
          " or else the number 0"),
    [%ev, %evec] : eigenvectors (%A),
    display (%ev, %evec),
    %mul : %ev[2],
    display (%mul),
    for j thru %n do (
        print ("j = ", j," %mul[j] = ", %mul[j]),
       if %mul[j] # 1 then %abort : true),
    if %abort  then return (" Found repeated eigenvalue"),
    for j thru %n do (
        %r[j] : %ev[1][j],
        print ("j = ", j," r[j] = ",%r[j])),
    for j thru %n do (
        %V[j] : cvec (%evec[j][1]),
        print ("j = ", j, " evec[j] = ", %V[j])),
    %Yindef : - invert (%A) . %B + sum (%k[j]*exp (%r[j]*t)*%V[j], j, 1, %n),
    display (%Yindef),
    %solns : colVecSolve (at (%Yindef, t = 0), %Y0),
   return ( at (%Yindef, %solns)))$

/* ODE1S (A, B, Y0)

The Maxima function ODE1S (A, B, Y0), uses matrix methods to solve
for the solution of dY/dt = A . Y(t) + B, in which Y is a matrix column vector with n elements
depending on t, A is a square n x n matrix of numerical elements, and B is either a given n
element
matrix column vector (with constant numerical values), or the number 0. Finally Y0 is an n
element
numerical matrix column vector giving the desired initial values Y(0).

For a problem in which there is no B term, and we are solving dY/dt = A . Y(t), with Y(0) = Y0,
we can either use ODE1S (A, 0, Y0) or  ODE1S (A, zeromatrix (n, 1), Y0), in which
n = length(A) = length (Y0).

  */



ODE1S (%A, %B, %Y0) :=
block ([%ev,%mul,%evec,%n, %abort : false, %Yindef], local (%r, %V, %k),
    if not squarep (%A) then return (" A must be a square matrix"),
    %n : length (%A),
    if length (%Y0) # %n then return ("Y0 must be a matrix column vector of length ", %n),
    if  (matrixp(%B) and length (%B) # %n) then
         return ("B must be a matrix column vector of length ",%n," or the number 0"),
```

```
    if (numberp(%B) and %B = 0) or (matrixp(%B) and %B = zeromatrix (%n, 1)) then
               return (expand (matrixexp (%A, t) . %Y0)),
     /* case %B present in problem */
    [%ev, %evec] : eigenvectors (%A),
    %mul : %ev[2],
    for j thru %n do  if %mul[j] # 1 then %abort : true,
    if %abort  then return (" Found repeated eigenvalue"),
    for j thru %n do (
        %r[j] : %ev[1][j],
        %V[j] : cvec (%evec[j][1])),
    %Yindef : - invert (%A) . %B + sum (%k[j]*exp (%r[j]*t)*%V[j], j, 1, %n),
    %solns : colVecSolve (at (%Yindef, t = 0), %Y0),
   return ( at (%Yindef, %solns)))$

/*

(%i87)  Ys : ODE1S (matrix ([5, -0.5], [-2, 5]), cvec ([-12, -24]), cvec ([12, 4]));
(Ys)    matrix([5*%e^(6*t)+4*%e^(4*t) + 3.0],[-10*%e^(6*t)+ 8*%e^(4*t) + 6.0])
(%i88)  at (Ys, t = 0);
(%o88)  matrix([12.0],[4.0])

(%i89)  Ys : ODE1S (matrix ([6,5], [1,2]), 0, cvec ([4, 1]));
(Ys)    matrix( [(25*%e^(7*t))/6 - %e^t/6],[(5*%e^(7*t))/6 + %e^t/6])
(%i90)  at (Ys, t = 0);
(%o90)  matrix([4],[1])

(%i91)  Ys : ODE1S (matrix ([6,5], [1,2]), zeromatrix (2,1), cvec ([4, 1]));
(Ys)    matrix([(25*%e^(7*t))/6 - %e^t/6],[(5*%e^(7*t))/6 + %e^t/6])
(%i92)  at (Ys, t = 0);
(%o92)  matrix([4],[1])

*/

/* ODE2S (A1, A2, B, Y0)

The Maxima function ODE2S (A1, A2, B, Y0), uses matrix methods to solve
for the solution of A1 . dY/dt = A2 . Y(t) + B, in which Y is a matrix column vector with n
elements
depending on t, A1 and A2 are square nxn matrices of numerical elements, and B is either a
given n element
matrix column vector (with constant numerical values), or the number 0. Finally Y0 is an n
element
numerical matrix column vector giving the desired initial values Y(0).

