SUGAR: A MEMS Simulation Program

18 August 2001

Contents

Part I

SUGAR User Manual and Reference

Chapter 1

Introducing SUGAR

1.1 What is SUGAR?

In less than a decade, the MEMS community has leveraged nearly all the integrated-circuit community's fabrication techniques, but little of the wealth of simulation capabilities. A wide range of student and professional circuit designers regularly use circuit simulation tools like SPICE, while MEMS designers often resort to back-of-the-envelope calculations. For three decades, development of IC CAD tools has gone hand-in-hand with the development of IC processes. Tools for simulation will play a similar role in future advances in the design of complicated micro-electromechanical systems.

SUGAR inherits its name and philosophy from SPICE. A MEMS designer can describe a device in a compact netlist format, and very quickly simulate the device's behavior. Using simple simulations in SUGAR, a designer can quickly find problems in a design or try out new ideas. Later in the design process, a designer might run more detailed simulations to check for subtle second-order effects; early in the design, a quick approximate solution is key. SUGAR provides that quick solution.

SUGAR is primarily written in MATLAB, in order to make it easier to install and improve. For performance reasons, some routines are written in C and pre-compiled as Matlab external functions, but this is transparent to the casual user. Because SUGAR runs inside MATLAB, users have access to the full power of the MATLAB environment as well as to the specialized analysis routines of SUGAR.

1.2 A first example

Let's first look at a simple example of how SUGAR is used. Perhaps we have just designed a simple test structure in the MUMPS process, a long cantilever anchored at one end. We want to know how much it will deflect if we apply a small force to the free end.

First, we need an input file, called a *netlist*, that describes the device. We save this netlist in the file `cantilever.net`:

```
uses mumps.net

anchor p1 [A]   [l=10u w=10u]
beam3d p1 [A B] [l=100u w=2u]
f3d    *  [B]   [F=50u oz=90]
```

The first line tells SUGAR that our device is made using the standard MUMPS process. The second and third lines tell SUGAR that the device is made of an anchor at node `A`, which is ten microns on a side; and a beam, two microns wide and one hundred long, which goes from the anchored end point `A` to the free end `B`. Both the anchor and the beam are made of the poly1 layer, or `p1`. The fourth line describes a force of 50 micro-Newtons applied at a right angle to the free end of the beam.

From within MATLAB, we only need to write three commands to see the effect of the force on the beam:

```
> net = cho_load('cantilever.net');
> q = cho_dc(net);
> cho_display(net, q);
```

The first command tells SUGAR to process the netlist description in `cantilever.net`. The result is a data structure stored in the `net` variable that describes the device to other SUGAR routines. The `cho_dc` function does a static (DC) analysis of the device, and returns a vector representing the equilibrium position. Finally, the third line causes SUGAR to display the displaced device.

1.3 Installing SUGAR

All the SUGAR software is available from the SUGAR home page at

```
http://www-bsac.eecs.berkeley.edu/~cfm
```

We update the software frequently; if you encounter problems, you may want to download the latest version before devoting hours of your time (or ours!) to debugging.

There is also a web interface to SUGAR at:

```
http://sugar.millennium.berkeley.edu/
```

1.3.1 System requirements

To require SUGAR, you will need MATLAB release 5.2 or later. Because the student edition of MATLAB 5 only handles matrices of a limited size, users of the student edition will only be able to simulate small devices.

SUGAR is regularly tested using MATLAB 5.3 on Windows, Sun, HP, and Alpha systems, and is tested using MATLAB 6.0 on Linux systems. We have not tested the software on other systems. If you would like to use SUGAR on a different system, you will need to compile the external routines for that system, as described later in this section.

1.3.2 Setting SUGAR paths

To use SUGAR, make sure that your Matlab path is set correctly. In particular, make sure the analysis and model subdirectories are included in your Matlab path. This can be done from within Matlab, e.g.

```
addpath /home/eecs/dbindel/sugar/analysis
addpath /home/eecs/dbindel/sugar/model
```

or from the shell, by setting the MATLABPATH environment variable. In `csh`, for instance, this might be

```
setenv MATLABPATH /home/eecs/dbindel/sugar/analysis:\
/home/eecs/dbindel/sugar/model
```

1.3.3 Compiling external routines

You will need to compile the external routines *only* if pre-compiled versions are not already available on your system. To compile the routines, change to the `compile` subdirectory and from Matlab type `makemex`. Then copy all the files beginning with `sugar_c` to the `analysis` subdirectory.

1.4 Getting (and giving) help

If you have concerns or difficulties using SUGAR which are not addressed in the manual sections, feel free to write to

`cfm@bsac.eecs.berkeley.edu`

We will try to respond promptly.

SUGAR *is* research software; if you would like to contribute models, analysis routines, or examples to the SUGAR project, let us know that, too!

Chapter 2

Describing devices

Devices in SUGAR are described by input files called *netlists*. In this chapter, we describe the features of the netlist language.

2.1 Dissecting an example

(TODO: Want an example that uses all the netlist features.)

2.2 Units and metric suffixes

By convention, the SUGAR model functions use the familiar MKS (meter-kilogram-second) system of units. This means that beam lengths, for example, are measured in meters instead of micrometers. In order to make it easier to type lengths of microns and pressures of gigapascals, we adopt a standard system of metric suffixes that can be appended to SUGAR numbers. For example, a hundred micron length in SUGAR could be represented as `100u` as well as in scientific notation (`100e-6`) or as a simple decimal (0.0001). Similarly, a Young's modulus of 150 gigapascals might be written as `150G`. Both suffixes and ordinary scientific notation can be used together, too; `0.1e7u` is a perfectly legitimate (if somewhat silly) way of writing the number 1.

The standard suffixes are:

`d`	deci	10^{-1}			
`c`	centi	10^{-2}	`h`	hecto	10^{2}
`m`	milli	10^{-3}	`k`	kilo	10^{3}
`u`	micro	10^{-6}	`M`	mega	10^{6}
`n`	nano	10^{-9}	`G`	giga	10^{9}
`p`	pico	10^{-12}	`T`	tera	10^{12}
`f`	femto	10^{-15}	`P`	peta	10^{15}
`a`	atto	10^{-18}	`E`	exa	10^{18}

2.3 Expressions

It's often convenient to do simple calculations inside a netlist. For example, suppose I have defined a variable `beamL` for the length of a beam in a device. Then we can define the length of another beam in terms of `beamL`:

```
uses mumps.net

beamL = 10u   % Make a beam ten microns long

% make a beamL by beamL/2 rectangle
beam3d p1 [A B] [w=2 l=beamL]
beam3d p1 [C D] [w=2 l=beamL]
beam3d p1 [A C] [w=2 l=beamL/2 oz=90]
beam3d p1 [C D] [w=2 l=beamL/2 oz=90]
```

SUGAR's expressions look much like Matlab or C expressions: we can add, subtract, multiply, divide, exponentiate, and negate numbers, and also evaluate functions. There are also comparison operators: `==`, `!=`, `<=`, `>=`, `<`, and `>`; and there are simple logic operators, "not" (`!`), "and" (`&`), and "or" (`|`). All operations are left associative, so `a + b + c` is evaluated as `(a + b) + c` rather than as `a + (b + c)`. The order of operations, from highest precedence to lowest precedence, is

Level	Operators
7	`|` (logical or)
6	`&` (logical and)
5	`==` (equality), `!=` (inequality), `>` (greater), `<` (less), `>=` (greater or equal), `<=` (less or equal)
4	`-` (add), `+` (subtract)
3	`*` (multiply), `/` (divide)
2	`-` (negate), `!` (logical not)
1	`^` (exponentiate)

As in C and Matlab, there is no separate logical type. Non-zero numbers are interpreted as "true," and zero is interpreted as "false." When a comparison or logical operation is true, it will evaluate to 1.

