Appendix B

from

Silicon Microwire Photovoltaics

Thesis by
Michael David Kelzenberg

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

2010

(Defended May 19, 2010)

Note

This appendix presents a method for importing optical generation profiles (or other arbitrary profiles) for use with the TCAD Sentaurus software suite. It is intended to assist others with the application of this technique. Introductory material and simulations performed using this technique are presented and discussed within the main chapters of the thesis.

Contents

Appendix B

Method of integrating *Lumerical FDTD* within *Sentaurus TCAD*

Sentaurus Device, part of the Synopsys *Sentaurus TCAD* software suite, is a robust semiconductor device physics simulation tool that is well-suited for simulating the performance of three-dimensional solar cell geometries such as Si microwire photovoltaics. It includes models for most of the device physics phenomena relevant to photovoltaics, which combined with its use of modern numerical methods, make it one of the most capable device-physics-based solar cell simulators available today. It can simulate arbitrary semiconductor geometries in two or three dimensions using a finite-element mesh (grid). This permits simulation of novel, nonplanar solar cell geometries, but can introduce several challenges in defining the device structure for simulations. In particular, for a typical Si solar cell, the following spatially varying quantities must be specified throughout the simulation volume:

- Optical generation rate
- Impurity concentration (e.g., dopants or traps)
- Carrier lifetime

These profiles can be easily specified or calculated for one-dimensional structures (e.g., Gaussian emitter doping profiles or exponential Beer's-law optical absorption profiles), but can become more complicated for arbitrary three-dimensional structures. For these reasons, the TCAD software includes numerous capabilities to specify analytical profiles, to simulate processing steps (e.g., diffusion doping), and to calculate optical absorption (e.g., ray-tracing, FDTD) in 3D structures. However, in certain situations we have found it useful to manually specify these profiles, based on external calculations, assumptions, or simulations. This appendix presents a method that has been developed to map arbitrary external profiles onto the numerical mesh used for Sentaurus Device simulations.

B.1 Introduction

In this thesis work, techniques were developed to import optical generation profiles calculated by FDTD simulations (*Lumerical FDTD Solutions*)* into device physics simulations (Sentaurus Device). This enabled comprehensive optoelectronic modeling of Si microwire-array solar cells, as presented in Chapter 2.

*Our choice of Lumerical FDTD software over the Sentaurus (internal) FDTD methods simply because of the expertise base for the former within our group.

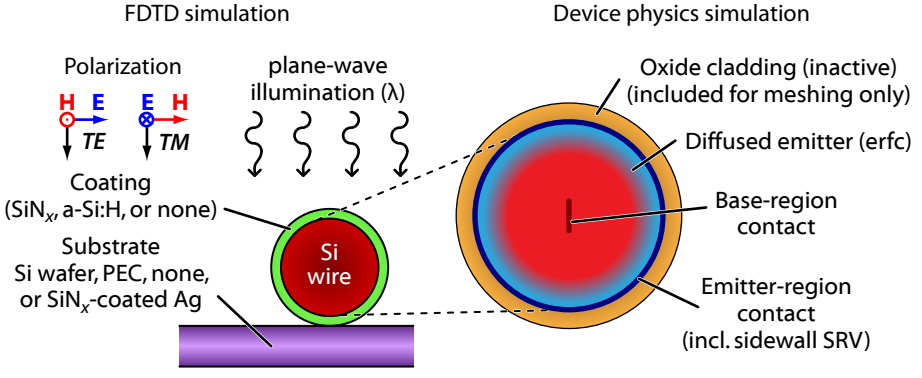


Figure B.1. Schematic diagram of simulation geometries for modeling optical (left) and electrical (right) behavior of horizontally oriented single-microwire Si solar cells.

This process was implemented within the *Sentaurus Workbench* (SWB) environment, extending its tool database to include the Lumerical CAD/FDTD programs as well as *MATLAB* scripts. MATLAB provides a convenient programming environment well-suited to the task of processing and storing the photogeneration profiles for device-physics simulations. Integrating these programs within SWB enabled us to automate the entire simulation process (structure generation, FDTD simulations, device physics simulations, and variable extraction), making use of the software's preprocessor to generalize the configuration of each simulation step. This allowed us to employ SWB's automated design features (such as parametric sweeps and numerical optimization), and also to take advantage of its project interface for generating, executing, and keeping track of a large number of simulations and parameters. For example, the FDTD simulations presented in Section 5.4.1 were implemented as a project within SWB, and can be run from start to finish with virtually no user interaction.

In this appendix, we describe the major steps of this simulation approach, and show how it has been integrated into the SWB environment. Source code is included for each key step, including a MATLAB script that generates mesh files containing arbitrary input profiles for Sentaurus simulations. To illustrate the use of these techniques, we describe a project that simulates the spectral response and solar J–V behavior of a single-wire solar cell structure like those fabricated in Chapter 5. The general structure of the simulated device is shown in Figure B.1. Both the optical (FDTD) and electrical (device physics) simulations are performed in two-dimensional coordinates. The particular details of the FDTD and device physics simulations are not discussed here; they closely follow those presented elsewhere in this thesis. Our discussion focuses instead on the procedures and tools required to import the optical generation profiles into Sentaurus Device utilizing the automation features of the SWB platform.

B.1.1 Requirements

This work made use of the following software under an x86_64 Linux environment:

- TCAD Sentaurus C-2009.06-SP2
- MATLAB R2009b
- Lumerical FDTD/CAD v6.5.5

The computational requirements vary depending on the size and complexity of the simulated structure. The two-dimensional structures discussed in this appendix can be easily simulated on modern personal computers having a few GB of RAM. The larger three-dimensional structures (i.e., the Si microwire-array solar cells presented in Chapter 2) were simulated on individual workstations in our lab, the most powerful of which were SunFire x2270 servers (Sun Microsystems) having 48 GB RAM (1300 MHz DDR3) and dual 64-bit processors (Intel Xeon 5500-series, 3 GHz).

Use of these techniques requires a moderate understanding of the TCAD software suite, including Sentaurus Device, the mesh generation tools, and the SWB preprocessor and project environment. These instructions also assume familiarity with Linux environments, MATLAB, and the Lumerical FDTD software.

B.2 Understanding the mesh files

Sentaurus operates on a finite element mesh, which consists of vertices, edges, and elements that store the discretized state of a device during simulations (e.g., doping, carrier concentration, or electric field). Information describing the layout of the mesh is known as the *grid*, which defines the physical position (coordinates) of each mesh element. Information describing the physical state of the device is known as the *data*, which provides the numerical value of each simulated quantity at each mesh element. Combined, the grid and data specify the device structure for Sentaurus simulation.

There are two file formats for storing grid and data information. The first and default type is "TDR"—a binary file format which combines both the grid and data information into a single `.tdr` file. This results in a smaller file size and eliminates any confusion about which grid belongs with which data. However it has proven difficult to read or write to these binary files without knowledge of their format. Our scripts do not presently support `.tdr` files. The other file format,"DF-ISE", is a text-based format which can be deciphered and written by relatively simple parsing scripts. A DF-ISE mesh consists of a grid (`.grd`) file and one or more data (`.dat`) files. For typical simulations, there is a single `.dat` file corresponding the `.grd` file, which contains data values for all simulation input profiles. However, the simulation input profiles can be divided amongst

multiple `.dat` files, each of which corresponds to the same `.grd` file but contains different physical quantities. For example, in the project highlighted herein, a Sentaurus mesh generation program generates the original `.grd` file as well as the `.dat` file describing the device doping profiles, while our MATLAB script generates a separate `.dat` file describing the optical generation profile. DF-ISE and TDR files can be merged, separated, and converted using the *Sentaurus Data Explorer* (TDX) program as described in the TCAD users manuals.

When using DF-ISE files, Sentaurus Device requires a grid file and a data file to specify the device structure (i.e., dimensions and doping profiles). Optionally, additional data files can be used to specify optical generation, SRH lifetime, or possibly other physical quantities (see the *File* section of the Sentaurus Device manual). Each `.dat` file must correspond to the same `.grd` file. During the simulation, Sentaurus will also save one or more new `.dat` files (depending on the command file settings), specifying the state of the mesh quantities (e.g., electric field) within the evolving simulation. A corresponding `.grd` file does not need to be saved (except, possibly, if mesh is modified by adaptive meshing which I have not explored).

Simulation meshes can be viewed using the program *Tecplot SV*. Tecplot can directly load and display TDR files since they contain both the grid and data components of the mesh. To view DF-ISE files in Tecplot, however, both a `.grd` and one (or more) `.dat` files must be specified. To simplify this, the primary `.dat` file is typically always given the same base filename as the `.grd` file. For a combined Lumerical/Sentaurus simulation, the typical set of mesh files for each experiment will include:

`n1_pof.grd`	the grid generated by the mesh generation tool
`n1_pof.dat`	the corresponding device data profiles (doping, etc.)
`n2_optgen.dat`	a data file generated by our MATLAB script (specifying the optical generation profile)
`n3_des.dat`	a data file automatically produced after the final iteration of Sentaurus Device, containing all physical quantities requested within the *Plot* section of the command file
`n3_ISC_des.dat`	a data file recorded at 0 V bias, produced by a instruction within the *Solve* section of the command file
`n3_nearVOC_des.dat`	a data file recorded when the simulated device current crosses 0 A (i.e., near V_{oc}), also produced by an instruction within the *Solve* section of the command file

In the above file names, the numbers *1*, *2*, and *3* would correspond to the node numbers (in SWB) of the mesh generation step, our MATLAB script step, and the Sentaurus Device simulation step, respectively. To visualize these mesh

files—for example, that of the MATLAB step, the following command can be used to invoke Tecplot SV at the shell prompt:*

```
tecplot_sv nA_pof.grd nB_optgen.dat &
```

B.2.1 Generating custom data files

The grid file contains all the information one needs to generate a new data file containing a user-specified (external) profile. The grid file maps each mesh element to a physical point in space (x,y), and it is usually straightforward to calculate the value of the external profile $P(x,y)$ for each mesh element. For our earlier work (PVSC 2009), I wrote a program that parsed each `.grd` file, constructing a map of the mesh in memory; then calculated the profile values for each mesh element and generated a properly formatted `.dat` file from scratch.

More recently, however, I have found an easier approach in which we instruct the Sentaurus mesh generation program to store the spatial coordinates of each mesh element within the initial `.dat` file (which normally only contains the doping data). This saves us the trouble of parsing the `.grd` file and building the mesh in memory. Reading the `.dat` file provides both the spatial coordinates (x,y) of each mesh element, as well as the correct order in which to write the desired profile values $P(x,y)$ in the new `.dat` file. This conceptually simpler approach is presented here.

