MATLAB Files for General CSTR Reactor Model

These program comprised by these files calculates the steady state concentrations and temperatures within a CSTR at steady state for a reaction network of arbitrary complexity. In addition to the files listed below, this program requires the following subroutines for the input and checking of data found in the MATLAB tutorial located on the 10.34 homepage :

get_input_scalar.m
assert_scalar.m
assert_vector.m
assert_matrix.m
assert_structure.m

File: CSTR_SS_F_vol_scan.m

This program calculates the steady state concentrations and temperature as a function of the volumetric flow rate through the reactor and makes the appropriate plots.

```
% CSTR_SS\CSTR_SS_F_vol_scan.m
%
% function iflag_main = CSTR_SS_F_vol_scan.m();
%
% This MATLAB program calculates the steady state concentration and
% temperature at steady state within a continuous stirred-tank reactor
% in overflow mode.  The model is written to support an arbitarily
% complex reaction network.
%
% First, a function is called that reads in the reactor geometry and
% reaction data from the keyboard.
%
% Then, program calls the built-in MATLAB nonlinear equation solver FSOLVE
% to solve the set of model equations for the unknown concentrations and
% temperature.
%
% This program repeats the calculation for multiple values of the inlet
% flow rate and plots the results.
%
% K. Beers
% MIT ChE
% 10/20/2001
```

```
function iflag_main = CSTR_SS_F_vol_scan();

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,iflag] = read_program_input;
   if(iflag <= 0)
      iflag_main = -1;
      error(['CSTR_SS: 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,iflag] = feval(file_name);

end


% We ask the user to input the range over which to plot the solution
% as a function of the volumetric flowrate.

check_real=1; check_sign=2; check_int=0;
prompt = 'Enter min. Reactor.F_vol for scan : ';
F_vol_min = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

check_real=1; check_sign=2; check_int=0;
prompt = 'Enter max. Reactor.F_vol for scan : ';
F_vol_max = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

num_grid = 50;

% Ask user whether to use logarithmic or linear scale.
check_real=1; check_sign=2; check_int=1;
prompt = 'Use linear (0) or logarithmic (1) x-axis for plot? : ';
i_use_log = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);

if(i_use_log)
   F_vol_grid = logspace(log10(F_vol_min),log10(F_vol_max),num_grid);
else
   F_vol_grid = linspace(F_vol_min,F_vol_max,num_grid);
end


% We begin our simulation at the maximum volumetric flowrate,
% because this gives the smallest conversion of product and
% thus the reactor concentrations and temperature are likely to
% be near those of the outlet.  This gives us a natural guess
% for the first simulation.
Guess = Inlet;


% We set the convergence parameters for the MATLAB nonlinear
% equation solver.
options = optimset('TolFun', 1e-8,'Display','off',...
   'LargeScale','off');


% Next, we allocate space to store the results of each
% calculation.

State_plot.conc = zeros(ProbDim.num_species,num_grid);
State_plot.Temp = linspace(0,0,num_grid);

for iter = 1:num_grid

   igrid = num_grid - iter + 1;
   Reactor.F_vol = F_vol_grid(igrid);


   % We now stack the guess data into the state vector.
   x_guess = stack_state(Guess,ProbDim.num_species);

   % Next, the MATLAB nonlinear equation solver is
   % invoked to solve for the unknown concentrations and
   % temperature.  If the calculation does not converge,
   % the user is asked to input a new guess.
   exitflag = 0;
   while(exitflag <= 0)
      [x_state,fval,exitflag,output] = fsolve(@CSTR_SS_calc_f,...
         x_guess,options, ...
         ProbDim,Reactor,Inlet,Physical,Rxn);
      if(exitflag > 0)
         break;
      end
      disp('CSTR_SS_F_vol_scan: fsolve did not converge');
      Reactor.F_vol, x_guess, x_state, fval,
      ichoose = ...
          input('Stop (0), retry with old result (1) with new guess (2) : ');
      if(~ichoose)
         error('CSTR_SS_F_vol_scan: Stopped calculation');
      elseif(ichoose==1)
         x_guess = x_state;
      else
         Guess.conc = input('Enter new guess of concentration vector : ');
         Guess.Temp = input('Enter new guess of temperature : ');
```

```
            x_guess = stack_state(Guess,ProbDim.num_species);
        end
    end


    % Unstack and store the results for later plotting.
    list_neg = find(x_state < 0);
    for count=1:length(list_neg)
        k = list_neg(count);
        x_state(k) = 0;
    end
    State = unstack_state(x_state,ProbDim.num_species);

    State_plot.conc(:,igrid) = State.conc;
    State_plot.Temp(igrid) = State.Temp;

    % Use the results of this calculation as an estimate of the
    % solution for the next value of the flowrate.
    Guess = State;

end


% Now, make plots of each concentration and temperature as a function
% of the volumetric flowrate through the reactor.

for ispecies = 1:ProbDim.num_species
    figure;
    if(i_use_log)
        semilogx(F_vol_grid,State_plot.conc(ispecies,:));
    else
        plot(F_vol_grid,State_plot.conc(ispecies,:));
    end
    title(['CSTR model : Effect of F_{vol} on conc. of species # ', ...
        int2str(ispecies)]);
    xlabel('F_{vol} = Volumetric Flowrate');
    ylabel(['c(', int2str(ispecies), ')']);
end

figure;
if(i_use_log)
    semilogx(F_vol_grid,State_plot.Temp);
else
    plot(F_vol_grid,State_plot.Temp);
end
title('CSTR model : Effect of F_{vol} on temperature');
xlabel('F_{vol} = Volumetric Flowrate');
ylabel('T');
```

