TR_1D_model1_SS\FinDiff_1D_FirstDeriv.m

```
% TR_1D_model1_SS\FinDiff_1D_FirstDeriv.m
%
% function [FirstDerivMatrix,iflag] = ...
%    FinDiff_1D_FirstDeriv(Grid,imask,iuse_upwind,velocity);
%
% This subroutine uses the finite difference method
% to return a matrix that discretizes the first
% derivative operator on a 1-D grid using either
% upwind or central differences.  The procedure is
% written to be compatible with the use of a
% non-uniform grid.  Values are returned only for
% discretization at the grid points where the value
% in an integer mask vector is non-zero.  A uniform
% velocity value is used to determine which one-sided
% difference formula to use for the upwind direction.
%
% INPUT :
% =======
% Grid          This data structure contains the 1D grid
%               informaion :
%               .num_pts = the total number of points
%               .z = the z-coordinates of each grid point
% imask         INT(Grid.num_pts)
%               This integer mask contains a non-zero value
%               only at the interior points at which the
%               first derivative operator is to be
%               discretized.
% iuse_upwind   INT
%               If this integer flag is non-zero, then use
%               upwind finite difference formula.  Otherwise,
%               use central finite differences.
%
% OUTPUT :
% ========
% FirstDerivMatrix   REAL(num_pts,num_pts) SPARSE
%               This sparse matrix contains the discretized
%               form of the first derivative operator at
%               each of the grid points with non-zero
%               imask values
% velocity      REAL
%               This is the value of the velocity in the
%               system (assumed constant by continuity)
%               and it is used to determine the upwind
%               direction from its sign.
%
% Kenneth Beers
% Massachusetts Institute of Technology
```

```
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/23/2001

function [FirstDerivMatrix,iflag] = ...
   FinDiff_1D_FirstDeriv(Grid,imask,iuse_upwind,velocity);


iflag = 0;

func_name = 'FinDiff_1D_FirstDeriv';


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


% check the input

% Grid
GridType.num_fields = 2;
% .num_pts
ifield = 1;
FieldType.name = 'num_pts';
FieldType.is_numeric = 1;
FieldType.num_rows = 1;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 1;
FieldType.check_int = 1;
GridType.field(ifield) = FieldType;
% .z
ifield = 2;
FieldType.name = 'z';
FieldType.is_numeric = 1;
FieldType.num_rows = Grid.num_pts;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 0;
FieldType.check_int = 0;
GridType.field(ifield) = FieldType;
% perform assertion
assert_structure(i_error,Grid,'Grid',func_name,GridType);

% imask
dim=Grid.num_pts; check_column=0;
check_real=1; check_sign=2; check_int=1;
```

```
assert_vector(i_error,imask,'imask', ...
   func_name,dim,check_real,check_sign, ...
   check_int,check_column);

% iuse_upwind
check_real=1; check_sign=2; check_int=1;
assert_scalar(i_error,iuse_upwind,'iuse_upwind', ...
   func_name,check_real,check_sign,check_int);

% velocity
check_real=1; check_sign=0; check_int=0;
assert_scalar(i_error,velocity,'velocity', ...
   func_name,check_real,check_sign,check_int);


%PDL> Set an integer flag to determine whether to
%   use the central, forward, or rearward finite
%   difference formula.

i_FD_central = 0;
i_FD_forward = 1;
i_FD_backward = -1;

% if use central finite differences automatically
if(iuse_upwind == 0)
   i_FD_formula = i_FD_central;

% else if use upwind finite differences
else
   if(velocity > 0)
      i_FD_formula = i_FD_backward;
   elseif(velocity < 0)
      i_FD_formula = i_FD_forward;
   else
      i_FD_formula = i_FD_central;
   end
end


%PDL> Initialize FirstDerivMatrix to all zeros

% find all poitns at which derivative is to be
% discretized
list_points = find(imask ~= 0);
num_eval_points = length(list_points);

max_nonzero = num_eval_points*3;
FirstDerivMatrix = spalloc(...
   Grid.num_pts,Grid.num_pts,max_nonzero);
```

```
%PDL> FOR every grid point m that has a non-zero
% integer mask value

for count=1:num_eval_points
   ipoint = list_points(count);


   % find indices of left, center, and right points
   i_center = ipoint;
   i_left = ipoint-1;
   i_right = ipoint+1;


   % use case statement to check for using different
   % finite difference options

   switch i_FD_formula;


   %    PDL> IF using central differences THEN

   case {i_FD_central}

%        PDL> Check to ensure that the point is
%          not a boundary point

      if(or((ipoint == 1),(ipoint == Grid.num_pts)))
         iflag = -3;
         message = [func_name, ': ', ...
            'Central FD not allowed for end point'];
         if(i_error ~= 0)
            if(i_error > 1)
               save dump_error.mat;
            end
            error(message);
         else
            return;
         end
      end


%        PDL> Set denom_inv = 1 / (grid_z(m+1) - grid_z(m-1))
%   Set the inverse of the denominator.

      denom = Grid.z(i_right) - Grid.z(i_left);
      denom_inv = 1 / denom;


%        PDL> FirstDerivMatrix(m,m+1) = denom_inv

      FirstDerivMatrix(i_center,i_right) = denom_inv;
```

```
%       PDL> FirstDerivMatrix(m,m-1) = -denom_inv

    FirstDerivMatrix(i_center,i_left) = -denom_inv;


%       PDL> ELSEIF using forward differences THEN

  case {i_FD_forward}


%       PDL> Check to ensure that the point is not the last one

    if(ipoint == Grid.num_pts)
      iflag = -3;
      message = [func_name, ': ', ...
        'Forward FD not allowed for last point'];
      if(i_error ~= 0)
        if(i_error > 1)
          save dump_error.mat;
        end
        error(message);
      else
        return;
      end
    end


%       PDL> Set denom_inv = 1/(grid_z(m+1)-grid_z(m))

    denom = Grid.z(i_right) - Grid.z(i_center);
    denom_inv = 1 / denom;


%       PDL> FirstDerivMatrix(m,m+1) = denom_inv

    FirstDerivMatrix(i_center,i_right) = denom_inv;


%       PDL> FirstDerivMatrix(m,m) = -denom_inv

    FirstDerivMatrix(i_center,i_center) = -denom_inv;


%       PDL> ELSEIF using rearward differences THEN

  case {i_FD_backward}


%       PDL> Check to ensure that the point is not the first one
```

```
      if(ipoint == 1)
         iflag = -3;
         message = [ func_name, ': ', ...
            'Backward FD not allowed for first point'];
         if(i_error ~= 0)
            if(i_error > 1)
               save dump_error.mat;
            end
            error(message);
         else
            return;
         end
      end


%        PDL> Set denom_inv = 1/(grid_z(m)-grid_z(m-1))

      denom = Grid.z(i_center) - Grid.z(i_left);
      denom_inv = 1 / denom;


%        PDL> FirstDerivMatrix(m,m) = denom_inv

      FirstDerivMatrix(i_center,i_center) = denom_inv;


%        PDL> FirstDerivMatrix(m,m-1) = -denom_inv

      FirstDerivMatrix(i_center,i_left) = -denom_inv;


   otherwise

      iflag = -4;
      message = [func_name, ': ', ...
         'Unknown FD case (', int2str(i_FD_formula), ...
         ') encountered'];
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end


%PDL> ENDIF
```

```
end  % for FOR loop

iflag = 1;

return;
```