B.3 Step 1: Generating the mesh

Structure generation for Sentaurus TCAD tools is typically accomplished using *Sentaurus Structure Editor* (also referred to as SDE). This provides a scriptable, graphical environment for specifying material shapes, doping, meshing parameters, etc. However, the mesh itself is not generated by SDE, rather, it is produced by a command-line meshing program that is invoked by SDE. When the mesh is requested, SDE converts its model into a command file for the meshing program, runs the program, and then displays the resulting mesh in its GUI. Although this generally simplifies the process of mesh generation, it is important to understand the behavior of the underlying mesh program and the format of the mesh command file prepared by SDE. Most importantly, to instruct the mesh program to store the x- and y- coordinates of each mesh element within the data file, the following commands must be appended to the mesh command file:

```
Definitions {
    AnalyticalProfile "XPosition" {
        Species = "PMIUserField0"
        Function = General(init="", function = "x", value = 10)
    }
    AnalyticalProfile "YPosition" {
```

*The `-mesa` rendering option is also required for compatibility with some remote X11 clients, such as *Cygwin-X* or *Xming*, which we often use for remote access from Windows-based machines.

```
        Species = "PMIUserField1"
        Function = General(init="", function = "y", value = 10)
    }
}

Placements {
    AnalyticalProfile "XPosition" {
        Reference = "XPosition"
        EvaluateWindow {
            Element = material ["Silicon"]
        }
    }
    AnalyticalProfile "YPosition" {
        Reference = "YPosition"
        EvaluateWindow {
            Element = material ["Silicon"]
        }
    }
}
```

These commands instruct the mesh program to store the value of the functions x and y as `PMIUserField0` and `PMIUserField1` in the mesh data file. The species names "`PMUuserField`*N*" must be used (rather than "`XPosition`" or "`ArbitraryName`") because they are valid *DATEX* fields. (See Sentaurus user's manuals for a list of valid DATEX fields.) Note that the above syntax could also be used to specify arbitrary analytical profiles, such as doping (`Species = "BoronActiveConcentration"`) or even optical absorption (`Species = "OpticalGeneration"`).

To issue the above commands to the meshing program, they must be saved as a text file within the project directory (for example, `mk_store_xy.cmd`) and then appended to the meshing command file prepared by SDE, using the following command syntax:

```
(sdedr:append-cmd-file "mk_store_xy.cmd")
```

This command should immediately precede the command to invoke the meshing program, `(sde:build-mesh)`, within the SDE command file.

B.3.1 Meshing strategy

Obtaining an optimal mesh is generally the most time-consuming step for a new Sentaurus project. TCAD Sentaurus includes three different mesh generators: *mesh*, *snmesh*, and *noffset3d*. Each offers its own benefits and drawbacks, and users should consult the manuals to understand and select the appropriate mesh tool for the desired structure. My basic understanding of the mesh options, based on my limited experience with 2D simulations, is as follows:

Mesh: Produces axis-aligned (rectangular) meshes with minimal triangulation. As the most basic meshing program, it offers fewer options for "intuitive" grid refinement, and usually requires a great deal of manual refinement instructions (i.e., multiboxes) to produce a suitable grid. Most of the simulations presented in this thesis were meshed using `mesh` with numerous multiboxes placed over the entire device extent. `Mesh` also supports offset mesh generation using the `-noffset` option, which I have not explored.

SNMesh: A more advanced axis-aligned mesher, `snmesh` offers more convenient refinement options (e.g., specifying finer meshing at surfaces or region boundaries), and tends to produce more smoothly varying grids with greater overall connectivity. **However, `snmesh` cannot produce DF-ISE meshes,** and is thus not directly compatible with our profile conversion scripts at this point. It may be possible to convert its `.tdr` meshes to DF-ISE format, but this has not been investigated.

NOffset3D: Produces meshes by "offsetting" material surfaces and boundaries based on specified grid densities and offset distances (i.e., an "onion peel" approach). After offsetting boundaries, it fills the remaining areas/volumes with triangular or tetrahedral meshes. It can produce very efficient grids for non-axis-aligned or 3D structures, but in my experience with the single-wire radial-junction structure shown in Figure B.1, has suffered from perplexing and erratic behavior that has requires fine-tuning of its input parameters as well as careful scrutiny of each grid it produces. I suspect that this is due to the presence of a closed circular boundary in the structure, which seems to causes gaps or overlaps in the grid of the concentric offset layers. I have nonetheless been able to produce a variety of efficient simulation grids for modeling these devices, and the method has worked reliably for other structures.

Figure B.2 is a screenshot illustrating the meshes produced by each of the three mesh generation programs (for the same radial p-n junction geometry). The oxide region (red) surrounding the wire provides a boundary at which grid refinement parameters can be specified (for `snmesh` and `noffset3d` only), but does not serve any other purpose in our simulations. `SNMesh` and `noffset3d` both produced suitable grids, whereas the `mesh`-generated grid would likely require further refinement near the oxide/Si interface. For this structure, `noffset3d` was ultimately favored for its efficient handling of the curved p-n junction interface and compatibility with DF-ISE-format mesh files. Within the project, SDE was employed to specify the device structure and invoke `noffset3d` to generate the mesh. However, `noffset3d` required that several additional offsetting parameters be specified in its command file (which were added using `append-cmd-file` in SDE), in order to generate a suitable mesh. Furthermore, following a careful inspection of the resulting grid, several refinement boxes were added (manually) to patch inadequately meshed areas that appeared to be caused by mismatches between the circumference of the innermost vs. outermost offset layers.

B.4 Step 2: Simulating optical generation profiles using *Lumerical FDTD*

Lumerical calculates the steady-state electromagnetic field phasor vectors $\vec{E}$ and $\vec{H}$ throughout the simulation volume. Assuming that all absorption is due to band-to-band absorption within the semiconductor material, the optical generation rate is determined by the energy loss per unit volume, or divergence of

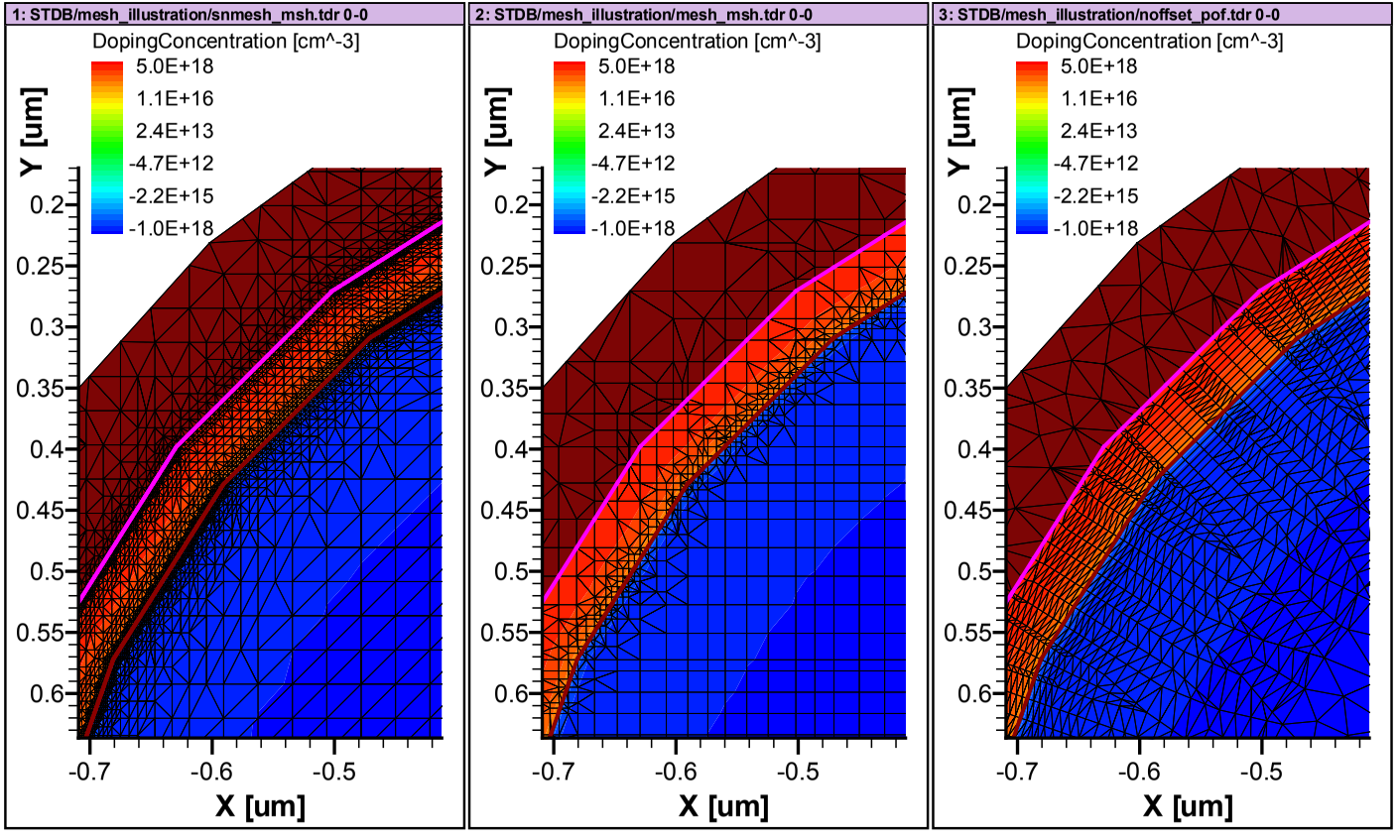


Figure B.2. Screenshot of simulation meshes produced by each tool. Left: `SNMesh`. Center: `Mesh`. Right: `NOffset3d`. Each mesh was generated for the same single-wire structure. The screenshot shows an area near the edge of the wire (viewed in Tecplot).

the Poynting vector. The equations work out such that G_{opt} can be determined directly from the electric field magnitude $\left|\vec{E}\right|$ and the imaginary part of the material's permittivity, ϵ'', as:

$$G_{\text{opt}} = \frac{\Re\left\{\nabla \cdot \vec{P}\right\}}{2E_{\text{ph}}} = \frac{\pi\epsilon''\left|\vec{E}\right|^2}{h} \tag{B.1}$$

These calculations can be performed within Lumerical, making use of its MATLAB-like programming environment and built-in function library. First, however, each simulation must be configured to record the quantities ϵ'' and $\left|\vec{E}\right|$. This is accomplished by a adding an index monitor (`"indexMonitor"` in our example code) and a frequency-domain power monitor (`"fldMonitor"`) throughout the simulation volume, and configuring them to record the appropriate quantities (e.g., E_x, E_y, E_z, etc.) After the simulation has been run, the following commands can be used to calculate G_{opt} throughout the simulation volume (note that this example is for two-dimensional simulations):

```
load("simulation_filename.fsp");

freq = getdata("fldMonitor", "f");
x = getdata("fldMonitor", "x");
y = getdata("fldMonitor", "y");

E2 = getelectric("fldMonitor");
n = getdata("indexMonitor", "index_x");

omega = 2 * pi * freq;
epsilon = eps0 * n^2;
Pabs = 0.5 * omega * E2 * imag(epsilon);
Ngen = Pabs * 1e-6/(6.626e-34 * freq); # cm^-3 s^-1
Current = 1.61e-19 * integrate(Ngen, 1:2, x, y);

matlabsave("output_filename.mat", x, y, Pabs, Ngen);
write("Absorbed photocurrent: " + num2str(Current) + " A per um");
```

The above code also uses the `integrate()` function to calculate the overall absorbed photocurrent of the structure (i.e., the maximally obtainable short circuit current of a solar cell). The resulting photogeneration profile variables are written to a MATLAB `.mat` file for further processing in the next step.

