Solis
1D Semiconductor Device Simulator

Sidi HAMADY
Full Professor, Dr. habil. Eng.
Université de Lorraine, France
LMOPS Lab., Université de Lorraine & CentraleSupélec, France
http://www.hamady.org

The latest Solis release (extracted size less than 10 MB) is freely available here:
http://www.hamady.org/download/solis_windows_64bit.zip
http://www.hamady.org/download/solis_linux_64bit.tgz
The latest version of this manual:
http://www.hamady.org/download/solis_simulator.pdf

Reference[1]:
Sidi Ould Saad Hamady. *Solis: a modular, portable, and high-performance 1D semiconductor device simulator*. Journal of Computational Electronics, 2020.

Solis simulator engine uses:
The Lua programming language, (C) Lua.org, PUC-Rio.
The Lua Just-In-Time Compiler, (C) Mike Pall.
The LIS linear solvers, (C) The SSI Project, Kyushu University, Japan.

Solis device editor, code editor, data plotter and calculator use:
The IUP GUI toolkit, (C) Tecgraf/PUC-Rio.
The Scintilla Component, (C) Neil Hodgson.

Summary

List of figures

List of tables

Prolog

Solis is a one-dimensional semiconductor device simulator with a high-performance and modular calculation engine coded in C++ and a user-friendly interface developed in C and completely independent from the calculation engine. Solis implements the drift-diffusion model and simulates graded or abrupt heterostructures (for solar cells, detectors, etc.) taking into account various recombinations mechanisms (Auger, radiative and Shockley-Read-Hall (SRH)), traps (donor-like and acceptor-like), incomplete dopants ionization, Schottky rectifying contacts, spontaneous and piezoelectric polarization in III-N materials, photogeneration using AM1.5 solar spectrum or any user-defined spectrum, etc. It outputs the current-voltage, capacitance-voltage and quantum efficiency, in addition to band diagram, spatial distribution of carriers density, ionized dopants and traps, generation/recombination, electric field, etc. Solis was designed with portability, flexibility and performance as the main criteria, e.g. its core simulation engine is coded in standard C++ with no dependency on any proprietary system, it natively supports Linux and Windows, all the physical models can be set using the integrated and fast scripting engine, and the discretization scheme, initial guess, voltage and wavelength sweep... can be defined by the user. The Solis input format is an easy-to-use plain text format with simple syntax and some useful features such as variable definition and mathematical parser. I started developing Solis in 2009 and presented the first testing release in the 37th International Symposium on Compound Semiconductors (ISCS) in 2010 in Japan [2].

The Solis distribution includes the following independent tools, as also shown in figure 1:

- The simulator driver, **soliscomp.exe** (or **soliscomp** under Linux), controlling the simulator engine implementing the drift-diffusion model. It solves the coupled Poisson-Continuity equations of the drift-diffusion model using a high-performance and fully modular C++ code. It implements a finite difference scheme and uses both fully coupled Newton method and the decoupled Gummel iterative method. The simulator integrates also a Lua scripting engine that can be used to implement user-defined physical models and parameters permitting an advanced control of the simulation process. Lua was choosen as a scripting language due to its speed (optimized, thanks to the included Just-In-Time LuaJIT interpreter), simplicity and light footprint. The simulator driver is a command line executable with a device input in a simple and clean text format. It can be easily controlled and automated by any language (Bash, Python... and Lua!) giving the user an unprecedented flexibility control over the simulation.

- A graphical device editor, **solisdevice.exe** (or **solisdevice** under Linux), implemented in

C. This tool gives an easy-to-use graphical frontend to Solis.

- An advanced code editor, **solisedit.exe** (or **solisedit** under Linux), implemented in C, to edit the device structure using the straightforward Solis syntax. This editor offers all functionality found in modern editors such as syntax highlighting, autocompletion, markers, indentation control, find/replace, file explorer... and are fully customizable. In addition to Solis syntax, it supports Lua, Python, C/C++, LaTeX... and can be used as a general code editor.

- A data plotter, **solisplot.exe** (Windows) or **solisplot** (Linux), implemented in C and C++. This tool is used by Solis to plot the simulation results but could also be used as a standalone data plotter.

- An advanced scientific calculator, **soliscalc.exe** (or **soliscalc** under Linux), implemented in C. In addition, Solis includes an interactive terminal emulator (**solisterm**, only under Linux), a standalone version of the embedded terminal in SolisEdit.

The whole Solis distribution size, including documentation and examples, is less that 10 MB.

These tools are independent. For instance, one can use the command line simulator driver (**soliscomp.exe** or **soliscomp** under Linux) alone (or launch it with Bash, Python, Lua...), or **solisedit.exe** to enter the device structure (and the models written in Lua, if any) and start automatically the simulator, or use the graphical device editor **solisdevice.exe** to graphically build the device and simulate it. I myself use the code editor and sometimes the graphical device editor. An input file created with the code editor can be open/modified by the graphical device editor and vice-versa. The Solis input format is not tightly linked to any particular tool.

To know if a new version is available, click *Menu/Help/Check for Update...* or visit my website: http://www.hamady.org

This Solis user manual is organized as follows:

The first chapter describes the physical theory used in Solis and the related implemented numerical methods.

The second chapter gives an overview on how to use Solis. It contains the description of the Solis input format as well as an overview of the physical model definition using the integrated Lua programming language. This chapter ended with the list of the included semiconductors along with the corresponding physical parameters.

The third chapter contains the description of the tools included in the Solis distribution, the code editor, the graphical device editor and the scientific calculator.

The fourth and last chapter gives a set of step-by-step examples.

The bibliography completes the manual.

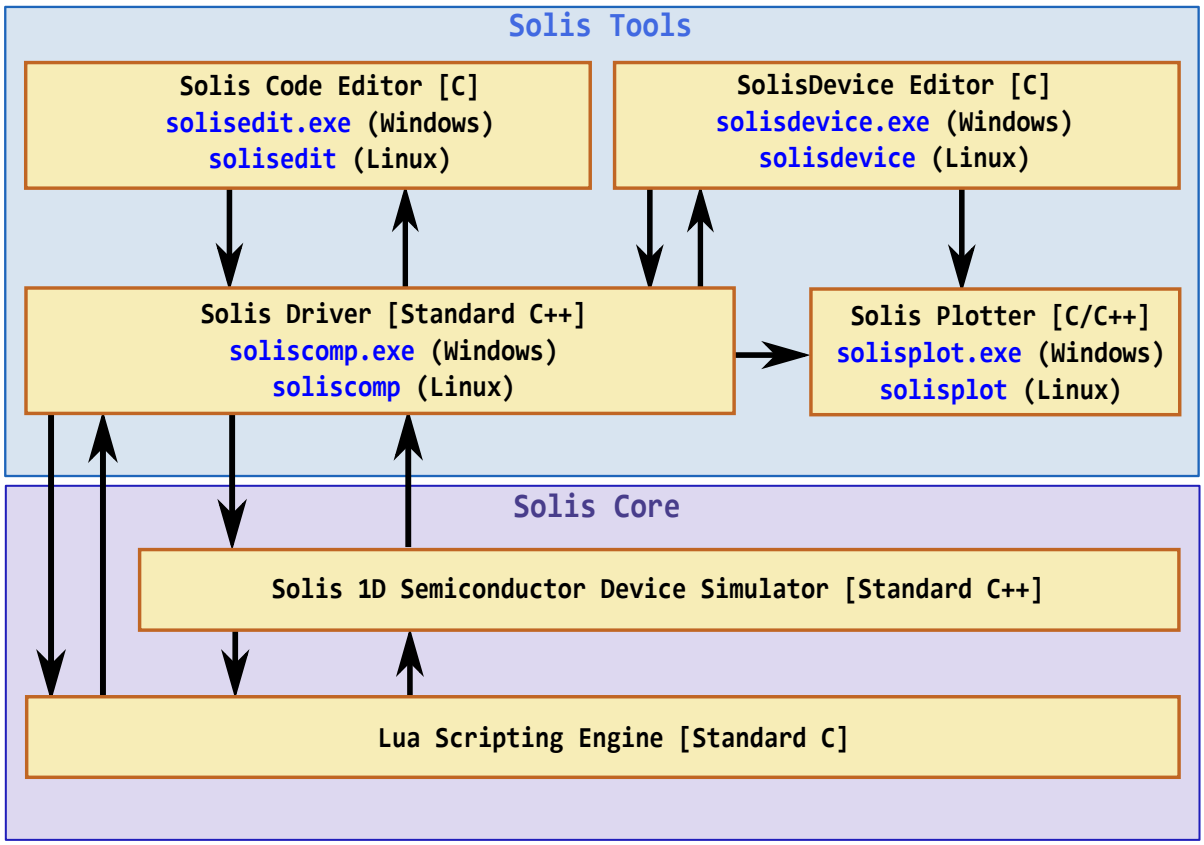

Figure 1: The Schematic Solis architecture. The Solis core part is entirely coded in standard C++ (the Lua engine being coded in standard C) and is easily portable to almost all operating systems, in addition to the natively supported Windows and Linux systems. The Solis code editor, device editor and the data plotter are developed in C (and partly in C++ for the plotter) and use the IUP GUI toolkit coded in C [3] and the Scintilla component coded in C++ [4].

Chapter 1
Theory and Numerical Procedure

Contents

Parameter	Normalization factor
Permittivity	ε_0
Potential	Thermal voltage kT/q
Energy	Thermal energy kT
Carrier concentration	$N = 10^{22}\ m^{-3}$
Space	Debye length $L = \sqrt{\varepsilon_0 kT/(q^2 N)}$
Mobility	$\mu_0 = 0.1\ m^2/Vs$
Recombination	$\beta_0 = 10^{-20}\ m^3/s$
Continuity factor	$\gamma^2 = \mu_0 \mathrm{kT}/\left(\beta_0 \mathrm{qNL}^2\right)$

Table 1.1: Normalization parameters used for the drift-diffusion model.

1.1 Drift-Diffusion Model

The drift-diffusion model is the standard physical model used to describe and calculate the electrical properties of semiconductor devices. It is deduced from the Boltzmann transport equation and is either versatile and precise to describe the physics of semiconductor devices. A comprehensive and impressive theoretical development could be found in the Selberherr's book [5].

The drift-diffusion model consists basically of three equations: the Poisson equation and the two current continuity equations as expressed in one-dimension and normalized form as following:

$$\begin{aligned}
\frac{d}{dz}\left(\varepsilon(z)\frac{dV(z)}{dz} + P(z)\right) &= n(z) - p(z) + N_A^-(z) - N_D^+(z) + \sum_t N_{tA}^-(z) - \sum_t N_{tD}^+(z) \\
\frac{dJ_n}{dz} &= -GR(z)/\gamma^2 \\
\frac{dJ_p}{dz} &= +GR(z)/\gamma^2 \\
J_n(z) &= -n(z)\mu_n \frac{d\phi_n(z)}{dz} \\
J_p(z) &= +p(z)\mu_p \frac{d\phi_p(z)}{dz}
\end{aligned} \tag{1.1}$$

With the electrostatic potential V; the spontaneous and piezoelectric polarization P, if any; the permittivity ε; the free electrons concentration n; free holes concentration p; the ionized donors $\mathrm{N_D^+}$ and ionized acceptors $\mathrm{N_A^-}$; the ionized donor-like traps $\mathrm{N_{tD}^+}$ and ionized acceptor-like traps $\mathrm{N_{tA}^-}$. Concentrations vary with the position coordinate z. $\mathrm{J_n}$ and $\mathrm{J_p}$ are the electron and hole currents and GR the net generation term. γ^2 is a normalization parameter (the continuity factor shown in table 1.1). μ_n and μ_p are the electrons and holes mobilities, ϕ_n and ϕ_p the quasi-Fermi levels. All parameters in these equations are *dimensionless*. The normalization

parameters are defined in table 1.1. Currents are expressed in term of quasi-Fermi levels for simplicity, and includes both drift and diffusion terms. The net generation term GR(z) includes light-induced carriers generation (photogeneration), and the various recombination phenomena including nonradiative Shockley-Read-Hall (SRH), radiative and Auger mechanisms.

The free electrons and holes concentrations are given by:

$$\begin{aligned} n(z) &= n_i e^{(+V(z)+V_n(z)-\phi_n(z))} \\ p(z) &= n_i e^{(-V(z)+V_p(z)+\phi_p(z))} \end{aligned} \tag{1.2}$$

Where n_i is the intrinsic carrier concentration. V_n and V_p are the band parameters for heterostructures and nonuniform graded materials (i.e. for homostructures $V_n = V_p = 0$).

The net generation term $GR(z)$ is expressed as:

$$\begin{aligned} GR(z) &= G(z) - R_D(z) - R_I(z) - R_A(z) \\ G(z) &= \int_\lambda \alpha(\lambda, z)\phi(\lambda) exp(-\alpha(\lambda, z)z)\, d\lambda \\ R_D &= \beta(np - n_i^2) \\ R_I &= \sum_t \frac{(np - n_i^2)}{\tau_p(n + n_i exp(E_t - E_i)) + \tau_n(p + n_i exp(-E_t + E_i))} \\ R_A &= (np - n_i^2)\,(C_n n + C_p p) \end{aligned} \tag{1.3}$$

Where G(z) is the photogeneration rate, $R_D(z)$ the direct (radiative) recombination rate, $R_I(z)$ the Shockley-Read-Hall recombination rate and $R_A(z)$ the Auger recombination rate. $R_I(z)$ account for dopants and traps (**t** index stands for **t**rap), each described by an activation energy E_t and electron and hole lifetimes τ_n and τ_p. E_i is the intrinsic Fermi level. $\alpha(\lambda)$ is the absorption coefficient, varying with the light wavelength λ, and ϕ the incident light flux. β is the direct recombination coefficient. C_n and C_p are the Auger coefficients for electrons and holes respectively.

1.2 Solving Methods

The drift-diffusion model as defined in equation 1.1 is solved to determine three parameters: the potential $V(z)$ and the quasi-Fermi levels $\phi_n(z)$ and $\phi_p(z)$. The other quantities are then calculated using these fundamental parameters. The first crucial step in solving these equations is the discretization scheme. I use the finite difference method with the Scharfetter-Gummel discretization method [6] that gives the numerical stability needed in these partial differential equation (PDE) system. The more classical finite difference schemes are completely inefficient to solve this system.

The discretized PDE system is then solved using two methods implemented in Solis, giving complete flexibility:

- The Gummel iterative decoupled method. In this method, illustrated in figure 1.1, the three

equations are solved iteratively until convergence: given a starting set of vectors for the three parameters (V(z), ϕ_n(z) and ϕ_p(z)) at each node, Solis solves the Poisson discretized system, using the Newton-Raphson method, and determine the new value of the potential vector V(z) in the discretized space. Then the electron continuity equation is solved using the obtained potential and the new value of the ϕ_n(z) vector is determined. After that, the hole continuity equation is solved, using the same obtained potential and the previously calculated ϕ_n(z), and the new value of the ϕ_p(z) vector is determined. And so on. The solver iterates until convergence. The Gummel method is usually efficient and Solis uses internally a damping strategy that ensures convergence in most cases. The Gummel method can converge slowly, particularly for problems where the three drift-diffusion equations are highly coupled.

- The Newton method that simultaneously solves the discretized full nonlinear drift-diffusion PDE system using the Newton-Raphson method. This method is suitable for highly coupled cases and can converge fast providing that a good starting solution is given. Solis uses the equilibrium solution, or the last saved solution, to provide a starting point for the solver. Solis can also use the Gummel method to determine the starting solution for the Newton method.

Internally both Gummel iterative decoupled method and Newton full method use the Newton-Raphson algorithm to solve the nonlinear system created by the discretization procedure. For both methods, Solis automatically calculates the starting solution vectors to ensure and speed up the convergence in most cases.

Solis handles both Dirichlet and Neumann boundary conditions. The device contacts can be ohmic (ideal or not) or rectifying (Schottky).

The numerical parameters, such as the starting initial guess, the damping parameter and tolerance, the boundary conditions, can be easily set by the user to handle specific structures.