Besides numbers, SUGAR supports a string type. String literals are denoted by double quotes. Unlike C and Matlab, the backslash does not quote characters in SUGAR expressions: `"\t"` is a backslash followed by a t, not a tab. Strings cannot be operands to arithmetic or logical operations, but they can be compared for equality or inequality.

SUGAR calls Matlab to evaluate almost all functions, so any functions that Matlab provides are available. Matlab functions called from SUGAR must return some string or floating point number. In addition to Matlab functions, SUGAR provides access to two intrinsic functions:

`cond(test, ifval, elseval)` - returns `ifval` when `test` is true, `elseval` otherwise.

`print(` **arguments** `)` - print out the arguments, and return the number of items printed. This function is provided purely for debugging convenience.

2.4 Rotations and Euler angles

In order to orient structures in SUGAR, users specify how each piece is rotated from a model coordinate frame into its actual orientation in the structure. Rotations are specified by a sequence of rotations about the x, z, and y axes, respectively. The amount to rotate about each axis is given by angles `ox`, `oz`, and `oy`, given in degrees; these three numbers are known as *Euler angles*, and can be used to describe any rotation.

For many applications, most of the structure will lie initially in a single plane, and the only rotations used to describe the device will be rotations about the z axis. For example, the lines

```
beam3d p1 [A B] [w=2 l=beamL]
beam3d p1 [C D] [w=2 l=beamL]
beam3d p1 [A C] [w=2 l=beamL/2 oz=90]
beam3d p1 [C D] [w=2 l=beamL/2 oz=90]
```

describe a rectangle; the first two beams run parallel to the x axis, and the latter two beams are rotated ninety degrees in the plane to run parallel to the y axis.

For more complicated examples, it is important to remember that order matters. To go from local coordinates to global coordinates, first the rotation about the x axis is applied, then the z axis, and then the y axis. For instance, in the model coordinate system, a beam points in the direction $(1, 0, 0)$, along the x axis. If we rotate the beam first 90 degrees about the x axis and then 90 degrees about the z axis, it would point along the y axis, in the $(0, 1, 0)$ direction. If, however, we were to rotate the beam first about the z axis and then about the x axis, it would end up pointing along the z axis.

2.5 Lexical notes

SUGAR 2.0 netlists are "free form"; that is, white space characters like tabs and carriage returns are not significant. For example, the netlists

```
uses mumps.net
beam2d p1
  [A B]
  [l = 100u
   w = 100u
  ]
```

and

```
uses mumps.net
beam2d p1 [A B] [l=100u w=100u]
```

are equivalent.

A comment in a netlist begins with `%` and extends to the end of the line.

A SUGAR identifier (like a C identifier) consists of a letter followed by a string of letters, numbers, and underscores. The keywords `uses`, `subnet`, `param`, `process`, `for`, and `if` are reserved, and cannot be used as identifiers.

2.6 `uses` statements

Netlists can contain `uses` statements to include other files. `uses` statements. A `uses` statement has the form

```
uses filename
```

For example, many netlists use the data for MUMPS process layers defined in `mumps.net`, and begin with the line

```
uses mumps.net
```

Files included by a `uses` statement are not particularly special. You can use `uses` to include files of process parameters, libraries of frequently-used subnets, etc.

A file will only be used once in a netlist. For example, if the file `subnets.net` started with

```
uses mumps.net
```

and a test netlist called `test.net` started with

```
uses subnets.net
uses mumps.net
```

then `test.net` would only include `mumps.net` once, and would not complain about the contents of `mumps.net` being defined multiple times.

2.7 Element lines

The basic unit of a SUGAR netlist is an element line. For example,

```
crossbeam beam2d p1 [A B] [l=100u w=2u]
```

is an element line describing a beam. This line consists of several fields:

- The first field, which is optional, is the name of the element; in this case, the element is named `crossbeam`.
- The second field is the name of the model for the element; in this case, it is the two-dimensional beam model `beam2d`. There are models for beams, anchors, electrical devices, etc.; a complete list of models, along with information on how to build new models, can be found in other SUGAR documentation.
- The third field is the name of the process parameter structure; in this case the beam is fabricated in the first layer of polysilicon in a MUMPS process, named `p1`. By specifying the process layer `p1`, a user informs the model function of common material parameters such as the Young's modulus for polysilicon and the thickness of the deposited layer. For models which require no process information, such as models for external forces, the process field may be set to `*`.
- After the process field comes a list of nodes, surrounded in brackets. In this case, the specified beam connects nodes `A` and `B`. Elements are connected together by sharing a common node. For instance, to attach a wider 100 micron beam to the `B` end of the beam above, we might write

  ```
  beam2d p1 [B C] [l=100u w=5u]
  ```

  Unlike in previous versions of SUGAR, node names in SUGAR 2.0 must begin with an alphabetic character.
- After the list of nodes comes a list of model parameters. In the example above, the model parameters consisted of the length and width of a beam; other models will require other parameters. A parameter specification always has the form `identifier = expression`.

2.8 Parameters and definitions

In order to allow users to experiment with variations on a simulation, or to do parameter sweeps, SUGAR supports named parameters. An example might be

```
param nfingers, l=10u
```

A parameter definition consists of the `param` keyword, followed by a comma-separated list of parameter names and default values. Default values are optional; however, it is an error to use the netlist without setting any parameters without defaults. Parameter defaults may be expressions depending on previously defined variables or parameters.

SUGAR netlists may also include definitions, such as

```
long_length = 200u
short_length = 100u
avg_length = (long_length + short_length)/2
```

Netlist variables are scoped, so that a definition made inside a subnet (see below) will not affect top-level element statements.

2.9 Process parameter structures

Physical parameters associated with a particular layer of a particular material are process parameters. An example of the baseline process information for the polysilicon layers in MUMPS (`default`) is provided below

```
process default = [
    Poisson = 0.3               %Poisson's Ratio = 0.3
    thermcond = 2.33            %Thermal conductivity Si = 2.33e-6/C
    viscosity = 1.78e-5         %Viscosity (of air) = 1.78e-5
    fluid = 2e-6                % Between the device and the substrate.
    density = 2300              %Material density = 2300 kg/m^3
    Youngsmodulus = 165e9       %Young's modulus = 1.65e11 N/m^2
    permittivity = 8.854e-12    %permittivity: C^2/(uN.um^2)=(C.s)^2/kg.um^3
    sheetresistance = 20        %Poly-Si sheet resistance [ohm/square]
]
```

In general a process definition has the form `process name = [...]`, where `process` is a keyword, `name` is the name to be given to the process information, and process definitions are given between the square brackets.

Process parameter structures may be derived from other process parameter structures. For example, a 2 micron poly layer named `p1` might be written

```
process p1 : default = [
    h = 2u
]
```

This layer automatically includes all the definitions made in the default process parameter structure.

2.10 Subnets

A subnet is analogous to a SPICE subcircuit, or to a function in C. Subnets provide users with a means to extend the set of available models without leaving SUGAR. An example subnet for a single unit of a serpentine structure taking up area is shown below:

```
subnet serpent [A E] [unitwid=* unitlen=* beamw=2u]
[
  len2 = unitlen/2
  beam2d parent [A b] [l=unitwid w=beamw oz=-90]
  beam2d parent [b c] [l=len2    w=beamw]
  beam2d parent [c d] [l=unitwid w=beamw oz=90]
  beam2d parent [d E] [l=len2    w=beamw]
]
```

Element lines using subnets are invoked in the same manner as element lines using model functions built in Matlab

```
serp1 serpent p1 [x y] [unitwid=10u unitlen=10u]
serpent p1 [y z] [unitwid=10u unitlen=10u w=3u]
```

The `parent` process for a subnet is the process specified in creating an instance of that subnet. In the above example, the `p1` process information would be used for the beams in the serpent subnet.

In general, a subnet definition consists of the keyword `subnet`, a name for the subnet model, a bracketed list of node names, a bracketed list of parameters, and a code block. The code block is enclosed in square brackets, and may include definitions, element lines, and array structures (see below).

