100 Elwood Davis Road ♦ North Syracuse, NY 13212 ♦ USA

SonnetLab User Guide

Specialists in High-Frequency Electromagnetic Software
(315) 453-3096 Fax: (315) 451-1694 http://www.sonnetsoftware.com

Table of Contents

1) Introduction

This document details the functionalities of the SonnetLab toolbox for Matlab (from here on called SonnetLab). The purpose of SonnetLab is to combine Sonnet's award winning simulation software with MathWork's Matlab scripting environment. SonnetLab can be used in Matlab scripts to automate circuit design and simulation. This document will guide users through the required steps to integrate SonnetLab into the Matlab environment and provide an in-depth discussion of some of the commonly used features of SonnetLab.

SonnetLab is a series of Matlab class definitions that are used to represent Sonnet projects in the Matlab environment. SonnetLab has functions to read Sonnet project files from the hard drive and convert them into Matlab objects. SonnetLab objects contain many variables which specify all of the information that describes a Sonnet project. SonnetLab is also able to build new Sonnet projects from scratch and save them to the hard drive as Sonnet project files. All Sonnet project files created with SonnetLab are compatible with Sonnet Software's circuit tools.

SonnetLab is designed to be flexible enough to allow users to modify any aspect of their Sonnet Projects while being straight-forward and easy to use. The most common use of SonnetLab is to open a large number of Sonnet project files, apply a particular modification to them and simulate them with the Sonnet analysis engine. In later sections of this document users will find directions regarding opening, modifying and simulating Sonnet projects. There are essentially three approaches used to make modifications to Sonnet project objects in Matlab.

The first approach is to modify the variable values in the Sonnet project object; this approach offers a lot of flexibility but may be difficult for some users. By modifying variable values a user can accomplish anything that can be done with Sonnet's circuit tool. Using this low-level functionality may require some understanding of the Sonnet Project File Format. The official Sonnet Project File Format specification document is available from the Sonnet website resource page: http://sonnetsoftware.com/resources/.

Although it is possible to make any desired modifications to the Sonnet project by modifying object

variables there are times when this can be an overwhelming task. SonnetLab includes functions that will perform many common operations. These functions will make some tasks much easier to perform than manually modifying variable values. These functions reduce the extent to which users will need to understand the Sonnet Project file format. Unfortunately, not everything that can be done with Sonnet's circuit tool can be accomplished using the built in functions. It is advisable to use the built in functions whenever possible and to only modify project variables when necessary.

2) Version Notes

This document describes the functionality for SonnetLab version 8.0. SonnetLab version 8.0 is designed for use with Sonnet version 12 and 13. SonnetLab is also designed and tested for use with Matlab versions 7.9.0 (R2009b) and above. No additional Matlab toolboxes are required in order to utilize SonnetLab to its full functionality.

SonnetLab has been tested on Windows XP 32-bit, Windows Vista 32-bit, Windows Vista 64-bit and Windows 7 64-bit. New in SonnetLab version 8.0 is support for Linux versions of Matlab.

3) Installation Notes

This guide assumes you have already downloaded the archive for SonnetLab and have extracted the files to a suitable location. Integrating SonnetLab with your Matlab environment is as simple as adding the scripts to your Matlab path. To add the folder of scripts to your Matlab path do the following:

1. Open up Matlab.
2. Go to File > Set Path. This will launch a new window.
3. Click the button labeled 'Add with Subfolders...'
4. Browse to the folder where you extracted SonnetLab.
5. Double click on the folder for SonnetLab

6. Double click on the Scripts folder
7. Click on the 'OK' button.
8. Click on the 'Save' Button.
9. The path has been set. You may now close the 'Set Path' window by either clicking on the 'Close' Button or clicking on the red X in the corner of the window.

Note: If you ever change the location of where SonnetLab is stored then you have to complete these steps again to add the interface to your Matlab path.

4) Core Functionality

4.1) Open a Sonnet Project File

The SonnetLab toolbox has the ability to open an existing Sonnet project file created with the Sonnet editor. A user can open a Sonnet project file named 'XYZ.son' in the Matlab working directory by invoking the following command:

```
>> aMatlabVariableName=SonnetProject('XYZ.son');
```

The above command will open the Sonnet project file, parse all the data elements and store them all in an object which is referenced by the Matlab variable aMatlabVariableName. It is important to remember that the project object is a handle (type 'help handle' in Matlab for information about handles). For example the following commands may provide undesired behavior for those unfamiliar with object oriented programming:

```
>> aMatlabVariableName=SonnetProject('XYZ.son');
>> aSecondMatlabVariableName=aMatlabVariableName;
```

The new variable that was created (aSecondMatlabVariableName) will be a duplicate reference to the same memory location as the source variable (aMatlabVariableName). Any modifications made to the variable aMatlabVariableName will automatically present the same

modification when examining aSecondMatlabVariableName because both variables point to the same memory location. Both variables can be assigned to independent memory locations by opening the project file once for each unique reference as follows:

```
>> aMatlabVariableName=SonnetProject('XYZ.son');
>> aSecondMatlabVariableName=SonnetProject('XYZ.son');
```

The above commands will create separate references to two different memory locations. Both variables are identical in that they store the same information but they point to separate memory locations; modifying one variable's values will not affect the other variable.

Alternatively users can make copies of Sonnet project objects using the clone() and quickClone() methods. These methods are explained in the section titled “Copy a Sonnet Project File” on page 10.

SonnetLab is mostly intended to be used for Sonnet Geometry projects although there is support for Sonnet Netlist projects as well. A Sonnet netlist project in the current working directory ('Netlist_XYZ.son') can be opened using the same command that is used to open a Sonnet geometry project:

```
>> aMatlabVariableName=SonnetProject('Netlist_XYZ.son');
```

SonnetLab will automatically determine if the supplied project is a Sonnet Geometry project or a Sonnet Netlist project and parse the project file.

4.2) Create a Sonnet Project Object

SonnetLab has the ability to create a new Sonnet project from scratch. This can be accomplished with the following command:

```
>> aMatlabVariableName=SonnetProject();
```

The above command will create an empty Sonnet project with a default set of values. The default values selected by SonnetLab are the same as those selected by Sonnet when clicking 'New Geometry' in the Sonnet toolbar. The default project has a particular box size (160x160 mils), default unit selections and no polygons.

SonnetLab is mostly intended to be used for Sonnet Geometries although there is support for Sonnet Netlist projects as well. When creating a new Sonnet project SonnetLab will create a default geometry project. Any Sonnet project can be converted to a default Sonnet netlist project as follows:

```
>> aMatlabVariableName=SonnetProject();
>> aMatlabVariableName.initializeNetlist();
```

When the user decides to save the new Sonnet project they should call the saveAs method to specify a filename for the new project. For example if we want to create a new Sonnet project and save it as ‘myProject.son’ on the hard drive the following commands will be executed:

```
>> aMatlabVariableName=SonnetProject();
>> aMatlabVariableName.saveAs( ‘myProject.son’ );
```

For more information on saving SonnetProject objects see the section titled “Save a Sonnet Project File” on page 8.

4.3) Save a Sonnet Project File

Once we have a Sonnet project loaded in Matlab's memory space, we can then interact with it in many ways. One of the most common operations is to save SonnetLab objects as Sonnet project files on the hard drive.

SonnetLab has two save functions which are modeled after the typical save system used with desktop software. There is a “save” method and a “saveAs” method. The “saveAs” method

will save a SonnetProject object as a file on the hard drive with a specified filename. If the SonnetProject object was created by opening a Sonnet project file stored on the hard drive or if the "saveAs" method has already been called on the project in the past the user may use the "save" method. The "save" method will save the SonnetProject object to the hard drive using the previously specified filename.

For example a file named 'XYZ.son' can be opened, modified and be re-written to its original filename using the following commands:

```
>> aMatlabVariableName=SonnetProject('XYZ.son');
    <Any modifications to the project>
>> aMatlabVariableName.save();
```

The above example shows how users can use the save() function to save the SonnetProject object to the hard drive using the same filename that was initially used to open the project. This makes it very easy to open a Sonnet project file in Matlab, make some modifications to the project (add polygons, move polygons, etc.) and write the project to the same file.

If the SonnetProject object is created from scratch, rather than by opening an existing project file, then the project will not yet have an associated filename. Users create Sonnet projects from scratch by calling SonnetProject()with no arguments.

If a user attempts to call save() before any filename is specified they will receive and error because the function will not know what filename to use for the Sonnet project file. If a user would like to save a project built from scratch, they will need to call "saveAs" which will save the project to the hard drive using the specified filename.

The "saveAs" method is useful for more than performing the initial save on new SonnetProject objects. There are times when the user may not want to overwrite the original project. Users may want to open a Sonnet project file from the hard drive, make some modifications and then save it to the hard drive under a different filename. This can be accomplished easily by using the "saveAs" method as follows:

```
>> aMatlabVariableName.saveAs('XYZ_1.son');
```

This will save the Sonnet project to a file on the hard drive named 'XYZ_1.son'. The passed string is a path so the project can be written to a file that is a relative path from the Current Working directory.

