Numerical model for the simulation of thick film semiconduction metal oxide gas sensors

February 19, 2020

Dissertation
der Mathematisch – Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Peter Bonanati

Tübingen
2020

Tag der mündlichen Prüfung: xx.xx.2014

Dekan: Prof. Dr. Wolfgang Rosenstiel

1. Berichterstatter: Prof. Dr. Udo Weimar
2. Berichterstatter: Prof. Dr. Reinhold Fink

Jupyter for scientific research

February 19, 2020

Contents

1 Introduction

1.1 Motivation

In the beginning of my academical career I was aiming an educational degree i math and physics. While studying at the University of Tübingen I was also worked as a research assistant in the Institute of Physical Chemistry. In the course of my studies I was allowed to spend 4 month in a school to gain first experiences in teaching physics and math to students between 12 and 20 years. Even though it was a great experience and I enjoyed it a lot to bring new ideas and concepts to the students, I also felt a strong urge to continue learning and explore. At this time the possibilities I saw in focusing on scientific research and development seemed more attractive to me. My reasoning was, that I might be able to combine my preference for teaching and research rather at the university than a college. Long story short, after some years I ended up doing my Phd. in the "Institute of Theoretical and Physical Chemistry" at the University of Tübingen. Lucky

as I was I directly got the possibility to work in a project with an industrial partner. The focus was on building a new state of the art gas sensors. Besides the benefits of working in a great teams, having enough financial support for research activities and learning to present and report my results on a regular basis, it also imposed some new problems needed to be solved. One of the biggest challenges was the continuous increase of experimental data in always shorter intervals. With the increasing industrialization and automatization of sensor production and testing, the quantity of high quality data increased dramatically.

Luck as I was, I was not the only one facing problems with the increasing amount of data needed to be processed and analyzed. In the beginning of each analysis, the specific task was not fully defined. It was rather an exploratory research to gain an idea about possible hidden opportunities. In this case the the traditional way of creating well defined software algorithms for a specific tasks did not fit too well.

I rather had the need for efficient tools to reduce time consuming parts regarding the data analysis, which had been performed manually until now. I was looking for a platform to combine multiple of these tools efficiently. So at this point of my career with my very basic knowledge in the area of programming/coding and the need to solve urgent tasks, I was looking for an efficient solutions with a steep learning curve.

Until then my standard procedure (and some others in our lab) of working with data was mainly based on manual feature extraction and analysis. When Working with a limited amount of samples, manually doing these steps was acceptable and efficient. With the industrial cooperation accelerated, the number of samples increased also rapidly. Soon we reached the point where the pre-processing of the data would easily consume a large amount of time. And that without even having started with a deeper analysis. Luckily I was not the only one facing such problems and a project from the Python community gained more and more interest. Articles like [Unp14] pointed me to a new way of dealing with data and indicated, that the Python programming language might be very helpful. Regarding the fact, that Python was already a well established programming language, the introduction of the "IPtyhon notebook ecosystem" was making the use of Python for in an scientific work flow very attractive. As mentioned in this article: [Dav13]

> "... what they do offer is an environment for exploration, collaboration, and visualization."

I also realized the large potential for my working field. By learning Python I got efficient tools for calculations, analysis and representations. Additionally the new tools have been build specially with the focus on the goal to easy report result including the way they have been gained. The environment around the so called "Juypter notebooks" was the ideal piece, which I experienced as a missing block in the scientific work I was doing.

Besides my work for our industrial partner I also did fundamental research about semiconducting metaloxide gas sensors. Based on great research done before my time on gas sensors, my focus was now on numerical calculations of the gas sensors. Typically a theoretical model of a gas sensor is developed and the predictions are compared with experimental results. When publishing the results the peer review system assures, that the publications are well written, documented the on a solid and proven basis. But when reading such papers, I had the experience, many of them presented excellent ideas, but unfortunately practical instructions on how to implement the presented models were often not given.

The work of rebuilding the model and recalculating the results was therefore often not possible

in a reasonable amount of time. In my experience this limits the direct comparison of new model with experimental data from other sensors. It is also worth to mention, that in my experience the average experimental oriented researcher does not have the required programming knowledge to easily implement algorithm based on the presented work. But I am confident, that if an algorithm was given in an appropriate way, most research would benefit from the presented code.

With the results I gained while my Phd. thesis, I was facing the same problem. Others understood the value of my work but could not transfer it to their particular problem. At this point my graphical representation in the form of presentations, my detailed description and the corresponding algorithms of my work had been three separated parts. An appropriated way of representing those results in unified matter did not exists yet. Facing this problem more and more, I decided not to ignore it anymore. I made the choice to try to combine representation, description and algorithm in one unified way.

I decided to orient myself back to my "educational roots" and use the powerful ecosystem around the "Jupyter notebooks" to calculate, represent and describe the results for my thesis for this task. This is why my thesis is written with the large focus on suppling a introduction to this excellent toolbox.

This work will (hopefully) allow more people to gain insides in the sensing properties of semi conducting metal oxide sensors (SMOX).Additionally this thesis is intended to be a useful introduction into Python, specially IPython notebooks, for scientific work.

These hopes are not unfunded. In my time working at the "Eberhard Karls University of Tübingen" i was also assigned to give multiple lectures and classes. One lecture branch has been the introduction to data mining. Often the lack of programming knowledge and the limited amount of lecture time did not allow a usage of Python as a supporting programming language. In these cases classical tools like Excel or Origin have been used to analyze example datasets for hidden facts. Most attendees have been very fast and understood the general concept of data mining. Simply not being able to translate the general concept into machine understandable instructions stopped them from using more advanced tools.

In cases, where enough time was available, a short introduction into programming with Python gained a lot of interest and was generally seen as a positive experience. With just a short introduction already many advanced tasks can be performed, which would often even have been possible with the "traditional" tools.

This thesis is therefore structured in such a way, that I will present my research results from the past years in a condensed form. Additionally I will use this opportunity to introduce and explain the importance of applying programming tools in the common work flow of scientific work.

The potential of "outsourcing" repetitive tasks to machine executable scripts is huge. By focusing on investing more time in creativity and intelligence solutions, the overall quality of the results will increase. My hope is to bring with this thesis not only a deeper insight in the understanding of SMOX based gas sensors, but also help others to start a interesting journey into the wide area of data mining and machine learning with python.

The thesis is structured in three major "work packages", each of them dealing with discrete elements of a typical scientific tasks. In a condensed form the following areas will be covered:

1. Solving differential equations and integral numerically
2. Using numerical models for simulation

3. Fitting numerical results with experimental data

I typically finish my introductions to Python by letting the students run their first commands. For other introductions found around the globe, this is a program which outputs "Hello World.". For Python I prefer to execute an other commands with boils almost down to the philosophical essence on how "instructions" should be. In a "Jupyter notebook", the next cell represents a "code block" and can be executed along the text of the document. The output of the executed code is then visible along with the text of the document.

```
[1]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```


1.2 Jupyter Notebooks

"The Zen of Python" might not always be the primary directive of each developer, but the Python community consists most probably of many people how would consider the latter points as important. So did also the inventors of the IPython and Jupyter. A quick search will reveal multiple sources in the world wide web giving a detailed picture about what "IPython Notebooks" are and how Jupyter in connected with them. Here I will not try to give an general overview about this

tool and rather stick to the phrase "Learning by doing.". A minimal understanding of working with Python will definitely be helpful for this thesis, but in not necessarily required. I will try to explain the used features this notebooks to guide an interested reader to the point where:

- understanding the fundamental instructions of Python
- using the basic functionality of the Notebooks
- fundamental understanding of SMOX based gas sensors

is gained.

It is worth mentioning, that the intention of such notebooks is to merge the essential tools of scientific work flows together. Data acquisition, preparation, analysis, representation and documentation all available in one place. The strength of not just sharing final conclusions in a nicely formatted way, but also being able to share the full stack of steps necessary to reach the final conclusion is essentially the strength of the Jupyter notebooks. This feature is already changing the way how scientific results are shared/published and was intentionally designed this way [RPGB17].

The default format of representing anything in a notebook is based on "Markdown". Wikipedia summarizes Markdown like this:

> Markdown is a lightweight markup language with plain text formatting syntax. Wikipedia

This means a document is formatted by writing plain text and special text blocks are interpreted as formatting commands. E.g. **BOLD** letters are generated by encapsulating the text with `** Text here **`, headings are generated by starting the heading with #. Depending on the number of #, subsections are created. I will not go into detail here about the features of Markdown. Many features are used in this notebook and are directly accessible by double-clicking the text element. By doing this, the formatted text will switch back to the "plain text representation" and will reveal the way it was created. By again executing the cell with `CTRL+ENTER`, its Markdown formatted representation is again rendered.

One other handy feature of the Jupyter ecosystem I use is the ability to transform notebooks into multiple other formats. Just to name some: HTML, WORD (DOC, DOCX), Latex. The tool `nbconvert` is used internally to convert the Markdown formatted representation into other formats. For this thesis the default option "Export as PDF" under the File option generates a Latex based PDF file. Mastering-markdown is a web page, where I found some hints on how to format my notebook. This example on web page demonstrates for instance how to generate block quotes:

As Kanye West said:

> We're living the future so the present is our past.

To learn how to use notebooks it is best to use them in an interactive environment. The next section will explain how to obtain one for free!

1.2.1 Installation guide

The easiest way to get started would be to use the Anaconda distribution. Anaconda bundles multiple different tools and installs them in the operation system. Anaconda will take care of cross dependencies and handle the update process of the software. This is not the only way to get

started with "Jupyter Notebooks" but surely an very fast and easy one. HERE are the slides I use for my lectures to guide students into the world of Python and here one example of it's usage.

1.2.2 Example Notebook - Sneak preview

Besides the first example `import this`, here is a very basic example to demonstrate some features of the notebook.

"Simple is better than complex." The Jupyter environment is equipped with "magic commands", which are not part of the base programming language (i.e. Python), but rather a helper instructions to simplify common tasks. Magic commands always start with `%` and are followed with an instruction. I will demonstrate in this example the use of the `%pylab inline` instruction. This modifies the current programming space to become a lab nicely equipped for scientific work tasks. For instance like a chemistry lab is commonly equipped with a balance, a water tab and a fire extinguisher, in this case a "pylab" is equipped with a data handing, a plotting and a calculation tool (besides many others). The additional parameter `inline` makes sure, that the figures will be along with this document. So let's setup a "pylab" and run some lines of code. (The plotting is handled in the background by Matplotlib [Hun07])
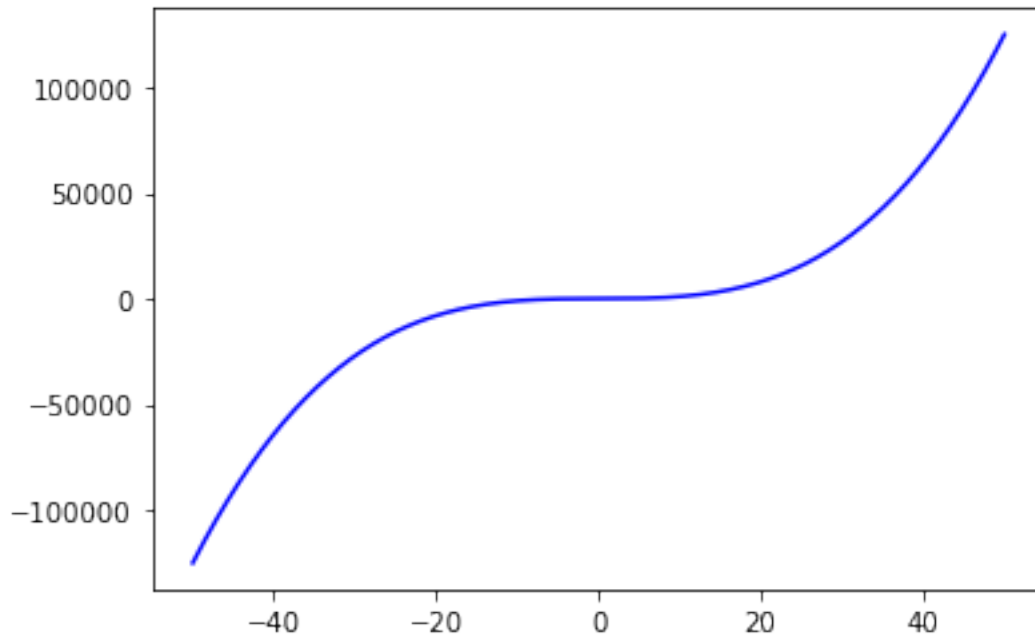
```
[3]: %pylab inline

     #get a list of 5000 points between -50 and 50
     xs = linspace(start=-50, stop=50, num=5000)

     # Python way of adding a comment is the '#' character
     # Python way of writer 'x to the power of y' is x**y
     # here I am calculating thethe third power of x for 5000 points between -50 and
      ↪50
     ys = xs**3

     #plot it
     fig = plt.plot(xs,ys,'b')
```

```
Populating the interactive namespace from numpy and matplotlib
```

The `linspace` function generates a 5000 linear distributed points in the interval [-50,50) and saves those points in the xs variable. ys is a list holding the third power of each point in xs. `plt.` is a submodule defined by the magic command `%pylab` which handels data plotting in a very simple way. `plt.plot(x,y,'r-')` for example plots x vs. y with a red line. The thesis is done in the notebook to offer the reader the possibility to directly work with newly gained knowledge. Therefore the upper block is a good opportunity to do the fist steps in Python. For instance change the line format to 'b' (blue). Or to 'b-.'. Change the exponent from 3 to 3.1. To do so:

- click on the code box above
- edit the code
- Add the following line `plt.title('The power law')`
- go back with CTRL-Z to correct your mistakes
- hit CTRL-ENTER to execute the code

1.3 SMOX based gas sensors

In this section a very short summary about **S**emi conducting **M**etal **OX**ide (SMOX) based gas sensors is provided. Relevant aspects for this thesis will be presented in a general way and additional literature references are be provided. The intention is to provided the minimum amount of information needed to follow the this work. The supplied references are a good staring point to gain a detailed understanding of SMOX based gas sensors.

1.3.1 SMOX material

A typical definition of a sensors describes two elements. The receptor and the transducer. The receptor interacts with the external stimulus, while the transducer translates this stimulus into a measurable response. In the case of SMOX sensors those two "parts" can't be separated directly,

since the SMOX material takes over the role of both. Generally the gas sensing with SMOX materials is based on the interaction of target molecules at its surface and a resulting change of the semiconductor itself [BW03].

Nevertheless the chemistry at the surface, like the adsorption/desorption of molecules, may be seen as the receptor of the sensor. In the process of adsorption/desorption at the surfaces, typically charges from the inside of the material are involved. In general this results in a modification of the charge distribution inside the semiconductor. As a consequence of an increase/decrease of the free charge carrier concentration the overall resistance of the material changes. The processes related to changes inside the semiconductor and hence the resistance, could be associated to the "transducer" part of the sensor. A detailed description of the physical and chemical properties that make tin oxide a suitable material gas sensing is described here:"[BD05].

For most SMOX the intrinsic electrical conductivity is very low due to a high band gab of the semiconductor. Typical preparation methods of the SMOX material generate defects inside the crystal structure due to missing oxygen atoms. For SnO_2, these defects act as additional electron donors and increase the number of free charge carriers and hence increase the conductivity.

Typically density of additional donors N_D of acceptors N_A plays a significant role in the sensing properties. The availability of additional free charge carries resulting from such defects depend also on the overall temperature of the sensor. Typical operation temperatures are around 300°C, but also higher and lower operations temperatures may be suitable depending on the use case. Besides effecting the semiconductors properties, the reactions at the surface of the sensor are also strongly dependent on the temperature.

1.3.2 SMOX based thick film sensors

The model of this work focuses on describing thick film SMOX gas sensors. In the case of such sensors the SMOX material is deposited as an porous thick film on top of a electric structure suitable to measure the resistance of the film. The film itself consists of multiple nano-sized grains in direct contact with each other. To measure the resistance of the film, a bias voltage is applied and the resulting current which passes through the network of multiple grains is measured. Depending on the percolation path, the total resistance of the sensor R_{Sensor} can be approximated as the product of a geometrical factor G associated to the film morphology and the resistance of one grain R_{Grain} as:

$$R_{Sensor} = G * R_{Grain}$$ (Resistance of sensor)

The following sketch represents the typical setup of a thick film sensor:
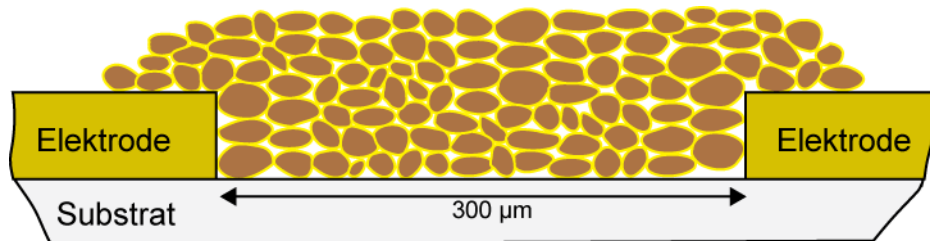
1.3.3 Surface potential

To better understand the processes related to the changes in the free charge carrier concentration inside the semiconductor, the semiconductor is typically described in a energy bands representation. For instance for SnO_2, trapping electrons at the surface results in an upward bending of the conduction band. The position of the conduction band at the surface is typically expresses in units $E_S = -eV_S$ (e is the charge of an electron). V_S is called the surface potential. In regard to a non-bended conduction band, the number of electrons are reduced in regions, where the band in elevated.

Experimentally the Kelvin probe method can be used to measure the change in the surface potential ΔV_S. The experimental setup and the requirements to obtain the value of ΔV_S are described in detail here: [OBW09]

The relation between ΔV_S and the resulting change in the resistance of the semiconductor R is then only determined by the semiconductor. The relation between ΔV_S and the resistance R of the semiconductor is defined independently of the actual surface chemistry. The relation depends strongly on the temperature T, the defect concentration N_D and the shape of the underlaying material. This work focuses on the deriving a model to predict the theoretical value of $\Delta R_{V_S} = \frac{R_{V_S}}{R_{V_S=0}}$ from a change in the surface potential ΔV_S.

With the definition of (Resistance of sensor) the value of ΔR_{V_S} of one grain can be directly calculated by the ratio of the experimentally measured value of the sensors R_{Sensor}.

$$\frac{R_{Sensor_{V_S}}}{R_{Sensor_{V_S=0}}} = \frac{G * R_{Grain_{V_S}}}{G * R_{Grain_{V_S=0}}} = \Delta R_{V_S} \tag{1}$$

1.3.4 Numerical model

Based on the example of SnO_2 as SMOX material, a numerical model describing effects of shape, size, defect concentration and temperature is developed for spherical grains. With such a model multiple conclusions about the dependency of these parameters on the sensitivity and performance of a sensor can be derived. Additionally the numerical model is able to provide the number of charges involved in the sensing process. This number is a important parameter involved in the description of the chemical reaction at the surface.

When solving this problem numerically, some assumptions will need to be introduction. These assumptions simplify the problem to a level where a numerical calculation is possible, but will reduce the validity of the results only to a small subset of all possible situations. The calculations in this thesis are also packed with assumptions and boundary conditions. My intention is to supply enough information to understand the relevance of the assumption and it's implications along with the supplied code. The way of presenting this knowledge should lead/motivate others to adapt the presented work to individual other cases with different boundary conditions.

2 Conclusion

Since the motivation for this "interactive" thesis should be clear now, I would like to come in the next section now to my actual research topic: "Numerical calculation of semiconducting metal oxide (SMOX) based gas sensors". In the next chapter I will demonstrate how theoretical numerical

calculations are used to for chemical sensors are used to better understand experimental results. In the following chapter I will demonstrate how such knowledge could be used to improve the performance of a sensor regarding it's sensitivity and selectivity.

Follow this link to come to the next section.

3 About the PDF-Version of this work

This notebook was not intended to be used as a printed hard copy or as a PDF. Its main intention is to serve as an easy entry point to Python supported science. Many of the implemented features in these notebooks like interactive widgets, animated data representations and live code examples will not work in the PDF-version. Only a static snapshot of the mutable representation can be represented outside the notebook, at best. The benefits of interacting with the presented work in a notebook should motivate the reader to use the notebook.

Nevertheless the integrated function to export a notebook to a latex based version offers a very nice way to publish results in a printable way. So please keep in mind, that the PDF-verison may not be able to represent all the features of the notebook as intended and some links might not work as expected. You are strongly encouraged to switch to the Jupyter presentation of this work and experience the full potential of such notebooks.

3.1 Equations

In a typical scientific thesis and textbook, relevant equations are referred by numbers of identifiers. In Latex this is done, by assigning a label and a tag to an equation. If this equation needs to be referred to, a pointer to the reference is added, and the tag is used for the representation of this equation. As an example, an arbitrary equation from this thesis is used. The (internal) reference of this equation is 'example', while the printed representation (tag) is "Example Equation".

$$\frac{dV^*}{dr^{*2}} = 1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*} \qquad \text{(Example Equation)}$$

To refer to this equation a reference can be added which will look like this (Example Equation). The underlying mechanisms will link the reference with the equation, print the label, and add an hyperlink allowing to directly jump to the equation. This feature works either in notebook or PDF representations. One drawback of the notebook representation is, that the equation and reference need to be in the same code block/cell. Otherwise the reference is not working. Instead of linking the equation correctly, ??? will be represented. My opinion is, that this will change with future versions of Jupyter. Since the PDF version is processed by a Latex interpreter in total, the "one code block" limitation does not exist there. To demonstrate this, I will try to reference the equation in the next code block.

Here the same reference to the equation: **??**. In the PDF-Version this will work correctly and in the Notebook-Version only ??? should be visible (until now).

3.2 Tables

With the numerical calculations performed in this thesis, data will need to be represented also. One way of doing this is by plotting the the data. This is also suitable for a static PDF-Version

of this thesis, which often is useful/necessary to have. This feature is well implemented and the transfer from Notebook to PDF version works well. Another way to display data are tables. Similar to the well known, omnipresent tool for working with tables, Excel, the Python universe has its own, but similar, tool. It's called Pandas. As a primer I will give here a very short introduction to Pandas. What sheets are for Excel, are Dataframes for Pandas. Here a simple example how to create a Dataframe in analogy to the previous programming example:

```
[6]: %pylab
     import pandas

     #create some x values
     x = linspace(start=0,stop=5,num = 10)

     #the y values will the the square of the x values
     y = x**2

     #Put them in a Dataframe and reference this new Dataframe with the variable `dF`
     dF = pandas.DataFrame({'x':x, 'y':y})
```

```
Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

This simple example of a Dataframe should demonstrate the basic concept of Dataframes. Once data is in the `DataFrame` format, the are infinit ways to transform/slice/merge/... it, to bring it into the desired shape. Along this thesis, some of the functionalities of Dataframes will be used and explained. Since the Jupyter environment is still "under construction", the transfer from Juypter to PDF for notebooks does not work to well until now. This does not mean, that is is not possible, it is just not yet implemented in the default/vanilla environment. This is way I want to highlight the small tweak that is at this point still needed to have a comparable output in Juypter notebooks and printed PDFs. To bypass this obstacle the default behavior of pandas needs to be altered (similar to this post: Latex-Tables Monkey Patch). This following patch brings DataFrames in the appropriate shape to by nicely represented in both representations.

3.2.1 Before the patch

```
[7]: display(dF)
```

```
            x          y
0    0.000000   0.000000
1    0.555556   0.308642
2    1.111111   1.234568
3    1.666667   2.777778
4    2.222222   4.938272
5    2.777778   7.716049
6    3.333333  11.111111
7    3.888889  15.123457
8    4.444444  19.753086
9    5.000000  25.000000
```

3.2.2 The patch

```
[8]: import pandas
     pandas.set_option('display.notebook_repr_html', True)

     def _repr_latex_(self):
         return r"""
         \begin{center}
         {%s}
         \end{center}
         """ % self.to_latex()

     pandas.DataFrame._repr_latex_ = _repr_latex_   # monkey patch pandas DataFrame
```

3.2.3 Prettier tables after the patch

```
[9]: display(dF)
     #"Test Tabel"
```

	x	y
0	0.000000	0.000000
1	0.555556	0.308642
2	1.111111	1.234568
3	1.666667	2.777778
4	2.222222	4.938272
5	2.777778	7.716049
6	3.333333	11.111111
7	3.888889	15.123457
8	4.444444	19.753086
9	5.000000	25.000000

In Jupyter notebooks the output will look very similar. In the PDF version of this thesis, those two output will differ(, unless a newer version of Jupyter fixed this issue.) To have this thesis in a PDF printable form, I will use the presented patch to unify the output. On the other side I suggest not to use this patch, and rather work with the Jupyter notebooks and its default representation, since the patch has also its downsides. But this would go beyond the scope of this remark about the patch.

