TR_1D_model1_SS\DAE_SS_solver_1.m

```
% TR_1D_model1_ss\DAE_SS_solver_1.m
%
% function [x_state,iflag_converge,f,f_init] = ...
%    DAE_SS_solver_1(x_guess,Solver,...
%    func_calc_A_int,func_calc_b_int, ...
%    func_implement_BC,epsilon,Param);
%
% This procedure is a generic solver for a
% DAE system arising from a discretized set
% of PDE's of the form :
%
% epsilon(k) * df_dt(k) = b(k) -
%       \sum_{j} {A(k,j)*x_state(j)}.
%
% The procedure is passed an initial guess of
% the state vector x_state, a structure of the
% system parameters, and the names of the
% subroutines that calculate from this information
% the A matrix, the b vector, and the Jacobian
% of the b vector.
%
% First, if Solver.max_iter_time is not 0,
% the solver starts off with a stage of implicit
% Euler time integration that robustly approaches
% the vicinity of a stable steady state.  If the
% norm of the function (b-Ax) vector drops below
% a magnitude Solver.atol_time, then the
% time integration stage ends.
%
% If the time integration procedure has reduced
% the function vector to an acceptable magnitude,
% or if Solver.max_iter_time is 0 signifies
% that Newton's method is to be performed directly,
% then the Newton's method stage begins.  If the
% function vector norm drops below a magnitude of
% Solver.atol_Newton, then convergence to steady
% state is deemed to have occurred, and the routine
% exists with iflag_converge = 1.  Otherwise, the
% Newton's method stage ends after a maximum of
% Solver.max_iter_Newton number of iterations
% with iflag_converge remaining zero or
% negative (if an error occurred).
%
% If Solver.iflag_Adepend is non-zero, then
% the value of the A matrix is state dependent and
% must be recalculated at each iteration.  If
% Solver.iflag_nonneg is non-zero, then the
```

```
% state variables are enforced to be non-negative
% at every iteration.
%
% INPUT :
% =======
% x_guess             REAL(num_DOF)
%                   This is the initial guess of the state
%                   vector.
% Solver              This data structure contains the parameters
%                   that control the operation of the solver.
%   The fields of this structure are :
%   .max_iter_time      INT
%                     the maximum number of implicit Euler time steps.
%                   If =0, then no time simulation is performed and the
%                     solver goes immediately to Newton's method
%   .dt               REAL
%                     the time step to be used in the implicit
%                   Euler simulation
%   .atol_time          REAL
%                     the norm of the function (time derivative) vector
%                   at which the time integration procedure is deemed
%                   to have been sufficiently converged
%   .max_iter_Newton    INT
%                     the maximum number of Newton's method iterations
%   .atol_Newton        REAL
%                     the norm of the function (time derivative) vector
%                     at which convergence to the steady state solution is
%                     deemed to have been achieved
%   .iflag_Adepend      INT
%                     if this integer flag is non-zero, then the A matrix
%                     is assumed to be state-dependent and so must be
%                     recalculated at every iteration
%   .iflag_nonneg       INT
%                     if this integer flag is non-zero, then the elements
%                     of the state vector are enforced to be non-negative
%                     at every iteration
%   .iflag_verbose      INT
%                     if this integer flag is non-zero, then the solver
%                   routine is instructed to print to the screen the
%                   progress of the solution process; otherwise, it
%                   runs silent
% func_calc_A_int   FUNCTION NAME
%                     This is the name of the function that takes the
%                     master state vector and the system parameters
%                     and returns the A matrix that discretizes the
%                     transport terms.
% func_calc_b_int           FUNCTION NAME
%                     This is the name of the function that takes
%                     the master state vector and the system
%                     parameters and returns the b vector that
%                     typically arises from the source term in a PDE.
```

```
%                 It also returns the Jacobian of the b
%                    vector, whose (m,n) element is the
%                    partial of b(m) with respect to x_state(n).
% func_implement_BC          FUNCTION NAME
%                    This is the name of the function that returns
%                    the values of A, b, and bJac for the boundary
%                    conditions, which are found where epsilon
%                    has values of zero.
% epsilon             INT(num_DOF = length(x_guess))
%                    this is a 1-D array of integers of the same size
%                    as x_state.  It contains a 1 at position k for
%                    every equation that is an ordinary differential
%                    equation, and a 0 for every algebraic equation,
%                    which in this problem arise from the boundary
%                    conditions
% Param
%                 This data structure contains the system parameters
%                 that are passed to the functions for calculating
%                 the DAE system elements.
