VibroSim COMSOL

COLLABORATORS			
	TITLE : VibroSim COMSOL		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 22, 2020	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Introduction

VibroSim_COMSOL is a toolkit for generating the vibrational and heat flow models of the VibroSim vibrothermographic NDE model. It is implemented as a set of MATLAB scripts which configure COMSOL to run the vibration and heatflow simulations.

PLEASE NOTE THAT VIBROSIM HAS NOT BEEN ADEQUATELY VALIDATED, THE NUMBERS BUILT INTO IT ALMOST CERTAINLY DO NOT APPLY TO YOUR VIBROTHERMOGRAPHY PROCESS. ITS OUTPUT CANNOT BE TRUSTED AND IS NOT SUITABLE FOR ENGINEERING REQUIREMENTS WITHOUT APPLICATION- AND PROCESS-SPECIFIC VALIDATION.

Vibrothermography is an NDE technique for finding cracks by vibration-induced crack heating. When a specimen containing a crack is vibrated at high amplitude, heat is generated at the crack. This is caused by a friction-like dissipative mechanism based on the relative motion of the two crack faces.

VibroSim_COMSOL separates out your geometric and material model of your specimen from VibroSim COMSOL's physics models of the vibrothermography process. The normal use of VibroSim_COMSOL is to predict vibrational response to a pulse, tone burst, or frequency sweep and to predict temperatures in response to particular levels of heat energy deposition along the crack faces. It is part of the VibroSim model, which simulates vibrothermographic response following a test process parallel to that which would be used in physical experiments.

There are two different test processes: One used with linear excitation and the other used with ultrasonic welder excitation.

Specifically, the linear excitation test process generally involves five steps:

1. A modal vibration analysis to identify resonances (VibroSim_COMSOL).
2. Frequency sweeps to analyze candidate vibrational resonances more precisely. (VibroSim_COMSOL)
3. A tone burst analysis to predict vibrational stresses on the crack (VibroSim_COMSOL)
4. Calculation of crack heating (VibroSim_Simulator)
5. Heatflow simulations predicting the resulting heat flow from each tone burst excitation (VibroSim_COMSOL)

The ultrasonic welder test process involves slightly different steps:

1. A modal vibration analysis to identify resonances (VibroSim_COMSOL).
2. Construction of a synthetic spectrum to verify correct damping behavior
3. Evaluation of the impulse response of the specimen to a welder impact (VibroSim_COMSOL)
4. Evaluation of crack motion (VibroSim_WelderModel)
5. Calculation of crack heating (VibroSim_Simulator)
6. Heatflow simulations predicting the resulting heat flow from crack heating (VibroSim_COMSOL)

Most of the steps substantially involve VibroSim_COMSOL. In general a VibroSim simulation requires an application-specific model construction script (the _comsol.m file). This needs to create a solid model within COMSOL, a suitable mesh, and attach "couplant" and "isolators" as appropriate.

Chapter 2

Installation

2.1 System Requirements

- COMSOL (Version 5.0 or higher), including the Structural Mechanics module, LiveLink for MATLAB, and (in most cases) the CAD Import Module
- MATLAB (Version R2014b or higher)

2.2 Installation Procedure

MATLAB and COMSOL must be installed. Follow the procedure in the LiveLink for MATLAB manual for configuring the version of MATLAB in the COMSOL desktop. To run the VibroSim scripts, you must either install the toolbox provided with the software, or add the `VibroSim_COMSOL` directory to your Matlab Path. The source directory should be used if changes to the source code need to be made.

2.3 Using VibroSim COMSOL

VibroSim scripts can be run directly from MATLAB. Your script can set up variables and then call BuildVibroModel() to build and optionally run a particular configuration of parameters, geometry, flaw, physics, and result processing.

In general you will want to build the VibroSim model from MATLAB with no graphical frontend connected to the COMSOL server (the graphical front end can sometimes interfere). Once the VibroSim scripts have constructed a model on the COMSOL server, you can connect to the COMSOL server from the COMSOL Desktop (an option in the Client/Server Menu), import the VibroSim model from the server (another option in the Client/Server Menu), and then work with the model from both MATLAB and the COMSOL Desktop. Best practice is to use the companion package VibroSim Simulator to manage the running of studies and management of data.

Chapter 3

Modeling Process

The modeling process depends on the goal of your modeling effort. In some cases you might be modeling a part that does not yet exist or that you cannot yet test, and predict what the crack detectability capability and reliability might be once you have the part and do the test. In other cases, you may have a process in place and want to use modeling to enhance your undertanding of the process.

3.1 Solid modeling

You will need to create a geometric solid model of your part in COMSOL, and generate or load a suitable mesh. You also need to adequately model your mounting apparatus and transducer.

In general, you pass a function or function pipeline as the `geometryfunc` parameter to `BuildVibroModel()`. This function is responsible for instantiating the geometry for the specimen, creating a `ModelWrapper`, applying a material, creating couplant and isolators, meshing, and applying boundary conditions. The geometryfunc takes three arguments: The top level `ModelWrapper M`, the top level wrapped geometry object `geom`, and the tag name for the specimen, `'specimen'`. The geometryfunc is presumed to return a single value, the wrapped specimen object.