Once the user executes the “saveAs” method the specified filename will be associated with the project. Any subsequent calls to the “save” method will utilize the newly specified filename. For example if we had run the following commands:

```
>> aMatlabVariableName=SonnetProject('XYZ.son');
>> aMatlabVariableName.saveAs('XYZ_1.son');
```

Any subsequent calls to save() will write the project to 'XYZ_1.son' and not to 'XYZ.son.' The “save” method is essentially a shortcut that allows users to not re-specify the most recently used filename every time they want to save a file.

4.4) Copy a Sonnet Project File

SonnetLab includes two built in methods that may be used to make a deep copy of a project object. Sonnet project objects created using these copy methods will reside in separate memory locations and be completely independent. The only difference between the original project and the copy is that the copy will not have a filename associated with it yet. This decision was made to minimize the likelihood that users will mistakenly overwrite project files.

The new project will need to be saved with the saveAs() command at least once to associate a filename with the project. The method for saving a Sonnet project is explained in the section titled “Save a Sonnet Project File” on page 8.

The below command will make a deep copy of a Sonnet project:

```
>> aProjectCopy= aOriginalProject.clone();
```

Alternatively users may prefer the quickClone() method which usually runs faster for large projects.

```
>> aProjectCopy = aOriginalProject.quickClone();
```

Project files can be copied including simulation data and output files with the SonnetProjectCopy function.

```
>> SonnetProjectCopy( 'aOriginalProjectName.son' , 'aNewProjectName.son' );
```

The above command will copy the project file for aOriginalProjectName.son and save it as aNewProjectName.son along with its simulation data.

4.5) Reset a Project

When an empty project is created, or when the "initialize" method is called on a project, all of the project's blocks get set to default values. One particularly important thing to note is that calling the "initialize" method one a project will reset the project to a default geometry project. If "initialize" is called on a netlist project object the project will be converted to a default geometry project. Alternatively users may call the "initializeGeometry" method which performs the same operation as "initialize".

```
>> aOriginalProject.initialize();
```

To convert the project into a default Sonnet netlist project use the "initializeNetlist" method. The "initializeNetlist" will convert any project, including geometry projects, into a netlist project.

```
>> aOriginalProject.initializeNetlist();
```

4.6) Compare two Sonnet Projects

The Sonnet suite includes a utility that compares Sonnet project files. The comparison utility can be called from Matlab with the SonnetProjectCompare function.

```
>> SonnetProjectCompare('Design1.son','Design2.son')
```

This function will return true if the two project files are identical. The comparison tool may generate some negatives that can be safely ignored. For example if the location of the label for a geometry variable is different between the two files the comparison function will return false even though the two designs are electromagnetically identical.

The comparison routine can also be executed on SonnetProject objects using the compare function.

```
>> isEqual = aProjectObject( 'aProjectFile.son' )
>> isEqual= aProjectObject(anotherProjectObject)
```

4.7) Modify Frequency Sweep Settings

Before a user can run a simulation they must first specify a frequency analysis sweep type along with its frequency parameters. For example Sonnet's adaptive band frequency sweep (ABS) requires start and stop frequency values. Frequency values are stored as doubles and the corresponding frequency unit is stored in the dimension block of the Sonnet project.

A Sonnet project may have many multiple sweeps defined. Although many frequency sweeps may be defined in a Sonnet project only one of them will be used for simulation. The selected frequency sweep is defined in the sweepType variable in the Sonnet project's control block.

If the sweepType variable is “STD” then the project is set to simulate using a “Frequency Sweep Combination.” Frequency sweep combinations include several frequency sweeps rather than a single frequency sweep. The sweeps that make up a frequency combination sweep are different than those that are not part of frequency combination sweeps. For example a user can not add an “ABS” sweep to a frequency sweep combination but they may add an “ABS Entry” sweep. The only difference between an “ABS” sweep and an “ABS Entry” sweep is that “ABS” sweeps may not be used in frequency sweep combinations and “ABS Entry” sweeps can only be used in frequency sweep combinations. The names of sweeps that exist in frequency combination sweeps are listed in the table below:

Sweep Name	Sonnet-Matlab Interface Class Name	Description
SWEEP	SonnetFrequencySweep	Linear frequency sweep
ABS_ENTRY	SonnetFrequencyAbsEntry	Adaptive Band Synthesis Sweep
ABS_FMAX	SonnetFrequencyAbsFmax	Find the maximum frequency response.
ABS_FMIN	SonnetFrequencyAbsFmin	Find the minimum frequency response.
DC_FREQ	SonnetFrequencyDcFreq	Analyze at a DC frequency point.
ESWEEP	SonnetFrequencyEsweep	Exponential frequency sweep.
LSWEEP	SonnetFrequencyLsweep	Linear sweep with a number of points.
STEP	SonnetFrequencyStep	Analyze at a discrete analysis frequency

A frequency sweep can be added to a Sonnet project by using the function calls in the table below. To obtain more information regarding any of these functions simply type “help SonnetProject.<function name>”.

Frequency Sweep Name	Function to add this type of frequency sweep
SonnetFrequencyAbs	AddAbsFrequencySweep (theStartFrequency,theEndFrequency)
SonnetFrequencySimple	AddSimpleFrequencySweep (theStartFrequency,theEndFrequency,theStepFrequency)
SonnetFrequencySweep	AddSweepFrequencySweep (theStartFrequency,theEndFrequency,theStepFrequency)
SonnetFrequencyAbsEntry	AddAbsEntryFrequencySweep (theStartFrequency,theEndFrequency)
SonnetFrequencyAbsFmax	AddAbsFmaxFrequencySweep (theStartFrequency,theEndFrequency,theMaximum)
SonnetFrequencyAbsFmin	AddAbsFminFrequencySweep (theStartFrequency,theEndFrequency,theMinimum)
SonnetFrequencyDcFreq	AddDcFrequencySweep (theMode,theFrequency)
SonnetFrequencyEsweep	AddEsweepFrequencySweep (theStartFrequency,theEndFrequency,theAnalysisFrequencies)
SonnetFrequencyLsweep	AddLsweepFrequencySweep (theStartFrequency,theEndFrequency,theAnalysisFrequencies)
SonnetFrequencyStep	AddStepFrequencySweep (theStepFrequency)

When a frequency sweep is added using one of these functions the selected frequency sweep gets changed to select the new frequency sweep. The selected frequency sweep can also be modified by using the changeSelectedFrequencySweep function. For example the selected frequency sweep can be changed to ABS using the following command:

```
>> aMatlabVariableName.changeSelectedFrequencySweep('ABS');
```

4.8) Calling Sonnet Tools

SonnetLab includes functions that directly interact with the Sonnet suite. These functions provide greater integration between the SonnetLab toolbox and the Sonnet Product Suite.

The SonnetLab tools described in this section are only compatible when run on a Microsoft Windows PC. The tools outlined in this section are not compatible with UNIX based operating systems. Support for such platforms is planned for a future release.

All of the above methods will automatically use the most recently installed version of Sonnet to perform the operation. Each method has an optional argument that will specify a particular version of Sonnet should be used to complete the operation. The version of Sonnet can be manually specified by passing the path to the Sonnet directory. The path may be either the base Sonnet installation directory or the Sonnet bin directory. The following examples will perform the desired operations using Sonnet version 12.52.

```
>> viewResponseData('C:\Program Files\sonnet.12.52')
>> viewCurrents('C:\Program Files\sonnet.12.52')
>> aSonnetProject.openInSonnet(false, 'C:\Program Files\sonnet.12.52');
>> [MegaBytesOfMemory NumberOfSubsections] = Project.estimateMemoryUsage('C:\Program
Files\sonnet.12.52');
```

4.8.1) Simulate a Sonnet Project

Sonnet does all of its simulations through a back-end known as ***em***. SonnetLab can send a project to ***em*** for analysis. This functionality allows the user to simulate a Sonnet Project directly from the command line or from a function. ***Em*** can export analysis results in many popular file formats used for electromagnetic data such as the touchstone format. With the appropriate scripts users can run an electromagnetic simulation, export the response data to a touchstone file (or one of several other file types) and utilize the response data in their Matlab environment by accessing the touchstone file (a touchstone file reader is available in the “Third Party Libraries” folder included with SonnetLab; many similar readers are available on the Matlab Exchange website).

An output file can be added to a Sonnet project by using the addFileOutput function. To get a description of the arguments needed to use the addFileOutput function please see the help information.

SonnetLab makes it easy to simulate Sonnet Projects. Any Sonnet Project can be simulated by calling the simulate() method.

```
>> aSonnetProject=SonnetProject('MainProj.son');
>> aSonnetProject.simulate();
```

When you call simulate the project will be automatically saved as a Sonnet project file on the hard drive. The function will then call Sonnet's built in simulation engine to simulate the project. If the project was created from scratch it will need to first be saved to the hard drive using the saveAs(filename) function before the project can be simulated.

Simulate can take an optional argument which allows the user to send options to the function. Options are passed all together as one string. Order of option switches does not matter and any unknown option switches are ignored. Supported switches are the following.

