A parametrized numerical model to simulate the semiconductor influence of thick film metal oxide gas sensors

March 20, 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
aus Mainz am Rhein

Tübingen
2020

Tag der mündlichen Prüfung: xx.xx.2020

Dekan: Prof. Dr. Wolfgang Rosenstiel

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

Jupyter for scientific research

March 20, 2020

Contents

1 Introduction

1.1 Motivation

In the beginning of my academical career I was aiming at an educational degree in math and physics. While studying at the University of Tübingen I also worked as a research assistant at the Institute of Physical Chemistry. In the course of my studies, I had the opportunity to spend 4 months in a school to gain first experiences in teaching physics and math to students between the ages 12 and 20. Even though it was a great experience and I enjoyed bringing new ideas and concepts to the students, I also felt a strong urge to continue learning and being able to 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 at school. Long story short, after some years I started my doctoral thesis in the "Institute of Theoretical and Physical Chemistry" at the University of Tübingen. I directly got the possibility to work in a project with an industrial partner. The focus was on building 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 to be solved. One of the biggest challenges was the continuous increase of experimental data in ever shorter intervals. With the increasing industrialization and automatization of sensor production and testing, the quantity of high quality data increased dramatically.

With the increased global interest in "data science", I was not the only one facing problems with the increasing amount of data needed to be processed and analyzed. At 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 traditional way of creating well defined software algorithms for specific tasks was not adequate.

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 tools efficiently. So at this point of my career with my very basic knowledge in the area of programming and the need to solve urgent tasks, I was looking for efficient solutions with a steep learning curve.

Until then my standard procedure 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. In parallel to this, 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 "I(nteractive)Python notebook ecosystem" was making the use of Python for in an scientific work flow very attractive.

As mentioned in this article related to IPython: [Dav13]

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

"Python at Cern" describes for instance the use of Python at the Large Hadron Collider (LHC) at

Cern. As the article mentions, “The interesting bits of data have to be stored, analyzed, shared and published”. “The ease of use and a very low learning curve makes Python a perfect programming language for many physicists and other people without the computer science background”. Python has been used at the LHC now for more than 20 years. Now in 2020 the general technological improvements allows also smaller institutions to work in a comparable way as for instance at the LHC. High computational power combined with large data storage facilities do not require a large financial investment anymore. As a consequence, in many research fields this “data driven scientific research” gains more and more interest. Nevertheless this is a slow process, and this work intends to help improving it.

Already the initial attempts of using Python, the large potential also for my working field was clear. By learning Python I got efficient tools for calculations, analysis and representations. Additionally the new tools have been build focusing on an simple way of reporting result while including the way they have been gained. The environment around the so called “IPython notebooks” was the ideal piece, which I experienced as a missing block in the scientific work I was doing. In 2015 toolset around IPython notebooks have been unified along with other programming languages in a project called Jupyter. In the following part of this work, IPython and Jupyter will be treaded as synonyms.

Besides my work for our industrial partner I also conducted fundamental research about semiconducting metaloxide gas sensors. Based on great research 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 and documented 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 during 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. Thus, I decided to try and combine representation, description and algorithm in one unified way.

For this I used 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 an introduction to this excellent toolbox.

This work will allow more people to gain insight into sensing properties of semiconducting 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 unfounded. While lecturing at the University of Tübingen the course, there

was never enough time to use Python as a supporting programming language in e.g., the introductory course on data mining. In these cases classical tools like Excel (tm) or Origin (tm) were used to analyze example datasets for hidden facts. While most students understood the general concept of data mining, 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 not 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 creative and intelligent 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. This work is intentionally structured in a way, to encourage modifications of the original work and simplify this process. The supplied program code is based entirely on well establish standard tools available for all operation systems. This makes this work easily reproducible for others.

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. Numerical solving of differential equations and integrals
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 other commands which in essence is a philosophical question on how instructions should be formulated.

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. The `import` statement in Python is used to add functionality or features to the current programming environment. `import this` is a special "easter egg" hidden in every Python version. It adds some Zen to the work space.

[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 scientist, but the Python community consists most probably of many people who would consider the latter points as important. So did also the inventors of the IPython and Jupyter. A quick search in the world wide web will provide a detailed picture about what "IPython Notebooks" are and how Jupyter in connected with them. Here I will not try to give a general overview of this tool but rather stick to the phrase "Learning by doing". A minimal understanding of working with Python will definitely be helpful for this thesis, but is not necessarily required. I will try to explain the used features this notebooks to guide an interested reader to the point where students can:

- understand the fundamental instructions of Python
- use the basic functionality of the Notebooks
- have a understanding of SMOX based gas sensors

It is worth mentioning, that the intention of such notebooks is to merge the essential tools of scientific work flows with 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 and PDF. 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

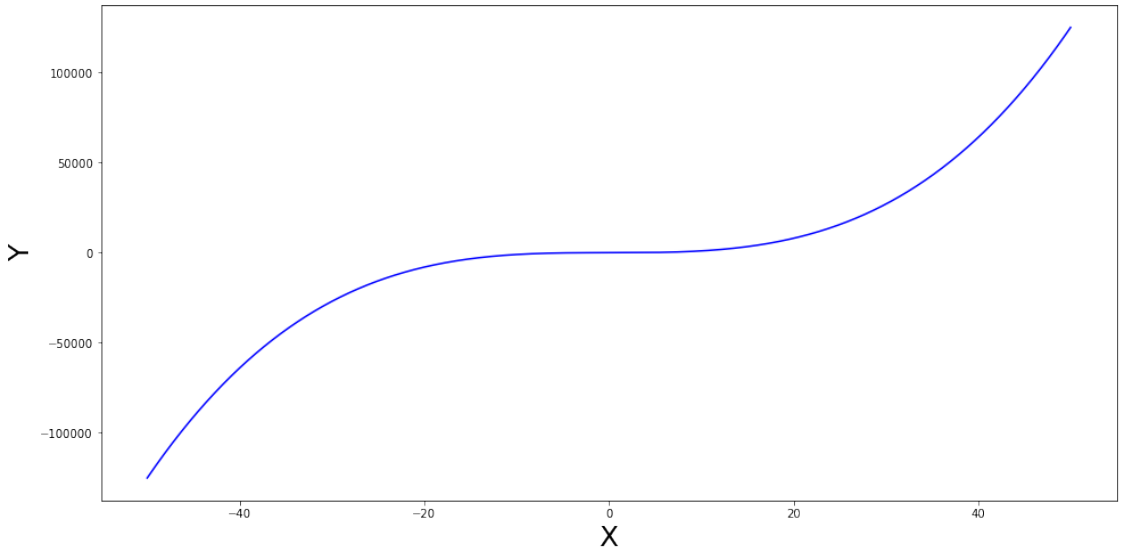
Besides the first example `import this`, here is a very basic example to demonstrate some features of the notebook.

"Simple is better than complex.": The Jupyter environment is equipped with "magic commands" which are not part of the Python programming language, but rather instructions to simplify common tasks. Magic commands always start with `%` and are followed with an instruction. I will demonstrate in this example using the `%pylab inline` instruction. This modifies the current programming space to become a lab nicely equipped for scientific work tasks. For instance, a chemistry lab is commonly equipped with a balance, a water tab and a fire extinguisher. Similarly, 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])

[22]:
```
%pylab inline

#The Python way of adding a comment is by starting a line with the '#' character

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


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

#plot it
fig = figure(figsize=(16,8))
plt.plot(xs,ys,'b');

#name axes
plt.xlabel('X', fontsize = 25)
plt.ylabel('Y', fontsize = 25);
```

```
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 take 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 will be provided. The intention is to provide the minimum amount of information needed to follow 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 gap 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.

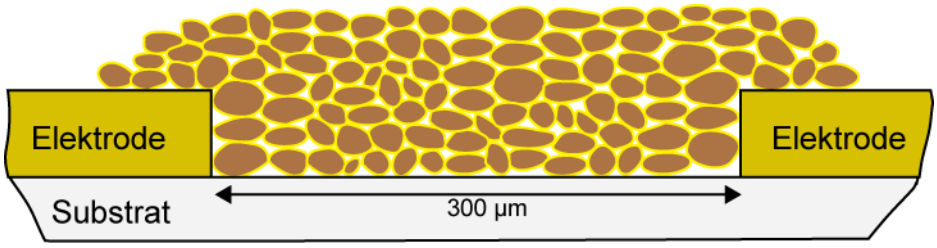
Typically, the density of additional donors N_D or acceptors N_A plays a significant role in the sensing properties. The availability of additional free charge carriers resulting from such defects depends also on the overall temperature of the sensor. Typical operation temperatures are around 300°C, 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:

$$R_{Sensor} \propto G * R_{Grain} \qquad \text{(Resistance of sensor)}$$

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 expressed 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 is 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 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}.

$$\frac{R_{Sensor_{V_S}}}{R_{Sensor_{V_S=0}}} = \frac{G * R_{Grain_{V_S}}}{G * R_{Grain_{V_S=0}}} = \Delta R_{V_S} \qquad (1)$$

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 an important parameter involved in the description of the chemical reaction at the surface.

When solving this problem numerically, some assumptions will need to be introduced. 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 Summary

Since the motivation for this "interactive" thesis should be clear now, I would like to now move to my actual research topic in the next section:

"A parametrized numerical model to simulate the semiconductor influence of thick film metal oxide gas sensors".

In the next notebook I will demonstrate how theoretical numerical calculations of chemical sensors are used to better understand experimental results.

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 interactive representation can be represented outside the Jupyter 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-version 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. A snapshot of the current folder can be downloaded here: Num_smox_sensor.zip

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".

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

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: Example 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 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, Microsoft Excel (tm), 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, Dataframes are for Pandas. Here a simple example how to create a Dataframe in analogy to the previous programming example:

[2]:
```
%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, there are infinite 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 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 why 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 in both representations.