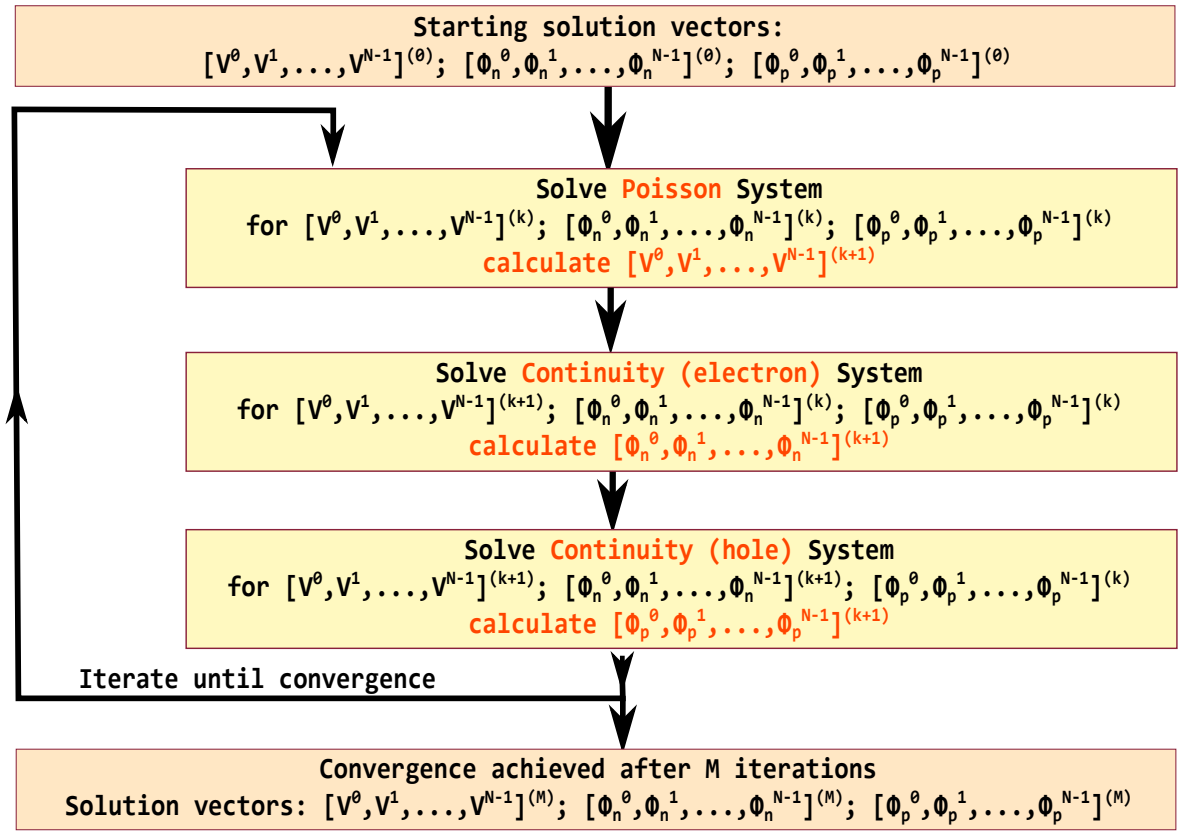

Figure 1.1: The Gummel method procedure.

Chapter 2
Getting Started with Solis

Contents

2.1 Installation

Solis is distributed in a portable version and do not need to be installed. Just download solis_windows_64bit.zip (Windows 64bit) or solis_linux_64bit.tgz (Linux 64bit), unzip/untar in any location (local user directory, USB key or Memory stick for example). The compressed Solis distribution size is less than 5 MB and the extracted distribution size is less than 10 MB. The Solis distribution includes a `bin` directory where the solis simulator driver, dynamic libraries and the Solis tools reside; a `doc` directory with the Solis documentation; an `examples` directory with templates used to define physical models for Solis and some useful input files including solar cells and ultraviolet detectors; an `icons` directory [1]; a `work` directory where to put simulation input and results; a `config` directory where Solis saves user settings.
In the `bin` directory are included the Solis simulation engine driver, core libraries and four tools:

- The simulation engine driver, **soliscomp.exe** (Windows) or **soliscomp** (Linux)
- The graphical device editor, **solisdevice.exe** (Windows) or **solisdevice** (Linux)
- The code editor, **solisedit.exe** (Windows) or **solisedit** (Linux)
- The data plotter, **solisplot.exe** (Windows) or **solisplot** (Linux)
- The scientific calculator, **soliscalc.exe** (Windows) or **soliscalc** (Linux)

As already presented in the prolog, these four tools are independent and are not mandatory to use. The simulator can be used alone, in the command line. The Solis input format is a completely open and clean text format (cf. section 2.3) and can be created/edited/parsed by any tool (text editor, scripting language, Bash...). Nonetheless, using the graphical device editor or the code editor is an efficient/fast and modern way to perform simulations.

To know if a new version is available, click *Menu/Help/Check for Update...* or visit my website: http://www.hamady.org
Under Linux, Solis includes also an interactive terminal emulator (solisterm), a standalone version of the embedded terminal in SolisEdit.
This terminal emulator is loaded and available to use if the VTE library is installed.
Usually the required VTE library is installed by default, but in some systems it must be installed:
Under CentOS, install vte by typing the following commands:

```
sudo yum install -y epel-release
sudo yum install vte
```

Under Ubuntu, install vte by typing the following command: `sudo apt-get install libvte9`

Under Ubuntu, if you encounter error such as:

```
error while loading shared libraries: libgtk-x11-2.0.so.0
```

reinstall the required library by typing in the terminal:

[1] The `icons` directory contains files useful to define an application icon and/or custom launcher for Linux desktop.

```
sudo apt-get -reinstall install libgtk2.0-0
```

Under Ubuntu/Debian 64bit, if you encounter error such as:
`failed to load module "canberra-gtk-module"`
reinstall the required library by typing in the terminal:
`sudo apt install libcanberra-gtk-module libcanberra-gtk3-module`

2.2 Solis Command Line Options

The Solis simulator driver, **soliscomp.exe** under Windows and **soliscomp** under Linux, is a command line tool to be executed in a console window. Open your favorite terminal under Linux or launch cmd.exe under Windows (or simply click the Command Prompt in Start/Accessories) and launch Solis with the appropriate options. The syntax is as follows:
`soliscomp -run infile [-out outfile]`
`infile` is the input filename described in the next section 2.3.
`-out outfile` is used to specify the output filename where Solis saves the simulation messages.
Examples:
`soliscomp -run solarcell.solis`
`soliscomp -run solarcell.solis -out simulmessages.txt`

2.3 Solis Input File Format

2.3.1 Solis Input Syntax

Solis simulates a device structure such as shown in figure 2.1. To describe such a device, Solis uses a very simple input file format, inspired by the INI format (click to see the Wikipedia definition of this format). With this text format, Solis input file are simple to generate/handle, to read and understand. The Solis input file extension is **.solis**.

The basic syntax of a Solis input is as follows:

```
# Solis <SL>
# the first line above is mandatory

$thick = 1
$dop = 1E+18

[Device]
   Name="PN"
   Temperature=300

[Layer1]
   Material="GaAs"
   Thickness=$thick
   Dopant.N=$dop
```

```
   Dopant.Type="Acceptor"

[Layer2]
   Material="GaAs"
   Thickness=10
   Dopant.N=1e+16
   Dopant.Type="Donor"

[Anode]
   Anode.Type="Ohmic"

[Cathode]
   Cathode.Type="Ohmic"

[Voltage]
   Voltage.Source="Sweep"
   Voltage.Start=0
   Voltage.End=1
   Voltage.Step=0.01
   Voltage.Jsup=0

[Light]
   Light.Source="AM1.5"
```

The first line in the Solis input (`# Solis <SL>`) should be always there. The input file is composed of sections. Each section contains `key=value` statements. The key name can be prefixed with the section name or not. For example, in the `[Voltage]` section, the source could be specified as `Voltage.Source="Sweep"` or `Source="Sweep"`. Each line (or text at the end of a line) beginning with the hash symbol (#) is treated as a comment and is ignored by the simulator. In this way, one can document the Solis input code.

Before the first section, one can define **variables** using the syntax:

```
$varName=varValue
```

One variable per line, and each variable name should begin with the $ symbol. Solis includes a mathematical parser (the one used by SolisCalc, detailed in section 3.4), and the variable value could be any valid mathematical expression, like in the following example:

```
$dsig = 0.5
$dop = 1e18 * exp(-1 / $dsig)
```

More generally, any numerical parameter could be given as a mathematical expression. The defined variable could be affected to any numerical parameter in any section, as in the following statement:

```
Dopant.N=$dop       # $dop defined previously
```

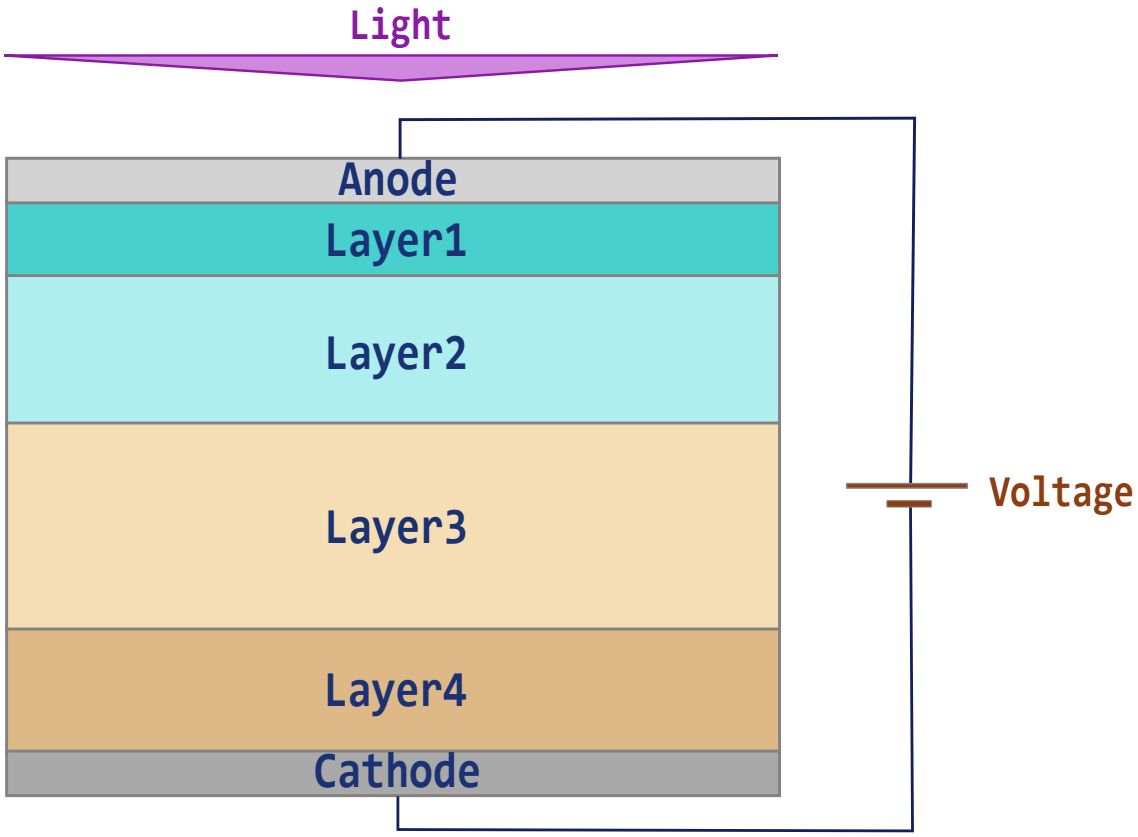

Figure 2.1: A schematic structure of a device as simulated by Solis.

2.3.2 Device Section

The first section in the Solis input file is the [Device] section:

```
[Device]
    # Device records
    Name="SolarCell"            # Device name
    Temperature=300             # Operating temperature in Kelvin
    Polarization.Scale=0        # Spontaneous/piezoelectric polarization scale
```

This section contains three keys: the device name (e.g. "GaAs solar cell"); the operating temperature in Kelvin and the spontaneous/piezoelectric polarization scale for III-N wurtzite materials. Only the device name is mandatory. If not given, the temperature will be set to 300 K. The polarization scale will be explained in section 2.3.3.6.

2.3.3 Layer Section

The following listing defines the first layer parameters:

```
[Layer1]                        # First layer
    Label="p-GaAs"              # Layer label
    Material="GaAs"             # Material name
    Composition.x=0             # for ternary alloys A(x)B(1-x)C
    Bandgap.Bowing=0            # Bandgap bowing for ternary alloys
    Affinity.Bowing=0           # Affinity bowing for ternary alloys
    Thickness=1                 # Layer thickness (micrometers)
    Bandgap=1.42                # Bandgap (eV)
    Permittivity=12.9           # Dielectric constant (in unity of eps0)
    NC=4.7e+17                  # Density of states in cond. band (1/cm3)
    NV=9e+18                    # Density of states in valence band (1/cm3)
    Mobility.N=4000             # Mobility of electrons (cm2/Vs)
    Mobility.P=400              # Mobility of holes (cm2/Vs)
    Dopant.N=1e+17              # Doping concentration (1/cm3)
    Dopant.E=0.045              # Dopant ionization energy (eV)
    Dopant.Taun=0.001           # Electron lifetime (seconds)
    Dopant.Taup=0.001           # Hole lifetime (seconds)
    Dopant.Type="Acceptor"      # Type ("Acceptor", "Donor" or "Intrinsic")
```

Listing 2.1: First Layer Section: Basic Parameters.

The next layers are defined in the same way, with the layer's section name incremented: [Layer1], [Layer2], ..., [Layer9]. Up to nine layers can be defined.

The listing 2.1 defines the basic layer parameters: the Thickness, Bandgap, Permittivity, density of states in in conduction NC band and in valence band NV, the mobility of electrons Mobility.N, the mobility of holes Mobility.P, the doping concentration Dopant.N and ionization energy Dopant.E, the electron lifetime Dopant.Taun, the hole lifetime Dopant.Taun and the doping type Dopant.Type. Solis uses the incomplete ionization model in all cases.

Solis includes a materials database including GaAs used in this example, as presented in section 2.6. It is not necessary to enter all parameters. On one hand, Solis uses the default

value for any parameter not explicitly defined in the layer section and, on the other hand, Solis saves only parameters with value different from the default one. Only the `Thickness` and the `Dopant.N` and `Dopant.Type` are mandatory [2]. If the `Material` statement is not defined, Solis uses monocrystalline silicon as the default material.

The layer section contains also keys defining physical models, as detailed in section 2.4. All parameters included in these models could be graded.

In addition to the basic layer parameters, one can define many other parameters:

2.3.3.1 Urbach tail

The **Urbach** parameters to describe disorder in the layer:

```
[Layer1]                          # First layer
                                  # ...
    Urbach.Enable="Yes"           # "Yes" or "No" to enable or disable Urbach
    Urbach.EA=0.02                # energy in eV for acceptors
    Urbach.ED=0.02                # energy in eV for donors
    Urbach.GA=2e+19               # density of states in 1/cm3/eV for acceptors
    Urbach.GD=2e+19               # density of states in 1/cm3/eV for donors
    Urbach.TaunA=1e-7             # electron lifetime in seconds for acceptors
    Urbach.TaupA=1e-7             # hole lifetime in seconds for acceptors
    Urbach.TaunD=1e-7             # electron lifetime in seconds for donors
    Urbach.TaupD=1e-7             # hole lifetime in seconds for donors
```

Listing 2.2: First Layer Section: Urbach Parameters.

These parameters are defined in the classical Urbach model [7] that is used to describe the sub-bandgap absorption in disordered materials such as amorphous silicon or high defect density III-N alloys. This sub-bandgap absorption can be written as:

$$\alpha(E) = \alpha_u exp\left(\frac{E - E_G}{E_A}\right) \quad or \quad \alpha_u exp\left(\frac{E - E_G}{E_D}\right) \tag{2.1}$$

The first term corresponds to the Urbach tail for acceptors and the second one for donors. α(E) the absorption coefficient ; E_A corresponds to the Solis parameter `Urbach.EA` ; E_D corresponds to `Urbach.EA` ; E is the incident photon energy and E_G the bandgap α_u the Urbach absorption parameter calculated by Solis to ensure the continuity of the α(E) at the bandedges. The Urbach density of states and lifetime are used in the recombination term in the continuity equations.

2.3.3.2 Refractive Index and Extinction Coefficient

The **complex refractive index**, with refractive index n and extinction coefficient k, is included in the layer section as follows:

[2] In fact, when the user defines a custom model for the doping, as detailed in section 2.4.3, it is not necessary to set `Dopant.N` and `Dopant.Type` since the user-defined model gives these parameters.

```
[Layer1]                                # First layer
                                        # ...
    Index.n=3.5                         # Refractive index
    Index.k=0.1                         # Extinction coefficient
                                        # ...
                                        # ...
```

Listing 2.3: First Layer Section: Refractive Index.

The given `Index.n` and `Index.k` are used to calculate the refractive index and extinction coefficient at a given photon energy E using relation 2.2:

$$k(E) = k_o \left(\frac{2E_G}{E}\right)\left(\frac{E - E_G}{E_G}\right)^m \tag{2.2}$$

Where E_G is the bandgap and k_o equal to `Index.k` given in the listing. `Index.k` corresponds then to the extinction coefficient at $E = 2 \times E_G$. The parameter `m` in the equation is equal to `1/2` if the bandgap is direct, and `2` otherwise. In this scheme, $k = 0$ if $E < E_G$, unless Urbach tail is considered. User-defined model for the complex index can be set using, e.g., data from experimental measurements. The definition of custom physical models is detailed in section 2.4.

2.3.3.3 Mobility

The **Mobility** is defined in the layer section as shown in the following listing:

```
[Layer1]                                # First layer
                                        # ...
    Mobility.N=1500                     # Electron mobility in cm2/Vs
    Mobility.P=500                      # Hole mobility in cm2/Vs
                                        # ...
```

Listing 2.4: First Layer Section: Mobility.

A custom mobility model (e.g. Caughey-Thomas) can be defined using the integrated scripting language, as detailed in section 2.4.

2.3.3.4 Doping

The **Doping** is defined in the layer section as shown in the following listing:

```
[Layer1]                                # First layer
                                        # ...
    Dopant.N=1e16                       # Concentration in 1/cm3
    Dopant.Nr=1e19                      # Reference concentration in 1/cm3
    Dopant.Taun=1e-4                    # Electron lifetime in seconds
    Dopant.Taup=1e-4                    # Hole lifetime in seconds
    Dopant.Type="Donor"                 # Dopant type: "Donor" or "Acceptor"
                                        # ...
```

Listing 2.5: First Layer Section: Doping.

