Short Tutorial on Matlab

Part 5. Using S-function blocks in Simulink®

I. Motivation: With the complexity of medium-size to large-size nonlinear models, it may be more efficient to use a set of differential equations written in an m-file. These m-files will be accessed by Simulink through the S-function block. Thus, this method mixes the advantages of an m-file which can be run directly by solvers such as **`ode45`**, with the graphical links to other Simulink blocks.

II. Example System:

Suppose we want to model the nonisothermal CSTR,

$$\frac{dC_a}{dt} = \left(\frac{F}{V}\right)\cdot\left(C_{af} - C_a\right) - k_0\cdot\exp\left[-\frac{E_a}{R\cdot(T+460)}\right]\cdot C_a$$

$$\frac{dT}{dt} = \left(\frac{F}{V}\right)\cdot\left(T_f - T\right) - \frac{\Delta H}{\rho\cdot C_p}\cdot\left[k_0\cdot\exp\left[-\frac{E_a}{R\cdot(T+460)}\right]\cdot C_a\right] - \left(\frac{U\cdot A}{\rho\cdot C_p\cdot V}\right)\cdot\left(T - T_j\right)$$

We want to model this system in which we will treat the jacket temperature, T_j, as the input (i.e. manipulated variable). We will also want to monitor concentration and temperature of the liquid in the CSTR as our outputs.

III. Write the m-file.

Recall that we could model the process by writing an m-file to be used by Matlab solvers such as **`ode45`**. One such file, which we will name as **`reactor.m`**, is shown in Figure 1.

Test the model to make sure it works. For instance, with T_j=55 :

```
>> [t,x]=ode45(@reactor,[0 10],[0.1;40],[],55);
```

Note/Recall: The command-line specifies: a simulation-time span of **`[0 10]`**, an initial-value column vector: **`[0.1;40]`**, a null placeholder,**`[]`** , for default options, and setting T_j with a value equal to 55.

```
function    dx = reactor(t,x,Tj)
%
%   model for reactor
%

    Ca  = x(1)          ;   % lbmol/ft^3
    T   = x(2)          ;   % oF

    Ea  = 32400         ;   % BTU/lbmol
    k0  = 15e12         ;   % hr^-1
    dH  = -45000        ;   % BTU/lbmol

    U   = 75            ;   % BTU/hr-ft^2-oF
    rhocp = 53.25       ;   % BTU/ft^3
    R   = 1.987         ;   % BTU/lbmol-oF
    V   = 750           ;   % ft^3
    F   = 3000          ;   % ft^3/hr
    Caf = 0.132         ;   % lbmol/ft^3
    Tf  = 60            ;   % oF

    A = 1221            ;   % ft^2

    ra  = k0*exp(-Ea/(R*(T+460)))*Ca;
    dCa = (F/V)*(Caf-Ca)-ra;
    dT  = (F/V)*(Tf-T)-(dH)/(rhocp)*ra...
            -(U*A)/(rhocp*V)*(T-Tj);

    dx =[dCa;dT];
```

Figure 1. File saved as **`reactor.m`**

Remarks:

1. We treat T_j as an argument/parameter. This is in anticipation that we will be varying T_j later as an input/manipulated variable.
2. The arguments **`x`** and **`dx`** are column vectors for state and derivative, respectively.
3. Writing a model first for direct ODE45 implementation is advisable, specially for complex processes. This way, one can check the validity of the model, prior to its incorporation to a Simulink model.

IV. Write an S-function file.

This file will also be saved as an m-file. It contains the protocol in which Simulink can access information from Matlab.

For our example, we show one such S-function file in Figure 2. We will save this file as **`reactor_sfcn.m`**.

```
function  [sys,x0,str,ts]=...
            reactor_sfcn(t,x,u,flag,Cinit,Tinit)
```

```
switch flag

    case 0  % initialize

        str=[]                      ;
        ts = [0 0]                  ;

        s = simsizes                ;

            s.NumContStates = 2     ;
            s.NumDiscStates = 0     ;
            s.NumOutputs = 2        ;
            s.NumInputs = 1         ;
            s.DirFeedthrough = 0    ;
            s.NumSampleTimes = 1    ;

        sys = simsizes(s)           ;

        x0 = [Cinit, Tinit]         ;

    case 1                          % derivatives

        Tj  = u                     ;
        sys = reactor(t,x,Tj)       ;

    case 3  % output

        sys = x;

    case {2 4 9}                    % 2:discrete
                                    % 4:calcTimeHit
                                    % 9:termination
        sys =[];

    otherwise

        error(['unhandled flag =',num2str(flag)])   ;

end
```

Figure 2. File saved as **reactor_sfcn.m**.

Let us deconstruct the S-function file given in Figure 2 to understand what the file needs to contain.

1. The first line specifies the input and output arguments.

```
function  [sys,x0,str,ts]=...
             reactor_sfcn(t,x,u,flag,Cinit,Tinit)
```

As it is with any Matlab functions, the variable names themselves are not as crucial as the positions of the variables in the list.

a) input arguments

(1) **`t`** - the time variable
(2) **`x`** - the column-vector of state variables
(3) **`u`** - the column-vector of input variables (whose value will come from other Simulink blocks)
(4) **`flag`** - indicator of which group of information and/or calculations is being requested by Simulink.

