Sesame Documentation

Benoit H. Gaury, Paul M. Haney

Sep 30, 2019

CONTENTS

About Sesame

Sesame is a Python3 package for solving the drift diffusion Poisson equations for multi-dimensional systems using finite differences.

Support for this project comes from the U.S. National Institute of Standards and Technology and the University of Maryland.

Summary

Version 2.0

Date Sep 30, 2019

Authors Benoit H. Gaury, Paul M. Haney

CHAPTER ONE

PRELIMINARIES

1.1 About Sesame

Sesame is a Python3 package for solving the drift diffusion Poisson equations for multi-dimensional systems using finite differences.

The software computes the steady state of a semiconductor between two contacts, and subject to voltage bias and/or illumination. It was developed to study polycrystalline solar cells containing charged interfaces (grain boundaries, sample surface). Sesame handles single charged states and continuum of defect states alike.

1.2 Installation instructions (for experienced users)

This section documents how to build Sesame.

1.2.1 Prerequisites

Building Sesame requires

- Python 3.4 or above,
- SciPy 0.9 or newer,
- LAPACK and BLAS, (other options are the free OpenBLAS or the nonfree MKL can be used.)

The following software is highly recommended though not strictly required:

- Matplotlib 1.1 or newer, for Sesame's plotting routines. Matplotlib is required when using the graphical interface of Sesame.
- MUMPS, a sparse linear algebra library that will in many cases speed up Sesame several times and reduce the memory footprint. (Sesame uses only the sequential, single core version of MUMPS. The advantages due to MUMPS as used by Sesame are thus independent of the number of CPU cores of the machine on which Sesame runs.)
- An environment which allows to compile Python extensions written in C, C++ and Fortran.

The graphical user interface requires PyQt5.

For users with no python installation, a convenient standalone installation which automatically includes all of the requisiste libraries and packages is Anaconda .

Sesame may be downloaded from https://github.com/usnistgov

1.2.2 Generic instructions

Standard build and install

Sesame can be built and installed following the usual Python conventions by running the following commands in the root directory of the Sesame distribution:

```
python setup.py build
python setup.py install
```

Depending on your system, you might have to run the second command with administrator privileges. The installation step can be done locally either by using the `--user` prefix:

```
python setup.py install --user
```

or by specifying the location where to install the package files with `--prefix=/path/of/directory`.

The tutorial examples can be found in the directory `examples` inside the root directory of the Sesame source distribution.

Build configuration

The setup script of Sesame parses the file `setup.cfg` in the root directory of the distribution to know if the graphical user interface should be installed, and whether or not to link Sesame against the MUMPS library.

The graphical user interface will be installed if `use = True` in the `GUI` section. The default is to install it (use `use = False` to avoid install). The `mumps` section provides the paths to relevant directories where MUMPS is installed. By default this section is commented out and MUMPS is not used.

Building the documentation

To build the documentation, the Sphinx documentation generator is required (version 1.4 or newer) with `numpydoc` extension (version 0.5 or newer), and Latex.

HTML documentation is built by entering the `doc` sub-directory of the Sesame package and executing `make html`. Open the file `index.html` in the directory `build/html` with a web browser to access the documentation. PDF documentation is generated by executing `make latex` followed by `make latexpdf`. The pdf file is generated in `build/latex`.

Because of some quirks of how Sphinx works, it might be necessary to execute `make clean` between building HTML and PDF documentation. If this is not done, Sphinx may mistakenly use PNG files for PDF output or other problems may appear.

As an alternative if `make` is not available, the HTML documentation can be built using the command from the root directory:

```
python setup.py build_sphinx
```

The documentation is produced in `doc/build/html`. To build the PDF file:

```
python setup.py build_sphinx -b latex
cd doc/build/latex
make all-pdf
```

The resulting PDF is produced in `doc/build/latex`.

1.2.3 Hints for specific platforms

Unix-like systems (GNU/Linux)

Sesame should run on all recent Unix-like systems.

1. Install the required packages.
2. Inside the Sesame source distribution's root directory run

```
python setup.py build
sudo python setup.py install
```

Run `python setup.py --help install` for installation options.

Microsoft Windows

The generic installation instructions given above also apply on Windows. However, since the only recommended way to compile Python extensions on Windows is using Visual C++, we are not able to provide guidelines as to how to build with the MUMPS library.

1.3 Installation instructions (for beginners)

This section documents how to build Sesame for those with zero Python experience.

1.3.1 Installing Python

For users with no Python installation, there are a number of convenient standalone installations which automatically includes all of the requisiste libraries and packages, including:

- Anaconda
- Canopy
- Pythonxy

These can be installed on any operating system (Windows, GNU/Linux, MacOS). This page walks through the process using Anaconda in a Windows environment.

First download and install Anaconda, using the default settings. After installation, you'll find a new folder with various programs in the windows `Start` button folder: `Start` → `All Programs` → `Anaconda`.

1.3.2 Downloading and Installing Sesame (on Windows)

To obtain Sesame, first open the Anaconda Prompt: `Start` → `All Programs` → `Anaconda` → `Anaconda Prompt`. A command line should appear (a primer on using the Windows command line can be found here). Sesame is downloaded using `git`. Make sure `git` is installed by first typing:

```
conda install -c anaconda git
```

Once you have `git`, obtain Sesame with the command:

```
git clone http://github.com/usnistgov/sesame
```

The git repository of Sesame is cloned in the directory where the command was issued. Enter the sesame repository, build and install Sesame with the commands:

```
cd sesame
python setup.py build
python setup.py install --user
```

The essential procedure for installing for other operating systems is the same.

1.3.3 Running Sesame

Upon installation, you can try some of the examples. Navigate the `examples` directory:

```
cd sesame\examples
```

Running a sesame python script is done with the command:

```
python 1dpn.py
```

The GUI is launched with the command from the sesame install directory:

```
python app.py
```

Note: some distributions of Anaconda are packaged with older versions of PyQt. If you find that graphics do not render, or that the GUI does not run, it may be because PyQt is not up to date (Sesame uses PyQt5). You can try this command to update PyQt if you have difficulty using graphics:

```
conda install -c anaconda pyqt
```

1.4 Authors of Sesame

The authors of Sesame are

- Benoit Gaury (NIST / University of Maryland)
- Paul M. Haney (NIST)

Benoit Gaury acknowledges support under the Cooperative Research Agreement between the University of Maryland and the National Institute of Standards and Technology Center for Nanoscale Science and Technology, Award 70NANB14H209, through the University of Maryland.

1.5 Citing Sesame

We provide Sesame as a free software to the research community. If you have used Sesame for a publication, we ask that you mention the software explicitly by name in the main text, and cite the introduction paper:

B. Gaury, Y. Sun, P. Bermel, P. M. Haney, *Sesame: A 2-dimensional solar cell modeling tool*, Sol. Energ. Mat. Sol. Cells 198, 53-62 (2019)

1.6 Sesame license

CHAPTER

TWO

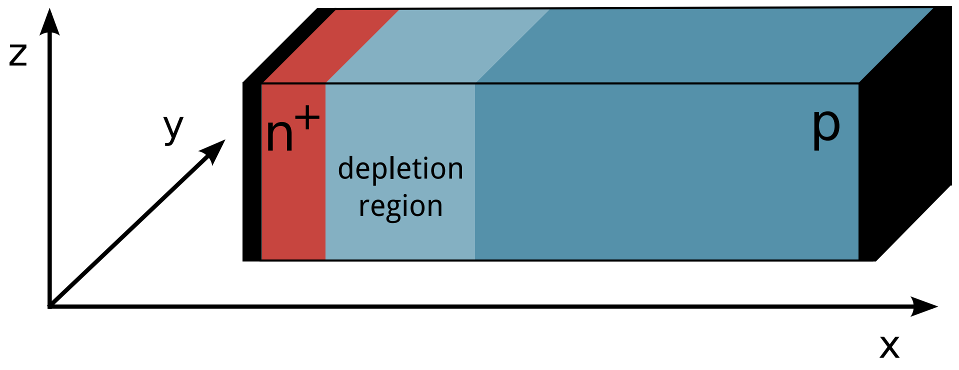
ANALYTICAL MODEL AND NUMERICAL IMPLEMENTATION

2.1 Physical model

Here we present the geometry with the system of coordinates that Sesame assumes, and the set of equations that it solves.

2.1.1 Geometry and governing equations

Our model system is shown below. It is a semiconductor device connected to two contacts at $x = 0$ and $x = L$. The doped regions are drawn for the example only, any doping profile can be considered.



The steady state of this system under nonequilibrium conditions is described by the drift-diffusion-Poisson equations:

$$\begin{aligned} \vec{\nabla}\cdot\vec{J_n} &= -q(G-R) \\ \vec{\nabla}\cdot\vec{J_p} &= q(G-R) \\ \vec{\nabla}\cdot(\epsilon\vec{\nabla}\phi) &= -\rho/\epsilon_0 \end{aligned} \tag{2.1}$$

with the currents

$$\begin{aligned} \vec{J_n} &= -q\mu_n n\vec{\nabla}\phi + qD_n\vec{\nabla}n \\ \vec{J_p} &= -q\mu_p p\vec{\nabla}\phi - qD_p\vec{\nabla}p \end{aligned} \tag{2.2}$$

where n, p are the electron and hole number densities, and ϕ is the electrostatic potential. $J_{n(p)}$ is the charge current density of electrons (holes). Here q is the absolute value of the electron charge. ρ is the local charge, ϵ is the dielectric constant of the material, and ϵ_0 is the permittivity of free space. $\mu_{n,p}$ is the electron/hole mobility, and is related the diffusion $D_{n,p}$ by $D_{n,p} = k_BT\mu_{n,p}/q$, where k_B is Boltzmann's constant and T is the temperature. G is the generation rate density, R is the recombination and we denote the net generation rate $U = G - R$. The natural length scale is the Debye length, given by $\lambda = \epsilon_0 k_B T/(q^2 N)$, where N is the concentration relevant to the problem. Combining Eqs. (2.1) and Eqs. (2.2), and scaling by the Debye length leads to the following system

$$\begin{aligned}
\tilde{\vec{\nabla}} \cdot \left(-\bar{n}\tilde{\vec{\nabla}}\bar{\phi} + \tilde{\vec{\nabla}}\bar{n}\right) &= \bar{U} \\
\tilde{\vec{\nabla}} \cdot \left(-\bar{p}\tilde{\vec{\nabla}}\bar{\phi} - \tilde{\vec{\nabla}}\bar{p}\right) &= -\bar{U} \\
\tilde{\vec{\nabla}} \cdot (\epsilon\vec{\nabla}\bar{\phi}) &= (\bar{n} - \bar{p}) + (\bar{N_A} - \bar{N_D})
\end{aligned}$$

where $\tilde{\vec{\nabla}}$ is the dimensionless spatial first derivative operator. $\bar{N}_{A,(D)}$ are the dimensionless ionized acceptor (donor) impurity concentration. The dimensionless variables are given below:

$$\begin{aligned}
\bar{\phi} &= \frac{q\phi}{k_B T} \\
\bar{n} &= n/N \\
\bar{p} &= p/N \\
\bar{U} &= \frac{U\lambda^2}{ND} \\
\bar{t} &= t\frac{q\mu N}{\epsilon_0} \\
\bar{J}_{n,p} &= J_{n,p}\frac{\lambda}{qDN}
\end{aligned}$$

with $D = k_B T\mu/q$ a diffusion coefficient corresponding to our choice of scaling for the mobility $\mu = 1\ \mathrm{cm}^2/(\mathrm{V}\cdot\mathrm{s})$. See the `Scaling()` class for the implementation of these scalings.

We suppose that the bulk recombination is through three mechanisms: Shockley-Read-Hall, radiative and Auger. The Shockley-Read-Hall recombination takes the form

$$R_{\mathrm{SRH}} = \frac{np - n_i^2}{\tau_p(n + n_1) + \tau_n(p + p_1)}$$

where $n_i^2 = N_C N_V e^{-E_g/k_B T}, n_1 = n_i e^{E_T/k_B T}, p_1 = n_i e^{-E_T/k_B T}$, where E_T is the energy level of the trap state measured from the intrinsic energy level, N_C (N_V) is the conduction (valence) band effective density of states. The equilibrium Fermi energy at which $n = p = n_i$ is the intrinsic energy level E_i. $\tau_{n,(p)}$ is the bulk lifetime for electrons (holes). It is given by

$$\tau_{n,p} = \frac{1}{N_T v_{n,p}^{\mathrm{th}} \sigma_{n,p}} \tag{2.3}$$

where N_T is the three-dimensional trap density, $v_{n,p}^{\mathrm{th}}$ is the thermal velocity of carriers: $v_{n,p}^{\mathrm{th}} = 3k_B T/m_{n,p}$, and $\sigma_{n,p}$ is the capture cross-section for (electrons, holes).

The radiative recombination has the form

$$R_{\mathrm{rad}} = B(np - n_i^2)$$

where B is the radiative recombination coefficient of the material. The Auger mechanism has the form

$$R_{\mathrm{A}} = (C_n n + C_p p)(np - n_i^2)$$

where C_n (C_p) is the electron (hole) Auger coefficient.

2.1.2 Extended line and plane defects

Additional charged defects can be added to the system to simulate, for example, grain boundaries or sample surfaces in a semiconductor. These extended planar defects occupy a reduced dimensionality space: a point in a 1D model, a line in a 2D model). The extended defect energy level spectrum can be discrete or continuous. For a discrete spectrum, we label a defect with the subscript d. The occupancy of the defect level f_d is given by[1]

$$f_d = \frac{S_n n + S_p p_d}{S_n(n + n_d) + S_p(p + p_d)}$$

where n (p) is the electron (hole) density at the defect location, S_n, S_p are recombination velocity parameters for electrons and holes respectively. n_d and p_d are

$$\bar{n}_d = n_i e^{E_d/k_BT}$$
$$\bar{p}_d = n_i e^{-E_d/k_BT}$$

where E_d is calculated from the intrinsic Fermi level E_i. The defect recombination is of Shockley-Read-Hall form:

$$R_d = \frac{S_n S_p (np - n_i^2)}{S_n(n + n_d) + S_p(p + p_d)}.$$

The charge density q_d given by a single defect depends on the defect type (acceptor or donor)

$$q_d = q\rho_d \times \begin{cases} (1 - f_d) & \text{donor} \\ (-f_d) & \text{acceptor} \end{cases}$$

where ρ_d is the defect density of state at energy E_d. S_n, S_p and ρ_d are related to the electron and hole capture cross sections σ_n, σ_p of the defect level by $S_{n,p} = \sigma_{n,p} v^{\text{th}}_{n,p} \rho_d$, where $v^{\text{th}}_{n,p}$ is the electron (hole) thermal velocity. Multiple defects are described by summing over defect label d, or performing an integral over a continuous defect spectrum.

2.1.3 Carrier densities and quasi-Fermi levels

Despite their apparent simplicity, Eqs. (2.1) are numerically challenging to solve. We next discuss a slightly different form of these same equations which is convenient to use for numerical solutions. We introduce the concept of quasi-Fermi level for electrons and holes (denoted by E_{F_n} and E_{F_p} respectively). The carrier density is related to these quantities as

$$\begin{aligned} n(x,y,z) &= N_C e^{(E_{F_n}(x,y,z) + q\phi(x,y,z) + \chi(x,y,z))/k_BT} \\ p(x,y,z) &= N_V e^{\left(-E_{F_p}(x,y,z) - q\phi(x,y,z) - E_g - \chi(x,y,z)\right)/k_BT} \end{aligned} \tag{2.4}$$

where the term χ is the electron affinity, ϕ is the electrostatic potential, and E_g is the bandgap. Note that all of these quantities may vary with position. Quasi-Fermi levels are convenient in part because they guarantee that carrier densities are always positive. While carrier densities vary by many orders of magnitude, quasi-Fermi levels require much less variation to describe the system.

The electron and hole current can be shown to be proportional to the spatial gradient of the quasi-Fermi level

$$\vec{J}_n = q\mu_n n \vec{\nabla} E_{F_n}$$
$$\vec{J}_p = q\mu_p p \vec{\nabla} E_{F_p}$$

These relations for the currents will be used in the discretization of Eq. (2.1).

[1]

23. Shockley, W. T. Read, Jr., *Phys. Rev.*, **87**, 835 (1952).

2.1.4 Boundary conditions at the contacts

Equilibrium boundary conditions

For a given system, Sesame first solves the equilibrium problem. In equilibrium, the quasi-Fermi level of electrons and holes are equal and spatially constant. We choose an energy reference such that in equilibrium, $E_F = E_{F_p} = E_{F_n} = 0$. The equilibrium problem is therefore reduced to a single variable ϕ. Sesame employs both Dirichlet and Neumann equilibrium boundary conditions for ϕ, which we discuss next.

