PNP Transport

Release 0.1

Oct 09, 2021

Contents

1 Dependencies **3**
- 1.1 FEniCS . . . 3
- 1.2 Python Modules . . . 4

2 Quick Start **7**
- 2.1 Quickstart: Finite Source Simulations . . . 8

3 Batch Analysis **11**
- 3.1 One-factor-at-a-time simulations . . . 11
- 3.2 One-factor-at-a-time analysis . . . 15

4 FEniCS Transport Simulations Code **25**
- 4.1 Finite Source Simulation Code . . . 25
- 4.2 Infinite Source Simulation Code . . . 25
- 4.3 Utils . . . 25
- 4.4 Confidence Intervals Methods . . . 28
- 4.5 Transport HD5 Storage System . . . 30
- 4.6 HD5 Storage . . . 30

5 PID Simulation Module **31**
- 5.1 Rsh Approximation Model . . . 31
- 5.2 Korol Conductivity Implementation . . . 31
- 5.3 Parameter Span . . . 32
- 5.4 Conductivty Interface . . . 37

6 Indices and tables **39**

Python Module Index **41**

Index **43**

This framework uses FEniCS to estimate the numerical solution to Poisson-Nernst-Planck equation to solve the transport kinetics of charged species in dielectrics and stacks of materials.

CHAPTER 1

Dependencies

1.1 FEniCS

In order to be able to run this code you need to have FEniCS installed. The best way to achieve this is to use a *dockerized* installation to run FEniCS. Refer to

FEniCS installation guide

FEniCS installation usually includes a minimum set of python libraries. However, you might need to install additional ones like.

It is recommended to create a named container with a folder shared with the local os:

```
$ docker run -ti -v $(pwd):/home/fenics/shared --name fenics-container quay.io/
↪fenicsproject/stable
```

To start the container run

```
$ docker start fenics-container
```

To stop the container run

```
$ docker stop fenics-container
```

To run the container we can create a shell script containing

Listing 1: run_fenics.sh

```
#!/bin/bash
docker exec -ti -u fenics fenics-container /bin/bash -l
```

Add execution permissions to the script

```
$ chmod +x run_fenics.sh
```

Then, we can just access the container by

```
$ ./run_fenics.sh
```

1.2 Python Modules

To run the analysis on the client side, make sure you have the following packages

1. Matplotlib
2. Scipy
3. h5py
4. pandas
5. tqdm

1.2.1 Installation of dependencies using PIP

Install the matplotlib package

```
$ pip install matplotlib
```

Install scipy

```
$ pip install scipy
```

Install the h5py package

```
$ pip install h5py
```

Install pandas

```
$ pip install pandas
```

Install tqdm (for progress bars)

```
$ pip install tqdm
```

1.2.2 Installation of dependencies using conda

Conda distributions usually come with matplotlib, scipy. In case your distribution does not include it you can run

```
$ conda install matplotlib
$ conda install scipy
```

Install the h5py package

```
$ conda install h5py
```

Install pandas

```
$ conda install pandas
```

Install tqdm (for progress bars)

```
$ conda install -c conda-forge tqdm
```

CHAPTER 2

Quick Start

```
$ cd executables
$ chmod +x *.sh
```

Running a finite source simulation

```
$ cd ./executables
$ ./simulate_fs.py --config input_example.ini
```

where the .ini file looks like

Listing 1: input_example.ini

```
[global]
# Simulation time in seconds
simulation_time = 345600.0
# Simulation temperature in °C
temperature = 85
# Surface source concentration in cm-2
surface_concentration = 1.0E+10
# Monolayer ingress rate (1/s)
rate_source = 1.000E-04
# The base filename for the output
filetag = output_file_Tag
# Number of time steps
time_steps = 720
# The surface mass transfer coefficient in cm/s
h = 1.0E-12
# The segregation coefficient
m = 1.0E+00
# The recovery time in seconds (default 0)
recovery_time = 43200.0
# The recovery voltage drop in the sinx layer
recovery_voltage = -7.5e-05
# Background concentration in cm-3
```

(continues on next page)

(continued from previous page)

```
cb = 1.000E-20
# Dielectric constant
er = 7.0

[sinx]
# The diffusivity of Na in cm^2/s
d = 3.92E-16
# The applied voltage stress in volts
stress_voltage = 7.500E-05
# The thickness of the layer in um
thickness = 0.075
# The number of points in the layer
npoints = 100

[si]
# The diffusivity of Na in cm^2/s
d = 1.000E-14
# The thickness of the layer in um
thickness = 1.0
# The number of points in the layer
npoints = 100
```

2.1 Quickstart: Finite Source Simulations

The simplest way to run the code is to run a simulation using an ini file from the command line.

Shell scripts are available at the *executables* folder in the root of the installation. If they do not already have execution permissions run:

```
$ cd executables
$ chmod +x *.sh
```

Running a finite source simulation. From the root of pnptransport run

```
$ ./simulate_fs.py --config input_example.ini
```

where the .ini file looks like

Listing 2: input_example.ini

```
[global]
# Simulation time in seconds
simulation_time = 345600.0
# Simulation temperature in °C
temperature = 85
# Surface source concentration in cm-2
surface_concentration = 1.0E+10
# Monolayer ingress rate (1/s)
rate_source = 1.000E-04
# The base filename for the output
filetag = output_file_Tag
# Number of time steps
time_steps = 720
# The surface mass transfer coefficient in cm/s
```

(continues on next page)

(continued from previous page)

```
h = 1.0E-12
# The segregation coefficient
m = 1.0E+00
# The recovery time in seconds (default 0)
recovery_time = 43200.0
# The recovery voltage drop in the sinx layer
recovery_voltage = -7.5e-05
# Background concentration in cm-3
cb = 1.000E-20
# Dielectric constant
er = 7.0

[sinx]
# The diffusivity of Na in cm^2/s
d = 3.92E-16
# The applied voltage stress in volts
stress_voltage = 7.500E-05
# The thickness of the layer in um
thickness = 0.075
# The number of points in the layer
npoints = 100

[si]
# The diffusivity of Na in cm^2/s
d = 1.000E-14
# The thickness of the layer in um
thickness = 1.0
# The number of points in the layer
npoints = 100
```

The sections of the input file are

2.1.1 global

This section contains parameters that are not layer specific including

simulation_time: str This corresponds to the total time to be simulated in seconds.

temperature: float The simulated temperature in °C. Used to determine ionic mobility in the dielectric, according to $\mu = Dq/k_{\mathrm{B}}T$.

surface_concentration: float The surface concentration at the source S, given in cm^{-2} . Used to determine the flux at the source, given by $J_0 = kS$, where k is the rate of ingress.

rate_source: float The rate of ingress of ionic contamination at the source, in s:sup:-*1*. Used to determine the flux at the source, $J_0 = kS$.

filetag: str The file tag used to generate the output folder and files.

time_steps: int The number of time intervals to simulate.

h: float The surface mass transfer coefficient in cm/s, for the segregation flux at the SiN_x / Si interface.

m: float The segregation coefficient at the dielectric/semiconductor interface.

recovery_time: float The additional simulation time in seconds without PID stress used for recovery.

recovery_voltage: float The bias used during recovery in V. This is applied to the dielectric layer and ideally needs to be negative.

cb: float The background concentration in cm^{-3} . Used as a finite initial concentration.

er: float The relative permittivity of the dielectric.

2.1.2 sinx

d: float The diffusion coefficient of the ionic species in the dielectric in cm^2 /s.

stress_voltage: float The applied voltage stress in the film in V.

thickness: float The thickness of the layer in um.

npoints: int The number of grid points to simulate.

2.1.3 si

d: float The diffusion coefficient of the ionic species in the semiconductor in cm^2 /s.

stress_voltage: float The applied voltage stress in the film in V.

thickness: float The thickness of the layer in um.

npoints: int The number of grid points to simulate.

The code needs to be run in a linux terminal. However, it is recommended to use a graphical environment to keep the processes alive if the remote connection with the server fails.

The default directory structure of the simulation will be

```
top_folder
|---base_folder
|   |---input_file.ini
|---results
|   |---constant-flux
|   |   |---filetag.h5
|   |   |---filetag.ini
```

The results folder can be specified by the optional argument *–output* to the *simulate_fs.py* script

```
./simulate_fs.py --config input_file.ini --output folder_output
```

which will generate a folder structure like this.

```
top_folder
|---base_folder
|   |---input_file.ini
|---output_folder
|   |---filetag.h5
|   |---filetag.ini
```

CHAPTER 3

Batch Analysis

3.1 One-factor-at-a-time simulations

The module *pnptransport.parameter_span* provides functions to multiple input files necessary to simulate the effect of the variation of a specific parameter.