3.1.1 Generating the solid model

COMSOL has an optional CAD import model that integrates the ParaSolid modeling kernel (the same kernel used in several major CAD packages) into COMSOL and support loading a variety of CAD formats. Geometries can also be constructed directly from 2D and 3D primitives and operations such as extrusions sweeps, etc.

In general generating the mesh for the model can be much harder than creating or importing the model per se, so it my be wise to choose your process for creating/importing the model to ease mesh generation. Specifically, if you want to define a programmatic mesh generation process -- for example to be able to accommodate changes in the model or to simplify mesh refinement studies -- you may need to be very careful in how the model is imported so that you can access geometric primitives by name.

Accessing primitives by name is very important so that you can identify specific edges, boundaries, or domains for use in the meshing process. Otherwise, these can only be manually identified by number. When a new import is performed, the numbering can change and each of those features must be manually re-identified. For objects created within COMSOL, enabling the "create selections" checkbox creates named selections that you can access during the meshing process.

Within VibroSim you can either load in a (optionally pre-meshed) geometry as an entire .mph file at the very start of your process, or build/load a geometry entirely within the geometryfunc that you pass to BuildVibroModel. If you choose to load an .mph file, your VibroSim script will start with

```
    InitializeVibroSimScript('mph_file_name.mph');
```

Otherwise you you pass no parameter to `InitializeVibroSimScript()`. Either way model construction, boundary identification, etc. can be performed by the function parameters to `BuildVibroModel()`

VibroSim is designed to make it much easier to swap out geometry than is normal for tools such as COMSOL. You usually only need to modify or replace the `geometryfunc()`. Your `geometryfunc()` is responsible for creating your specimen geometry, specifying the mesh, specifying the material, and applying suitable boundary conditions (isolator and couplant boundary conditions are automatically generated by the routines that create the isolators and couplant).

For a simple example, look at `CreateRectangularBarSpecimen()`. This is about as simple as you can make a geometry function. There is no couplant or isolators (without couplant you can only do modal analysis). The meshing is defined and the material is defined. Also a 3D view is defined to provide a convenient default representation that looks at the crack.

For a more advanced example, look at `CreateRectangularBarGeometry()`, which calls `CreateRectangularBarSpecimen()` and then creates isolators and couplant and also applys static boundary conditions to the isolators and couplant.

`vibroSim_example5` uses a unique `geometryfunc()` to generate a model of a gear. Models like this are difficult to build and require a lot of back and forth between the gui and the code. The best way to create a model like this is to take it step by step. Start with a script to generate a similar model and look at the generated output in the gui. Then use the gui tools to make an edit to the model and save out the model as a `.m` file. Integrate the new lines of code into the `geometryfunc()` and repeat until the entire model can be generated with the `geometryfunc()`. Use `vibroSim_example5` as a guide.

3.1.2 Generating the mesh

You will need to generate a mesh that is suitable both for the vibration analysis and the heat flow analysis. Detailed meshing instructions can be found in the COMSOL documentation. If you manually construct a set of meshing nodes, and then save the entire .mph file, you can load it using `InitializeVibroSimScript()` as described above. The problem with this approach is that if you use VibroSim to add a hypothesized crack, isolators and/or couplant, you will need to remesh the geometry. (You could get around this problem by creating these additional geometry elements in advance, and then your `geometryfunc` and `flawfunc` would identify the preexisting geometry and wrap it with a `ModelWrapper` object.

Often, it is better to define a mesh generation process based on identifying geometric features by name, and using those features as boundaries of meshing zones. Meshing is nominally done after geometry creation, using `BuildLater` functionality (although `BuildLater` is not strictly required for specimen meshing so long as all geometric features -- including any crack -- already exist.).

Create a `meshbuilder` `BuildLater` object by creating and passing a mesh building function to `BuildLater` (here the variable `specimen` is presumed to be the `ModelWrapper` for the specimen object):

```
specimen.mesh=BuildLater(M,[specimen.tag '_mesh'],...
                'meshbuilder', ...
                @(M,mesh,obj)  ...
                BuildMeshDCObject(M,geom,mesh,specimen,obj, ...
                 'spc', ...  % dcprefix
                 specimen.getdomainselection, ...
                 [ geom.tag '_', tag, '_edg'], ...
                 specimen.gettopfaceselection, ...
                 specimen.getbottomfaceselection));
```

The mesh building function will be called with three parameters: The top level model wrapper `M`, the top level mesh object `mesh`, and the created `BuildLater` mesh object `meshobj`. The example above creates inline a MATLAB anonymous function (`@` notation) which takes those three arguments and more variables from the parent scope (`geom` and `specimen`).

The mesh building function needs to instantiate COMSOL nodes to mesh the entire geometry of the object. In general regions of the object with complicated geometry or fine detail (such as holes, a crack, etc.) should be meshed separately from bulk material. This requires that the regions be separated into multiple COMSOL domains, such as with a `Partition` object. In general swept meshes are very effective for extrusions, but will fail if anything (such as a crack, hole, etc.) interrupts the extrusion. `FreeTet` is a very effective mesher for small, constrained domains, but it does not seem to work very well for large complicated volumes. `FreeTri` can be used to tesselate boundaries prior to running `FreeTet`. See the chapter "Meshing" in the COMSOL Multiphysics Reference Manual for more information.