3.2.1 Before the patch

[3]:
```
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

[4]:
```
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

[5]:
```
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

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

Solving the poisson equation for spherical grains

March 20, 2020

Contents

1 Abstract

A semiconducting 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 it 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 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 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 mechanism 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 can 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. It is assumed that adsorbed oxygen at the surface of the semiconductor lead to an interaction with the charges inside the thick film grains. By the adsorbed oxygen at the surface charge carriers are trapped at the surface and a depletion layer forms. 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. For example, it could be shown that for an SnO_2 based gas sensor 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 [BRW15].

The following figure shows the experimental results from this publication. In the very beginning of the experiment, a SnO_2 based gas sensor was exposed to pure nitrogen. It is assumed that the resistance under nitrogen corresponds to the flat band situation. It should be mentioned that this may not always be the case for all sensor materials. Nevertheless an increase of the resistance then 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 which injects additional free charge carriers in the conduction band. In the following figure we clearly see that a conduction band switch is present.

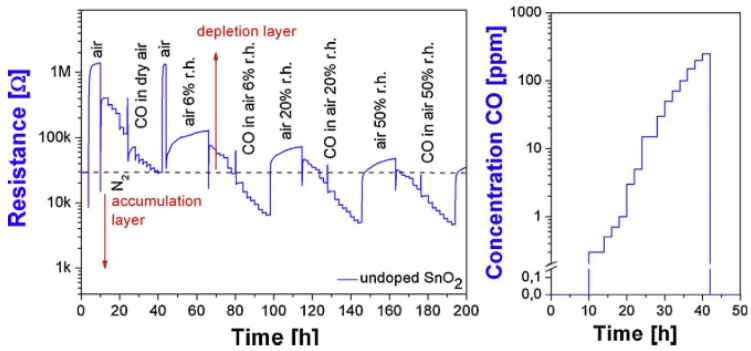


Fig. 1. Left: DC electrical resistance measurement of an undoped SnO_2 sensor (polycrystalline thick film sensing layer) during exposure to N_2, and 0–250 ppm CO in different background conditions. Right: The profile of CO exposure as a function of time, which is valid for all background conditions.

With the absence of the depletion layer also most of the commonly used 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 surface condition needs a more general descriptions, which includes a depletion and accumulation layer controlled transduction mechanism.

The processes inside a semiconductor can be described by a set of differential equations available in literature. As mentioned, common simplifications are not valid for a generalize solution of the equations. Additionally finding an analytically solution exceeded by far my intellectual capabilities. Therefore, a numerical solution for this problem was developed.

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 which is certainly a drawback of this numerical 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 properties, the resulting dataset can then again be used to gain insights about the fundamental relations of the parameters. Also the comparison of the numerical results with experimental data might reveal 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 those different parts and how they have been simplified, solved and combined again will be described. In the 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 requires 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] it is known that the grain size and the number charges trapped at the surface have strong impact on the potential and charge distribution

inside the grain. For large grains, compared to their Debye length (L_D), additional charges trapped at the surface may leave the bulk region unaffected. In contrast to the large grains, relative small grains may be affected through the whole grain by surface charges. The simulation of this effect will be the major goal of this chapter and a graphical representation this is shown at the end.

With those results the influence of the charge transfer at the surface on the resulting charge density inside the grain can be described. Hence the free charge carrier concentration and a position dependent resistivity inside the grain can be calculated.

When investigating the total resistance of the SMOX material, the pathway of the current through the material plays a major role. With the results presented in this chapter the anisotropic resistivity distribution inside the material can be used to derive the total resistance of the grain.

3.2 Semiconductor properties of the SMOX grains

The advantages of industrialized production techniques are inline with general advantages of the SMOX-based sensor technology. 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. Typical production techniques result in almost spherical grains with a narrow size distribution and a high surface to volume ration beneficial for high reactive surface area. When deployed as a thick film the high number of grain-grain contacts have an additional 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 desired increase in performance is not explained. Since the aim is to gain a fundamental understanding of the shape influence while staying close to an 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 this work are 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 variety in parameters like diameter, doping level, band gap and material composition are not favorable for understanding their influence on the overall sensing properties by comparing the numerical model with experimental data.

3.4 Poisson's equation

Surface reactions induce a charge transfer between the bulk and the surface of the grain. 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. 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 are reasonable for the investigated cases, but do not allow 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:

$$\nabla\phi = -\frac{\rho}{\epsilon\epsilon_0} \tag{Poisson}$$

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

It is assumed 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 at 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 for SnO_2. Since all surface sites will be accessible by the gas, the solution of equation: (Poisson) should have a rotational symmetry.

In case of an rotational symmetric shape of a SMOX grain, equation (Poisson) can be expressed as a ordinary differential equation of only the radius:

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

3.5 Charge density

As mentioned, the scope of this work is to investigate the transduction mechanism. Specially including the phenomena of a switch from a depletion- to accumulation layer controlled transduction. The previously shown measurement demonstrated the conduction mechanism switch 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 an effective way to describe and simplify the Poisson equation. In the case of an 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 conduction band E_C and the Fermi level energy E_{Fermi} is high enough:

$$E_C - E_{Fermi} \gg 3k_BT \tag{1}$$

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

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

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 it will be demonstrated when using the Boltzmann approximation is a good approximation of the Fermi-Dirac function and when not.

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

The Fermi-Dirac equation is transformed to a suitable format, which will reflect the the occupation probability at energies relative to the initial conduction band position E_C:

Fermi-Dirac:

$$f(E) = \frac{1}{exp(\frac{E-E_{Fermi}}{k_BT})+1} = \frac{1}{exp(\frac{E-E_C+E_C-E_{Fermi}}{k_BT})+1} = \frac{1}{exp(\frac{E_C-E_{Fermi}}{k_BT}) * exp(\frac{E-E_C}{k_BT})+1} \qquad \text{(Fermi)}$$

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:

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

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:

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

Typically this equation is simplified to the following form:

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

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. Such an analytical equation is useful for further theoretical calculations but is not necessary for our numerical approach of this thesis. It is worth-wise to mention that this simplification is only valid if the Boltzmann approximation is also valid.

Equation $n(E_C)$ is solved numerically and compared with the results obtained with the common approximations. $m_e^* = 0.3 m_e$ 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 shielding 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 the latter equation it is possible to calculate how the surface potential is electrically screened 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:

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

In this case the Debye length is the distance required to screen a potential V until its value reaches $\frac{V}{e}$. Even if the Boltzmann approximation may not be valid in all cases, the Debye length can still be used as material specific property. However it will most likely not represent the characteristic screening distance anymore, which is required to reduce the surface potential by the factor $\frac{1}{e}$.

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 $\frac{k_B T}{e}$:

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

- 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. It will be shown in the next part of this thesis that the numerical calculation is also possible with the initial spherical Poisson equation (Poisson spherical (V)). However without using the unitless representation of the Poisson equation 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 necessary calculations would need to be redone. The benefit of the latter derived unitless 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

A direct advantage of the numerical approach 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:

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

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).

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

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

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. 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.

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

3.7 Constants

For the numerical calculations some physical constants are required. 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` following 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 the course of this thesis. Additionally some useful functions, e.g. the conversion from Celsius to Kelvin and vise versa are added. Such functions will be of major importance when transferring gained knowledge in the 'semiconductor regime' (Kelvin is the useful scale here) to application relevant conditions (where Celsius is the common temperature scale).

Before implementing a custom class for the constants, a already available class of constants is imported from SciPy package. SciPy is library, which includes fundamental algorithms for scientific computing [VGO+20] (SciPy 1.0: fundamental algorithms for scientific computing in Python). This package is a very powerful and well established set of tools, which will be used for most calculations in this work. With the line: `from scipy import constants as scipyConst`, the class constants is imported from the scipy package. The `as` statement is used, to give this imported class a unique name which includes the SciPy package in its name.

[2]:
```
from scipy import constants as scipyConst
class Constant:
    def __init__(self):
```

```
        self.K0 = scipyConst.convert_temperature(0,'C','K')
        self.kB=scipyConst.k
        self.EPSILON_0 = scipyConst.epsilon_0
        self.E_CHARGE = scipyConst.elementary_charge
        self.h = scipyConst.h
        self.MASS_E = scipyConst.electron_mass
        self.NA= scipyConst.N_A
        self.VOL_mol = 22.4
        self.mole_per_l = self.NA/self.VOL_mol

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

    def C_to_K(self, C):
        return scipyConst.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, a simplified numerical representation of the investigated semiconducting material is implemented. 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.

Starting again 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`

[1]:
```
%pylab inline
```

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

3.8.1 Solving integrals numerically

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

From my experience, many students and researches are excellently trained to define the set of equations describing their model mathematically. Second, also the evaluation of the individual

equations with multiple variables imposes no problems. 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 only to specific boundary conditions.

As mentioned solving integrals numerically is fairly easy, even if one might not feel very comfortable with counting squares. But 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 function to be integrated can be evaluated for each point between the boundaries of the integral, not much stands 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:

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

The `quad` function: The `quad` function from the `scipy.integrate` package will be used to integrate the given function. `quad` needs as comma separated inputs the

1. function to integrate
2. the lower integration boundary
3. the upper integration boundary

From following description comes from the documentation of `quad`:

> 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 demonstrated 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 for most of them is how to technically implement the numerical solving algorithm in Python/code. So here it comes:

[4]:
```
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 value. 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 probability distribution" function:

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

By definition a probability distribution is normalized and the integral from $-\infty$ to ∞ is 1:

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

Finding the analytical solution of this integral is already a rather advanced task, but still solvable. In the following cell the algorithm to solve this integral numerically is demonstrated.

[11]:
```
def f(x):
    return 1/(2*pi)**0.5*e**(-(x**2/2))

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'`.

Inside a formatted string any variable name surrounded by curly parenthesis is replaced with its 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, feel 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 derived in his thesis [Hue11] a way to calculate the energetic position of the conduction band E_C relative to the Fermi-Energy level E_F from the temperature T, concentration of defects in the bulk N_D and the effective mass of electrons in the semiconductor m_e^*:

$$\Delta E_{CF} = E_C - E_F \tag{13}$$

Since ΔE_{CF} 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 on special assumption only valid for SnO_2. The definition from [Hue11] is translated into a Python algorithm and used as a 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.

