CONTENTS

I Introduction 1

1 Installation instructions 3
  1.1 Prerequisites 3
  1.2 Windows installation procedure 3
  1.3 Linux installation procedure 4
  1.4 Quick start guide 4
  1.5 Bug reports / Feedback 5

2 GMSH Quick Tutorial 7
  2.1 GMSH Example 1D 7
  2.2 GMSH Example 2D 9

II Input 13

3 tiberCAD Input File 15
  3.1 Description of Input file structure 15
  3.2 Device section 16
  3.3 Modules 19
  3.4 Simulation section 24
  3.5 Output description 24
  3.6 Material database 25
  3.7 Input files Examples 27

4 Atomistic generator 37
  4.1 General Options 38
  4.2 Optional Blocks 39
  4.3 Importing an external file 41
  4.4 Applying periodicity 41
  4.5 Use of random alloy generator 42
  4.6 Example 43
### III Modules

#### 5 Elasticity

- 5.1 Theory ............................................. 47
- 5.2 Module options .................................. 49
- 5.3 Solution/Plot variables ......................... 49
- 5.4 Solver section .................................... 49
- 5.5 Physics section ................................ 49
- 5.6 Boundary conditions ............................. 53
- 5.7 Example 1: Strain in heterostructures ....... 55
- 5.8 Example 2: Piezoelectric nanogenerator ..... 57

#### 6 Thermal

- 6.1 Theory ............................................. 61
- 6.2 Solution/Plot variables ......................... 61
- 6.3 Solver section .................................... 61
- 6.4 Physics section ................................ 61
- 6.5 Boundary conditions ............................. 63
- 6.6 Example 1 ......................................... 64
- 6.7 Example 2 ......................................... 66

#### 7 Drift-diffusion simulation of electrons and holes

- 7.1 Theory ............................................. 73
- 7.2 Solution/Plot variables ......................... 73
- 7.3 Module options .................................. 73
- 7.4 Solver section .................................... 74
- 7.5 Physics section ................................ 75
- 7.6 Schrödinger/Poisson/Drift-Diffusion calculations ............................... 94
- 7.7 Example 1: pn diode ............................. 98
- 7.8 Example 2: Mosfet .............................. 99

#### 8 Quantum EFA calculations

- 8.1 Module efaschroedinger ........................ 107
- 8.2 Module opticskp .................................. 113
- 8.3 Example ............................................ 116

#### 9 Dye Solar Cell

- 9.1 Introduction ....................................... 125
- 9.2 Theory ............................................. 126
- 9.3 Device ............................................. 129
- 9.4 Module DSC ....................................... 129
- 9.5 Sweep .............................................. 133
- 9.6 Output .............................................. 133
- 9.7 Example ............................................ 133

#### 10 Valence Force Field

- 10.1 Module efaschroedinger ........................ 143
Part I

Introduction
CHAPTER ONE

INSTALLATION INSTRUCTIONS

In the following, installpath denotes the directory where tiberCAD 2.0.0 gets installed. Version 2.5.0 of GMSH (http://www.geuz.org/gmsh) will be installed together with tiberCAD.

1.1 Prerequisites

Get the installer package for your OS/architecture from http://www.tiberlab.com or by contacting support@tiberlab.com. Table Installer packages lists the packages available for download. To run tiberCAD you will also need a license file that you will have to copy into the installation directory of tiberCAD.

1.2 Windows installation procedure

To install tiberCAD in Windows, please run the setup program tibercad-2.0.0-i686-mingw32_setup.exe.

During the installation you can choose the installation directory. After finishing installation, copy your license file tibercad.lic into the license subdirectory of the tiberCAD installation directory (installpath/license), without changing the filename.

<table>
<thead>
<tr>
<th>Installer</th>
<th>Target architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>tibercad-2.0.0-i686-mingw32_setup.exe</td>
<td>Windows 32-bit</td>
</tr>
<tr>
<td>tibercad-2.0.0-i686-linux_installer.sh</td>
<td>Linux 32-bit</td>
</tr>
<tr>
<td>tibercad-2.0.0-x86_64-linux_installer.sh</td>
<td>Linux 64-bit</td>
</tr>
</tbody>
</table>

Installer Packages
1.3 Linux installation procedure

To install tiberCAD under Linux, download and run the self-extracting installer tibercad-2.0.0-ARCH_installer.sh, where ARCH corresponds to your hardware architecture, and follow the installation instructions.

After installation, copy your license file tibercad.lic into the license subdirectory of the tiberCAD installation path (installpath/license) without changing the filename. You can also provide the license file during installation.

tiberCAD is launched by means of a shell script, which is installed together with the tiberCAD executable. It takes care of setting all necessary environment variables.

If you have to call the executable directly, you have to set the environment variable TIBERCADROOT to the tiberCAD installation directory (installpath).

1.4 Quick start guide

In the examples subdirectory you can find several examples ready to run. More examples may be available on http://www.tiberlab.com or http://www.tibercad.org.

tiberCAD has the following command line options:

- \(-v\) print the version number and exit
- \(-b\) run in batch mode, without asking for user input.

The \(-b\) option is useful only for the windows version, which attends a keystroke by the user before exiting.

1.4.1 Windows

Open Windows Explorer and go to the tiberCAD installation directory. If you have write permission in the installation directory, you can browse to an example directory and start the simulation by double clicking the input file, e.g. bulk.tib in tut0step1. If not, copy the whole directory to a location in your personal area and run the examples from there.

If you cannot run tiberCAD by double clicking a tiberCAD input file (the files with extension .tib), then the input files may not be correctly associated with the tiberCAD executable. In this case, try to establish the association by right-clicking the input file, choosing

    open with... >> Choose Program... >> Browse...,

browsing to the tiberCAD installation directory and choosing the tiberCAD executable, tibercad.exe.
1.4.2 Linux

After a correct installation of tiberCAD you should be able to run tiberCAD from the command line using the command `tibercad`. If not, you probably have to add the `bin` subdirectory of the tiberCAD installation path to your `PATH` environment variable or start the tiberCAD executable using the absolute path (`installpath/bin/tibercad`).

Copy the directory of the example you want to run to your home directory or any place you have write permissions for. Change to the example directory and run tiberCAD by (assuming `tut0step1`)

```bash
$ tibercad bulk.tib
```

1.5 Bug reports / Feedback

Please send bug reports, feedback or suggestions to support@tibercad.org. When submitting bug reports, please always include the full version number of tiberCAD you are running. The full version number appears in the first line of output when running the program:

```bash
$ tibercad -v
tiberCAD version 2.0.0 (x86_64-linux)
```
To use tiberCAD, as a first step you need to model the device and generate a suitable mesh grid. This can be done by using DEVISE module of ISE-TCAD 9.5 software package or GMSH program.

In the following we will see how to write a basic GMSH script in 1 and 2D; for any details please refer to GMSH manual (http://geuz.org/gmsh/).

2.1 GMSH Example 1D

2.1.1 Step 1: Modeling the device

Here we will refer to the Example 0 Bulk Silicon in 1D in the Example directory. See Input_Ex1 for the description of the corresponding input file.

In a GMSH script, several variables can be defined and given a value in this way:

\[
\begin{align*}
L &= 1; \\
\Delta &= 0.01;
\end{align*}
\]

these are valid GMSH variables: \(L\) is just the length of the Si sample; \(\Delta\) is the value of a characteristic mesh length (see below).

GMSH modeling strategy: bottom-up design

In gmsh the idea is to design the model with a “bottom-up” strategy. So, first, points are defined, then lines connecting points, surface connecting lines, and so on. No superimposing objects are allowed. This means that, once defined your points, you may connect them with lines but different lines must not have parts in common (just points); the same works for surfaces: they may have only lines in common, but no intersections between surfaces are allowed.
If a geometrical model with not null intersections between entities (points, lines, surfaces, volumes) is created, unpredictable results may occur (gmsh crashes during meshing, a mesh is created which is not valid, etc.).

Definition of the geometrical entities Points

Point(1) = {0, 0, 0, d};
Point(2) = {L, 0, 0, d};

In the definition of a geometrical point, the first three expressions inside the braces on the right hand side give the three X, Y and Z coordinates of the point; the last expression $d$ sets the characteristic mesh length at that point, that is the size of a mesh element, defined as the length of the segment for a line mesh element, the radius of the circumscribed circle for a triangle mesh element and the radius of the circumscribed sphere for a tetrahedron mesh element.

Thus, the smaller is the value of $d$, the greater is the mesh density close to that point. The size of the mesh elements will then be computed in GMSH by linearly interpolating these characteristic lengths in the whole mesh.

In a 1D simulation it is assumed that the geometrical model is restricted to the $x$ axis. Any other geometrical orientation could give unpredictable results.

Definition of a geometrical entity Line

Line(1) = {1, 2};

The two expressions inside the braces on the right hand side give the identification numbers of the start and end points of the line.

Definition of the physical entity Physical Line “bulk” and of two physical entities Physical Point

Convenient Physical Names are to be assigned to the Physical entities. Physical Names consist of strings enclosed between quotation marks. The syntax is the following:

Physical Line("bulk") = {1}
............
Physical Point("Anode") = {1};
Physical Point("Cathode") = {2};

The expression(s) inside the braces on the right hand side give the identification numbers of all the geometrical lines that need to be grouped inside the Physical Line or Physical Point. In this way, in general, physical regions are created which associate together geometrical regions, and then the related mesh elements, which share some common physical properties. It’s only these physical regions which can be referred to outside GMSH. In tiberCAD, this is done by associating one or
more physical regions to a tiberCAD region through the keywords Region and mesh_regions (see Input File).

In general, in a n-Dimension (nD) simulation, (n-1)D physical regions (points in 1D, lines in 2D, surfaces in 3D) are used by tiberCAD to impose the required boundary conditions. Each (n-1)D physical region defined in this way in GMSH will be associated in tiberCAD to a boundary condition (Contact) region. Thus, in this case, Physical points Anode and Cathode will be associated respectively to two Contacts (see Input_Ex1).

2.1.2 Step 2: Meshing the device

The .geo script file with the geometrical description can be run in GMSH, to display the modelled device and to mesh it through the GMSH graphical interface. To generate the mesh, select Mesh in the main menu of GMSH and click on 1D, 2D or 3D depending on the dimension of your simulation. This will create a file .msh in your working directory.

Alternatively, a non-interactive mode is also available in GMSH, without graphical user interface. For example, to mesh this 1D tutorial in non-interactive mode, just type in the command line

```
gmsh bulk.geo -1 -o bulk.msh
```

where bulk.geo is the geometrical description of the device with GMSH syntax; -1 means 1D mesh generation;

some command line options are:

-1, -2, -3

to perform 1D, 2D or 3D mesh generation, respectively.

-o mesh_file.msh

to specify the name of the mesh file to be generated

In this way, a .msh has been generated and is ready to be read in tiberCAD.

2.2 GMSH Example 2D

In this second example we will refer to the Example 4 that you can find in the Example directory (mosfet.geo). See Input_Ex2 for a description of the Input file.
2.2.1 Modeling the device

Again, as a first step, we have to model the device.

Geometrical Points and Lines are defined to design the device structure; the fourth parameter in Point assignment is the characteristic length associated to that point: this is an essential feature to control the mesh density and refine it where necessary (usually in the channel region).

In a 2D simulation it is assumed that the geometrical model is restricted to the xy-plane \((z = 0)\). Any other geometrical orientation could give impredictable results

```
Point(1) = {0, -h, 0, lsub};
Point(2) = {0, 0, 0, lc};
Point(3) = {xmax,-h,0.0,lsub};
Point(4) = {-xmax,-h,0.0,lsub};
Point(5) = {xmax,0,0.0,lh};
Point(6) = {-xmax,0,0.0,lh};
..........................
Line(1) = {4,1};
Line(2) = {3,13};
Line(6) = {4,14};
Line(7) = {10,9};
Line(8) = {12,2};
Line(9) = {8,7};
Line(10) = {11,8};
Line(11) = {9,12};
Line(13) = {7,6};
..........................
```

Definition of a surface

First a line loop is composed, listing all the lines constituting the boundary of the surface; then this line loop is assigned to a Plane Surface object (this procedure can be alternatively performed through the graphical interface).

```
Line Loop(40) = {28,2,-34,33,8,29,-31,-30,-6,1};
Plane Surface(41) = {40};
..........................
```

The obtained geometrical surface is shown n Fig. Surface

Definition of the Physical Surfaces

Each of the Physical Surfaces is composed by one or more geometrical Plane Surface. For example, Physical surface contact comprises in one single physical region the two separated contact geometrical regions, while Physical surface oxide corresponds to the oxide region. The Physical
**User Manual, Release 2.5.0**

**Figure 2.1: Surface**

*surfaces* are the 2D Physical regions of the mesh and will be assigned to the related tiberCAD regions through the keyword *Region* and *mesh_regions*. (See *Input_Ex2*)

```
Physical Surface("substrate") = {41}; // n-Si
Physical Surface("contact") = {44,47}; // n+-Si
Physical Surface("oxide") = {46}; // SiO2
```

**Definition of the Physical Lines**

In this 2D simulation, 1D physical regions are used to carry information about boundary condition regions. In other words, each *Physical Line* corresponds to a boundary condition (a contact in the case of a drift diffusion calculation). Thus *Physical Line source* refers to the source contact, *Physical Line gate* to the gate contact, *Physical Line drain* to the drain contact. The names of these *Physical Lines* will be assigned to tiberCAD *Contacts*.

```
Physical Line("source") = {13}; // source
Physical Line("gate") = {39,38}; // gate
Physical Line("drain") = {19}; // drain
```

The final geometrical model is shown in Fig. *Geometrical model*

### 2.2.2 Meshing the device

The *.geo* script file with the geometrical description can be run in GMSH, to display the modelled device and to mesh it through the GMSH graphical interface. Alternatively, a textual mode is also available in GMSH, without graphical user interface. For example, to mesh this 2D tutorial in non-interactive mode, just type:

```
gmsh mosfet.geo -2 -o mosfet.msh
```

The final meshed model is shown in Fig. *2D meshing*
Figure 2.2: Geometrical model

Figure 2.3: Meshed model
Part II

Input
TIBERCAD INPUT FILE

Input for tiberCAD is composed by an input file e.g. input.tib and a mesh file generated by a mesher software: as for now, mesh files from GMSH (*.msh) and from Synopsys TCAD (*.grd) are supported. Be sure that the material files are in the correct directory.

To run the program, type:

    tibercad input_file_name

3.1 Description of Input file structure

A valid input file for tiberCAD is a text file with the structure described in the following. In the whole input file, everything following a “#” is considered as a comment and is ignored; blank lines can be present anywhere and are ignored, too. An input file is composed by several blocks:

A block is enclosed between curly braces (“{”, “}”) and may include one or more blocks. Each block has a header made of one or two keywords. Each block may contain zero or any number of parameter assignments of the form tagname = tagvalue, where

- tagname is a string
- tagvalue is a single numerical or string item or a list of items in parentheses and separated by commas. e.g. (cathode, anode)

Format is free for the parameter assignments, provided that they are separated by spaces. Everything which follows a “#” is considered as a comment and is ignored.

For example:

    electron_mobility field_dependent
    {
        region = buffer
        # type = doping_dependent
        low_field_model = constant
    }
Here and in the whole input file a string item can include a combination of characters, special characters and numbers, except spaces; if a space is found, the string item is assumed to be terminated.

The input file is composed by the following main classes of blocks:

```
Device, Module, Simulation
```

which will be described in the following.

### 3.2 Device section

```
Device hemt
{
    meshfile = hemt.msh

    Region buffer
    {
        ...
        Doping
        {
            ...
        }
    }

    Region barrier
    {
        ...
        ...
    }
}
```

The **Device** section includes the geometrical description of the device to be simulated. An optional device name can be associated to the device object after the `Device` keyword. In the **Device** section, two types of blocks can be present: **Region** and **Cluster** blocks. Outside of these blocks, general options common to all the device can be defined. The most important one is the specification of the mesh file, which is mandatory.

- **meshfile** [string] name of the mesh file, including file name extension (*grd* for Synopsys devise, *msh* for GMSH mesh files)
- **mesh_units** [double] units used in the meshfile in meters, default is $1e-6$ corresponding to micrometers

**For a 1D simulation the geometrical model (mesh) has to be drawn on the x axis. For a 2D simulation the geometrical model (mesh) has to be drawn in the xy-plane (z=0). Any other orientation will produce wrong results.**
The **Region** blocks contain the description of the device in continuous media approach. The **Cluster** blocks define logical groups of regions, which may have different materials or different physical properties. In this way it is possible to easily refer to sets of regions by using the cluster name.

### 3.2.1 Region block

**Region** blocks are started with the keyword **Region**, followed by the name of the tiberCAD region. The name of the tiberCAD region can coincide with the name of a mesh region, as defined during the modeling of the device. In this case, if the keyword `mesh_regions` is not used, the tiberCAD region will be associated to the mesh region identified by the given name. Otherwise, the tiberCAD region will be associated with the mesh regions specified using the keyword `mesh_regions`. For example, in the following, the tiberCAD Region **QuantumWell** will comprise the two mesh regions **well1** and **well2**, defined during the geometrical modeling of the device.

```
Region QuantumWell
{
  mesh_regions = (well1, well2)

  Doping
  {
    density = 1e17
    type = donor
    level = 0.025
  }
}
```

In the following, instead, the **Region well1** takes its name from the mesh region **well1** it refers to.

```
Region well1
{

  Doping
  {
    density = 1e17
    type = donor
    level = 0.025
  }
}
```

The available keywords inside a **Region** block are the following:

- `material` [string] name of the material associated to the region, e.g. Si or AlGaAs
- `x` [double, 0 < x < 1] alloy concentration, expressed as the molar fraction of the first component of the alloy. To create e.g. an alloy Al_{0.2}Ga_{0.8}As, we specify
AlGaAs for the keyword material, and 0.2 for x

**mesh_regions** [string, list] a list of region names as specified in the meshing program.

**[x, y, z]-growth-direction** [3-/4-tuple] Bravais vectors with Miller indices for wurtzite (4-tuple) or zincblende (3-tuple) crystal along the x, y and z directions.

A common material, crystal structure and growth directions can be defined for all device regions by defining them outside of the Region blocks.

The optional subblock Doping as in the example above contains the keywords:

- **type** [string] The dopant type. Can be donor or acceptor.
- **density** [double] The doping concentration in cm\(^{-3}\).
- **level** [double] The energy level of the dopant given as the distance from the conduction band edge (for donors) or from the valence band edge (for acceptors) in eV.
- **g** [integer] Level multiplicity. Defaults to 2 for donors and 4 for acceptors.

### 3.2.2 Cluster block

The definition of cluster blocks must be preceded by the keyword Cluster, followed by the name of the Cluster. For example

```
Cluster Quantum
{
    regions = (well, barrier1, barrier2)
}
```

groups the regions well, barrier1 and barrier2 in a logical unit with name Quantum. The regions option is mandatory and contains the list of mesh or tiberCAD regions to be grouped together.

Regions and clusters represent the macroscopical description of the device or structure to be be simulated in tiberCAD. In the rest of the input file, the regions associated to the Modules will be indicated by means of tiberCAD region names and cluster names.

### 3.2.3 Atomistic structure

The definition of an atomistic structure can be done using the keyword Atomistic.
Atomistic qw
{
  regions = (all)
  reference_region = barrier1
}

The atomistic generator produces an atomistic model over all the specified regions. Currently the
structure can only be a pseudomorphic structure. The reference region is used to define the crystal
lattice for the whole structure. More details about the atomistic generator can be found in the
module Empirical Tight Binding (ETB).

### 3.3 Modules

Module driftdiffusion
{
  name = dd
  #regions = all

  plot = (Ec, Ev, Eg, eQFermi, hQFermi,
         eDensity, hDensity, eMobility, hMobility,
         Polarization, ElPotential, ElField,
         eCurrentDensity, hCurrentDensity,
         CurrentDensity, ContactCurrents,
         eConductivity, hConductivity)

  Solver
  {
    max_iterations = 15
    relative_tolerance = 1e-12
    step_tolerance
  }

  Physics
  {
    strain_simulation = strain

    polarization (piezo, pyro) {}

    mobility
    {
      type = field_dependent
      low_field_model = doping_dependent
    }
  }
}
Contact cathode
{  
  voltage = $Vd
}

Contact anode
{  
  voltage = 0.0
}

One or more module-blocks may be present: each module-block must be preceded by the keyword Module, followed by the (single-word) module name. This must be the name of one of the tiberCAD modules. Here are the Modules implemented until now:

- **driftdiffusion**: Poisson-driftdiffusion transport of electrons and holes
- **thermal**: Heat balance simulation
- **elasticity**: Calculation of elastic deformations in heterostructures
- **efaschroedinger**: Envelop Function Approximation (EFA) solution of single particle Schroedinger equation for electrons and holes
- **opticskp**: Optical properties (spontaneous emission spectrum)
- **DSC**: Simulation of a DSC solar cell
- **vff**: Relaxation of an atomistic structure with valence force field
- **empirical_tb**: Atomistic quantum calculations with an Empirical Tight-binding solver
- **sweep**: Parameterized execution of a module simulation (e.g. for the calculation of output current characteristics)
- **selfconsistent**: coupled calculations of different simulation modules

Each module-block usually contains a list of general options, such as `plot` and others specific to each module. Then, two main blocks define the `Physics` and the `Solver` models and parameters for this module.

- **Solver** contains the solver parameters; depending on the Module, it can contain a LinearSolver definition subblock.

- **Physics** usually contains the definition of the `physical models` used in the simulation.

Examples of `physical models` are, for the **driftdiffusion** module, **recombination**, **electron mobility**, **trap**, **polarization** and so on. A particular model is the **Boundary** model, which has an alias `Contact` for driftdiffusion module, as we will see in the following. The declaration of these models obey to the following syntax:
model_keyword type_specifier

where model_keyword is the name of the physical model to be declared (e.g. the trap model),
type_specifier is the name of a particular one among the available descriptions for that model and
the following block contains the options for the model. For example, for a trap model of type acceptor

trap acceptor
{
  regions = buffer
  Nt = 7e16
  Et = 0.5
  reference = cb
}

An alternative multiple declaration is possible if no options, other than default, are declared:

model_keyword (type specifier1, type specifier2,...)

For example:

recombination (srh, direct, auger) { }

In this example, several recombination models are defined (srh, auger, direct) each one with default parameters.

A very useful feature is the possibility to restrict the application of a given physical model to one
or more regions of the simulated device. This can be obtained with the keyword regions. For example:

regions = (buffer,substrate)

where buffer and substrate are valid regions of the device. In the following example

valence_band
{
  regions = QWell
  density_of_states bulk
  {
    dos_mass = 0.80
  }
}
the bulk DOS model of band properties for the valence band is restricted to the region named QWell:

```
regions = QWell
```

It is also possible to use the keyword regions to exclude one or more regions from the defined physical model, simply using a minus sign before the name of the region(s):

```
band_properties
{
    regions = -QWell
    density_of_states bulk_kp
    {
        strain_simulation = strain
    }
}
```

Here the region named QWell is excluded from bulk kp DOS model:

```
regions = -QWell
```

Thus this model will be applied to all the device, except the region QWell.

For a detailed description of the models, please refer to the reference guide.

Two special modules are the sweep and the selfconsistent modules, which will be described in the following sections.

### 3.3.1 Module sweep

```
Module sweep
{
    name = sweep_drain
    solve = driftdiffusion
    variable = $Vd
    start = 0.0
    stop = 2.0
    steps = 20
    plot_data = true
}
```

Each sweep Module defines a set of calculations applied to a boundary region (e.g. a set of bias values to be assigned to a drain contact of a MOSFET for the calculation of an output drain IV.
characteristic), in this Guide referred to as sweep calculation.

The following keywords are defined for this feature:

- **variable** name of the variable to which the sweep is applied:

  Writing e.g. `variable = $Vg` means that a sweep on the variable $Vg$ will be performed. (see *Variables*).

Further options controlling a sweep are:

- **name** A user defined name. This is important if several sweeps are defined.
- **solve** names of the simulations (modules) to be solved at each sweep step. The names of other sweeps may be provided to realize nested sweeps.
- **start, stop, steps** The sweep goes from start to stop in steps steps.
- **min_step** The minimum allowed step size.
- **max_step** The maximum allowed step size.
- **plot_data** If set to true, output data will be written after each step of the sweep, otherwise this will be done only for the final sweep value. The default is false to limit output data. The amount of written data can be controlled using the **max_step** option.

Once a sweep calculation has been defined, it is treated as an ordinary simulation and may therefore be used just like any other simulation by adding it to a solve statement (see below).

### 3.3.2 Module selfconsistent

In this Module it is possible to define a self-consistent calculation based on two different simulation modules (e.g. drift diffusion and thermal).

```plaintext
Module selfconsistent
{
    name = sc_all
    solve = (tb, dens_el, dens_hl, dd)
    max_iterations = 5
    absolute_tolerance = 1e-3
    relative_tolerance = 1e-6
}
```

In solve the list of simulations to be performed self-consistently is specified.

See also *Selfconsistent*. 

---

3.3. Modules 23
3.4 Simulation section

Simulation
{}
    temperature = 300
    verbose = 3
    solve = (strain, dd, sweep_g)
    logfile = output_Nt5e16_A10.215_SiN/hemt.log

    resultpath = output_Nt5e16_A10.215_SiN
    output_format = vtk

In this block one can specify several general parameters such as the temperature and the settings for the actual calculation to be run, that is the process-flow of the simulation.

    searchpath: path for material files, default is the system-defined material directory.

    temperature: temperature of the system [K]

    solve: list of simulations to be executed, in the order of execution
        solve = (strain, driftdiffusion, quantum_electrons, quantum_holes)

    resultpath: path for output directory

    output_format: format of the output data: grace for ascii data column type vtk for VTK based output.

3.5 Output description

At the end of the execution, the program will write the results of the simulation in the directory specified by resultpath, with the format specified by output_format. The output variables for each Module are specified in the list plot.

tiberCAD output is divided in two classes: mesh-based and mesh-independent quantities.

The available formats are

- vtk for 2D and 3D output data
- grace for 1D output data

Output results from 2D or 3D simulations may be visualized with the open source visualization and post-processing tool paraview: http://www.paraview.org
Output results from 1D simulations are ascii data files which can be visualized e.g. with the open source plotting tools:

**xmgrace**: [http://plasma-gate.weizmann.ac.il/Grace](http://plasma-gate.weizmann.ac.il/Grace) (for Linux)


### 3.5.1 Mesh-based quantities

The mesh-based quantities are all the quantities associated with the nodes of the mesh, such as Fermi level, electron and hole density, conduction and valence band, etc., together with all the quantities associated with the elements of the mesh, such as current density.

