TR_1D_model1_SS\read_program_input.m

```
% TR_1D_model1_SS\read_program_input.m
%
% function [ProbDim,Reactor,Physical,Rxn,Grid,iflag] = ...
%    read_program_input();
%
% This procedure reads in the simulation parameters that are
% required to define the problem.  This is done only for an
% initial simulation and need not be done on a restart.  For
% the first version of the program, the input is made from
% the screen, so the routine also prints out sufficient
% information to explain the meaning of each input
% parameter.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/25/2001



function [ProbDim,Reactor,Physical,Rxn,Grid,iflag] = ...
   read_program_input();

func_name = 'read_program_input';

iflag = 0;


disp(' ');
disp(' ');
disp('Input the system parameters : ');
disp('The parameters are input in real units, where ');
disp('     L = unit of length');
disp('     M = unit of mass');
disp('     t = unit of time');
disp('     E = unit of energy');
disp('     T = unit of temperature');



% REACTOR DATA ----------------------------------------
% PDL> Input first the reactor size and flow rate data :
%       Reactor.len,Reactor.dia,Reactor.Qflow

disp(' ');
```

```
disp(' ');
disp('Input the reactor dimensions : ');
disp(' ');



% Perform assertion that a real scalar positive number has
% been entered.  This is performed by a function assert_scalar
% that gives first the value and name of the variable, the name
% of the function that is making the assertion, and values of
% 1 for the three flags that tell the assertion routine to make
% sure the value is real, positive, and not to check that it is
% an integer.

% Reactor.len
check_real=1; check_sign=1; check_int=0;
prompt = 'Input the length of the reactor (L) : ';
Reactor.len = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

% Reactor.dia
check_real=1; check_sign=1; check_int=0;
prompt = 'Input the diameter of the reactor (L) : ';
Reactor.dia = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

% Reactor.Qflow
check_real=1; check_sign=1; check_int=0;
prompt = 'Input the volumetric flow rate (L^3/t) : ';
Reactor.Qflow = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


%PDL> Calculate other static reactor properties :
%       reactor_volume, reactor_cross_area, reactor_surf_area,
%       reactor_velocity

Reactor.cross_area = pi/4*Reactor.dia*Reactor.dia;
Reactor.surf_area = pi*Reactor.dia*Reactor.len;
Reactor.volume = Reactor.cross_area*Reactor.len;
Reactor.velocity = Reactor.Qflow/Reactor.cross_area;


% PDL> Input reactor coolant variables :
%       Reactor.Temp_cool, Reactor.U_HT

disp(' ');

% Reactor.Temp_cool
check_real=1; check_sign=1; check_int=0;
prompt = 'Input the jacket coolant temperature (T) : ';
```

```
Reactor.Temp_cool = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

% Reactor.U_HT
check_real=1; check_sign=1; check_int=0;
prompt = ['Input the jacket heat transfer coefficient', ...
      ' (E/t/(L^2)/T) : '];
Reactor.U_HT = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% PDL> Input number of species, ProbDim.num_species

disp(' ');
disp(' ');

% ProbDim.num_species
check_real=1; check_sign=1; check_int=1;
prompt = 'Input the number of species : ';
ProbDim.num_species = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% PDL> Input reactor inlet properties :
%           Reactor.conc_in, Reactor.Temp_in

disp(' ');
disp(' ');
disp(['Input the inlet concentrations (mol/L^3) ', ...
   'and temperature (T).']);
disp(' ');

Reactor.conc_in = linspace(0,0,ProbDim.num_species)';

for ispecies = 1:ProbDim.num_species

   % Reactor.conc_in(ispecies)
   check_real=1; check_sign=2; check_int=0;
   prompt = ['Enter inlet concentration of species ', ...
         int2str(ispecies), ' : '];
   Reactor.conc_in(ispecies) = get_input_scalar( ...
      prompt,check_real,check_sign,check_int);

end

disp(' ');

% Reactor.Temp_in
check_real=1; check_sign=1; check_int=0;
prompt = 'Enter temperature of inlet : ';
Reactor.Temp_in = get_input_scalar(prompt, ...