%
% OUTPUT :
% ========
% x_state             REAL(num_DOF)
%                 This is the output estimate of the steady state
%                 solution of the DAE system.
% iflag_converge    INT
%                    This integer flag is set equal to 1 if the
%                    solution procedure has converged.  A value
%                    of 0 means that the method did not converge.
%                    A negative value indicates an error.
% f                 REAL(num_DOF)
%                 This is the time derivative vector (for boundary
%                 points it is a measure of error in the boundary
%                 condition) whose magnitude tells how far the
%                 output estimate is from the steady state.
% f_init             REAL(num_DOF)
%                 The time derivative vector for State_init
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/23/2001
```

```
function [x_state,iflag_converge,f,f_init] = ...
   DAE_SS_solver_1(x_guess,Solver,...
   func_calc_A_int,func_calc_b_int, ...
   func_implement_BC,epsilon,Param);
```

```
iflag_converge = 0;

func_name = 'DAE_SS_solver';


% This integer flag controls what action to take
% in case of an assertion failure.  See the
% assertion routines for further details.
i_error = 2;


% check input data

% signify not yet completed checking of
% initial data
iflag_converge = -1;

% check data structure that contains
% solver parameters.  We do this with a
% routine that performs the appropriate
% assertions on a structure.

SolverType.num_fields = 8;

% .max_iter_time
ifield = 1;
FieldType.name = 'max_iter_time';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 1;
SolverType.field(ifield) = FieldType;
% .dt
ifield = 2;
FieldType.name = 'dt';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 1;
FieldType.check_int = 0;
SolverType.field(ifield) = FieldType;
% .atol_time
ifield = 3;
FieldType.name = 'atol_time';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
```

```
FieldType.check_real = 1;
FieldType.check_sign = 1;
FieldType.check_int = 0;
SolverType.field(ifield) = FieldType;
% .max_iter_Newton
ifield = 4;
FieldType.name = 'max_iter_Newton';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 1;
SolverType.field(ifield) = FieldType;
% .atol_Newton
ifield = 5;
FieldType.name = 'atol_Newton';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 1;
FieldType.check_int = 0;
SolverType.field(ifield) = FieldType;
% .iflag_Adepend
ifield = 6;
FieldType.name = 'iflag_Adepend';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 1;
SolverType.field(ifield) = FieldType;
% .iflag_nonneg
ifield = 7;
FieldType.name = 'iflag_nonneg';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 1;
SolverType.field(ifield) = FieldType;
% .iflag_verbose
ifield = 8;
FieldType.name = 'iflag_verbose';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
```

```
FieldType.check_sign = 2;
FieldType.check_int = 1;
SolverType.field(ifield) = FieldType;
% perform assertion on Solver structure
assert_structure(i_error,Solver,'Solver', ...
   func_name,SolverType);

if(Solver.iflag_verbose ~= 0)
   disp(' ');
   disp(' ');
   disp('Starting DAE_SS_solver_1() ...');
end


% check that x_guess is vector of proper type
% and set number of degrees of freedom to
% the length of x_guess.

if(Solver.iflag_nonneg ~= 0)
   check_sign = 1;
else
   check_sign = 0;
end
dim=0; check_column=1;
check_real=1; check_sign=0; check_int=0;
assert_vector(i_error,x_guess,'x_guess', ...
   func_name,dim,check_real, ...
   check_sign,check_int,check_column);
num_DOF = length(x_guess);

% check that func_calc_A_int is a
% character string
if(~ischar(func_calc_A_int))
   error([func_name, ': ', ...
   'func_calc_A_int is not a character string']);
end

% check that func_calc_b_int is a character string
if(~ischar(func_calc_b_int))
   error([func_name, ': ', ...
   'func_calc_b_int is not a character string']);
end

% check that func_implement_BC is a character string
if(~ischar(func_implement_BC))
   error([func_name, ': ', ...