The reference concentration is used to take into account the variation of the carrier lifetime with the doping concentration, using the following formula:

$$\begin{aligned} \tau_n &= \frac{\tau_{n0}}{1 + \frac{N}{N_r}} \\ \tau_p &= \frac{\tau_{p0}}{1 + \frac{N}{N_r}} \end{aligned} \tag{2.3}$$

Where N is the doping concentration, τ_{n0} and τ_{p0} correspond to the Solis parameters `Dopant.Taun` and `Dopant.Taup`, and N_r corresponds to `Dopant.Nr`.

A dopant model can be defined with the possibility to set any doping profile, as detailed in section 2.4.

2.3.3.5 Traps

In addition to the continuous band tail distribution, using the Urbach model, Solis handles discrete trap levels in the layer bandgap. Up to five discrete trap levels can be defined. Each trap is defined as follows:

```
[Layer1]                              # First layer
                                      # ...
    Trap1.N=1e16                      # Trap density in 1/cm3
    Trap1.Nr=1e19                     # Reference density in 1/cm3
    Trap1.E=0.0                       # Activation energy (eV) with respect...
                                      # ... to the intrinsic Fermi level
    Trap1.Taun=1e-8                   # Electron lifetime in seconds
    Trap1.Taup=1e-8                   # Hole lifetime in seconds
    Trap1.Type="Donor"                # Trap type: "Donor" or "Acceptor"
```

Listing 2.6: First Layer Section: Traps.

Adding more traps is done in the same way, with the trap's number incremented: `Trap1`, `Trap2`, ..., `Trap5`.

The discrete trap parameters are defined in the same way than the dopant parameters in listing 2.5, except that the trap energy is defined relatively to the intrinsic Fermi level (close to the mid of the bandgap). The variation of the carrier lifetime with the trap density is defined as in equation 2.3.

User-defined trap model can be set with any profile, as detailed in section 2.4.

2.3.3.6 Lattice

Solis can include the effect of the spontaneous and piezoelectric polarization for III-N (mainly GaN, AlN, InN, AlGaN and InGaN) compound semiconductors.

The spontaneous and piezoelectric polarization is included in the Poisson equation (section 1.1, equation 1.1).

The lattice parameters used to define the total polarization are set as follows:

```
[Layer1]                          # First layer
                                  # ...
    Lattice.a=0.311               # Lattice a parameter in nm
    Lattice.c=0.498               # Lattice c parameter in nm
    Lattice.e31=-0.58             # Piezoelectric coefficient e31 in C/m2
    Lattice.e33=1.55              # Piezoelectric coefficient e33 in C/m2
    Lattice.c13=99                # Elastic coefficient c13 in GPa
    Lattice.c33=389               # Elastic coefficient c33 in GPa
    Lattice.Psp=-0.081            # spontaneous polarization coefficient in C/m2
```

Listing 2.7: First Layer Section: Lattice.

Where `Lattice.a` and `Lattice.c` are the lattice parameters; `Lattice.e31` and `Lattice.e33` the piezoelectric coefficients; `Lattice.c13` and `Lattice.c33` the elastic coefficients; P_{sp} the spontaneous polarization.

The total polarization P is the sum of spontaneous P_{sp} and piezoelectric P_{pi} polarization, calculated as follows, in the case of c-oriented III-N layers [8]:

$$\begin{aligned} P_{total} &= Polarization.Scale \times (P_{psp} + P_{pi}) \\ P_{pi} &= \sum_j e_{ij}\varepsilon_j = e_{33}\varepsilon_3 \ + \ e_{31}(\varepsilon_1 \ + \ \varepsilon_2) \\ \varepsilon_3 &= -2\frac{c_{13}}{c_{33}}\left(\frac{a-a_0}{a_0}\right) \quad ; \quad \varepsilon_1 = \varepsilon_2 = \frac{a-a_0}{a_0} \end{aligned} \tag{2.4}$$

a are the a-lattice parameter of the strained layer; a_0 that of the relaxed layer.

`Polarization.Scale` is a ponderation coefficient, defined in the `[Device]` section as follows:

```
[Device]
                                  # ...
    Polarization.Scale=1          # Spontaneous/piezoelectric polarization scale
```

The `Polarization.Scale` value varies between −10 and 10. For `Polarization.Scale = 0`, there is no spontaneous/piezoelectric polarization at all. For `Polarization.Scale = 10` (the maximum allowed value), the value is ten times higher than the value expected from the lattice parameters. To invert the polarization, just give a negative `Polarization.Scale` value. The `Polarization.Scale` parameter is useful to study the effect of the spontaneous/piezoelectric polarization on the device (e.g. solar cell or detector) performances. The lattice and polarization parameters can be user-defined using custom physical model, as detailed in section 2.4.

2.3.3.7 Mesh

The number of discretization (mesh) points in the layer can be set using:

```
[Layer1]                          # First layer
                                  # ...
    Mesh.Points=100               # The number of mesh points
```

Listing 2.8: First Layer Section: Mesh.

This parameter will be taken into account only if the `Mesh` key in the `Numeric` section is set to `"Layer"`, as explained in section 2.3.7 below. In this way, one can apply a mesh scheme on each layer to handle devices containing, for example, layers with very different thickness/doping/...

The number of mesh points should be set to a value between 5 and 5000. On the other hand, the total number of points in the whole device should between 50 and 20000.

2.3.4 Anode and Cathode Sections

Two contacts are defined for a given device: the anode (top contact) and the cathode (bottom contact). Each contact can be either ohmic or rectifying (Schottky) with a flexibility to define an intermediate ohmic-Schottky behavior using the barrier height.

The anode is defined as follows:

```
[Anode]                              # Anode section
    Anode.Type="Ohmic"               # Material name
    Anode.Phi=0.0                    # Barrier height (eV)
    Anode.Thickness=0.005            # Thickness in micrometers
    Anode.Rn=1e7                     # Recombination speed for electrons (m/s)
    Anode.Rp=1e7                     # Recombination speed for holes (m/s)
    Anode.Index.n=2.0                # Refractive index n
    Anode.Index.k=0.0                # Extinction coefficient k
```

Listing 2.9: Anode Section: Parameters.

And the cathode is defined is the same way, with different parameters if necessary:

```
[Cathode]                            # Cathode section
    Cathode.Type="Ohmic"             # Material name
    Cathode.Phi=0.0                  # Barrier height (eV)
    Cathode.Thickness=0.005          # Thickness in micrometers
    Cathode.Rn=1e7                   # Recombinaison speed for electrons (m/s)
    Cathode.Rp=1e7                   # Recombinaison speed for holes (m/s)
    Cathode.Index.n=2.0              # Refractive index n
    Cathode.Index.k=0.0              # Extinction coefficient k
```

Listing 2.10: Cathode Section: Parameters.

Recombination speed for electrons and holes can be set to model nonideal ohmic contact. The refractive index `n` and extinction coefficient `k` is defined to account for reflection and absorption in anode and cathode. A custom contact refractive index and extinction coefficient spectra can be defined (based, for example, on experimental data), as detailed in section 2.4.

2.3.5 Voltage Section

The voltage, applied between anode and cathode, is defined in the following listing:

```
[Voltage]                          # Voltage section
    Voltage.Source="Sweep"         # "Sweep", "None" or "CustomBias.txt"
    Voltage.Start=0.0              # Voltage start value in volts
    Voltage.End=1.0                # Voltage end value in volts
    Voltage.Step=0.01              # Voltage step value in volts
    Voltage.Jinf=-1.0              # Minimal current density in A/cm2
    Voltage.Jsup=1.0               # Maximal current density in A/cm2
```

Listing 2.11: Voltage Section: Parameters.

If the parameter `Voltage.Source` is set to `"None"`, the simulation will be performed at equilibrium.
If `Voltage.Source` is set to `"Sweep"`, the voltage will be increased from `Voltage.Start` to `Voltage.End` with a step equal to the value set in `Voltage.Step`.
The `Voltage.Source` can also be set using text file defined by the user. This file should contain the voltage values (one value per line) in an increasing order. In this way, one can completely adapt the voltage sweep to a specific device. The `custom_voltage.txt` file in the `examples` directory gives an example of custom voltage source.

`Voltage.Jinf` and `Voltage.Jsup` define a range in which the calculated current is constrained: simulation stops if the current reach one of those limits. These current constraints are very useful when simulating devices such as solar cells where the current varies between the short-circuit value and zero. Then for solar cells one can set `Voltage.Jsup` to `0.0` and set `Voltage.Jinf` to a value larger (in modulus) than the expected short-circuit current in A/cm^2, e.g. `Voltage.Jinf=-0.1` for a PN solar cell [3].

2.3.6 Light Section

A light source, as shown in figure 2.1, is defined in the following way:

```
[Light]                            # Light section
    Light.Source="AM1.5"           # "Mono", "AM1.5" or "None"
    # The following parameters are optional if Light.Source set to "AM1.5"
    Light.Start=0.4                # Light start wavelength in micrometers
    Light.End=1.2                  # Light end wavelength in micrometers
    Light.Step=0.01                # Light step wavelength in micrometers
    Light.Flux=1e17                # Light flux in 1/cm2/s
```

Listing 2.12: Light Section: Parameters.

If `Light.Source` is set to `"AM1.5"`, Solis uses the standard solar spectrum AM1.5G as defined in [9] and shown in figure 2.2. This solar spectrum is saved in the file `solar_spectrum_am15g.txt` located in the `examples` directory. In this case, it is not necessary to specify the other parameters. If these parameters are specified in this case, only the solar spectrum range from `Light.Start` to `Light.End` will be considered.

[3] for NP solar cell, just invert the values: `Voltage.Jinf=0` and `Voltage.Jsup=0.1`

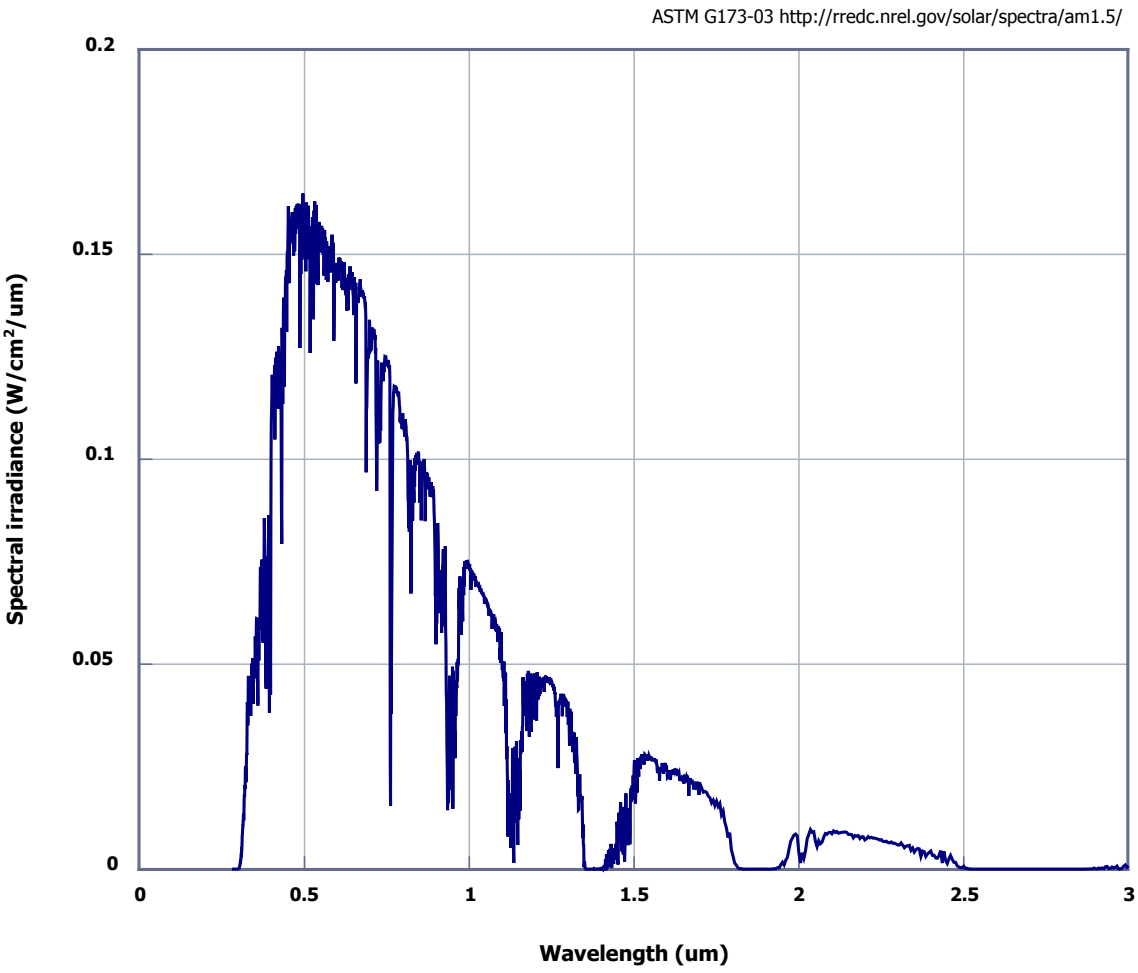

Figure 2.2: Standard AM1.5G solar spectrum [9].

`Light.Source` can also be set to `"Mono"` to define a monochromatic light at the wavelength defined by `Light.Start` (in um) and a flux defined by `Light.Flux` (in $1/cm^2/s$).

Set `Light.Source` to `"EQE"` to simulate the quantum efficiency spectrum from `Light.Start` to `Light.End` using `Light.Step`.

If `Light.Source` is set to `"None"`, no light source is considered in the simulation.

As for voltage, the `Light.Source` can be set using text file defined by the user. This file contains the wavelength (in um) and flux (in $1/cm^2/s/um$) values (two values per line, separated by tabulation) in an increasing wavelength order. The `custom_spectrum.txt` file in the `examples` directory is an example of such a custom light source.

2.3.7 Numeric Section

Some Solis numerical parameters, mainly the output options, solver method, tolerance, damping factor, mesh, initial guess, can be defined to meet the user needs.

These parameters can be defined in this section as in listing 2.13.

"bicgstab"	biconjugate gradient stabilized method
"bicgstabl	biconjugate gradient stabilized method (l)
"bicgsafe"	biconjugate gradient (safety variant) method
"gmres"	generalized minimal residual method
"fgmres"	flexible generalized minimum residual method
"minres"	minimum residual method
"tfqmr"	transpose-free quasi-minimal residual method

Table 2.1: The iterative linear solvers used by Solis.

```
[Numeric]                                # Numeric section
    Numeric.Verbose="Yes"                # Print out all messages: "Yes", "No"
    Numeric.SaveFull="No"                # Save all output data: "Yes", "No"
    Numeric.Method="Gummel"              # Set the method: "Newton", "Gummel"
    Numeric.InnermostSolver="tfqmr"      # Set the innermost solver
    Numeric.Tol=1e-4                     # Tolerance to reach
    Numeric.Damp=0.2                     # Damping factor
    Numeric.Mesh="Layer"                 # Mesh: "Layer", "Uniform" or text file
    Numeric.Points=200                   # The device mesh number of points
    Numeric.Guess="custom_guess.txt"     # The user-defined initial guess
```

Listing 2.13: Numeric Section: Parameters.

The `Numeric.Verbose` parameter can be set to `"No"` to print out only core messages, or to `"Yes"` to print out all simulator messages.

If `Numeric.SaveFull` is set to `"No"`, only current-voltage, capacitance-voltage or spectral response are saved with the equilibrium and last calculated spatial distribution of potential, field, carrier concentrations, recombination rates, band diagram, etc. If set to `"Yes"`, all the output data are saved for every voltage and wavelength, increasing accordingly the disk usage and slowing down the simulation.

With `Numeric.Method`, one can set the algorithm to `"Gummel"` or `"Newton"`. These methods are previously presented in section 1.1. `Numeric.InnermostSolver` set the innermost solver for the linear system arisen from the discretization of the drift-diffusion system, as explained is section 1.1. `Numeric.InnermostSolver` can be set to one of the solvers listed in table 2.1.

A comprehensive description and implementation of these iterative linear solvers can be found in [10, 11]. Solis uses `tfqmr` by default.

The `Numeric.Tol` parameter is set to the solver tolerance. Smaller is `Numeric.Tol`, the more precise is the solution and slower is the simulation, obviously. In fact, it is usually unnecessary to set the tolerance to a small value (less than 10^{-5}) since when the convergence is achieved, the obtained error is far smaller than the given tolerance. Solis can automatically change the given tolerance to seek for a faster convergence.

The `Numeric.Damp` parameter is set to the internal damping factor used to ensure convergence of the Gummel or Newton methods, as presented in section 1.1. A value of 0.2 means that the maximum allowed variation of potential and Fermi levels is $\frac{1}{0.2} = 5$ in unit of kT. This value plays an important role in the convergence and can be automatically adapted by Solis to enhance convergence in some situations. Usually it is not necessary to manually set it, except when the convergence fails.

