SLALOM: Open-Source, Portable and Easy-to-use Solar Cell Optimizer. Application to the Design of InGaN and CZTS Solar Cells.

Sidi Ould Saad Hamady [1,2,*] and Nicolas Fressengeas [1,2]

[1] *Université de Lorraine, Laboratoire Matériaux Optiques, Photonique et Systèmes, Metz, F-57070, France*
[2] *Laboratoire Matériaux Optiques, Photonique et Systèmes, CentraleSupélec, Université Paris-Saclay, Metz, F-57070, France*
[*] *Corresponding author: Sidi Ould Saad Hamady, sidi.hamady@univ-lorraine.fr*

Abstract

SLALOM (SoLAr CelL Multivariate OptiMizer) [1] is a set of open-source Python programs implementing a rigorous mathematical methods for the optimization of solar cells using as backend a drift-diffusion device simulator [1]. It aims to be simple to use, to maintain and to extend. It includes a core optimizer using the well tested robust mathematical methods, a set of user interface utilities and some complete and working examples easily adaptable to new solar cell technologies. SLALOM uses, as device simulator, the Silvaco® Atlas tool. It can be easily extended to use any simulator that have a standard input format and a command line interface.

1 Install

Usually the device simulator is installed on a high performance and robust Linux server with Red Hat, CentOS or Debian distribution and far more rarely on Windows or macOS. Linux offers the stability, flexibility, performance and durability needed for optoelectronic devices simulation. Figure 1 shows a schematic view of the two main configurations where SLALOM is used:

- Client / Server configuration, where the device simulator (e.g. Silvaco®) is installed on a remote high-performance calculation server under Linux (Red Hat, CentOS or Debian) and the optimizer in installed on the user local machine (laptop, thin client, workstation or even tablet or smartphone). User connect to the remote server using the standard SSH (Secure Shell) protocol using public / private keys. In this configuration SLALOM is installed on the client side and controls the remotely installed simulator to peform the solar cell optimization. This configuration is the standard one in academic institutions or laboratories where a high performance calculation server (or a clutser server) is remotely used for TCAD and simulation.
- Local configuration, where the device simulator and SLALOM are installed on the same computer.

SLALOM handle these configurations transparently and should be installed, in both cases, locally.

1.1 SLALOM requirements

- Python version 2.7.x or later [2] (SLALOM is coded and tested using Python 2.7.12).
- NumPy version 1.5 or later. NumPy is a package for linear algebra and numerical computing [3].

[1] SLALOM is freely available on the github repository https://github.com/sidihamady/SLALOM or by downloading the up-to-dare archive http://www.hamady.org/photovoltaics/slalom_source.zip

