Optimization of the Length Of a Stub Using SonnetLab

By Bashir Souid

This work was supervised by Dr. Serhend Arvas.

In this tutorial we will open a Sonnet project that already has a throughline, then we will generate several projects with different length stubs, we will simulate the designs and determine which length has the lowest return loss at 5 Ghz.

The .m file for this tutorial can be found in

```
<SonnetLab directory>\Tutorials\Stub Optimization\StubOptimization.m
```

The first thing we do in this tutorial is set some parameters that will be used for the optimization. In the following three lines we will set a variable for the minimum length of the stub, the maximum length of the stub and the number of iterations we should attempt. The user may modify these values to be whatever values they like but be aware that if the maximum length value is too large the stub may extend outside of the box.

```
aMaxIterations=10;
aMinStubLength=75;
aMaxStubLength=100;
```

Next we will define two more variables that will aid us in the optimization routine. These values will not need to be changed by the user. These variables simply keep track of which design has the lowest return loss.

```
aCircuitWithBestLoss=0;
aBestLossSoFar=inf;
```

The script then enters the main optimization loop where aMaxIterations number of designs are generated, simulated and analyzed. The optimization loop will open the Sonnet project `'Optimization_Start.son'` which is our starting design.

```
Project=SonnetProject('Optimization_Start.son');
```

The starting design has the desired stackup, a throughline and two ports on either side of the throughline. The optimization routine needs to add a stub, of a particular length, to the throughline. Before we add the stub we need to determine an appropriate length for the stub:

```
aLength=randi([aMinStubLength aMaxStubLength],1);
```

The coordinates of the polygon verticies can be expressed in terms of aLength and the polygon can be added to the project with a call to the `addMetalPolygonEasy()` method.

```
anArrayOfXCoordinates=[70;90;90;70];
anArrayOfYCoordinates=[130;130;130-aLength;130-aLength];
Project.addMetalPolygonEasy(0,anArrayOfXCoordinates,anArrayOfYCoordinates);
```

At this point we have successfully modified our initial design project and need to save the project before simulating. If we call the save() or simulate() methods they will save the project over the original design file (`'Optimization_Start.son'`); this is undesirable because we don't want to modify our initial design file. We can save the project under a new filename by using the saveAs() method. After the saveAs() method is called any subsequent calls to save() or simulate() will use the new filename for the project rather than the original filename. We want a new Sonnet project file for each iteration of our simulation so we will use the following lines to generate a new filename and save the project:

```
aFilename=['Optimization_iteration_' num2str(iCounter) '.son'];
Project.saveAs(aFilename);
```

The above command will save the Sonnet project specified by the variable Project to the hard drive as the file 'Optimization_iteration_#.son'.

Now that we have specified the simulation settings we can call Sonnet's simulation engine to simulate the project using the command:

```
Project.simulate();
```

The easiest way to analyze the results of an ***em*** simulation is to have the project export a touchstone file and read the data using a touchstone reader. This tutorial will use the third party touchstone reader included with SonnetLab; users can use any touchstone reader they like to parse the touchstone data.

```
aSnPFilename=['Optimization_iteration_' num2str(iCounter) '.s2p'];
[F, Data, Zo, DataCell] = TouchstoneRead(aSnPFilename);
```

The S11 data returned by the `TouchstoneRead()` function can be converted into dB using the following line:

```
aLossOfCurrentIteration=20*log10(abs(permute(Data(1,1,:),[3 2 1])));
```

Our optimization goal is to minimize the return loss at 5 Ghz. We will want to compare the return loss of the current iteration of the optimization to the return loss of the best iteration we have encountered so far; this will make it easy to keep track of which circuit design iteration has the lowest return loss. We can accomplish this task with the following lines:

```
if aLossOfCurrentIteration < aBestLossSoFar
    aCircuitWithBestLoss=iCounter;
    aBestLossSoFar=aLossOfCurrentIteration;
end
```

That is the end of our optimization routine. The script will analyze the desired number of circuits and eventually leave the optimization routine with a value for the best circuit design. All that we have to do after that is tell the user which iteration was best and to copy the file from the best iteration and call it '`Optimization_End.son`'.

This concludes the first example of how to optimize a circuit design using Matlab. SonnetLab makes it trivial to open project files, modify project settings, simulate project files and analyze simulation results.