[12]:
```
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
    """

    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 Michi's Thesis page 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 = 1e22
mass_e_eff_factor =0.3

EDCF_eV = calc_EDCF_by_temp(T_C, ND, mass_e_eff_factor)
print(f'Distance between conduction band and fermi level: {EDCF_eV:.2f}eV')
```

```
Distance between conduction band and fermi level: 0.31eV
```

Define the smox-material class: With the helper functions a new class describing the actual SMOX material can be defined. Besides combining the relevant parameters of the semiconductor, the new `class Material` should also hold a method to calculate the concentration of charge carries in the conduction band.

[13]:
```
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
from patch_pandas_latex import *


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)
        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:

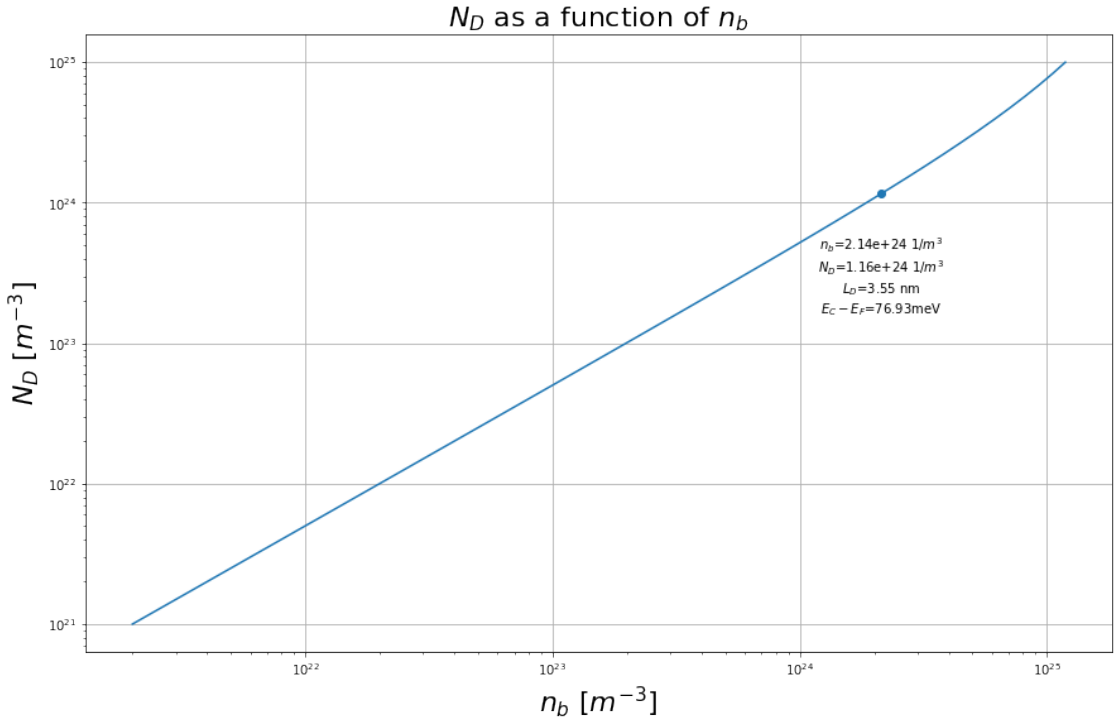
> "It can save time when an expensive or I/O bound function is periodically called with the same arguments"

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 in bytes.

Relation between n_b and N_D In the thesis of Julia Rebolz [Reb16] 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] have been calculated. Those values will be used to check the results calculated from the `Material` class. The numerical calculation 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.

[14]:
```
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)

# 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' )



#Checking one value from the Thesis of Julia Rebholz
#is done here
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');
```



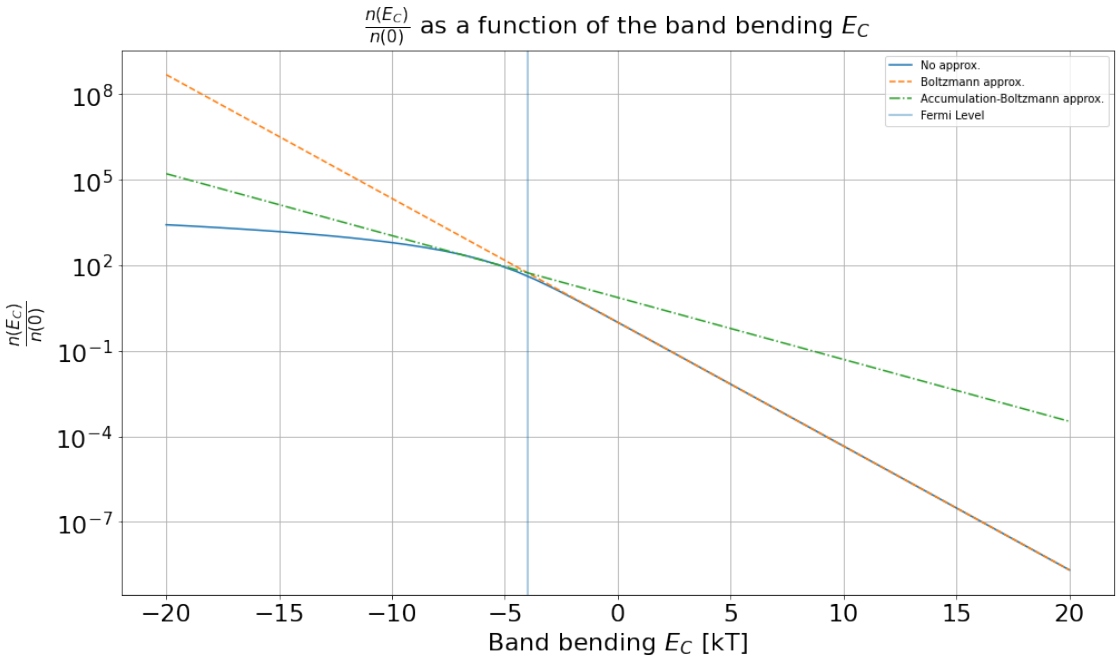
Free charge carrier conc. using the Boltzmann approximation Besides the full numerical solution, also the solutions derived from the Boltzmann approximations need to be defined. This will allow to compare the different solutions and check the validity of the different approximations.

[15]:
```
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 in the conduction band as a function of E_C in units of kT. $E_C = 0$ represents the position of the conduction band in a unaffected bulk.

[24]:
```
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 defined by the `Material` class. For one grain, the Poisson equation with spherical symmetry is solved for the positions inside the grain. Deriving from the properties of a material the implications for an actual three dimensional 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 volume play an important role. Very small grains may have relatively high concentration of

surface sites but lack the electrons needed for the reaction at the surface. In such a case 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 grain size can only be analyzed, if the transition from a material to an actual grain is performed.

To solve the Poisson equation, the solver will need to be feed 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 "**i**nital **v**alue **p**roblem" is **solve**d with the scipy tool solve_ivp.

[17]:
```
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
        #and directly discard non physical solutions
        #Also crossing the flatand is not realistic and should be stopped

        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
        return y[1]
    hit_ground.terminal = True


    #see the docstring why I chose the metohd BDF
    data = solve_ivp(fun = self.deriv_E_E_dot,
                     t_span = (r_LD[-1],r_LD[0]),
                     y0 = [E_init_kT,E_dot_init_kt],
                     t_eval=r_LD[::-1],
                     events=hit_ground,
                     method = 'BDF')

    #since we start from the surface and iterate towards
    #the center, the results have to be revered to be again
    #sorted by increasing values of r*

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

    #since the evaluation might stop  earlier
    #the missing elements need to be fileed up again
    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: Now with the `Grain` class defined, actual numerical gains can be initialized . Here an example on how to use the `Grain` class is provided.

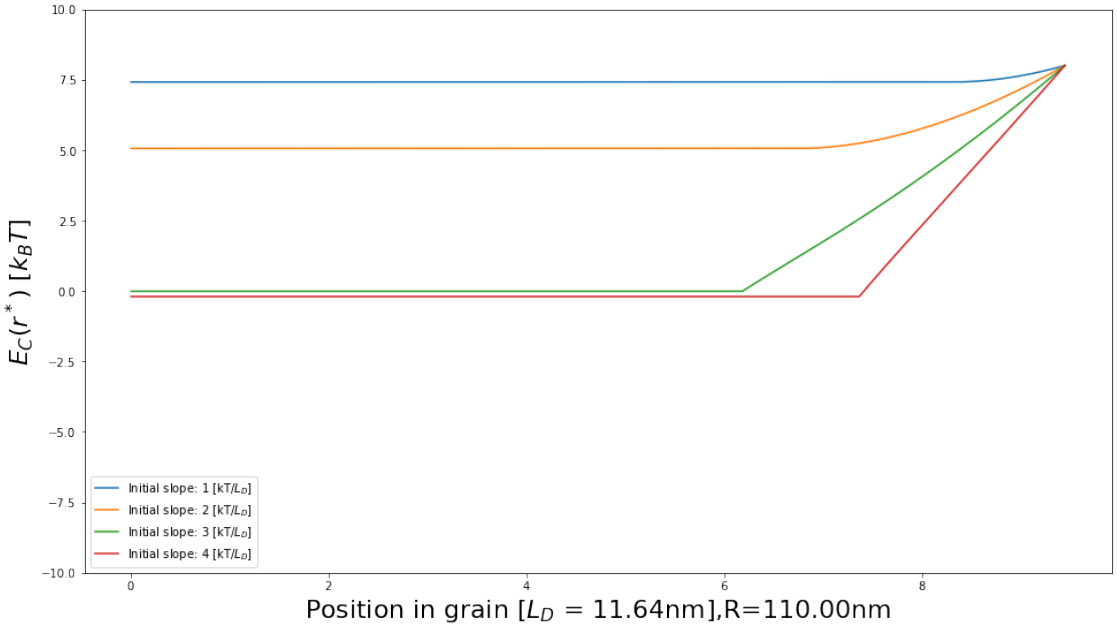
[29]:
```
#define a grain with a specific material
def create_grain(grainsize, T_C, ND):
    mass_e_eff_factor =0.3
    material = Material(T_C,ND)
    grain = Grain(grainsize_radius=grainsize,material=material)
    return grain

#Check the influence of LD

g = create_grain(50e-9, T_C=300, ND=9e24*4)
print(f'This grain has a size of {g.R/g.material.LD:.2f} LD')
```

```
This grain has a size of 45.48 LD
```

In the next example a grain is defined with a fixed surface potential. Based on this fixed potential 3 different "guesses" of the initial slope are used to simulate the shape of the conduction band inside the grain.