4 Bibliography section

References

[BD05]   BATZILL, Matthias ; DIEBOLD, Ulrike: The surface and materials science of tin oxide. In: *Progress in Surface Science* 79 (2005), Nr. 2-4, S. 47–154. http://dx.doi.org/10.1016/j.

progsurf.2005.09.002. – DOI 10.1016/j.progsurf.2005.09.002. – ISBN 0079–6816

[BW03]   Bârsan, N. ; Weimar, U.: Understanding the fundamental principles of metal oxide based gas sensors; the example of CO sensing with SnO2 sensors in the presence of humidity. In: *Journal of Physics Condensed Matter* 15 (2003), Nr. 20, R813–R839. http://dx.doi.org/10.1088/0953-8984/15/20/201. – DOI 10.1088/0953–8984/15/20/201. – ISBN 0953–8984

[Dav13]   Davenport, Thomas H.: *The Rise of Data Discovery*. https://www.datanami.com/2016/05/04/rise-data-science-notebooks/. Version: 2013

[Hun07]   Hunter, John D.: Matplotlib: A 2D graphics environment. In: *Computing in Science and Engineering* 9 (2007), may, Nr. 3, 99–104. http://dx.doi.org/10.1109/MCSE.2007.55. – DOI 10.1109/MCSE.2007.55. – ISSN 15219615

[OBW09]   Oprea, Alexandru ; Bârsan, Nicolae ; Weimar, Udo: Work function changes in gas sensitive materials: Fundamentals and applications. In: *Sensors and Actuators, B: Chemical* 142 (2009), Nr. 2, 470–493. http://dx.doi.org/10.1016/j.snb.2009.06.043. – DOI 10.1016/j.snb.2009.06.043. – ISSN 09254005

[RPGB17]   Randles, Bernadette M. ; Pasquetto, Irene V. ; Golshan, Milena S. ; Borgman, Christine L.: Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study. In: *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries* (2017). http://dx.doi.org/10.1109/JCDL.2017.7991618. – DOI 10.1109/JCDL.2017.7991618. – ISBN 9781538638613

[Unp14]   Unpingco, José: *Python for signal processing: Featuring IPython notebooks*. Bd. 9783319013. Cham : Springer International Publishing, 2014. – 1–128 S. http://dx.doi.org/10.1007/978-3-319-01342-8. http://dx.doi.org/10.1007/978-3-319-01342-8. – ISBN 9783319013428

This page is intentionally left blank.

Solving the poisson equation for spherical grains

February 19, 2020

Contents

1 Abstract

A semi conducting metal oxide (SMOX) based gas sensor translates chemical reactions at the surface into a change of resistance. Free charge carriers being trapped or injected in the material due to the surface reactions, result in a change in resistance. While the specific chemistry at the surface is by itself a large research topic being actively investigated, the effect inside the grain

can be investigated, to some extend, separately. Even if is not possible to fully describe a SMOX sensor without combining the (chemical) effects related to the surface reactions with the processes inside the semiconductor such a work will still provide new insights. The topic of the next two chapters of this work will be focusing on deriving such a numerical model describing describing the processes happening inside the semiconductor. By applying well known theoretical models about semiconductors to the SMOX sensor, the influence of the material and it's shape will result in a better understanding of the transduction of chemical reactions into measurable resistances. Besides the resistance of the semiconductor, it is also possible to measure with the Kelvin Probe method one additional parameter crucial for a deeper understanding: the work function of the semiconductor. The goal of this thesis is to provide a tool, with which experimental data from simultaneous work function and resistance measurements and be analyzed in detail to gain a better understanding about the overall processes involved in sensing with SMOX materials.

2   Motivation

The research on semiconducting metal oxide gas sensors was focusing in the past mostly on scenarios, where oxygen is the most dominant reactive, gaseous, species in the proximity of the sensor. The adsorbed oxygen at the surface of the semiconductor lead to an interaction with the charges inside the thick film grains. Due to the trapping of the charge carrier by oxygen, a depletion layer is formed in the surface region of the semiconductor. Based on this depletion layer assumption, many investigations have successfully lead to a deep understanding of the sensing mechanism.

Nevertheless,the existence of an depletion layer is not always valid. Recent experimental results have shown, that even under atmospheric conditions which are common in real live, the dominant impact of oxygen may be gone. It could be shown, that under conditions of 50% r.h. and low concentrations of CO (<100 ppm) in synthetic air, the depletion layer is not present anymore and a accumulation layer manifests. The following figure shows the experimental results from such a measurement. In the beginning of the experiment a SnO_2 based gas sensor was exposed to pure nitrogen. For this sensor it can be assumed, that under this condition no band bending is present. This means, that the resistance under nitrogen corresponds to the flat band situation. An increase of the resistance indicates the presence of a depletion layer since less free charge carriers are available for the conductions. A decrease of the resistance would indicate the presence of an accumulation layer. In the following figure we clearly see, that a conduction band switch is present. A detailed description of the finding can be found here [BRW15].
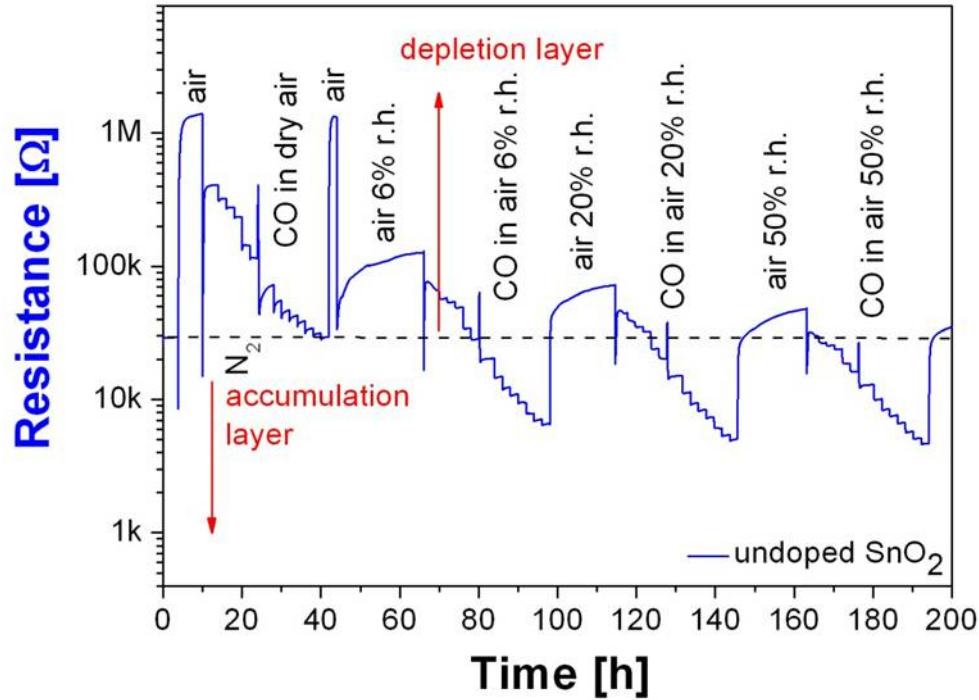
With the absence of the depletion layer also most simplifications are not valid anymore. Mainly the validity of the Schottky and Boltzmann approximation may not be given anymore. Facing those facts the equations to describe the transduction mechanism for a specific electrical band configurations needed a more general descriptions, which includes a depletion and accumulation layer controlled transduction mechanism.

To describe such a semiconductor the set of differential equations is available in literature. Common simplifications for such problems have not been valid in the generalized case needed to describe the sensor and finding an analytically solution exceeded by far my intellectual capabilities. This is why a I chose to go a different path by numerically deriving solutions for the problem.

While working in the field of SMOX sensors already some years, I was used to describe transduction processes by assigning different parts of an analytical solution to properties of the sensor. With a numerical solution this is not possible anymore and this was certainly a drawback of this method. On the other side the relation between intrinsic properties can still be studied also in detail. By solving the equations numerically for multiple combinations of the intrinsic property the resulting dataset can then again be used gain insights about the fundamental relations of the parameters. Also the comparison of the numerical results with experimental data might reveals properties, which can not be measured easily otherwise. Therefore the goal was first to break the problem of describing a SMOX sensor into smaller discreet parts and second trying to solve each of it individually.

In the upcoming chapters I will describe the different part and how they have been simplified, solved individually and combined again. In last chapter of the thesis, experimental data is compared with the numerically gained results.

3 Numerical calculation of semiconductors gas sensors

3.1 Introduction

To elaborate the modeling of sensing, equations such as the shape dependent Poisson equation, the electro-neutrality equation and the geometry dependent electrical current path, must be solved. In most cases, this is an extensive mathematical effort and therefore, the numerical computing environments Python will be used to derive numerical solutions for equations, which cannot be solved analytically, as shown below. From literature [RK04] the grain size dependency on the charges trapped at the surface has different impact on the potential and charge distribution inside the grain. For large grains compared to their Debye length (L_D), a charge transfer at the surface may be leave the bulk unaffected. In contrast to large grains, relative small grains may be affected through the whole grain. This effect will be the major outcome of the chapter and a graphical representation is shown at the end.

To understand the influence of the charge transfer at the surface on the resulting charge density inside the grain is from main importance. With this knowledge the free charge carrier concentration and a position dependent resistivity can be defined.

When investigating the total resistance of one grain, the pathway through the grain plays a major role. The total resistance may vary a lot based on the an-isotropic resistivity distribution.

3.2 Semiconductor properties of the SMOX grains

The advantages of industrialized production techniques are inline with general advantages of the SMOX-based sensor technologie. Both are:

- upscalable

- highly reproducible

- low cost

Besides the benefits for the industrialization of such a material, also the resulting morphological are beneficial for a good sensor performance. The typical spherical grains with a narrow size distribution result in high surface to volume ration. Besides the high surface area, the high number of grain-grain contacts have a positive effect on the sensitivity of the sensor, due to the high number of back to back Schottky barriers. As described in [BW03] [BHW11] such barriers are of major importance for the sensing properties.

In the literature other geometries claim exceptional performances for multiple other shapes. As from hollow spheres to nano-rods, often the mechanism which explains the measured increase in performance is not explained. Since the aim is to gain a fundamental understanding of the shape influence while staying close to a industrialize material I will keep the focus in this work on spherical grains, which are commonly used in commercial products. Nevertheless the techniques described in my work should be transferable to arbitrary and more complex shapes.

Besides the shape also the defect concentration and stochastic composition varies a lot with the preparation process. One additional goal of this thesis is to gain a better understanding about the relation of these two properties and the sensor performance.

3.3 Choice of geometric model

Most SMOX grains can be well approximated as spherical grains. The typical diameters of the grains are from 5nm to 200nm. The benefit by choosing of such a shape with a the rotational symmetry is the reduction of the complexity for the numerical calculation. Therefore the approximation of the SMOX particles as spheres was chosen. The second benefit of choosing materials prepared by rather standardized preparation routes is the availability of multiple different materials form varying laboratories around the world. These materials may vary in sizes and defect concentration but are often similar in shape. This fact is favorable when the numerical results are compared and validated with experimental data.

Other available materials with more specialized shapes as hallow spheres or fibers do exist, but will not be investigated in the research. In the first place the complex numerical description of such geometries will increase the calculation duration. Also the limited availability and variety in parameters as diameter, doping level, band gap, material composition are not favorable for to check the numerical model with experimental data.

3.4 Poisson's equation

Reactions at the surface (see [sec:Surface-reactions]) result in a charge transfer between the bulk of a grain and the surface. This modification of the charge density distribution inside the grain causes again a change in resistivity . By moving charges to/away from the surface, electrical potentials through the grain are generated. The electrical potential at the surface and therefore the work function of the semiconductor changes (see [subsec:Work-function]). A detailed description of work function measurements with the Kelvin Probe method is described here [OBW09].

Previous studies have described the direct relation between surface potential, surface charge and resistance exists. The latter studies initially define certain approximations which have been adapted and reasonable for the investigated cases, but do not allow a predictions outside the boundaries of the pre-assumption. Also the direct impact of size and geometric on the transduction is not taken fully into consideration. In order to have a more general model of the SMOX materials and to include the geometric effects, the charge distribution has to be solved in a more general way.

Identical to the previous studies the relation between surface potential and charge distribution has to be solved initially. This relation is defined by the Poisson-Law:

$$\nabla \phi = -\frac{\rho}{\epsilon \epsilon_0} \qquad \text{(Poisson)}$$

ϕ=electrical potential, ρ=free charge density, ϵ=vacuum permittivity, ϵ_0=relative permittivity.

It is assume that ϵ does not vary inside the grain. The charge destiny ρ on the other side is directly influenced by the charge transfer. Since the transfer of charges to the surface influences the work function and the energetic position of the conduction band, ρ is a function of ϕ. ϕ again depends on the position in the grain. At the surface $\phi(r = r_S)$ corresponds to the surface potential ϕ_s, while in the center $\phi(r = 0) = \phi_b$ may have a different value. The exact shape of $\phi(r)$ is gained from solving the Poisson equation (*Poisson*).

It will be assumed, that the reactions take place a the surface of the crystal and bulk diffusion will be neglected. Even if there are reports of oxygen bulk diffusion for certain materials , this is not

the general behavior and does not apply apply for SnO_2. Since all surface sites will be accessible by the gas, the solution of equation: Poisson should have a rotational symmetry. In the case of multiple grains with grain-grain contacts this assumptions needs to be validated again.

In case of an rotational symmetric shape of a SMOX grain, equation [eq:poisson_eq]can be expressed as a ordinary differential equation of only the radius:

$$\left(\frac{1}{r}\right)\frac{d}{dr}\frac{rd\phi(r)}{dr} = -\frac{\rho(r)}{\epsilon\epsilon_0} \qquad \text{(Poisson spherical 1)}$$

3.5   Charge density

As mentioned, the scope of my work was to investigate the transduction mechanism. Specially including the phenomena of a switch from a depletion- to accumulation layer controlled transduction mechanism. The previously shown conduction mechanism switch an experiment is shown, where such a switch occurs under application relevant environmental conditions (50% r.h. and ~3 ppm CO). The findings are described in detail in [BRW15].

In cases of depletion layer controlled transduction the Schottky-Approximation was proven to be a effective way to describe and simplify the Poisson equation. In the case of a accumulation layer the assumption of a fully depleted space charge layer is not valid anymore.

It should be mentioned, that a common second approximation often used together with Schottky's approximation is Boltzmann's approximation. The Boltzmann approximation is valid if the energetic difference between the energy E and the Fermi level energy E_{Fermi} is high enough:

$$E - E_{Fermi} \gg 3k_BT \qquad (1)$$

In such a case the Fermi-Dirac distribution $f(E)$ can be expressed with the Boltzmann distribution b(E):

$$f(E) = \frac{1}{exp(\frac{E-E_{Fermi}}{k_BT})+1} \xrightarrow{Boltzmann\,Conditions} b(E) = exp(-\frac{(E-E_{Fermi})}{k_BT}) \qquad (2)$$

Based on the findings, that the flat band situation is reached in application relevant conditions, the Boltzmann approximation is not always valid anymore. Operando Kelvin Probe experiments indicate, that the surface potential may drop up to 1eV below the nitrogen level [BRW15]. The typical difference $E_{Conduction} - E_{Fermi}$ is between 50meV and 300meV. It is reasonable to expect, that the conduction band may even cross the Fermi level and therefore also the conditions necessary for the Boltzmann approximation do not exist. In the upcoming calculations I will calculate and show when using the Boltzmann approximation is legit and when not.

Since the goal of this work is to unify the calculations for all transduction regions, the Fermi-Dirac distribution, without further simplification, is used to calculate the charge distributions.

We will begin by rewrite the Fermi-Dirac equation to a suitable format, which will reflect the the occupation probability at energies relative to the initial conduction band position E_C:

Ferimi-Dirac:

$$f(E) = \frac{1}{exp(\frac{E-E_{Fermi}}{k_B T})+1} = \frac{1}{exp(\frac{E-E_C+E_C-E_{Fermi}}{k_B T})+1} = \frac{1}{exp(\frac{E_C-E_{Fermi}}{k_B T})*exp(\frac{E-E_C}{k_B T})+1} \quad \text{(Fermi)}$$

The density of states $g_{E_C}(E)$ with the energy E and the conduction band at E_C is given by [SN07] as followed:

$$g_{E_C}(E) = \frac{\sqrt{2}}{\Pi^2} \frac{\sqrt{E-E_C}}{\hbar^3} m^{*\frac{3}{2}} = 4\Pi * \frac{(2*m^*)^{\frac{3}{2}}}{h^3} * \sqrt{E-E_C} \quad (3)$$

The integral of the Fermi occupation probability $f(E)$ (*Fermi*) and the density of states $g_{E_C}(E)$ results in the the number of charges in the conduction band:

$$n(E_C) = \int_{E_C}^{inf} g_{E_C}(E) * f(E)\, dE \quad (n(E_C))$$

Typically this equation is simplified to the following form. Such an analytical equation is useful for further analytical calculations but is not necessary for our numerical approach:

$$n(E_C) = N_C exp\left(\frac{E_F - E_C}{k_B T}\right) \quad (4)$$

with $N_C = 2\left(\frac{2\Pi m_e^* k_B T}{h}\right)^{\frac{3}{2}}$, the effective density of states in the conduction band. It is worth-wise to mention, that this simplification is only valid if also the Boltzmann approximation is valid.

Equation $n(E_C)$ is solved numerically and compared with the results obtained with the common approximations. $m_e^* = 0.3m_e$ for SnO_2 was chosen based on [BD05].

Above a operation temperature of 300°C, all donors are ionized and available as free charge carriers in the conduction band [BRW15]. If some of those electrons are trapped at the surface due to surface reaction, a positive charge remains localized in the crystal at the donors position. Additionally the energetic position of the conduction band increases with electron trapped at the surface. Out of the combination of conduction band shift $E = E_C - E_{C_b}$ and equation $n(E_C)$, one can calculate the free charge carrier density ρ from Poisson's equation.

In the case of an unaffected bulk, $n(E_{C_b}) \equiv n_b$ is the density of electrons in the conduction band. In case of a charge transfer to/from the surface, the number of electrons in the conduction band will change. The relation between the density of charges in the conduction band $n(E_C)$ and the shifted, new energetic position of the conduction band E_C is fixed by equation $n(E_C)$. The difference between n_b and $n(E_C)$ is the density of the positive, ionized donors remaining in the crystal. Those remaining donors are the cause of the electrical screening of the surface potential. The decay of the energetic conduction band position from the surface energy level back to the 'bulk position'

depends directly on that number. With this relation a energy dependent charge density can be formulated as followed:

$$\frac{\rho(E)}{e} = n(E_{C_b}) - n(E) = n_b - n(E) \tag{5}$$

Equation Poisson spherical 1 becomes then:

$$\left(\frac{1}{r}\right) \frac{d}{dr} \frac{rd\phi}{dr} = -\frac{\rho\left(E\left(r\right)\right)}{\epsilon\epsilon_0} = -\frac{e\left(n\left(E_{C_b}\right) - n\left(E\right)\right)}{\epsilon\epsilon_0} = -\frac{e\left(n_b - n\left(E\right)\right)}{\epsilon\epsilon_0} \qquad \text{(Poisson spherical 2)}$$

With $E = V * e = (\Phi_0 - \Phi) * e$

"In discussions of semiconductos, it is useful to define a"band bending" function V such that eV is related to the potential energy of an electron"[Bel07]::

$V = \phi_b - \phi, \ E = V * e$

With this relation equation (Poisson spherical 2) becomes:

$$\left(\frac{1}{r}\right) \frac{d}{dr} \frac{rdV}{dr} = \frac{e\left(n_b - n\left(E\right)\right)}{\epsilon\epsilon_0} \qquad \text{(Poisson spherical (V))}$$

With this relation is is possible to calculate how the potential at the surface is electrically screen by the remaining positive charges inside the grain. An important parameter for such calculations is the Debye length. In cases where the Boltzmann approximation is valid, the Debye length can be approximated with the following formula. In this case the given length is the distance needed to screen a potential V until its value reaches $\frac{V}{e}$. Even if the Boltzmann approximation may not be valid in all the cases to be calculated, the Debye length will still be used as material specific property.

$$L_D = \sqrt{\frac{\epsilon\epsilon_0 k_B T}{n_b e^2}} \tag{6}$$

With the definition of the Debye length all relevant variables of the calculation can be express without physical units as ratios of material specific parameters.

- The distance inside the grain r is expressed in units of the Debye length L_D :

  $r^* = \frac{r}{L_D}, \frac{dr^*}{dr} = \frac{1}{L_D} \longrightarrow dr = dr^* * L_D$

- The position of the conduction band inside the grain in units of the $\frac{k_B T}{e}$:

  $V^* = \frac{e}{k_B T} * V, \frac{dV^*}{dV} = \frac{e}{k_B T} \longrightarrow dV = dV^* * \frac{k_B T}{e}$

22

- And the number of free charge carries in units of the intrinsic number of charges n_b:

$n^*(V^*) = \frac{n(V)}{n_b}$

By substituting those unit-less parameters in equation (Poisson spherical (V)), one obtains the a unit-less Poisson equation suitable for the numerical calculations:

$$\frac{1}{r^{*2}}\frac{d}{dr^*}r^{*2}\frac{dV^*}{dr^*} = 1 - n^*(V^*) \qquad \text{(Unitless Poisson equation)}$$

This step of substituting the equation with unit less parameters is not obligatory for the the numerical calculations. As I will show in the next part of the thesis, the numerical calculation is also be possible with the initial spherical Poisson equation (Poisson spherical (V)). Therefore the material specific parameters need to be given to the algorithm. The downside would be, that for every new material with any parameter changing, the calculation would need to be redone. The benefit of the latter derived equation is that it is valid for multiple combinations of intrinsic parameters. The solution would only depend on three parameters:

- Grainsize R in units of L_D
- Temperature T as in $\frac{k_B T}{e}$
- The doping level of the semiconductor described with n_b

The advantage of the numerical approch is now, that for typical values of these parameters the solution are computed and used for further understanding of their influence on sensing with SMOX material. Typical values of the relevant parameters are:

- Typical grainsizes reach from 0.1 to 100 L_D
- Typical temperatures are in the range of 100°C to 400°C
- The doping level n_b range from $10^{20}\frac{1}{m^3}$ to $10^{25}\frac{1}{m^3}$

This indicates just the typical materials, but solutions for other parameters are alos possible. For the scope of my work I will nevertheless concentrate on the gives ranges.

3.6 Poisson equation as system of ODEs

The Python SciPy package ([JOPO15], [VGO$^+$20]) will now be used to numerically solve the derived equations. The "odesolvers" of SciPy are able to solve first order ODEs, or systems of first order ODES. To solve a second order ODE, if must first by converted by changes of variables to a system of first order ODES.

Equation (Unitless Poisson equation) is an ODE of second order, so it it needs to be express as a system of first order ODES.

Practically a functions needs to be defined, which gets as input an list of functions and returns an list of the derived functions:

$$derive\_func\left(V^*, \frac{dV^*}{dr^*}\right) \longrightarrow \frac{dV^*}{dr^*}, \frac{d^2V^*}{dr^{*2}} \qquad (7)$$

The second input term $\frac{dV^*}{dr^*}$ corresponds already to the first output term. So no special work needs to be done here. But also the second output parameter can be calculated with the given input parameters by using (Unitless Poisson equation).

$$\frac{1}{r^{*2}}\frac{d}{dr^*}r^{*2}\frac{dV^*}{dr^*} = 1 - n^*(V^*) = \frac{2r^*}{r^{*2}}\frac{dV^*}{dr^*} + \frac{r^{*2}}{r^{*2}}\frac{dV^*}{dr^{*2}} = \frac{2}{r^*}\frac{dV^*}{dr^*} + \frac{dV^*}{dr^{*2}} = 1 - n^*(V^*) \qquad (8)$$

$$\frac{dV^*}{dr^{*2}} = 1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*} \qquad \text{(Second derivative)}$$

The odesolver needs beside the derive-function additional parameters. Namely a set of initial start values for V^* and $\frac{dV^*}{dr^*}$ and boundaries in between the solver should calculate the solution. Since we first want to calculate the shape of the conduction band resulting from different surface potentials $V^*_{Surface}$, the initial parameter V^*_{init} is already defined as $V^*_{Surface}$. Also the boundaries should be the full grain, so r^* is between 0 and the grain radius R^*. Only the $\frac{dV^*}{dr^*}|_{init}$ can not directly be defined. Nevertheless a the correct solution of the differential equation can still be found. This will be done by solving the equation for multiple initial values of $\frac{dV^*}{dr^*}|_{init}$ and "picking" the right solution. This step will be described in detail in a later part of this chapter.

$$odesolver(derive\_func, [V^*_{init}, \tfrac{dV^*}{dr^*}{}_{init}], r) \longrightarrow V^*(r^*), \tfrac{dV^*}{dr*}(r*)$$

3.7 Constants

For the numerical calculations I will need to use some constants to refer to. To structure this notebook it is favorable to concentrate the definition at a single point and refer always back to this definition. This reduces the potential error of typos when using some constants over and over again.

One way to generate a global object which groups the relevant information together and allows to access them easily are classes. Such classes do not only store the relevant information but offer also some useful functionalities related to the stored information.

In the following code block, the class is defined with the statement `class`. What the follows the `class` statement is the name of the class. By convention class names start with a capital letter. Inside the `class` initially one function called `__init__` is defined with the `def __init__(self):` statement. This special function is always automatically executed when an instance of the class is created. Here I define some constants, which are relevant for course of this thesis. I also add some useful functions, e.g. the conversion from Celsius to Kelvin and vise versa. Such functions will be of mayor importance when doing the knowledge transfer gained in the 'semiconductor regime' (Kelvin is the useful scale here) to application relevant conditions (where Celsius is the usual temperature scale).

```
[1]: from scipy import constants as sciConst
     class Constant:
         def __init__(self):
             self.K0 = sciConst.convert_temperature(0,'C','K')
             self.kB=sciConst.k
```

```
        self.EPSILON_0 = sciConst.epsilon_0
        self.E_CHARGE = sciConst.elementary_charge
        self.h = sciConst.h
        self.MASS_E = sciConst.electron_mass
        self.NA= sciConst.N_A
        self.VOL_mol = 22.4
        self.mole_per_l = self.NA/self.VOL_mol

    def K_to_C(self, K):
        return sciConst.convert_temperature(K, 'K', 'C')

    def C_to_K(self, C):
        return sciConst.convert_temperature(C, 'C', 'K')

    def eV_to_J(self,eV):
        return eV*self.E_CHARGE

    def J_to_eV(self,J):
        return J/self.E_CHARGE


CONST = Constant()
```

3.8   Materials

Once the basic constants are defined, creating also a simplified numerical representation of the investigated semiconducting material. As explained in the theoretical section, the charge distribution n, depends on the position of the conduction band and is the result from solving an integral. For our calculations this integral will be solved numerically. So before all the relevant parameters of the SMOX material can be defined, a short introduction into solving integrals with Python is useful. Again start with setting up a numerical python lab, which will output all results "inline" with this document. As shown in the introduction this is done by using the magic command %pylab inline

```
[4]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib

/usr/lib/python3.8/site-packages/IPython/core/magics/pylab.py:159: UserWarning:
pylab import has clobbered these variables: ['f']
`%matplotlib` prevents importing * from pylab and numpy
  warn("pylab import has clobbered these variables: %s"  % clobbered +
```

3.8.1   Solving integrals numerically

Introduction   Many observables in nature can be predicted with the solution an integral of a certain function. In this section I will make a short excursion on how to solve integrals numerically.

