**SUGAR II v3.3**

**A Design, Modeling, and Simulation Package for Micro Electro Mechanical Systems**

(User’s guide)

Richa Bansal, Prabhakar Marepalli, Jason V. Clark

Electrical and Computer Engineering, Mechanical Engineering

Purdue University

**Revision Date: March 04, 2011**

Table of Contents

1. Introducing Sugar 5

1.1 What is Sugar? 5

1.2 A first example 5

1.3 Installing SUGAR 6

1.3.1 System requirements 7

1.3.2 Setting SUGAR paths 7

1.3.3 Compiling external routines 7

1.4 Getting (and giving) help 7

2. Describing Devices 8

2.1 Units and metric suffixes 8

2.2 Expressions 8

2.3 Rotations and Euler angles 10

2.4 Lexical notes 11

2.5 uses statements 12

2.6 Element Lines 12

2.7 Parameters and definitions 13

2.8 Process parameter structures 14

2.9 Subnet statements 15

2.10 Arrays 16

2.11 Conditionals 17

3. Analyzing devices 19

3.1 Types of analysis 19

3.2 Static analysis 19

3.3 Steady state and Modal Analysis 21

3.4 Transient Analysis 22

3.5 Future analysis routines 22

4. Available models 22

4.1 Available models 22

4.2 Model descriptions and interfaces 23

4.2.1 beam2d 23

4.2.2 beam2de 23

4.2.3 beam3d 24

4.2.4 beam3de 25

4.2.5 anchor 25

4.2.6 beam3dlink 26

4.2.7 circbeam 26

4.2.8 f2d 27

4.2.9 f3d 28

4.2.10 gap2d 28

4.2.11 Vsrc 29

4.2.12 Isrc 29

4.2.13 eground 30

4.2.14 R 30

4.2.15 L 30

4.2.16 C 31

5. Function reference 31

5.1 Load netlist 31

5.2 Device display 32

5.3 Viewing displacements 33

5.4 Static analysis 33

5.5 Modal analysis 34

5.5.1 Analysis routine 34

5.5.2 Display routine 35

5.6 Steady state analysis 36

5.7 Transient analysis 37

6. Examples with explanations 38

6.1 Cantilever example 38

6.2 Multiple beam example 41

6.3 Beam3dlink example 42

6.4 Array definition (demo array\_use) 43

6.5 Parameter definition (demo param\_use) 43

6.6 Beam gap structure (demo beamgap) 45

6.7 Modal analysis (demo mirror) 46

6.8 Steady state analysis (demo\_ss) 50

7. More examples 52

7.1 V-beam thermo-mechanical actuator (ETA\_netlist\_demo) 52

7.2 Carbon nanotube (CNT) structure 55

7.2.1 Armchair structure 55

7.2.2 Zigzag structure 57

8. Code architecture and source organization 58

9. Model function interface 59

9.1 Model overview 59

9.2 Model function arguments 60

9.3 Variable definitions 60

9.4 Parameter checking 61

9.5 Local matrices for linear terms 62

9.6 Local force contributions 63

9.7 Node positioning 64

9.8 Element display 64

10. Matlab structures, assembly, and analysis 65

10.1 The netlist data structure 65

10.2 Assembly and Display 66

11. Parsing and pre-processing 67

11.1 Translator structure 68

11.1.1 Scanning 68

11.1.2 Parsing and intermediate representation 68

11.1.3 Matlab code generation 68

11.2 Matlab post-translation processing 69

11.2.1 Argument sanity checks 69

11.2.2 Index assignment 69

11.2.3 Node positioning 69

12. SugarCube 70

12.1 What is SugarCube? 70

12.2 SugarCube Graphical User Interface 70

12.2.1 MEMS display window 70

12.2.2 Parameterization and optimization window 70

12.2.3 Plot window 72

12.3 Parameterization of MEMS 72

12.3.1 Static analysis 72

12.3.2 Modal analysis 74

12.3.3 Transient analysis 74

12.4 Optimization of MEMS 75

References 77

# Introducing Sugar

## Whatis 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.

## 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, which describes the device. We save this netlist in the file called 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=pi/2]

The first line tells SUGAR that our device is made using the standard MUMPS process. The second tell SUGAR that the device is made of an anchor at node A, which is ten microns in length and 10 microns in width. The third line tells SUGAR that the anchor is attached to a 3 dimensional beam that is two microns wide and one hundred microns long. The beam goes from the anchored end point A to the free end B. Both the anchor and the beam are made of the poly1 layer, orp1that is defined in the process file mumps.net. 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);

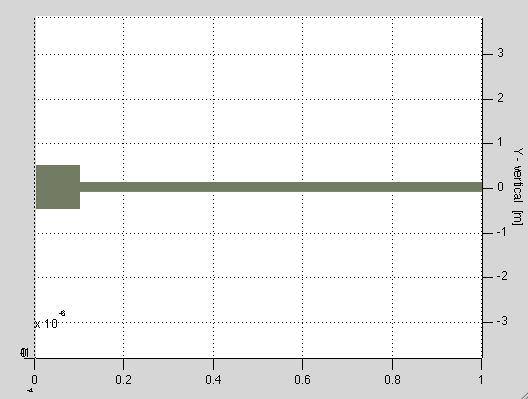


Figure 1: Cantilever beam in SUGAR

The cho\_load 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\_dcfunction 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.

## 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/>

### System requirements

To use 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.

### 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

### 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 compilesubdirectory and from Matlab type makemex. Then copy all the files beginning with sugar\_cto the analysis subdirectory.

## 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](mailto:cfm@bsac.eecs.berkeley.edu)

We will try to respond promptly. SUGAR is research software; if you would like to contribute models, analysis routines, orexamples to the SUGAR project, let us know that, too!

# Describing Devices

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

## 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 parameter values defined in SUGAR. For example, a 100 micron length in SUGAR could be represented as 100u, 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; 0.1e7uis a perfectly legitimate (if somewhat silly) way of writing the number 1.

The standard suffixes are:

Table 1: Units and metric suffices

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| d | deci | 10-1 | h | hecto | 102 |
| c | centi | 10-2 | k | kilo | 103 |
| m | milli | 10-3 | M | mega | 106 |
| u | micro | 10-6 | G | giga | 109 |
| n | nano | 10-9 | T | terra | 1012 |
| p | pico | 10-12 | P | peta | 1015 |
| f | femto | 10-15 | E | exa | 1018 |
| a | atto | 10-18 |  |  |  |

## Expressions

It is often convenient to do simple calculations inside a netlist. For example, suppose wehave defined a variablebeamL for the length of a beam. 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, negate numbers, and also evaluate functions. We have comparison operators: ==, !=, <=, >=, <, and >; and 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 as follows:

Table 2: Precedence order of expressions in SUGAR

|  |  |
| --- | --- |
| Level | Operators |
| 9 | | (logical or) |
| 8 | & (logical and) |
| 7 | == (equality), != (inequality), > (greater), < (less), >= (greater or equal), <= (less or equal) |
| 6 | - (subtract) |
| 5 | + (add) |
| 4 | \* (multiply) |
| 3 | / (divide) |
| 2 | - (negate), ! (logical not) |
| 1 | ^ (exponentiate) |

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: for example, "\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 in SUGAR as well. 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. Thisfunction is provided purely for debugging convenience.

## Rotations and Euler angles

In order to orient structures in SUGAR, users need to 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 radians; these three numbers are known as *Euler angles*, and can be used to describe any rotation.

For example, the following netlist describes 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.

uses mumps.net

beamL = 100u

beam3d p1[A B][w=2u l=beamL]

beam3d p1[B C][w=2u l=beamL/2 oz=pi/2]

beam3d p1[C D][w=2u l=beamL oz=pi]

beam3d p1[D A][w=2u l=beamL/2 oz=-pi/2]

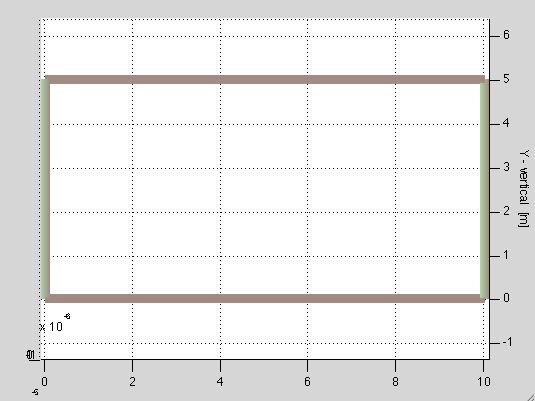


Figure 2: Rectangle

A thing to note is that if no rotation angle is specified, then the beam will be oriented in the default directionalong the *x* axis. Also, all angle are measured from the default orientation, that is, *x*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.

## 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 following two netlists are equivalent.

uses mumps.net

beam2d p1

[A B] [l = 100u

w = 100u]

and

uses mumps.net

beam2dp1[A B][l=100u w=100u]

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.

## uses statements

Netlists can contain uses statements to include other files. A uses statementhas 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. If a subnet is used in a netlist, then that file needs to be included in the netlist using uses statement. For example:

uses mumps.net

uses serpent\_subnet.net

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 mumps.net

uses subnets.net

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

## Element Lines

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

crossbeam beam2dp1[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 isnamed 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, circular beams, electrical devices, etc. in Sugar library; a complete list of models, along with information on how to build new models, can be found in the sections that follow.
* The third field is the name of the process parameter structure. In this case the beam is fabricated in the first layer of poly-silicon in a MUMPS process, named p1. This mumps process is available in the model mumps.net. By specifying the process layer p1, a user specifies the thickness of the deposited layer to be the default value 2e-06. For models that 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, enclosed in brackets. In above example, the specified beam connects nodes A and B. Elements are connected together by sharing a common node. For instance, to attach a 100 micron by 5 micron beam to the B end of the first beam, we might write

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

Unlike in previous versions of SUGAR, node names in SUGAR 3.0 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 may require other parameters. A parameter specification always has the form identifier = expression.

## Parameters and definitions

In order to allow user defined parameters or to experiment with variations on a simulation, SUGAR supports named parameters. For example:

param nfingers, length1=10u

A parameter definition consists of the param keyword, followed by a comma separated list of parameter name and default values. Default values are optional; however, it is an error to use the netlist without setting any parameters without defaults. If no default values are given, then the parameter value must be obtained through the user. 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 (Section 2.10)will not be visible outside the subnet and hence will not affect top-level element statements.

## 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 = 1.3 W cm-1∞C-1.

viscosity = 1.78e-5 %Viscosity (of air) 1.78e-05

fluid = 2e-6 %Fluid layer thickness between device andsubstrate

density = 2300 %Material density = 2300 kg/m3

Youngsmodulus = 165e9 %Young’s modulus = 165e09 N/m2

permittivity = 8.854e-12 %Permittivity=8.854e-12 = (C.s)2/kg.um3

sheetresistance = 20 %Poly-Si sheet resistance = ohm/square

straingradient=0 %Strain gradient = /m

thermalexpansion=2.6e-06 %Thermal expansion of Si = C-1

ambienttemperature=0 %Ambient temperature = K

]

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.