```

```
   check_real,check_sign,check_int);



% PHYSICAL DATA ----------------------------------------
% PDL> Input physical data :
%           Physical.diffusivity, Physical.density,
%           Physical.Cp, Physical.thermal_conductivity

Physical.diffusivity = linspace(0,0,ProbDim.num_species)';

disp(' ');
disp(' ');
disp('Next, input the diffusivities of each species (L^2/t) : ');
disp(' ');

for ispecies = 1:ProbDim.num_species

   % Physical.diffusivity(ispecies)
   check_real=1; check_sign=2; check_int=0;
   prompt = ['Input diffusivity of species ', ...
             int2str(ispecies), ' : '];
   Physical.diffusivity(ispecies) = ...
      get_input_scalar(prompt, ...
         check_real,check_sign,check_int);

end

disp(' ');

% Physical.density
check_real=1; check_sign=1; check_int=0;
prompt = 'Input density of medium (M / L^3) : ';
Physical.density = get_input_scalar( ...
   prompt,check_real,check_sign,check_int);

disp(' ');

% Physical.Cp
check_real=1; check_sign=1; check_int=0;
prompt = ['Input heat capactity of medium', ...
       ' (E/T) : '];
Physical.Cp = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

disp(' ');

% Physical.thermal_conduct
check_real=1; check_sign=1; check_int=0;
prompt = ['Input thermal conductivity of medium', ...
       ' (E/t/L/T) : '];
```

```
Physical.thermal_conduct = get_input_scalar( ...
   prompt,check_real,check_sign,check_int);


% PDL> Set Physical.thermal_diff equal to
%           (thermal_conductivity/density/Cp)

Physical.thermal_diff = Physical.thermal_conduct / ...
   Physical.density / Physical.Cp;



% REACTION DATA ----------------------------------------

% PDL> Input the number of reactions,
%      ProbDim.num_rxn

disp(' ');
disp(' ');
disp('Now, enter the kinetic data for the reaction network');
disp(' ');
disp(' ');


% ProbDim.num_rxn
check_real=1; check_sign=1; check_int=1;
prompt = 'Enter the number of reactions : ';
ProbDim.num_rxn = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% PDL> Input the reaction data, one-by-one for each reaction :
%           Rxn.stoich_coeff, Rxn.is_rxn_elementary,
%           Rxn.ratelaw_exp, Rxn.k_ref,
%           Rxn.T_ref, Rxn.E_activ, Rxn.delta_H


% allocate a structure for the reaction data

Rxn.stoich_coeff = zeros(ProbDim.num_rxn,ProbDim.num_species);
Rxn.ratelaw_exp = zeros(ProbDim.num_rxn,ProbDim.num_species);
Rxn.is_rxn_elementary = linspace(0,0,ProbDim.num_rxn)';
Rxn.k_ref = linspace(0,0,ProbDim.num_rxn)';
Rxn.T_ref = linspace(0,0,ProbDim.num_rxn)';
Rxn.E_activ = linspace(0,0,ProbDim.num_rxn)';
Rxn.delta_H = linspace(0,0,ProbDim.num_rxn)';


disp(' ');
disp(' ');
disp('Now enter the kinetic data for each reaction.');
```

```
disp(' ');

for irxn = 1:ProbDim.num_rxn


   % We use a while loop to repeat the process of inputing the
   % reaction network until we accept it.  This is because
   % inputing the kinetic data is most prone to error.

   iflag_accept_Rxn = 0;

   while (iflag_accept_Rxn ~= 1)

      disp(' ');
      disp(' ');
      disp(' ');
      disp(['Enter kinetic data for reaction # ', ...
         int2str(irxn)]);

      disp(' ');
      disp('Stoichiometric coefficients ---');
      disp(' ');

      for ispecies = 1:ProbDim.num_species

         % Rxn.stoich_coeff(irxn,ispecies)
         check_real=1; check_sign=0; check_int=0;
         prompt = ['Enter stoich. coeff. for species # ', ...
               int2str(ispecies), ' : '];
         Rxn.stoich_coeff(irxn,ispecies) = ...
            get_input_scalar(prompt, ...
               check_real,check_sign,check_int);

      end

      disp(' ');

      % Rxn.is_rxn_elementary(irxn)
      check_real=1; check_sign=2; check_int=1;
      prompt = ['Is this reaction elementary ? ', ...