Sometimes it may be necessary to access variables attached to nodes internal to a netlist. For example, in the above example we might be interested in the version of node `b` for subnet instance `serp1`. In the analysis functions, that node would be referred to as `serp1.b`. It would not be valid to refer to node `x` as `serp1.A`, since `x` already has a name defined outside the subnet.

Subnet instances that are not explicitly named, like the second `serpent` element in the example above, are assigned names consisting of `anon` followed by some number. It is possible to use a name like `anon1.b` to refer to the `b` node in the second line, but it is not recommended since the internal naming schemes for anonymous elements are subject to future change.

2.11 Arrays

SUGAR provides syntactic support for arrays of structures. For example, the following code fragment creates a spring composed of twenty of the serpentine units from the subnet example and anchors it at one end:

```
a1 anchor p1 [x(1)] []
for k = 1:20
[
  link(k) serpent p1 [x(k) x(k+1)] [unitwid=10u unitlen=10u]
]
```

Note that both element names and node names may be indexed. It is possible to have names with multiple indices as well (eg `link(i,j)`). The index variable is only valid within the loop body.

The general syntax of a for loop is

```
for index = lowerbound : upperbound
[
  ... code lines ...
]
```

where `index` is the name of the index variable, `lowerbound` is an expression for the lower bound of the loop, and `upperbound` is an espression for the upper bound of a loop.

2.12 Conditionals

SUGAR supports `if` statements of the form

```
if expression [
  code
]
```

and

```
if expression [
  code
] else [
  code
]
```

The main purpose of `if` statements is to give some flexibility to subnet writers. For example, suppose I wanted a beam that would automatically compute its electrical resistance if one was not provided. I could do that with the following subnet:

```
subnet mybeam [A B] [l=* w=* h=* R=-1 resistivity=*]
[
  % If R is -1, then the user didn't specify anything to override
  % the default, so we'll help calculate the resistance.
  if (R == -1) [
    resistance = resistivity * l/(w*h)
    beam3d parent [A B] [l=l w=w h=h]
    R      parent [A B] [R=resistance]
  ]

  % Otherwise, we'll just accept whatever the user wrote in
  else [
    beam3d parent [A B] [l=l w=w h=h]
    R      parent [A B] [R=R]
  ]
]
```

There are a few caveats that go with this example:

1. Usually, `resistivity` would be defined in the process information when an instance of `mybeam` was created.

2. An obvious, simple, and incorrect way to write this example would be

```
subnet mybeam [A B] [l=* w=* h=* R=-1 resistivity=*]
[
  if (R == -1) [
    resistance = resistivity * l/(w*h)
  ] else [
    resistance = R
  ]
  beam3d parent [A B] [l=l w=w h=h]
  R      parent [A B] [R=resistance]
]
```

the problem with this solution is SUGAR's scoping rules. The variable `resistance` defined in the `if` case and the `else` case is invisible outside of the `if` statement where it was defined.

3. The SUGAR netlist language was designed to be simple, with just enough features to easily describe a device. By using the `if` statement, it is possible to write arbitrarily complicated netlists, with constructs like recursive subnets or even more subtle beasts. Exercise good taste when you write netlists, and try to relegate any subtle and complicated coding tasks to Matlab instead of to the SUGAR netlist language.

Chapter 3

Analyzing devices

3.1 Types of analysis

SUGAR supports three basic styles of analysis:

Static analysis : In static analysis, we find the equilibrium state of a device. Static analysis is sometimes called DC analysis by analogy to the equilibrium analysis for direct current circuits.

Linearized analysis : A linearized approximation to a system near equilibrium can provide valuable information about the stability of the system and the nature of small oscillations about equilibrium. SUGAR provides two flavors of linearized analysis:

- In *modal analysis*, the characteristic modes of the system (and their corresponding frequencies) are determined. SUGAR can display the shapes of the displacements corresponding to various modes.
- In *steady-state analysis*, SUGAR computes the frequency response of a user-specified variable when another user-specified variable is sinusoidally excited. The output of steady state analysis is Bode plots.

Transient analysis : In transient analysis (or dynamic analysis), the motion of the system is integrated forward in time. Transient analysis in SUGAR is still somewhat unreliable; we hope to have better support for it soon.

3.2 Static analysis

In static analysis, we attempt to find an equilibrium state for a MEMS device. In the most general case, the equilibrium may not be unique; in this case, SUGAR will usually find the equilibrium position closest to where it starts looking (which, by default, is the undisplaced position).

The equilibrium state is characterized by a collection of force and moment balance equations (and their electrical and thermal analogues):

$$F(x) = 0$$

where x is a vector of displacements from the original positions (and voltages, temperatures, etc) of the device. We solve these equations using a standard Newton-Raphson iteration. For linear

problems, a Newton-Raphson iteration will converge in one steps; for nonlinear problems, the iteration may never converge. Currently, SUGAR assumes the iteration has converged when the size of the change between iterations is sufficently small in an appropriately scaled norm. If convergence has not set in after 40 iterations, the routine exits with a diagnostic message.

The function to perform static analysis is `cho_dc`:

```
function [q, converged] = cho_dc(net, q0, is_sp)
```

The first argument, `net`, is the netlist structure returned from `cho_load`. The other arguments are optional. The starting value for the iteration is given by `q0`; by default, the iteration starts at the undisplaced position (`q0` = 0). The flag `is_sp` tells the routine whether it should use sparse solvers or not; by default, the flag is true (sparse solvers are used). The function returns a vector of displacements to reach the computed equilibrium (`q`), and a flag that indicates whether the iteration converged (`converged`).

In some cases, it is possible to find tricky equilibrium positions by approaching them step-by-step. For example, suppose we wanted to determine the equilibrium position of a device near a pull-in voltage. As we approach the critical voltage, it becomes more difficult to find the equilibrium position, and past the critical voltage, no equilibrium exists. If the commands

```
params.V = Vfinal;
net = cho_load('device.net', params);
q = cho_dc(net)
```

fail, we could try

```
q = [];
for V = 0:.5:Vfinal
  net = cho_load('device.net', params);
  q = cho_dc(net, q);
end
```

Even if we were still unable to find the equilibrium position, we might get useful information from seeing how nearly we were able to approach the final voltage, and what the equilibrium was at the last point where we were able to find it.

There are two ways to view the results of a static analysis:

1. We can view individual components of the displacement vector using the command `cho_dq_view`:

   ```
   % Find displacement of the y coordinate at node 'tip'
   tipy = cho_dq_view(q, net, 'tip', 'y');
   ```

   Alternately, we could look up the index of the tip y coordinate, and then look at the corresponding entry of the `q` vector:

   ```
   % Find displacement of the y coordinate at node 'tip'
   tipy_index = lookup_coord(net, 'tip', 'y');
   tipy = q(tipy_index);
   ```

2. We can display the shape of the displaced structure using the `cho_display` routine:

```
% Display the undisplaced structure in figure 1,
% and the displaced structure in figure 2.
q = cho_dc(net);
figure(1); cho_display(net);
figure(2); cho_display(net, q);
```

3.3 Steady-state and Modal Analysis

(From Jason's old text)

To determine the steady-state response, SUGAR first linearizes the system of ordinary differential equation at the point of static equilibrium. The high order system of ODEs is then converted into first order form given by

$$\dot{x} = Ax + Bu \tag{3.1}$$
$$y = Cx + Du \tag{3.2}$$

where x is the system dynamic state variable, u is the sinusoidal external excitation, and y is the system dynamic response. A, B, C, and D are the system, input coupling, output, and feed forward matrices respectively [1]. The solution of equation (3.3) provides Bode plots as well as modal analysis.

