% CSTR_ODE\CSTR_ODE.m

```
%
% function iflag_main = CSTR_ODE();
%
% This MATLAB program calculates the dynanamic concentration and
% temperature profiles within a continuous stirred-tank reactor.
% This program models a CSTR in overflow mode (no volume change),
% or, if the flow rate is set to zero, the model simulates a batch
% reactor.  At the moment, semi-batch reactors with a volume
% that changes with time is not supported.
%
% This program uses the MATLAB ODE-IVP solver ode15s, a multistep
% method that works well for stiff systems.
%
% K. Beers
% MIT ChE
% 10/28/2001


function iflag_main = CSTR_ODE();


iflag_main = 0;



% First, read in the problem definition data from the keyboard or
% from an input file.

imode_input = input('Select mode of input (0 = keyboard, 1 = file) : ');

if(~imode_input)
   [ProbDim,Reactor,Inlet,Physical,Rxn,StateInit,iflag] = ...
      read_program_input;
   if(iflag <= 0)
      iflag_main = -1;
      error(['CSTR_ODE: Error returned from read_program_input = ', ...
         int2str(iflag)]);
   end

else
   file_name = input('Enter input file name (without .m) : ','s');
   [ProbDim,Reactor,Inlet,Physical,Rxn,StateInit,iflag] = ...
      feval(file_name);

end


% We now stack the initial state data into the state vector.
```

```
x_init = stack_state(StateInit,ProbDim.num_species, ...
   Reactor.nonisothermal);


% Then, the options for the MATLAB ODE-IVP solver are set.
options = odeset('RelTol',1e-4,'AbsTol',1e-6, ...
   'Refine',10,'Stats','on');


% The user is now asked to the length of the time period to
% be simulated.
% time_final
prompt = 'Enter final simulation time : ';
check_real=1; check_sign=1; check_int=0;
time_final = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% Now, the ODE-IVP solver is called.
[t,x_traj] = ode15s(@CSTR_ODE_calc_f, ...
   [0 time_final],x_init, ...
   options,ProbDim,Reactor,Inlet,Physical,Rxn);


% Plot the simulation results.
for ispecies=1:ProbDim.num_species
   figure;
   plot(t,x_traj(:,ispecies));
   title(['Concentration of species # ', ...
         int2str(ispecies)]);
   xlabel('time');
   ylabel(['c (',int2str(ispecies),')']);
end

if(Reactor.nonisothermal)
   figure;
   plot(t,x_traj(:,ProbDim.num_species+1));
   title('Reactor Temperature');
   xlabel('time');
   ylabel('T');
end


% Save the results to a binary output file.
save CSTR_ODE_results.mat;

iflag_main = 1;

return;
```

% CSTR_ODE\read_program_input.m

```
%
% function [ProbDim,Reactor,Inlet,Physical,Rxn,StateInit,iflag] = ...
%    read_program_input();
%
% This procedure reads in the simulation parameters that are
% required to define a single CSTR dynamic simulation.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 10/28/2001
%
% Version as of 10/28/2001



function [ProbDim,Reactor,Inlet,Physical,Rxn,StateInit,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 volume and heat transfer data :

disp(' ');
disp(' ');
disp('Input the reactor data : ');
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.

disp(' ');
disp('Reactor size information : ');

% Reactor.volume
check_real=1; check_sign=1; check_int=0;
prompt = 'Input the volume of the reactor (L^3) : ';
Reactor.volume = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

% Reactor.F_vol
check_real=1; check_sign=1; check_int=0;
prompt = 'Input volumetric flowrate through reactor (L^3/t) : ';
Reactor.F_vol = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% Now, the user selects whether to perform a non-isothermal
% calculation or to keep the temperature constant at the
% value of Temp_cool.

% Reactor.nonisothermal
check_real=1; check_sign=1; check_int=2;
prompt = 'Simulate isothermal (0) or non-isothermal (1) reactor? : ';
Reactor.nonisothermal = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% If operate in isothermal mode, only input set temperature.
if(~Reactor.nonisothermal)

   % Reactor.Temp_cool
   check_real=1; check_sign=1; check_int=0;
   prompt = 'Enter the constant reactor temperature : ';
   Reactor.Temp_cool = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);

   % We now set the other non-needed coolant information
   % to dummy values.
   Reactor.F_cool = 1;
   Reactor.U_HT = 1;
   Reactor.A_HT = 1;
   Reactor.Cp_cool = 1;


% otherwise, if simulating non-isothermal reactor, input
```

```
% the data for the coolant jacket.
else

   disp(' ');
   disp('Reactor coolant information : ');

   % Reactor.F_cool
   check_real=1; check_sign=1; check_int=0;
   prompt = 'Input the coolant flowrate (L^3/t) : ';
   Reactor.F_cool = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);

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

   % Reactor.A_HT
   check_real=1; check_sign=1; check_int=0;
   prompt = 'Input the jacket heat transfer area (L^2) : ';
   Reactor.A_HT = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);

   % Reactor.Cp_cool
   check_real=1; check_sign=1; check_int=0;
   prompt = 'Input the reactor coolant heat capacity (E/mol/T) : ';
   Reactor.Cp_cool = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);

end


% 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 inlet properties :
%           Inlet.conc, Inlet.Temp
% and the initial concentrations and temperature within
% the reactor.

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

Inlet.conc = linspace(0,0,ProbDim.num_species)';
StateInit.conc = Inlet.conc;
StateInit.Temp = 0;

for ispecies = 1:ProbDim.num_species

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

   % StateInit.conc(ispecies)
   check_real=1; check_sign=2; check_int=0;
   prompt = ['Enter initial reactor concentration of species ', ...
         int2str(ispecies), ' : '];
   StateInit.conc(ispecies) = get_input_scalar( ...
      prompt,check_real,check_sign,check_int);

end

disp(' ');

% Inlet.Temp
if(Reactor.F_vol)
   check_real=1; check_sign=1; check_int=0;
   prompt = 'Enter temperature of inlet : ';
   Inlet.Temp = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);
else
   Inlet.Temp = 1;
end

% StateInit.Temp
```

```
check_real=1; check_sign=1; check_int=0;
prompt = 'Enter initial reactor temperature : ';
StateInit.Temp = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);



% PHYSICAL DATA ----------------------------------------
% PDL> Input physical data :
%           Physical.Cp_data(num_species,4)
% This is done only in nonisothermal mode.