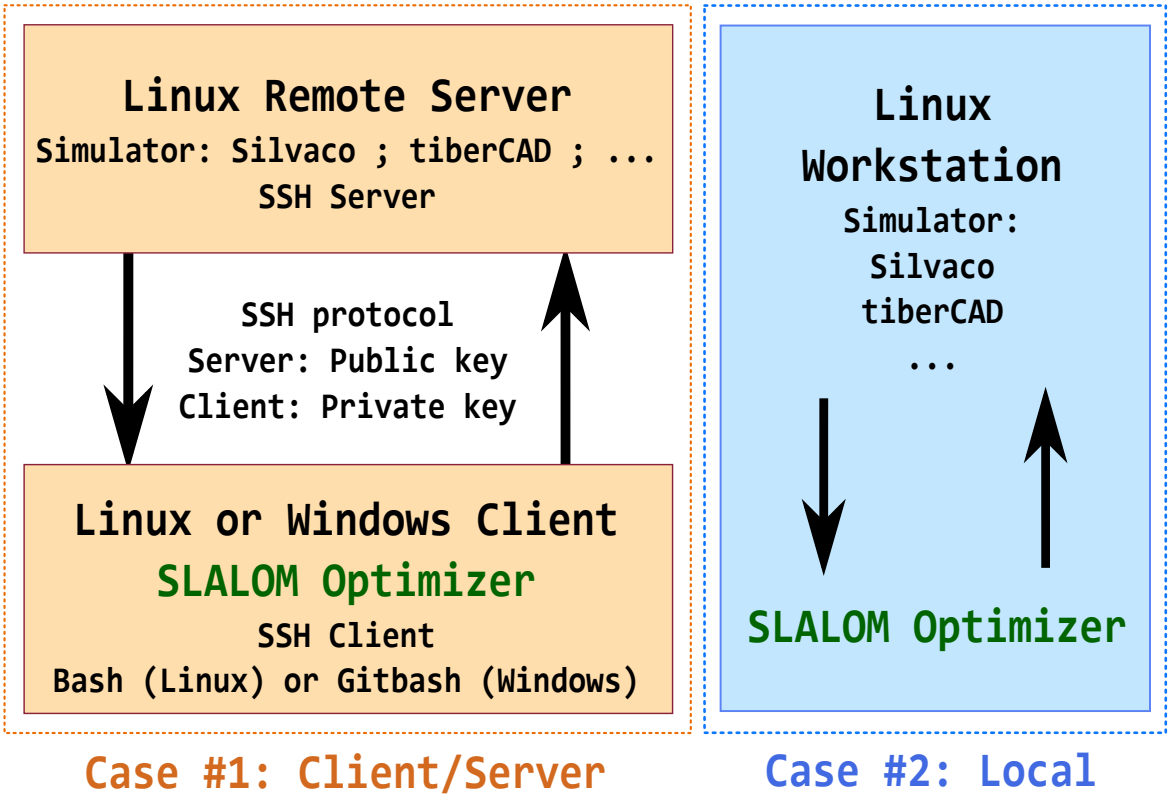


Figure 1: SLALOM install cases. Case 1 is the standard way: the device simulator (Silvaco®, or any other simulator) is installed on a remote calculation server and accessed from a local computer or thin client using the SSH protocol. Case 2 correspond to the case where the device simulator is installed in the user machine and can then be accessed locally.

- SciPy version 0.13.1 or later. SciPy contains a more specialized scientific routines for, e.g., mathematical optimization among many other routines [4].
- Matplotlib version 1.3.x or later. Matplotlib is a 2D plotting and visualization package [5].
- Tkinter 8.5 or later (optional, required only for the client GUI monitor and the device definition module). Tkinter is a package for building multi-platform graphical user interface (GUI) [6].
- Gitbash for Windows (for SSH in Client / Server configuration) [7]: Linux-like console for Windows including standard and Git tools and commands.

1.2 Linux

Python is already installed with almost any Linux distribution.
For Red Hat (or clones such as CentOS or Scientific Linux, or Fedora), NumPy, SciPy and Matplotlib can be installed using yum:

sudo yum install python-numpy python-scipy python-matplotlib tkinter python-matplotlib-tk

If the default Python version is older than 2.7.x (as for CentOS 6.x and Red Hat 6.x), you have to install a newer Python / NumPy / SciPy / Matplotlib version. To check your Python version, just type:

/usr/bin/python -V

1.3 Windows

Two methods (at least!) to install Python / NumPy / SciPy / Matplotlib under Windows:

- Method 1 (recommended): Download and install the Anaconda distribution:

  https://www.continuum.io/downloads

  Choose Python 2.7 version.

  Anaconda contains almost everything you need for scientific programming with Python and is well maintained.

- Method 2: Install the official Python distribution from:

  https://www.python.org/downloads/release/python-2712/

  Choose preferably Python 2.7.12 and install it.

  When Python is installed, download and install NumPy/SciPy/Matplotlib from:

  http://www.lfd.uci.edu/ gohlke/pythonlibs/

  The only advantage of this latter method is a smaller size since only vanilla Python and selected modules are installed.

Under Windows you need SSH client to access the remote calculation server if SLALOM is used in the Client / Server configuration (as usually done). For that you need to install Git for Windows from here:

https://git-for-windows.github.io/

Git for Windows offers a Linux-like console under Windows (Gitbash) including a very good SSH client in addition to Git and many other standard commands and tools.

To test Gitbash, start it and type some commands:

cd /C/Users/MyName/.ssh/
ls
vi ./config

If you are not familiar to Linux, you need to learn some basic commands and tools such as *ls*, *cd*, *mkdir*, *cp*, *mv*, *pwd*, *cat*, *touch*, *chmod*, *chown*, *ssh*, etc.

Note: under Gitbash, file path use the Linux notation:

/C/Users/MyName/.ssh/config is the Gitbash equivalent for: *C:\Users\MyName\.ssh\config*

1.4 SSH communication

As highlighted previously, SLALOM use the Secure Shell (SSH) to control the simulator installed on the remote server.

To configure the SSH client (under Windows with Gitbash or under Linux), perform the followings steps:

- edit the */.ssh/config*. You can use *vi* to edit this file (or any other editor):

  cd /.ssh/
  ls
  vi ./config

  Put the remote server information, like:

  Host slalom
  Hostname 192.168.1.10
  User myname

  Replace with your server information given by your system administrator.

  To access the remote calculation server defined in */.ssh/config*, you need to use SSH with keys. The idea is simply to use cryptographic keys to communicate with the server and avoid using password. Two keys are necessary, a private one located on the client side and a public one located on the server. To create the two keys, start a Linux console (or Gitbash under Windows) and type:

ssh-keygen -t rsa -b 4096 -C "me@mail.com"

Replace with your e-mail.
ssh-keygen generated a private/public keys pair. You can view the public key by typing:

cat /.ssh/id_rsa.pub

After generating the keys, you need to send the public one to the server:

cat /.ssh/id_rsa.pub | ssh myname@slalom 'cat >> /home/myname/.ssh/authorized_keys'

Replace with your server information as included in */.ssh/config*.
As a last step, connect to the server using SSH and when asked choose to add it to the known hosts:

ssh myname@slalom

The next time the connexion will be estabslihed without asking for password. The public / private keys pair ensure a secure communication between the client and the server. SLALOM will use this reliable and secure mechanism to control the remote simulator.
If the connexion failed, you can check and, if necessary, correct the remote SSH file access permissions:
Connect to the server using password and type:

chmod 700 .ssh
chmod 600 .ssh/authorized_keys

Disconnect from the server, and retry to connect without using password.