## Subnet statements

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 is shown below

subnet serpent[A E][unitwid=\* unitlen=\* beamw=2u]

[

len2 = unitlen/2

beam2d parent [A b] [l=unitwid w=beamw oz=-pi/2]

beam2d parent [b c] [l=len2 w=beamw]

beam2d parent [c d] [l=unitwid w=beamw oz=pi/2]

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 serpentp1[x y][unitwid=10u unitlen=10u]

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

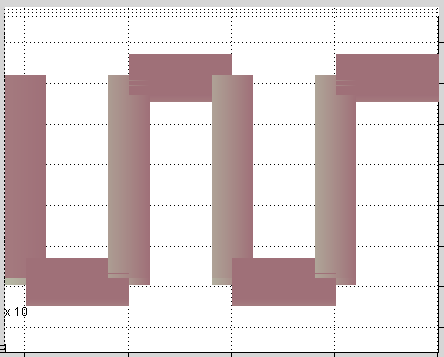


Figure 3: Use of subnet

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 section 2.11).Note that giving a name to the subnet instance (serp1) is optional. However, it might be useful when referring to the internal nodes of the subnet (see below).

For the subnet instance above, only nodes A and E are visible outside the subnet. It is possible to have as many nodes visible as needed by mentioning them in the subnet definition. Sometimes it may be necessary to access variables attached to nodes internal to a netlist, which are not otherwise visible outside the subnet. 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.

## 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 anchorp1[x(1)][l=5u w=5u]

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), x(i,j)). The index variable ‘i’is only valid within the loop body. The array produces the following serpentine structure:

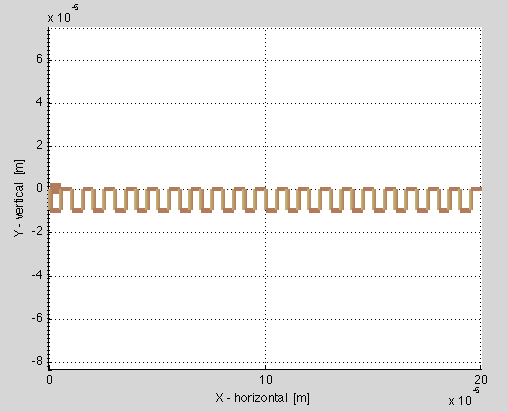


Figure 4: Use of arrays and for loop

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 boundof the loop, and upperbound is the upper bound of a loop.

## Conditionals

The general syntax of an if statement is:

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 the user wants a beam that would automatically compute its electrical resistance if one was not provided. He 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

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. The user can also provide a different value resistivity by the use of ‘param’ and passing it to the process file.
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 definedin the if case and the else case is invisible outside of the if statement where it was defined.

1. 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 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.

# Analyzing devices

## 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.

## 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):

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 sufficiently 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 fromcho\_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 solver 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

Vfinal=5;

param.V = Vfinal;

net = cho\_load('device.net', param);

q = cho\_dc(net);

fail, we could try

Vfinal = 5

q = [];

for V = 0:.5:Vfinal

param.V = V

net = cho\_load('device.net, param);

q = cho\_dc(net, q);

end

If we are still unable to find the equilibrium position, we might get useful information from seeing how closely were we 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')

1. 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);

## Steady state and Modal Analysis

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

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 these equations provides Bode plots as well as modal analysis.

The system behavior near equilibrium is determined by analyzing the linearized system. In modal analysis, we find the resonant behavior of the structure, assuming no damping, by solving the eigenproblem:

The eigenvalues give the resonant frequencies, and the corresponding eigenvectors give the resonant modes. Theroutine cho\_mode returns selected frequencies and mode shapes for a structure, along with the operating pointat which linearization took place. Mode shapes can be viewed graphically using the cho\_modeshape commands. For small problems, the default dense solvers are adequate; for larger problems, users should select the number of modes they want, and those modes will be computed using a less expensive iterative method.

To use the steady-state analysis routine, a user specifies a single input degree of freedom and a single output degree of freedom, usually by naming a nodal variable. SUGAR then draws a Bode plot illustrating the amplitude gain and phase shift between a harmonic excitation at the input and a measured harmonic at the output. Note that, unlike the modal analysis routine, the steady state routine does not discard damping terms.

## Transient Analysis

This solver calculates the transient response of a MEMS device, which may contain nonlinearelements 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 multi step 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

## 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. Sensitivity analysis has not yet been integrated into SUGAR 3.0.

# Available models

## 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** mechanical fixed node

**beam3dlink**rigid 3d link

**circbeam** 3D mechanical arced beam

**f2d** planar force or moment

**f3d** 3D force or moment

**gap2d**two planar electrostatic mechanical beams, resistors

**Vsrc** Voltage source

**Isrc** Current source

**eground** Electronic ground

**R** constant resistor

**L** Inductor

**C** Capacitor

## Model descriptions and interfaces

### beam2d

Describes an in-plane beam connecting two nodes.

**Example**

beam2d p1[A B][l=100u w=5u oz=pi/4]

**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 information)

**ox** initial rotation about beam’s x-axis (required if not 0)

**oy** initial rotation about beam’s y-axis (required if not 0)

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

### beam2de

Similar to beam2d but adds electronic resistance to the beam.

**Example**

beam2d p1[A B][l=100u w=5u oz=pi/2 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 information)

**ox** initial rotation about beam’s x-axis (required if not 0)

**oy** initial rotation about beam’s y-axis (required if not 0)

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

### beam3d

Similar to beam2d but can be rotated out-of-plane, hence it is 3 Dimensional.

**Example**

beam3d p1[A B][l=100u w=5u oy=pi/2 oz=pi/4 ox=pi/4]

**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 information)

**oy** initial rotation about beam’s y-axis (required if not 0)

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

**ox** initial rotation about beam’s x-axis (required if not 0)

### beam3de

Similar to beam3d but adds electronic resistance to the beam.

**Example**

beam3d p1[A B][l=100u w=5u oy=pi/2 oz=-pi/4 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 information)

**oy** initial rotation about beam’s y-axis (required if not 0)

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

**ox** initial rotation about beam’s x-axis (required if not 0)

### 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 information)

**oy** initial rotation about beam’s y-axis (required if not 0)

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

**ox** initial rotation about beam’s x-axis (required if not 0)

### beam3dlink

Describes a mechanically rigid link.

**Example**

beam3d p1[A B][w=10u l=100u]

beam3dlink p1[B C][w=10u l=100u oz=pi/2 L1=5u]

**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 information)

**L1**rigid link length (required)

**oy** initial rotation about beam’s y-axis (required if not 0)

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

**ox** initial rotation about beam’s x-axis (required if not 0)

**oy1** initial rotation about beam’s y-axis (optional)

**oz1** initial rotation about beam’s z-axis (optional)

**ox1** initial rotation about beam’s x-axis (optional)

See Section 6 for a detailed example on the use of beam3dlink.

### circbeam

Describes a 3D mechanical curved beam.

**Example**

uses mumps.net

beam3d p1[A B][l=20u w=5u oz=pi/2]

circbeam p1[B C][w=5u radius=20u alpha=pi/2 oz=pi/2]

**Nodal variables:**

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

**Parameters:**

**radius**radius of the arc (required); positive in counterclockwise direction

**alpha**curvature range (required); has to be less than pi/2, positive in counterclockwise direction

**w** beam width in meters (required)

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

**oy** initial rotation about beam’s y-axis (required if not 0)

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

**ox** initial rotation about beam’s x-axis (required if not 0)

### f2d

Describes an in-plane external force at a node.

**Example**

f2d \* [A] [F=10u oz=pi/4]

f2d \* [A] [M=1u oz=pi/4]

**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 beam’s y-axis (required if not 0)

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

**ox** initial rotation about beam’s x-axis (required if not 0)

### f3d

Describes a 3D external force at a node.

**Example**

f3d \* [A] [F=10u oy=pi/2 oz=pi/4]

f3d \* [A] [M=1u oz=pi/4]

**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 beam’s y-axis (required if not 0)

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

**ox** initial rotation about beam’s x-axis (required if not 0)

### gap2d

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

**Example**

gap2dp1[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)

**w2** beam2 width in meters (required)

**gap** initial gap spacing (required)

**R1** beam1 resistance in ohms (required)

**R2** beam2 resistance in ohms (required)

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

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

### Vsrc

Describes a voltage source.

**Example**

Uses stdlib.net

Vsrc \* [A B] [V=5]

Note: only mechanical element display.Stdlib.net required.

**Nodal variables:**

{e} at both nodes

**Parameters:**

V voltage in volts (required)

### Isrc

Describes a voltage source.

**Example**

uses stdlib.net

Isrc \* [A B] [I=5u]

Note: only mechanical element display. Stdlib.net required.

**Nodal variables:**

{e} at both nodes

**Parameters:**

**I** Current in Amperes (required)

### eground

Describes an electronic ground.

**Example**

uses stdlib.net

eground \* [A] []

Note: only mechanical element display. Stdlib.net required.

**Nodal variables:**

{e} at nodes

**Parameters:**

None

### R

Describes a constant resistor.

**Example**

Uses stdlib.net

R \* [A B] [R=50]

Note: only mechanical element display. Stdlib.net required.

**Nodal variables:**

{e} at both nodes

**Parameters:**

**R** Resistance in ohms (required if G not specified)

**G** Conductance in mhos (required if R not specified)

### L

Describes a Inductor.

**Example**

Uses stdlib.net

L \* [A B] [L=2m]

Note: only mechanical element display. Stdlib.net required.

**Nodal variables:**

{e} at both nodes

**Parameters:**

**L** Inductance in Henry (required)

### C

Describes a capacitor.

**Example**

Uses stdlib.net

C \* [A B] [C=4u]

Note: only mechanical element display. Stdlib.net required.

**Nodal variables:**

{e} at both nodes

**Parameters:**

**C** Capacitance in Farad (required)

# Function reference

## Load netlist

**Calling sequence**

function [net] = cho\_load(name, param);

Loads and processes a netlist.

**Inputs**

**name** - String naming the netlist file to be loaded.

**param** (Optional) - A structure whose entries are the values of the parameters to be overridden.

**Example**

param.nfingers=10; % Set the nfingers parameter

net = cho\_load('comb.net', param); % Load netlist

## Device display

**Calling sequence**

function cho\_display(net, q);

Displays 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

q = cho\_dc(net)

cho\_display(net);

figure(1); %Display undisplaced structure in figure 1

cho\_display(net, q);

figure(2); %Display displaced structure in figure 2

**Caveats**

There is no display of electrical components.