The following scripts provides an example of the usage:

Listing 1: one_factor_at_a_time.py

```
"""
This program will create all the necessary input files to run pnp transport simulations with 'one factor at a time'
variations.

The variations on the relevant parameters are described in  pidsim.parameter_span.one_factor_at_a_time documentation.
These variations are submitted through a csv data file.

The rest of the parameters are assumed to be constant over all the simulations.

Besides the input files, the code will generate a database as a csv file with all the simulations to be run and the
parameters used for each simulation.

@author: <erickrmartinez@gmail.com>
"""
import numpy as np
import pidsim.parameter_span as pspan

# The path to the csv file with the conditions of the different variations
csv_file = r'G:\My Drive\Research\PVRD1\Manuscripts\Device_Simulations_draft\simulations\one_factor_at_a_time_lower_20201028_h=1E-12.csv'
# Simulation time in h
```

(continues on next page)

(continued from previous page)

```
simulation_time_h = 96.
# Temperature in °C
temperature_c = 85
# Relative permittivity of SiNx
er = 7.0
# Thickness of the SiNx um
thickness_sin = 75E-3
# Modeled thickness of Si um
thickness_si = 1.0
# Number of time steps
t_steps = 1440
# Number of elements in the sin layer
x_points_sin = 100
# number of elements in the Si layer
x_points_si = 100
# Background concentration in cm^-3
cb = 1E-20


if __name__ == '__main__':
    pspan.one_factor_at_a_time(
        csv_file=csv_file, simulation_time=simulation_time_h*3600, temperature_
↪c=temperature_c, er=er,
        thickness_sin=thickness_sin, thickness_si=thickness_si, t_steps=t_steps, x_
↪points_sin=x_points_sin,
        x_points_si=x_points_si, base_concentration=cb
    )
```

The script will use the *csv* file defined in *csv_file* which has the following form

Table 1: Parameter Span CSV input

Parameter	base	span
sigma_s	1.00E+10	1E10,1E11,1E12,1E13,1E14
zeta	1.00E-04	1E-8, 1E-7, 1E-6, 1E-5,1E-4
DSF	1.00E-14	1E-14,1E-15,1E-16
E	1.00E+04	1E1,1E2,1E3,1E4,1E5,1E6
m	1.00E+00	1
h	1.00E-12	1E-12,1E-10,1E-8
recovery time	43200	43200
recovery electric field	-1.00E+04	-1.00E+04

sigma_s corresponds to the surface concentration at the source S in cm^{-2} , **zeta** corresponds to the rate of ingress from the source k in s^{-1} , **DSF** is the diffusivity of Na in the stacking fault D_{SF} in cm^2/s, **E** is the electric field in SiN_xgiven in V/cm, h and m are the surface mass transfer coefficient (cm/s) and segregation coefficient at the SiN_x/Si interface, respectively. The **recovery time** is indicated in seconds and corresponds to time added to the simulation where the system is either (1) allowed to relax by removing the electric stress or, (2) stressed under the PID stress in reverse polarity. The value of the recovery electric field is indicated in the last row of the table.

The column *base* corresponds to the base case to compare the rest of the simulations. The span column indicates all the variations from the base case for the respective parameter in the row, while keeping the rest of the parameters constant.

After running the code, the following file structure will be created

which will generate a folder structure like this.

```
base_folder
|---one_factor_at_a_time.csv
|---input
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+01Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+01Vcm.ini
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+02Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+02Vcm.ini
|   |--- ...
|   |---ofat_db.csv
|   |---batch_YYYYMMDD.sh
```

With n .ini files corresponding to all of the variations. The naming convention for the .ini files is *constant_source_flux_* + PID stress time in hours + _ + T in °C + *C_* + S in cm^{-2}+ *pcm2_z* + k in s^{-1} + *ps_DSF* + D_{SF} in cm^2/s + _ + E in V/cm + *_h* + h in cm/s + *_m* + m + *_rt* + recovery time in hours + *h_rv* + recovery voltage (at the SiN_x) in V/cm.

Additionally, the scripts create a batch script *batch_YYYYMMDD.sh* that needs to be run in the location where *simulate_fs.py* can be reached. This will send all the simulations as separate python jobs to the OS.

Lastly, the script generates a *ofat_db.csv* table containing the list of all simulations with all the parameters used in each case:

Table 2: OFAT Database

config file	sigma_s (cm^-2)	zeta (1/s)	D_SF (cm^2/s)	E (V/cm)	h (cm/s)	m	time (s)	temp (C)	bias (V)	recovery time (s)	recovery E (V/cm)	recovery bias (V)	thickness sin (um)	thickness si (um)	er	cb (cm^-3)	t_steps	x_points sin	x_points si
constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+10	1.00E-04	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+11pcm2_z1E-04ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+11	1.00E-04	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+12pcm2_z1E-04ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+12	1.00E-04	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+13pcm2_z1E-04ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+13	1.00E-04	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+14pcm2_z1E-04ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+14	1.00E-04	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+10pcm2_z1E-08ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+10	1.00E-08	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+10pcm2_z1E-07ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+10	1.00E-07	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+10pcm2_z1E-06ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+10	1.00E-06	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+10pcm2_z1E-05ps_DSF1E-14_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+10	1.00E-05	1.00E-14	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-15_1E+04Vcm_h1E-12_m1E+00_rt12h_rv-1E+04Vcm.ini	1.00E+10	1.00E-04	1.00E-15	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100
constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-16_1E+04Vcm_h1E-	1.00E+10	1.00E-04	1.00E-16	1.00E+04	1.00E-12	1	345600	85	0.075	43200	-10000	-0.075	0.075	1	7	1.00E+20	720	100	100

Depending on the choice of parameters, large concentration and potential gradients can lead to non-convergent simulations. The batch script will run the remainder of the simulations that do converge. Adjustments to the time step and mesh elements might be needed to reach convergence. In any case, it is convenient to add an additional column to *ofat_db.csv* to flag if the simulation converged for further batch analysis.

3.2 One-factor-at-a-time analysis

A script is provided to analyze the output of one-factor-at-a-time batch simulations, which plots the concentration profile as a function of time for each simulation and also simulates the P_{mpp} and R_{sh} as a function of time using a Random Forrest Regression model fit from previous Sentaurus simulations.

Listing 2: one_factor_at_a_time_analysys.py

```
"""
This code runs basic analysis on simulations that were computed using the 'one at a time analysis'.

You must provide the path to the csv database with the parameters of each simulation.

Functionality:

1. Plot the last concentration profile over the layer stack.
2. Plot the Rsh(t) estimated with the series resistor model.
3. Estimate the integrated sodium concentration in SiNx and Si at the end of the simulation.
"""
import numpy as np
import pandas as pd
from mpl_toolkits.axes_grid1 import make_axes_locatable
# import pidsim.rsh as prsh
import pidsim.ml_simulator as pmpp_rf
import h5py
import os
import platform
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.ticker as mticker
import matplotlib.gridspec as gridspec
from scipy import integrate
import pnptransport.utils as utils
from tqdm import tqdm

path_to_csv = r'G:\My Drive\Research\PVRD1\Manuscripts\Device_Simulations_draft\simulations\inputs_20201028\ofat_db.csv'
path_to_results = r'G:\My Drive\Research\PVRD1\Manuscripts\Device_Simulations_draft\simulations\inputs_20201028\results'
t_max_h = 96.  # h

pid_experiment_csv = None #'G:\My Drive\Research\PVRD1\DATA\PID\MC4_Raw_IV_modified.csv'

color_map = 'viridis_r'

defaultPlotStyle = {
    'font.size': 11,
```

(continues on next page)