Option	Description
-c	Cleans the project data before simulating
-x	Does not clean the project data before simulating (default)
-w	Displays a simulation status window (default)
-t	Does not display a simulation status window
-r	Runs the simulation instantaneously (default)
-p	Does not run the simulation instantaneously (requires the status window)
-v	Calls a particular version of Sonnet to do the simulation

<VERSION>	
-s <DIRECTORY>	To manually specify the Sonnet directory to use for the simulation. The directory may either be the base Sonnet directory or the version's bin directory.

SonnetLab will try to find an entry for Sonnet within the Windows registry. If there are no entries for Sonnet in the windows registry then Sonnet must have been improperly installed on the computer. SonnetLab will be unable to simulate a Sonnet project if it cannot locate Sonnet's simulation engine. If a user has multiple versions of Sonnet installed, SonnetLab will select the most recently installed version of Sonnet. The most recently installed version may not necessarily correspond to the latest Sonnet version. For example, if a user were to install Sonnet version 12.52 and then later install Sonnet version 11.52 SonnetLab will select Sonnet version 11.52 to do simulations. The Sonnet version selection can be overridden by using the '-v <version>' switch when running simulate. Users can also use the '-s <directory>' switch to specify a particular directory where the desired version of Sonnet is installed. The '-v' and the '-s' switches are useful for simulating a project with a particular version of Sonnet (many users have more than one version of Sonnet installed).

By default Sonnet simulations are done with the same status window that is used when Sonnet's circuit tools simulate projects. It is also possible to call Sonnet's simulation engine directly which will not launch the status window. Sonnet's simulation status engine is very light weight but if users need to simulate a large number of extremely small projects it may be beneficial to not display the status window when simulating. The '-t' switch allows users to simulate a project without loading Sonnet's built in status window. Whether or not a status window is displayed, the simulate function will return a Boolean to indicate whether the simulation was successful and the function will also return any error messages encountered as a second output argument. This makes it easy for users to halt their scripts if they encounter simulation errors. A user can pass the '-p' switch to the simulate function to indicate that they do not wish to start the simulation engine in the paused state. This may be useful if a user would like to modify Sonnet cluster settings before allowing an ***em*** job to run. The'-p' switch will do nothing when simulating a project without using the status window.

The '-c' switch is used to force the simulator to delete previous simulation data before running the simulation. Cleaning a Sonnet project will force ***em*** to analyze all frequencies during the current simulation operation. Using the '-x' option will allow ***em*** to check the data and only re-simulate if necessary. The '-x' option is the default option and leads to better simulation times when some data points for the project have already been calculated.

The following are a few examples of various ways to call the Simulate function.

```
>> aSonnetProject.simulate()
>> aSonnetProject.simulate('-t')
>> aSonnetProject.simulate('-t -c')
>> aSonnetProject.simulate('-w -x -r')
>> aSonnetProject.simulate('-s C:\Program Files\sonnet.12.56');
```

In the first example the project is written to a file and simulated using the status window. In the second example the project is written to a file and simulated without displaying the status window. In the third example the project is written to a file, cleaned and then simulated without a status window. The fourth example has all default switches and will perform the same operation as the first example where no switches were passed. The fifth example specifies which Sonnet installation should be used to simulate the project.

Alternatively there is a function in SonnetLab that can be used independently on Sonnet project files which exist on the hard drive rather than in memory. This allows a user to simulate a project without loading the project using SonnetLab. This function is called SonnetCallEm()which accepts the same option tags as simulate().

4.8.2) Sonnet Editor

SonnetLab includes a method called openInSonnet() which will save the project and open it in Sonnet. The function takes an optional Boolean argument that is true if Matlab should halt execution until the Sonnet window is closed. If the user omits the argument then the script will assume a value of true. If the argument was omitted or specified to be true then

any saved changes from Sonnet will be transferred to the Matlab version of the project.

Either of the following two commands will save the project, open the project in the Sonnet editor, wait for the editor to be closed, and re-import the project with any changes made by the Sonnet editor.

```
>> aMatlabVariableName.openInSonnet();
>> aMatlabVariableName.openInSonnet(true);
```

The following command will save the Sonnet project, open the project in the Sonnet editor but not wait for the editor to be closed and not import changes to the project.

```
>> aMatlabVariableName.openInSonnet(false);
```

4.8.3) Sonnet Response Viewer

SonnetLab includes a function to open the project's response data in Sonnet's response viewer. The command to open Sonnet's data response viewer is the following:

```
>> aMatlabVariableName.viewResponseData();
```

4.8.4) Sonnet Current Viewer

If a Sonnet project is simulated with current calculations turned on, then the current data can be viewed with the following command:

```
>> aMatlabVariableName.viewCurrents();
```

4.8.5) Sonnet Memory Estimator

SonnetLab can also call Sonnet's memory estimator engine to estimate the required amount of memory to simulate a Sonnet project file. Calling the memory estimator will first save the Sonnet project object. The memory estimator will return two values: the first returned value is the estimated number of megabytes for simulation and the second returned value is the number of subsections. The memory estimator can be called with the following command:

```
>> [MegaBytes, Subsections] = aMatlabVariableName.estimateMemoryUsage();
```

5) General Information and Usage

The following sections describe SonnetLab functionality that is shared between both Sonnet geometry projects and Sonnet netlist projects. This information provides general knowledge regarding how to properly utilize SonnetLab and provides some insight into the inner workings of how Sonnet projects are represented in Matlab.

5.1) Sonnet Blocks

The Sonnet project file format isolates components of projects into segments known as blocks. When a Sonnet project is loaded in Matlab each of the blocks is implemented using a separate class. Each class in SonnetLab is designed to store data for a block or for a component of a block.

Separating the project data into self contained blocks helps organize the data. The differences between geometry projects and netlist projects are expressed by having different block compositions.

For example geometry projects have a geometry block which stores settings for the box, polygons, ports, etc. Netlist projects have a circuit block which stores settings for circuit elements such as resistor elements, transmission line elements, etc.

Some types of blocks are common to both project types. One such example is the dimension block which stores the selected units for length, frequency, etc.

A Sonnet Geometry project will have the following blocks:

HeaderBlock
DimensionBlock
ControlBlock
VariableSweepBlock
OptimizationBlock
FrequencyBlock
GeometryBlock

A Sonnet Netlist project will have the following blocks:

HeaderBlock
DimensionBlock
ControlBlock
VariableSweepBlock
OptimizationBlock
FrequencyBlock
CircuitElementsBlock
ParameterBlock

Each of these blocks holds information that is used to describe the project. Detailed descriptions of what data is stored in each block are available in the project format document available from http://www.SonnetSoftware.com/. A brief description of what information each block stores is provided below:

- **HeaderBlock** - Stores information regarding the program that created the project. The header block also contains information regarding when the project was created and

when it was modified. Users typically do not need to modify header block values.

- **DimensionBlock** - Stores the measurement units that are used for the project.
- **ControlBlock** - Stores some of the unique options of the project including which frequency sweep to run.
- **VariableSweepblock** - Stores the information to be fed into Sonnet's analysis engine. This block is only used when performing an analysis with regard to sweeping variables.
- **Optimizationblock** - Stores the information to be fed into Sonnet's optimization engine. This block is only used when performing an optimization analysis.
- **FrequencyBlock** - Stores the information regarding saved sweep types with their options.
- **Geometryblock** - Stores polygons, dielectric layers, dimension parameters, ports, and other geometric components of the circuit.
- **CircuitElementsBlock** - This class defines the Circuit portion of a SONNET netlist project file. This class is a container for arrays of all the circuit elements contained in the netlist project.
- **ParameterBlock** - This class defines the VAR block for a Sonnet netlist project. It stores all the variables that are used for a Sonnet netlist project.

Other blocks often exist in geometry projects. One such example is the FileOutBlock which holds settings for output files. If the project is set to output analysis results to a file then this block will be present but if there are no selected output files this block may or may not be present.

A project may contain blocks that SonnetLab does not understand. The interface will save each of these blocks as 'Unknown Sonnet Blocks.' Unknown Sonnet Blocks are common because many third party circuit tools create extra blocks that are ignored by Sonnet. SonnetLab will still write out these unknown blocks when the project is saved. All Sonnet blocks are printed out in the same order in which they were read in. This allows SonnetLab to maintain compatibility with other circuit tools that create custom blocks.

5.2) Modify Project Settings

There are many ways to make modifications to project information from within Matlab. The most visual way for users to interact with Sonnet project objects is to use the Matlab Variable Editor. The Matlab Variable Editor can be used to examine the contents of matrices, structures and objects in the Matlab environment.

Many users may find the Variable Editor to be useful when working with SonnetLab projects. In order to examine the contents of a Sonnet project using the Variable Editor, first either create a new Sonnet Project or open an existing one. When a Sonnet project object is created, an entry for it will be made in the Workspace panel in Matlab (if your interface has the workspace panel hidden it can be made visible by clicking on the menu item 'Desktop'>>'Workspace'). Double clicking on the object for your Sonnet Project will present the contents of the object in the Variable Editor. The Sonnet project contains several more objects; each one represents information that is part of the Sonnet Project and whose values can be modified either directly from within the Variable Editor or from the command line. Modifying Sonnet project variable values may require some knowledge of the Sonnet project file format.