If `Numeric.Mesh` is set to `"Uniform"`, a uniform discretization will be applied to the whole device, using `Numeric.Points` that is the mesh number of points to set from `50` to `20000`. The uniform mesh is the mesh scheme applied to devices where the thicknesses (and/or other parameters such as doping) are comparable. When the device contains layers with very different thicknesses (such as solar cell with a very thin highly doped emitter and thick absorber), the uniform mesh is no longer adapted since it implies a huge increase in the number of points (to have sufficient points in the thinnest layer) that slows down the simulation. In this case, set `Numeric.Mesh` to `"Layer"` and, in each layer section, set `Mesh.Points` to a value adapted to the layer thickness. User can also define a custom mesh scheme, by setting `Numeric.Mesh` to a text file. This file should contain the position values (in micrometers, one value per line) in an increasing order from `0` to the total thickness of the device. Using a custom mesh allows the user to define an adaptive discretization scheme with, for example, small step at the interfaces (or in highly doped regions, etc.) and larger step elsewhere.

User can also give a custom initial guess to the Solis engine. Solis internally guess the initial potential profile at equilibrium to give a starting solution to the solver to ensure and speed up convergence in most cases. The internal Solis procedure can be replaced if the user set the `Numeric.Guess` parameter to a text file containing the potential and Fermi levels (for electrons and holes). This file should contain four values per line, separated by tabulation: position in micrometers, potential in V, Fermi levels for electrons and holes in eV. The position, in increasing order, varies from from `0` to the total thickness of the device. Listing 2.14 gives an example of such a file. The initial guess is given at equilibrium and the Fermi levels can be set to zero (Solis uses the Fermi level at equilibrium as the energy origin). Giving a good initial guess speed up the simulation, particularly for complex optimization procedure.

```
# Initial guess for Solis
# z(um) V(V)                Efn(eV) Efp(eV)
0       -0.306262653854     0       0
0.01    -0.306262653854     0       0
0.02    -0.30625226779      0       0
0.03    -0.306231438238     0       0
# ....
# ....
1.97     0.30629510655      0       0
1.98     0.306308626896     0       0
2        0.306308626896     0       0
```

Listing 2.14: Example of initial guess.

One can use a solution previously found by Solis as a starting initial guess to simulate a similar structure. A strategy well suited to a particular device can be found and implemented using, for example, the embedded Lua scripting engine or any other language.

2.3.8 Command Section

This section contains some Solis general parameters:

```
[Command]                              # Command section
    Command.OutDir="/tmp/"             # The output directory
                                       # where Solis saves the output data
    Command.OutDirDate="No"            # Append date to the output directory
    Command.Plot="Yes"                 # Set to "Yes" to plot output data
                                       # using the SolisPlot tool
```

Listing 2.15: Command Section: Parameters.

With `Command.OutDir`, one can set an output directory where Solis saves output data. If set to nothing (`""`) or not defined at all, Solis create, in the working directory (where the current input file is), an output directory named `simulout` and a subdirectory using the input name. Section 2.5 gives details about the Solis output files. If the `Command.OutDirDate` is set to `"Yes"`, the current date and time is used for the output directory name, to preserve output files between simulations of the same device.

One can set `Command.Plot` to `"Yes"` to plot output data (current-voltage, capacitance-voltage or spectral response characteristic, spatial distribution of potential, field, carrier concentrations, recombination rates, band diagram, etc.) using the SolisPlot tool.

2.4 Define Physical Models using the Lua programming language

One of the strengths of Solis is the use of an embedded scripting language, letting the user customize/define the physical models and input parameters. This flexibility is of great importance in order to simulate precisely and realistically semiconductor devices. The Lua programming language was choosen mainly because of his simplicity, lightness, small footprint and speed. A very good tutorial, written by one of the main Lua developers, can be found here: https://www.lua.org/pil/contents.html.

Model templates are integrated in the Solis code editor and included in the `examples` directory. These templates define the:

- bandgap model in a layer section (`[Layer1]` to `[Layer9]`)
- permittivity model in a layer section
- doping model in a layer section
- trap model in a layer section

- mobility model in a layer section
- recombination model in a layer section
- lattice model in a layer section
- refractive index n and extinction coefficient k model in a layer, anode and cathode sections

Each model is defined by a Lua function with a given mandatory name and list of arguments. The function prototype (name, list of arguments and returned values) is fixed but the function content is, of course, completely user-defined. All functions related to a given device could be defined in one Lua file per layer or separated, with one file per function. It depends only on the user preference.

2.4.1 Bandgap Model

This model is defined as in the listing 2.1 and partially reproduced below:

```
[Layer1]                              # First layer
                                      # ...
                                      # ...
    Bandgap.Model="bandgap.lua"       # Bandgap model using Lua
                                      # ...
                                      # ...
```

Listing 2.16: Bandgap Model Definition in the Layer Section.

`bandgap.lua` is the Lua source file containing the model function, always named `sol_bandgap`. The following code gives an example of such a Lua function:

```
function sol_bandgap(position, thickness, temperature, concentration)
    eg = 1.12        -- bandgap (eV)
    chi = 4.17       -- affinity (eV)
    nc = 2.8e19      -- density of states in conduction band in 1/cm3
    nv = 1.04e19     -- density of states in valence band in 1/cm3
    graded = false
    -- ...
    -- ...
    return eg, chi, nc, nv, graded, true
end
```

Listing 2.17: Bandgap Model Lua Function.

The `sol_bandgap` takes as input the position (in micrometers) in the layer (from 0 to the layer thickness); the layer thickness (in micrometers); the temperature in Kelvin and the doping concentration (in cm^{-3}). The function must return four parameters: the bandgap value (in eV); the affinity (in eV); the density of states respectively in conduction and valence band (in cm^{-3}). All these parameters can be graded i.e. depending on position inside the layer, giving a very flexible way to study heterostructures with non-homogeneous properties. Only these four parameters are always returned. The temperature and doping concentration given as input

parameters permit implementing physical models such as bandgap narrowing, Burstein-Moss shift and/or Varshni relation.

If the parameters are **graded**, the function must return the fifth parameter, a boolean, set to true. It is important to return this boolean to avoid unnecessary calls to the function and then speed up the simulation. By default, if the graded parameter is not returned, the layer is considered to be not graded (thus homogeneous) and the function will not be called again for the next position in the layer. *The graded parameter is always returned as the penultimate parameter for all Solis Lua functions.*

The last (sixth) returned parameter is the boolean **status**: if false, the simulator will stop the calculation. This value is usually set to true, but can be set to false to abort the simulation process in some circumstances. *The status is always returned as the last parameter for all Solis Lua functions.*

2.4.2 Permittivity Model

As for the bandgap, the permittivity model is defined as below:

```
[Layer1]                              # First layer
                                      # ...
    Permittivity.Model="perm.lua"     # Permittivity model using Lua
```

Listing 2.18: Permittivity Model Definition in the Layer Section.

In this Solis input, `perm.lua` is the Lua source file containing the model function, always named `sol_permittivity`, as given in the following code:

```
function sol_permittivity(position, thickness, temperature)
    eps = 11.8        -- relative permittivity
    graded = false
    -- ...
    return eps, graded, true
end
```

Listing 2.19: Permittivity Model Lua Function.

The `sol_permittivity` takes as input the position and layer thickness (in micrometers) and the temperature (in Kelvin). It returns the relative permittivity and, as for the bandgap function presented in 2.4.1, the (optional) graded and status parameters. Note that if the bandgap and related parameters are graded, the permittivity is usually graded also to keep the physical consistency.

2.4.3 Doping Model

Solis can handle custom doping profile. With the Lua model, one can code any varying doping parameters in a given layer and not only a constant doping concentration as usually done in the semiconductor device simulators. In solar cells, for example, one can study the effect of a doping gradient in the absorber.

The dopant model is defined in the Solis input as follows:

```
[Layer1]                              # First layer
                                      # ...
    Dopant.Model="doping.lua"         # Dopant model using Lua
```

Listing 2.20: Doping Model Definition in the Layer Section.

`doping.lua` is the Lua source file containing the dopant model function, always named `sol_doping`, as given in the following code:

```
function sol_doping(position, thickness)
    Na = 0              -- acceptor density (1/cm3)
    Nd = 1e16           -- donor density (1/cm3)
    Nr = 1e19           -- reference density (1/cm3)
    Taun = 1e-6         -- electron lifetime (seconds)
    Taup = 1e-6         -- hole lifetime (seconds)
    E = 0               -- activation energy (eV)
    F = 1               -- degeneracy factor
    graded = false
    -- ...
    return Na, Nd, Nr, Taun, Taup, E, F, graded, true
end
```

Listing 2.21: Doping Model Lua Function.

The `sol_doping` takes as input the position in the layer in micrometers, the layer thickness, and returns the doping parameters: acceptor density (in cm^{-3}), donor density (in cm^{-3}), reference density (in cm^{-3}), electron lifetime (in s), hole lifetime (in s), activation energy (in eV) and degeneracy factor. These parameters are already described in 2.3.3.4. The last two returned parameters are the graded and status parameters, as already defined in 2.4.1.

The following code gives an example on how to define a Gaussian p-doping profile in a layer with a background n-doping:

```
function sol_doping(position, thickness)
    Na = 1e18           -- acceptor density at the layer surface (1/cm3)
    Nd = 1e16           -- background donor density (1/cm3)
    Nr = 1e19           -- reference density (1/cm3)
    Taun = 1e-6         -- electron lifetime (seconds)
    Taup = 1e-6         -- hole lifetime (seconds)
    E = 0               -- activation energy (eV)
    F = 1               -- degeneracy factor
    graded = true       -- set graded to true in this case

    Na = Na * math.exp(-position * position / 0.25)

    return Na, Nd, Nr, Taun, Taup, E, F, graded, true
end
```

Listing 2.22: Doping Model Lua Function: Example.

One can also give the simulator an experimental doping profile as measured, e.g., by the C-V (capacitance-voltage) technique.

2.4.4 Trap Model

As for doping, Solis can handle custom trap parameters to simulate a spatially nonuniform trap distribution. The trap model is defined in the Solis input as follows:

```
[Layer1]                              # First layer
                                      # ...

    Trap.Count=1                      # The total number of traps
    Trap.Model="trap.lua"             # Trap model using Lua

                                      # ...
```

Listing 2.23: Trap Model Definition in the Layer Section.

The `Trap.Count` parameter is mandatory when using Lua to define traps. It is the total number of traps (between 1 and 5).

`trap.lua` is the Lua source file containing the trap model function, named `sol_trap`, with the following prototype:

```
function sol_trap(number, position, thickness)
    Na = 1e15         -- density for acceptor-like trap (1/cm3)
    Nd = 0            -- density for donor-like trap (1/cm3)
    Nr = 1e19         -- reference density (1/cm3)
    Taun = 1e-8       -- electron lifetime (seconds)
    Taup = 1e-8       -- hole lifetime (seconds)
    E = 0             -- trap energy relatively to Ei (eV)
    F = 1             -- degeneracy factor
    graded = false
    -- ...
    -- ...
    return Na, Nd, Nr, Taun, Taup, E, F, graded, true
end
```

Listing 2.24: Trap Model Lua Function.

The `sol_trap` takes as input `number`, the trap number (from 1 to N where N is the total number of traps in the layer), the position and layer thickness in micrometers.

The function returns the trap parameters: acceptor density, donor density, reference density, electron lifetime, hole lifetime, activation energy, and degeneracy factor. These parameters are already described in 2.3.3.5. The fact that the activation energy is defined relatively to the intrinsic Fermi level. The last two returned parameters are the graded and status parameters, as already defined in 2.4.1.

In the following example, one trap is defined with an exponentially decreasing density:

```
function sol_trap(number, position, thickness)
    Na = 1e17         -- density for acceptor-like trap (1/cm3)
    Nd = 0            -- density for donor-like trap (1/cm3)
    Nr = 1e19         -- reference density (1/cm3)
    Taun = 1e-8       -- electron lifetime (seconds)
    Taup = 1e-8       -- hole lifetime (seconds)
    E = 0             -- trap energy relatively to Ei (eV)
    F = 1             -- degeneracy factor
    graded = true

    Na = Na * math.exp(-position / 0.5)

    return Na, Nd, Nr, Taun, Taup, E, F, graded, true
end
```

Listing 2.25: Trap Model Lua Function: Example.

2.4.5 Mobility Model

The mobility model is defined in the Solis input as follows:

```
[Layer1]                              # First layer
                                      # ...

    Mobility.Model="mobility.lua"     # Mobility model using Lua

                                      # ...
```

Listing 2.26: Mobility Model Definition in the Layer Section.

`mobility.lua` is the Lua source file containing the mobility model function `sol_mobility`, defined as follows:

```
function sol_mobility(position, thickness, temperature, concentration)
    mun = 1500        -- electron mobility in cm2/Vs
    mup = 500         -- hole mobility in cm2/Vs
    graded = false
    -- ...
    -- ...
    return mun, mup, graded, true
end
```

Listing 2.27: Mobility Model Lua Function.

The `sol_mobility` takes as input the position (in micrometers) in the layer; the layer thickness (in micrometers); the temperature in Kelvin and the doping concentration (in cm^{-3}).

The function returns the electrons and holes mobility. The last two returned parameters are the graded and status parameters, as already defined in 2.4.1.

Models such as Caughey-Thomas can be easily implemented in `sol_mobility`.

2.4.6 Recombination Model

The recombination parameters, used by Solis to implement the Auger, direct and Shockley-Read-Hall mechanisms, can be user-defined in the Solis input as follows:

```
[Layer1]                                          # First layer
                                                  # ...

    Recombination.Model="recombination.lua"       # Recombination model using Lua

                                                  # ...
```

Listing 2.28: Recombination Model Definition in the Layer Section.

`recombination.lua` is the Lua source file containing `sol_recombination`, the recombination function, defined as follows:

```
function sol_recombination(position, thickness)
    Beta = 1.1e-14          -- Radiative recombination coefficient (cm3/s)
    Cn = 8.3e-32            -- Auger recombination constant for electrons (cm6/s)
    Cp = 1.8e-31            -- Auger recombination constant for holes (cm6/s)
    Taun = 1e-7             -- Electron lifetime (seconds)
    Taup = 1e-7             -- Hole lifetime (seconds)
    graded = false
    --
    --
    return Beta, Cn, Cp, Taun, Taup, graded, true
end
```

Listing 2.29: Recombination Model Lua Function.

The `sol_recombination` takes as input the position and the layer thickness (in micrometers). It returns the radiative (direct) recombination coefficient (in cm^3/s); Auger recombination constants (in cm^6/s); electron and hole lifetime (in s). The last two returned parameters are the graded and status parameters, as already defined in 2.4.1.

2.4.7 Lattice Model

The lattice parameters can be defined in the layer section as follows:

```
[Layer1]                                      # First layer
                                              # ...

    Lattice.Model="lattice.lua"               # Lattice model using Lua

                                              # ...
```

Listing 2.30: Lattice Model Definition in the Layer Section.

`lattice.lua` is the Lua source file containing the lattice function `sol_lattice`, defined as follows:

```
function sol_lattice(position, thickness)
    a    =  0.3189          -- Lattice a parameter (nm)
    e31  = -0.34            -- Piezoelectric parameter (C/m2)
    e33  =  0.67            -- Piezoelectric parameter (C/m2)
    c13  = 100              -- Elastic parameter (GPa)
    c33  = 392              -- Elastic parameter (GPa)
    Psp  = -0.03            -- Spontaneous polarization (C/m2)
    graded = false
    --
    --
    return a, e31, e33, c13, c33, Psp, graded, true
end
```

Listing 2.31: Lattice Model Lua Function.

The `sol_lattice` takes as input the position and the layer thickness (in micrometers). It returns the a-lattice, piezoelectric and elastic, spontaneous polarization parameters. These parameters are already presented in section 2.3.3.6. The last two returned parameters are the graded and status parameters, as already defined in 2.4.1.

2.4.8 Refractive Index and Extinction Coefficient Model

The refractive index `n` and the extinction coefficient `k` can be defined in Lua to reflect the user custom data.

The refractive index model is defined in the Solis input as follows:

```
[Layer1]                            # First layer
                                    # ...
    Index.Model="index.lua"         # Refractive index model using Lua
                                    # ...
```

Listing 2.32: Refractive Index Model Definition in the Layer Section.

`index.lua` is the Lua source file containing the refractive index model function, always named `sol_index`, with the following prototype:

```
function sol_index(position, thickness, bandgap, lambda)
    n = 1
    k = 0
    graded = false
    -- ...
    return n, k, graded, true
end
```

Listing 2.33: Refractive Index Model Lua Function.

The `sol_index` input parameters are the position and the layer thickness (in micrometers), the bandgap (in eV) and the light wavelength (in micrometers). It returns the refractive index `n` and the extinction coefficient `k`. The last two returned parameters are the graded and status parameters, as already defined in 2.4.1.

2.5 Solis Output Files

As specified in 2.3.8, one can set `Command.OutDir` parameter to an output directory where Solis saves the simulation results, and set `Command.OutDirDate` to `"Yes"` to use the current date and time for the output directory name.

By default, Solis create, in the working directory (where the current input file is located), an output directory named `simulout` and a subdirectory using the input file name and stores the following files:

- `simuldevice.txt` contains a synthetic description of the simulated device as in the following example:

```
# Solis <DE>
# The above first line is mandatory
#
# c-Si Solar Cell
#
Device 'c-Si Solar Cell': 2 layers:
   Layer1:
     t = 1 ; NLx = 1e+020 ; Eg = 1.12 ; un = 500 ; up = 500 ; N
   Layer2:
     t = 200 ; NLx = 1e+017 ; Eg = 1.12 ; un = 500 ; up = 500 ; P
```

- `simulog.txt` contains simulator internal messages for debugging and testing purposes.