(continued from previous page)

```
    'font.family': 'Arial',
    'font.weight': 'regular',
    'legend.fontsize': 11,
    'mathtext.fontset': 'stix',
    'xtick.direction': 'in',
    'ytick.direction': 'in',
    'xtick.major.size': 4.5,
    'xtick.major.width': 1.75,
    'ytick.major.size': 4.5,
    'ytick.major.width': 1.75,
    'xtick.minor.size': 2.75,
    'xtick.minor.width': 1.0,
    'ytick.minor.size': 2.75,
    'ytick.minor.width': 1.0,
    'xtick.top': False,
    'ytick.right': False,
    'lines.linewidth': 2.5,
    'lines.markersize': 10,
    'lines.markeredgewidth': 0.85,
    'axes.labelpad': 5.0,
    'axes.labelsize': 12,
    'axes.labelweight': 'regular',
    'legend.handletextpad': 0.2,
    'legend.borderaxespad': 0.2,
    'axes.linewidth': 1.25,
    'axes.titlesize': 12,
    'axes.titleweight': 'bold',
    'axes.titlepad': 6,
    'figure.titleweight': 'bold',
    'figure.dpi': 100
}

if __name__ == '__main__':
    if platform.system() == 'Windows':
        path_to_csv = r'\\?\' + path_to_csv
        path_to_results = r'\\?\' + path_to_results
        if pid_experiment_csv is not None:
            pid_experiment_csv = r'\\?\' + pid_experiment_csv

    t_max = t_max_h * 3600.
    # Create an analysis folder within the base dir for the database file
    working_path = os.path.dirname(path_to_csv)
    analysis_path = os.path.join(working_path, 'batch_analysis')
    # If the folder does not exists, create it
    if not os.path.exists(analysis_path):
        os.makedirs(analysis_path)

    # If an experimental profile is provided load the csv
    if pid_experiment_csv is not None:
        pid_experiment_df = pd.read_csv(pid_experiment_csv)

    # Read the database of simulations
    simulations_df = pd.read_csv(filepath_or_buffer=path_to_csv)
    # pick only the simulations that converged
    simulations_df = simulations_df[simulations_df['converged'] == 1].reset_
↪index(drop=True)
    # Count the simulations
```

(continues on next page)

(continued from previous page)

```
    n_simulations = len(simulations_df)
    integrated_final_concentrations = np.empty(n_simulations, dtype=np.dtype([
        ('C_SiNx average final (atoms/cm^3)', 'd'), ('C_Si average final (atoms/cm^3)
↪', 'd')
    ]))
    # Load the style
    mpl.rcParams.update(defaultPlotStyle)
    # Get the color map
    cm = mpl.cm.get_cmap(color_map)
    # Show at least the first 6 figures
    max_displayed_figures = 6
    fig_counter = 0
    for i, r in simulations_df.iterrows():
        filetag = os.path.splitext(r['config file'])[0]
        simga_s = r['sigma_s (cm^-2)']
        zeta = r['zeta (1/s)']
        dsf = r['D_SF (cm^2/s)']
        e_field = r['E (V/cm)']
        h = r['h (cm/s)']
        m = r['m']
        time_max = r['time (s)']
        temp_c = r['temp (C)']

        source_str1 = r'$S_{{\mathrm{{s}}}} = {0} \; (\mathrm{{cm^{{-2}}}})$'.format(
            utils.latex_order_of_magnitude(simga_s))
        source_str2 = r'$k = {0} \; (\mathrm{{1/s}})$'.format(utils.latex_order_of_
↪magnitude(zeta))
        e_field_str = r'$E = {0} \; (\mathrm{{V/cm}})$'.format(utils.latex_order_of_
↪magnitude(e_field))
        h_str = r'$h = {0} \; (\mathrm{{cm/s}})$'.format(utils.latex_order_of_
↪magnitude(h))
        temp_str = r'${0:.0f} \; (\mathrm{{°C}})$'.format(temp_c)
        dsf_str = r'$D_{{\mathrm{{SF}}}} = {0} \; (\mathrm{{cm^2/s}})$'.format(utils.
↪latex_order_of_magnitude(dsf))
        # Normalize the time scale
        normalize = mpl.colors.Normalize(vmin=1E-3, vmax=(t_max / 3600.))
        # Get a 20 time points geometrically spaced
        requested_time = utils.geometric_series_spaced(max_val=t_max, min_delta=600,␣
↪steps=20)
        # Get the full path to the h5 file
        path_to_h5 = os.path.join(path_to_results, filetag + '.h5')
        # Create the concentration figure
        fig_c = plt.figure()
        fig_c.set_size_inches(5.0, 3.0, forward=True)
        fig_c.subplots_adjust(hspace=0.1, wspace=0.1)
        gs_c_0 = gridspec.GridSpec(ncols=1, nrows=1, figure=fig_c)
        # 1 column for the concentration profile in SiNx
        # 1 column for the concentration profile in Si
        # 1 column for the colorbar
        gs_c_00 = gridspec.GridSpecFromSubplotSpec(
            nrows=1, ncols=2, subplot_spec=gs_c_0[0], wspace=0.0, hspace=0.1, width_
↪ratios=[2.5, 3]
        )
        ax_c_0 = fig_c.add_subplot(gs_c_00[0, 0])
        ax_c_1 = fig_c.add_subplot(gs_c_00[0, 1])

        # Axis labels
```

(continues on next page)

(continued from previous page)

```
        ax_c_0.set_xlabel(r'Depth (nm)')
        ax_c_0.set_ylabel(r'[Na] ($\mathregular{cm^{-3}}$)')
        # Title to the sinx axis
        ax_c_0.set_title(r'${0}\; \mathrm{{V/cm}}, {1:.0f}\; \mathrm{{°C}}$'.format(
            utils.latex_order_of_magnitude(e_field), temp_c
        ))
        # Set the ticks for the Si concentration profile axis to the right
        ax_c_1.yaxis.set_ticks_position('right')
        # Title to the si axis
        ax_c_1.set_title(r'$D_{{\mathrm{{SF}}}} = {0}\; \mathrm{{cm^2/s}},\; E=0$'.
↪format(
            utils.latex_order_of_magnitude(dsf)
        ))
        ax_c_1.set_xlabel(r'Depth (um)')
        # Log plot in the y axis
        ax_c_0.set_yscale('log')
        ax_c_1.set_yscale('log')
        ax_c_0.set_ylim(bottom=1E10, top=1E20)
        ax_c_1.set_ylim(bottom=1E10, top=1E20)
        # Set the ticks for the SiNx log axis
        ax_c_0.yaxis.set_major_locator(mpl.ticker.LogLocator(base=10.0, numticks=6))
        ax_c_0.yaxis.set_minor_locator(mpl.ticker.LogLocator(base=10.0, numticks=60,␣
↪subs=np.arange(2, 10) * .1))
        # Set the ticks for the Si log axis
        ax_c_1.yaxis.set_major_locator(mpl.ticker.LogLocator(base=10.0, numticks=6))
        ax_c_1.yaxis.set_minor_locator(mpl.ticker.LogLocator(base=10.0, numticks=60,␣
↪subs=np.arange(2, 10) * .1))
        ax_c_1.tick_params(axis='y', left=False, labelright=False)
        # Configure the ticks for the x axis
        ax_c_0.xaxis.set_major_locator(mticker.MaxNLocator(4, prune=None))
        ax_c_0.xaxis.set_minor_locator(mticker.AutoMinorLocator(4))
        ax_c_1.xaxis.set_major_locator(mticker.MaxNLocator(3, prune='lower'))
        ax_c_1.xaxis.set_minor_locator(mticker.AutoMinorLocator(4))
        # Change the background colors
        # ax_c_0.set_facecolor((0.89, 0.75, 1.0))
        # ax_c_1.set_facecolor((0.82, 0.83, 1.0))
        # Create the integrated concentration figure
        fig_s = plt.figure()
        fig_s.set_size_inches(4.75, 3.0, forward=True)
        fig_s.subplots_adjust(hspace=0.1, wspace=0.1)
        gs_s_0 = gridspec.GridSpec(ncols=1, nrows=1, figure=fig_s)
        gs_s_00 = gridspec.GridSpecFromSubplotSpec(
            nrows=1, ncols=1, subplot_spec=gs_s_0[0], hspace=0.1,
        )
        ax_s_0 = fig_s.add_subplot(gs_s_00[0, 0])
        # Set the axis labels
        ax_s_0.set_xlabel(r'Time (h)')
        ax_s_0.set_ylabel(r'$\bar{C}$ ($\mathregular{cm^{-3}}$)')
        # Set the limits for the x axis
        ax_s_0.set_xlim(left=0, right=t_max / 3600.)
        # Make the y axis log
        ax_s_0.set_yscale('log')
        # Set the ticks for the y axis
        ax_s_0.yaxis.set_major_locator(mpl.ticker.LogLocator(base=10.0, numticks=6))
        ax_s_0.yaxis.set_minor_locator(mpl.ticker.LogLocator(base=10.0, numticks=60,␣
↪subs=np.arange(2, 10) * .1))
        # Set the ticks for the x axis
```

(continues on next page)