Variable values can also be modified using Matlab's command line interpreter. For example, if a user wanted to locate the frequency unit for a Sonnet project all they would need to do is access the frequency unit variable in the dimension block of the Sonnet project. The command to access the selected frequency unit would be the following:

```
>> aProjectObject.DimensionBlock.FrequencyUnit
    ans = GHZ
```

This line accesses the frequency unit for the project referenced by the aProjectObject variable. The class system in Matlab uses the “dot” notation. To access a property of an object one only has to place a dot after the parent object and specify the name of the property. It is important to understand that a property of an object may be another object. In the above example, we saw that in order to reference the frequency unit for the project we had to access the dimension block (DimensionBlock) of the project which is also an object.

The frequency units for the project can be changed from 'GHZ' to 'HZ' in the variable editor by

double clicking the value and changing it to 'HZ'. The value can also be changed using the command window by entering the following:

```
>> aProjectObject.DimensionBlock.FrequencyUnit='HZ';
```

In the above example, we looked at how we could change the frequency unit that is used for our Sonnet project. It is important to only use frequency unit values that are accepted by Sonnet. SonnetLab will allow you to choose invalid unit descriptors but if the project is opened by Sonnet, an error message may be displayed. To find a list of supported units, please see the file format documentation for your particular release of Sonnet.

5.3) Modify Project Settings Using Methods

The previous section demonstrated how to manually modify the variable which stores the frequency dimension unit. Although the presented method is perfectly valid it is non-ideal. There are many cases where manually modifying variable values may be difficult. For example, adding a new metal polygon to a project requires the modification of a large number of variables. Fortunately, SonnetLab contains many functions that facilitate common tasks. One of the advantages of using the built-in functions is that users will not need to have as much knowledge of the Sonnet project file format.

For example, a particular user may not know that the project's selected frequency unit is stored inside the dimension block object of a Sonnet project object. If a user wants to modify that particular variable value manually, they will need to know where that variable is located in the file format structure. SonnetLab designers wanted to simplify common operations so that users could use SonnetLab without knowing the project file structure. SonnetLab contains many functions that facilitate common operations.

For example, a user may change a project's frequency units to 'HZ' by using the changeFrequencyUnit() function as follows:

```
>> aMatlabVariableName.changeFrequencyUnit('HZ');
```

This function will modify the same variable that was changed manually in the previous section; both methods of changing the frequency unit will produce the same result. In many cases using the built-in functions will be simpler than manually modifying variable values.

Although SonnetLab includes a large set of helping methods there are some operations that don't currently have helping methods. Any suggestions for new methods may be submitted to the author. The designers of SonnetLab are always looking for ways to simplify the usage of the interface including adding additional functions and making existing functions easier to use.

5.4) Copy a Project Element

All SonnetLab classes have a "clone" method which will perform a deep copy of a SonnetLab component. The copy of the object will have all the same property values as the original object but will reside in a separate memory location which means that modifications made to either object will not affect the other object.

Copying an object is very useful for backing up elements of a Sonnet project. A user could make a copy of a polygon, modify the original polygon, simulate the project, analyze the simulation results, and restore the backed up polygon object.

The following example will get a reference to the first polygon in a project, make a copy of the polygon, delete the original polygon, and insert the copied polygon into the project.

```
% Open a project titled 'DemoOriginal.son'; you may use any geometry project file
Project=SonnetProject('DemoOriginal.son');

% Get a reference to the first polygon in the project.
aPolygon=Project.getPolygon(1);

% Make a copy of the polygon
```

```
aPolygonCopy=aPolygon.clone();

% Delete the original polygon
Project.deletePolygon(aPolygon);

% Insert the copied polygon into the project
Project.addPolygon(aPolygonCopy);

% Save the project
Project.saveAs('DemoModified.son');
```

The result of the above operation should produce a file called ‘DemoModified.son’ which is identical to ‘DemoOriginal.son’.

It is worthwhile to note that although this functionality is available for all SonnetLab classes there are shortcuts available for copying polygon objects. More information about copying polygon objects can be found in the section titled “Copying Polygons” on page 37.

The “clone” method is available for all SonnetLab classes including SonnetProject. Calling the “clone” method on an entire Sonnet project has a few exceptions that should be taken into account. For more information regarding copying entire Sonnet projects see the section titled “Copy a Sonnet Project File” on page 10.

5.5) Reset a Project Element

The section titled “Reset a Project” explained how to reset an entire SonnetProject object to default settings. Similar functionality is available for all SonnetLab objects. This functionality is particularly useful when applied to individual blocks of a SonnetProject.

Users can also call the initialize method for any block to reset its data back to the default. For example a user can easily clear the geometry block for a project by doing the following:

```
>> aGeometryProject.GeometryBlock.initialize();
```

Resetting the geometry block will delete all the polygons in the project. Resetting the geometry block does not affect other blocks in the project. Calling initialize on one particular block can be an easy way to reset part of a project to default settings rather than creating a new project from scratch.

5.6) Delete a Project Element

Although it is undesirable to remove Sonnet project blocks it is appropriate to remove elements such as polygons, ports, bricks, etc. These elements are stored in cell arrays in the project. Removing an element from a Sonnet project is as simply as removing the element from its array.

The cell arrays of objects are organized by type. Elements present in geometry projects (such as polygons, ports, components, etc.) can be found in the "GeometryBlock" object. More information regarding the buildup of a Sonnet geometry project can be found in the section titled "Sonnet Blocks".

Elements present in netlist projects (such as data file elements, network elements, etc.) can be found in the "CircuitBlock" object. More information regarding the buildup of a Sonnet netlist project can be found in the section titled "Sonnet Blocks".

The following example shows how to remove a polygon object from a geometry project by removing it from the array of polygons. All polygon objects are stored in a cell array called "ArrayOfPolygons". For the sake of simplicity this example will delete the first polygon in the array of polygons.

```
>> Project.GeometryBlock.ArrayOfPolygons(1)=[];
```

SonnetLab includes many methods that simplify the removal of such objects. The section titled "Delete Polygons" presents several functions that greatly simplify the process of deleting a polygon from a project. The above method of removing polygons may appeal to users who

wish to manually remove/modify elements in unique ways.

6) SonnetLab with Geometry Projects

The following sections describe SonnetLab functionality that is specifically designed for Sonnet geometry projects. Techniques described in this section are not applicable towards Sonnet netlist projects.

As described in the section titled “Open a Sonnet Project File” SonnetLab can open project files with the below command. SonnetLab will automatically detect if the specified project file is a geometry project file.

```
>> aMatlabVariableName=SonnetProject( ‘XYZ.son’ );
```

As described in the section titled “Create a Sonnet Project Object” SonnetLab can create a new Sonnet geometry project from scratch with the below command. By default all newly created Sonnet projects will be Sonnet geometry projects.

```
>> aMatlabVariableName=SonnetProject();
```

6.1) Polygons

There are essentially three ways to differentiate one polygon object from another; every polygon has a unique handle, ID and index. Each polygon in a project is an instance of a class; each instance of a class has a unique handle. The handle for an object can be used when calling functions which are presented in the following paragraph. All the polygons in a project are stored in a cell array at <ProjectVariableName>.GeometryBlock.ArrayOfPolygons. Many methods will allow the user to pass a polygon's cell array index in order to specify a particular polygon. Every polygon has a unique debug ID value (sometimes simply called its ID value). This ID value can be sent to methods in order to specify a particular polygon. The benefit of

keeping track of a particular polygon's ID rather than its index is that its index may change if polygons are added or removed from the cell array of polygons; a polygon's ID (and its handle) will not change when the project adds or removes polygons.

SonnetLab includes many functions that interact with geometry polygons. Methods that interact with polygons typically come in three flavors. One flavor of the method will use the polygon's handle, another will use a polygon ID and the third will use the polygon's index. For example there is a method called flipPolygonX(Polygon) which will accept either a polygon reference or the polygon's ID. The method flipPolygonXUsingId(ID) will also perform the same operation given the polygon's ID or a polygon reference. There is a method called flipPolygonXUsingIndex(Index) which will accept either the polygon's index in the array of polygons or a reference to the polygon itself. This is the general pattern for all the methods that deal with polygons.

6.1.1) Adding Polygons to Geometry Projects

Polygons are one of the major components of Sonnet geometry projects. Polygons may include planar metal polygons, dielectric bricks and rectangular/circular via's. Polygons are stored in the GeometryBlock as a cell array called ArrayOfPolygons.

There are several ways to create a new polygon. The low-level way would be to make a new empty polygon by constructing it without any parameters and manually changing all its relevant values.

```
>> aPolygonObject=SonnetGeometryPolygon();
>> aPolygonObject.MetalizationLevelIndex=0;
>> aPolygonObject.XMinimumSubsectionSize=0;
Etc.
```

After all the properties of the polygon have been modified the polygon would still need to be added to the cell array of polygons. Using this approach to add a new polygon to a project can be overwhelming. It is easier to use the addMetalPolygon function instead.

