MATLAB PoSB Package Companion:

Overview of tool

In this companion text, we will use the MATLAB PoSB Package to model and simulate our various parts and systems.

Syntax and semantics

In order to construct a system model and simulate it, you first need to instantiate a Matlab object that will store the system:

```
a_system = BioSystem();
```

Then, you create a set of compositors:

```
a_compositor = Compositor('compositor_name', initial_value);
```

and add each to the system model:

```
a_system.AddCompositor(a_compositor);
```

Next, you create parts. A general part definition has the following format (where "..." signifies a continuation of the same MATLAB program line and "…" signifies a set of similar objects):

```
a_part = ...
   Part('mechanism_name', …
        [State_Variable_1 State_Variable_2 … State_Variable_X], ...
        [Rate('rate equation for State Variable 1'), ...
         Rate('rate equation for State Variable 2'), ...
         Rate('rate equation for State Variable X')]);
```

We use the following notation to refer to a state variable's value within a rate equation definition:

```
val(a_state_variable)
```

Each part is then added to the system model:

```
a_system.AddPart(a_part);
```

and then you are ready to run a simulation of the system:

```
[T, Y] = a_system.run(tspan);
```

Information from the simulation will be stored in T and Y, where T contains specific time values within the simulation and Y contains values of the state variables of each of these time values. The simulation time is provided by the input variable `tspan`.

Chapter 3 examples

A Simple Enzymatic Reaction

Here we consider a MATLAB implementation of the simple enzymatic reaction from Chapter 3. First, we define our system:

```
sys1 = BioSystem();
```

Then, we define the compositors and add them to our system. Remember, compositors take in rates from parts and provide the proper transformations of the state variables-- in this case the change in concentration of each species.

Here we define the individual compositors (our version of MATLAB already knows what the label “Compositor” means):

```
E1 = Compositor('E1', 10);

M1 = Compositor('M1', 10);

M2 = Compositor('M2', 0);
```

Here we actually add the three compositors to our system:

```
sys1.AddCompositor(E1);

sys1.AddCompositor(M1);

sys1.AddCompositor(M2);
```

Next we define and add the parts of our system. In this example we only have one part. We will set $k_{cat} = 10$ and $k_m = 5$. Note that the part "knows" how many state variables it affects and provides a vector containing the change in all three of its variables.

Here we define Part P1 based on the mechanism, $k_{cat}$and$k_m$we have previously determined:

```
      P1 = …

        Part('M1-(E1)->M2', ...

             [M1 E1 M2], ...

             [Rate('(-10 * val(E1) * val(M1)) / (5 + val(M1))'), ...

              Rate('0'), ...

              Rate('(10 * val(E1) * val(M1)) / (5 + val(M1))')]);
```

The system we defined will next be converted to ODEs and solved with a standard numerical solver. The conversion of our system is straightforward and performed automatically, but it is helpful to go through the mechanics once.

Above, we initialized the system such that the relevant values at t=0 are as follows:

$$[E_1] = 10$$

$$[M_1] = 10$$

$$[M_2] = 0$$

$$\frac{d[E_1]}{dt} = 0$$

$$\frac{d[M_1]}{dt} = 0$$

$$\frac{d[M_2]}{dt} = 0$$

Initially these derivatives are all 0 because we have not yet added any parts to the system. When we add a part, we add a term that changes the state of those species. For example, when we add part P1 to our system:

```
        sys1.AddPart(P1);
```

the above equations are modified automatically by the tool, based on the definition above of P1, to the following:

$$\frac{d[E_1]}{dt} = 0 \quad \textit{// remember that } [E_1] \textit{is constant for this part}$$

$$\frac{d[M_1]}{dt} = -10 \cdot \frac{val(E_1) \cdot val(E_2)}{5 + val(M_1)}$$

$$\frac{d[M_2]}{dt} = 10 \cdot \frac{val(E_1) \cdot val(E_2)}{5 + val(M_1)}$$

Although we have defined separate compositors, the equations they define are coupled-- they share process terms and therefore change state in coordinated ways.

With the parts and compositors for the system defined, we can try to analytically solve the system (this is hard for these sorts of equations) or perform a simulation. We choose to do the latter using our MATLAB PosB package.

```
        [T,Y] = sys1.run(3);
```

Inside `sys1.run()` is the code that executes the numerical solver for this equation. A numerical solver essentially discretizes time; that is, it calculates a small *Δt*. The current values for the species' states are plugged into the right hand side of the equations and essentially multiplied by that *Δt* to estimate the change in state. The molecular state is then updated with this change. When this is done repeatedly, a trajectory of concentration over time is created. The "3" in the call is how long to run it -- three time units. Thus at the end of the call we have a trajectory of concentrations from t=0 to t=3. The numerical solver uses methods that choose *Δt* values to minimize errors introduced by the discretization, but we can plot the trajectories on an evenly spaced series of points.