In writing your mesh building function you will need to refer to geometric entities in a consistent way. The best way to do this is through automatically generated named selections. This process is documented on the COMSOL website: *Automatically Handling Selections in COMSOL Multiphysics*. COMSOL creates these named selections for geometric features when the 'Create selections' checkbox is selected for the geometric feature.

Be warned that geometry changes (such as adding a crack, adding isolators, or adding couplant) will change the numbering of geometric entities including changing the meaning of manually-created named selection. Using the automatically-generated named selections seems to be the *only* reliable way to identify geometry by its construction for use in the meshing process.

Please note that all COMSOL meshing nodes must be explicitly mapped to specific domains. If you leave a meshing node set to mesh all remaining geometry, it will attempt to mesh the couplant and isolators that aren't built until a later phase, and then conflict with the meshes built specifically for those objects.

The mesh needs to be fine enough for accurate analysis. Usually the vibration analysis is the most sensitive to inadequate meshing. In general, there should be at least several elements per wavelength, with wavelengths $\lambda = c/f$determined by the wavespeeds c of the different kinds of waves and partial waves involved.

Obviously it is the lowest c that matters. Usually this will be the bending wave. For a thin plate with a vibrational wave of frequency f,

$$c_b = \left(\frac{4\pi^2 E h^2 f^2}{12\rho(1-\nu^2)} \right)^{\frac{1}{4}}$$

but this prediction may be unnecessarily conservative, as out-of-plane curvature can increase the effective bending stiffness dramatically and the wavespeed with it.

3.1.3 Applying material properties

Material properties are usually specified in your geometry creation function with a `BuildLater` of class `applymaterial` that is called with two parameters: The top level `ModelWrapper` `M` and the `BuildLater` object `obj`. The example below applies a named material ('Titanium') to the object. A function, GetDomain(M,geom,specimen) is passed that will extract and return the COMSOL domain entity numbers corresponding to `specimen`'s COMSOL node that will be given this material:

```
BuildLater(M,[specimen.tag '_applymaterial'],...
           'applymaterial', ...
           @(M,obj) ...
           ReferenceNamedMaterial(M,geom,specimen,...
           'Titanium', ...
           @(M,geom,specimen) GetDomain(M,geom,specimen)));
```

3.1.4 Creating isolators and couplant

Couplant is the layer of protective material between your excitation transducer and your specimen. Isolators are similar layers of material between your specimen and its mount. Couplant and isolators form the boundary of the vibrational simulation: The details of your excitation transducer and mounting apparatus are beyond the couplant and isolators and are *not* explicitly modeled.

Proper use of couplant and isolators is necessary to make vibrothermographic testing repeatable. Otherwise the unpredictability of the contact nonlinearity and the complexity of the dynamic response of the mounts and transducer combine to add a huge amount process variability to the physical system. With such inconsistent experimental behavior, modeling is not meaningful.

Couplant, modeled as a viscoelastic layer between your specimen and your transducer, acts as a spring that serves to isolate the dynamic behavior of the transducer from the dynamic behavior of your specimen. It inhibits the formation of complicated and unpredictable system resonances between the transducer and specimen (in the condition that the couplant mechanical mobility is much larger than the transducer mechanical mobility), and allows the transducer to be dynamically modeled as a displacement boundary condition on the opposite side of the couplant from the specimen.

Isolators, likewise modeled as viscoelastic layers between your specimen and mounts, act as springs that isolate the dynamic behavor of the mounts from the dynamic behavior of your specimen. They inhibit the formation of unpredictable system resonances between mounts and specimen (in the condition that couplant mechanical mobility is much larger than mount mechanical mobility), and allow the mounts to be dynamically modeled as zero displacement boundary conditions.

To create isolators and couplant, you can either create them explicitly in your `geometryfunc` with CreateThinIsolator() and/or CreateThinCouplant(), or you can wrap your `geometryfunc` with a call to `AttachThinCouplantIsolators()`. An advantage of using `AttachThinCouplantIsolators()` is that it creates the couplant and isolators using `BuildLaterWithNormal()` which automatically orients the couplant and isolator normal to the specimen. This function takes seven arguments:

1. The top level ModelWrapper (`M`)
2. The top level wrapped geometry (`geom`)
3. Wrapped COMSOL object representing the specimen geometry.
4. A single row-vector representing the coordinates for placing the couplant (`couplant_coord`).
5. A matrix -- series of row vectors -- each of which represent the desired position of an isolator (`isolator_coords`).