There is also a function to add dielectric bricks called addDielectricBrick and a method to add vias called addViaPolygon.

```
>> aSonnetProject.addMetalPolygon (theMetalizationLevelIndex, theMetalType, theFillType,
theXMinimumSubsectionSize, theYMinimumSubsectionSize, theXMaximumSubsectionSize,
theYMaximumSubsectionSize, theMaximumLengthForTheConformalMeshSubsection, theEdgeMesh,
theXCoordinateValues, theYCoordinateValues)
```

The above command will add a metal polygon to a Sonnet geometry project. The user must specify a value for all of the above fields. The metal type field specifies the material from which the polygon is constructed. The user may specify '0' for lossless or the index for the requested material in the array of metal types. The user can alternatively specify the name of the metal type. The requested metal type must be defined for the project before it can be used; type 'help SonnetProject.defineNewMetalType' for information on adding metal types to a Sonnet project. The following example adds a gold metal polygon to layer zero of the project.

```
>> x=[5,10,10,5,5];
>> y=[10,10,20,20,10];
>> aSonnetProject.addMetalPolygon(0,'Gold','N',0,0,50,100,0,'Y',x,y);
```

A new blank geometry project will have one metallization level which is level zero. The polygon's (X, Y) coordinate values are expected to be in a matrix. In the above example, the vertices of the polygon are (5, 10), (10, 10), (10, 20), (5, 20) and (5, 10). The Sonnet file format requires that the last point of a polygon be the same as the first point in order to close the polygon; SonnetLab will apply this requirement automatically when using polygon creation functions. The user can specify the (X, Y) points in either a clockwise or counterclockwise direction.

It is unusual for a user to want to specify some of the fields for adding a polygon. Many users may prefer to make a polygon using default settings. SonnetLab contains a method called addMetalPolygonEasy which takes three arguments for the polygon and uses default values for the rest. Similarly there is an addDielectricBrickEasy and an addViaPolygonEasy function as well which simplify adding new dielectric bricks and vias to the project.

For example a two by two square metal polygon on level zero can be placed at the origin easily with the addMetalPolygonEasy function as follows:

```
>> aSonnetProject.addMetalPolygonEasy(0, [0,0,2,2], [0,2,2,0])
```

The above command will also accept an optional argument that specifies the metal type for the new polygon. The value may be either the index for the desired metal type in the array of metal types or the name of the metal type. The user may add a lossless polygon by omitting the metal type argument, passing a zero as the metal type index or passing 'Lossless'. The metal type must be defined before it can be used for polygons (Lossless metal is implicitly defined). In the below example the user adds a gold polygon to the project.

```
>> aSonnetProject.addMetalPolygonEasy(0, [0,0,2,2], [0,2,2,0],'gold')
```

6.1.2) Searching for Polygons

Sonnet geometry projects store their polygons in a cell array. Large projects may have thousands of polygons which may make it difficult to locate a particular polygon. The naive approach to finding a particular polygon in the array of polygons is to loop through every polygon in the array searching for a particular polygon; this can be a difficult and lengthy process. The designers of SonnetLab felt that it would be beneficial to include several functions to facilitate searching for polygons. The following methods can be useful for obtaining references to polygons when the physical location of the polygon is known.

Every polygon in a Sonnet project is assigned a (X, Y) coordinate pair for its centroid and it’s mean. The X component for the centroid is calculated by taking the largest X value for the polygon and subtracting it from the smallest X coordinate and dividing by two. This is also done for the Y coordinates to obtain the Y component for the centroid.

Every polygon has a (X, Y) coordinate pair that corresponds to the polygon's mean point.

The X component for the mean point is an average of all the X coordinate values; similarly the Y component is an average of all the Y coordinate values.

If the centroid location or the mean location of a polygon is known then it is easy to search for the polygon. The proceeding examples will all make use of the polygons whose coordinates are shown below:

```
>> aSonnetProject=SonnetProject('MainProj.son');
>> aSonnetProject.GeometryBlock.ArrayOfPolygons{1}.XCoordinateValues
        ans = [0]    [0]    [11.4000]    [11.4000]    [0]
>> aSonnetProject.GeometryBlock.ArrayOfPolygons{1}.YCoordinateValues
        ans = [22.60]  [25.60]  [25.60]  [22.60]    [22.60]
```

The above commands printed out the values for the X and Y coordinates for the first polygon in the project. It is similarly simple to display the components of the centroid and the mean point.

```
>> aSonnetProject.GeometryBlock.ArrayOfPolygons{1}.CentroidXCoordinate
        ans =5.7000
>> aSonnetProject.GeometryBlock.ArrayOfPolygons{1}.CentroidYCoordinate
        ans =24.1000
>> aSonnetProject.GeometryBlock.ArrayOfPolygons{1}.MeanXCoordinate
        ans =5.7000
>> aSonnetProject.GeometryBlock.ArrayOfPolygons{1}.MeanYCoordinate
        ans =46.7000
```

These are the centroid and mean component values for the first polygon in our project. If a user knew the centroid location of a polygon but did not know where it was in the array of polygons they could still easily obtain a reference to the polygon by using the function findPolygonUsingCentroidX as follows:

```
>> aSonnetProject.findPolygonUsingCentroidX(5.7);
```

This function will locate any and all polygons that have a centroid X coordinate of 5.7. The function will return a handle for each polygon, the DebugId for each polygon and the index

for each polygon in the array of polygons.

The DebugId for a polygon is a unique number that represents a particular polygon. Knowing a polygon's DebugId allows you to locate it easily even when the polygon is moved or re-sized. Similarly there are functions to search for polygons given the centroid Y coordinate value or both the X and Y coordinate values. All of the same functions exist for use with the mean of the polygon.

When working with a project interactively it may be beneficial to simply use the displayPolygons() function to print out all the properties (including centroid and mean points) of the project's polygons to the command window in an easy to read table. Although this can helpful when using SonnetLab interactively, it is not as useful when used in a function.

There are cases when knowing a polygon's location may not be enough to find the polygon easily in a script. For example it may be difficult to obtain a reference to a particular polygon if a project has lots of polygons at the same location on different metallization levels. Another example is if the project had concentric circular via's with the same centroid but are different sizes. The find functions allow us to specify the layer that we would like to search in and they allow us to specify the size of the polygon.

Indicating a layer to search in can help reduce the number of returned results from a polygon search. To do this you include another argument which selects a layer. We can use Matlab's built in length function to see how many polygons are returned in our examples when searching on layer zero and searching on layer one.

```
>> length(aSonnetProject.findPolygonUsingCentroidX(5.7,0))
        ans =1
>> length(aSonnetProject.findPolygonUsingCentroidX(5.7,1))
        ans = 0
```

The above commands indicate that on level zero there was one polygon that had a centroid X coordinate of 5.7; there were no polygons on level 1 which had a centroid X coordinate of

5.7.

The user can also specify the size of the polygon being searched for. The size of a polygon is simply the sum of the polygon's total X range and total Y range. The size of the polygon used in the previous example was 14.4. If we search for polygons of that size we will locate the polygon; if we search for polygons of size 14.5 we will not locate it or any polygons because no polygons in this project are 14.5 units in size.

```
>> length(aSonnetProject.findPolygonUsingCentroidX(5.7,0,14.4))
        ans =1
>> length(aSonnetProject.findPolygonUsingCentroidX(5.7,0,14.5))
        ans = 0
```

Alternatively a polygon can be found using the findPolygonUsingPoint function which takes an X and Y coordinate as inputs and returns polygons that encapsulate that point. The findPolygonUsingPoint function optionally takes an integer as an argument to specify the metallization level index. The command below will return all polygons that encompass the point (0, 0).

```
>> aSonnetProject.findPolygonUsingPoint(0,0)
```

SonnetLab also includes a find function which allows users to specify what polygons should be returned. The findPolygonUsingFunction function takes a function pointer as an argument and uses the user specified function to determine if a polygon should be returned or not.

The function that is passed to findPolygonUsingPoint should accept a polygon as an argument (polygons are instances of the SonnetGeometryPolygon class) and return a Boolean. The Boolean should be true if the user would like the polygon to be included in the solution set and false otherwise.

The findPolygonUsingFunction method gives users the ability to perform custom searches flexibly and effortlessly. The following example will find all polygons in a project that have

an X centroid value that is greater than 50. The following function will appropriately check if a polygon meets the desired condition:

```
function result=greaterThan50(aPolygon)
  if aPolygon.CentroidXCoordinate > 50
    result=true;
  else
    result=false;
  end
end
```

The following command will search the project's polygon array for any polygons with an X centroid value greater than 50 by using the defined function.

```
>> aSonnetProject.findPolygonUsingFunction(@greaterThan50)
```

6.1.3) Moving Polygons

Another group of functions deal with moving a polygon from one location to another. The two most commonly used functions for moving polygons are movePolygon and movePolygonRelative. The function movePolygon will change the polygon's X and Y coordinates such that the polygon's centroid is at a particular X and Y value. For example we can call the movePolygon function on the polygon from our previous example as follows:

```
>> aSonnetProject=SonnetProject('MainProj4.son');
>> aPolygon = aSonnetProject.GeometryBlock.ArrayOfPolygons{1};
>> SonnetProject.movePolygon(aPolygon,0,0);
```

This will move the first polygon in our array of polygons such that its centroid is at the grid location (0, 0). It is important to understand that Sonnet uses an inverted Y axis for its grid so the point (0, 0) is actually the top left corner of the Sonnet window.