## Viewing displacements

**Callingsequence**

function [dqcoord] = cho\_dq\_coord(q, net, node, coord);

Extract the displacement corresponding to a particular coordinate from a displacement vector q (as output by the function cho\_dc), or an array of displacement vectors (as output by cho\_ta).

**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 q associated with the indicated coordinate.

**Example**

net = cho\_load('beamgap.net'); % Load netlist

q = cho\_dc(net); % Perform static analysis

dy = cho\_dq\_view(q, net, 'B', 'y'); % Get the y displacement at node B

## Static analysis

**Callingsequence**

function [q] = cho\_dc(net, q0, is\_sp)

Finds a solution of the equilibrium equations using a Newton-Raphson method.

**Inputs**

**net** - Netlist structure returned from calling cho\_load

**q0** (Optional) - Starting guess for an equilibriumposition. If noq0 is provided, or if q0 = [], then the search will start at q0 of 0.

**Is\_sp** (Optional) - If true, the code will use sparse solvers. The default is true, that is,

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

q = cho\_dc(net); % Perform static analysis

cho\_display(net, q); % 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 diagnostic warning:

Warning: DC solution finderdidnotconvergeafter40iterations

In this case, the result q returned by the routine may not be correct. Failure to find equilibrium occurs in cases such as as an electrostatically actuated gap operating near pull-in voltage.

A system may have more than one equilibrium point. When SUGAR finds an equilibrium point, it may not always be the desired equilibrium point. For example, in the case of an electrostatically actuated gap, 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.

As SUGAR will usually find the equilibrium position closest to where it starts looking, a good starting guess might be quite useful. Finding the equilibrium point for a “nearby” problem can often attain this. For example, in trying to find the equilibrium point for an 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.

## Modal analysis

### Analysis routine

**Callingsequence**

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, or if q0 = [], the routine will consider q0 = 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.

**Example**

net = cho\_load('beamgap.net');

[freq, egv, q0] = cho\_mode(net, 1, [], 1);

### Display routine

**Callingsequence**

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 that normally move a few microns, a scale factor of 10-4often 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. Also note that the modal analysis routine neglects damping forces.

As noted above, using cho\_modeshapewith 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.

## 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 pointq0.

**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, 'node5', 'y', 'node10', 'y');

**Caveats**

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

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

## Transient analysis

**Calling sequence**

function [T,Q,C,G] = cho\_ta(net,tspan,q0)

This function simulates 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

plot(T, dy); % plot how it moves over t

**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.

# Examples with explanations

This section describes some examples explaining the usage of Sugar3.0. 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.

## Cantilever example

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 the method of making the netlist, running static analysis, and obtaining the graphical output.

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

**Netlist:**

A netlist is created as follows:

* Open the text editor in Matlab
* Write the netlist text
* Save it as filename.net or filename.m. Both are valid, however, remember to use the correct file extension while loading the file in SUGAR.

The 4 lines in our netlist represent an anchor, beam, and force.

% cantilever.net

uses mumps.net

anchor anchorp1[substrate][l=10u w=10u]

beam beam3dp1[substrate tip][l=100u w=2u h=2u, oz=pi/4]

force f3d\*[tip][F=2u oz=pi/2]

The first line in the netlist includes a process file ‘mumps.net’. All process information such as layer thickness, Young’s modulus and other material properties is defined in the process file. The user can also define his own process. The fabrication layer of this process is p1.

The second 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. 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 provides the geometry and orientation of the anchor. Here the length and width are 10 microns.

The third line in the netlist representsa flexible 3D beam. The model used for this beam is called beam3d. The fabrication layer with which this beam is composed of is the p1 layer (i.e. the first layer of polysilicon, defined in process file mumps.net). 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 parameter sectionof this line provides geometry and orientation for the 3D beam. 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 micro-Newtons. The orientation of the force vector is in the y-direction.

**Running a netlist**

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
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 on the structure. The cho\_dc command takes net as its input. Using the parameter values given in the netlist and the parameterized element models, it calculates the deflection of the structure. The output of cho\_dc is the displacement vector dq.

SUGAR can graphically display the deflected structure using the cho\_display command.

Cho\_displayusesgeometries and orientations from net, and node displacements from dq as input to display the structure.

To display original, non-deflected structure, simply type

cho\_display(net);

To display the deflected structure, type

cho\_display(net, dq);

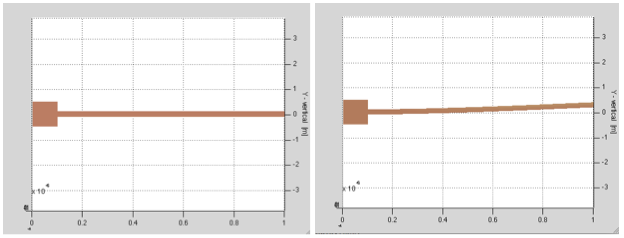


Figure 5: Cantilever example

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.

## Multiple beam example

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

**Netlist:**

% 'multibeam.net'

uses mumps.net

anchor p1[substrate][l=10u w=10u]

beam3d p1[substrate A][l=100u w=2u]

beam3d p1[A B][l=50u w=4u oz=pi/4]

beam3d p1[A C][l=50u w=4u oz=3\*pi/4]

beam3d p1[C D][l=50u w=4u oz=pi/2]

f3d \* [D] [M=1n oz=pi/2]

As before, each element is connected at shared nodes. The commands to load the netlist, performstatic analysis, and display the non-deflected and deflected structures are as follows:

net = cho\_load('multibeam.net'); dq = cho\_dc(net);

figure(1); cho\_display(net);

figure(2); cho\_display(net, dq);

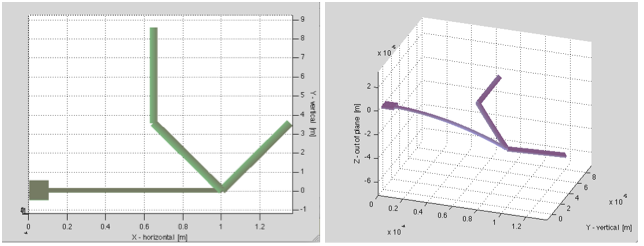


Figure 6: Multibeam example

## Beam3dlink example

This example demonstrates the usage of the model function beam3dlink.

Notice, the difference produced by the two netlists.

uses mumps.net

beamL = 20u

beam3d p1[A B][w=2u l=beamL]

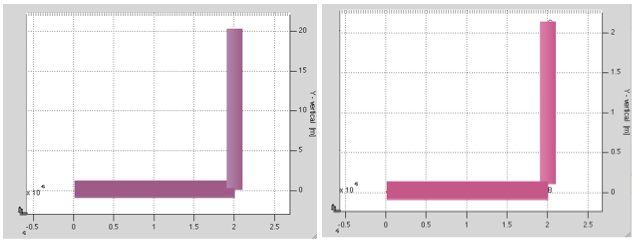
beam3d p1[B C][w=2u l=beamL oz=pi/2]

uses mumps.net

beamL = 20u

beam3d p1[A B][w=2u l=beamL]

beam3dlink p1[B C][w=2u l=beamL oz=pi/2 L1=1u]

****

(a) (b)

Figure 7: 7(a) 2 beams connected at their nodes, 7(b) 2 beams connected through beam3dlink

In SUGAR, the elements are connected to each other at the nodes. However, sometimes there is a need for accurate representation of the dimensions of various elements. Beam3dlink places an invisible rigid link between the end node of the previous element and the next element. Here, the second element is essentially shifted by a distance of 1 micron and rotated by oz=pi/2, before it connects to the first element. Parameters l, w and h define the dimensions of the connecting beam; L1 is the length of the rigidlink; ox, oy and oz are the global rotation angles; and ox1, oy1 and oz1 define the local rotation of the rigid link.

For small forces, figure 7(b) is equivalent to 7(a).

## Array definition (demo array\_use)

This example demonstrates the use of the arrays in SUGAR. Each beam is rotated from the previous beam by an angle of 30deg.

**Netlist:**

%demonstrate use of arrays

uses mumps.net

anchor p1[a][l=10u w=10u]

beam3d p1[a b(1)][l=50u w=5u]

for i=1:5

[

beam3d p1 [b(i) b(i+1)] [l=50u w=5u oz=(i-1)\*pi/6]

]

Here, b is an array of length 5. Arrays are extremely useful for making repetitive structures with the help of for loop.

The final structure is shown below. Note again that the orientation of each beam is defined from the positive x-axis.

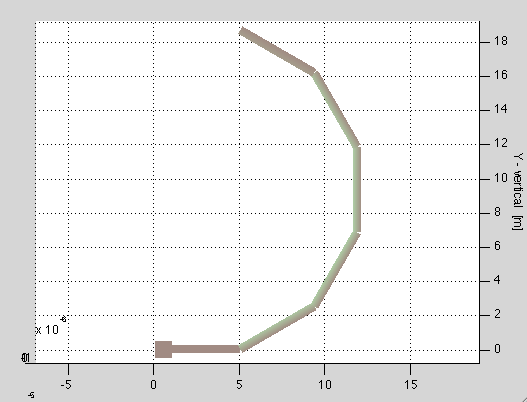


Figure 8: Array example

## Parameter definition (demo param\_use)

**Netlist:**

uses mumps.net

l\_anchor=15u

w\_anchor=10u

h\_anchor=2u

param l\_hot=300u

param w\_hot=0.8u

h\_hot=h\_anchor

param l\_cold=l\_hot\*0.75

param w\_cold=20u

h\_cold=h\_anchor

param gap=4u

l\_bridge=w\_cold/2+w\_hot/2+gap

w\_bridge=3u

h\_bridge=h\_anchor

l\_short=l\_hot\*0.25

w\_short=2u

h\_short=h\_anchor

param T\_top=600

param T\_bottom=300

thermalexpansion=2.3e-6

anchor p1[A][l=l\_anchor w=w\_anchor h=h\_anchor oz=pi/2]

anchor p1[B][l=l\_anchor w=w\_anchor h=h\_anchor oz=3\*pi/2]

beam3d p1[A a][l=l\_hot w=w\_hot h=h\_hot T=T\_top thermalexpansion=thermalexpansion]%hot arm

beam3d p1[a c][l=l\_bridge w=w\_bridge h=h\_bridge oz=3\*pi/2 ]%bridge

beam3d p1[b c][l=l\_cold w=w\_cold h=h\_cold oz=0 T=T\_bottom thermalexpansion=thermalexpansion]%cold arm

beam3d p1[b d][l=w\_cold/2-w\_short/2 w=2u oz=pi/2]%connecter to anchor

beam3d p1[B d][l=l\_short w=w\_short h=h\_short]%short arm

Here the following parameters are parametrized:

l\_hot, w\_hot, l\_cold, w\_cold, gap, T\_top and T\_bottom.