1.5 Installing SLALOM

When the required packages (Python / NumPy / SciPy / Matplotlib) are installed and SSH configured, you can install SLALOM by downloading it from:

https://github.com/sidihamady/SLALOM

http://www.hamady.org/photovoltaics/slalom_source.zip.

Unzip and that's all!

2 Step-by-step Guide

2.1 SLALOM directory structure

The SLALOM distribution structure is shown in figure 2.

- The **Device** directory contains the input files for Silvaco® simulator. For the simulator, the input files are coded with four mandatory prerequisites: simplicity, modularity, clearness and

reusability. For Silvaco® par exemple all the material parameters and physical model are defined in a specialized C external files and reused and adapted for new structure. To define a new solar cell structure, juste reuse these files, keep the same structure and prerequisites, and adapt to your specific study.

For Silvaco® simulator, two solar cell structures are included in SLALOM: a **CdS/CZTS (Copper Zinc Tin Sulfide)** structure and a **InGaN** (Indium Gallium Nitride) PN structure. For instance, there is one CdS/CZTS input file for Deckbuild (*CZTS_NP.in*) and seven C files for physical models and parameters: *CdS_Index.c* for the refractive index and extinction coefficient of CdS; *CZTS_Bandgap.c* for the bandgap, affinity and density of states model; *CZTS_Index.c* for the refractive index and extinction coefficient of CZTS; *CZTS_Mobility.c* for the CZTS mobility model; *CZTS_Permittivity.c* for the CZTS dielectric permittivity model; *CZTS_Recomb.c* for the CZTS recombinations model; and *CZTS_Parameters.h* header file containing the CZTS parameters used by the above C files.

- The **Guide** directory contains this PDF guide.
- The **Images** directory contains various icons and images used by the GUI part of SLALOM.
- The **Remote** directory is only used during the optimization to upload simulation files to the remote calculation server and is, otherwise, empty.
- The **SLALOM Python files** with:
  - **slalom.py**: the SLALOM startup module. This module set the parameters, the optimization method and control the whole process.
  - **slalomCore.py**: the SLALOM core class, implementing the mathematical optimization methods provided by the SciPy package and controlling the device simulation engine, e.g. Silvaco®. As a class with well defined and modularized functionalities it can be easily externded by creating a new inherited class. This inheritance mechanism is very useful and allows modularity, clearness, reliability and keeping unchanged a highly tested codebase.
  - **slalomDevice.py**: class defining a set of devices (e.g. *InGaN_PN*, *CZTS_NP*, etc.) for easier and more robust optimization work. For every project, this class can be reimplemented to include any set of relevant devices.
  - **slalomDeviceGui.py**: class providing a useful interface to create a new device type. It is only used if the device type specified in *slalom.py* is not defined in *slalomDevice.py*. *slalomDeviceGui.py* uses the Tkinter graphical toolkit that is already installed on the client (this is generally the case, except for some CentOS or Red Hat machines). If Tkinter is not installed, this class is not used.
  - **slalomMonitor.py**: used to monitor the optimizer either in client / server configuration using SSH or locally if it runs on the same machine than the optimizer. slalomMonitor uses the slalomWindow class that provide visualisation and control functionalities. If Tkinter is not installed, this class is not used.
  - **slalomSimulator.py**: class interfacing the solar cell simulator and encapsulates functionality specific to the simulator, e.g. Silvaco®, and can be easily extended to include any simulator that can be launched from the command line and output results in text files (i.e. any well designed simulator).
  - **slalomWindow.py**: common class providing the visualisation and control functionalities used by the SLALOM GUI part, only available if Tkinter is installed.

SLALOM include predefined solar cell structures for the Silvaco® simulator. Among these structures are an Indium Gallium Nitride (InGaN) and Copper Zinc Tin Sulfide (CZTS) solar cells.