In addition to the direct calculation of photogeneration profiles (G_{opt}), it is also useful to determine the overall absorption of the structure using appropriately placed* field monitors. The total absorbed photocurrent calculated by volume integration above (`Current`) should always concur with the absorption calculated based on how much energy passes through the monitors surrounding the device (the latter calculation is aided by the `transmission()` function in Lumerical).

*By *appropriately placed*, we mean that the monitors should form a closed (Gaussian) surface around the absorption volume.

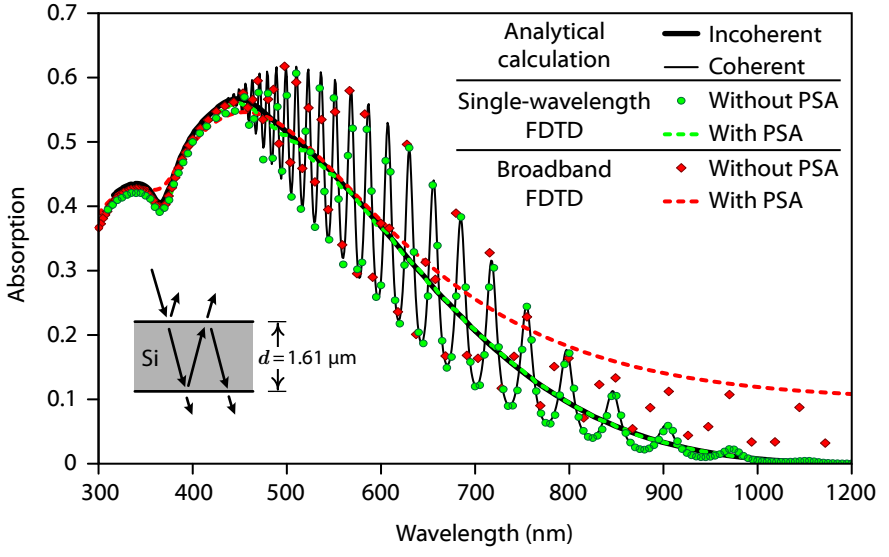


Figure B.3. Effect of partial spectral averaging (PSA) on Lumerical FDTD simulations. The absorption of a 1.61 µm thick planar Si slab (in air) is plotted, as calculated by single-wavelength (green) and broadband (red) FDTD simulations, with and without PSA using Δf values calculated by equation B.2 ($k = 1$). All simulations used Lumerical's most-dense automatic grid setting. The broadband simulation represents the best material fit we were able to obtain for Si, using up to 18 fit parameters and trying a variety of tolerance and bandwidth settings for the fit. The black lines plot the absorption calculated by analytical means, with and without interference considerations.

B.4.1 Partial spectral averaging

Thin, partially transparent structures can exhibit large fluctuations in their reflection and transmission as the illumination wavelength is varied, due to interference (the phenomena responsible for Newton's rings). Similar effects are also observed in FDTD, and can result in apparently "noisy" absorption calculations as the wavelength is varied (unless the simulation wavelengths are very closely spaced). To circumvent this problem, Lumerical offers *partial spectral averaging* (PSA), which averages simulation results over a small frequency range (via Lorentzian weighting) surrounding the principal simulation frequency. Given a device structure of thickness d, refractive index n, and a simulation wavelength of λ (and frequency f), the spectral half-width (Δf) of k interference fringes is given by the equation:

$$\Delta f = \frac{f}{1 + \frac{k d_{\mathrm{Si}} n}{\lambda}} \qquad [\mathrm{Hz}] \tag{B.2}$$

I have found that using the value of Δf corresponding to $k = 1$ gives good results when using Lumerical's partial spectral averaging. This concept is illustrated in Figure B.3, in which FDTD simulations have been performed to calculate

the absorption of a simple planar Si slab of thickness $d = 1.61$ μm. The results of the FDTD simulations can be compared to the analytical solution for the absorption of this structure, calculated as described in Section 3.1.3. The absorption calculated by FDTD simulations with vs. without PSA nearly exactly agrees with the analytical solutions for incoherent vs. coherent absorption, respectively. The figure also illustrates the difficulties we encountered attempting to perform broadband simulations of Si structures in Lumerical. For these reasons, all FDTD modeling presented in this thesis was performed using single-wavelength simulations, with partial spectral averaging where appropriate. The syntax to employ PSA in the single-microwire Si solar cell structures discussed herein is shown in the source code listings of section B.10.

B.5 Step 3: Generating the external-profile data file

The external-profile data file is generated using a MATLAB script (included in the source code listings of section B.10). This script first opens the `.dat` file produced in the initial mesh generation step, reading the values of `PMIUserField0` and `PMIUserField1` for each mesh element. Then, it produces a new `.dat` file containing the values of the external profile. A user-definable function determines the profile value at each spatial position. The script can be modified in several locations to specify the names of the input and output files, the names of the regions to process, the name of the output field, and most importantly, the function that determines the external profile value as a function of spatial coordinates. Comments within the script file provide details of how these modifications are made. The results of the script can be verified in Tecplot SV, as illustrated in Figure B.4.

To map profile values from the fixed-pitch (rectangular) FDTD grid to the varying element dimensions of the finite-element mesh, it is most conceptually simple to employ bilinear interpolation (the method used in the provided source code listing).* However, this method could potentially introduce aliasing artifacts if the absorption profiles vary on a shorter length scale than the finite element mesh. For example, mild distortion of the sinusoidal profile is visible in the center of the wire shown in Figure B.4, where the finite-element grid is most coarsely spaced.

One solution to this problem might be to apply an antialiasing (averaging) filter to the FDTD dataset that conserves the total optical generation within the simulation volume (assuming that such "blurring" of G_{opt} is insignificant in terms of the electrical behavior of the device). A better solution is to use MATLAB's built-in library of Delaunay triangulation functions to average the FDTD grid cells over the Voronoi region corresponding to each mesh element. An implementation of this approach is depicted in Figure B.5. We start by constructing the Delaunay triangulation corresponding to the finite-element mesh.

*Because the FDTD grid pitch must always be smaller than length scales of absorption or resonant field profiles, it is usually suitable to employ nearest-neighbor interpolation, which is considerably faster for large simulation volumes. We use bilinear interpolation here because it does not add significant processing time for the example 2D structures.

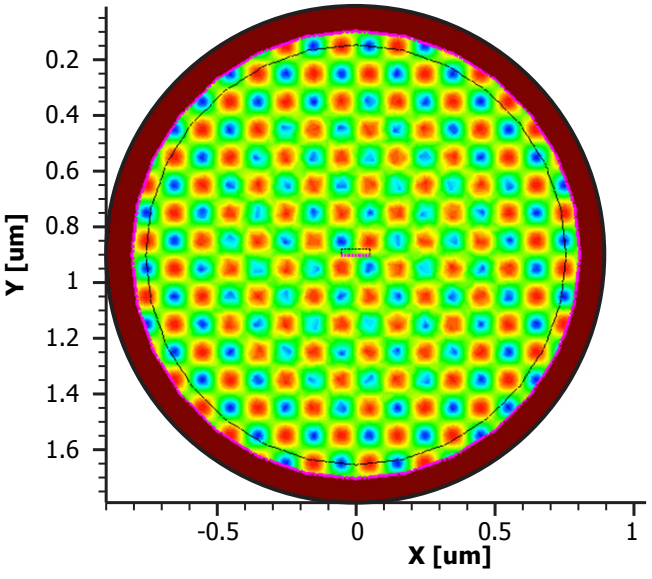


Figure B.4. Result of running the MATLAB script on the `noffset3d`-generated mesh from Figure B.2, using a sinusoid profile function to test the mapping (visualized in Tecplot SV).

We then iterate across the FDTD grid, assigning each grid cell (or "pixel") to the nearest finite-element vertex. We can finally iterate through each element of the finite-element mesh and calculate the profile value to store in the resulting `.dat` file: elements to which one or more FDTD cells were mapped are assigned the average value of these cells, whereas elements to which zero cells were mapped are assigned the value of the nearest FDTD cell (or an interpolated value). This approach is not strictly conservative, but is certainly more so than linear interpolation. In our implementation, we utilized a script that parsed the `.grd` file so that a constrained Delaunay mesh could be constructed for each region. However, it might also be feasible to implement this approach with only the knowledge of the mesh coordinates and physical extent of each region, masking the FDTD grid-mapping by region instead of confining the Delaunay triangulation by region.

In most cases, simple interpolation has proven suitable for importing FDTD generation profiles onto Sentaurus Device simulation meshes. For example, this approach yields accurate results for the simulation grid shown in Figure B.5 for $\lambda \gtrsim 400$ nm, below which wavelengths the shallow excitation profile is not conservatively mapped to the finite-element mesh. It is important to remember that, even if perfectly conservative optical generation mapping is performed, the finite-element mesh must still be dense enough to accurately represent the actual optical generation profile for device physics simulations. Thus, the most straightforward way to ensure accuracy in the grid conversion process is to simply increase the density of the device-physics mesh (if computationally feasible) until the largest mesh cells are of similar size as the FDTD cells, at which point

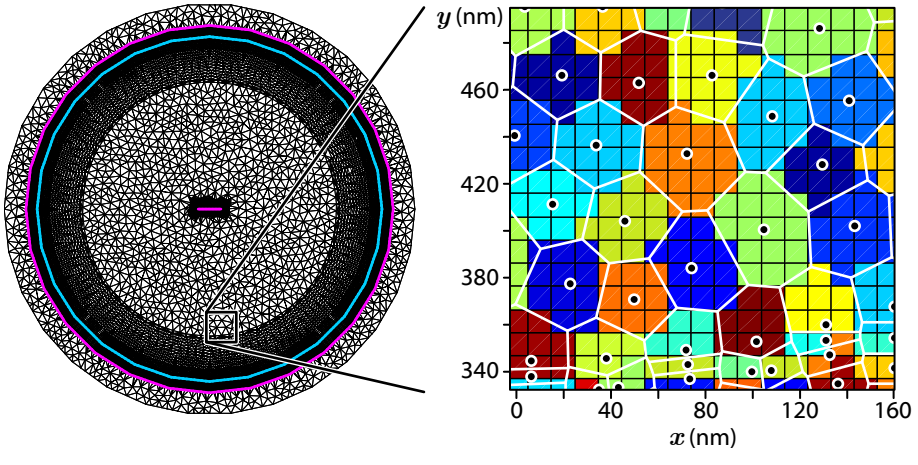


Figure B.5. Use of Delaunay triangulation to improve the accuracy of profile conversion. **Left:** finite-element simulation mesh for a $d = 1.61$ um single-wire solar cell structure. The magenta lines indicate the contact electrodes, and the blue line indicates the p-n junction. **Right:** profile conversion algorithm applied to a central region of the wire. The orthogonal grid lines lines delimit the FDTD grid cells ("pixels"), and the black markers indicate the vertices of the finite-element mesh. The white lines show the Voronoi polygons for each mesh region. Each FDTD cell is colored based on which mesh element to which it is mapped (the colors are randomly chosen to illustrate the mapping).

aliasing is not a concern. As a test, numerical integration over the FDTD grid (as calculated in Lumerical or MATLAB) should produce the same approximate result as numerical integration over the device-physics mesh in Tecplot SV.*

B.6 Step 4: Loading the external profile in Sentaurus.