Physical.Cp_data = zeros(ProbDim.num_species,4);

if(Reactor.nonisothermal)

   disp(' ');
   disp(' ');
   disp('Input molar heat capacities of each species (E/mol/T) : ');
   disp('Temperature dependence : Cp = C0 + C1*T + C2*T^2 + C3*T^3');
   disp(' ');

   for ispecies = 1:ProbDim.num_species
      prompt = ['Input molar heat capacity data of species ', ...
            int2str(ispecies), ' : '];
      disp(prompt);

      % Physical.Cp_data(ispecies,:)
      check_real=1; check_sign=0; check_int=0;
      for j=1:4
         prompt = ['Enter C', int2str(j-1), ' : '];
         Physical.Cp_data(ispecies,j) = ...
            get_input_scalar(prompt, ...
               check_real,check_sign,check_int);
      end
   end

else

   % use dummy value for isothermal case
   Physical.Cp_data(:,1) = 1;
end



% 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, meaning only
% that the rate law exponents are automatically set by
% the stoichiometric coefficients.
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


iflag = 1;

return;
```

% CSTR_ODE\stack_state.m

```
%
% function [x_state,iflag] = stack_state(State, num_species, ...
%              isnonisothermal);
```

```
%
% This procedure stacks the concentration and temperature
% data into a single master array.
%
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 10/20/2001
%
% Version as of 10/28/2001



function [x_state,iflag] = stack_state(State,num_species, ...
   isnonisothermal);

iflag = 0;

func_name = 'stack_state';


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

% check for errors in input parameters


% check dimensions
assert_scalar(1,num_species,'num_species', ...
   func_name,1,1,1);


% check if State is proper structure type
StateType.num_fields = 1;
if(isnonisothermal)
   StateType.num_fields = 2;
end
% .conc
ifield = 1;
FieldType.name = 'conc';
FieldType.is_numeric = 1;
FieldType.num_rows = num_species;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 0;
StateType.field(ifield) = FieldType;
% .Temp
```

```
if(isnonisothermal)
   ifield = 2;
   FieldType.name = 'Temp';
   FieldType.is_numeric = 1;
   FieldType.num_rows = 1;
   FieldType.num_columns = 1;
   FieldType.check_real = 1;
   FieldType.check_sign = 1;
   FieldType.check_int = 0;
   StateType.field(ifield) = FieldType;
end
% call assertion routine for structure
assert_structure(i_error,State,'State', ...
   func_name,StateType);


% allocate x_state column vector and
% initialize to zeros
num_DOF = num_species;
if(isnonisothermal)
   num_DOF = num_DOF + 1;
end
x_state = linspace(0,0,num_DOF)';


%PDL> First, we stack the concentrations

%PDL> Set pos_counter to zero

pos_counter = 0;


%PDL> FOR ispecies FROM 1 TO ProbDim.num_species

for ispecies = 1:num_species

   x_state(pos_counter+1) = State.conc(ispecies);

%   PDL> Increment pos_counter by num_pts

   pos_counter = pos_counter + 1;


%PDL> ENDFOR

end


%PDL> Next, we stack the temperature

if(isnonisothermal)
```

```
   x_state(pos_counter+1) = State.Temp;
end

iflag = 1;

return;
```

% CSTR_ODE\unstack_state.m

```
%
% function [State,iflag] = unstack_state(x_state, num_species, ...
%         isnonisothermal);
%
% This procedure stacks the concentration and temperature
% data into a single master array.
%
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 10/20/2001
%
% Version as of 10/28/2001



function [State,iflag] = unstack_state(x_state,num_species, ...
   isnonisothermal);

iflag = 0;

func_name = 'unstack_state';


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

% check for errors in input parameters


% check dimensions
check_real=1; check_sign=1; check_int=1;
assert_scalar(i_error,num_species,'num_species', ...
   func_name,check_real,check_sign,check_int);

% check x_state vector
num_DOF = num_species;
```

```
if(isnonisothermal)
   num_DOF = num_DOF + 1;
end

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


% Allocate space for State data.
State.conc = linspace(0,0,num_species)';
if(isnonisothermal)
   State.Temp = 0;
end



%PDL> First, we unstack the concentrations

%PDL> Set pos_counter to zero

pos_counter = 0;


%PDL> FOR ispecies FROM 1 TO ProbDim.num_species

for ispecies = 1:num_species

   State.conc(ispecies) = x_state(pos_counter+1);

%   PDL> Increment pos_counter by num_pts

   pos_counter = pos_counter + 1;


%PDL> ENDFOR

end


%PDL> Next, we stack the temperature

if(isnonisothermal)
   State.Temp = x_state(pos_counter+1);
end

iflag = 1;

return;
```

% CSTR_ODE\CSTR_ODE_calc_f.m

```
%
% function [fval,iflag] = CSTR_ODE_calc_f(x_state, ...
%    ProbDim,Reactor,Inlet,Physical,Rxn);
%
% This MATLAB file calculates the vector of time derivatives of
% the concentrations and temperature within a CSTR in overflow
% mode.  This model incoporates a general reaction network.
%
% K. Beers
% MIT ChE
% 10/28/2001

function [fval,iflag] = CSTR_ODE_calc_f(time,x_state, ...
   ProbDim,Reactor,Inlet,Physical,Rxn);

iflag = 0;

if(Reactor.nonisothermal)
   num_DOF = ProbDim.num_species + 1;
else
   num_DOF = ProbDim.num_species;
end
fval = linspace(0,0,num_DOF)';


% Next, unstack the state vector.
State = unstack_state(x_state,ProbDim.num_species, ...