            '(1 = yes, 0 = no) : '];
      Rxn.is_rxn_elementary(irxn) = ...
         get_input_scalar(prompt, ...
            check_real,check_sign,check_int);


      % if the reaction is elementary, then the
      %rate law exponents can be obtained directly from
      % the stoichiometry coefficients

      if(Rxn.is_rxn_elementary(irxn) == 1)
```

```
   % initialize ratelaw_exp values to zero
   Rxn.ratelaw_exp(irxn,:) = ...
      linspace(0,0,ProbDim.num_species);

   % make a list of all reactants
   list_reactants = ...
      find(Rxn.stoich_coeff(irxn,:) < 0);

   % for each reactant, the rate law exponent is the
   % negative of the stoichiometric coefficient
   for i_reactant = 1:length(list_reactants)
      ispecies = list_reactants(i_reactant);
      Rxn.ratelaw_exp(irxn,ispecies) = ...
         -Rxn.stoich_coeff(irxn,ispecies);
   end


   % if the reaction is not elementary, we need
   % to input separate values of the rate
   % law exponents

else

   disp(' ');

   for ispecies = 1:ProbDim.num_species

      % Rxn.ratelaw_exp(irxn,ispecies)
      check_real=1; check_sign=0; check_int=0;
      prompt = ...
      ['Enter the rate law exponent for species # ', ...
            int2str(ispecies) ' : '];
      Rxn.ratelaw_exp(irxn,ispecies) = ...
         get_input_scalar(prompt, ...
            check_real,check_sign,check_int);

   end

end


% Now, enter the reference rate law constants

disp(' ');
disp(' ');

% Rxn.T_ref(irxn)
check_real=1; check_sign=1; check_int=0;
prompt = ['Enter the kinetic data reference ', ...
   'temperature (T) : '];
```

```
Rxn.T_ref(irxn) = get_input_scalar( ...
   prompt,check_real,check_sign,check_int);

disp(' ');

% Rxn.k_ref(irxn)
check_real=1; check_sign=2; check_int=0;
prompt = 'Enter the rate constant at reference T : ';
Rxn.k_ref(irxn) = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% Finally , the activation energy (divided by the value of
% the ideal gas constant in the chosen units) and the heat
% of reaction are input.

disp(' ');

% Rxn.E_activ(irxn)
check_real=1; check_sign=2; check_int=0;
prompt = ['Enter activation energy divided ', ...
      'by gas constant (T) : '];
Rxn.E_activ(irxn) = get_input_scalar( ...
   prompt,check_real,check_sign,check_int);

disp(' ');

% Rxn.delta_H(irxn)
check_real=1; check_sign=0; check_int=0;
prompt = 'Enter the heat of reaction (E / mol) : ';
Rxn.delta_H(irxn) = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);



% We now write out the kinetic data to the
% screen and ask if this is to be accepted.

disp(' ');
disp(' ');
disp('Read-back of entered kinetic data : ');

% List the reactants.
list_reactants = ...
   find(Rxn.stoich_coeff(irxn,:) < 0);
disp(' ');
disp('Reactant species and stoich. coeff. :');
for count=1:length(list_reactants)
   ispecies = list_reactants(count);
   disp([int2str(ispecies), '     ', ...
      num2str(Rxn.stoich_coeff(irxn,ispecies))]);
```

```
   end

   % List the products.
   list_products = ...
      find(Rxn.stoich_coeff(irxn,:) > 0);
   disp(' ');
   disp('Product species and stoich. coeff. : ');
   for count=1:length(list_products)
      ispecies = list_products(count);
      disp([int2str(ispecies), '     ', ...
         num2str(Rxn.stoich_coeff(irxn,ispecies))]);
   end

   % Tell whether the reaction is elementary.
   disp(' ');
   if(Rxn.is_rxn_elementary(irxn) == 1)
      disp('Reaction is elementary');
   else
      disp('Reaction is NOT elementary');
   end

   % Write the ratelaw exponents
   disp(' ');
   list_ratelaw_species = ...
      find(Rxn.ratelaw_exp(irxn,:) ~= 0);
   disp('Rate law exponents : ');
   for count = 1:length(list_ratelaw_species)
      ispecies = list_ratelaw_species(count);
      disp([int2str(ispecies), '     ', ...
         num2str(Rxn.ratelaw_exp(irxn,ispecies))]);
   end

   % Write the kinetic data.