For a problem in which there is no B term, and we are solving A1 . dY/dt = A2 . Y(t), with Y(0)
= Y0,
we can either use ODE2S (A1,A2, 0, Y0) or  ODE2S (A1,A2, zeromatrix (n, 1), Y0), in which
n = length(A) = length (Y0).

The solution method is to multiply every term by invert(A1) (provided det(A1) # 0), which
reduces the set of equations to the form dY/dt = D . Y + E, and then call ODE1S.

  */

ODE2S (_A1, _A2, _B, _Y0) :=
block (
   if det (_A1) = 0 then return (" ODE2S method depends on det (A1) not being zero"),
   ODE1S (invert (_A1) . _A2, invert (_A1) . _B, _Y0))$



/*  fll(L) returns the first element of the list L, the last element of the list L,
    and the length of the list L  */


fll(x) := [first(x),last(x),length(x)]$
```

```
/* Dowling ch 20 */

NumSufficient(d11,d1,d21,d2) :=
 block (
    if not numberp (d11) then display(d11),
    if not numberp (d1) then display (d1),
    if not numberp (d21) then display(d21),
    if not numberp (d2) then display (d2),
    if (d11 < 0 and d1 > 0) and (d21 < 0 and d2 > 0)  then  print ("global maximum")
    else if (d11 <= 0 and d1 >= 0) and  (d21 <= 0 and d2 >= 0) then print ("relative maximum")
    else if (d11 > 0 and d1 > 0) and (d21 > 0 and d2 > 0) then  print ("global minimum")
    else if (d11 >= 0 and d1 >= 0) and  (d21 >= 0 and d2 >= 0) then print ("relative minimum")
    else print ("neither maximized nor minimized: saddle point"))$




Extremal(FF) :=
block ([ffx,ffxp,ffxpt,ffxpx,ffxpxp,xsoln,varOde, AA,BB, CCL : [ ], CC, Eqn],
    kill(%k1),
    ffx : diff (FF,x),
    ffxp : diff (FF, xp),
    ffxpt : diff (ffxp, t),
    ffxpx : diff (ffxp,x),
    if not lfreeof ([x, xp, xpp], listofvars (ffxpx)) then (print(" nonlinear ode"),
    return(done )),
    ffxpxp : ratsimp (diff (ffxp, xp)),
    if not lfreeof ([x, xp, xpp], listofvars (ffxpxp)) then (print(" nonlinear ode"),
    return(done )),
    if details then  display (ffx, ffxp, ffxpt, ffxpx, ffxpxp),
   /* print ("Euler's Equation"),
    print (ffx - ffxpt - ffxpx*xp  - ffxpxp*xpp," = 0"),  */
    /* no derivatives case */
    if ffxpx = 0 and ffxpxp  = 0 then (
       xsoln : solve(ffx - ffxpt,x)[1],
    /*    display (xsoln),  */
        return (xsoln))
    /* no xpp term case : fxpxp = 0, fxpx # 0 */
    else if ffxpxp = 0 then (
        xsoln : x = %k1 + integrate ( (ffx - ffxpt)/ffxpx, t),
    /*    display (xsoln),  */
        return (xsoln)),
    /* case  we have an xpp term */
    varOde : ffx - ffxpt - ffxpx*xp - ffxpxp*xpp,
    if details then display (varOde),
    varOde : expand (varOde/coeff (varOde,xpp)),
    if details then display (varOde),
    AA : coeff(varOde, xp),
    BB : coeff (varOde,x),
    if details then display (AA, BB),
    for j thru length (varOde) do
       if lfreeof ([x, xp, xpp], part (varOde, j)) then CCL : cons (- part(varOde,j), CCL),
    if details then display (CCL),
     CC : apply ("+", CCL),
    if details then display (CC),
     Eqn : xpp + AA*xp + BB*x = CC,
    print ("ode: ", Eqn),
    Eqn : 'diff (x,t,2) + AA*'diff (x,t) + BB*x = CC,
    ode2(Eqn, x, t))$

NumSuffCond(F) :=
 block ([H1, H2, D11,D1, D21, D2 ],
    H1 : hessian (F, [x,xp]),
    if details then display (H1),
    D11 : H1[1,1],
```