The output values for these quantities are reported in the files `simname_msh.ext`, where `simname` is the Module used for the calculations and `ext` is the extension of the chosen file format, e.g. `vtu` for paraview output.

```
strain_msh.vtu
```

In the case a sweep calculation is performed and the `plot_data` keyword is set to `true`, the output files are of the kind `simname_sweepvariable_stepvalue_msh.ext`, where `sweepvariable` is the variable with respect to which the sweep is performed (e.g. gate voltage) and `stepvalue` is the value of this variable at that step; e.g the result at the step $V_b = 1.1$ will be found in the file:

```
dd_Vb_1.1_msh.vtu
```

### 3.5.2 Mesh-independent quantities

The mesh-independent quantities are the quantities which are not associated to the mesh, for example current at the contacts of a diode or quantized energy levels in a quantum well.

These mesh-independent quantities are displayed in separated files, with the format `simname.ext`, e.g `quantum_electrons.dat`, where `simname` is the name of the Module associated to the results and `ext` is the filename extension (usually `dat`). If a sweep is performed, the output file gets the format `sweep_name_simname.ext`, where `sweep_name` is the name of the sweep performed, for example

```
sweep_drain_driftdiffusion.dat
```

The mesh independent results for every sweep step are stored in this file.

### 3.6 Material database
### Material list

<table>
<thead>
<tr>
<th>Name</th>
<th>Crystal structure</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AlAs</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>AlAsP</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlAsSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlGaAs</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlGaN</td>
<td>wz</td>
<td>ternary</td>
</tr>
<tr>
<td>AlGaP</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlGaSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlInAs</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlInN</td>
<td>wz</td>
<td>ternary</td>
</tr>
<tr>
<td>AlInP</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlInSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlN</td>
<td>wz</td>
<td>binary</td>
</tr>
<tr>
<td>AlP</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>AlPSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>AlSb</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>GaAs</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>GaAsP</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>GaAsSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>GaInP</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>GaInSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Crystal structure</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GaN</td>
<td>wz</td>
<td>binary</td>
</tr>
<tr>
<td>GaP</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>GaPSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>GaSb</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>Ge</td>
<td>zb</td>
<td>simple</td>
</tr>
<tr>
<td>InAs</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>InAsP</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>InAsSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>InGaAs</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>InGaN</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>InN</td>
<td>wz</td>
<td>binary</td>
</tr>
<tr>
<td>InP</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>InPSb</td>
<td>zb</td>
<td>ternary</td>
</tr>
<tr>
<td>InSb</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>Pentacene</td>
<td>-</td>
<td>molecular</td>
</tr>
<tr>
<td>Si</td>
<td>zb</td>
<td>simple</td>
</tr>
<tr>
<td>SiN</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>SiO2</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>TiO2mes</td>
<td>-</td>
<td>mesoporous</td>
</tr>
<tr>
<td>ZnO</td>
<td>zb</td>
<td>binary</td>
</tr>
<tr>
<td>ZrO2</td>
<td>zb</td>
<td>binary</td>
</tr>
</tbody>
</table>

Table 3.1: List of Materials

The parameters of the most important semiconductor materials are collected in the material database. See List of materials for a complete list of the materials defined in tiberCAD. *zb* stands for zincblende crystal structure, *wz* for wurtzite. Materials can be simple elements, like Si and Ge, binary compounds, such as GaAs or GaN, and ternary alloys, like AlGaAs. For the simple and binary compounds the lattice, strain and band properties are included in each material file. The ternary alloy material file instead contains the name of the two parent materials, from which the parameters of the alloy material are calculated, depending on the components concentration, according to a quadratic law, e.g. for the energy gap $E_g$

$$E_g(A_xB_{1-x}) = xE_g(A) + (1-x)E_g(B) - x(1-x)C$$

where the *bowing* parameter $C$ accounts for the deviation from a linear interpolation (virtual-crystal approximation) between the two binary compounds $A$ and $B$ and $x$ is the concentration of the binary $A$. Values of $C$ for one or more parameters are reported in the alloy material file. User-defined materials can be freely added, provided that the correct syntax for the parameters entry is followed.
3.7 Input files Examples

3.7.1 1D Bulk Silicon

This is the Example 0 in the Example directory. This simple example calculates the IV characteristic of a piece of slightly doped bulk silicon. The mesh file bulk.msh has been generated by GMSH, based on the script bulk.geo, described in *GMSH Example 1D*.

```plaintext
# Description of the device physical regions
Device
{
   meshfile = bulk.msh

   Region bulk
   {
      material = Si

      Doping
      {
         density = 1e16
         type = donor
      }
   }
}

Here the geometrical and physical structure of the device to be simulated is described. Note that the name of the only Region of this Device is bulk, that is the name of the Physical Line in *GMSH Example 1D*.

# Definition of Simulation Models

Module driftdiffusion
{

   # name = driftdiffusion # this is the default name

   #regions = all # 'all' is the default

   # what we want to plot
   plot = (Ec, Ev, eQFermi, hQFermi, ContactCurrent)

   Contact anode { voltage = $Vb }
   Contact cathode { }
}
```
Here the block for Module `driftdiffusion` is defined.

Note that the names of the two `Contact` boundary regions, `anode` and `cathode`, are the names of the two `Physical Point` objects created in `GMSH Example 1D`.

```plaintext
# we want to sweep over the anode voltage
Module sweep
{
    solve = driftdiffusion
    variable = $Vb
    start = 0.0
    stop = 1
    steps = 10
    # for each step we want to plot the solution variables
    # specified in the driftdiffusion module
    plot_data = true
}
```

In this example, we calculate Poisson and transport equations for a set of bias values, as described by `Sweep` block.

```plaintext
Simulation
{
    # this increases the amount of information
    # written to the screen
    verbose = 2

    solve = sweep

    resultpath = output
    output_format = grace
}
```

Finally, the `Simulation` block defines the simulations to be executed and their order. In this case this amounts to just the `sweep` block calculation.

### 3.7.2 2D Mosfet

This is the Example 4 in the Example directory (see also `Example 2: Mosfet`) This example calculates the IV characteristics of a silicon Mosfet. The mesh file `mosfet.msh` has been generated by GMSH, based on the script `mosfet.geo`, described in `GMSH Example 2D`.

```plaintext
Device mosfet
{
    meshfile = mosfet.msh
}
```
material = Si

Region substrate
{
  Doping
  {
    density = 1e18
    type = acceptor
  }
}

Region contact
{
  Doping
  {
    density = 5e19
    type = donor
  }
}

Region oxide
{
  material = SiO2
}

Note that the Regions substrate, contact, oxide correspond to the Physical surfaces defined in the geo script of GMSH (see GMSH Example 2D).

Module driftdiffusion
{

  # we solve for electrons only
  #coupling = electrons

  plot = (Ec, Ev, eQFermi, eDensity, eCurrentDensity, eMobility, hQFermi, hDensity, hCurrentDensity, hMobility, NetRecombination, ElField, ElPotential, ContactCurrents)

  Solver
  {
    type = linesearch

    linear_solver
    {
      method = pconly
    }

3.7. Input files Examples
preconditioner = lu
}
}

Physics
{
  recombination srh {}
}

mobility
{
  type = field_dependent
  low_field_model = doping_dependent
}
}

Contact gate
{
  type = schottky
  barrier_height = 3.0
}

voltage = $V_g$

# assume a gate width of 1 mm = 0.1 cm
area_factor = 0.1
}

Contact source
{
  voltage = 0.0
  area_factor = 0.1
}

Contact backcontact
{
  voltage = 0.0
  area_factor = 0.1
}

Contact drain
{
  voltage = $V_d$
  area_factor = 0.1
}
Note that the names of the Contact boundary regions, are the names of the Physical Line objects created in *GMSH Example 2D*.

### 3.7.3 Example of Input file: 2D HEMT

Here is an example of the input file template:

```
Device hemt1
{
   meshfile = hemt.msh
   mesh_units = 1e-6

   material = GaN
   y-growth-direction = (0,0,0,1)
   z-growth-direction = (1,0,-1,0)
   x-growth-direction = (-1,2,-1,0)

   Region barrier
   {
      material = AlGaN
      x = 0.14
   }
   Region barrier_doped
   {
      mesh_regions = (barrier_dop_s, barrier_dop_d)
      material = AlGaN
      x = 0.14
      Doping
      {
         Nd = 1e21
         type = donor
         doping_level = 0.026
      }
   }

   Region buffer {}

   Region buffer_doped
   {
      mesh_regions = (buffer_dop_s, buffer_dop_d)
      Doping
      {
         Nd = 1e21
         type = donor
         doping_level = 0.026
      }
   }
```

---

3.7. Input files Examples
Region cap
{
  Doping
  {
    Nd = 5e18
    type = donor
    doping_level = 0.026
  }
}
Region cap_doped
{
  mesh_regions = (cap_dop_s, cap_dop_d)
  Doping
  {
    Nd = 1e21
    type = donor
    doping_level = 0.026
  }
}
Region passivation
{
  mesh_regions = (passiv1, passiv2, passiv3, passiv4)
  material = SiN
}
Cluster semiconductor
{
  regions = -passivation
}
Module driftdiffusion
{
  name = dd
  regions = all
  coupling = electrons
  save_state = true
  #load_state = output_Nt7e16_out/dd_Vd_50.tsv

  plot = (Ec, Ev, Eg, eQFermi, hQFermi, eDensity, hDensity, Polarization, eCurrentDensity, hCurrentDensity, CurrentDensity, ContactCurrents, ElPotential, ElField, eMobility, hMobility, eConductivity, hConductivity)

NonlinearSolver linesearch
{
  # type = ls
  relative_tolerance = 1e-15
  step_tolerance = 5e-3
max_iterations = 15
LinearSolver petsc
{
  ksp_type = pconly
  preconditioner = lu
}

Physics
{

  band_properties
  {
    density_of_states bulk_kp
    {
      strain_simulation = strain
    }
  }

  recombination (srh, direct, auger) {}

  electron_mobility field_dependent
  {
    region = buffer
    low_field_model = constant
  }

  electron_mobility doping_dependent
  {
    region = (barrier, barrier_doped, cap, cap_doped, buffer_doped)
  }

  trap fixed_charge
  {
    region = GaN/SiN
    Nt = 2.74e13
  }

  trap acceptor
  {
    region = buffer
    Nt = 7e16
    Et = 0.5
    reference = cb
  }

  polarization (piezo) {strain_simulation = strain}
  polarization (pyro) {}

}

3.7. Input files Examples
Contact drain
{
  voltage = @Vd
}

Contact source
{
  voltage = 0.0
}

Contact gate
{
  regions = (gate1, gate2)
  type = schottky
  work_function = 1.5272
  voltage = @Vg
}

Module elasticity
{
  name = strain
  regions = -passivation # all the device except Region "passivation"

  plot = (Strain, StrainCell, Stress, Displacement )
  lin_rel_tol = 1e-6

  Physics
  {
    body_force lattice_mismatch
    {
      reference_material = GaN

      structure = wz
      y-growth-direction = (0,0,0,1)
      z-growth-direction = (1,0,-1,0)
      x-growth-direction = (-1,2,-1,0)
    }
  }
  
  Contact substrate
  {type = clamp}
}
Module sweep
{
    name = sweep_g
    variable = Vg
    solve = sweep_d
    start = -1
    stop = 1.0
    steps = 2
    max_step = 0.25
    min_step = 1e-4
    #plot_data = true
}

Module sweep
{
    name = sweep_d
    variable = Vd
    solve = driftdiffusion
    start = 0.0
    stop = 50.0
    steps = 200
    min_step = 1e-4
    plot_data = true
}

Module selfconsistent
{
    name = relaxation
    solve = (macrostrain, driftdiffusion)
    absolute_tolerance = 1e-3
    relative_tolerance = 1e-5
}

Simulation
{
    temperature = 300
    verbose = 3
    solve = (strain, dd, sweep_g)
    logfile = output_Nt5e16_A10.215_SiN/hemt.log
    # these parameters can be defined in modules, too
    resultpath = output_Nt5e16_A10.215_SiN
    output_format = vtk
}
The Atomistic generator is a tool of tibercad which allows to generate an atomic basis associated with the finite element mesh which belongs to a given physical region, based on the material specifications and the growth directions defined for that region.

For example, the atomistic structure `tb` is created by the Atomistic generator through the following block of instructions contained in the Device section:

```plaintext
Atomistic tb
{
    reference_region = barrier_left
    regions = Quantum
    passivation = yes
    print = (xyz, gen, tgn)
    random_alloy = true

    radial_distribution
    {
        species = (In,Ga)
        cutoff_radius = 1.0
        resolution = 0.001
    }

    alloy_statistics
    {
        control_volume_radius = 1.0
        plot_alloy_composition = true
    }
}
```

The internal atomistic generator has the following features:

- It supports any crystal structure with fcc, bcc, cubic and hexagonal Bravais lattice. Information about basis and primitive vectors is taken from the material database.
- It performs hydrogen passivation for any crystal structure supported.
For the generation of heterostructures a pseudomorphic growth is assumed, with the same lattice constants of the defined reference region.

The definition of a reference region is needed. It provides the specifications on the material and parameters (growth direction, molar fraction, crystal phase) which will be used to build the crystal structure.

In this way, only the atomic species are changed according to the different materials specifications, while the lattice is the reference material lattice.

The structure may be eventually relaxed by projecting strain calculated with a continuum elasticity model, or by minimizing the energy of the system applying a valence force field (VFF) model.

### 4.1 General Options

The following keywords may be defined in Atomistic block:

- **regions** [string] defines the physical Regions or Cluster where to build the atomistic structure.
- **reference_region** [string] defines the name of the region whose material and growth directions are taken as a reference for lattice construction.
- **reference_material** [string] defines a general reference material and growth directions that are taken as a reference for lattice construction (may be useful if one chooses a reference material not present in any device region).
- **translation** [real vector] defines a translation vector as \((dx, dy, dz)\) used to move the atomistic structure w.r.t. the mesh regions. It is useful for fine alignments of atoms to the mesh.
- **passivation** [boolean] if true an hydrogen passivation is applied to the structure. Hydrogen atoms are inserted at a defined distance in the same direction of the original crystal bond.
- **hydrogen_distance** [real] when passivation = true, defines the bond length at which Hydrogen atoms are placed from the surface atoms (default 1.2 Angstrom).
- **periodicity** [list of boolean] add periodicity along one or more of the three directions \(x,y,z\), depending on the mesh dimension: for a 1D mesh, add periodicity in \(x\) direction (periodicity = true); for a 2D mesh, add periodicity along \(x\) and/or \(y\) directions (e.g. periodicity = (true,true)); for a 3D mesh, add periodicity along \(x,y\) and \(z\) (e.g. periodicity = (true,true,true)). More examples in the following.
- **supercell_size_y** [real] defines a lateral supercell size on \(y\)-axis (in Angstrom) in 1D (quantum wells) and 2D (nanowires) structures.
- **supercell_size_z** [real] defines a lateral supercell size on \(z\)-axis (in Angstrom) in 1D and 2D structures.
**load_structure** [string] specifies a (relative) path/file for the loading of an external structure. Valid formats are tgn, xyz, gen. The external structure is trimmed outside the regions on which the atomistic structure is defined in the input file. Note: when loading xyz files, the supercell vectors must be defined through the keyword lattice_vectors (see below)

**lattice_vectors** [list of floating] specifies the periodicity vectors used when importing an external atomic structure in xyz format (see load_structure). The format is (x1,y1,z1,x2,y2,z2, x3,y3,z3), values are in Å, e.g.( 52.728 , 0.0, 0.0 , 0.0, 45.664 , 0.0, 0,0, 42.758 ).

**random_alloy** [boolean] if true, a random alloy structure is generated (default = false). More examples in the following.

**random_generator_seed** [integer] an integer defining the random seed. Useful to produce exactly the same random alloy structure in a set of executions.

**fix_mean_alloy_concentration** [boolean] if true the number of substituted atoms in a random alloy is as close as possible to the nominal molar fraction of the alloy. If false each atom is substituted with a probability proportional to the defined nominal molar fraction. The resulting actual number of substituted atoms may differ from the nominal concentration value.

**clustering** [specie name] Alloy structures with clustering on the given atomic specie are produced, e.g. clustering = In. This is done by placing a fraction of seed atoms (cluster_seed) with uniform distribution, then the remaining fraction of atoms is placed biased with a larger probability near existing ions.

**cluster_seeds** [real] defines the fraction of ions placed with uniform distribution. Must be > 0. (default = 0.02).

**reference_atom** [integer] sets the atom label (typically 1=cations, 2=anions) where local concentration is computed. Default is cations.

**meshdata_format** [string] defines the output format of the alloy statistics (default = “vtk”) plotted when plot_alloy_composition = true (see next paragraph).

**print** [string] print the generated atomic structure with one of the following formats: xyz: xyz file format; gen: gen file format; tgn: tibercad format which combines informations on the mesh and the atoms.

### 4.2 Optional Blocks

Two optional blocks may be present in Atomistic block:
4.2.1 Block alloy_statistics

This block performs the calculation of the local concentration by averaging over small local spheres whose radius is defined by the keyword control_volume_radius.

Example:

alloy_statistics
{
  control_volume_radius = 1.0
  plot_alloy_composition = true
}

The following keywords may be defined in alloy_statistics block:

- control_volume_radius: real sets the radius of the control volume (nm), that is of a sphere on which local concentration is computed (default = 0.5 nm).
- plot_alloy_composition [boolean] if true the alloy composition is mapped on the mesh and plotted in a vtk file format.

4.2.2 Block radial_distribution

If this block is declared, the radial distribution function, g(r), is computed. The distribution represents the probability density of finding an atom at a given distance from a reference atom. It is computed by counting all atoms that are found at a distance (r, r+dr) and normalized to the total volume and total number of atoms.

Example:

radial_distribution
{
  species = (In, Ga)
  cutoff_radius = 1.0
  resolution = 0.001
}

The following keywords may be defined in radial_distribution block:

- cutoff_radius [real] is used to set the maximum distance of the distribution.
- resolution [real] is used to specify the binning distance step (dr).
- species [specie names] For each specie a radial distribution function is computed. The distribution is also resolved into the other species present in the atomistic structure. Files are produced in output with names given by <structure_name_Specie_radial_suffix>.dat, with column structure
  r specie_1 specie_2 ...
In practice each column represents the probability density to find a specie $\text{specie}_i$ at a distance $r$ from the reference specie. The radial distribution can be computed also after VFF optimization, by adding `radial_distribution` to the `plot` instruction.

### 4.3 Importing an external file

An external atomic structure in a valid file format may be loaded through the keyword `load_structure`. Valid file formats are `xyz`, `gen` and `tgn`. While `xyz` and `gen` are commonly used atomic descriptions, `tgn` format is an internal format used in tibercad which contains information on the physical regions and mesh elements associated to the atoms.

When loading an external structure file in `xyz` format, the supercell vectors of the structure must be defined through the keyword `lattice_vectors`. When loading a `gen` file this information is already included. After loading, the external atomic structure is trimmed outside the physical regions defined in Atomistic block. Then atoms are associated to the current mesh.

**Note**: It is assumed that the atomic positions of the structure are consistent with the mesh.

### 4.4 Applying periodicity

Periodicity may be applied to the generated atomistic structure in 1, 2 and 3D. Default treatment of periodicity is different for each mesh dimension, to take into account most common cases.

For a 1D mesh, a periodicity is always applied in $y$ and $z$ directions. As a default, a minimal cell is built in $yz$ plane ($x$ being the predefined growth direction). This refers to the case of a quantum well structure. The periodicity vectors are by default those of the minimal cell for the given material. It is possible to build a supercell in the $yz$ plane, by defining the supercell size through the `supercell_size_y` and `supercell_size_z` keywords. In this case, appropriate periodicity vectors will be obtained and applied. It is possible to add a periodicity along the $x$ (growth) direction with the keyword `periodicity = true`, to design a superlattice structure.

For a 2D mesh, the generated structure is periodical along $z$ axis. It is possible to add a periodicity in $x$ and $y$ directions with the keyword `periodicity = (<x-per>,<y-per>)`, where `<x-per>` and/or `<y-per>` may be `true` or `false`.

For a 3D mesh, the generated structure is not periodical along any axis (`cluster`). It is however possible to add a periodicity in any directions with the keyword `periodicity = (<x-per>,<y-per>,<z-per>)`, where `<x-per>` and/or `<y-per>` and/or `<z-per>` may be `true` or `false`.

It must be noted that here we refer to the dimension of the geometrical and mesh models. In 1 and 2D, however, the atomic structures are in any case built with at least one layer along the directions which are not present in the geometrical model, according to the material lattice parameters.
4.5 Use of random alloy generator

By default, the atomic structure of an alloy material is generated based on Virtual Crystal Approximation (VCA). The VCA considers an alloy $A_xB_{1-x}C$ as a fictitious material whose properties are a weighted average of the properties of the AC and BC components. In continuous media models, nonlinearities in the alloy parameters such as bandgap, effective masses or piezoelectric tensors are usually modelled with suitable bowing parameters, derived experimentally or from bulk DFT calculations. In ETB, tight-binding parameters are averaged between those of the alloy components, weighting with the nominal alloy concentration.

An alternative to VCA is the random alloy approach, where the atoms of the components species are randomly distributed in the alloy region and their parameters are not averaged. Using atomistic approaches it is possible to model in detail realistic alloy distributions within the nanostructures including the effect of local strain, thus leading to more accurate electronic calculations and allowing to study alloy fluctuations in a region of a device.

To this aim, it is necessary to generate several random samples of the alloy structure, in order to perform a statistical analysis of the fluctuations of alloy properties. In TiberCAD, a random alloy structure is generated if $random\_alloy = true$ in Atomistic block. In this case, the alloy is created by random substitution of one of the components of the host material with the substitution ion. For example, InGaN alloy is created by substituting Ga atoms with In atoms in a GaN lattice. The modality of this substitution is ruled by $fix\_mean\_alloy\_concentration$: if this is true, then a fixed number of atoms (obtained from molar concentration) is substituted, in randomly chosen positions of the lattice. If $fix\_mean\_alloy\_concentration$ is false, then each Ga atom in the lattice is substituted with a probability proportional to the defined nominal molar fraction. This implies that the actual number of substitutions, and thus the actual mean concentration of the present random sample, differs from the nominal value. In this way one can take into account fluctuations of the global alloy concentration.

Random generation is ruled by a random seed, which is in general different for each execution. In case one needs to repeat calculations with exactly the same random configuration, it is possible to keep it unmodified by defining the same $random\_generator\_seed$ in all the executions.

By default, the random alloy structure is built with an uniformly random approach. An option of random alloy generation allows to generate clustering in random configuration. This is made by defining the atomic specie subjected to clustering with $clustering = specie\_name$, e.g. $clustering = In$ (by default no clustering is applied). In this way, the alloy structure is not built entirely with an uniformly random approach. Instead, the formation of clusters of substitution atoms (e.g. In atoms in InGaN alloy) is favoured. The keyword $clustering\_seeds$ define the fraction (default is 0.02) of substitution ions which are placed with an uniform distribution. The remaining fraction of ions is placed with a higher probability in positions close to other substitution ions.
4.6 Example

Here is an example of generation of an atomistic structure

Region well1
{
    material = InGaN
    x = 0.27
}

Region qbarrier1
{
    material = GaN
}

Region qbarriert
{
    material = GaN
}

Atomistic tb
{
    reference_region = qbarrier1
    regions = (well1, qbarrier1, qbarriert)
    passivation = yes

    print = (xyz)

    # clustering = In

    #load = output_x0.20_1200/tb.xyz

    random_alloy = true
    random_generator_seed = 5
    plot_alloy_composition = true
    control_volume_radius = 0.5
    extract_alloy_statistics = true

    #supercell_size_z = 20
    #supercell_size_y = 60
An atomistic structure named *tb* is created based on the physical regions *well1, qbarrierl, qbarriert*. The reference region is *qbarrierl*, which means that the reference lattice will be that of GaN. Passivation of surface states is *true*. Since the mesh dimension is 1D, periodicity is imposed along y and z axis.

*random_alloy = true* means that the InGaN alloy in the well1 region will be generated with random substitution. The random distribution will be uniform, since *clustering = In* is commented. We define a *random_generator_seed = 5*, which can be used in a following calculation to obtain exactly the same atomic structure. *extract_alloy_statistics = true* will write info on local concentration, obtained averaging on local spheres defined by *control_volume_radius = 0.5*.

*plot_alloy_composition = true* will plot the local concentration on the mesh for a nice visualization.
Part III

Modules
CHAPTER

FIVE

ELASTICITY

5.1 Theory

Elasticity is a Finite Element solver for mechanical equilibrium problems. It brings features typically developed for force balance problems into device modeling. The coupled treatment of the electro-mechanical problem within a unique framework results very useful to explore for multidisciplinary ideas. Details about the theory of continuous elasticity applied to device mechanical deformation can be found in [Povolotskyi].

TiberCAD computes the mechanical deformation of a body subjected to external forces by means of the equilibrium mechanical equation, i.e.

\[
\frac{\partial \sigma_{ij}}{\partial x_j} = f_i
\]  

(5.1)

where \( \sigma \) is the stress second-rank tensor and \( f \) is the external body force applied to the system. As we will see below, any strain and stress source can be appropriately mapped into an equivalent body force.

The stress tensor \( \sigma \) is related to the mechanical strain by means of the constitutive relationships \( \sigma_{ij} = C_{ijkl} \epsilon_{lk} \) where \( C \) is the stiffness fourth-rank tensor. The strain is related to the displacement \( u \) by the expression

\[
\epsilon_{kl} = \frac{1}{2} \left( \frac{\partial u_l}{\partial x_k} - \frac{\partial u_k}{\partial x_l} \right)
\]  

(5.2)

Because of the symmetry of the strain and stiffness tensor the final equation reads as

\[
\frac{\partial}{\partial x_j} C_{ijkl} \frac{\partial u_l}{\partial x_k} = f_i
\]  

(5.3)

the computed displacements are used to deform the mesh.
5.1.1 Stiffness constant

By default the stiffness constant is anisotropic. For the zincobland crystal structure the independent coefficients are (with the Voigt notation) $C_{11}$, $C_{12}$, $C_{44}$.

$$
C = \begin{pmatrix}
C_{11} & C_{12} & C_{12} & 0 & 0 & 0 \\
C_{12} & C_{11} & C_{12} & 0 & 0 & 0 \\
C_{12} & C_{12} & C_{11} & 0 & 0 & 0 \\
0 & 0 & 0 & C_{44} & 0 & 0 \\
0 & 0 & 0 & 0 & C_{44} & 0 \\
0 & 0 & 0 & 0 & 0 & c_{44}
\end{pmatrix}
$$

(5.4)

For the wurtzite structure we have $C_{11}$, $C_{12}$, $C_{13}$, $C_{44}$.

$$
C = \begin{pmatrix}
C_{11} & C_{12} & C_{13} & 0 & 0 & 0 \\
C_{12} & C_{11} & C_{12} & 0 & 0 & 0 \\
C_{13} & C_{12} & C_{11} & 0 & 0 & 0 \\
0 & 0 & 0 & C_{44} & 0 & 0 \\
0 & 0 & 0 & 0 & C_{44} & 0 \\
0 & 0 & 0 & 0 & 0 & C_{44}
\end{pmatrix}
$$

(5.5)

While the anisotropic model is included by default, the isotropic one must be explicitly indicated.