   'func_implement_BC is not a character string']);
end

% check that epsilon is a vector of the proper type
dim = num_DOF; check_real = 1;
```

```
check_sign = 2; check_int = 1;
assert_vector(i_error,epsilon,'epsilon',...
   func_name,dim,check_real,check_sign, ...
   check_int,check_column);


% signify that finished checking input data

iflag_converge = 0;


% Initialize x_state to x_guess

x_state = x_guess;


%PDL> Calculate b, bJac using function
%   func_calc_b_int

[b_int,bJac_int,iflag_func] = feval(...
   func_calc_b_int,x_state,epsilon,Param);
if(iflag_func <= 0)
   message = [ func_name, ': ', ...
         'error (', int2str(iflag_func), ') ',
         'in feval calling of func_calc_b_int'];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end

% check that routine returns a proper b vector
% and bJac matrix

dim = num_DOF; check_real = 1; check_sign = 0;
check_int = 0; check_column = 1;
assert_vector(i_error,b_int,'b_int',...
   func_name,dim,check_real,check_sign, ...
   check_int,check_column);

num_rows = num_DOF; num_columns = num_DOF;
check_real = 1; check_sign = 0; check_int = 0;
assert_matrix(i_error,bJac_int,'bJac_int',...
   func_name,num_rows,num_columns, ...
   check_real,check_sign,check_int);
```

```
%PDL> Calculate the matrix A using the
%   function func_calc_A_int

[A_int,iflag_func] = feval( ...
   func_calc_A_int,x_state,epsilon,Param);
if(iflag_func <= 0)
   message = [ func_name, ': ', ...
         'error (', int2str(iflag_func), ') ',
         'in feval calling of func_calc_A_int'];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end
num_rows = num_DOF; num_columns = num_DOF;
check_real = 1; check_sign = 0; check_int = 0;
assert_matrix(i_error,A_int,'A_int',func_name,...
   num_rows,num_columns, ...
   check_real,check_sign,check_int);


%PDL> Call function func_implement_BC to set the
%       elements of A and b that implement the
%       boundary conditions

[A_BC,b_BC,bJac_BC,iflag_func] = feval(...
   func_implement_BC,x_state,epsilon,Param);
if(iflag_func <= 0)
   message = [ func_name, ': ', ...
         'error (', int2str(iflag_func), ') ',
         'in feval calling of func_implement_BC'];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end
% check A_BC
num_rows = num_DOF; num_columns = num_DOF;
check_real = 1; check_sign = 0; check_int = 0;
assert_matrix(i_error,A_BC,'A_BC',func_name, ...
   num_rows,num_columns, ...
   check_real,check_sign,check_int);
% check b_BC
```

```
dim = num_DOF; check_column = 1;
check_real = 1; check_sign = 0; check_int = 0;
assert_vector(i_error,b_BC,'b_BC',func_name, ...
   dim,check_real,check_sign, ...
   check_int,check_column);
% check_bJac_BC
num_rows = num_DOF; num_columns = num_DOF;
check_real = 1; check_sign = 0; check_int = 0;
assert_matrix(i_error,bJac_BC,'bJac_BC',func_name, ...
   num_rows,num_columns, ...
   check_real,check_sign,check_int);


%PDL> Calculate function Jacobian, Jac = bJac - A

Jac = (bJac_int + bJac_BC) - (A_int + A_BC);


% calculate function (time derivative) vector

f = (b_int + b_BC) - (A_int + A_BC) * x_state;


% store the initial function (time derivative)
% vector for return

f_init = f;



%PDL> If Solver.max_iter_time IS NOT 0 THEN
%       perform time simulation

iflag_converge_time = 0;
if(Solver.max_iter_time ~= 0)

  if(Solver.iflag_verbose ~= 0)
     disp(' ');
     disp(['    ', ...