[30]:
```
#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()
```



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

Unfortunately this value is in most cases unknown. To solve this problem the previously derived function (Second derivative) can be used:

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

This equation is transformed into following form:

$$\int_0^R \frac{dV^*}{dr^{*2}} dr^* = \left[\frac{dV^*}{dr^*}\right]_0^R = \left.\frac{dV^*}{dr^*}\right|_R = \tag{14}$$

$$\left.\frac{dV^*}{dr^*}\right|_{Surface} = \int_0^R 1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*} dr^* \qquad \text{(Surface slope)}$$

With this relation each solution can be verified. For a pair of valid starting conditions the resulting solution should also be a valid solution for equation: (Surface slope).

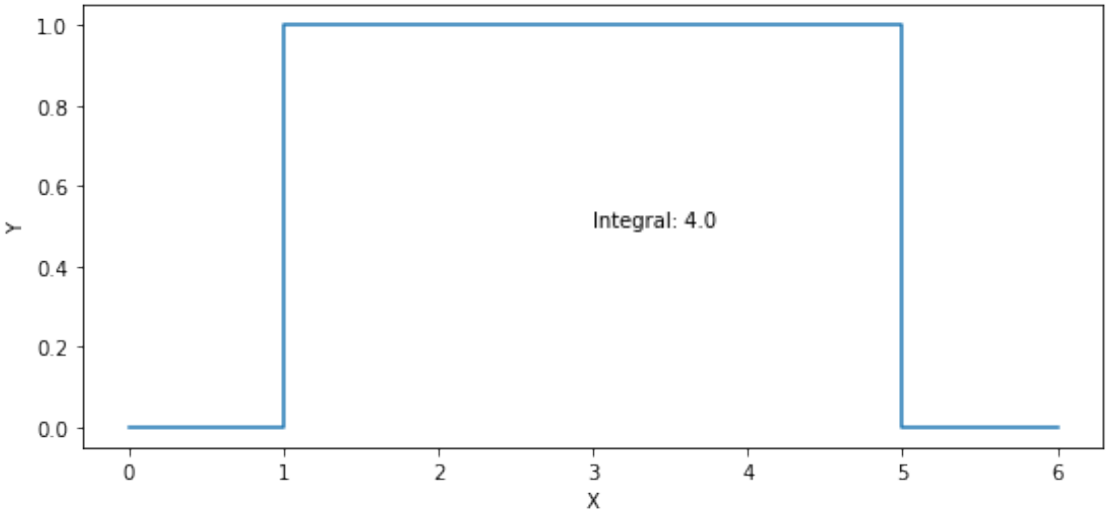
The right side of (Surface slope) will again be evaluated numerically. The expression $n^*(V^*)$ can be calculated for each V^* with the function defined in `class material`. From the `ode_solver` the values of $\frac{dV^*}{dr^*}$ are known inside the grain at the positions of r^*. Since all the elements of the integral are known, the numerical evaluation is not difficult.

However the elements of the integral in this case are not functions anymore and can't be evaluated individually for each point. From the iterative solving algorithm of the `ode_solver` only lists of values are available. In such a case 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:

> 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`

[20]:
```
x = [0,1,1,2,3,4,5,5,6]
y = [0,0,1,1,1,1,1,0,0]
fig, axe = subplots(figsize = (9,4))
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}');
```



From equation Surface slope the following condition to validate the solution is defined :

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

The left side of the equation can be calculated for multiple values of the inital slope at the surface $\left.\frac{dV^*}{dr^*}\right|_{R^*}$. With the correct value of $\left.\frac{dV^*}{dr^*}\right|_{R^*}$ the left side of the equation will be minimized. Since similar minimization problems have been solved 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 the SciPy package:

```
from scipy.optimize import minimize_scalar
```

To use `minimize_scalar`, an additional function needs to be defined. The new function fulfills two steps. First the initial value problem (ivp) is solved with the supplied initial slope for a fixed surface potential. With the solution of the ivp, the error defined by (**??**) is calculated and returned. The function `minimize_scalar` then takes care of varying the initial slope to minimize the error.

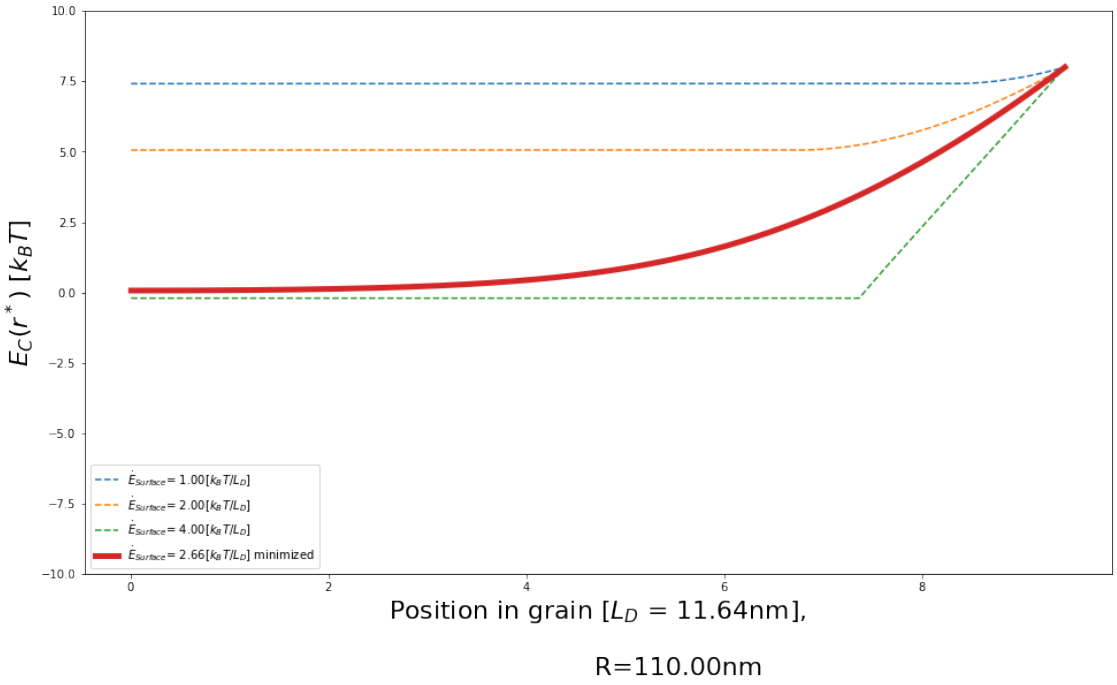
[31]:
```
from scipy.optimize import minimize_scalar


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(fun = min_vdot,
                              args=(grain, E_init_kT, debug),
                              method = 'Bounded',
                              bounds = bounds
                             )
    else:
        res = minimize_scalar(fun = min_vdot,
                              args=(grain, E_init_kT, debug)
                             )
    return res
```

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

[22]:
```
#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',
         linewidth = 5)
```

```
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();
```



The thick line is the result corresponding to the solution minimal error

3.8.4 Additional relevant parameters

From the above solution for spherical grains, additional properties can already be calculated without much computational effort. For instance the total number of free charge carries left inside the grain. From the charge distribution inside the grain, a simple integral over the volume of the sphere will reveal this value. The difference of this value to the value from the flatband situation will reveal the number of charges $N^-_{Surface}$ involved and trapped by the surface reactions.

[49]:
```
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
    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	5.6e-21
grain_sur...	1.5e-13
n	[2.41291...
all_c_at_...	1.1e+03
charges_t...	4e+02
surface_v...	1.5
temp	300
mass_eff	2.7e-31
ND	1e+23
EPSILON	9.9
nb	2e+23
E_Fermi_kT	-4

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

Those results will lead 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.

To do the time consuming calculations (finding the right start conditions) only once, each correct solution we will save in a `DataFrame`. A `DataFrame` is a data structure to organize information similar to Excel Worksheets (tm). 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.

[39]:
```
import pandas as pd
dF_calc = pd.DataFrame()
```

In the following code cell helper functions, which are needed for the parallelization of the jobs are defined.

[40]:
```
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):
    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

The function `calc_solution_by_parameters(T_C, ND, grainsize)` will calculate a specific grain defined by the following parameters:

- T_C = Temperature
- ND = number of donors
- grainsize = Radius of the SMOX grain

For the screening of the multiple combinations of those 3 parameters, first a list of all combinations is generated. In Python a fast way to achieve this is `itertools.product`. This function will generate a list of all combination of its arguments.