3.1.5 Creating the crack or flaw

The function to create a simulated crack should be passed to `BuildVibroModel` as the `flawfunc` parameter. It is called with three parameters: The top level `ModelWrapper` `M`, the top level wrapped geometry object `geom`, and the wrapped specimen object as returned from `geometryfunc`, `specimen`. Usually, this will just be a call to CreateCrack. In our first example, the crack is located at fixed location `[ 0.2 0.4 0.0 ]` meters with (major axis) half-length `.003` m and (minor axis) depth `.002`m. The major axis is `[0,1,0]` (along y) and the minor axis is `[0,0,-1]` (along $-z$). The subradii on which to break up the mesh are given as `[ 0.001, 0.002, 0.003 ]` meters. The crack provides its strain measurement from and derives its vibrothermographic heating calculation from the vibrations calculated in COMSOL physics model `solidmech_harmonicburst`, and `solidmech_harmonicsweep`. A text file is passed where heating data on the crack can be stored. Finally a cracktype is provided that can be either `penny` or `through`.

```
bldcrack = @(M,geom,specimen) CreateCrack(M,geom,'crack',specimen, ...
            { 0.2, 0.4, 0.0 }, ... % crackx, cracky, crackz
            0.003, ...  % semi major axis length
            0.002, ...  % semi minor axis length
            [0,1,0], ...  % axis major direction (growth along surface)
            [0,0,-1], ... % axis minor direction (growth into depth)
            [ .001, .002, .003 ], ... % subradii for different COMSOL boundaries
            {'solidmech_harmonicsweep','solidmech_harmonicburst'}, ... % vibration physics ↩
                models where the crack should be present
            dc_dummy_heatingdata_href{1},... % Text file to hold crack heating energies.
            cracktype);
```

Please note that since CreateCrack() adds additional geometric objects to the specimen and forms a union between these additional geometric objects and the specimen, that this union should be considered the final representation of the specimen. The final representation of specimen and crack is accessible within MATLAB as a property `crack.unionwithspecimen` of the crack object.

3.1.6 Applying boundary conditions

Boundary conditions in VibroSim are created as `BuildLater` objects through the `AddBoundaryCondition()` function. Most of the boundary conditions needed are automatically created. For example, when you create couplant and isolators, the continuity boundary conditions between couplant or isolator and specimen are created. In addition, the dynamic boundary condition on the far side of the couplant or isolator is created -- fixed condition for couplant or excitation displacement for isolator -- is also created automatically. By contrast, static boundary conditions holding the couplant and isolators in place are *not* created automatically because the forces are dependent on how the specimen is mounted. So if you want to do static analysis, you will need to apply static constraints as additional boundary conditions.

Boundary conditions are specified by physics class -- which represents which classes of physics models will get the boundary condition (each physics model has a physics class specified on creation) -- and by boundary condition class. Boundary condition classes are selected for activation when the physics is used in a study step. For a boundary condition to be active on a physics model in a study step, the physics model must be of one of the physics classes specified for that boundary condition, *and* at least one of the boundary condition classes specified for the boundary condition must be specified for the study step.

Physics classes
Physics class: `solidmech_modal`, *Use:* Dynamic modal analysis
Physics class: `solidmech_static`, *Use:* Static deformation analysis
Physics class: `solidmech_harmonicper`, *Use:* Dynamic harmonic perturbation of static deformation

Physics class: `solidmech_harmonic`, *Use:* Dynamic harmonic analysis -- instantiated as both `solidmech_harmonicsweep` and `solidmech_harmonicburst`
Physics class: `solidmech_multisweep`, *Use:* Multi-sweep analysis -- used to generate ultrasonic welder impulse response
Physics class: `heatflow`, *Use:* Conduction heat transfer

Boundary condition classes
Boundary condition class: `continuities`, *Use:* Continuity boundary conditions between different pieces of an assembly
Boundary condition class: `staticloading`, *Use:* Static boundary conditions (DC constraint) on specimen
Boundary condition class: `excitation`, *Use:* Dynamic excitation boundary condition
Boundary condition class: `fixedisolators`, *Use:* Dynamic fixed constraint boundary condition on isolators
Boundary condition class: `crackheating`, *Use:* Heat flow on to crack faces

The following example illustrates a static boundary condition on an isolator with `ModelWrapper` `specimen.blisolator`. The new boundary condition gets tag `<specimen.blisolator tag name>_blfixed` for physics `solidmech_static` and boundary condition class `staticloading`. An anonymous function that wraps `BuildFaceFixedBC()` will be stored in a BuildLater object that will be cloned and constructed within each applicable physics model.

```
AddBoundaryCondition(M,geom,specimen.blisolator,[specimen.blisolator.tag '_blfixed'], ...
         'solidmech_static', ...  % physics
         'staticloading', ...     % BC class
         @(M,physics,bcobj) ... % parameters passed to BC creation
         BuildFaceFixedBC(M,geom,physics, ...
               specimen.blisolator, ...
               bcobj, ...
               specimen.blisolator.getfreefaceselection)); % function to identify boundary
```

3.2 Modal Analysis

Typically, once you have loaded and meshed a new geometry, you will want to do a modal analysis. The modal analysis determines the vibrational resonant structure of the specimen. It is controlled by the `simulationeigsaround` parameter and the `simulationneigs` parameter, which determine near what frequency (in Hz) to search and how many modes to find.

The result of the modal analysis is a set of mode frequencies, each each with a different mode shape. Usually you would select some subset to focus on in a vibrothermography test (if you use a broadband vibrothermography system), or you may be exciting a range of them simultaneously through exciter tip contact nonlinearity (if you use a narrowband vibrothermography system). Typically you will want to determine which resonances give large amounts of dynamic strain across the region of likely crack locations. Your goal in performing the NDE test will be to excite enough of these modes with enough amplitude to detect a crack regardless of the exact position of the crack.