From my experience, many students and researches excellently trained to define the set of equations describing their problems mathematically. Second, also the evaluation of the individual equations with multiple variables imposes (in most cases) no problem. And third, it is also part of the common knowledge, that the integral of any function is equivalent to the area below the curve. A college of mine once told me, that she learn calculating the integral in school by: 1. drawing the function for multiple points on a paper 2. combine them with a line 3. count the squares below the curve

Nothing else is done, when solving integrals numerically with Python. And we will see, that this method is quite accurate.

On the other side solving an integral analytically requires in many cases advanced mathematical skills and often approximation/simplifications are introduces to be able to solve the problem. Those tasks are often hard to master for many people (including me) and the simplifications often reduce the solution to specific boundary conditions (as for example the Boltzmann approximation).

As mentioned solving integrals numerically is fairly easy, even if one might not feel very comfortable with counting squares. But even if counting is not an option, there are modern tools to solve this task very efficiently! If haven't been introduced yet, here they come!

So if the elements to be integrated can evaluated for each point between the boundaries of the integral, not much is in the way to solve the integral numerically. Here a simple example of solving:

$$\int_3^5 x^5 dx \tag{9}$$

The analytical solution solution is:

$$\left[1/6x^6\right]_3^5 = 1/6 * 5^6 - 1/6 * 3^6 \simeq 2482.67 \tag{10}$$

The quad **function** The quad function from the `scipy.integrate` package will be used to integrate the given function. The quad needs as comma separated inputs the 1. function to integrate 2. the lower integration boundary 3. the upper integration boundary

The help file of quad says: >Integrate func from a to b (possibly infinite interval) using a technique from the Fortran library QUADPACK.

This description reveals, that the Fortran library QUADPACK is used in the background. So nothing new is shown here from the "scientific" point of view. I'd rather like to point out, how easy this can be applied in a Jupyter notebook. From discussion with colleagues I know, that the biggest challenge is how to technically implement the numerical solving algorithm in Python. So here it comes:

```
[5]: from scipy.integrate import quad

def f(x):
    # return x to the power 5
```

```
    return x**5

#numerical solution
num_sol, num_error = quad(f,3,5)

#analytical solution
ana_sol = 1/6*(5**6-3**6)

print(f'Numerical solution: {num_sol:.2f} +- {num_error:.2f}')
print(f'Analytical solution: {ana_sol:.2f}')
```

```
Numerical solution: 2482.67 +- 0.00
Analytical solution: 2482.67
```

In this case, the numerical and the analytical solution result in the same results. It is worth to mention, that quad not only returns the actual value of the integral, but also "an estimate of the absolute error in the result.".

How about a more complex problem? Let's look at the "Normal distribution":

$$f(x) = \frac{1}{\sqrt{2\Pi}} * e^{-\frac{x}{2}} \tag{11}$$

Since the probability distribution is normalized the integral from $-\infty$ to ∞ is 1:

$$\int_{-\infty}^{\infty} f(x)dx = 1 \tag{12}$$

The analytical solution of this integral is already a rather advanced task, but still doable. The numerical results are obtained in the following cell.

```
[6]: def f(x):
         return 1/(2*pi)**0.5*e**(-(x**2/2))

     num_sol, num_error = quad(f, -np.inf,np.inf)
     ana_sol = 1

     num_sol, num_error = quad(f, -inf,inf)
     ana_sol = 1

     print(f'Numerical solution: {num_sol:.12f} +- {num_error:.12f}')
     print(f'Analytical solution: {ana_sol:.12f}')
```

```
Numerical solution: 1.000000000000 +- 0.000000010178
Analytical solution: 1.000000000000
```

Also here a numerical solution is in line with the expected result from the analytic solution, if we can accept and error of 10ppb. In may cases such a small error is acceptable.

These two examples cover functions, where a analytical solution is known. In the following parts of this thesis, many analytical solutions are not known. In this case the shortcut of using a numerical solution instead of relaying on the exact solution is reasonable.

Side-Note: The `print()` statement is used to add the results in the output of the notebook. The `print` command requires a text *string* between its parenthesis. In Python a *string* consists of multiple characters between quotation marks: e.g. `'L3TT3R5'`. Additionally another rather new feature of Python is used here. This feature is called *formatted strings*. *Formatted strings* are constructed with an `f` in front of the *string*: `f'L3TT3R5'`.

For formatted strings variables inside curly parenthesis are then replaced with their string representation. The formatting of the representation may be given after `:`. For example `.12f` tells the formatter to represent the variable as a float with 12 digits after the decimal separator. When reading this as an interactive notebook, feels free to modify the formatting statement and check the result.

3.8.2 Numerical description of the semiconductor

Helper functions for semiconductor calculations Dr. Michael Hübner (page 50) derived in his thesis a way to approximate the energetic difference between the the level of the Fermi-Level and the conduction band (in a flat band situation) depending of the temperature T, concentration of defects in the bulk of the grain N_D and the effective mass of electrons inside the semiconductor m_e^*. N_D is difficult to measure experimentally so this thesis will initially start with reasonable values found in literature.

For a the later deep analysis N_D may be screen for the analysis of their influence on the overall performance of a sensing material.

Based on Michael Hübners work it is possible to derive a value for the energetic distance between Fermi level and conduction band $\Delta E_{CF} = E_C - E_F$.

Since the energetic distance from the Fermi level mainly defines the occupation probability of the states in the conduction band, this term is of major importance. It should be pointed out, that the calculation in the thesis are based special assumption only valid for SnO_2. The definition is translated into a Python algorithm and used as an starting point for further calculations. Besides this function also two other "helper functions" are defined which will be used at multiple places in the upcoming calculations.

```
[7]: import scipy
     def calc_kT(T_C):
         """
         Calculate the kT value for a temp. in °C
         T_C = Temp in °C
         """

         kT = CONST.kB*(CONST.C_to_K(T_C))
         return kT

     def calc_eff_density_of_states(T_C,mass_e_eff_factor):
         """
         Calculate the eff. densitiy of states in the conduction band
         T_C = Temp in °C
         mass_e_eff_factor = material specific factor to calculate the effective mass
                             from the electron mass
```

```python
    """
    kT = calc_kT(T_C)
    MASS_E_EFF = mass_e_eff_factor*CONST.MASS_E
    NC = 2*(2*np.pi*MASS_E_EFF*kT/(CONST.h**2))**(3.0/2.0)
    return NC

def calc_EDCF_by_temp(T_C, ND,mass_e_eff_factor):
    """
    T_C = Temperature in °C

    ND = number of donors per m³
    ND = 9e21 # 9*10**15 cm**3 Mich Thesis Seite 50

    mass_e_eff_factor = material specific factor to calculate the effective mass
                        from the electron mass
    """


    kT = calc_kT(T_C)

    NC = calc_eff_density_of_states(T_C,mass_e_eff_factor)

    ED1C_eV = 0.034
    ED2C_eV = 0.140

    a = np.exp(CONST.eV_to_J(ED1C_eV)/kT)
    b = np.exp(CONST.eV_to_J(ED2C_eV)/kT)
    t3 = 1.0
    t2 = (1.0/b-0.5*NC/ND)
    t1 = -1.0/b*NC/ND
    c = -1.0/(2*a*b)*NC/ND

    poly_params = (c,t1, t2, t3)


    solutions=numpy.roots(poly_params)
    EDCFs = []
    for sol in solutions:
        if sol.imag == 0:
            EDCF = np.log(sol.real)
            EDCFs.append(-EDCF*kT/CONST.E_CHARGE)
    if len(EDCFs)>1:
        raise Exception('Should not be...')
    else:
        return EDCFs[0]
```

```
T_C = 300
ND = 1e24
mass_e_eff_factor =0.3

EDCF_eV = calc_EDCF_by_temp(T_C, ND, mass_e_eff_factor)
print(f'{EDCF_eV:.2f}eV')
```

```
0.08eV
```

Define the smox-material class With the helper functions a new class for the describing the actual SMOX material. This class should be initialized with the relevant parameters (in the scope of this thesis). Besides this the `class Material` should hold a method to calculate the concentration of charge carries in the conduction band.

```
[8]: from scipy.integrate import quad
     from scipy.interpolate import interp1d
     import scipy
     from functools import lru_cache
     import numpy as np
     import pandas as pd




     class Material:
         def __init__(self,T_C,ND,
                       mass_e_eff_factor = 0.3, EPSILON = 9.86, DIFF_EF_EC_evolt =␣
      ↪None):
             '''
             T_C = Temperature of the material
             ND = number of donors per m³
             DIFF_EF_EC_evolt = E_condution - E_Fermi
             '''
             self.EPSILON = EPSILON
             self.ND = ND
             self.MASS_E_EFF = mass_e_eff_factor*CONST.MASS_E
             self.T_C = T_C
             self.kT = calc_kT(self.T_C)
             self.NC = calc_eff_density_of_states(T_C,mass_e_eff_factor)


             if DIFF_EF_EC_evolt:
                 self.Diff_EF_EC_evolt = DIFF_EF_EC_evolt
             else:
                 self.Diff_EF_EC_evolt = calc_EDCF_by_temp(T_C, ND, mass_e_eff_factor)
             self.Diff_EF_EC = CONST.eV_to_J(self.Diff_EF_EC_evolt)


             self.nb, self.nb_err = self.n(0)
```

```python
        self.LD = np.sqrt((self.EPSILON*CONST.EPSILON_0*self.kT)
                           /(self.nb*(CONST.E_CHARGE**2)))

    def J_to_kT(self,J):
        return J/self.kT

    def kT_to_J(self,E_kT):
        return E_kT*self.kT

    def densitiy_of_states(self,E, E_c):
        return 4*np.pi*(2*self.MASS_E_EFF)**(3.0/2.0)/CONST.h**3*(E-E_c)**0.5

    def fermic_dirac(self,E_c):
        '''
        Calculate the value for the Fermi-Dirac distribution for an energetic
        position relative to the material specific conduction band E_c
        E = E_c+Diff_EF_EC+E_Fermi
        So the term in the Fermi-Dirac distribution E-E_Fermi will become
        E_c+Diff_EF_EC+E_Fermi-E_Fermi = E_c+Diff_EF_EC
        TODO: THIS SHOULD BE IN THE TEXT ABOVE SOMEWHERE
        '''
        if (E_c+self.Diff_EF_EC)/self.kT>100:
            f = 0
        else:
            f=1.0/(1+np.exp((E_c+self.Diff_EF_EC)/self.kT))

        return f

    def n_E(self,E,E_c):
        if E<E_c:
            n = 0
        else:
            n = self.densitiy_of_states(E, E_c)*self.fermic_dirac(E)
        return n

    @lru_cache(maxsize=512*512*512)
    def n(self, E_c):
        '''
        Calculate the number of charges in the conduction band at the position␣
↪E_C

        E_C  = the postition of the conduction band in J
        '''
        n, n_err = quad(lambda E:self.n_E(E, E_c),E_c,E_c+self.kT*100)
        return n, n_err


T_C = 300
```

```
ND = 1.16e23
mass_e_eff_factor =0.3

EDCF_eV = calc_EDCF_by_temp(T_C, ND, mass_e_eff_factor)

print(f'''For SnO2 at {T_C}°C with a defect concentration of {ND} 1/m³,
      the value of EDCF_eV is {EDCF_eV:.3f} eV''')

material = Material(T_C,ND, DIFF_EF_EC_evolt=EDCF_eV)
```

```
For SnO2 at 300°C with a defect concentration of 1.16e+23 1/m³, the value of
EDCF_eV is 0.190 eV
```

Hint: `@lru_cache(maxsize=512*512*512)` is a decorator for the function n(self, E_c).

> "By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it."
> (https://realpython.com/primer-on-python-decorators/)

This Python decorator is used to speed up the calculation process. The lru_cache ("Last Recently Used") is used to cache the input and output of a certain function. As the description of the function says:

> "It can save time when an expensive or I/O bound function is periodically called with the same arguments"

Since in our numerical calc. we will often need to derive the charge density the `@lru_cache` is of great use here. The maxsize argument in the brackets defines the maximal size of the cache in the memory of the computer.

Relation between n_b and N_D   In the thesis of Julia Rebolz (page 106) some values for n_b and L_D and the distance of the conduction band to the Fermi-Level ($E_{C,Flatband} - E_F$)in units of [eV]. The numerical calculation with the provided `Material` class are in good agreement with the presented results. The code cell below can be used to check values found in literature with the presented model.

[9]:
```python
T_C = 300
a = np.array([(Material(T_C, ND_temp).nb, ND_temp) for ND_temp in np.
  ↪logspace(21, 25)])
fig, axe = subplots(figsize = (((1+5**0.5)/2)*9,9))
x = a[:,0]
y = a[:,1]
axe.plot(x,y)
axe.set_yscale('log')
axe.set_xscale('log')
axe.set_ylabel('$N_D$ $[m^{-3}]$', fontsize=22)
axe.set_xlabel('$n_b$ $[m^{-3}]$',fontsize=22)
axe.grid()
axe.set_title('$N_D$ as a function of $n_b$', fontsize = 22)
```

```python
# To calcaulte the ND from nb, the numiercal data is interpolated.
# Generally this way is much simpler than deriving the inverse function to␣
 ↪calculate ND from a given nb
ND_from_nb = scipy.interpolate.interp1d(x,y, kind='cubic' )

#Chekcing one value from the Thesis of Julia Rebholz
nb_check = 2.14e24
ND_check = ND_from_nb(nb_check)
mat_check = Material(T_C, ND_check)
LD_check = mat_check.LD
Ec_Ef_eV_check = mat_check.Diff_EF_EC_evolt
axe.scatter(nb_check, ND_check)
axe.text(nb_check,ND_check/2,
         f'$n_b$={nb_check:.2e} $1/m^3$\n$N_D$={ND_check:.2e} $1/
 ↪m^3$\n$L_D$={LD_check*1e9:.2f} nm\n$E_C-E_F$={Ec_Ef_eV_check*1000:.2f}meV',
         verticalalignment='top', horizontalalignment='center');
```


N_D as a function of n_b

n_b=2.14e+24 1/m³
N_D=1.16e+24 1/m³
L_D=3.55 nm
E_C-E_F=76.93meV
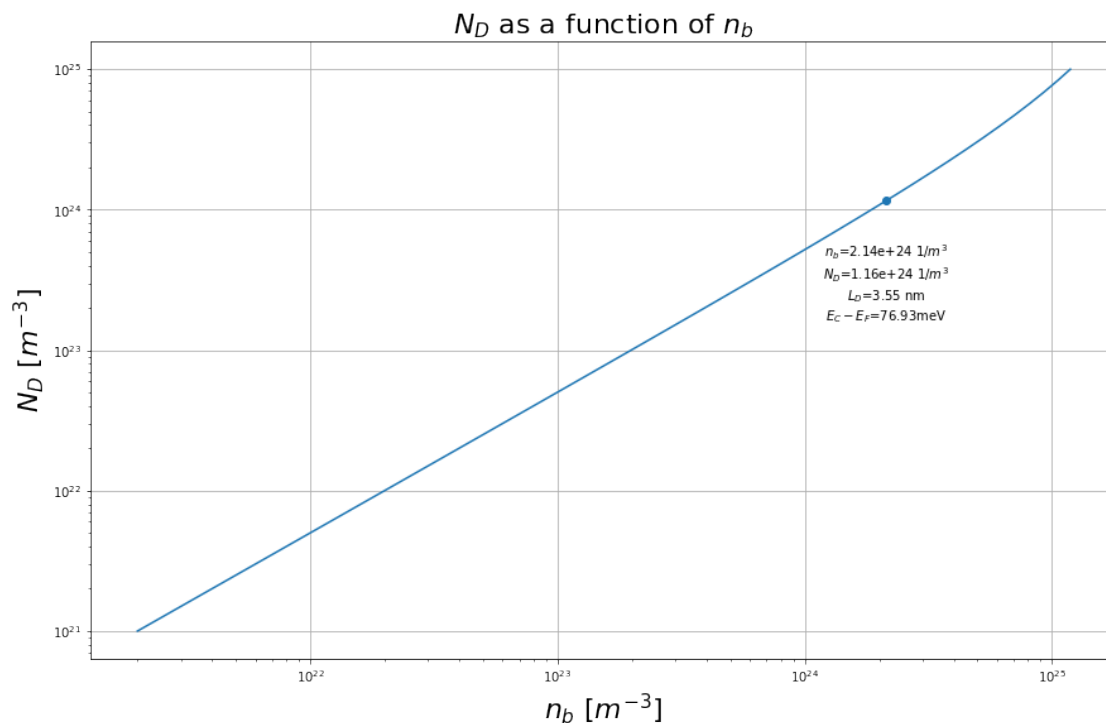
Free charge carrier conc. using the Boltzmann approximation   Besides the full numerical solution, also the solutions derived from the Boltzmann approximations need to be defined. This will allow to compare the different solutions and check the validity of the different approximations.

```
[10]:  def boltzmann_acc(material, E_c):
           return np.exp(-(E_c+material.Diff_EF_EC)/(material.kT*2))

       def boltzmann(material,E_c):
           return np.exp(-(E_c+material.Diff_EF_EC)/material.kT)

       def densitiy_of_states(material,E, E_c):

           return 4*np.pi*(2*material.MASS_E_EFF)**(3.0/2.0)/CONST.h**3*(E-E_c)**0.5

       def n_boltzmann(material,E_c):

           return boltzmann(material,E_c)*material.NC

       def n_boltzmann_acc(material,E_c):

           return boltzmann_acc(material,E_c)*material.NC
```

Compare the numerical solution with the approximations   With all the definitions in place,
the different solutions can be compared. This will be done be representing the charge carrier
concentration *n* for the different solutions for multiple positions of the conduction band E_C in
units of *kT*.

```
[11]:  def plot_material_char(mat):
           ns = []
           n_boltzs = []
           n_boltzs_acc = []
           E_c_kts = []
           for i in np.linspace(-20,20):
               E_c = mat.kT_to_J(i)
               E_c_kts.append(i)
               ns.append(mat.n(E_c)[0]/mat.nb)
               n_boltzs.append(n_boltzmann(mat, E_c)/mat.nb)
               n_boltzs_acc.append(n_boltzmann_acc(mat, E_c)/mat.nb)


           fermi_level_pos_kt = -mat.J_to_kT(mat.Diff_EF_EC)

           fig, axe = subplots(1,figsize = (16,9))

           axe.plot(E_c_kts, ns, label='No approx. ')
           axe.plot(E_c_kts, n_boltzs, '--', label='Boltzmann approx.')
           axe.plot(E_c_kts, n_boltzs_acc, '-.', label='Accumulation-Boltzmann approx.')
           axe.set_yscale('log')
           axe.set_title('$\\frac{n(E_C)}{n(0)}$ as a function of the band bending␣
       ↪$E_C$', fontsize=22)
           axe.set_xlabel('Band bending $E_C$ [kT]', fontsize=22)
```
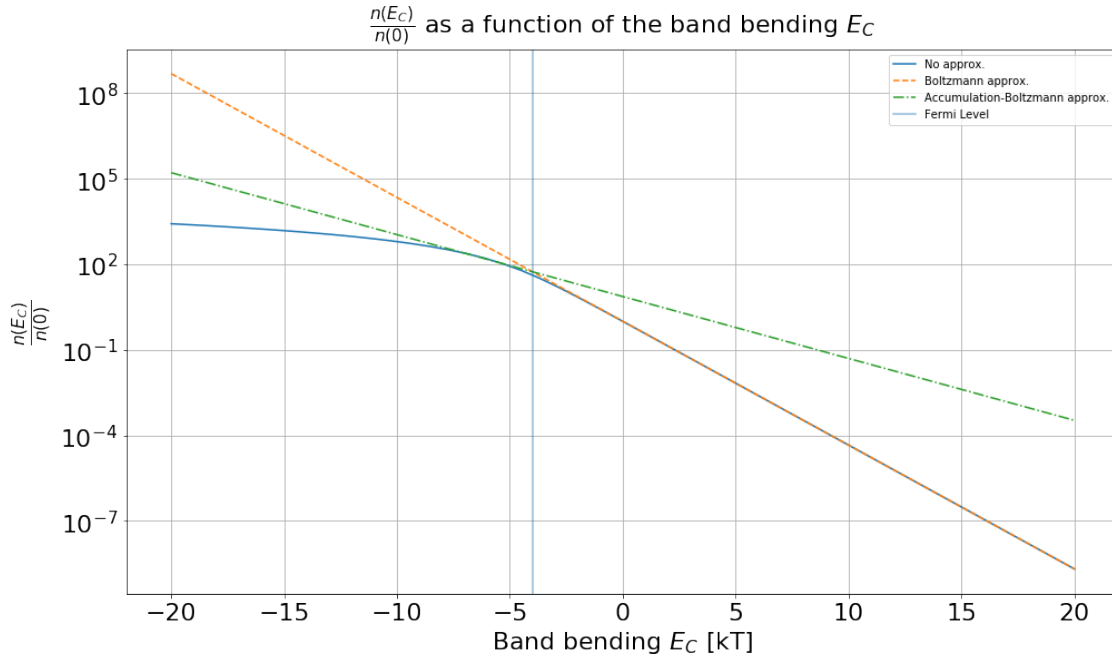
```
    axe.set_ylabel('$\\frac{n(E_C)}{n(0)}$', fontsize=22)
    axe.axvline(fermi_level_pos_kt, label='Fermi Level', alpha=0.5)
    axe.tick_params(axis='both', which='major', labelsize=22)
    axe.legend()
    axe.grid(b=True)
    return fig
material = Material(300,1e23)
fig = plot_material_char(material)
```


The numerical solution and the approximations are inline with each other in their specific regions of E_C. The Boltzmann approximation is identical to the numerical solution starting ~3kT above the Fermi energy level. The approximation for the accumulation layer has its validity in a region where an accumulation is present. This was already shown in [BHW11].

3.8.3 Numerical description of the semiconductor grains

In this section we define a SMOX grain. We approximate the grain as a sphere composed out of a material we previously defined. For one grain, the Poisson equation with spherical symmetry is solved. The transfer of the findings from a material to an actual grain are important for multiple reasons.

On the one side the ratio of the available surface sites to react with the semiconductor and its bulk size play an important role. Very small grains may have relatively high concentration of surface sites but lack of electrons needed for the reaction at the surface. So a grain may get fully depleted which has significant influence on the overall conduction.

On the other side, the conduction path through the grain differers depending on the free charge

carrier concentration. Depending on the size of the grain and the charge distribution, the charge current may have its conduction path along the center of the grain, or along the surface.

These effects of the grainsize can only be analyzed, if the transfer from a material to an actual grain is solved.

To solve the Poisson equation, we will need to supply the solver with the initial values. As described, two values need to be supplied. One is the surface potential. This value can experimentally be measured with the Kelvin Probe method. The second start parameter, which needs to be supplied is the slope of the potential at the surface. With these two parameters, the solver iterates from the starting condition stepwise though the grain and calculates for each step new values based on the previous iteration.