   Reactor.nonisothermal);
% if operating in isothermal mode, set temperature to that
% of the coolant jacket.
if(~Reactor.nonisothermal)
   State.Temp = Reactor.T_cool;
end

% Next, calculate the heat capacities for the inlet and
% reactor temperatures for each species.  This only need
% be done for nonisothermal simulations, otherwise dummy
% values are set.
if(Reactor.nonisothermal)
   Cp = linspace(0,0,ProbDim.num_species)';
   Cp_avg = 0;  % molar average heat capactity of medium

   Cp_in = linspace(0,0,ProbDim.num_species)';
   Cp_in_avg = 0;  % molar average heat capacity of inlet

   for ispecies = 1:ProbDim.num_species
      Cp_in(ispecies) = Physical.Cp_data(ispecies,1) + ...
         Physical.Cp_data(ispecies,2)*Inlet.Temp + ...
```

```
            Physical.Cp_data(ispecies,3)*Inlet.Temp^2 + ...
            Physical.Cp_data(ispecies,4)*Inlet.Temp^3;
         Cp_in_avg = Cp_in_avg + ...
            Cp_in(ispecies)*State.conc(ispecies);
         Cp(ispecies) = Physical.Cp_data(ispecies,1) + ...
            Physical.Cp_data(ispecies,2)*State.Temp + ...
            Physical.Cp_data(ispecies,3)*State.Temp^2 + ...
            Physical.Cp_data(ispecies,4)*State.Temp^3;
         Cp_avg = Cp_avg + Cp(ispecies)*State.conc(ispecies);
      end
      Cp_in_avg = Cp_in_avg / sum(Inlet.conc);
      Cp_avg = Cp_avg / sum(State.conc);

   else
      Cp = linspace(1,1,ProbDim.num_species)';
      Cp_avg = 1;
      Cp_in = Cp;
      Cp_in_avg = 1;
   end

   % Next, calculate the reaction rate using the general reaction network.
   [RxnRate, iflag2] = reaction_network_model(...
      ProbDim.num_species,ProbDim.num_rxn, ...
      State.conc,State.Temp,Rxn,1,Cp_avg);
   if(iflag2 <= 0)
      iflag = iflag2;
      error(['CSTR_ODE_calc_f: reaction_network_model returns error = ', ...
            int2str(iflag2)]);
   end


   % Calculate the function vector.

   % Mass Balances on each species

   for ispecies = 1:ProbDim.num_species
      fval(ispecies) = Reactor.F_vol/Reactor.volume * ...
         (Inlet.conc(ispecies) - State.conc(ispecies));
      source = 0;
      for irxn = 1:ProbDim.num_rxn
         source = source + ...
            Rxn.stoich_coeff(irxn,ispecies)*RxnRate.rate(irxn);
      end
      fval(ispecies) = fval(ispecies) + source;
   end


   % enthalpy balance, only done for nonisothermal mode simulation
   if(Reactor.nonisothermal)
      iDOF = ProbDim.num_species + 1;
```

```
   % inlet/outlet fluxes of enthalpy
   fval(iDOF) = 0;
   for ispecies = 1:ProbDim.num_species
      fval(iDOF) = fval(iDOF) + ...
         Inlet.conc(ispecies)*Cp_in(ispecies)*Inlet.Temp - ...
         State.conc(ispecies)*Cp(ispecies)*State.Temp;
   end
   fval(iDOF) = fval(iDOF)*Reactor.F_vol/Reactor.volume;

   % enthalpy due to net chemical reaction
   source = 0;
   for irxn = 1:ProbDim.num_rxn
      source = source - Rxn.delta_H(irxn)*RxnRate.rate(irxn);
   end
   fval(iDOF) = fval(iDOF) + source;

   % enthalpy flux to coolant jacket
   Q_cool = Reactor.F_cool*Reactor.Cp_cool* ...
      (State.Temp - Reactor.T_cool) * ...
      (1 - exp(-Reactor.U_HT*Reactor.A_HT/ ...
            Reactor.F_cool/Reactor.Cp_cool));
   fval(iDOF) = fval(iDOF) - Q_cool/Reactor.volume;
end


iflag = 1;

return;
```

% CSTR_ODE\reaction_network_model.m

```
%
% function [RxnRate, iflag] = ...
%    reaction_network_model(num_species,num_rxn, ...
%    conc_loc,Temp_loc,Rxn,density,Cp);
%
% This procedure evaluates the rates of each reaction
% and the derivatives of the rates with respect to the
% concentrations and temperature for a general reaction
% network.  The rate laws are characterized by the
% product of each concentration raised to an
% exponential power.  The rate constants are temperature
% dependent, according to an Arrhenius expression based
% on an activation energy and the value of the rate
% constant at a specified reference temperature.
% Also, the contributions to the time derivatives of
% the concentrations and the temperature due to the
% total effect of reaction are returned.
```

```
%
% INPUT :
% =======
% num_species       INT
%               The number of species
% num_rxn           INT
%               The number of reactions
% conc  REAL(num_species)
%           This is a column vector of the concentrations
%           of each species at a single point
% Temp  REAL
%           This is the temperature at a single point
%
% Rxn   This structure contains the kinetic data
%       for the general reaction network.  The fields
%       are :
% .stoich_coeff      REAL(num_rxn,num_species)
%                                 the stoichiometric coefficients
%               possibly fractional) of each
%                                       species in each reaction.
% .ratelaw_exp      REAL(num_rxn,num_species)
%                                       the exponential power (possibly fractional)
%               to which the concentration of each species
%               is raised each reaction's rate law.
% .is_rxn_elementary    INT(num_rxn)
%                                       if a reaction is elementary, then the
%               rate law exponents are zero for the
%               product species and the negative of the
%               stoichiometric coefficient for the
%               reactant species.  In this case, we need
%               not enter the corresponding components of
%               ratelaw_exp since these are determined by
%               the corresponding values in stoich_coeff.
%               We specify that reaction number irxn is
%               elementary by setting
%               is_rxn_elementary(irxn) = 1.
%               Otherwise (default = 0), we assume that
%               the reaction is not elementary and require
%               the user to input the values of
%               ratelaw_exp for reaction # irxn.
% .k_ref                    REAL(num_rxn)
%               the rate constants of each reaction at a
%               specified reference temperature
% .T_ref                    REAL(num_rxn)
%                                       This is the value of the reference
%               temperature used to specify the
%               temperature dependence of each
%                                       rate constant.