The move functions require the polygon object to be passed to the function along with the new centroid coordinates. The function will move the polygon such that the centroid is at the specified location while maintaining the polygon's shape and size. Alternatively the polygon's DebugId can be passed to the move functions instead of the polygon object. In the ongoing example the DebugId for the polygon is 12 so the polygon can be moved using the following function call:

```
>> SonnetProject.movePolygon(12,0,0);
```

Because SonnetLab has polygon search functions (such as findPolygonUsingCentroidX) that return both references to polygons and polygon DebugId's it is simple to use either the polygon itself or the DebugId.

```
>> aPolygon = aSonnetProject.findPolygonUsingCentroidX(1,1);
>> SonnetProject.movePolygon(aPolygon,0,0);
```

This will move the polygon that was at centroid position (1, 1) to a position such that its centroid is at location (0, 0). In this particular case, this move constitutes decreasing the X and Y coordinates by one.

The function movePolygonRelative takes the same arguments as movePolygon but rather than moving the polygon such that the centroid is at the passed location it instead moves the polygon by a specified distance. For example the same one cell move can be performed using the following:

```
>> aPolygon = aSonnetProject.findPolygonUsingCentroidX(1,1);
>> SonnetProject.movePolygonRelative(aPolygon,-1,-1);
```

This will decrease all the X coordinates by one and decrease all the Y coordinates by one which will move the polygon located at position (1,1) (if there is a polygon located at position (1,1)) to location (0,0).

The polygon object itself has support for the move function. A particular polygon can be moved by using the following command:

```
>> aPolygon = aSonnetProject.findPolygonUsingCentroidX(1,1);
>> aPolygon.movePolygonRelative(-1,-1);
```

Calling a move function on a polygon directly requires only the coordinate values; the function does not take a reference to the polygon (itself) or the polygon's DebugId (its own DebugId) as a parameter.

6.1.4) Copying Polygons

It is common to want to want to make a copy of a polygon object. Polygon objects, like all SonnetLab objects, are Matlab handles and cannot be copied properly using the equals sign.

An example of the improper way to copy a polygon is presented in the following code block. In this code block we will open a SonnetProject file named “XYZ.son”, get a reference to the first polygon in the project and copy that polygon using the equal (“=”) sign operator.

```
>> Project = SonnetProject( “XYZ.son” );
>> aFirstPolygon = Project.getPolygon(1);
>> aSecondPolygon = aFirstPolygon;       % Wrong
```

The above commands will actually have aSecondPolygon point to the same memory location as aFirstPolygon. This means that if the data contained in either variable is changed it will modify the data of the other variable. A unique copy of a polygon can be made in the same way as any other SonnetLab object can be copied: the clone method. For example an acceptable way to copy a polygon object is the following:

```
>> aSecondPolygon = aFirstPolygon.clone(); % Correct
```

For more information on the clone() method see the section titled “Copy a Project Element”.

Although the above method will appropriately copy the polygon object SonnetLab includes several methods that may simplify the process of copying polygons.

There SonnetLab includes two additional approaches to copying a polygon. One approach is with the “copyPolygon” method and the other is with the “duplicatePolygon” method. There are variations on both methods which allow users to specify that they would like to perform the operation by specifying the desired polygon in a different way than its ID or reference (ex: copyPolygonUsingIndex).

The only difference between the “copyPolygon” method and the “duplicatePolygon” method is that the “copyPolygon” will only make a copy of the specified polygon and return a reference to it; the “duplicatePolygon” method will make a copy of the specified polygon, add it to the project and return a reference to the polygon. The “duplicatePolygon” method is a shortcut to add the newly created polygon to the same project.

Both methods will accept either the polygon’s ID or a reference to the polygon. The following code block will open the Sonnet project file “XYZ.son”, make a copy of the polygon with an ID value of 12, add the polygon to the newly copied file to the project, and save the project.

```
>> Project = SonnetProject( “XYZ.son” );
>> Project.duplicatePolygon(12);
>> Project.save();
```

6.1.5) Change Polygon Type

The default metal type for newly created metal polygons in the Sonnet editor or SonnetLab is lossless metal. It is often desirable to change the metal type of such a polygon. Similarly it is desirable for there to be a means in SonnetLab to change the composition of dielectric

bricks and via polygons.

Before a polygon can be changed to another composition type the type must be defined in the Sonnet project. For example if the user would like to change a metal polygon such that it is comprised of copper metal then the project must have a copper metal type defined. The copper metal type can't be automatically defined by SonnetLab because information such as the metal thickness cannot be assumed (execute "help SonnetProject.defineNewMetalType" for information on defining such types). Similarly dielectric brick materials must be defined before they can be used.

The format for via materials has changed between Sonnet version 12 and Sonnet version 13. In Sonnet version 12 via polygons were comprised of the same materials as planar metal polygons. In Sonnet 13 via polygons are comprised of new via metal types. The most recent version of SonnetLab supports both Sonnet version 12 and Sonnet version 13 project files. Users will not be able to change via polygons contained in Sonnet version 13 projects to planar metal types. Users will not be able to change via polygons in Sonnet version 12 projects to the new via metal types because via metal types can't be defined in Sonnet version 12 files.

The semantics for changing a composition of a polygon is very simple. Planar metal polygons, dielectric bricks, and via polygons can all be modified with the same method: "changePolygonType". The "changePolygonType" method accepts two parameters: the first parameter is either a reference to the desired polygon or the polygon's ID, the second parameter is the name of the desired material. If the polygon is a metal polygon then the material name should be the name of a defined metal type (if the metal type is not defined an error will be thrown). If the polygon is a dielectric brick then material name should be the name of a defined dielectric brick material type, etc.

The following example will convert the polygon with an ID of 12 to a defined planar metal type called "ThinCopper". If there is no polygon with an ID of 12 or if the "ThinCopper" material type is not defined then an error will be thrown.

```
>> Project.changePolygonType(12,'ThinCopper');
```

SonnetLab also includes a method called “changePolygonTypeUsingIndex” which instead of accepting a polygon ID will accept an index in the polygon array. Both methods will accept a polygon reference.

The following example will convert the 5th polygon in a project to “ThinCopper”.

```
>> Project.changePolygonTypeUsingIndex(5,'ThinCopper');
```

6.1.6) Delete Polygons

SonnetLab includes functions that allow users to remove polygons from Sonnet Geometry projects. Polygons can be removed from Sonnet projects by removing them from the array of polygons as described in the section titled “Delete a ” on page 27. Because the removal of polygons is such a commonly used operation SonnetLab includes a few operations that simplify their removal.

The “deletePolygon” method can be used to delete planar metal polygons, dielectric brick polygons and via polygons. If the polygon has any connected ports, edge vias, or parameters then they will be deleted as well.

The “deletePolygon” method will accept either a polygon ID or a polygon reference. The following example will delete a polygon with an ID of 12.

```
>> Project.deletePolygon(12);
```

SonnetLab also includes a method called “deletePolygonUsingIndex” which instead of accepting a polygon ID will accept an index in the polygon array. Both methods will accept a polygon reference. The following example will delete the 5th polygon in a project.

```
>> Project.deletePolygonUsingIndex(5);
```

6.2) Ports

Ports are directly connected to planar metal polygons or via polygons. SonnetLab includes several methods that add, remove and search for ports. Sonnet supports three types of ports: standard, autogrounded and co-calibrated.

Typically ports are managed using either their port number, their index in the array of ports, or a reference to the port. A brief explanation of each is provided below:

- The port number is the number that is displayed over the port in the Sonnet editor.
- All the ports in a project are stored in a cell array (Project.GeometryBlock.ArrayOfPorts); users can use the port's index in the cell array as a means of specifying a particular port. Using a port index is useful when trying to reference a recently added port; a newly created port is added to the end of the array of ports. If a user wants to modify the most recently added port they can access it as the last element in the array of ports.
- A port reference is the port object itself. If a user already has the port object that they would like to utilize they may send that port object directly to SonnetLab methods.

An important thing to note is that if a polygon is deleted then any connected ports are automatically deleted.

6.2.1) Adding Ports

SonnetLab includes functions that allow users to add ports to Sonnet Geometry projects. Sonnet has support for three types of ports: standard ports, auto-grounded ports and co-calibrated ports (for a detailed description of how Sonnet models each type of port please see Sonnet's documentation).

SonnetLab provides methods that give users fine tuned control over their port properties and methods that make it simple to add ports without having to specify values for every property.