Sentaurus Device is invoked using the following `File` section syntax to specify the external optical generation file that was generated generated in the previous step.

```
File
{
  Grid                  = "n@node|sde@_pof.grd"
  Doping                = "n@node|sde@_pof.dat"
  OpticalGenerationFile = "n@node|matlab@_optgen.dat"
  ...
}
```

The `File` section can also include directives that cause Sentaurus Device to load external profile values for several other fields, including carrier lifetime, emission rate, or trapped charge density.

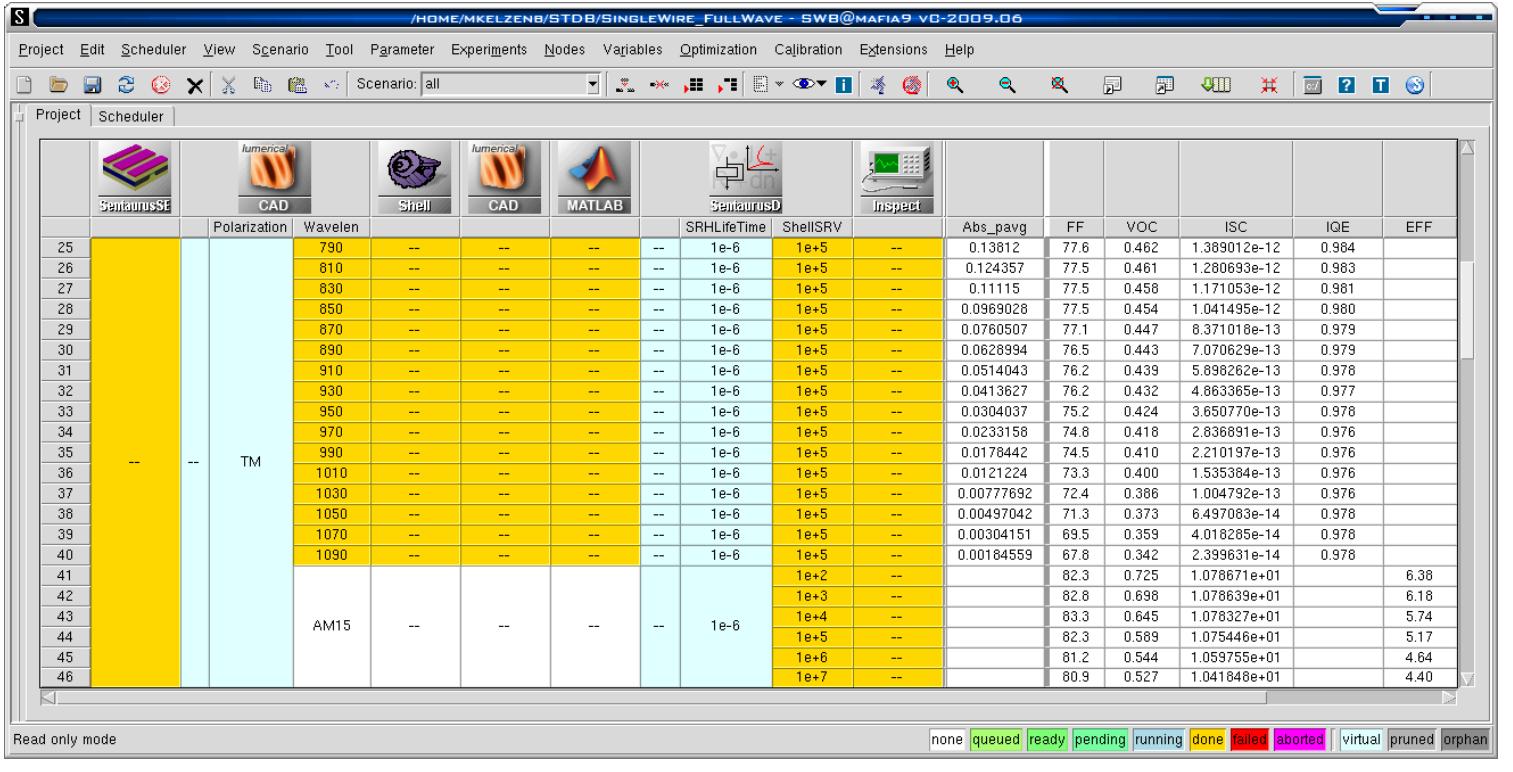
B.7 Integrating these steps into SWB

The above steps, as well as additional steps to run and process the Lumerical FDTD simulations, can be integrated into Sentaurus Workbench (SWB) as user-configured tools. User-configured tools are specified in the user's *tooldb* (tool database) file, which is located in the STDB directory, and can be edited from within SWB by selecting `Edit→Tool DB→User`. To add the features of this appendix to SWB, one can copy the Tcl code provided in section B.10 into his or her `tooldb` file (creating a new file if it does not exist).

The provided `tooldb` file allows SWB projects to script and invoke Lumerical FDTD/CAD and MATLAB steps as part of the simulation process flow. A screenshot of a project that simulates a single-microwire Si solar cell structure (like that depicted in Figure B.1) is shown in Figure B.6. The process steps employed by this project are:

1. **Sentaurus Structure Editor**—builds the finite-element mesh. Parameters for this step could include device dimensions or doping levels.

*Tecplot SV is unaware of whether two-dimensional simulation data correspond to planar or cylindrical devices. For cylindrical devices, the simulation-plane density profiles (e.g., `OpticalGeneration`) must be (manually) multiplied by the circumferential depth ($2\pi x$, assuming cylindrical symmetry about $x = 0$) to yield physically meaningful values when integrated. This can be accomplished by specifying a new data set in Tecplot SV (via "alter data"), such as the following:
`{CylOptGen [umz^-1 umr^-1 s^-1]} = 6.2832*1E-4*{OpticalGeneration [cm^-3*s^-1]}*{X [um]}`



			Polarization	Wavelen					SRHLifeTime	ShellSRV		Abs_pavg	FF	VOC	ISC	IQE	EFF
25	--	--	TM	790	--	--	--	--	1e-6	1e+5	--	0.13812	77.6	0.462	1.389012e-12	0.984	
26				810	--	--	--	--	1e-6	1e+5	--	0.124357	77.5	0.461	1.280693e-12	0.983	
27				830	--	--	--	--	1e-6	1e+5	--	0.11115	77.5	0.458	1.171053e-12	0.981	
28				850	--	--	--	--	1e-6	1e+5	--	0.0969028	77.5	0.454	1.041495e-12	0.980	
29				870	--	--	--	--	1e-6	1e+5	--	0.0760507	77.1	0.447	8.371018e-13	0.979	
30				890	--	--	--	--	1e-6	1e+5	--	0.0628994	76.5	0.443	7.070629e-13	0.979	
31				910	--	--	--	--	1e-6	1e+5	--	0.0514043	76.2	0.439	5.898262e-13	0.978	
32				930	--	--	--	--	1e-6	1e+5	--	0.0413627	76.2	0.432	4.863365e-13	0.977	
33				950	--	--	--	--	1e-6	1e+5	--	0.0304037	75.2	0.424	3.650770e-13	0.978	
34				970	--	--	--	--	1e-6	1e+5	--	0.0233158	74.8	0.418	2.836891e-13	0.976	
35				990	--	--	--	--	1e-6	1e+5	--	0.0178442	74.5	0.410	2.210197e-13	0.976	
36				1010	--	--	--	--	1e-6	1e+5	--	0.0121224	73.3	0.400	1.535384e-13	0.976	
37				1030	--	--	--	--	1e-6	1e+5	--	0.00777692	72.4	0.386	1.004792e-13	0.976	
38				1050	--	--	--	--	1e-6	1e+5	--	0.00497042	71.3	0.373	6.497083e-14	0.978	
39				1070	--	--	--	--	1e-6	1e+5	--	0.00304151	69.5	0.359	4.018285e-14	0.978	
40				1090	--	--	--	--	1e-6	1e+5	--	0.00184559	67.8	0.342	2.399631e-14	0.978	
41				AM15	--	--	--	--	1e-6	1e+2	--		82.3	0.725	1.078671e+01		6.38
42										1e+3	--		82.8	0.698	1.078639e+01		6.18
43										1e+4	--		83.3	0.645	1.078327e+01		5.74
44										1e+5	--		82.3	0.589	1.075446e+01		5.17
45										1e+6	--		81.2	0.544	1.059755e+01		4.64
46										1e+7	--		80.9	0.527	1.041848e+01		4.40

Figure B.6. Screenshot of SWB project that integrates Lumerical and MATLAB processing steps.

2. **Lumerical CAD**—prepares the Lumerical FDTD simulation structure. Parameters could include wavelength and polarization.

3. **Shell**—executes the Lumerical FDTD simulation using a `csh` script.

4. **Lumerical CAD**—loads the completed FDTD simulation, processes the results, and saves the optical generation profile and the FDTD grid in a MATLAB-format (`.mat`) data file.

5. **MATLAB**—loads the finite-element mesh from step 1 and the optical generation profiles from step 4. Generates the `.dat` file specifying the optical generation profile on the finite-element mesh.

6. **Sentaurus Device**—simulates the I–V characteristics of the solar cell using the optical generation profile data produced in step 5. Typical parameters could include carrier lifetimes and surface recombination velocities.

7. **Inspect**—extracts the operating parameters of the device (e.g., I_{sc}, V_{oc}, FF, η, E.Q.E., or I.Q.E.) and exports them as project variables in the SWB table.

This framework provides means to automate all of the simulation steps presented in this appendix, allowing each experiment to vary the device and simulation parameters such as geometry, material quality, and illumination wavelength. This platform has proven remarkably useful in my work, not only for performing elaborate optoelectronic simulations, but also for automating simple parametric sweeps in FDTD simulations. The SWB environment is particularly advantageous from an organizational and archival standpoint: SWB projects not only provide a tabulated record of the simulation parameters and results for each experiment, but also retain the input and output files for each simulation step in their directories, all of which are easily referenced by node number.

B.8 Simulating solar illumination

In the project shown in Figure B.6, each experiment corresponds to a single illumination wavelength (and polarization state). This allows the polarization-dependent spectral response of the device to be simulated in a straightforward manner. To simulate broadband, unpolarized illumination such as sunlight (i.e., the AM 1.5G spectrum), several general approaches are possible:

Single-experiment, broadband illumination: A single Lumerical simulation can utilize a broadband light source which spans the solar spectrum. During post-processing, the data can be normalized and weighted to yield the simulated response under solar illumination. Further details of this method are provided in the Lumerical reference guide. (🗗)

Although this approach may be the simplest, I have thus far been unable to adequately model the dispersion of Si across the solar spectrum in

Lumerical (see Figure B.3). Aside from the stark inaccuracy of my broadband simulation attempts, they have required manyfold greater memory and CPU time than the combined requirements of single-wavelength simulations spanning $\lambda = 300$ to 1100 nm in 20 nm increments.

Multi-experiment, summed monochromatic illumination: For each device geometry, numerous single-wavelength simulations are run to span the solar spectrum (each as a separate experiment, stemming from a common SDE node). To simulate solar illumination, an additional experiment is configured with a special keyword or value in place of a normal numerical value for the parameter *wavelength* (such as `"AM15"` or `15`). This experiment's CAD/FDTD steps are skipped; instead, the combined and weighted FDTD results from the preceding single-wavelength simulations provide the `OpticalGeneration` profile for its Sentaurus Device step. The special *wavelength* keyword triggers the MATLAB script to load the single-wavelength results, then sum and weight them appropriately for the desired solar spectrum, and store this composite photogeneration profile in a `.dat` file for the Sentaurus simulation.

