Optimize Coupler Size Using Dimension Parameters and SonnetLab

By Bashir Souid

This work was supervised by Dr. Serhend Arvas.

This tutorial will demonstrate how to add dimension parameters to a Sonnet project and demonstrate how parameters can be used to optimize a layout. In this tutorial we will use a modified version of 2Branch.son from the Sonnet examples (the cell size has been decreased). The script will open the Sonnet project, add dimension parameters, and optimize the layout all from within Matlab. The dimension parameters will modify the coupler such that its size is increased or decreased by up to 10%.

The .m file for this tutorial can be found in

```
<SonnetLab directory>\Tutorials\Coupler Optimization With
Parameters\CouplerParameters.m
```

A new Sonnet project can be generated with the following call. The new project will be the same as a new project created with the Sonnet editor.

```
Project=SonnetProject();
```

Before we can add dimension parameters to the polygons that make up the coupler we must get references (or IDs) for the polygons that should be modified. Users can open the project file with a text editor (like notepad), find the IDs for the desired polygons and use them for the script but this tutorial will instead show the user how to search for polygons using SonnetLab methods.

One of the typical ways of getting a reference to a particular polygon is to search for the polygon using its centroid coordinate pair; a polygon's centroid coordinate is located in the center of the polygon. The X component of the centroid coordinate pair is halfway between the largest and smallest X coordinates of the polygon; the Y component of the centroid coordinate pair is also halfway across the range of polygon Y coordinates.

We can determine which polygon is at the top of the circuit by finding the one with the lowest Y centroid coordinate value (polygon coordinates with low Y values are closer to the top of the box then the bottom because Sonnet uses an inverse Y grid. The origin for the grid is the top left corner of the box). The bottom edge of the coupler will have the highest Y coordinate. The left and right edges of the coupler will have Y centroid coordinates that are about half of the box size. The polygon that makes up the left edge of the coupler has a smaller X centroid coordinate value than the polygon on the right edge of the coupler.

We can get all of the polygons' centroid coordinates along with references to the polygon objects using the following command.

```
[aArrayOfCentroidXCoordinates, aArrayOfCentroidYCoordinates, ~, ~,
aArrayOfPolygons]=Project.getAllPolygonCentroids();
```

We can get a reference to the top polygon by finding the polygon has the smallest centroid Y coordinate value. This can be accomplished by using Matlab's min() function on aArrayOfCentroidYCoordinates to get the index of the desired polygon. Once we know the index we can get a reference to the appropriate polygon in aArrayOfPolygons.

```
[~,aIndexInArray]=min(aArrayOfCentroidYCoordinates);
aTopPolygon=aArrayOfPolygons{aIndexInArray};
```

We can similarly find the bottom polygon by searching for the polygon that has the highest Y centroid coordinate value.

```
[~,aIndexInArray]=max(aArrayOfCentroidYCoordinates);
aBottomPolygon=aArrayOfPolygons{aIndexInArray};
```

Next we will obtain references for the polygons that make up the left and right edges of the coupler. The left and right edges of the coupler have centroid Y coordinate values that are about equal to the box height divided by two. We can find a polygon that has a Y coordinate value closest to the desired value and get an appropriate handle using the following commands:

```
[~,aFirstIndexInArray]=min(abs(aArrayOfCentroidYCoordinates-
Project.yBoxSize/2));
aFirstPolygon=aArrayOfPolygons{aFirstIndexInArray};
```

The second polygon with a Y centroid coordinate near the middle of the box can be found using the following commands. It is important to slice aArrayOfCentroidYCoordinates such that we are only looking at the values after aFirstIndexInArray because we don't want the same polygon to be returned twice.

```
[~,aSecondIndexInArray]=min(abs(aArrayOfCentroidYCoordinates(aIndexInArray
+1:length(aArrayOfCentroidYCoordinates))-Project.yBoxSize/2));
aSecondPolygon=aArrayOfPolygons{aFirstIndexInArray+aSecondIndexInArray};
```

Now that we have two polygons that represent the left and right edges of the coupler we just need to figure out which polygon corresponds to which edge. This can be accomplished by comparing the polygons' X centroid coordinates:

```
if aFirstPolygon.CentroidXCoordinate < aSecondPolygon.CentroidXCoordinate
    aLeftPolygon=aFirstPolygon;
    aRightPolygon=aSecondPolygon;
else
    aLeftPolygon=aSecondPolygon;
    aRightPolygon=aFirstPolygon;
end
```

Now that we have references to the desired polygons we can add dimension parameters that will modify their widths. In this tutorial we will use symmetric dimension parameters; SonnetLab also includes the ability to add anchored dimension parameters. Symmetric dimension parameters require two reference points and two point sets. The value for the dimension parameter is the distance between the reference points. The polygon vertices contained in the first point set move the same distance as the first reference point: if the first reference point moves 5 mils then the coordinates that are part of the first point set will move 5 mils in the same direction. The points in the second point set move along with the second reference point.

addSymmetricDimensionParameter takes the following arguments:

1) The parameter name (Ex: 'Width')
2) Handle for first reference polygon or the polygon's ID
3) The vertex number used for the first reference polygon
4) Handle for second reference polygon or the polygon's ID
5) The vertex number used for the second reference polygon
6) A cell array of any polygons that have points that should be included in the first point set. If there is only one polygon to be altered then this parameter does not need to be a cell array. Polygons in the

first point set are the ones to be altered in the same way as the first reference point.