(continued from previous page)

```
        # Configure the ticks for the x axis
        ax_s_0.xaxis.set_major_locator(mticker.MaxNLocator(6, prune=None))
        ax_s_0.xaxis.set_minor_locator(mticker.AutoMinorLocator(2))
        # Create the mpp figure
        fig_mpp = plt.figure()
        fig_mpp.set_size_inches(4.75, 3.0, forward=True)
        fig_mpp.subplots_adjust(hspace=0.1, wspace=0.1)
        gs_mpp_0 = gridspec.GridSpec(ncols=1, nrows=1, figure=fig_mpp)
        gs_mpp_00 = gridspec.GridSpecFromSubplotSpec(
            nrows=1, ncols=1, subplot_spec=gs_mpp_0[0], hspace=0.1,
        )
        ax_mpp_0 = fig_mpp.add_subplot(gs_mpp_00[0, 0])
        # Set the axis labels
        ax_mpp_0.set_xlabel(r'Time (h)')
        ax_mpp_0.set_ylabel(r'$R_{\mathrm{sh}}$ ($\mathrm{\Omega\cdot cm^2}}$)')

        # Vfb figure
        fig_vfb = plt.figure()
        fig_vfb.set_size_inches(4.75, 3.0, forward=True)
        fig_vfb.subplots_adjust(hspace=0.1, wspace=0.1)
        gs_vfb_0 = gridspec.GridSpec(ncols=1, nrows=1, figure=fig_vfb)
        gs_vfb_00 = gridspec.GridSpecFromSubplotSpec(
            nrows=1, ncols=1, subplot_spec=gs_vfb_0[0], hspace=0.1,
        )
        ax_vfb_0 = fig_vfb.add_subplot(gs_vfb_00[0, 0])
        # Set the axis labels
        ax_vfb_0.set_xlabel(r'Time (h)')
        ax_vfb_0.set_ylabel(r'$V_{\mathrm{FB}}$ (V)')

        with h5py.File(path_to_h5, 'r') as hf:
            # Get the time dataset
            time_s = np.array(hf['time'])
            # Get the vfb dataset
            vfb = np.array(hf.get(name='vfb'))
            # Get the sinx group
            grp_sinx = hf['L1']
            # get the si group
            grp_si = hf['L2']
            # Get the position vector in SiNx in nm
            x_sin = np.array(grp_sinx['x']) * 1000.
            thickness_sin = np.max(x_sin)
            x_si = np.array(grp_si['x']) - thickness_sin / 1000.
            x_sin = x_sin - thickness_sin
            thickness_si = np.amax(x_si)
            n_profiles = len(time_s)
            requested_indices = utils.get_indices_at_values(x=time_s, requested_
↪values=requested_time)
            time_profile = np.empty(len(requested_indices))

            model_colors = [cm(normalize(t)) for t in time_s / 3600.]
            scalar_maps = mpl.cm.ScalarMappable(cmap=cm, norm=normalize)
            with tqdm(requested_indices, leave=True, position=0) as pbar:
                for j, idx in enumerate(requested_indices):
                    time_j = time_s[idx] / 3600.
                    time_profile[j] = time_j
                    # Get the specific profile
                    ct_ds = 'ct_{0:d}'.format(idx)
```

(continues on next page)

(continued from previous page)

```
                    try:
                        c_sin = np.array(grp_sinx['concentration'][ct_ds])
                        c_si = np.array(grp_si['concentration'][ct_ds])
                        color_j = cm(normalize(time_j))
                        ax_c_0.plot(x_sin, c_sin, color=color_j, zorder=0)
                        ax_c_1.plot(x_si, c_si, color=color_j, zorder=0)
                        pbar.set_description('Extracting profile {0} at time {1:.1f}␣
↪h...'.format(ct_ds, time_j))
                        pbar.update()
                        pbar.refresh()
                    except KeyError as ke:
                        print("Error reading file '{0}'.".format(filetag))
                        raise ke

            # Estimate the integrated concentrations as a function of time for each␣
↪layer
            c_sin_int = np.empty(n_profiles)
            c_si_int = np.empty(n_profiles)
            with tqdm(range(n_profiles), leave=True, position=0) as pbar:
                for j in range(n_profiles):
                    # Get the specific profile
                    ct_ds = 'ct_{0:d}'.format(j)
                    c_sin = np.array(grp_sinx['concentration'][ct_ds])
                    c_si = np.array(grp_si['concentration'][ct_ds])
                    c_sin_int[j] = abs(integrate.simps(c_sin, -x_sin )) / thickness_
↪sin
                    c_si_int[j] = abs(integrate.simps(c_si, x_si)) / thickness_si
                    pbar.set_description('Integrating profile at time {0:.1f} h: S_N:
↪{1:.2E}, S_S: {2:.3E} cm^-2'.format(
                        time_s[j] / 3600.,
                        c_sin_int[j],
                        c_si_int[j]
                    ))
                    pbar.update()
                    pbar.refresh()

        ax_s_0.plot(time_s / 3600., c_sin_int, label=r'$\mathregular{SiN_x}$')
        ax_s_0.plot(time_s / 3600., c_si_int, label=r'Si')
        # ax_s_0.plot(time_s / 3600., c_si_int + c_sin_int, label=r'Si + $\mathregular
↪{SiN_x}$')

        ax_vfb_0.plot(time_s / 3600., vfb)
        ax_vfb_0.set_xlim(left=0, right=t_max_h)

        integrated_final_concentrations[i] = (c_sin_int[-1], c_si_int[-1])

        leg = ax_s_0.legend(loc='lower right', frameon=True)

        # Set the limits for the x axis of the concentration plot
        ax_c_0.set_xlim(left=np.amin(x_sin), right=np.amax(x_sin))
        ax_c_1.set_xlim(left=np.amin(x_si), right=np.amax(x_si))
        # Add the color bar
        divider = make_axes_locatable(ax_c_1)
        cax = divider.append_axes("right", size="7.5%", pad=0.03)
        cbar = fig_c.colorbar(scalar_maps, cax=cax)
        cbar.set_label('Time (h)\n', rotation=90, fontsize=14)
        cbar.ax.tick_params(labelsize=11)
```

(continues on next page)

(continued from previous page)

```
        plot_c_sin_txt = source_str1 + '\n' + source_str2
        ax_c_0.text(
            0.95, 0.95,
            plot_c_sin_txt,
            horizontalalignment='right',
            verticalalignment='top',
            transform=ax_c_0.transAxes,
            fontsize=11,
            color='k'
        )

        plot_c_si_txt = h_str + '\n$m=1$'
        ax_c_1.text(
            0.95, 0.95,
            plot_c_si_txt,
            horizontalalignment='right',
            verticalalignment='top',
            transform=ax_c_1.transAxes,
            fontsize=11,
            color='k'
        )

        # Identify layers
        ax_c_0.text(
            0.05, 0.015,
            r'$\mathregular{SiN_x}$',
            horizontalalignment='left',
            verticalalignment='bottom',
            transform=ax_c_0.transAxes,
            fontsize=11,
            fontweight='bold',
            color='k'
        )

        ax_c_1.text(
            0.05, 0.015,
            'Si',
            horizontalalignment='left',
            verticalalignment='bottom',
            transform=ax_c_1.transAxes,
            fontsize=11,
            fontweight='bold',
            color='k'
        )

        # set the y axis limits for the integrated concentration plot
        ax_s_0.set_ylim(bottom=1E5, top=1E20)
        title_str = source_str1 + ', ' + source_str2 + ', ' + dsf_str

        plot_txt = e_field_str + '\n' + temp_str + '\n' + h_str
        ax_s_0.set_title(title_str)
        ax_s_0.text(
            0.65, 0.95,
            plot_txt,
            horizontalalignment='left',
            verticalalignment='top',
```

(continues on next page)