- `simulout_jv.txt` contains the current-voltage characteristic, if any. Its content looks like:

```
# Solis <JV>
# The above first line is mandatory
#
# c-Si Solar Cell
#
# Voltage = 0 V ; Current = 0.0115406 A/cm2 ;
#
# Voltage(V)    J(A/cm2)
#
-0.69        -0.0161024
-0.68        -0.00206067
-0.67        0.00749823
-0.66        0.0140025
-0.65        0.0184273
-0.64        0.021437
#...
```

  The voltage and current values are separated with a tabulation. Similarly, `simulout_cv.txt` contains the capacitance-voltage characteristic.

- `simulout_qw.txt` contains the external and internal quantum efficiency spectra, if any. Its content looks like:

```
# Solis <QW>
# The above first line is mandatory
#
# GaN Detector
#
# Voltage = 0 V ; Current = -2.89747e-010 A/cm2 ;
#
# Wavelength(um)    EQE             IQE
#
0.202           0.624345        0.624345
0.204           0.63732         0.63732
0.206           0.650023        0.650023
0.208           0.662497        0.662497
0.21            0.674743        0.674743
0.212           0.686753        0.686753
0.214           0.698521        0.698521
0.216           0.710042        0.710042
0.218           0.721311        0.721311
0.22            0.732321        0.732321
0.222           0.743071        0.743071
#...
```

  The wavelength and efficiencies values are separated with a tabulation.

- `simulout_dist_NNN.txt` files, where `NNN` is a number, contain, for each voltage (if current-voltage characteristic) or wavelength (if quantum efficiency spectra), the spatial distribution of potential, electric field, Fermi levels, conduction and valence band, carrier concentration, doping and traps concentration, currents (electrons, holes and total), photogeneration and recombination (Auger, direct and SRH). These files are not saved for every voltage (or wavelength) if the parameter `Numeric.SaveFull` is set to `"No"`. In this latter case, those file are saved only at equilibrium (`simulout_dist_001.txt`) and for the last applied voltage (or wavelength).

- Other files such as `simulplot.txt` are used by the Solis tools.

Alongside with the simulation results, Solis saves also the simulation input `simulinput.solis` and related Lua models, if any, in the output directory. This permits to keep the input code and models used to produce the results with the results, for easier analysis and archiving.

2.6 Materials Parameters

Solis includes a materials database with a set of the most used semiconductors, particularly common in photovoltaics and optoelectronics.

All the materials properties are extracted from published experimental data and are considered to be highly reliable. The Madelung's handbook is one of the main references in this matter [12].

Table 2.2 gives the main parameters of the included semiconductors. The other parameters, recombination (Auger, direct, SRH), Urbach, Bandgap narrowing, refractive index and extinction coefficient spectra, mobility... are taken from well-established experimental data [13, 12, 14, 15].

In addition, Solis includes alloys such as Si_xGe_{1-x}, $In_xGa_{1-x}As$, $In_xGa_{1-x}P$, $Al_xGa_{1-x}As$, $Al_xGa_{1-x}N$, $In_xGa_{1-x}N$. The composition x, the bandgap bowing and the affinity bowing for these alloys are defined in the corresponding layer section, as shown in 2.3.3: `Composition.x`, `Bandgap.Bowing`, `Affinity.Bowing`. The alloy parameters are deduced from the two binaries (or elements for Si_xGe_{1-x}) using the Vegard law (modified to include the bandgap and affinity bowing factors).

	$E_G(eV)$	$\chi(eV)$	$N_C(cm^{-3})$	$N_V(cm^{-3})$	ϵ_r	$\beta(cm^3/s)$
Ge	0.66	4.0	10^{19}	5×10^{18}	16.2	6.41×10^{-14}
Si	1.12	4.17	2.8×10^{19}	1.04×10^{19}	11.8	1.1×10^{-14}
$a-Si$	1.8	3.9	2×10^{20}	2×10^{20}	11.9	1.1×10^{-14}
$GaAs$	1.42	4.07	4.7×10^{17}	9×10^{18}	12.9	7.2×10^{-10}
$InAs$	0.354	4.9	8.7×10^{16}	6.6×10^{18}	15.15	10^{-14}
InP	1.344	4.38	5.7×10^{17}	1.1×10^{19}	12.5	10^{-14}
$InSb$	0.17	4.59	4.2×10^{16}	7.3×10^{18}	16.8	5×10^{-11}
$GaSb$	0.726	4.06	2.1×10^{17}	1.8×10^{19}	12.5	10^{-10}
C $(Diamond)$	5.5	7.2	10^{20}	10^{19}	5.5	10^{-10}
AlN	6.2	2.26	6.3×10^{18}	4.8×10^{20}	9.14	5×10^{-11}
GaN	3.4	4.1	1.2×10^{18}	4.1×10^{19}	8.9	1.1×10^{-8}
InN	0.7	5.26	9×10^{17}	5.3×10^{19}	15.3	2×10^{-10}
$6H-SiC$	3.0	3.4	9×10^{18}	2.5×10^{19}	9.66	1.5×10^{-12}
$CIGS$	1.15	4.2	2.2×10^{18}	1.8×10^{19}	13.4	10^{-12}
$CZTS$	1.5	4.5	2.1×10^{18}	8.9×10^{18}	6.5	10^{-12}
CdS	2.42	4.2	1.8×10^{19}	2.4×10^{18}	9	10^{-12}
In_2S_3	2.9	4.3	2.2×10^{18}	1.8×10^{19}	13.5	10^{-12}
$CdTe$	1.47	4.4	2.2×10^{18}	1.8×10^{19}	10.2	10^{-12}
ZnO	3.3	4.4	2.2×10^{18}	1.8×10^{19}	9	10^{-12}

Table 2.2: Basic parameters of semiconductors included in the Solis database. E_G is the bandgap, χ the electronic affinity, N_C and N_V the density of states in conduction and valence band respectively, ϵ_r the relative permittivity, and β the direct recombination parameter. The experimental data are taken from [13, 12, 14, 15].

Chapter 3
Solis Tools

Contents

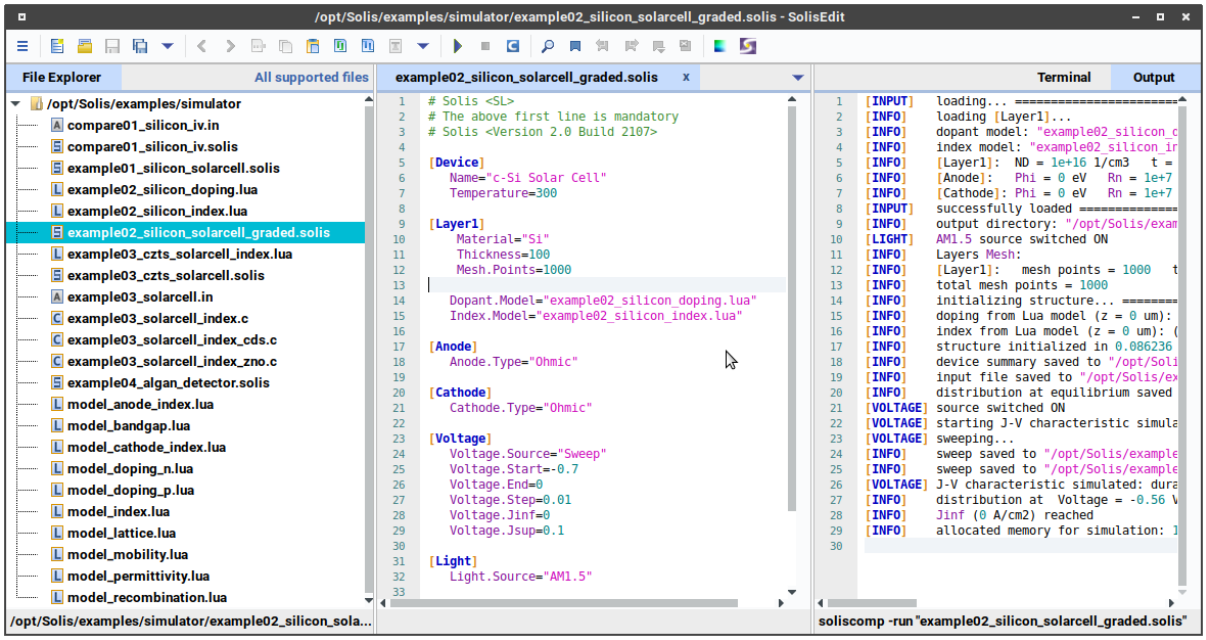

Figure 3.1: SolisEdit Screenshot.

Four independent tools are distributed with Solis:

- A code editor, **solisedit.exe** (or **solisedit** under Linux).
- A graphical device editor, **solisdevice.exe** (or **solisdevice** under Linux).
- A data plotter, **solisplot.exe** (or **solisplot** under Linux).
- An scientific calculator, **soliscalc.exe** (or **soliscalc** under Linux).

In addition, Solis includes an interactive terminal emulator (**solisterm**, only under Linux), a standalone version of the embedded terminal in SolisEdit.

To know if a new version is available, click *Menu/Help/Check for Update...* or visit my website: http://www.hamady.org

3.1 Code Editor

The Solis code editor, `SolisEdit`, offers all functionality found in modern editors such as syntax highlighting, autocompletion, markers, indentation control, find/replace, file explorer... and is fully customizable (screenshot in figure 3.1).

`SolisEdit` can be used to edit the Solis input files (with extension **.solis**) and model files (with extension **.lua**) and, in addition, it supports a set of languages used by scientists and engineers such as C/C++, Bash, Python, Octave, Fortran, LaTeX[1] and Makefile. The language

[1]This manual was composed in LaTeXusing `SolisEdit`.

is automatically selected based on the file extension.

`SolisEdit` integrates a `File Explorer` to work easily with files, that can be set to show only (filter) some files based on their extension (right-click the root directory and select the corresponding filter).

`SolisEdit` includes a system of coloured `Markers`, shown in the margins, to show in realtime the modified/saved sections of the current documents. The `Markers` can be reset at any time by selecting *Edit/Remove* Markers menu.
One particular marker is the bookmark, used to mark a specific line in the code for easier navigation.

The classic `Search/Replace` functionality are included in `SolisEdit`: *Search* menu.
Almost every aspect of the `SolisEdit` user interface can be customized: *Options* menu.

With `SolisEdit`, one can configure specific tool, such as a compiler or a Bash script, to run for a known file type (Solis and Lua of course, but also C, Python, LaTeX, etc.). To configure a tool, select *Tools/Compiler* menu and type the command to use to build/compile the corresponding file type. For example, for Solis, the command is `solis -run %s.solis` where `%s` will be replaced by the filename. It is useful to call a batch file (under Windows) or bash script (under Linux) to build a specific file. This can be done in the following way:

- For Windows, one can create a batch file (named `build.bat` for example), put the command in this file and add the batch name `build.bat` in the *Tools/Compiler* dialog after selecting the corresponding file type (example: LaTeX). The following example gives a typical `build.bat` content for building LaTeXdocument under Windows using MiKTeX and the Sumatra PDF viewer:

```
@echo off
@del /f /q myreport.pdf >nul 2>&1
latexmk -pdf -pvc- -halt-on-error myreport.tex
if %errorlevel% equ 0 (
    start "" "SumatraPDF.exe" myreport.pdf
) else (exit /b 1)
```

Listing 3.1: build.bat file to compile LaTeXdocument under Windows.

- For Linux, similarly you can create a Bash file (named `build.sh` for example), put the command in this file and add `./build.sh` in the *Tools/Compiler* dialog after selecting the corresponding file type (example: LaTeX).

- For Python (for both Windows or Linux), just put a command like this (replace with your installed Python interpreter):
  `python -u %s.py`

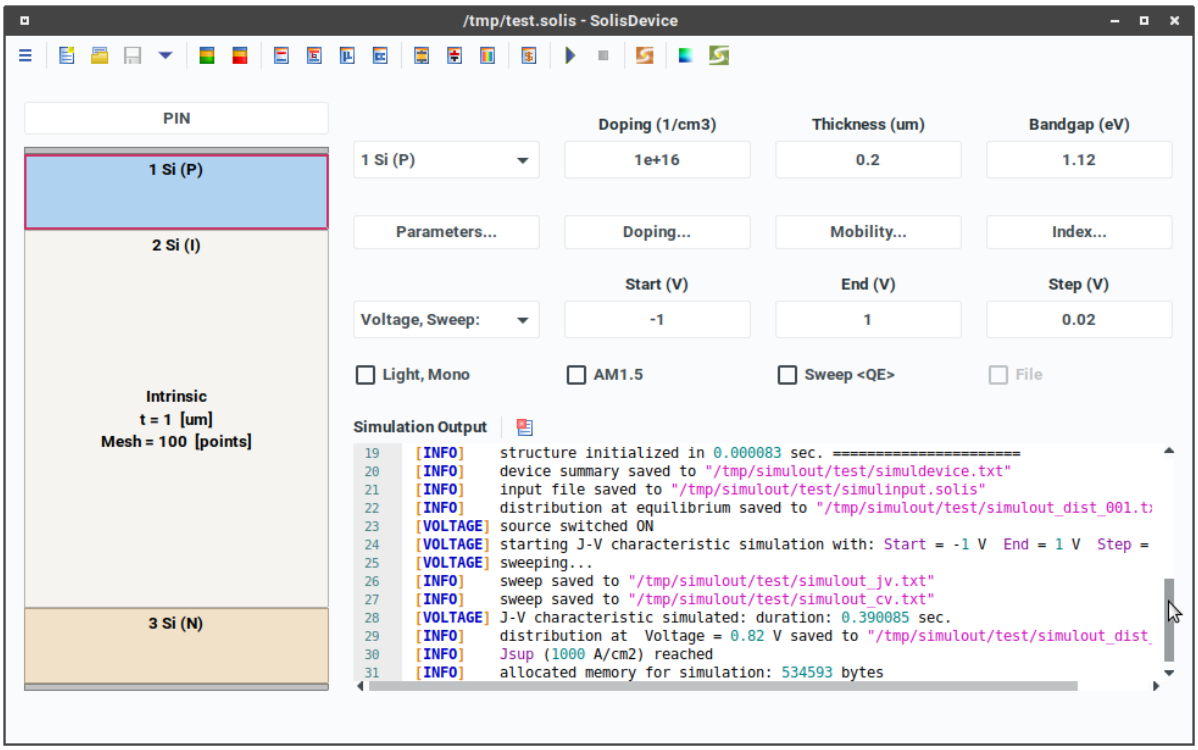

Figure 3.2: SolisDevice Screenshot.

In the *Tools/Compiler* dialog, one can check the *Redirect standard output* option to let `SolisEdit` to print out all the text generated by the tool to the `Output Window`. If this option is unchecked, `SolisEdit` will launch a command window and run the tool inside it. In this latter case, one can check the *Close when restart* option to close the previous command window before starting a new one. For the Solis simulator it is better to check the *Redirect standard output* option (it is checked, by default) to benefit from functionality such as syntax coloring. Under Linux, SolisEdit includes an embedded `terminal` emulator. This terminal emulator is loaded and available to use if the VTE library is installed. Usually the required VTE library is installed by default, but in some systems (CentOS 7 for example), it must be installed as follows (for CentOS):

```
sudo yum install -y epel-release
sudo yum install vte
```

3.2 Graphical Device Editor

`SolisDevice` is the Solis graphical device editor to build graphically the device to simulate. It offers a very easy way to perform simulations with a simple and functional user interface (screenshot in figure 3.2). It generates the same input file than entered with `SolisEdit`. An input graphically created with `SolisDevice` can be edited with `SolisEdit` and vice-versa.

`SolisDevice` functionality are accessible with the *menu*, *toolbar* or by right-clicking or move

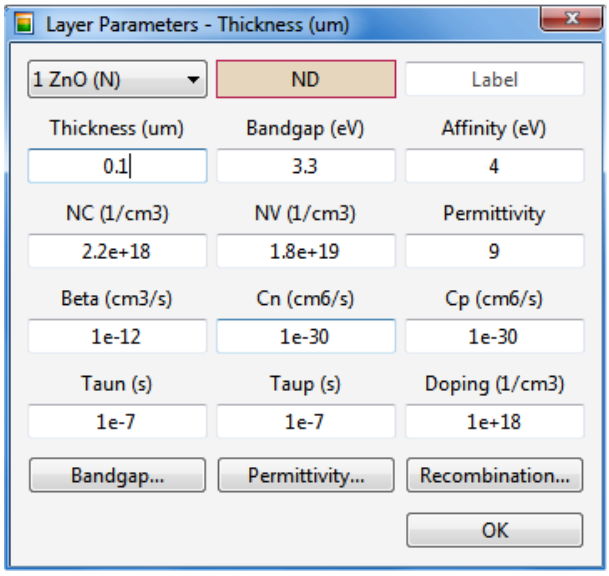

Figure 3.3: SolisDevice: Parameters Dialog.