Dirichlet boundary conditions

Sesame uses Dirichlet boundary conditions as the default. This is the appropriate choice when the equilibrium charge density at the contacts is known *a priori*, and applies for Ohmic and ideal Schottky contacts. For Ohmic boundary conditions, the carrier density is assumed to be equal and opposite to the ionized dopant density at the contact. For an n-type contact with N_D ionized donors at the $x = 0$ contact, Eq. (2.4) yields the expression for $\phi^{eq}(x = 0)$:

$$\phi^{eq}(0, y, z) = k_B T \ln (N_D/N_C) - \chi(0, y, z)$$

Similar reasoning yields expressions for ϕ^{eq} for p-type doping and at the $x = L$ contact. For Schottky contacts, we assume that the Fermi level at the contact is equal to the Fermi level of the metal. This implies that the equilibrium electron density is $N_C \exp[-(\Phi_M - \chi)/k_B T]$ where Φ_M is the work function of the metal contact. Eq. (2.4) then yields the expression for ϕ^{eq} (shown here for the $x = 0$ contact):

$$\phi^{eq}(0, y, z) = -\Phi_M|_{x=0\ contact}$$

An identical expression applies for the $x = L$ contact.

Neumann boundary conditions

Sesame also has an option for Neumann boundary conditions, where it is assumed that the electrostatic field at the contact vanishes:

$$\frac{\partial \phi^{eq}}{\partial x}(0, y, z) = \frac{\partial \phi^{eq}}{\partial x}(L, y, z) = 0 \tag{2.5}$$

The equilibrium potential ϕ^{eq} determines the equilibrium densities n_{eq}, p_{eq} according to Eqs. (2.4) with $E_{F_n} = E_{F_p} = 0$.

Out of equilibrium boundary conditions

Out of thermal equilibrium, we impose Dirichlet boundary conditions on the electrostatic potential. For example, in the presence of an applied bias V at $x = L$, the boundary conditions are

$$\begin{aligned} \phi(0, y, z) &= \phi^{eq}(0, y, z) \\ \phi(L, y, z) &= \phi^{eq}(L, y, z) + qV \end{aligned}$$

For the drift-diffusion equations, the boundary conditions for carriers at charge-collecting contacts are typically parameterized with the surface recombination velocities for electrons and holes at the contacts, denoted respectively by S_{c_p} and S_{c_n}

$$\begin{aligned}
\vec{J}_n(0,y,z)\cdot\vec{u}_x &= qS_{c_n}(n(0,y,z)-n_{\mathrm{eq}}(0,y,z))\\
\vec{J}_p(0,y,z)\cdot\vec{u}_x &= -qS_{c_p}(p(0,y,z)-p_{\mathrm{eq}}(0,y,z))\\
\vec{J}_n(L,y,z)\cdot\vec{u}_x &= -qS_{c_n}(n(L,y,z)-n_{\mathrm{eq}}(L,y,z))\\
\vec{J}_p(L,y,z)\cdot\vec{u}_x &= qS_{c_p}(p(L,y,z)-p_{\mathrm{eq}}(L,y,z))
\end{aligned} \tag{2.6}$$

where $n(p)_{\mathrm{eq}}$ is the thermal equilibrium electron (hole) density.

References

2.2 Numerical treatment of the drift diffusion Poisson equations

In this section we present the procedure followed to discretize the drift diffusion Poisson set of equations, the algorithm used to solve it and its implementation.

2.2.1 Scharfetter-Gummel scheme

To solve the drift diffusion Poisson equations numerically, we utilize a simple spatial discretization. Recall that densities are defined on sites, and fluxes (such as current flux, electric field flux) are defined on links. It's important to note that *sites* and *links* in the discretized grid are fundamentally different objects, as shown in the figure below.

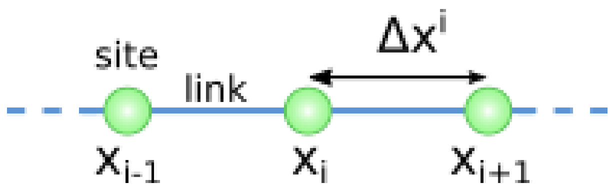


Fig. 1: Sites versus links. We take the indexing convention that Δx^i represents the space between sites i and $i+1$.

We consider a one-dimensional system to illustrate the model discretization. First, we want to rewrite the currents in semi-discretized form for link i (link i connects discretized points i and $i+1$):

$$\begin{aligned}
J_n^i &= q\mu_n n_i \frac{\partial E_{F_n,i}}{\partial x}\\
J_p^i &= q\mu_p p_i \frac{\partial E_{F_p,i}}{\partial x}
\end{aligned} \tag{2.7}$$

Note that link indices are denoted with a superscript, while site indices are denoted with a subscript.

Next, a key step to ensure numerical stability is to integrate the above in order to get a completely discretized version of the current J^i. This discretization is known as the Scharfetter-Gummel scheme[1]. First, rewrite the hole density in terms of the quasi-Fermi level.

$$p(x) = e^{\left(-\chi(x)-E_g(x)-E_{F_p}(x)-q\phi(x)+k_BT\ln(N_V)\right)/k_BT}$$

[1]

8. (k) Gummel, IEEE Transactions on Electron Devices, **11**, 455 (1964).

It's convenient to define $\psi_p = \chi + E_g + E_g - k_B T \ln(N_V)$. We plug this form of p into Eq. (2.7):

$$J_p^i = q\mu_p e^{-\psi_p(x)/k_B T} \frac{\partial E_{F_p}}{\partial x},$$

next multiply both sides of the hole current by $e^{\psi_p(x)/k_B T}\, dx$, and integrate over link i

$$\int J_p^i e^{\psi_p(x)/k_B T} \mathrm{d}x = q\mu_p \int e^{-E_{F_p}/k_B T} \mathrm{d}E_{F_p} \tag{2.8}$$

Now we assume that ψ_p varies linearly between grid points,

$$\psi_p(x) = \frac{\psi_{p_{i+1}} - \psi_{p_i}}{\Delta x^i}(x - x_i) + \psi_{p_i},$$

which enables the integral on the left hand side above to be performed:

$$\int_{x_i}^{x_{i+1}} \mathrm{d}x e^{\psi_p(x)/k_B T} = k_B T \Delta x^i \frac{e^{\psi_{p_{i+1}}/k_B T} - e^{\psi_{p_i}/k_B T}}{\psi_{p_{i+1}} - \psi_{p_i}} \tag{2.9}$$

Plugging Eq. (2.9) into Eq. (2.8) and solving for J_p^i yields

$$J_p^i = \frac{q\mu_p^i}{\Delta x^i} \frac{\psi_{p_i} - \psi_{p_{i+1}}}{e^{\psi_{p_{i+1}}/k_B T} - e^{\psi_{p_i}/k_B T}} \mu_p \left[e^{-E_{F_p,i+1}/k_B T} - e^{-E_{F_p,i}} \right] \tag{2.10}$$

Where $\mu_p^i = (\mu_{p_i} + \mu_{p_{i+1}})/2$. A similar procedure leads to the following expression for J_n^i:

$$J_n^i = \frac{q\mu_n^i}{\Delta x^i} \frac{\psi_{n_{i+1}} - \psi_{n_i}}{e^{-\psi_{n_{i+1}}/k_B T} - e^{-\psi_{n_i}/k_B T}} \left[e^{E_{F_n,i+1}/k_B T} - e^{E_{F_n,i}/k_B T} \right] \tag{2.11}$$

where $\psi_n = q\phi + \chi + k_B T \ln(N_C)$.

2.2.2 Newton-Raphson algorithm

We want to write the continuity and Poisson equations in the form $f(x) = 0$, and solve these coupled nonlinear equations by using root-finding algorithms. The appropriate form is given by:

$$\begin{aligned}
f_p^i &= \frac{2}{\Delta x^i + \Delta x^{i-1}} \left(J_p^i - J_p^{i-1} \right) + G_i - R_i \\
f_n^i &= \frac{2}{\Delta x^i + \Delta x^{i-1}} \left(J_n^i - J_n^{i-1} \right) - G_i + R_i \\
f_v^i &= \frac{2}{\Delta x^i + \Delta x^{i-1}} \left(\left(\frac{\epsilon_i + \epsilon_{i-1}}{2} \right) \left(\frac{\phi_i - \phi_{i-1}}{\Delta x^{i-1}} \right) - \left(\frac{\epsilon_{i+1} + \epsilon_i}{2} \right) \left(\frac{\phi_{i+1} - \phi_i}{\Delta x^i} \right) \right) - \rho_i
\end{aligned}$$

These equations are the discretized drift-diffusion-Poisson equations to be solved for the variables $\{E_{F_n,i}, E_{F_p,i}, \phi_i\}$, subject to the boundary conditions given in introduction.

We use a Newton-Raphson method to solve the above set of equations. The idea behind the method is clearest in a simple one-dimensional case as illustrated on the figure below. Given a general nonlinear function $f(x)$, we want to find its root $\bar{x} : f(\bar{x}) = 0$. Given an initial guess x_1, one can estimate the error δx in this guess by assuming that the function varies linearly all the way to its root

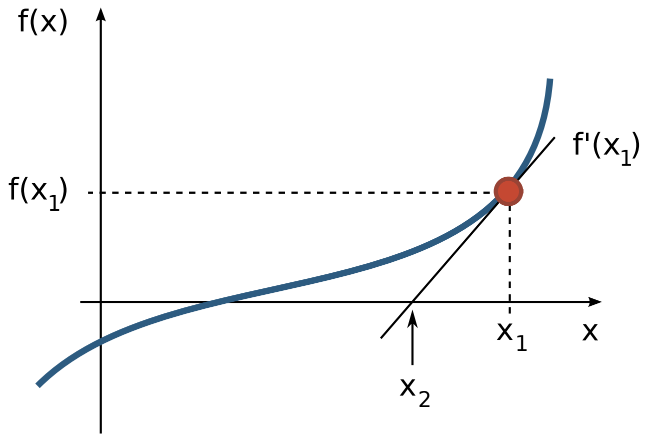


Fig. 2: Schematic for the Newton-Raphson method for root finding.

$$\delta x = \left(\frac{df}{dx}(x_1) \right)^{-1} f\left(x_1\right) \tag{2.12}$$

An updated guess is provided by $x_2 = x_1 - \delta x$.

In multiple dimensions the last term in Eq. (2.12) is replaced by the inverse of the Jacobian, which is the multi-dimensional generalization of the derivative. In this case, Eq. (2.12) is a matrix equation of the form:

$$\delta \mathbf{x} = A^{-1} \mathbf{F}\left(\mathbf{x}\right)$$

where

$$A_{ij} = \frac{\partial F_i}{\partial x_j}$$

Here is a small subset of the A matrix for our problem. We have only explicitly shown the row which corresponds to f_n^i (here we drop the super/sub script convention set up to distinguish between sites and links, for the sake of writing things more compactly):

$$\begin{pmatrix} & \cdots & & & & & & & & & \\ \vdots & & & & & & & & & & \\ \cdots & \frac{\partial f_n^i}{\partial E_{F_n}^{i-1}} & \frac{\partial f_n^i}{\partial E_{F_p}^{i-1}} & \frac{\partial f_n^i}{\partial \phi^{i-1}} & \frac{\partial f_n^i}{\partial E_{F_n}^{i}} & \frac{\partial f_n^i}{\partial E_{F_p}^{i}} & \frac{\partial f_n^i}{\partial \phi^{i}} & \frac{\partial f_n^i}{\partial E_{F_n}^{i+1}} & \frac{\partial f_n^i}{\partial E_{F_p}^{i+1}} & \frac{\partial f_n^i}{\partial \phi^{i+1}} & \cdots \\ \vdots & & & & & & & & & & \\ & \cdots & & & & & & & & & \end{pmatrix} \begin{pmatrix} \vdots \\ \delta E_{F_n}^{i-1} \\ \delta E_{F_p}^{i-1} \\ \delta \phi^{i-1} \\ \delta E_{F_n}^{i} \\ \delta E_{F_p}^{i} \\ \delta \phi^{i} \\ \delta E_{F_n}^{i+1} \\ \delta E_{F_p}^{i+1} \\ \delta \phi^{i+1} \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ f_n^{i-1} \\ f_p^{i-1} \\ f_v^{i-1} \\ f_n^{i} \\ f_p^{i} \\ f_v^{i} \\ f_n^{i+1} \\ f_p^{i+1} \\ f_v^{i+1} \\ \vdots \end{pmatrix} \tag{2.13}$$

Note that for this problem, finding derivatives numerically leads to major convergence problems. We derived the derivatives and implemented them in the code for this reason.

2.2.3 Multi-dimensional implementation

We do the standard *folding* of the multi-dimensional index label (i, j, k) into the single index label s of the sites of the system:

$$s = i + (j \times n_x) + (k \times n_x \times n_y)$$

where n_x (n_y) is the number of sites in the x-direction (y-direction).

Using sparse matrix techniques is key fast to fast computation. We provide below the number of non-zero elements in the Jacobian for periodic boundary conditions in the y- and z-directions.

Dimension	Number of stored values in the Jacobian
1	19 (n_x-2) + 20
2	n_y [29 (n_x - 2) + 28]
3	n_y n_z [39 (n_x - 2) + 36]

By default the Newton correction is computed by a direct resolution of the system in Eq. (2.13). This is done using the default Scipy solver. We recommend using the MUMPS library instead, which yields faster performace. Note that for large systems, and especially for 3D problems, the memory and the computing time required by the direct methods aforementioned become so large that they are impractical. It is possible to use an iterative method to solve Eq. (2.13) in these cases.

References

CHAPTER

THREE

TUTORIAL: LEARNING SESAME THROUGH EXAMPLES

3.1 Tutorial 1: I-V curve of a one-dimensional *pn* homojunction

In this tutorial we show how to build a simple one-dimensional pn^+ homojunction and compute its I-V curve. In this and other tutorials we assume a rudimentary knowledge of python (e.g. basic syntax, function definition, etc.) and numpy (e.g. array declarations).

See also:

The example treated here is in the file `1d_homojunction.py` located in the `examples\tutorial1` directory of the distribution. The same simulation's GUI input file is `1d_homojunction.ini`, also located in the `examples\tutorial1` directory.

The 1-dimensional pn^+ homojunction band diagram under short-circuit conditions is shown below.

3.1.1 A word for Matlab users

Sesame uses the Python3 language and the scientific libraries Numpy and Scipy. A documentation on the similarities and differences between Matlab and Numpy/Scipy can be found here.

3.1.2 Constructing a mesh and building the system

We start by importing the sesame and numpy packages:

```
import sesame
import numpy as np
```

Choosing a good mesh is a crucial step in performing the simulation. An overly coarse mesh will give inaccurate results, while an excessively fine mesh will make the simulation slow. The best mesh for most systems is nonuniform: being highly refined in regions where the solution changes rapidly, and coarse in regions where the solution varies slowly. After the tutorials the user should have a sense of how to construct an appropriate mesh. In this example, we create a mesh which contains more sites in the *pn* junction depletion region:

```
L = 3e-4 # length of the system in the x-direction [cm]
x = np.concatenate((np.linspace(0,1.2e-4, 100, endpoint=False),      # depletion region
                    np.linspace(1.2e-4, L, 50)))                      # neutral region
```

Note: Sesame assumes all quantities of length are input in units of cm. Other assumed input units are: time in s, energy in eV.

To make a system we use Sesame's *Builder()* class. The input to *Builder()* are grids along the different dimensions of the simulation. For a 1-dimensional simulation, we provide only the *x* grid as input. *Builder()* returns an object `sys` which contains all the information needed to describe the simulation. Additional simulation settings will be set by calling various methods of `sys`:

```
sys = sesame.Builder(x)
```

3.1.3 Adding material properties

Next we add a material to our system. A material is defined using a python `dictionary` object, which is added to the system using the *add_material()* method of `sys`:

```
material = {'Nc':8e17, 'Nv':1.8e19, 'Eg':1.5, 'affinity':3.9, 'epsilon':9.4,
        'mu_e':100, 'mu_h':100, 'Et':0, 'tau_e':10e-9, 'tau_h':10e-9, 'Et':0}

sys.add_material(material)
```

Here `Nc` (`Nv`) is the effective density of states of the conduction (valence) band (cm^{-3}), `Eg` is the material band gap (eV), `epsilon` is the material's dielectric constant, `mu_e` (`mu_h`) is the electron (hole) mobility ($\mathrm{cm}^2/(\mathrm{V}\cdot\mathrm{s})$), `Et` is the energy level of the bulk recombination defect, as measured from the intrinsic energy level, and `tau_e` (`tau_h`) is the electron (hole) bulk lifetime (s). For the full list of material parameters available, see the documentation of the method *add_material()*.

3.1.4 Adding dopants

Let's add dopants to make a *pn* junction. This requires specifying the regions containing each type of dopant, which is done by using a python function. Here's an example: let's suppose the n-type region is between x=0 and x=50 nm. We write a function which returns a value of `True` when the input `pos` belongs to this region, and `False` otherwise:

```
junction = 50e-7 # extent of the junction from the left contact [cm]

def n_region(pos):
    x = pos
    return x < junction
```

We add donors by calling the `sys` method *add_donor()*, whose input arguments are the donor concentration (in units cm^{-3}), and the function defining the doped region (`n_region` for this example)

```
# Add the donors
nD = 1e17 # [cm^-3]
sys.add_donor(nD, n_region)
```

Similarly, we add acceptors by defining a function `p_region` to specify the p-type region, and add it to the system with the *add_acceptor()* method:

```
def p_region(pos):
    x = pos
    return x >= junction

```

(continues on next page)

(continued from previous page)

```
# Add the acceptors
nA = 1e15 # [cm^-3]
sys.add_acceptor(nA, p_region)
```

Note: The `lambda` keyword provides a more efficient way to define simple functions with python. For example, the `p_region` function can be defined in the single line: `p_region = lambda x:  x<=junction`. See python documentation for more details on defining "anonymous functions" using `lambda`.