       'Starting implicit Euler time integration ...']);
  end


%        PDL> Iterate over every time step starting at
%       time = 0
%       FOR iter FROM 1 TO Solver.max_iter_time

  time = 0;

  for iter=1:Solver.max_iter_time
```

```
%  PDL> Modify every row of Jac corresponding to
%  an algebraic equation to the correct form required
%  to enforce the corresponding algebraic equation
%  after the delta_x update is made.  This is a
%  special trick that works for the linear
%  algebraic equations found in the DAE systems
%  that commonly result from a PDE

    % LHS matrix for updating equation
    % with implicit Euler
    D = speye(num_DOF) - Solver.dt * Jac;

    % RHS vector for updating equation
    % with implicit Euler
    g = Solver.dt * f;

    % identify rows corresponding
    % to boundary points
    list_bound = find(epsilon == 0);

    % for every boundary condition equation,
    % replace the implicit Euler updating equation
    % elements with the values required to yield
    % a delta_x that satisfies the boundary
    % conditions at the new time step (with current A).
    for count=1:length(list_bound)
       iDOF = list_bound(count);
       D(iDOF,:) = A_BC(iDOF,:) - bJac_BC(iDOF,:);
       g(iDOF) = b_BC(iDOF) - ...
          dot(A_BC(iDOF,:),x_state);
    end


%  PDL> Calculate delta_x, the change in x_state
%  for this time step using the implicit Euler method.

    delta_x = D\g;


%  PDL> Update the value of x_state

    x_state = x_state + delta_x;


%  PDL> IF Solver.iflag_noneg IS NOT 0 signifying
%  that the elements of the state vector should be
%  non-negative THEN find and make any negative
%  values zero

    if(Solver.iflag_nonneg ~= 0)
```

```
      list_neg = find(x_state < 0);
      for count=1:length(list_neg)
         iDOF = list_neg(count);
         x_state(iDOF) = 0;
      end
   end
```

```
%PDL> Calculate new b, bJac using function
%   func_calc_b_int
```

```
   [b_int,bJac_int,iflag_func] = ...
      feval(func_calc_b_int,x_state, ...
      epsilon,Param);
   if(iflag_func <= 0)
      message = [func_name, ': ', ...
            'error (', int2str(iflag_func), ') ', ...
            'in feval calling of func_calc_b_int'];
      if(i_error > 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

   % check that routine returns a proper b vector
   % and bJac matrix
   % b_int
   dim = num_DOF; check_column = 1;
   check_real = 1; check_sign = 0; check_int = 0;
   assert_vector(i_error,b_int,'b_int',func_name,...
      dim,check_real,check_sign, ...
      check_int,check_column);
   % bJac_int
   num_rows = num_DOF; num_columns = num_DOF;
   check_real = 1; check_sign = 0; check_int = 0;
   assert_matrix(i_error,bJac_int,'bJac_int',...
      func_name,num_rows,num_columns, ...
      check_real,check_sign,check_int);
```

```
%PDL> IF Solver.iflag_Adepend is non-zero signifying
%          a state-dependence of the A matrix, calculate a new
%          value of A using the function func_calc_A_int
```

```
   if(Solver.iflag_Adepend ~= 0)
      [A_int,iflag_func] = feval(...
         func_calc_A_int,x_state,epsilon,Param);
```

```
      if(iflag_func <= 0)
         message = [func_name, ': ', ...
            'error (', int2str(iflag_func), ') ', ...
            'in feval calling of func_calc_A_int'];
         if(i_error ~= 0)
            if(i_error > 1)
               save dump_error.mat;
            end
            error(message);
         else
            return;
         end
      end
      % check A_int
      num_rows = num_DOF; num_columns = num_DOF;
      check_real = 1; check_sign = 0; check_int = 0;
      assert_matrix(i_error,A_int,'A_int', ...
         func_name,num_rows,num_columns, ...
         check_real,check_sign,check_int);
   end


%PDL> Call function func_implement_BC to set the
%   forms of A and b for implementing the
%   boundary conditions

   [A_BC,b_BC,bJac_BC,iflag_func] = feval(...
      func_implement_BC, ...
      x_state,epsilon,Param);
   if(iflag_func <= 0)
      message = [func_name, ': ', ...
         'error (', int2str(iflag_func), ') ', ...