An isotropic stiffness can be defined within the block Stiffness with the keyword isotropic. In this case, the only independent parameters are the Young modulus $E$ (young in GPa) and the Poisson’s ratio $\nu$ (poisson).

$$
C = \frac{E}{(1+\nu)(1-2\nu)} \begin{pmatrix}
1-\nu & \nu & \nu & 0 & 0 & 0 \\
\nu & 1-\nu & \nu & 0 & 0 & 0 \\
\nu & \nu & 1-\nu & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2}
\end{pmatrix}
$$

(5.6)

Example:

Stiffness isotropic
{
  young = 129
  poisson = 0.349
}

5.2 Module options

The following options influence the behaviour of the Elasticity module:

- **mesh_deformation** [bool] if `true` the strain is computed iteratively until the convergence on the structure deformation is reached (default = `false`)

- **shape_error** [double] defines the maximum tolerated error in non-linear strain computation (default = `1e-3`)

- **shape_iterations** [integer] defines the number of shape iterations for non-linear strain computation. The default is `20`.

- **strain_atomistic_structure** [string] defines the name of the atomistic structure to which deformation will be applied according to the strain calculated with `elasticity`

To apply a strain deformation to a defined atomistic structure, it is more convenient to define a `mesh_deformation` with the default number of `shape_iterations`. In this way, strain will be computed iteratively until the convergence on the structure deformation is reached. It is also important to define the name of the atomistic structure to be deformed, with `strain_atomistic_structure`. For example

```plaintext
mesh_deformation = true
strain_atomistic_structure = dot_atoms
```

5.3 Solution/Plot variables

The solution variables available for plotting and for interaction with other modules are given in *Plotting variables*.

5.4 Solver section

The **Solver** section of the **Elasticity** module refers to a linear solver. See section *Linear solvers* for details on linear solver options.

5.5 Physics section

In the following we describe all the submodels. As mentioned in the Introduction in section *Input-File Getting*, submodels can be restricted to a subset of simulation regions.

The user can specify the following physical models:
5.5.1 Body force

The Body force model implements an external force applied to the device. 

The keyword is body_force, i.e

```
body_force [type]
{
}
```

The type can be constant, converse_piezo, lattice_mismatch, thermal_stress.

Constant

A constant value body force can be included by means of the keyword constant.

Example:

```
body_force constant
{
   F = 1
}
```

F is a force in N/m³

Lattice mismatch

When two crystals with different crystal structure are put in contact the lattice mismatch between them may induce a strain $\epsilon^{LM}$ and, therefore, a stress $\sigma = C\epsilon^{LM}$.

This contribution can be mapped in the body force

$$f_i = -\frac{\partial}{\partial x_j}C_{ijkl}\epsilon^{LM}_{lk} \quad (5.7)$$

and can be included with the keyword lattice_mismatch.

The strain source can be computed only once the reference lattice is identified. The reference material and its growth axis must be included following the same syntax of the device section (see InputFileGetting).

Example:
body_force lattice_mismatch
{
    reference_material = AlGaN
    structure = wz
    x = 0.2
    x-growth-direction = (1,0,0,0)
    y-growth-direction = (0,1,-1,0)
    z-growth-direction = (0,0,0,1)
}

When a lattice mismatch force is included in the calculation, the displacement will be computed with respect to the reference lattice.

Converse piezoelectric effect

In presence of an electric field, an additional stress source may be present due to the so-called converse piezoelectric effect, given by:

$$\sigma^{SC}_{il} = -e_{ijk}E_k$$ (5.8)

The converse piezoelectric effect can be included with the keyword `converse_piezo` and an electrostatic simulation must be indicated.

The effective body force reads as:

$$f_i = -\frac{\partial}{\partial x_j}\sigma^{CP}_{ij}$$ (5.9)

Example:

```plaintext
body_force converse_piezo
{
    poisson_simulation = dd
}
```

where `poisson_simulation` indicates the simulation providing the electric field strength.

Thermal stress

Mechanical stress can be induced by thermal expansion in two ways:

1. different materials with different expansion coefficients touching each other
2. a temperature gradient induces different expansion in different spatial points
Thermal stress is included as a strain source of the form

$$\epsilon_{ij} = \alpha_{ij}(T - T_0)$$

(5.10)

where $\alpha_{ij}$ is the thermal expansion coefficient tensor, and $T_0$ is a reference temperature. $T$ is the local lattice temperature, which is or the simulation temperature or the temperature obtained from a different simulation module.

Thermal stress can be defined in the simulation using the `body_force` model `thermal_stress`:

```plaintext
body_force thermal_stress
{
    thermal_coefficient = (3.17e-6, 5.59e-6)
    thermal_simulation = th
    reference_temperature = 300
}
```

`thermal_coefficient` can be used to override the material database value for the thermal expansion coefficient (which may be anisotropic, but diagonal). `thermal_simulation` (optional) is the name of a thermal simulation providing the lattice temperature. `reference_temperature` is the reference temperature $T_0$. Its default value is equal to the simulation temperature.

### 5.5.2 Stiffness

The keyword is `stiffness`, i.e

```plaintext
stiffness [type]
{
}
```

The type can be `isotropic` or `anisotropic`. The latter is the default.

#### isotropic

When the stiffness constant is isotropic, the user has to provide the Young module (`young`) and Poisson ratio (`poisson`).

Example:

```plaintext
stiffness isotropic
{
    young = 129
    poisson = 0.349
}
```
anisotropic

The anisotropic constant is set by default, with the values taken from the material file; however, it is possible to override them in the input file.

Example:

\begin{verbatim}
stiffness anisotropic
  
  C11 = 55
  C22 = 123
  C44 = 103

\end{verbatim}

5.6 Boundary conditions

The following boundary conditions can be implemented for Elasticity simulations, by means of the block Contact.

• Surface force
• Clamp
• Custom

5.6.1 Surface force

Surfaces forces $f^0$ are applied by imposing $\sigma_{ij} n_j = f_i^0$ along the surface with normal $n$. This boundary condition can be used with the keyword surface_force. An example is shown below.

\begin{verbatim}
Contact base
  
  type = surface_force
  force = (0,0,0.5)

\end{verbatim}

where force is the applied force in GPa.

5.6.2 Clamp

On the other hand, one may want to fix some surface of the device. The keyword clamp freezes all nodes of a given surface.
Example:

Contact substrate
{
    type = clamp
}

5.6.3 Custom

Custom imposes a general boundary condition by means of the matrix and vector constraint $H$ and $R$ respectively, which are related to the displacement by means of $Hu = R$.

Example:

Contact substrate
{
    type = Custom
    $H = ((1,0,0),(0,1,0),(0,0,1))$
    $R = (1,0,1)$
}

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strain</td>
<td>total strain</td>
<td>-</td>
</tr>
<tr>
<td>StrainCrystal</td>
<td>total strain in the crystal system</td>
<td>-</td>
</tr>
<tr>
<td>Stress</td>
<td>total stress</td>
<td>GPa</td>
</tr>
<tr>
<td>StressCrystal</td>
<td>total stress in the crystal system</td>
<td>GPa</td>
</tr>
<tr>
<td>Displacement</td>
<td>displacement</td>
<td>m</td>
</tr>
<tr>
<td>ForceSource</td>
<td>force</td>
<td>N/m3</td>
</tr>
<tr>
<td>StrainSource</td>
<td>strain source</td>
<td>-</td>
</tr>
<tr>
<td>StressSource</td>
<td>stress source</td>
<td>GPa</td>
</tr>
<tr>
<td>EnergyDensity</td>
<td>elastic energy density</td>
<td>J/m³</td>
</tr>
</tbody>
</table>

Table 5.1: Solution/Plot variables
5.7 Example 1: Strain in heterostructures

In the following example we will compute the strain induced by the lattice mismatch in a system comprising a layer of GaN between two contacts of $\text{Al}_x\text{Ga}_{1-x}\text{N}$.

We begin the input file by defining the device regions

Device
{
  dimension = 3
  meshfile = mesh.msh
  mesh_units = 1e-9
  material = AlGaN
  x = 0.2
  x-growth-direction = (-1,0,1,0)
  y-growth-direction = (-1,2,-1,0)
  z-growth-direction = (0,0,0,1)
  Region Well
  {material = GaN}

  Region Barr1
  {
  }

  Region Barr2
  {
  }
}

All the options included in this section, such as the material, axis growth and alloy concentration, apply for all the Regions defined in the whole structure. If we want to specify different properties for a Region, we may define them in the relevant Region block. E.g.

Region Well
{material = GaN}

The second part of the input file is devoted to the declaration of Module elasticity

Module elasticity
{
  Physics
  {
    body_force lattice_mismatch
    {
      x-growth-direction = (-1,0,1,0)
      y-growth-direction = (-1,2,-1,0)
    }
  }
}
z-growth-direction = (0,0,0,1)
reference_material = AlGaN
x = 0.2

Contact Base
{type = clamp}

In Physics we declare the physical models to be applied to our device. In our case, with body_force lattice_mismatch we are adding a body force into the system, induced by the lattice mismatch of the materials defined in Device section with the reference lattice which in this case is $\text{Al}_{0.2}\text{Ga}_{0.8}\text{N}$, as defined by

reference_material = AlGaN
x = 0.2

In order to avoid a free standing device we may want to freeze a surface. With the Boundary type Clamp we fix, in this case, the region Base.

![3D map of the strain component $\varepsilon_{xx}$](image)

**Figure 5.1**: 3D map of the strain component $\varepsilon_{xx}$

Finally, the **Simulation** block reads

(}
User Manual, Release 2.5.0

Figure 5.2: Strain components along growth direction

```
solve = elasticity
```

where the default name ‘‘elasticity’‘ is used.

By default, files for a 3D system are written in vtu format which can be read from Paraview. Strain output data are shown below.

In Fig. Strain map the 3D picture of the $\varepsilon_{xx}$ strain component shows that the GaN region is subjected to a compression (negative value of $\varepsilon_{xx}$).

Fig. Strain components shows the strain components along a cut in the growth direction.

### 5.8 Example 2: Piezoelectric nanogenerator

In this example we will compute the output potential of a piezoelectric nanogenerator based on a vertical compressed ZnO nanowire. The cylindrical column, oriented along the $z$-axis, has a radius of 150 nm and is 4 $\mu$m high.

In the device section we specify the name of the region (Column) and the doping.

```
Device
{
    meshfile = mesh.msh
```

---

5.8. Example 2: Piezoelectric nanogenerator 57
mesh_units = 1e-6
material = ZnO
dimension = 3

Region Column
{
  material = ZnO

  Doping
  {
    Nd = 1e16
    type = donor
    Ed = 0.035
  }
}

In the Physics section of Module elasticity we define an isotropic elastic constant by means of the Young module and Poisson ratio.

Module elasticity
{
  plot = (Strain, Stress, Displacements)

  Physics
  {
    stiffness isotropic
    {
      young = 149
      poisson = 0.349
    }
  }
}

On the upper surface we apply a normal force toward the base of the nanowire. The units are in GPa.

Contact Upper
{
  type = surface_force
  force = (0,0,-0.00625)
}
Finally, the boundary condition *clamp* frozes the column base.

Contact Lower
{
  type = clamp
}

Since we want to perform a simulation at the equilibrium, in Module *driftdiffusion* we only need to solve the Poisson equation. This can be specified by

coupling = poisson

The lower contact is ohmic. The output potential can be taken at the top of the nanowire, where the force is applied, with respect to the electrical potential at the base. This can be done by setting a reference contact, i.e.

reference_contact = Lower

With

default_boundary_condition = zero_displacement

we enforce zero total electric displacement field (*D*) along the simulation boundary.

Module *driftdiffusion*
{

coupling = poisson
reference_contact = Lower
default_boundary_condition = zero_displacement
plot = (ElPotential,ElField,eDensity,Ec,Ev,Polarization,IonizedDonors)

By

strain_simulation = elasticity

we specify the strain simulation, which will provide the strain map.

Physics
{
  strain_simulation = elasticity
}

Contact Lower
{
  type = ohmic
  regions = Lower
  voltage = 0.0
}

To obtain the desired piezopotential in output, we first compute *elasticity* and then *driftdiffusion*.

5.8. Example 2: Piezoelectric nanogenerator
Simulation
{
  solve = (elasticity,driftdiffusion)
  resultpath = output

  output_format = vtk
}

The Figure *Piezopotential* shows the electrical potential across a 200 nm length region at the tip of the NW. The output potential is $\phi_M = -42 \text{mV}$. This value is a result of the competition between the piezoelectric field and the screening due to the free carriers.

![Piezopotential](image.png)

Figure 5.3: Piezopotential
6.1 Theory

Thermal module computes the balance between the heat generation and the heat dissipation. Details about irreversible thermodynamics can be found in [Wachutka]. The heat flux, within the diffusive approach, is given by the Fourier’s law, i.e.

\[ J = -\kappa \nabla T \]  

(6.1)

where \( \kappa \) is thermal conductivity (which is assumed temperature independent). The power balance is ensured by the continuity equation for the thermal flux. i.e.

\[ \nabla \cdot J = H \]  

(6.2)

where \( H \) is the total heat source, which can be a combination of several models (see Heat source).

6.2 Solution/Plot variables

The solution variables available for plotting and for interaction with other modules are given in Plotting variables.

6.3 Solver section

The Solver section of the Thermal module refers to a linear solver. See section Linear solvers for details on linear solver options.

6.4 Physics section

In the following we will describe all the physical models. As mentioned in the Introduction in section InputFileGetting, submodels can be restricted to a subset of simulation regions.
The user can specify the heat source and the thermal conductivity models. If no thermal conductivity is specified, the constant model will be automatically added and the values will be read from the database.

6.4.1 Heat source

The keyword is `heat_source`, i.e.

```
heat_source [type]
{
}
```

The type can be constant or joule

**Constant**

The constant value heat source can be included with `heat_source constant`. Example:

```
heat_source constant
{
    H=1e6
}
```

In this case the heat source is given by a constant power density $H$, expressed in $W/m^3$.

**Joule**

Heat generated by the Joule’s effect can be included with the model type `joule`. This model includes the heat generated by electrons and holes.

Transport simulation, e.g. `driftdiffusion`, from which joule heating is calculated is given by the keyword `transport_simulation`.

Example:

```
heat_source joule
{
    transport_simulation = driftdiffusion
}
```
6.4.2 Thermal conductivity

With this model, a thermal conductivity model can be implemented for a given region or the whole device. As for now, only the constant model type is available.

The keyword is ThermCond, i.e.

```plaintext
thermal_conductivity constant
{
  ThermCond = 0.55
}
```

If the thermal conductivity model is missing, the database values for the device materials are used.

6.5 Boundary conditions

The boundaries of the thermal simulation domain are set by default to ideal thermal insulator. Different boundary conditions can be implemented, namely Dirichlet boundary conditions, surface boundary resistance, thermal flux b.c., by means of the block Contact.

6.5.1 Dirichlet b.c.

Dirichlet boundary conditions can be imposed with the keyword heat_reservoir.

This boundary condition imposes a fixed temperature $T = T_0$, where $T_0$ is the prescribed temperature.

```plaintext
Contact anode
{
  type = heat_reservoir
  temperature = 300
}
```

6.5.2 Surface resistance

A surface boundary resistance can be added with the keyword surface_resistance.

```plaintext
Contact substrate
{
  type = surface_resistance
  r_surf = 0.01
  temperature = 300
}
```
where the surface resistance is given by the keyword \texttt{r\_surf} (in \(m^2 W/K\)).

This boundary condition imposes a surface resistance \(R\), so that the heat flux is constrained to

\[
J_i n_i = \frac{(T - T_0)}{R}
\]

where \(T_0\) is the reference temperature and \(n\) the normal to the surface.

### 6.5.3 Thermal flux

Finally, thermal flux can be imposed in the following way

```java
Contact substrate
{
    type = thermal_flux
    heat_flux = 30
}
```

where \texttt{heat\_flux} is the thermal flux in W/m\(^2\).

This boundary condition imposes a fixed thermal flux, i.e. \(J \cdot n = J_0\), where \(n\) is the normal to the surface and \(J_0\) the prescribed thermal flux.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>LatticeTemp</td>
<td>Lattice temperature</td>
<td>K</td>
</tr>
<tr>
<td>ThermalFlux</td>
<td>Thermal flux</td>
<td>W/m(^2)</td>
</tr>
<tr>
<td>HeatSource</td>
<td>total heat source</td>
<td>W/m(^3)</td>
</tr>
<tr>
<td>ThermCond</td>
<td>thermal conductivity</td>
<td>W/mK</td>
</tr>
</tbody>
</table>

Table 6.1: Solution/Plot variables

### 6.6 Example 1

In the following example, we get the temperature profile of a rectangular device with an hotspot in the center.

The device block specifies the material (Silicon) and the physical regions. In this case we have the \texttt{HotSpot} and the \texttt{Base} region
Device
{
    material = Si
    Region HotSpot{}
    Region Base{}
}

Let us now declare the Module thermal. The temperature is fixed at the left and right side of the domain, by defining the two Contact blocks left and right.

Module thermal
{
    Physics
    {

        heat_source constant
        {
            regions = HotSpot
            H = 1e10
        }

    }

    Contact left
    {
        type = heat_reservoir
        temperature = 300
    }
    Contact right
    {
        type = heat_reservoir
        temperature = 300
    }

}

The hotspot is included only in the HotSpot region through

regions = HotSpot

In the simulation block we define the solution of thermal.

Simulation
{
    solve = thermal
}

The temperature map obtained from the simulation is shown below in Fig. *Lattice temperature*
6.7 Example 2

In the following example, we will see how to perform a thermal-driftdiffusion self-consistent simulation. We include the Seebeck and Peltier effects. The heat conduction through the environment is modelled by adding an air region all around the diode.

We assume that far from the junction the system is in thermal equilibrium with a thermal bath at 300 K. The heat dissipation through the substrate is taken into account by introducing a $0.5cm^2K/W$ thermal surface resistance.

The model of the device (including air region) is shown in Fig. Device Model, together with the mesh.

Furthermore, we rely on the cylindrical symmetry with respect to the growth axis and thus we can consider only a 2D slice of the system.

In this way we simulate a 3D device by performing a 2D simulation (with much less computational time).

Device
{
    material = Si
meshfile = tut_09.msh
dimension = 2
symmetry = cylindrical

Region n_side{
  Doping
  {
    type = donor
    Nd = 1e18
  }
}

Region p_side{
  Doping
  {
    type = acceptor
    Nd = 1e18
  }
}

Region air { material = Air }

We select a 2D dimension for our simulation (dimension = 2), but then we specify the cylindrical symmetry with

symmetry = cylindrical

Region n-side corresponds to the n-doped region situated at the bottom of the device, while the region p-side refers to the top p-doped region. Air is modelled with Region Air.

Here is the Module driftdiffusion

Module driftdiffusion
{
  name = dd

  plot = (Ec, Ev, eQFermi, hQFermi, eDensity, hDensity,
          eMobility, hMobility, NetRecombination, Pn, Pp,
          RecombHeat, ePeltier, hPeltier, eJoule, hJoule,
          ePowerFlux, hPowerFlux, ContactCurrents)

  regions = (n_side, p_side)
The drift-diffusion simulation is performed only over the device domain (excluding the Air region). Therefore, in Module driftdiffusion we have to indicate
regions = (n_side,p_side)

In order to make the drift-diffusion simulation use the results of the chosen thermal simulation model, we write

thermal_simulation = tt

The thermoelectric power model used is the diffusivity model (see Thermoelectric power models) and we can include it by the notation

thermoelectric_power diffusivity_model{}

As for the thermal model

Module thermal
{
    name = tt
    plot = ( LatticeTemp, ThermalFlux, HeatSource,ThermCond,MaxTemp)

    Physics
    {
        heat_source joule
Heat generated by the Joule effect is included by defining an *heat source* model of type *joule*. In this way, the heat generated by electrons and holes is included.

With

```plaintext
transport_simulation = dd
```

we specify the transport simulation which provides the particles for the Joule model (in this case it is the drift-diffusion simulation, but in general it could be any appropriate implemented transport model).

The *dirichlet boundary condition* is set by defining a *Contact* with the type *heat_reservoir* and the temperature specified as 300 K.

The *thermal surface resistance* is included with a *Contact* with the type *surface_resistance*. Its value is given by \( r_{surf} \).

The *selfconsistent* module solves the *thermal* and the *driftdiffusion* simulations (named *tt* and *dd* respectively) in a self consistent way.

```plaintext
Module selfconsistent
{
    solve = (dd,tt)
    absolute_tolerance = 1e-5
    monitor = true
}
```

The sweep section runs the selfconsistent simulation for each anode bias step.
Module sweep
{
    solve = selfconsistent
    variable = $Vb
    start = 0.0
    stop = 1.2
    steps = 12
    plot_data = true
}

Since we have to solve a sweep simulation, we write

\[ \text{solve} = \text{sweep} \]

in the Simulation block

Simulation
{
    verbose = 3
    solve = sweep
    resultpath = output
    output_format = vtk
}

Output of simulation is shown in Fig. Temperature map, Thermal flux and Thermoelectric power. All results displayed are calculated at 1.2 V bias value. One can see that the substrate removes most of the heat generated.
Figure 6.2: Device model and mesh

Figure 6.3: Temperature Map
Figure 6.4: Heat sources

Figure 6.5: Thermoelectric power
7.1 Theory

The semi-classical transport simulation of electrons and holes is based on the drift-diffusion approximation (see [Selberherr]).

Beside the electric potential the electro-chemical potentials are used as variables such that the system of PDEs to be solved reads as follows

\[-\nabla (\varepsilon \nabla \phi - P) = -e(n - p - N_d^+ + N_a^-)\]
\[-\nabla (\mu_n n (\nabla \phi_n + P_n \nabla T)) = R\]
\[-\nabla (\mu_p p (\nabla \phi_p + P_p \nabla T)) = -R\]

(7.1)

\(P\) is the electric polarization due to e.g. piezoelectric effects. \(N_d^+\) and \(N_a^-\) are the densities of ionized donors and acceptors, respectively. \(R\) is the net recombination rate, i.e. recombination rate minus generation rate, and \(P_n\) and \(P_p\) are the electron and hole thermoelectric powers, respectively. The models for the mobilities and the net recombination rates can be specified in the Physics section as described in the following.

7.2 Solution/Plot variables

The solution variables available for plotting and for interaction with other modules are given in Plotting variables.

7.3 Module options

The following options influence the behaviour of the Drift-Diffusion module:
**coupling** [string] defines which equations to solve. The default is `full`, meaning that the full system consisting of the Poisson, electron continuity and hole continuity equations is solved. Other possible values are `poisson` (for equilibrium calculations), `electrons` or `holes`. For the last two cases local equilibrium is assumed such that $\phi_n = \phi_p$.

**enforce_local_charge_neutrality** [bool] If set to true, local charge neutrality will be enforced by setting the charge density to zero. This may be useful for solving the Poisson equation involving only dielectrics.

**guess_el_qfermi** [double] If this option is set, then before resolving the system the given number will be set as a guess for the electron electro-chemical potential.

**guess_hl_qfermi** [double] If this option is set, then before resolving the system the given number will be set as a guess for the hole electro-chemical potential.

**default_boundary_condition** [string] With this option the user can control the default boundary condition for the electric field on all external boundaries without explicit boundary model. Possible values are zero field (default), or zero displacement. The two differ only in presence of electric polarization fields.

**quadrature_rule** [string] This option allows to chose between trapezoidal and Gauss type numeric integration rules. The default rule is `gauss`, but in some cases trapez may prevent density peaks near badly resolved material interfaces.

**save_state** [bool] If set to `true` the current solution will be written to a compressed file after each solve. The file name follows the same rules as the result files, having file extension `.tsv`.

**load_state** [string] Reload a formerly saved solution. The provided string value has to be the relative or absolute path to a `.tsv` file created using the option `save_state`.

**solve_after_load** [bool] If set to `true`, the system will be solved after having reloaded a saved state. Otherwise it will not be solved, which is the default behaviour.

Currently the reload of saved solutions only works correctly when using the identical mesh. Otherwise there will be undefined behaviour or failure.

### 7.4 Solver section

The Solver section of the Drift-Diffusion module refers to a nonlinear solver. See *Nonlinear solvers* for details on nonlinear solver options.
7.5 Physics section

The Physics block contains generic options for the bulk physical model and the definition of submodels. The generic options are:

- **thermal_simulation** If you are doing coupled electrothermal simulations, you have to specify the name of the thermal simulation providing the lattice temperature.

- **strain_simulation** If you are doing simulations on strained systems, you have to specify the name of a strain simulation. The strain values obtained from this model will be used to calculate strain dependent parameters like band parameters and piezoelectric polarizations.

- **relax_polarization** With this option one can specify a global relaxation factor for the electric polarization field. This can be useful if the amount of total electric polarization has to be treated as fitting parameter.

In the following we describe all submodels. As mentioned in the Introduction (REF HERE) submodels can be restricted to a subset of simulation regions.

### 7.5.1 Band parameter models

The model and parameters to calculate band parameters like band edges, band gaps etc. can be controlled by special submodel blocks in different ways:

1. a single `band_properties` block containing parameters for conduction and for valence band
2. `conduction_band` and `valence_band` blocks to control each band independently

Band parameter models may be defined through the block `density_of_states`, described in the following.

### 7.5.2 Density of states (DOS)

The density of states for electrons and/or holes can be set using the `density_of_states` submodel block inside the `band_properties` block, as follows:

```plaintext
band_properties # or conduction_band or valence_band
{
    density_of_states [type]
    {
        ...
    }
}
```
There are several types of density of states models.

3D Bulk DOS (default)

This is the default model describing the 3D DOS of a parabolic band semiconductor. Band edge energies and the DOS mass are taken from the database, or can be provided from the input file. In the current version, if band parameters have to be given from the input file, it is necessary to use the `conduction_band` and `valence_band` blocks instead of `band_properties`. In the `density_of_states` subblock, one can then give the band edge and the DOS mass the keywords `level`, and `dos_mass`, respectively. A degeneracy can be specified with the `degeneracy` keyword. The material whose band parameters are given in this way is defined through the keyword `regions`, which is in general used in the input file to restrict a model to one or more regions of the device. Thus the entry

```
regions = QWell
```

means that the following parameters refer to the material contained in region `QWell`.

