Top nine hints for working with Comsol

From Site

Contents

Hint 1. Save as Java

The greatest time-saving hint of all.

Steps

1. `File > Reset History`

Comsol remembers every edit you had made. So if you had added a rectangle and then deleted it, both of these commands will appear in the generated Java file. Resetting history will give you a Java file listing steps for going from an empty model to the final state of the model directly, with the steps logically grouped.

2. `File > Save As Model Java-File...`

3. Edit the file and comment out entries for running the simulations

Such as

```
model.sol("sol1").runAll();
```

4. Compile

Using a command such as

```
C:/Programs/COMSOL43a/bin/win32/comsolcompile.exe -jdkroot C:/Programs/Java/jdk160/ example.java
```

Note that with the current version of Comsol (4.3.2.189 as of this writing, 13 January 2014), JDK 7 cannot be used yet. (Note that for Linux, JDK 6 may be downloaded as a self-extracting archive with extension `bin`.)

5. Open in Comsol

Select the `Compiled Model Java File (*.class)` file type from the `Open` dialog box.

When you open the file, all of the commands in the Java file are processed, and you get your model back.

What this allows you to do

- Copy and paste anything from one model file to another. (By copying the Java commands for creating these.) This saves many hours.
- Change a model from 2D axisymmetric to 3D. (Don't forget to delete all of the geometry first. You might have to change other items, if an error message appears when loading.)
- Generate complicated geometry and import it into Comsol by automating the process of manually adding the features via the Comsol user interface. For example, I needed to generate a hexagonal network of hundreds of edges, and there was no easy way to accomplish this from within Comsol.
- Load a model made in a later version of Comsol into an earlier version. (For example: 4.3 to 4.2a.)

You don't need to know Java in order to work with a Comsol-generated Java file! These files use only the simplest Java syntax. They are essentially lists of commands. It's easy to modify the code by following the examples.

Hint 2. Watch the absolute and relative tolerances

By default these are usually set too high for certain types of physics, such as electric currents. So the solution you get may look OK, but be nothing like the true solution, especially for time-dependent simulations. I usually set these to 1e-8 each, and if that doesn't work, go to 1e-5, but not higher.

Somewhat counter-intuitively, setting these values lower also reduces the amount of convergence errors in some cases.

Hint 3. Use Livelink for Matlab for post-processing

The most flexible approach is to extract the data using `mphinterp(..)`, which accepts a set of points and returns the value of the solution at those points. Then process the data using e.g. Matlab.

If working on a server, you may need to do it like this:

```
comsol server -tmpdir $TMPDIR </dev/null >/dev/null &
matlab -nodesktop -nosplash

addpath('/usr/local/comsol/4.3b/mli')
mphstart()
```

Note that the `/dev/null` parts may be required, for some reason, for the Comsol server to not terminate immediately. The variable `$TMPDIR` should point to a directory with enough space.

Hint 4. Use `mphnavigator`

When working with a complex model, such as a nested parametric sweep with a time-dependent simulation inside, it may not be obvious what to supply to the `mphinterp(..)` function for the `'dataset'`, `'outersolnum'`, etc. arguments. Then `mphnavigator` is great for investigating the structure of the model file. It will answer questions such as:

- Which dataset corresponds to which solution?
- How do solution tags map to parametric sweep numbers (`'outersolnum'` arguments)?

Hint 5. Build trivial models to learn how things work

For example, learning the difference between bidirectional and unidirectional constraints is much easier by trying it out in a model than by reading about it in the documentation.

It's often suitable to use 1D models for experimentation, where meshing is not a problem and solutions are quick. (Note that in Comsol, all models are conceptually three-dimensional, but the geometry can be 1D, 2D, 3D, as well as the axisymmetric variants. The meaning is that the model is redundant, in other words symmetric, in 2 or 1 dimension, or radially about a point or an axis.)

Hint 6. Run a super-coarse-mesh pilot version of your model

This will solve quickly and will let you troubleshoot many issues with short turn-around time. Otherwise you might find yourself in a situation where post-processing your 96 GB 36-hour solution reveals that the voltages in the region of interest are zero, as a result of a small typo.

Hint 7. Geometry object selection tricks

Use wireframe rendering. This way it's easy to select internal faces visually, by clicking a few times on the same pixel.

Select domains, boundaries, edges, points by typing in their numbers. (E.g. in case an error occurs and the message give you a few IDs.) To do this, use a spare *Explicit* selection, and click the *Paste* icon. Type in the numbers in the window that appears; format is like so: `1, 4, 8-14` .

Hint 8. Stay with default units

Under *Geometry*, you can set the *Length unit* to whatever is comfortable given the scale of your model, for example micrometres, but **don't do that!** When it's time to work with the model through Matlab, some of the length units will need to be given in the scaled unit (micrometres) and some in the default unit (metres), and it's easy to get confused and waste time misinterpreting unexpected results.

Instead, think in micrometres, type that number directly, followed by "e-6", think in millimetres, type "e-3". Same for microseconds and milliseconds. This is much easier than figuring out how many zeros to put after the decimal point, and easier to read, because you don't have to take the extra mental step (even if simple) of converting what's written to the units you're thinking in.

Hint 9. Try assemblies for flexibility with meshing

For simple models, *Adaptive Mesh Refinement* is the best and easiest way to build an appropriate mesh. (An efficient way to use AMR that is relevant to many time-dependent simulations, is to first run a stationary solver with AMR, and then use the mesh it produces.) But sometimes you need to have a high degree of control of your meshes. In this case, assemblies let you mesh parts of your model completely independently.

In *Geometry*, switch the method under *Form Union* to *Form Assembly*. This avoids the final geometry step of fusing the geometric entities together. Instead, under *Geometry*, add *Union* items from the *Boolean operations* submenu, one for each part of the model that you wish to mesh separately.

The boundaries between the remaining geometric entities, after these final *Union* steps, appear as usual, but each boundary is actually two, coinciding in space: one belonging to the domain on one side of it, and the other to the domain on the other side, in a different assembly unit. If a variable is defined only on one of those domains, then it cannot be directly referenced from the other domain across this boundary. To reference it from a boundary condition on the domain across the assembly boundary, (suppose the variable is called `V`), write it as `src2dst_ap1(V)`, where `ap1` is the name of the identity pair that has been created when making an assembly. Check how the source and destination of the identity pair have been defined under *Definitions* of the model; this expression may need to be `dst2src_ap1(V)`. It helps making explicit selections from the identity pair *Source* and *Destination* boundary lists, and naming them in a useful way (such as `au12` for the boundaries belonging to assembly unit 1, interfacing with assembly unit 2): use the "copy" icon next to the list.

The BCs should be defined on the more finely meshed side of each pair. This applies whether you use references such as `src2dst_ap1(V)` explicitly, or the special BCs from the *Pairs* submenu, such as *Continuity*, which use them implicitly (as can be seen via *Equation View*). In other words, if a variable is referred to another side (suppose it exists on both), it should be referred from the coarsely meshed side to the finely meshed side. Otherwise the model is likely to run into convergence problems. (Because there are not enough constraints, as was explained to me by Comsol support.) To see what I mean, try the two cases in a 2D model made up of a couple of squares as assembly units.

In one case, I found that a complex boundary condition (simulating an RC layer at an internal boundary) could not be written such that we refer each variable from the coarser to the finer side. The solution that still gives the advantages of flexible meshing, was to avoid making that an assembly-unit boundary, and put the assembly-unit boundary inside the domain near it, just a continuity one.