The user can change the default values of any of there parameters from the command window in Matlab.

param.l\_hot = 400e-06;

param.w\_hot = 4e-06;

param.T\_top = 1000;

net = cho\_load('heatuator1.m', param);

q = cho\_dc(net);

cho\_display(net, q);

The structure deflects because of the difference between the temperature of the top and the bottom beams.

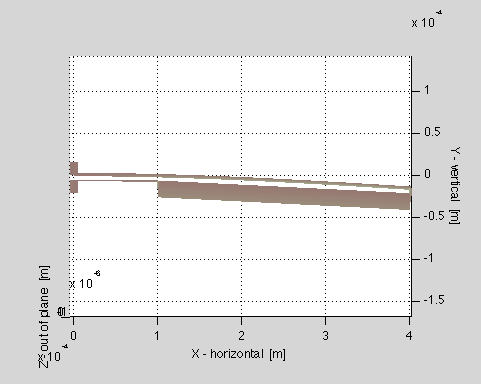


Figure 9: Use of identifier 'param'

## 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 mumps.net

uses stdlib.net

Vsrc \* [A fred] [V=12 sv=0.1 sph=0]

eground \* [fred] []

anchor p1[A][l=5u w=10u oz=pi]

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=-pi/2]

anchor p1[E][l=5u w=10u oz=-pi/2]

eground \* [E] []

Equilibrium displacements have been calculated at an input voltage of 12V. The undeflected and deflected structures are shown below.

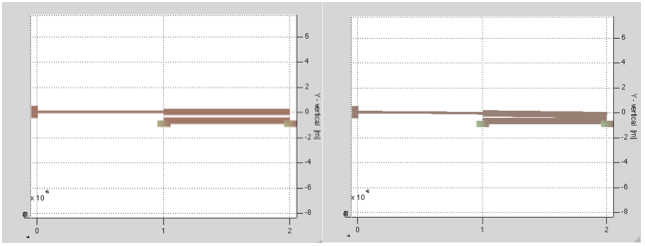


Figure 10: Beamgap structure

## 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. The netlist and structure is shown below.

**Netlist:**

uses mumps.net

%anchors and their torsion hinges:

anchor p1[b][ l=10u w=10u oz=pi/2 h=8u]

beam3d p1[b c][l=80u w=2u oz=-pi/2 h=2u]

beam3d p1[d e][l=80u w=2u oz=-pi/2 h=2u]

anchor p1[e][l=10u w=10u oz=-pi/2 h=8u]

%outer frame:

beam3d p1[c f][l=100u w=20u oz=-pi/2 h=4u]

beam3d p1[f d][l=100u w=20u oz=-pi/2 h=4u]

beam3d p1[c1 f1][l=100u w=20u oz=-pi/2 h=4u]

beam3d p1[f1 d1][l=100u w=20u oz=-pi/2 h=4u]

beam3d p1[c c1][l=200u w=20u h=4u]

beam3d p1[d d1][l=200u w=20u h=4u]

%inner torsion hinges:

beam3d p1[g3 f1][l=40u w=2u h=2u]

beam3d p1[f g6][l=40u w=2u h=2u]

%inner solid "plate":

beam3d p1[g6 g3][l=120u w=140u h=4u]

%rear lever:

beam3d p1[h f][l=75u w=80u h=4u]

The un-displaced structure is shown below:

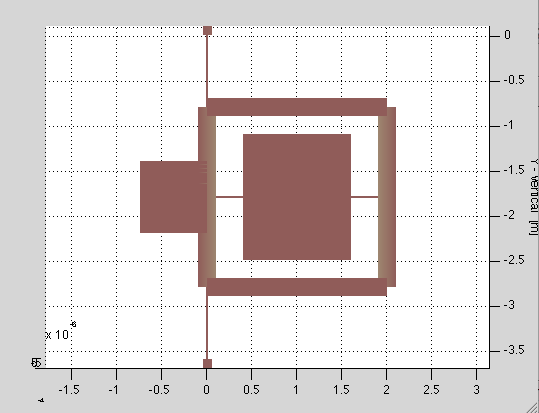


Figure 11: Model analysis: undisplaced mirror structure

The various frequency modes are given below:

For doing Model Analysis, type the following commands in the Matlab command window:

net = cho\_load('mirror.m'); % Load the netlist

cho\_display(net); % Display the original structure

[f, egv, dq] = cho\_mode(net); % Perform mode calculation

cho\_modeshape(net, f, egv, dq, mode\_num);

for mode\_num = 1:4

figure(mode\_num)

cho\_modeshape(net, f, egv, dq, mode\_num);

end

The four structures for each mode are shown below:

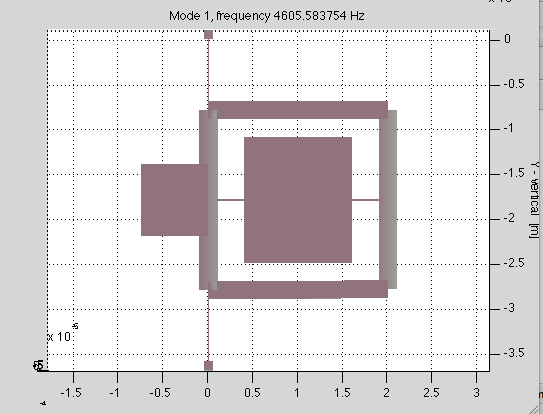


Figure 12: Model analysis: 1st mode for mirror structure

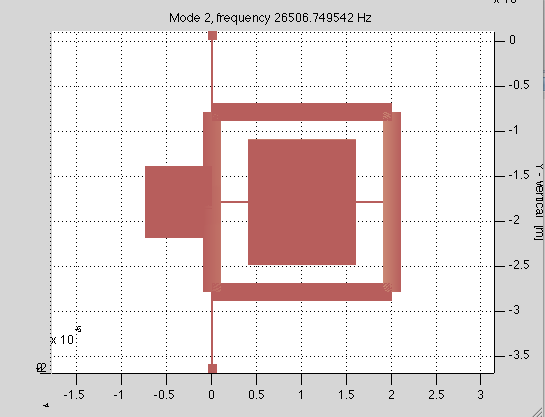


Figure 13: Model analysis: 2nd mode for mirror structure

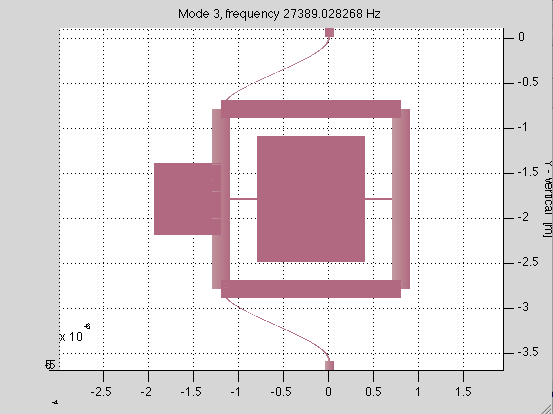


Figure 14: Model analysis: 3rd mode for mirrir structure

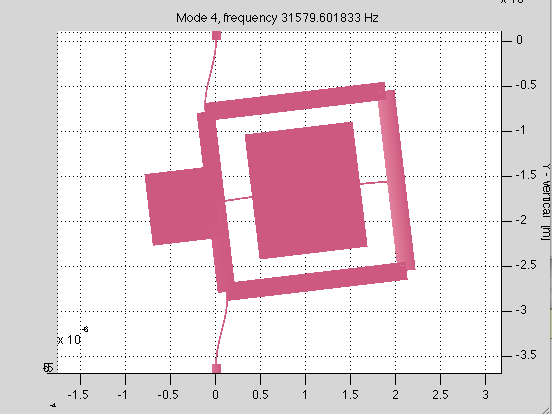


Figure 15: Model analysis: 4th mode for mirror structure

## Steady state analysis (demo\_ss)

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

**Netlist:**

uses mumps.net

a1 anchorp1[n2][l=5u oz=0 w=10u R=100]

b1 beam2dp1[n2 n5][l=150u oz=pi w=2u R=1000]

a2 anchorp1[n3][l=5u oz=0 w=10u R=100]

b2 beam2dp1[n3 n6][l=150u oz=pi w=2u R=1000]

b1v beam2dp1[n5 n7][l=50u oz=pi/2 w=5u R=500]

b2v beam2dp1[n5 n6][l=50u oz=-pi/2 w=5u R=500]

b3v beam2dp1[n6 n8][l=50u oz=-pi/2 w=5u R=500]

b3 beam2dp1[n7 n9][l=150u oz=0 w=2u R=1000]

b4 beam2dp1[n8 n13][l=150u oz=0 w=2u R=1000]

b1m beam2dp1[n9 n10][l=50u oz=0 w=20u R=100]

b2m beam2dp1[n10 n16][l=50u oz=0 w=20u R=100]

b3m beam2dp1[n10 n11][l=75u oz=-pi/2 w=20u R=100]

b4m beam2dp1[n11 n12][l=75u oz=-pi/2 w=20u R=100]

b5m beam2dp1[n13 n12][l=50u oz=0 w=20u R=100]

b6m beam2dp1[n12 n17][l=50u oz=0 w=20u R=100]

bh beam2dp1[n11 n14][l=300u oz=0 w=2u R=1500]

bm beam2dp1[n14 n15][l=196u oz=0 w=116u R=100]

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

net = cho\_load('multimode\_m.m');

figure(1);cho\_display(net);

dq = cho\_dc(net); % Find a DC equilibrium point;

% Steady state analysis and bode plot;

% Specify the input excitation at 'node10' along 'y' direction;

% Specify the output response of 'y' displacement of the mass at 'node14';

figure(2);

find\_ss(net,dq,'n10','y','n14','y');

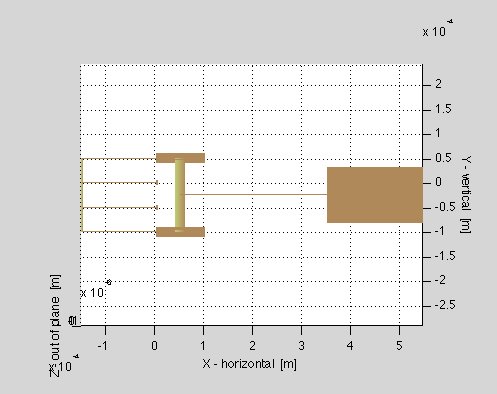


Figure 16: Steady state analysis: Undisplaced resonator structure

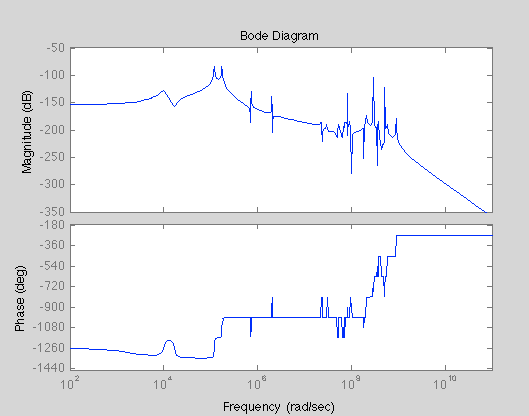


Figure 17: Steady state analysis: Bode plot

# More examples

## V-beam thermo-mechanical actuator (ETA\_netlist\_demo)

This example demonstrates the use of param, arrays and subnets. The netlist simulates a v-beam thermo-mechanical actuator. The user can change the dimensions and material properties of the actuator. The user needs to specify the average temperature of the v-beams. The v-beams expand due to the temperature rise and due to the symmetry of the structure, the shuttle translates in the negative x-direction.