There are six types of request that Simulink performs, each of which is designated by an integer number:

`flag` value	**Job/Data Request**
0	***Initialization:*** a) Setup of input/output vector sizes and other setup modes b) Specification/calculation of initial conditions for the state variables.
1	***Derivative Equation Updating:*** a) Calculations involving input vectors b) Calculation of the derivatives
2	***Discrete Equation Updating*** (will not be used for our example)
3	***Output Calculations:*** Evaluating output variables as a function of the elements of the state vector (and in some case, also the elements of the input vector)
4	***Get Time of Next Variable Hit*** (will not be used for our example)
9	***Termination:*** Additional routines/calculations at the end of the simulation run. (will not be used for our example)

(5) **`Cinit,Tinit`** - additional supplied parameters.

In our case, these are the initial conditions for concentration and temperature.

Note: We do not specify what the values of the input arguments are. Their values will be specified by Simulink during a simulation run.

b) output arguments

(1) **`sys`** - the main vector of results requested by Simulink. Depending on the **`flag`** sent by Simulink, this vector will hold different information.

If **flag** = 0	**sys = [a,b,c,d,e,f,g]** where, **a =** number of continuous time states **b =** number of discrete time states **c =** number of outputs (*Note: this is not necessarily the number of states*) **d =** number of inputs **e =** 0 (*required to be* 0*, not currently used*) **f** = 0(no) or 1(yes) for direct algebraic feed through of input to output. (*this is relevant only if during flag=3, the output variables depend algebraically on the input variables*.) **g** = number of sample times. (*for continuous process, we set this equal to* 1)
If **flag** = 1	**sys =** a column vector of the derivatives of the state variables
If **flag** = 3	**sys =** a column vector of the output variables
If **flag** = 2,4,9	since these flags are not used in our example, they can just send out a null vector: **sys=[]**

The next set of 3 output arguments are needed by Simulink only when **flag** = 0, otherwise they are ignored:

(2) **x0** - column vector of initial conditions.

(3) **str** - need to be set to null. This is reserved for use in future versions of Simulink.

(4) **ts** - an array of two columns to specify sampling time and time offsets. Since our example will deal only with continuous systems, this will be set to **[0 0]** during initiation phase.

2. After the first line, the S-function file is split into the different cases determined by **`flag`**. As shown in Figure 3, we show the bare structure of the "containers" for the different cases. We have left out the details for case 1, 2 and 3. For case 2, 4, and 9, we simply set **`sys=[]`**. The last two lines to catch an exceptional case where a bug occurs during the Simulink run.

```
switch flag

    case 0                              % initialize

      %...

    case 1                              % derivatives

      %...

    case 3                              % output

      %...

    case {2 4 9}                        % 2:discrete
                                        % 4:calcTimeHit
                                        % 9:termination
        sys =[];

    otherwise

        error(['unhandled flag =',num2str(flag)])   ;

end
```

Figure 3.

Now, let us fill the details.

For case 0 (initialization),
a) define **`str`**, **`ts`** and **`x0`**

```
        str=[]                          ;
        ts = [0 0]                      ;
        x0 = [Cinit, Tinit]             ;
```

b) create a row vector which specifies the number of inputs and outputs, etc.
To aid in this, we invoke the **`simsizes`** command.
Without arguments, **`simsizes`** will creates a strucure variable which we can then fill with the required values:

```
 s = simsizes                           ;
```

```
s.NumContStates  = 2   ;
s.NumDiscStates  = 0   ;
s.NumOutputs     = 2   ;
s.NumInputs      = 1   ;
s.DirFeedthrough = 0   ;
s.NumSampleTimes = 1   ;
```

Using the command **simsizes** again with the structure variable as the argument actually translates the values in the structure, **`s`**, into a row vector which gets sent to Simulink via **`sys`**:

```
sys = simsizes(s)          ;
```

For case 1 (derivative calculations)

We set the input **`u`** to **`Tj`** and then apply it to the m-file we wrote earlier, i.e. **`reactor.m`**:

```
case 1                          % derivatives

   Tj  = u                      ;
   sys = reactor(t,x,Tj)        ;
```

For case 3 (output calculations)

```
case 3                          % output

   sys = x;
```

V. Insert the S-Function block into the Simulink.

In the Simulink Library browser, go to the [**`User-Define Functions`**] subdirectory. Then drag-drop the **`S-Function`** block (see Figure 4).

Double-click on the S-function block and fill in the parameters. Change the S-function name to **`reactor_sfcn`**. Also, fill in the parameters. In our case, we input **`0.1,40`** (which is the value for **`Cinit`** and **`Tinit`**) as shown in Figure 5.