For example:

```
valence_band
{
    regions = QWell

density_of_states bulk
{
    dos_mass = 0.80
}
}
```

Here, the Bulk DOS model is chosen for the `valence band` parameters, only for the case of the region named `QWell`. The band parameter given is the DOS effective mass (`dos_mass = 0.80`).

Bulk kp DOS

This model for bulk band parameters can be chosen by specifying `density_of_states bulk_kp`. It calculates the band edge energies and masses from bulk kp theory, including Pikus-Bir strain corrections. To include strain corrections, the keyword `strain_simulation` has to be used, providing the name of the module instance which calculates strain. In the current version it is not possible to provide kp parameters from the input file. For example:

```
```
defines the bulk kp model for all the materials of the device.

It may happen that one would like to apply the bulk kp model to all the device, except one or more regions, where some band parameters should be provided directly from input file. This can be obtained with the following syntax:

```
band_properties
{
    regions = -QWell

density_of_states bulk_kp
{
    strain_simulation = strain
}
}
```

```
conduction_band
{
    regions = QWell

density_of_states bulk_kp
{
    strain_simulation = strain
}
}
```

```
valence_band
{

regions = QWell

density_of_states bulk
{
  dos_mass = 0.80
}
}

In this example the bulk kp model is applied to all the regions (first block), except the region QWell. This is expressed by the syntax

regions = -QWell

a minus sign before the name of the region(s) excludes this (or those) from the model.

Then, in the second block, it is stated that the bulk kp model is to be applied to the region QWell, but only for the case of conduction band. Finally, in the third block, the bulk DOS model is assigned to the valence band of the region QWell and a value of dos mass is given in input (dos_mass = 0.80). To summarize, in this way the bulk kp model is applied to all the regions of the device, except for one (QWell), where the bulk kp is used for CB, while the bulk model, with a user defined value of dos mass, is used for the material in the regione named QWell.

Quantum DOS

With this model, the local DOS is obtained from the solution of the Schrödinger equation in a defined system. The following keywords are used to control this model:

- **quantum_simulation** The name of a quantum density simulation. This will use the quantum mechanical particle density in the regions it was calculated. More than one simulations can be specified as a vector. In this case the sum of all quantum densities is taken.

If quantum_simulation is specified, the following additional options control the mixing between classical and quantum density:

- **barrier_regions** Regions which can be regarded as pure barriers. In these regions a classical density will be added using the barrier materials bulk band edge.

A further submodel block may be added to define an alternative model to be used to obtain the DOS inside the barrier regions defined by barrier_regions, or when the quantum density is not available, for example because it has not been yet calculated in the simulation. This block is called classical_DOS, and its type may be any one of the types defined for density_of_states model, as follows:
classical_DOS [type]
{
  ...
}

For example:

conduction_band
{
  density_of_states quantum
  {
    # where to get the quantum density from
    quantum_simulation = quantum_el
    barrier_regions = buffer_quantum

    classical_DOS bulk_kp
    {
      strain_simulation = strain
    }
  }
}

Here, quantum DOS, and thus quantum density, for electrons, is obtained from the simulation quantum_el, for example an EFA calculation. A classical DOS is defined, to be obtained with the model bulk_kp.

**Constant DOS**

The constant DOS model is identified by the identifier constant. The constant DOS has the following form, for $E_{\text{min}} \leq E \leq E_{\text{max}}$:

$$g(E)dE = \frac{N_0}{E_{\text{max}} - E_{\text{min}}}dE$$

and $g(E) = 0$ otherwise. In the input file the constant DOS is set using three keywords, $N_0$, level and Ewidth. The first one is used to set the maximum occupied density $N_0 (cm^{-3})$. If a value for level is not specified in the input file, it will be set by default as the conduction band edge ($E_C$) for electrons, and as the valence band edge ($E_V$) for holes. The values of $E_C$ and $E_V$ are taken from material files. For electrons $E_{\text{min}} = \text{level}$, and $E_{\text{max}} = \text{level} + \text{Ewidth}$; for holes $E_{\text{max}} = \text{level}$, and $E_{\text{min}} = \text{level} - \text{Ewidth}$.
Gaussian DOS

The gaussian DOS model is identified by the identifier *gauss*. The gaussian DOS has the following form

\[ g(E, \sigma) dE = \frac{N_0}{\sigma \sqrt{2\pi}} \exp \left( -\frac{1}{2} \left( \frac{E - E_{C,V}}{\sigma} \right)^2 \right) dE \]

where \( N_0 (cm^{-3}) \) is the maximum occupied density of carriers; \( \sigma (eV) \) is the variance of the gaussian, hence it is a measure of the energy disorder; \( E_C \) and \( E_V \) are the conduction and valence band edges respectively. Band edges are set in the material files of TiberCAD, but a different center for the gaussian can be specified with the keyword *level = [value]*. \( N_0 \) and \( \sigma \) can be set with the keywords *N0 = [value]* and *sigma = value* respectively.

The gaussian DOS can be used also to define a distribution of trap states (see section *Trap models*), for example:

```
trap eNeutral
{
  regions = ...
  Nt = 1e18
  Et = 1.0
  reference = cb

  density_of_states
  {
    type = gauss
    N0 = 1
    sigma = 0.1
  }
}
```

According with the conventions used in TiberCAD the maximum density of traps is obtained as the product \( N_t \times N_0 \), hence one can set indifferently \( N_t \) or \( N_0 \), leaving one parameter set to 1. The center of the gaussian is set with the value \( E_t \), thus it is not necessary to provide a value for *level*. \( E_t \) is specified with respect to the conduction band (*reference = cb*), or the valence band (*reference = vb*) or the midgap (*reference = m*). The example sets a gaussian distribution of *eNeutral* traps whose center is 1 eV below the conduction band.

The gaussian DOS can be used also to define a distribution of trap states (see section *Trap models*). This is particularly useful when modeling metal/semiconductor interfaces using the IDIS model (Induced Density of Interface States, see [Santoni]). According to the IDIS model a density \( N_{trap} \) of traps is distributed on the interface and a Charge Neutrality Level (CNL) is defined in such a way that when the fermi level is below the CNL the interface is positively charged, and when the fermi level is above the CNL the interface is negatively charged. This situation can be simulated using *eNeutral* traps over the CNL, and *hNeutral* traps below the CNL. For Alq3 the CNL is...
1.68 $eV$ above the valence band and $N_{\text{trap}} = 2.63 \times 10^{19} \, cm^{-3}eV^{-1}$. Thus if we want to simulate a constant distribution of $e\text{Neutral}$ traps above the CNL extending up to the conduction band edge (bandgap for $Alq_3$ is 2.9 $eV$) we have to set:

```plaintext
trap eNeutral
{
    regions = ...
    Nt = 3.21e19
    Et = 1.68  #$E_t$level of the constant DOS is automatically fixed by this value
    reference = vb #$E_t$ is set w. r. t. valence band edge

    density_of_states
    {
        type = constant
        Ewidth = 1.22  #$Alq_3$ bandgap - $E_t$
        N0 = 1
    }
}
```

And to set a constant distribution of $h\text{Neutral}$ traps between the valence band edge and the CNL we write:

```plaintext
trap hNeutral
{
    regions = ...
    Nt = 4.42e19
    Et = 1.68
    reference = vb

    density_of_states
    {
        type = constant
        Ewidth = 1.68  #$E_t$ is the same value of Et because we have set $E_t$ as the CNL w. r. t. valence band
        N0 = 1
    }
}
```

According with the conventions used in TiberCAD the maximum density of traps is obtained as the product $N_t \times N_0$, hence one can set indifferently $N_t$ or $N_0$, leaving one parameter set to 1. We have also to keep in mind that the constant distribution is normalized in energy, so $N_t$ (or $N_0$) is automatically divided by $E_{\text{width}}$. But the density of states $N_{\text{trap}}$ is already in $cm^{-3}eV^{-1}$ units, so we have to multiply it by 2.9 - 1.68 = 1.22 $eV$ for $e\text{Neutral}$ and by 1.68 $eV$ for $h\text{Neutral}$ in order to get the correct volume density for $N_t$ (or $N_0$).
7.5.3 Recombination/generation models

This section describes the currently available generation/recombination models. Note that all recombination models can be applied also to surfaces/interfaces. Recombination models are controlled by means of recombination submodel blocks inside the Physics section. Different recombination models having the same (or no) options can be enabled in a single statement by writing:

recombination (model1, model2, ...) {} 

**Shockley-Read-Hall (SRH) recombination**

The SRH recombination model can be enabled by defining a recombination submodel of type srh.

SRH recombination is defined as follows:

\[
R_{SRH} = \frac{np - n_i^2}{(n + n_i e^{E^*/k_BT})\tau_p + (p + n_i e^{-E^*/k_BT})\tau_n} 
\]

\(E^* = E_{trap} - (E_c + E_v)/2\) is the trap level with respect to the midband energy.

\(n_i\) is the intrinsic carrier density, \(\tau_n\) and \(\tau_p\) are the recombination times.

The parameters are taken from the material database. The recombination times are dependent on temperature and doping density, e.g.

\[
\tau_n = \tau_0^n \left( \frac{T}{T_0} \right)^{\alpha_n} e^{\beta(T/T_0 - 1)} 
\]

\[
\tau_0^n = \tau_{min,n} + \frac{\tau_{max,n} - \tau_{min,n}}{1 + (N/N_{ref})^{\gamma}} 
\]

where \(T_0\) is the reference temperature (300 K). Table 2.3 shows the corresponding parameters for the material data files. The parameters for holes and electrons have to be specified in an array, e.g. \(\tau_{min} = (1e - 5, 3e - 6)\)

The recombination times and trap level can be overridden from the input file by using the keywords of Table 2.4.

The SRH recombination model can be applied also to surfaces and interfaces. In this case, you can provide the recombination velocities using the keywords rec_velocity_n and rec_velocity_p instead of tau_n and tau_p.

**Direct (radiative) recombination**

The direct recombination model can be enabled in the input file by by defining a recombination submodel of type direct.
Direct recombination is modeled as follows:

\[ R_{\text{direct}} = C(np - n_i^2) \]  

(7.3)

The material data file and the input file use the same keyword C for the parameter C. The database value can be overridden from the input file as described for SRH recombination.

**Langevin (radiative) recombination**

The Langevin recombination model can be enabled in the input file by defining a recombination submodel of type `langevin`.

Langevin recombination is modeled as follows:

\[ R_{\text{Langevin}} = \gamma \frac{e}{\varepsilon} (\mu_n + \mu_p)(np - n_0p_0) \]  

(7.4)

\( \mu_n \) and \( \mu_p \) are electron and hole mobilities respectively and their values are taken from the proper model specified in the simulation (see *Mobility models*); \( n_0 \) and \( p_0 \) are equilibrium electron and hole densities respectively; \( \gamma \) is a constant that can be set using the keyword `gamma = [value]` (by default value is set to 1).

**Auger recombination**

The Auger recombination model can be enabled in the input file by defining a recombination submodel of type `auger`.

Auger recombination is modeled by the following equation

\[ R_{\text{auger}} = (C_n n + C_p p)(np - n_i^2) \]  

(7.5)

with temperature dependent parameters

\[ C_{\{n,p\}} = \left(A + B \frac{T}{T_0} + C \left(\frac{T}{T_0}\right)^2\right) \left(1 + He^{-\{n,p\}/N_0}\right) \]

The parameters \( A; B; C; H \) and \( N_0 \) are taken exclusively from the database. They are different for \( C_n \) and \( C_p \) and have to be specified as arrays with keywords \( A, B, C, H, N0 \), e.g. \( A = (1e-31, 1e-32) \). The calculated values for \( C_n \) and \( C_p \) can be overridden from the input file by specifying values for the keywords `Cn` and `Cp`.

**Optical generation**

A very simple model for photoelectric generation of electron-hole pairs is implemented in tiberCAD. It is enabled by specifying a generation submodel of type `optical` The model imposes a constant generation rate which has to be provided by the keyword G in units of \((\text{cm}^{-3} \cdot \text{s}^{-1})\).
Usually the simulation should define a sweep on the value of $G$ from 0 to the desired generation.

### 7.5.4 Thermoelectric power models

The thermoelectric power models are the same for electrons and holes.

The keyword is `thermoelectric_power`, i.e.

```plaintext
germs.pngthermoelectric_power [type]
{
}
```

The type can be `constant` (i.e. the thermoelectric powers are read from the database) or `diffusivity_model` where the thermoelectric powers are computed by

\[
P_n = -\frac{k_b}{q} \left( \frac{5}{2} + \frac{e\phi_n + E_n - e\varphi}{k_bT} \right) \\
P_p = \frac{k_b}{q} \left( \frac{5}{2} - \frac{e\phi_p + E_v - e\varphi}{k_bT} \right)
\]  \hspace{1cm} (7.6)

The default is $P_n = P_p = 0$

### 7.5.5 Mobility models

The models to be used for electrons and holes can be defined in a single submodel block or independently using two blocks. The corresponding keywords are `mobility` or `electron_mobility` and `hole_mobility`, i.e.

```plaintext
gear.pngmobility [type]
{
}
```

```plaintext
gear.pngelectron_mobility [type]
{
}
```

```plaintext
gear.pnghole_mobility [type]
{
}
```

When using the first approach, both carriers will use the same model, and parameters provided in the input file will also be used by both carriers. When mixing the different definitions, the blocks `electron_mobility` and `hole_mobility` will override the common `mobility` block.
The default model is the constant mobility model. The parameters for the different mobility models are needed for both electrons and holes. In the material files they are specified with a common keyword in arrays, e.g.

<table>
<thead>
<tr>
<th>Mobility Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>mobility</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>mu_max</strong></td>
</tr>
<tr>
<td><strong>exponent</strong></td>
</tr>
</tbody>
</table>

Table 7.1: Mobility Model

**Constant mobility model**

The constant mobility model (identifier constant) assumes a mobility which depends only on temperature by means of the following formula:

\[
\mu_{\text{const}} = \mu_0 \left( \frac{T}{T_0} \right)^{-\gamma} \tag{7.7}
\]

In the material data file \( \mu_0 \) and \( \gamma \) have to be specified with the keywords mu_max and exponent. \( \mu_0 \) can be overridden from the physical_model section using the keyword mu or from the Region sections using the keywords mu_e and mu_h.

**Doping dependent mobility model**

The doping dependent mobility model (identifier doping_dependent) implements two models for mobility depending on the total doping density and the temperature. The model that is used depends on the value of the mobility_formula parameter.

**Model by Masetti et al.**

The model by Masetti et al. Masetti is identified by mobility_formula = 1. It uses the following formula:

\[
\mu = \mu_{\text{min},1} \cdot e^{-P_c/N} + \frac{\mu_{\text{const}} - \mu_{\text{min},2}}{1 + (N/C_p)^a} - \frac{\mu_{1}}{1 + (C_s/N)^{\beta}} \tag{7.8}
\]

where \( N \) is the total doping density and \( \mu_{\text{const}} \) the mobility obtained from the constant mobility model. The parameters are specified in the material file as given in Table 2.5.

**Model by Arora**

The model by Arora Arora is identified by mobility_formula = 2. It reads:

\[
\mu = \mu_{\text{min}} + \frac{\mu_d}{1 + (N/N_0)^{A_v}} \tag{7.9}
\]
with

\[ \mu_{\text{min}} = A_{\text{min}}(T/T_0)^{\alpha_{\text{min}}} \quad \mu_d = A_d(T/T_0)^{\alpha_d} \]
\[ N_0 = A_N(T/T_0)^{\alpha_N} \quad A^* = A_a(T/T_0)^{\alpha_a} \]

The parameters are given in table at the end of the Chapter.

**Field dependent mobility model**

The field dependent mobility model describes the degradation of mobility at high driving fields. It is identified by the identifier field_dependent. The electric field component in direction of the current flow, the gradient of the electro-chemical potential or a so called “field parameter” can be chosen as driving force:

\[ \text{driving force} = \text{efield} \mid \text{grad} \_\text{fermi} \mid \text{field parameter} \]

The default driving force is the gradient of the corresponding electro-chemical potential \( \nabla \phi \). field_parameter uses a field parameter given by \( \sqrt{E \cdot \nabla \phi} \) as driving force [Zakhleniuk].

The model is based on the Caughey-Thomas model, refined by Canali [6]:

\[ \mu = \frac{\mu_{\text{lowfield}}}{\left( 1 + \left( \frac{\mu_{\text{lowfield}}|E|}{v_{\text{sat}}} \right)^{\frac{1}{\beta}} \right)^{1/\beta}} \]  
(7.10)

with

\[ \beta = \beta_0(T/T_0)^b \]

\(|E|\) is the modulus of the driving field, multiplied by a damping factor \( n/(n + n_0) \), where \( n \) is the electron or hole density, and \( n_0 \) is a parameter with default \( 10^9 \). \( \mu_{\text{lowfield}} \) is the low-field mobility. For the latter one can specify the model to be used using the parameter lowfield_model. As default the doping dependent model is used. The damping factor is needed to improve convergence at low carrier densities. There are two models for vsat, identified with Vsat_Formula = 1 and 2.

Formula 1 reads

\[ v_{\text{sat}} = v_{\text{sat},0}(T/T_0)^{-\gamma} \]  
(7.11)

Formula 2 reads

\[ v_{\text{sat}} = \max(A_{\text{vsat}} - B_{\text{vsat}}(T/T_0), v_{\text{min}}) \]  
(7.12)

The parameters for the field dependent mobility model are summarized in Table 2.7.
Field assisted mobility model

The field assisted mobility model describes the enhancement of the carrier mobility by an electric field in organic semiconductors. It is identified by the identifier field_enhanced.

The model is given by equation (7.13):

$$\mu = \mu_0 e^{\sqrt{|E|/E_0}}$$

where $|E|$ is the modulus of the driving field, $\mu_0$ is the zero-field mobility and $E_0$ is a critical field strength.

The parameters for the field assisted mobility model are the following (summarized in Table Field assisted mobility parameters):

- **$\mu_0$** The mobility at low electric field.
- **$E_0$** The critical electric field strength.

<table>
<thead>
<tr>
<th>parameter name</th>
<th>default value</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu_0$</td>
<td>0.0054</td>
<td>cm$^2$/Vs</td>
</tr>
<tr>
<td>$E_0$</td>
<td>$3 \cdot 10^5$</td>
<td>V/cm</td>
</tr>
</tbody>
</table>

Table 7.2: Field assisted mobility parameters

Hopping mobility model

The hopping mobility model is obtained as a parametrical fit of numerical solutions of a system of master equations implementing the Miller-Abrahams hopping model between localized states with a gaussian energy distribution (see [Pasveer]). It is identified by the identifier hopping_mobility. The expression for mobility is:

$$\mu(T, \rho, F) = \mu(T, \rho)f(T, F)$$

$$\mu(T, \rho) = \mu_0 c_1 \exp \left[ -c_2 \left( \frac{\sigma}{k_B T} \right)^2 \right] \exp \left[ \frac{1}{2k_B T} \left( \frac{\sigma^2}{k_B T} - \sigma \right) (2\rho a^3) \right]$$

$$f(T, F) = \exp \left\{ 0.44 \left[ \left( \frac{\sigma}{k_B T} \right)^{3/2} - 2.2 \right] \left[ \sqrt{1 + 0.8 \left( \frac{eF a}{\sigma} \right)^2} - 1 \right] \right\}$$

where $\rho$ is the carrier density (electrons or holes); $F$ is the electric field; $T$ the temperature $a = N_0^{-1/3}$ is the average distance between sites ($N_0$ is the maximum carrier density (cm$^{-3}$) for the
gaussian DOS); \(\sigma\) is the variance of the gaussian DOS; \(\nu_0\) is the attempt to jump frequency of the Miller-Abrahams model; \(c_1 = 1.8 \times 10^{-9}\); \(c_2 = 0.42\) and

\[
\delta = 2 \frac{\log (s^2 - s) - \log (\log 4)}{s^2} \quad \mu_0 = \frac{a^2 \nu_0 \epsilon}{\sigma}
\]

where \(s = \frac{\sigma}{k_B T}\).

It is important to notice that this formula makes sense physically only if the condition \((\frac{\sigma}{k_B T})^2 < 2.2 > 0\) is fulfilled. In the TiberCAD input file the hopping mobility model has to be set providing values for \(\sigma, N_0\) and \(\nu_0\) using the respective keywords \(\text{sigma = [value]}, N0 = ... \text{ and nu0 = ... }\). This mobility model makes sense only if used together with the gaussian density of states, hence the values of \(\text{sigma}\) and \(\text{N0}\) have to be the same as the corresponding values set in the density of states model for the gaussian DOS.

As explained in [Santoni] two cut-off have been implemented. For \(F > \frac{2\sigma}{e\alpha}\)

\[
f(T, F) = f(T, \frac{2\sigma}{e\alpha})
\]

For \(\rho > 0.1N_0\)

\[
\mu(T, \rho) = \mu(T, 0.1N_0)
\]

### 7.5.6 Polarization models

For simulations involving materials with nonzero electric polarization (such as nitrides) it is important to include the effect of polarization. This is done by specifying the models for spontaneous (pyro-) and piezoelectric polarization using the keyword \text{polarization} with the types \text{pyro} and \text{piezo}

\[
polarization \ pyro \ {} \\
polarization \ piezo \ {} \\
\]

As for all models, if they do not have individual options, they can be specified together by writing

\[
polarization \ (\text{pyro, piezo}) \ {} \\
\]

### Spontaneous polarization

The spontaneous polarization model (sometimes also called ‘pyroelectric polarization’) imposes a constant electric polarization \(P\) along the symmetry-breaking direction of the crystal. Crystals with wurtzite structure like Nitrides have strong polarization fields along the c-direction. The value of the polarization usually is taken from the database, but it can be overridden from the input file by specifying the option \(Pz\), meaning the value of the spontaneous polarization along c-direction ([0001]). Alternatively, one can specify explicitly a polarization vector using the option \(P = (Px, Py, Pz)\). This is useful to impose an arbitrary constant polarization field.
Piezopolarization

The piezoelectric polarization is strain induced and given by the linear relationship

\[ P_{pz} = e_{ikl} \varepsilon_{kl} \]  

(7.14)

where \( \varepsilon_{kl} \) is the strain tensor. The piezoelectric coefficients \( e_{ikl} \) are stored in the database. The strain is obtained from the simulation specified in the Physics section, but it can be overridden by providing a name for the strain simulation inside the polarization block using the strain_simulation option.

polarization (piezo) {strain_simulation = strain}

7.5.7 Trap models

Currently single level traps are implemented in TiberCAD. Traps can be normally neutral or normally charged electron or hole traps, or a fixed charge. Common options for all models are

- **type** The type of traps. One of eNeutral, hNeutral, donor, acceptor or fixed_charge. (Only necessary if not provided as second keyword).
- **Nt** The trap density in cm\(^3\) (or cm\(^2\) for surface traps).
- **Et** The trap level in eV with respect to the reference energy.
- **reference** The reference energy. The default is m for midgap. Possible values are cb, vb or m
- **recombination_center** Flag to switch on and off the recombination through a trap

For \( \text{reference} = \text{m} \) for example, the trap energy is given as \( E_{\text{trap}} = E_{\text{midgap}} + Et \). In the other cases it is \( E_{\text{trap}} = E_c - Et \) or \( E_{\text{trap}} = E_v + Et \). The following trap types are implemented:

- **eNeutral** The trapped electron density is given by

\[ n_t = \frac{N_t}{1 + \exp\left(\frac{E_{\text{trap}} - E_{F,n}}{k_B T}\right)} \]  

(7.15)

- **hNeutral** The trapped hole density is given by

\[ p_t = \frac{N_t}{1 + \exp\left(-\frac{E_{\text{trap}} - E_{F,p}}{k_B T}\right)} \]  

(7.16)

- **donor** The density of ionized traps is given by

\[ N_{t}^+ = N_t - \frac{N_t}{1 + \exp\left(\frac{E_{\text{trap}} - E_{F,n}}{k_B T}\right)} \]  

(7.17)
**acceptor**  The density of ionized traps is given by

\[
N_t^+ = N_t - \frac{N_t}{1 + \exp(-\frac{E_{\text{trap}} - E_{F,p}}{k_B T})}
\]  

(7.18)

If traps are specified, the total charge density in the Poisson equation is modified to include the charged trap densities:

\[
\rho = e \left( p - n + N_D^+ - N_A^- - \sum n_t + \sum p_t + \sum N_t^+ - \sum N_t^- \right)
\]

(7.19)

Additionally, each trap can induce a SRH recombination term of the form

\[
R_t = \frac{v_{th}^n \sigma^n v_{th}^p \sigma^p (np - n_t^2)}{v_{th}^n \sigma^n (n + n_1) + v_{th}^p \sigma^p (p + p_1)}
\]

(7.20)

where \(\sigma^n, p\) are the capture cross sections, \(v_{th}^n, p\) the thermal velocities and (for Boltzmann statistics)

\[
n_1 = n_{i,\text{eff}} \exp(E_{\text{trap}}/k_B T), \quad p_1 = n_{i,\text{eff}} \exp(-E_{\text{trap}}/k_B T)
\]

(7.21)

The SRH (see Recombination/generation models) recombination model associated with a trap has to be enabled explicitly by using the option `recombination_center = true` in the trap definition. If `recombination_center = true` is specified, trap-assisted tunneling (TAT) as described in the following section can be enabled by adding a subblock `trap_assisted_tunneling` as follows:

```plaintext
trap_assisted_tunneling
{
    tunneling_mass = 0.45
}
```

This will activate the Hurkx TAT model with the specified tunneling mass, using the trap energy level given in the trap description.

### 7.5.8 Tunneling models

Several simplified tunneling models are implemented in tiberCAD. They are derived from quantum mechanical models under simplifying assumptions and cast into local recombination-generation models.

#### Trap-assisted tunneling

Trap-assisted tunneling (TAT) in both forward and reverse bias is taken into account with a recombination model proposed by Hurkx (see [Hurkx]). In this model, a modified expression for the SRH recombination is found, where the carrier SRH lifetimes are modified by field-effect functions \(\Gamma_n\) and \(\Gamma_p\) as \(\tau_{n,p} = \tau_{n,p,0}/(1 + \Gamma_{n,p})\). The field-effect functions vanish for weak electric fields, yielding the conventional SRH formula.
The model for the Trap-assisted tunneling can be enabled by defining a recombination sub-model of type srh, adding the keyword `trap_assisted_tunneling` as follows:

```plaintext
recombination srh
{
  Et = 0.56
  reference = cb
  trap_assisted_tunneling = true
}
```

The parameters are the following (see also section *Trap models*):

- **Et** The trap level in eV with respect to the reference energy.
- **reference** The reference energy. The default is \( m \) for midgap. Possible values are \( cb \), \( vb \) or \( m \)
- **trap_assisted_tunneling** if true tunneling through the defined trap is switched on

(see [Hurkx]).

**Band-to-band tunneling**

For direct band-to-band tunneling, a model that is also due to Hurkx is implemented. This model can be written as a local generation-recombination process:

\[
G^{b2b} = B E^\alpha exp(E_0/E)
\]  

(7.22)

The model is activated by defining a recombination submodel of type `band2band` as follows:

```plaintext
recombination band2band
{
  B = 4e14
  E0 = 1.9e7
  sigma = 2.5
}
```

The parameters for the band to band tunneling model are summarized in Table *Hurkx band to band tunneling parameters*, together with their default value: (see [Hurkx]).

<table>
<thead>
<tr>
<th>parameter name</th>
<th>default value</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>( 4 \cdot 10^{14} )</td>
<td>( \text{I/cm}^3\text{s} )</td>
</tr>
<tr>
<td>E0</td>
<td>( 1.9 \cdot 10^{7} )</td>
<td>( \text{V/cm} )</td>
</tr>
<tr>
<td>sigma</td>
<td>2.5</td>
<td>( \text{l} )</td>
</tr>
</tbody>
</table>

Table 7.3: Hurkx Band to band tunneling parameters
7.5.9 Boundary conditions

Boundary conditions are implemented for ohmic contacts, Schottky contacts, free surfaces and interfaces. Contacts are boundary models that allow a nonzero normal electrical current. The applied voltage is specified with the option $voltage$.

A variable can be assigned to this, using the $-$syntax. On ohmic or schottky contacts one can define surface recombination velocities for electrons and holes using the options $\text{rec_velocity}_e$ and $\text{rec_velocity}_h$. This will impose Robin type boundary conditions for the continuity equations of the form

$$\begin{align*}
-\nabla\left[\mu_n n (\nabla \phi_n + P_n \nabla T)\right] &= v_n (n - n_0) \\
-\nabla\left[\mu_p p (\nabla \phi_p + P_p \nabla T)\right] &= -v_p (p - p_0)
\end{align*}$$

The options $\text{zero_field}$, $\text{zero_grad_fermi}_e$ and $\text{zero_grad_fermi}_h$ can be used, which when set to $\text{true}$ will impose zero normal electric field and zero normal gradient of the electron and hole electro-chemical potential, respectively. The latter are special cases of surface recombination velocities ($v_{\text{rec}} = 0$).

Contacts are defined by blocks with keyword Contact, for example:

```
Contact anode
{
    type = ohmic
    [regions = (anode1, anode2)]
    voltage = $Vd$
}
```

An area factor can be specified for contacts using the keyword $\text{area_factor}$. The contact current will be multiplied by this factor.

For interfaces and surfaces, the same syntax can be used (optionally one can use the keywords $\text{Interface}$ or $\text{Boundary}$), however they do usually not need to be defined explicitly.

**Ohmic contact**

The ohmic contact (identifier $\text{ohmic}$) has no further parameters.

**Schottky contact**

A Schottky contact (identifier $\text{schottky}$) has the additional parameter $\text{barrier}$, which signifies the energy difference between the semiconductor band edge and the fermi energy in the metal. As default, the barrier is taken with respect to the conduction band. By specifying $\text{band} = \text{v}$ the barrier can be imposed with respect to the valence band (p-type contact). Alternatively, the metal work function can be defined using the keyword $\text{work_function}$ or the keyword $\text{metal_fermilevel}$. The latter is just the work function with inverted sign.
The value given in `work_function` or `metal_fermilevel` has to be aligned with the band energies given in the material files, not with that resulting from simulation.

The `fixed_barrier` controls the behaviour of the barrier height for strained materials. If it is set to true, the barrier will be independent of strain (default behaviour). If it is set to false, the given barrier is used as barrier for the unstrained case and will depend on strain during simulation. If the metal work function is specified, the barrier will be strain dependent as default. Thermionic emission is by default switched on, but can be disabled by specifying `thermionic_emission = false`.

If a Schottky contact is touching different materials, one should specify the work function instead of the barrier.

`barrier_lowering = true` activates the image charge lowering of the Schottky barrier (see [Santoni]); if it is not specified, by default it is set to false. The Schottky barrier $\phi_B$ is lowered by a quantity depending on the electric field $F$:

$$
\phi_B = \phi_{B0} - \sqrt{\frac{eF}{4\pi\varepsilon}}
$$

Internally TiberCAD sets conduction and valence band edges, then it fixes the barrier setting a value of the fermi level inside the band gap. It is important to notice that it is not possible to lower the barrier for both electrons and holes, in fact lowering the barrier for electrons (`band = c`) means to shift the fermi level toward the conduction band, and accordingly the barrier for holes raises, and vice versa lowering the barrier for holes (`band = v`) means to shift the fermi level toward the valence band, and accordingly the barrier for electrons raises.

`scott_malliaras = true` tells TiberCAD to use the [Scott] and Malliaras model of recombination velocity at the contact; if it is not specified, by default it is set to false and the thermal velocity is used instead.

$$
v_{rec} = v_{SM} (f) = \frac{v_{SM} (0)}{4} \left( \frac{1 - \psi^2 (f)}{\psi^2 (f)} - f \right)
$$

$$
v_{SM} (0) = \frac{16\pi\varepsilon (k_B T)^2 \mu}{e^3}
$$

$$
\psi (f) = f^{-1} + f^{-\frac{1}{2}} - f^{-1} \left( 1 + 2f^{\frac{1}{2}} \right)^{\frac{1}{2}}
$$
\[ f = \frac{eF_C}{k_B T}, \quad r_C = \frac{e^2}{4\pi\varepsilon k_B T} \]

The Scott and Malliaras model takes into account the effect of the electric field, and it is used in organic semiconductors because it does not depend on the effective mass, a parameter not well defined in organic materials.

**Interface/surface model**

The free surface or interface model (identifier interface) can include surface charges due to traps and surface recombination. Their definition can be found in section *Trap models*.

Each trap model will induce automatically a SRH recombination model as in the bulk case.

### 7.6 Schrödinger/Poisson/Drift-Diffusion calculations

tiberCAD is able to perform selfconsistent Schrödinger-Poisson or Schrödinger-Drift-Diffusion calculations. For this purpose, a density of states model of type quantum has to be specified in Physics for at least one of the carriers, and a selfconsistent simulation should be defined in the Selfconsistent block (see *Selfconsistent*). The following option, to be specified in the Physics section, controls the behaviour of the selfconsistent simulation.

`use_density_predictor` When set to true, a predictor-corrector scheme will be adopted in the selfconsistent cycle. The Poisson/Drift-Diffusion solver does not just take the particle densities as given by the Schrödinger calculation, but it will assume a dependency of the density on the potentials of the form

\[
\rho(\varphi, \phi_n, \phi_p) = \frac{\rho_{\text{quantum}}(\varphi^0, \phi^0_n, \phi^0_p)}{\rho_{\text{classical}}(\varphi^0, \phi^0_n, \phi^0_p)} \rho_{\text{classical}}(\varphi, \phi_n, \phi_p)
\]

(7.23)

where \((\varphi^0, \phi^0_n, \phi^0_p)\) are the potentials for which the quantum density was calculated. use_density_predictor = true is the preferred method for selfconsistent Schrödinger-Poisson/Drift-Diffusion calculations and is enabled by default.

For example, in Module driftdiffusion

```
Physics
|
| conduction_band
| |
| density_of_states quantum
| |
| # where to get the quantum density from
```
quantum_simulation = quantum_el
barrier_regions = buffer_quantum

classical_DOS bulk_kp
{
    strain_simulation = strain
}
}

valence_band
{
    density_of_states quantum
    {
        # where to get the quantum density from
        quantum_simulation = quantum_hl
        barrier_regions = buffer_quantum

        classical_DOS bulk_kp
        {
            strain_simulation = strain
        }
    }
}

In this example, we include both electrons and holes in the selfconsistent simulation, by defining a quantum model of DOS for both conduction_band and valence_band. Here quantum_el and quantum_hl are the efaschroedinger simulations which calculate the quantum densities for electron and holes, respectively.

Then, in Module selfconsistent

Module selfconsistent
{
    solve = (quantum_el, quantum_hl, driftdiffusion)
    # we do not use relaxation, but a predictor-corrector scheme
    #relaxation_factor = 0.5
    max_iterations = 10
    absolute_tolerance = 1e-3
    relative_tolerance = 1e-8
    monitor = true
    #xmonitor = true
}

Again, quantum_el and quantum_hl are the efaschroedinger simulations which calculate the quantum densities for electron and holes. The solve statement here specifies the order of

7.6. Schrödinger/Poisson/Drift-Diffusion calculations
execution in the self-consistent cycle, which is repeated until the requested tolerance is reached.

### Solution Table

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ec</td>
<td>Conduction band edge</td>
<td>eV</td>
</tr>
<tr>
<td>Ev</td>
<td>Valence band edge</td>
<td>eV</td>
</tr>
<tr>
<td>eQFermi</td>
<td>Electro-chemical potential of electrons</td>
<td>eV (−eφ&lt;sub&gt;n&lt;/sub&gt;)</td>
</tr>
<tr>
<td>hQFermi</td>
<td>Electro-chemical potential of holes</td>
<td>eV (−eφ&lt;sub&gt;p&lt;/sub&gt;)</td>
</tr>
<tr>
<td>Ec0</td>
<td>Conduction band edge without electric potential</td>
<td>eV</td>
</tr>
<tr>
<td>Ev0</td>
<td>Valence band edge without electric potential</td>
<td>eV</td>
</tr>
<tr>
<td>Eg</td>
<td>Band gap</td>
<td>eV</td>
</tr>
<tr>
<td>ConductionBands</td>
<td>Minimal of all conduction bands</td>
<td>eV</td>
</tr>
<tr>
<td>ValenceBands</td>
<td>Maximum of all valence bands</td>
<td>eV</td>
</tr>
<tr>
<td>EIPotential</td>
<td>Electric potential</td>
<td>V</td>
</tr>
<tr>
<td>eDensity</td>
<td>Electron density</td>
<td>cm&lt;sup&gt;−3&lt;/sup&gt;</td>
</tr>
<tr>
<td>hDensity</td>
<td>Hole density</td>
<td>cm&lt;sup&gt;−3&lt;/sup&gt;</td>
</tr>
<tr>
<td>eMobility</td>
<td>Electron mobility</td>
<td>cm&lt;sup&gt;2&lt;/sup&gt;V&lt;sup&gt;−1&lt;/sup&gt;s&lt;sup&gt;−1&lt;/sup&gt;</td>
</tr>
<tr>
<td>hMobility</td>
<td>Hole mobility</td>
<td>cm&lt;sup&gt;2&lt;/sup&gt;V&lt;sup&gt;−1&lt;/sup&gt;s&lt;sup&gt;−1&lt;/sup&gt;</td>
</tr>
<tr>
<td>eConductivity</td>
<td>Electron conductivity</td>
<td>S/cm</td>
</tr>
<tr>
<td>hConductivity</td>
<td>Hole conductivity</td>
<td>S/cm</td>
</tr>
<tr>
<td>EField</td>
<td>Electric Field</td>
<td>Vcm&lt;sup&gt;−1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Polarization</td>
<td>Electric Polarization</td>
<td>Cm&lt;sup&gt;−2&lt;/sup&gt;</td>
</tr>
<tr>
<td>CurrentDensity</td>
<td>Total current density</td>
<td>Acm&lt;sup&gt;−2&lt;/sup&gt;</td>
</tr>
<tr>
<td>eCurrentDensity</td>
<td>Electron current density</td>
<td>Acm&lt;sup&gt;−2&lt;/sup&gt;</td>
</tr>
<tr>
<td>hCurrentDensity</td>
<td>Hole current density</td>
<td>cm&lt;sup&gt;−3&lt;/sup&gt;</td>
</tr>
<tr>
<td>IonizedDonors</td>
<td>Ionized donor density</td>
<td>cm&lt;sup&gt;−3&lt;/sup&gt;</td>
</tr>
<tr>
<td>IonizedAcceptors</td>
<td>Ionized acceptor density</td>
<td>cm&lt;sup&gt;−3&lt;/sup&gt;</td>
</tr>
<tr>
<td>IonizedElectronTraps</td>
<td>Ionized electron trap density</td>
<td>cm&lt;sup&gt;−3&lt;/sup&gt;</td>
</tr>
<tr>
<td>IonizedHoleTraps</td>
<td>Ionized hole trap density</td>
<td>cm&lt;sup&gt;−3&lt;/sup&gt;</td>
</tr>
<tr>
<td>eThElPower</td>
<td>Electron thermoelectric power</td>
<td>V K&lt;sup&gt;−1&lt;/sup&gt;</td>
</tr>
<tr>
<td>hThElPower</td>
<td>Hole thermoelectric power</td>
<td>V K&lt;sup&gt;−1&lt;/sup&gt;</td>
</tr>
<tr>
<td>NetRecombination</td>
<td>The net recombination rate for each recombination/generation model and the total rate cm&lt;sup&gt;−3&lt;/sup&gt;s&lt;sup&gt;−1&lt;/sup&gt;</td>
<td>A/cm&lt;sup&gt;3−d&lt;/sup&gt;</td>
</tr>
<tr>
<td>ContactCurrent</td>
<td>The electric current on each contact</td>
<td>A/cm&lt;sup&gt;3−d&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Table 7.4: Solution/Plot variables
### Semiconductor Table

<table>
<thead>
<tr>
<th>keyword</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ec</td>
<td>conduction band edge (eV)</td>
</tr>
<tr>
<td>Ev</td>
<td>valence band edge (eV)</td>
</tr>
<tr>
<td>m_dos_e</td>
<td>conduction band effective DOS mass ($m_e$)</td>
</tr>
<tr>
<td>m_dos_h</td>
<td>valence band effective DOS mass ($m_e$)</td>
</tr>
<tr>
<td>Nc</td>
<td>conduction band effective DOS (cm$^{-3}$)</td>
</tr>
<tr>
<td>Nv</td>
<td>valence band effective DOS (cm$^{-3}$)</td>
</tr>
</tbody>
</table>

Table 7.5: Parameters for the simple semiconductor model

### SRH Table

<table>
<thead>
<tr>
<th>parameter</th>
<th>keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{\text{min}}$</td>
<td>taumin</td>
</tr>
<tr>
<td>$\tau_{\text{max}}$</td>
<td>taumax</td>
</tr>
<tr>
<td>$N_{\text{ref}}$</td>
<td>Nref</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>gamma</td>
</tr>
<tr>
<td>$E^*$</td>
<td>Etrap</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Talpha</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Tcoeff</td>
</tr>
</tbody>
</table>

Table 7.6: SRH material data file parameters

### SRH parameters Table

<table>
<thead>
<tr>
<th>tau_n</th>
<th>tau_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>tau_p</td>
<td>tau_p</td>
</tr>
<tr>
<td>$E^*$</td>
<td>E_t</td>
</tr>
</tbody>
</table>

Table 7.7: SRH input file parameters

---

7.6. Schrödinger/Poisson/Drift-Diffusion calculations
7.7 Example 1: pn diode

The following example shows a minimal Drift-Diffusion module definition for a pn junction.
Module driftdiffusion
{
  name = dd
  #regions = (pside, nside)
  plot = (Ec, Ev, eDensity, hDensity)

  Physics
  {
    recombination srh {}
    mobility doping_dependent {}
  }

  Contact anode
  {
    voltage = $Vd
  }

  Contact cathode
  {
    voltage = 0
  }
}

7.8 Example 2: Mosfet

In this second example we show a 2D simulation of a silicon Mosfet device. The GMSH model (see GMSH Example 2D) is shown in Fig. GMSH model of the Mosfet.

The model consists of a p-doped Si substrate (substrate), two highly n-doped access regions (contact), a thin gate oxide (oxide) and source, gate, drain and back-side contacts.

We want to simulate a set of output characteristics and a transcharacteristic for this Mosfet, using two distinct input files: outputchar.tib and transchar.tib. In these two files we define only the Module sweep and Simulation blocks. The device and model definitions are put into a third file mosfet.tib, which is included in the other two files using the syntax

@include mosfet.tib

The device definition found in mosfet.tib is shown in the following listing:

Device mosfet
{
  meshfile = mosfet.msh
}
Figure 7.1: GMSH model of the Mosfet showing the mesh and the region labels

```plaintext
material = Si

Region substrate
{
    Doping
    {
        density = 1e18
        type = acceptor
    }
}

Region contact
{
    Doping
    {
        density = 5e19
        type = donor
    }
}

Region oxide
{
    material = SiO2
}
```

The material is defined globally in the Device section. For the oxide it has to be overridden in the correspondent Region block.

The following shows the module definition for the Drift-Diffusion simulation:
Module driftdiffusion
{
    #coupling = electrons

    plot = (Ec, Ev, eQFermi, eDensity, eCurrentDensity, eMobility,
           hQFermi, hDensity, hCurrentDensity, hMobility,
           NetRecombination, ElField, ElPotential, ContactCurrents)
}

Solver
{
    type = linesearch

    linear_solver
    {
        method = pconly
        preconditioner = lu
    }
}

Physics
{
    recombination srh {}

    mobility
    {
        type = field_dependent
        low_field_model = doping_dependent
    }
}

Contact gate
{
    type = schottky
    barrier_height = 3.0

    voltage = $Vg

    area_factor = 0.1
}

Contact source
{
    voltage = 0.0
    area_factor = 0.1
}
Contact backcontact
{
    voltage = 0.0
    area_factor = 0.1
}

Contact drain
{
    voltage = $V_d
    area_factor = 0.1
}

The commented option coupling = electrons shows how a unipolar simulation can be set up. This example however will be simulated in bipolar mode. The Solver defines options for the nonlinear solver (the shown options are the default ones). In this case, a linesearch approach is used to refine the nonlinear Newton steps. The linear solver for each Newton step (defined in the linear_solver block) uses a complete LU factorisation as preconditioner (preconditioner = lu). In this case, the linear solver method can be chosen to be the application of the preconditioner only (method = pconly), instead of using an iterative approach.

The Physics block contains the definition of a few physical models to be used. For the particle density we use Fermi-Dirac statistics, which is the default. We use Shockley-Read-Hall recombination since we solve for both electrons and holes. For the mobility, we use a field-dependent model instead of the default constant mobility model. The low-field mobility is chosen to be calculated from a doping dependent model (which is the default).

The contacts are defined in the Contact blocks. For the gate we specify schottky as type (the default is ohmic contact), providing a suitable barrier height. The area_factor = 0.1 indicates that we assume a transistor with 1 mm gate width.

Next, we create a file transchar.tib containing the definitions for the simulation of the transcharacteristic, as given in the following listing:

@include mosfet.tib

Module sweep
{
    name = sweep_drain
    solve = driftdiffusion

    variable = $V_d
    start = 0.0
    stop = 1.0
    steps = 5
}

Module sweep

102 Chapter 7. Drift-diffusion simulation of electrons and holes
On the first row we include the device definition from `mosfet.tib`. Then we define two sweeps `sweep_drain` and `sweep_gate`. The first one will ramp the drain to 1.0 V in 5 steps, without plotting the results. The second one will then perform a sweep on the gate voltage from -0.5 V to 1.5 V, plotting the results after each step and producing the transfer characteristic. The option `max_step = 0.1` limits the maximum voltage step to 0.1 V. This is useful, since the solver has first to reach the initial gate voltage of -0.5 V, starting from 0 V. Using this option it will do this in steps of 0.1 V. In the `Simulation` block we simply have to specify the two sweeps in the correct order.

Running the simulation with

tiberccad transchar.tib

will produce in particular a set of files `*.vtu` for each step of the sweep `sweep_gate`, and the file `sweep_gate_driftdiffusion.dat` containing the voltage-current characteristics for each contact, shown in Fig. *Mosfet transcharacteristic*.

For the simulation of the output characteristics we create a file `outputchar.tib` with the following content:

```
@include mosfet.tib

Module sweep
{
    name = sweep_drain
    solve = driftdiffusion
}
```

7.8. Example 2: Mosfet
variable = $V_d$
start = 0.0
stop = 2.0
steps = 40

plot_data = true
}

Module sweep
{
    name = sweep_gate
    solve = sweep_drain

    variable = $V_g$
    start = 0.0
    stop = 1.5
    steps = 6
}

Simulation
{
    solve = sweep_gate
    resultpath = output_outputchar

    output_format = vtk
}
As before, we include the device definition using the `@include` statement. Then we define a sweep on the drain voltage with name `sweep_drain` and a second sweep `sweep_gate` to sweep the gate voltage. In the latter, we specify `sweep_drain` in the `solve` option, creating thus a nested sweep. For each gate voltage, a sweep over the drain voltage will be performed.

Running the simulation will produce a file for each couple of values ($V_g$, $V_d$), and a file containing the output characteristic for each value of $V_g$. The resulting set of output characteristics is shown in Fig. *Mosfet output characteristics*.

![Mosfet output characteristics](image)

**Figure 7.3:** Mosfet output characteristics
In tiberCAD, it is possible to perform quantum calculations in the framework of Envelope Function Approximation (EFA): eigenstates, eigenfunctions and quantum density of a given system and dispersion of quantum states can be obtained by means of the module:

- **Module efaschroedinger**

The optical properties are calculated by the module

- **Module opticskp**

### 8.1 Module efaschroedinger

The efaschroedinger simulation tool of tiberCAD is developed in order to solve a single-particle *Schrödinger* equation for electrons and holes in a semiconductor crystal. This problem is an eigenvalue problem that is treated as a generalized complex eigenvalue problem

\[
H\psi = ES\psi, \tag{8.1}
\]

where \(H\) and \(S\) are the Hamiltonian and S-matrix, respectively. The EFA calculations are performed by the **Module efaschroedinger**. A typical example is the following

```plaintext
Module efaschroedinger
{
    name = quantum_el
    regions = quantum
    plot = (EigenFunctions, EigenEnergy, EnergyLevels, QuantumDensity)

    poisson_simulation = driftdiffusion # potential from driftdiffusion
    strain_simulation = strain
}
```
number_of_eigenstates = 10

Physics
{
    particle = el
    model = single_band
}

In this example, Schrödinger equation is solved for electrons
particle = el
with a single band model
model = single_band

We calculate 10 eigenstates by specifying:
number_of_eigenstates = 10

The electron quantum density is calculated by default if the keyword QuantumDensity is present in the plot list:
plot = (ProbabilityDensity, EigenEnergy, QuantumDensity)

A similar definition may be used in a Module efaschroedinger for the calculation of hole quantum states
For holes, one can choose, for example, a 6 bands $k \cdot p$ model in this way:
Physics
{
    model = 6x6
}

8.1.1 Module options

The following options are to be defined in the main block of Module efaschroedinger:

poisson_simulation [string] defines the name of the simulation (e.g. driftdiffusion) that can provide electric potential

strain_simulation [string] defines the name of the simulation (e.g. elasticity) that can provide elastic strain
number_of_eigenstates [integer] defines the number of eigenvalues and eigenfunctions to be found.

num_valence_eigenvalues [integer] defines the number of valence band eigenvalues and eigenfunctions to be found (for 8x8 $k \cdot p$ model).

num_conduction_eigenvalues [integer] defines the number of conduction band eigenvalues and eigenfunctions to be found (for 8x8 $k \cdot p$ model).

regions [string] defines the regions associated to this EFA simulation

8.1.2 Solver section

The Solver section of the Module efaschroedinger contains the following options:

Dirichlet_bc_everywhere [boolean] if true (default value), Dirichlet boundary conditions are imposed over all the boundaries of the simulation region

solver [string] defines the solver for the eigenvalue problem, possible values are: arnoldi, lapack, krylovshur. The default value is krylovshur.

In the case of the lapack solver all the eigenvalues are computed. In the case of arnoldi or krylovshur solver it is necessary to specify which and how many eigenvalues have to be computed. The idea is that the iterative solver calculates several eigenvalues that are close to a specific number, referred to as the guess.

max_iteration_number [integer] maximum number of iteration, used as a stop condition

eigen_solver_tolerance [double] numerical eigensolver tolerance used as a convergence criterium

guess [double] the algorithm try to find the closest eigenvalues to this absolute value of energy (eV). If not defined, then by default it is calculated internally based on the band edges.

ksp_type [string] Krylov subspace method type: bcgs, gmres, cg

pc_type [string] preconditioner type: cholesky, jacobi, ilu, composite.

8.1.3 Physics section

particle [string] defines for which particle (electron or hole) Schrödinger equation is solved with a single band model. Possible values are el and hl. A different Module efaschroedinger has to be defined for each particle to be solved with a single band model.

model [string] possible values are 6x6 for a 6 bands $k \cdot p$ model, 8x8 for a 8 bands $k \cdot p$ model, single_band for a single band model ($\Gamma$ point).
In case *single_band* model is applied to electrons, the relevant mass is read from the material file. When it is applied to holes, in *Physics* section the following options have to be defined:

- **effective_mass** [double(units of $m_0$)] effective mass to be used in *single_band* calculations for holes
- **band_edge** [double] optional override of band edge energy defined in the material file ($E_v$)

In the following some examples of use.

For a single **conduction** band calculation for electrons:

```plaintext
Module efaschroedinger
{
  name = quantum_electrons
  regions = Quantum_1

  ............

  number_of_eigenstates = 6

  Physics
  {
    particle = el
    model = single_band
  }
}
```

In this case the total number of calculated eigenvalues is given by $number_of_eigenstates = 6$.

For a single **valence** band calculation, *Physics* block becomes:

```plaintext
Physics
{
  particle = hl
  model = single_band
  effective_mass = ...
}
```

Note that the **effective_mass** input value is compulsory.

For a 6x6 $k \cdot p$ model:

```plaintext
Module efaschroedinger
{
  name = quantum_holes
  ........
```
number_of_eigenstates = 12

Physics
{
  model = 6x6
}

In this case the total number of calculated (valence band) eigenvalues is given by `number_of_eigenstates = 12`.

For a 8x8 \( k \cdot p \) model:

Module efaschroedinger
{
  name = quantum_8x8
  ........

  num_valence_eigenvalues = 12
  num_conduction_eigenvalues = 6

  Physics
  {
    model = 8x8
  }
}

Note that in this case it is possible to define separately the number of calculated eigenvalues for valence band and conduction band respectively, through the keywords `num_valence_eigenvalues` and `num_conduction_eigenvalues`. In this example the number of calculated valence band eigenvalues is 12 and the number of calculated conduction band eigenvalues is 6.

In case `number_of_eigenstates` is used instead, e.g.

`number_of_eigenstates = 10`

the total number of calculated eigenstates will be 20, 10 in CB and 10 in VB.

### 8.1.4 Quantum density calculation

The particle (electron, hole) **quantum density** is calculated by default if the keyword `QuantumDensity` is present in the plot list:

`plot = (ProbabilityDensity, EigenEnergy, QuantumDensity)`

The quantum density is calculated analytically in the following way. For each eigenstate we calculate the effective mass assuming quadratic dispersion. Then the charge density is calculated as
follows:

\[
\rho_{1D}(r) = g \frac{mkT}{2\pi\hbar^2} |\psi(r)|^2 \ln \left( 1 + \exp \left( \pm \frac{\mu - E}{kT} \right) \right) \tag{8.2}
\]

\[
\rho_{2D}(r) = g |\psi(r)|^2 \frac{1}{2} \sqrt{\left( \frac{mkT}{2\pi\hbar^2} \right) F_{-1/2} \left( \pm \frac{\mu - E}{kT} \right)}, \tag{8.3}
\]

where \( \rho_{1D} \) and \( \rho_{2D} \) are the 1D and 2D charge densities; \( m \) is the averaged mass (the mass is different for each quantized state and is position independent); \( g \) is the degeneracy of the states. The + sign is for electrons, the − sign is for holes.

### 8.1.5 Calculation of quantum dispersion

Within the Module `efaschroedinger` it is possible to calculate the dependence of quantum eigenstates on \( k \)-vector. Such dependence gives the quantum state dispersion. To calculate the quantum dispersion we need to define the block `Dispersion`

```plaintext
Dispersion
{
   k-path = G-K-M
   number_of_nodes = 10
   k_max = 1
}
```

The dispersion of quantum states is calculated at k-points that are nodes of a mesh in k-space.

The main parameters are:

- **k-path**: path of the direction in k-space; it must be a string of the kind G-K-M
- **number_of_nodes**: number of nodes in the mesh along each direction
- **k_max**: maximum value of \( k \) (in units of 1/nm) in each direction. Can be specified as vector in order to have different extensions in different directions.

The dispersion can be calculated in general in a k-space dimension between 1 and \((3-\text{simdim})\), where \( \text{simdim} \) is the simulation dimension. That is a 1D k-space can be defined for 1D and 2D simulations and a 2D k-space for 1D simulations. Accordingly, **number_of_nodes** is an array of dimension equal to the k-space dimension.

For example, in a simulation of a quantum well (1D), to create a 2D k-space one defines a 2D mesh by writing e.g.