Using this approach, both the 1-sun efficiency and the spectral response of the cell can be easily recorded in the project table within SWB. However, this method is somewhat cumbersome as it requires diligence in experimental layout and execution order, since all the single-wavelength FDTD simulations must be run before the broadband illumination profile can be calculated. For this reason it is not amenable to optimization within SWB. Nonetheless this is the technique I employ for most applications. By separating experiments by scenario, and by careful naming of the optical generation files, I am able to complete this process in two steps: the first, to specify and run all single-wavelength simulations; and the second, to run the broadband simulations.

Single-experiment, summed monochromatic illumination: The single-wavelength approach above can be condensed into a single experiment within SWB. The first Lumerical CAD step is scripted to prepare multiple single-wavelength simulation files to span the solar spectrum (under a single project node prefix). The FDTD execution step is scripted to run all of these simulations. Similarly, the second Lumerical and the MATLAB steps are altered to sequentially process a multitude of individual simulations. Finally, after processing all single-wavelength simulations, the MATLAB script can then weight and sum them appropriately to produce a single `OpticalGeneration` profile representing broadband solar illumination.

With this approach, wavelength and polarization would no longer appear as experiment parameters, and thus spectral response data would not be tabulated within the SWB project view. However, the scripts could easily be modified to record spectrally resolved simulation results as separate output files (just as Sentaurus Device produces `.plt` files recording the results of its internal I–V sweeps). This approach is also compatible with numerical optimization in SWB.

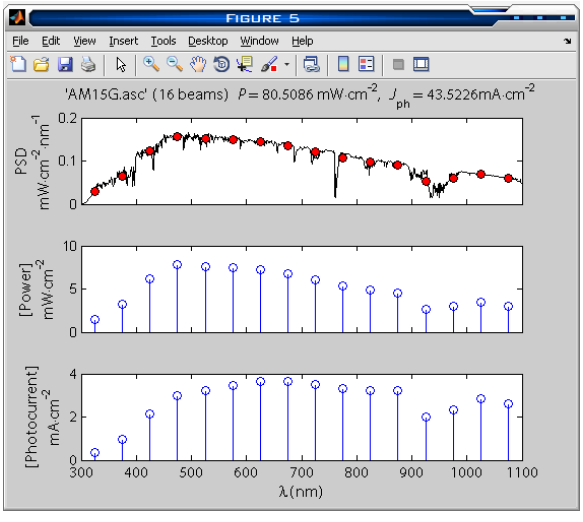


Figure B.7. Discrete solar spectrum script output.

A tabulation of weighting factors for discrete single-wavelength simulations to approximate solar illumination (corresponding to AM 1.5 global and direct reference spectra) is provided in section B.11 at the end of this appendix. Also included is a general-purpose MATLAB script for integrating or binning solar spectra. A screenshot of a discrete solar spectrum produced by this script is shown in Figure B.7.

B.8.1 Tips

- A single-wavelength source magnitude of `86.6` in Lumerical corresponds to 1 mW·cm^{-2} illumination intensity. This simplifies the weighting of each profile, as each can be directly multiplied by the desired illumination intensity (in mW·cm^{-2}) of the discrete ("binned") solar spectrum. The source magnitude (E_0) is specified in units of V·m^{-1}, thus the following equation determines illumination intensity:

$$\langle S \rangle = \frac{1}{2\mu_0 c} E_0^2 = \frac{\epsilon_0 c}{2} E_0^2 \qquad [\mathrm{W{\cdot}m^{-2}}] \tag{B.3}$$

- If polarization-dependent simulation data are not needed, each single-wavelength FDTD simulation can be run (simultaneously or consecutively) for both TE and TM polarization and then averaged in a single project node.

B.9 Tool information

B.9.1 Lumerical CAD (`lumcad`)

Lumerical CAD is used twice in our method of SWB integration. The first CAD step prepares and saves (but does not execute) the simulation structure. An intermediate shell step executes the FDTD simulation. The second CAD step then loads the simulation results, extracts the optical generation profiles, and saves this information to a MATLAB `.mat` file for subsequent processing.*

Input files :

- **Script file** (`lumcad_lcs.lsf`): The Lumerical script file, which will be pre-processed prior to invoking CAD. Scripts should call `exit(2)` at the end of their routine, otherwise the CAD window will remain open.
- **Template structure** (`lumcad_template.fsp`): Instead of using a script to generate the entire Lumerical simulation from scratch, it is often convenient to manually generate the structure beforehand as a "template" file. This way, a simple script can load the template structure, make only changes pertaining to the experiment's parameters, and then save the resulting structure under the appropriate filename for the current node.

Output files :

- **Lumerical structure** (`n@node@_lumstr.fsp`): The simulation structure, saved and ready for simulation.
- **Lumerical shell script** (`n@node@_lumstr.sh`): This is the shell script that Lumerical automatically generates when the above structure is saved. Executing it invokes the MPI FDTD program to run the simulation. It is configured within CAD under the menu `Simulation→Set parallel options` menu.
- **Lumerical log file** (`n@node@_lumstr_p0.log`): This is the log file produced by Lumerical FDTD as it runs the simulation.
- **MATLAB MAT file** (`n@node@_lummat.mat`): The optical generation profile information extracted from a completed simulation, saved for subsequent MATLAB processing.

*Lumerical CAD does not have a "batch mode." It will briefly open the GUI window while each script is executed.

B.9.2 Shell

The shell tool is a standard part of TCAD Sentaurus, and is not modified by the provided `tooldb` file. We utilize a C-shell (`csh`) script to launch the Lumerical FDTD simulations, and to examine their output files to make sure they ran to completion. The shell script command file is listed in section B.10.

Note: My FDTD simulations rely on Lumerical's *auto-shutoff* feature to determine when to terminate a simulation. If a simulation reaches the end of its allotted duration ("100% completion") before encountering the auto-shutoff criteria, I generally consider the results to be invalid. Thus the provided shell script will mark the FDTD nodes as *failed* unless auto-stop is reached.

B.9.3 MATLAB

Batch-mode MATLAB scripts are run in a nongraphical instance of MATLAB, and can make use of the full library of nongraphical MATLAB functions—including the parallelization toolkit, which is partularly useful for dealing with large simulation structures.

Input files :

- **MATLAB m-file** (`matlab_mat.m`): When MATLAB nodes are executed, this script file is pre-processed and then piped to the MATLAB command prompt. Note that a new instance of MATLAB will be invoked for each MATLAB step; thus each node will not have access to workspace variables in other MATLAB sessions. Upon completion, the script should call `exit(0)` to quit the MATLAB session, otherwise SWB will wait indefinitely for the program to terminate. For this reason, it is also best to enclose all code in a `try` block, so that `exit(1)` can be called in the event of any error.
- **Other input files:** One or more `.mat` files containing optical generation profiles from previous Lumerical CAD steps, and the DF-ISE `.dat` file from the mesh generation step.

Output files :

- **DF-ISE mesh data** (`n@node@_optgen.dat`): The optical generation profile mapped to the simulation grid.

- **Other output files:** The provided script will also store the mapped photogeneration data as a MATLAB `.mat` format, using the naming convention:

  `n@node|sde@_OptGen@Wavelen@@Polarization@.mat`

  This facilitates loading, weighting, and summing the numerous single-wavelength profiles when the special wavelength value of "15" is specified (see above).

B.10 Selected source code

This section provides the `tooldb` file used to integrate Lumerical and MATLAB as tools in Sentaurus Workbench. Source code is also provided for the following simulation steps of the single-microwire Si solar cell model highlighted above:

- The Lumerical structure generation script
- The shell script for launching Lumerical FDTD simulations from SWB
- The Lumerical data extraction script
- The MATLAB grid conversion script

Although each file is somewhat specific to the single-microwire Si solar cell structure shown here, they have been prepared in hopes of providing a clear example of the approach we have developed for importing arbitrary input profiles for simulations with Sentaurus Device. A MATLAB script for binning the solar spectrum for discrete single-wavelength simulations is also provided.