You can identify suitable modes by viewing the modal analysis results and selecting each mode in turn. You will want to identify modes where there are large strains in the relevant region. Crack heating is believed to originate from strains normal to the crack face and from shearing strains on the crack face, so verify that the strains in the relevant region include components in the necessary directions.

The modal analysis model can be created by `CreateVibroModal()` and is instantiated with `VibroPhysics()` when `mode` is set to `'modal'`, `'broadbandprocess'`, `'welderprocess'`, or `'all'`.

3.3 Static Analysis

Static analysis is used to determine the static forces on the couplant and isolators as well as any static deformation of the specimen. It is not part of the normal vibrothermography process, but can be useful in some circumstances:

- Where couplant or isolators are nonlinear-elastic, to determine the linearization point for harmonic or harmonic perturbation analysis.
- To consider (in combination with harmonic perturbation analysis, and with the nonlinear flags set) geometric nonlinearity and the effect of geometric nonlinearity on resonant frequencies.

- To consider (in combination with harmonic perturbation analysis) the total (static+dynamic) stresses applied to a part and to make sure that those stresses do not approach the fatigue limit or have a risk of reducing the lifetime of the part

The static analysis model can be created by `CreateVibroStatic()` and is instantiated with `VibroPhysics()` (but without the nonlinear flag set) when `mode` is set to `'static'` or `'all'`.

3.4 Harmonic Perturbation Analysis

Harmonic perturbation analysis is used to determine vibrational perturbations that are superimposed over static deformation. It is useful

- Where couplant or isolators are nonlinear-elastic, to perform linearized vibration analysis around a linearization point.
- To consider (in combination with static analysis, and with the nonlinear flags set) geometric nonlinearity and the effect of geometric nonlinearity on resonant frequencies.
- To consider (in combination with static analysis) the total (static+dynamic) stresses applied to a part and to make sure that those stresses do not approach the fatigue limit or have a risk of reducing the lifetime of the part

The harmonic perturbation analysis model can be created by `CreateVibroHarmonicPer()`. It is not instantiated `VibroPhysics()`.

3.5 Harmonic Analysis -- Sweep mode

Harmonic analysis in sweep mode is used to fine-tune the identification of resonant modes and frequencies. Typically you will enter a range of frequencies immediately surrounding a desired resonance. This is important because not all characteristics -- for example material and couplant/isolator loss factors -- are modeled identically in harmonic analysis vs. modal analysis, so there can be a slight frequency shift from that predicted in modal analysis. Similar narrowband frequency sweeps are similarly used in physical experiments to identify small shifts in frequency that tend to occur as amplitude is increased, due to nonlinearity in the transducer and/or specimen.

The sweep mode harmonic analysis generates a pair of spectral plots of motion at the selected laser vibrometer location (parameters `laserx`, `lasery`, `laserz` control the location and `laserdx`, `laserdy`, and `laserdz` control the direction of sensitivity) and of strain magnitude across the hypothesized crack. These are useful in fine-tuning the frequency as well as identifying how much heating might occur.

The harmonic sweep physics is controlled by three DC parameters:

1. `simulationfreqstart`: Starting frequency for the sweep
2. `simulationfreqstep`: Step frequency for the sweep
3. `simulationfreqend`: end frequency for the sweep

The harmonic analysis model can be created by `CreateVibroHarmonic()` and a sweep-mode form is instantiated by `VibroPhysics()` when `mode` is set to `'harmonicsweep'`, `'broadbandprocess'`, or `'all'`.

3.6 Harmonic Analysis -- Burst mode

Harmonic analysis in burst mode is used to simulate a vibrothermographic test at a particular excitation frequency. The magnitude of the engineering dynamic strain across the crack (defined as the vector magnitude of the complex magnitude of the tensor product $\varepsilon_{ij}n_j$ between engineering dynamic strain tensor ε_{ij} and the crack face normal n_j) is usually used as an input to the crack heating model to determine the energy flux boundary condition for the heat flow analysis.

The term "engineering strain" is used in this context to refer to the strain that would exist at the crack location if the crack were not present. This parameter is used to predict heating for several reasons:

1. The presence of the crack perturbs the strain field in its immediate vicinity in a complicated way. There is no clear alternative.
2. Engineering strain is the parameter captured during the calibration experiments used to evaluate the parameters of the crack heating model.
3. In almost all cases where predicting crack heating is important, the hypothetical crack is small enough that its effect on the modal vibration pattern and frequency will be minimal.

Therefore it is very important that the burst mode harmonic analysis used to calculate the vibrational input to the crack heating model should *not* include a discontinuity between crack faces. This is accomplished by not setting the `crackdiscontinuity` flag to `CreateVibroHarmonic()`. It can be verified by observing surface plots of the resulting stress field and confirming that no stress concentration is visible around the crack.

The harmonic analysis model can be created by `CreateVibroHarmonic()` and a burst-mode form is instantiated by `VibroPhysics()` when `mode` is set to `'harmonicburst'`, `'broadbandprocess'`, or `'all'`. The frequency of the burst is controlled by the DC parameter `simulationburstfreq`.