```plaintext
number_of_nodes = (5,5)
```

By default, the length in all the directions in k-space is given by **k_max**. E.g., in a 2D k-space, dispersion is calculated on a grid given by y-axis and z-axis between 0 and **k_max**.

If the optional keyword
k-path

is present, then the dispersion is calculated in 1D along a defined path P1-P2-P3, for example G-K-M. Available symmetry points are G, K, M, M'. In this case, number_of_nodes is an integer. The dispersion is calculated for each of the quantum states previously defined in the simulation.

8.1.6 Output

The available output variables for Module efaschroedinger, to be specified in the plot option, are the following:

- **EigenEnergy**: List of Eigenenergies in eV
- **ProbabilityDensity**: square modules $|\psi(r)|^2$ of the eigenstate wavefunctions
- **QuantumDensity**: if present, quantum density is calculated
- **k-space_dispersion**: output of the quantum dispersion
- **Occupation**: probability to find the state occupied. It is calculated assuming Fermi distribution and mean electrochemical potential and temperature:

$$\bar{\mu} = \langle \psi | \mu(r) | \psi \rangle$$  \hspace{1cm} (8.4)

$$\bar{T} = \langle \psi | T(r) | \psi \rangle$$  \hspace{1cm} (8.5)

If **ProbabilityDensity** is specified as plot variable, then **EigenEnergy** will plot the levels of the states as constant values on the simulation mesh in addition to the textual file listing all energies.

8.2 Module opticskp

By defining the Module opticskp, calculation of optical properties is enabled; in particular, the optical kp matrix elements are calculated from the quantum models specified in the Module.

The optical spectrum from spontaneous emission is calculated in the following way

$$P(h\omega) = \sum_{i,j} \frac{1}{2\pi^2} \frac{\omega_{ij}^2 e^2}{m^2 c^3} |M_{ij}|^2 f_i(E_i)(1 - f_j(E_j)) \frac{\Gamma/2}{(h\omega_{ij} - h\omega)^2 + (\Gamma/2)^2} d\Omega,$$  \hspace{1cm} (8.6)

where $f_i$ and $f_j$ are the Fermi distributions and $M_{i,j}$ is the optical matrix element between the states $i$ and $j$.

**opticskp** also calculates the stimulated emission and gain spectra.
Module opticskp
{
  name = optics
  regions = quantum
  plot = (optical_spectrum_k_0 )
  initial_state_model = quantum_el
  final_state_model = quantum_hl

  #initial_eigenstates = (0, 9)
  #final_eigenstates = (0, 15)

  polarization = (0, 0, 1)
  Emin = 2.8
  Emax = 3.6
  dE = 0.001
}

The main parameters are:

  initial_state_model: quantum simulations associated to the initial state of the optical transition

  final_state_model: quantum simulations associated to the final state of optical transition

  Emin: minimum energy

  Emax: max energy

  dE: energy step

  polarization [vector] light polarization

  plot: to select spectrum in $k=0$ ($\Gamma$ point) or spectrum integrated in k-space

Here, initial_state_model and final_state_model are, respectively, the quantum simulations (efaschroedinger module) associated respectively to the initial state of the optical transition (e.g. electron), and to the final state of optical transition (e.g. hole).

By default, all the eigenstates calculated in the electron and hole quantum simulations are taken in account for the optical calculations.

Optionally, one can define the range of states to be used for optical transitions, by means of the keywords initial_eigenstates and final_eigenstates.

A range of energy values must be defined for the optical spectrum, in this way:

Emin = 3.0
Emax = 5.0
dE = 0.001

where Emin, Emax and dE are respectively the minimum, maximum and interval energy.
By specifying `optical_spectrum_k_0` in the `plot` statement, the emission recombination optical spectrum for \( k=0 \) (\( \Gamma \) point) is calculated.

### 8.2.1 Integrated spectrum

For 1D and 2D calculations, it is possible to perform an integration of the optical spectrum in \( k\text{-space} \), by specifying `optical_spectrum` in the `plot` statement

\[
\text{plot} = (\text{optical_spectrum})
\]

In this case, one has to define a \textbf{k-integration} block inside \textbf{Module opticskp}, in this way

\[
\text{k-integration}
\]

\{
  
  \text{k_max} = 0.5 \\
  \text{number_of_elements} = (5,5) \\
  \text{quadrature_type} = \text{gaussian} \\
  \text{quadrature_order} = \text{third} \\

  \text{refine_k_space} = \text{false} \\
  \text{refine_fraction} = 0.5 \\
  \text{relative_accuracy} = 0.001

\}

The parameters for the \( k\text{-space} \) integration are the following

- \textbf{k_max}: maximum value of \( k \) (in each direction, if given as vector)
- \textbf{quadrature_type}: type of integration, default is \texttt{gaussian}
- \textbf{quadrature_order}: order of integration, default is \texttt{third}
- \textbf{refine_k_space}: default is \texttt{false}, if \texttt{true}, then \textit{adaptive} \( k \)-mesh refinement is enabled
- \textbf{refine_fraction}: refinement parameter for \textit{adaptive} \( k \)-mesh refinement
- \textbf{relative_accuracy}: refinement tolerance for \textit{adaptive} \( k \)-mesh refinement

If \texttt{refine_k_space = true} , that is adaptive \( k \)-mesh refinement is enabled, all the elements whose error is greater than the value \((1-\text{refine_fraction})^\times \) (maximum error) are going to be refined. In this case, “Error” is just the integrated quantity. The refinement will end when the tolerance defined by \texttt{relative_accuracy} is satisfied.

To run an opticskp simulation, one needs to define it in the `solve` statement, in this way:

\[
\text{Simulation}
\]

\{

\}

**8.2. Module opticskp**
solve = (strain, dd, optics)
}

where \textit{optics} is the name of the defined \textit{opticskp} simulation. Note that, in this way, the quantum (efaschroedinger) simulations needed for \textit{opticskp} are executed but their results are not shown in the output. To plot quantum results you should explicitly define \textit{efaschroedinger} simulations in the \textit{solve} statement, in this way:

Simulation
{
  solve = (strain, dd, quantum_el, quantum_hl, optics)
}

8.2.2 Output

The output variables for optics calculations are:

- \textit{optical_spectrum_k_0}: optical emission spectrum for \textit{k}=0.
- \textit{optical_spectrum}: optical emission spectrum integrated in \textit{k}-space.

\textbf{The two output variables are mutually exclusive.}

8.3 Example

In this 1D example we will see how to calculate quantum properties of a GaAs/InGaAs \textbf{quantum well}. \textit{Schrödinger} equation is solved with a single-band effective mass model for conduction band and with a 6-band \textbf{k.p} model for valence band. Eigenvalues and eigenfunctions are calculated to get energy levels and wavefunctions in the quantum well. Here is the device definition:

Device
{
  meshfile = InGaAs_1D.msh

  # mesh is drawn in nm
  mesh_units = 1e-9

  Region buffer
  {
    material = GaAs
  }

  # some other parameters...
Region barrier_1
{
    mesh_regions = (barrier1_cl, barrier1_q)
    material = GaAs
}

Region QWell
{
    material = InGaAs
    x = 0.40
}

Region barrier_2
{
    mesh_regions = (barrier2_q, barrier2_cl)
    material = GaAs
}

The InGaAs well region (QWell) and the two barrier regions are collected in the Cluster Quantum_1. Quantum calculation will be restricted to this Cluster

Cluster Quantum_1
{
    regions = (barrier1_q, QWell, barrier2_q)
}

First, a strain calculation for the GaAs/InGaAs/GaAs heterostructure, with GaAs as a reference substrate, is performed.

Module elasticity
{
    name = strain
    regions = all
    plot = (Strain)
Physics
{
  body_force lattice_mismatch
  
  reference_material = GaAs
  structure = zb

}

Contact cathode
{type = clamp}

See _Elasticity_ for a detailed explanation.

Simulation is performed at equilibrium, so that we solve _driftdiffusion_ (see _Drift-diffusion simulation of electrons and holes_) with the option
coupling = poisson

which means that only Poisson equation will be solved.

Module driftdiffusion
{
  ............
  coupling = poisson

  Physics
  {

    band_properties {
      density_of_states bulk_kp
      
      strain_simulation = strain
    }

  }

  polarization pyro {} 
  polarization piezo { strain_simulation = strain }

  recombination srh {}
  recombination direct {}

}
Also, we define a **density_of_states** model of type *bulk kp* to obtain band parameters. The elasticity simulation is used to obtain strain tensor necessary to calculate correctly band parameters: 

\[
\text{strain\_simulation = strain.}
\]

Finally, we define the model *efaschroedinger*, for quantum effective mass calculations. We are going to study quantized states of electrons and holes in the quantum well. Since the structure is 1D, each eigenstate is characterized by the energy level number *n* and the *k vector* that is perpendicular to the growth direction. In this example, we define two simulations that solve *Schrödinger* equation for a single *k-vector* (\( k = 0 \)), one for electrons and one for holes. For electrons

Module *efaschroedinger*

{

    name = quantum_electrons
    regions = Quantum_1

    plot = (ProbabilityDensity, EigenEnergy)

    poisson_simulation = driftdiffusion  # potential from driftdiffusion
    strain_simulation = strain

    number_of_eigenstates = 6

    Physics
    {
        particle = el
        model = single_band
    }

}

Note that we must define *poisson* and *strain* simulation from which potential and strain data have to be taken

poisson_simulation = driftdiffusion  # potential from driftdiffusion
strain_simulation = strain

Electron eigenstates will be obtained from a single band model, so we define

Physics
{
    particle = el
    model = single_band
}

In a similar way, for holes

8.3. Example
Module efaschroedinger
{
    name = quantum_holes
    regions = Quantum_1

    plot = (ProbabilityDensity, EigenEnergy)

    poisson_simulation = driftdiffusion  # potential from driftdiffusion
    strain_simulation = strain

    number_of_eigenstates = 12

    Physics
    {
        model = 6x6
    }
}

where, in this case, we use a 6x6 $k \cdot p$ model to calculate 12 hole eigenstates

Physics
{
    model = 6x6
}

Finally, in the block Simulation, we state the solve order.

solve = (strain, driftdiffusion, quantum_electrons, quantum_holes)

We need to compute strain and driftdiffusion modules first, to get the correct parameters for $k \cdot p$ quantum calculations

Thus, the block Simulation reads

Simulation
{
    dimension = 1
    temperature = 300
    solve = (strain, driftdiffusion, quantum_electrons, quantum_holes)
    resultpath = output
    output_format = grace
}
Output of simulation is shown in the following.

First we look at the output of drift-diffusion calculation. In Fig. *Conduction and valence bands* we have the conduction and valence band profiles in equilibrium condition (contained in the output file *driftdiffusion_msh.dat*); the Fermi level is in correspondance of the zero of energy.

![Figure 8.1: Conduction and valence bands](image)

Then in Fig. *Conduction band levels* we show the conduction band profile and the the quantized levels for electrons (from the file *quantum_electrons_msh.dat*) in the InGaAs quantum well; for this 8 nm-wide quantum well only the first two energy levels are confined.

The first 12 quantized levels for the holes (from the file *quantum_holes_msh.dat*) in the InGaAs quantum well are shown in Fig. *Valence band levels* (there are 6 couples of degenerated energy levels).

Fig. *CB wavefunctions* show the wavefunctions (square module) for the energy states in the conduction band: only the first two states are confined in the conduction band and are shown here.

Finally, in Fig. *VB wavefunctions* there are the wavefunctions for the first 10 confined energy states in the valence band. The first of each couple of degenerated states is shown. States from the heavy hole and the light hole bands are visible.
Figure 8.2: Conduction band levels

Figure 8.3: Valence band levels
Figure 8.4: CB wavefunctions

Figure 8.5: VB wavefunctions
9.1 Introduction

The basic structure of a DSC is the following:

1. A photoanode made of a mesoporous semiconductor medium $TiO_2$ surrounded by a liquid electrolyte.

2. A region of liquid electrolyte only in contact with a counter-electrode (cathode) covered by a thin platinum layer.

The surface of the semiconductor is covered by a mono-layer of dye molecules which make the oxide photoactive. The mesoporous material is obtained by sintering together nanoparticles of $TiO_2$ (15-20 nm in diameter). The diameter of the nanoparticles is fundamental to obtain the right porosity of the material, in fact in order to have a high light harvesting from the dye molecules is necessary to have a large effective area that can be obtained only with a porous material, moreover the porosity of the semiconductor allows the electrolyte to be in contact with dye molecules.

The light-to-electricity conversion is the following: a photon is absorbed by a dye, it excites an electron which is transferred into the conduction band of the porous material. The charge transfer is extremely fast, in the order of tens of femtoseconds. Then, the electron percolates inside the porous material until it reaches the anode contact: a transparent conductive oxide (TCO). The ionized dye is regenerated by the electrolyte via a redox process where iodide is oxidized and triiodide is produced. The dye is now ready to absorb another photon. The regeneration consumes also iodide producing triiodide. This generates a concentration gradient that moves iodide toward the porous region and triiodide toward the cathode. Finally, the triiodide ions reach the platinum where they are reduced and transformed back in iodide ions. The cycle of charge carriers is completed and a current has been established inside the cell exploiting the energy of the photon absorbed.

Two important facts must be noted:

1. In the cell there is no permanent chemical transformation;

2. The split of the electron-hole pair created by the photon occurs directly at the dye level, the electron percolates inside the $TiO_2$ while the hole transport is handled by the electrolyte.
This means that charge carriers in DSCs are majority carriers and the recombination is suppressed, it occurs only at the interface between $\ce{TiO2}$ and electrolyte. That is the reason why DSCs achieve reasonable efficiencies despite they are made of very disordered materials.

### 9.2 Theory

For a brief list of literature about Dye Solar cells (DSC) see [Kalyanasundaram]. For a review of the model see [Gagliardi].

The model consists in a set of drift-diffusion equations for the propagation of ions and electrons coupled with Poisson equation:

\[
\nabla \cdot (\mu_i n_e \nabla \phi_e) = (G - R) \tag{9.1}
\]

\[
\nabla \cdot (\mu_\text{I} I - n_\text{I} - n_\text{I} \nabla \phi_\text{I}) = -\frac{3}{2}(G - R) \tag{9.2}
\]

\[
\nabla \cdot (\mu_\text{I3} I_3 - n_\text{I3} \nabla \phi_\text{I3}) = \frac{1}{2}(G - R) \tag{9.3}
\]

\[
\nabla \cdot (\mu_e n_e \nabla \phi_c) = 0, \tag{9.4}
\]

where $\mu_\alpha$ refers to carrier mobilities, $n_\alpha$ to charge concentrations and $\phi_\alpha$ to electrochemical potentials. $R$ is the recombination term and $G$ the generation term due to illumination. In order to take into account the trap density we use a density dependent mobility developed within multi-trapping model:

\[
\mu_e(n_e) = \frac{\mu_0}{\alpha} \left( \frac{n_e}{N_t} \right)^{\frac{1-\alpha}{\alpha}} \tag{9.5}
\]
where $a$ is the trap exponent, $N_t$ the trap density and $\mu_0$ a constant. The energy trap density is assumed to form an exponential tail below the conduction band edge of the semiconductor:

$$g_T(E) = \frac{aN_t}{kT} e^{\frac{aE}{kT}}. \quad (9.6)$$

The Poisson equation to handle the internal electrical potential drop:

$$-\varepsilon \Delta \varphi = q[n_e + N_D^+ - n_{I^-} - n_{I_3^-} - (n_e - \bar{n}_e)], \quad (9.7)$$

where $N_D^+$ is the amount of ionized dyes and it is equal to:

$$N_D^+ = \frac{G}{k_{dye}} \quad (9.8)$$

with $G$ the generation term and $k_{dye}$ the rate constant of dye regeneration. The dielectric constant, $\varepsilon$, of the mesoporous material is a mix of the two dielectric functions for semiconductor and electrolyte. We use the Maxwell-Garnet model where the dielectric function of the mixed medium results as:

$$\varepsilon = \frac{\varepsilon_s \varepsilon_e + 2\varepsilon_s^2 + 2\varepsilon_p \varepsilon_e - 2\varepsilon_s \varepsilon_p}{\varepsilon_e + 2\varepsilon_s - \varepsilon_p \varepsilon_e + \varepsilon_s \varepsilon_p} \quad (9.9)$$

with $\varepsilon_s$ and $\varepsilon_e$ dielectric constants of semiconductor and electrolyte, respectively, and $\varepsilon_p$ porosity of the medium. The recombination term depends largely on the loss mechanisms at the electrolyte/oxide interface [Frank]. From the chemical path we can get a formula for the interface recombination:

$$R = k_0 \left[ \left( \frac{n_e}{\bar{n}_e} \right)^\beta \frac{n_{I_3^-}}{n_{I^-}} - \bar{n}_e \sqrt{\frac{n_{I_3^-} n_{I^-} - \bar{n}_{I_3^-}}{\bar{n}_{I^-}^3}} \right], \quad (9.10)$$
where the electron rate $k_0$ is the recombination rate constant. The recombination rate constant depends on whether traps are present or not within Multi-trapping model. It looks like:

$$k_0 = k_e \left( \frac{N_c}{n_e} \right)^{1-\beta}$$

(9.11)

$$k_0 = k_e \left( \frac{N_t}{n_e} \right)^{1-\beta} \left( \frac{N_t}{N_c} \right)$$

(9.12)

and

(9.13)

where the first equation is related to no traps condition (with $N_c$ the effective conduction band density of the semiconductor and $\beta$ the non linear recombination constant for electrons). The trap assisted recombination takes into account also the effective density of states of the traps ($N_t$).

For the boundary conditions of the model we assume at the photoanode:

- $j_e = -q\nu_e(n_e - \bar{n}_e \exp(V/(k_B T)))$: rate equation for electron collection, $k_B$ Boltzmann constant;
- $\nabla \phi_{I^-} = 0$: no iodide current at the photoanode;
- $\nabla \phi_{I_3^-} = 0$: no triiodide current at the photoanode;
- $\nabla \phi_c = 0$: no cationic current;

at the cathode:

- $\nabla \phi_e = 0$: no electronic current at the cathode;
- $-q\mu_{I^-} n_{I^-} \nabla \phi_{I^-} = \frac{3}{2} j_0 \left( e^{\frac{\phi}{2k_B T}} - e^{\frac{-\phi}{2k_B T}} \right)$: split of the current between the ionic species, the current is evaluated using a Butler-Volmer equation. $j_0$ is the exchange current;
- $-q\mu_{I_3^-} n_{I_3^-} \nabla \phi_{I_3^-} = -\frac{1}{2} j_0 \left( e^{\frac{\phi}{2k_B T}} - e^{\frac{-\phi}{2k_B T}} \right)$: split of the current between the ionic species, the current is evaluated using a Butler-Volmer equation. $j_0$ is the exchange current;
- $\nabla \phi_c = 0$: no cationic current;

integral boundary conditions for conservation of ionic species:

- $\int_{\Omega} \left[ \frac{1}{3} n_{I^-}(r) + n_{I_3^-}(r) \right] dr = \left( \frac{1}{3} \bar{n}_{I^-} + \bar{n}_{I_3^-} \right) \Omega$: conservation of iodine ions within the cell;
- $\int_{\Omega} n_c(r) dr = \bar{n}_c \Omega$: conservation of cation within the cell;

where $\Omega$ is the volume of the cell, $n_\alpha$ the density of charged species and the index $\alpha$ stands for cation (c), iodide ($I^-$), triiodide ($I_3^-$) and electrons (e).

$E_{red}$ is the redox potential. The redox potential is evaluated within Nernst approximation:

$$E_{red} = E_{Pt}^0 - \frac{kT}{2} \ln \left( \frac{n_{I^-}/n_{St}}{(n_{I^-}/n_{St})^3} \right).$$

(9.14)
9.3 Device

The Device section for a DSC simulation is defined as following:

```plaintext
# Description of the device physical regions
Device
{
    meshfile = <name_of_the_mesh>
    Region TiO2
    {
        material = TiO2mes
        porosity = 0.5
    }
    Region electrolyte
    {
        material = TiO2mes
        TiO2 = false
    }
}
```

The Device section must contain the name of the mesh file `meshfile = <name_of_the_mesh>`. As usual, to each region of the device we associate a material file from the database. In this case, for both regions we define the same material `TiO2mes`, which is a special material which contains standard parameters for both $TiO_2$ and electrolyte. Then, for every region we must specify if it contains $TiO_2$ or electrolyte or both. This can be done setting two flags called `TiO2` and `electrolyte`. When one of these two keywords is set to `true`, the corresponding material is present in the region, otherwise it is not present. By default they are assumed both `true` (which defines a porous region). In the second region of the example shown here we want only electrolyte and thus `TiO2 = false` is explicitly specified. In case both materials are present (porous region) like in Region `TiO2` above, a value for `porosity` must be defined, through the keyword `porosity` (in the range between 0, $TiO_2$ only, and 1, electrolyte only). If one of the two is not present the porosity is automatically set to 0 or 1.

9.4 Module DSC

The simulation of DSCs is performed by the Module `dssc`:

```plaintext
Module dssc
{
    name = dssc
    plot = (ElPotential, eDensity, CurrentDensity, eMobility, ContactCurrents)
    Solver linesearch
    {
        max_iterations = 300
        step_tolerance = 1e-5
        linear_solver
    }
}
```

9.3 Device
Beyond the name of the simulation and the list of plotted variables, this section contains information about the Contacts, parameters for the Solver and the Physics sections.

### 9.4.1 Contacts

Information for the contacts are inserted in the Module section, in the Contact subsections. The two contacts are the photoanode and the cathode. The photoanode must be part of the boundary of a region where $TiO_2$ is present, on the contrary the cathode must be part of the boundary of a region where the electrolyte is present.

Contact anode
{
  type = ohmic
  bias = $V[0.0]
  kinetic_rate = 1e3
}

Contact cathode
{
  type = Pt
  Ex_curr = 0.1
}

The anode is modeled as a contact of type ohmic, with an electron collection rate given by kinetic_rate. The default value of kinetic_rate is $10^6$. Usually a sweep over the bias is performed (considering the cathode as the reference contact). The associated variable is indicated as $V$ above. The cathode is a contact of the special type Pt (platinum): it is modeled as a Butler-Volmer equation, the only needed parameter is the exchange current $j_0$, expressed in $A/cm^2$ ($Ex_{curr} = 0.1$ in the example above).
9.4.2 Physics

In this section one can define the physical parameters enlisted in table Physical parameters. In this case they are valid for the whole device. Alternatively, one can set one or more of these parameters in one of the Regions of the Device, to limit its validity to that Region. For example, the trap effective density of states is given by $N_t$, which by default is $N_t = N_c$, while recombination constant rate is given by $k_e$. If we are simulating a device under illumination the Physics section must contain at least the setting of the generation module, given by

\[
generation = \text{dssc\_generation}
\]

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Poisson equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>porosity</td>
<td>Porosity of porous material</td>
</tr>
<tr>
<td>perm_oxide</td>
<td>Relative perm. of the TiO$_2$</td>
</tr>
<tr>
<td>perm_electrolyte</td>
<td>Relative perm. of the electrolyte</td>
</tr>
<tr>
<td>k_dye</td>
<td>Oxidized Dye regeneration rate constant $k$</td>
</tr>
<tr>
<td>trap_exp</td>
<td>Trap exponential tail $a$</td>
</tr>
<tr>
<td>trap_DOS</td>
<td>Trap effective density of states $N_t$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Drift Diffusion equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ne</td>
<td>Dark electron density $n$</td>
</tr>
<tr>
<td>nI</td>
<td>Dark iodide density $n$</td>
</tr>
<tr>
<td>nI3</td>
<td>Dark electron density $n$</td>
</tr>
<tr>
<td>mu_e</td>
<td>Electron mobility $\mu$</td>
</tr>
<tr>
<td>D_I</td>
<td>Iodide diffusion constant $D_I$</td>
</tr>
<tr>
<td>D_I3</td>
<td>Triiodide diffusion constant $D_{I3}$</td>
</tr>
<tr>
<td>D_C</td>
<td>Cation diffusion constant $D_C$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Recombination term</th>
</tr>
</thead>
<tbody>
<tr>
<td>k_e</td>
<td>Recombination rate $k$</td>
</tr>
<tr>
<td>rec_non_linearity</td>
<td>Non-linearity exponent in electron density recombination $\beta$</td>
</tr>
</tbody>
</table>

Table 9.1: Physical parameters that can be set for the model divided in subsets relative to different processes in the cell.

9.4.3 Generation module

If the simulated device is under illumination, we have to define the generation term. The generation term is related to the flux of photons which reaches the active TiO$_2$ regions and the dye present in the cell. We assume a simple Lambert-Beer exponential decay for charge generation of the form:

\[
G = \int \alpha(\lambda)\Phi(\lambda)e^{-\alpha(\lambda)x}d\lambda, \quad (9.15)
\]
where $\alpha$ is the absorption coefficient (in $\mu m^{-1}$) of the chosen Dye, $\Phi(\lambda)$ the intensity of the light at wavelength $\lambda$ of the light source.

The parameters for the generation can be defined in the auxiliary Module \textit{dssc\_generation}, defined as following:

```plaintext
Module dssc\_generation
{
  regions = TiO2
  plot = (Distance, Generation)
  light\_direction = (1, 0, 0)
  light\_intensity = $x$
  dye = N719

  Contact anode
  {
  }
}
```

In the \textit{dssc\_generation} module, the following keywords must be specified:

- \textbf{regions}: the regions where we want the generation to take place (where the Dye is present);
- \textbf{plot}: if we want to plot the generation within the cell during generation sweep;
- \textbf{light\_direction}: the vector which fixes the direction from where the light comes;
- \textbf{light\_intensity}: the light intensity, in units of Sun;
- \textbf{dye}: the dye used in the cell (file name);
- \textbf{illuminati\_spectrum}: the source spectrum (file name), by default a 1.5 AM solar spectrum;

The light intensity is usually defined as a variable used in a sweep (see below). It can be set to increasing values until it reaches 1, that means one Sun, or a larger or smaller illumination intensity (0.1, 2.0, etc.). The keyword \textit{illuminati\_spectrum} defines the spectrum of the source. The default file spectrum is the standard 1.5 AM sun spectrum contained in the material database in the file \texttt{Sun1p5am}.

The \textit{dye} keyword defines the absorption spectrum of the used dye. Different spectrum files can be used, by adding them in the material folder: the first line of the file contains the \textit{energy gap} between the HOMO and the LUMO of the dye in eV, the second part of the file is the absorption intensity in $\mu m^{-1}$ in the interval 380-800 nanometers.

The last part of the \textit{Module dssc\_generation} is the definition of the \textit{Contact} boundary. This contact defines the physical region taken as the illuminated external surface.
9.5 Sweep

Two sweeps are needed for the calculation of the entire I-V characteristic under illumination.

Module sweep
{
    name = sweep_gen
    solve = (dssc_generation, dssc)
    variable = $x
    values = (0, 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 0.1, 1)
    plot_data = true
}

Module sweep
{
    name = sweep_V
    solve = dssc
    variable = $V
    values = (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.62, 0.64, 0.66, 0.68, 0.7, 0.72, 0.74, 0.76, 0.78, 0.8)
    plot_data = true
}

The first sweep, (sweep_gen), is needed to perform the transition from dark condition to full short-circuit condition under illumination. Then a second sweep, the voltage sweep (sweep_v), computes the I-V characteristic under illumination. In case of dark simulation (application of an external bias without illumination) the first sweep is not needed. The intensity of illumination can be changed by defining a final value for $x$ different from 1 (1 = 1 Sun of power).

9.6 Output

The output variables to be plotted should be enlisted in the plot statement in Module dssc. The available variables are reported in tables Nodal quantities, Elemental quantities and Scalar quantities.

9.7 Example

In this Tutorial we will show how to model and simulate a Dye Sensitized Solar Cell (DSC).

To run this tutorial the following files should be present in the working directory:

DSC.tib: input file (physical parameters of the device).
### Nodal quantities

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>eQFermi</td>
<td>Electron electrochemical potential</td>
<td>$eV (-e\phi_n)$</td>
</tr>
<tr>
<td>IQFermi</td>
<td>Iodide electrochemical potential</td>
<td>$eV (-e\phi_I)$</td>
</tr>
<tr>
<td>I3QFermi</td>
<td>Triiodide electrochemical potential</td>
<td>$eV (-e\phi_{I_3})$</td>
</tr>
<tr>
<td>CQFermi</td>
<td>Cation electrochemical potential</td>
<td>$eV (-e\phi_C)$</td>
</tr>
<tr>
<td>ElPotential</td>
<td>Electrostatic potential</td>
<td>$eV (\phi)$</td>
</tr>
<tr>
<td>Eredox</td>
<td>Electrolyte electrochemical potential</td>
<td>$eV (-e\phi_{Red})$</td>
</tr>
<tr>
<td>eDensity</td>
<td>Electron density (it includes trapped electrons)</td>
<td>$\text{cm}^{-3}$</td>
</tr>
<tr>
<td>IDensity</td>
<td>Iodide density</td>
<td>$\text{cm}^{-3}$</td>
</tr>
<tr>
<td>I3Density</td>
<td>Triiodide density</td>
<td>$\text{cm}^{-3}$</td>
</tr>
<tr>
<td>CDensity</td>
<td>Cation density</td>
<td>$\text{cm}^{-3}$</td>
</tr>
<tr>
<td>Generation</td>
<td>The net electron generation rate</td>
<td>$\text{cm}^{-3}\text{s}^{-1}$</td>
</tr>
<tr>
<td>NetRecombination</td>
<td>The net recombination rate</td>
<td>$\text{cm}^{-3}\text{s}^{-1}$</td>
</tr>
<tr>
<td>eMobility</td>
<td>Electron mobility (including density dependent mob. for multi trap.)</td>
<td>$\text{cm}^2\text{s}^{-1}\text{V}^{-1}$</td>
</tr>
</tbody>
</table>

Table 9.2: Nodal quantities. The flags Potential and Density allow to plot all the electrochemical potentials and all the densities, including generation and recombination.

### Elemental quantities

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElField</td>
<td>Electric Field</td>
<td>$\text{V cm}^{-1}$</td>
</tr>
<tr>
<td>CurrentDensity</td>
<td>Total current density</td>
<td>$\text{A cm}^{-2}$</td>
</tr>
<tr>
<td>eCurrentDensity</td>
<td>Electron current density</td>
<td>$\text{A cm}^{-2}$</td>
</tr>
<tr>
<td>ICurrentDensity</td>
<td>Iodide current density</td>
<td>$\text{A cm}^{-2}$</td>
</tr>
<tr>
<td>I3CurrentDensity</td>
<td>Triiodide current density</td>
<td>$\text{A cm}^{-2}$</td>
</tr>
<tr>
<td>CCurrentDensity</td>
<td>Cation current density</td>
<td>$\text{A cm}^{-2}$</td>
</tr>
</tbody>
</table>

Table 9.3: Elemental quantities. The flag Current allows to plot all of them in the x, y and z components.

### Scalar quantities

<table>
<thead>
<tr>
<th>ContactCurrents</th>
<th>Contact currents</th>
<th>*$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*\$a\$: depends on dimension.

Table 9.4: Scalar quantities.

**DSC.msh**: mesh generated by GMSH from the script (geometry of the device).

**DSC.geo**: script for GMSH.
9.7.1 Device structure

In this tutorial we will see a simple 1D model, made of two physical regions:

1. TiO$_2$ region, which is the mesoporous medium where the light absorption and recombination take place.

2. Electrolyte region.

Two boundary condition points represent the photoanode (left) and the counter-electrode (right).

![Diagram of the 1D device](image)

Figure 9.2: The scheme of the 1D device.

Definition of dsc.tib:

```
# Description of the device physical regions
Device
{
  meshfile = DSC.msh

  Region TiO2
  {
    material = TiO2mes
    porosity = 0.5
  }

  Region electrolyte
  {
    material = TiO2mes
    TiO2 = false
  }
}
```
For both region (TiO2 and electrolyte) the same material for the database is used (TiO2mes), the parameter porosity defines which porosity has the material. The flag TiO2 = false defines instead that in the electrolyte region only the electrolyte is present and there is no semiconductor. By default the dssc module assumes that a certain region contains both the semiconductor and the electrolyte (porous material).

9.7.2 Definition of Simulation Modules and Boundary Conditions

```
# Definition of Simulation Models and associated Boundary Conditions
Module dssc
{
  name = dssc
  plot = (Potential, Density, Current, Mobility, ContactCurrents)

  Solver linesearch
  {
    max_iterations = 300
    step_tolerance = 1e-4
    linear_solver
    {
      preconditioner = lu
    }
  }

  Physics
  {
    generation = dssc_generation
  }

  Contact anode
  {
    type = ohmic
    bias = $V[0.0]
    kinetic_rate = 1e3
  }

  Contact cathode
  {
    type = Pt
    Ex_curr = 0.1
  }
```
The module \texttt{dssc} contains the name of the simulation (\texttt{dssc}) and the list of output plotted (see \textit{Output}), the subsections for the \textit{solver}, the \textit{physics} and the \textit{contacts}.

The \textit{physics} subsection contains a keyword to set the \texttt{dssc\_generation module}, since the device is illuminated. The non linear exponent in the electron density recombination $\beta$ is assumed to be 1 (linear case), but can be changed to a different value for non-linear recombination.

We define two \textit{Contacts}: anode (for the photoanode) and cathode (for the counter-electrode). The two contacts are respectively of type \texttt{ohmic} (anode) and \texttt{Pt} (cathode). The external bias is applied at the anode (\textit{bias}), while the counter-electrode (cathode) is modeled as a Butler-Volmer equation.

\begin{verbatim}
Module dssc_generation
{
  regions = TiO2
  light_direction = (1, 0, 0)
  light_intensity = $x
  dye = N719

  Contact anode
  {
  }
}
\end{verbatim}

The \texttt{Module dssc\_generation} defines the photogeneration within the device. The generation model must define which is the region(s) where photogeneration occurs (\texttt{regions}), the intensity of light (\texttt{light\_intensity}), the direction of the light rays (\texttt{light\_direction}), the kind of Dye used in the cell (\texttt{dye}) and the boundary region illuminated (in our example \texttt{anode}).

Another keyword, \texttt{illumination\_spectrum = <source\_spectrum>} can be set if we want to change the solar spectrum for the generation, in case the light source we want to simulate is not the conventional sun spectrum.

By default the flag is set to \texttt{illumination\_spectrum = Sun1p5am}, the conventional sun spectrum 1.5 AM.

\begin{verbatim}
Module sweep
{
  name = sweep_gen
  solve = (dssc\_generation, dssc)
  variable = $x
  values = (0, 1e-12, 1e-11, 1e-10, 1e-9, 1e-8, 1e-7,
            1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 0.1, 1)
  plot_data = true
}
\end{verbatim}

9.7. Example

137
Module sweep
{
    name = sweep_V
    solve = dssc
    variable = $V$
    values = (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.62, 0.64, 0.65, 0.66, 0.67, 0.68, 0.69, 0.7, 0.71, 0.73, 0.75, 0.77, 0.79, 0.8)
    plot_data = true
}

These are the two sweeps needed to solve the cell: the first sweep is needed to push the device to short circuit condition under illumination, the second sweep calculates the IV characteristic under illumination.

# Definition of model-independent parameters of the Simulation
Simulation
{
    searchpath = ../../materials
    solve = (sweep_gen, sweep_V)
    resultpath = output
    output_format = grace
}

The two sweeps are to be executed in the shown order in the solve statement in Simulation section. First, the sweep on the variable \( x \) (sweep_gen) increases the generation until 1 Sun, then the sweep on bias (sweep_V), produces the IV characteristic under illumination.

In a DSC the current is fundamentally diffusion driven, the electric field and the consequent drift current is rather small.

9.7.3 Output

The output of the calculation is set in the Module dssc. It is possible to collect a set of Output variables, as defined in tables Nodal quantities, Elemental quantities and Scalar quantities, with some special keywords.

plot = (Potential, Density, Current, Mobility, ContactCurrents)

Thus, with Potential we plot the internal potential profiles which include electrostatic potential, with Density, all the densities which include electron, iodide, triiodide, cation densities and recombination and generation profiles, with Current all the currents which include the x, y and z components of the currents (electronic, iodide, triiodide and cation, total current) and the elec-
Figure 9.3: Sweeps for generation and applied bias.

The mesh-based output for this 1D simulation can be found in the files dssc_sweepvariable_stepvalue_msh.dat, e.g. dssc_V_0.8_msh.dat.

In Fig. Potential profiles the results for ionic and electrostatic potential in the cell are shown, in short-circuit condition.

In Fig. Current components, the values of the iodide, triiodide and electron current components along the cell are shown.

The mesh-independent output is printed in the files sweep_V_dssc.dat and sweep_gen_dssc.dat. From the first one, the IV characteristic of the cell is shown in Fig. IV char.
Figure 9.4: Potential profiles for the ionic species and the electrostatic potential (short-circuit condition).

Figure 9.5: Current contributions within the cell (short-circuit condition).
Figure 9.6: I-V characteristic of the device.
VALENCE FORCE FIELD

10.1 Theory

The Valence Force Field (VFF) is an empirical method which allows to calculate the total energy of crystalline structures as a sum of two center and three center terms which depend on bond stretching and bond bending. The total potential $U$ is calculated as a sum of atomic contributions $U = \sum_i U_i$ where $i$ is an index running on the atoms and the form of $U_i$ depends on the crystal structure. TiberCAD includes potentials suitable for Zinc Blende and Wurtzite crystals.

In the case of Zinc Blende the original Keating potential is used:

$$U_i = \frac{3\alpha}{16r_0^2} \sum_{j=1}^4 (r_{ij} - r_0^2)^2 + \frac{3\beta}{8r_0^2} \sum_{j=1}^4 \sum_{k>j}^4 (r_{ij} r_{ik} - r_0^2 \cos \theta_0)^2$$

where $\alpha$ and $\beta$ are the bond stretching and bond bending constants, $r_0$ and $\theta_0$ the reference bond length and bond angle of the bulk crystal.

In the case of Wurtzite a modified Keating potential is used. In Zinc Blende each atom is bonded by four identical bonds to four first neighbor atoms placed on the vertices of an ideal tetrahedron. In Wurtzite crystals the nearest neighbors are placed on the vertices of a distorted tetrahedron, such that three bonds are identical but the fourth one has a different length and angle. The non ideality is usually described by an internal parameter $u$.

Therefore in the case of Wurtzite the following potential is used:

$$U_i = \frac{3\alpha}{16r_0^2} \sum_{j=1}^3 (r_{ij} - r_0^2)^2 + \frac{3\alpha'}{16r_0^2} (r_{i4} - r_0'^2)^2 +$$

$$+ \frac{3\beta}{8r_0^2} \sum_{j=1}^3 \sum_{k>j}^3 (r_{ij} r_{ik} - r_0^2 \cos \theta_0)^2 + \frac{3\beta'}{8r_0 r_0'} \sum_{k=1}^3 (r_{i4} r_{ik} - r_0 r_0' \cos \theta_0')^2$$

where different reference bond length and angles are used if bonds along the [0001] crystal direction are considered. In principle, also different coefficients $\alpha$, $\alpha'$ and $\beta$, $\beta'$ can be used. Once the
potential as a function of atomic coordinates is known, the system geometry can be relaxed by minimizing the VFF potential.

The stiffness constants used in continuum elasticity can be written in terms of VFF coefficients as well, for further details refer to the work of Keating [P.N. Keating Phys. Rev., 145 (1966), p. 637] for the Zinc Blende and to the work of Camacho and Niquet [D. Camacho and Y.M.Niquet, Physica E, 42 (2009), p.1361], even though through a set of linearly dependent equations. Therefore it is not possible to define coefficients which correspond to a given set of stiffness constants. If no parameters are provided via input file by the user, tiberCAD will calculate the parameters internally by minimizing the mismatch between the stiffness constants provided in database and the one calculated from the VFF. This allows to ensure the best consistency possible between continuum and atomistic model.

10.2 Module vff

In tiberCAD, it is possible to perform a relaxation of an atomistic structure by means of the Module vff. Here is an example

Module vff
{
    regions = all
    plot = (StrainNodes)

    atomistic_structure = tb
    boundary_conditions = all_around

    Solver
    {
        absolute_tolerance = 1e-4
        print_level = 0
    }
}

A VFF calculation is desirable in combination with other atomistic methods, such as empirical tight binding, as it allows to take into account internal strain.

10.2.1 Module options

The VFF module is controlled by the following options:

**atomistic_structure** [string] Defines the atomistic structure on which the VFF must be calculated. The atomistic structure must have been declared in the appropriate Device section.
**boundary_conditions** [string] Defines if and how some atoms are fixed to simulate different boundary conditions. The available options are *substrate*, *free_standing* and *all_around*. The default is *free_standing*, meaning that all the atoms are allowed to relax. If *all_around* is specified, all the outer atoms are fixed. Each atom which is not bonded to 4 atoms which belong to the same structure, or each atom bonded to a passivation hydrogen, is considered an outer atom. If *substrate* is specified, all the atoms belonging to an initial layer orthogonal to a given direction are fixed. This last keyword is used to simulate epitaxial heterostructures. The behaviour of *substrate* is determined by additional keywords here enlisted.

**substrate_plane** [string] It can take values *x*, *y* or *z*. If *boundary_conditions* is set to *substrate*, it specifies the direction orthogonal to the substrate in the cartesian reference. The atoms with smallest coordinates, within a given tolerance, along the given direction are fixed (i.e., it is assumed that the structure is oriented along positive cartesian direction, with substrate underneath). If *boundary_conditions* is not set to *substrate*, this keyword has no effect.

**substrate_updown** [bool] If set to true, both atoms with smallest and largest coordinates are fixed, to simulate a substrate condition on both the lowest and highest structure plane. This can be useful to combine different relaxation methods. If *boundary_conditions* is not set to *substrate*, this keyword has no effect.

**substrate_tol** [double] Define the thickness of the layer of fixed atoms (in Angstrom). The default is 1.0, which is usually enough to fix the first atomic layer. If *boundary_conditions* is not set to *substrate*, this keyword has no effect.

### 10.2.2 Solver options

The VFF supports a structure optimization through a non-linear conjugate gradient algorithm. The tolerance is expressed as a force in eV/A and the default value is 1e-3. The tolerance can be modified with the keyword *absolute_tolerance*. Example:

```plaintext
Solver
{
  absolute_tolerance = 1e-4
}
```

### 10.2.3 Physics

The Physics section of the input file is used to allow the definition of a *custom* bond stretching parameter $\alpha$ and a *custom* bond bending parameter $\beta$ through special blocks *keating* with the type *user*. As a default, all these parameters are calculated internally. An example of custom parameters is
Physics
{
  keating user
  {
    regions = InN
    alpha = 67.38
    beta = 10.01
  }
  keating user
  {
    regions = GaN
    alpha = 88.35
    beta = 20.92
  }
}

In this example the user explicitly set the model coefficients for all the regions made of Indium Nitride and for all the region made of Gallium Nitride. The coefficients are specified in N/m.

If at least one of the material is an alloy, the user may choose between a random alloy approximation or a virtual crystal approximation (VCA). In a random alloy approximation the two center parameters are assigned according to a reference bulk material which depend on the bond atoms (e.g. a Ga-N bond will get bulk GaN parameters assigned), and three center parameters will be assigned as an average of corresponding bulk terms (e.g. a In-N-Ga bond will have parameters averaged between bulk InN and GaN). In the case of random alloy the user can specify both components as in the following example, where we consider two regions, one associated to alloy InGaN and one associated to GaN.

Physics
{
  keating user
  {
    regions = InGaN
    component InN
    {
      alpha = 67.38
      beta = 10.01
    }
    component GaN
    {
      alpha = 88.35
      beta = 20.92
    }
  }
  keating user
  {
    regions = GaN

Chapter 10. Valence Force Field
For InGaN, parameters for each alloy component, InN and GaN are defined separately through the subblock component, where the keyword component is followed by the name of the alloy component, e.g. InN, as in the following:

component InN
{
    alpha = 67.38
    beta = 10.01
}

In the virtual crystal approximation the coefficients are automatically calculated in order to approximate properly the alloy stiffness coefficient calculated with Vegard law, in order to ensure consistency with continuum models. A user may override these value by specifying the alloy coefficients, as in this example:

Physics
{
    keating user
    {
        regions = InGaN
        alpha = 70.05
        beta = 12.00
    }
    keating user
    {
        regions = GaN
        alpha = 88.35
        beta = 20.92
    }
}

Different coefficients can be defined for different regions just by specifying the correct region names. Note that this settings have to be consistent with the definition random_alloy in the Atomistic Generator.

10.2.4 Output

The available output variables for Module vff, to be specified in the plot option, are the following and correspond to the strain tensor extrapolated from atom positions in the unit cell after relaxation:

**StrainNodes**: strain tensor associated to nodes
**StrainCells:** strain tensor associated to cells

### 10.3 Example

First, the Device structure is defined for a InGaN/GaN quantum well heterostructure

```plaintext
Device
{
    meshfile = well.msh
    mesh_units = 1e-9  #nanometers
    dimension = 1

    structure = wz
    x-growth-direction = (0,0,0,1)
    y-growth-direction = (1,0,-1,0)
    z-growth-direction = (-1,2,-1,0)