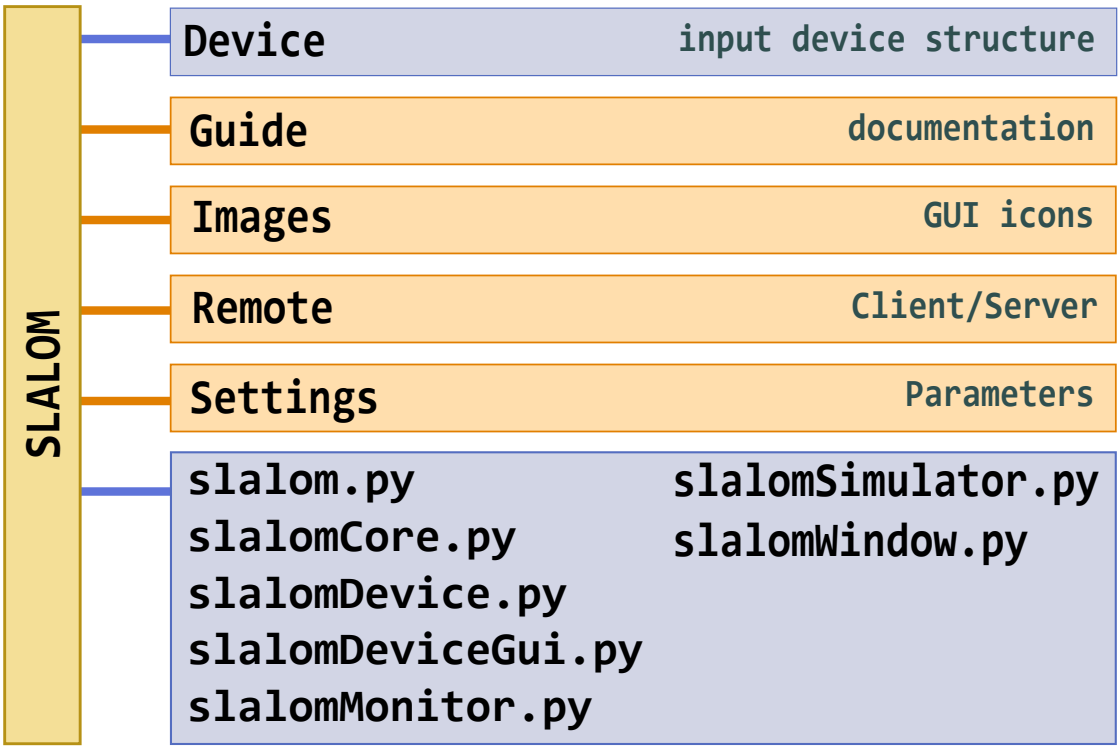


Figure 2: The SLALOM distribution structure. The Device directory contains the file :TODO:

2.2 Case Study: InGaN PN Solar Cell

The InGaN PN solar cell simulation files for the Silvaco® simulator are located in the *SLALOM/Device* directory.

For Silvaco®, The Deckbuild *InGaN_PN.in* can be edited, tested and modified in the Deckbuild editor. As underlined previously, all the material parameters and models are defined in C source files for modularity, simplicity and reliability.
These C files can be modified, updated and adapted to the state-of-the-art.

After checking the input (*InGaN_PN.in*) and C files for Silvaco®, open the SLALOM device class file: *slalomDevice.py*. This class define a set of devices for a solar cell technology of interest to the user. By default it contains the InGaN PN solar cell definition, among others. This class is very useful by saving a lot of time while permitting a far more reliable optimization work.
In *slalomDevice.py*, each device type is defined by:

- its associated input filename (class member *inputFilename*)
- the parameters name array (*paramName*) as defined in the simulator input file. For Silvaco® for example, every parameter name in the array corresponds to a variable definition in the Deckbuild input file as: *set paramName = parameValue*. Example: *set NLayerThick = 1.0*.
- start values (*paramStart*). Array containing the lower limit for each parameter.
- initial values (*paramInit*). Array containing the initial value for each parameter.
- final values (*paramEnd*). Array containing the upper limit for each parameter.

- the number of points (*paramPoints*). Array containing, for each parameter, the number of points between start and final values.
- the variation type (*paramLogscale*). For each parameter, set to *True* for logarithmic variation (e.g. for doping) and *False* for linear variation (e.g. for thickness, bandgap, etc.).
- the normalization values (*paramNorm*). Array of values used in the optimization, corresponding to a "standard" value for each parameter (e.g. 10^{17} for doping).
- numeric format (*paramFormat*). C-style format for each parameter. The simple rule is to use "$\%.8f$" for linearly varying parameters (e.g. thickness) and "$\%.5e$" for logarithmic variation (e.g. for doping). *paramFormatShort* corresponds to the compact version of *paramFormat*.

In the present case, we want to optimize the InGaN PN solar cell with respect to five parameters: The N-layer doping concentration and thickness, the P-layer doping concentration and thickness and the Indium composition. The thickness and composition vary linearly while the doping variation is logarithmic.