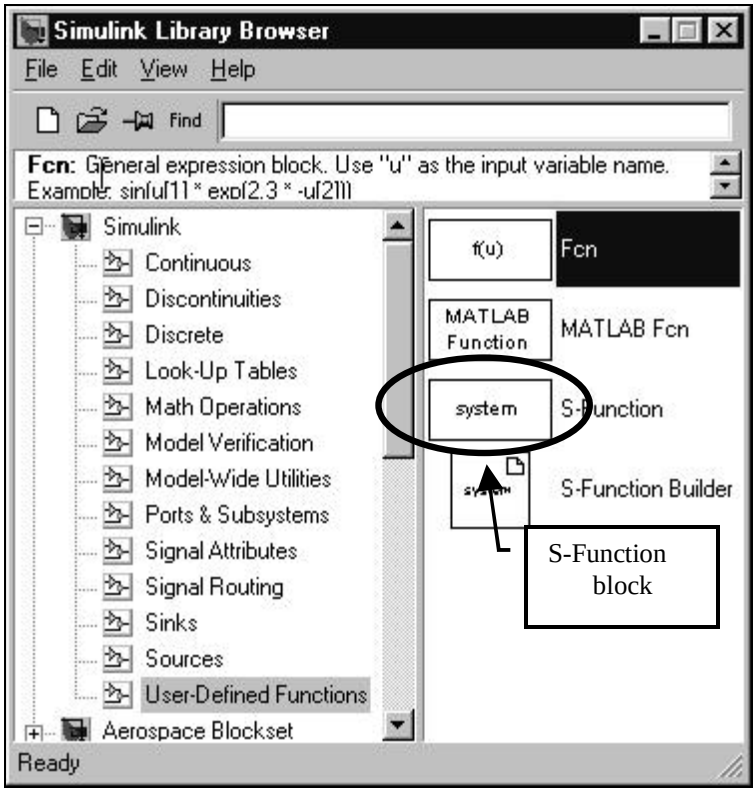

Figure 4.

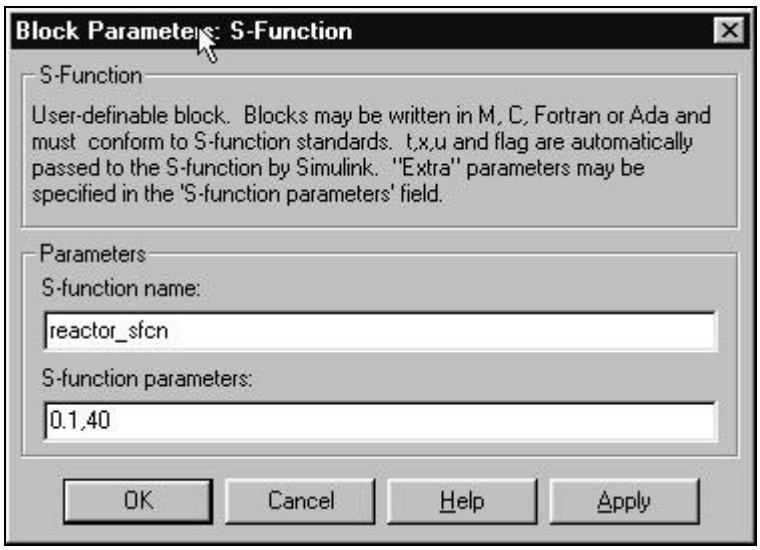

Figure 5.

VI. Add other Simulink blocks and simulate.

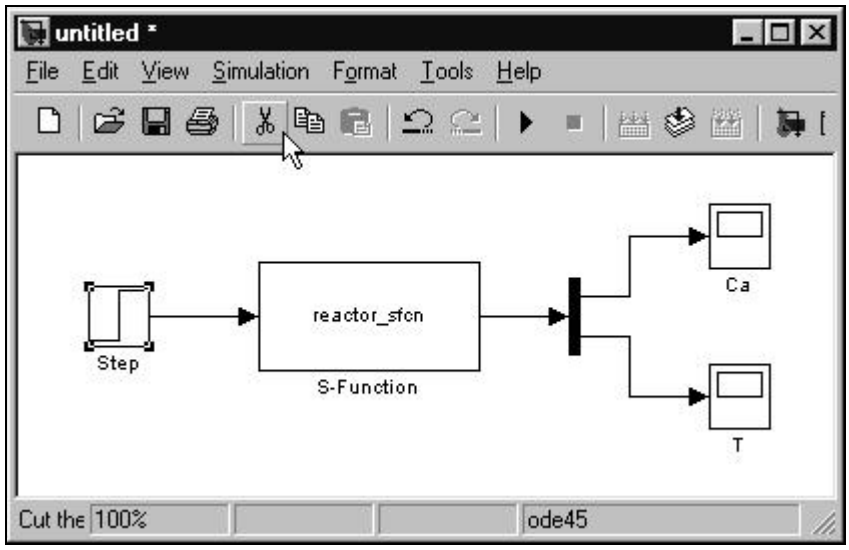

Figure 6.

Remark: In figure 6, we include a **`demux`** block (which stands for demultiplexer) to split the output vector to the 2 elements. In other applications where the input vectors has more than one element, we need a **`mux`** block (which stands for multiplexer). Both **`mux`** and **`demux`** blocks reside in the **`Signal Routing`** subdirectory of the Simulink Library browser.