(continued from previous page)

```
            transform=ax_s_0.transAxes,
            fontsize=11,
            color='k'
        )

        # rsh_analysis = prsh.Rsh(h5_transport_file=path_to_h5)
        ml_analysis = pmpp_rf.MLSim(h5_transport_file=path_to_h5)
        time_s = ml_analysis.time_s
        time_h = time_s / 3600.
        requested_indices = ml_analysis.get_requested_time_indices(time_s)
        pmpp = ml_analysis.pmpp_time_series(requested_indices=requested_indices)
        rsh = ml_analysis.rsh_time_series(requested_indices=requested_indices)

        simulated_pmpp_df = pd.DataFrame(data={
            'time (s)': time_s, 'Pmpp (mW/cm^2)': pmpp, 'Rsh (Ohm cm^2)': rsh,
            'vfb (V)': vfb
        })
        simulated_pmpp_df.to_csv(os.path.join(analysis_path, filetag + '_simulated_
↪pid.csv'), index=False)

        ax_mpp_0.plot(time_h, rsh, label='Simulation')
        ax_mpp_0.set_xlim(0, np.amax(time_h))
        if pid_experiment_csv is not None:
            time_exp = pid_experiment_df['time (s)']/3600.
            pmax_exp = pid_experiment_df['Pmax']
            ax_mpp_0.plot(time_exp, pmax_exp / pmax_exp.max(), ls='None', marker='o',␣
↪fillstyle='none', label='Experiment')
            leg = ax_mpp_0.legend(loc='lower right', frameon=True)
        ax_mpp_0.set_yscale('log')
        ax_mpp_0.set_xlabel('time (h)')
        ax_mpp_0.set_ylabel('$R_{\mathrm{sh}}\;(\Omega \cdot \mathregular{cm^2})$')
        # ax_mpp_0.set_ylabel('Normalized Power')

        ax_mpp_0.xaxis.set_major_locator(mticker.MaxNLocator(6, prune=None))
        ax_mpp_0.xaxis.set_minor_locator(mticker.AutoMinorLocator(2))

        title_str = source_str1 + ', ' + source_str2 + ', ' + dsf_str

        plot_txt = e_field_str + '\n' + temp_str + '\n' + h_str
        ax_mpp_0.set_title(title_str)
        ax_mpp_0.text(
            0.65, 0.95,
            plot_txt,
            horizontalalignment='left',
            verticalalignment='top',
            transform=ax_mpp_0.transAxes,
            fontsize=11,
            color='k'
        )

        ax_vfb_0.set_title(title_str)
        ax_vfb_0.text(
            0.65, 0.95,
            plot_txt,
            horizontalalignment='left',
            verticalalignment='top',
            transform=ax_vfb_0.transAxes,
```

(continues on next page)

(continued from previous page)

```
417             fontsize=11,
418             color='k'
419         )
420 
421         fig_c.tight_layout()
422         fig_s.tight_layout()
423         fig_mpp.tight_layout()
424         fig_vfb.tight_layout()
425 
426         fig_c.savefig(os.path.join(analysis_path, filetag + '_c.png'), dpi=600)
427         fig_c.savefig(os.path.join(analysis_path, filetag + '_c.svg'), dpi=600)
428         fig_s.savefig(os.path.join(analysis_path, filetag + '_s.png'), dpi=600)
429         fig_mpp.savefig(os.path.join(analysis_path, filetag + '_p.png'), dpi=600)
430         fig_mpp.savefig(os.path.join(analysis_path, filetag + '_p.svg'), dpi=600)
431         fig_vfb.savefig(os.path.join(analysis_path, filetag + '_vfb.png'), dpi=600)
432         fig_vfb.savefig(os.path.join(analysis_path, filetag + '_vfb.svg'), dpi=600)
433 
434         plt.close(fig_c)
435         plt.close(fig_s)
436         plt.close(fig_mpp)
437         plt.close(fig_vfb)
438 
439         del fig_c, fig_s, fig_mpp, fig_vfb
440 
441     simulations_df['C_SiNx average final (atoms/cm^3)'] = integrated_final_
    ↪concentrations['C_SiNx average final (atoms/cm^3)']
442     simulations_df['C_Si average final (atoms/cm^3)'] = integrated_final_
    ↪concentrations['C_Si average final (atoms/cm^3)']
443 
444     simulations_df.to_csv(os.path.join(analysis_path, 'ofat_analysis.csv'),␣
    ↪index=False)
445 
446 
```

Change the path to the csv file using the variable *path_to_csv* on line 28 to point to the csv *ofat_db.csv* from the previous simulation. Also update the path to which the results are expected to be stored by changing the variable *path_to_results* on line 29.

t_max_h in line 30 represents the maximum time to be plotted.

For each simulation the following plots are generated

1. A plot of the Na concentration profile as a function of time indicated in color scale. Saved in png and svg (vector) formats.
2. A plot of P_{mpp} as a function of time. Saved in png and svg (vector) formats. Additionally a csv file is generated with the data from the plot.
3. A plot showing the average concentration in each layer, as a function of time. Save in png format.

The output is saved within the *path_to_output* using the following structure

```
output_folder
|---batch_analysis
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+01Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+01Vcm_c.png
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+01Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+01Vcm_c.svg
```

(continues on next page)

(continued from previous page)

```
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+01Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+01Vcm_p.png
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+01Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+01Vcm_p.svg
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+01Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+01Vcm_s.svg
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+01Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+01Vcm_simulated_pid.csv
|   |---constant_source_flux_96_85C_1E+10pcm2_z1E-04ps_DSF1E-14_1E+02Vcm_h1E-12_
↪m1E+00_rt12h_rv-1E+02Vcm_c.png
|   |--- ...
|   |---ofat_analysis.csv
```

This is an example of a concentration plot

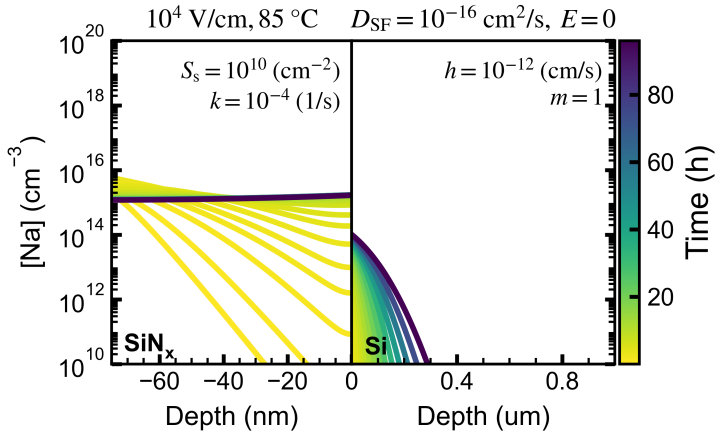

Fig. 1: Example of a concentration plot from the batch analysis.

This is an example of a P_{mpp} plot

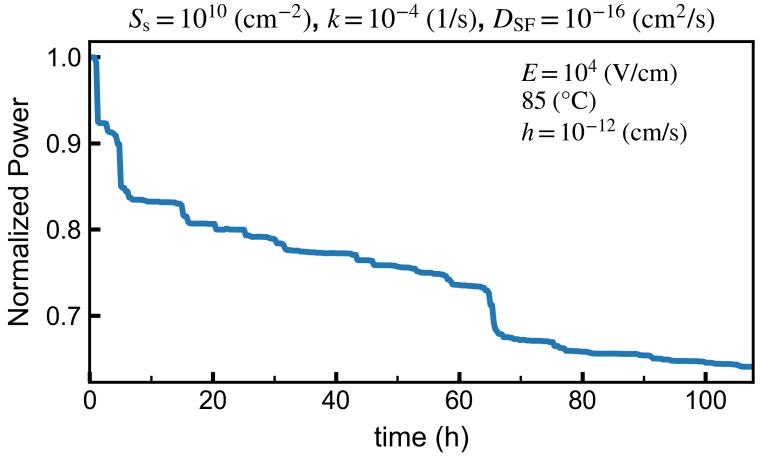

Fig. 2: Example of a P_{mpp} plot from the batch analysis.

CHAPTER 4

FEniCS Transport Simulations Code

4.1 Finite Source Simulation Code

4.2 Infinite Source Simulation Code

4.3 Utils

pnptransport.utils.**evaluate_arrhenius**(*a0: float*, *Ea: float*, *temp: float*) → float

Evaluate an Arrhenius variable

$$A = A_0 \exp\left(-\frac{E_\mathrm{a}}{k_\mathrm{B}T}\right)$$

Parameters

- **a0** (*float*) – The exponential prefactor
- **Ea** (*float*) – An activation energy in eV
- **temp** (*float*) – The temperature

Returns **x** – The evaluated variable

Return type float

pnptransport.utils.**fit_arrhenius**(*temperature_axis*, *y*, ***kwargs*)

Fits the experimental data to an Arrhenius relationship

Parameters

- **temperature_axis** ([*double*]) – The temperature axis
- **y** ([*double*]) – The dependent variable
- ****kwargs** –

  inverse_temp: boolean True if the units of the temperature are 1/T

temp_units: string The units of the temperature. Valid units are K and °C

pnptransport.utils.**format_pid_hhmmss_csv**(*path_to_csv: str*)

This function process the input csv so that a column with the time in seconds is added based on the input time column in hh:mm:ss

Parameters **path_to_csv** (*str*) – The pid csv file

pnptransport.utils.**format_time_str**(*time_s: float*)

Returns a formatted time string

Parameters **time_s** (*float*) – The time in seconds

Returns **timeStr** – A string representing the time

Return type str

pnptransport.utils.**geometric_series_spaced**(*max_val: float*, *min_delta: float*, *steps: int*, *reverse: bool = False*, ***kwargs*) → numpy.ndarray