3.1.5 Specifying contact types

Next we need to specify the contact boundary conditions. For this example, we'll use selective Ohmic contacts. We first specify contact type with the `sys` method *`contact_type()`*, which takes two input arguments: the contact type at $x = 0$ ("left" contact), and the contact type at $x = L$ ("right" contact). Note that the order of arguments matters: the right contact type is the first agument, the left contact type is the second argument. Then we'll specify the recombination velocities for electrons and holes at left and right contacts with the `sys` method *`contact_S()`*. Again, the order of the input to *`contact_S()`* should be as shown below:

```
# Define Ohmic contacts
sys.contact_type('Ohmic', 'Ohmic')

# Define the surface recombination velocities for electrons and holes [cm/s]
Sn_left, Sp_left, Sn_right, Sp_right = 1e7, 0, 0, 1e7  # cm/s
sys.contact_S(Sn_left, Sp_left, Sn_right, Sp_right)
```

3.1.6 Computing an I-V curve

To compute an I-V curve under illumination, we specify the generation profile with a function. For this example, we use an exponentially varying generation profile defined in the function `gfcn`:

```
phi = 1e17       # photon flux [1/(cm^2 s)]
alpha = 2.3e4    # absorption coefficient [1/cm]

# Define a function for the generation rate
def gfcn(x):
    return phi * alpha * np.exp(-alpha * x)
```

Adding the illumination profile to the simulation is accomplished with the `sys` method *`generation()`*, which takes the function we've defined as input:

```
sys.generation(gfcn)
```

Finally we compute the I-V curve under illumination. We do this with the sesame method `IVcurve()`, whose the input arguments are the system object `sys`, an array of applied voltage values, the equilibrium solution we just computed, and a string which is the seedname for the output files.:

```
voltages = np.linspace(0, 0.95, 40)
j = sesame.IVcurve(sys, voltages, '1dhomo_V')
j = j * sys.scaling.current
```

Note: The `IVcurve` method returns the dimensionless current. We convert it to dimension-ful form by multiplying by the constant `sys.scaling.current`.

The output data files will have names like `1dhomo_V_0.gzip` where the number 0 labels the the `voltages` array index. These data files contain all the information about the simulation settings and solution. *tutorial 4* discusses how to access and plot this detailed data.

3.1.7 Saving and plotting the I-V curve

In this section we show different ways to save the computed current and voltage values.

First we store the data we wish to save in a dictionary object:

```
result = {'v':voltages, 'j':j}
```

Then we use the numpy function `save` to save the data as a numpy array. The first argument is the filename for the saved data (note the file will receive a .npy extension), the second argument is the dictionary to save:

```
np.save('jv_values', result)
```

The data dictionary can subsequently be loaded with the command:

```
result = np.load("jv_values.npy").
```

We can also save the data in a simple ascii file with the command:

```
np.savetxt('jv_values.txt', (v, j))
```

An alternative is to save the data in a Matlab-readable .mat file. This is accomplished with the function `savemat` in the scipy library:

```
import scipy.io.savemat as savemat
savemat('jv_values.mat', result)
```

Note: In the tutorial script, we've added commands to check if the scipy library is installed. We omit these commands in this tutorial for the sake of clarity.

The library Matplotlib is commonly used for plotting in python. The code for generating a simple current-voltage plot is shown below:

```
import matplotlib.pyplot as plt
plt.plot(voltages, j, '-o')
plt.xlabel('Voltage [V]')
plt.ylabel('Current [A/cm^2]')
plt.grid()      # add grid
plt.show()      # show the plot on the screen
```

We discuss loading and plotting results in *Tutorial 4*. As a preview, we show the code used to generate the band diagram we showed at the beginning of this tutorial:

```
sys, result = sesame.load_sim('1dhomo_V_0.gzip')  # load data file
az = sesame.Analyzer(sys,result)                   # get Sesame analyzer object
p1 = (0,0)
p2 = (3e-4,0)
az.band_diagram((p1,p2))                           # plot band diagram along line␣
↪from p1 to p2
```

3.2 Tutorial 2: I-V curve of a one-dimensional *pn* heterojunction

In this tutorial we consider a more complex system in 1-dimension: a heterojunction with a Schottky back contact. The n-type material is CdS and the p-type material is CdTe. The structure of the script is the same as in the last tutorial, however we must provide more detail to describe a more complex system.

See also:

The example treated here is in the file `1d_heterojunction.py` located in the `examples\tutorial2` directory of the distribution. The same simulation's GUI input file is `1d_heterojunction.ini`, also located in the `examples\tutorial2` directory.

The band diagram for this system under short-circuit conditions is shown below .

3.2.1 Constructing a grid and building the system

We first define the thicknesses of the n-type and p-type regions:

```
t1 = 25*1e-7     # thickness of CdS
t2 = 4*1e-4      # thickness of CdTe
```

The mesh for a heterojunction should be very fine in the immediate vicinity of the materials interface. We define a distance `dd`, which determines the thickness of the highly-refined mesh near an interface. We form the overall system mesh by concatenating meshes for different parts of the system as follows:

```
dd = 3e-6 # 2*dd is the distance over which mesh is highly refined
x = np.concatenate((np.linspace(0, dd, 10, endpoint=False),                     # L␣
↪contact interface
            np.linspace(dd, t1 - dd, 50, endpoint=False),                        #␣
↪material 1
            np.linspace(t1 - dd, t1 + dd, 10, endpoint=False),                  #␣
↪interface 1
            np.linspace(t1 + dd, (t1+t2) - dd, 100, endpoint=False),            # material␣
↪2
            np.linspace((t1+t2) - dd, (t1+t2), 10)))                            # R␣
↪contact interface
```

As before we make a system with `Builder()`:

```
sys = sesame.Builder(x)
```

3.2.2 Adding material properties

We make functions to define the n-type and p-type regions as in the last tutorial:

```
def CdS_region(pos):
    x = pos
    return x<=t1

def CdTe_region(pos):
    x = pos
    return x>t1
```

Now we add materials to our system. We define two dictionaries to describe the two material types:

```
CdS = {'Nc': 2.2e18, 'Nv':1.8e19, 'Eg':2.4, 'epsilon':10, 'Et': 0,
    'mu_e':100, 'mu_h':25, 'tau_e':1e-8, 'tau_h':1e-13, 'affinity': 4.}

CdTe = {'Nc': 8e17, 'Nv': 1.8e19, 'Eg':1.5, 'epsilon':9.4, 'Et': 0,
    'mu_e':320, 'mu_h':40, 'tau_e':5e-9, 'tau_h':5e-9, 'affinity': 3.9}
```

As in the last tutorial, we add materials using the `sys` method *`add_material()`*. This time we specify the material location using the functions we defined above as additional input arguments to *`add_material()`*:

```
sys.add_material(CdS, CdS_region)      # adding CdS
sys.add_material(CdTe, CdTe_region)      # adding CdTe
```

3.2.3 Adding dopants

Adding the dopants works as in the last tutorial:

```
nD = 1e17  # donor density [cm^-3]
sys.add_donor(nD, CdS_region)
nA = 1e15  # acceptor density [cm^-3]
sys.add_acceptor(nA, CdTe_region)
```

3.2.4 Specifying contact types

Next, we'll add a left Ohmic contact and a right Schottky contact. For Schottky contacts, we must to specify the work function of the metal. As in the previous tutorial, we add contacts to the system using the `sys` method *`contact_type()`*; however this time we provide the additional arguments of the left and right contact work functions to *`contact_type()`*:

```
Lcontact_type, Rcontact_type = 'Ohmic', 'Schottky'
Lcontact_workFcn, Rcontact_workFcn = 0, 5.0    # eV

sys.contact_type(Lcontact_type, Rcontact_type, Lcontact_workFcn, Rcontact_workFcn)
```

Note that for Ohmic contacts, the metal work function doesn't enter into the problem, so its value is unimportant - we therefore simply set the left contact work function equal to 0. Having defined the contact types, we next specify the contact recombination velocities as before. For this system, we'll assume the contacts are non-selective:

```
Sn_left, Sp_left, Sn_right, Sp_right = 1e7, 1e7, 1e7, 1e7  # cm/s
sys.contact_S(Sn_left, Sp_left, Sn_right, Sp_right)
```

3.2.5 Computing an I-V curve

We've now completed the system definition. As in the last example, we compute the equilibrium solution, add illumination, and compute the I-V curve

Warning: Sesame does not include interface transport mechanisms of thermionic emission and tunneling.

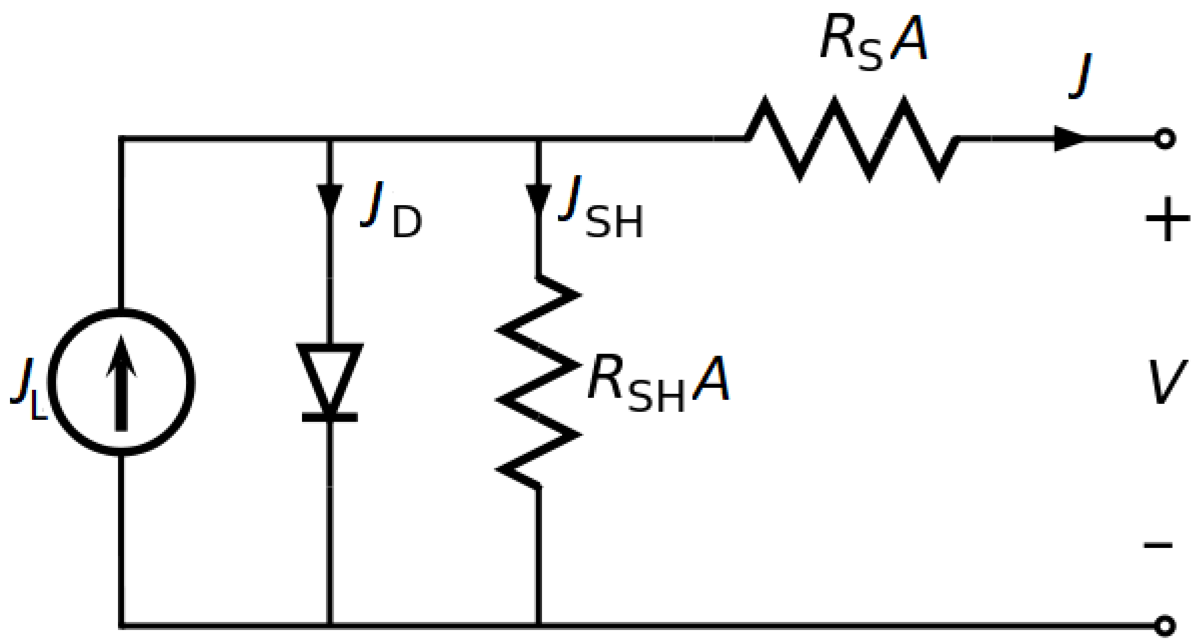
```
phi = 1e21 # photon flux [1/(cm^2 s)]
alpha = 2.3e6 # absorption coefficient [1/cm]

# Define a function for the generation rate
f = lambda x: phi * alpha * np.exp(-alpha * x)
sys.generation(f)

voltages = np.linspace(0, 0.95, 40)
j = sesame.IVcurve(sys, voltages, '1dhetero_V')
# convert dimensionless current to dimension-ful current
j = j * sys.scaling.current
```

The current can be saved and plotted as in the previous tutorial:

```
result = {'v':voltages, 'j':j}    # store j, v values
np.save('jv1d_hetero', result)     # save the j-v curve

import matplotlib.pyplot as plt
plt.plot(voltages, j,'-o')         # plot j-v curve
plt.xlabel('Voltage [V]')
plt.ylabel('Current [A/cm^2]')
plt.grid()                        # show grid lines
plt.show()                        # show plot
```

3.2.6 Adding contact and shunt resistance

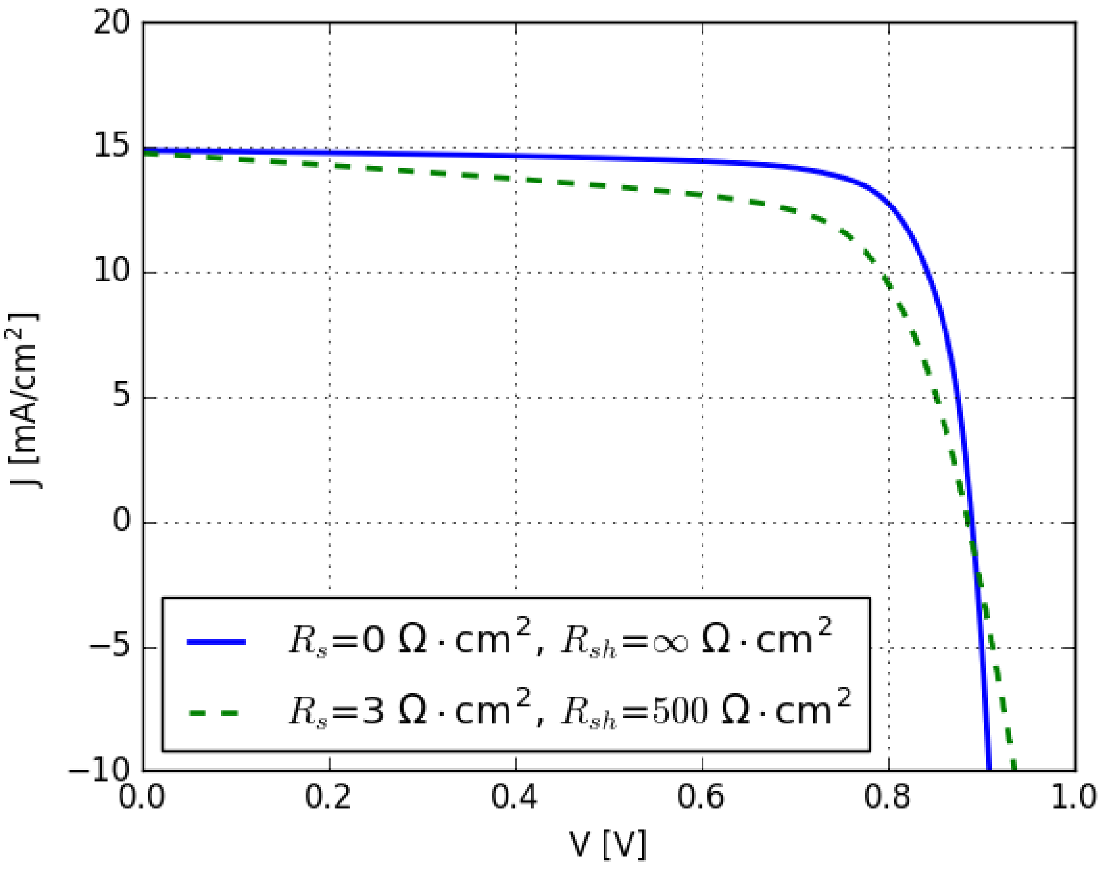
We next demonstrate how to include the effect of series and shunt resistance. The example treated here is in the file `1d_heterojunction_with_Rs_Rsh` located in the `examples\tutorial2` directory of the distribution. The classic equivalent circuit model for a solar cell is given below (note we use current density J and resistance-area product to characterize the circuit).



For our model, the diode in this circuit is replaced by the numerically computed current-voltage relation shifted by the computed short-circuit current, so that $J^{\mathrm{dark}}_{\mathrm{diode}}(0) = 0$). The light source current J_L is given by the numerically computed short-circuit current density. The current-voltage relation of the above circuit is given by the following implicit equation:

$$J = J_L - J^{\mathrm{dark}}_{\mathrm{diode}}(V - JR_{\mathrm{s}}A) - \frac{V + JR_{\mathrm{s}}A}{R_{\mathrm{sh}}A}.$$

For a fixed potential drop across the circuit V, the above equation is solved numerically to find the total current through the circuit J. Below we show the effect of finite series and contact resistance values (given by R_{s} and R_{sh} respectively) on the current-voltage relation computed in the first part of the tutorial:



We refer the reader to the example script for more details of how the mathematics of the equivalent circuit model is implemented.

3.3 Tutorial 3: Two-dimensional *pn* junction with a grain boundary

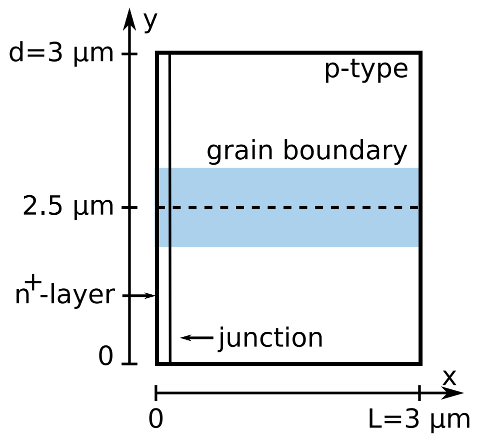
In this tutorial we show how to build a two-dimensional pn^+ junction containing a grain boundary.