Keep *slalomDevice.py* untouched and open the SLALOM startup file: (*slalom.py*). This module control the whole optimization process: set the device parameters, the mathematical method, the communication parameters and so on. Usually by properly defining devices in *slalomDevice.py* it is only necessary to set the device type and SSH parameters in *slalom.py*. Check and set the folowing values to reflect your installation (specifically *currentDir*, *remoteDir* and *SSH parameters*):

```
deviceSimulator = "atlas"

# your local device directory
currentDir = "N:\TCAD\SLALOM\Device\Silvaco\"

# remoteDir is the name of the remote directory...
# ... used when using SSH to connect to a remote...
# ... server where the simulator is installed.
remoteDir = "/home/user/SLALOM/Device/Silvaco/"

# put here the SSH host configured previously
remoteSSHhost = "user@slalom"

# the device type defined in slalomDevice.py
deviceType = "InGaN_PN"

optimType = "Optim"

minimizeMethod = "SLSQP"
```

If you are using the local configuration (simulator and SLALOM installed locally on the same machine) put an empty value in *remoteDir* and *remoteSSHhost*:

```
remoteDir = ""

remoteSSHhost = ""
```

After setting these parameters in *slalom.py*, open a Linux console (or Gitbash under Windows) and *cd* to the SLALOM directory...:

cd /SLALOM

... and start the optimizer:

python slalom.py

The optimization process will start and the SLALOM monitor launched (if Tkinter installed). The monitor gives the solar cell performances (open-circuit voltage, short-circuit current and efficiency). It gives also the current-voltage characteristic (click the *View JV* button), the spectral response (*View SR* button) and the raw values for every set of optimization parameters (*View data* button), permitting to precisely and interactively follow the process. If the monitor is not started, open a console and run it:
python slalomMonitor.py

The optimizer can be stopped at any time by clicking the *Stop* button. The time statistics (simulation mean, minimum and maximum duration and so on) are shown in realtime. You can close the monitor (without stopping the optimizer) by clicking the *Close* button (or the window close button) and restart it later with the command: *python slalomMonitor.py*

When the optimization is finished, the results are zipped and downloaded locally for in-detail analysis. On the main server used to develop SLALOM (a Red Hat Linux server with two 8-core Xeon processors and 32 GB of RAM), every simulation takes one minute up to ten minutes (strongly depending on device, mesh, parameters, required tolerance, solver, etc.) and the whole optimization process takes one hour up to fifty hours.

The whole set of results is stored in a sub-directory in the *output* directory with a name prefixed with the date and device type (for example: *20170210_1755_InGaN_PN*). Each result file name is prefixed with *simuloutput_* and the iteration index and contains, as header, the corresponding set of parameters. If the first line in these files is *v Atlas* then they can be visualized by the Silvaco® Tonyplot tool. Other files can also be plotted using Tonyplot providing that the header lines (beginning with #) are removed since Tonyplot unfortunately do not ignore comment lines. In all cases these files are in plain ASCII format and can be visualized and treated by any plotting and analysis software (just skip the header lines to plot the included data).
The file(s) **prefixed** with:

- *simuloutput_all* correspond to distribution (electric field, potential, etc.) as calculated by Silvaco® for each set of parameters choosen by the optimizer.
- *simuloutput_band* corresponds to the band diagram at equilibrium.
- *simuloutput_mesh_all* correspond to the mesh input generated by the simulator.
- *simuloutput_popt* contains the incident optical power density in mW/cm^2.
- *simuloutput_jv* correspond to the current-voltage characteristic for each set of parameters choosen by the optimizer. The files prefixed with *simuloutput_jvp* correspond also to current-voltage characteristic for each set of parameters choosen by the optimizer but only up to the open-circuit voltage.
- *simuloutput_pv* correspond to power-voltage characteristic.

- *simuloutput_photocurrent* correspond to the available and source photocurrent, used to calculate the external and internal quantum efficiencies.
- *simuloutput_spectralresponse_eqe* and *simuloutput_spectralresponse_iqe* for the external and the internal quantum efficiency spectra.
- *simuloutput_optimized* contains all the points set by the optimizer and, for each point, the solar cell performances.
- *simuloutput_log* contains the SLALOM output including all the messages printed out during the optimization.
- *simuloutput_stdout* contains the simulator standard output including all the messages printed out during the simulation. Useful for debugging purpose, if the simulator is diverging for example.

The result directory is self-contained and includes even the input file and C model files used to produce the results it contains, for reliable analysis and archiving. This directory is zipped at the end of the optimization process.

In this study case, SLALOM was started with the default optimization method but it has three modes (represented by the *optimType* variable in *slalom.py*):

- **Snap**: only one simulation is performed, corresponding to the set of parameters *paramInit* defined in *slalomDevice.py*. This mode is usually used to test one particular set of parameters before performing a full optimization.
- **Brute**: Brute force optimization: for each parameter, an iteration is done between *paramStart* and *paramEnd* with the number of points specified by *paramPoints* as defined in *slalomDevice.py*. This mode and can take a very long time to perform. For example with four parameters and ten points per parameter, 10^4 simulations will be performed with a overall duration of 10^4 minutes (considering roughly one minute per simulation) and 166 hours or seven days!
- **Optim**: Mathematical optimization using one of the following methods:
  - **L-BFGS-B**: a variant of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, a quasi - Newton iterative method to solve nonlinear optimization problems [8]. The **L** stands for **L**imited memory as this variant use a modified algorithm that need to store in memory only a part of the Hessian matrix. In this method name, **B** stands for **B**ound constrained, since this variant is adapted to handle constraints on the parameters with a specified variation range, condition necessary in solar cell optimization since each parameter is constrained by the physics and the current technology.
  - **SLSQP**: Sequential Least Squares Programming, a Newton method for constrained problems using a Hessian approximation [9]. The SLSQP needs less evaluations and usually converges faster, for the drift-diffusion simulation considered here, than L-BFGS-B.