B.10.1 `tooldb` file

Listing B.1. User `tooldb` file (Tcl).

```
#BEGIN FILE

# Lumerical / MATLAB integration for Sentaurus Workbench
# Michael Kelzenberg, 2010
# California Institute of Technology

#SPECIAL_SETTINGS BEGIN
global tcl_platform
global env
#SPECIAL_SETTINGS END

#FILE-TYPES BEGIN
lappend WB_tool(file_types) lumscript
set WB_tool(lumscript,ext) lsf
lappend WB_tool(file_types) lumstructure
set WB_tool(lumstructure,ext) fsp
lappend WB_tool(file_types) lumbat
set WB_tool(lumbat,ext) sh
lappend WB_tool(file_types) lumlog
set WB_tool(lumlog,ext) log
```

```
lappend WB_tool(file_types) matlabm
set WB_tool(matlabm,ext) m
lappend WB_tool(file_types) matlabmat
set WB_tool(matlabmat,ext) mat
#FILE-TYPES END

#TOOL BEGIN lumcad
set WB_tool(lumcad,category) device
set WB_tool(lumcad,visual_category) device_old
set WB_tool(lumcad,acronym) lcs
set WB_tool(lumcad,after) all
set WB_manual(lumcad) /usr/lumerical/docs/FDTD_reference_guide.pdf
set Icon(lumcad) $env(STDB)/ico/cad.gif
set WB_tool(lumcad,exec_mode) batch ; # (interactive | batch)
set WB_tool(lumcad,setup) { os_ln_rel @lumscript@ n@node@_lcs.lsf "@pwd@" }
set WB_tool(lumcad,epilogue) \
     { make_sh_executable "$wdir" @node@; extract_vars "$wdir" @stdout@ @node@ }
set WB_binaries(tool,lumcad) CAD-noaccel
set WB_tool(lumcad,cmd_line) "n@node@_lcs.lsf"
set WB_tool(lumcad,input) [list lumscript lumstructure]
set WB_tool(lumcad,input,lumscript,file)  @toolname@_lcs.lsf
set WB_tool(lumcad,input,lumscript,newfile)  @toolname@_lcs.lsf
set WB_tool(lumcad,input,lumscript,label)  "Script file..."
set WB_tool(lumcad,input,lumscript,editor) text
set WB_tool(lumcad,input,lumscript,parametrized)  1
set WB_tool(lumcad,input,lumstructure,file)  @toolname@_template.fsp
set WB_tool(lumcad,input,lumstructure,newfile)  @toolname@_template.fsp
set WB_tool(lumcad,input,lumstructure,label)  "Template structure..."
set WB_tool(lumcad,input,lumstructure,editor) lumstructure
set WB_tool(lumcad,input,lumstructure,parametrized)  0
set WB_tool(lumcad,output) [list lumstructure lumbat lumlog matlabmat]
set WB_tool(lumcad,output,lumstructure,file)  n@node@_lumstr.fsp
set WB_tool(lumcad,output,lumbat,file)  n@node@_lumstr.sh
set WB_tool(lumcad,output,lumlog,file)  n@node@_lumstr_p0.log
set WB_tool(lumcad,output,matlabmat,file)  n@node@_lummat.mat
set WB_tool(lumcad,output,files) "n@node@_* pp@node@_*"
set WB_tool(lumcad,interactive,option)  "-edit"
set WB_tool(lumcad,batch,option)  "-run"
lappend WB_tool(all) lumcad
#TOOL END

#TOOL BEGIN matlab
set WB_tool(matlab,category) gridgen
set WB_tool(matlab,visual_category) gridgen
set WB_tool(matlab,acronym) mat
set WB_tool(matlab,after) all
set WB_manual(matlab) /usr/matlab/help/begin_here.html
set Icon(matlab) $env(STDB)/ico/matlab.gif
set WB_tool(matlab,exec_mode) batch ; # (interactive | batch)
set WB_tool(matlab,setup) { os_ln_rel @matlabm@ n@node@_mat.m "@pwd@" }
set WB_tool(matlab,epilogue) { extract_vars "$wdir" @stdout@ @node@ }
set WB_binaries(tool,matlab) "matlab"
set WB_tool(matlab,cmd_line) "< n@node@_mat.m"
set WB_tool(matlab,input) [list matlabm]
set WB_tool(matlab,input,matlabm,file)  @toolname@_mat.m
set WB_tool(matlab,input,matlabm,newfile)  @toolname@_mat.m
set WB_tool(matlab,input,matlabm,label)  "Matlab m-file..."
set WB_tool(matlab,input,matlabm,editor) text
set WB_tool(matlab,input,matlabm,parametrized)  1
set WB_tool(matlab,output) [list doping grid]
set WB_tool(matlab,output,doping,file)  n@node@_optgen.dat
set WB_tool(matlab,output,grid,file)  n@node@_optgen.grd
set WB_tool(matlab,output,files) "n@node@_* pp@node@_*"
set WB_tool(matlab,interactive,option)  ""
set WB_tool(matlab,batch,option)  "-nojvm -nodisplay"
```

```
lappend WB_tool(all) matlab
#TOOL END

#INPUT-EDITORS BEGIN
set WB_binaries(editor,text) gedit
lappend WB_editor(all) text
set WB_binaries(editor,lumstructure) CAD
lappend WB_editor(all) lumstructure
#INPUT-EDITORS END

#OUTPUT-VIEWERS BEGIN
set WB_viewer(lumstructure,files) "\{*n@node@_*.fsp\}"
set WB_viewer(lumstructure,label) ".fsp Files (Lumerical CAD)"
set WB_viewer(lumstructure,nbfiles) 3
set WB_viewer(lumstructure,cmd_line) @files@
set WB_viewer(lumstructure,exec_dir) @pwd@
set WB_binaries(viewer,lumstructure) CAD-noaccel
lappend WB_viewer(all) lumstructure

set WB_viewer(lumbat,files) "\{*n@node@_*.sh\}"
set WB_viewer(lumbat,label) "Shell file (for MPI-Lumerical)"
set WB_viewer(lumbat,nbfiles) 3
set WB_viewer(lumbat,cmd_line) @files@
set WB_viewer(lumbat,exec_dir) @pwd@
set WB_binaries(viewer,lumbat) gedit
lappend WB_viewer(all) lumbat

set WB_viewer(lumlog,files) "\{*n@node@_lumstr_p0.log\}"
set WB_viewer(lumlog,label) "Lumerical log file"
set WB_viewer(lumlog,nbfiles) 5
set WB_viewer(lumlog,cmd_line) @files@
set WB_viewer(lumlog,exec_dir) @pwd@
set WB_binaries(viewer,lumlog) "gnome-terminal -x tail -f -n +1"
lappend WB_viewer(all) lumlog

set WB_viewer(matlabmat,files) "\{*n@node@_matlabmat.mat\}"
set WB_viewer(matlabmat,label) "Matlab MAT file"
set WB_viewer(matlabmat,nbfiles) 3
set WB_viewer(matlabmat,cmd_line) @files@
set WB_viewer(matlabmat,exec_dir) @pwd@
set WB_binaries(viewer,matlabmat) matlab
lappend WB_viewer(all) matlabmat
#OUTPUT-VIEWERS END


#RunLimits
#accepted values for restriction_model:
#none,per_project,per_user,per_swb
set WB_limits(restriction_model) "per_user"
set WB_limits(lumcad,run_limit) 4
set WB_limits(matlab,run_limit) 4
#Run Limits end

#TCL-SOURCE BEGIN
# MK 2009
proc make_sh_executable { wdir node} {
    foreach file [glob -nocomplain -directory $wdir n${node}*.sh] {
        file attributes $file -permissions 00755
    }
}
#TCL-SOURCE END
```

B.10.2 Lumerical structure generation script

Listing B.2. Lumerical script for preparing FDTD simulation structures (`lumcad_lcs.lsf`).

```
########################################################################################
#
# Lumerical structure-generating script
# (c) 2009 Michael Kelzenberg
# California Institute of Technology
#
# This script loads a "template" file and modifies it according to the paramters
# set for this node in SWB (i.e., wavelength and polarization).  The input/
# output files are defined below.
#
# Tip: Be carefule using "#" to comment out lines.  This can confuse the
# sentaurus pre-processor.  As a rule, always put a space after "#" if you are
# writing comments.
#
########################################################################################

clear;

template_file = "lumcad_template.fsp";
lumstr_file = "n@node@_lumstr.fsp";

Polarization = "@Polarization@";
Lambda = @Wavelen@ * 1e-9; # (m)

WireDiameter = 1.61e-6;
BoxWidth = 1.8e-6; #This is the width of the monitors surrounding the wire
IllumWidth = 4e-6; #This is the width of the illumination source

write("Single-wire solar cell FDTD structure generator script for LUMERICAL");
write("Michael Kelzenberg, 2010");
write("Settings: "+Polarization+"-polarization, "+num2str(Lambda*1e+9)+" nm.");
write("Loading template file: " + template_file);

load(template_file);

switchtolayout;

setparallel("Shell/batch file type","Linux multi-processors");
setparallel("Create parallel shell/batch file when saving fsp file",1);
setparallel("Number of processors per node",2);

simulation;
select("FDTD");

# Select a reasonable simulation duration (longer than will be required)
set("simulation time",1e-12);
if( Lambda > 800e-9 ) {
    set("simulation time",5e-12);
}
if( Lambda > 950e-9 ) {
    set("simulation time",10e-12);
}
if( Lambda > 1070e-9 ) {
    set("simulation time",20e-12);
}

# Set correct boundary conditions depending on polarization
if( Polarization == "TE" ) {
    set("x min bc","Anti-Symmetric");
```

```
} else {
    set("x min bc","Symmetric");
}

# Set wavelength and polarization of the source
sources;
select("singleSource");
set("amplitude",86.8);  #86.8 V/m is a illum. intensity of 1 mW/cm2
set("polarization",Polarization);
set("wavelength start",Lambda);
set("wavelength stop",Lambda);

# Set the bandwith for partial spectral averaging
monitors;
freq = 2.998e+8/Lambda;
n = real(getindex("SiAspnes",freq));
waveFract = 1; # 2 is for half-wave, 1 for full-wave
deltaF = freq / (1 + (waveFract*WireDiameter*n/Lambda) );

write("Partial spectral averaging deltaF = " + num2str( deltaF/1e12 ) + " THz");

select("topMonitor");
set("delta",deltaF);
select("bottomMonitor");
set("delta",deltaF);
select("leftMonitor");
set("delta",deltaF);
select("rightMonitor");
set("delta",deltaF);
select("wireMonitor");
set("delta",deltaF);

write("Saving modified structure: " + lumstr_file);

save(lumstr_file);

write("Completed successfully");

exit(2);
```

B.10.3 FDTD execution shell script

Listing B.3. Shell script (`csh`) for launching MPI FDTD simulations from SWB (`cshell_csh.cmd`).

```
# Shell script for invoking FDTD simulations
# Michael Kelzenberg, 2010 (c)
# California Institute of Technology

# This script executes the .sh file generated by the prior Lumerical CAD step
# then parses the output to ensure that the simulation ran to completion.

#setdep @previous@

# TCL code in our tooldb file now takes care of setting the execut bit for
# Lumerical's MPI shell scripts, so the following is not needed here:
# chmod 755 *.sh

echo "*******************************************************************************"
echo "Shell script for running Lumerical FDTD simulations"
echo "Michael Kelzenberg, 2010"
echo ""
echo "Warning!  Using 'Abort' in SWB will NOT terminate MPI FDTD simulations."
```

```
echo "You must manually kill each process."
echo ""
echo "*******************************************************************************"

./n@node|lumcad@_lumstr.sh


echo ""
echo "*******************************************************************************"
echo "FDTD complete."
echo "*******************************************************************************"
echo "Last 20 lines of simulation log file:"
echo ""
tail -n 20 n@node|lumcad@_lumstr_p0.log
echo ""
echo "*******************************************************************************"
if ( { grep -q "Simulation completed successfully" \
                                                n@node|lumcad@_lumstr_p0.log } ) then
  if ( { grep -q "the autoshutoff criteria are satisfied" \
                                                n@node|lumcad@_lumstr_p0.log } ) then
    echo "Auto-shuttoff criteria were reached.  Simulation complete."
    echo -n "*******************************************************************************"
    echo "*******"
    exit 0
else
  echo "ERROR: The auto-shutoff criteria were not reached.  Simulation is incomplete."
  echo "This typically means that simulation results are invalid.  This node has been"
  echo "marked as FAILED to prevent unintentional use of these simulation results."
  echo ""
  echo "If these results are indeed correct, this error check can be disabled within "
  echo "the shell script for this tool."
  endif
else
  echo "ERROR: The simulation does not appear to have finished successfully.  "
  echo "Please check the log file for more information."
endif

echo "*******************************************************************************"
exit 1
```