See also:

The example treated here is in the file `2d_homojunction_withGB.py` located in the `examples\tutorial3` directory in the root directory of the distribution. The same simulation's GUI input file is `2d_homojunction_withGB.ini`, also located in the `examples\tutorial3` directory.

3.3.1 Building a two-dimensional system

We want to simulate a two-dimensional pn^+ junction (homojunction) with a columnar grain boundary as depicted below.



As usual, we start by importing the sesame and numpy packages. We construct the mesh of the system and make an instance of the `Builder()`. Notice that in this case we provide the `Builder()` function with both x and y grids; this automatically tells the code to build a two-dimensional system:

```
import sesame
import numpy as np

# dimensions of the system
Lx = 3e-4 # [cm]
Ly = 3e-4 # [cm]

# position of p-n junction [cm]
junction = .1e-4

# Mesh
x = np.concatenate((np.linspace(0, .2e-4, 30, endpoint=False),       # mesh near the␣
↪contact
                    np.linspace(0.2e-4, 1.4e-4, 50, endpoint=False),     # mesh in␣
↪depletion region
                    np.linspace(1.4e-4, 2.7e-4, 50, endpoint=False),     # mesh in bulk
                    np.linspace(2.7e-4, 2.98e-4, 30, endpoint=False),    # mesh near the␣
↪GB end point
                    np.linspace(2.98e-4, Lx, 10)))                     # mesh near the␣
↪contact

y = np.concatenate((np.linspace(0, 1.25e-4, 50, endpoint=False),
                    np.linspace(1.25e-4, 1.75e-4, 50, endpoint=False),   # mesh near the␣
↪GB core
                    np.linspace(1.75e-4, Ly, 50)))

# Create a system
sys = sesame.Builder(x, y)
```

We define and add a material as before:

```
# Dictionary with the material parameters
mat = {'Nc':8e17, 'Nv':1.8e19, 'Eg':1.5, 'affinity': 3.9, 'epsilon':9.4,
        'mu_e':200, 'mu_h':200, 'tau_e':10e-9, 'tau_h':10e-9, 'Et':0}

# Add the material to the system
sys.add_material(mat)
```

We next define functions delimiting the regions with different doping values. Because the model is 2-dimensional, the function input argument `pos` is a `tuple` of the form `(x,y)`. In the functions below, we first "unpack" the x and y coordinate, and determine the location of the x-coordinate relative to the junction:

```
# Add the donors
def n_region(pos):
    x, y = pos
    return x < junction

nD = 1e17  # [cm^-3]
sys.add_donor(nD, n_region)

# Add the acceptors
def p_region(pos):
    x, y = pos
    return x >= junction

nA = 1e15  # [cm^-3]
sys.add_acceptor(nA, p_region)
```

We specify contacts as before:

```
# Use perfectly selective Ohmic contacts
sys.contact_type('Ohmic', 'Ohmic')
Sn_left, Sp_left, Sn_right, Sp_right = 1e7, 0, 0, 1e7
sys.contacts(Sn_left, Sp_left, Sn_right, Sp_right)
```

Note: Sesame assumes that the contact properties (e.g. recombination velocity, metallic work function) are uniform along the y-direction.

3.3.2 Adding a grain boundary

Now we add a line of defects to simulate a grain boundary using the `sys` method `add_line_defects()`. The necessary inputs are the grain boundary defect's electrical properties (e.g. capture cross sections, energy level, defect density, and charge states), and the endpoints defining the grain boundary location (recall a grain boundary is represented by a line in a 2-dimensional simulation). Below we show code defining these properties for our example, and adding the grain boundary to the simulation:

```
# gap state characteristics
rho_GB = 1e14               # defect density [1/cm^2]
S_GB = 1e-15                # trap capture cross section [cm^2]
E_GB = 0.4                  # energy of gap state (eV) from intrinsic energy level


# Specify the two points that make the line containing additional charges
p1 = (.1e-4, 1.5e-4)   # [cm]
p2 = (2.9e-4, 1.5e-4)  # [cm]

# Add the line of defects to the system
sys.add_line_defects([p1, p2], rho_GB, S_GB, E=E_GB, transition=(1/-1))
```

The type of the charge transition α/β is specified by assigning the `transition` input value as shown above. In our example we chose a mixture of donor and acceptor at energy E. An acceptor would be described by (-1,0) and a donor by (1,0).

Note:

- Avoid adding charges on the contacts of the system, as these will not be taken into account. The code is not equiped to deal with such boundary conditions.
- In order to add another gap state at a different energy at the same location, one repeats the exact same process.
- Here we assumed equal electron and hole surface recombination velocities. The function `add_line_defects()` takes two surface recombination velocities as argument. The first is for electrons, the second for holes. To use different values write

```
sys.add_line_defects([p1, p2], rho_GB, Sn_GB, Sp_GB, E=E_GB)
```

- A continuum of states can be considered by omitting the energy argument above. The density of states can be a callable function or a numerical value, in which case the density of states is independent of the energy.

3.3.3 Computing the IV curve

The computation of the IV curve proceeds as in the previous tutorials. We show the code below:

```
# Solve equilibirum problem first
solution = sesame.solve(sys, 'Poisson')


# define a function for generation profile
f = lambda x, y: 2.3e21*np.exp(-2.3e4*x)
# add generation to the system
sys.generation(f)

# Specify applied voltages
voltages = np.linspace(0, .9, 10)
# Compute IV curve
j = sesame.IVcurve(sys, voltages, '2dGB_V', guess=solution)
# rescale to dimension-ful current
j = j * sesame.scaling.current

# Save the computed IV data
result = {'voltages':voltages, 'j':j}
np.save('2dGB_IV', result)
```

3.3.4 Plotting system variables

The 2-dimensional solutions can be plotted with tools we describe more fully in *tutorial 4*. As a preview, we list the commands for loading and plotting the electrostatic potential:

```
sys, results = sesame.load_sim('2dGB_V_0.gzip')
sesame.plot(sys, results['v'])
```

The output is shown below:

Note: As discussed more fully in Tutorial 4, quantities in Sesame are dimensionless by default. The electrostatic potential shown above is dimensionless, scaled by the thermal voltage. The `scaling` field of `sys` provides the relevant quantites needed to rescale quantities to dimension-ful form.

3.3.5 Spatial variation of material parameters

See also:

The example treated here is in the file `2d_homojunction_withGB_nonuniform_mobility.py` in the `examples\tutorial4` directory in the root directory of the distribution.

Suppose we want to have a reduced mobility around the line defects compared to the rest of the system. To do so, we add another material which is defined in the region of non-uniform mobility. It has the same properties as the original material, except that the mobility is not longer a scalar, but a function:

```
# function defining region of reduced mobility
xGB = 1.5e-4   # GB x-coordinate
Lmu = .25e-4   # distance from GB over which mobility is reduced
def reduced_mu_region(pos):
    x, y = pos
    return ((x < xGB+Lmu) & (x > xGB-Lmu) & (y > .1e-4) & (y < 2.9e-4))

# function defining region of reduced mobility
def my_mu(pos):
    muGB = 10
    x, y = pos
    # mobility varies linearly between GB core and Lmu
    return 10 + 310*np.abs((x-xGB)/Lmu)

mat2 = {'Nc': 8e17, 'Nv': 1.8e19, 'Eg': 1.5, 'epsilon': 9.4, 'Et': 0,
        'mu_e': my_mu, 'mu_h': 40, 'tau_e': 10 * 1e-9, 'tau_h': 10 * 1e-9}

# Add the material to the system
sys.add_material(mat2, reduced_mu_region)

sesame.plot(sys, sys.mu_e)
```

3.4 Tutorial 4: Saving, loading, and analyzing simulation data

In this tutorial we describe the input and output data formats of Sesame, and show how to use Sesame's built-in tools to analyze the solution.

See also:

The examples treated here are in the files `analyze_data.py` and `plot_data.py`, located in the `examples\tutorial4` directory of the distribution. This tutorial uses the output files from `examples\tutorial3` python script, so it's necessary to copy these gzip data files to `examples\tutorial4`.

3.4.1 Saving data

In the previous tutorials, detailed solution data was saved automatically in the `IVcurve()` function. It's also possible to manually save simulations with Sesame's *`save_sim()`* function. This function saves both the system object containing all of the simulation settings, and the solution dictionary. (The solution dictionary contains the keys `'v'`, `'efn'`, and `'efp'`.) An example of *`save_sim()`* is shown below:

```
sesame.save_sim(sys, results, "my_sim")
```

The saved output file is named "my_sim.gzip". Note that the gzip extension indicates the data is compressed, and the data structures are stored using python's `pickle` module.

The data can also be saved in a Matlab-readable format (.mat file), by adding `fmt='mat'` as an additional input argument:

```
sesame.save_sim(system, result, "my_sim", fmt='mat')
```

In this case the arrays defining the system properties (including `Eg`, `Nc`, `Nv`, etc) are saved in a `system` data structure, and the solution (`'efn'`, `'efp'`, `'v'`) is saved in a `results` data structure.

Note: The saved data is "folded" into 1-d arrays. For example, the solution array of a two-dimensional system with n_x x-grid points and n_y y-grid points is stored as a one-dimensional array of length $n_x \times n_y$.

The postprocessing tools packaged with Sesame are built to work with python data files. It's therefore more convenient to save the data in python format if substantial postprocessing and analysis will be performed.

3.4.2 Loading data

Loading a saved simulation is accomplished with the `load_sim` command. This returns the system object and result dictionary:

```
sys, result = sesame.load_sim("my_sim")
```

3.4.3 Analysis of data: the Analyzer object

Next we show how to extract and analyze the data computed by the solvers. To facilitate data analysis, Sesame has an *`Analyzer()`* class which contains many methods to compute typical quantities of interest, such as total defect, radiative, or radiative recombination, total current, carrier densities, grain boundary recombination, etc. A summary and the descriptions of the methods available via the *`Analyzer()`* object are detailed in Sec. *Core modules*.

We'll demonstrate the use of some of these methods, using the system created in *tutorial 3*. We start as always by importing sesame and numpy:

```
import numpy as np
import sesame
```

Note: In the rest of this tutorial, it's necessary to copy the output gzip files you obtained in tutorial3 to the `examples\tutorial4` directory

Our data analysis will begin with computing carrier densities and currents, and plotting data. In the code below we load a data file and create an instance of Analyzer class. The *`Analyzer()`* object is initialized with a system and a dictionary of results. This dictionary must contain the key `'v'`, and can include `'efn'`, `'efp'` when computed.:

```
sys, results = sesame_loadsim('2dpnIV.vapp_0.npz')
az = sesame.Analyzer(sys, results)
```

We start with how to obtain integrated quantities like the steady state current. In the code below we compute the current for all applied voltages of the IV curve, using the *`Analyzer()`* method `full_current()`:

```
J = []
for i in range(40):
    filename = '2dpnIV.vapp_{0}.npz'.format(i)     # construct file name
    sys, results = sesame.load_sim(filename)       # load file
    az = sesame.Analyzer(sys, results)             # create Analyzer
    current = az.full_current()                    # compute current
    J.append(current)                              # add to array of current values
```

Non-integrated quantities are often plotted along lines. We define such lines by two points. Given two points in real coordinates, the method *line()* returns the dimensionless curvilinear abscissae along the line, and the grid sites:

```
p1 = (2e-4, 0)    # [cm]
p2 = (2e-4, 3-6)   # [cm]

X, sites = az.line(sys, p1, p2)
```

Scalar quantities like densities or recombination are obtained either for the entire system, or on a line:

```
# For the entire system
n2d = az.electron_density()
n2d = n2d * sys.scaling.density              # convert to dimension-ful form
n2d = np.reshape(n2d, (sys.ny, sys.nx))      # reshape to 2-d array

# On the previously defined line
n1d = az.electron_density((p1, p2))
n1d = n1d * sys.scaling.density              # convert to dimension-ful form
```

Note: Note that the Analyzer methods return values in dimensionless form. It is therefore necessary to convert to dimension-ful form using the quantities stored in the `scaling` field of `sys`. Available dimensions are: density, energy, mobility, time, length, and generation. These dimensions (except mobility) depend on the temperature and the unit length (meter or centimeter) given when creating an instance of the class *Builder()* (default is 300 K and centimeters).

Vectorial quantities (i.e. currents) are computed either on a line or for the entire system, by component. For instance, to compute the electron current in the x-direction:

```
# For the entire system
jn = az.electron_current(component='x')

# On the previously defined line
jn = az.electron_current(location=(p1, p2))
```

We now turn to the treatment of the extended line defects introduced in our system. The following code retrieves the solution along the grain boundary core:

```
# Line endpoints of the grain boundary core
p1 = (20e-7, 1.5e-4)    #[cm]
p2 = (2.9e-4, 1.5e-4)   #[cm]
# get the coordinate indices of the grain boundary core
X, sites = az.line(syst, p1, p2)

# obtain solution data along the GB core
efn_GB = results['efn'][sites]
efp_GB = result['efp'][sites]
v_GB   = result['v'][sites]
```

In this code we compute the integrated defect recombination along the grain boundary core:

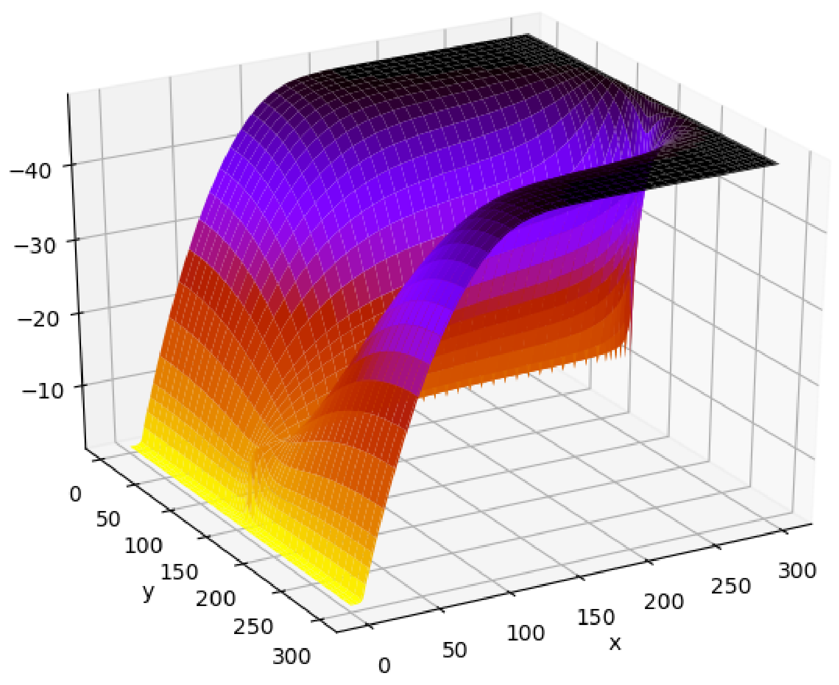
```
# Get the first planar defect from the system
defect = sys.defects_list[0]
# Compute the defect recombination rate as a function of position along the planar␣
↪defect
R_GB = az.defect_rr(defect)    # R_GB is an array

# Compute the integrated recombination along the line defect
Rtot_GB = az.integrated_defect_recombination(defect)  #Rtot_GB is a number
```

The attribute of Builder called `defects_list` is a list of named tuples. This list stores the parameters of each defect originally added to the system. The field names of the named tuples are `sites`, `location`, `dos`, `energy`, `sigma_e`, `sigma_h`, `transition`, `perp_dl`. The last field contains the lattice distance perpendicular to the line of defects. This quantity is used to normalize the recombination velocity and the density of states.