  Other methods can be easily integrated in SLALOM. The two choosen methods were practically demonstrated to have a good compromise between robustness and speed when applied to our complex drif-diffusion problem in solar cells.

The computing duration of the mathematical optimization methods is at least two orders of magnitude less than the brute force method. For the case taken previously (seven days needed by brute force) it will take only three hours or less for the L-BFGS-B method to find the optimal set of parameters. Nonetheless, the mathematical optimization methods, even very fast, will not necessarily find the absolute

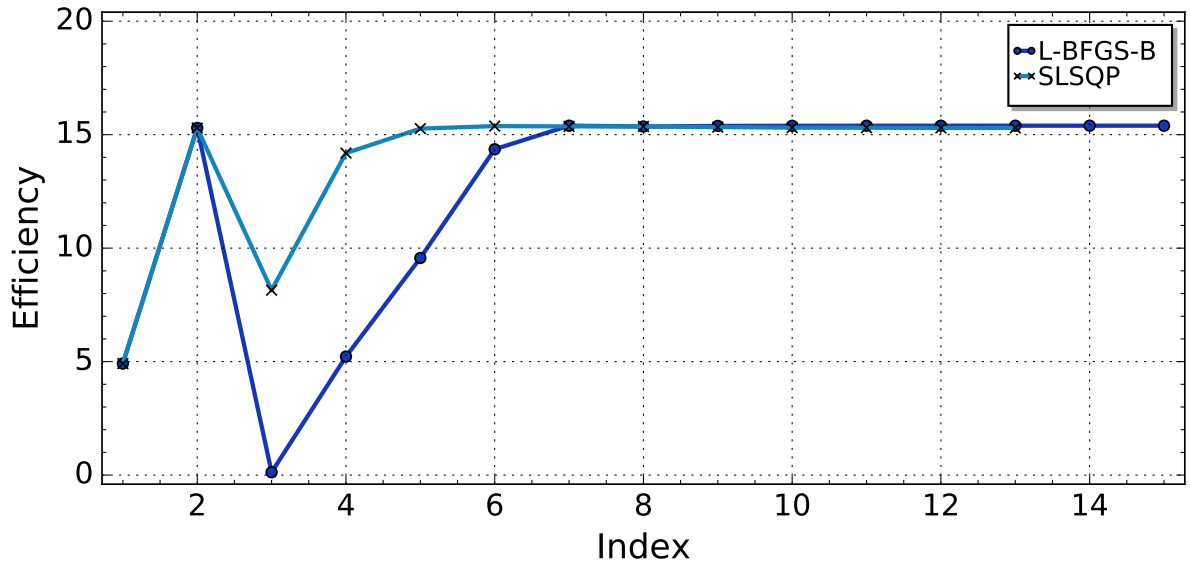
(global) optimum since they can stick on a local one, specially for this complex problem involving the resolution of the nonlinear coupled drif-diffusion equations. This is typically an example of the so-called non-convex optimization problem. There are at least two ways of dealing with this common issue. Firstly we can give initial set of parameters close to what we expect to be the global optimum. This can be generally done according to the user experience. The second method is to perform few optimizations starting each time from a random set of parameters. The absolute optimum will be given by the global optimal efficiency among these local optima. One effective strategy the we use is to calculate efficiency for a dozen of points (or more) randomly choosen in a well-defined variation domain and give the best one to the optimizer as the initial set of parameters. Another strategy, only feasible for three parameters or less, is to do a rough brute force optimization (for exemple 4 points per parameters and then 64 total number of points for three parameters) and choose the range where the brute force gives the best efficiency as the optimizer variation domain. The convergence of the optimizer depends also on the internal implementation and numerical parameters, mainly the required tolerance (*tolerance* in *slalomCore.py*), the maximum number of iterations and the Jacobian calculation method and resolution (*jaceps* in *slalomCore.py*).The tolerance default value can be changed to handle particular situations. If increased, the convergence will be faster but the probability to stick on a local optimum becomes higher. On the contrary, if the tolerance is decreased, the convergence becomes slower but the probability to find the absolute optimum usually increases.