Produces an array of values spaced according to a geometric series

$$S_n = a + ar + ar^2 + ... + ar^{n-2} + ar^{n-1}$$

For which $S_n = a(1 - r^n)/(1 - r)$

Here, a is the minimum increment (min_delta) and n is the number of steps and r is determined using Newton's method

Example:

```
import pnptransport.utils as utils

utils.geometric_series_spaced(max_val=3600, min_delta=1, steps=10)
# output:
# array([0.00000000e+00, 1.00000000e+00, 3.33435191e+00, 8.78355077e+00,
# 2.15038985e+01, 5.11976667e+01, 1.20513371e+02, 2.82320618e+02,
# 6.60035676e+02, 1.54175554e+03, 3.60000000e+03])
```

Parameters

- **max_val** (*float*) – The maximum value the series will take
- **min_delta** (*float*) – The minimum delta value it will take
- **steps** (*int*) – The number of values in the array
- **reverse** (*bool*) – If true solve for r = 1 / p
- ****kwargs** (*keyword arguments*) –

  n_iterations: int The number of Newton iterations to estimate r

Returns A vector with geometrically spaced values

Return type np.ndarray

pnptransport.utils.**get_indices_at_values**(*x: numpy.array*, *requested_values: numpy.array*) → numpy.ndarray

Constructs an array of valid indices in the x array corresponding to the requested values

Parameters

- **x** (*np.array*) – The array from which the indices will be drawn

- **requested_values** (*np.array*) –

Returns An array with the indices corresponding to the requested values

Return type np.array

pnptransport.utils.**latex_format**(*x*, *digits=2*) → str
Creates a LaTeX string for matplotlib plots.

Parameters

- **x** (*str*) – The value to be formatted
- **digits** (*int*) – The number of digits to round up to.

Returns The math-ready string

Return type str

pnptransport.utils.**latex_format_with_error**(*num*, *err*)
Parses a measurement quantity and it's error as a LaTeX string to use in matplotlib texts.

Parameters

- **num** (*float*) – The measured quantity to parse
- **err** (*float*) – The error associated error.

Returns The quantity and its error formatted as a LaTeX string

Return type str

pnptransport.utils.**latex_order_of_magnitude**(*num: float*, *dollar=False*)
Returns a LaTeX string with the order of magnitude of the number (10^x)

Parameters

- **num** (*float*) – The number
- **dollar** (*bool*) – If true, enclose string between $. Default False

Returns The LaTeX-format string with the order of magnitude of the number.

Return type str

pnptransport.utils.**tau_c**(*D: float*, *E: float*, *L: float*, *T: float*) → float
Estimates the characteristic constant for the Nernst-Planck equation in the low concentration approximation

$$\tau_c = \frac{L}{\mu E} + \frac{D}{\mu^2 E^2}\left[2 \pm \left(1 + \frac{qE}{kT}L\right)^{1/2}\right]$$

Since $\mu = qD/kT$

$$\tau_c = \left(\frac{L}{D}\right)X + \left(\frac{1}{D}\right)X^2\left[2 \pm \left(1 + \frac{L}{X}\right)^{1/2}\right],$$

with $X = kT/qE$

When $\mu E \tau_c$ is negligible, compared with the diffusive term $2\sqrt{D\tau_c}$, it returns

$$\tau_c = \frac{L^2}{4D}$$

Parameters

- **D** (*float*) – The diffusion coefficient in cm^2/s
- **E** (*float*) – The electric field in MV/cm = 1E6 V/cm
- **L** (*float*) – The distance in cm
- **T** (*float*) – The temperature in °C

Returns The characteristic time in s

Return type float

4.4 Confidence Intervals Methods

pnptransport.confidence.**confidence_interval**(*res: scipy.optimize.optimize.OptimizeResult*, ***kwargs*)

This function estimates the confidence interval for the optimized parameters from the fit.

Parameters

- **res** (*OptimizeResult*) – The optimized result from least_squares minimization
- ****kwargs** –

  confidence: float The confidence level (default 0.95)

Returns ci: The confidence interval

Return type np.ndarray

pnptransport.confidence.**confint**(*n: int*, *pars: numpy.ndarray*, *pcov: numpy.ndarray*, *confidence: float = 0.95*, ***kwargs*)

This function returns the confidence interval for each parameter

---

Note: Adapted from http://kitchingroup.cheme.cmu.edu/blog/2013/02/12/Nonlinear-curve-fitting-with-parameter-confidence-intervals/ Copyright (C) 2013 by John Kitchin. https://kite.com/python/examples/702/scipy-compute-a-confidence-interval-from-a-dataset

---

Parameters

- **n** (*int*) – The number of data points
- **pars** (*np.ndarray*) – The array with the fitted parameters
- **pcov** (*np.ndarray*) – The covariance matrix
- **confidence** (*float*) – The confidence interval

Returns The matrix with the confindence intervals for the parameters

Return type np.ndarray

pnptransport.confidence.**get_rsquared**(*x: numpy.ndarray, y: numpy.ndarray, popt: numpy.ndarray, func: Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray]*)

This function estimates R^2 for the fitting

Reference: http://bagrow.info/dsv/LEC10_notes_2014-02-13.html

Parameters

- **x** (*np.ndarray*) – The experimetnal x points
- **y** (*np.ndarray*) – The experimental y points
- **popt** (*np.ndarray*) – The best fit parameters
- **func** (*Callable[[np.ndarray, np.ndarray]*) – The fitted function

Returns The value of R^2

Return type float

pnptransport.confidence.**mean_squared_error**(*yd: numpy.ndarray*, *ym: numpy.ndarray*)

This function estimates the mean squared error of a fitting.

Parameters

- **yd** (*np.ndarray*) – The observed data points
- **ym** (*np.ndarray*) – The datapoints from the model

Returns The mean squared error

Return type float

pnptransport.confidence.**predband**(*x: numpy.ndarray, xd: numpy.ndarray, yd: numpy.ndarray, p: numpy.ndarray, func: Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray], conf: float = 0.95*)

This function estimates the prediction bands for the specified function without using the jacobian of the fit https://codereview.stackexchange.com/questions/84414/obtaining-prediction-bands-for-regression-model

Parameters

- **x** (*np.ndarray*) – The requested data points for the prediction bands
- **xd** (*np.ndarray*) – The experimental values for x
- **yd** (*np.ndarray*) – The experimental values for y
- **p** (*np.ndarray*) – The fitted parameters
- **func** (*Callable[[np.ndarray, np.ndarray], np.ndarray]*) – The optimized function
- **conf** (*float*) – The confidence level

Returns

- *np.ndarray* – The value of the function at the requested points (x)
- *np.ndarray* – The lower prediction band
- *np.ndarray* – The upper prediction band

pnptransport.confidence.**predint**(*x: numpy.ndarray, xd: numpy.ndarray, yd: numpy.ndarray, func: Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray], res: scipy.optimize.optimize.OptimizeResult, **kwargs*)

This function estimates the prediction bands for the fit (see https://www.mathworks.com/help/curvefit/confidence-and-prediction-bounds.html)

Parameters

- **x** (*np.ndarray*) – The requested x points for the bands
- **xd** (*np.ndarray*) – The x datapoints

- **yd** (*np.ndarray*) – The y datapoints
- **func** (*Callable[[np.ndarray, np.ndarray]]*) – The fitted function
- **res** (*OptimizeResult*) – The optimzied result from least_squares minimization
- **kwargs** (*dict*) –

  confidence: float The confidence level (default 0.95)

  simulateneous: bool True if the bound type is simultaneous, false otherwise

  mode: [functional, observation] Default observation

pnptransport.confidence.**predint_multi**(*x: numpy.ndarray, xd: numpy.ndarray, yd: numpy.ndarray, func: Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray], res: scipy.optimize.optimize.OptimizeResult, **kwargs*)

This function estimates the prediction bands for the fit

(See https://www.mathworks.com/help/curvefit/confidence-and-prediction-bounds.html)

Parameters

- **x** (*np.ndarray*) – The requested x points for the bands
- **xd** (*np.ndarray*) – The x datapoints
- **yd** (*np.ndarray*) – The y datapoints
- **func** (*Callable[[np.ndarray, np.ndarray]]*) – The fitted function
- **res** (*OptimizeResult*) – The optimzied result from least_squares minimization
- **kwargs** (*dict*) –

  confidence: float The confidence level (default 0.95)

  simultaneous: bool True if the bound type is simultaneous, false otherwise

  mode: [functional, observation] Default observation

Returns

- *np.ndarray* – The predicted values.
- *np.ndarray* – The lower bound for the predicted values.
- *np.ndarray* – The upper bound for the predicted values.