3.4 Transient Analysis

(From Jason's old text)

This solver calculates the transient response of a MEMS device, which may contain nonlinear elements and excitations that are functions of time t and state vector q. Several ODE solvers are available, whereby speed may be traded for accuracy and long-term stability. These numerical methods include an implicit second order Rosenbrock solver for stiff problems where low accuracy is acceptable, an explicit Runge-Kutta 4th-5th order solver for non-stiff systems, an implicit multistep integration method of varying order for stiff problems requiring higher accuracy, and a simple explicit Euler algorithm. The transient solvers require the system ODEs to be in first order form. We do this in the standardized way [2] by introducing a new state vector Q where

$$\dot{Q} = \frac{d}{dt} \begin{matrix} x \\ \dot{x} \end{matrix} = f(t, x)$$

3.5 Future analysis routines

In the future, we plan to also support *sensitivity analysis*. Sensitivity analysis is not an independent style of analysis as much as it is an extension to the forms of analysis listed above. For example, a static sensitivity analysis might tell how the equilibrium position would change due to variations from the nominal material properties, layer thicknesses, etc. Similarly, sensitivity analysis used with the linearized analysis routines might tell how the fundamental frequencies of the device would change if the device properties were perturbed, and sensitivity analysis of transient results would tell how the dynamic response would be affected by perturbing device properties. Some forms of sensitivity analysis were supported in SUGAR 1.01, but that code has not yet been integrated into SUGAR 2.0.

Chapter 4

More examples

(From Jason's demo20.pdf – figures removed. Need to put them back in, and also add some more sophisticated examples (using parameters, etc))

This section describes how to use SUGAR2.0 by examples. The netlists, commands for running analysis, and output are shown. For convenience, all netlist files given here are available in the SUGAR demo directory. Netlist format is defined in chapter 2.

4.1 Examples with explanations

4.1.1 cantilever example (demo_dc1)

This demo shows how to simulate the deflection of a beam due to an external force, where the beam is fixed at one end. It shows to make the netlist, how run static analysis, and obtain graphical output.

To model the beam a 3D linear beam model called beam3d will be used. If a planar beam is desired, simply replace the model b3dm with beam2d in line 2 of following netlist.

Netlist

The following netlist is created by opening a text editor, entering the 3 lines of netlist text shown below, and saved as cantilever.net. The 3 text lines represent an anchor, beam, and force.

```
% 'cantilever.net'
uses mumps.net
anchor  anchor  p1 [ substrate]  [l=10u w=10u h=10u]
beam beam3d  p1 [substrate  tip]  [l=100u w=2u h=2u]
force  f3d  * [tip]  [F=2u oz=90]
```

The first line in the netlist includes a process file "mumps.net". All process information such as layer thickness, Young's modulus etc. are defined in "mumps.net".

The first line in the netlist represents the anchor element. Anchors are the MEMS components that mechanically ground flexible structures to the substrate. Without anchors, structures would be statically indeterminate.

The material properties of the anchor are given in process file "mumps.net". The fabrication layer of this process is p1.

The anchor is attached to the substrate by the node labeled substrate. Notice that both the anchor element and beam element (described below) contain the node labeled substrate. The anchor is coupled to the beam through node substrate.

The parameters section of this line of text provides the geometry and orientation, where the length, width are 10 microns.

The second element is a flexible beam. The model used for this beam is called beam3d, which is described in appendix. The fabrication layer with which this beam is composed of is the p1 layer (i.e. the first layer of polysilicon). It is fixed on one end due to its connection to the anchor through the node labeled substrate. The opposite end of the beam, labeled tip, is free to move.

The last section of this line provides geometry and orientation. The beam extends to the right, from node substrate to node tip.

The final line is a force applied at the free end of the beam (node tip). The magnitude of the force is given as 2 microNewtons. The orientation of the force vector is in the y- direction since it was rotated from its default position along the positive x-axis.

Command

Once the netlist text file is created, load it into Matlab with the cho_load command. Then static analysis may be performed, which finds a final equilibrium state of the system. Running static analysis on the above netlist requires 3 commands. Within the Matlab workspace:

1. load the netlist,
2. perform static analysis on it, and
3. display the results:

```
net = cho_load('cantilever.net');
dq = cho_dc(net);
cho_display(net,dq);
```

The first command loads the text file called cantilever.net into a variable called net. The net variable contains all of the important information in the netlist file, but converted into structured form favorable to the SUGAR algorithms.

The second line performs the static analysis. The cho_dc command takes net as its input. Using the parameter values given in the netlist and the parameterized element models described in appendix, it calculates the deflection of the structure. The displacement vector dq is the output of cho_dc.

Incidentally, cho stands for the basic building locks of sugar (i.e. carbon, hydrogen, oxygen).

Using geometries and orientations from net and node displacements from dq as input to cho_display, SUGAR can graphically display the deflected structure in Matlab (figure IV.1).

Figure 4.1

To display original, non-deflected structure, simply type

```
cho_display(net);
```

After the structure is displayed, left clicking and dragging within the display window may adjust the view. The magnification buttons in the display window may be used to zoom in and out by first clicking on, say the zoom-in (+), followed by pointing and clicking on the display window at the precise position that is to be magnified. Figure IV.2 shows the cantilever3D structure rotated to a different point of view.

4.1.2 multiple beam example (demo_dc2)

This demo is similar to above demo with the exception that it uses multiple beams and it is deflected by a moment. Netlist is as following:

Netlist

```
% 'multibeam.net'
uses mumps.net
anchor p1 [substrate]   [l=10u w=10u oz=180]
beam3d p1 [substrate A] [l=100u w=2u oz=0]
beam3d p1 [A B]         [l=50u w=4u oz=45]
beam3d p1 [A C]         [l=50u w=4u oz=-45]
beam3d p1 [C D]         [l=50u w=4u oy=-45]
f3d *  [D] [M=1n]
```

As before, each element is connected at shared nodes. The commands to load the netlist, do the analysis, and display the non-deflected and deflected figures are

```
net = cho_load('multibeam.net');
dq = cho_dc(net);
figure(1); cho_display(net);
figure(2); cho_display(net,dq);
```

Figure 4.3
Figure 4.4

4.1.3 Beam gap structure (demo_beamgap)

This is a 2D coupled electrical and mechanical domain analysis. It contains electrical voltage source, electrical ground, electro-mechanical anchors, beam and gap. The netlist and structure of this demo are as following:

Netlist

```
% 'beamgap2e.net'
uses mupms.net
Vsrc     * [A fred]  [V=10 sv=0.1 sph=0]
eground  * [fred] []
anchor   p1 [A]   [l=5u w=10u  oz=180]
beam2de  p1 [A b] [l=100u w=2u h=2u oz=0 R=100]
gap2de   p1 [b c D E] [l=100u w1=5u w2=5u oz=0 gap=2u R1=100 R2=100]
eground  *  [D] []
anchor  p1 [D]   [l=5u w=10u oz=-90]
anchor  p1 [E]   [l=5u w=10u oz=-90]
eground  *  [E] []
```

Equilibrium displacements have been calculated at an input voltage 10v. The deflected structure is shown as following:

4.1.4 Modal analysis (demo_mirror)

This is a 3D mechanical modal analysis for a mirror structure. 3D mechanical anchors and beams are included. Resonant frequencies have been calculated and the first to fourth mode shapes are displayed. Below are the netlist and demo pictures