% .E_activ                          REAL(num_rxn)
%                                       the constant activation energies of
%               each reaction divided by the ideal
```

```
%                gas constant
% .delta_H                       REAL(num_rxn)
%                                    the constant heats of reaction
%
% density          REAL
%                    the density of the medium
% Cp             REAL
%                    the heat capacity of the medium
%
% OUTPUT :
% ========
% RxnRate   data structure containing the following fields :
% .time_deriv_c            REAL(num_species)
%           this is a column vector of the time derivatives of the
%           concentration due to all reactions
% .time_deriv_T                    REAL
%           this is the time derivative of the temperature due to
%           the effect of all the reactions
% .rate                            REAL(num_rxn)
%           this is a column vector of the rates of each reaction
% .rate_deriv_c            REAL(num_rxn,num_species)
%           this is a matrix of the partial derivatives of each reaction
%           rate with respect to the concentrations of each species
% .rate_deriv_T            REAL(num_rxn)
%           this is a column vector of the partial derivatives of each
%           reaction rate with respect to the temperature
% k                REAL(num_rxn)
%       this is a column vector of the rate constant values at the
%       current temperature
% .source_term      REAL(num_rxn)
%       this is a column vector of the values in the rate law expression
%       that are dependent on concentration.
%       For example, in the rate law :
%          R = k*[A]*[B]^2,
%       the source term value is [A]*[B]^2.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/25/2001
```

```
function [RxnRate, iflag] = ...
   reaction_network_model(num_species,num_rxn, ...
   conc_loc,Temp_loc,Rxn,density,Cp);

iflag = 0;
```

```
% this integer flag controls the action taken
% when an assertion fails.  See the assertion
% routines for a description of its use.
i_error = 1;

func_name = 'reaction_network_model';


% Check input

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

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

% conc_loc
dim = num_species; check_column=0;
check_real=1; check_sign=0; check_int=0;
assert_vector(i_error,conc_loc,'conc_loc', ...
   func_name,dim,check_real,check_sign, ...
   check_int,check_column);
% now, make sure all concentrations are non-negative
list_neg = find(conc_loc < 0);
for count=1:length(list_neg)
   ispecies = list_neg(count);
   conc_loc(ispecies) = 0;
end

% Temp_loc
check_real=1; check_sign=0; check_int=0;
assert_scalar(i_error,Temp_loc,'Temp_loc', ...
   func_name,check_real,check_sign,check_int);
% make sure the temperature is positive
trace = 1e-20;
if(Temp_loc <= trace)
   Temp_loc = trace;
end

% Rxn
RxnType.struct_name = 'Rxn';
RxnType.num_fields = 7;
% Now set the assertion properties of each field.
% .stoich_coeff
ifield = 1;
FieldType.name = 'stoich_coeff';
```

```
FieldType.is_numeric = 1;
FieldType.num_rows = num_rxn;
FieldType.num_columns = num_species;
FieldType.check_real = 1;
FieldType.check_sign = 0;
FieldType.check_int = 0;
RxnType.field(ifield) = FieldType;
% .ratelaw_exp
ifield = 2;
FieldType.name = 'ratelaw_exp';
FieldType.is_numeric = 2;
FieldType.num_rows = num_rxn;
FieldType.num_columns = num_species;
FieldType.check_real = 1;
FieldType.check_sign = 0;
FieldType.check_int = 0;
RxnType.field(ifield) = FieldType;
% .is_rxn_elementary
ifield = 3;
FieldType.name = 'is_rxn_elementary';
FieldType.is_numeric = 1;
FieldType.num_rows = num_rxn;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 1;
RxnType.field(ifield) = FieldType;
% .k_ref
ifield = 4;
FieldType.name = 'k_ref';
FieldType.is_numeric = 1;
FieldType.num_rows = num_rxn;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 0;
RxnType.field(ifield) = FieldType;
% .T_ref
ifield = 5;
FieldType.name = 'T_ref';
FieldType.is_numeric = 1;
FieldType.num_rows = num_rxn;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 1;
FieldType.check_int = 0;
RxnType.field(ifield) = FieldType;
% .E_activ
ifield = 6;
FieldType.name = 'E_activ';
FieldType.is_numeric = 1;
```

```
FieldType.num_rows = num_rxn;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 2;
FieldType.check_int = 0;
RxnType.field(ifield) = FieldType;
% .delta_H
ifield = 7;
FieldType.name = 'delta_H';
FieldType.is_numeric = 1;
FieldType.num_rows = num_rxn;
FieldType.num_columns = 1;
FieldType.check_real = 1;
FieldType.check_sign = 0;
FieldType.check_int = 0;
RxnType.field(ifield) = FieldType;
% call assertion routine for structure
assert_structure(i_error,Rxn,'Rxn',func_name,RxnType);

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

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


%PDL> Initialize all output variables to zeros

RxnRate.time_deriv_c = linspace(0,0,num_species)';
RxnRate.time_deriv_T = 0;
RxnRate.rate = linspace(0,0,num_rxn)';
RxnRate.rate_deriv_c = zeros(num_rxn,num_species);
RxnRate.rate_deriv_T = linspace(0,0,num_rxn)';
RxnRate.k = linspace(0,0,num_rxn)';
RxnRate.source_term = linspace(0,0,num_rxn)';


%PDL>  For every reaction, calculate the rates and
%   their derivatives with respect to the
%   concentrations and temperatures
%   FOR irxn FROM 1 TO num_rxn

for irxn = 1:num_rxn


%PDL> Calculate rate constant at the current temperature
```

```
   factor_T = exp(-Rxn.E_activ(irxn) * ...
      (1/Temp_loc - 1/Rxn.T_ref(irxn)));
   RxnRate.k(irxn) = Rxn.k_ref(irxn)*factor_T;


%PDL> Calculate the derivative of the rate constant with
%   respect to temperature

   d_rate_k_d_Temp = RxnRate.k(irxn) * ...
      Rxn.E_activ(irxn)/(Temp_loc^2);


%PDL> Set ratelaw_vector to be of length num_species whose
%   elements are the concentrations of each species
%   raised to the power ratelaw_exp(irxn,ispecies).
%   If the exponent is 0, automatically set corresponding
%   element to 1.