This "inital value problem" is solved with the scipy tool solve_ivp.

```python
[12]: from scipy.integrate import solve_ivp
class Grain:
    def __init__(self,grainsize_radius,material,rPoints=1000):
        self.R = grainsize_radius
        self.material = material
        self.rs = np.linspace(self.R/1000, self.R, rPoints)


    def solve_with_values(self,E_init, E_dot_init):
        r_LD = self.rs/self.material.LD
        E_init_kT = self.material.J_to_kT(E_init)
        E_dot_init_kt = self.material.J_to_kT(E_dot_init)

        #the solver should stop, when the slope is zero.
        #This is reasonable since if the slope is zero, this should be the lowest
        #point of the graph so, when we "hit_ground" the solver should stop,
        #to save some computational time
        def hit_ground(t, y):
            #print(y)
            if y[0]:
                if E_init_kT<0:
                    if y[0]>0:
                        return 0
                    if y[0]<E_init_kT:
                        return 0
                else:
                    if y[0]<0:
                        return 0
                    if y[0]>E_init_kT:
                        return 0

            if y[1]:
                if abs(y[1])<0.0001:
                    return 0
```

```python
            return y[1]
        hit_ground.terminal = True


        #see the docstring why I chose the metohd BDF
        data = solve_ivp(self.deriv_E_E_dot,(r_LD[-1],r_LD[0]), ␣
↪[E_init_kT,E_dot_init_kt],
                         t_eval=r_LD[::-1], events=hit_ground, method = 'BDF')

        #since we start the iteration to solve the equation from the outside,
        #the results have to be revered

        r = data.t[::-1]
        v = data.y[0][::-1]
        v_dot = data.y[1][::-1]

        #sinde we stop the evaluation earlier, when v_dot = 0,
        #the missing elements are fileed up
        missing_elements_count = len(r_LD)-len(r)
        r = np.concatenate((r_LD[:missing_elements_count], r))
        v = np.concatenate((np.ones(missing_elements_count)*v[0],v))
        v_dot = np.concatenate((np.ones(missing_elements_count)*v_dot[0],v_dot))



        return r,v, v_dot, data


    def deriv_E_E_dot(self,r_, U_U_dot):
        U = U_U_dot[0]
        U_dot = U_U_dot[1]
        E = self.material.kT_to_J(U)
        n = self.material.n(E)
        U_dot_dot = 1-n[0]/self.material.nb -2/r_*U_dot
        return [U_dot, U_dot_dot]
```

Example of differnt grains

```python
[13]: #define a grain with a specific material
      def create_grain(grainsize, T_C, ND):
          mass_e_eff_factor =0.3
          material = Material(T_C,ND)
          grain = Grain(grainsize_radius=grainsize,material=material)
          return grain

      #Check the influence of LD
```

```
g = create_grain(100e-9, T_C=300, ND=9e21)
print(g.R/g.material.LD)

g = create_grain(50e-9, T_C=300, ND=9e24*4)
print(g.R/g.material.LD)
```

```
2.5852950260293164
45.47592892276805
```

In the next example a grain is defined with a fixed surface potential. Based on the surface potential and 3 different guesses of the initial slope of the potential, the shape of the conduction bad inside the grain is calculated.

```
[14]: #defining a new grain
      T_C = 300
      ND = 1e23
      grain = create_grain(110e-9, T_C, ND)

      #initalize the plot
      fig, axe = subplots(figsize = (16,9))
      axe.set_ylim(-10,10)

      #fix the surface potential to 8kT
      E_init_kT = 8
      #express the surface potential in J
      E_init = grain.material.kT_to_J(E_init_kT)

      #for 3 values of the inial slope in units of 1kT/LD
      #calculate the conduction band
      # In 1LD the potential drops of the factor 1/e in a infite plane
      #8/e ~ 3; so 1,2,4 should be fine as a guess
      for E_dot_init_kT in [1,2,3,4]:
          #convert to SI units for the numerical calc.
          E_dot_init = grain.material.kT_to_J(E_dot_init_kT)

          #solve with the inital values
          r,v, v_dot, data = grain.solve_with_values(E_init, E_dot_init)


          axe.plot(r,v, label=f'Initial slope: {E_dot_init_kT} [kT/$L_D$]')
          axe.set_xlabel
      axe.set_ylabel('$E_C(r^*)$ [$k_BT$]', fontsize =22)
      axe.set_xlabel(f'Position in grain [$L_D$ = {grain.material.LD*1e9:.2f}nm],
                     R={grain.R*1e9:.2f}nm', fontsize =22)
      leg = axe.legend()
```
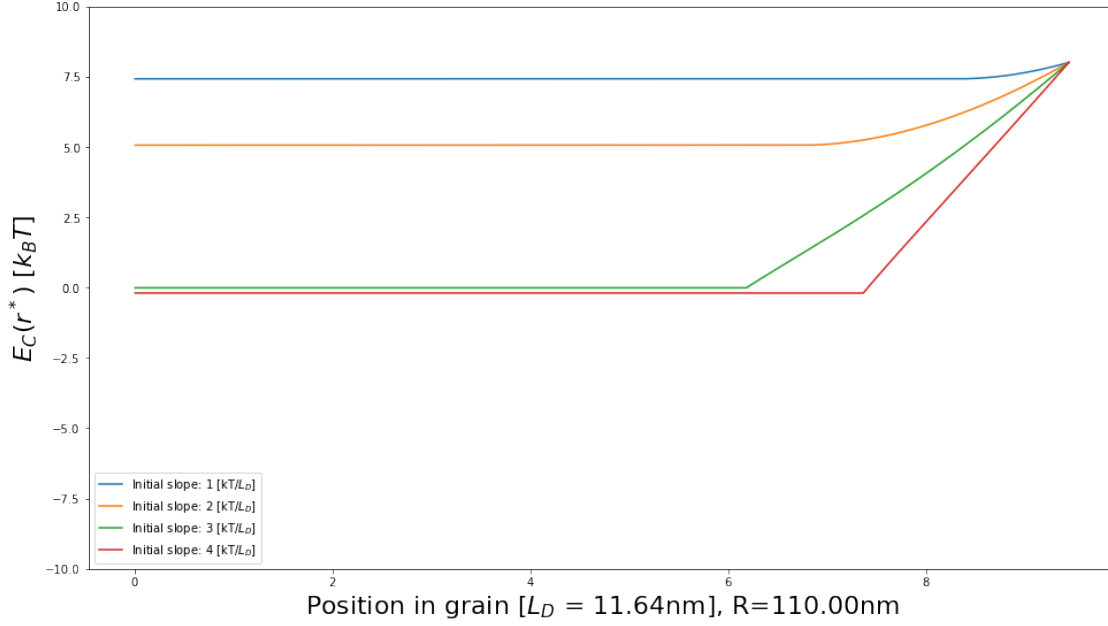
Position in grain [$L_D = 11.64$nm], R=110.00nm

From this graph it is obvious, that the initial slope has a major influence on the result.

Unfortunately this value is not known. On the other side, I can relay on an equation to check my solution. We will use the derived equation (Surface slope) from the Poisson equation:

$$\frac{dV^*}{dr^{*2}} = 1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*} \qquad \text{(Surface slope)}$$

This equation can be transformed into following form:

$$\int_0^R \frac{dV^*}{dr^{*2}}dr^* = \left[\frac{dV^*}{dr^*}\right]_0^R = \frac{dV^*}{dr^*}\bigg|_R = \qquad (13)$$

$$\frac{dV^*}{dr^*}\bigg|_{Surface} = \int_0^R 1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*}dr^* \qquad (14)$$

With this relation we can check each solution. For each starting condition the solution should also be valid for this equation. The right side will again be evaluated numerically. The expression $n^*(V^*)$ can be calculated for each V^* with the function defined in `class material`. From the solver of the differential equation $\frac{dV^*}{dr^*}$ is known inside the grain for each r^*. Since all the elements of the integral are known, the numerical evaluation is not difficult. Since the elements of the integral in this case are not functions, which can be calculated individually for each point, but rather a list of values, the integration is slightly different. For the numerical integration of a list of values y corresponding to a set of x values, the numpy function `trapz` is used:
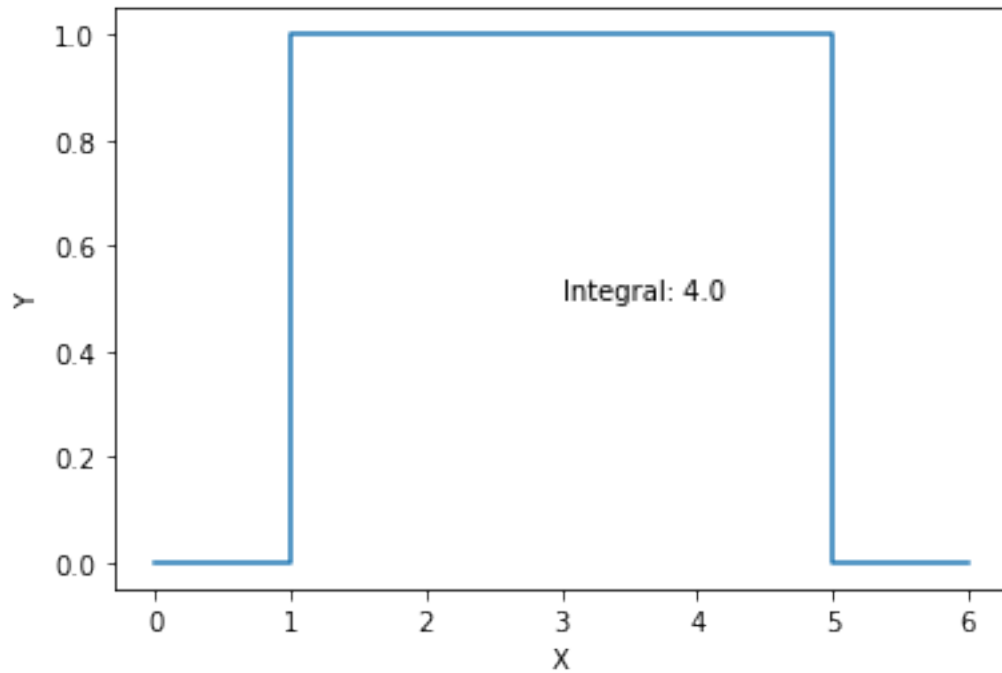
39

numpy.trapz(y, x=None, dx=1.0, axis=-1)

Integrate along the given axis using the composite trapezoidal rule.

Example usage of `np.trapz`

```
[15]: x = [0,1,1,2,3,4,5,5,6]
      y = [0,0,1,1,1,1,1,0,0]
      fig, axe = subplots()
      axe.plot(x,y)
      axe.set_xlabel('X')
      axe.set_ylabel('Y')
      numerical_integral = np.trapz(y,x)
      axe.text(3, 0.5, f'Integral: {numerical_integral}')
```

```
[15]: Text(3, 0.5, 'Integral: 4.0')
```


From the latter relation between the inital slope a the surface and charge distribution on the center (*Surfaceslope*) the following boundary condition for the solution is defined:

$$\left.\frac{dV^*}{dr^*}\right|_{R^*} - \int_0^{R^*} \left(1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*}\right) dr^* = 0. \tag{15}$$

Tle left side of the equation can be calculated for multiple values of $\left.\frac{dV^*}{dr^*}\right|_{R^*}$. The right value of $\left.\frac{dV^*}{dr^*}\right|_{R^*}$ needs to be found to minimize the left side of the equation. Definitely similar problems

have been done before, Python/SciPy has already a solution for this ready. The tools needed to solve this problems can be found in the `scipy.optimize` package. The function `minimize_scalar` will be used to minimize the left side of the equation by varying the scalar parameter $\left.\frac{dV^*}{dr^*}\right|_{R^*}$.

The following line is used to load the required function: `from scipy.optimize import minimize_scalar`

To use this function, a function needs to be defined, which is then minimized by changing the input parameter. This function, which takes the initial slope as an argument, returns the "error" based on the previous equation. The algorithm should then find the best initial slope parameter to have a valid solution and minimize the "error"

```python
[16]: from scipy.optimize import minimize_scalar


def min_vdot(vdot_init, grain, vinit, debug = False):
    #solve the ivp with the given values
    r,v,vdot, data = grain.solve_with_values(grain.material.kT_to_J(vinit),
                                              grain.material.kT_to_J(vdot_init))


    #for each point of the solution the element in the integral is calculated
    integrand = [(1-grain.material.n(grain.material.kT_to_J(v_i))[0]/grain.
    →material.nb)-2/r_i*vdot_i for r_i,
                 v_i, vdot_i in zip(r, v, vdot)]

    #the integral is numerically calculated
    dV = np.trapz(y=integrand,x=r)

    #The integral should be the same as the slope at the surface,
    #the difference is the error to be minimized
    res = abs((dV-vdot[-1]))

    if debug:
        print(vdot_init, dV, vinit, res)
    return res

def find_best_E_dot_init(E_init_kT, grain,debug = False, bounds = None):
    #if bounds are given as a hint for the minimize algorithm, then the
    →method='Bounded' can be used
    #In this case the algorithm will search in the biven interval
    if bounds:
        res = minimize_scalar(min_vdot,args=(grain, E_init_kT, debug), method =
    →'Bounded',
                              bounds = bounds)
    else:
        res = minimize_scalar(min_vdot,args=(grain, E_init_kT, debug))
    return res
```

41

So the previous example with randomly guessed initial values can be extended with a better guess.

```
[17]:  #grain = Grain(100e-9,material)


       T_C = 300
       ND = 1e23
       grain = create_grain(110e-9, T_C, ND)

       fig, axe = subplots(figsize = (16,9))
       axe.set_ylim(-10,10)

       E_init_kT = 8
       E_init = grain.material.kT_to_J(E_init_kT)

       for E_dot_init_kT in [1,2,4]:
           E_dot_init = grain.material.kT_to_J(E_dot_init_kT)
           r,v, v_dot, data = grain.solve_with_values(E_init, E_dot_init)
           axe.plot(r,v, label='$\dotE_{Surface}$'+ f'= {E_dot_init_kT:.2f}[$k_BT$/
       ↪$L_D$]')


       #Now we will try to find the best inital slope to solve the equation
       res = find_best_E_dot_init(E_init_kT, grain)

       E_dot_init_kT = res.x
       E_dot_init = grain.material.kT_to_J(E_dot_init_kT)
       r,v, v_dot, data = grain.solve_with_values(E_init, E_dot_init)
       axe.plot(r,v, '-*', label='$\dotE_{Surface}$'+ f'= {E_dot_init_kT:.2f}[$k_BT$/
       ↪$L_D$] minimized')

       axe.set_ylabel('$E_C(r^*)$ [$k_BT$]', fontsize =22)
       axe.set_xlabel(f"""Position in grain [$L_D$ = {grain.material.LD*1e9:.2f}nm],\n
                       R={grain.R*1e9:.2f}nm""", fontsize =22)
       axe.legend();
```
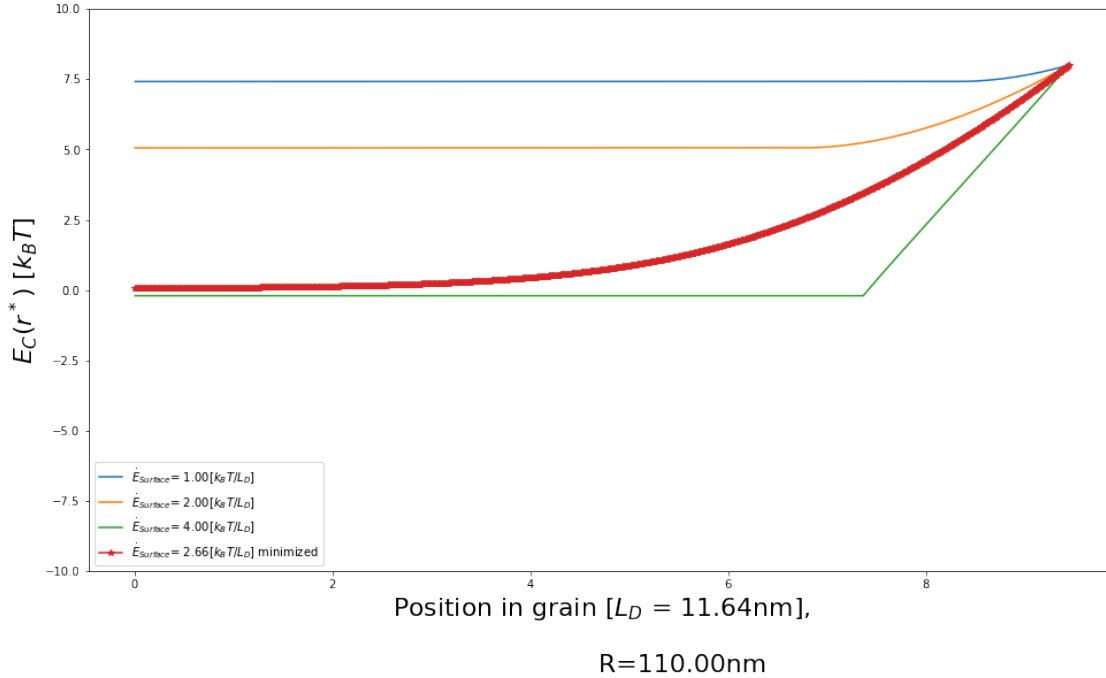
Figure axis labels: $E_C(r^*)\ [k_BT]$ (vertical axis), Position in grain $[L_D = 11.64nm]$, R=110.00nm (horizontal axis)

Legend:
$\dot{E}_{Surface} = 1.00\,[k_BT/L_D]$
$\dot{E}_{Surface} = 2.00\,[k_BT/L_D]$
$\dot{E}_{Surface} = 4.00\,[k_BT/L_D]$
$\dot{E}_{Surface} = 2.66\,[k_BT/L_D]$ minimized

3.8.4 Additional relevant paramters

From the above solution some additional results can be calculated without too much of computational effort. For instance the total number of free charge carries inside the grain left. From the charge distribution inside the grain, a simple integral of the values over the volume of the sphere will reveal this value. The difference of this value to the value in the flat band situation is then be the number of charges trapped at the surface $N^-_{Surface}$. Additionally from the concentration of defects N_D inside the grain, it is possible to calculate the amount of defects resulting in the $1\mathring{A} = 0.1nm$ if they would equally project on the surface.

```
[18]: def calc_sum_of_charges(grain, r, v):
          ser = {}
          #Geometric properties
          ser['R'] = grain.R
          grain_volume = 4.0/3.0*pi*(grain.R**3)
          grain_surface = 4.0*pi*(grain.R**2)
          ser['grain_vol'] = grain_volume
          ser['grain_surface'] = grain_surface

          # calcualte the acctual free charge carrier conc. from the position inside
          ↪the grain
          n = [grain.material.n(v_J)[0] for v_J in grain.material.kT_to_J(v)]
          ser['n'] = n

          # charges in grain and at the surface
```

43

```python
    all_c_at_flatband = 4.0/3.0*pi*(grain.R**3)*grain.material.n(0)[0]
    charges_at_surface = all_c_at_flatband-np.trapz(n,(r*grain.material.LD)**3*4/
↪3*pi)
    ND_projection_surf = grain.material.ND*((grain.R+0.1*1e-9)**3 - grain.
↪R**3)*4/3*pi

    ser['all_c_at_flatband'] = all_c_at_flatband
    ser['charges_trapped_at_surface'] = charges_at_surface
    ser['surface_vacancies_projection'] = ND_projection_surf




    #additional paramters
    ser['temp'] = grain.material.T_C
    ser['mass_eff'] = grain.material.MASS_E_EFF
    ser['ND'] = grain.material.ND
    ser['EPSILON'] = grain.material.EPSILON
    ser['nb'] = grain.material.nb
    ser['E_Fermi_kT'] = -grain.material.J_to_kT(grain.material.Diff_EF_EC)

    return ser

calc_charges = calc_sum_of_charges(grain, r, v)


display(pd.DataFrame(pd.Series(calc_charges)))
```

	0
R	1.1e-07
grain_vol	5.57528e-21
grain_surface	1.52053e-13
n	[1.8453235121620025e+23, 1.8453235121620025e+2...
all_c_at_flatband	1106.69
charges_trapped_at_surface	937.262
surface_vacancies_projection	1.52191
temp	300
mass_eff	2.73282e-31
ND	1e+23
EPSILON	9.86
nb	1.985e+23
E_Fermi_kT	-3.99825

3.9 Putting the pieces together

With a description of the semiconductor itself by the class `material` and the semiconductor grain by the class `grain` the screening of multiple parameters can start. In the following combinations will be screened:

- 4 different defect concnetrations N_D: $[10^{21}, 10^{22}, 10^{23}, 10^{24}]$ $[\frac{1}{m}]$
- Temperature of the meraterial: 300°C.
- Surface potentials [-20,20] $k_B T$
- Grain radii (R): 6.25nm, 12.5nm, 25nm, 50nm and 100nm.

Those results will lead in a second step to an understanding of the relation between surface reaction, resistance change and grain size. But for now, data for further analysis will be generated. This part has a high computational effort. Since the solutions of each combination do not depend on each other, this job can be parallelized easily.

As we would like to do the time consuming calculations (finding the right start conditions) only once, we will save the correct solution in a `DataFrame`. As mentioned earlier, a `DataFrame` is a data structure to organize information similar to Excel Worksheets. As in "Excel Worksheets" data can be stored, accessed and manipulated. A `Dataframe` is a part of the `pandas` Python library. To shorten the command for pandas I will import it and add an alias to it. The following code part import `pandas` and creates a `Dataframe`, where all our results will be stored.

```
[19]: dF_calc = pd.DataFrame()
```

Next, we need to create the numerical representation of the material. Here we define helper functions, which are needed for the parallelization of the jobs.

```
[20]: def solve_grain_for_E_init_kT(E_init_kT,grain, debug = False, bounds = None):
          res = find_best_E_dot_init(E_init_kT, grain,debug = debug, bounds = bounds)

          ser_temp = pd.Series(dtype=float)
          ser_temp['Einit_kT'] = E_init_kT
          ser_temp['E_dot_init_kT'] = res.x
          ser_temp['res'] = res.fun

          E_init_J = grain.material.kT_to_J(E_init_kT)
          E_dot__init_J = grain.material.kT_to_J(res.x)
          r,v,vdot, data = grain.solve_with_values(E_init_J,E_dot__init_J)

          ser_temp['v'] = v
          ser_temp['v_dot'] = vdot
          ser_temp['r'] = r

          derived_values_dict = calc_sum_of_charges(grain, r, v)
          ser_temp = ser_temp.append(pd.Series(derived_values_dict))
          return ser_temp


      def calcualte_conduction_band(grain):
```

```python
    dF_calc_temp = pd.DataFrame()
    #for E_init_kT in [-8,0,8]:
    #for E_init_kT in [-8,-4,-2,-1,0,1,2,4,8]:
    for E_init_kT in list(list(range(-20,21))):
        ser_temp = solve_grain_for_E_init_kT(E_init_kT,grain)
        dF_calc_temp = dF_calc_temp.append(ser_temp, ignore_index=True)
    return dF_calc_temp


def calc_solution_by_parameters(T_C, ND, grainsize):
    grain = create_grain(grainsize, ND=ND, T_C=T_C)
    dF_calc_temp = calcualte_conduction_band(grain)
    return dF_calc_temp
```

3.9.1 Defining the parameters to be screen

Then a list is created including all the paramters combinbations. In Python a fast way to achieve this is `itertools.product`. This function will gerate a list of all combination of its arguments.

```python
[21]: import itertools
      import pprint

      Ts = [300]
      ND_ref = 1e21
      NDs = [ND_ref, ND_ref*1e1, ND_ref*1e2, ND_ref*1e3]
      Rs = [6.25e-9, 12.5e-9, 25e-9, 50e-9, 100e-9]

      combinations = list(itertools.product(Ts,NDs,Rs))
      pprint.pprint(combinations)
```

```
[(300, 1e+21, 6.25e-09),
 (300, 1e+21, 1.25e-08),
 (300, 1e+21, 2.5e-08),
 (300, 1e+21, 5e-08),
 (300, 1e+21, 1e-07),
 (300, 1e+22, 6.25e-09),
 (300, 1e+22, 1.25e-08),
 (300, 1e+22, 2.5e-08),
 (300, 1e+22, 5e-08),
 (300, 1e+22, 1e-07),
 (300, 1e+23, 6.25e-09),
 (300, 1e+23, 1.25e-08),
 (300, 1e+23, 2.5e-08),
 (300, 1e+23, 5e-08),
 (300, 1e+23, 1e-07),
 (300, 1e+24, 6.25e-09),
 (300, 1e+24, 1.25e-08),
```

```
(300, 1e+24, 2.5e-08),
(300, 1e+24, 5e-08),
(300, 1e+24, 1e-07)]
```

3.9.2 Starting a parallelized calculation

In the next cell block the actual calculation will take place. The first line: `from multiprocessing import Pool`, adds multi processing capabilities to the programming environment. The imported class `Pool` represents a pool of worker processes, which are used to execute the task in parallel. In line 11 `Pool(8)` initializes a pool with 8 parallel processes. On this pool the function `starmap` takes care of executing a certain function with specific arguments. In this case the function will be `calc_solution_by_parameters` and the 'parameters' will be the list of combinations we just created in the cell above. Additionally the time is calculated for the full process. The output shows, that the total calculation time is around 1h. This was performed on a regular desktop PC. This examples shows, that performing scientific numerical calculations do not need necessarily special dedicated hardware to get started. Regarding the low workload most computer have in average over a day, most modern PCs should be very well suited for such a task.

```python
[43]: #using multiple processors of the system to calculate the solutions in parallel.
      from multiprocessing import Pool
      import time

      #save the start timestamp
      start_calc_time = time.time()

      #use 8 cores of the cpus
      #starmap takes over the work of running the tasks
      #The pool of 8 cores is used to distribute the work
      with Pool(8) as p:
          all_res_list = p.starmap(calc_solution_by_parameters, combinations)
          pass

      #All solution is are returned in a list, which needs then to be combined again
      dF_calc = pd.concat(all_res_list)
      dF_calc.index = range(len(dF_calc))

      calc_duration_sec = time.time()-start_calc_time
      print(f'Calc duration: {calc_duration_sec/60:.1f}min.')
```

3.9.3 Export/Import data

The date will be saved for later use and to avoid a re-calculation. It is helpful to directly re-import the data to see if any mistakes have happened while saving the date.

```python
[38]: dF_calc.to_hdf('results.h5', 'raw', mode='w')
```

```
[66]: pd.set_option('display.max_columns', 6)
      pd.set_option('display.precision', 2)
      pd.set_option('display.max_colwidth', 12)

      calc_dF_all = pd.read_hdf('results.h5', 'raw')
      calc_dF_all.head()
```

```
[66]:    EPSILON  E_Fermi_kT  E_dot_init_kT  ...   temp           v       v_dot
      0     9.86        -8.6       -1028.08  ...  300.0  [-9.3515...  [-0.0582...
      1     9.86        -8.6        -915.45  ...  300.0  [-9.2795...  [-0.1319...
      2     9.86        -8.6        -806.08  ...  300.0  [-9.2049...  [-0.1626...
      3     9.86        -8.6          -5.01  ...  300.0  [-17.0, ...  [-5.0107...
      4     9.86        -8.6          -4.16  ...  300.0  [-16.0, ...  [-4.1571...

      [5 rows x 19 columns]
```

With all the results now in single DataFrame, we will analyze and represent individual rows of the full set to gain some insides about the data. Since the DataFrame only saves the resulting numbers, but not the corresponding classes `Grain` and `Material`, we need to create the numerical `Grain` again from this numbers. A helper function, which takes one or multiple rows and returns the corresponding numerical `Grain` class, will be very handy an is constructed in the next cell.

```
[67]: def create_grain_from_data(dF):
          if type(dF)==pd.Series:
              dF = pd.DataFrame([dF])

          if len(dF['temp'].unique())==1:
              T_C = dF['temp'].unique()[0]
          else:
              raise Exception('Multiple paramters for one grain are invalid.')

          if len(dF['ND'].unique())==1:
              ND = dF['ND'].unique()[0]
          else:
              raise Exception('Multiple paramters for one grain are invalid.')

          if len(dF['mass_eff'].unique())==1:
              mass_e_eff_factor = dF['mass_eff'].unique()[0]/CONST.MASS_E
          else:
              raise Exception('Multiple paramters for one grain are invalid.')

          if len(dF['R'].unique())==1:
              grainsize_radius = dF['R'].unique()[0]
          else:
              raise Exception('Multiple paramters for one grain are invalid.')
```

```
    material = Material(T_C,ND)
    grain = Grain(grainsize_radius=grainsize_radius,material=material)

    return grain
```

3.9.4 Refine algorithm

Visualize incorrectly found minima   Sometimes the minimization of the function does not work was expected. In most cases the searching algorithm may find a "just" a local minimum and miss the global one. Generally this problem could be solved by suppling additional hints to the minimization algorithm, where to look for the minimum. To do so we could first find the the points, where the minimization "error" is still too high and the recalculate those points with additional hints.

