TR_1D_model1_SS\implement_Dankwert_BC.m

```
% TR_1D_model1_SS\implement_Dankwert_BC.m
%
% function [A_BC,b_BC,bJac_BC,iflag] = ...
%    implement_Dankwert_BC(x_state,epsilon,Param);
%
% This procedure calculates the components in the DAE form :
%
%   epsilon(k) * df_dt(k) = b(k) -
%          sum_{j} {A(k,j)*x_state(j)}
%
% that discretize the boundary conditions on the PDE's.
% These are governed by the locations where epsilon(k)
% is zero.  This procedure implements Dankwert's boundary
% conditions at the inlet and outlet of a 1-D tubular reactor.
% Second order accurate formulas are used to approximate the
% derivatives at the boundaries.
%
% INPUT :
% =======
% x_state              REAL(num_DOF)
%                      This is the vector of the state variables.
% epsilon             REAL(num_DOF)
%                      This is a vector with 1's at the interior
%                      points and 0 at the boundary points.
% Param                This is a data structure containing the
%                      system parameters.  This routine uses
%                      the fields .ProbDim, .Grid, .Reactor,
%                      .Physical
%
% OUTPUT :
% ========
% A_BC                 REAL(num_DOF,num_DOF)
%                      This sparse matrix discretizes is used in
%                      the linear formulation of the boundary
%                      conditions.
% b_BC                 REAL(num_DOF)
%                      This RHS vector is used in the linear
%                      formulation of the boundary conditions.
% bJac_BC              REAL(num_DOF,num_DOF)
%                      The Jacobian of b_BC.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
```

```
% Version as of 7/24/2001


function [A_BC,b_BC,bJac_BC,iflag] = ...
   implement_Dankwert_BC(x_state,epsilon,Param);


iflag =0;

func_name = 'implement_Dankwert_BC';


% This integer flag controls what action to take
% in the case of an assertion or called routine
% error.
i_error = 2;


% check input

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

% set number of state variables
num_DOF = length(x_state);

% epsilon
dim=num_DOF; check_column=1;
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);

% Extract system parameters from Param structure
if(~isstruct(Param))
   iflag = 1;
   message = [func_name, ': ', ...
         'Param is not a structure'];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      A_BC=0; b_BC=0; bJac_BC=0;
      return;
   end
```

```
end
% ProbDim
field_name = 'ProbDim';
if(~isfield(Param,field_name))
   iflag = -1;
   message = [func_name, ': ', ...
         'Param does not contain ',field_name];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      A_BC=0; b_BC=0; bJac_BC=0;
      return;
   end
end
ProbDim = Param.ProbDim;
% Grid
field_name = 'Grid';
if(~isfield(Param,field_name))
   iflag = -1;
   message = [func_name, ': ', ...
         'Param does not contain ',field_name];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      A_BC=0; b_BC=0; bJac_BC=0;
      return;
   end
end
Grid = Param.Grid;
% Reactor
field_name = 'Reactor';
if(~isfield(Param,field_name))
   iflag = -1;
   message = [func_name, ': ', ...
         'Param does not contain ',field_name];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      A_BC=0; b_BC=0; bJac_BC=0;
      return;
   end
end
```

```
Reactor = Param.Reactor;
% Physical
field_name = 'Physical';
if(~isfield(Param,field_name))
   iflag = -1;
   message = [func_name, ': ', ...
         'Param does not contain ',field_name];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      A_BC=0; b_BC=0; bJac_BC=0;
      return;
   end
end
Physical = Param.Physical;



% make an integer mask of the interior points from
% the first field values of epsilon

imask_int = epsilon(1:Grid.num_pts);


% From this, make a list of the boudary points

list_bound = find(imask_int == 0);
num_bound = length(list_bound);


%PDL> Initialize A, b, and bJac to zeros.  For the
%   linear algebraic equations obtained from the
%   Dankwert's BC's, the return value of bJac is zero

max_nonzero = num_bound*(ProbDim.num_species+1)*3;
A_BC = spalloc(num_DOF,num_DOF,max_nonzero);

b_BC = linspace(0,0,num_DOF)';

bJac_BC = spalloc(num_DOF,num_DOF,1);


%PDL> First, we set the linear equations at the inlet
%   using Dankwert's condition that is essentially a
%   flux balance around the inlet

%PROCEDURE: discretize_boundary_deriv
%PDL> Calculate the coefficients that discretize the first
```

```
%   derivative operator at z1 from the values of the field
%   at z1, z2, and z3.  The coefficients are for the form :
%   d\phi_dz = a1 * \phi(z1) + a2*\phi(z2) + a3*\phi(z3)
%ENDPROCEDURE

[a1,a2,a3,iflag_func] = discretize_boundary_deriv(...
   Grid.z(1),Grid.z(2),Grid.z(3));
if(iflag_func <= 0)
   iflag_func = -2;
   message = [func_name, ': ', ...
         'Error (',int2str(iflag_func), ') ', ...
         'returned from discretize_boundary_deriv'];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end


%PDL> Set the inlet BC on the species balances

%PDL> For every species balance
%   FOR ispecies FROM 1 TO ProbDim.num_species

for ispecies = 1:ProbDim.num_species


%PDL> Set pos_offset = (ispecies-1)*Grid.num_pts
   % set position offset to add to point number
   % to obtain the chosen concentration in the
   % master state vector.