   ratelaw_vector = linspace(1,1,num_species)';
   list_species = find(Rxn.ratelaw_exp(irxn,:) ~= 0);
   for count=1:length(list_species)
      ispecies = list_species(count);
      ratelaw_vector(ispecies) = ...
         conc_loc(ispecies) ^ Rxn.ratelaw_exp(irxn,ispecies);
   end


%PDL> Calculate the ratelaw source term that is the product
%   of all elements of ratelaw_vector

   RxnRate.source_term(irxn) = prod(ratelaw_vector);


%PDL> The rate of reaction # irxn is equal to the product of
%   the ratelaw source term with the value of the rate constant

   RxnRate.rate(irxn) = RxnRate.k(irxn) * ...
      RxnRate.source_term(irxn);


%PDL> Set rxn_rate_deriv_T(irxn) to be equal to the product of
%        the temperature derivative of the rate constant times the
%          ratelaw source term

   RxnRate.rate_deriv_T(irxn) = ...
      d_rate_k_d_Temp * RxnRate.source_term(irxn);


%PDL> FOR EVERY ispecies WHERE
%       ratelaw_exp(irxn,ispecies) IS non-zero
```

```
   for count=1:length(list_species)
      ispecies = list_species(count);


%PDL> Set vector_work = ratelaw_vector and replace the
%          ispecies element with
%          ratelaw_exp(irxn,ispecies)*
%   conc(ispecies)^(ratelaw_exp(irxn,ispecies)-1)
%   If ratelaw_exp(irxn,ispecies) is exactly 1, then do
%   special case where replace element with 1

      vector_work = ratelaw_vector;
      if(Rxn.ratelaw_exp(irxn,ispecies) == 1)
         vector_work(ispecies) = 1;
      else
         exponent = Rxn.ratelaw_exp(irxn,ispecies);
         vector_work(ispecies) = exponent * ...
            (conc_loc(ispecies) ^ (exponent-1));
      end


%       PDL> Set rxn_rate_deriv_c(irxn,ispecies) equal to the
%                       product of all components of this vector
%                       multiplied by the rate constant

      RxnRate.rate_deriv_c(irxn,ispecies) = ...
         RxnRate.k(irxn) * prod(vector_work);


%       PDL> ENDFOR for sum over participating species

   end


%       PDL> FOR EVERY ispecies WHERE
%                Rxn.stoich_coeff(irxn,ispecies) IS non-zero

   list_species = find(Rxn.stoich_coeff(irxn,:) ~= 0);
   for count=1:length(list_species)
      ispecies = list_species(count);


%                PDL> Increment rxn_time_deriv_c(ispecies) by
%                       Rxn.stoich_coeff(irxn,ispecies)
%                       multiplied with the rxn_rate(irxn)

      RxnRate.time_deriv_c(ispecies) = ...
         RxnRate.time_deriv_c(ispecies) + ...
         Rxn.stoich_coeff(irxn,ispecies) * ...
            RxnRate.rate(irxn);
```

```
%       PDL> ENDFOR over participating species
   end


%       PDL> Increment rxn_time_deriv_T by the negative of
%                Rxn.delta_H divided by the product
%                of density and heat capactity
%                and then multiply by rxn_rate(irxn)

   RxnRate.time_deriv_T = RxnRate.time_deriv_T - ...
      (Rxn.delta_H(irxn)/density/Cp)*RxnRate.rate(irxn);


%PDL> ENDFOR over reactions

end


iflag = 1;

return;
```

% CSTR_ODE\get_input_scalar.m

```
%
% function value = get_input_scalar(prompt, ...
%    check_real,check_sign,check_int);
%
% This MATLAB m-file gets from the user an
% input scalar value of the appropriate type.
% It asks for input over and over again
% until a correctly typed input value
% is entered.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/25/2001



function value = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


func_name = 'get_input_scalar';
```

```
name = 'trial_value';


input_OK = 0;

while (input_OK ~= 1)
   trial_value = input(prompt);
   [iflag_assert,message] = ...
      assert_scalar(0,trial_value, ...
         name,func_name, ...
         check_real,check_sign,check_int);
    if(iflag_assert == 1)
       input_OK = 1;
       value = trial_value;
    else
       disp(message);
    end

 end

 return;
```

% CSTR_ODE\assert_scalar.m

```
%
% function [iflag_assert,message] = assert_scalar( ...
%   i_error,value,name,func_name, ...
%   check_real,check_sign,check_int,i_error);
%
% This m-file contains logical checks to assert than an
% input value is a type of scalar number.  This function is
% passed the value and name of the variable, the name of
% the function making the assertion, and four integer
% flags that have the following usage :
%
% i_error : controls what to do if test fails
%       if i_error is non-zero, then use error()
%       MATLAB command to stop execution, otherwise
%       just return the appropriate negative number.
%       if i_error > 1, then dump current state to
%       dump_error.mat before calling error().
%
% check_real : check to examine whether input number is
% real or not.  See table after function header for set
% values of these case flags
% check_real = i_real (make sure that input is real)
% check_real = i_imag (make sure that input is
%          purely imaginary)
```

```
% any other value of check_real (esp. 0) results
%       in no check
%
%   check_real
%       i_real = 1;
%       i_imag = -1;
%
% check_sign : check to examine sign of input value
% see table after function header for set values
% of these case flags
% check_sign = i_pos (make sure input is positive)
% check_sign = i_nonneg (make sure input is non-negative)
% check_sign = i_neg (make sure input is negative)
% check_sign = i_nonpos (make sure input is non-positive)
% check_sign = i_nonzero (make sure input is non-zero)
% check_sign = i_zero (make sure input is zero)
% any other value of check_sign (esp. 0)
%       results in no check
%
% check_sign
%       i_pos = 1;
%       i_nonneg = 2;
%       i_neg = -1;
%       i_nonpos = -2;
%       i_nonzero = 3;
%       i_zero = -3;
%
% check_int : check to see if input is an integer
% if = 1, then check to make sure input is an integer
% any other value, perform no check
%
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/21/2001


function [iflag_assert,message] = assert_scalar( ...
   i_error,value,name,func_name, ...
   check_real,check_sign,check_int);


iflag_assert = 0;
message = 'false';


% First, set case values of check integer flags.
```

```
% check_real
i_real = 1;
i_imag = -1;

% check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

iflag_assert = 0;


% Check to make sure input is numerical and not a string.
if(~isnumeric(value))
   message = [ func_name, ': ', ...
            name, ' is not numeric'];
   iflag_assert = -1;
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end


% Check to see if it is a scalar.