4.5 Transport HD5 Storage System

4.6 HD5 Storage

CHAPTER 5

PID Simulation Module

5.1 Rsh Approximation Model

5.2 Korol Conductivity Implementation

class pidsim.korol_conductivity.**KorolConductivity**

Bases: *pidsim.conductivity_interface.ConductivityInterface*

This class provides methods to map a concentration of Na atoms in Si to a conductivity value.

Example

```
from pidsim.korol_conductivity import KorolConductivity
import h5py

conductivity_model: KorolConductivity = KorolConductivity()
# Assume every Na atom contributes 1 conduction electron
conductivity_model.activated_na_fraction = 1.
# Get a concentration profile from a transport simulation
h5_path = './transport_simulation_output.h5'
# Get the profile at index 20
idx = 20
with h5py.File(h5_path, 'r') as hf:
    c = np.array(hf['/L2/concentration/ct_{0:d}'.format(idx)])
# Update the concentration profile in the model
conductivity_model.concentration_profile = c
conductivity_model.segregation_coefficient = 1.
conductivity = conductivity_model.estimate_conductivity()
```

__sodium_profile

The Na concentration profile in cm^{-3} .

Type np.ndarray

__activated_na_fraction
The activated fraction of Na atoms to compute the conductivity $0 < f < 1$.

Type float

__segregation_coefficient
Deprecated since version 0.1: This value represents the segregation coefficient of Na in the stacking fault assuming a mechanism driven by bulk diffusion + segregation at the SF. Use 1.0.

Type float

activated_na_fraction

concentration_profile

conductivity_model(*concentration: numpy.ndarray*) → numpy.ndarray
Implementation of the conductivity_model model.

Model simplifications

1. The Na to Si ratio in the stacking fault is obtained from the ratio between Na concentration and Si concentration in the bulk of a perfect crystal (does not consider the specific geometry of a stacking fault)
2. Conductivity is calculated based on depth-resolved Hall-effect measurements of mobility and carrier density in Na-implanted Si (Korol et al.)

Reference Korol, V. M. "Sodium ion implantation into silicon." Physica status solidi (a) 110.1 (1988): 9-34.

Parameters **concentration** (*np.ndarray*) – The sodium concentration in the Si bulk

Returns The conductivity_model profile

Return type np.ndarray

estimate_conductivity()

segregation_coefficient

5.3 Parameter Span

pidsim.parameter_span.**append_to_batch_script**(*filetag: str*, *batch_script: str*)
Appends an execution line to the batch script

Parameters

- **filetag** (*str*) – The file tag for the .ini configuration file to run.
- **batch_script** (*str*) – The path to the batch script to append to.

pidsim.parameter_span.**create_filetag**(*time_s: float*, *temp_c: float*, *sigma_s: float*, *zeta: float*, *d_sf: float*, *ef: float*, *m: float*, *h: float*, *recovery_time: float = 0*, *recovery_e_field: float = 0*, *d_sin: float = None*) → str
Create the file_tag for the simulation input file

Parameters

- **time_s** (*float*) – The simulation time in seconds.
- **temp_c** (*float*) – The temperature in °C

- **sigma_s** (*float*) – The surface concentration of the source, in atoms/ cm^2 .
- **zeta** (*float*) – The rate of ingress in 1/s
- **d_sf** (*float*) – The diffusivity at the SF in cm^2 /s
- **ef** (*float*) – The applied electric field in SiNx in V/cm
- **m** (*float*) – The segregation coefficient
- **h** (*float*) – The surface mass transfer coefficient at the SiNx/Si interface in cm/s
- **recovery_time** (*float*) – The simulated recovery time (additional to the PID simulations) in s. Default: 0.
- **recovery_e_field** (*float*) – The electric field applied under recovery (ideally with sign opposite to the PID stress) units: V. Default: 0 V

Returns The file_tag

Return type str

pidsim.parameter_span.**create_input_file**(*simulation_time: float*, *temperature_c: float*, *sigma_s: float*, *zeta: float*, *d_sf: float*, *e_field: float*, *segregation_coefficient: float*, *h: float*, *thickness_sin: float*, *thickness_si: float*, *base_concentration: float*, *er: float*, *t_steps: int*, *x_points_sin: int*, *x_points_si: int*, *out_dir: str*, *recovery_time: float = 0.0*, *recovery_e_field: float = 0*, *d0_sinx: float = 1e-14*, *ea_sinx: float = 0.1*, *d_sin: float = None*) → str

Creates an inputfile for the finite source simulation

Parameters

- **simulation_time** (*float*) – The simulation time in seconds
- **temperature_c** (*float*) – The simulation temperature °C
- **sigma_s** (*float*) – The surface concentration of the source in atoms/cm^2 zeta: float The surface rate of ingress of Na in (1/s)
- **d_sf** (*float*) – The diffusion coefficient in the SF in cm^2 /s
- **e_field** (*float*) – The electric field in SiNx in V/cm
- **segregation_coefficient** (*float*) – The segregation coefficient at the SiNx/Si interface
- **h** (*float*) – The surface mass transfer coefficient at the SiNx/Si interface in cm/s
- **thickness_sin** (*float*) – The thickness of the SiNx layer in um.
- **thickness_si** (*float*) – The thickness of the Si layer in um
- **base_concentration** (*float*) – The base Na concentration prior to the simulation.
- **er** (*float*) – The relative permittivity of SiNx
- **t_steps** (*int*) – The number of time steps to simulate
- **x_points_sin** (*int*) – The number of grid points in the SiNx layer
- **x_points_si** (*int*) – The number of grid points in the Si layer
- **out_dir** (*str*) – The path to the output dir
- **recovery_time** (*float*) – Additional time used to model recovery (s). Default: 0.

- **recovery_e_field** (*float*) – Electric field applied during the recovery process in V/cm. Default: 0
- **d0_sinx** (*float*) – The Arrhenius prefactor for the diffusion coefficient of Na in SiN_x , in cm^2 /s. (only used if d_sin is *None*)
- **ea_sinx** (*float*) – The activation energy of the diffusion coefficient of Na in SiN_x, given in eV. (only used if d_sin is *None*)
- **d_sin** (*float*) – The diffusivity of Na in SiN_x , in cm^2 /s. Default: *None*.

Returns The file_tag of the generated file

Return type str

pidsim.parameter_span.**create_span_file**(*param_list: dict*, *simulation_time: float*, *temperature_c: float*, *thickness_sin: float*, *thickness_si: float*, *base_concentration: float*, *er: float*, *t_steps: int*, *x_points_sin: int*, *x_points_si: int*, *out_dir: str*)

A wrapper for create_input_file that unpacks the values of the parameter span contained in param_list

Parameters

- **param_list** (*dict*) – A dictionary with the values of the parameters that are being varied. Must contain: - sigma_s: The surface concentration in atoms/cm^2 - zeta: the rate of transfer in 1/s - dsf: the diffusion coefficient of Na in the SF - e_field: The electric field in V/cm - segregation coefficient: The segregation coefficient at the SiNx/Si interface - h: The surface mass transfer coefficient at the SiNx/Si interface - recovery_time: The recovery time added to the simulation - recovery_e_field: The electric field applied under recovery
- **simulation_time** (*float*) – The simulation time in seconds
- **temperature_c** (*float*) – The simulation temperature in °C
- **thickness_sin** (*float*) – The thickness of the SiNx layer in um.
- **thickness_si** (*float*) – The thickness of the simulated SF in um
- **base_concentration** (*float*) – The bulk base impurity concentration for all layers in 1/cm^3 er: float The relative permittivity of SiNx
- **t_steps** (*int*) – The number of time steps to simulate.
- **x_points_sin** (*int*) – The number of elements to simulate in the SiNx layer.
- **x_points_si** (*int*) – The number of elements to simulate in the Si layer.
- **out_dir** (*str*) – The path to the output directory

Returns The name of the input file for the simulation.

Return type str

pidsim.parameter_span.**efield_plus_d_sin**(*csv_file: str*, *simulation_time: float*, *temperature_c: float*, *sigma_s: float*, *zeta: float*, *h: float*, *segregation_coefficient: float*, *dsf: float*, *er: float = 7.0*, *thickness_sin: float = 0.075*, *thickness_si: float = 1*, *t_steps: int = 720*, *x_points_sin: int = 100*, *x_points_si: int = 200*, *base_concentration: float = 1e-20*)

Generate the input files to simulate simultaneous variations in :math: *D_{mathrm{SiN}}* and :math: *E*.