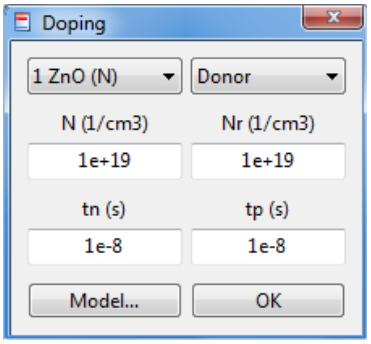

Figure 3.4: SolisDevice: Doping Dialog.

(e.g. resizing a layer).

With `SolisDevice` one can add/move/resize/delete a layer and set the **material parameters**: *Device* menu or the linked toolbar buttons. In the layer parameters dialog (shown by double-clicking the layer or pressing the *Parameters* button), one can enter the semiconductor material properties: bandgap, affinity, permittivity, direct, Auger and SRH recombination, etc.

The layer **material** can be set using the integrated Solis database: *Device/Material...* menu. In this dialog, one can also select Lua model files for bandgap and permittivity.

The **layer doping** parameters can be set using the corresponding dialog (figure 3.4) shown by pressing the *Doping...* button. These parameters are already presented in section 2.3.3.4 and correspond to the dopant type and concentration, the reference concentration, the electron and hole lifetime. The doping Lua model file can also be selected in this dialog (refer to section 2.4 for detail in implementing Lua models for Solis).

Traps can be added/deleted/modified using the corresponding dialog shown by selecting the *Device/Traps...* menu (figure 3.5). As for doping, these parameters are already presented in

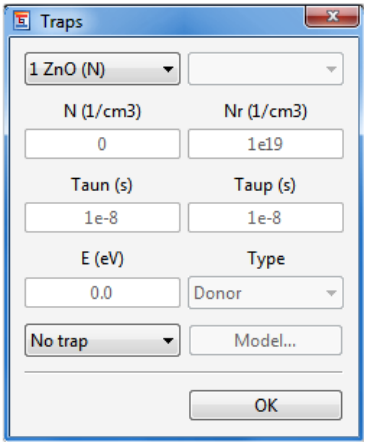

Figure 3.5: SolisDevice: Traps Dialog.

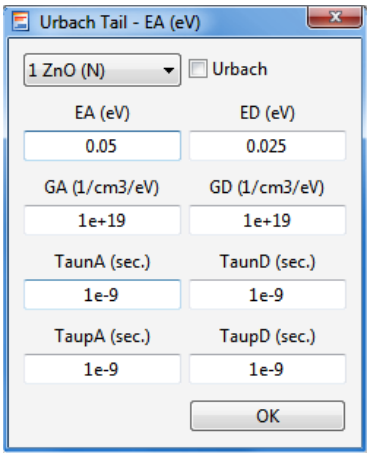

Figure 3.6: SolisDevice: Urbach Tail Dialog.

section 2.3.3.5 and correspond to the defect type and concentration, the reference concentration, the activation energy, the electron and hole lifetime, trap degeneracy. A Lua model file for traps can also be selected in this dialog, as detailed in section 2.4.

Urbach parameters, as defined is 2.3.3.1, can be set using the dialog shown by selecting the *Device/Urbach Tail...* menu (figure 3.6).

Mobility parameters defined is 2.3.3.3 are set in the dialog shown by selecting the *Device/-Mobility...* menu (figure 3.7). The mobility Lua model file can also be selected in this dialog by pressing the *Model...* button.

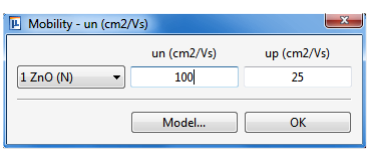

Figure 3.7: SolisDevice: Mobility Dialog.

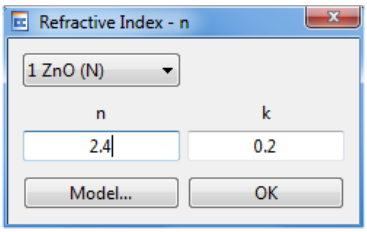

Figure 3.8: SolisDevice: Refractive Index and Extinction Coefficient Dialog.

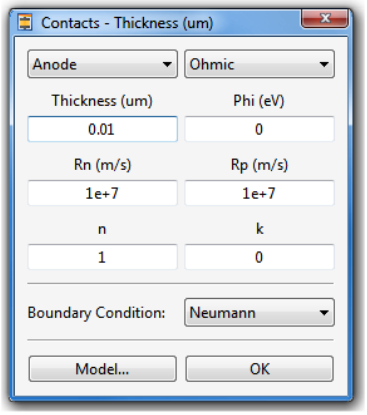

Figure 3.9: SolisDevice: Anode and Cathode Dialog.

Refractive Index and Extinction Coefficient are defined in the dialog shown by selecting the *Device/Index...* menu (figure 3.8). The Lua model file for refractive index and extinction coefficient spectra can also be selected in this dialog by pressing the *Model...* button.

Anode and Cathode parameters are defined in the dialog shown by selecting the *Device/-Contacts...* menu (figure 3.9). To define the Lua model file for the anode and cathode refractive index and extinction coefficient spectra, press the *Model...* button.

To define the **Voltage** and **Light** parameters, use the corresponding dialog shown by selecting the *Device/Voltage...* or the *Device/Light...* menu (figure 3.10). To set custom voltage sweep, press *File...* in the voltage dialog. The voltage sweep file format was already defined in section 2.3.5. Similarly, one can define custom light spectrum by selecting *File...* in the light dialog (for details on the custom spectrum format, refer to section 2.3.6).

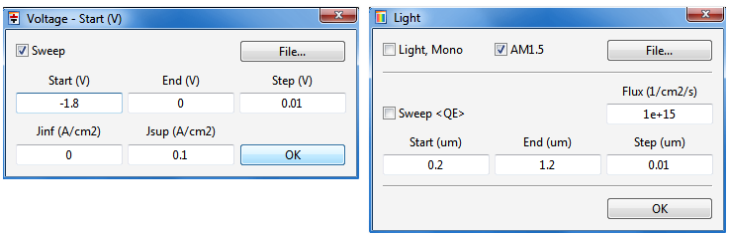

Figure 3.10: SolisDevice: Voltage and Light Dialog.

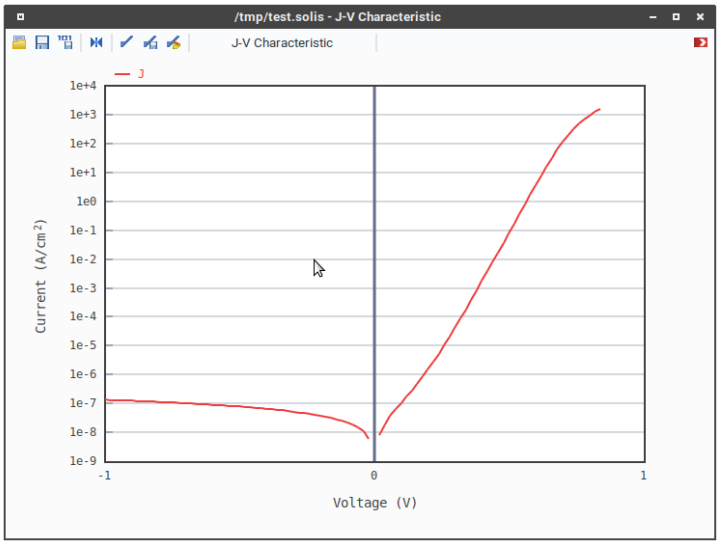

Figure 3.11: SolisPlot Screenshot.

3.3 Data Plotter

The Solis data plotter, `SolisPlot` (screenshot in figure 3.11), is usually not used as a standalone application. It is rather used by the simulator driver `soliscomp`, the code editor `SolisEdit` or the graphical device editor `SolisDevice` to plot the simulation results. But it can be used alone to directly plot simulation results or other data. To do so, just launch `SolisPlot`, browse to the simulation output directory and select the corresponding file. With `SolisPlot`, one can control every aspect of the plot curves, axes, scale, add lines, etc. The plot can be saved in the SVG format (and PDF format under Linux). This vectorial format retains high graphic quality and can be directly used in reports/papers or converted to PDF format to use in a LaTeXdocument for example.

3.4 Solis Scientific Calculator

The Solis scientific calculator, `SolisCalc`, is an advanced mathematical expression-based calculator (screenshot in figure 3.12). It supports the most common and useful functions. It's easy to use: to evaluate an expression, simply write it, using operators (+ - * / ^), parenthesis and mathematical functions and press ENTER (or F12). One can also use the numeric keypad to enter numbers and operators. One can set variables (with any non-reserved name), using fundamental constants, etc. `SolisCalc` menu gives an easy way to use the software functionality.

The following mathematical **functions** are supported:

```
exp(x)                  # exponential
ln(x)                   # natural logarithm
log(x)                  # decimal logarithm
pow(x,n)                # xn
```

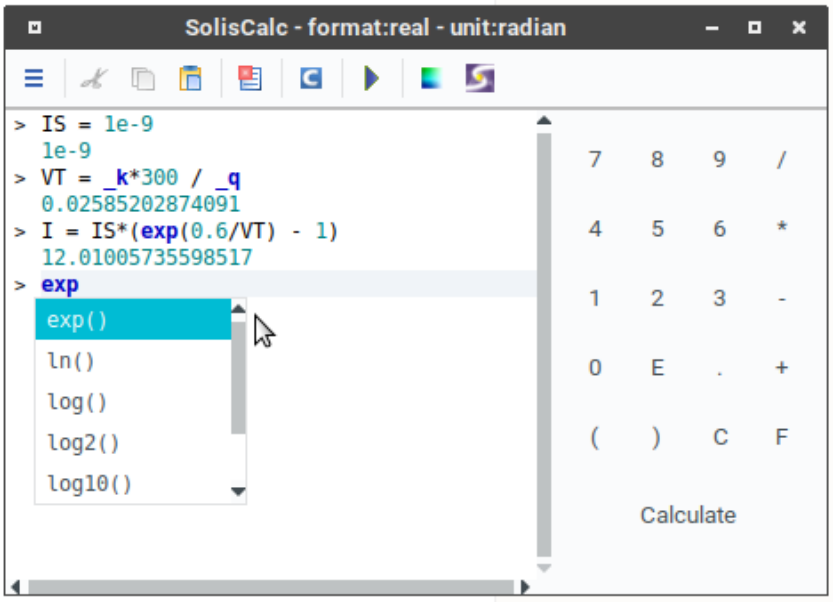

Figure 3.12: SolisCalc Screenshot.

```
sin(x)                  # sine
cos(x)                  # cosine
tan(x)                  # tangent
asin(x)                 # arc sine
acos(x)                 # arc cosine
atan(x)                 # arc tangent
sinh(x)                 # hyperbolic sine
cosh(x)                 # hyperbolic cosine
tanh(x)                 # hyperbolic tangent
abs(x)                  # absolute value
sqrt(x)                 # square root
ceil(x)                 # ceiling, the smallest integer not less than x
floor(x)                # floor, the largest integer not greater than x
fmod(x,y)               # x modulo y
erf(x)                  # error function
jn(n,x)                 # Bessel function of x of the first kind of order n
yn(n,x)                 # Bessel function of x of the second kind of order n
bern(x)                 # Bernoulli function: x / (exp(x) - 1)
gauss(x,m,s)            # Gauss function: exp((x - m)^2 / 2s^2)
lorentz(x,m,s)          # Lorentz function: s / ((x - m)^2 + s^2)
hypot(x,y)              # hypotenuse, sqrt(x^2 + y^2)
min(x,y)                # smallest value of x and y
max(x,y)                # largest value of x and y
rand(x)                 # random number between 0 and 1
time()                  # elapsed time in seconds since January 1, 1970
sign(x)                 # sign of x (-1 if x < 0, +1 if x > 0 and 0 if x = 0)
hypot(x,y)              # sqrt(x^2+y^2)
erf(x)                  # error function
gammaln(x)              # ln(gamma(x))
beta(x)                 # beta(x)
```

```
trunc(x)                # nearest integer
round(x)                # nearest integer, rounding
rint(x)                 # rounds the floating-point to an integer
```

Listing 3.2: Mathematical Functions in SolisCalc.

Constants:

```
pi                      # 3.14169...
_q                      # electron charge in C
_m                      # electron mass in kg
_k                      # Boltzmann constant in J / K
_h                      # Planck constant in m2 kg / s
_c                      # speed of light in vacuum in m / s
_e                      # vacuum permittivity in F / m
_n                      # Avogadro constant
```

Listing 3.3: Constants in SolisCalc.

Commands:

```
cls                     # clear the console
format:real             # set the numerical output format to real
format:int              # set the numerical output format to integer (10-base)
format:dec              # set the numerical output format to integer (10-base)
format:bin              # set the numerical output format to integer (2-base)
format:oct              # set the numerical output format to integer (8-base)
format:hex              # set the numerical output format to integer (16-base)
format?                 # print out the current numerical format
unit:degree             # set the angle unit to degree
unit:radian             # set the angle unit to radian
unit?                   # print out the angle unit
```

Listing 3.4: Commands in SolisCalc.

You can create **variables** (with any non-reserved name):

```
a = 6*2
```

The last expression evaluation can be accessed using the internal variable `ans`:

```
ans
```

Append a semicolon (;) to the expression to suppress the output:

```
a = 6*2;
```

A comment can be added at the end of an expression, using # :

```
y=sin(pi/4)   # comment
```

Previous calculated expressions can be reused by pressing up or down arrows.

`SolisCalc` can be executed from the **command line**:

```
soliscalc -run input [-out outfile]
```

`input` may be a filename or a double-quoted expression and `outfile` is the output filename:

```
soliscalc -run "a=1;b=2;c=a*sin(b)"
soliscalc -run calcin.txt -out calcout.txt
```

Integer arithmetic in binary, octal and hexadecimal bases:
One can perform integer calculations in binary, octal, decimal and hexadecimal bases in 32 bits unsigned format directly within the console, by prefixing the number with `0b` for binary base, `0o` for octal base and `0x` for hexadecimal base: just type the expression and press enter:

```
0xFFF7 + 12 + 0b111 + 0o547
```

To show result in hexadecimal, set accordingly the numerical output format (see listing 3.4):

```
format:hex
```

To **solve a nonlinear equation f(x) = 0**:

```
solve(x^2 - 2, 0, 10)
```

The syntax is `solve(function, xl, xh)` where `function` is the f(x) function expression (example: `x^2 - 2`), `xl` the x lower limit of the interval where the solution is to be found and `xh` the x higher limit (`xl` and `xh` are optional).

A session (calculations history) can be saved in a text file (with extension *.soliscalc*) and retrieved later: menu *File/Save* and *File/Open*.

Chapter 4
Examples of Using Solis

Contents

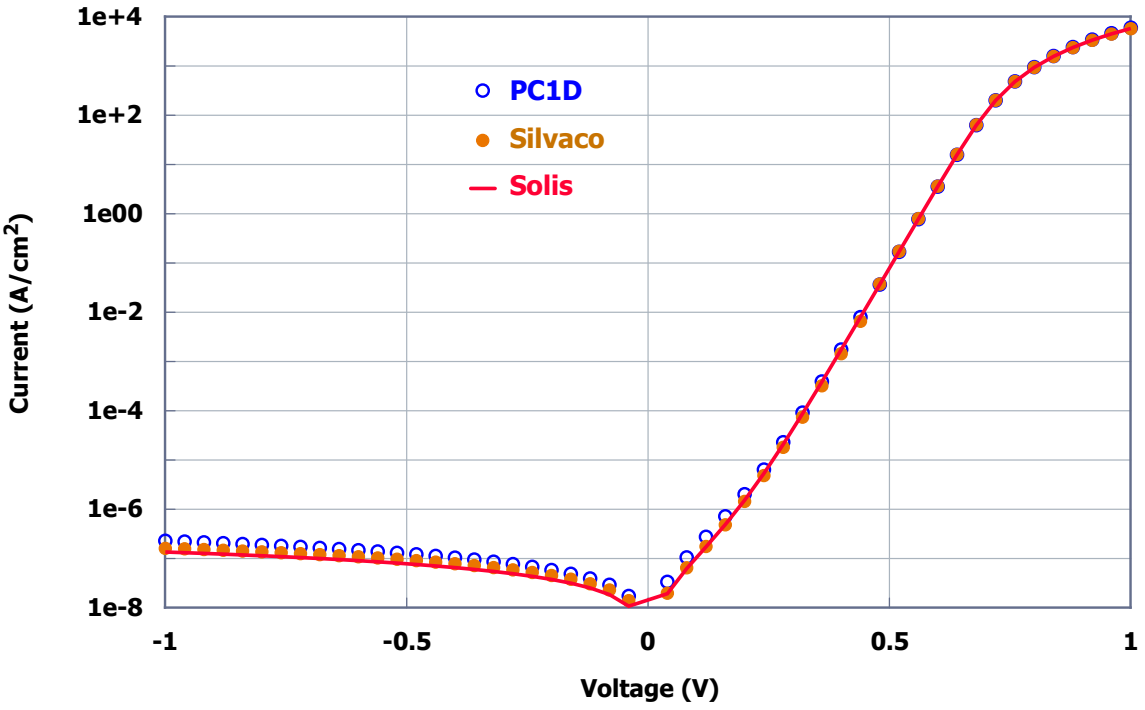

Figure 4.1: Monocrystalline silicon PN diode current-voltage characteristic simulated using Solis, PC1D and Atlas®. The symbols indicates the characteristics simulated using Atlas® from Silvaco® and PC1D, and the straight line the Solis simulated one.