if(max(size(value)) ~= 1)
   message = [ func_name, ': ', ...
            name, ' is not scalar'];
   iflag_assert = -2;
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end


% Then, check to see if it is real.

switch check_real;
```

```
case {i_real}
   if(~isreal(value))
      message = [ func_name, ': ', ...
                name, ' is not real'];
      iflag_assert = -3;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_imag}
   if(real(value))
      message = [ func_name, ': ', ...
                name, ' is not imaginary'];
      iflag_assert = -3;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

end


% Next, check sign.

switch check_sign;

case {i_pos}
   if(value <= 0)
      message = [ func_name, ': ', ...
                name, ' is not positive'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
```

```
   end

case {i_nonneg}
   if(value < 0)
      message = [ func_name, ': ', ...
                name, ' is not non-negative'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_neg}
   if(value >= 0)
      message = [ func_name, ': ', ...
                name, ' is not negative'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_nonpos}
   if(value > 0)
      message = [ func_name, ': ', ...
                name, ' is not non-positive'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_nonzero}
   if(value == 0)
      message = [ func_name, ': ', ...
                name, ' is not non-zero'];
```

```
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_zero}
   if(value ~= 0)
      message = [ func_name, ': ', ...
                name, ' is not zero'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

end
```

% Finally, check to make sure it is an integer.

```
if(check_int == 1)
   if(round(value) ~= value)
      message = [ func_name, ': ', ...
                name, ' is not an integer'];
      iflag_assert = -5;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end
end
```

% set flag for succesful passing of all checks

```
iflag_assert = 1;
```

```
message = 'true';

return;
```

% CSTR_ODE\assert_vector.m

```
%
% function [iflag_assert, message] = ...
%    assert_vector( ...
%    i_error,value,name,func_name,num_dim, ...
%    check_real,check_sign,check_int,check_column);
%
% This m-file contains logical checks to assert
% than an input value is a vector of a given type.
% This function is passed the value and name of
% the variable, the name of the function making the
% assertion, the dimension that the vector is
% supposed to be, and five integer flags
% that have the following usage :
%
% i_error : controls what to do if test fails
%       if i_error is non-zero, then use error()
%       MATLAB command to stop execution, otherwise
%       just return the appropriate negative number.
%       if i_error > 1, create file dump_error.mat
%       before calling error()
%
% check_real : check to examine whether input is real
% see table after function header for set
%   values of these case flags
% check_real = i_real (make sure that input is real)
% check_real = i_imag (make sure that input
%       is purely imaginary)
% any other value of check_real (esp. 0)
%       results in no check
%
%   check_real
%       i_real = 1;
%       i_imag = -1;
%
% check_sign : check to examine sign of input
% see table after function header for set
%   values of these case flags
% check_sign = i_pos (make sure input is positive)
% check_sign = i_nonneg (make sure input is non-negative)
% check_sign = i_neg (make sure input is negative)
% check_sign = i_nonpos (make sure input is non-positive)
% check_sign = i_nonzero (make sure input is non-zero)
% check_sign = i_zero (make sure input is zero)
```

```
% any other value of check_sign (esp. 0)
%       results in no check
%
%   check_sign
%       i_pos = 1;
%       i_nonneg = 2;
%       i_neg = -1;
%       i_nonpos = -2;
%       i_nonzero = 3;
%       i_zero = -3;
%
% check_int : check to see if input is an integer
% if = 1, then check to make sure input is an integer
% any other value, perform no check
%
% check_column : check to see if input is a
%       column or row vector
% check_column = i_column (make sure input is
%           column vector)
% check_column = i_row (make sure input is
%           row vector)
% any other value, perform no check
%
%   check_column
%       i_column = 1;
%       i_row = -1;
%
% if the dimension num_dim is set to zero, no
% check as to the dimension of the vector is made.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/21/2001



function [iflag_assert,message] = ...
    assert_vector( ...
    i_error,value,name,func_name,num_dim, ...
    check_real,check_sign,check_int,check_column);


% First, set case values of check integer flags.

% check_real
i_real = 1;
i_imag = -1;
```

```
% check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;

% check_column
i_column = 1;
i_row = -1;


iflag_assert = 0;
message = 'false';


% Check to make sure input is numerical and
% not a string.

if(~isnumeric(value))
   message = [ func_name, ': ', ...
            name, 'is not numeric'];
   iflag_assert = -1;
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end


% Check to see if it is a vector of the proper length.

num_rows = size(value,1);
num_columns = size(value,2);

% if it is a multidimensional array
if(length(size(value)) > 2)
   message = [ func_name, ': ', ...
            name, 'has too many subscripts'];
   iflag_assert = -2;
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
```

```
      return;
   end
end

% if both the number of rows and number of columns are
% not equal to 1, then value is a matrix instead
% of a vector.
if(and((num_rows ~= 1),(num_columns ~= 1)))
   message = [ func_name, ': ', ...
            name, 'is not a vector'];
   iflag_assert = -2;
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end

% if the dimension of the vector is incorrect
if(num_dim ~= 0)
   if(length(value) ~= num_dim)
      message = [ func_name, ': ', ...
               name, 'is not of the proper length'];
      iflag_assert = -2;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end
end


% check to make sure that the vector is of the
% correct type (e.g. column)

switch check_column;

case {i_column}
   % check to make sure that it is a column vector
   if(num_columns > 1)
      message = [ func_name, ': ', ...
               name, 'is not a column vector'];
      iflag_assert = -2;
      if(i_error ~= 0)
```

```
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end

case {i_row}
   if(num_rows > 1)
      message = [ func_name, ': ', ...
                  name, 'is not a row vector'];
      iflag_assert = -2;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

end



% Then, check to see if all elements are of
% the proper complex type.

switch check_real;

case {i_real}

   % if any element of value is not real
   if(any(~isreal(value)))
      message = [ func_name, ': ', ...
                  name, ' is not real'];
      iflag_assert = -3;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_imag}
```

```
   % if any element of value is not
   % purely imaginary
   if(any(real(value)))
      message = [ func_name, ': ', ...
                name, ' is not imaginary'];
      iflag_assert = -3;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

end


% Next, check sign.