```
   if details then display (D11),
    D1 : float (determinant (H1)),
   if details then display (D1),
    H2 : hessian (F, [xp, x]),
   if details then display (H2),
    D21 : H2[1,1],
    if details then  display (D21),
    D2 : float (determinant (H2)),
    if details then display (D2),
    NumSufficient (D11, D1, D21, D2),
    done)$

NumDynamic (F) :=
block ([cdex],
   cdex : Extremal (F),
   print (" candidate extremal: ", cdex),
    NumSuffCond (F), done)$



Dynamic (F) :=
block ([H1, H2, D11,D1, D21, D2, Fx, Fxp, Fxpt, Fxpx, Fxpxp,
        xp, xpp, ode, A, B, C, eqn, soln ],
    H1 : hessian (F, [x,xp]),
/*   display (H1),  */
    D11 : H1[1,1],
/*    display (D11), */
    D1 : float (determinant (H1)),
/*    display (D1), */
    H2 : hessian (F, [xp, x]),
/*     display (H2),  */
    D21 : H2[1,1],
/*    display (D21), */
    D2 : float (determinant (H2)),
/*    display (D2), */
    sufficient (D11, D1, D21, D2),
    Fx : diff (F,x),
    Fxp : diff (F, xp),
    Fxpt : diff (Fxp, t),
    Fxpx : diff (Fxp,x),
    Fxpxp : diff (Fxp, xp),
/*    display (Fx, Fxp, Fxpt, Fxpx, Fxpxp), */
    ode : Fx - Fxpt - Fxpx*xp - Fxpxp*xpp,
    ode : expand (ode/coeff (ode,xpp)),
    A : coeff(ode, xp),
    B : coeff (ode,x),
/*     display (A, B),     */
    for j thru length (ode) do
      if numberp (part (ode, j)) then C : - part(ode, j),
    if not numberp(C) then C : 0,
/*    display (C),  */
    eqn : 'diff (x,t,2) + A*'diff (x,t) + B*x = C,
/*    display (eqn), */
    soln : ode2(eqn, x, t),
/*     display (soln), */
    print (" candidate extremal is "),
    print (" x(t) = ", rhs(soln)),
    done)$

/* Dowling Ch. 21 code */

ConcaveTest (FF, xx, yy) :=
   block ([ d1,d11, dd1, d2, d21, dd2, %simply : false],
    d1 : hessian (FF, [xx, yy]),
    display (d1),
    d11 : d1[1,1],
    display (d11),
```

```
    dd1 : determinant (d1),
    display (dd1),
    if d11 < 0 and dd1 > 0 then (
         print("strictly concave"),
          return (done)),
    if d11 <= 0 and dd1 >= 0 then (
       /* print("check simply"), */
        d2 : hessian (FF, [yy, xx]),
        display (d2),
        d21 : d2[1,1],
        display (d21),
        dd2 : determinant (d2),
        display (dd2),
          if d21 <= 0 and dd2 >= 0 then %simply : true),
    if %simply then print ("simply concave")
     else  print (" no sufficient conditions"),
     done)$



Zindef (%A, %B ) :=
block ([%ev,%mul,%evec, %n, %abort : false ], local (%r, %V, %k),
    %n : length (%A),
    [%ev, %evec] : eigenvectors (%A),
    %mul : %ev[2],
    for j thru %n do  if %mul[j] # 1 then %abort : true,
    if %abort  then return (" Found repeated eigenvalue"),
    for j thru %n  do (
        %r[j] : %ev[1][j],
        %V[j] : cvec (%evec[j][1])),
     - invert (%A) . %B + sum (%k[j]*exp (%r[j]*t)*%V[j], j, 1, %n))$

Zindef_float (%A, %B ) :=
block ([%ev,%mul,%evec, %n, %abort : false ], local (%r, %V, %k),
    /* display (%A, %B),  */
    %n : length (%A),
    [%ev, %evec] : eigenvectors (%A),
    %mul : %ev[2],
    for j thru %n do  if %mul[j] # 1 then %abort : true,
    if %abort  then return (" Found repeated eigenvalue"),
    %ev : float(%ev),
    %evec : float(%evec),
    for j thru %n  do (
        %r[j] : %ev[1][j],
        %V[j] : cvec (%evec[j][1])),
     - invert (%A) . %B + sum (%k[j]*exp (%r[j]*t)*%V[j], j, 1, %n))$




dofactor : true$
doratsimp : true$
details : false$
fpprintprec:5$
ratprint:false$
```