function imain_flag = TR_1D_model1_SS();

This program calculates the steady state concentration and temperature profiles in a 1-D tubular reactor for an arbitrary number of species and an arbitrary reaction network. The reaction network is specified by the stoichiometric coefficients and the exponential powers to which the concentrations of each species are raised in the rate laws. The effective diffusivities for each species and the density and heat capacity of the medium are assumed to be constant. The heats of reaction are likewise assumed constant, and the temperature dependence of each rate constant is specified by the value of the rate constant at a reference temperature and a constant activation energy. The heat transfer coefficient for the cooling jacket is assumed constant. Dankwert’s boundary conditions are applied at the inlet and outlet. A constant superficial velocity, obtained from knowledge of the reactor dimensions and volumetric flow rate, is used to quantify the convective contribution to the fluxes of each species’ concentration and the enthalpy.

% PROGRAM INPUT/OUTPUT DATA
% =========================
% problem_dimension_data (struct ProbDim)
% -----------------------------
% .num_species IN INT
% the number of species
% .num_rxn IN INT
% the number of reactions
% reactor_data (struct Reactor)
% -----------------------------
% .len IN REAL
% the length of the tubular reactor
% .dia IN REAL
% the diameter of the tubular reactor
% .Qflow IN REAL
% the volumetric flow rate through the reactor. Along with the dimensions of the reactor, it defines the superficial velocity used in the convective terms of the species and enthalpy balances.
% .Temp_cool IN REAL
% the temperature of the reactor coolant jacket
% .U_HT IN REAL
% the overall heat transfer coefficient of the reactor
% .conc_in IN REAL(ProbDim.num_species)
% the concentrations of each species in the reactor inlet
% .Temp_in IN REAL
% the temperature of the reactor inlet
% .volume PROG REAL
% the volume of the reactor
% .cross_area PROG REAL
% the cross sectional area of the reactor
% .surf_area PROG REAL
% the surface area of the reactor available for heat transfer to the cooling jacket
% .velocity PROG REAL
% the superficial velocity in the reactor that is included in the convective flux terms
%
% physical_data (struct Physical)
% -----------------------------------------------
% .diffusivity IN REAL(num_species)
% the constant diffusivities of each species
% .density IN REAL
% the constant density of the medium
% .Cp IN REAL
% the constant heat capacity of the medium
% .thermal_conduct IN REAL
% the constant thermal conductivity of the medium
% .thermal_diff PROG REAL
% the constant thermal diffusivity of the medium
%
% rxn_data (struct Rxn)
% ----------------------
% .stoich_coeff IN
REAL(ProbDim.num_rxn,ProbDim.num_species)
the stoichiometric coefficients
possibly fractional) of each
species in each reaction.
.ratelaw_exp IN REAL(ProbDim.num_rxn,ProbDim.num_species)
the exponential power (possibly fractional)
to which the concentration of each species
is raised each reaction’s rate law.
is_rxn_elementary IN INT(ProbDim.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 IN REAL(ProbDim.num_rxn)
the rate constants of each reaction at a
specified reference temperature
.T_ref IN REAL(ProbDim.num_rxn)
This is the value of the reference
temperature used to specify the
temperature dependence of each
rate constant.
.E_activ IN REAL(ProbDim.num_rxn)
the constant activation energies of
each reaction divided by the value
of the ideal gas constant
.delta_H IN REAL(num_rxn)
the constant heats of reaction

PROGRAM IMPLEMENTATION NOTES
===============

Section 1. Method of discretizing PDE's:
To discretize the partial differential equations that describe the balances on the species concentrations and the enthalpy, use the method of finite differences. To avoid spurious oscillations when convection dominates and the local Peclet number is greater than two, use upwind differencing. Implement the finite difference procedure so that the grid point spacing may be non-uniform.