As illustration, Figure 3 shows the efficiency of the InGaN PN solar cell as determined by the optimizer using the L-BFGS-B and SLSQP methods with the initial point choosen far from the global optimum (first case, Figure 3a) and near it (second case, Figure 3b). Each index corresponds to a set of parameters choosen by the optimizer in the variation domain defined in *slalomDevice.py*. None of the strategies described above was used. Both methods converge to same set of optimal parameters in about two hours for L-BFGS-B and one hour for SLSQP on a Red Hat Linux server with two 8-core Xeon processors and 32 GB of RAM. Usually, for the tested solar cell structures, SLSQP is faster but, as other methods, can optimize to a local point.
When the initial point was choosen in the close vicinity of the global optimum (second case, 3b), the algorithm obviously converges faster to the optimum (fifty minutes for SLSQP and ninety minutes for L-BFGS-B). This point illustrates the fact that it is necessary to use a "reasonable" strategy, as explained previously, to find the global solar cell optimal efficiency considering the complexity of the underlying physical model (drift-diffusion) and its non-convex nature. Nonetheless, finding a local optimum in a constrained parameters range could be interesting for technological and feasibility reasons (a feasible lower optimum is practically more interesting than a higher unreachable one).

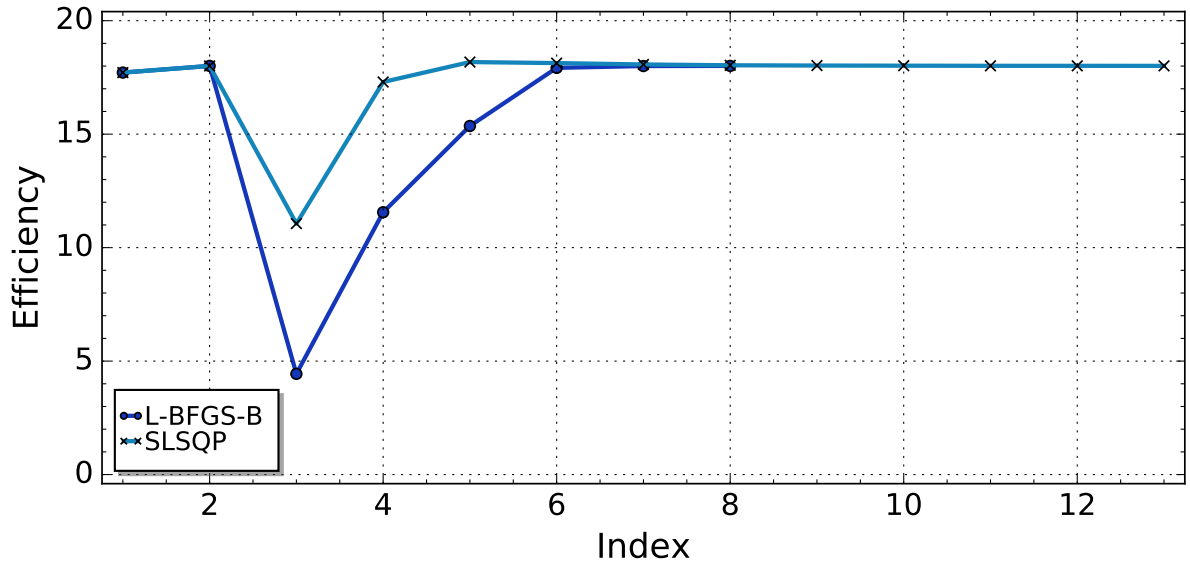
The optimal set of parameters ("point") lies in the vicinity of the following set of values (giving an optimal efficiency of about 18%, a short-circuit current of $26.8 mA/cm^2$, an open-circuit voltage of $0.85V$ and a fill factor of 78%):
(P-layer thickness of $0.01\mu m$, N-layer thickness of $1\mu m$, P-layer doping of $10^{19} cm^{-3}$, N-layer doping of $4 \times 10^{16} cm^{-3}$, Indium composition of 56%).

Figure 4a shows the current-voltage characteristic near optimal point while Figure 4b plots the external and internal quantum efficiency spectra.
When using the brute force method with only 5 points per parameter, the total number of points is 3125 ($= 5^5$). The total duration is about 100 hours on the same machine. The resolution is equal to the range divided by the number of points per parameter. If seeking higher resolution, the brute force duration become completey infeasible: it varies as n^m where n is the number of points and m the number of parameters. For thickness, for example, the resolution is about $0.2\mu m$ (for 5 points per parameter) meaning that the optimal thickness is known within this uncertainty. For the optimization methods (L-BFGS-B and SLSQP) the resolution is determined by the Jacobian step which is, in this case, equal to the range divided by 50, meaning $0.02\mu m$, ten times better than previously. Therefore, with a far

better resolution, the L-BFGS-B and SLSQP methods are *at least* two orders of magnitude faster than the brute force method.