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

iflag_main = 1;

return;
```

File: CSTR_SS.m

This program calculates the steady state concentrations and temperature for a single simulation.

```
% CSTR_SS\CSTR_SS.m
%
% function iflag_main = CSTR_SS.m();
%
% This MATLAB program calculates the steady state concentration and
% temperature at steady state within a continuous stirred-tank reactor
% in overflow mode.  The model is written to support an arbitarily
% complex reaction network.
%
% First, a function is called that reads in the reactor geometry and
% reaction data from the keyboard.
%
% Then, program calls the built-in MATLAB nonlinear equation solver FSOLVE
% to solve the set of model equations for the unknown concentrations and
% temperature.
%
% K. Beers
% MIT ChE
% 10/20/2001


function iflag_main = CSTR_SS();


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,iflag] = read_program_input;
   if(iflag <= 0)
      iflag_main = -1;
      error(['CSTR_SS: 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,iflag] = feval(file_name);
```

```
end



% Next, we set the solver parameters for the MATLAB nonlinear
% equation solver.

% First, the user is asked to input the initial guess of the solution.

disp(' ');
disp('Input the guess of the solution : ');

Guess.conc = linspace(0,0,ProbDim.num_species)';

for ispecies = 1:ProbDim.num_species
   check_real=1; check_sign=1; check_int=0;
   prompt = ['Guess concentration of species # ', ...
         int2str(ispecies), ' : '];
   Guess.conc(ispecies) = get_input_scalar(prompt, ...
      check_real,check_sign,check_int);
end

check_real=1; check_sign=1; check_int=0;
prompt = 'Guess reactor temperature : ';
Guess.Temp = get_input_scalar(prompt, ...
   check_real,check_sign,check_int);


% We now stack the guess data into the state vector.

x_guess = stack_state(Guess,ProbDim.num_species);


% Next, the MATLAB nonlinear equation solver is invoked to solve for the
% unknown concentrations and temperature.

options = optimset('TolFun', 1e-8,'Display','off',...
   'LargeScale','off');
[x_state,fval,exitflag,output] = ...
    fsolve(@CSTR_SS_calc_f,x_guess,options, ...
   ProbDim,Reactor,Inlet,Physical,Rxn);
if(exitflag <=0)
   iflag_main = -2;
   error(['CSTR_SS: fsolve exited with error = ', ...
         int2str(exitflag)]);
end

% Unstack and print the results.

State = unstack_state(x_state,ProbDim.num_species);
```

```
disp(' ');
disp(' ');
disp('Calculated concentrations : ');
disp(State.conc);

disp('Calculated temperature : ');
disp(State.Temp);


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

iflag_main = 1;

return;
```

File: CSTR_SS_input4.m

This is an example input file for the simulation.