% grid_data (struct Grid)
% -----------------------
% .num_pts PIN INT
% the number of grid points in the axial direction
% .z POUT REAL(Grid.num_pts)
% the values of the z-coordinate at the grid points

% state_data (struct State)
% ------------------------
% .conc POUT REAL(Grid.num_pts,ProbDim.num_species)
% the values of the species' concentrations at grid points
% .Temp POUT REAL(Grid.num_pts)
% the values of the temperature at each grid point

% Section 2. Method of solving for the steady state profiles :
% ---------------------------------------------------------------

% To solve for the steady-state profiles, we will use a robust two-step procedure. We will initially assume that the inlet conditions hold uniformly throughout the reactor. As this is likely to be far from the true solution, we will first perform a number of implicit Euler time integration steps to get within the vicinity of the stable steady state solution. The time integration will proceed until a maximum number of time steps have been performed or until the norm of the time derivative vector falls below a specified value. If the time derivative has become sufficiently small, we will switch to Newton's method with a weak-line search to aid global convergence.
If one wishes to use only Newton's method to solve for the steady state profile (for example to find an unstable steady state), then Solver.max_iter_time is set to 0. Otherwise, if the maximum number of time integration steps has been performed and the time derivative is still too large, the program exits without performing any Newton's method iterations.

A restart utility will be added so that if convergence is not achieved, executing the program again will start from the previously saved results. Upon a restart, new time step and convergence tolerances are input.

At each time or Newton's method iteration, the values of the concentration and temperatures at each grid point will be constrained to be non-negative.

iflag_restart PIN INT
This integer flag indicates whether the simulation is a restart of a previous simulation, in which only new convergence parameters need be input, or is an initial simulation in which all system parameters must be input. If iflag_restart is non-zero, then it is a restart, if 0 then it is an initial simulation.

imain_flag POUT INT
This integer flag signifies whether the solution method has converged. A positive value signifies that convergence to the steady state value has been attained. A negative value indicates some error.

solver_data (struct Solver)

.max_iter_time PIN INT
the maximum number of implicit Euler time steps.
If =0, then no time simulation is performed and the solver goes immediately to Newton's method
.dt PIN REAL
the time step to be used in the implicit Euler simulation
.atol_time PIN REAL
the norm of the function (time derivative) vector at which the time integration procedure is deemed to have been sufficiently converged
.max_iter_Newton PIN INT
% the maximum number of Newton's method iterations
% .atol_Newton PIN REAL
% the norm of the function (time derivative) vector
% at which convergence to the steady state solution is
% deemed to have been achieved
% .iflag_Adepend PROG INT
% if this integer flag is non-zero, then the A matrix
% is assumed to be state-dependent and so must be
% recalculated at every iteration
% .iflag_nonneg PROG INT
% if this integer flag is non-zero, then the elements
% of the state vector are enforced to be non-negative
% at every iteration
% .iflag_verbose PROG INT
% if this integer flag is non-zero, then the solver
% routine is instructed to print to the screen the
% progress of the solution process; otherwise, it
% runs silent
%
% Interaction with Section 1. Method of discretizing PDE's:
%
% Each time that the program runs, the solver will overwrite the
% value of the concentration and temperature profiles. It could
% be that too large of a time step is used or that Newton's method
% has a problem converging, so that the quality of the solution
% is poorer than it was before the solver was called. The next
% restart should therefore start from the old, better solution
% and not necessarily the most recent. To guard against this,
% if the output solution estimate appears farther from steady
% state than the input estimate, a warning message will be
% returned and two separate output files will be created. The
% results of the solver will be written to the standard output
% file, but a second file will be written that retains the initial
% results. If these previous results are to be used in a
% subsequent restart, the user copies this file to the name of
% the standard output file before running again. User discretion
% is required in this case, because the dynamics of some systems
% have an induction period. In this case, the magnitude of the
% time derivative vector will naturally increase in the course
% of approaching the stable steady state.
%
% Kenneth Beers
% Massachusetts Institute of Technology
% Department of Chemical Engineering
% 7/2/2001
function imain_flag = TR_1D_model1_SS();

    func_name = 'TR_1D_model1_SS';

    imain_flag = 0;

    % This integer flag controls what to do if an assertion fails.
    % See assertion routines for meaning.
    i_error = 2;

    % PDL> Ask if it is a restart, read answer to iflag_restart

    disp('Starting TR_1D_model1_SS');
    iflag_restart = input('Is this a restart? (0=no, 1=yes) : '); 
    check_real=1; check_sign=2; check_int=1; 
    assert_scalar(i_error,iflag_restart,'iflag_restart',... 
    func_name,check_real,check_sign,check_int);

    % PDL> IF it is not a restart, THEN

    if(iflag_restart == 0)