Netlist

```
% 'mirror.net'
uses mumps.net
anchor  p1 [ b   ] [ l=10e-6 w=10e-6 ox=0 oy=0 oz=90 h=8e-6]
beam3d  p1 [ b  c ] [ l=80e-6 w=2e-6  ox=0 oy=0 oz=-90 h=2e-6]
beam3d  p1 [ d  e ] [ l=80e-6 w=2e-6  ox=0 oy=0 oz=-90 h=2e-6]
anchor  p1 [ e   ] [ l=10e-6 w=10e-6 ox=0 oy=0 oz=-90 h=8e-6]
 %outer frame:
beam3d  p1 [ c  f  ] [ l=100e-6 w=20e-6 ox=0 oy=0 oz=-90 h=4e-6]
beam3d  p1 [ f  d  ] [ l=100e-6 w=20e-6 ox=0 oy=0 oz=-90 h=4e-6]
beam3d  p1 [ c1 f1 ] [ l=100e-6 w=20e-6 ox=0 oy=0 oz=-90 h=4e-6]
beam3d  p1 [ f1 d1 ] [ l=100e-6 w=20e-6 ox=0 oy=0 oz=-90 h=4e-6]
beam3d  p1 [ c  c1 ] [ l=200e-6 w=20e-6 ox=0 oy=0 oz=0 h=4e-6]
beam3d  p1 [ d  d1 ] [ l=200e-6 w=20e-6 ox=0 oy=0 oz=0 h=4e-6]
 %inner torsion hinges:
beam3d  p1 [ g3 f1 ] [ l=40e-6 w=2e-6 ox=0 oy=0 oz=0 h=2e-6]
beam3d  p1 [ f  g6 ] [ l=40e-6 w=2e-6 ox=0 oy=0 oz=0 h=2e-6]
 %inner solid "plate":
beam3d  p1 [ g6 g3 ] [ l=120e-6 w=140e-6 ox=0 oy=0 oz=0 h=4e-6]
 %rear lever:
beam3d  p1 [ h  f  ] [ l=75e-6 w=80e-6 ox=0 oy=0 oz=0 h=4e-6]
```

4.1.5 Steady state analysis (demo_ss)

This is a 2D steady state analysis for a resonator as following:

Netlist

```
% 'multmode_m.net'
uses mumps.net
a1 anchor p1 [2]  [l=5e-6   oz=0     w=10e-6  R=100]
b1 beam2d  p1 [2  5]  [l=150e-6 oz=180   w=2e-6   R=1000]
a2 anchor p1 [3]  [l=5e-6   oz=0     w=10e-6  R=100]
b2 beam2d  p1 [3  6]  [l=150e-6 oz=180   w=2e-6   R=1000]
b1v beam2d p1 [5  7]  [l=50e-6  oz=90    w=5e-6  R=500]
b2v beam2d p1 [5  6]  [l=50e-6  oz=-90   w=5e-6  R=500]
b3v beam2d p1 [6  8]  [l=50e-6  oz=-90   w=5e-6  R=500]
b3 beam2d  p1 [7  9]  [l=150e-6 oz=0     w=2e-6  R=1000]
b4 beam2d  p1 [8  13] [l=150e-6 oz=0     w=2e-6  R=1000]
b1m beam2d  p1 [9  10] [l=50e-6  oz=0     w=20e-6 R=100]
b2m beam2d   p1 [10 16] [l=50e-6  oz=0     w=20e-6 R=100]
b3m beam2d   p1 [10 11] [l=75e-6  oz=-90   w=20e-6 R=100]
```

```
b4m beam2d   p1 [11 12] [l=75e-6  oz=-90   w=20e-6 R=100]
b5m beam2d   p1 [13 12] [l=50e-6  oz=0     w=20e-6 R=100]
b6m beam2d   p1 [12 17] [l=50e-6  oz=0     w=20e-6 R=100]
bh  beam2d   p1 [11 14] [l=300e-6 oz=0     w=2e-6  R=1500]
bm  beam2d   p1 [14 15] [l=196e-6 oz=0     w=116e-6 R=100]
```

An input excitation of sinusoidal force is applied on the resonator as shown as following: The bode plot of the y direction response at the mass is shown below.

4.1.6 Transient analysis (demo_ta1gap)

This is a 3D electromechanical transient analysis for a gap-closing actuator. A piecewise linear voltage V(t) is applied across the page. The voltage V(t) ramps from 5V at t=10usec to 12V at t=500usec, and then drops to 0V. The displacement component of node C in the direction of force is observed below. The initial voltage step causes the device to oscillate. At the voltage increases at a linear rate, the gap decreases at a nonlinear rate due to the electrostatic force increasing proportionally to $1/gap(q)^2$. This force also causes the period of oscillation to increase. Once the voltage is removed, the actuator exponentially decays back to equilibrium due to viscous Couette air damping between the beams and the substrate.

Netlist

```
uses mumps.net
%beam and it's anchor:
anchor p1 [a] [l=5u w=10u oz=180  ox=0 oy=0 h=10u]
beam3d p1  [a b] [l=100u w=4u oz=0  ox=0 oy=0 h=4u]
%electrostatic
gapV2 p1 [b c d e] [V1=5 t1=10e-6  V2=12 t2=500e-6 l=100u w1=4u w2=4u oz=0
ox=0 oy=0 h=4u gap=2u]
%gap anchors
anchor p1 [d] [l=5u w=10u oz=-90 ox=0 oy=0 h=10u]
anchor p1 [e] [l=5u w=10u oz=-90 ox=0 oy=0 h=10u]
```

Chapter 5

Available models

(Taken from Jason's appendix A)

5.1 Available models

beam2d planar mechanical beam

beam2de planar mechanical beam and electric resistor

beam3d 3D mechanical beam

beam3de 3D mechanical beam and electronic resistor

anchor 0D mechanical fixed node

f2d planar force or moment

f3d 3D force or moment

gap2de two planar electrostatic mechanical beams, resistors

gap3de two electrostatic 3D mechanical beams, resistors

Vsrc Voltage source

eground Electronic ground

comb2d N-finger electrostatic comb, 2D mechanical

R constant resistor

Isrc constant current source

nmos nmos model

pmos pmos model

5.2 Model descriptions and interfaces

5.2.1 beam2d

Describes an in-plane beam connecting two nodes.

Example:

```
beam2d p1 [A B][l=100u w=5u oz=10]
```

Nodal variables:

{x, y, rz} at both nodes

Parameters

l beam length in meters (required)

w beam width in meters (required)

h thickness of beam in meters (optional; supplied in process info)

oz initial rotation about beam's z-axis (required if not 0)

5.2.2 beam2de

Similar to beam2d but adds electronic resistance to the beam.

Example

```
beam2d p1 [A B][l=100u w=5u oz=10 R=100]
```

Nodal variables

{x, y, rz, e} at both nodes

Parameters

l beam length meters (required)

w beam width in meters (required)

R beam resistance in ohms (required)

h thickness of beam in meters (optional; supplied in process info)

oz initial rotation about beam's z -axis (required if not 0)

5.2.3 beam3d

Similar to beam2d but can be rotated out-of-plane.

Example

```
beam3d p1 [A B][l=100u w=5u oy=20 oz=10 ox=45]
```

Nodal variables

{x, y, z, rx, ry, rz} at both nodes

Parameters

l beam length meters (required)

w beam width in meters (required)

h thickness of beam in meters (optional; supplied in process info)

oy initial rotation about y-axis (required if not 0)

oz initial rotation about beam's z -axis (required if not 0)

ox initial twist about the beam's x-axis (required if not 0)

5.2.4 beam3de

Similar to beam3d but adds electronic resistance to the beam.

Example

```
beam3d p1 [A B][l=100u w=5u oy=20 oz=10 ox=45 R=100]
```

Nodal variables

{x, y, z, rx, ry, rz, e} at both nodes

Parameters

l beam length meters (required)

w beam width in meters (required)

R beam resistance in ohms (required)

h thickness of beam in meters (optional; supplied in process info)

oy initial rotation about y-axis (required if not 0)

oz initial rotation about beam's z -axis (required if not 0)

ox initial twist about the beam's x-axis (required if not 0)

5.2.5 anchor

Describes a mechanically fixed node. 3D or 2D.

Example

```
anchor p1 [A B][l=10u w=10u h=10u]
```

Nodal variables

{x, y, z, rx, ry, rz} at both nodes

Parameters

l beam length meters (required)

w beam width in meters (required)

h thickness of beam in meters (optional; supplied in process info)

oy initial rotation about y-axis (required if not 0)

oz initial rotation about beam's z -axis (required if not 0)

ox initial twist about the beam's x-axis (required if not 0)

5.2.6 f2d

Describes an in-plane external force at a node.