[41]:
```
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))
print("T_C, ND, grainsize")
pprint.pprint(combinations)
```

```
T_C, ND, grainsize
[(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 duration of for the full process is measured. The output shows, that the total calculation time is around 1h, when using 8 processes in parallel. This was performed on a desktop PC with the following configuration:

- Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
- 10 cpus
- 64GB memory

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.

[49]:
```
#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.')
```

```
Calc duration: 60.9min.
```

3.9.3 Export/Import data

The date will be saved for later use 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. As a sanity check,some parts of the re-imported data are displayed as a table.

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

[50]:
```
calc_dF_all = pd.read_hdf('results.h5', 'raw')

display(calc_dF_all.iloc[:,0:7].tail())
```

	EPSILON	E_Fermi_kT	E_dot_init_kT	Einit_kT	ND	R	all_c_at_flatband
815	9.86	-1.71	4.69	16.0	1.00e+24	1.00e-07	7797.54
816	9.86	-1.71	4.81	17.0	1.00e+24	1.00e-07	7797.54
817	9.86	-1.71	4.94	18.0	1.00e+24	1.00e-07	7797.54
818	9.86	-1.71	5.05	19.0	1.00e+24	1.00e-07	7797.54
819	9.86	-1.71	5.16	20.0	1.00e+24	1.00e-07	7797.54

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.

[43]:
```
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
```

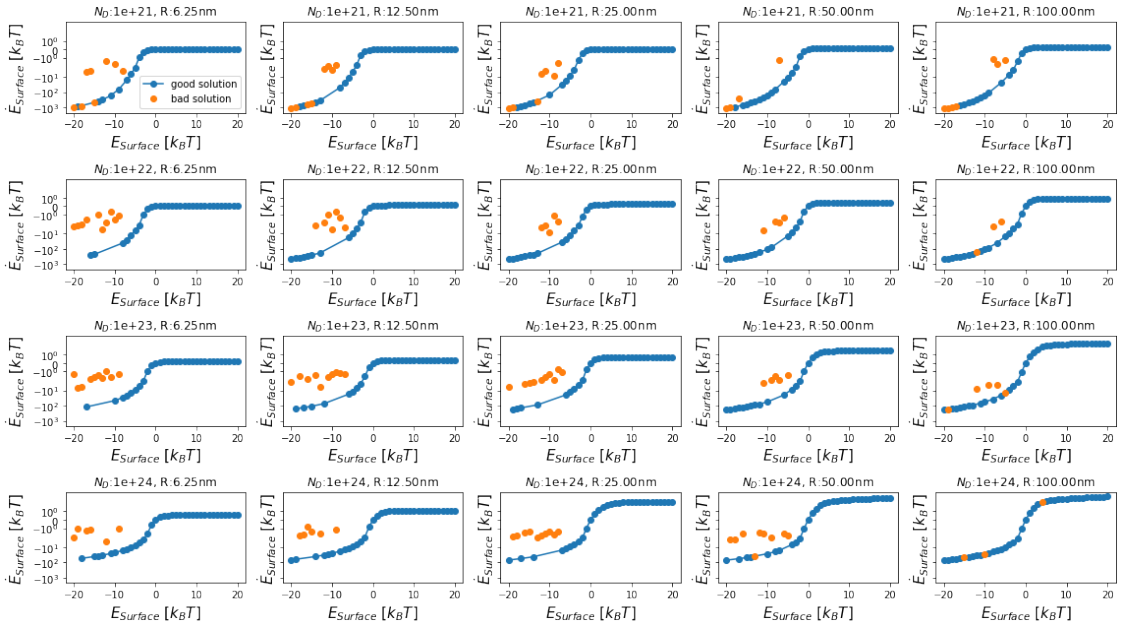
3.9.4 Refine algorithm

Visualize incorrectly found minimum: Sometimes the minimization algorithm does not work as expected. In some cases the minimization algorithm will find "just" a local minimum and miss the global one. Generally this problem could be solved by suppling additional hints to the minimization algorithm, like a range where to look for the global minimum. To do so, first the points, where the minimization "error" is still too high are identified and then recalculate 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.

In this representation the y-axis scaling is set to `symlog`. The description of `symlog` from Matplotlib documentation describes `symlog` is: "The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin." This scaling type overcomes the problem of representing exponential dataset with even negative values.

[44]:
```
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 solution')
    axe.plot(g_bad['Einit_kT'], g_bad['E_dot_init_kT'],'o',
             label='bad solution')

    #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)
fig.axes[0].legend()
fig.tight_layout()
```

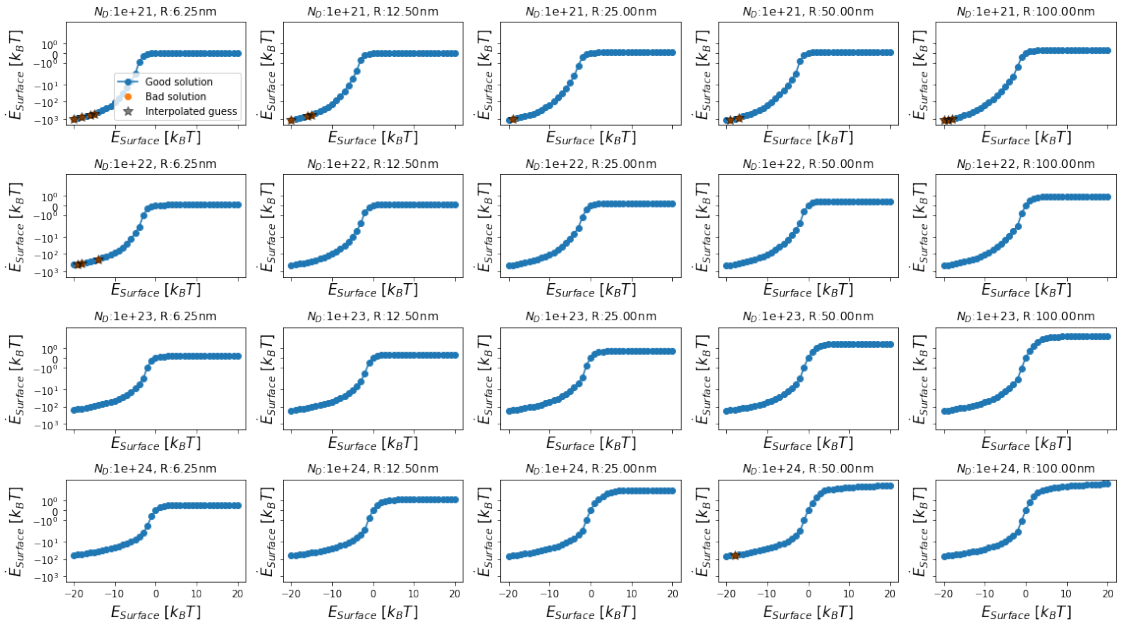


Solutions with an high error after the minimization are represented in green, good results in blue. Since enough results can be considered as good, an estimation of the better values for the "bad" results can be derived.

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 lay. As humans can, so does Python. For this problem the “good” solutions are used to estimate the correct values of the “bad” 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.

[47]:
```
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)
```



Now the green stars represent the guesses of the correct solutions

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.

[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
```

[36]:
```
#filter only rows which need to be recalculated
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.

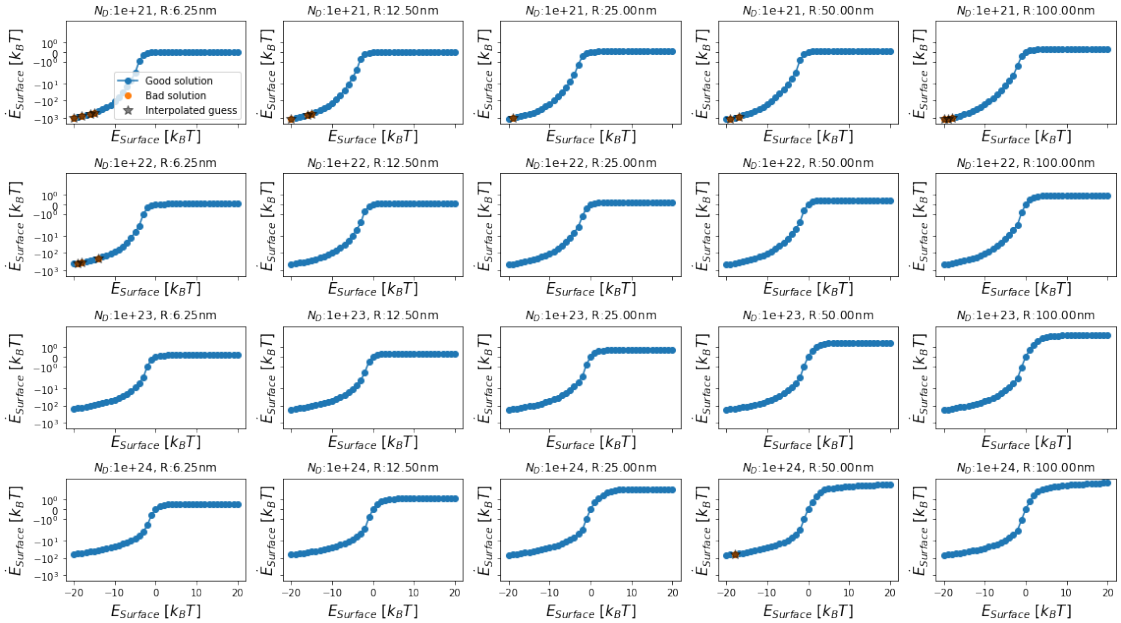
[29]:
```
for s in new_sers:
    calc_dF_all.loc[s.name] = s

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 its quality.