    Region QW
    {
        material = InGaN
        x=0.1
    }

    Region barrier_left
    {
        material = GaN
    }

    Region barrier_right
    {
        material = GaN
    }

    Region device_left
    {
        material = GaN
    }

    Region device_right
    {
        material = GaN
    }

    # group together the regions for the quantum simulation
```

# group together the regions for the quantum simulation
Cluster Quantum
{
  regions = (barrier_left, barrier_right, QW)
}

Atomistic tb
{
  reference_region = device_left
  regions = Quantum
  passivation = no
  print = (xyb, xyz, gen, tgn)
}
}

An atomistic block defines the region where an atomistic structure needs to be generated, that is the QW and the barrier regions. Based on the GaN/InGaN/GaN atomistic structure generated, ETB calculations will be performed, so that we define a Module empirical_tb

Module empirical_tb
{
  regions = Quantum
  name = tb
  atomistic_structure = tb
  Harrison_scaling = true

  strain_simulation = strain
  potential_simulation = driftdiffusion

  sparse_format = full

  projection_length = 1.0
  dangling_bond_scaling = 1.0
  dangling_bond_onsite = -5.0

  plot = (tbstates, ProbabilityDensity)
  jmol_output_format = cube

  #assemble_hamiltonian = false

  Solver
  {
    #load_states = true
    #load_path = output3
    num_valence_eigenvalues = 2
    num_conduction_eigenvalues = 2
  }
}
long_tolerance = 1e-6
guess_valence = -1.4
)
)

A **Module** opticstb is defined to calculate optical properties with ETB

Module opticstb
{

name = spectrum
plot = optical_spectrum_k_0

initial_state_model = tb
final_state_model = tb

Emin = 3.0
Emax = 5.0
dE = 0.001

}

A **Module** elasticity is defined to calculate macroscopic strain as a first guess for VFF relaxation

Module elasticity
{

name = strain
mesh_deformation = true

plot = (Strain,Stress,Displacement)

Physics
{

body_force lattice_mismatch
{

reference_material = GaN
x-growth-direction = (0,0,0,1)
y-growth-direction = (1,0,-1,0)
z-growth-direction = (-1,2,-1,0)

}
Contact cathode{type = clamp}

The Module vff is here defined in such a way to be applied to the whole atomistic structure \(tb\), with boundary conditions fixing all the outer atoms (all_around). In output, strain tensor extrapolated from vff relaxation is plotted (StrainNodes)

Module vff
{
    regions = all
    plot = (StrainNodes)

    atomistic_structure = tb
    boundary_conditions = all_around

    Solver
    {
        absolute_tolerance = 1e-4
        print_level = 0
    }
}

Finally, drift-diffusion is defined for an equilibrium calculation of the band profiles

Module driftdiffusion
{

    name = driftdiffusion
    regions = all
    coupling = poisson

    #plot = (Ec, Ev, eQFermi, hQFermi, ElField , eDensity, hDensity, Polarization)
    plot = (Polarization, Ec, Ev, eQFermi, hQFermi, ElField, eDensity, hDensity, ElPotential)

    save_state = true
    #load_state = dd.tsv
    # solve_after_load = true

    Physics
    {

        strain_simulation = strain

        use_density_predictor = true
    }

10.3. Example
background_conductivity = 1e-6

recombination srh {
}

recombination direct {
  C = 1.1e-8
}

mobility {
  type = doping_dependent
  #model = field_dependent
  #low_field_model = doping_dependent
}

#endphysics

Contact cathode {
  type = ohmic
  voltage = $Vb[0.0]
  rec_velocity_p = 1e7
}

Contact anode {
  type = ohmic
  voltage = 0.0
  rec_velocity_n = 1e7
}

#driftdiffusion

We solve first for strain with elasticity, which applies in this case a mesh deformation to the device, accounting for the calculated displacements. Then, based on this deformed mesh, vff is applied, which relaxes the atomistic structure associated to the mesh. Finally, an equilibrium solution is found for potentials and then empirical tight-binding Hamiltonian is calculated on the relaxed atomic structure (tb), to obtain eigenstates and optical properties (spectrum)

Simulation {
  verbose = 4
  temperature = 300
}
solve = (strain, vff, driftdiffusion, tb, spectrum)

resultpath = output
output_format = grace

Note that it is always highly advisable to solve first elasticity, before a VFF relaxation. This will provide a reasonable first guess for the VFF solver and also will apply the correct deformation to the mesh, assuring that atoms are still associated to the correct FEM elements.
11.1 Theory

11.1.1 Tight Binding approach

The Tight binding method (TB) is a technique to describe the electronic and optical properties of a material through an atomistic description. A corpuscular approach is needed in devices with nanometric features, where a detailed description of electronic and optical properties cannot be reached with an effective description.

In the following we will give a brief description of general features of Tight Binding Model. Then we will focus on the technique implemented in TiberCAD, namely Empirical Tight Binding (ETB).

The TB model has been introduced by Slater and Koster in a notable publication in 1954 (see [SlaterKoster]).

The starting point is the LCAO (linear combination of atomic orbitals) representation.

If we assume that the electronic states $\Psi$ in the system under observation are perturbations of the atomic states, it’s convenient to use the LCAO basis:

$$|\Psi\rangle = \sum_{\alpha R} C_{\alpha R} |\alpha, R\rangle$$  \hspace{1cm} (11.1)

Where $R$ is the atom position and $\alpha$ a quantum number to distinguish between different orbitals.

Using the previous relation in the time independent Schrödinger equation, we obtain the expression:

$$\sum_{\alpha' R'} C_{\alpha' R'} \left[ H_{\alpha' R' \alpha R} - E S_{\alpha' R' \alpha R} \right] = 0$$  \hspace{1cm} (11.2)

where
\( H_{\alpha'}R^\alpha R \) and \( S_{\alpha'}R^\alpha R \) are the Hamiltonian matrix element and the overlap matrix and are respectively given by:

\[
H_{n'\alpha',n\alpha} = \langle n'\alpha' | H | n\alpha \rangle \\
S_{n'\alpha',n\alpha} = \langle n'\alpha'|n\alpha \rangle
\]  

(11.3)

The Hamiltonian may possibly include an external potential \( V_{ext} \) term, including an externally applied potential and mean field corrections. The overlap matrix is the identity when the basis is orthogonal. This is not the case in general, since usually atomic orbitals have non zero overlap. However, the basis set can be orthogonalized using a Lowdin orthogonalization. Anyway, resulting wavefunctions have longer range, and it determines a worst numerical formulation than when using a non orthogonal basis. The overlap matrix \( S \) can be calculated from the basis function, but in order to solve the generalized eigenvalue problem the matrix elements of the Hamiltonian need to be evaluated. The method used to obtain these matrix elements classifies the different TB implementations.

The Hamiltonian matrix elements can be classified in four categories (see [DiCarlo review])

- **On-site**: when the atomic wavefunctions and the potential are centered on the same site
- **Two-centre**: when wavefunction and potential are on the same site and the other wavefunctions are on different sites
- **Three-centre**: when the wavefunctions and the potential are all on different sites
- **Local environment correction**: when the two wavefunctions are on the same site and the potential is on a different site

In the following we will consider only on-site and two-centre contributions, a reasonable trade-off between computational effort and accuracy.

### 11.1.2 Empirical Tight Binding

As seen in previous section, the type of matrix element included in a TB representation is one of the discriminants between one model and another. The other features which differentiate TB models are the number of interacting neighbours (usually nearest neighbours or second nearest neighbours), the orthogonality of the basis set and the method used to calculate the matrix elements. Focusing on the last point, there are three main techniques to evaluate these parameters: empirical, semi-empirical and ab initio methods.

In empirical tight binding (ETB), matrix elements are calculated as fitting parameters of characteristic bulk quantities, such as effective mass, energy gap and split-off energy.

In order to solve the Schrodinger equation we need to evaluate two different kinds of matrix element: **on-site** matrix elements and **hopping** matrix elements.

The *empirical tight binding* technique consists in finding these quantities by fitting the bulk properties of semiconductors, assuming an orthogonal basis, i.e. \( S = I \) with \( I \) the identity matrix.
The physical meaning of the on-site term is straight: it’s the eigenenergy of the corresponding orbital. The hopping energy can be written in different notations. In the original work of Slater and Koster (see [SlaterKoster]), hopping elements are given by linear combinations of atomic orbitals in a two centre approximation. Note that Slater and Koster work is based on a Lowdin orthogonalized basis, leading to zero overlap between orbitals. This is a crucial point in ETB technique, as it allows to solve an eigenvalue problem instead of a generalized eigenvalue problem, with a much lower computational effort, and it’s the key for the efficiency of ETB respect to ab initio techniques.

It means that any interatomic parameter is labeled as $V_{\alpha\beta\mu}$, where $\alpha, \beta = (s, p, d)$ are the atomic orbitals involved and $\mu$ is the component of angular momentum around the bond axis, i.e. $\mu = \sigma$, $\mu = (\sigma, \pi)$, $\mu = (\sigma, \pi, \delta)$ respectively for $ss, sp, sd, pp$ and $dd$ bonds. This is the so called molecular orbital notation. The relationship between these quantities and hopping matrix elements (Koster notation) is shown in Slater and Koster work.

The chemistry of localized states in covalent semiconductors needs at least an eight band parameterization (one $s$ and three $p$ orbitals with spin degeneracy) (see [Vogl]).

Historically, the first approach to empirical tight binding has been performed by using these $sp^3$ basis, often expanding the interaction to second nearest neighbours. In fact first-neighbour $sp^3$ models fail in many points: it’s proved that they can not reproduce an indirect gap in diamond and zincblende materials and they cannot fit even the lowest conduction bands of semiconductors as Ge, Si, AlAs and GaP.

However a second-neighbour basis is not desirable as it cannot be used to describe random alloys and heterointerfaces in a straight way, while first neighbour parameterizations are used under the assumption that parameters are portable when we’re not dealing with bulk structures, and locally the neighbours interact as in the corresponding bulk material.

It is possible to overcome these deficiencies by including an excited $s$ state, i.e. the $s^*$ state, on each atom (see [Vogl]). With this approach it’s possible to get a ten band nearest neighour $sp^3s^*$ parameterization which describe with a good degree of accuracy bulk IV group semiconductors, zincblend and wurtzite III-V materials and ternary alloys.

The $sp^3s^*$ parameterization has been used for years and it’s still routinely applied. However, it still suffers some inner limitations. For example, it cannot fit the $X$ valley transverse mass, making its application critical in cases where $X$ valley contribution are important, such as highly confined structures or materials with a band minimum in $X$ like Silicon. There are two ways to overcome these limitation: increasing the number of neighbours or increasing the number of parameters.

For the reasons previously explained, we find more convenient not to discard the nearest neighbour approach. Luckily we can rely on modern parameterizations that include $d$ orbitals, leading to a 30 bands description. The $sp^3s^*d$ parameterization by Jancu (see [Jancu], [Jancuwz]), gives a very accurate description of C, Si, Ge, AlP, InP, GaAs, AlAs, InAs, GaSb, AlSb, InSb, GaN, AlN and InN. This parameterization introduces seven additional hopping matrix elements but, at the cost of an increased computational cost, it offers a very accurate description of nanostructures.
11.2 Module empirical_tb

In tiberCAD, it is possible to perform atomistic quantum calculations in the framework of Empirical Tight Binding (ETB): eigenstates, eigenfunctions and quantum density of a given system can be obtained by solving a tight-binding Hamiltonian by means of the module:

- **Module** empirical_tb

The optical properties are calculated by the module

- **Module** opticstb

A typical example of tight-bind calculations is the following

Module empirical_tb
{

    regions = Quantum
    name = tb
    atomistic_structure = tb
    Harrison_scaling = true

    strain_simulation = strain
    potential_simulation = driftdiffusion

    plot = (tbstates)

    Solver
    {
        #load_states = true
        #load_path = output3

        num_valence_eigenvalues = 2
        num_conduction_eigenvalues = 2
        long_tolerance = 1e-6
        guess_valence = -1.4
    }

}

In the previous example, TB calculations are applied to the physical region Quantum (usually a Cluster)
regions = Quantum

An atomistic structure must be coupled with this physical region, in order to apply the TB model
atomistic_structure = tb

This is made through the **Atomistic generator**, the tool of tibercad which allows to generate an atomic basis associated with the finite element mesh which belongs to a given physical region, based on the material specifications and the growth directions defined for that region.

See section **Atomistic generator** for a detailed description of the Atomistic generator.

### 11.2.1 Module options

The following options influence the behaviour of the **Module** `empirical_tb`:

- **name** [string] defines the name of empirical_tb simulation
- **regions** [string] defines the regions associated to this ETB simulation
- **atomistic_structure** [string] defines the name of the atomistic structure which will be used by ETB to create the TB Hamiltonian. It must be the name of a structure created through an **Atomistic** block in **Device** section.
- **potential_simulation** [string] defines the name of the simulation (e.g. `driftdiffusion`) that can provide electric potential
- **strain_simulation** [string] defines the name of the simulation (e.g. `elasticity`) that can provide elastic strain. The strain tensor can then be used to calculate the scaling of some ETB parameters.
- **Harrison_scaling** [boolean] if true, scaling of ETB parameters is applied (usually in presence of material deformation which causes atom displacement from equilibrium position). This option is by default set to **true** whenever a strain simulation is performed on the system, otherwise is by default **false**.

### 11.2.2 Solver section

The Solver section of the **Module** `empirical_tb` contains the following options:

- **num_valence_eigenvalues** [integer] defines the number of valence band eigenvalues and eigenfunctions to be found.
- **num_conduction_eigenvalues** [integer] defines the number of conduction band eigenvalues and eigenfunctions to be found.
- **load_states** [boolean] if true, a list of quantum states (eigenvalues and eigenfunctions) are loaded from the file specified by `load_path`. If
the number of loaded electron or hole states is lower of respectively
num_conduction_eigenvalues or num_valence_eigenvalues, then the remaining
states are calculated.

**load_path** [string] path of the file from which tb states can be loaded

**guess_conduction** [double] the solver algorithm tries to find the closest eigen-
values to this absolute value of energy (eV). If guess_conduction is not defined,
then by default it is calculated internally based on the band edges.

**guess_valence** [double] the solver algorithm tries to find the closest eigenvalues
to this absolute value of energy (eV). If guess_valence is not defined, then by
default it is calculated internally based on the band edges.

**long_tolerance** [double] solver convergence criterion (default 1e-9)

With the keyword load_states = true it is possible to resume a calculation starting from the last
of the states loaded from a file. Also, one can use a set of already calculated states for an optical
spectrum calculation (see in the following). Just run empirical_tb_simulation to load the states and
then optictb to calculate spectrum based on those states.

### 11.2.3 Output

The available output variables for Module empirical_tb, to be specified in the plot option,
are the following:

**tbstates** : states eigenfuctions in cube format

**MeshStatesNodes** : plots states on the mesh. MeshStatesNodes is not supported
in 1D calculations

**MeshStates** : plots states on the mesh in 1D calculations (e.g. Quantum wells).

By default the list of eigenvalues[eV] is printed in the output directory in the output file simulation_name.dat. This file contains the values of all the calculated electron and hole eigenvalues (EigenEnergy), the Fermi level energy (FermiLevel) and the occupation index (Occupation) of each state.

Through the keyword tbstates, the states eigenfuctions may be stored in a cube file for visualization
of wavefunctions with the open-source viewer jmol (http://jmol.sourceforge.net).

By defining MeshStatesNodes (or MeshStates in 1D), square modules of the eigenstate wave-
functions may be plotted on the FEM mesh through the output file simulation_name_msh.vtu, e.g
tb_msh.vtu.
11.3 Module opticstb

The Module opticstb implements the calculation of optical properties based on the tb states; in general, the optical matrix elements are calculated from the tb models specified in `initial_state_model` and `final_state_model`. In the current version the model must be the same in both cases. For example

```plaintext
initial_state_model = tb2
final_state_model = tb2
```

The main parameters are:

- **initial_state_model**: tb simulation associated to the initial state of the optical transition
- **final_state_model**: tb simulation associated to the final state of optical transition
- **Emin**: minimum energy
- **Emax**: max energy
- **dE**: energy step
- **plot**: `optical_spectrum_k_0` to select optical emission spectrum in \( k=0 \) (\( \Gamma \) point)
  - `matrix_elements` to select calculation of the optical matrix elements

The output file for optical emission spectrum in \( k=0 \) (\( \Gamma \) point) is named `<simulation_name>_spectrum_k_0.dat` and is composed by several columns with the following format

```plaintext
```

The output file for optical matrix elements is named `<simulation_name>.dat` and is composed by 5 columns with the following format

```plaintext
initial_state final_state |Px|^2 |Py|^2 |Pz|^2
```

11.4 Example 1

In this example we will see how to calculate quantum states of a GaN quantum dot with a tight-binding model based on a sp3s*d parameterization. Strain due to the lattice mismatch is taken in account with elasticity calculations and the related atom displacements are projected on the atomic structure. Potential profiles including piezo and pyro polarizations are calculated by solving Poisson equation and projected as well on the TB Hamiltonian.
First, the Device structure is defined

Device qdot
{
  meshfile = box_in_cube.msh
  mesh_units = 1e-9
  dimension = 3

  y-growth-direction = (-1,0,1,0)
  x-growth-direction = (-1,2,-1,0)
  z-growth-direction = (0,0,0,1)
}

Region ball
{
  material = GaN

  Doping {
    Nd = 1e15
    type = donor
    Ed = 0.025
  }
}

Region qbox
{
  material = AlN

  Doping {
    Nd = 1e15
    type = donor
    Ed = 0.025
  }
}

Region intrinsic
{
  material = AlN

  Doping {
    Nd = 1e16
    type = donor
    Ed = 0.025
  }
}
A GaN quantum dot (QD) is surrounded by an AlN quantum box: these two regions, named ball and qbox, constitute the atomistic cluster. Based on the finite element grid defined by this cluster, an atomistic structure is built, as defined in Atomistic block of the Device section. The reference lattice used to build the structure is defined by
reference_region = nside
to be that of the AlN material of one of the other regions which form the Device: two doped and
one intrinsic AlN regions.

Strain due to lattice mismatch between GaN and AlN (taken as reference substrate) is taken into
account through the Module elasticity

Module elasticity
{

name = str
regions = all

mesh_deformation = true
strain_atomistic_structure = tbl

plot = (Strain, Stress, Displacement )

Physics
{

body_force lattice_mismatch
{

reference_material = AlN

y-growth-direction = (-1,0,1,0)
x-growth-direction = (-1,2,-1,0)
z-growth-direction = (0,0,0,1)

}

}

Contact cathode
{type = clamp}

)

Note that to apply the strain deformation to the defined atomistic structure, we define
mesh_deformation = true. In this way, strain will be computed iteratively until the convergence
on the structure deformation is reached. It is also important to define the name of the atomistic
structure to be deformed, with strain_atomistic_structure = tbl.

Tight binding calculation of quantum states in the system made by the QD and the AlN surrounding
box is defined by the Module empirical_tb
Module empirical_tb
{
    regions = atomistic
    name = tb

    atomistic_structure = tb1
    potential_simulation = dd
    strain_simulation = str

    plot = (tбstates, MeshStatesNodes)

    Solver
    {
        num_valence_eigenvalues = 1
        num_conduction_eigenvalues = 1
        long_tolerance = 1e-4
    }
}

Note that the results of potential (from drift diffusion) and strain (from elasticity) simulations are recalled, to project the correct potential profile and atom displacement to the TB Hamiltonian. In output, the ground states of conduction and valence bands are saved together with the wavefunctions. States are also plotted on the FEM grid (through MeshStatesNodes).

Based on these results, optical properties are calculated through the Module opticstb

Module opticstb
{
    name = opt
    regions = atomistic

    initial_state_model = tb2
    final_state_model = tb2

    compute_strengths = true

    plot = (matrix_elements, optical_spectrum_k_0)
    output_format = grace

    Emin = 4.25
    Emax = 4.75
    dE = 0.001
}

The initial and final state for the spontaneous emission recombination are defined by

11.4. Example 1
initial_state_model = tb2
final_state_model = tb2

where “tb2” is the name of the empirical tb simulation which performs calculations. Usually, this is the same for electrons and holes.

The energy range for the calculation of the optical spectrum is defined through

\[ \begin{align*}
E_{\text{min}} &= 4.25 \\
E_{\text{max}} &= 4.75 \\
dE &= 0.001
\end{align*} \]

The output of calculation is composed by the matrix elements file `opt.dat` and by the optical spectrum in \( \Gamma \) point, in file `opt_spectrum_k_0.dat`. For the latter output, the keyword

`compute_strengths = true`

is necessary, to calculate emission spectrum power assuming all occupations equal to one, since the simulation is performed at equilibrium and thus with nominal zero occupation of states.

### 11.5 Example 2

In this second example, we will see how to perform two atomistic calculations on atomistic structure with different size. First a full atomistic structure will be generated, representing a quantum dot with its WL and buffer layers. On this full structure, a VFF relaxation will be executed. Then, EETB will be applied to a smaller subset of the relaxed structure, including only qdot and WL.

First, the Device structure is defined

```
Device qdot
{ meshfile = qdot_Zb.msh
  mesh_units = 1e-9
  dimension = 3

  structure = zb
  x-growth-direction = (1,0,0)
  y-growth-direction = (0,1,0)
  z-growth-direction = (0,0,1)

  Region box
  { material = GaAs }

  Region buffer_up
  { material = GaAs
```
Region buffer_down
{
    material = GaAs
}

Region Atomistic_cube_up
{
    material = GaAs
}

Region Atomistic_cube_down
{
    material = GaAs
}

Region dot
{
    material = InGaAs
    x = 0.7
}

Region buffer_WL
{
    material = InGaAs
    x = 0.7
}

Region Atomistic_cube_WL
{
    material = InGaAs
    x = 0.7
}

# group together the regions for the quantum simulation
Cluster Quantum_Atomistic
{
    regions = (Atomistic_cube_WL, Atomistic_cube_down, Atomistic_cube_up, dot)
}

Cluster Quantum_buffer
{
    regions = (Atomistic_cube_WL, Atomistic_cube_down, Atomistic_cube_up, dot,
               buffer_up, buffer_down, buffer_WL)
}
Here, two Clusters are defined: Quantum_buffer refers to the full device region where we want to generate an atomistic description; Quantum_Atomistic is a subset of the previous one, and will be used for ETB calculations.

Next, the Atomistic block defines the atomistic structure dot_atoms, associated to Cluster Quantum_buffer

Atomistic dot_atoms
{

regions = Quantum_buffer
#reference_region defines the reference lattice
reference_region = Atomistic_cube_down
#Enable hydrogen passivation
passivation = yes

print = (xyz, tgn)
# save the structure in several formats
}

Next, Elasticity Module is defined. In this way, macroscopic strain is calculated, deformation is applied to the mesh (mesh_deformation = true) and displacements are projected to the atomistic structure (strain_atomistic_structure = dot_atoms).

Module elasticity
{

name = strain
plot = (Strain,Stress,Displacement)

mesh_deformation = true
strain_atomistic_structure = dot_atoms

Physics {
    body_force lattice_mismatch
    {
        reference_material = GaAs
    }
}

Solver
{
    relative_tolerance = 1e-6
}

Contact bottom {type = clamp}
The result of **elasticity** is taken as a initial guess for next calculations. With **vff** Module, atomic positions of *dot_atoms* structure are relaxed by imposing a fixed position for all the atoms on the region boundary (*boundary_conditions = all_around*).

Module **vff**

```plaintext

{ 
  atomistic_structure = dot_atoms 
  boundary_conditions = all_around 
  plot = (xyz) 
}
```

**Drift-diffusion Module** will be executed to solve Poisson at equilibrium.

Module **driftdiffusion**

```plaintext

{ 
  name = dd 
  plot = (Ec, Ev, ElPotential, ElField, Polarization) 
  regions = all 
  coupling = poisson 
  integration_order = 2 

  #save_state = true
  #load_state = ./prova2/dd.tsv

  Physics
  { 
    band_properties 
    { 
      density_of_states bulk_kp 
      { 
        strain_simulation = strain 
      } 
    } 

    recombination srh { } 
    polarization (pyro) {} 
    polarization (piezo) {strain_simulation = strain}
  }
}
```

In alternative, a previously calculated solution from drift-diffusion may be loaded through

**11.5. Example 2**

169
Finally, we define **empirical_tb** Module

```plaintext
Module empirical_tb
{
    name = tb

    # Physical regions to which etb is applied
    regions = Quantum_Atomistic

    # Atomistic structure on which etb is calculated
    atomistic_structure = dot_atoms

    potential_simulation = dd
    strain_simulation = strain

    plot = (MeshStates,MeshStatesNodes)

    Solver
    {
        # load_states = true
        # load_path = prova

        num_valence_eigenvalues = 0
        num_conduction_eigenvalues = 2
        long_tolerance = 1e-7

        guess_conduction = 0.15
        guess_valence = -0.10
    }
}
```

The **ETB** calculations will be performed on the atomic structure *dot_atoms*, as deformed by the **VFF** relaxation.

By default, reasonable guesses are generated internally to help the solver to get convergence. Usually these values are good enough to get a steady convergence of solutions. However, in this particular case of small energy gap material (InGaAs), it may be necessary to set manually a guess close to the expected value of the eigenstate.

```plaintext
guess_conduction = 0.15
guess_valence = -0.10
```
In this way, we avoid erroneous states due to the folding of the Hamiltonian. The reference of the guess is the valence top band edge at 0.0 eV.

Note that ETB will be applied to the `atomistic_structure` `dot_atoms` (`atomistic_structure = dot_atoms`) , but only to the subset of it whose atoms are associated to the regions included in `Cluster Quantum_Atomistic` (`regions = Quantum_Atomistic`).

```plaintext
Simulation
{
  temperature = 300
  solve = (strain,vff,dd,tb)
  resultpath = output
  format = vtk
}
```

We solve for `strain` from `elasticity`, then for `vff` relaxation. `dd` calculates band structure at equilibrium based on which `tb` simulation is applied.

*Note that it is always highly advisable to solve first elasticity, before a VFF relaxation. This will provide a reasonable first guess for the VFF solver and also will apply the correct deformation to the mesh, assuring that atoms are still associated to the correct FEM elements.*
Part IV

Controlling the Simulation
12.1 Variables

In tiberCAD, the parameters defined in a module (e.g. voltage bias, temperature, material parameters) may be associated to a user-defined variable, in such a way that they their numerical value can be controlled during the simulation. The assignment to a variable is done in this way

\[ \text{parameter} = \$\text{my\_var}[0.0] \]

where \text{my\_var} is a user-defined variable and the value in square brackets \([0.0]\) is the default value of the variable.

For example

Contact anode
  
  \{ 
    type = ohmic 
    voltage = $\text{Vbias}[0.0] 
  
  
\}

here the value of the voltage in the defined Contact is given by the value of the variable \text{Vbias}

A typical application of a user-defined variable is in the Sweep calculation.

12.2 Sweep

Performs a linear or logarithmic sweep for a given variable.

Options and parameters:

- **type** [string] type of sweep, linear (default) or log
- **solve** [string] a list of simulations to be solved, eg. (\text{strain}, \text{dd})
variable [string] the sweep variable, eg. $Vd

start [double] the first value of the sweep

stop [double] the last value of the sweep

steps [integer] the number of steps the interval (start, stop) gets subdivided in. For logarithmic sweeps, default number of steps is calculated assuming a factor of 10 between successive values.

values [double] instead of start, stop and steps, provide the sweep values explicitly

plot_data [boolean] set to true if you want to plot after each step the simulation results (default is false)

file_mode [string] controls the behaviour for writing the data file containing global data. Can be one of append, overwrite (default) or no-overwrite

min_step [double] the minimum absolute step size

max_step [double] the maximum absolute step size

initial_step [double] the absolute initial step size

min_relative_step [double] minimum relative step size, default is 1e-3

max_relative_step [double] maximum relative step size, default is 1

initial_relative_step [double] initial relative step size, default is 1

The relative step sizes refer to each single sweep step. If max_relative_step is less than one, each sweep step will be subdivided in smaller steps. If a simulation fails, the step gets reduced by half until the simulation succeeds, or until the minimum relative or absolute step size is reached. In the latter case, the sweep is assumed to have failed. As for any module, a name can be given to a sweep block. This is important when several sweeps are defined, and in particular when nested sweeps (a solve option of one sweep refers to another sweep) are used.

Example:

Module sweep
{
  solve = (dd, thermal)
  variable = $Vbias
  start = 0.0
  stop = 1.2
  steps = 12
  plot_data = true
}

In this example, at each step of the sweep the two simulations dd and thermal are performed, in this order, while the variable is Vbias and assumes values between 0 and 1.2.
12.3 Selfconsistent

This tool may be used to solve models in an iterative way to obtain a selfconsistent solution, optionally using a relaxation approach.

Options and parameters:

- **name** [string] optional name of the selfconsistent block
- **max_iterations** [integer] the maximum number of iterations. When the maximum number of iterations is reached, the program issues a warning, exits from the self-consistent loop and proceeds to the next operation.
- **relative_tolerance** [double] the relative convergence tolerance for the observed variable in terms of the l2-norm
- **absolute_tolerance** [double] the absolute convergence tolerance for the observed variable in terms of the maximum-norm
- **relaxation_factor** [double] an optional relaxation factor (default is 1) to be applied to the observed variable
- **solve** [(list of) string(s)] the simulations to be solved

The observed variable on which convergence control and relaxation is done is the system variable of the last simulation specified in `solve`.

Example:

```
Module  selfconsistent
{
   name = sc_all
   solve = (quantum_el,quantum_hl, dd)
   max_iterations = 25  # 5

   absolute_tolerance = 1e-3
   relative_tolerance = 1e-5
}
```

In this example, the `selfconsistent` solution named `sc_all` is defined, which solves the simulations defined in the relevant modules as `quantum_el`, `quantum_hl` and `dd`, in this order.

Then, in Simulation, we execute `sc_all`, e.g.

```
Simulation
{
   verbose = 3

   solve = (strain, sweep, scall)
```
resultpath = output
output_format = grace
}

13.1 Numerical Solvers

The Solver block inside a module description contains the options for the numerical solver. The solver type the Solver block is describing (linear, nonlinear, eigenvalue solver) depends on the module. The options and parameters for the nonlinear and the linear solvers are presented in the following.

13.1.1 Nonlinear solvers

The Solvers implemented are of two types:

- **Petsc solver** = uses the PETSc nonlinear solver (SNES) (default)
- **Linesearch solver** = uses a linear linesearch implemented in TiberCAD

Nonlinear solvers are based on iterative methods, solving in each iteration a linear system. The linear solver used for this solution can be controlled by providing a block with the keyword linear_solver containing the options for the linear solver (see Linear solvers).

**Petsc solver**

- **relative_tolerance** [double] convergence criterion based on relative residual 1_2-norm, default 1e-6
- **absolute_tolerance** [double] convergence criterion based on the 1_2-norm of the residual, default 1e-50
- **max_iterations**: integer maximum number of iterations, default 500
- **step_tolerance** [double] tolerance criterion based on the 1_2-norm of the correction step
- **max_step** [double] maximum linesearch step (1_2-norm)
**divergence_tolerance** [double] divergence criterion

**Linesearch solver**

**absolute_tolerance** [double] convergence criterion based on the $l_2$-norm of the residual, default 1e-50

**relative_tolerance** [double] convergence criterion based on relative residual $l_2$-norm, default 1e-9

**step_tolerance** [double] tolerance criterion based on the $l_{\infty}$-norm of the correction step, default 1e-3

**max_iterations** [integer] maximum number of iterations, default 25

### 13.1.2 Linear solvers

**Petsc solver**

The Krylov subspace method to be used can be selected with the keyword **method**. For example:

```
method = cgs
```

Here are the available values for **method**; default is **bcs**:

- **bcs**: BiCGstab
- **bcgs**: BiCGstab(L)
- **gmres**: Generalized Minimal Residual
- **bicg**: BiConjugate Gradient
- **cg**: Conjugate Gradient
- **cgs**: Conjugate Gradient Squared
- **richardson**: Richardson

Other options:

- **pconly**: only apply preconditioner
- **relative_tolerance**: convergence criterion based on relative residual $l_2$-norm, default 1e-6
- **absolute_tolerance**: convergence criterion based on the $l_2$-norm of the residual, default 1e-50
- **max_iterations**: maximum number of iterations, default 1500
Preconditioner can be selected with the keyword `preconditioner`. For example

```plaintext
preconditioner = ilu
```

Available types of Preconditioner are:

- **lu**: LU
- **ilu**: incomplete LU
- **jacobi**: Jacobi
- **cholesky**: Cholesky
- **none**: no preconditioning
- **composite**: use a combination of Jacobi and iLU