Example

```
f3d * [A][F=10u oz=45]
f3d * [A][M=1u oz=45]
```

Nodal variables

{x, y, rz} at node

Parameters

F force in Newtons (required if M is not used)

M moment in Newton-meters (required is F is not used)

oy initial rotation about y-axis (required if not 0)

oz initial rotation about the vetors's z -axis (required if not 0)

5.2.7 f3d

Describes a 3D external force at a node.

Example

```
f3d * [A][F=10u oy=35 oz=45]
f3d * [A][M=1u oy=35 oz=45]
```

Nodal variables

{x, y, z, rx, ry, rz} at node

Parameters

F force in Newtons (required if M is not used)

M moment in Newton-meters (required is F is not used)

oy initial rotation about y-axis (required if not 0)

oz initial rotation about the vectors's z -axis (required if not 0)

5.2.8 gap2de

Describes a 2D electrostatic gap, which consists of two electronic, mechanical beams.

Example

```
gap p1 [a b c d][l=100u w1=5u w2=5u oz=0 gap=2u R1=100 R2=100]
```

Nodal variables

{x, y, z, rx, ry, rz, e} at all four nodes

Parameters

l beam length meters (required)

w1 beam1 width in meters (required)

w1 beam2 width in meters (required)

gap initial gap spacing

h thickness of both beams in meters (optional; supplied in process info)

R1 beam1 resistance in ohms (required)

R2 beam2 resistance in ohms (required)

oz initial rotation about beam1's z -axis (required if not 0)

5.2.9 gap3de

Describes a 2D electrostatic gap, which consists of two electronic, mechanical beams.

Example

```
gap p1 [a b c d][l=100u w1=5u w2=5u oz=0 gap=2u R1=100 R2=100]
```

Nodal variables

{x, y, z, rx, ry, rz, e} at all four nodes

Parameters

l beam length meters (required)

w1 beam1 width in meters (required)

w1 beam2 width in meters (required)

gap initial gap spacing

h thickness of both beams in meters (optional; supplied in process info)

R1 beam1 resistance in ohms (required)

R2 beam2 resistance in ohms (required)

oy initial rotation about y-axis (required if not 0)

oz initial rotation about beam1's z -axis (required if not 0)

ox initial twist about the beam1's x-axis (required if not 0)

5.2.10 Vsrc

Describes a voltage source.

Example

```
Vsrc * [e d] [V=5 sv=0.1 sph=0]
```

See Fig 5 for nodes. Note: only mechanical element display.

Nodal variables

{e} at both nodes

Parameters

V voltage in volts (required)

5.2.11 eground

Describes a electronic ground.

Example

```
eground * [e] []
```

See Fig 5. Note: only mechanical element display.

Nodal variables

{e} at nodes

Parameters

none

Chapter 6

Function reference

6.1 Load netlist

Calling sequence

```
function [net] = cho_load(name, params);
```

Loads and processes a netlist.

Inputs

`name` String naming the netlist file to be loaded

`params` (Optional) - a structure whose entries are the values of the parameters to be overridden.

Example

```
params.nfingers = 10;                % Set the nfingers parameter
net = cho_load('comb.net', params);  % Load netlist
```

6.2 Device display

Calling sequence

```
function cho_display(net, q);
```

Display the mechanical structure described by a netlist.

Inputs

`net` Netlist structure returned from calling cho_load

`q` (Optional) - the displacement of the original structure, as returned by the static analysis routine or the transient analysis routine. If `q` is unspecified, the undisplaced structure will be displayed.

Example

```
net = cho_load('beamgap.net');  % Load netlist for beam-gap system
cho_display(net);               % Display undisplaced structure
```

Caveats

There is no display of electrical components.

6.3 Viewing displacements

Calling sequence

```
function [dqcoord] = cho_dq_coord(dq, net, node, coord);
```

Extract the displacement corresponding to a particular coordinate from a displacement vector q (as output by `cho_dc`), or an array of displacement vectors (as output by `cho_ta`).

Note that a node "node" completely internal to an instance "foo" of some subnet would be named "node.foo".

Inputs

`dq` : Displacement vector or array of displacement vectors.

`net` : Netlist structure returned from calling cho_load

`node` : Name of node.

`coord` : Name of coordinate at indicated node.

Outputs

`dqcoord` : Value (or vector of values) from `dq` associated with the indicated coordinate.

Example

```
net = cho_load('beamgap.net');       % Load netlist
dq = cho_dc(net);                    % Perform static analysis
dy = cho_dq_view(Q, net, 'c', 'y');  % Get the y component at c
```

6.4 Static analysis

Calling sequence

```
function [q] = cho_dc(net, q0, is_sp)
```

Finds a solution of the equilibrium equations

$$Kx = F(x)$$

using a Newton-Raphson method.

Inputs

`net` : Netlist structure returned from calling cho_load

`q0` : (Optional) Starting guess at the displacements from starting position for an equilibrium position. If no `q0` is provided, or if `q0 = []` the search will start at `q0` of 0.

`is_sp` : (Optional) If true, the code will use sparse solvers. The current default is true (use sparse solvers).

Outputs

`q` : The computed equilibrium state, expressed as displacements from the initial position.

Example

```
net = cho_load('beamgap.net');  % Load netlist for beam-gap system
dq = cho_dc(net);               % Perform static analysis
cho_display(net, dq);           % Display the deflected structure
```

Caveats

The zero-finder currently used by SUGAR is very simple, and may fail to converge for some problems. If the function fails to converge after 40 iterations, it will exit and print a warning diagnostic:

```
Warning: DC solution finder did not converge after 40 iterations
```

In this case, the result `q` returned by the routine is likely to be useless. Failure to find equilibrium occurs in such cases as an electrostatically actuated gap operating near pull-in voltage.

When SUGAR finds an equilibrium point, it may not always be the equilibrium point desired. In the case of an electrostatically actuated gap, for example, there are two equilibria below pull-in voltage: one stable and one unstable. When the equilibria are close together, especially with respect to the distance from the starting point `q0`, the solver may move to an unstable equilibrium.

Good initial guesses for an equilibrium point can often be attained by finding the equilibrium point for a "nearby" problem. For example, in trying to find the equilibrium point for a electrostatically actuated gap operating near pull-in, a good initial guess `q0` would be the output of a static analysis for the same device at a lower voltage.

6.5 Modal analysis

Analysis routine

Calling sequence

```
function [freq, egv, q0] = cho_mode(net, nmodes, q0, find_dc);
```

Find the resonating frequencies and corresponding mode shapes (eigenvalues and eigenvectors) for the linearized system about an equilibrium point.

Inputs

`net` : Netlist structure returned from calling cho_load.

`nmodes` : (Optional) If `nmodes > 0`, use sparse solvers to get nmodes modes If `nmodes == 0`, use the usual dense solver to get all the modes If `nmodes < 0`, solve with `eig(K \ M)` rather than `eig(M,K)`. This last option can potentially cause trouble (for instance, if K is singular), but it is faster.

`q0` : (Optional) Equilibrium operating point, or initial guess for a search for an equilibrium operating point. If not supplied, of if `q0 = []`, the routine will search for an equilibrium point near 0.

`find_dc` : (Optional) If true, search for an equilibrium point near the supplied q0.

Outputs

freq : Vector of resonating frequencies (eigenvalues)

egv : Array of corresponding mode shapes (eigenvectors)

q : Equilibrium point about which the system was linearized

Display routine

Calling sequence

`function cho_modeshape(net, freq, egv, q0, s, num);`

Display the shape of a resonating mode of the mechanical structure.

Inputs

`net` : Netlist structure returned from calling cho_load.

`freq` : Vector of resonant frequencies from `cho_mode`.