switch check_sign;

case {i_pos}
   % if any element of value is not positive
   if(any(value <= 0))
      message = [ func_name, ': ', ...
                name, ' is not positive'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_nonneg}
   % if any element of value is negative
   if(any(value < 0))
      message = [ func_name, ': ', ...
                name, ' is not non-negative'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
```

```
      else
         return;
      end
   end

case {i_neg}
   % if any element of value is not negative
   if(any(value >= 0))
      message = [ func_name, ': ', ...
               name, ' is not negative'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_nonpos}
   % if any element of value is positive
   if(any(value > 0))
      message = [ func_name, ': ', ...
               name, ' is not non-positive'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_nonzero}
   % if any element of value is zero
   if(any(value == 0))
      message = [ func_name, ': ', ...
               name, 'is not non-zero'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
```

```
   end

case {i_zero}
   % if any element of value is non-zero
   if(any(value ~= 0))
      message = [ func_name, ': ', ...
                name, ' is not zero'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

end


% Finally, check to make sure it is an integer.

if(check_int == 1)
   if(any(round(value) ~= value))
      message = [ func_name, ': ', ...
                name, ' is not an integer'];
      iflag_assert = -5;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end
end


% set flag for succesful passing of all checks

iflag_assert = 1;
message = 'true';

return;
```

% CSTR_ODE\assert_matrix.m

```
%
% function [iflag_assert,message] = assert_matrix( ...
%    i_error,value,name,func_name, ...
%    num_rows,num_columns, ...
%    check_real,check_sign,check_int);
%
% This m-file contains logical checks to assert
% than an input value is a matrix of a given type.
% This function is passed the value and name of the
% variable, the name of the function making the
% assertion, the dimension that the matrix is supposed
% to be, and four integer flags that have the
% following usage :
%
% i_error : controls what to do if test fails
%       if i_error is non-zero, then use error()
%       MATLAB command to stop execution, otherwise
%       just return the appropriate negative number.
%       if i_error > 1, create file dump_error.mat
%       before calling error()
%
% check_real : check to examine whether input is real
% see table after function header for set values
%       of these case flags
% check_real = i_real (make sure that input is real)
% check_real = i_imag (make sure that input is
%       purely imaginary)
% any other value of check_real (esp. 0)
%       results in no check
%
%   check_real
%       i_real = 1;
%       i_imag = -1;
%
% check_sign : check to examine sign of input
% see table after function header for set
%       values of these case flags
% check_sign = i_pos (make sure input is positive)
% check_sign = i_nonneg (make sure input
%           is non-negative)
% check_sign = i_neg (make sure input is negative)
% check_sign = i_nonpos (make sure input
%           is non-positive)
% check_sign = i_nonzero (make sure input is non-zero)
% check_sign = i_zero (make sure input is zero)
% any other value of check_sign (esp. 0)
%           results in no check
%
%   check_sign
%       i_pos = 1;
%       i_nonneg = 2;
```

```
%       i_neg = -1;
%       i_nonpos = -2;
%       i_nonzero = 3;
%       i_zero = -3;
%
% check_int : check to see if input value
%           is an integer
% if = 1, then check to make sure input
%       is an integer
% any other value, perform no check
%
% if the dimensions num_rows or num_columns
% are set to zero, no check as to that
% dimension of the matrix is made.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/21/2001



function [iflag_assert,message] = assert_matrix( ...
   i_error,value,name,func_name, ...
   num_rows,num_columns, ...
   check_real,check_sign,check_int);


% First, set case values of check integer flags.

% check_real
i_real = 1;
i_imag = -1;

% check_sign
i_pos = 1;
i_nonneg = 2;
i_neg = -1;
i_nonpos = -2;
i_nonzero = 3;
i_zero = -3;


iflag_assert = 0;
message = 'false';


% Check to make sure input is numerical and
% not a string.
```

```
if(~isnumeric(value))
   message = [ func_name, ': ', ...
         name, ' is not numeric'];
   iflag_assert = -1;
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end


% Check to see if it is a matrix of
% the proper length.

% if it is a multidimensional array
if(length(size(value)) > 2)
   message = [ func_name, ': ', ...
         name, ' has too many subscripts'];
   iflag_assert = -2;
   if(i_error ~= 0)
      if(i_error > 1)
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end

% check that value has the proper number of rows
if(num_rows ~= 0)
   if(size(value,1) ~= num_rows)
      message = [ func_name, ': ', ...
            name, ' has the wrong number of rows'];
      iflag_assert = -2;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end
end
```

```
% check that value has the proper number of columns
if(num_columns ~= 0)
   if(size(value,2) ~= num_columns)
      message = [ func_name, ': ', ...
            name, ' has the wrong number of columns'];
      iflag_assert = -2;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end
end


% Then, check to see if all elements are of
% the proper complex type.

switch check_real;

case {i_real}

   % if any element of value is not real
   if(any(~isreal(value)))
      message = [ func_name, ': ', ...
            name, ' is not real'];
      iflag_assert = -3;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_imag}

   % if any element of value is not purely imaginary
   if(any(real(value)))
      message = [ func_name, ': ', ...
            name, ' is not imaginary'];
      iflag_assert = -3;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
```

```
            error(message);
        else
            return;
        end
    end

end


% Next, check sign.

switch check_sign;

case {i_pos}
    % if any element of value is not positive
    if(any(value <= 0))
        message = [ func_name, ': ', ...
                name, ' is not positive'];
        iflag_assert = -4;
        if(i_error ~= 0)
            if(i_error > 1)
                save dump_error.mat;
            end
            error(message);
        else
            return;
        end
    end

case {i_nonneg}
    % if any element of value is negative
    if(any(value < 0))
        message = [ func_name, ': ', ...
                name, ' is not non-negative'];
        iflag_assert = -4;
        if(i_error ~= 0)
            if(i_error > 1)
                save dump_error.mat;
            end
            error(message);
        else
            return;
        end
    end

case {i_neg}
    % if any element of value is not negative
    if(any(value >= 0))
        message = [ func_name, ': ', ...
                name, ' is not negative'];
        iflag_assert = -4;
```

```
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_nonpos}
   % if any element of value is positive
   if(any(value > 0))
      message = [ func_name, ': ', ...
            name, ' is not non-positive'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_nonzero}
   % if any element of value is zero
   if(any(value == 0))
      message = [ func_name, ': ', ...
               name, 'is not non-zero'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

case {i_zero}
   % if any element of value is non-zero
   if(any(value ~= 0))
      message = [ func_name, ': ', ...
               name, ' is not zero'];
      iflag_assert = -4;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
```

```
         end
         error(message);
      else
         return;
      end
   end

end


% Finally, check to make sure it is an integer.