         'in feval calling of func_implement_BC'];
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end
   % A_BC
   num_rows = num_DOF; num_columns = num_DOF;
   check_real = 1; check_sign = 0; check_int = 0;
   assert_matrix(i_error,A_BC,'A_BC', ...
         func_name,num_rows,num_columns, ...
         check_real,check_sign,check_int);
  % b_BC
  dim = num_DOF; check_column = 1;
  check_real = 1; check_sign = 0; check_int = 0;
```

```
assert_vector(i_error,b_BC,'b_BC',func_name,...
   dim,check_real,check_sign, ...
   check_int,check_column);
 % bJac_BC
num_rows = num_DOF; num_columns = num_DOF;
check_real = 1; check_sign = 0; check_int = 0;
assert_matrix(i_error,bJac_BC,'bJac_BC',...
   func_name,num_rows,num_columns, ...
   check_real,check_sign,check_int);
```

%PDL> Calculate new function vector, f = b - A*x_state

```
f = (b_int+b_BC) - (A_int+A_BC)*x_state;
```

%PDL> Calculate function Jacobian, Jac = bJac - A

```
Jac = (bJac_int+bJac_BC) - (A_int+A_BC);
```

%PDL> Update time value, time = time + Solver.dt

```
time = time + Solver.dt;
```

%PDL> Calculate norm of f, as norm_f = max(abs(f))

```
f_norm = max(abs(f));

if(Solver.iflag_verbose ~= 0)
   disp([ '    ',int2str(iter), '     ',  ...
          num2str(time), '     ', ...
          num2str(f_norm)]);
end
```

%PDL> If norm_f is LESS THAN Solver.atol_time THEN
% sufficient convergence of the time integration
% has been achieved. Set a local integer flag to
% signify this and EXIT the time iteration FOR loop

```
if(f_norm < Solver.atol_time)
   iflag_converge_time = 1;
   if(Solver.iflag_verbose ~= 0)
      disp(['   ', ...
       'Time simulation convergence attained']);
   end
   break;
end
```

```
%   PDL> ENDFOR for time iterations

  end


if(Solver.iflag_verbose ~= 0)
  if(iflag_converge_time ~= 1)
     disp(['    ', ...
          'Time simulation did not converge']);
  end
end


%PDL> ENDIF if(Solver.max_iter_time ~= 0)

end




%PDL> IF Solver.max_iter_time IS 0 signifying no time
%   simulation was performed OR IF a local integer flag
%   shows convergence of the time integration THEN
%   begin Newton's method iterations

if(or((Solver.max_iter_time==0),(iflag_converge_time==1)))

  if(Solver.iflag_verbose ~= 0)
     disp( ' ');
     disp('Starting Newton method iterations ...');
  end


%   PDL> Iterate over the Newton's method steps
%      FOR iter FROM 1 TO Solver.max_iter_Newton

  for iter=1:Solver.max_iter_Newton


%      PDL> Calculate the full Newton's method
%         update delta_x

      D = Jac;
      g = -f;
      delta_x = D\g;


%               PDL> Perform weak line search to find
%         update that satisfies descent criterion
%         using the 2-norm
```

```
      f_2norm_old = dot(f,f);

      lambda_max = 1;
      max_line_search = 50;
      for iweak_LS = 0:max_line_search
         lambda = lambda_max*(2^(-iweak_LS));

         y = x_state + lambda*delta_x;


%        PDL> IF Solver.iflag_noneg IS NOT 0
%                  signifying that the elements of the
%                  state vector should be non-negative
%              THEN find and make any negative
%                  values zero

         if(Solver.iflag_nonneg ~= 0)
            list_neg = find(y < 0);
            for count=1:length(list_neg)
               iDOF = list_neg(count);
               y(iDOF) = 0;
            end
         end


%           PDL> Calculate new b, bJac using
%             function func_calc_b_int

         [b_int,bJac_int,iflag_func] = feval( ...
            func_calc_b_int,y, ...
            epsilon,Param);
         if(iflag_func <= 0)
            message = [func_name, ': ', ...
                  'error (', int2str(iflag_func), ') ', ...
                  'in feval calling of func_calc_b_int'];
            if(i_error ~= 0)
               if(i_error > 1)
                  save dump_error.mat;
               end
               error(message);
            else
               return;
            end
         end
         % check that routine returns a proper b vector
         % and bJac matrix
         % b_int
         dim = num_DOF; check_column = 1;
         check_real = 1; check_sign = 0; check_int = 0;
         assert_vector(i_error,b_int,'b_int',func_name,...