`egv` : Array of mode shape vectors from `cho_mode`.

`q0` : Equilibrium point from `cho_mode`.

`s` : Scale factor. Eigenvectors from `cho_mode` are normalized to be unit length; for eigenvectors with significant components (within a few orders of magnitude of 1) in directions corresponding to components which normally move a few microns, a scale factor of 10^{-4} often makes the display of the mode more comprehensible.

`num` : Number of the mode to display. Modes are numbered in order of decreasing frequency.

Example

```
% Show the first (lowest-frequency) mode shape for the system,
% scaled by a factor of 0.1
net = cho_load('beamgap.net');
[freq, egv, q] = cho_mode(net);
cho_modeshape(net, freq, egv, q0, 0.1, 1);
```

Caveats

While SUGAR will attempt to find an appropriate linearization point, it is not guaranteed to converge to one. See the caveats for static analysis.

The modal analysis routine neglects damping forces.

As noted above, using `cho_modeshape` with a too-large scaling factor often results in the displayed device being stretched to incomprehensible proportions. Currently, trial-and-error guesses at an appropriate scale factor seem to work best.

6.6 Steady state analysis

Calling sequence

```
function find_ss(net, q0, in_node, in_var, out_node, out_var)
```

Make Bode plots of the frequency response of the linearized system about an equilibrium point `q0`.

Inputs

`net` : Netlist structure returned from calling cho_load

`q0` : Equilibrium position for the system, as determined via the static analysis routine `cho_dc`

`in_node` : Name of the node at which an input signal is to be applied.

`in_var` : Name of the nodal variable to be excited.

`out_node` : Name of the node where the response is to be observed.

`out_var` : Name of the nodal variable to be observed.

Example

```
net = cho_load('multimode.net');
dq = cho_dc(net);
find_ss(net, dq, 'node10', 'y', 'node14', 'y');
```

Caveats

Steady-state frequency response analysis currently fails for devices involving purely algebraic constraint. Such devices include electrical resistor networks with no inductances or capacitances considered, for example.

The steady-state analysis routines currently use functions from Matlab's Control Toolbox, which may be unavailable to some Matlab users.

6.7 Transient analysis

Calling sequence

```
function [T,Q,C,G] = cho_ta(net,tspan,q0)
```

Simulate the behavior of the device over some time period.

Inputs

`net` : Netlist structure returned from calling cho_load

`tspan` : Two-element vector `[tstart tend]` indicating the start and end times for the simulation.

`q0` : (Optional) Initial state at `tstart`. If `q0` is not provided, the default is zero.

Outputs

`T` : Time points where the solution was sampled

`Q` : Array of state vectors sampled at the times in `T` (i.e. `Q(i,:)` is the state vector at time `T(i)`)

Example

```
net = cho_load('beamgap.net')        % Load the netlist
[T,Q] = cho_ta(net,[0 1e-3]);        % Simulate 1 ms behavior
dy = cho_dq_view(Q, net, 'c', 'y');  % Get the y component at c, and
plot(T, dy);                         %  plot how it moves over time
```

Caveats

The transient analysis routines currently take an impractically long time to simulate even some simple examples over modest time spans (like a millisecond). Mixed electrical-mechanical simulations are particularly problematic.

Like frequency-response analysis, the transient analysis routine fails completely for devices involving purely algebraic constraints.

Chapter 7

Netlist formal grammar

Below is the formal grammar for the netlist language. This grammar has undefined symbols *ID*, *INT*, *FLOAT*, and *STRING* corresponding to identifiers, integer constants, floating-point constants, and string constants respectively. Literal symbols are in `typewriter` style, and ϵ denotes an empty string. This is the grammar used in the YACC netlist parser, with only minor modifications for readability.

netlist:
    netlistDefs

netlistDefs:
    netlistDefs netlistDef
    netlistDef

netlistDef:
    paramLine
    processLine
    subnet
    codeLine

paramLine:
    `param` *paramList*

paramList:
    param
    paramList `,` *param*

param:
    def
    ID

processLine:
    `process` *ID* `=` `[` *defs* `]`
    `process` *ID* `:` *ID* `=` `[` *defs* `]`

subnet:
    subnetHead codeBlock

subnetHead:
    `subnet` *ID* `[` *names* `]` `[` *subnetParams* `]`

subnetParams:
    subnetParams subnetParam
    ϵ

subnetParam:
    def
    ID `=` `*`

codeLines:
    codeLines codeLine
    codeLine

codeLine:
    codeBlock
    def
    elementLine
    forLine
    ifLine

codeBlock:
    `[` *codeLines* `]`

elementLine:
    elementHead `[` *names* `]` `[` *defs* `]`

elementHead:
    ID `(` *exprs* `)` *ID ID*
    ID ID ID
    ID `(` *exprs* `)` *ID* `*`
    ID ID `*`
    ID ID
    ID `*`

forLine:
    forHead codeBlock

forHead:
    for ID `=` *expr* `:` *expr*

ifLine:
    `if` *expr codeLine*

`if` *expr codeLine* `else` *codeLine*

def:
ID `=` *expr*

defs:
defs def
ϵ

names:
names name
ϵ

name:
ID
ID `(` *exprs* `)`

expr:
expr `+` *expr*
expr `-` *expr*
expr `*` *expr*
expr `/` *expr*
expr `^` *expr*
`-` *expr*
`!` *expr*
expr `&` *expr*
expr *expr*
expr `==` *expr*
expr `!=` *expr*
expr `>` *expr*
expr `<` *expr*
expr `>=` *expr*
expr `<=` *expr*
`(` *expr* `)`
INT
FLOAT
STRING
ID
ID `(` *optexprs* `)`

exprs:
exprs `,` *expr*
expr

optexprs:
exprs
ϵ

Comments and `uses` statements are implemented with the tokenizer rather than as part of the parser. Comments begin at a `%` character and extend to the end of the line. `uses` statements must appear on a line of their own, and consist of the keyword `uses` followed by the name of the file to be used, optionally including a path.

Part II

SUGAR Implementor's Guide

Chapter 8

Code architecture and source organization

There are three major components to SUGAR:

- Model functions which describe how to check parameters, write equations, and display individual elements of a device.
- Routines to assemble and analyze device equations and display results; and
- A parser and preprocessor, written using C and the compiler tools bison and flex;

The three components are respectively in the `model`, `analysis`, and `compile` subdirectories of the SUGAR distribution. We have listed the pieces in order from "most likely to be modified by a casual user" to "least likely to be modified by a casual user." It seems likely that many users will want to add their own model functions, and the code is organized so that it should be possible to do this without learning about any of the rest of the code base. Some users may want to try out new forms of analysis, or implement new solvers; in order to do this, they will need a working knowledge of how to work with the Matlab representation of the device, and how analysis routines interact with the model functions. The brave few who decide to extend the netlist language, or otherwise modify SUGAR in a fundamental way will need to understand the parser and preprocessor routines along with everything else. However, we have tried to keep the design modular, so that it is possible to change pieces of the code without understanding the entirety.

(Still need to describe the communication between the major components of the system, perhaps draw a pretty picture, and tell users which chapter they need to look at to make which common extension. But even with that, this will be a short section.)

Chapter 9

Model function interface

9.1 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...

9.2 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.

9.3 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.

