1 Model overview

SUGAR 2.0 provides a general interface for incorporating new device models. SUGAR formulates the governing equations for a device as

$$Mx'' + Dx' + Kx = F(x, x', t)$$

The system matrices M, D, and K, and the forcing function F are all constructed from local quantities contributed by the elements in the device. In this document, we describe the interface for building the model functions to describe these elements.

A set of basic SUGAR model functions is provided in the `model` subdirectory. All model functions begin with `MF_` in order to prevent conflicts with other Matlab function names. For instance, the model function for a simple electrical resistor is named `MF_R.m`. The prefix is invisible to a user who just wants to write netlists and does not care about extending the set of intrinsic model functions.

A model function serves several purposes: it is used to form the system of governing equations, to check the validity of input parameters, to determine positions of the mechanical nodes in the system, and to display the device. Each of these functions is handled by a different case in the model. Therefore, at a high level, a model function looks like

```
% A model function skeleton

function [output] = MF_template(flag, R, params, q, t, nodes);

switch (flag)

case 'vars'

  % Code for case vars

% ... more cases

otherwise

  output = [];

end
```

Most model functions will not have a use for all the possible cases; for example, there is no display case associated with a purely electrical element like a voltage source. By including an `otherwise` clause at the end of the model function, any cases which are not defined are handled using an automatic default. Ending with an `otherwise` clause also allows users to extend SUGAR with additional capabilities, possibly supported by new model function cases, without breaking pre-existing model functions.

In the remainder of this section, we will walk through the steps in writing a simple model function. Our particular example will be a gap-closing actuator consisting of two parallel beams.

This is probably a good place for a figure...

1.1 Model function arguments

All model functions have the same argument signature. The arguments are

flag Name of the case that the caller needs performed. The possible cases are described further below.

R 3-by-3 rotation matrix mapping from the local coordinate system into global coordinates.

params Structure containing parameters to the model function for this element, such as orientation (ox,oy,oz), resistance (R), etc. This structure will also contain any parameters inherited from the process information.

q A state vector consisting of [x; xdot]. This should only be used in evaluating the contributions to the forcing function F and its Jacobian.

t Time. This should only be used in evaluating contributions to F and its Jacobian.

nodes The structures associated with the nodes this element affects. For more information on the node info structure, see the comments in [parse_enrich.m]. *I know this deserves a section, too.*

Model functions also have a single output parameter (labelled `output` in the skeleton example above). The exact nature of the output depends on which case is called.

1.2 Variable definitions

The 'vars' case defines node variables and branch variables associated with the element. The output is a structure which can contain three fields: ground, dynamic, and branch.

Ground variables are variables which are always forced to zero, such as the voltage at an electrical ground, or the displacement at a mechanical anchor. Note that the variables which one element considers dynamic may be grounded by another element. Dynamic variables are nodal variables which appear as free variables in the equations for this model. Branch variables are free variables associated with the element itself rather than with any particular node. Examples include the branch current for an inductor.

The format of `output.dynamic` (or `output.ground`) is

```
{ nodeid1 {'var1' 'var2' ...};
  nodeid2 {'var1' ...};
  % ...
}
```

i.e. it is a cell array of rows, one for each node. The first entry on a row identifies the number of a node, and the second entry is a cell array containing the names of the associated variables. The format of `output.branch` is

```
{ 'var1' 'var2' 'var3' ...}
```

where 'var1', etc. are the names of branch variables.

If a model contributes no variables of a particular type, then the corresponding field should not appear in the model function output. For example, our parallel-plate gap closing actuator model only contributes dynamic variables.

```
case 'vars'

  output.dynamic = {1 {'x' 'y' 'z' 'rx' 'ry' 'rz'} ;
                    2 {'x' 'y' 'z' 'rx' 'ry' 'rz'} ;
                    3 {'x' 'y' 'z' 'rx' 'ry' 'rz'} ;
                    4 {'x' 'y' 'z' 'rx' 'ry' 'rz'} };
```

The order in which variables appear in the output for this case determines the assignment of the local indices for the variables. For example, the fact that the y-displacement variable `'y'` for the second node appears eighth in the overall list means that the eighth component of the input state vector q will by that y-displacement, the eighth component of the output for the local contribution to F will be the force on node 2 in the y direction, and so on. All dynamic variables are ordered before branch variables. Grounded variables are not assigned local indices.

1.3 Parameter checking

The 'check' case of the model function performs parameter checking. This may simply entail ensuring that required parameters are present, but it may also include checks to ensure that the parameters are legal (for example, no negative lengths or widths). If there is an error, a descriptive string is returned via `output`; otherwise, an empty array is returned.