7) A cell array of vectors that indicate which polygon vertices should be in the first point set. If there is only one polygon to be altered then this parameter does not need to be a cell array.
8) A cell array of any polygons that have points that should be included in the second point set. If there is only one polygon to be altered then this parameter does not need to be a cell array. Polygons in the second point set are the ones to be altered in the same way as the first reference point.
9) A cell array of vectors that indicate which polygon vertices should be in the first point set. If there is only one polygon to be altered then this parameter does not need to be a cell array.
10) The direction of movement; this may be 'x', 'X', or 'XDir' for the X direction and 'y', 'Y', or 'YDir' for the Y direction.

Please execute "help SonnetProject.addSymmetricDimensionParameter" for additional information and examples regarding adding symmetric dimension parameters to a project.

This tutorial will add four dimension parameters to the project. In order to specify the reference points and point sets we need to give the addSymmetricDimensionParameter() method the references to aTopPolygon, aBottomPolygon, aLeftPolygon, and aRightPolygon. We will also need to specify which vertex coordinate to use for each reference point and point set.

The desired vertices for the polygons can easily be selected by using methods like lowerLeftVertex(). Methods such as lowerLeftVertex() are only intended to be used for rectangular polygons and may provide undesired results when used with non-rectangular polygons (Example: what is the lower left corner for a spiral?).

```
Project.addSymmetricDimensionParameter('Width1',...
    aTopPolygon,aTopPolygon.lowerLeftVertex(),...
    aTopPolygon,aTopPolygon.upperLeftVertex(),...
    aTopPolygon,aTopPolygon.lowerRightVertex(),...
    aTopPolygon,aTopPolygon.upperRightVertex(),'Y');
Project.addSymmetricDimensionParameter('Width2',...
    aBottomPolygon,aBottomPolygon.lowerLeftVertex(),...
    aBottomPolygon,aBottomPolygon.upperLeftVertex(),...
    aBottomPolygon,aBottomPolygon.lowerRightVertex(),...
    aBottomPolygon,aBottomPolygon.upperRightVertex(),'Y');
Project.addSymmetricDimensionParameter('Width3',...
    aLeftPolygon,aLeftPolygon.lowerLeftVertex(),...
    aLeftPolygon,aLeftPolygon.upperLeftVertex(),...
    aLeftPolygon,aLeftPolygon.lowerRightVertex(),...
    aLeftPolygon,aLeftPolygon.upperRightVertex(),'X');
Project.addSymmetricDimensionParameter('Width4',...
    aRightPolygon,aRightPolygon.lowerLeftVertex(),...
    aRightPolygon,aRightPolygon.upperLeftVertex(),...
    aRightPolygon,aRightPolygon.lowerRightVertex(),...
    aRightPolygon,aRightPolygon.upperRightVertex(),'X');
```

It is important to output any simulation data in a format that can be read easily by Matlab. The touchstone format is popular and many touchstone readers are available from the Matlab file

exchange; a touchstone reader ships with SonnetLab as a third party script. We can instruct Sonnet to output a touchstone file whenever the project is simulated by using the following command:

```
Project.addTouchstoneOutput();
```

The project can then be saved as using the following command. This will be the file that the circuit design iterations will be based on.

```
Project.saveAs('Optimization_Start.son');
```

We will now define a few variables that will be used for the optimization loop. The first one will be aMaxIterations which defines how many circuit iterations will be generated. The next is aCircuitWithBestLoss which stores the index for the circuit with the best loss. Next we define aBestLossSoFar which will store the best loss we have encountered so far.

```
aMaxIterations=10;
aCircuitWithBestLoss=0;
aBestLossSoFar=inf;
```

At this point the script enters its optimization loop where it will generate aMaxIterations number of circuits that have varying parameter values. To do this we will first open the base project.

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

We will then determine the amount that which each parameter should be modified. We want to increase/decrease the value of each parameter by at most 10%. We will determine the amount of variation by using Matlab's random number generator.

```
aDeltaFactor=rand(1)*.20-.10;
```

Now that we have determined the amount of change we should introduce into this design iteration we can change the parameters' values to reflect the desired change. The value for a dimension parameter can be changed with the modifyVariableValue() method.

```
% Modify Width1 to be its initial value +/- 10%
aCurrentValue=Project.getVariableValue('Width1');
aNewValue=aCurrentValue+aCurrentValue*aDeltaFactor;
Project.modifyVariableValue('Width1',aNewValue);

% Modify Width2 to be its initial value +/- 10%
aCurrentValue=Project.getVariableValue('Width2');
aNewValue=aCurrentValue+aCurrentValue*aDeltaFactor;
Project.modifyVariableValue('Width2',aNewValue);

% Modify Width3 to be its initial value +/- 10%
aCurrentValue=Project.getVariableValue('Width3');
aNewValue=aCurrentValue+aCurrentValue*aDeltaFactor;
Project.modifyVariableValue('Width3',aNewValue);

% Modify Width4 to be its initial value +/- 10%
aCurrentValue=Project.getVariableValue('Width4');
aNewValue=aCurrentValue+aCurrentValue*aDeltaFactor;
Project.modifyVariableValue('Width4',aNewValue);
```

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

initial 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 commands 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 with a file named '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;
    aFilenameOfBestIteration=aFilename;
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.