The netlist and undisplaced structure is shown below:

%V-beam Thermo-mechanical Actuator

%make demo file for it in demo folder

uses mumps.net

param l\_vbeam=330u

param w\_vbeam=8u

param theta\_vertical=pi/6

param n\_vbeam=5

param l\_hsink=40u

param w\_hsink=4u

param n\_hsink=3

param w\_shuttle=60u

param ambienttemperature=298

param T\_vbeam=298

param thermalexpansion=2.5e-06

%subnet for heat sinks

subnet sub1\_ETA[A B][]%this means that the subnet starts at node A and ends at node B

[

beam3d parent [A B] [l=(w\_hsink/2+16u+w\_hsink/2) w=w\_shuttle h=3.5u] %the distance between each heat sink is 16u

beam3dlink parent [B C] [l=l\_hsink w=w\_hsink oz=pi/2 h=3.5u L1=w\_shuttle/2]

beam3dlink parent [B D] [l=l\_hsink w=w\_hsink oz=-pi/2 h=3.5u L1=w\_shuttle/2]

]

%subnet for V-beams

subnet sub2\_ETA[A B][]

[

beam3d parent [A B] [l=32u w=w\_shuttle h=3.5u]

beam3d parent [B C] [l=l\_vbeam w=w\_vbeam h=3.5u oz=(pi/2 theta\_vertical) T=T\_vbeam thermalexpansion=thermalexpansion]

beam3d parent [B D] [l=l\_vbeam w=w\_vbeam h=3.5u oz=-(pi/2-theta\_vertical) T=T\_vbeam thermalexpansion=thermalexpansion]

%anchors are needed on each side of the two beam

anchor parent [C] [l=10u w=10u h=3.5u]

anchor parent [D] [l=10u w=10u h=3.5u]

]

%make the front heat sinks

for j = 1:n\_hsink %n\_hsink=3

[

sub1\_ETA p1 [yy(j) yy(j+1)] [l=40u w=4u h=3.5u] %pass default values

]

%connecting front heat sinks to V-beams

beam3dlink p1[yy(n\_hsink+1) xx(1)][l=1n w=w\_shuttle h=3.5u L1=0]

%make the V-beams

for j = 1:n\_vbeam %n\_vbeam = 5

[

sub2\_ETA p1 [xx(j) xx(j+1)] [l=300u w=8u h=3.5u] %pass default values

]

%connecting the V-beams to rear heat sinks

beam3d p1[xx(n\_vbeam+1) zz(1)][l=40u w=w\_shuttle h=3.5u]

%make the rear heat sinks

for j = 1:n\_hsink

[

sub1\_ETA p1 [zz(j) zz(j+1)] [l=40u w=4u h=3.5u]

]

beam3d p1[zz(n\_hsink+1) ss][l=16u+w\_hsink w=w\_shuttle h=3.5u]

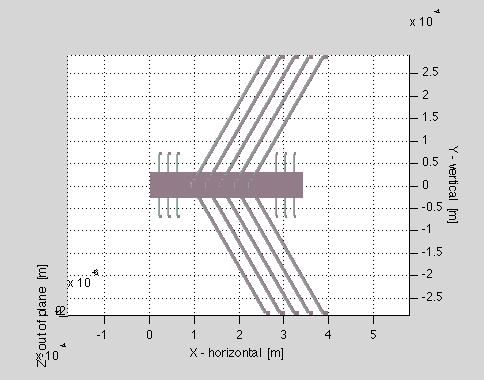


Figure 18: V-beam thermo-mechanical actuator

The user can then change the values of the parameters. The model prompts the user to change each parameter value. The command for this will be:

param.’parametername’=’parameter value’

For example, the user can change the number of v-beam pairs and length of v-beams of the actuator. He can also change the temperature of the v-beams to say 600K. The user can also find the displacement of any node. Here, the we find the x-displacement of first node yy(1).

param.l\_vbeam = 400e-06;

param.n\_vbeam = 3;

param.T\_vbeam = 1000;

net = cho\_load('ETA\_netlist\_demo.m', param);

q = cho\_dc(net);

cho\_display(net, q);

cho\_dq\_view(q, net, 'yy(1)', 'x')

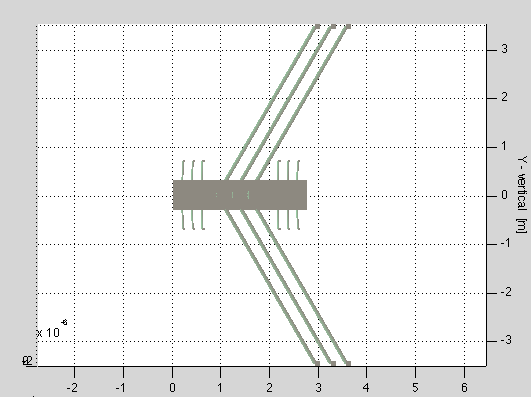


Figure 19: V-beam thermo-mechanical actuator: user modified parameters

## Carbon nanotube (CNT) structure

### Armchair structure

This example discusses the use of a new model in sugar that simulates armchair carbon nanotubes. Armchair structure is characterized by equal chirality (n=m). The use can simulate any armchair CNT of desired chirality and lengthin SUGAR by passing them as parameters to the netlist.

The netlist is given below:

%index number n1, m1

uses mumps.net

param n1=5

param m1=5

param n\_rings=6

anchor p1[c(1,1)][l=0.001n w=0.001n h=0.001n]

%chirality (n,m)

%here we simulate a armchair SWNT, hence m=0

%user inputs the chirality (n,m)

d = (0.245/pi)\*sqrt(n1^2 + m1^2 + n1\*m1) %diameter of a SWNT

%by knowing the diameter, we can find the number of hexagons in the

%lattice, i.e. N

subnet link\_SWNT[a(1,1) b(1,1) c(1,1) c(1,2) a(2,1) b(2,1)][l=\* w=\* h=\*] %replace beam3d with beam3d\_SWNT to have have properties of C-C bond.

[

beam3d parent [a(1,1) c(1,1)] [l=0.1415n w=0.001n h=0.001n oy=-pi/(2\*n1) oz=(pi+pi/3)]

beam3d parent [c(1,1) a(2,1)] [l=0.1415n w=0.001n h=0.001n oy=-pi/(2\*n1) oz=-pi/3]

beam3d parent [a(2,1) b(2,1)] [l=0.1415n w=0.001n h=0.001n]

beam3d parent [b(2,1) d(1,1)] [l=0.1415n w=0.001n h=0.001n oy=pi/(2\*n1) oz=pi/3]

beam3d parent [d(1,1) b(1,1)] [l=0.1415n w=0.001n h=0.001n oy=pi/(2\*n1) oz=(pi/2 + pi/6)]

beam3d parent [d(1,1) c(1,2)] [l=0.1415n w=0.001n h=0.001n oy=pi/n1]

]

for j = 1:n\_rings

[

for k = 1:n1

[

link\_SWNT p1 [a(j,k) b(j,k) c(j,k) c(j,k+1) a(j+1,k) b(j+1,k)] [l=0.1415n w=0.001n h=0.001n oy=(k-1)\*(2\*pi/n1)]

beam3dlink p1 [c(j,n1+1) x(j,1)] [l=0n w=0.001n h=0.001n L1=0.001n]

]

]

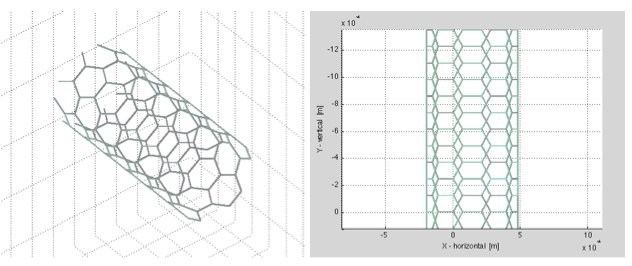


Figure 20: Two different views of (5, 5) armchair CNT

### Zigzag structure

This example discusses the use of a new model in sugar that simulates zigzag carbon nanotubes. Zigzag structure is characterized by m=0, where (n,m) is the chirality of the CNT. The use can simulate any zigzag CNT of desired chirality and length in SUGAR by passing them as parameters to the netlist.

**Netlist:**

uses mumps.net

param n1=10

param m1=0

param n\_rings=6

anchor p1[x(1,1)][l=0.001n w=0.001n h=0.001n]

%chirality (n,m)

%here we simulate a zigzag SWNT, hence m=0

%user inputs the chirality (n,m)

d = (0.245/pi)\*sqrt(n1^2 + m1^2 + n1\*m1) %diameter of a SWNT

%by knowing the diameter, we can find the number of hexagons in the

%lattice, i.e. N

N=floor((pi\*d)/0.245)

subnet link\_SWNT[c(1,1) c(1,2) c(2,1)][l=\* w=\* h=\*]%nodes c(1,1), c(1,2) and c(1,3) are visible outside the subnet

[

beam3d parent [c(1,1) b(1,1)] [l=0.1415n w=0.001n h=0.001n oz=-pi/6]

beam3d parent [b(1,1) c(1,2)] [l=0.1415n w=0.001n h=0.001n oy=pi/N oz=pi/6]

beam3d parent [b(1,1) c(2,1)] [l=0.1415n w=0.001n h=0.001n oz=-pi/2]

]

for j = 1:n\_rings

[

for k = 1:N

[

link\_SWNT p1 [x(j,k) x(j,k+1) x(j+1,k)] [l=0.1415n w=0.001n h=0.001n oy=(j-1)\*(pi/N) + (k-1)\*(2\*pi/N)]

beam3dlink p1 [x(j,N+1) x(j,1)] [l=0n w=0.001n h=0.001n L1=0.001n]

]

]

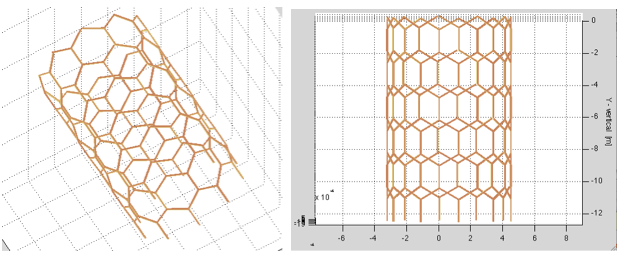


Figure 21: Two different views of (5, 0) zigzag CNT

# Code architecture and source organization

There are three major components to SUGAR:

1. Model functions which describe how to check parameters, write equations, and display individual elements of a device.
2. Routines to assemble and analyze device equations and display results; and
3. 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.