For our electrostatic parallel-plate gap example, we have

```
case 'check'

  if (~isfield(params, 'density')       | ...
      ~isfield(params, 'fluid')         | ...
      ~isfield(params, 'viscosity')     | ...
      ~isfield(params, 'Youngsmodulus') | ...
```

```
    ~isfield(params, 'permittivity'))
  output = 'Missing process parameters';
elseif ~isfield(params, 'l')
  output = 'Missing length';
elseif ~isfield(params, 'w1')
  output = 'Missing width w1 of first beam';
elseif ~isfield(params, 'w2')
  output = 'Missing width w2 of second beam';
elseif ~isfield(params, 'gap')
  output = 'Missing gap between beams';
elseif ~isfield(params, 'h')
  output = 'Missing beam height';
elseif ~isfield(params, 'V')
  output = 'Missing voltage difference V';
else
  output = [];  % All checks passed!
end
```

1.4 Local matrices for linear terms

The local contributions to the mass, damping, and stiffness matrices are returned by cases 'M', 'D', and 'K', respectively. As noted above, the order in which the variables are indexed should correspond to the order of declaration from the 'vars' case.

The 'M' case for our sample model function makes use of another model function to build the necessary matrices. Since the gap consists of two beams, and since our indexing puts the variables in the same order that the `beam3d` model uses, first for the beam connecting nodes 1 and 2 and then for the beam connecting 3 and 4, we are able to call through directly to `MF_beam3d` to get the two blocks.

```
case 'M'

  params.w = params.w1;
  output(1:6,1:6) = MF_beam3d('M', R, params);

  params.w = params.w2;
  output(7:12,7:12) = MF_beam3d('M', R, params);
```

The 'D' and 'K' cases are analogous.

1.5 Local force contributions

The presence of an electrostatic forcing function is the only thing that makes our sample electrostatic gap model more than just a pair of beams that happen

to be close to each other. The code is included in this manual primarily to illustrate the use of the rotation matrix argument `R` to translate from global to local coordinates and back.

```
case 'F'

x1 = R'*x(1:3);    % xyz displacement of node 1
x2 = R'*x(7:9);    % xyz displacement of node 2
x3 = R'*x(13:15);  % xyz displacement of node 3
x4 = R'*x(19:21);  % xyz displacement of node 4

V = param.V;   % Constant voltage

gap = param.gap;           % Gap width
l = param.l;               % Length of the beams
A = l * param.h;           % Area of attracting surfaces
e0 = param.permittivity;   % Permittivity of free space
c = 0.5 * e0 * V*V * A;    % Constant in attraction magnitude

% Distance squared between node 1-3 and between node 2-4
d1 = sum(( [0; gap; 0] + x1 - x3 ).^2);  % |x1 - x3|^2
d2 = sum(( [0; gap; 0] + x2 - x4 ).^2);  % |x2 - x4|^2

F1 = [R*[ 0; -c/(2*d1);            0];  % Force on node 1, local y direction
      R*[ 0;         0; -c*l/(12*d1)]]; % Moment on node 1

F2 = [R*[ 0; -c/(2*d2);            0];  % Force on node 2, local y direction
      R*[ 0;         0; +c*l/(12*d2)]]; % Moment on node2

% Forces and moments on nodes 3 and 4 are opposite those on 1 and 2
output = [F1; F2; -F1; -F2];
```

The cases 'dFdx' and 'dFdxdot' compute the local contributions to the Jacobian of F with respect to x and x' respectively. These Jacobians are used in the Newton-Raphson iteration to compute the static solution $Kx = F(x)$, in computing the linearized system at an equilibrium point for static and modal analysis, and in stiff solvers for the transient equations.

1.6 Node positioning

SUGAR determines the position of nodes in a mechanical structure by querying the model functions in the structure to find the relative positions of the nodes they reference. For example, the parameters `l`, `gap`, `w1`, and `w2` in our example model completely determine the relative positions of the four nodes. The case 'pos' computes these relative positions. Column i of the `output` matrix from the 'pos' case is the relative position of node i.

```
case 'pos'

  l = params.l;
  g = params.gap + (params.w1 + params.w2)/2;

  output = R * ...
      [0  l   0  l;
       0  0  -g -g;
       0  0   0  0];
```

It is worth noting that all three coordinates should be specified even for two-dimensional models.

If no node is assigned absolute coordinates, the first mechanical node to appear in the netlist is placed at the origin. However, it is possible to specify the absolute coordinates of a node. For example, the following case ('abspos') in our sample model function would allow the user to specify explicitly the absolute coordinates of node 1.

```
case 'abspos'

  if (isfield(params, 'x') & isfield(params, 'y') & isfield(params, 'z'))
    output = [params.x; params.y; params.z];
  else
    output = [];
  end
```

1.7 Element display

The 'display' case displays the element as part of a picture of the device. All the existing functions display using the `displaybeam` function, which takes as parameters the six 3-d degrees of freedom for each of its end points, the position of the first end point, and a parameter structure describing the beam geometry (length `l`, width `w`, and height `h`). More sophisticated types of displays, such as displays which are shaded according to stress contours, are also possible.

The display case for our simple gap model simply calls `displaybeam` to show the two component beams.

```
case 'display'

  params.w = params.w1;
  displaybeam(q(1:12), nodes(1).pos, params);

  params.w = params.w2;
  displaybeam(q(13:24), nodes(3).pos, params);
```