Parameters

- **csv_file** (*str*) – The path to the csv file containing the parameter variations

- **simulation_time** – The simulation time (s).
- **temperature_c** – The simulation temperature in °C
- **sigma_s** (*float*) – The surface concentration :math: *S_0* in cm^{-2}.
- **zeta** (*float*) – The rate of ingress :math: *k* in s^{-1}.
- **dsf** (*float*) – The diffusion coefficient of Na in the stacking fault in cm^2/s
- **h** (*float*) – The surface mass transfer coefficient between the SiN_xfilm and the silicon layer. In cm/s.
- **segregation_coefficient** (*float*) – The segregation coefficient :math: *m*.
- **er** (*float*) – The relative permittivity of SiN: sub:*x*.
- **thickness_sin** (*float*) – The thickness of SiN: sub:*x*in um.
- **thickness_si** – The thickness of the silicon layer in um.
- **t_steps** (*int*) – The number of time steps to simulate.
- **x_points_sin** (*int*) – The number of grid points for the SiN: sub:*x*layer.
- **x_points_si** (*int*) – The number of grid points in the Si layer.
- **base_concentration** (*float*) – The base concentration to simulate (cm:sup: *-3*).

pidsim.parameter_span.**one_factor_at_a_time**(*csv_file: str*, *simulation_time: float*, *temperature_c: float*, *er: float = 7.0*, *thickness_sin: float = 0.075*, *thickness_si: float = 1*, *t_steps: int = 720*, *x_points_sin: int = 100*, *x_points_si: int = 200*, *base_concentration: float = 1e-20*)

Generates input files and batch script to run one-factor-at-a-time parameter variation

Parameters

- **csv_file** (*str*) – The path to the csv file containing the base case and the parameter scans to simulate: Format of the file

Parameter name	Base case	span
sigma_s	1E+11	1E10,1E11,…
zeta	1E-4	1E-4,1E-3,…
DSF	1E-14	1E-12,1E-14,…
E	1E4	1E2,1E4,…
m	1	1
h	1E-8	1E-8,1E-7,…

- **simulation_time** (*float*) – The total simulation time in s.
- **temperature_c** (*float*) – The simulation temperature in °C
- **er** (*float*) – The relative permittivity of SiNx. Default 7.0
- **thickness_sin** (*float*) – The thickness of the SiNx layer in um. Default: 0.075
- **thickness_si** (*float*) – The thickness of the Si layer in um. Default 1 um
- **t_steps** (*int*) – The number of time steps for the integration.
- **x_points_sin** (*int*) – The number of grid points in the SiN layer
- **x_points_si** (*int*) – The number of grid points in the Si layer.

- **base_concentration** (*float*) – The background impurity concentration in cm^-3. Default 1E-20 cm^{-3} .

pidsim.parameter_span.**sigma_efield_variations**(*sigmas: numpy.ndarray*, *efields: numpy.ndarray*, *out_dir: str*, *zeta: float*, *simulation_time: float*, *dsf: float*, *h: float*, *m: float*, *temperature_c: float*, *er: float = 7.0*, *thickness_sin: float = 0.075*, *thickness_si: float = 1.0*, *t_steps: int = 720*, *x_points_sin: int = 100*, *x_points_si: int = 200*, *base_concentration: float = 1e-20*)

Generates inputs for a combination of the initial surface concentrations and electric fields defined in the input. Every other parameter remains fixed.

Parameters

- **sigmas** (*np.ndarray*) – An array containing the values of the surface concentration to vary (in ions/cm^2)
- **efields** (*np.ndarray*) – An array containing the values of the electric fields to vary (in V/cm)
- **out_dir** (*str*) – The path to the folder to store the generated input files.
- **zeta** (*float*) – The value of the rate of ingress at the surface (1/s)
- **simulation_time** (*float*) – The time length of the simulation in seconds.
- **dsf** (*float*) – The diffusion coefficient of Na in the stacking fault.
- **h** (*float*) – The surface mass transfer coefficient at the SiNx/Si interface in (cm/s)
- **m** (*float*) – The segregation coefficient at the SiNx/Si interface
- **temperature_c** (*float*) – The temperature °C
- **er** (*float*) – The relative permittivity of the dielectric. Default 7.0
- **thickness_sin** (*float*) – The thickness of the SiNx layer in um.
- **thickness_si** (*float*) – The thickness of the Si layer in um.
- **t_steps** (*int*) – The number of time steps. Default: 720
- **x_points_sin** (*int*) – The number of mesh points in the SiNx layer. Default 100
- **x_points_si** (*int*) – The number of mesh points in the Si layer. Default 200
- **base_concentration** (*float*) – The background concentration at the initial condition in atoms/cm^3

pidsim.parameter_span.**sin_bias_from_e**(*e_field: float*, *thickness_sin: float*) → float

Estimates the bias in SiNx based on the value of the electric field and the thickness of the layer.

Parameters

- **e_field** (*float*) – The electric field in the SiNx layer (V/cm)
- **thickness_sin** (*float*) – The thickness of the SiNx layer in (um).

Returns The corresponding bias in V

Return type float

pidsim.parameter_span.**string_list_to_float**(*the_list: str*) → numpy.ndarray

Takes a string containing a comma-separated list and converts it to a numpy array of floats

Parameters **the_list** (*str*) – The comma-separated list

Returns The corresponding array

Return type np.ndarray

5.4 Conductivty Interface

class pidsim.conductivity_interface.**ConductivityInterface**
Bases: object

concentration_profile

estimate_conductivity()

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pidsim.conductivity_interface, 37
pidsim.korol_conductivity, 31
pidsim.parameter_span, 32
pnptransport.confidence, 28
pnptransport.utils, 25

Index

Symbols

__activated_na_fraction (*pidsim.korol_conductivity.KorolConductivity attribute*), 31
__segregation_coefficient (*pidsim.korol_conductivity.KorolConductivity attribute*), 32
__sodium_profile (*pidsim.korol_conductivity.KorolConductivity attribute*), 31

A

activated_na_fraction (*pidsim.korol_conductivity.KorolConductivity attribute*), 32
append_to_batch_script() (*in module pidsim.parameter_span*), 32

C

concentration_profile (*pidsim.conductivity_interface.ConductivityInterface attribute*), 37
concentration_profile (*pidsim.korol_conductivity.KorolConductivity attribute*), 32
conductivity_model() (*pidsim.korol_conductivity.KorolConductivity method*), 32
ConductivityInterface (*class in pidsim.conductivity_interface*), 37
confidence_interval() (*in module pnptransport.confidence*), 28
confint() (*in module pnptransport.confidence*), 28
create_filetag() (*in module pidsim.parameter_span*), 32
create_input_file() (*in module pidsim.parameter_span*), 33
create_span_file() (*in module pidsim.parameter_span*), 34

E

efield_plus_d_sin() (*in module pidsim.parameter_span*), 34
estimate_conductivity() (*pidsim.conductivity_interface.ConductivityInterface method*), 37
estimate_conductivity() (*pidsim.korol_conductivity.KorolConductivity method*), 32
evaluate_arrhenius() (*in module pnptransport.utils*), 25

F

fit_arrhenius() (*in module pnptransport.utils*), 25
format_pid_hhmmss_csv() (*in module pnptransport.utils*), 26
format_time_str() (*in module pnptransport.utils*), 26

G

geometric_series_spaced() (*in module pnptransport.utils*), 26
get_indices_at_values() (*in module pnptransport.utils*), 26
get_rsquared() (*in module pnptransport.confidence*), 28

K

KorolConductivity (*class in pidsim.korol_conductivity*), 31

L

latex_format() (*in module pnptransport.utils*), 27
latex_format_with_error() (*in module pnptransport.utils*), 27
latex_order_of_magnitude() (*in module pnptransport.utils*), 27

M

mean_squared_error() (*in module pnptransport.confidence*), 29

O

one_factor_at_a_time() (*in module pidsim.parameter_span*), 35

P

pidsim.conductivity_interface (*module*), 37
pidsim.korol_conductivity (*module*), 31
pidsim.parameter_span (*module*), 32
pnptransport.confidence (*module*), 28
pnptransport.utils (*module*), 25
predband() (*in module pnptransport.confidence*), 29
predint() (*in module pnptransport.confidence*), 29
predint_multi() (*in module pnptransport.confidence*), 30

S

segregation_coefficient (*pidsim.korol_conductivity.KorolConductivity attribute*), 32
sigma_efield_variations() (*in module pidsim.parameter_span*), 36
sin_bias_from_e() (*in module pidsim.parameter_span*), 36
string_list_to_float() (*in module pidsim.parameter_span*), 36

T

tau_c() (*in module pnptransport.utils*), 27