```
% CSTR_SS\CSTR_SS_input4.m
%
% This input file sets parameters for a steady state
% calculation of a single CSTR using the program CSTR_SS.m
%
% K. Beers
% MIT ChE
% 10/20/2001


function [ProbDim,Reactor,Inlet,Physical,Rxn,iflag] = ...
   CSTR_SS_input1();

iflag = 0;


% Set reactor data
radius = 5;
height = 10;
Reactor.volume = height*pi*radius^2;
Reactor.F_vol = 20;
Reactor.F_cool = 1000;
Reactor.T_cool = 1;
Reactor.U_HT = 1000;
Reactor.A_HT = 2*pi*radius*height;
Reactor.Cp_cool = 1;

% Set number of species
ProbDim.num_species = 4;

% Set inlet concentrations and temperature
Inlet.conc = [1; 2; 0; 0];
Inlet.Temp = 1;

% Set heat capacity data
Physical.Cp_data(1,:) = [1 0 0 0];
Physical.Cp_data(2,:) = [1 0 0 0];
Physical.Cp_data(3,:) = [1 0 0 0];
Physical.Cp_data(4,:) = [1 0 0 0];

% Set number of reactions
ProbDim.num_rxn = 2;

% Allocate storage for 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)';

% Set reaction # 1 data
irxn = 1;
Rxn.stoich_coeff(irxn,:) = [-1 -2 1 0];
Rxn.is_rxn_elementary(irxn) = 0;
Rxn.ratelaw_exp(irxn,:) = [1 2 0 0];
Rxn.k_ref(irxn) = 1;
Rxn.T_ref(irxn) = 1;
Rxn.E_activ(irxn) = 1;
Rxn.delta_H(irxn) = -1;

% Set reaction # 2 data
irxn = 2;
Rxn.stoich_coeff(irxn,:) = [0 -1 -1 1];
Rxn.is_rxn_elementary(irxn) = 0;
Rxn.ratelaw_exp(irxn,:) = [0 1 1 0];
Rxn.k_ref(irxn) = 0.1;
Rxn.T_ref(irxn) = 1;
Rxn.E_activ(irxn) = 10;
Rxn.delta_H(irxn) = -1;


iflag = 1;

return;
```

File: read_program_input.m

```
% CSTR_SS\read_program_input.m
%
% function [ProbDim,Reactor,Inlet,Physical,Rxn,iflag] = ...
%    read_program_input();
%
% This procedure reads in the simulation parameters that are
% required to define a single CSTR simulation.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 10/20/2001
%
% Version as of 10/20/2001



function [ProbDim,Reactor,Inlet,Physical,Rxn,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);

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);



% 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

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

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

for ispecies = 1:ProbDim.num_species

   % Inlet.conc_in(ispecies)
   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);

end

disp(' ');

% Inlet.Temp_in
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);
```

```
% PHYSICAL DATA ------------------------------------------
% PDL> Input physical data :
%           Physical.Cp_data(num_species,4)

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

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


% 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



iflag = 1;

return;
```

File: stack_state.m

```
% CSTR_SS\stack_state.m
%
% function [x_state,iflag] = stack_state(State, num_species);
%
% 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 10/20/2001



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

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 = 2;
% .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
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;
% 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+1;
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

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

```
iflag = 1;

return;
```

File: unstack_state.m

```
% CSTR_SS\unstack_state.m
%
% function [State,iflag] = unstack_state(x_state, num_species);
%
% 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 10/20/2001



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

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 + 1;

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)';
State.Temp = 0;


%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

State.Temp = x_state(pos_counter+1);

iflag = 1;

return;
```

File: CSTR_SS_calc_f.m

```
% CSTR_SS_calc_f.m
%
% This MATLAB file calculates the vector of function residuals that is
% zero for a solution to the set of equations that models a steady-state
% CSTR with an arbitrarily complex reaction network.
%
% K. Beers
% MIT ChE
% 10/20/2001

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

iflag = 0;

num_DOF = ProbDim.num_species + 1;
fval = linspace(0,0,num_DOF);


% Next, unstack the state vector.
State = unstack_state(x_state,ProbDim.num_species);


% Next, calculate the heat capacities for the inlet and
% reactor temperatures for each species.

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);
```

```
% 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_SS_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
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;


iflag = 1;

return;
```

File: reaction_network_model.m

```
% CSTR_SS\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;
```