3.4.4 Plotting data

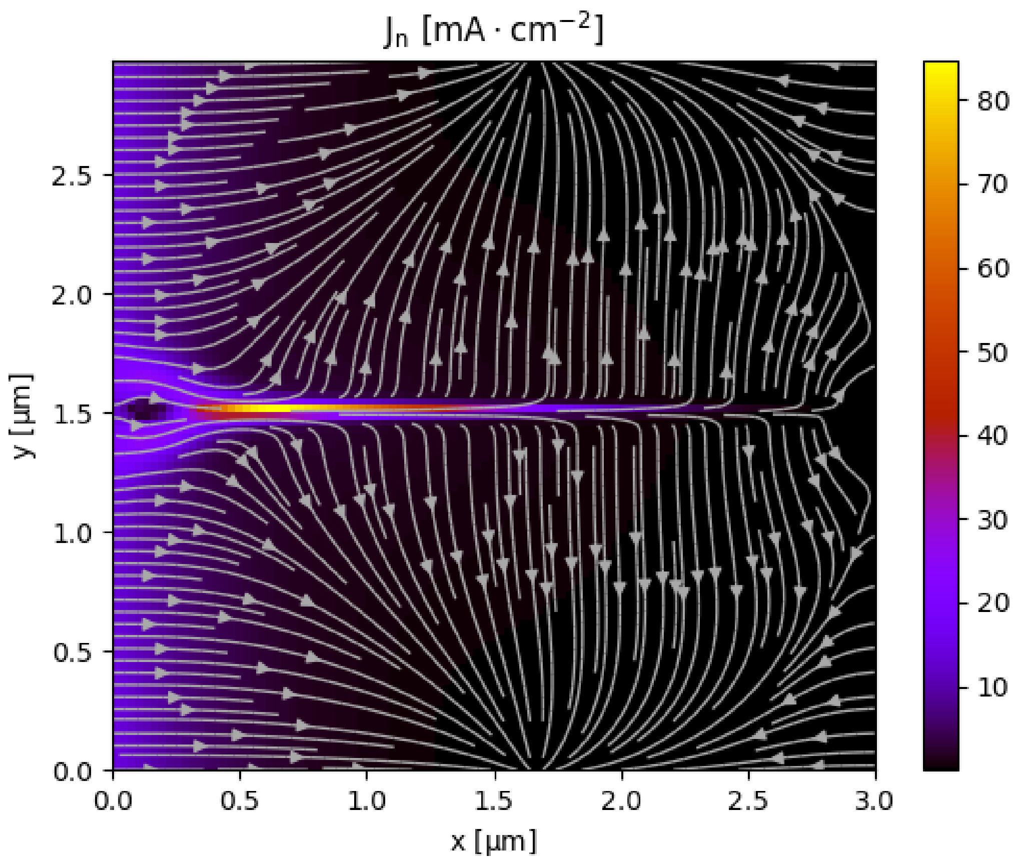
To facilitate the visualization of two- and three-dimensional plots, `sesame` provides a few functions (requiring `matplotlib`) that represent quantities in 2D or 3D. For example, one can visualize the electrostatic potential at zero bias in 3D with:

```
sys, results = sesame.load_sim('2dGB_V_0.gzip')
az = sesame.Analyzer(sys, results)
az.map3D(results['v']) # units of kT/q
```



or plot the electron current accross the system:

```
sys, results = sesame.load_sim('2dGB_10.gzip')
az = sesame.Analyzer(sys, results)
az.electron_current_map()
```



We finally show how to use native `matplotlib` functionality to plot the data obtained with the *`Analyzer()`* object. In the following code we make plots of the electron density that we obtained earlier:

```
# points define line perpendicular to GB
p1 = (2e-4, 0)    # [cm]
p2 = (2e-4, 3-6)   # [cm]
X, sites = az.line(syst, p1, p2)

# make 1d figure
plt.figure(1)
plt.plot(X, np.log(n1d))
plt.xlabel('Position [cm]')
plt.ylabel('Log[n]')
```

Here we represent the 2-dimensional electron density with a contour plot:

```
# For the entire system
n2d = az.electron_density()
n2d = n2d * sys.scaling.density              # convert to dimnsion-ful form
n2d = np.reshape(n2d, (sys.ny, sys.nx))      # reshape to 2-d array

# make 2d contour plot
plt.figure(2)
plt.contourf(sys.xpts, sys.ypts, np.log(n2d))
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Position [cm]')
plt.ylabel('Position [cm]')
plt.colorbar()
plt.title('ln(n)')

plt.show()  # show figures on screen
```

3.4.5 Advanced possibilities

In case the methods available in the *Analyzer()* are not enough (especially in 3D), the module *sesame.observables()* gives access to low-level routines that compute the carrier densities and the currents for any given sites on the discretized system.

In the table below we show the syntax used to get some attributes of the *Builder()* that can then be useful:

Attribute	Syntax
grid nodes	`syst.xpts`, `syst.ypts`, `syst.zpts`
number of grid nodes	`syst.nx`, `syst.ny`, `syst.nz`
grid distances	`syst.dx`, `syst.dy`, `syst.dz`

The exhaustive list of all accessible attributes is in the documentation of the *Builder()* class itself. Note that the grid nodes are in the units given in the system definition, while the grid distances are dimensionless.

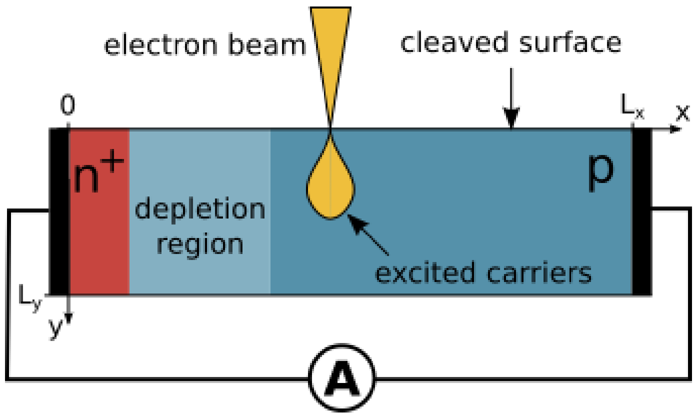
3.5 Tutorial 5: Simulating an EBIC/CL experiment

In this tutorial we build a 2-dimensional simulation to describe experiments with a localized carrier generation profile, such as electron beam induced current (EBIC), or cathodoluminescence (CL).

See also:

The example treated here is in the file `2d_EBIC.py` located in the `examples\tutorial5` directory of the distribution. The same simulation's GUI input file is `2d_EBIC.ini`, also located in the `examples\tutorial5` directory.

For this case we'll need to define a system as before, and then define a custom carrier generation rate density profile associated with electron beam excitation. We'll then cycle over beam positions and compute the total current and total radiative recombination as a function of beam position.



3.5.1 Building the system

The system we want to build is a 2-dimensional p-n junction in which the “top” of the system represents the exposed sample surface. Building the 2-d pn junction proceeds as in the previous tutorials, and the code is shown below. One important difference for this example is that the top/bottom boundary conditions are “hard-wall” (in the previous cases, we used periodic boundary conditions, which are the default). This is specified by calling `Builder()` with an extra argument `Periodic=False`:

```python
## dimensions of the system
Lx = 3e-4   #[cm]
Ly = 3e-4   #[cm]

# extent of the junction from the left contact [cm]
junction = .1e-4    # [cm]

# Mesh
x = np.concatenate((np.linspace(0,.2e-4, 30, endpoint=False),
                    np.linspace(0.2e-4, 1.4e-4, 60, endpoint=False),
                    np.linspace(1.4e-4, 2.7e-4, 60, endpoint=False),
                    np.linspace(2.7e-4, Lx, 10)))

y = np.concatenate((np.linspace(0, .25e-4, 50, endpoint=False),
                    np.linspace(.25e-4, 1.25e-4, 50, endpoint=False),
                    np.linspace(1.25e-4, Ly, 50)))

# Create a system
sys = sesame.Builder(x, y, periodic=False)

# Dictionary with the material parameters
mat = {'Nc':8e17, 'Nv':1.8e19, 'Eg':1.5, 'epsilon':9.4, 'Et': 0,
       'mu_e':320, 'mu_h':40, 'tau_e':10*1e-9, 'tau_h':10*1e-9}

# Add the material to the system
sys.add_material(mat)

# define a function specifiying the n-type region
def n_region(pos):
    x, y = pos
    return x < junction
# define a function specifiying the p-type region
def p_region(pos):
    x, y = pos
    return x >= junction

# Add the donors
nD = 1e17 # [cm^-3]
sys.add_donor(nD, n_region)
# Add the acceptors
nA = 1e15 # [cm^-3]
sys.add_acceptor(nA, p_region)

# Use Ohmic contacts
sys.contact_type('Ohmic','Ohmic')
Sn_left, Sp_left, Sn_right, Sp_right = 1e7, 1e7, 1e7, 1e7
sys.contact_S(Sn_left, Sp_left, Sn_right, Sp_right)
```

3.5.2 Adding surface recombination

Adding recombination at the sample surface is accomplished with a planar defect along the line $y = L_y$. We consider a neutral surface, so that the charge state of the defect is always 0. This is implemented by setting `transition=(0, 0)` as an input argument to `add_line_defects()`. As described in the previous tutorial, the values given in `transition` set the charge of the defect when its occupied or unoccupied:

```
p1 = (0, Ly)
p2 = (Lx, Ly)

E = 0                        # energy of gap state (eV) from intrinsic level
rhoGB = 1e14                 # density of defect states [cm^-2]
s = 1e-14                    # defect capture cross section [cm^2]

sys.add_line_defects([p1, p2], rhoGB, s, E=E, transition=(0,0))
```

3.5.3 Electron beam excitation

Next we review the physics of the electron beam excitation. For a beam focused at (x_0, y_0), a simple parameterization of the generation rate density profile is given by a Gaussian:

$$G(x,y) = \frac{G_{\rm tot}}{A} \times \exp\left(-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}\right) \tag{3.1}$$

For our geometry, x_0 is the lateral beam position, while the depth of the excitation from the sample surface is y_0. The total generation rate (units $1/s$) is approximated by[4]:

$$G_{tot} \approx \frac{I_{\rm beam}}{q} \times \frac{E_{\rm beam}}{3E_g} \tag{3.2}$$

The length scale of the excitation σ is determined by the electron beam energy and material mass density, and is written in terms of the interaction distance R_B:

$$R_B = r_0 \left(\frac{0.043}{\rho/\rho_0}\right) \times (E_{\rm beam}/E_0)^{1.75} \tag{3.3}$$

The constants in Eq. (3.3) are $r_0 = 1\ \mu{\rm m},\ \rho_0 = 1\ {\rm g/cm}^3,\ E_0 = 1\ {\rm keV}$. The length scale of the Guassian σ and the distance from the surface y_0 are related to R_B as[5]:

$$\sigma = \frac{R_B}{\sqrt{15}}$$
$$y_0 = 0.3 \times R_B$$

The normalization constant A has units of volume. The standard normalization of a 2-dimensional Gaussian is $2\pi\sigma^2$, which has units of area. An appropriate choice for the additional length factor in A is the electron diffusion length L_D, so that:

4

3. (j) Wu and D. B. Wittry, J. App. Phys., **49**, 2827,(1978).

5

1. (e) Grun, Zeitschrift fur Naturforschung, **12a**, 89, (1957).

$$A = 2\pi\sigma^2 L_D \tag{3.4}$$

To code $G(x, y)$, Eq. (3.1) we start by making the necessary definitions of constants:

```
q = 1.6e-19        # C
Ibeam = 10e-12     # A
Ebeam = 15e3       # eV
eg = 1.5           # eV
density = 5.85     # g/cm^3
kev = 1e3          # eV

Gtot = Ibeam/q * Ebeam / (3*eg)
Rbulb = 0.043 / density * (Ebeam/kev)**1.75      # given in micron
Rbulb = Rbulb * 1e-4                             # converting to cm

sigma = Rbulb / sqrt(15)                         # Gaussian spread
y0 = 0.3 * Rbulb                                 # penetration depth

Ld = np.sqrt(sys.mu_e[0] * sys.tau_e[0]) * sys.scaling.length  # diffusion length
```

3.5.4 Perfoming the beam scan

To scan the lateral position x_0 of the beam, we first define the list of x_0 values:

```
x0list = np.linspace(.1e-4, 2.5e-4, 11)
```

We define an array to store the computed current at each beam position:

```
jset = np.zeros(len(x0list))
```

Next we scan over x_0 with a `for` loop. At each value of x_0, we define a function as given in Eq. (3.1), and add this generation to the system:

```
for idx, x0 in enumerate(x0list):

    def excitation(x,y):
        return Gtot/(2*np.pi*sigma**2*Ld) *
           np.exp(-(x-x0)**2/(2*sigma**2)) * np.exp(-(y-Ly+y0)**2/(2*sigma**2))

    sys.generation(excitation)
```

Note: Using the GUI is more awkward for this type of simulation, because only one variable definition is allowed in the generation function definition. Therefore all of the numerical prefactors must be computed by the user and input by hand.

Now we solve the system:

```
solution = sesame.solve(sys, periodic_bcs=False, tol=1e-8)
```

We obtain the current and store it in the array:

```
# get analyzer object with which to compute the current
az = sesame.Analyzer(sys, solution)
# compute (dimensionless) current and convert to dimension-ful form
tj = az.full_current() * sys.scaling.current * sys.scaling.length
# save the current
jset[idx] = tj
```

It can be informative to plot the current normalized to the total generation rate. The (dimensionless) total generation rate for a simulation is contained in the `gtot` field of `sys`. As always, we must use `scaling` factors to make this a dimension-ful quantity. The code for this is shown below:

```
# obtain total generation from sys object
gtot = sys.gtot * sys.scaling.generation * sys.scaling.length**2
jratio[idx] = tj/(q * gtot)
```

We can also compute the radiative recombination at each beam position point, thereby simulation a cathodoluminescence experiment. This code shown below:

```
# compute (dimensionless) total radiative recombination and convert to to   dimension-
↪ful form
cl = az.integrated_radiative_recombination() * sys.scaling.generation *   sys.scaling.
↪length**2
# save the CL
rset[idx] = cl
rad_ratio[idx] = cl/gtot
```

The current and CL can be saved and plotted as in previous tutorials.

References

3.6 Tutorial 6: Batch submission for computing clusters

See also:

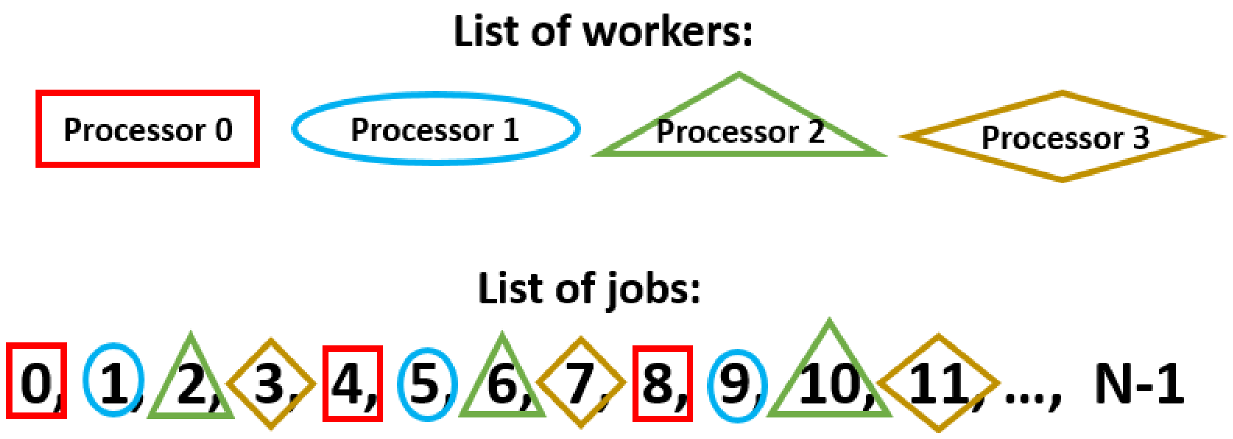
The example treated here is in the file `sesame_batch_job.py` located in the `examples\tutorial6` directory of the distribution.

3.6.1 Running Sesame on a cluster

Next we give an example of running Sesame on a computing cluster. This can be accomplished in several different ways; the most efficient way depends on the details of the cluster environment. For our example, the prerequisites libraries are:

- MPI Library
- mpi4py

The basic idea is illustrated in schematic below (note we follow MPI and python convention where indices start from 0). We have more than one processor available to execute all of the (independent) jobs. In this case, we'll assign the jobs as shown in the figure:



Any executable which uses MPI is called from the command line using the prefix `mpirun`. For example, the script "parallel_batch_example.py" is run on 32 processors with the following command:

```
mpirun -np 32 python3 parallel_batch_example.py
```

3.6.2 Parallel script description

We first import some additional packages:

```
import numpy as np
import sesame
import itertools
from mpi4py import MPI
```

The first half of the script contains a function called `system`. The `system` function takes parameter values as input and constructs a system object. This works as in previous tutorials, so we'll skip over it for now and begin with the second half contains a block of code, which begins:

```
if __name__ == '__main__':
```

Calling "parallel_batch_example.py" runs the code within the block contained in this "main" block. In the next section we describe the "main" code, which performs the actual parallel-ization of the job.

3.6.3 Cycling over parameters on a computing cluster

The primary task of the "main" script is to distribute independent serial jobs among an arbitrary number of processors. This is the easiest type of parallel computing (known as "embarassingly parallel"), and requires only a couple of additional lines of code.

We first initialize the MPI library with the command:

```
mpi_comm = MPI.COMM_WORLD
```

We next determine the total number of processors available with `mpi_comm.Get_size()`, and the "rank" of a particular instance of the program using `mpi_comm.Get_rank()`. In the schematic shown at the beginning, `mpisize=4`, and each processor is assigned its own rank: 0, 1, 2, or 3. The processors will all run the code identically, with the exception that each has its own unique value of mpirank:

```
mpirank = mpi_comm.Get_rank()
mpisize = mpi_comm.Get_size()
```

We define the set of parameter lists we want to study:

```
rho_GBlist = [1e11, 1e12, 1e13]          # [1/cm^2]
E_GBlist = [-.3, 0, .3]                  # [eV]
S_GBlist = [1e-14, 1e-15, 1e-16]         # [cm^2]
taulist = [1e-7, 1e-8, 1e-9]             # [s]
```

We use the itertools product function to form a list of all combinations of parameter values. The total number of jobs (`njobs` is equal to the product of the length of all parameter lists. This can get quite large if we vary several parameters (for this case we have 180 jobs):

```
paramlist = list(itertools.product(rho_GBlist, E_GBlist, S_GBlist,  taulist))
njobs = len(paramlist)
```

Here's where the parallel-ization of the batch processes enters. Each node only needs to compute a subset of all parameters. We set the relevant parameters for each node with the function `range`. The inputs are: starting index, ending index, step size:

```
my_param_indices = range(mpirank,njobs,mpisize)
```

This partitions the jobs as shown at the top of the page: each processor starts on a different job (the first job index equals the `mpirank` value) and skips over `mpisize` jobs to the next one. In this way we cover all jobs roughly equally between all the processors.