# Model function interface

## Model overview

SUGAR 3.0 provides a general interface for incorporating new device models. SUGAR formulates the governing equations for a device as

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 the following:

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 that 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.

## 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**- A 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].

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.

## Variable definitions

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

Ground variables are variables that 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 that 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’, ‘var2’. are the names of branch variables. If a model contributes no variables of a particular type, then the corresponding field should notappear 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 be the y-displacement of second node, 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.

## 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

## 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.

## 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 node 2

%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 , in computing the linearized system at an equilibrium point for static and modal analysis, and in stiff solvers for the transient equations.

## 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

## 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 that 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);

# Matlab structures, assembly, and analysis

## 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 thesame 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 hierarchy 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\_anchorp1[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.

## 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 corresponding 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, andtherefore will not have a corresponding global index. The matrix rows and columns corresponding 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 omitted, 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.

# Parsing and pre-processing

In order to easily support a more powerful netlist language, the parser in SUGAR 2.0 was rewritten 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.

## Translator structure

### 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 language 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.

### 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 rudimentary. Besides the automatic detection of parse errors, the parsing routines check only for undefined variables and invalid process layers.

### 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 that keeps a separate stack for subnet state, similar to the runtime stack kept by most modern languages.

## Matlab post-translation processing

### 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.

### 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.

### 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 that 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.

# SugarCube

## What is SugarCube?

SugarCube is a tool that adds a simple and intuitive graphical user interface (GUI), parameterization, and optimization features to SUGAR. This tool is available online at nanoHUB.org through a web interface and remote computation. SugarCube has a hierarchical library, which consists of multitude of complete networks of popular or useful MEMS that are based on the published work of experts. Once a user selects and loads a system from the library, default values of its key geometric, material, and loading parameters are displayed. These parameters can be held constant or modified. With these parameters, the user can investigate the system by performing static, modal, steady state, or transient analysis. The deflected system is displayed, and the results of the analysis are available through plots or numerical data. Top-down analysis can also be performed in SugarCube, i.e. the user can determine the optimum values of parameters (geometric, material, or dynamic) as a function of desired performance. The desired performance characteristic might be resonance frequency, deflection per voltage or temperature, chip real estate size, etc. What is new and different about SugarCube is that, it does not require reading of manuals, programming, or any MEMS expertise to use. This makes SugarCube amenable to novice users at various educational levels to explore what-if scenarios of MEMS, and to experts who do not have time or desire to learn a traditional tool for modeling and simulating a device based on a common system. SugarCube may also be useful for manufacturers in surveying the performances of various MEMS for use in their new product [3].

This chapter is organized as follows. Graphical user interface(GUI) of SugarCube is described in section 12.2. Section 12.3 describes the parameterization of the MEMS using SugarCube. This section provides the examples of how static, modal, and transient analyses can be performed. Section 12.4 describes the optimization feature of SugarCube with an example.

## SugarCube Graphical User Interface

### MEMS display window

The models loaded into the tool are displayed in this window. Each model in SugarCube is a network of predefined lumped elements which are connected together at nodes. Some of these key nodes are displayed at their corresponding locations in the model (see Fig. 22). After simulation, the deflected shape of the MEMS is displayed in this window.

### Parameterization and optimization window

Through this window the user is also able to load a model, simulate and save results, download simulation data, and optimize the design parameters of the model as a function of desired performance.

Upon clicking the Load button, a window appears showing hierarchical library of ready-made MEMS models (See Figure22). By selecting a category, a list of corresponding MEMS is displayed. An image preview and description accompany each selection in the library for easy identification. Upon selecting a MEMS from the library, the device is displayed in main MEMS display window and predefined parameters associated with this model appear in the parameter window. These parameters may include geometries, material properties, and dynamic performances; all with common default values i.e. the user can run each device in the library at once without providing any input. If the user chooses to modify default values, each parameter has editable minimum, divisions, and maximum entry fields for parameter sweeps. User can investigate the performance of the loaded MEMS with variation in single or multiple parameters. When a single parameter is selected to perform parameter sweep, SugarCube displays a 2D plot of performance vs parameter. In case of two parameters, a 3D surface plot is displayed. Sliders are provided beneath each parameter to make the modification much simpler to the user. When the user selects a single-valued parameter, i.e. selects for a performance at only a single value of parameter, SugarCube simply displays the numerical value of performance. The solver options are given at the bottom of the parameterization window. Threeoptions are available in SugarCube: static, modal,and transient analyses. The lumped-element node and coordinate at which the performance is desired is also selectable for plotting. The detailed procedure of how each parameter sweep can be performed and how each solver can be used is provided in section 12.3.

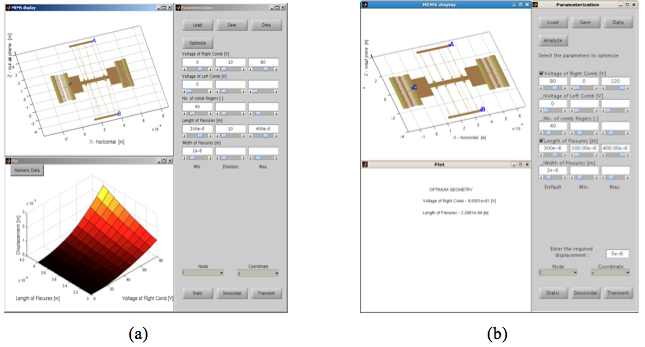


Figure 22: SugarCube GUI. (a) Parameterization of MEMS. In the above example, the variation of deflection of the comb-drive of a tang resonator is investigated with the variation of voltage on right comb and the length of flexures. (b) Optimization of MEMS. In this case, the optimum parameters of the same resonator that gives the desired performance are obtained.

In order to optimize the selected model as a function of desired performance, optimize button is selected. Upon clicking this button, all the parameters associated with the model are displayed and a checkbox is provided for each of them. User can select the parameters to be optimized using these checkboxes. Then the practical bounds in which each parameter can vary, needs to be specified by using the minimum and maximum fields or the sliders beneath them. This step is important to ensure that the optimized solution is realistic. Once the parameters and their bounds are selected, the desired performance is entered in the field provided at the bottom of the window. Recall that the desired performance is for the selected lumped-element node and coordinate of the model. SugarCube formulates an optimization problem using the above input and displays the optimum values of each parameter in the plot window. Examples of how this feature can be used are provided in section 12.4.

### Plot window

Output values of static parameters, 2D curves, or 3D manifolds of parameter sweeps appear in this window. For modal analysis, a drop down menu appears which lists different mode numbers, where each mode, frequency, and deformed shape is displayed in this window. A performance vs time plot is displayed for transient analysis. Higher dimensional numeric data may be downloaded for further analysis in all the above cases.

## Parameterization of MEMS

### Static analysis

In static analysis, SugarCube solves for the static response of the MEMS for the applied excitation force. This feature is explained in the following 3 examples:

1. ***Static analysis of a fixed-fixed beam***

A fixed-fixed beam is loaded from the flexurescategory of hierarchical MEMS library. With the default parameters loaded, the vertical deflection of the center node of the beam can be investigated as follows: As shown in figure 23, the center node is represented by node ‘B’ in the MEMS display window. This node is selected from the Node pull-down menu. For vertical deflection, the y-coordinate is selected from Coordinate pull-down menu. Clicking on the Static button performs the analysis of the beam using default parameter values and displays the deflection in plot window. In this case, the vertical deflection of ‘b’ is 39.457e-6m.

1. ***Static analysis of thermal actuator***

In this example, we investigate the variation of the performance of the thermal actuator with a single parameter sweep. The actuator in this example is available in Thermal Actuators category of the MEMS library. The performance of this device is controlled by the various parameters like length and width of hot arm, gap between hot and cold arm, coefficient of thermal expansion, and temperature of hot and cold arm. For the sake of this demonstration, we investigate the variation of performance of the actuator with width of hot arm. The minimum and maximum values of width are entered in min. and max. fields as 0.8e-6m and 6.0e-6m respectively. 10 divisions are selected as resolution. This creates 10 equally spaced values of width from 0.8e-6m to 6.0e-6m. Then the node ‘A’ and coordinate ‘y’ are selected to investigate the deflection of the actuator tip. Clicking on Staticbutton computes the tip deflection at all the values of width and the output is displayed as a 2-D plot as shown in figure 24(a).

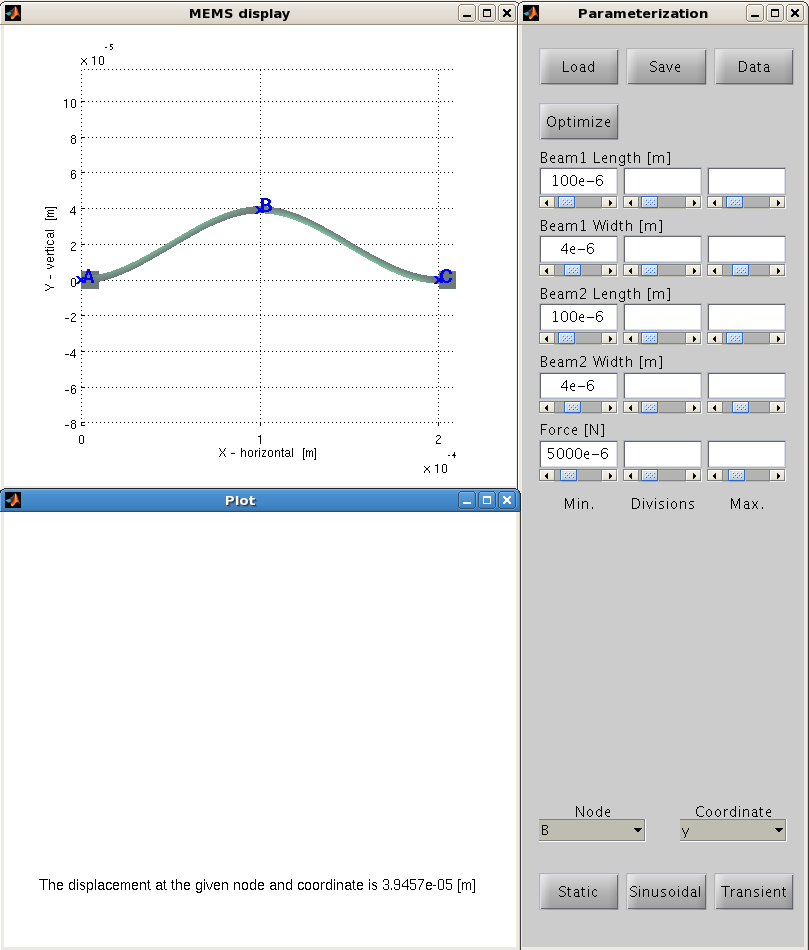


Figure 23: Static analysis of a fixed-fixed beam. This figure demonstrates an example of single parameter analysis. The vertical deflection of the center node of the beam is investigated. In this case, it is observed that the beam deflects by 39.457e-6m for the give parameter values.