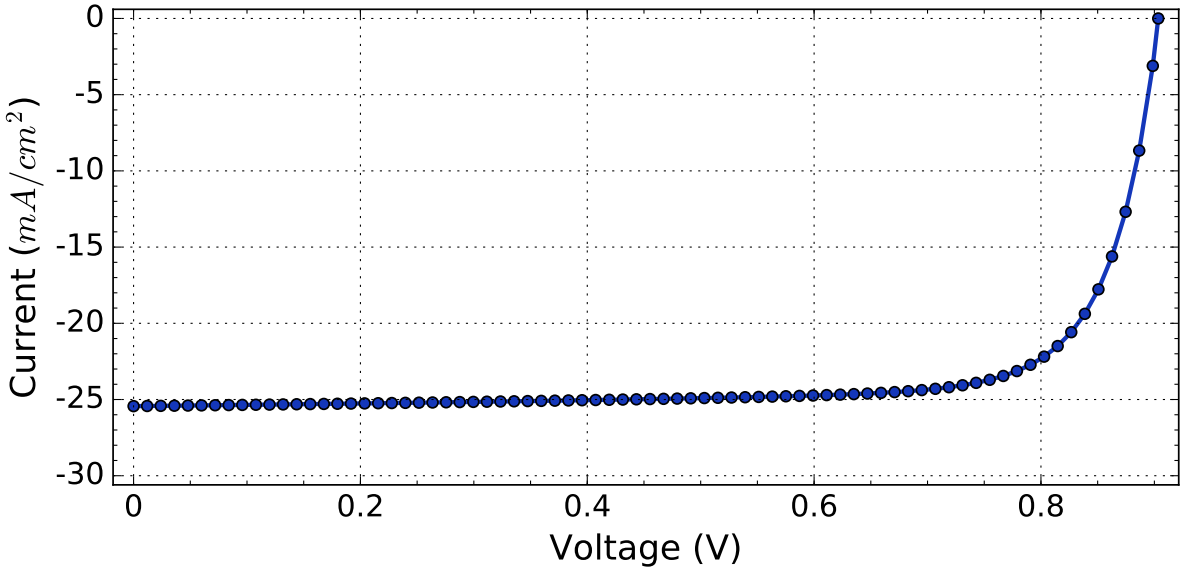


(a) Initial point choosen relatively far from the global optimum.

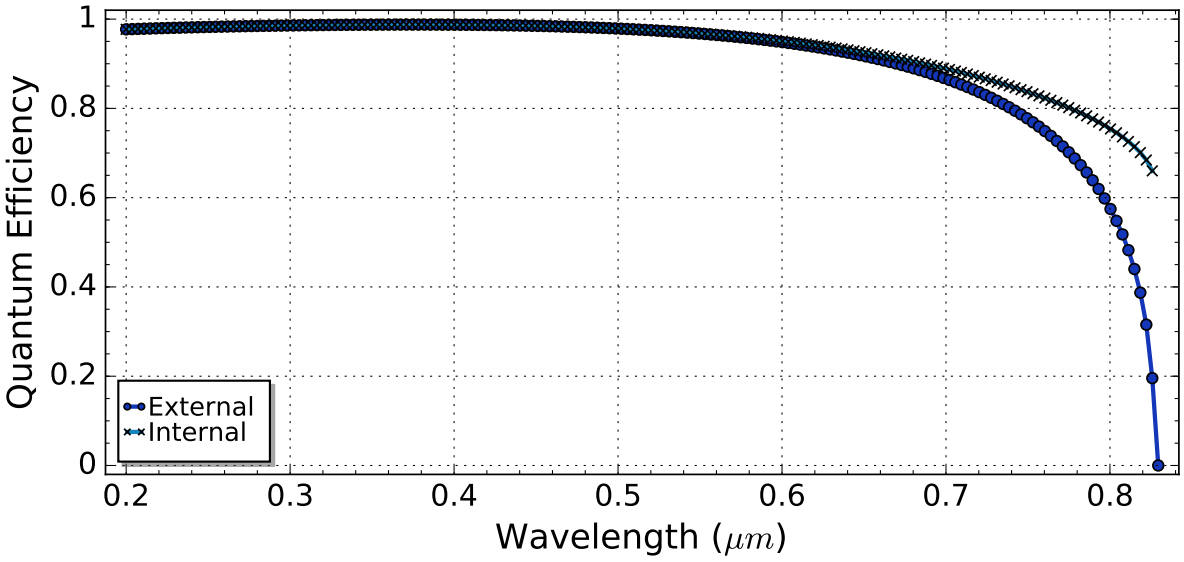


(b) Initial point choosen in the close vicinity of the global optimum.

Figure 3: Efficiency of the InGaN PN solar cell determined by SLALOM using the L-BFGS-B and SLSQP methods.



(a) Current-voltage characteristic.



(b) External and internal quantum efficiency spectra.

Figure 4: Current-voltage characteristic and external and internal quantum efficiency spectra of the InGaN PN solar cell at the optimum.

References

[1] Sidi Ould Saad Hamady and Nicolas Fressengeas. SLALOM: Open-Source Solar Cell Multivariate Optimizer. *TODO*, 10(10-20):10, 2017.

[2] Guido van Rossum. Python programming language. *https://www.python.org/*, 2017.

[3] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[4] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. `http://www.scipy.org/`.

[5] John D Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[6] John W Shipman. Tkinter 8.5 reference: a GUI for Python. *http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html*, 2013.

[7] Johannes Schindelin. Git for windows: the windows port of git. *https://git-for-windows.github.io/*, 2017.

[8] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[9] Stephen J Wright and Jorge Nocedal. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.