[48]:
```
calc_dF_all = pd.read_hdf('results.h5', 'corr')
print(len(calc_dF_all[calc_dF_all['res']>5]))
check_solutions(calc_dF_all)
print(f'The maximal "error" of the minimization is :{calc_dF_all["res"].max()}')
```

```
2
The maximal "error" of the minimization is :53.768085981645115
```



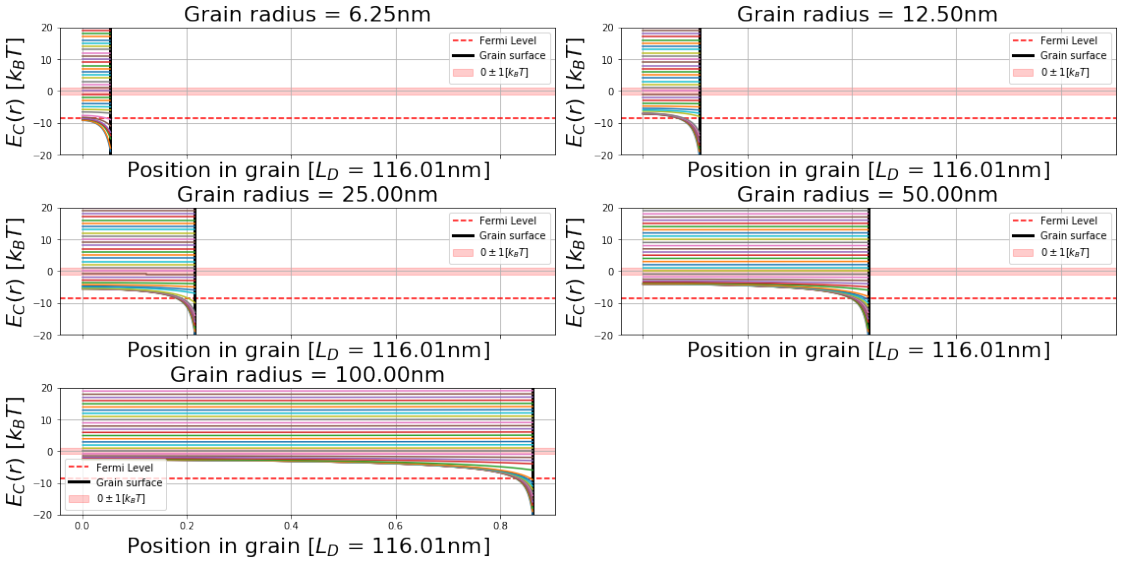
3.10 Shape of the potential drop inside the grain

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

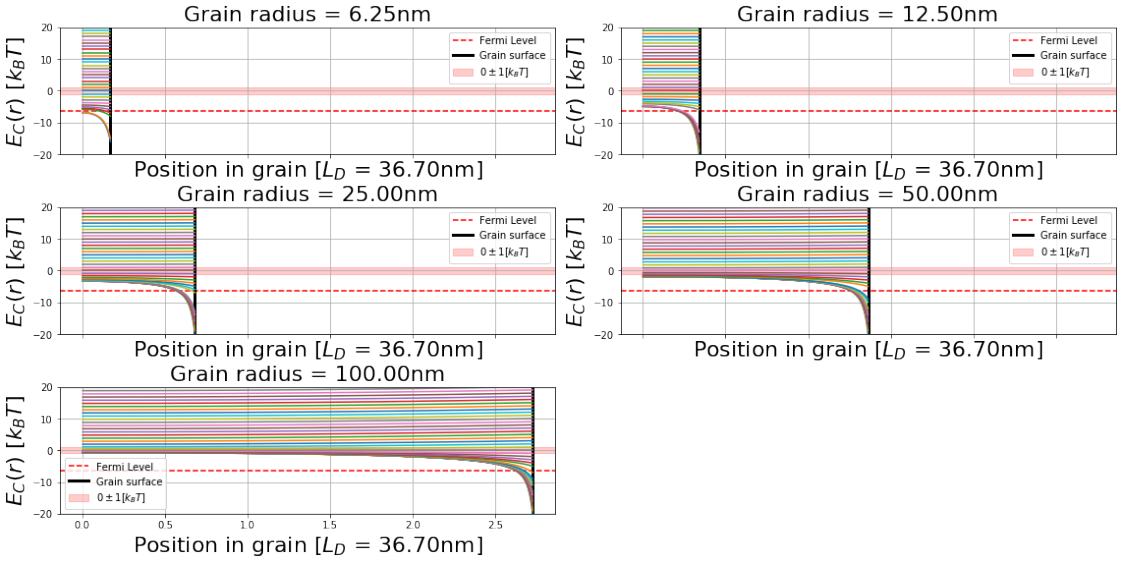
[74]:
```
for ND, calc_dF in calc_dF_all.groupby('ND'):
    fig, axes= subplots(3,2,figsize = (16,9), sharex=True)

    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)

        grain = create_grain_from_data(calc_dF_grainsize)
```

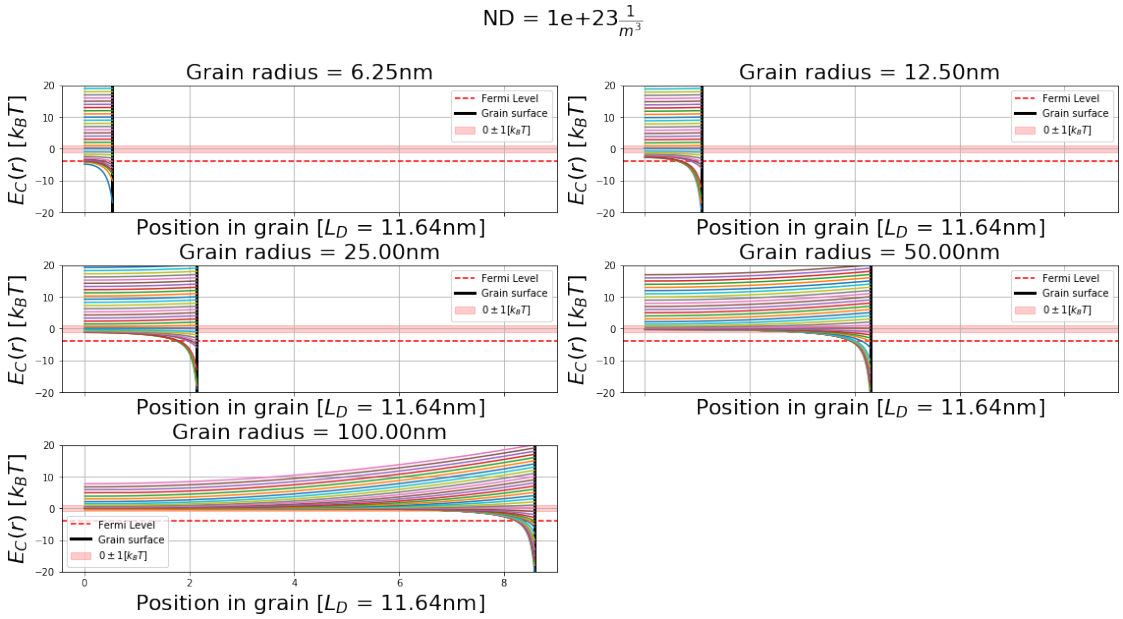
```
        axe.axhline(-grain.material.J_to_kT(grain.material.Diff_EF_EC),
                    linestyle='--',color='r', label='Fermi Level')

        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()
```

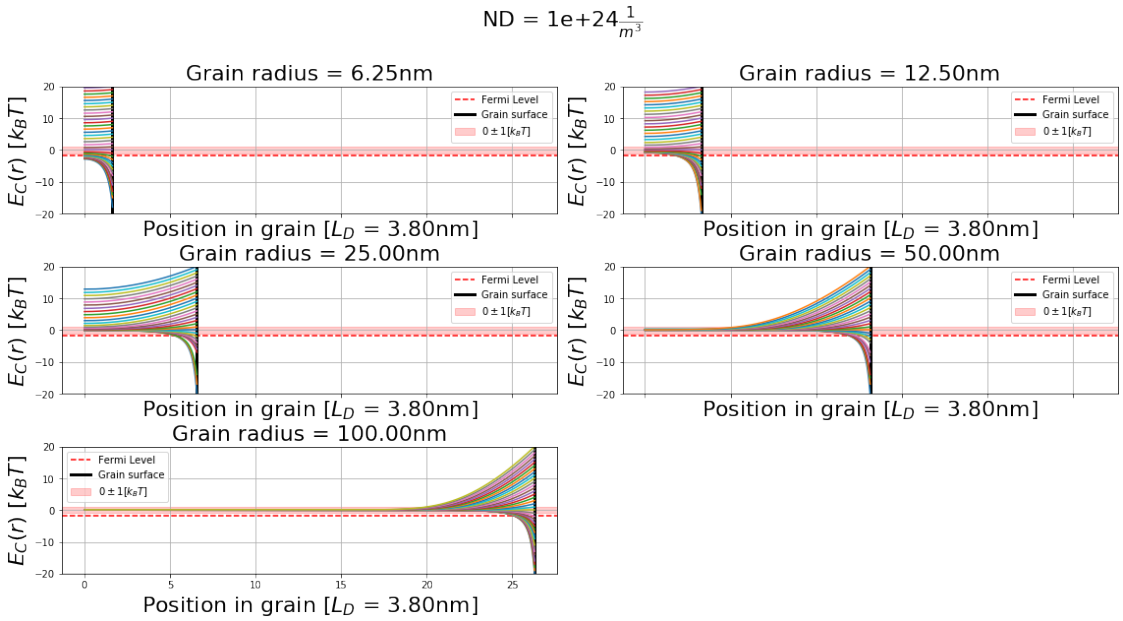
ND = 1e+21$\frac{1}{m^3}$



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




ND = 1e+23 $\frac{1}{m^3}$
Grain radius = 6.25nm
Grain radius = 12.50nm
Grain radius = 25.00nm
Grain radius = 50.00nm
Grain radius = 100.00nm
$E_C(r)$ $[k_BT]$
Position in grain [L_D = 11.64nm]
Fermi Level
Grain surface
$0 \pm 1[k_BT]$



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 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 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 additionally introduces directly structure in the data.

In the next notebook this calculated data will be used derive the total resistance of a grain. 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

[Hue11] HUEBNER, Michael: *New Approaches for the Basic Understanding of Semiconducting Metal Oxide Based Gas Sensors : Sensing , Transduction and Appropriate Modeling*. 2011. – 156 S. – ISBN 978–3–8440–0893–7

[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

[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

[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

From SMOx-grains to sensor resistance

March 20, 2020

Contents

1 Abstract

The type of gas sensors under investigation consist of multiple, small semiconducting grains. In the previous chapter the effect of surface reactions onto the SMOX grain were simulated. The results show that the charge distribution inside the grain depends on the surface potential, defect concentration and radius. When such a grain is used as a sensor, a bias voltage is applied and the resulting electrical current is measured to calculate the total resistance. The applied bias voltage is selected in way, that it is not interfering with the semiconductor itself. Typically the voltage is chosen to be below 1 $k_B T/e$ per grain.

To numerically derive the total resistance from a specific charge distribution inside the grain, the actual electrical conduction path through the grain needs to be simulated first. In this part of the thesis this conduction path will be first numerically simulated and in a second step the total resistance of the grain will be derived from the simulated conduction path.