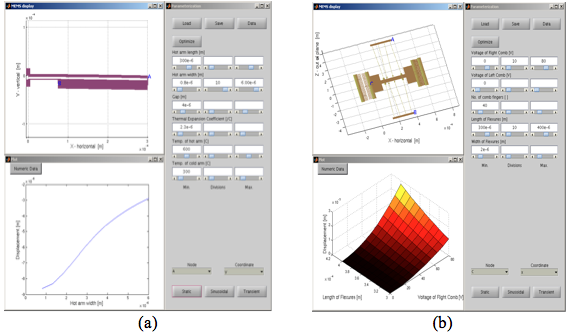
****

Figure 24: Static analysis (a) Thermal actuator. Here a 2D plot of tip displacement as a function of width is displayed; b) Tang resonator. A 3D surface plot of the combdrive displacement as a function of voltage applied and flexure length is displayed.

1. ***Static analysis of a tang-resonator***

A tang-resonator is demonstrated as the final example in static analysis. The values of the selected parameters are entered in a similar procedure described above. The only difference is that, here two parameters i.e. the voltage of right comb and length of flexures are selected. The ‘x’ displacement of the node on the base of the combfinger (node ‘C’) is selected for investigation. Clicking on Static button computes the deflection at all the values of the parameters. A 3-D response surface is displayed as shown in figure 24(b).

### Modal analysis

Performing modal analysis in SugarCube is quite simple as explained below. The device to investigate is loaded from MEMS library. Once loaded, the Sinusoidal button is clicked. This prompts for the user to enter the number of modes required. Once the number of modes is entered, SugarCube performs the modal analysis and a drop down menu appears in plot window. This menu lists different mode numbers, where each mode frequency and deformed shape is displayed. An example of modal analysis of accelerometer is shown in figure 25.

### Transient analysis

In order to perform transient analysis in SugarCube, the user loads the model, enters the parameter values, selects the node and coordinate to investigate the performance, and clicks on Transient button. This button prompts an input from the user to enter the start and end time of simulation. Once these values are entered, SugarCube performs the transient analysis and displays a plot of performance of the selected degree of freedom vs time. As an example, the transient analysis of a crab-leg flexure in water is shown in figure 26(a). An important point here is that, transient analysis of SugarCube accepts only single valued parameters unlike the static analysis that can perform parametric sweep. Currently Matlab’s ode45 solver is used to find the transient response. Reduced order models using Krylov subspace methods are being developed for transient analysis, and they will be included in the next version of SugarCube. As an example, the transient response of the above crab-leg flexure in air calculated using Krylov subspace methods is shown in figure 26(b).

* + 1. Steady **state response**

SugarCube can compute the steady state response of the device and display the bode-plot. This is done using transfer function method. Example of steady state response of a tang-resonator is shown in figure 27.

## Optimization of MEMS

Another interesting feature of SugarCube is optimization of MEMS. In this, the user can select a MEMS from the library and obtain the optimum design parameters of the device that can give the desired performance. This is demonstrated in a simple example shown below. We choose the same tang resonator that was previously demonstrated in Static analysis. Once the device is loaded from the library, the user clicks on Optimize button. This shows all the design parameters with each of them accompanied by a check-box, see Figure 28. The user can select the parameters to optimize using these check-boxes. Once the parameters are selected, the default value, minimum allowed value, and maximum allowed value of each selected parameter are entered. This is to ensure that the optimized design parameters lie in the physically possible limits. Then the user specifies the desired performance by selecting the lumped node, coordinate,and desired deflection at that degree of freedom. Clicking on Static button then formulates an optimization problem with the entered parameters, solves it using Matlab’s optimization toolbox and displays the optimized values of each parameters in plot window (see figure 26).

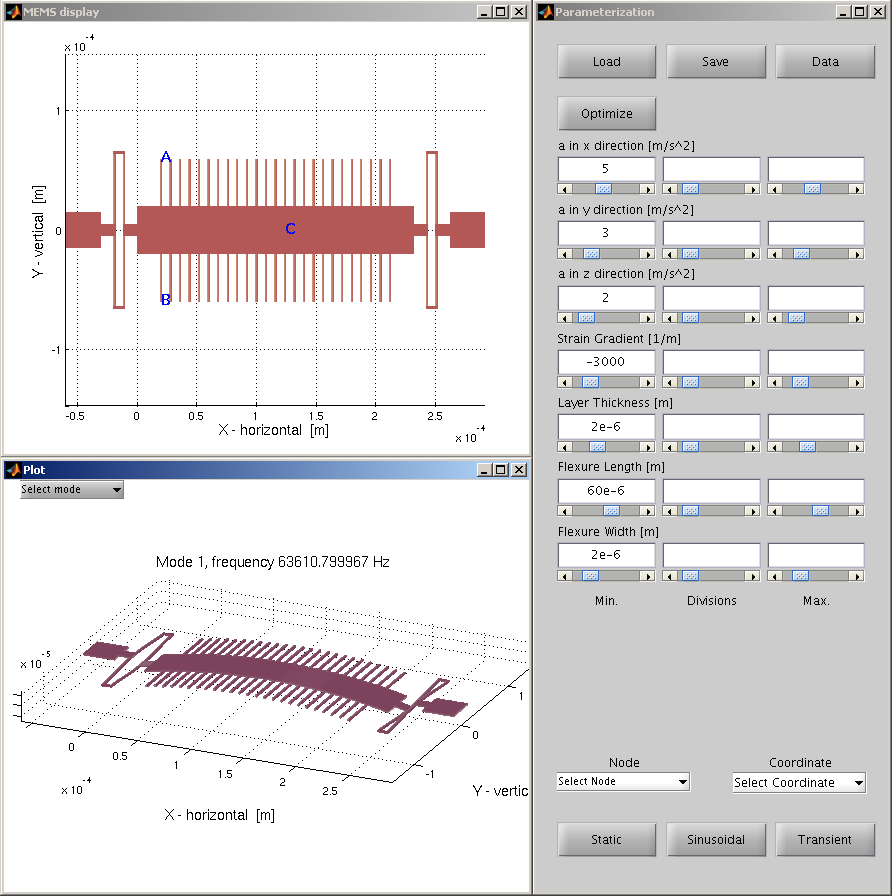
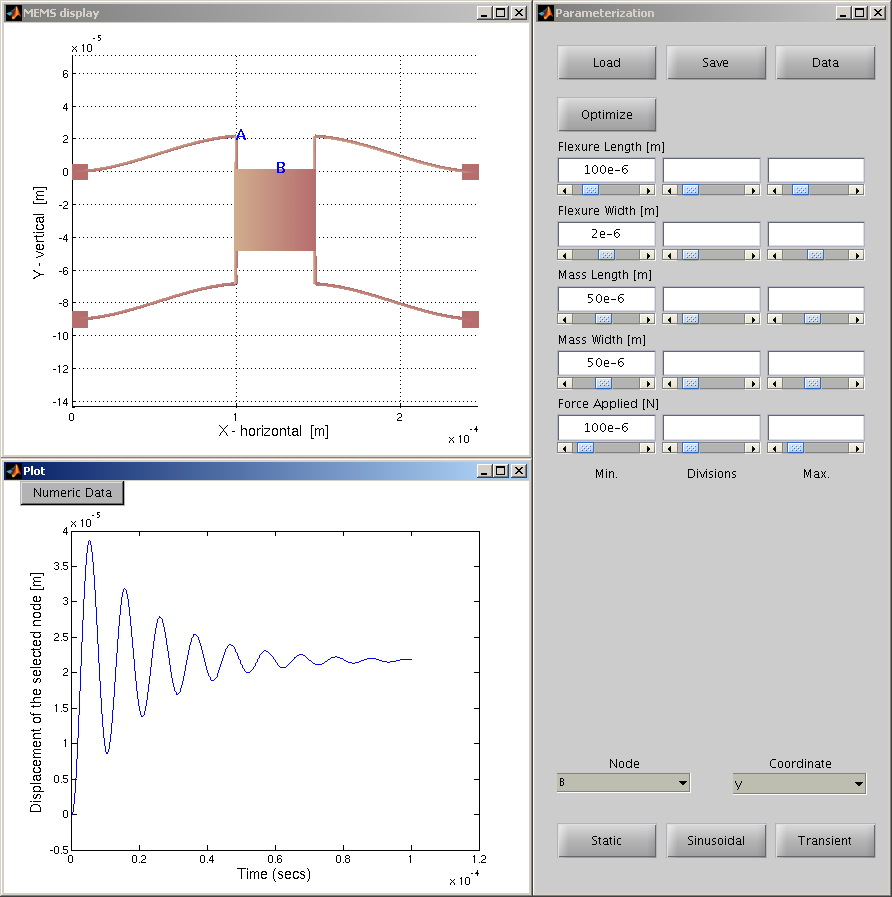
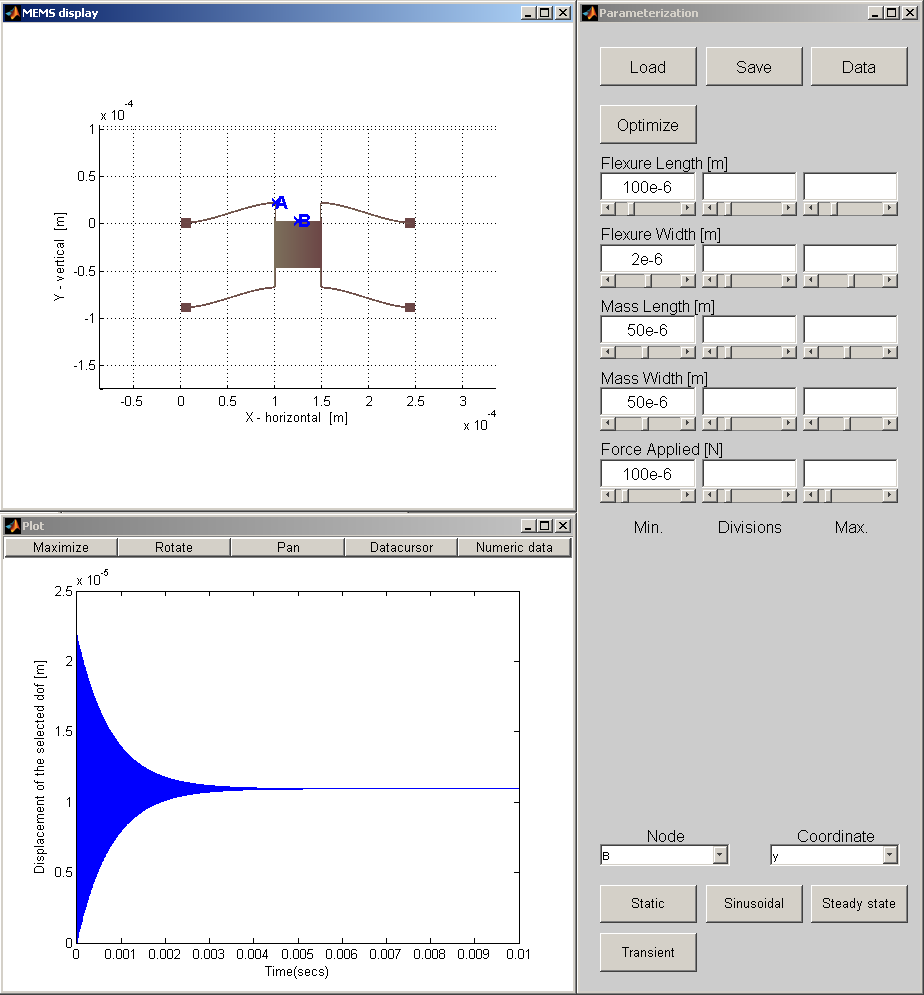
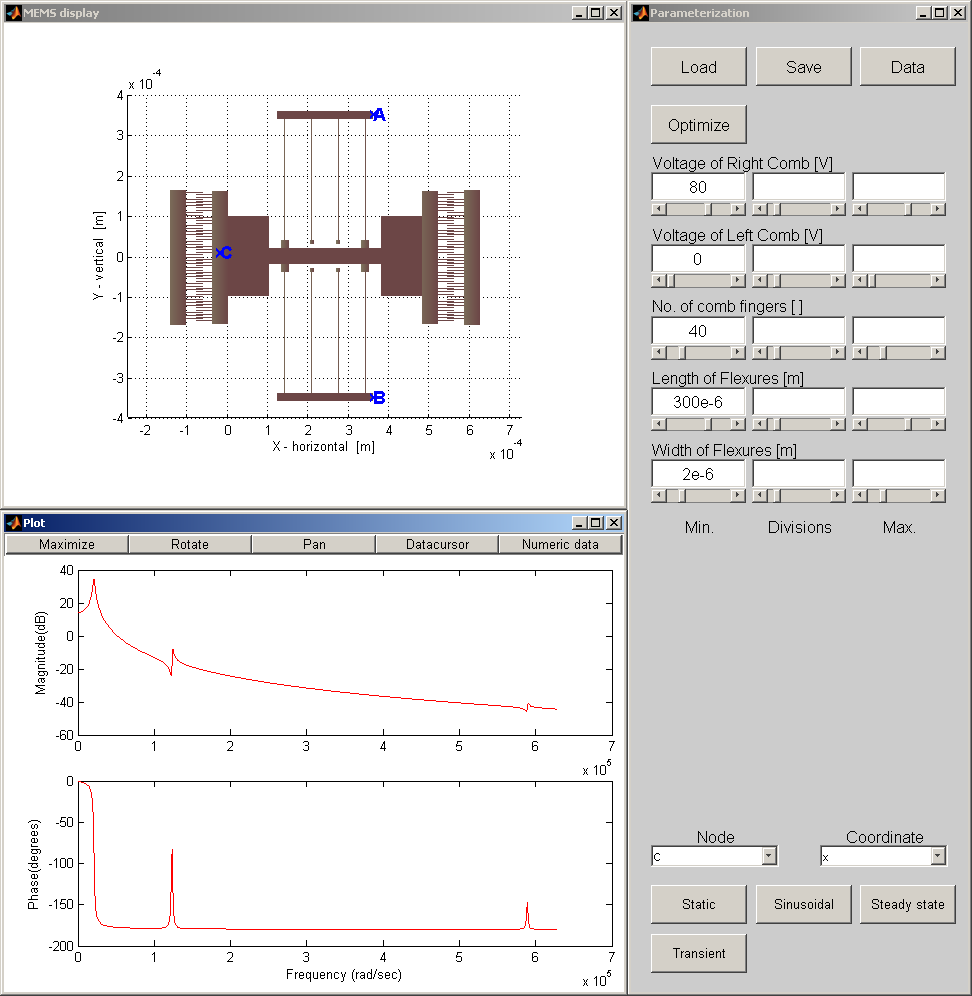