To plot the results:

```
        num_compositors = size(Y,2);

        for i=1:num_compositors

          subplot(num_compositors, 1, i);

          plot(T, Y(:,i));

          xlabel('time');
```

```
    ylabel(strcat('[',sys1.compositors(i).name,']'), 'Rotation', 0);
  end
```

Designing a Desired Output for the Simple Enzymatic Reaction

As discussed in the textbook, we may want to design the simple enzymatic system to reach 90% completion at a given time. Here is program code to generate a data matrix and corresponding heat map relating values of k_{cat} and K_m to the difference between desired and predicted completion time:

```
%
% sim_single_enzyme_engineering
%
% simulate changes in Kcat and Km for an enzyme
%
clear;
%
% overall algorithm:
%
% for x=[low_Kcat high_Kcat]
%  for y=[low_Km high_Km]
%    construct system
%    run simulation
%    determine completion rate
%    compute difference from desired completion rate
%    store this value in a matrix
%  end for
% end for

desired_completion_90pct_time = 4;
simulation_run_time = 10;
timestep = 0.01;
tspan = 0:timestep:simulation_run_time;
```

```
init_E1_val = 10;
init_M1_val = 10;
init_M2_val =  0;

E1_index = 1;
M1_index = 2;
M2_index = 3;

M1_completion_threshold = 0.1 * init_M1_val;

Kcat_vals = 10.^[0.01:0.005:1.0];
Km_vals   = 10.^[0.10:0.10:2.0];

Kcat_indices = 1:length(Kcat_vals);
Km_indices   = 1:length(Km_vals);

delta_completion_time = zeros(length(Kcat_indices),length(Km_indices));

for Kcat_idx = Kcat_indices
    for Km_idx = Km_indices

        %
        % do several iterations of simulations with decreasing
        % window size to find good approximation of completion time
        %
        done_iterations = false;
        num_iterations = 1;

        E1_val = init_E1_val;
        M1_val = init_M1_val;
        M2_val = init_M2_val;

        time_offset = 0;

        while done_iterations == false
            %
```

```
% construct the system
%
sys1 = BioSystem();

E1 = Compositor('E1', E1_val);
M1 = Compositor('M1', M1_val);
M2 = Compositor('M2', M2_val);

sys1.AddCompositor(E1);
sys1.AddCompositor(M1);
sys1.AddCompositor(M2);


%
% Part M1-(E1)->M2
%
M1_rate_str = strcat(...
    '(-', num2str(Kcat_vals(Kcat_idx)), ...
    ' * val(E1) * val(M1)) …
    / (', num2str(Km_vals(Km_idx)), ' + val(M1))');
M2_rate_str = strcat(...
    '(', num2str(Kcat_vals(Kcat_idx)), ...
    ' * val(E1) * val(M1)) / (', ...
    num2str(Km_vals(Km_idx)),' + val(M1))');
P1 = Part('M1-(E1)->M2', [M1 E1 M2], ...
    [Rate(M1_rate_str), Rate('0'), Rate(M2_rate_str)]);

sys1.AddPart(P1);

disp(strcat('simulation: start time=', …
       num2str(time_offset), ...
     ', Kcat = ', num2str(Kcat_vals(Kcat_idx)), ...
     ', Km = ', num2str(Km_vals(Km_idx))));

[T,Y] = sys1.run(tspan);

%
% plot simulation
```

```
        %
        % determine time to complete 90 percent of reaction
        % (estimated by depletion of M1)
        c = 1;
        while (c<length(T)) && ...
              (Y(c,M1_index) > M1_completion_threshold)
            c = c+1;
        end

        completion_90pct_time = T(c);
        time_offset = time_offset + T(c-1);

        % get initial values for next iteration of simulation
        E1_val = Y(c-1,E1_index);
        M1_val = Y(c-1,M1_index);
        M2_val = Y(c-1,M1_index);

        num_iterations = num_iterations-1;
        if (num_iterations < 1)
            done_iterations = true;
        end
    end

    delta_completion_time(Kcat_idx,Km_idx) = abs(...
        time_offset + completion_90pct_time -
        desired_completion_90pct_time);

end
if (Kcat_idx > 2)
    figure(1001);
    h = contour(delta_completion_time);
    pcolor(delta_completion_time);
    shading interp;
    g = gca;
    colorbar;
    title('Difference from desired completion time');
    xlabel('Km (log)');
```

```
        ylabel('Kcat (log)');
    end
end

figure;
g = contour(delta_completion_time);
pcolor(delta_completion_time);
shading interp;
colorbar;
title('Difference from desired completion time');
xlabel('Km (log)');
ylabel('Kcat (log)');

figure;
contour(delta_completion_time(1:60,1:12))
pcolor(delta_completion_time(1:60,1:12))
shading interp;
colorbar;
title('Difference from desired completion time');
xlabel('Km (log A.U.)');
ylabel('Kcat (log A.U.)');
```