Four examples are given here:

- A silicon solar cell, using the core functionality of the simulator. A monocrystalline N^+P Silicon solar cell is considered and the effect of the absorber thickness is highlighted.
- A monocrystalline N^+P Silicon solar cell using a Gaussian doping gradient created using a Lua defined model.
- A ZnO/CdS/CZTS solar cell [1].
- An AlGaN ultraviolet Schottky detector, using custom user-defined models code in Lua. This wide bandgap material and its alloys (in particular InGaN, AlGaN and BGaN) are particularly interesting for optoelectronic applications [16, 17, 18, 19, 20].

All the input and model files are included in the Solis `examples/simulator` directory. It is advised to copy the `examples` directory content to another location to keep the original examples untouched.

4.1 Comparison between Solis, PC1D and Atlas

In addition to the examples detailed in the following sections, one can compare results obtained with other simulators and, in particular, the reference commercial simulator Atlas® from

Silvaco® and the main one-dimensional academic simulator PC1D. To compare results obtained with different simulators, one have to set the same physical parameters and to check that the used models, the excitation parameters and numerical methods are expected to give comparable results.

Figure 4.1 shows, as an example, a monocrystalline silicon PN diode current-voltage characteristic simulated using Solis, PC1D and Atlas® from Silvaco®. The simulation files for Solis, Atlas® and PC1D are included in the Solis `examples/simulator` directory as:
`compare01_silicon_iv.solis` for Solis
`compare01_silicon_iv.in` for Atlas®
`compare01_silicon_iv.prm` for PC1D

4.2 Example 1: Silicon Solar Cell

The simulated silicon solar cell is a standard N^+P monocrystalline cell with a thin (1 μm) highly n-doped emitter and a thick ($>$ 50 μm) p-doped absorber. To simulate this solar cell, one can use: the simulator deriver `solis` in command line, the code editor `SolisEdit`, or the graphical device editor `SolisDevice`:

- **using the simulator driver `soliscomp` in command line.** Open a command prompt window or a console and go to the directory where you copied the `examples/simulator` directory content. If not yet done, add the `bin` directory to the `PATH` (refer to the operating system to see how to do so) to be able to run the Solis tools from any location. When ready, just type:

  ```
  soliscomp -run example01_silicon_solarcell.solis
  ```

  The `example01_silicon_solarcell.solis` input file content is shown in listing 4.1 and located in the `examples/simulator` directory.

- **using the Solis code editor `SolisEdit`.** Launch `SolisEdit`, located in the `bin` directory, and open `example01_silicon_solarcell.solis`. Click the *Run* button (or select *Run/Run* menu, or press *F12* key).

- **using the Solis graphical device editor `SolisDevice`.** Launch `SolisDevice`, located in the `bin` directory, and open `example01_silicon_solarcell.solis`. Click the *Run* button (or select *Run/Run* menu, or press *F12* key).

Using `SolisDevice` is probably the easiest way since everything is done graphically. It is not even necessary to know the Solis syntax to perform simulation when using `SolisDevice`. But if you prefer to have a greater control over the code, you can use `SolisEdit` that offers all needed functionality. With `SolisEdit`, one can also create and edit Lua models, using the same environment and, in addition, edit documents in other formats such as LaTeX, Python or C.

```
# Solis <SL>
# The above first line is mandatory
# Solis <Version 2.0 Build 2107>

[Device]
   Name="c-Si Solar Cell"
   Temperature=300

[Layer1]
   Material="Si"
   Thickness=1
   Dopant.N=1e+20
   Dopant.Taun=1e-4
   Dopant.Taup=1e-4
   Dopant.Type="Donor"
   Mesh.Points=10

[Layer2]
   Material="Si"
   Thickness=100
   Dopant.N=1e+17
   Dopant.Taun=1e-4
   Dopant.Taup=1e-4
   Dopant.Type="Acceptor"
   Mesh.Points=1000

[Anode]
   Anode.Type="Ohmic"
[Cathode]
   Cathode.Type="Ohmic"

[Voltage]
   Voltage.Source="Sweep"
   Voltage.Start=-0.7
   Voltage.End=0
   Voltage.Step=0.01
   Voltage.Jinf=0
   Voltage.Jsup=0.1

[Light]
   Light.Source="AM1.5"

[Numeric]
   Numeric.Method="Gummel"
   Numeric.Mesh="Layer"

[Command]
   Command.Plot="Yes"
```

Listing 4.1: Silicon Solar Cell using Core Functionality.

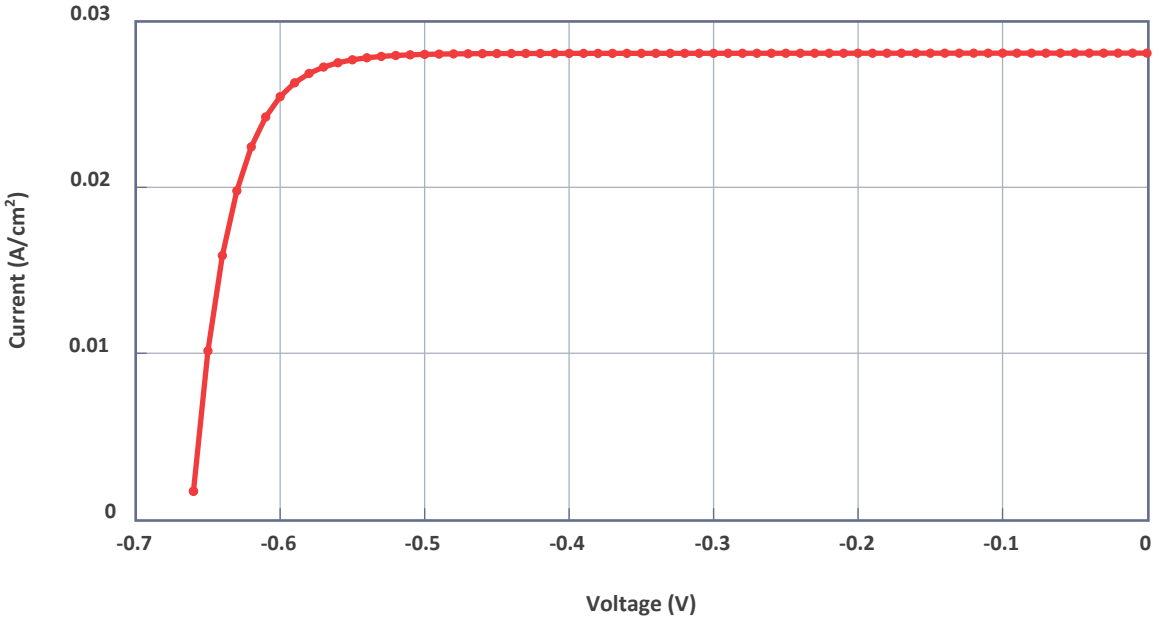

Figure 4.2: The monocrystalline silicon solar cell current-voltage characteristic under AM1.5 spectrum. The solar cell Solis input code is shown in the listing 4.1. The photovoltaic performances obtained for this solar cell are: efficiency $\eta = 15.696\ \%$; short-circuit current $J_{SC} = 0.028\ A/cm^2$; open-circuit voltage $V_{OC} = 0.661\ V$; fill factor $FF = 83.87\ \%$; series resistance $R_S = 1\ \Omega.cm^2$; shunt resistance $R_P = 33980\ \Omega.cm^2$.

The simulation ended after a few seconds (less than two seconds on an Intel i7 2.9GHz quad core processor) and outputs the solar cell current-voltage characteristic under AM1.5 solar illumination and the spatial distribution of potential, electric field, Fermi levels, conduction and valence band, carrier concentration, doping and traps concentration, currents (electrons, holes and total), photogeneration and recombination (Auger, direct and SRH).

The current-voltage characteristic under AM1.5 is shown in figure 4.2. The solar cell performance (open-circuit voltage, short-circuit current, fill factor and efficiency) are automatically determined by Solis.

In the code (or using the graphical device editor), perform simulation by varying the **thickness** from `5` to `300` micrometers and note the efficiency vs thickness. Figure 4.3 shows this variation of efficiency reproducing what experimentally expected for silicon solar cell: an efficiency saturation beyond a given thickness. Of course, this behavior depends strongly on the other physical parameters, such as diffusion length, mobility, doping and absorption. A rigorous mathematical optimization of this solar cell (with respect to, e.g., thickness and doping) could be performed using the SLALOM tool [21] with Solis as a backend simulation engine.

4.3 Example 2: Silicon Solar Cell with Gradient Doping

In the previous example, we use only Solis core functionality to simulate a silicon solar cell. It is also possible to use the embedded Lua scripting engine to include, for example, a doping

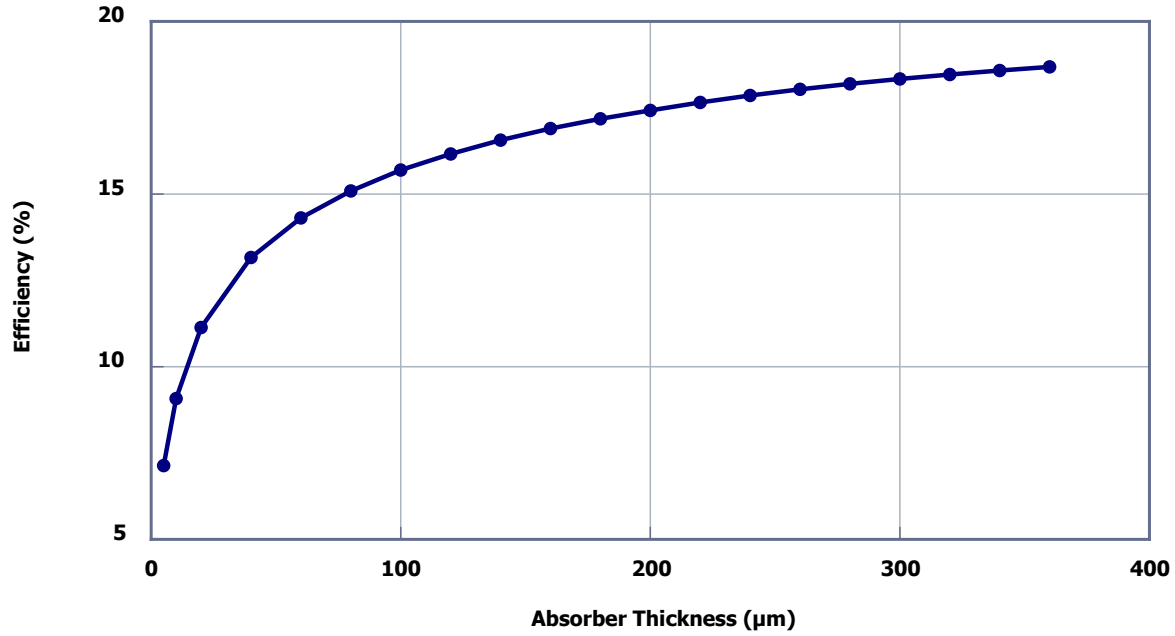

Figure 4.3: The monocrystalline silicon solar cell efficiency variation with the absorber thickness. The solar cell Solis input code is shown in the listing 4.1 with an absorber thickness varying from 5 to 360 micrometers.

profile either theoretical or deduced from experimental measurements using, e.g., the capacitance-voltage technique. To illustrate this feature, let us simulate a monocrystalline silicon solar cell consisting of a p-type wafer implanted with phosphorous to create the N^+P solar cell with a thin N^+ region. To simulate this solar cell, one can create a Solis device with only one layer [1], corresponding to the p thick absorber, and a n-doping profile (Gaussian, exponential or empirical) in Lua.

This solar cell input code is shown in listing 4.2.

```
# Solis <SL>
# The above first line is mandatory
# Solis <Version 2.0 Build 2107>

[Device]
   Name="c-Si Solar Cell"
   Temperature=300

[Layer1]
   Material="Si"
   Thickness=100
   Mesh.Points=1000
   Dopant.Model="example02_silicon_doping.lua"
   Index.Model="example02_silicon_index.lua"
```

[1] Since Solis natively handles grading, devices can be simulated with only one layer and a graded doping (and/or graded bandgap, etc.).

```
[Anode]
   Anode.Type="Ohmic"

[Cathode]
   Cathode.Type="Ohmic"

[Voltage]
   Voltage.Source="Sweep"
   Voltage.Start=-0.7
   Voltage.End=0
   Voltage.Step=0.01
   Voltage.Jinf=0
   Voltage.Jsup=0.1

[Light]
   Light.Source="AM1.5"

[Numeric]
   Numeric.Method="Gummel"
   Numeric.Mesh="Layer"

[Command]
   Command.Plot="Yes"
```

Listing 4.2: Silicon Solar Cell with Gradient Doping using Lua Defined Model.

This input file, named `example02_silicon_solarcell_graded.solis`, is located in the `examples/simulator` directory. The doping profile and refractive index and extinction coefficient model are defined in Lua files located also in the `examples/simulator` directory, and shown in listings 4.3 and 4.4.

```
-----------------------------------------------------------------
-- File: example02_silicon_doping.lua
-- Solis <Version 2.0 Build 2107>
-----------------------------------------------------------------
function sol_doping(position, thickness)
    Na = 1e15           -- background acceptor density in 1/cm3
    N0 = 1e20           -- donor density at the layer surface in 1/cm3
    Nr = 1e19           -- reference density in 1/cm3
    Taun = 1e-4         -- electron lifetime (seconds)
    Taup = 1e-4         -- hole lifetime (seconds)
    E = 0               -- activation energy (eV)
    F = 1               -- degeneracy factor
    graded = true       -- set graded to true in this case
    posr = 0.2
    Nd = N0 * math.exp(-(position * position) / (posr * posr))
    return Na, Nd, Nr, Taun, Taup, E, F, graded, true
end
```

Listing 4.3: Gradient Doping Lua Model.

```
------------------------------------------------------------
-- File: example02_silicon_index.lua
-- Solis <Version 2.0 Build 2107>
------------------------------------------------------------

function sol_index(position, thickness, bandgap, lambda)
    Eph = 1.23984 / lambda
    Eg = 1.12
    kA = 0.05
    graded = false
    n = 3.42    -- refractive index
    k = 0       -- extinction coefficient
    if (Eph >= Eg) then
        k = (kA * ((Eph - Eg) / Eg) * ((Eph - Eg) / Eg))
    else
        k = 0.0
    end
    return n, k, graded, true
end
```

Listing 4.4: Index Lua Model.

Launch `SolisEdit` or `SolisDevice` and open `example02_silicon_graded.solis`. Click the *Run* button (or select *Run/Run* menu, or press *F12* key). The found efficiency, $\eta = 14.01$ %, could be optimized by studying in detail the effect of the doping gradient, thickness, diffusion length, etc. This kind of study is also useful for thin films solar cells where the control of the doping profile plays an important role in the efficiency optimization.

4.4 Example 3: CZTS Solar Cell

The CZTS, Copper Zinc Tin Selenide $Cu_2ZnSnSe_4$, is become a promising material for thin films solar cells [22] mainly because of the abundance of its elements and its photovoltaic efficiency, comparable, at least for the best laboratory samples, to the efficiency of well-established CIGS solar cells [23, 24]. For such a "new" material, it is often necessary to be able to define custom physical models based on or extrapolated from experimental data. We will use, in this example, the Lua scripting language to define one of these physical models: the index/absorption.

This ZnO/CdS/CZTS solar cell input code is shown in listing 4.5.

```
# Solis <SL>
# The above first line is mandatory
# Solis <Version 2.0 Build 2107>

[Device]
```

```
    Name = "CZTS Solar Cell"
    Temperature = 300

[Layer1]
    Material = "ZnO"
    Thickness = 0.1
    Mesh.Points = 100
    Dopant.N = 1e+18
    Dopant.Type = "N"
    Dopant.Taun = 1e-7
    Dopant.Taup = 1e-7

    Index.Model = "example03_czts_solarcell_index.lua"

[Layer2]
    Material = "CdS"
    Thickness = 0.1
    Mesh.Points = 100
    Dopant.N = 1e+18
    Dopant.Type = "N"
    Dopant.Taun = 1e-7
    Dopant.Taup = 1e-7

    Index.Model = "example03_czts_solarcell_index.lua"

[Layer3]
    Material = "CZTS"
    Thickness = 1
    Mesh.Points = 500
    Dopant.N = 1e+16
    Dopant.Type = "P"
    Dopant.Taun = 1e-7
    Dopant.Taup = 1e-7

    Index.Model = "example03_czts_solarcell_index.lua"

[Anode]
    Anode.Type = "Ohmic"

[Cathode]
    Cathode.Type = "Ohmic"

[Voltage]
    Voltage.Source = "Sweep"
    Voltage.Start = -1
    Voltage.End = 0
    Voltage.Step = 0.01
    Voltage.Jinf = 0
    Voltage.Jsup = 0.1
```

```
[Light]
    Light.Source = "AM1.5"

[Numeric]
    Numeric.Method = "Gummel"
    Numeric.Mesh = "Layer"

[Command]
    Command.Plot = "Yes"
```

Listing 4.5: CZTS Solar Cell with Index Lua Defined Model.