The user guide will primarily focus on standard ports. Similar functions are available for adding auto-grounded and co-calibrated ports to projects and can be found in the method reference document that ships with SonnetLab. The function to add a standard port to a project is “addPortStandard”. The example command will add a port to the second edge of the polygon with an ID of 11. The port has a resistance of 50 ohms with zero inductance, capacitance, or reactance.

```
>> Project.addPortStandard(11,1,50,0,0,0);
```

The polygon edge number is relative to the polygon's coordinate vertices. In the above example we added a port to the second edge of the polygon. The second edge of the polygon is the edge that spans the polygon's second and third coordinate pairs. So for example if a polygon has X coordinates of [34, 227, 227, 34, 34] and Y coordinates of [105, 105, 75, 75, 105] then edge two is between the points (227,105) and (227, 75). Since the port is attached to the middle of the second edge of the polygon the port is located at (227, 90).

The “addPortStandard” method may be overcomplicated for many users. SonnetLab includes a function called “addPortToPolygon” which only needs two arguments: the DebugId of the attached polygon and the vertex number that the port should be attached to. We can add a port to the second edge of the polygon represented by the polygon ID number 11 with the following command:

```
>> Project.addPortToPolygon(11,1);
```

There is a third, simpler, way to add the desired port to the project. SonnetLab has a method called “addPortAtLocation” which allows users to add a port to a project without knowing the ID of the polygon or the polygon edge number. The “addPortAtLocation” only requires

an (X, Y) coordinate pair which specifies the approximate location where the port should be added. The function will search the list of polygons for a polygon with the closest edge to the passed point. If the desired port location is too far away from any polygon edges then an error will be thrown.

```
>> Project.addPortAtLocation (5, 10);
```

The add port methods listed above return a reference to the newly created port. This object reference can be passed to other methods or manually modified. The following example lines of code will use the "addPortAtLocation" method to add a port, get a reference to the port, and then use that reference to modify the port's resistance value:

```
>> aPort = Project.addPortAtLocation(5, 10);
>> aPort.Resistance = 75;
```

6.2.2) Deleting a Port

SonnetLab includes functions that allow users to remove ports from Sonnet Geometry projects. Ports can be removed from Sonnet projects by removing them from the array of ports as described in the section titled "Delete a Project Element" on page 27. Because the removal of ports is such a commonly performed operation SonnetLab includes a few operations that simplify the removal of ports.

The "deletePorts" method can be used to delete ports. The "deletePorts" method provides a means for deleting a port from a project using the associated port number. In the following example port number two can be removed from the project by executing:

```
>> Project.deletePort(2);
```

SonnetLab also includes a method called "deletePortUsingIndex" which instead of accepting a port number will accept an index in the port array. The following example will

delete the 3rd port in a project.

```
>> Project.deletePortUsingIndex(3);
```

6.2.3) Searching for a Port

SonnetLab contains several methods that provide the ability to search for ports in a project. All of the below methods will return the same information: a reference to the located port(s), the port number(s), and the port index(s).

The "findPort" method accepts a port number. This method is useful if you know a port's number but would like to know its index in the array of ports or have a reference to the port object. The following example will search for a port with a port number of three.

```
>> [PortReference, PortNumber, Index] = Project.findPort(3);
```

The "findPortUsingPoint" method allows users to search for a port based on its approximate location in the Sonnet box. The "findPortUsingPoint" method accepts an X and Y coordinate that should be close to the desired port. An important thing to keep in mind is that Sonnet uses an inverse Y grid. Low Y values are at the top of the Sonnet box and high Y values are at the bottom of the Sonnet box. The following example will search for ports near the location (0,120):

```
>> [PortReference, PortNumber, Index] = Project.findPortUsingPoint(0,120);
```

The "findPortUsingPoint" method will optionally accept an integer that will restrict results to a particular metallization level. The following example restricts the previous results to metallization level zero:

```
>> [PortReference, PortNumber, Index] = Project.findPortUsingPoint(0,120,0);
```

The “findPortsInGroup” method allows users to search for ports that are present in a particular co-calibrated port group. The “findPortsInGroup” method accepts a group name as an argument. The following example will return all ports that are in co-calibrated port group ‘A’.

```
>> [PortReference, PortNumber, Index] = Project.findPortsInGroup( ‘A’ );
```

6.3) Components

SonnetLab supports resistor, capacitor, inductor and data file (S-parameter) components. All components in a Sonnet geometry project are stored in a cell array called “ArrayOfComponents” within the GeometryBlock. Users may interact with components by directly accessing the desired component found within the array of components. SonnetLab includes several methods that simplify the operation of creating, searching for, and deleting components. Such methods are demonstrated in the following sections.

There are essentially three ways to differentiate one component object from another; every component has a unique handle, ID and index. A brief description of each is provided below:

- Each component in a project is an instance of a class; each instance of a class has a unique handle. The handle for an object can be used when calling functions which are presented in the following subsections.
- All the components in a project are stored in a cell array at <ProjectVariableName>.GeometryBlock.ArrayOfComponents. Many methods will allow the user to pass a component's cell array index in order to specify a particular component.
- Every component has a unique debug ID value (sometimes simply called its ID value). This ID value can be sent to methods in order to specify a particular component. The benefit of keeping track of a particular component's ID rather than its index is that its index may change if components are added or removed from the cell array of components; a component's ID (and its handle) will not change when the project adds or removes components.

SonnetLab includes many functions that interact with components. Methods that interact with components typically come in three flavors. One flavor of the method will use the component's handle, another will use a component’s ID and the third will use the component's index.

For example there is a method called deleteComponent(Component) which will accept either a component reference or the component 's ID. The method deleteComponentUsingId(ID) will also perform the same operation given the component's ID or a component reference. There is a method called deleteComponentUsingIndex(Index) which will accept either the component 's index in the array of components or a reference to the component itself. This is the general pattern for all the methods that deal with components.

6.3.1) Add Components

Components can be added to SonnetProjects using a set of methods. The following table indicates which methods can be used to add particular types of components to a project.

Component Type	SonnetProject Method
Resistor	addResistorComponent
Capacitor	addCapacitorComponent
Inductor	addInductorComponent
Data File	addDataFileComponent

For example the “addResistorComponent” method will add an ideal resistor component to a Sonnet geometry project. The above methods will return a reference to the newly created component.

All four types of component add methods accept the same arguments. The add component methods require at least four arguments and may optionally accept a fifth.

1) The component name
2) The component' value
3) Level number
4) A 2x2 matrix of the component port locations.
    The first row should be the first port's X value, then its Y value
    The second row should be the second port's X value, then its Y value
5) (Optional) The terminal width. This value should be either
    - "Feed" to use the feedline width (Default)
    - "Cell" for one cell width
    - A number which represents a custom width

The first argument, the component name, may be any name that the user chooses. The second argument is the component's value. For an ideal resistor component the component value should be the desired resistance value, for capacitor components this value should be the capacitance value, for inductor components this value should be the inductance value and for data file components this value should be the filename for the data file. The units for the resistance/capacitance/inductance value are specified by the project's global unit selection (default is for resistance is ohms). The third argument is the metallization level on which the component should be placed. The fourth argument specifies the location for the components ports which ultimately sets the location for the component itself.

If the user does not specify a value for the fifth argument, the terminal width, then the component's terminal width will be the feedline width. The user can explicitly specify that they would like to use the feedline width by passing the fifth argument as 'Feed'. The fifth argument can alternatively be 'Cell' to indicate that the terminal width should be the size of one cell on the Sonnet grid. The user may also pass any number if they would like to specify a particular value for the terminal width.

6.3.2) Delete Components

SonnetLab includes functions that allow users to remove components from Sonnet Geometry projects. Components can be removed from Sonnet projects by removing them from the array of components as described in the section titled "Delete Components" on page 47. Because the removal of components is such a commonly used operation

SonnetLab includes a few operations that simplify the removal of components.

The “deleteComponent” method can be used to delete components. The “deleteComponent” method provides a means for deleting a component from a project using the associated components name, the component’s ID, or a reference to the component. For example in the following example the component named ‘R1’ will be removed from the project by executing:

```
>> Project. deleteComponent ( ‘R1’ );
```

SonnetLab also includes a method called “deleteComponentUsingIndex” which instead of accepting a component ID will accept an index in the components array. The user may alternatively still use a component reference or the name of a component instead of an index. The following example will delete the 3rd component in a project.

```
>> Project. deleteComponentUsingIndex (3);
```

6.3.3) Searching for Components

SonnetLab contains several methods that can be used to search for a particular component in a SonnetProject.

SonnetLab contains methods that will search the array of components and only return components that are of a particular type (resistor, capacitor, inductor, data file). For example the “getResistorComponents” method will return a vector of all of the resistor components in a project.

```
>> aArrayOfResistorComponents = Project.getResistorComponents( );
```

Similarly SonnetLab has the “getCapacitorComponents” method, the “getInductorComponents” method, and the “getDataFileComponents” method.

The "getComponent" method will return a component in the project based on its index in the array of components. For example the following line will return the 5th component in a SonnetProject.

```
>> Component=Project.getComponent(5);
```

The "findComponent" method will search for a component in the array of components based on its name, or ID. The following example will return the index and a reference to the component named 'R1'.

```
>> [Index, Polygon]=Project.findComponent( 'R1' )
```

7) SonnetLab with Netlist Projects

The following sections describe SonnetLab functionality that is specifically designed for Sonnet netlist projects. Techniques described in this section are not applicable towards Sonnet geometry projects.

