TR_1D_model1_SS\TR_1D_modlel1_SS_solver.m

```
% TR_1D_model1_SS\TR_1D_model1_SS_solver.m
%
% function [State,iflag_converge,f,f_init] = ...
%    TR_1D_model1_SS_solver(State_init,Solver,ProbDim, ...
%    Reactor,Physical,Rxn,Grid);
%
% This procedure updates the solution estimate
% of the steady state concentration and
% temperature profiles in a 1D tubular
% reactor with an arbitrary reaction network.
% If desired, an initial stage of implicit Euler
% time integration is performed for either a
% maximum number of steps or until the norm of
% the function (time derivative) vector drops
% below a certain magnitude.  At this point, a
% Newton's method with a weak line search is
% performed until convergence to steady state
% is achieved.  The new estimates of the steady
% state concentration and temperature profiles
% are then returned along with an integer flag
% signaling 1 if convergence has been achieved.
%
% The procedure first provides routines to convert
% the problem to the generic form :
%
% epsilon(k) * df_dt(k) = b(k) -
%          \sum_{j} {A(k,j)*x_state(j)}
%
% Here x_state is a master 1-D array of the
% unknown state variables and packing and unpacking
% routines will be provided to convert between
% state_data and x_state.  For equation (row) k,
% epsilon(k) is 1 if the equation is an ODE and is
% 0 if it is an algebraic equation arising from the
% boundary conditions.  b(k) is a non-linear source
% term and A is a matrix that discretizes the
% diffusive and convective transport terms.  Names
% of functions will be set that calculate A,
% calculate the source term vector b, and
% the Jacobian of b given the input state data in
% the master array form x.  Three functions are used -
% the first calculates the A elements for the interior
% points, the second calculates the b and bJac elements
% for the interior points, and the third calculates
% the A, b, and bJac elements for the boundary conditions
% that are implemented as algebraic equations.
%
```

```
% The names of these functions are then passed to
% a generic solver routine that does the actual
% calculation and that may be reused for other problems.
% A flag is passed to this solver routine that enforces
% that all state variables be non-negative at
% every time and Newton's method iteration.
%
% INPUT :
% =======
% State_init          copy of State data structure containing
%                     the initial values of the concentration
%                     and temperature profiles
% Solver              see TR_1D_model1_SS.m for description
% ProbDim             see TR_1D_model1_SS.m for description
% Reactor             see TR_1D_model1_SS.m for description
% Physical            see TR_1D_model1_SS.m for description
% Rxn                 see TR_1D_model1_SS.m for description
% Grid                see TR_1D_model1_SS.m for description
%
% OUTPUT :
% ========
% State               data structure (see TR_1D_model1_ss.m for format)
%                     that contains the output estimate of the steady
%                     state solution obtained by the solver
% 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 7/25/2001
```

```
function [State,iflag_converge,f,f_init] = ...
   TR_1D_model1_SS_solver(State_init,Solver,ProbDim, ...
   Reactor,Physical,Rxn,Grid);
```

```
%PDL> Initialize iflag_converge to 0 to
%   designate lack of convergence

iflag_converge = 0;

func_name = 'TR_1D_model1_SS_solver';


% This integer flag controls what action to take
% in case of an error.  A value > 1 results in
% a dump_error.mat file being written before the
% MATLAB error() function is invoked.
i_error = 2;

% Since this routine is so closely coupled with the
% main program, no checks on input are added.

if(Solver.iflag_verbose ~= 0)
   disp(' ');
   disp(' ');
   disp('Starting TR_1D_model1_SS_solver()...');
end


%PDL> Set names of routines to be used for calculating
%   the A matrix, b vector, and the Jacobian of b in
%   the standard DAE form.  For the interior points,
%   use a function to calculate A and another to
%   calculate b and bJac.  Use a separate function
%   to implement the boundary conditions.

func_calc_A_int = 'TR_1D_model1_func_calc_A_int';
func_calc_b_int = 'TR_1D_model1_func_calc_b_int';
func_implement_BC = 'implement_Dankwert_BC';


%PROCEDURE: stack_state
%PDL> Stack the state variables into the master
%   1-D array x_state
%ENDPROCEDURE

[x_init,iflag_func] = stack_state(State_init, ...
   ProbDim.num_species,Grid.num_pts);
if(iflag_func <= 0)
   if(i_error > 1)
      save dump_error.mat;
   end
   message = [func_name, ': ', ...
         'Error (',int2str(iflag_func),')', ...
         ' returned from stack_state'];
   error(message);
```

```
end


%PROCEDURE: calc_epsilon
%PDL> Set the epsilon vector telling the solver
%   which are the ordinary differential equations
%   and which are the algebraic equations
%ENDPROCEDURE

% set an integer mask that has 0's at the
% boundary points and 1's at the interior points.
imask_int = linspace(1,1,Grid.num_pts)';
imask_int(1) = 0;
imask_int(Grid.num_pts)= 0;

% set total number of fields
num_fields = ProbDim.num_species + 1;

% calculate the epsilon vector for the DAE system
% that has 1's for every ODE and a 0 for every
% algebraic equation
[epsilon,iflag_func] = calc_epsilon( ...
   Grid.num_pts,imask_int,num_fields);
if(iflag_func <= 0)
   iflag_converge = -1;
   if(i_error > 1)
      save dump_error.mat;
   end
   message = [func_name, ': ', ...
      'error (',int2str(iflag_func),')', ...
      ' returned from calc_epsilon'];
   error(message);
end


%PDL> Provide a master structure to stack the
%   data to be passed to the general solver
%   routine and from there to the functions
%   that calculate A and b, bJac.

Param.ProbDim = ProbDim;
Param.Reactor = Reactor;
Param.Physical = Physical;
Param.Rxn = Rxn;
Param.Grid = Grid;



%PROCEDURE: DAE_SS_solver_1
%PDL> Pass x_state, the system parameters
%   and the solver_data parameters to a generic
```

```
%   solver routine to update the steady state
%   solution estimate.  This routine returns
%   the new solution estimate in x_state, the
%   appropriate value to iflag_converge, and
%   the final value of the function (b-Ax)
%   vector.
%ENDPROCEDURE

[x_state,iflag_func,f,f_init] = ...
   DAE_SS_solver_1(x_init,...
   Solver,func_calc_A_int,func_calc_b_int, ...
   func_implement_BC,epsilon,Param);

iflag_converge = iflag_func;

if(iflag_converge < 0)
   message = [func_name, ': ', ...
         'Error (',int2str(iflag_func),')', ...
         ' returned from DAE_SS_solver_1'];
   if(i_error > 1)
      save dump_error.mat;
   end
   error(message);
end


%PROCEDURE: unstack_state
%PDL> Unstack the new solution estimate
%   to the state_data variable names
%ENDPROCEDURE

[State,iflag_func] = unstack_state(x_state, ...
   ProbDim.num_species,Grid.num_pts);
if(iflag_func <= 0)
   message = [func_name, ': ', ...
         'Error (',int2str(iflag_func),')', ...
         ' returned from unstack_state'];
   if(i_error > 1)
      save dump_error.mat;
   end
   error(message);
end


%PDL> Return the new state_data variable
%   values, the new value of iflag_converge,
%   and the function vector for
%   the new estimate

return;
```