B.10.4 MATLAB mesh conversion script

Listing B.4. MATLAB mesh conversion script (`matlab_mat.m`).

```
# MATLAB optical generation grid conversion script for lumsen project
# (c) 2010 Michael Kelzenberg
# California Institute of Technology

#setdep @previous@
#setdep @node|sde@
#setdep @node|lumcad@

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB script for importing arbitrary profiles onto DF-ISE simulation
% grids for use with Sentaurus TCAD.
%
% Michael Kelzenberg, 2010
%
% This script reads a DF-ISE .dat file to determine the spatial coordinates of
% of each grid point.  It then generates a new .dat file containing an
% OpticalGeneration profile that was calculated by FDTD.  The key input/output
% settings and the mapping function are indicated by comments in the code.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%DAT file
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%This should be a valid DF-ISE .dat file (i.e. generated by mesh or
%noffset3d.  The meshing program must be scripted to store the x- and y-
%position of each vertex of the grid as "PMIUserField0" and
%"PMIUserField1", respectively.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

datFile = 'n@node|sde@_msh.dat';
grdFile = 'n@node|sde@_msh.grd';

%FDTD MAT file %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This should be the Matlab MAT file generated by the Lumerical CAD script
%including:
% Pabs_x,Pabs_y     X and Y specification of grid (m)
% freq          Freq. of simulation (Hz)
% Pabs *        Matrix of power absorption (W/m3)
% Ngen *        Matrix of optical generation rate (per cm3 per s)
% IntgPwr *     Total power absorption (W/m)
% Current *     Total photocurrent (A per um device length)
% Absfrac *     Fraction of absorbed light, i.e. Absorption Quantum Efficiency
%   *these variables followed by '_pavg' corespond to partial spectral averaging
%
% Note: presently, only Pabs_x, Pabs_y, and Ngen_pavg are used by this script.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

FDTDFile = 'n@node|lumcad1@_lummat.mat';

%Regions to process %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%These are the regions to perform the optical generation mesh conversion.
%This must be a cell array of region names, including double-quotes (")
%around each region name!!!
%  Example syntax: regionsToProcess = {'"Base_region"', "Emitter_region"' };
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
regionsToProcess = {'"InnerContact_region"' '"Emitter_region"' '"Base_region"'};

%The output dat and grd files are used for monochromatic-illumination device
% physics simulations (the next step in this experiment).
outputFile = ['n@node@_optgen.dat'];
outputGrid = ['n@node@_optgen.grd'];

%The export data file (.mat) is saved so that a MATLAB script can sum together
% multiple single-wavelength simulations to approximate solar illumination.
% We chose a file name that is unique to the wavelength, polarization, and
% the device physics simulation grid:
exportFile = 'n@node|sde@_OptGen@Wavelen@@Polarization@.mat';

%Number of data values to output per line in output DAT file
numperline = 10;

try

disp('');
disp('------------------------------------------------------------------');
disp(['OptGenConverter Version 2']);
disp(['(c) 2010 Michael Kelzenberg']);
disp(['California Institute of Technology']);
disp('------------------------------------------------------------------');
disp(' ');

% A wavelength value of '15' is the signal to assemble 1-sun AM1.5G illumination
if (@Wavelen@ == 15)
    nodenum = @node@;  nodenum_sde = @node|sde@;
    disp(['Invoking am15proc.m script to generate combined-wavlength OptGen' ...
        ' profile...']);
    disp(' ');
    disp('This will fail if the prerequisite single-wavelength FDTD profiles');
```

```
    disp('  have not been generated for this device physics simulation grid,');
    disp('  (n@node|sde@)');
    disp(' ');
    am15proc;
    exit(0);
end

disp(['Opening DAT file ' datFile ]);

grd = fopen(datFile);
if (grd < 1)
    error(['Error opening file ' datFile ' for reading.']);
    %exit
end

if ( ~isequal( fgetl(grd), 'DF-ISE text'))
    disp('Error with grid file format: It might not be a DF_ISE text file.');
    disp('Please double-check input file.  The first line should read:');
    disp('     DF-ISE text');
    error('File parse error');

end
fln = 1;

verts = [];
regions = {};

nl = fgetl(grd); fln=fln+1;
while( isempty( regexp(nl, 'nb_vertices *= *[0-9]+') ) && ~feof(grd) )
    nl = fgetl(grd); fln=fln+1;
end
tmp=regexp(nl, '[0-9]+','match');
numverts = str2num(tmp{1});
disp(['  File reports ' num2str(numverts) ' vertices']);

nl = fgetl(grd); fln=fln+1;
while( isempty( regexp(nl, 'nb_edges *= *[0-9]+') ) && ~feof(grd) )
    nl = fgetl(grd); fln=fln+1;
end
tmp=regexp(nl, '[0-9]+','match');
numedges = str2num(tmp{1});
disp(['  File reports ' num2str(numedges) ' edges']);

nl = fgetl(grd); fln=fln+1;
while( isempty( regexp(nl, 'nb_elements *= *[0-9]+') ) && ~feof(grd) )
    nl = fgetl(grd); fln=fln+1;
end
tmp=regexp(nl, '[0-9]+','match');
numelems = str2num(tmp{1});
disp(['  File reports ' num2str(numelems) ' elements']);

nl = fgetl(grd); fln=fln+1;
while( isempty( regexp(nl, 'nb_regions *= *[0-9]+') ) && ~feof(grd) )
    nl = fgetl(grd); fln=fln+1;
end
tmp = regexp(nl, '[0-9]+','match');
numregions = str2num(tmp{1});
disp(['  File reports ' num2str(numregions) ' regions']);

%Advance to data section of file...
nl = fgetl(grd); fln=fln+1;
while( isempty( regexp(nl, 'Data.*\{', 'once') ) && ~feof(grd) )
    nl = fgetl(grd); fln=fln+1;
end
```

```
if ( feof(grd) )
    disp('Unexpected end-of-file, no data processed.');
    disp(['Line: ' num2str(fln)]);
    error('File parse error.');
end


regionArray = [];
disp(' ');
disp('Reading data points...');
%Main reading loop.  Look for PMIUserField 0 or 1 data sets...
while ~feof(grd)

    nl = fgetl(grd); fln=fln+1;
    while ( isempty( regexpi(nl,'\s*function\s*=\s*PMIUserField[01]','once'))...
            && ~feof(grd) )
        nl = fgetl(grd); fln=fln+1;
    end
    if (feof(grd))
        break
    end

    tmp = regexp(nl, '[01]','match');
    axisNumber = str2num(tmp{1});

    nl = fgetl(grd); fln=fln+1;
    while ( isempty( regexpi(nl, '\s*validity\s*=\s*\[\s*".*"\s*\]', 'once'))...
            && ~feof(grd) )
        nl = fgetl(grd); fln=fln+1;
    end

    if (feof(grd))
        error(['File Parse Error near line ' num2str(fln)]);
        break
    end
    tmp = regexp(nl, '".*"','match');
    regionName = tmp{1};

    nl = fgetl(grd); fln=fln+1;
    while ( isempty( regexpi(nl, '\s*Values\s*\(\s*[0-9]+\s*\)', 'once') )...
            && ~feof(grd) )
        nl = fgetl(grd); fln=fln+1;
    end

    if (feof(grd))
        disp(['File Parse Error near line ' num2str(fln)]);
        break
    end
    tmp = regexp(nl, '[0-9]+','match');
    numElems = str2num(tmp{1});

    dataPoints = [];
    while (1)
        nl = fgetl(grd); fln = fln+1;
        if(isempty(regexp(nl,'[0-9]+') ) )
            break
        else
            thisline = regexp(nl,'[\.\-\e\E\+0-9][\s\.\-\e\E\+0-9]*','match');
            thisline = thisline{1};
            dataPoints = [dataPoints str2num(thisline)];
        end
        if ( ~isempty(regexp(nl,'}','once') ))
            break
        end
    end
```

```
    disp(['  Region ' regionName ' read ' num2str(length(dataPoints)) '/' ...
        num2str(numElems) ' elements for axis ' num2str(axisNumber) ]);

    %Error if data points disagree with number stated in header
    if ( numElems ~= length(dataPoints) )
        disp(['Error: number of data points does not match file header']);
        disp(['Parse error near line ' num2str(fln)]);
        error(['File structure error in region ' regionName]);
    end

    existingRegion = 0;
    for n=1:length(regionArray)
        canRegion = regionArray{n};
        if (isequal(regionName,canRegion.name))
            existingRegion = n;
        end
    end

    if (existingRegion)
        if (axisNumber == 0)
            regionArray{existingRegion}.xdata = dataPoints;
        else
            regionArray{existingRegion}.ydata = dataPoints;
        end

        if ~isequal( length(regionArray{existingRegion}.xdata), ...
                length(regionArray{existingRegion}.ydata) )
            disp(['Error: number of x data points does not match number of ' ...
                    'y data points']);
            error(['File structure error in region ' regionName ]);
        end

    else
        newRegion.name = regionName;
        if (axisNumber == 0)
            newRegion.xdata = dataPoints;
            newRegion.ydata = [];
        else
            newRegion.ydata = dataPoints;
            newRegion.xdata = [];
        end
    newRegion.gdata = zeros(size(dataPoints));
        regionArray{end+1} = newRegion;
    end

end

for n=1:length(regionArray)
    if ~isequal( length(regionArray{n}.xdata), length(regionArray{n}.ydata) )
            disp(['Error: number of x data points does not match number of '...
                'y data points']);
            error(['File structure error in region ' regionArray{n}.name ]);
    end
end

disp(' ');
disp('Completed reading DAT file');
disp(['  Read ' num2str(length(regionArray)) ' region(s)']);
disp(' ');
fclose(grd);

%Now ensure that data was successfully read for all requested regions
regionsToProcess = unique(regionsToProcess);
for n=1:length(regionsToProcess)
```

```
    reqName = regionsToProcess{n};
    hasRegion = 0;
    for m=1:length(regionArray)
        if isequal( reqName, regionArray{m}.name )
            hasRegion=1;
        break;
        end
    end
    if ~hasRegion
        disp(['Error: Vertex position information for requested region ' ...
            reqName ' not contained within this grid.']);
        error(['Unable to process region: ' reqName ]);
    end
end

% We're done parsing grid file -- Now we load FDTD results and define the
% mapping function.  Note that the spatial translation applied to the x- and
% y-coordinates in the mapping function is specific to the project geometry:
disp(' ');
disp(['Loading MAT file ' FDTDFile ]);
load(FDTDFile);
optGenMatrix = OptGen';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Mapping function:
newoptgen = @(xi, yi) interp2(Pabs_x,Pabs_y, optGenMatrix, xi*1e-6,yi*1e-6 );
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Now ready to write the output data file...
disp(['Opening output file ' outputFile ]);

ogo = fopen(outputFile,'w');
if (ogo < 1)
    error(['Error opening file ' outputFile ' for writing.']);
end

fprintf(ogo, 'DF-ISE text\n\n');
fprintf(ogo, ...
   'Info {\n  version     = 1.0\n  type        = dataset\n  dimension   = 2\n');
fprintf(ogo, '  nb_vertices = %d\n  nb_edges    = %d\n  nb_faces    = 0\n',...
    numverts, numedges);
fprintf(ogo, '  nb_elements = %d\n  nb_regions  = %d\n  datasets    = [ ',...
    numelems, numregions);
for n=1:length(regionsToProcess)
    fprintf(ogo,'"OpticalGeneration" ');
end
fprintf(ogo, ']\n  functions   = [ ');
    for n=1:length(regionsToProcess)
        fprintf(ogo,'OpticalGeneration ');
    end
fprintf(ogo, ']\n}\n\nData {\n\n');

for n=1:length(regionsToProcess)
    reqName = regionsToProcess{n};
    hasRegion = 0;
    for m=1:length(regionArray)
        if isequal( reqName, regionArray{m}.name )
            hasRegion=m;
            break;
        end
    end
    if (hasRegion)
        reg = regionArray{hasRegion};

        disp( ['Proessing Optical Generation for region ' reg.name '...'] );
```

```
        fprintf(ogo,['Dataset ("OpticalGeneration") {\n  function  = '...
            'OpticalGeneration\n  type      = scalar\n  dimension = 1\n'...
            'location  = vertex\n  validity  = [ ' reg.name ' ]\n' ] );
        fprintf(ogo, '  Values (%d) {\n', length(reg.xdata) );

    gdata = zeros(size(reg.xdata));
        nl = 1;
        for nv=1:length(reg.xdata)

            ogi = newoptgen(reg.xdata(nv), reg.ydata(nv));
            fprintf(ogo, ' %22e', ogi);
        gdata(nv) = ogi;
            nl = nl + 1;
            if (nl > 10)
                fprintf(ogo, '\n');
                nl = 1;
            end
        end
        if (nl > 1)
            fprintf(ogo,'\n');
        end
        fprintf(ogo, '  }\n}\n\n');

        disp( ['    ' num2str(length(reg.xdata))  ' processed'] );
        regionArray{hasRegion}.gdata = gdata;
      end
 end

fprintf(ogo,'\n\n}');
fclose(ogo);
disp(['Finished writing output file ' outputFile ]);

disp(' ');
disp(['Copying from grid file: ' grdFile]);
copyfile(grdFile,outputGrid);
disp(['To grid file: ' outputGrid]);

disp(' ');
disp(['Exporting generation profile: ' exportFile ]);
save( exportFile ,'regionArray','numverts','numedges','numelems','numregions');

disp(' ');
disp('Processing complete!');

exit(0);

catch ME
    disp(ME);
    exit(1); %This will mark the node as 'failed' in SWB.
end
```

B.11 Solar spectrum weightings for discrete simulations

Table B.1. Discrete solar spectrum, 100 nm bins.