9.4 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
```

9.5 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.

9.6 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.

9.7 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
```

9.8 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);
```

Chapter 10

Matlab structures, assembly, and analysis

10.1 The netlist data structure

The parsed representation of the netlist returned by `cho_load` is one of SUGAR's central data structures. The netlist structure contains the following fields:

- elements(i) Information structure for the ith element.
  - name name of element
  - model model function name
  - node_ids indices of nodes involving this element
  - parameter structure mapping model parameters to value
  - var_ids list of indices of variables for this element. Grounded variables are assigned index 0.
- nodes(j) Information for the jth node.
  - name name of node. An element with branch variables has a dummy node with the same name as the element, and any branch variables are assigned to that node.
  - elt_ids indices of elements involving this node
  - vars structure mapping variable names to indices
  - pos coordinates of the undisplaced node (mechanical nodes only)
- dof number of (ungrounded) degrees of freedom
- scales(k) characteristic size of the kth ungrounded variable

The final netlist data structure does not have the same heirarchy as the original design, which might contain subnets and array constructs. The only way that the structure of the original netlist is portrayed in the final data structure is in the node and element name fields. Element names have the form

```
subnet element.subnet element. ... .element.
```

For example, an element named bar in a subnet instance named foo would be called foo.bar. Similarly, the structure of a node name is

```
subnet element.subnet element. ... .node.
```

A single node may have several aliases if it is used as an argument to a subnet. For instance, in the netlist fragment

```
subnet silly_anchor [x] [l=* w=*]
[ anchor parent [x] [l=l w=w]
]

silly_anchor an_anchor p1 [A] [l=5u w=5u]
```

the node A could also be called an anchor.x. In such cases, the name assigned to the node is the name at the highest scoping level; in this example, A. Elements which are not explicitly named are assigned unique names of the form anon plus a number. Anonymous elements use the same scoping rules as normal elements, so that an anonymous element inside an anonymous subnet instance has a name like anon5.anon10.

10.2 Assembly and Display

The assembly and display routines have very simple (and very similar) structure. Each routine loops through all the elements in turn, calls the model function to get a local contribution, and then merges the local contribution into a global structure. The body of assemble system is a typical example. Here net is the netlist data structure and mflag is a flag describing whether the mass ('M'), damping ('D'), or stiffness ('K') matrix should be assembled:

```
for i = 1:length(net.elements)
  elt = net.elements(i);

  % Get the local contribution
  [Mlocal] = feval( elt.model, mflag, elt.R, elt.parameter );

  % If this element has anything to contribute, incorporate it
  if (~isempty(Mlocal))

       j = find(elt.var_ids ~= 0);            % Find ungrounded variables
       jdx = elt.var_ids(j);                  % Get associated global indices

       M(jdx,jdx) = M(jdx,jdx) + Mlocal(j,j); % Add local contribution

  end
end
```

Note the use of two different indexing systems. The element "stamps" returned by the model functions are ordered according to a local variable ordering cor- responding to the order in which variable names appear in the output of the 'vars' case. Entry i in the var ids field then gives the global index for the ith local variable. However, some local variables may be grounded, and

therefore will not have a corresponding global index. The matrix rows and columns cor- responding to grounded variables, represented in var ids by zero entries, are not added into the global matrix. The matrix assembly routines take a flag is sp to indicate whether the matrices should be assembled using Matlab's sparse data structures. If the is sp flag is ommitted, sparse output is assumed by default. There are only a few assembly functions:

- assemble_system assembles the linear mass, damping, and stiffness matrices
- assemble_F assembles the forcing term
- assemble_dFdx assembles the Jacobian with respect to the position variables
- assemble_dFdxdot assembles the Jacobian with respect to velocity variables
- cho_display assembles local display output into a complete device picture

Additionally, the structure of the netlist checking function check netlist is similar to the structure of the assembly routines.

Chapter 11

Parsing and pre-processing

(Taken from implementor.pdf – out of date)

In order to easily support a more powerful netlist language, the parser in SUGAR 2.0 was re-written using the UNIX tools flex and bison. flex and bison are GNU versions of the classic lex and yacc compiler construction tools. For those who wish to understand or modify the parser code, a good source of information on the tools used can be found in the book lex & yacc by John Levine, Tony Mason, and Doug Brown, published by O'Reilly and Associates. Online resources on flex and bison include the GNU man pages and info pages. In the current version of SUGAR, the cho load command to load netlists basically proceeds in two phases. In the first phase, the yacc translator, which is compiled into an external Matlab routine (MEX file) converts the user netlist into a Matlab function (called nettemp.m by default). In the second phase, the Matlab function is executed to partially build the final data structure. After that, additional Matlab routines process the data structure to assign global variable indices, compute the undisplaced positions of the mechanical nodes, and sanity check the model function parameters.

11.1 Translator structure

11.1.1 Scanning

The scanner, sugar.lex, is responsible for tokenizing the input file and for managing the inclusion of files via the uses statement. The token description is straightforward, and the curious reader is referred to the source code for further details. The handling of uses statements is slightly more complicated.

A SUGAR uses statement is a combination of the uses statement in a lan- guage like Delphi (Object Pascal) and the C #include statement. The text of a file included via uses is processed only at the first place it is encountered. So, for example, if subnets.net uses mumps.net, and foo.net uses both subnets.net and mumps.net, the text of mumps.net will only be used once. Consequently, the scanner file keeps two structures to keep track of uses statements: a stack which keeps track of nested uses, and a list of files which have been included already.

Before attempting to open a file for inclusion, the scanner calls the function which file, defined in the general library file sugar lib.c. When called from within a MEX file, which file scans the Matlab path for files of the given name.

11.1.2 Parsing and intermediate representation

The parser file sugar.y is little more than a copy of the formal grammar for the SUGAR netlist language. The parser actions call routines in parse.c in order to build an intermediate representation of the netlist parse tree. The intermediate representation is described in parse.h.

The error checking done in the current version of the parser is very rudi- mentary. Besides the automatic detection of parse errors, the parsing routines check only for undefined variables and invalid process layers.

11.1.3 Matlab code generation

The final output of the SUGAR netlist translator is a Matlab script. The task of the code generation routines in codegen.c is to recursively traverse the intermediate representation of the netlist structure built by sugar.y and parse.c and output Matlab code that will create appropriate corresponding data structures.

Perhaps the most confusing aspect of the current code generation code is its treatment of subnets. When the code generator starts on a subnet instance, it stores relevant state, such as the name and coordinate system associated with the subnet, or the assignment of local node names to global node indices, into several different places. Some such information is kept with the parse tree data structure; other information is kept in local variables in the run-time stack of the generator code. Work is underway on a version of the code generator which keeps a separate stack for subnet state, similar to the runtime stack kept by most modern languages.

11.2 Matlab post-translation processing

11.2.1 Argument sanity checks

The first step after the generated Matlab script generates a partial version of the netlist data structure is to sanity check the input arguments. This is done by the check netlist routine, which calls the 'check' clause of each model function in turn. The model functions are responsible for returning a diagnostic message if they lack some piece of information needed for later analysis, or if the arguments they receive are inconsistent or out of range.

11.2.2 Index assignment

Global variable index assignment proceeds in two steps. First, grounded variables are identified and marked. Then, nodal and branch variables which were not identified as grounded in the first phase are assigned indices. Index assignment is done in the file parse enrich2.m.

In the current implementation, index assignment requires some string comparisons, which slows it down substantially. Index assignment and node positioning take substantially more time than other parts of the netlist loading process. This is likely to change in future versions, as more of the phases that are currently done by post-processing move into the code generator.

11.2.3 Node positioning

Before determining where nodes should be located, the node positioning code determines which nodes should be located. This is done by scanning through the list of elements and determining which nodes are assigned a relative position by the 'pos' clause of some element. It is possible for a single model function to contain some mechanical nodes (which have positions) and some nodes

which do not have an associated position. In this case, the number of columns returned by the 'relpos' clause of the model will be smaller than the number of nodes.

The node positioning routine then does a breadth-first traversal of the position graph. The first node visited is arbitrarily assigned to be positioned at the origin, and subsequently visited nodes are assigned locations based on the relative position information contained in the model functions and on the already-determined locations of their neighboring nodes. If the mechanical nodes are not all in a single connected component, the positioning routine will issue a warning message and position at the origin the first node it encounters in each component.

Ideally, an additional pass after node positions were determined would check that the locations were consistent with the relative position information for every element. Such a geometry check is not yet implemented.

Node positioning is implemented in find pos.m.