3.7 Multisweep analysis

Multisweep analysis is used for ultrasonic welder excitation modeling to obtain a minimally sampled frequency domain representation of the specimen response to an impact (impulse) at the welder contact location. By using different frequency domain spacings over different frequency ranges, the number of samples required to get an impulse response can be drastically reduced compared to a direct time-domain calculation. The multisweep analysis is split into four segments (seg1, seg2, seg3, seg4) that can be computed in series or (potentially) parallel. The process_multisweep step of VibroSim_Simulator then converts the results from the four frequency domain segments into a time-domain impulse response.

3.8 Heat Flow Analysis

Heat flow analysis is used to simulate the flow of heat from the partial annuli that make up the crack faces. The amount of heat flow can be specified as coming from the file given as the second to last argument `heatingfile` to CreateCrack().

When viewing the results of the heat flow analysis it is common to subtract COMSOL's default background temperature of 20 deg. C (293.15 deg. K) and also to add in synthetic camera noise. The recommended expression is `T-293.15 + cameranoise(x,y,z)`. The noise-equivalent temperature difference (NETD) of the camera noise is specified using the DC parameter `simulationcameranetd`.

The heatflow analysis model can be created by `CreateVibroHeatFlow()` and is instantiated by `VibroPhysics()` when `mode` is set to `'heatflow'`, `'broadbandprocess'`, `'welderprocess'`, or `'all'`.

Chapter 4

Structure

4.1 Wrapping COMSOL nodes with MATLAB objects

All COMSOL nodes used during construction of the COMSOL model are wrapped with MATLAB objects of `class ModelWrapper`. The ModelWrapper provides many benefits:

- Provides pass-by-reference semantics within MATLAB, in contrast to most other MATLAB objects, which are pass-by-value (because `ModelWrapper` is indirectly derived from `handle`.
- Provides the ability to dynamically add additional properties to an instance (because `ModelWrapper` is derived from `dynamicprops`.
- Provides a consistent means -- through object properites -- to store common characteristics, such as the tag name, the COMSOL node, and the COMSOL parent (used in case we want to delete the COMSOL node).
- Provides means -- through object properites -- to store related objects.
- All `ModelWrapper`s are entered in a master model registry stored in a global MATLAB variable, so they can be looked-up by tag. This is very useful, for example, if your model building script fails and you want to interactively debug. The top level model object can for example be extracted with

  ```
  M=FindWrappedObject([],'Model');
  model=M.node;
  ```

  Other objects can be found by tag name via

  ```
  WrappedObject=FindWrappedObject(M,tagname);
  ```

The `ModelWrapper` for the top level COMSOL object is generally referred to as `M` and is passed as a parameter to most VibroSim functions.

4.2 BuildLater objects

A common situation when constructing a COMSOL model tree is the desire to *specify* objects to be constructed at one phase of the construction process, but to actually build them later. One example of this is in meshing. Usually you don't want to start the meshing process until the geometry is finalized, but in many cases you want to specify how each geometry component is to be meshed as it is constructed.

To address cases like these, VibroSim defines the `BuildLater` object, which represents an object that is specified at one phase of model construction, but where the COMSOL node is instantiated at a later phase. `BuildLater` objects are created with the `BuildLater` constructor which takes four parameters: The top level model object `M`, the tag for the new object, the BuildLater class for the new object (which controls when it will be instantiated and what parameters are provided for the instantiation), and the code object that will perform the instantiation. Usually the code object is an "anonymous function": A dynamically-generated MATLAB function that provides instructions for building the object.

4.2.1 Use of anonymous functions

Anonymous functions, sometimes alternatively referred to as "lambdas" are short, inline, dynamically generated MATLAB functions that make it relatively easy to pass code as a function parameter. The basic syntax for a MATLAB Anonymous function is:

```
gettopface  = @(M, geom, blockobj) GetBlockFace(M,geom,blockobj,[0 0 1]);
```

The parentheses following the '@' specify the parameters that must be specified when calling the function. Other values and parameters used are taken from the context where the anonymous function was defined. The anonymous function is a code object that can be passed around and stored like any other MATLAB variable, and can be executed when needed.

4.2.2 Functions for instantiating `BuildLater`objects

Function: `BuildLater`, *Use:* Class constructor (general purpose
Function: `BuildLater`, *Use:* Class constructor (general purpose
Function: `AddBoundaryCondition`, *Use:* Specifying boundary conditions for later instantiation during assembly of the physics tree
Function: `BuildLaterWithNormal`, *Use:* Specifying geometric (and other) objects -- usually attachments such as couplant and isolators -- to be built after the main geometry has been constructed and surface normal vectors can be evaluated, so that the attachment can be properly oriented.
Function: `RunLater`, *Use:* Specifying code (as opposed to an object to construct) that needs to be run in a later phase of model construction

4.3 Phases of model construction

(These phases correspond to the execution of `BuildVibroModel()`)