   pos_offset = (ispecies-1)*Grid.num_pts;


%   PDL> A(pos_offset+1,pos_offset+1) =
%   1 - Physical.diffusivity(ispecies)/Reactor.velocity*a1

   A_BC(pos_offset+1,pos_offset+1) = ...
      1 - Physical.diffusivity(ispecies)/Reactor.velocity*a1;


%   PDL> A(pos_offset+1,pos_offset+2) =
%            -Physical.diffusivity(ispecies)/Reactor.velocity*a2

   A_BC(pos_offset+1,pos_offset+2) = ...
      -Physical.diffusivity(ispecies)/Reactor.velocity*a2;
```

```
%   PDL> A(pos_offset+1,pos_offset+3) =
%        -Physical.diffusivity(ispecies)/Reactor.velocity*a3

   A_BC(pos_offset+1,pos_offset+3) = ...
      -Physical.diffusivity(ispecies)/Reactor.velocity*a3;


%   PDL> b(pos_offset+1) = Reactor.conc_in(ispecies)

   b_BC(pos_offset+1) = Reactor.conc_in(ispecies);


%PDL> ENDFOR

end


%PDL> For the enthalpy balance, set the inlet boundary condition

%   PDL> Set pos_offset = ProbDim.num_species*Grid.num_pts

   pos_offset = ProbDim.num_species*Grid.num_pts;


%        PDL> A(pos_offset+1,pos_offset+1) =
%                  1 - Physical.thermal_diff/Reactor.velocity*a1

   A_BC(pos_offset+1,pos_offset+1) = ...
      1 - Physical.thermal_diff/Reactor.velocity*a1;


%        PDL> A(pos_offset+1,pos_offset+2) =
%                  -Physical.thermal_diff/Reactor.velocity*a2

   A_BC(pos_offset+1,pos_offset+2) = ...
      -Physical.thermal_diff/Reactor.velocity*a2;


%        PDL> A(pos_offset+1,pos_offset+3) =
%                  -Physical.thermal_diff/Reactor.velocity*a3

   A_BC(pos_offset+1,pos_offset+3) = ...
      -Physical.thermal_diff/Reactor.velocity*a3;


%        PDL> b(pos_offset+1) = Reactor.Temp_in

   b_BC(pos_offset+1) = Reactor.Temp_in;
```

```
% PDL> Set the outlet BC using Dankwert's condition that
% the axial derivative is zero

%PROCEDURE: discretize_boundary_deriv
%PDL> Calculate the coefficients that discretize the first
%   derivative operator at z(num_pts) from the values of the
%   field at the last three points.  The coefficients are
%   for the form :
% d\phi_dz = b3 * \phi(z(num_pts-2)) + b2*\phi(z(num_pts-1)) +
%   b1*\phi(z(num_pts))
%ENDPROCEDURE

[b1,b2,b3,iflag_func] = discretize_boundary_deriv(...
   Grid.z(Grid.num_pts),Grid.z(Grid.num_pts-1),...
   Grid.z(Grid.num_pts-2));
if(iflag_func <= 0)
   iflag = -2;
   message = [func_name, ': ', ...
         'Error( ',int2str(iflag_func),') ', ...
         'returned from discretize_boundary_deriv'];
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end


%PDL> Set the outlet BC on each species balance
%        FOR ispecies FROM 1 TO ProbDim.num_species

for ispecies = 1:ProbDim.num_species


%        PDL> Set pos_offset = (ispecies-1)*Grid.num_pts

   pos_offset = (ispecies-1)*Grid.num_pts;


%        PDL> A(pos_offset+num_pts,pos_offset+num_pts-2) = b3

   A_BC(pos_offset + Grid.num_pts, ...
      pos_offset + Grid.num_pts - 2) = b3;


%        PDL> A(pos_offset+num_pts,pos_offset+num_pts-1) = b2
```

```
   A_BC(pos_offset + Grid.num_pts, ...
      pos_offset + Grid.num_pts-1)= b2;


%        PDL> A(pos_offset+num_pts,pos_offset+num_pts) = b1

   A_BC(pos_offset + Grid.num_pts, ...
      pos_offset + Grid.num_pts) = b1;


%        PDL> b(pos_offset+num_pts) = 0

   b_BC(pos_offset + Grid.num_pts) = 0;


%PDL> ENDFOR

end


%PDL> Set the outlet BC for the enthalpy balance

%        PDL> Set pos_offset =
%                 ProbDim.num_species*Grid.num_pts

   pos_offset = ProbDim.num_species*Grid.num_pts;


%        PDL> A(pos_offset+num_pts,pos_offset+num_pts-2) = b3

   A_BC(pos_offset + Grid.num_pts, ...
      pos_offset + Grid.num_pts - 2) = b3;


%        PDL> A(pos_offset+num_pts,pos_offset+num_pts-1) = b2

   A_BC(pos_offset + Grid.num_pts, ...
      pos_offset + Grid.num_pts - 1) = b2;


%        PDL> A(pos_offset+num_pts,pos_offset+num_pts) = b1

   A_BC(pos_offset + Grid.num_pts, ...
      pos_offset + Grid.num_pts) = b1;


%        PDL> b(pos_offset+num_pts) = 0

   b_BC(pos_offset + Grid.num_pts) = 0;
```

```
iflag = 1;

return;
```