2 Review

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

[4]:
```
%pylab inline
from part2 import *
```

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

3 Load the results

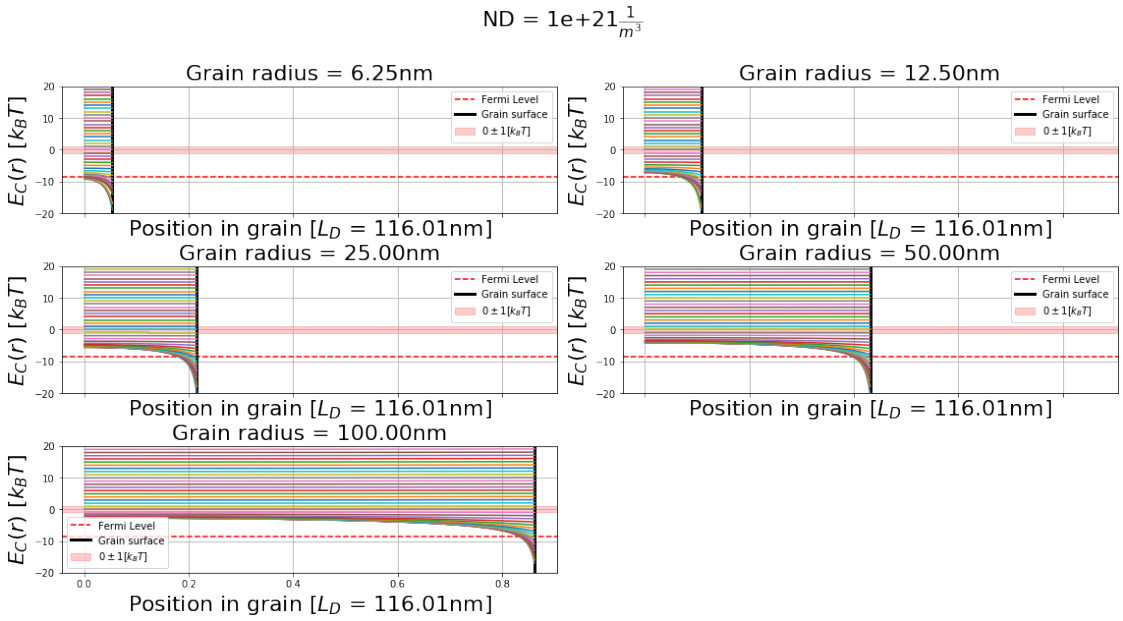
[24]:
```
calc_dF = pd.read_hdf('results.h5', 'corr')
calc_dF.index = range(len(calc_dF))
calc_dF_all = calc_dF[calc_dF['res']<2]
```

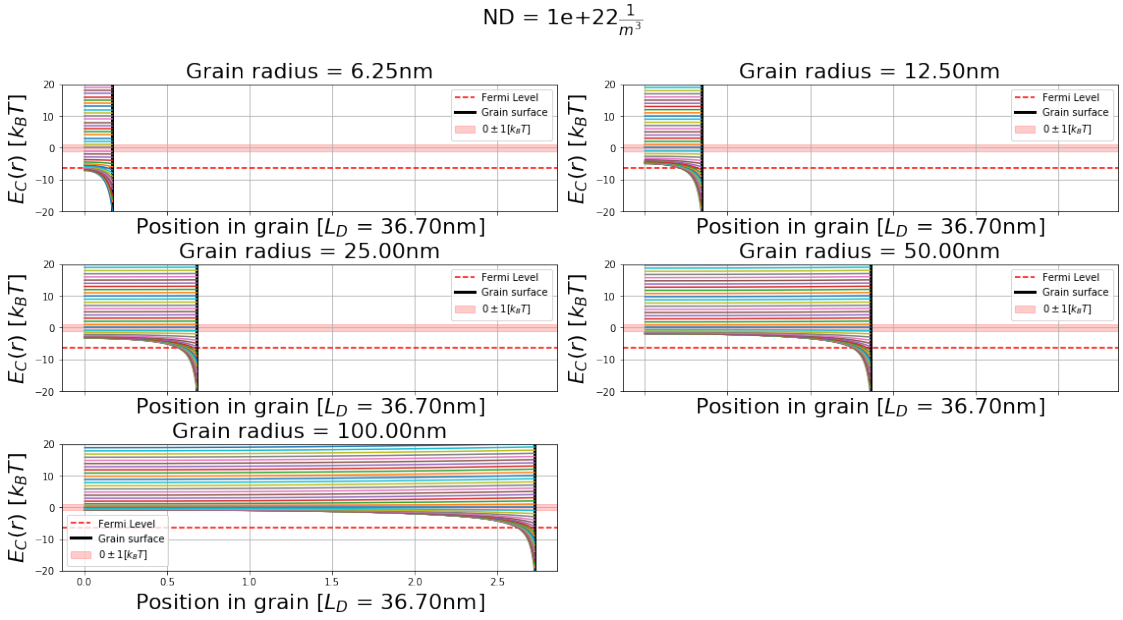
Again in this notebook the final results from the previous chapter are represented. The redundancy of such a representation (especially in the printed version) may seem overexaggerated. But when working with the interactive Jupyter notebook this choice seems legitimated, since each notebook represents 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.

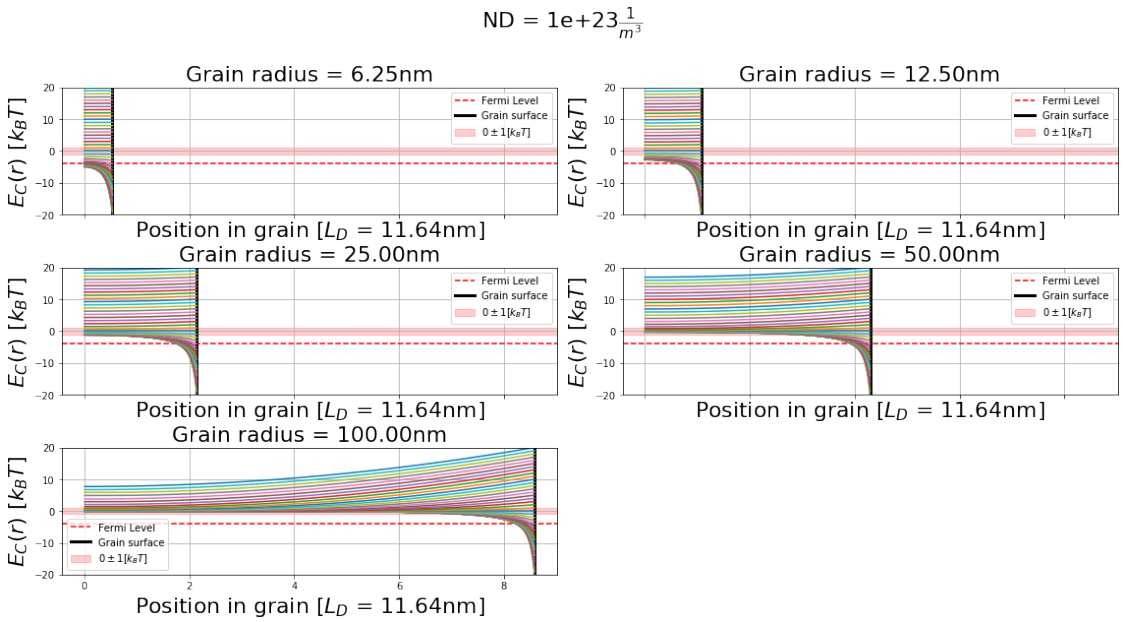
Finding the compromise of this thesis between the printed version and the interactive notebook generates in this case some “glitches”. With the strong focus to introduce the Jupyter notebook environment in this thesis the redundant representation of the figures was chosen in this case.