Here again, a graphical representation is helpful for a better understanding. Since we have solution for multiple doping levels (N_D) and multiple grain radii (R), I will represent each combination individually. The Pandas `groupby` function allows to split `DataFrames` temporally by a given group label. By grouping all the results by the label-tuple (N_D, R), Pandas will do the work and we can represent the individual result separately. Since the data holds also a column named `res` with the final "error" of the minimization, we can distinguish good and bad results easily.

```
[49]: gs = calc_dF_all.groupby(['ND','R'])
      fig, axe = subplots(len(gs)//5,5, figsize=(16,9), sharey=True)

      calc_dF_all['E_dot_init_kT_estimation'] = None
      for ax_i, ((ND,R), g) in enumerate(gs):
          #select the axe
          axe = fig.axes[ax_i]

          #selecting on ly the good results by using the 'res' field
          g_good = g[g['res']<2]

          #the bad ones are the complent of the good ones; droping the good
          #ones from all results leaves the bad ones back
          g_bad = g.drop(g_good.index)

          #plotting the good and bad results
          axe.plot(g_good['Einit_kT'], g_good['E_dot_init_kT'],'o-', label='good')
          axe.plot(g_bad['Einit_kT'], g_bad['E_dot_init_kT'],'o', label='bad')

          #some plotting sugar
          axe.set_yscale('symlog')
          axe.set_title(f'$N_D$:{ND}, R:{R*1e9:.2f}nm')
          axe.set_xlabel(r'$E_{Surface}$ [$k_BT$]', fontsize = 15)
          axe.set_ylabel(r'$\dot{E}_{Surface}$ [$k_BT$]', fontsize = 15)
```
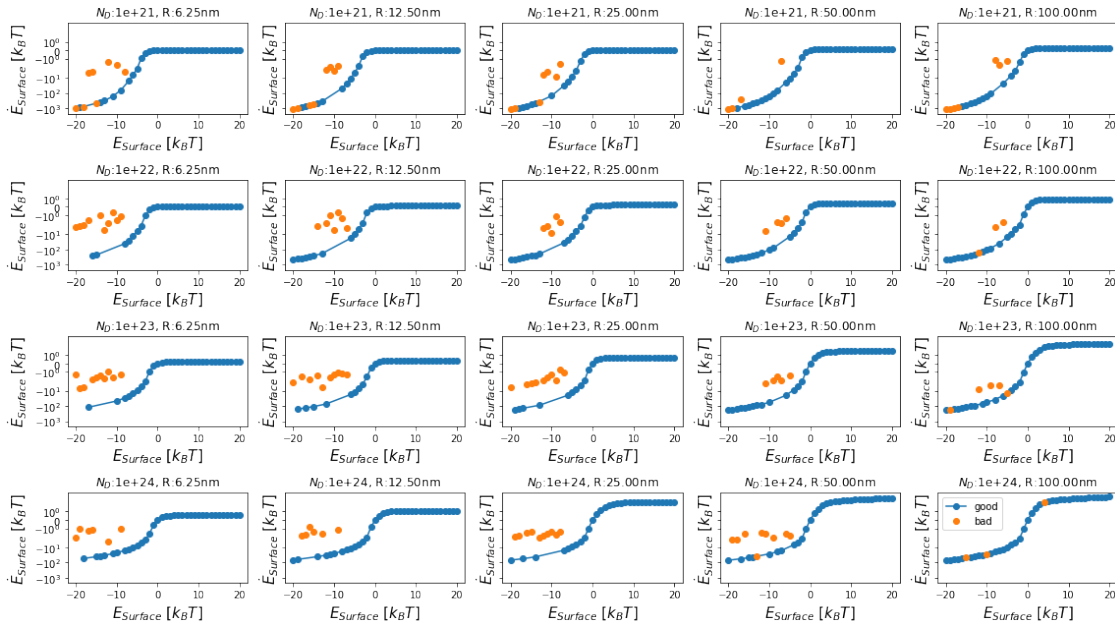
```
axe.legend()
fig.tight_layout()
```


Interpolate better solutions of incorrectly found minimum   The good results can be distinguished well from the bad ones, where the algorithm failed. Additionally a trained eye is able to estimate the interval, where the correct solution should lye. As humans can, so does Python. For this problem a the good solutions are used to estimate the correct solution of the failed points. In the following plot, this is done. Since we will need the predicted result as a starting point for a second minimization, we will add this value to the DataFrame holding all solutions and naming it appropriately . Since this function might become very handy to check the solution, we will wrap it into a function and reuse it later.

```python
[50]: from scipy.interpolate import interp1d
      def check_solutions(calc_dF_all):
          gs = calc_dF_all.groupby(['ND','R'])
          fig, axes = subplots(len(gs)//5,5, figsize=(16,9), sharex=True, sharey=True)

          calc_dF_all['E_dot_init_kT_estimation'] = None
          for ax_i, ((ND,R), g) in enumerate(gs):
              axe = fig.axes[ax_i]
              g_good = g[g['res']<2]
              g_bad = g.drop(g_good.index)
              axe.plot(g_good['Einit_kT'], g_good['E_dot_init_kT'],'o-', label='Good␣
      ↪solution')
```

```
        axe.plot(g_bad['Einit_kT'], g_bad['E_dot_init_kT'],'o', label='Bad␣
↪solution')
        axe.set_yscale('symlog')
        axe.set_title(f'$N_D$:{ND}, R:{R*1e9:.2f}nm')
        axe.set_xlabel(r'$E_{Surface}$ [$k_BT$]', fontsize =15)
        axe.set_ylabel(r'$\dot{E}_{Surface}$ [$k_BT$]', fontsize =15)

        x = g_good['Einit_kT']
        y = g_good['E_dot_init_kT']
        w = g_good['res']

        #create the interpolation
        interp = interp1d(x,y,kind='cubic',bounds_error=False,␣
↪fill_value='extrapolate')

        g_bad_correct_y  = interp(g_bad['Einit_kT'])
        axe.plot(g_bad['Einit_kT'], g_bad_correct_y, '*k',
                markersize=10, alpha=0.5, label='Interpolated guess')


        calc_dF_all.loc[g_bad.index, 'E_dot_init_kT_estimation'] =␣
↪g_bad_correct_y


    axe = fig.axes[0]
    axe.legend(*axes[0][0].get_legend_handles_labels())
    fig.tight_layout()
check_solutions(calc_dF_all)
```
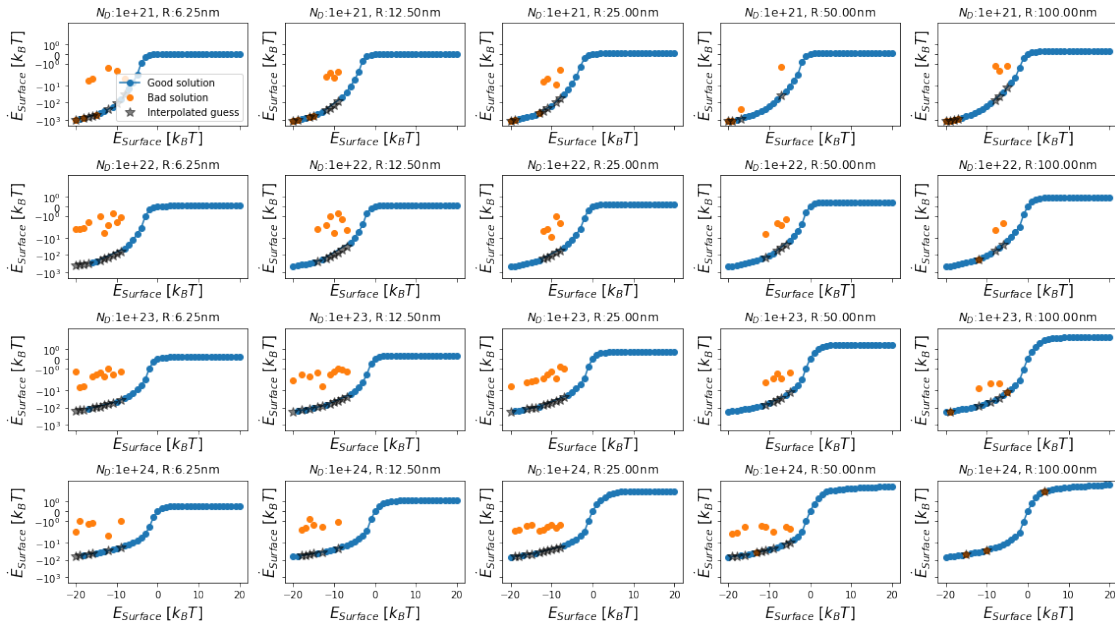
3.9.5 Recalculating the incorrect minima (with boundaries)

By using the estimated correct values of the slope at the surface of the grain, the function `find_best_E_dot_init` for the bad solutions is repeated. Since the good solutions will have no value for the "estimated correct value", we will use this fact to select the rows in the DataFrame which need to be recalculated.

```
[27]: def recalculate_by_index(index):
          ser_temp = calc_dF_all.loc[index].copy()
          grain = create_grain_from_data(ser_temp)
          E_init_kT = ser_temp['Einit_kT']



          #This is the estimated value from the interpolation of the good solutions
          estim = ser_temp['E_dot_init_kT_estimation']

          #Since the correct solution sould be in proximity of this solution, an
      ↪interval
          #is created from this value (here +-10%)
          bounds = sorted((estim*0.9, estim*1.1))

          ser_new = solve_grain_for_E_init_kT(E_init_kT,grain, debug = False, bounds =
      ↪bounds)
          ser_new.name = ser_temp.name
          ser_temp.update(ser_new)
          return ser_temp

      #calc_dF_all.loc[ser_temp.name] = ser_new
```

```
[28]: index_to_recalc = list(calc_dF_all[calc_dF_all['res']>2].
      ↪dropna(subset=['E_dot_init_kT_estimation']).index)
      from multiprocessing import Pool

      if __name__ == '__main__':
          p = Pool(8)
          new_sers = p.map(recalculate_by_index, index_to_recalc)
```

And finally the old solutions in the DataFrame are replaced with the (hopefully) better ones and saved again in a file.

```
[29]: for s in new_sers:
          calc_dF_all.loc[s.name] = s

      calc_dF_all.to_hdf('results.h5', 'corr', mode='a')
```

3.9.6 Checking the corrected solutions

Now the solution should all be available and a simple representation should reveal some insight about its quality.

```
[51]: calc_dF_all = pd.read_hdf('results.h5', 'corr')
      print(len(calc_dF_all[calc_dF_all['res']>1]))
      check_solutions(calc_dF_all)
      print(f'The maximal "error" of the minimization is :{calc_dF_all["res"].max()}')
```

```
71
The maximal "error" of the minimization is :53.768085981645115
```
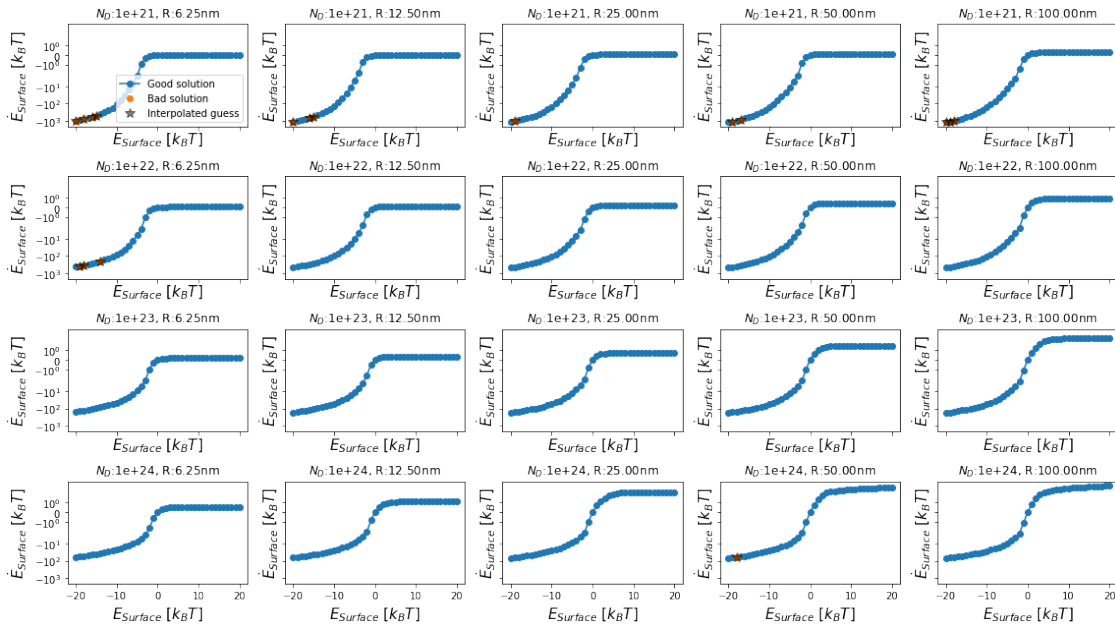

3.10 Shape of the potential drop inside the grain

Since all data is available now, the shape of the conduction band inside the grain can be represented for the different combinations of parameters.

```
[74]: for ND, calc_dF in calc_dF_all.groupby('ND'):
          fig, axes= subplots(3,2,figsize = (16,9), sharex=True)

          fig.suptitle(f'ND = {ND}'+ r'$\frac{1}{m^3}$', fontsize = 22)


          for ax_i, (R, calc_dF_grainsize) in enumerate(calc_dF.groupby('R')):
              axe = fig.axes[ax_i]
              axe.set_ylim(-20,20)
```

```python
        grain = create_grain_from_data(calc_dF_grainsize)


        axe.axhline(-grain.material.J_to_kT(grain.material.Diff_EF_EC),
                    linestyle='--',color='r', label='Fermi Level')

        axe.axvline(grain.R/grain.material.LD,
                    linewidth=3, color='k', label='Grain surface')



        axe.axhspan(-1,+1,color='r', alpha=0.2, label='$0\pm 1 [k_BT]$')

        for vinit, ser_temp in calc_dF_grainsize.iterrows():

            #discarde bad solutions from the plot
            if ser_temp['res']>5:
                continue

            r = ser_temp['r']
            v = ser_temp['v']
            vdot = ser_temp['v_dot']

            axe.set_title(f'Grain radius = {grain.R*1e9:.2f}nm', fontsize=22)

            axe.plot(r,v, '-', label = "")
            axe.set_ylabel('$E_C(r)$ [$k_BT$]', fontsize =22)
            axe.set_xlabel(f'Position in grain [$L_D$ = {grain.material.LD*1e9:.
↪2f}nm]',
                           fontsize =22)
        axe.legend()
        axe.grid(b=True)

    fig.axes[-1].set_axis_off()
    fig.tight_layout()
    fig.subplots_adjust(top=.85)
    close()
    display(fig)
    for i in range(5):print()
```
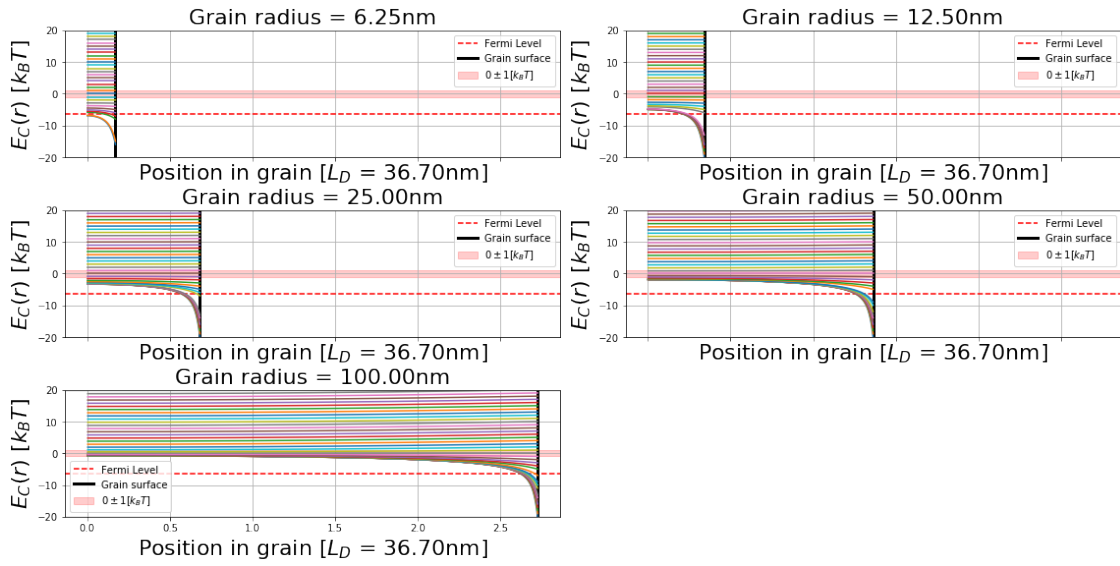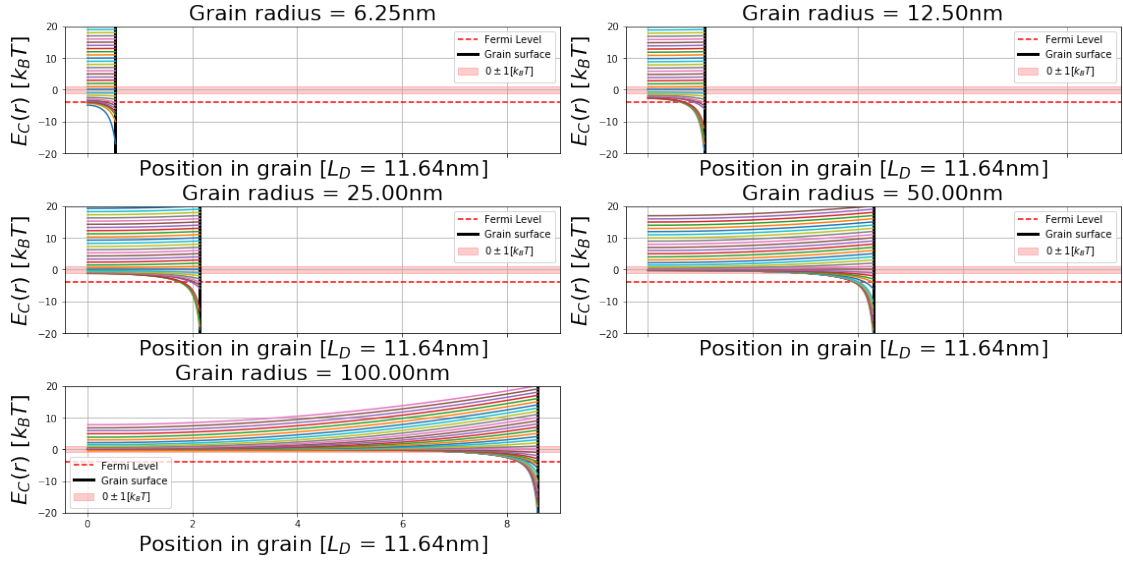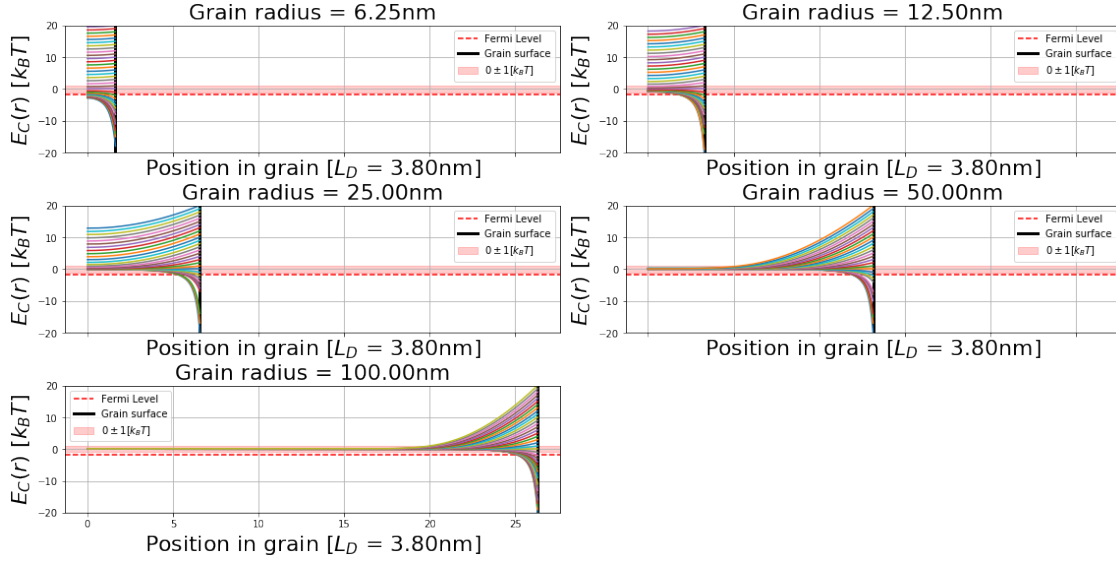
ND = $1e+21\frac{1}{m^3}$

Grain radius = 6.25nm

Grain radius = 12.50nm

Grain radius = 25.00nm

Grain radius = 50.00nm

Grain radius = 100.00nm

ND = $1e+22\frac{1}{m^3}$

Grain radius = 6.25nm

Grain radius = 12.50nm

Grain radius = 25.00nm

Grain radius = 50.00nm

Grain radius = 100.00nm

$$ND = 1e+23 \frac{1}{m^3}$$

$$ND = 1e+24\frac{1}{m^3}$$

This graph shows how a surface potential is shielded by the remaining ionized donors. In the case of on deletion layer ($E_{C_{Surface}} > 0$)), the total number of charges shielding the surface potential is rather small compared to the amount of charges in an accumulation layer ($E_{C_{Surface}} < 0$)). The result of such an asymmetry is visible in the graph. The width of the accumulation layer is by far smaller then the width of the depleted are.

4 Summary

In this notebook the flowowing steps have been accomplished:

- numerically calculate the charge density in a semiconductor
- solve the Poisson equation for spherical grains
- Calculate the solutions for multiple grain sizes and surface potentials

Those calculations have been derived with a standard set of Python tools. By using mainly the `numpy`, `scipy`, `matplotlib` and `pandas` these results have been achieved.

To avoid a to large blocks of information in one notebook I like to introduce a breakpoint here. At such breakpoints it is helpful to save all the relevant gathered data in a `DataFrame`, save it to the filesystem, and pick it up again in a fresh notebook. This keeps each notebooks close to one topic and and introduces directly structure in the data.

In the next notebook this calculated date will be used derive the total resistance of a grain under

different conditions. The anisotropic charge carrier distribution inside the grain has a mayor influence on the total resistance. For two extreme cases, the conduction path inside the grain differs a lot. Those cases are:

1. Accumulation layer at the surface
2. Depletion layer at the surface

In the case of 1., the current will most likely run along the highly conductive surface of the grain. In the second case, the current will need to overcome a highly resistive surface layer and then propagate through the inside of the relatively low resistive bulk of the grain.

Since all information to numerically derive the effects are now pre-calcualted, the next notebook will start at this point and continue to calcualte the total resistance.Non-PDF readers, could use this link to guide them to the next notebook.