        % PROCEDURE: read_program_input
        % PDL> Read in the program input data (intent IN)
        % PDL> Among PIN data, read grid_data:num_pts
        % ENDPROCEDURE

        disp('Reading program input ...');

        [ProbDim,Reactor,Physical,Rxn,Grid,iflag_func] = ...
            read_program_input;
        if(iflag_func <= 0)
            imain_flag = -1;
            if(i_error > 1)
                save dump_error.mat;
            end
            error([func_name, ' : ', ...
                'Error (', int2str(iflag_func), ') ', ...]
% PROCEDURE: set_grid_1D
% PDL> Specify the locations of the grid points in z_grid.
% For the moment, simply use a uniform grid, although
% write the rest of the program to be compatible with
% the use of a non-uniform grid
% ENDPROCEDURE

disp('Setting grid ...');
[Grid.z,iflag_func] = set_grid_1D(Grid.num_pts,Reactor.len);
if(iflag_func <= 0)
    imain_flag = -2;
    if(i_error > 1)
        save dump_error.mat;
        end
        error([func_name, ': ', ...
            'Error (', int2str(iflag_func), ')', ...
            'returned from set_grid_1D']);
    end
end

% PDL> Initialize the concentration and temperature profiles
% by setting them to be uniformly equal to the inlet
% conditions.

State.conc = zeros(Grid.num_pts,ProbDim.num_species);
for ispecies = 1:ProbDim.num_species
    State.conc(:,ispecies) = Reactor.conc_in(ispecies);
end

State.Temp = linspace(...
    Reactor.Temp_in,Reactor.Temp_in,Grid.num_pts);

% PDL> ELSE IF NOT a restart THEN

else

% PDL> Read in the file TR_1D_model1_SS.mat

disp('Reading file TR_1D_model1_SS.mat');
load TR_1D_model1_SS.mat;
% PDL> ENDIF

end

% PROCEDURE: read_solver_input
% PDL> Input the values of the PIN variables that control
% the solver operation
% ENDPROCEDURE

[Solver,iflag_func] = read_solver_input;
if(iflag_func <= 0)
    imain_flag = -3;
    if(i_error > 1)
        save dump_error.mat;
        error(['func_name', ': ', ...
            'Error (', int2str(iflag_func), ') ', ...
            'returned from read_solver_input']);
    end
end

% Save the initial concentration and temperature
% profiles in back-up variables for possible later
% use in a restart in case the solver behaves badly.

State_init = State;

% PROCEDURE: TR_1D_model1_SS_solver
% PDL> Call the solver to update the estimate
% of the solution vector
% ENDPROCEDURE

[State,iflag_converge,f,f_init] = ...
    TR_1D_model1_SS_solver(State_init, ... Solver,ProbDim,Reactor,Physical,Rxn,Grid);

% Write the results of the simulation to
% the file TR_1D_model1_SS.mat

save TR_1D_model1_SS.mat;
% PDL> CASE : Select course of action based on
% value of iflag_converge returned from
% steady state solver

switch iflag_converge;

% PDL> IF iflag_converge IS 0,
% signifying no convergence

case {0}

% PDL> Set integer flag of main program,
% imain_flag to 0

  imain_flag = 0;

% PDL> If the norm of the function (time derivative)
% vector is greater after the solver operation
% than it was before, set the return value of
% imain_flag to indicate this. Then, write the
% old profiles to the file
% TR_1D_model1_SS_backup.mat and set
% imain_flag as indicator

  norm_f_init = max(abs(f_init));
  norm_f = max(abs(f));

  if(norm_f > norm_f_init)
    disp('');
    disp(['Final estimate had larger error ',...
          'than initial estimate']);
    imain_flag = -4;
    State = State_init;
    clear State_init;
    save TR_1D_model1_SS_backup.mat;
  end

% PDL> IF iflag_converge IS 1, signfying convergence
% PDL> Print convergence message and set
% imain_flag to 1
case {1}
  imain_flag = 1;
  disp(' ');
  disp('Solver converged');

  if iflag_converge IS negative, signifying error
  PDL> Print error message and set imain_flag to -1
  otherwise

  imain_flag = -5;
  disp(['Error encountered with iflag_converge = ', ...
       'int2str(iflag_converge)));

  PDL> ENDCASE

end

% PDL> Make plots of the solver output results

plot_results(ProbDim.num_species,Grid,State);

return;