Wavelength		AM 1.5G		AM 1.5D	
Beam (nm)	Bin (nm)	P mW·cm^{-2}	J_{ph} mA·cm^{-2}	P mW·cm^{-2}	J_{ph} mA·cm^{-2}
350	(300–400)	4.792	1.353	3.193	0.901
450	(400–500)	14.059	5.103	11.625	4.219
550	(500–600)	15.093	6.695	13.374	5.933
650	(600–700)	13.898	7.286	12.572	6.591
750	(700–800)	11.302	6.837	10.331	6.249
850	(800–900)	9.440	6.472	8.758	6.004
950	(900–1000)	5.637	4.320	5.298	4.060
1050	(1000–1100)	6.444	5.457	6.084	5.153
1150	(1100–1200)	3.168	2.939	3.022	2.803
1250	(1200–1300)	4.300	4.335	4.111	4.145
1350	(1300–1400)	1.167	1.270	1.121	1.220
1450	(1400–1500)	0.701	0.820	0.682	0.797
1550	(1500–1600)	2.554	3.193	2.481	3.102
1650	(1600–1700)	2.209	2.940	2.149	2.860
1750	(1700–1800)	1.444	2.038	1.407	1.987
1850	(1800–1900)	0.023	0.034	0.022	0.033
1950	(1900–2000)	0.283	0.446	0.279	0.438
2050	(2000–2100)	0.688	1.138	0.678	1.120
2150	(2100–2200)	0.848	1.470	0.835	1.448
2250	(2200–2300)	0.699	1.268	0.690	1.252
2350	(2300–2400)	0.483	0.915	0.478	0.906

Table B.2. Discrete solar spectrum, 50 nm bins.

Wavelength		AM 1.5G		AM 1.5D	
Beam (nm)	Bin (nm)	P mW·cm^{-2}	J_{ph} mA·cm^{-2}	P mW·cm^{-2}	J_{ph} mA·cm^{-2}
325	(300–350)	1.416	0.371	0.817	0.214
375	(350–400)	3.246	0.982	2.272	0.687
425	(400–450)	6.192	2.123	4.925	1.688
475	(450–500)	7.779	2.980	6.606	2.531

Discrete solar spectrum, 50 nm bins

Wavelength		AM 1.5G		AM 1.5D	
Beam (nm)	Bin (nm)	P mW·cm^{-2}	J_{ph} mA·cm^{-2}	P mW·cm^{-2}	J_{ph} mA·cm^{-2}
525	(500–550)	7.642	3.236	6.708	2.841
575	(550–600)	7.460	3.460	6.667	3.092
625	(600–650)	7.198	3.628	6.500	3.277
675	(650–700)	6.719	3.658	6.087	3.314
725	(700–750)	6.006	3.512	5.467	3.197
775	(750–800)	5.318	3.324	4.884	3.053
825	(800–850)	4.874	3.243	4.510	3.001
875	(850–900)	4.575	3.228	4.256	3.003
925	(900–950)	2.652	1.978	2.488	1.856
975	(950–1000)	2.977	2.341	2.802	2.203
1025	(1000–1050)	3.470	2.868	3.269	2.703
1075	(1050–1100)	2.986	2.589	2.825	2.449
1125	(1100–1150)	1.190	1.080	1.134	1.029
1175	(1150–1200)	1.961	1.858	1.871	1.774
1225	(1200–1250)	2.259	2.232	2.157	2.131
1275	(1250–1300)	2.046	2.104	1.958	2.014
1325	(1300–1350)	1.186	1.268	1.140	1.218
1375	(1350–1400)	0.002	0.002	0.002	0.002
1425	(1400–1450)	0.123	0.141	0.119	0.137
1475	(1450–1500)	0.570	0.678	0.555	0.660
1525	(1500–1550)	1.289	1.586	1.252	1.540
1575	(1550–1600)	1.265	1.607	1.229	1.562
1625	(1600–1650)	1.154	1.513	1.121	1.470
1675	(1650–1700)	1.057	1.428	1.029	1.390
1725	(1700–1750)	0.892	1.241	0.870	1.210
1775	(1750–1800)	0.556	0.796	0.542	0.776
1825	(1800–1850)	0.023	0.033	0.022	0.033
1875	(1850–1900)	0.000	0.000	0.000	0.000
1925	(1900–1950)	0.014	0.022	0.014	0.022
1975	(1950–2000)	0.266	0.424	0.261	0.416
2025	(2000–2050)	0.309	0.504	0.304	0.496
2075	(2050–2100)	0.379	0.634	0.373	0.625
2125	(2100–2150)	0.448	0.768	0.441	0.756
2175	(2150–2200)	0.400	0.702	0.394	0.691

Discrete solar spectrum, 50 nm bins

Wavelength		AM 1.5G		AM 1.5D	
Beam (nm)	Bin (nm)	P mW·cm^{-2}	J_{ph} mA·cm^{-2}	P mW·cm^{-2}	J_{ph} mA·cm^{-2}
2225	(2200–2250)	0.374	0.671	0.369	0.662
2275	(2250–2300)	0.325	0.597	0.322	0.590
2325	(2300–2350)	0.276	0.517	0.273	0.512
2375	(2350–2400)	0.208	0.398	0.206	0.394

Listing B.5. MATLAB script for integrating/binning the solar spectrum.

```
function [totalinputcurrent totalinputpower centerpts beampwr specpwr ...
    beamphot warnstring] = spectralCalculator(bins, centerpts)
% spectralCalculator(bins, centerpts) Calculates the discrete weightings
%                                     for optical simulations based on
%                                     'binning' a continuous reference
%                                     spectrum (e.g. the solar spectrum)
%                                     into individual 'beams'.
%
% Arguments:
%   centerpts - (n) discrete wavelengths for simulations ('beams')
%      Note: centerpts can be left empty to have the 'beam' wavelengths
%      chosen automatically, in a manner which conserves energy as well
%      as photon flux.  (Arbitrarily chosen beam energies will not
%      genearlly conserve power.)
%   bins - a list of (n+1) wavelengths, specifying the wavelength range
%   over which to sum photons for each 'beam'.  The power of the n^th
%   beam is calculated by summing the photons from bins(n) to bins(n+1).
%
% Environment:
%   The workspace must contain the global variable SPECDATA:
%     (first column)  - reference spectrum wavelengths (nm)
%     (second column) - power spectral density (mW/cm2/nm)
%     (third column)  - photon flux spectral density (per cm2 per nm)
%   See also: loadSpectrum.m
%
% Outputs
%   totalinputcurrent - Photocurrent of reference spectrum within bin
%                       range (mA/cm2)
%   totalinputpower   - Power within binned range of reference spectrum
%                       (mW/cm2)
%   centerpts         - The wavelength of each 'beam' (nm)
%   beampwr           - The power of each 'beam' (mW/cm2)
%   specpwr           - The spectral power density of each beam
%                       (mW/cm2/nm)
%   beamphot          - The photon flux of each beam (per cm2 per s)
%   warnstring        - Diagnostic error message
%
% Example use:
%   spectralCalculator([280 1100],800)
%   ans =
%      43.7352
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global SPECDATA;

wl_ = SPECDATA(:,1);
phot_ = SPECDATA(:,3);
```

```
pwr_ = SPECDATA(:,2);

if (isempty(centerpts))
    autocenter = 1;
    centerpts = zeros(1,length(bins)-1);
else
    autocenter = 0;
end

numBins = length(centerpts);

beampwr = zeros(1,numBins);
specpwr = zeros(1,numBins);
beamphot = zeros(1,numBins);


photspectraldensity = @(wl) interp1(wl_, phot_, wl);
pwrspectraldensity = @(wl) interp1(wl_, pwr_, wl);
binphots = @(wl1, wl2) quadgk( photspectraldensity, wl1, wl2, ...
    'MaxIntervalCount', 5000);
binpwr = @(wl1, wl2) quadgk( pwrspectraldensity, wl1, wl2, ...
    'MaxIntervalCount', 5000);

totalInputCurrent = 1e3*1.61E-19*binphots(bins(1), bins(end)); %mA/cm2
totalInputPower = quad( pwrspectraldensity, bins(1), bins(end)); %mW/cm2

warnstring = [];

for nBin=1:numBins
    photsInBin = binphots( bins(nBin), bins(nBin+1) );  %num/cm2/sec
    energyInBin = binpwr( bins(nBin), bins(nBin+1) ) / 1000 / 1.61E-19;
                                                          %eV/cm2/sec
    if (autocenter)
        photEnergy = energyInBin / photsInBin;
        photWl = 1.24 / photEnergy * 1000;
        centerpts(nBin) = photWl;
    else
        photEnergy = 1.24 / (centerpts(nBin) / 1000);
    end
    beamphot(nBin) = photsInBin;  %num/cm2/sec
    beampwr(nBin) = photsInBin * photEnergy * 1.61E-19 * 1000; %mW/cm2
    specpwr(nBin) = beampwr(nBin) / (bins(nBin+1) - bins(nBin));
                                                          %mW/cm2/nm
    if ( centerpts(nBin) > bins(nBin+1) || centerpts(nBin) < bins(nBin) )
        warnstring = ['Warning: At least one bin wavelength lies '...
            'outside the boundaries for that bin.'];
    end
end

totalinputcurrent = totalInputCurrent;
totalinputpower = totalInputPower;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function FileName = loadSpectrum
%  loadSpectrum      load raw spectum data from the ASTM G173-03 file:
%                      col 1: Wavelength nm
%                      col 2: W*m-2*nm-1
%
%  Inputs: none (prompts file dialog)
%  Outputs:
%    FileName - the name of the selected file
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global SPECDATA;
```

```
[FileName,PathName] = uigetfile('*.asc','Select spectrum data file');

SPECDATA = load([PathName FileName]);

%convert to mW/cm2/nm
SPECDATA(:,2) = SPECDATA(:,2)*1000/100/100;

%add col. for photons/cm2/nm
energies = 1.24 ./ (SPECDATA(:,1)./1000) * 1.61E-19; %energy/photon (J)

SPECDATA(:,3) = SPECDATA(:,2)./1000./energies;

%add col. for photocurrent/cm2/nm
SPECDATA(:,4) = SPECDATA(:,3).*1.61E-19;
```