```

```
      dim,check_real,check_sign, ...
      check_int,check_column);
   % bJac_int
   num_rows = num_DOF; num_columns = num_DOF;
   check_real = 1; check_sign = 0; check_int = 0;
   assert_matrix(i_error,bJac_int,'bJac_int',...
      func_name,num_rows,num_columns, ...
      check_real,check_sign,check_int);


%     PDL> IF Solver.iflag_Adepend is non-zero
%       signifying a state-dependence of the A matrix,
%       calculate a new value of A using the function
%       func_calc_A_int

   if(Solver.iflag_Adepend ~= 0)
      [A_int,iflag_func] = feval( ...
         func_calc_A_int,y,epsilon,Param);
      if(iflag_func <= 0)
         message = [func_name, ': ', ...
            'error (', int2str(iflag_func), ') ', ...
            'in feval calling of func_calc_A_int'];
         if(i_error ~= 0)
            if(i_error > 1)
               save dump_error.mat;
            end
            error(message);
         else
            return;
         end
      end
      % check A_int
      num_rows = num_DOF; num_columns = num_DOF;
      check_real = 1; check_sign = 0; check_int = 0;
      assert_matrix(i_error,A_int,'A_int', ...
         func_name,num_rows,num_columns, ...
         check_real,check_sign,check_int);
   end


%     PDL> Call func_implement_BC to obtain the values
%       of A and b, bJac that are used to enforce the
%       boundary conditions

   [A_BC,b_BC,bJac_BC,iflag_func] = feval( ...
      func_implement_BC, ...
      y,epsilon,Param);
   if(iflag_func <= 0)
      message = [func_name, ': ', ...
         'error (', int2str(iflag_func), ') ', ...
         'in feval calling of func_implement_BC'];
```

```
            if(i_error ~= 0)
               if(i_error > 1)
                  save dump_error.mat;
               end
               error(message);
            else
               return;
            end
         end
         % A_BC
         num_rows = num_DOF; num_columns = num_DOF;
         check_real = 1; check_sign = 0; check_int = 0;
         assert_matrix(i_error,A_BC,'A_BC', ...
               func_name,num_rows,num_columns, ...
               check_real,check_sign,check_int);
         % b_BC
         dim = num_DOF; check_column = 1;
         check_real = 1; check_sign = 0; check_int = 0;
         assert_vector(i_error,b_BC,'b_BC',func_name,...
            dim,check_real,check_sign, ...
            check_int,check_column);
         % bJac_BC
         num_rows = num_DOF; num_columns = num_DOF;
         check_real = 1; check_sign = 0; check_int = 0;
         assert_matrix(i_error,bJac_BC,'bJac_BC',...
            func_name,num_rows,num_columns, ...
            check_real,check_sign,check_int);


%           PDL> Calculate new function vector,
%              f = b - A*x_state

         f = (b_int+b_BC) - (A_int+A_BC)*y;


%           PDL> Calculate norm of f, as
%                 f_norm = max(abs(f))

         f_norm = max(abs(f));


         % check for descent criterion

         f_2norm = dot(f,f);

         if(f_2norm < f_2norm_old)
            break;
         end

      end      % iweak_LS for loop
```

```
%       PDL> Update x_state with results of
%            weak line search delta_x

      x_state = y;
      if(Solver.iflag_verbose ~= 0)
         disp([ '    ', int2str(iter), '     ', ...
               num2str(f_norm), '     ', ...
               num2str(f_2norm)]);
      end


%     PDL> Calculate Jacobian, Jac = bJac - A

      Jac = (bJac_int+bJac_BC) - (A_int+A_BC);


%     PDL> If norm_f is LESS THAN Solver.atol_Newton
%        THEN we have achieved satisfactory convergence
%        to steady state.  Set iflag_converge = 1 and
%        exit the FOR loop running the Newton method
%        iterations

      if(f_norm < Solver.atol_Newton)
         iflag_converge = 1;
         if(Solver.iflag_verbose ~= 0)
            disp('Newton Method Convergence achieved');
            break;
         end
      end


%PDL> ENDFOR - end Newton's method iterations

   end

% ENDIF for performing Newton's iterations
end


if(Solver.iflag_verbose ~= 0)
   disp(' ');
   if(iflag_converge < 0)
      disp('DAE_SS_solver_1() had error');
   elseif (iflag_converge ~= 1)
      disp('DAE_SS_solver_1() unconverged');
   end
   disp('Exiting DAE_SS_solver_1()');
end
return;
```