Figure 25: SugarCube features. Modal analysis of accelerometer. The frequency of the first mode and the corresponding mode shape are displayed

** **

**Figure 26: Transient response of a crab-leg flexure (a) Response computed using Matlab’s inbuilt ode45 solver (highly damped medium like water) b) Response computed using Krylov subspace reduced order modeling (medium is air in this case). Note: All the parameters are not the same in (a) and (b) and hence the difference in equilibrium position**

****

**Figure 27. Steady state response of a tang resonator. The plot window displays the bode-plot of the device.**

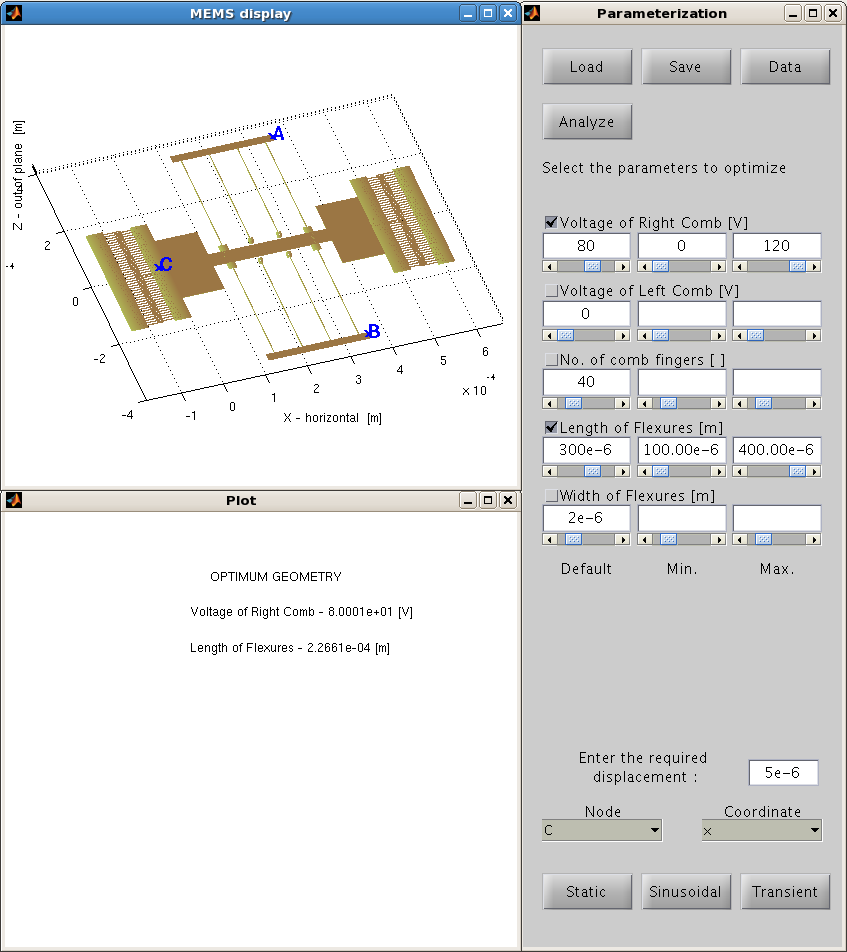


Figure 28: Optimization of MEMS in SugarCube. In order to optimize a MEMS, the device is loaded from the library and the Optimize button is clicked. Upon clicking this button, all the design parameters of the MEMS are displayed with a check-box accompanying each of them. The parameters to optimize are selected using these check-boxes. Then the desired performance at the desired lumped node and coordinate is entered. Finally, clicking on Static button displays the optimum design parameters in plot window.

# Sugar to SugarCube

# In this section, we present how expert users can import their models into SugarCube. This process requires just an addition of a new line to the existing sugar netlist of the model. The expert user identifies the important parameters in the model like the width of the beam, temperature, force applied, etc. These are the parameters which will be shown in the parameterization window of SugarCube. Once the parameters are identified, their default values and their extreme possible values (maximum and minimum) are defined. Finally, as Sugar uses node-based netlist description language, the important nodes at which the performance has to be evaluated should be selected. The selection of important nodes is required to avoid complexity that would otherwise be created by displaying all the available nodes.

# All these parameters are included in the netlist with a ‘sugarcube’ tag using the following syntax:

# sugarcube \* [ ] [ parameter1name='parameter1label,units,default value, minimum value, maximum value ' parameter2name = 'parameter2label,units,default value, minimum value, maximum value' ……………..

# ……………..

# nodes='node1, node2, node3,…. ']

# where the parameter name in the above syntax is the name used for a variable in the netlist and parameter label is the label that needs be displayed in the parameterization window once the model is loaded into SugarCube.

# As a demonstration, we show the following lines of netlist to import tang resonator (Figure 24(b) and 26) to SugarCube.:

# sugarcube \* [] [ VR = ' Voltage of Right Comb, V, 80, 0, 120 '

# VL = ' Voltage of Left Comb, V, 0, 0, 120 '

# nf = ' No. of comb fingers, , 40, 20, 120 '

# L = ' Length of Flexures, m, 300e-6, 100e-6, 400e-6 '

# w = ' Width of Flexures, m, 2e-6, 1e-6, 6e-6 '

# nodes = ' ttR, tbR, L(20) ']

# The last line of the above example provides the node names that are to be displayed in SugarCube. 'ttR,tbR,L(20)' are the node names used inside Sugar netlist description.

# References

1. David Hanson. C,*Interfaces and Implementations*, Addison-Wesley, 1996.
2. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, *Reference Manual of the Programming Language Lua, 4.0*, TeCGraf, PUC-Rio, October 2001. Available at www.lua.org.
3. Prabhakar Marepalli, “Tools for MEMS design and simulation”, *M.S. thesis in writing.*
4. SUGAR 3.0: A MEMS Simulation Program (User’s Guide), April 2002