Phase: Between execution of InitializeVibroSimScript() and BuildVibroModel(), *Purpose:* Custom initialization and parameter setting
Phase: Parameter setting via `paramfunc()`, *Purpose:* Set parameters of the simulation
Phase: Geometry construction, *Purpose:* Build specimen, define isolators, contactors, materials, boundary conditions, etc. via `geometryfunc()`
Phase: Flaw construction, *Purpose:* Instantiate flaw via `flawfunc()`
Phase: Initial mesh instantiation, *Purpose:* Define a mesh over the specimen prior to normal extraction
Phase: Normal extraction, *Purpose:* Extraction of surface normals of specimen, use of those normals to properly orient and instantiate previously-defined objects such as isolators and contactors.
Phase: Remaining mesh instantiation, *Purpose:* Meshing of all remaining objects such as those created after normal extraction
Phase: Applying materials, *Purpose:* Defining which materials apply to which geometric domains
Phase: Building physics and studies, *Purpose:* Create physics and studies, and instantiate boundary conditions via `physicsstudyfunc()`
Phase: Selection of boundary conditions, *Purpose:* Boundary conditions for each physics must be selected once all physics nodes have been created
Phase: Instantiate results nodes, *Purpose:* Create plots, etc. via `resultsfunc`
Phase: Save and run model, *Purpose:* Save model -- in case of trouble executing -- and run all studies if desired
Phase: Save final results, *Purpose:* Save model.

4.4 Parameters and DCParameters

VibroSim provides support for internally and externally specified and controlled parameters through an internal parameter database `Paramdb` which can supplement external parameters known as `DCParams`. These two types of parameters can be used identically within VibroSim; the only difference is in how they are initialized. Local `Paramdb` entries are instantiated from `AddParamToParamdb()` whereas `DCParams` are instantiated externally and accessed via remote procedure calls (currently

implemented using the D-Bus RPC engine). Attempting to create a local Paramdb entry will fail if a DCParam of the same name exists.

For the most part, numeric Paramdb entries and DCParams are transformed into COMSOL parameters (in COMSOL's 'Definitions' node) via the ObtainDCParameter() call. Note that changing the local Paramdb or remote DCParam after the ObtainDCParameter() call will not affect the COMSOL parameter.

In some cases, such as for string parameters or for numeric parameters where numeric evaluation within MATLAB is required (or where COMSOL requires numbers in place of a named parameter), the Paramdb entry or DCParam is instead extracted directly via GetDCParamNumericValue() or GetDCParamStringValue.

4.4.1 Algebraic expressions represented as strings

In most cases COMSOL supports using expressions in place of numeric input. These expressions can use COMSOL parameters. Provided a Paramdb entry or DCParam has been extracted via ObtainDCParameter(), its name can be used like a variable in such expressions.

It is fairly common to need to do arithmetic or form mathematical expressions based on such parameters, without wanting to evaluate them directly (so the parameter could be changed later and the expression would still be correct). This is done extensively within VibroSim and is a common practice within the code. One example of this is the function magnitude_cellstr_array() which takes as a parameter a MATLAB cell array of strings. Each string is presumed to represent a vector component. The function (code shown below) assembles the needed expression as a string using MATLAB bracket concatenation ([and]). *Please note that when assembling expressions using bracket concatenation it is very important to parenthesize every externally-provided input to ensure that the input is used as a unit and cannot be broken by operator precedence -- such as the multiplication operator multiplying only the last term of a sum.*

```
function magnitude=magnitude_cellstr_array(array)
% function magnitude=magnitude_cellstr_array(array)
%
% Calculate the vector magnitude of a vector represented
% as a cell array of strings

% Initialize magnitude-squared to 0
magnitudesq='0.0';

% Accumulate the absolute square of each element
for cnt=1:length(array)
  magnitudesq=[ magnitudesq ' + abs(' array{cnt} ')^2' ];
end

% return the square root.
magnitude=[ 'sqrt(' magnitudesq ')' ];
```

Chapter 5

DC Parameters

Variables that VibroSim needs to use are stored in a parameter database. Generally these variables are called DC (Datacollect) parameters. VibroSim COMSOL can be run concurrently with a Datacollect session, allowing for simulations to be run on data as is being collected. Two parameter databases are used, one kept in MATLAB that holds any parameters that may be used. Parameters can be added to this database using the `AddParamToParamdb` function. The second database is kept in COMSOL itself, and is intended to only have the parameters that are in active use in the COMSOL model. The function `ObtainDCParameter` will pull the DC parameter from the Paramdb and place it in the COMSOL parameter database.

This chapter documents all the DC Parameters in active use in VibroSim COMSOL at the present time.

DC PARAMETERS

blmountoffsetx Bottom left mount offset.

brmountoffsetx Bottom right mount offset.

calcvib description

calcvib2 description

couplantdashpotcoeff Dashpot coefficient of the couplant. The isolator is a spring foundation with damping.

couplantfacemethod Face method for the couplant. Defaults to `FreeQuad`

couplantlength Length of the couplant.

couplantmeshsize Size of the mesh when there are no geometric constraints.

couplantmeshtype Mesh type of the couplant. Defaults to `HEXAHEDRAL`.

cracksemimajoraxislen Crack length in the semi-major axis. For an half-ellipse crack, the semimajor axis extends along the surface of the material.

cracksemiminoraxislen Crack length in the semi-minor axis. For an half-ellipse crack, the semiminor axis extends into the depth of the material.