7.1) Adding Circuit Elements

Sonnet Netlist projects may have the following netlist elements: resistor elements, capacitor elements, inductor elements, transmission line elements, physical transmission line elements, data response elements, project file elements and network elements.

Each of these elements can be added to a Sonnet Netlist project with one of the functions in the chart below. To see more information regarding how to use each individual function, please see their corresponding help string (Ex: help SonnetProject.addResistorElement)

Element Type	Function Name

Resistor Element	`addResistorElement`
Capacitor Element	`addCapacitorElement`
Inductor Element	`addInductorElement`
Transmission Line Element	`addTransmissionLineElement`
Physical Transmission Line Element	`addPhysicalTransmissionLineElement`
Data Response Element	`addDataResponseFileElement`
Project File Element	`addProjectFileElement`
Network Element	`addNetworkElement`

If a user would like to add a resistor element to a Sonnet Netlist project they can execute the following command:

```
>> aNetlistProject.addResistorElement(1,2,50);
```

This will add a resistor element to the first network in the project. The resistor will be connected between ports one and two of the network and have a resistance of 50 (the units for resistance are specified in the dimension block of the Sonnet project; default is ohms).

If the circuit contains multiple networks the user can specify the network for the resistor by including an additional argument:

```
>> aNetlistProject.addResistorElement(1,2,50,2);
```

This will add a resistor element to the second network in the project between nodes one and two. The number for the network is the network element's index in the array of networks (aMatlabVariableName.CircuitBlock.ArrayOfNetworkElements). Alternatively the user can specify the name of the network rather than the network's index.

```
>> aNetlistProject.addResistorElement(1,2,50,'NetworkName');
```

This will add the resistor element to the network named 'NetworkName'. If the project does not contain a network named 'NetworkName' an error will be thrown.

7.2) Modify Network Ports

The "setNetworkPorts" method will allow users to modify the port numbers for a network. The method can be used to modify the ports for a particular network of a Sonnet project. The following example will modify the ports of the second network in a Sonnet netlist project to be [1, 2, 3, 4, 5].

```
>> aNetlistProject.setNetworkPorts(2, [1 2 3 4 5]);
```

The user may omit the network index argument if they would like to modify the last network in the project. The last network in a netlist project is the external network. The following example will modify the ports of the last network in the project to be the numbers one through ten.

```
>> aNetlistProject.setNetworkPorts(1:10);
```

7.3) Netlist Variables

Sonnet supports variables in netlist projects. Variables allow users to modify the value of multiple circuit elements simultaneously. The value of a circuit component can be assigned to a netlist variable such that its value will be consistent with other netlist elements. The following line of code will modify the resistance value of a particular resistor element to be equal to the variable "Res".

```
>> aResistorElement.ResistanceValue = 'Res';
```

Before a variable can be used it should be defined and assigned a value. The following

subsections will explain how to declare and modify netlist variables.

7.3.1) Define a Netlist Variable

Netlist variables can be declared using the “defineVariable” method. The “defineVariable” method requires two arguments: the first argument is the desired variable name; the second argument is the variable value. The following example will declare a variable called “Z0” which has an initial value of 50.

```
>> aNetlistProject.defineVariable('Z0',50);
```

7.3.2) Get a Netlist Variable Value

The value of a netlist variable can be read using the “getVariableValue” method. The “getVariableValue” method requires the name of the desired variable. If the specified variable name has not been previously defined then an empty matrix ([]) will be returned.

```
>> aNetlistProject.getVariableValue ('Z0')
        ans = 50
```

7.3.3) Modify Netlist Variable Value

After a netlist variable has been declared its value can modified using the “modifyVariableValue” method. This method expects to receive a valid name for a variable defined in the project. If the specified variable name is invalid or nonexistent an error will be thrown. The following example will modify the value of the variable “Z0” to the value 55.

```
>> aNetlistProject.modifyVariableValue('Z0',55);
```

8) Tips / Tricks / Pitfalls

1. SonnetLab objects may exist independently from projects

   Users can construct SonnetLab objects that are completely independent from any Sonnet project objects. This can be helpful if a user wants to manually build an object (for example a polygon object) and insert it into many Sonnet projects. This technique may also be useful for backing up an aspect of a Sonnet project. An example of constructing a polygon that is independent from any projects is the following command:

   ```
   >> aPolygon=SonnetGeometryPolygon( );
   ```

   Objects constructed independently from Sonnet projects will have default settings; this often means that all the properties will be empty matrices. The user may need to manually assign values to properties.

2. SonnetLab objects may be shared between projects

   One of the strengths of the interface's handle nature is that blocks and other objects can be shared between SonnetLab projects. For example the fileout block of a Sonnet project can be shared with another project by executing the following

   ```
   >> aFirstProject.FileOutBlock=aSecondProject.FileOutBlock;
   ```

   The above command will cause both projects to share the same fileout block. If any changes are made to one project's fileout block then the same modification is made to the other project.

   One potential use of this functionality would be to generate many projects that have unique layouts but share the same box stackup. The layouts can be simulated with different stackups by changing the stackup of any one of the projects. This may be useful for

comparing identical circuits that have different layouts.

This same functionality can be used to make projects share polygons, circuit elements, etc. This functionality can be very useful for optimization problems where many projects may be interconnected.

3. Be careful when cloning polygons or building them manually

The clone() method is very convenient for making an exact copy of a polygon (or any other SonnetLab object) but one thing to be careful about is that polygons are expected to have a unique debug ID value. If two polygons in a project have identical debug ID values the project will still be read properly by the Sonnet Suite but multiple identical ID's may cause confusion when using SonnetLab tools. This issue is partially avoided because SonnetLab will generate a random ID value for the new polygon. It is unlikely that this random ID value will match any existing ID values but it is possible that it will; because polygon objects can be constructed independently of project objects and polygons may be shared between projects it is impossible to generate a unique ID without knowing which project the polygon will be added to.

SonnetLab has a method called generateUniqueId() that will find a unique debug ID that the user may assign to the polygon. See the following example:

```
>> aPolygon.DebugId=Project.generateUniqueId();
```

Alternatively, users can use the following method:

```
>> Project.assignUniqueDebugId(aPolygon);
```

The above commands allow users to avoid issues with multiple polygons having the same ID. Users should utilize one of the above methods whenever a new polygon is incorporated into a project.

The above methods are not needed when creating polygons using SonnetLab's add/duplicate polygon methods. The add/duplicate methods will automatically obtain unique ID values for newly created polygons.

4. Class methods are the best way to perform an operation with some exceptions.

   The designers of SonnetLab dedicated a lot of work to create a large number of class methods to help users perform common operations on Sonnet projects. Although these methods can greatly simplify many tasks, there occasionally may be situations where it is faster to do an operation manually by editing object properties.

   One such example is if you wanted to add ten thousand polygons to a Sonnet project. SonnetLab's "addMetalPolygon" method must choose a unique ID for each polygon. Although the routine to find a unique ID has been designed to run as fast as possible the operation may be computationally cheaper if the user manually specifies an ID. A user may be able to add such polygons to a project faster if they manually set the ID themselves using a counter or an algorithm.

   Perhaps the fastest way to add ten thousand polygons to a project would be to give all the existing polygons in the project sequential ID's (this can be accomplished using the "assignAllPolygonsSequencialIds" method). The user should then reallocate the array to be a larger size so that all of the entire array will not have to be re-sized while performing the operation. Each newly created polygon could then be constructed manually and its ID value can be the polygon's index in the polygon array.

   SonnetLab allows you to take any approach you like for circuit design; including manually modifying the project. Most users will not need to optimize their code to perform operations faster than any included routines but the functionality to do so is available.

5. SonnetLab will automatically delete inconsistent project elements.

Polygons may have other objects associated with them including ports, edge vias and dimensional parameters. This has many benefits; if a polygon is moved then any ports, edge vias, and dimensional parameters associated with it also move to the new location. This can raise issues since it is necessary to delete any associated items when a polygon is deleted. SonnetLab will automatically delete ports, edge vias and dimensional parameters associated with a deleted polygon. This functionality mimics what the Sonnet project editor does when a polygon is deleted.

Although most users will not want to keep ports etc. that were attached to deleted polygons SonnetLab does provide a means to disable the automatic deletion of these objects. Sonnet-Matlab interface projects contain a property called "AutoDelete" which when set to true will automatically delete inconsistent project elements. "AutoDelete" is set to true by default; to disable automatic deletion set the property to false.

9) Contact

Your feedback is important to us. If you have any questions or comments about SonnetLab, please contact Sonnet Support by email at support@sonnetsoftware.com.

Please make sure you are using the most up to date version of SonnetLab before submitting a bug report. When submitting a bug report please include the Sonnet project file that generated the error (Sonnet project files have the extension .son) and the output from the command "SonnetMatlabVersion". The more information that that we receive the faster it will be for us to resolve the issue and contact you back.