if(check_int == 1)
   if(any(round(value) ~= value))
      message = [ func_name, ': ', ...
            name, ' is not an integer'];
      iflag_assert = -5;
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end
end


% set flag for succesful passing of all checks

iflag_assert = 1;
message = 'true';

return;
```

% CSTR_ODE\assert_structure.m

```
%
% function [iflag_assert,message] = assert_structure(...
%    i_error,Struct,struct_name,func_name,StructType);
%
% This MATLAB m-file performs assertions on a data
% structure. It makes use of assert_scalar,
% assert_vector, and assert_matrix for the
% fields.
%
% INPUT :
```

```
% =======
% i_error   controls what to do if test fails
%          if i_error is non-zero, then use error()
%          MATLAB command to stop execution, otherwise
%          just return the appropriate negative number.
%          if i_error > 1, then dump current state to
%          dump_error.mat before calling error().
% Struct    This is the structure to be
%          checked
% struct_name   the name of the structure
% func_name     the name of the function making the
%              assertion
% StructType    this is a structure that contains the typing
%              data for each field.
%   .num_fields is the total number of fields
%   Then, for i = 1,2, ..., StructType.num_fields, we have :
%   .field(i).name          the name of the field
%   .field(i).is_numeric    if non-zero, then field is numeric
%   .field(i).num_rows      # of rows in field
%   .field(i).num_columns   # of columns in field
%   .field(i).check_real    value of check_real passed to assertion
%   .field(i).check_sign    value of check_sign passed to assertion
%   .field(i).check_int     value of check_int passed to assertion
%
% OUTPUT :
% =======
% iflag_assert  an integer flag telling of outcome
% message       a message passed that describes
%              the result of making the assertion
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
%
% Version as of 7/25/2001



function [iflag_assert,message] = assert_structure(...
   i_error,Struct,struct_name,func_name,StructType);

iflag_assert = 0;
message = 'false';


% first, check to make sure Struct is a structure

if(~isstruct(Struct))
   iflag_assert = -1;
   message = [func_name, ': ', struct_name, ...
```

```
        ' is not a structure'];
   if(i_error ~= 0)
      if(i_error > 1);
         save dump_error.mat;
      end
      error(message);
   else
      return;
   end
end


% Now, for each field, perform the required assertion.

for ifield = 1:StructType.num_fields

   % set shortcut to current field type
   FieldType = StructType.field(ifield);

   % check if it exists in Struct
   if(~isfield(Struct,FieldType.name))
      iflag_assert = -2;
      message = [func_name, ': ', struct_name, ...
            '  does not contain ', FieldType.name];
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end
   end

   % extract value of field
   value = getfield(Struct,FieldType.name);

   % if the field is supposed to be numeric
   if(FieldType.is_numeric ~= 0)

      % check to make sure field is numeric
      if(~isnumeric(value))
         iflag_assert = -3;
         message = [func_name, ': ', ...
               struct_name, '.', FieldType.name, ...
               ' is not numeric'];
         if(i_error ~= 0)
            if(i_error > 1)
               save dump_error.mat;
            end
            error(message);
```

```
   else
      return;
   end
end


% decide which assertion statement to use based on
% array dimension of field value

% If both num_rows and num_columns are set equal
% to zero, then no check of the dimension of this
% field is made.

if(and((FieldType.num_rows == 0), ...
     (FieldType.num_columns == 0)))

   message = [func_name, ': ', ...
         struct_name,'.',FieldType.name, ...
         ' is not checked for dimension'];
   if(i_error ~= 0)
      disp(message);
   end


% else, peform check of dimension to make sure
% it is a scalar, vector, or matrix (i.e. a
% two dimensional array).

else

   % check that is is not a
   % multidimensional array
   if(length(size(value)) > 2)
      iflag_assert = -4;
      message = [func_name, ': ', ...
         struct_name,'.',FieldType.name, ...
         ' is multidimensional array'];
      if(i_error ~= 0)
         if(i_error > 1)
            save dump_error.mat;
         end
         error(message);
      else
         return;
      end

   % else if scalar
   elseif(and((FieldType.num_rows == 1), ...
           (FieldType.num_columns == 1)))
      assert_scalar(i_error,value, ...
         [struct_name,'.',FieldType.name], ...
```

```
            func_name,FieldType.check_real, ...
            FieldType.check_sign,FieldType.check_int);

         % else if a column vector
         elseif (and((FieldType.num_rows > 1), ...
                  (FieldType.num_columns == 1)))
            dim = FieldType.num_rows;
            check_column = 1;
            assert_vector(i_error,value, ...
               [struct_name,'.',FieldType.name], ...
               func_name,dim,FieldType.check_real, ...
               FieldType.check_sign,FieldType.check_int, ...
               check_column);

         % else if a row vector
         elseif (and((FieldType.num_rows == 1), ...
                  (FieldType.num_columns > 1)))
            dim = FieldType.num_columns;
            check_column = -1;
            assert_vector(i_error,value, ...
               [struct_name,'.',FieldType.name], ...
               func_name,dim,FieldType.check_real, ...
               FieldType.check_sign,FieldType.check_int, ...
               check_column);

         % otherwise, a matrix
         else
            assert_matrix(i_error,value, ...
               [struct_name,'.',FieldType.name], ...
               func_name, ...
               FieldType.num_rows,FieldType.num_columns, ...
               FieldType.check_real,FieldType.check_sign, ...
               FieldType.check_int);

         end % selection of assertion routine

      end    % if perform check of dimension

   end       % if (FieldType.is_numeric ~= 0)

end          % for loop over fields


% set return results for succesful assertion

iflag_assert = 1;
message = 'true';

return;
```