Next we define two arrays in which to store the computed I-V values. One of them is a local array, the other is a "global" array into which all the computed values will be set at the end of the program:

```
# Define array to store computed J-V values
jvset_local = np.zeros([njobs, len(voltages)])
jvset = np.zeros([njobs, len(voltages)])
```

Most of the parallel-ization is completed. The only remaining parallel component of code occurs at the very end, when we compile the results from all the processors into a single array.

3.6.4 Defining the system

We define a function which takes in a list of parameters. In this case, the parameters are ρ_{GB}, E_{GB}, S_{GB}, τ. The construction of the system follows the previous examples:

```
def system(params):

    # we assume the params are given in order: [rho_GB, E_GB, S_GB, tau]
    rho_GB = params[0]
    E_GB = params[1]
    S_GB = params[2]
    tau = params[3]

    # Dimensions of the system
    Lx = 3e-4   # [cm]
    Ly = 3e-4   # [cm]

    # Mesh
```

(continues on next page)

(continued from previous page)

```
    x = np.concatenate((np.linspace(0, 0.2e-4, 30, endpoint=False),
                        np.linspace(0.2e-4, 1.4e-4, 60, endpoint=False),
                        np.linspace(1.4e-4, 2.7e-4, 60, endpoint=False),
                        np.linspace(2.7e-4, 2.98e-4, 30, endpoint=False),
                        np.linspace(2.98e-4, Lx, 10)))

    y = np.concatenate((np.linspace(0, 1.25e-4, 60, endpoint=False),
                        np.linspace(1.25e-4, 1.75e-4, 50, endpoint=False),
                        np.linspace(1.75e-4, Ly, 60)))

    sys = sesame.Builder(x, y)      # Create a system

    # Dictionary with the material parameters
    mat = {'Nc': 8e17, 'Nv': 1.8e19, 'Eg': 1.5, 'epsilon': 9.4, 'Et': 0,
           'mu_e': 320, 'mu_h': 40, 'tau_e': tau, 'tau_h': tau}

    sys.add_material(mat)           # Add the material to the system

    junction = .1e-4   # [cm]

    # Define a function specifiying the n-type region
    def region1(pos):
        x, y = pos
        return x < junction

    # Define a function specifiying the p-type region
    def region2(pos):
        x, y = pos
        return x >= junction

    nD = 1e17                            # Donor density [cm^-3]
    sys.add_donor(nD, region1)           # Add the donors
    nA = 1e15                            # Acceptor density [cm^-3]
    sys.add_acceptor(nA, region2)                # Add the acceptors


    # Define contacts: CdS and CdTe contacts are Ohmic
    sys.contact_type('Ohmic', 'Ohmic')
    Sn_left, Sp_left, Sn_right, Sp_right = 1e7, 1e7, 1e7, 1e7
    sys.contact_S(Sn_left, Sp_left, Sn_right, Sp_right)

    # Specify the two points that make the line containing additional   charges
    p1 = (0.1e-4, 1.5e-4)     # [cm]
    p2 = (2.9e-4, 1.5e-4)     # [cm]

    # Add donor defect along GB
    sys.add_line_defects([p1, p2], rho_GB, S_GB, E=E_GB, transition=(1, 0))
    # Add acceptor defect along GB
    sys.add_line_defects([p1, p2], rho_GB, S_GB, E=E_GB, transition=(0, -1))

    return sys
```

Here we define the set of applied voltages:

```
# Specify applied voltages
voltages = np.linspace(0, .9, 10)
```

Now we cycle over all the parameter sets which apply to a given node:

```
# cycle over all parameter sets
for myjobcounter in my_param_indices:

    # Get system for given set of parameters
    params = paramlist[myjobcounter]
    sys = system(params)

    # Get equilibrium solution
    eqsolution = sesame.solve(sys, 'Poisson')

    # Define a function for generation profile
    f = lambda x, y: 2.3e21 * np.exp(-2.3e4 * x)
    # add generation to the system
    sys.generation(f)

    # Specify output filename for given parameter set
    outputfile = ''
    for paramvalue in params:
        outputfile = outputfile + '{0}_'.format(paramvalue)

    # Compute J-V curve
    jv = sesame.IVcurve(sys, voltages, outputfile, guess=eqsolution)
    # Save computed J-V in array
    jvset_local[myjobcounter,:] = jv
```

To combine the output of all the processers, we use `mpi_comm.Reduce`. The first argument is the local value of jv; the second argument is the global jv array. The local arrays will be added together and stored in the global array:

```
mpi_comm.Reduce(jvset_local,jvset)
```

Finally we save the global array of jv values, together with the list of parameters in a file “JVset”:

```
np.savez("JVset", jvset, paramlist)
```

CHAPTER

FOUR

USING SESAME WITH THE GUI

The Sesame GUI can be launched from the command line by navigating to the base sesame directory and executing:

```
python app.py
```

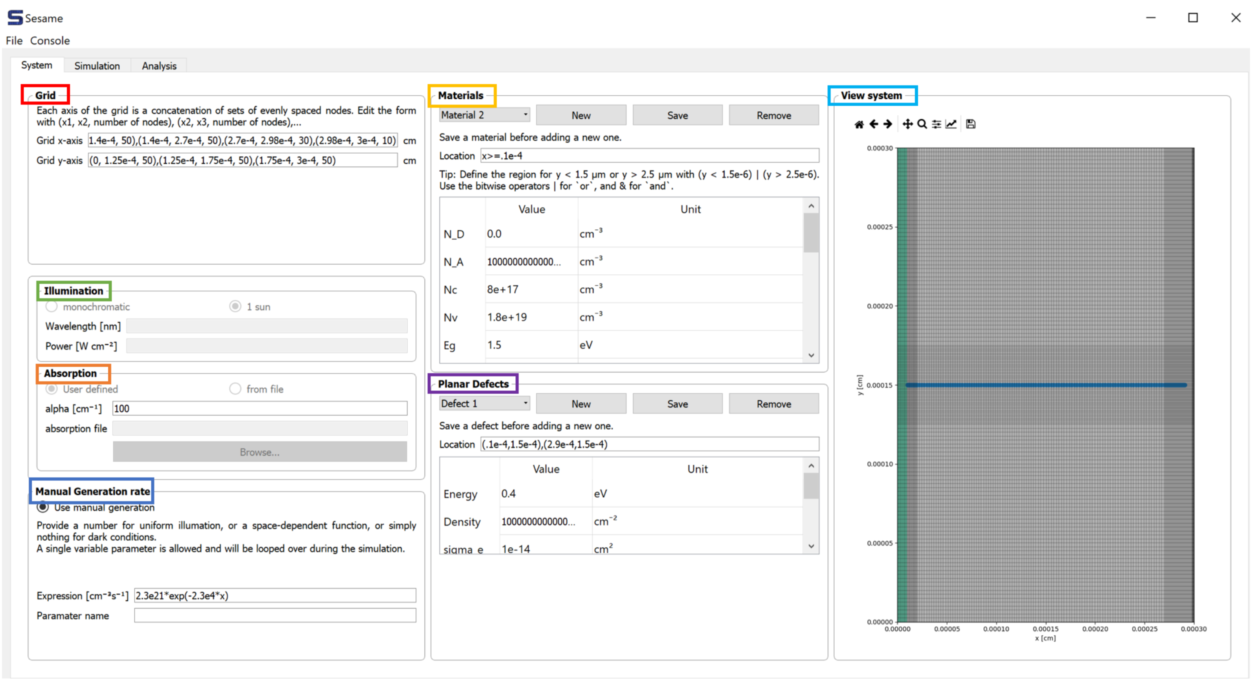
Alternatively, the GUI can be installed as a standalone executable (currently available only for Windows).

The Sesame GUI has three main tabs - system, simulation, and analysis. The File menu enables users to save and load GUI simulation files. The View menu provides options for viewing the spatial distribution of defects and certain system properties (like mobility and lifetime). The `Console` menu provides the option of an interactive python console below the main window.

In the following subsections we detail the content and use of each of the window tabs.

4.1 System Tab

The first step in performing simulations with the GUI is defining the system. The layout of the system tab is shown below:



Grid

Each axis of the grid is a concatenation of sets of evenly spaced nodes. Edit the form with (x1, x2, number of nodes), (x2, x3, number of nodes),...

Grid x-axis: 1.4e-4, 50),(1.4e-4, 2.7e-4, 50),(2.7e-4, 2.98e-4, 30),(2.98e-4, 3e-4, 10) cm

Grid y-axis: (0, 1.25e-4, 50),(1.25e-4, 1.75e-4, 50),(1.75e-4, 3e-4, 50) cm

Illumination

monochromatic [✓ 1 sun]

Wavelength [nm]

Power [W cm⁻²]

Absorption

[✓ User defined] from file

alpha [cm⁻¹]: 100

absorption file

Browse...

Manual Generation rate

[✓ Use manual generation]

Provide a number for uniform illumation, or a space-dependent function, or simply nothing for dark conditions.
A single variable parameter is allowed and will be looped over during the simulation.

Expression [cm⁻³s⁻¹]: 2.3e21*exp(-2.3e4*x)

Paramater name

Materials

Material 2 | New | Save | Remove

Save a material before adding a new one.

Location: x>=.1e-4

Tip: Define the region for y < 1.5 µm or y > 2.5 µm with (y < 1.5e-6) | (y > 2.5e-6).
Use the bitwise operators | for `or`, and & for `and`.

	Value	Unit
N_D	0.0	cm^{-3}
N_A	1000000000000...	cm^{-3}
Nc	8e+17	cm^{-3}
Nv	1.8e+19	cm^{-3}
Eg	1.5	eV

Planar Defects

Defect 1 | New | Save | Remove

Save a defect before adding a new one.

Location: (.1e-4,1.5e-4),(2.9e-4,1.5e-4)

	Value	Unit
Energy	0.4	eV
Density	1000000000000...	cm^{-2}
sigma_e	1e-14	cm^2

Grid: The upper left frame contains inputs for the user-defined grid. The dimensionality of the system is inferred from the grid entries: entering a grid for the x-axis only implies a 1-d system. Adding a grid for the y-axis implies a 2-d system, etc.

Materials: The middle-top frame contains the entries for the bulk material properties. Multiple materials can be added to the system. Their location is specified in the `Location` field.

View system: Provides a depiction of the system geometry, including the x and y meshes, different materials regions, and the planar defects (grain boundaries).

Planar Defects: Planar defects are added in the middle-bottom frame. For a 1-d system, a planar defect location is specified by 1 point. For a 2-d system, a planar defect is specified by 2 points. The energy level of the defect (with respect to the intrinsic energy level), the defect density, capture cross sections, and charge states are specified here.

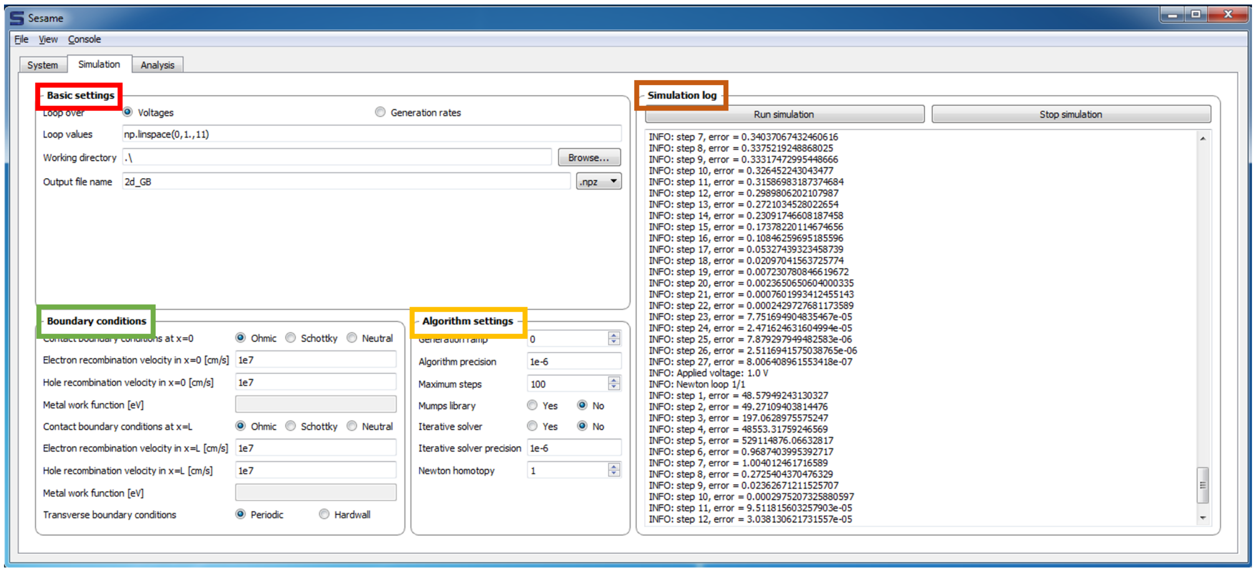
Illumination Specify the illumination on the system. Options are 1. monochromatic, with user-defined wavelength and power, and 2. 1 sun illumination, where the spectral density is taken from https://www.nrel.gov/grid/solar-resource/spectra-am1.5.html (Direct+circumsolar spectrum).

Absorption Specify the material absorption. Options are 1. a user-defined single absorption coefficient, and 2. loading in absorption spectrum from an external file. Tutorial2 has a sample absorption file. Note that the format of the file is assumed to be two column, with first column containing wavelength of light in units of [nm], and the second column containing the associated absorption coefficient in units of [1/m].

Generation Rate: The carrier generation rate density is entered in the lower-left frame. There is an option to vary 1 parameter in this profile. In cases where the generation profile changes, the parameter which is varied must be specified in the field `parameter name`. If the generation profile does not change, this field may be left blank.

4.2 Simulation Tab

The system tab is used to define the conditions of the simulation (e.g. the applied voltage and contact boundary conditions). The system window is shown below:



Basic settings

Loop over: [✓ Voltages] Generation rates
Loop values: np.linspace(0,1.,11)
Working directory: .\
Output file name: 2d_GB .npz

Boundary conditions

Contact boundary conditions at x=0: [✓ Ohmic] Schottky Neutral
Electron recombination velocity in x=0 [cm/s]: 1e7
Hole recombination velocity in x=0 [cm/s]: 1e7
Metal work function [eV]:
Contact boundary conditions at x=L: [✓ Ohmic] Schottky Neutral
Electron recombination velocity in x=L [cm/s]: 1e7
Hole recombination velocity in x=L [cm/s]: 1e7
Metal work function [eV]:
Transverse boundary conditions: [✓ Periodic] Hardwall

Algorithm settings

Generation ramp: 0
Algorithm precision: 1e-6
Maximum steps: 100
Mumps library: Yes [✓ No]
Iterative solver: Yes [✓ No]
Iterative solver precision: 1e-6
Newton homotopy: 1

Simulation log

```
INFO: step 7, error = 0.340370674324606l6
INFO: step 8, error = 0.3375219248868025
INFO: step 9, error = 0.3331747299544866 6
INFO: step 10, error = 0.326452243043477
INFO: step 11, error = 0.31586983187374684
INFO: step 12, error = 0.2989806202107987
INFO: step 13, error = 0.2721034528022654
INFO: step 14, error = 0.23091746608187458
INFO: step 15, error = 0.17378220114674656
INFO: step 16, error = 0.10846259695185596
INFO: step 17, error = 0.05327439323458739
INFO: step 18, error = 0.02097041563725774
INFO: step 19, error = 0.007230780846619672
INFO: step 20, error = 0.0023650650604000335
INFO: step 21, error = 0.0007601993412455143
INFO: step 22, error = 0.0002429727681173589
INFO: step 23, error = 7.751694904835467e-05
INFO: step 24, error = 2.471624631604994e-05
INFO: step 25, error = 7.879297949482583e-06
INFO: step 26, error = 2.5116941575038765e-06
INFO: step 27, error = 8.006408961553418e-07
INFO: Applied voltage: 1.0 V
INFO: Newton loop 1/1
INFO: step 1, error = 48.57949243130327
INFO: step 2, error = 49.27109403814476
INFO: step 3, error = 197.0628975575247
INFO: step 4, error = 48553.31759246569
INFO: step 5, error = 529114876.06632817
INFO: step 6, error = 0.9687403995392717
INFO: step 7, error = 1.004012461716589
INFO: step 8, error = 0.2725404370476329
INFO: step 9, error = 0.02362671211525707
INFO: step 10, error = 0.0002975207325880597
INFO: step 11, error = 9.511815603257903e-05
INFO: step 12, error = 3.038130621731557e-05
```

Basic Settings The GUI allows looping over two types of variables: the applied voltage and a user-defined parameter in the generation rate (this variable is defined in the bottom-left field of the `System` tab). The Loop values can be specified with `numpy` functions for array construction (as in the example shown above), or can be specified manually (e.g. [0, .1, .2]).