5 Bibliography section

References

[BD05]   BATZILL, Matthias ; DIEBOLD, Ulrike: The surface and materials science of tin oxide. In: *Progress in Surface Science* 79 (2005), Nr. 2-4, S. 47–154. `http://dx.doi.org/10.1016/j.progsurf.2005.09.002`. – DOI 10.1016/j.progsurf.2005.09.002. – ISBN 0079–6816

[Bel07]   BELTON, P. S.: *Preface*. Plenum Press, New York, 2007. – 1–247 S. `http://dx.doi.org/10.1002/9780470995792`. `http://dx.doi.org/10.1002/9780470995792`. – ISBN 1405121270

[BHW11]   BÂRSAN, N. ; HÜBNER, M. ; WEIMAR, U.: Conduction mechanisms in SnO2 based polycrystalline thick film gas sensors exposed to CO and H2 in different oxygen backgrounds. In: *Sensors and Actuators, B: Chemical* 157 (2011), Nr. 2, S. 510–517. `http://dx.doi.org/10.1016/j.snb.2011.05.011`. – DOI 10.1016/j.snb.2011.05.011. – ISBN 0925–4005

[BRW15]   BARSAN, Nicolae ; REBHOLZ, Julia ; WEIMAR, Udo: Conduction mechanism switch for SnO<inf>2</inf> based sensors during operation in application relevant conditions; Implications for modeling of sensing. In: *Sensors and Actuators, B: Chemical* 207 (2015), feb, Nr. Part A, 455–459. `http://dx.doi.org/10.1016/j.snb.2014.10.016`. – DOI 10.1016/j.snb.2014.10.016. – ISSN 09254005

[BW03]   BÂRSAN, N. ; WEIMAR, U.: Understanding the fundamental principles of metal oxide based gas sensors; the example of CO sensing with SnO2 sensors in the presence of humidity. In: *Journal of Physics Condensed Matter* 15 (2003), Nr. 20, R813–R839. `http://dx.doi.org/10.1088/0953-8984/15/20/201`. – DOI 10.1088/0953–8984/15/20/201. – ISBN 0953–8984

[JOPO15]   JONES, Eric ; OLIPHANT, Travis ; PETERSON, Pearu ; OTHERS: *SciPy: Open Source Scientific Tools for Python, 2001 (http://www.scipy.org/)*. `http://www.scipy.org/`. Version: 2015

[OBW09] OPREA, Alexandru ; BÂRSAN, Nicolae ; WEIMAR, Udo: Work function changes in gas sensitive materials: Fundamentals and applications. In: *Sensors and Actuators, B: Chemical* 142 (2009), Nr. 2, 470–493. http://dx.doi.org/10.1016/j.snb.2009.06.043. – DOI 10.1016/j.snb.2009.06.043. – ISSN 09254005

[RK04] ROTHSCHILD, Avner ; KOMEM, Yigal: The effect of grain size on the sensitivity of nanocrystalline metal-oxide gas sensors. In: *Journal of Applied Physics* 95 (2004), Nr. 11 I, S. 6374–6380. http://dx.doi.org/10.1063/1.1728314. – DOI 10.1063/1.1728314. – ISSN 00218979

[SN07] SZE, S.M. ; NG, Kwok K.: *Physics of Semiconductor Devices*. 2007. – 293–373 S. http://dx.doi.org/10.1049/ep.1970.0039. http://dx.doi.org/10.1049/ep.1970.0039. – ISBN 0471143235

[VGO+20] VIRTANEN, Pauli ; GOMMERS, Ralf ; OLIPHANT, Travis E. ; HABERLAND, Matt ; REDDY, Tyler ; COURNAPEAU, David ; BUROVSKI, Evgeni ; PETERSON, Pearu ; WECKESSER, Warren ; BRIGHT, Jonathan ; WALT, Stéfan J. ; BRETT, Matthew ; WILSON, Joshua ; MILLMAN, K J. ; MAYOROV, Nikolay ; NELSON, Andrew R J. ; JONES, Eric ; KERN, Robert ; LARSON, Eric ; CAREY, C J. ; POLAT, İlhan ; FENG, Yu ; MOORE, Eric W. ; VANDERPLAS, Jake ; LAXALDE, Denis ; PERKTOLD, Josef ; CIMRMAN, Robert ; HENRIKSEN, Ian ; QUINTERO, E A. ; HARRIS, Charles R. ; ARCHIBALD, Anne M. ; RIBEIRO, Antônio H ; PEDREGOSA, Fabian ; MULBREGT, Paul van ; SCIPY 1.0 CONTRIBUTORS: SciPy 1.0: fundamental algorithms for scientific computing in Python. In: *Nature methods* (2020). http://dx.doi.org/10.1038/s41592-019-0686-2. – DOI 10.1038/s41592–019–0686–2. – ISSN 1548–7105

This page is intentionally left blank.

From SMOx-grains to sensor resistance

February 19, 2020

Contents

1 Abstract

The gas sensors being investigated consists of multiple, small semiconducting grains. In the previous chapter the effect various of surface conditions onto the SMOX grain were modeled and calculated. The results show, that the charge distribution inside the grain varies and depends on the surface potential and defect concentration of the semicondutor. When such a grain is used as a sensor, a bias voltage is applied and the resulting resistance is measured. To numerically derive the total resistance from a specific charge distribution inside the grain, actual conduction path through the grain needs to be understood. In this part the conduction path will be numerically derived and in a second step the dependency on the overall resistance will be derived.

2 Review

In the last notebook/chapter the semiconductor part of the SMOX grains was addressed. This included the numerical calculation the charge carrier density as as function of the conduction band bending by solving the Poisson equation for spherical grains. The results for multiple grains are saved to file and can now be used here again without recalculating them. Additionally a Python module was created. In this module all the functions and variables from the previous notebook are merged together and saved in file called part2.py. By importing this file, all elements will be accessible also in this notebook. The following command will do the job: `from part2 import *`

```python
[1]: %pylab inline
     %load_ext autoreload
     %autoreload 2
     from part2 import *
```

```
Populating the interactive namespace from numpy and matplotlib
```

3 Load the results

```python
[2]: calc_dF = pd.read_hdf('results.h5', 'corr')
     calc_dF.index = range(len(calc_dF))
     calc_dF_all = calc_dF[calc_dF['res']<2]
```

Again in this notebook the final results from the previous chapter are represented. The redundancy of such a representation (especially in the printed version) may seem overexaggerated. But when working with the interactive Jupyter notebook this choice seems legitimated, since each notebook represents a self-contained element of research. It would be optimal if all data and representation are available at place. Since the representation of the data as seen in the last chapter is of essential importance, it should be added here again.

Finding the compromise of this thesis between the printed version and the interactive notebook generates in this case some "glitches". With the strong focus to introduce the Jupyter notebook environment in this thesis the redundant representation of the figures was chosen in this case.

```python
[3]: for ND, calc_dF in calc_dF_all.groupby('ND'):
         fig, axes= subplots(3,2,figsize = (16,9), sharex=True)

         fig.suptitle(f'ND = {ND}'+ r'$\frac{1}{m^3}$', fontsize = 22)


         for ax_i, (R, calc_dF_grainsize) in enumerate(calc_dF.groupby('R')):
             axe = fig.axes[ax_i]
             axe.set_ylim(-20,20)

             grain = create_grain_from_data(calc_dF_grainsize)


             axe.axhline(-grain.material.J_to_kT(grain.material.Diff_EF_EC),
```

```python
                    linestyle='--',color='r', label='Fermi Level')

        axe.axvline(grain.R/grain.material.LD,
                    linewidth=3, color='k', label='Grain surface')



        axe.axhspan(-1,+1,color='r', alpha=0.2, label='$0\pm 1 [k_BT]$')

        for vinit, ser_temp in calc_dF_grainsize.iterrows():

            #discarde bad solutions from the plot
            if ser_temp['res']>5:
                continue

            r = ser_temp['r']
            v = ser_temp['v']
            vdot = ser_temp['v_dot']

            axe.set_title(f'Grain radius = {grain.R*1e9:.2f}nm', fontsize=22)

            axe.plot(r,v, '-', label = "")
            axe.set_ylabel('$E_C(r)$ [$k_BT$]', fontsize =22)
            axe.set_xlabel(f'Position in grain [$L_D$ = {grain.material.LD*1e9:.
↪2f}nm]',
                           fontsize =22)
        axe.legend()
        axe.grid(b=True)

    fig.axes[-1].set_axis_off()
    fig.tight_layout()
    fig.subplots_adjust(top=.85)
    close()
    display(fig)
    for i in range(5):print()
```
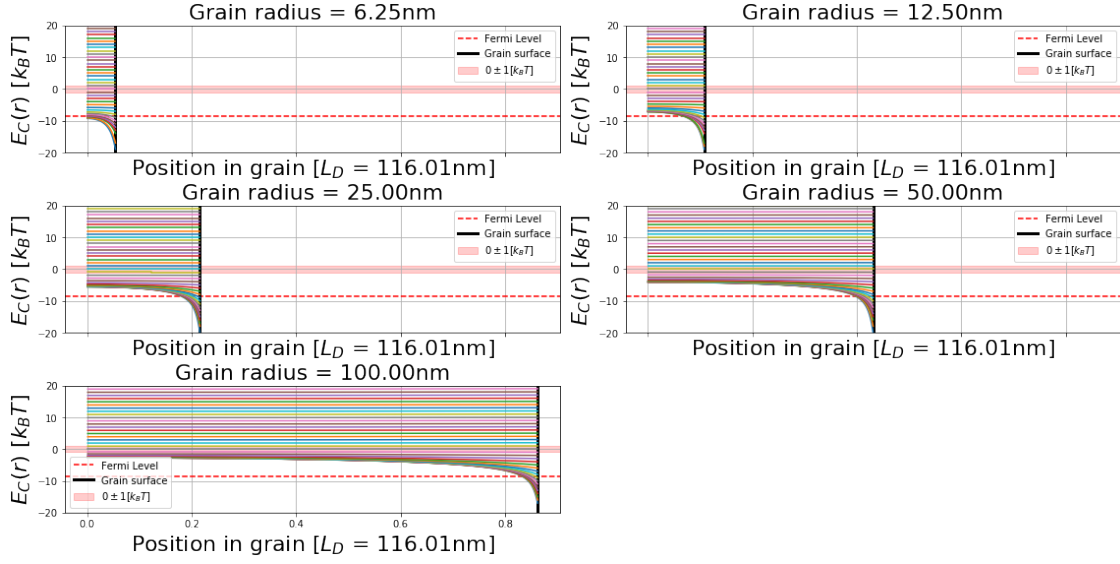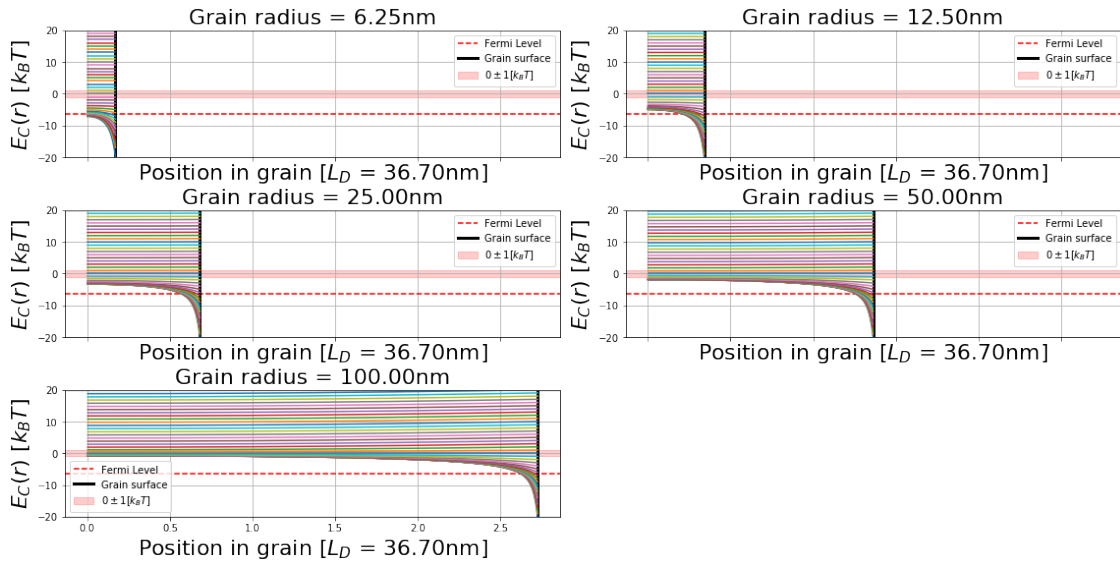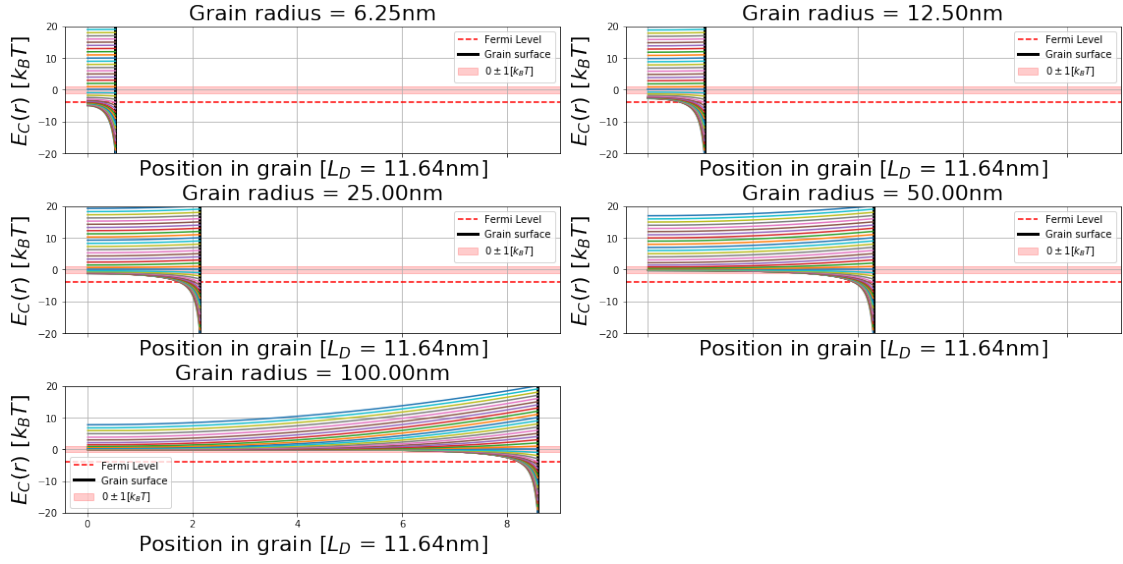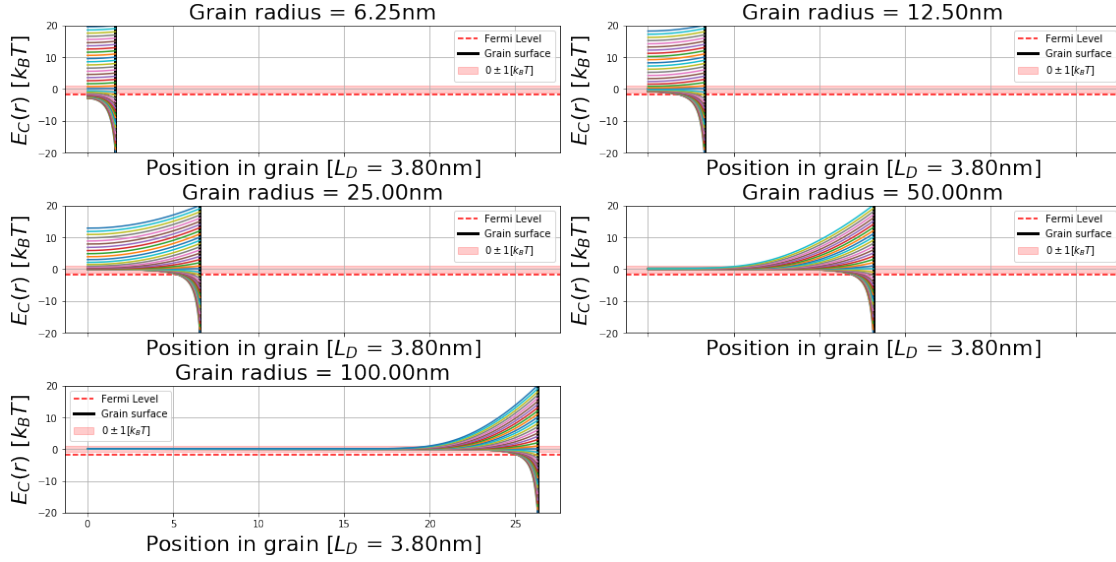
ND = $1e+21\frac{1}{m^3}$

ND = $1e+22\frac{1}{m^3}$

$$ND = 1e+23 \frac{1}{m^3}$$

$$ND = 1e + 24 \frac{1}{m^3}$$

This graph shows how a surface potential is shielded by the remaining ionized donors. In the case of on deletion layer ($E_{C_{Surface}} > 0$)), the total number of charges shielding the surface potential is rather small compared to the amount of charges in an accumulation layer ($E_{C_{Surface}} < 0$)). The result of such an asymmetry is visible in the graph. The width of the accumulation layer is by far smaller then the width of the depleted are.

4 From charge distribution to resistance

4.1 The "numerical" grain

With the previous tools and calculation the position of the conduction band inside the grain is known. With the previously defined `Material` it is possible to calculate the exact number of free charges in the conduction band. With these information it is now possible to assign each point inside the grain a certain charge density n. From the charge density the conductivity can be derived. The conductivity of a semiconductor is defined by:

$$Conductivity = \sigma = q * (n * \mu_n + p * \mu_p) \tag{1}$$

Here q is the electrical charge of an electron, n the density of electrons in the conduction band and μ_n the mobility of the electrons. Focusing on the description of SnO_2, which is an n-type semiconductor with $n >> p$, the conductivity can be simplified to the following equation:

$$Conductivity = \sigma = q * n * \mu_n \tag{2}$$

The relation between resistivity ρ and the conductivity is given by:

$$R = \frac{\rho * l}{A} \tag{3}$$

$$Resistivity = \rho = \frac{1}{\sigma} \tag{4}$$

To derive from the known conductivity inside the grain, the total resistance of the grain, the current path needs to be known. The current flow along the field lines inside the grain, which are equivalent to the gradient of the potential. Therefore the potential distribution inside the grain needs to be known first. To do this, the grain is represented by a numerical model. This model is created by slicing it into equal distributed cubes of the same size. Each cube will have a defined conductivity and potential. The colored areas in the picture indicate the areas, where a bias potential will be applied to generate a virtual electrical field.
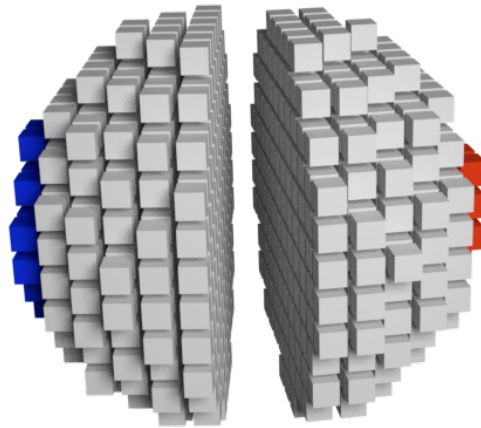
To simulate the spread of the bias potential areas inside the grain, a technique called relaxation will be used. The general idea is to guess an initial potential distribution, and then, based on the laws of physics, iteratively correct this guess. The correction is done by re-calculating each time the potential U_0 of one center-"cube" based on the potentials U_i and conductivity σ of the direct neighbors.

By doing this for each "cube", the potential distribution will more and more apply to the physical solution. When approaching to the solution, the overall changes in the potential of each cube will get smaller and smaller. In an ideal case it will not change anymore. In this case the potential of each cube will be just as it should be to fulfill the laws of physics.

To understand how this process can be supported by the means of modern matrix operations, I will shorty derive how U_0 is calculated from the surrounding U_i. In a second step matrix convolution will be used to solve the problem efficiently. First we will need to combine Ohm's law and

Kirchhoff's first law:

$$R = \frac{\Delta U}{I} = \frac{\rho * l}{A} \ and \sum_i I_i = 0 \tag{5}$$

$$\rightarrow \sum_i \frac{\Delta U_i}{R_i} = \sum_i \frac{U_0 - U_i}{\rho_i} \frac{A}{l} = 0 \tag{6}$$

$$\rightarrow \sum_i \frac{U_0 - U_i}{\rho_i} \frac{A}{l} = 0 \tag{7}$$

$$\rightarrow \sum_i \frac{U_0}{\rho_i} = \sum_i \frac{U_i}{\rho_i} \tag{8}$$

$$\rightarrow U_0 \sum_i \frac{1}{\rho_i} = \sum_i \frac{U_i}{\rho_i} \tag{9}$$

$$\rightarrow U_0 = \frac{\sum_i \frac{U_i}{\rho_i}}{\sum_i \frac{1}{\rho_i}} \tag{10}$$

$$\rightarrow U_0 = \frac{\sum_i U_i * \sigma_i}{\sum_i \sigma_i} = \frac{\sum_i U_i * q * n * \mu_n}{\sum_i q * n * \mu_n} \tag{11}$$

$$= \frac{q * \mu_n}{q * \mu_n} \frac{\sum_i U_i * n_i}{\sum_i n_i} = \frac{\sum_i U_i * n_i}{\sum_i n_i} \qquad (U_0 \text{ from } U_i)$$

$$\tag{12}$$

With A the cube face area and l the distance between the cube centers, which vanish in the equation when all cubes have equal sizes. Additionally μ_n is assumed to be constant inside the grain. This simplification is not necessary for the further calculation and could also be treated as a position dependent variable like σ_i. In the course of this thesis μ_n will be kept constant.

To evaluate the each n_i at arbitrary points r inside the grain, one additional step is needed. Due to the nature of the numerical solution from the previous notebook we know the value of n only at specific points. For values between those fix-points, an interpolation between the neighbors can be used. Again, SciPy and Python offer here also a easy to use and robust solution. `from scipy import interpolate` adds the `interpolate` module into the kernel. The `interp1d` function of this module is described (here) as follows: >Interpolate a 1-D function. > >x and y are arrays of values used to approximate some function f: y = f(x). This class returns a function whose call method uses interpolation to find the value of new points.

Since the values of n and r exist already precalculated for specific points in the Dataframe, the following function is used to create the appropriate function for continuous values of r.

```
[4]: from scipy import interpolate

     def get_interpolated_n_v(ser,grain):

         v = ser['v']
         r = ser['r']
         n = ser['n']

         r[0] = 0
```

```python
    n_int = interpolate.interp1d(r*grain.material.LD, n, kind='cubic')
    v_int = interpolate.interp1d(r*grain.material.LD, v, kind='cubic')

    return n_int, v_int
```

As mentioned earlier, the positions of the applied virtual bias potential will only be the cubes on the far left and right, as indicated in the picture of the sliced cube. By arranging the bias voltage like this, the potential inside the grain will have a rotational symmetry along the axis connecting the two poles. The benefit of the resulting symmetry is that the potential inside the grain can be described by a $N \times N$ matrix, where N are the number of cubes inside the grain. Since $N \times N$ data structures are very common in modern application fields like computer vision and image recognition, many algorithm dealing with such data structures are available and optimized. In this notebook we will not reach out for potentially even faster state-of-the-art implementations like PyTorch or Tensorflow to deal with this matrix/tensor, but rather stick to the well established tool of SciPy. A nice review on about what SciPy is can be found here: [VGO$^+$20] (SciPy 1.0: fundamental algorithms for scientific computing in Python). On advantage of using "just" SciPy is, that it is easily available on most operation system and second the performance is good enough for this use case.

First we create now the data structure of the grain for further simulations. The $N \times N$ cubes will be represented by a `numpy` array.

```python
[5]: def initaliz_d_v(d_v, d_mask, v):
        d_v[:,1] = -v
        d_v[:,0] = -v
        d_v[:,-2] = +v
        d_v[:,-1] = +v
        d_v = d_v*d_mask
        return d_v


    def pos_to_r(xi,yi,grain, cube_size, d):
        '''
        By passing the xi and yi indices, the grain and one array, the position (r)
        inside the grain is return
        '''

        cx = d.shape[0]//2+1-1 #find the center; length divided by two without rest;␣
    ↪+1;
                             #-1 since we start counting at 0
        cy = d.shape[1]//2+1-1

        ri = (((xi-cx))**2+((yi-cy))**2)**0.5

        r = (ri*cube_size)
        return r

    def r_to_pos(r, grain, cube_size, d_v):
```

```python
    center = d_v.shape[0]//2
    return int(round(r/cube_size))+center


def create_numerical_grain_matrix( grain, ser,cube_size):
    #these functions are needed to calculate the value of n and
    #v at arbitrary positions
    n_int, v_int = get_interpolated_n_v(ser, grain)

    #calc. number of cubes inside the grain
    # having a uneven number ensures having a defined center layer
    nx = ny = 1+2*int(round((grain.R/cube_size)))

    #initalize the data with zeros

    #data for voltages =d_v
    #data for the conductivity = d_cond
    d_v = np.zeros((nx,ny))
    d_cond = np.zeros((nx,ny))

    #and additionally a mask
    #for values outside the grain
    d_mask = np.zeros((nx,ny))

    # now the data arrays will be filled with values
    for xi in range(d_cond.shape[0]):
        for yi in range(d_cond.shape[1]):

            #calcualte the position iside the grain
            #from the cubes position
            r = float(pos_to_r(xi,yi,grain, cube_size, d_v))

            try:
                #if r is outside the grain, n_int(r) raise an error
                #and the function jumps to the "except" part

                #otherwise the conductivity will be saved in units of nb
                #inside the d_cond array

                condu = n_int(r)/grain.material.nb
                d_cond[xi, yi] = condu

                #since this point is inside the grain, the mask is 1
                d_mask[xi,yi] = 1

            except ValueError:
                #outside the grain
```

70

```
                    d_cond[xi, yi] = 0
                    d_mask[xi,yi] = 0
        d_v = initaliz_d_v(d_v, d_mask, 1000)
        return d_v, d_cond, d_mask
```

4.2 Precalc the numerical grains for all conditions

The grain data structure can now a represented graphically. For faster interactive response, we will pre initialize all the grains for the data available. Due to the similarity of the $N \times N$ data structure to common pixel based pictures, the function `imshow` is very handy to represent the data.

```
[18]: d_cond_plots = []
      for i, ser in calc_dF_all.iterrows():
          print(f'Initalized {i+1} of {len(calc_dF_all)}.', end='\r', flush=True)
          grain = create_grain_from_data(ser)
          d_v, d_cond,d_mask = create_numerical_grain_matrix( grain,␣
       ↪ser,cube_size=grain.R/50)
          d_cond_plot = d_cond.copy()
          d_cond_plot[np.where(d_mask==0)]=None
          d_cond_plots.append(d_cond_plot)
      calc_dF_all.loc[:, 'd_cond'] = d_cond_plots
```

```
Initalized 820 of 803.
```