   disp(' ');
   disp([ 'T_ref = ', num2str(Rxn.T_ref(irxn))]);
   disp(' ');
   disp([ 'k_ref = ', num2str(Rxn.k_ref(irxn))]);
   disp(' ');
   disp([ 'E_activ = ', num2str(Rxn.E_activ(irxn))]);
   disp(' ');
   disp([ 'delta_H = ', num2str(Rxn.delta_H(irxn))]);

   disp(' ');
   prompt = 'Accept these rate parameters? (0=no, 1=yes) : ';
   check_real=1;check_sign=2;check_int=1;
   iflag_accept_Rxn = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);


end   % for while loop to accept data
```

```
end       % irxn for loop



% GRID DATA ----------------------------------
% PDL> Input number of grid points, Grid.num_pts

disp(' ');
disp(' ');

% Grid.num_pts
check_real=1; check_sign=1; check_int=1;
prompt = 'Enter the number of grid points in the z direction : ';
Grid.num_pts = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

% allocate space for the column vector of z
% grid point values
disp('Initializing grid points to all zeros');
Grid.z = linspace(0,0,Grid.num_pts)';


iflag = 1;

return;
```

TR_1D_model1_SS\read_solver_input.m

```
% TR_1D_model1_SS\read_solver_input.m
%
% function [Solver,iflag] = read_solver_input();
%
% This procedure reads in from the screen the simulation
% parameters that control the solver operation.  New values
% of these parameters are read every time the program runs,
% even on restarts.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/25/2001



function [Solver,iflag] = read_solver_input();

iflag = 0;

func_name = 'read_solver_input';


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


disp(' ');
disp(' ');
disp('Enter the parameters for the steady state solver.');


% PDL> Input Solver.max_iter_time

disp(' ');
disp('First, enter the maximum number of iterations of the');
disp('implicit Euler method that is used to approach the');
disp('vicinity of the steady state solution.  If a value of');
disp('0 is entered, then no implicit Euler steps are performed');
disp('and the solver goes directly to Newtons method.');
disp(' ');

% Solver.max_iter_time
check_real=1; check_sign=2; check_int=1;
prompt = 'Enter max. # of time iterations : ';
```

```
Solver.max_iter_time = get_input_scalar( ...
   prompt,check_real,check_sign,check_int);


% PDL> If Solver.max_iter_time IS NOT 0 THEN

if(Solver.max_iter_time ~= 0)

   disp(' ');
   disp('Enter data for the time integration stage.');


%        PDL> Input Solver.dt

   check_real=1; check_sign=1; check_int=0;
   prompt = 'Enter the time step dt (t) : ';
   Solver.dt = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);


%        PDL> Input Solver.atol_time

   check_real=1; check_sign=1; check_int=0;
   prompt = 'Enter the abs. tolerance for the time integration : ';
   Solver.atol_time = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);


% Otherwise, set dummy values

else

   Solver.dt = 1;
   Solver.atol_time = 1;


%PDL> ENDIF

end


%PDL> Input data for Newton's method solver,
%           Solver.max_iter_Newton, Solver.atol_Newton

disp(' ');
disp('Now enter parameters for Newtons method solver.');

% Solver.max_iter_Newton
check_real=1; check_sign=2; check_int=1;
prompt = 'Enter max. # of Newtons method iterations : ';
Solver.max_iter_Newton = get_input_scalar(...
```

```
   prompt,check_real,check_sign,check_int);

% Solver.atol_Newton
check_real=1; check_sign=1; check_int=0;
prompt = 'Enter abs. tolerance for Newtons method solver : ';
Solver.atol_Newton = get_input_scalar(...
   prompt,check_real,check_sign,check_int);


% PDL> Set Solver.iflag_Adepend to 0 to signify that
%   the A matrix obtained by discretizing the system
%   is not state-dependent.

Solver.iflag_Adepend = 0;


% PDL> Set Solver.iflag_nonneg to 1 to signify that
%   the components of the state vector should be
%   enforced to be non-negative at every iteration
%   of the solution procedure.

Solver.iflag_nonneg = 1;


% PDL> Input desired value for Solver.iflag_verbose

disp(' ');
check_real=1; check_sign=0; check_int=0;
prompt = 'Solver to be verbose (enter 1) or silent (other value) : ';
Solver.iflag_verbose = get_input_scalar( ...
   prompt,check_real,check_sign,check_int);


iflag = 1;

return;
```