The working directory and output file name determine the location and name of simulation output files. The files can be saved in python format (.npz) or matlab format (.mat). The output filenames are appended by "_x", where x labels the loop index.

Boundary Conditions Boundary conditions are specified next. Possible choices for contact boundary conditions Ohmic, Schottky, or Neutral, and recombination velocity for electrons/holes are specified here. Transverse boundary conditions can be periodic or hardwall.

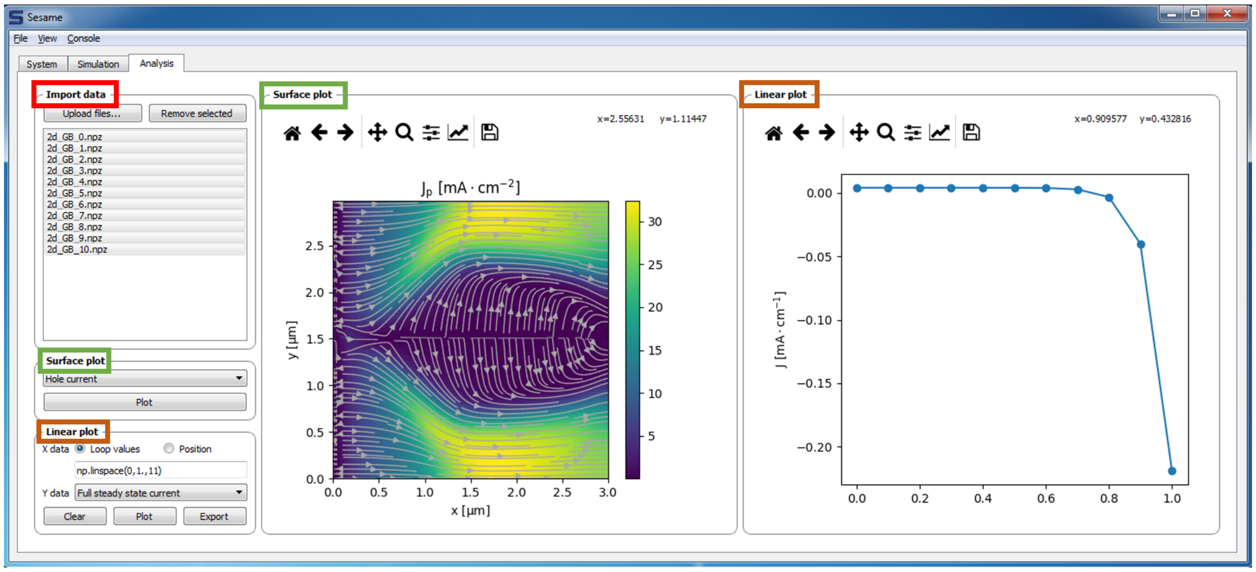
Algorithm Settings The algorithm settings are as follows: There's an option to ramp the generation rate. This can be useful to assist in convergence. If a system under illumination fails to converge, one can increase the generation ramp value. Setting the value to "n" means the initial illumination is decreased by a factor of 10^n. The generation is increased by factors of 10 until the desired generation rate is reached.

The `Algorithm precision` sets the error tolerance for the root-finding algorithm. A value of 1e-6 is typically acceptable. `Maximum steps` determines how many steps the Newton-Raphson algorithm attempts before aborting. The `Mumps library` option can be selected if the MUMPS library is installed. This library increases the speed up to several times, depending on system size. There is also an option for `iterative solver`, which is less accurate, but faster for very large systems (such as 3-d systems). The iterator solver precision is specified in the following field.

Simulation Log The simulation is launched with the `Run simulation` button on the top right. The log window gives a real-time output of the error in the solution obtained with the Newton-Algorithm. The solution procedure can be aborted at any time by clicking `Stop simulation`.

4.3 Analysis Tab

After successful completion of the simulation, the `Analysis tab` provides tools to plot and analyze the solution.



Import data The upper-left frame lets the user specify the output files to analyze. This list of files will be automatically populated by the output files of a successful simulation specified in the `Simulation tab`.

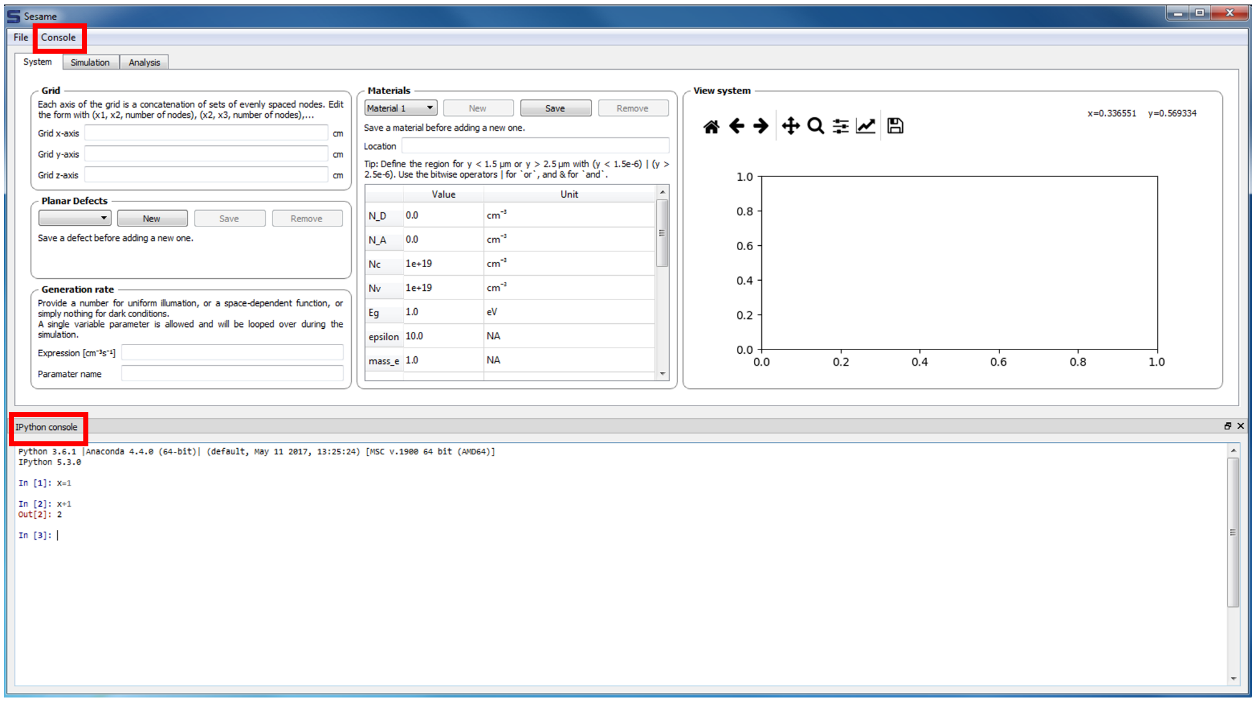
Surface plot For plots of quantities in 2-dimensions, select an outputfile, and in the `surface plot`, select a quantity to plot. Clicking `Plot` will give the spatially resolved color contour plot of the given quantity.

Linear plot For 1-d plots, two options are available: "loop values" and "position". For "loop values", a single scalar quantity (e.g. current density, total recombination) is plotted against the values of the looped parameter. Selecting "loop values" automatically selects all output files, and automatically fills in the X data field with the loop values. This is shown in the graphic above.

If “Position” is selected, then a plot of a quantitiy versus position is given. For a 2-d simulation, the position is given by two endpoints of a line (note the position of the endpoints are assumed to be given in units of [cm]). This is specified by the user in the “S data” field. For a 1-d simulation, the “X data” field is automatically set to the entire 1-d grid.

4.4 Running scripts in the GUI

The standalone Sesame GUI executable contains a full Python distribution. This can be accessed by selecting the `Console` → `Show Console` option from the window. An IPython console is displayed, as shown below. Below we show how Sesame scripts may be run from the python console:



Each axis of the grid is a concatenation of sets of evenly spaced nodes. Edit the form with (x1, x2, number of nodes), (x2, x3, number of nodes),...

Save a defect before adding a new one.

Provide a number for uniform illumation, or a space-dependent function, or simply nothing for dark conditions.
A single variable parameter is allowed and will be looped over during the simulation.

Save a material before adding a new one.

Tip: Define the region for y < 1.5 µm or y > 2.5 µm with (y < 1.5e-6) | (y > 2.5e-6). Use the bitwise operators | for `or`, and & for `and`.

	Value	Unit
N_D	0.0	cm^{-3}
N_A	0.0	cm^{-3}
Nc	1e+19	cm^{-3}
Nv	1e+19	cm^{-3}
Eg	1.0	eV
epsilon	10.0	NA
mass_e	1.0	NA

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)]
IPython 5.3.0

In [1]: x=1

In [2]: x+1
Out[2]: 2

In [3]: |
```

The first step is to navigate to the directory with the python script you would like to run. This is accomplished with the following commands, entered into the IPython prompt:

```python
import os
os.chdir("examples/tutorial1")
```

Next the python script (in this case called “1d_homojunction.py”) is called with the following command:

```python
exec(open("1d_homojunction.py").read())
```

The example scripts and GUI input files provided in the examples/tutorials folders define identical simulations. The user can modify one or the other as needed, and run either one within the GUI. Scripts provide more flexibility for system definition and for simulation actions, e.g. looping over several variables.

CHAPTER

FIVE

CORE MODULES

The following modules are used for internal and external use.

5.1 `sesame` – Top level package

We provide short names for a few objects and methods from the sub-packages. Otherwise, this package has no functionality of its own.

The version of Sesame is available under the name `__version__`.

5.1.1 From `sesame.builder`

Scaling([input_length, T])	An object defining the scalings of the drift-diffusion-Poisson equation.
Builder(xpts[, ypts, input_length, T, periodic])	A system discretized on a mesh.

5.1.2 From `sesame.plotter`

plot(sys, data[, cmap, alpha, fig, title])	Plot a 2D map of a parameter (like mobility) across the system.
plot_line_defects(sys[, ls, fig])	Plot the sites containing additional charges located on lines in 2D.

5.1.3 From `sesame.solvers.default`

solve(system[, compute, guess, tol, . . .])	Solve the drift diffusion Poisson equation on a given discretized system out of equilibrium.
IVcurve(system, voltages, file_name[, . . .])	Solve the Drift Diffusion Poisson equations for the voltages provided.

5.1.4 From `sesame.analyzer`

Analyzer(sys, data)	Object that simplifies the extraction of physical data (densities, currents, recombination) across the system.

5.1.5 From `sesame.utils`

save_sim(sys, result, filename[, fmt])	Utility function that saves a system together with simulation results.
load_sim(filename)	Utility function that loads a system together with simulation results.

5.2 `sesame.builder` – High-level construction of systems

class `sesame.builder.`**Scaling**(*input_length='cm'*, *T=300*)

An object defining the scalings of the drift-diffusion-Poisson equation. The temperature of the system and the reference unit for lengths are specified when an instance is created. The default unit for length is cm, and the default temperature is 300 K.

Parameters input_length: string

Reference unit for lengths. Acceptable entries are 'cm' for centimeters and 'm' for meters.

T: float

Temperature for the simulation.

Attributes

denisty: float	Density scale taken equal to 10^{19} cm^{-3}.
energy: float	Energy scale.
mobility: float	Mobility scale taken equal to 1 cm^2/(V.s).
time: float	Time scale.
length: float	Length scale.
generation: float	Scale of generation and recombination rates.
velocity: float	Velocity scale.
current: float	Electrical current density scale.

class `sesame.builder.`**Builder**(*xpts*, *ypts=array([0.])*, *input_length='cm'*, *T=300*, *periodic=True*)

A system discretized on a mesh.

This type discretizes a system on a mesh provided by the user, and takes care of all normalizations. The temperature of the system is specified when an instance is created. The default is 300 K.

Parameters xpts, ypts, zpts: numpy arrays of floats

Mesh with original dimensions.

input_length: string

Reference unit for lengths. Acceptable entries are 'cm' for centimeters and 'm' for meters.

T: float

Temperature for the simulation.

Attributes

xpts, ypts, zpts: numpy arrays of floats	Mesh with original dimensions.
dx, dy, dz: numpy arrays of floats	Dimensionless lattice constants in the x, y, z directions.
nx, ny: integers	Number of lattice nodes in the x, y directions.
Nc, Nv: numpy arrays of floats	Dimensionless effective densities of states of the conduction and valence bands.
Eg: numpy array of floats	Dimensionless band gap.
ni: numpy array of floats	Dimensionless intrinsic density.
mu_e, mu_h: numpy arrays of floats	Dimensionless mobilities of electron and holes.
tau_e, tau_h: numpy arrays of floats	Dimensionless bulk lifetime for electrons and holes.
n1, p1: numpy arrays of floats	Dimensionless equilibrium densities of electrons and holes at the bulk trap state.
bl: numpy array of floats	Electron affinity.
g: numpy array of floats	Dimensionless generation for each site of the system. This is defined only if a generation profile was provided when building the system.
gtot: float	Dimensionless integral of the generation rate.
defects_list: list of named tuples	List of named tuples containing the characteristics of the defects in the order they were added to the system. The field names are sites, location, dos, energy, sigma_e, sigma_h, transition, perp_dl.

`add_acceptor`(*density*, *location=<function Builder.<lambda>>*)

Add acceptor dopants to the system.

Parameters density: float

Doping density [cm^{-3}].

location: Boolean function

Definition of the region containing the doping. This function must take a tuple of real world coordinates (e.g. (x, y)) as parameters, and return True (False) if the lattice node is inside (outside) the region.

`add_defects`(*location*, *N*, *sigma_e*, *sigma_h=None*, *E=None*, *transition=(1, -1)*)

Add additional charges (for a grain boundary for instance) to the total charge of the system. These charges are distributed on a line.

Parameters location: float or list of two array_like coordinates [(x1, y1), (x2, y2)]

Coordinate(s) in [cm] of a point defect or the two end points of a line defect.

N: float or function

Defect density of states [cm^{-2}]. Provide a float when the defect density of states is a delta function, or a function returning a float for a continuum. This function should take a single energy argument in [eV].

sigma_e: float

Electron capture cross section [cm^2].

sigma_h: float (optional)

Hole capture cross section [cm^2]. If not given, the same value as the electron capture cross section will be used.

E: float

Energy level of a single state defined with respect to the intrinsic Fermi level [eV]. Set to *None* for a continuum of states (default).

transition: tuple

Charge transition occurring at the energy level E. The tuple (p, q) represents a defect with transition p/q (level empty to level occupied). Default is (1,-1).

`add_donor`(*density*, *location=<function Builder.<lambda>>*)
Add donor dopants to the system.

Parameters density: float

Doping density [cm^{-3}].

location: Boolean function

Definition of the region containing the doping. This function must take a tuple of real world coordinates (e.g. (x, y)) as parameters, and return True (False) if the lattice node is inside (outside) the region.

`add_material`(*mat*, *location=<function Builder.<lambda>>*)
Add a material to the system.

Parameters mat: dictionary

Contains the material parameters Keys are Nc (Nv): conduction (valence) effective densities of states [cm^{-3}], Eg: band gap [eV], epsilon: material's permitivitty, mu_e (mu_h): electron (hole) mobility [m^2/(V s)], tau_e (tau_h): electron (hole) bulk lifetime [s], Et: energy level of the bulk recombination centers [eV], affinity: electron affinity [eV], B: radiation recombination constant [cm^3/s], Cn (Cp): Auger recombination constant for electrons (holes) [cm^6/s], mass_e (mass_h): effective mass of electrons (holes). All parameters can be scalars or callable functions similar to the location argument.

location: Boolean function

Definition of the region containing the material. This function must take a tuple of real world coordinates (e.g. (x, y)) as parameter, and return True (False) if the lattice node is inside (outside) the region.

`contact_S`(*Scn_left*, *Scp_left*, *Scn_right*, *Scp_right*)
Create the lists of recombination velocities that define the charge collection at the contacts out of equilibrium.

Parameters Scn_left: float

Surface recombination velocity for electrons at the left contact [cm/s].

Scp_left: float

Surface recombination velocity for holes at the left contact [cm/s].

Scn_right: float

Surface recombination velocity for electrons at the right contact [cm/s].

Scn_right: float

Surface recombination velocity for electrons at the right contact [cm/s].

`contact_type` (*left_contact*, *right_contact*, *left_wf=None*, *right_wf=None*)

Define the boundary conditions for the contacts at thermal equilibrium. 'Ohmic' or 'Schottky' impose a value of the electrostatic potential, 'Neutral' imposes a zero potential derivative.

Parameters left_contact: string

Boundary conditions for the contact at x=0.

right_contact: string

Boundary conditions for the contact at x=L.

left_wf: float

Work function for a Schottky contact at x=0.

right_wf: float

Work function for a Schottky contact at x=L.

Notes

Schottky contacts require work functions. If no values are given, an error will be raised.

`generation` (*f*, *args=[]*)

Distribution of generated carriers.

Parameters f: function

Generation rate [cm^{-3}].

args: tuple

Additional arguments to be passed to the function.

5.3 `sesame.plotter` – Plotting of systems

`plot`(sys, data[, cmap, alpha, fig, title])	Plot a 2D map of a parameter (like mobility) across the system.
`plot_line_defects`(sys[, ls, fig])	Plot the sites containing additional charges located on lines in 2D.

5.3.1 sesame.plotter.plot

`sesame.plotter.`**`plot`** (*sys*, *data*, *cmap='gnuplot'*, *alpha=1*, *fig=None*, *title=''*)

Plot a 2D map of a parameter (like mobility) across the system.

Parameters sys: Builder

The discretized system.

data: numpy array

One-dimensional array of data with size equal to the size of the system.

cmap: string

Name of the colormap used by Matplolib.

alpha: float

Transparency of the colormap.

fig: Maplotlib figure

A plot is added to it if given. If not given, a new one is created and displayed.

5.3.2 sesame.plotter.plot_line_defects

`sesame.plotter.`**`plot_line_defects`**(*sys*, *ls='-o'*, *fig=None*)

Plot the sites containing additional charges located on lines in 2D. The length scale of the graph is 1 micrometer by default.

Parameters sys: Builder

The discretized system.

ls: string

Line style of the plotted paths.

fig: Maplotlib figure

A plot is added to it if given. If not given, a new one is created and a figure is displayed.

5.4 `sesame.solvers` – Equilibrium and nonequilibrium solvers

Sesame offers several solvers to address quickly specific problems (equilibrium potential, IV curve) without having to manage a cumbersome machinery to get to the solution.

5.4.1 `sesame.solvers.default` – Default solver

The functions below belong to the default solver *`sesame.solvers.default`* which is created upon loading sesame. We made these functions available once sesame is loaded making the solver completely transparent for the user. The solver stores the equilibrium potential of the system as soon as it has been computed.

`solve`(system[, compute, guess, tol, . . .])	Solve the drift diffusion Poisson equation on a given discretized system out of equilibrium.
`IVcurve`(system, voltages, file_name[, . . .])	Solve the Drift Diffusion Poisson equations for the voltages provided.

sesame.solvers.default.solve

`classmethod` `default.`**`solve`**(*system*, *compute='all'*, *guess=None*, *tol=1e-06*, *periodic_bcs=True*, *maxiter=300*, *verbose=True*, *htp=1*)

Solve the drift diffusion Poisson equation on a given discretized system out of equilibrium. If the equilibrium electrostatic potential is not yet computed, the routine will compute it and save it for further computations.

Parameters system: Builder

The discretized system.

compute: string

Set to 'all' to solve the full drift-diffusion-Poisson equations, or to 'Poisson' to only solve the Poisson equation. Default is set to 'all'.

guess: dictionary of numpy arrays of floats

Contains the one-dimensional arrays of the initial guesses for the electron quasi-Fermi level, the hole quasi-Fermi level and the electrostatic potential. Keys should be 'efn', 'efp' and 'v'.

tol: float

Accepted error made by the Newton-Raphson scheme.

periodic_bcs: boolean

Defines the choice of boundary conditions in the y-direction. True (False) corresponds to periodic (abrupt) boundary conditions.

maxiter: integer

Maximum number of steps taken by the Newton-Raphson scheme.

verbose: boolean

The solver returns the step number and the associated error at every step if set to True (default).

htp: integer

Number of homotopic Newton loops to perform.

Returns solution: dictionary with numpy arrays of floats

Dictionary containing the one-dimensional arrays of the solution. The keys are the same as the ones for the guess. An exception is raised if no solution could be found.

sesame.solvers.default.IVcurve

classmethod `default.`**`IVcurve`**(*system*, *voltages*, *file_name*, *guess=None*, *tol=1e-06*, *periodic_bcs=True*, *maxiter=300*, *verbose=True*, *htp=1*, *fmt='npz'*)

Solve the Drift Diffusion Poisson equations for the voltages provided. The results are stored in files with `.npz` format by default (See below for saving in Matlab format). The steady state current is computed at the end of the voltage loop and returned. Note that the potential is always applied on the right contact.

Parameters system: Builder

The discretized system.

voltages: array-like

List of voltages for which the current should be computed.

file_name: string

Name of the file to write the data to. The file name will be appended the index of the voltage list, e.g. `file_name_0.npz`.

guess: dictionary of numpy arrays of floats (optional)

Starting point of the solver. Keys of the dictionary must be 'efn', 'efp', 'v' for the electron and quasi-Fermi levels, and the electrostatic potential respectively.

tol: float

Accepted error made by the Newton-Raphson scheme.

periodic_bcs: boolean

Defines the choice of boundary conditions in the y-direction. True (False) corresponds to periodic (abrupt) boundary conditions.

maxiter: integer

Maximum number of steps taken by the Newton-Raphson scheme.

verbose: boolean

The solver returns the step number and the associated error at every step, and this function prints the current applied voltage if set to True (default).

htp: integer

Number of homotopic Newton loops to perform.

fmt: string

Format string for the data files. Use `mat` to save the data in a Matlab format (version 5 and above).

Returns J: numpy array of floats

Steady state current computed for each voltage value.

Notes

The data files can be loaded and used as follows:

```
>>> results = np.load('file.npz')
>>> efn = results['efn']
>>> efp = results['efp']
>>> v = results['v']
```

5.4.2 Creating another solver

Making another solver is done by creating an instance of the *sesame.solvers.Solver* class. This can be used to turn off the use of the MUMPS library even when the library is available.

`sesame.solvers.Solver` – Sesame solver oject

class sesame.solvers.**Solver**(*use_mumps=True*)

An object that creates an interface for the equilibrium and nonequilibrium solvers of Sesame, and stores the equilibrium electrostatic potential once computed.

Parameters **use_mumps: boolean**

Flag for the use of the MUMPS library if available. The flag is set to True by default. If the MUMPS library is absent, the flag has no effect.

Attributes

equilibrium: numpy array of floats	Electrostatic potential computed at thermal equilibrium.

5.5 `sesame.analyzer` – Computing densities, recombination and currents

Sesame provides several functions to compute densities, recombination and other integrated quantities to simplify the analysis of simulation data. We give a summary of these functions below:

`line`(p1, p2)	Compute the path and sites between two points.
`band_diagram`(location[, fig])	Compute the band diagram between two points defining a line.
`electron_density`([location])	Compute the electron density across the system or on a line defined by two points.
`hole_density`([location])	Compute the hole density across the system or on a line defined by two points.
`bulk_srh_rr`([location])	Compute the bulk Shockley-Read-Hall recombination across the system or on a line defined by two points.
`auger_rr`([location])	Compute the Auger recombination across the system or on a line defined by two points.
`radiative_rr`([location])	Compute the radiative recombination across the system or on a line defined by two points.
`defect_rr`(defect)	Compute the recombination for all sites of a defect (2D and 3D).
`total_rr`()	Compute the sum of all the recombination sources for all sites of the system.
`electron_current`([component, location])	Compute the electron current either by component (x or y) across the entire system, or on a line defined by two points.
`hole_current`([component, location])	Compute the hole current either by component (x or y) across the entire system, or on a line defined by two points.
`electron_current_map`([cmap, scale])	Compute a 2D map of the electron current.
`map3D`(data[, cmap, scale])	Plot a 3D map of data across the entire system.
`integrated_bulk_srh_recombination`()	Integrate the bulk Shockley-Read-Hall recombination over an entire system.
`integrated_auger_recombination`()	Integrate the Auger recombination over an entire system.
`integrated_radiative_recombination`()	Integrate the radiative recombination over an entire system.
`integrated_defect_recombination`(defect)	Integrate the recombination along a defect in 2D.
`full_current`()	Compute the steady state current in 1D and 2D.

All the functions are gathered in the *`sesame.analyzer.Analyzer()`* class.

class `sesame.analyzer.`**`Analyzer`**(*sys*, *data*)

Object that simplifies the extraction of physical data (densities, currents, recombination) across the system.

Parameters sys: Builder

A discretized system.

data: dictionary

Dictionary containing 1D arrays of electron and hole quasi-Fermi levels and the electrostatic potential across the system. Keys must be 'efn', 'efp', and/or 'v'.

`auger_rr`(*location=None*)

Compute the Auger recombination across the system or on a line defined by two points.

Parameters location: array-like ((x1,y1), (x2,y2))

Tuple of two points defining a line over which to compute the recombination.

Returns r: numpy array

An array with the values of recombination.

`band_diagram`(*location*, *fig=None*)
Compute the band diagram between two points defining a line. Display a plot if fig is None.

Parameters **location: array-like ((x1,y1), (x2,y2))**

Tuple of two points defining a line over which to compute a band diagram.

fig: Maplotlib figure

A plot is added to it if given. If not given, a new one is created and displayed.

`bulk_srh_rr`(*location=None*)
Compute the bulk Shockley-Read-Hall recombination across the system or on a line defined by two points.

Parameters **location: array-like ((x1,y1), (x2,y2))**

TUple of two points defining a line over which to compute the recombination.

Returns r: numpy array

An array with the values of recombination.

`defect_rr`(*defect*)
Compute the recombination for all sites of a defect (2D and 3D).

Parameters **defect: named tuple**

Container with the properties of a defect. The expected field names of the named tuple are sites, location, dos, energy, sigma_e, sigma_h, transition, perp_dl.

Returns r: numpy array of floats

An array with the values of recombination at each sites.

`electron_current`(*component='x'*, *location=None*)
Compute the electron current either by component (x or y) across the entire system, or on a line defined by two points.

Parameters **component: string**

Current direction `'x'` or `'y'`. By default returns all currents in the x-direction.

location: array-like ((x1,y1), (x2,y2))

Tuple of two points defining a line over which to compute the electron current.

Returns jn: numpy array of floats

`electron_current_map`(*cmap='gnuplot'*, *scale=10000.0*)
Compute a 2D map of the electron current.

Parameters **cmap: Matplotlib color map**

Color map used for the plot.

scale: float

Scale to apply to the axes of the plot.

`electron_density`(*location=None*)
Compute the electron density across the system or on a line defined by two points.

Parameters **location: array-like ((x1,y1), (x2,y2))**

Tuple of two points defining a line over which to compute the electron density.

Returns n: numpy array of floats

See also:

hole_density

full_current()
Compute the steady state current in 1D and 2D.

Returns J: float

The integrated full steady state current.

hole_current(*component='x'*, *location=None*)
Compute the hole current either by component (x or y) across the entire system, or on a line defined by two points.

Parameters **component: string**

Current direction `'x'` or `'y'`. By default returns all currents in the x-direction.

location: array-like ((x1,y1), (x2,y2))

Tuple of two points defining a line over which to compute the hole current.

Returns jp: numpy array of floats

hole_current_map(*cmap='gnuplot'*, *scale=10000.0*)
Compute a 2D map of the hole current of a 2D system.

Parameters **cmap: Matplotlib color map**

Color map used for the plot.

scale: float

Scale to apply to the axes of the plot.

hole_density(*location=None*)
Compute the hole density across the system or on a line defined by two points.

Parameters **location: array-like ((x1,y1), (x2,y2))**

Tuple of two points defining a line over which to compute the hole density.

Returns p: numpy array of floats

See also:

electron_density

integrated_auger_recombination()
Integrate the Auger recombination over an entire system.

Returns JR: float

The integrated Auger recombination.

Warning: Not implemented in 3D.

integrated_bulk_srh_recombination()
Integrate the bulk Shockley-Read-Hall recombination over an entire system.

Returns JR: float

The integrated bulk recombination.

Warning: Not implemented in 3D.

`integrated_defect_recombination`(*defect*)
Integrate the recombination along a defect in 2D.

Returns JD: float

The recombination integrated along the line of the defect.

Warning: Not implemented in 3D.

`integrated_radiative_recombination`()
Integrate the radiative recombination over an entire system.

Returns JR: float

The integrated radiative recombination.

Warning: Not implemented in 3D.

`static line`(*p1*, *p2*)
Compute the path and sites between two points.

Parameters system: Builder

The discretized system.

p1, p2: array-like (x, y)

Two points defining a line.

Returns s, sites: numpay arrays

Curvilinear abscissa and sites of the line.

Notes

This method can be used with an instance of the Analyzer():

```
>>> az = sesame.Analyzer(sys, res)
>>> X, sites = az.line(sys, p1, p2)
```

or without it:

```
>>> X, sites = sesame.Analyzer.line(sys, p1, p2)
```

`map3D`(*data*, *cmap='gnuplot'*, *scale=1e-06*)
Plot a 3D map of data across the entire system.

Parameters data: numpy array

One-dimensional array of data with size equal to the size of the system.

cmap: string

Name of the colormap used by Matplolib.

scale: float

Relevant scaling to apply to the axes.

radiative_rr(*location=None*)

Compute the radiative recombination across the system or on a line defined by two points.

Parameters **location: array-like ((x1,y1), (x2,y2))**

Tuple of two points defining a line over which to compute the recombination.

Returns r: numpy array

An array with the values of recombination.

total_rr()

Compute the sum of all the recombination sources for all sites of the system.

Returns r: numpy array of floats

An array with the values of the total recombination at each sites.

5.6 `sesame.observables` – Low-level routines for computing densities and currents

These routines are to be used when the *Analyzer()* class does not provide the desired methods.

get_n(sys, efn, v, sites)	Compute the electron density on the given sites.
get_p(sys, efp, v, sites)	Compute the hole density on the given sites.
get_jn(sys, efn, v, sites_i, sites_ip1, dl)	Compute the electron current between sites `site_i` and `sites_ip1`.
get_jp(sys, efp, v, sites_i, sites_ip1, dl)	Compute the hole current between sites `site_i` and `sites_ip1`.

5.6.1 sesame.observables.get_n

sesame.observables.**get_n**(*sys*, *efn*, *v*, *sites*)

Compute the electron density on the given sites.

Parameters **sys: Builder**

The discretized system.

efn: numpy array of floats

Values of the electron quasi-Fermi level.

v: numpy array of floats

Values of the electrostatic potential.

sites: list of integers

The sites where the electron density should be computed.

Returns n: numpy array

5.6.2 sesame.observables.get_p

`sesame.observables.`**`get_p`**(*sys*, *efp*, *v*, *sites*)

Compute the hole density on the given sites.

Parameters **sys: Builder**

The discretized system.

efp: numpy array of floats

Values of the hole quasi-Fermi level.

v: numpy array of floats

Values of the electrostatic potential.

sites: list of integers

The sites where the hole density should be computed.

Returns p: numpy array

5.6.3 sesame.observables.get_jn

`sesame.observables.`**`get_jn`**(*sys*, *efn*, *v*, *sites_i*, *sites_ip1*, *dl*)

Compute the electron current between sites `site_i` and `sites_ip1`.

Parameters **sys: Builder**

The discretized system.

efn: numpy array of floats

Values of the electron quasi-Fermi level for the entire system (as given by the drift diffusion Poisson solver).

v: numpy array of floats

Values of the electrostatic potential for the entire system (as given by the drift diffusion Poisson solver).

sites_i: list of integers

Indices of the sites the current is coming from.

sites_ip1: list of integers

Indices of the sites the current is going to.

dl: numpy arrays of floats

Lattice distances between sites `sites_i` and sites `sites_ip1`.

Returns jn: numpy array of floats

5.6.4 sesame.observables.get_jp

`sesame.observables.`**`get_jp`**(*sys*, *efp*, *v*, *sites_i*, *sites_ip1*, *dl*)

Compute the hole current between sites `site_i` and `sites_ip1`.

Parameters **sys: Builder**

The discretized system.

efp: numpy array of floats

Values of the hole quasi-Fermi level for the entire system (as given by the drift diffusion Poisson solver).

v: numpy array of floats

Values of the electrostatic potential for the entire system (as given by the drift diffusion Poisson solver).

sites_i: list of integers

Indices of the sites the current is coming from.

sites_ip1: list of integers

Indices of the sites the current is going to.

dl: numpy arrays of floats

Lattice distances between sites `sites_i` and sites `sites_ip1`.

Returns jp: numpy array of floats

5.7 `sesame.utils` – Miscellaneous routines

save_sim(sys, result, filename[, fmt])	Utility function that saves a system together with simulation results.
load_sim(filename)	Utility function that loads a system together with simulation results.

5.7.1 sesame.utils.save_sim

`sesame.utils.`**`save_sim`**(*sys*, *result*, *filename*, *fmt='npy'*)

Utility function that saves a system together with simulation results.

Parameters **sys: Builder**

The discretized system.

result: dictionary

Dictionary of solution, containing 'v', 'efn', 'efp'

filename: string

Name of outputfile

fmt: string

Format of output file, set to 'mat' for matlab files. With the default numpy format, the Builder object is saved directly.

5.7.2 sesame.utils.load_sim

`sesame.utils.`**`load_sim`**(*filename*)

Utility function that loads a system together with simulation results.

Parameters **filename: string**

Name of inputfile

Returns system: Builder object

A discritized system.

result: dictionary

Dictionary containing 1D arrays of electron and hole quasi-Fermi levels and the electrostatic potential across the system. Keys are 'efn', 'efp', and/or 'v'.

- genindex

PYTHON MODULE INDEX