To visualize the grain, the conductivity is represented for different initial surface potentials expressed in units of $\frac{k_B T}{e}$

```
[22]: def plot_grain_states(calc_dF_grainsize, vmax=None, vmin=None):
          fig, axes = subplots(3,3, figsize = (16,10))

          grain = create_grain_from_data(calc_dF_grainsize)
          for axe in fig.axes:axe.set_visible(False)

          plot_E_init_kT = [-20,-10,-5,-1,0,1,5,10,20]
          plot_dF = calc_dF_grainsize.loc[calc_dF_grainsize['Einit_kT'].
       ↪isin(plot_E_init_kT)].sort_values(by='Einit_kT')
          for ax_i, (vinit, ser) in enumerate(plot_dF.iterrows()):
              axe = fig.axes[ax_i]
              axe.set_visible(True)
              axe.set_facecolor('grey')

              Einit_kT = ser['Einit_kT']

              axe.set_title(r'$E_{C_{Surface}}=$'+f'{Einit_kT}[$k_BT$]')
              axe.set_ylabel('x [nm]')
              axe.set_xlabel('y [nm]')
```

```python
        d_cond_plot =ser['d_cond']# calc_dF.loc[ser.name, 'd_cond']

        #using axe.imshow to plot the data on the axe
        axe.grid(b=True, zorder=-5)
        im = axe.imshow(vmax-np.log(d_cond_plot)+vmin, interpolation='bicubic',
                        extent=(-grain.R*1e9, grain.R*1e9, -grain.R*1e9, grain.
 ↪R*1e9),

                        vmax=vmax*2, vmin=vmin, cmap='hot', zorder=2)

    fig.tight_layout()
    fig.subplots_adjust(top=0.9)
    ND = calc_dF_grainsize['ND'].unique()[0]
    fig.suptitle(f'ND: {ND}', fontsize=22)



def plot_conductivity(GrainRadius, ND=1e21):
    R = GrainRadius/1e9
    calc_dF_grainsize = calc_dF_all.groupby(['ND','R']).get_group((ND,R))
    max_n = np.log(calc_dF_grainsize['d_cond'].apply(lambda c:np.nanmax(c))).
 ↪max()
    min_n = np.log(calc_dF_grainsize['d_cond'].apply(lambda c:np.nanmin(c))).
 ↪min()
    plot_conduction_band(calc_dF_grainsize)


    plot_grain_states(calc_dF_grainsize, vmax = max_n, vmin = min_n)



def plot_conduction_band(calc_dF_grainsize):
    fig, axe = subplots(figsize = (16,10))
    grain = create_grain_from_data(calc_dF_grainsize)

    axe.axhline(-grain.material.J_to_kT(grain.material.Diff_EF_EC),
                linestyle='--',color='r', label='Fermi Level')

    axe.axvline(grain.R/grain.material.LD,
                linewidth=3, color='k', label='Grain surface')

    axe.grid()


    axe.axhspan(-1,+1,color='r', alpha=0.2, label='$0\pm 1 [k_BT]$')

    for vinit, ser_temp in calc_dF_grainsize.iterrows():

        #discarde bad solutions from the plot
```

```python
        if ser_temp['res']>5:
            continue

        r = ser_temp['r']
        v = ser_temp['v']
        vdot = ser_temp['v_dot']

        axe.set_title(f'Grain radius = {grain.R*1e9:.2f}nm', fontsize=22)

        axe.plot(r,v, '-', label = "")
        axe.set_ylabel('$E_C(r)$ [$k_BT$]', fontsize =22)
        axe.set_xlabel(f'Position in grain [$L_D$ = {grain.material.LD*1e9:.
↪2f}nm]',
                       fontsize =22)
        axe.legend()


use_interactive_controls = False

if use_interactive_controls:
    from ipywidgets import interact, interactive, fixed, interact_manual
    import ipywidgets as widgets
    grainsizes = list(calc_dF_all['R'].unique())
    interact(plot_conductivity,
             GrainRadius=np.array(grainsizes)*1e9,
             ND=list(calc_dF_all.groupby(['ND']).groups.keys()),
             text='Select a grainsize:');
else:
    GrainRadius = 50
    ND = 1e24
    plot_conductivity(GrainRadius, ND)
```
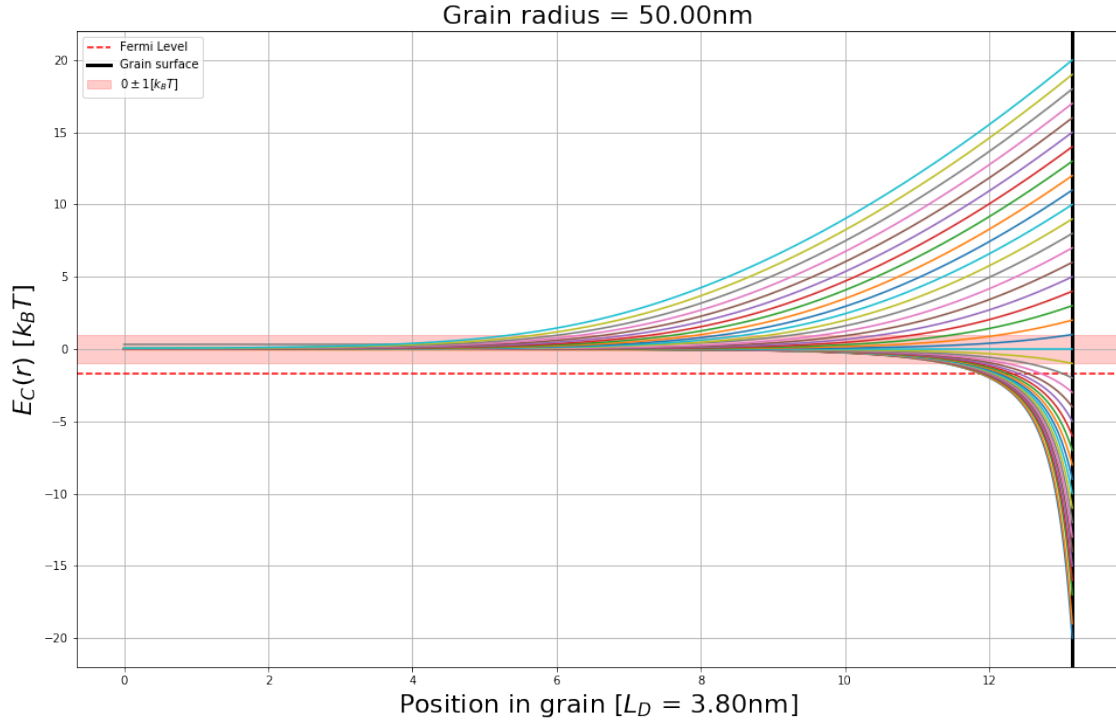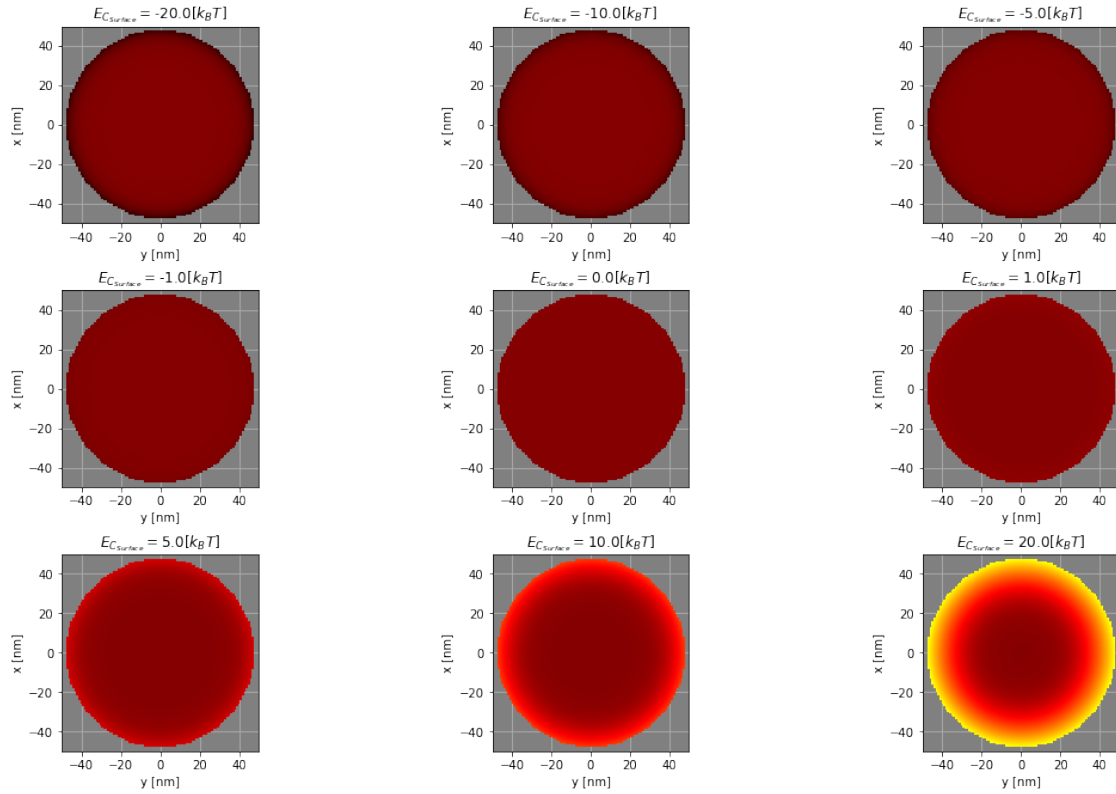
Grain radius = 50.00nm

ND: 1e+24

The top figure shows, how the position of the conduction band propagates inside the grain starting for surface energies from 20 $k_B T$ to -20 $k_B T$. The lower figure is the representation of $n^*(r) = \frac{n(r)}{n_b}$ inside the grain. For a surface potential of 20 $k_B T/e$ a large depleted surface layer is visible.

4.3 Relaxation

4.3.1 Convolution algorithm

Equation (U_0 from U_i) is the basis of the relaxation process. The potential at each cube will be recalculated according to: $U_0 = \frac{\sum_i U_i * n_i}{\sum_i n_i}$. The indices i stand for the direct neighbors of U_0. The following simple example should explain how `convolve2d` can be used to solve our task efficiently. The function needs two parameters as inputs. The first is the matrix itself, while the second is the description of the convolution operation. In a very short description, this is what the algorithm will do:

1. goto on datapoint i_x_y
2. multiply the neighbors of i_x_y with the corresponding value of the second argument
3. sum up the results and save it a the position of the datapoint i_x_y
4. do this for all data points

It would be out of the scope to dig deeper into the details of convolutions, but the following example should reveal the main concept it.

```python
[61]: from scipy import signal

      # the potential of the cubes at a certain point
      #with its direct neighbors
      U = np.array([[1,2,3],
                    [1,2,3],
                    [1,2,3]])
      print('U:')
      print(U)
      print()

      #the conductivity of each cube
      c = np.array([[1,1,1],
                    [10,10,10],
                    [100,100,100]])
      print('c:')
      print(c)
      print()

      #calculating the product of U and c
      print('U*c')
      print(U*c)
      print()
```

```
#the convolution matrix
conv = np.array([[0,1,0],
                 [1,0,1],
                 [0,1,0]])
print('Conv')
print(conv)
print()

#calculating the convolution
signal.convolve2d(U*c, conv, boundary='fill', mode='same', fillvalue=0)
```

U:
[[1 2 3]
 [1 2 3]
 [1 2 3]]

c:
[[1   1   1]
 [10 10 10]
 [100 100 100]]

U*c
[[1   2   3]
 [10 20 30]
 [100 200 300]]

Conv
[[0 1 0]
 [1 0 1]
 [0 1 0]]

[61]: array([[12, 24, 32],
             [121, 242, 323],
             [210, 420, 230]])

4.3.2 Example with 'convolve2d'

This example shows, how convolve2d is helpful for solving the relaxation problem. For instance the sum of the direct neighbors of the center is: $10 + 2 + 30 + 200 = 242$, This is exactly the value returned by convolve2d. The process of calculating the convoluted matrix is done for the nominator and the denominator. After the division U_0 is obtained. Some additional steps as masking the potentials outside the grain and setting the bias again are added. If the potential down not change anymore, the iterations can be stopped.

```
[63]: from scipy import signal


      def solve_relaxation(d_v, d_cond, d_mask, n = 10000000):
          res_new = 1000
          #shortly disable the error when dividing by zero (denominator)
          old_settings = np.seterr()
          np.seterr(divide='ignore', invalid='ignore')
          conv = [[0,1,0],[1,0,1],[0,1,0]]
          denominator = signal.convolve2d(d_cond, conv, boundary='fill',
                                          mode='same', fillvalue=0)
          for i in range(n):


              numerator = signal.convolve2d(d_v*d_cond, conv, boundary='fill',
                                            mode='same', fillvalue=0)

              d_v_new = (numerator/denominator)*d_mask
              d_v_new = np.nan_to_num(d_v_new,0)

              d_v_prev = d_v.copy()



              d_v = d_v_new.copy()
              d_v = initaliz_d_v(d_v, d_mask, 1000)

              res_pre = res_new
              res_new = np.abs(np.sum(d_v_prev-d_v))

              if i%10000==1:
                  #print(res_pre,res_new)
                  if ((res_pre - res_new)==0) and (i>40000):
                      break
          #setting back the defaults
          np.seterr(**old_settings)
          return d_v, d_cond, d_mask
```

4.3.3 See how a single solution evolves

```
[65]: import matplotlib.animation as animation

      c_dF = calc_dF_all.copy()

      ser =c_dF[(c_dF['R']==100e-9) & (c_dF['Einit_kT']==-8) & (c_dF['ND']==1e22)].
        →iloc[0]
```

```python
vinit = ser.name
cube_size = grain.R/50
ser['cube_size'] = cube_size
grain = create_grain_from_data(ser)


cube_size_value = cube_size


d_v, d_cond, d_mask = create_numerical_grain_matrix( grain,␣
 ↪ser,cube_size=cube_size_value)


ns = 1
def update(frame):
    axe.clear()
    axe_v.clear()
    n = 5
    n = conv_runs[frame]
    global d_v
    global ns
    ns+=n
    axe.set_title(f'Number relaxation interations: {ns}')
    axe_v.set_title('Potential inside from middle-left to middle-right')

    d_v, _, _ = solve_relaxation(d_v = d_v , d_cond=d_cond, d_mask=d_mask, n=n)
    d_v_plot = d_v.copy()
    d_v_plot[np.where(d_mask==0)]=None
    img = axe.imshow(d_v_plot,interpolation='bicubic',)


    axe_v.plot(d_v[r_to_pos(0,grain, cube_size, d_v),:])

    #plot_grad(axe_g, axe_c, d_v=d_v, d_mask=d_mask)

    return img



fig, axes = subplots(1,2, figsize = (16,9))
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

axe = axes[0]

img = axe.imshow(d_v)
cb = colorbar(img, ax = axe)
cb.ax.set_ylabel('Volage [V]')

axe_v = axes[1]
```
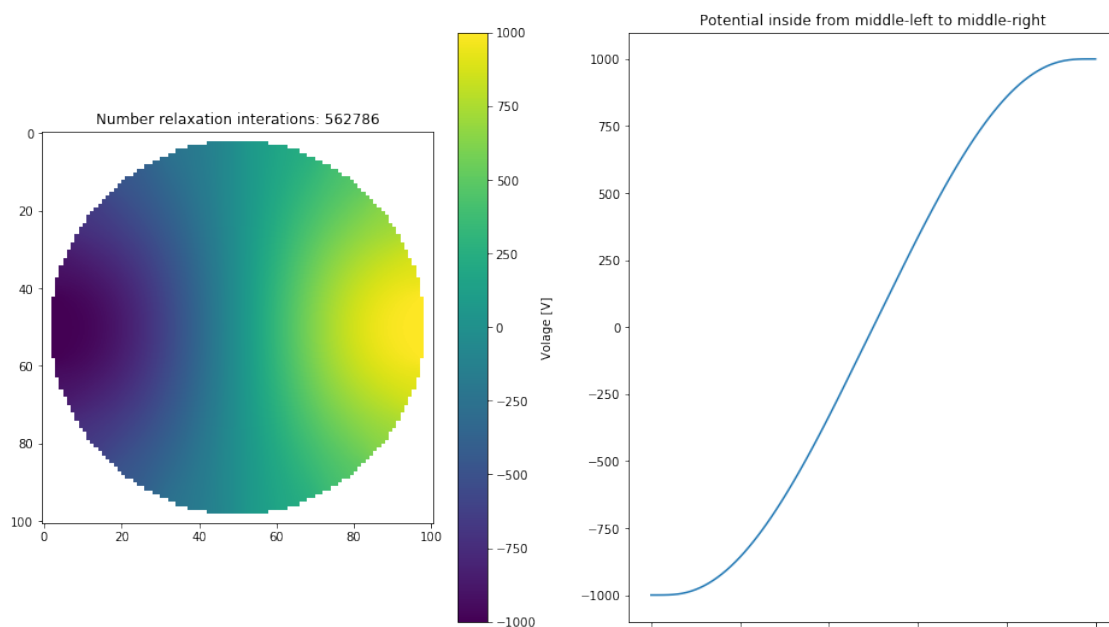
```
max_frames = 100
conv_runs = list(np.round(np.logspace(0,4,max_frames),0).astype(int)*5)
ani = FuncAnimation(fig, update, frames = list(range(max_frames)),␣
 ↪interval=10,blit=False, repeat = False)

# Set up formatting for the movie files
Writer = animation.writers['ffmpeg']
writer = Writer(fps=10, metadata=dict(artist='Me'), bitrate=1800)
ani.save('im.mp4', writer=writer)

plt.show()
```


And also save the animation as a video. Save this animation

The video can then be loaded again and played back in the notebook.

```
[66]: from IPython.display import Video
display(Video('./im.mp4'))
```

```
<IPython.core.display.Video object>
```

4.3.4 Calculate total resistance

Once the potentials inside the grain is solved with the relaxation algorithm, it is time to calculate the total resistance of the grain. Since the relaxation was solved by applying a "virtual" potential

difference ($\Delta\Phi$) to the grain, the total resistance R_{Total} could be calculated by Ohm's low: $R_{Total} = \frac{\Delta\Phi}{Current}$, where $\Delta\Phi = V_{Total}$ is the potential difference.

Therefore only the current trough the grain needs to be calculated. This can be done by calculating the total current passing the slice in the center of the grain. The index of the center slice is c. And the potential at one cube at position r_i of the center slice is U_{c_i}. Therefore the the potential difference ΔU_{c_i} across the center cube is given by:

$$\Delta U_{c_i} = U_{(c-1)_i} - U_{(c-1)_i} \tag{13}$$

The current passing the cube at in the center slice c at position r_i is therefore:

$$I_{c_i} = \frac{\Delta U_{c_i}}{R_{c_i}} = \frac{U_{(c-1)_i} - U_{(c-1)_i}}{\frac{d}{q*\mu_n*n_{c_i}*A}} = \frac{U_{(c-1)_i} - U_{(c-1)_i}}{\frac{d}{q*\mu_n*n_{c_i}*d*d}} \tag{14}$$

$$= (U_{(c-1)_i} - U_{(c-1)_i}) * n_{c_i} * q * \mu_n * d \tag{15}$$

In this equation $q * \mu_n * d$ is constant for the full grain and will be named k in the following calculations. Since the actual grain has a rotational symmetry and this model just represents the two dimensional slice, the I_{c_i} needs to be multiplied by $2 * \Pi * r_i$ to take the volume contribution of grain into account. The total current I_{Total} can therefore be calculated by:

$$I_{Total} = \sum I_{c_i} * 2 * \Pi * r_i = \sum (U_{(c-1)_i} - U_{(c-1)_i}) * n_{c_i} * 2 * \Pi * r_i * k \tag{16}$$

The total resistance R_{Total} of the grain is then defined by:

$$R_{Total} = \frac{V_{Total}}{I_{Total}} = \frac{V_{Total}}{\sum (U_{(c-1)_i} - U_{(c-1)_i}) * n_{c_i} * 2 * \Pi * r_i * k} \tag{17}$$

In the field of gas sensor research the absolute value of the resistance is not necessarily the best figure to analyze the performance. Often the ratio of the resistances offers more insights. In this case also the total resistance R_{Total} is not as interesting as the resistance change from a reference condition. As reference condition the resistance at the flatband situation R_{Total_0} is selected. This reference situation is typically reached under the expose of pure nitrogen. In the presence of nitrogen, no reactive species is interacting with the surface and the number of trapped charges a the surface can be considered equal to 0, hence no band bending is present. In rapport to the resistance at the flatband situation, where the surface potential is $0 k_B T / e$, the resistance at a specific surface potential V_S will be named $R_{Total_{V_S}}$. When calculating the ratio of both, the previously introduced constant k has no importance anymore. The only value which needs to be derived from the model is then:

$$I^*_{Total_{V_S}} = \sum (U_{(c-1)_i} - U_{(c-1)_i}) * n_{c_i} * 2 * \Pi * r_i \tag{18}$$

Once $I^*_{Total_{V_S}}$ is calculated for all surface potentials V_S, the ratio of those values with $I^*_{Total_0}$ will reveal how a specific grain changes its resistance under different conditions. The relative change in resistance can simply be calculated be dividing each $I^*_{Total_{V_S}}$ by its corresponding value $I^*_{Total_0}$ in the flat band situation. As final result of this calculation the relative change in resistance $\Delta R_{V_S} = \frac{R_{V_S}}{R_0} = \frac{I_{Total_0}}{I_{Total_{V_S}}} = \frac{I^*_{Total_0}}{I^*_{Total_{V_S}}}$ of a grain in rapport to the flat band situation will be gained.

```python
[69]: def calc_current_center(d_v, d_cond, d_mask, cube_size, grain):

          center_pos = r_to_pos(0, grain, cube_size, d_v)
          center_current = (d_v[:,center_pos+1]-d_v[:,center_pos-1])*d_cond[:
      ↪,center_pos]

          r = np.array([float(pos_to_r(xi,center_pos,grain, cube_size, d_v)) for xi in↪
      ↪range(len(center_current))])

          center_current_tot = np.sum(center_current*2*pi*r)
          return center_current_tot, center_current, r
```

4.3.5 Precalcualtion of all conditions

With all functions available the calculation of all conditions in the DataFrame can take place. Again, as demonstrated in the previous chapter, this task will be parallelized.

```python
[71]: def calc_conv_by_ser(ser):

          vinit = ser.name
          grain = create_grain_from_data(ser)
          cube_size_value = ser['cube_size']

          d_v, d_cond, d_mask = create_numerical_grain_matrix( grain,↪
      ↪ser,cube_size=cube_size_value)
          d_v = initaliz_d_v(d_v, d_mask, 1000)
          d_v, d_cond, d_mask = solve_relaxation(d_v, d_cond, d_mask, n = 10000000)

          center_current_tot, center_current, r = calc_current_center(d_v, d_cond,↪
      ↪d_mask, cube_size_value, grain)

          ser_out = ser.copy()

          ser_out.loc['current'] = center_current_tot
          ser_out['d_v'] = d_v
          ser_out['d_mask'] = d_mask
          ser_out['cube_size_value']=cube_size_value
          return ser_out
```

```python
[24]: start_t = time.time()
      from multiprocessing import Pool

      # cubesize defined by radius
      calc_dF_all['cube_size'] = calc_dF_all['R']/50

      ser_list = []
      for i, ser in calc_dF_all.iterrows():
          ser_list.append(ser)
      with Pool(12) as p:
          all_res_list = p.map(calc_conv_by_ser, ser_list)
      calc_dF_sol = pd.DataFrame(all_res_list)

      duration = time.time()-start_t
      print('Duration of the calculation: {duration/60:.2f} min.')
```

3108.441292285919

The previously described calculation of ΔR_{V_S} from $I^*_{Total_{V_S}}$ is implemented in the the next cell in Python.

```python
[160]: def get_flatband_current(dF):
           flatband_current = dF[dF['Einit_kT']==0].iloc[0]['current']
           return flatband_current

       def calc_res_change(dF):
           flatband_current = get_flatband_current(dF)
           rel_res = flatband_current/dF['current']
           dF['rel_res_change'] = rel_res
           return dF
       calc_dF_sol_with_rel_res_change = calc_dF_sol.groupby(['R', 'temp','ND']).
        ↪apply(calc_res_change)
```

```python
[161]: calc_dF_sol_with_rel_res_change.to_hdf('numerical_sol.h5','raw',␣
        ↪mode='w',complevel=9,complib='lzo')
```

/usr/lib/python3.8/site-packages/pandas/core/generic.py:2489:
PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed,key->block1_values]
[items->Index(['n', 'r', 'v', 'v_dot', 'd_v', 'd_mask'], dtype='object')]

  pytables.to_hdf(

```python
[168]: calc_dF_sol = pd.read_hdf('numerical_sol.h5','raw')
```

4.3.6 Representing final results

```
[158]: %matplotlib inline
```

```
[169]: fig, axes = subplots(len(calc_dF_sol['ND'].unique())//2,2,figsize = (16,9),␣
       ↪sharey=True, sharex=True)

       for ax_i,(ND,calc_dF_n) in enumerate(calc_dF_sol.groupby('ND')):

           axe = fig.axes[ax_i]
           axe.set_title(f'$N_D$='+f'{ND}')

           axe_up = axe.twiny()

           for R,calc_dF_grainsize in calc_dF_n.groupby('R'):

               grain = create_grain_from_data(calc_dF_grainsize.iloc[0])

               #s = calc_dF_grainsize['d_v'].iloc[0].shape[0]

               flat_band = calc_dF_grainsize[calc_dF_grainsize['Einit_kT']==0].
       ↪iloc[0]['rel_res_change']
               res = calc_dF_grainsize['rel_res_change']


               v = calc_dF_grainsize['Einit_kT']
               rel_size = grain.R/grain.material.LD

               axe.plot(v, res, 'o-', label = f'GrainRadius = {R*1e9:.
       ↪2f}nm$\equiv${rel_size:.2f}LD', linewidth=5)
               #have a additional graph in units of meV on the x axis
               axe_up.plot(v*CONST.J_to_eV(grain.material.kT)*1000, res,alpha=0)


               axe.set_yscale('log')
               axe.set_ylabel(r'$\frac{R}{R_{Flatband}}$', fontsize =15)
               axe.set_xlabel('$E_{C_{Surface}}$ [$k_BT$]', fontsize =22)
               axe.tick_params(axis='both', which='major', labelsize=15)
               axe.grid(b=True)


           #draw the Fermi-Level
           axe.axvline(-grain.material.J_to_kT(grain.material.Diff_EF_EC),␣
       ↪label='Fermi-Level')
           axe.axvline(0, color = 'r', linewidth=5, alpha=0.3, label='Flat band')

           if ax_i in [0,1]:
```
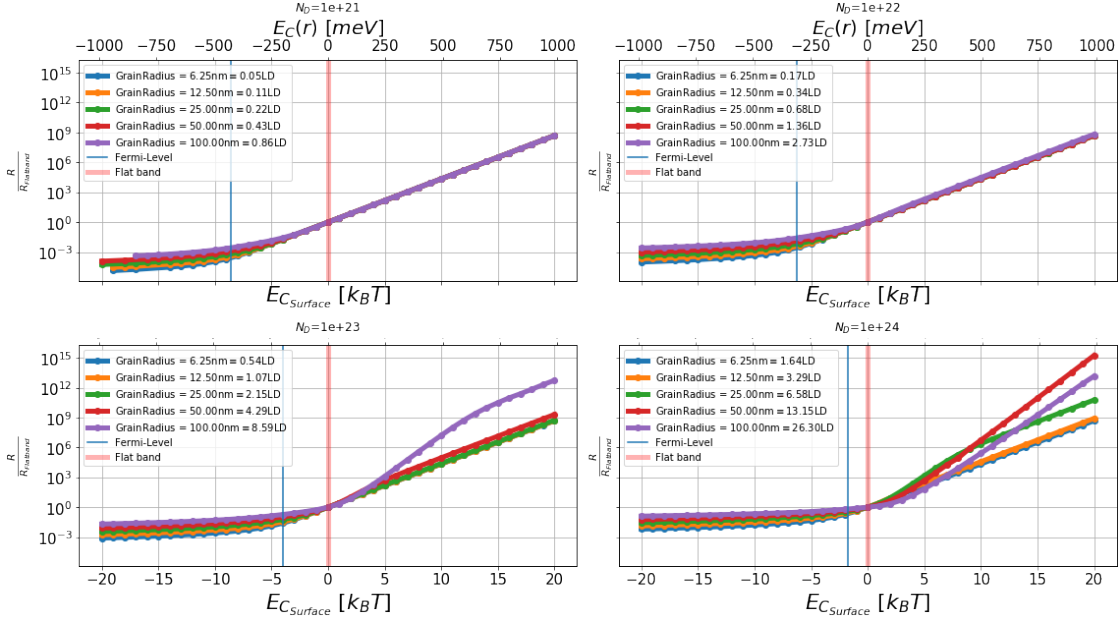
```
        axe_up.set_xlabel('$E_C(r)$ [$meV$]', fontsize =20)
        axe_up.tick_params(axis='both', which='major', labelsize=15)
    else:
        axe_up.tick_params(axis='both', which='major', labelsize=0)
        pass


    fig.tight_layout()
    axe.legend()
```


4.4   Summary

Before using the new dataset to interpret experimental results, some interesting features can already be seen in the last figure.