couplantsweepelements Number of elements in the sweep direction.

couplantthickness Thickness of the couplant.

couplantwidth Width of the couplant.

couplantYoungsModulus Youngs modulus of the couplant. The couplant is simulated as a spring foundation. The stiffness of the spring being the Young's modulus divided by the thickness.

excitation_t0 Vibration timing: the start of the envelope ramp-up.

excitation_t1 Vibration timing: end of the envelope ramp-up.

excitation_t2 Vibration timing: start of the envelope ramp-down.

excitation_t3 Vibration timing: end of envelope ramp-down.

impulseexcitation_t0 Center of the impulse exciation.

impulseexcitation_width With of the impulse excitation.

isolatordashpotcoeff Dashpot coefficient of the isolator. The isolator is a spring foundation with damping.

isolatorfacemethod Face method of the isolator. Defaults to `FreeQuad`.

isolatorlength Length of the isolator.

isolatormeshsize Size of the mesh when there are no geometric constraints.

isolatormeshtype Mesh type of the isolator. Defaults to `HEXAHEDRAL`.

isolatorsweepelements Number of elements in the sweep direction.

isolatorthickness Thickness of the isolator.

isolatorwidth Width of the isolator.

isolatorYoungsModulus Youngs modulus of the isolator. The isolator is simulated as a spring foundation. The stiffness of the spring being the Young's modulus divided by the thickness.

laserx Location of the first laser vibrometer spot, x component.

lasery Location of the first laser vibrometer spot, y component.

laserz Location of the first laser vibrometer spot, z component.

laserdx Direction that the first laser vibrometer is pointing, x component.

laserdy Direction that the first laser vibrometer is pointing, y component.

laserdz Direction that the first laser vibrometer is pointing, z component.

laser2x Location of the second laser vibrometer spot, x component.

laser2y Location of the second laser vibrometer spot, y component.

laser2z Location of the second laser vibrometer spot, z component.

laser2dx Direction that the second laser vibrometer is pointing, x component.

laser2dy Direction that the second laser vibrometer is pointing, y component.

laser2dz Direction that the second laser vibrometer is pointing, z component.

meshsize Size of the mesh when there are no geometric constraints.

meshsizemin Minimum mesh size. COMSOL will try to limit the mesh size to this value when meshing around tight curvatures.

seg1_freqstart Starting frequency for a frequency sweep.

seg1_freqstep Frequency step for a frequency sweep.

seg1_freqend Final frequency for a frequency sweep.

seg2_freqstart Starting frequency for a frequency sweep.

seg2_freqstep Frequency step for a frequency sweep.

seg2_freqend Final frequency for a frequency sweep.

seg3_freqstart Starting frequency for a frequency sweep.

seg3_freqstep Frequency step for a frequency sweep.

seg3_freqend Final frequency for a frequency sweep.

seg4_freqstart Starting frequency for a frequency sweep.

seg4_freqstep Frequency step for a frequency sweep.

seg4_freqend Final frequency for a frequency sweep.

simulationburstfreq Excitation frequency for a single frequency burst study.

simulationcameranetd Camera noise equivalent temperature difference. A measure of the noise present in the image.

simulationcrackx Crack center location, x component.

simulationcracky Crack center location, y component.

simulationcrackz Crack center location, z component.

simulationeigsaround The modal analysis will search for modes around this frequency. Set it to 1 to get all frequencies starting at zero.

simulationfreqend Ending frequency for a frequency sweep.

simulationfreqstart Starting frequency for a frequency sweep.

simulationfreqstep Step frequency for a frequency sweep.

simulationsurfaceemissivity Surface emissivity to be used in the simulation. A black-body radiator has a surface emissivity of 1.0, this is the default.

simulationtimeend When to end the heatflow simulation. Typically aligned with the end of the excitation.

simulationtimestart When to start the heatflow simulation. This will align with the beginning of the excitation. Will usually be zero as the excitation usually starts at t=0.

simulationtimestep Time step for the heatflow simulation.

spclength Length of the specimen. Used in the `CreateRectangularBarSpecimen()` function.

spcmaterial Material of which the specimen is made.

spcrayleighdamping_alpha Rayleigh damping coefficient alpha. Rayleigh damping is visous damping that is proportional to the mass and stiffness matrices of the specimen. Alpha is the coefficient of the mass matrix in the equation. Conceptually, increasing alpha will increase the damping in the low frequencies.

spcrayleighdamping_beta Rayleigh damping coefficient alpha. Rayleigh damping is visous damping that is proportional to the mass and stiffness matrices of the specimen. Beta is the coefficient of the stiffness matrix in the equation. Conceptually, increasing beta will increase the damping in the high frequencies.

spcthickness Thickness of the specimen. Used in the `CreateRectangularBarSpecimen()` function.

spcviscousdamping Velocity dependent damping of the motion of the specimen.

spcwidth Width of the specimen. Used in the `CreateRectangularBarSpecimen()` function.

timedomain_end_time End time of the time domain impulse response study.

timedomain_start_time Start time of the time domain impulse response study.

timedomain_step_time Time step of the time domain impulse response study.

tlmountoffsetx Top left mount offset.

trmountoffsetx Top right mount offset.

Chapter 6

Reference