This input file, named `example03_czts_solarcell.solis`, and index model, as shown in 4.6 (for ZnO, CdS and CZTS), are located in the `examples/simulator` directory. The refractive index model would have been defined for each material in a separate file. Using one file is just for clarity (the code could also be adapted in the function, using the bandgap value).

```
-----------------------------------------------------------------
-- Solis <Version 2.0 Build 2107>
-----------------------------------------------------------------

function sol_index(position, thickness, bandgap, lambda)
    Eph = 1.23984 / lambda
    Eg  = bandgap
    kA  = 0.2
    graded = false

    n = 2.59
    if (Eph >= Eg) then
        k = (2.0 * Eg / Eph) * kA * sqrt((Eph - Eg) / Eg)
    else
        k = 0.0
    end

    return n, k, graded, true
end
```

Listing 4.6: Index Lua Model for ZnO, CdS and CZTS.

Start `SolisEdit` (or the graphical device editor `SolisDevice`), browse to the directory where you copied the `examples/simulator` directory content, open the file:
`example03_czts_solarcell.solis`
and run it.
The obtained current-voltage characteristic under AM1.5 is shown in figure 4.5 and band diagram in 4.4. The simulation take about eight seconds (on an Intel i7 2.9GHz quad core processor).

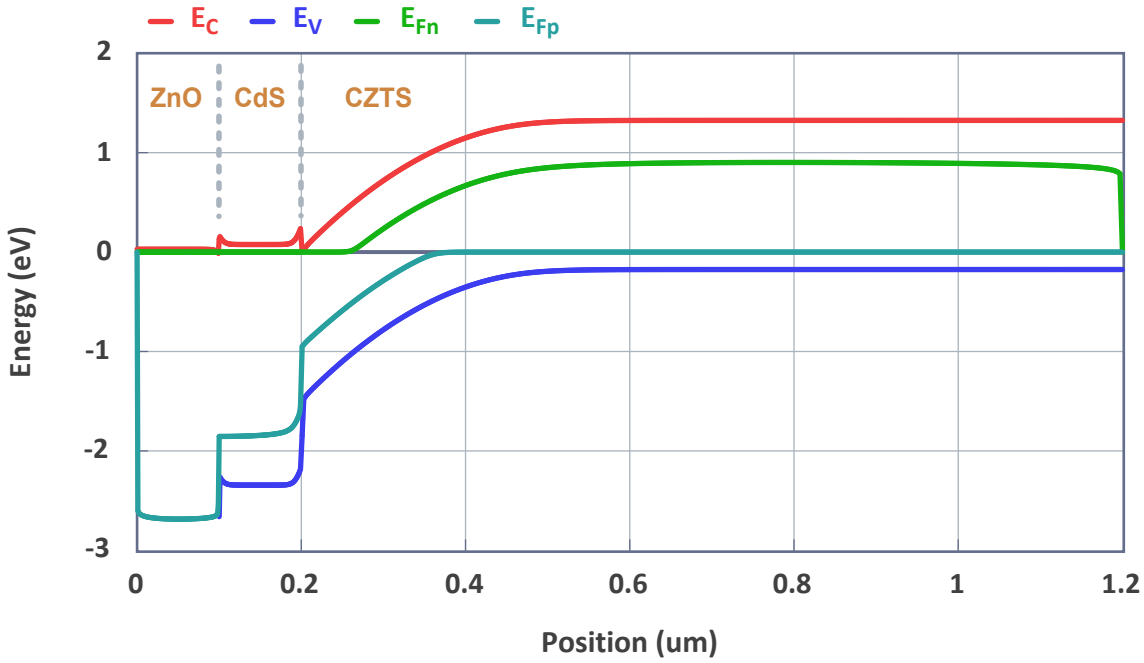

Figure 4.4: The CZTS solar cell band diagram under AM1.5 spectrum. The solar cell Solis input code is shown in the listing 4.5. The band diagram is obtained under short-circuit. E_C and E_V are the band edges and E_{Fn} and E_{Fp} the quasi-Fermi levels.

4.5 Example 4: AlGaN ultraviolet Schottky detector

AlGaN is a III-N wide bandgap compound semiconductor particularly interesting in ultraviolet (UV) light emission (lasers, LED) and detection [16, 17, 18, 19]. Insensitivity to visible light, resistance to high radiation levels or high temperature, possibility to cover a large UV range (from GaN to AlN bandgap) are the main advantages of using AlGaN in UV detection. The AlGaN Schottky detector do not include any p-doped layer and is thus particularly interesting since the AlGaN p-doping is difficult to master in these III-N semiconductors.

The input listing in 4.7 is used to simulate an AlGaN Schottky detector with an absorbing 5 nm thick contact layer.

Start `SolisEdit` (or the graphical device editor `SolisDevice`), browse to the directory where you copied the `examples/simulator` directory content, open the file:
`example03_algan_detector.solis`
and run it. The obtained external quantum efficiency is shown in figure 4.6. The simulation take less than one second (on an Intel i7 2.9GHz quad core processor).

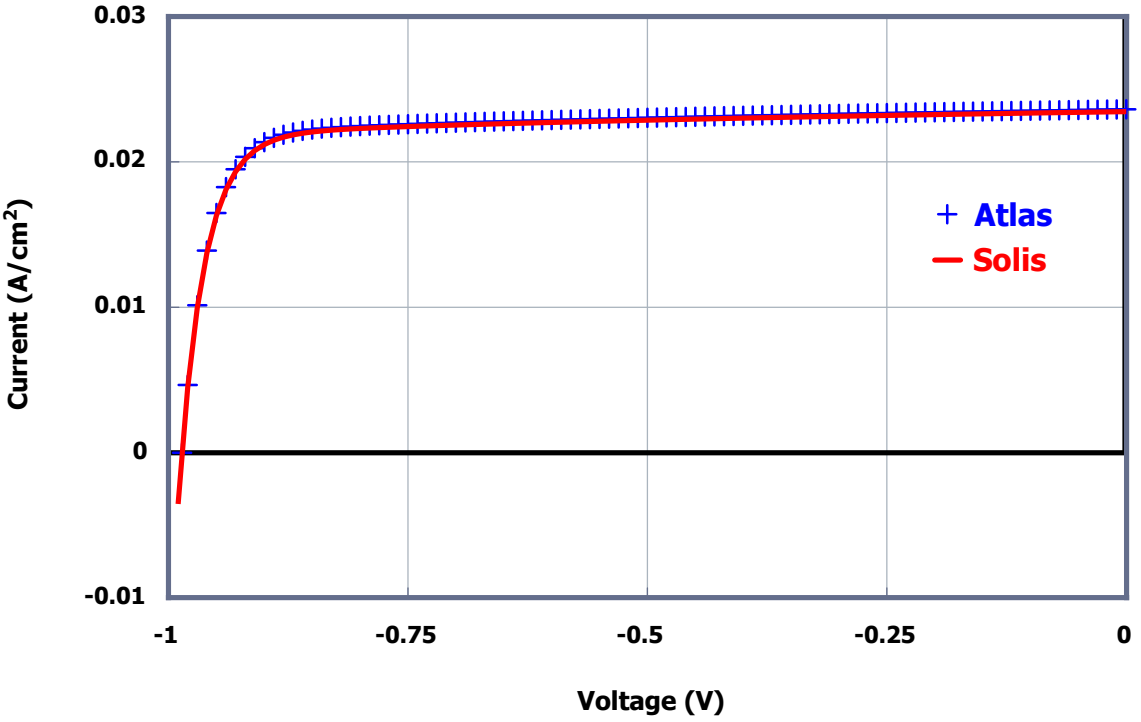

Figure 4.5: The CZTS solar cell current-voltage characteristic under AM1.5 spectrum. The solar cell Solis input code is shown in the listing 4.5. The symbols indicates the characteristic simulated using Atlas® from Silvaco® and the straight line the Solis simulated one. The photovoltaic performances obtained for this solar cell are: efficiency $\eta = 19.263$ % ; short-circuit current $J_{SC} = 0.0234$ A/cm^2 ; open-circuit voltage $V_{OC} = 0.985$ V ; fill factor FF $= 82.73$ % ; series resistance $R_S = 1.5$ Ω.cm^2 ; shunt resistance $R_P = 1039$ Ω.cm^2 ; with an integrated AM1.5 power of 0.099271 W/cm^2. The Atlas® calculated performances are: $\eta = 19.260$ % ; $J_{SC} = 0.0236$ A/cm^2 ; $V_{OC} = 0.986$ V ; FF $= 82.77$ % ; with an integrated AM1.5 power of 0.100037 W/cm^2. The solar Atlas® input code, `example03_czts_solarcell.in`, is located in the `examples/simulator` directory.

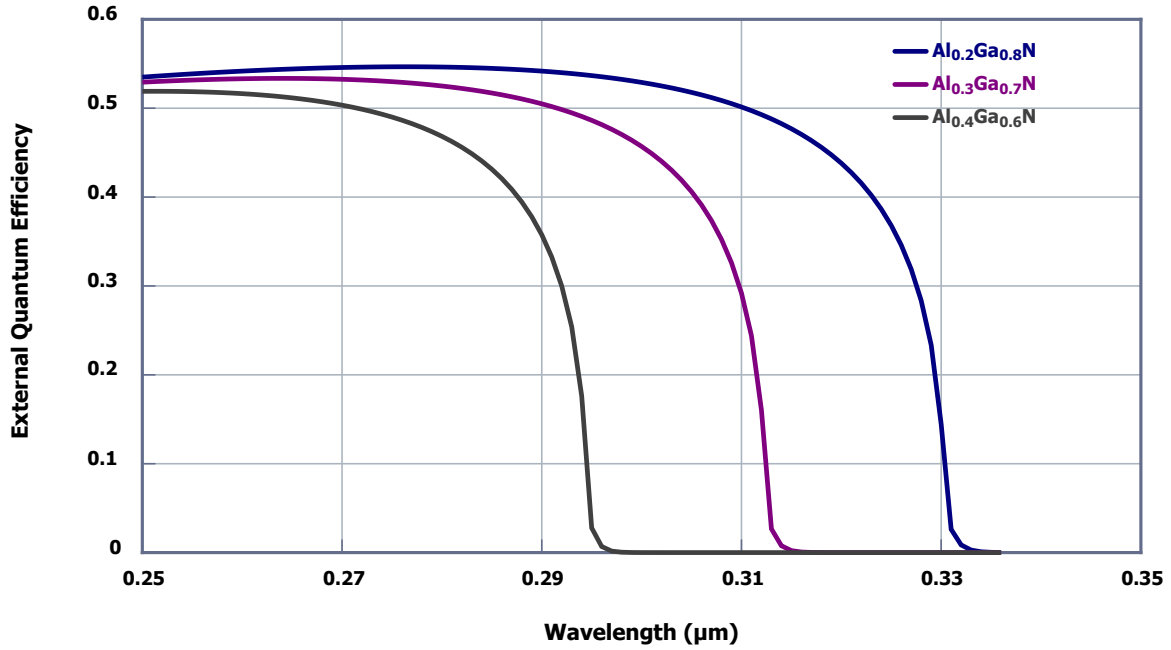

Figure 4.6: The AlGaN Schottky UV detector external quantum efficiency spectrum for three aluminum composition x in $Al_xGa_{1-x}N$. The corresponding Solis input code is shown in the listing 4.7.

```
# Solis <SL>
# The above first line is mandatory
# Solis <Version 2.0 Build 2107>

$xcomp = 0.2
$Workfunc = 5.65
$Affinity = (2.26 * $xcomp) + (4.1 * (1 - $xcomp))
$BarrierHeight = $Workfunc - $Affinity

[Device]
    Name = "AlGaN Schottky UV Detector"
    Temperature = 300

[Layer1]
    Material = "AlGaN"
    Thickness = 1
    Mesh.Points = 200
    Composition.x = $xcomp
    Bandgap.Bowing = 1.3        # H Angerer et al. APL 71 (11) 1997
    Dopant.N = 1e16
    Dopant.Nr = 1e20
    Dopant.Taun = 1e-9
    Dopant.Taup = 1e-9
    Dopant.Type = "D"

[Anode]
    Anode.Type = "Schottky"
    Anode.Phi = $BarrierHeight
    Anode.Index.n = 1.1         # refractiveindex.info
    Anode.Index.k = 2.2
    Anode.Thickness = 0.005
[Cathode]
    Cathode.Type = "Ohmic"

[Voltage]
    Voltage.Source = "None"
[Light]
    Light.Source = "Sweep"
    Light.Start = 0.25
    Light.End = 0.35
    Light.Step = 0.001
    Light.Flux = 1e+15

[Numeric]
    Numeric.Method = "Gummel"
    Numeric.Mesh = "Layer"
[Command]
   Command.Plot = "Yes"
```

Listing 4.7: GaN ultraviolet Schottky detector using Lua defined models.

Bibliography

[1] S Ould Saad Hamady. "Solis: a modular, portable, and high-performance 1D semiconductor device simulator". In: *Journal of Computational Electronics* (2020), pp. 1–8 (cited p. i, 52).

[2] S Ould Saad Hamady. "SigmaSim: Enhanced One-Dimensional Simulator for Semiconductor Devices". In: *37th International Symposium on Compound Semiconductors (ISCS), Takamatsu, Japan. Oral.* 2010 (cited p. 1).

[3] *The IUP GUI toolkit, (C) 1994-2017 Tecgraf/PUC-Rio.* 2017 (cited p. 3).

[4] *The Scintilla Component, (C) Neil Hodgson.* 2016 (cited p. 3).

[5] S Selberherr. *Analysis and simulation of semiconductor devices.* Springer Science & Business Media, 2012 (cited p. 6).

[6] DL Scharfetter and HK Gummel. "Large-signal analysis of a silicon read diode oscillator". In: *IEEE Transactions on electron devices* 16.1 (1969), pp. 64–77 (cited p. 7).

[7] F Urbach. "The long-wavelength edge of photographic sensitivity and of the electronic absorption of solids". In: *Physical Review* 92.5 (1953), p. 1324 (cited p. 17).

[8] F Bernardini, V Fiorentini, and D Vanderbilt. "Spontaneous polarization and piezoelectric constants of III-V nitrides". In: *Physical Review B* 56.16 (1997), R10024 (cited p. 20).

[9] NREL. *Reference Solar Spectral Irradiance: ASTM G-173.* 2004 (cited p. 22, 23).

[10] A Nishida. "Experience in developing an open source scalable software infrastructure in Japan". In: *International Conference on Computational Science and Its Applications.* Springer. 2010, pp. 448–462 (cited p. 24).

[11] *The LIS linear solvers, (C) The SSI Project, Kyushu University, Japan.* 2013 (cited p. 24).

[12] O Madelung. *Semiconductors: data handbook.* Springer Science & Business Media, 2012 (cited p. 36, 37).

[13] JW Slotboom and HC De Graaff. "Measurements of bandgap narrowing in Si bipolar transistors". In: *Solid-State Electronics* 19.10 (1976), pp. 857–862 (cited p. 36, 37).

[14] *RefractiveIndex.info.* 2010 (cited p. 36, 37).

[15] *Ioffe.* 2010 (cited p. 36, 37).

[16] Q Chen, JW Yang, A Osinsky, S Gangopadhyay, B Lim, MZ Anwar, M Asif Khan, D Kuksenkov, and H Temkin. "Schottky barrier detectors on GaN for visible–blind ultraviolet detection". In: *Applied Physics Letters* 70.17 (1997), pp. 2277–2279 (cited p. 52, 61).

[17] E Munoz, E Monroy, JL Pau, F Calle, F Omnes, and P Gibart. "III nitrides and UV detection". In: *Journal of Physics: Condensed Matter* 13.32 (2001), p. 7115 (cited p. 52, 61).

[18] S Ould Saad Hamady. "A simulation of doping and trap effects on the spectral response of AlGaN ultraviolet detectors". In: *Journal of Semiconductors* 33.3 (2012), p. 034002 (cited p. 52, 61).

[19] S Ould Saad Hamady. "Simulation of AlGaN and BGaN metal-semiconductor-metal ultraviolet photodetectors". In: *Physica Status Solidi (c)* 9.3-4 (2012), pp. 1099–1104 (cited p. 52, 61).

[20] S Ould Saad Hamady, A Adaine, and N Fressengeas. "Numerical simulation of InGaN Schottky solar cell". In: *Materials Science in Semiconductor Processing* 41 (2016), pp. 219–225 (cited p. 52).

[21] S Ould Saad Hamady and N Fressengeas. "SLALOM: Open-source, portable, and easy-to-use solar cell optimizer. Application to the design of InGaN solar cells". In: *EPJ Photovoltaics* 9 (2018), p. 13 (cited p. 55).

[22] M Jiang and X Yan. "Cu_2ZnSnS_4 thin film solar cells: present status and future prospects". In: *Solar Cells-Research and Application Perspectives*. InTech, 2013 (cited p. 58).

[23] M Kaelin, D Rudmann, and AN Tiwari. "Low cost processing of CIGS thin film solar cells". In: *Solar Energy* 77.6 (2004), pp. 749–756 (cited p. 58).

[24] D Mencaraglia, S Ould Saad Hamady, and Z Djebbour. "Admittance spectroscopy for non-crystalline thin film devices characterization: comparison of $Cu(In, Ga)Se_2$ and $a - Si : H$ cases". In: *Thin Solid Films* 431 (2003), pp. 135–142 (cited p. 58).