4.4.1   Fully depleted small grains:

The position of the conduction band at the surface $E_{C_{Surface}}$ is far away from the position of the Fermi energy E_{Fermi} and the grain radius R is small compared to the Debye length ($R < 5 * L_D R$). In this case the resistance changes in the depletion layer controlled conduction mechanism similar for all grains. This can be explained by the fact, that the grains are fully depleted and the effect of the distinct depletion layer is not preset. For those small grains, the effect of the different grain size has an impact on the sensing properties regarding only when the conduction band bending approaches the Fermi level energy.

4.4.2 Large grains:

For large grains ($R < 5 * L_D R$), which are not fully depleted, the effect of a high resistive surface layer add up with the reduction of the area of a low resistive bulk. Since large grains are not fully depleted, the low resistive area in the center of the grain get smaller with increasing surface potential. Those two effects add up and the total resistance changes. The small grains, which are fully depleted this effect is not present. Once the larger grains get full depleted, the volume effect of the surface layer is not present anymore and the change in resistance is comparable to the smaller grains.

4.4.3 Flat band situation:

Additionally it can be noticed, that the flat band situation may not be the point, where the conduction starts to change. Rather the proximity to the Fermi level is responsible for altering the conduction mechanism.

4.5 Conclusion

Real experimental data from simultaneous work function and resistance measurements can be fitted to this dataset. Since the detection of the grain size is rather simple, the fitting of the corresponding plots will reveal the corresponding Debye length and doping level N_D. As additional feature for further understanding the surface reactions, the number of involved trapped charges at the surface can be extracted from the numerical model. Non-PDF readers, could use this link to guide them to the next notebook.

5 Bibliography section

References

[VGO+20] VIRTANEN, Pauli ; GOMMERS, Ralf ; OLIPHANT, Travis E. ; HABERLAND, Matt ; REDDY, Tyler ; COURNAPEAU, David ; BUROVSKI, Evgeni ; PETERSON, Pearu ; WECKESSER, Warren ; BRIGHT, Jonathan ; WALT, Stéfan J. ; BRETT, Matthew ; WILSON, Joshua ; MILLMAN, K J. ; MAYOROV, Nikolay ; NELSON, Andrew R J. ; JONES, Eric ; KERN, Robert ; LARSON, Eric ; CAREY, C J. ; POLAT, İlhan ; FENG, Yu ; MOORE, Eric W. ; VANDERPLAS, Jake ; LAXALDE, Denis ; PERKTOLD, Josef ; CIMRMAN, Robert ; HENRIKSEN, Ian ; QUINTERO, E A. ; HARRIS, Charles R. ; ARCHIBALD, Anne M. ; RIBEIRO, Antônio H ; PEDREGOSA, Fabian ; MULBREGT, Paul van ; SCIPY 1.0 CONTRIBUTORS: SciPy 1.0: fundamental algorithms for scientific computing in Python. In: *Nature methods* (2020). http://dx.doi.org/10.1038/s41592-019-0686-2. – DOI 10.1038/s41592–019–0686–2. – ISSN 1548–7105

This page is intentionally left blank.

Comparing numerical results with experimental data
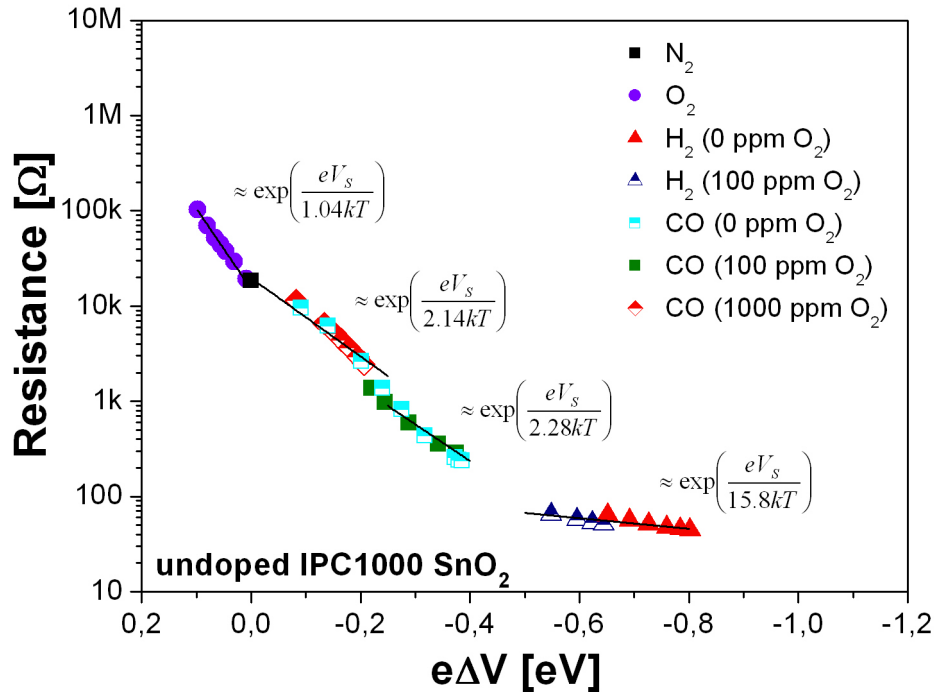
February 19, 2020

Contents

1 Abstract

Experimental data from simultaneous work function and resistance measurements will be compared with the results from the numerical calculations. Results from an SnO_2 gas sensor measured at 300°C will be used to demonstrate, how numerical data can be used to gain more insights about the measured material. The chosen dataset the the graphical representation originates from the Phd. thesis of Julia Rebholz: [?].

The data was generated by exposing the sensor to various gas compositions of H_2, CO, O_2 and N_2. The surface potential changes ΔV resulting from the different gas atmosphere have been obtained with the Kelvin probe technique. Simultaneously the corresponding resistance was measured. The data point point at $0e\Delta V$ corresponds to the situation in nitrogen.

These experimental data points will be compared with the results obtained from the numerical model.

The plot shows Resistance [Ω] versus eΔV [eV] for undoped IPC1000 SnO$_2$, with the following data series in the legend:

- N$_2$
- O$_2$
- H$_2$ (0 ppm O$_2$)
- H$_2$ (100 ppm O$_2$)
- CO (0 ppm O$_2$)
- CO (100 ppm O$_2$)
- CO (1000 ppm O$_2$)

$$\approx \exp\left(\frac{eV_s}{1.04kT}\right)$$

$$\approx \exp\left(\frac{eV_s}{2.14kT}\right)$$

$$\approx \exp\left(\frac{eV_s}{2.28kT}\right)$$

$$\approx \exp\left(\frac{eV_s}{15.8kT}\right)$$

2 Fitting experimental data to numerical results

2.1 Importing experimental and numerical data

The data is saved in an excel file, which will be loaded by using the tools provided by `pandas`.

```
[43]:  #Setting up the env.
       from part2 import *
       import pandas as pd
       %pylab inline

       #importing the data
       calc_dF = pd.read_hdf('numerical_sol.h5','raw')
       dF_1000 = pd.read_excel('Kelvin_Data.xlsx', sheet_name='ipc1000').
        ↪sort_values(by='dV')

       #instead of unsing the row number
       #each row has the value of dV as index
       dF_1000.index = dF_1000['dV']
```

Populating the interactive namespace from numpy and matplotlib

2.2 Representing the raw data

```python
[47]: def format_axe(axe, ylabel = None, set_ylim=False):
          labelsize = 30
          if set_ylim:
              axe.set_ylim((1e-4,1e3))
          axe.set_yscale('log')
          axe.set_xlim((0.2,-1.2))
          if ylabel:
              axe.set_ylabel(ylabel, fontsize = labelsize)
          else:
              axe.set_ylabel(r'$\frac{R_{V_S}}{R_{(V_S=0)}}$', fontsize = labelsize)
          axe.set_xlabel('$qV_S$ $[eV]$', fontsize = labelsize)
          axe.tick_params(axis='both', which='both', labelsize=labelsize)


      fig, axe = subplots(figsize=(16,10))

      sens, dF = 'IPC 1000',dF_1000

      v_exp = dF['dV']

      res_exp = dF['res']
      axe.set_title(sens, fontsize = 30)
      axe.scatter(v_exp,res_exp, s=100)
      format_axe(axe,ylabel='$Resistance$ [$\Omega$]')
      axe.set_ylim(res_exp.min()/2,res_exp.max()*2);
      axe.set_ylim(10,10e9);
      axe.grid()
```
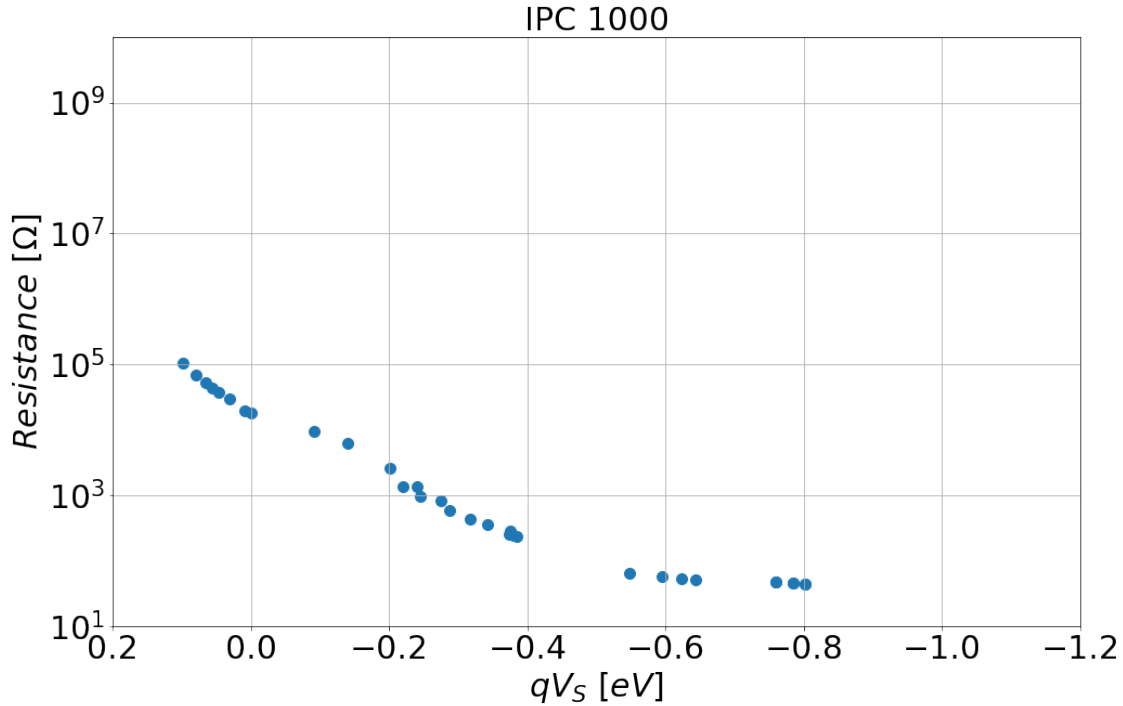
IPC 1000

2.3 From R_{V_S} to ΔR_{V_S}

In the experimental dataset the value at $0qV_s$ represent the data points measured under nitrogen. Therefore $\Delta R_{V_S} = \frac{R_{V_S}}{R_0}$ is calculated by :

- First derive the resistance under nitrogen R_0
- Second devide all resitance values by this value

```
[48]: from scipy.optimize import curve_fit
      from scipy.interpolate import interp1d


      fig, axe = subplots(1, figsize=(16,10))

      #get the value of the flatband (if needed)
      #by interpolation
      interp_res = interp1d(v_exp,res_exp)
      res_flatband = interp_res(0)

      #calcualte the rel. res change
      rel_res_exp = dF['res']/res_flatband


      #represent it
```
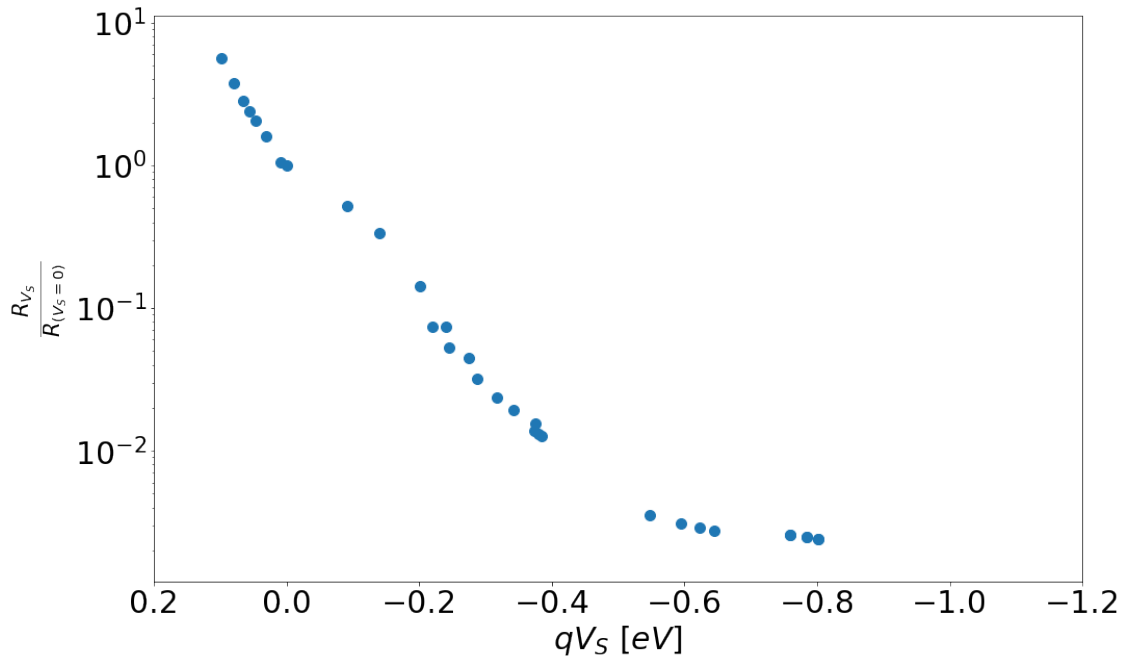
```
format_axe(axe)
axe.scatter(dF['dV'],rel_res_exp, s=100)
axe.set_ylim(rel_res_exp.min()/2,
              rel_res_exp.max()*2);
```


2.4 Interpolating the numerical values

In the previous section, the numerical solution for multiple start parameters have been calculated. Nevertheless most probably the calculated dataset will not hold exactly the same values gathered from the experiment. To obtain the numerical value for a specific experimental value of qV_s a interpolation between of the existing numerical values will be used again. For all different numerical grains ΔR_{V_S} will be calculated for all the experimental values of qV_s. Once this is done, the different between the numerical model and the exp. data can be calculated and evaluated.

```
[49]: #The dataframe to hold the different
      #of the exp. values to the numerical ones
      #Will be used to find the best fitting num. solution
      num_data_at_exp_pos_dF = pd.DataFrame(index = v_exp)

      #group the num. data by its paramters (T, R and ND)
      data_by_grain = calc_dF.groupby(['temp','R','ND'])

      for (T, R,ND), calc_dF_grain in data_by_grain:

          num_data_at_exp_pos_dF[(T, R,ND)] = None
```

```
        grain = create_grain_from_data(calc_dF_grain)

        flat_band_data = calc_dF_grain[calc_dF_grain['Einit_kT']==0].iloc[0]

        rel_res_num = calc_dF_grain['rel_res_change']

        #express the surace potential in eV
        #to be comparable with the exp. data
        v_num = calc_dF_grain['Einit_kT']*CONST.J_to_eV(grain.material.kT)

        #use interpolation to get the values for the positions
        #of the experiment data points
        interp_rs_num = interp1d(v_num, rel_res_num,bounds_error=False)
        interp_v_num = interp1d(rel_res_num,v_num, bounds_error=False)

        #caculate the numerical value of rel. res at the position
        # of V from the experiment
        res_num_at_exp_pos = interp_rs_num(v_exp)

        #save those values in the new DataFrame
        num_data_at_exp_pos_dF.loc[:,(T, R,ND)] = res_num_at_exp_pos
```

2.5 Calculating the fit error

num_data_at_exp_pos_dF contains now the values of ΔR_{V_S} at the positions qV_S. From these values the relative error needs to be calculated. The following formula is used to derive the error:

$$\epsilon_{V_S} = \left(\frac{R_{numerical}(qV_s) - R_{experiment}(qV_s)}{R_{experiment}(qV_s)} \right)^2 \tag{1}$$

The sum of all ϵ_{V_S} is the total error of the fit. The numerical model with the lowest value of $\sum \epsilon_{V_S}$ is the model which fits best to the experimental data. The average grain diameter of the material "IPC100" is known to be in average 110nm (Nanoparticle engineering for gas sensor optimization: Improved sol-gel fabricated nanocrystalline SnO2 thick film gas sensor for NO2 detection by calcination, catalytic metal introduction and grinding treatments, 1999). The dataset we created in the previous section includes models for grains with radii of 50nm and 100nm. Therefore we can narrow the fit algorithm down, to take only models with a radius of 50nm and 100nm in account

```
[50]:  abs_error = num_data_at_exp_pos_dF.subtract(rel_res_exp, axis='index')
       rel_error = abs_error.divide(rel_res_exp, axis='index')
       rel_error_square = rel_error**2
       sum_of_squares = rel_error_square.sum()

       valid_index = [i for i in sum_of_squares.index if i[1] in [50e-9,100e-9]]
```

```
sum_of_squares_grainsize = sum_of_squares.loc[valid_index].sort_values()

grain_min_error_tuple = sum_of_squares_grainsize.idxmin()
display(pd.DataFrame({'error':sum_of_squares_grainsize}))
grain_min_error_tuple
```

	error
(300.0, 5e-08, 1e+22)	6.82
(300.0, 1e-07, 1e+22)	7.67
(300.0, 1e-07, 1e+21)	14.36
(300.0, 5e-08, 1e+21)	18.77
(300.0, 5e-08, 1e+23)	175.20
(300.0, 1e-07, 1e+23)	1256.74
(300.0, 5e-08, 1e+24)	9905.88
(300.0, 1e-07, 1e+24)	39709.78

[50]: (300.0, 5e-08, 1e+22)

2.6   Representation of the fit

Finally the best fit results can be represented graphically.

```
[54]: fig, axe = subplots(figsize = (16,10))
#for grain_tuple in num_data_at_exp_pos_dF.keys():
for grain_tuple in sum_of_squares.index:
    if grain_tuple == grain_min_error_tuple:
        linestyle = '*-'
        linewidth = 5
        alpha = 0.5
        label = 'Best Fit'
    elif grain_tuple in sum_of_squares_grainsize.index[0:2]:
        linestyle = '-o'
        linewidth = 5
        alpha = 0.3
        label = 'Two best fits'
    else:
        linestyle = '-.'
        linewidth = 1
        alpha = 0.3
        label = 'Other solution'




    axe.plot(num_data_at_exp_pos_dF.index,
                num_data_at_exp_pos_dF[grain_tuple],
```

```
              linestyle, linewidth=linewidth, alpha = alpha,
              label =label)


    last_x  = num_data_at_exp_pos_dF.index[0]
    last_y  = num_data_at_exp_pos_dF.iloc[0][grain_tuple]

    if grain_tuple in sum_of_squares_grainsize.index[0:2]:
        axe.text(last_x-0.05,last_y,
            f'''R:{grain_tuple[1]*1e9:.2f}nm\n$N_D$:{grain_tuple[2]:.2}''')
format_axe(axe)

axe.scatter(rel_res_exp.index,
            rel_res_exp,
            s=100,
            label = 'Exp. data'
            )

axe.set_ylim(rel_res_exp.min()/10,
              rel_res_exp.max()*2);


l = {h[1]:h[0] for h in zip(*axe.get_legend_handles_labels())}.keys()
h = {h[1]:h[0] for h in zip(*axe.get_legend_handles_labels())}.values()
axe.legend(h,l,loc=1, fontsize = 22)
```
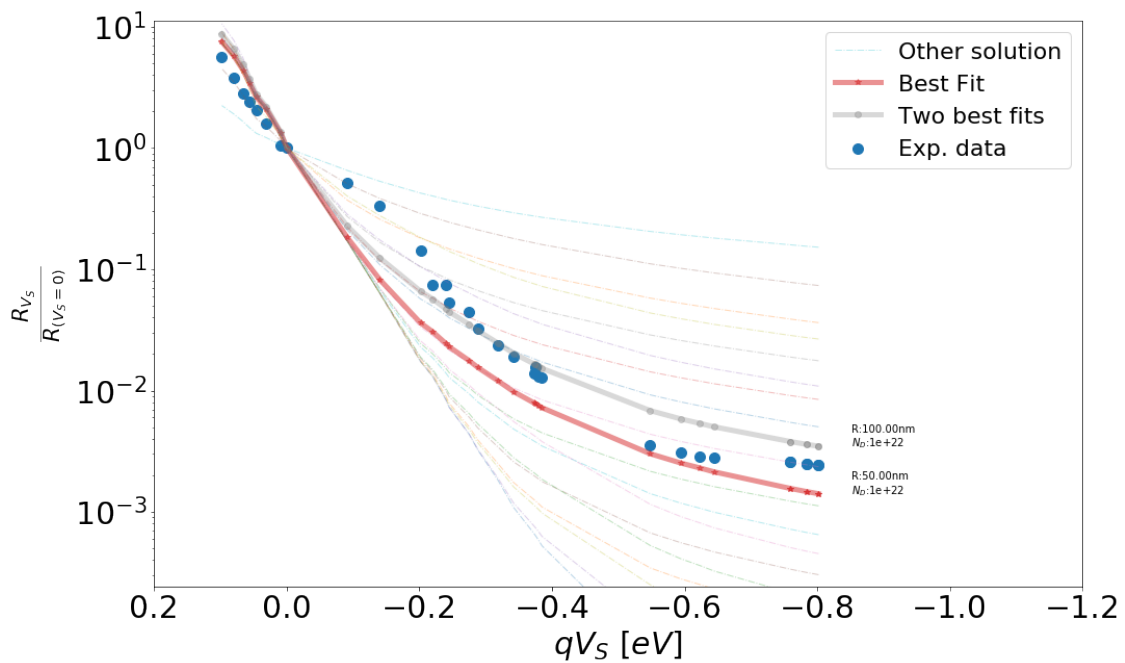
[54]: <matplotlib.legend.Legend at 0x7fb19c3de490>

3 Conclusion

We can see, that both fits do not fit perfectly to the experimental data. One obvious reason is the curse screening of the grain size and and N_D. A second iteration of creating models with radii between 50nm and 100nm might find a better fit. Also a finer screening of N_D will turn out to be helpful for a better result. On the other side the fitting shows, that the experimental data fits will to a grain with approximately 50nm radius and a defect concentration of around $N_D = 1 * 10^{22} 1/m$ at 300°C.

4 Bibliography section

References

[Reb16] REBHOLZ, Julia M.: *Influence of Conduction Mechanism Changes and Related Effects on the Sensing Performance of Metal Oxide Based Gas Sensors*. Shaker Verlag, 2016. – 127 S. – ISBN 978–3–8440–4832–2