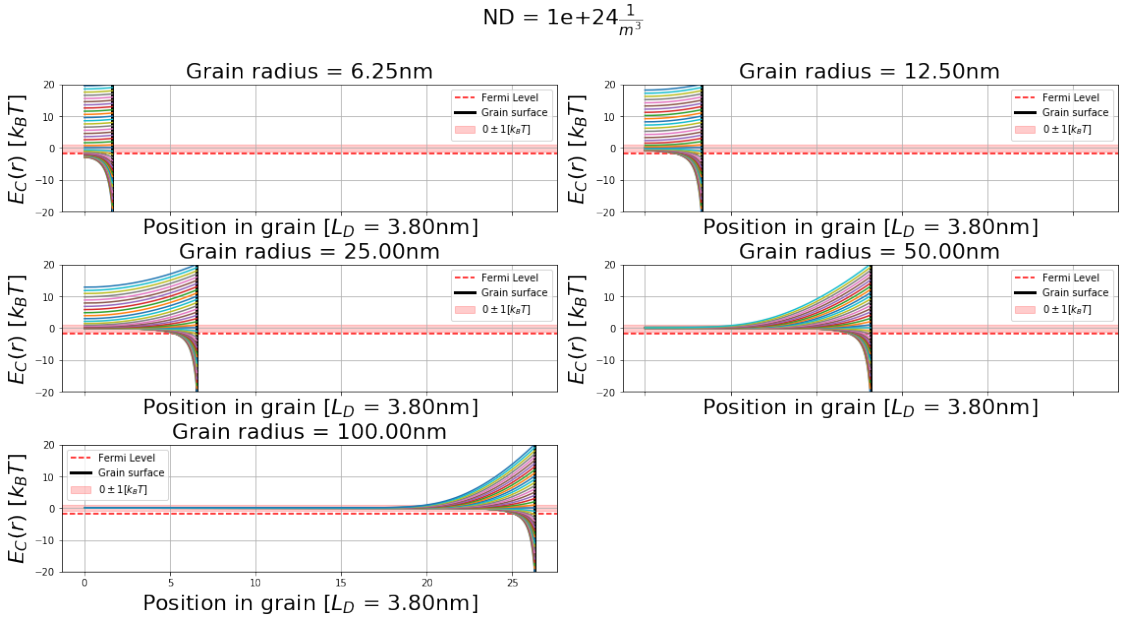
[26]:
```
for ND, calc_dF in calc_dF_all.groupby('ND'):
    fig, axes= subplots(3,2,figsize = (16,9), sharex=True)

    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),
                    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+23 $\frac{1}{m^3}$
Grain radius = 6.25nm
Grain radius = 12.50nm
Grain radius = 25.00nm
Grain radius = 50.00nm
Grain radius = 100.00nm
$E_C(r)$ [k_BT]
Position in grain [L_D = 11.64nm]
Fermi Level
Grain surface
0 ± 1[k_BT]



These graphs show 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 calculations the position of the conduction band inside the grain, resulting from a specific band bending, is known. The previously defined `Material` class allows to calculate the exact number of free charges in the conduction band. Using these information it is now possible to assign to each position 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:

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

Here q is the electrical charge of an electron, n the density of electrons, p the density of holes 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:

$$Resistivity = \rho = \frac{1}{\sigma} \tag{3}$$

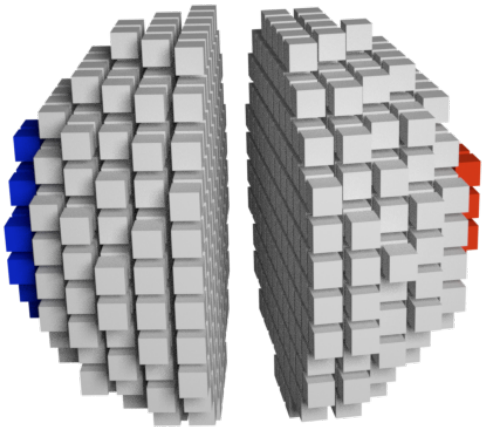
To derive from the known conductivity inside the grain the total resistance of the grain, the current path needs to be known. The current flow along the field lines inside the grain, which are equivalent to the gradient of the potential. Therefore the potential distribution inside the grain needs to 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. With this model a solid grain is approximated as a network for connected small resistors. For each cube i with the dimension $lxlxl$, the resistance R_i of the cube can be calculated as follows:

$$R = \frac{\rho * l}{A} = \frac{\rho * l}{l * l} = \frac{\rho}{l} \tag{4}$$

$$\tag{5}$$

A is the area of one side of the cube.

The colored areas in the picture indicate the areas, where a bias potential will be applied to generate a virtual electrical field.



To simulate how the electrical field causes by the bias voltage propagates 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 converge 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.

This relaxation process can be supported by the means of modern matrix operations. For this I will shorty derive how U_0 is calculated from the surrounding U_i and then describe how a "matrix convolution" will be used to solve the problem efficiently.

First we will need to combine Ohm's law and Kirchhoff's first law:

$$R = \frac{\Delta U}{I} \qquad \text{(Ohm's law)}$$

$$\sum_i I_i = 0 \qquad \text{(Kirchhoff's first law)}$$

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

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

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

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

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

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

$$= \frac{q * \mu_n}{q * \mu_n} \frac{\sum_i U_i * n_i}{\sum_i n_i} = \frac{\sum_i U_i * n_i}{\sum_i n_i} \qquad (U_0 \text{ from } U_i) \tag{12}$$

The cube's face area A and l cube's length vanish from the equation since 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 calculate the value of each n_i at arbitrary points r inside the grain, one additional step is required. Due to the nature of the numerical solution from the previous notebook we know the value of n only at specific points. To gain the value 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) like this:

> 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

    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 the data structure of the grain for further simulations is implemented. The $N \times N$ cubes will be represented by a `numpy` array.

[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
```

```
    '''

    #find the center:
    #length divided by two without rest; +1;
    #-1 since we start counting at 0

    cx = d.shape[0]//2+1-1
    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):
    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
                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 a faster response of the interactive elements, we will pre initialize all the grains for all available data. Due to the similarity of the $N \times N$ data structure to common pixel based pictures, Matplotlib's function `imshow` is very handy to represent such 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
```

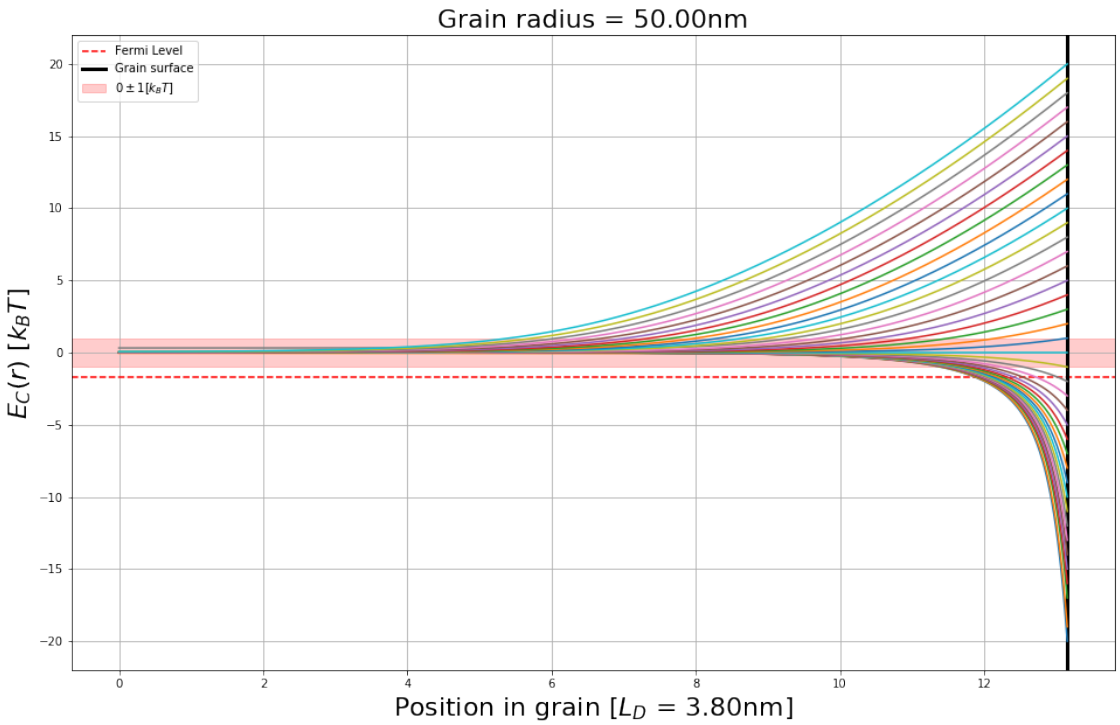
```
Initalized 820 of 803.
```

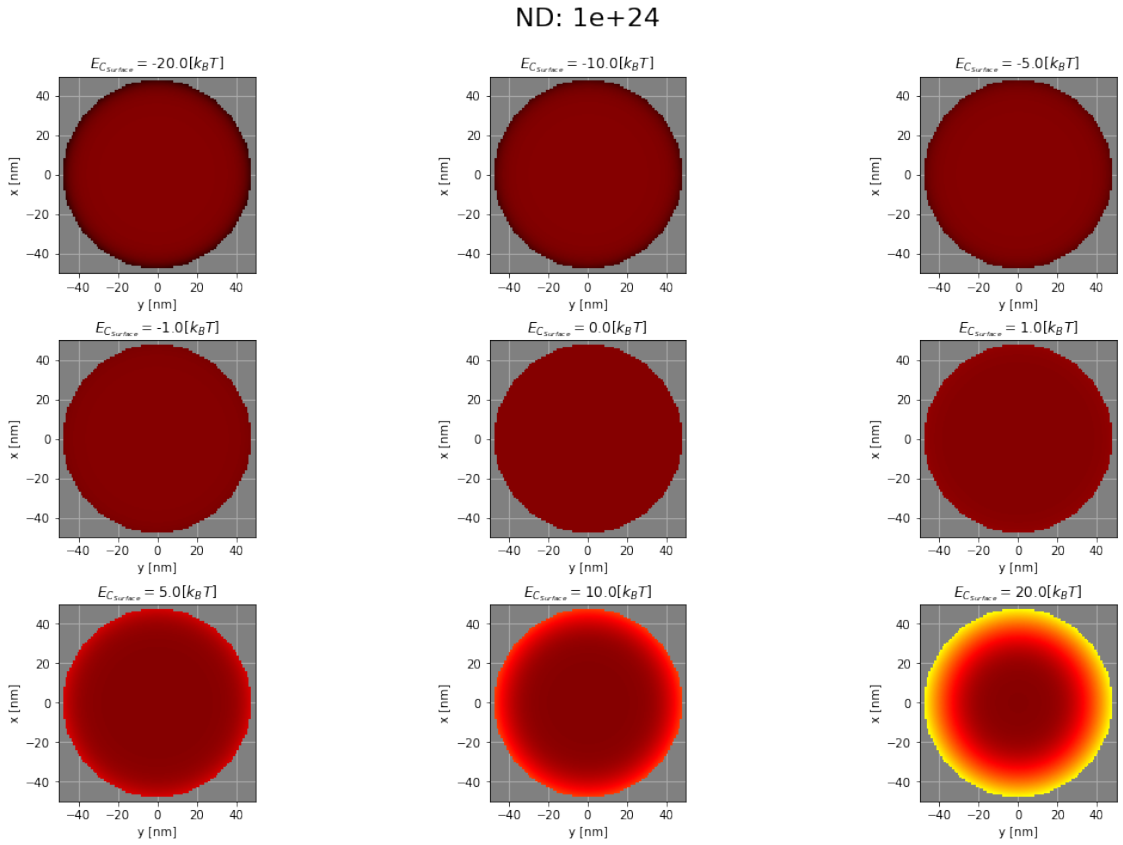
To visualize the grain, the conductivity is represented for different initial surface potentials expressed in units of $\frac{k_B T}{e}$

[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]')

        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
        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
Fermi Level
Grain surface
$0 \pm 1[k_BT]$
$E_C(r)$ $[k_BT]$
Position in grain [L_D = 3.80nm]



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

[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 of equation (U_0 from U_i). 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 does not change anymore, the iterations can be stopped.

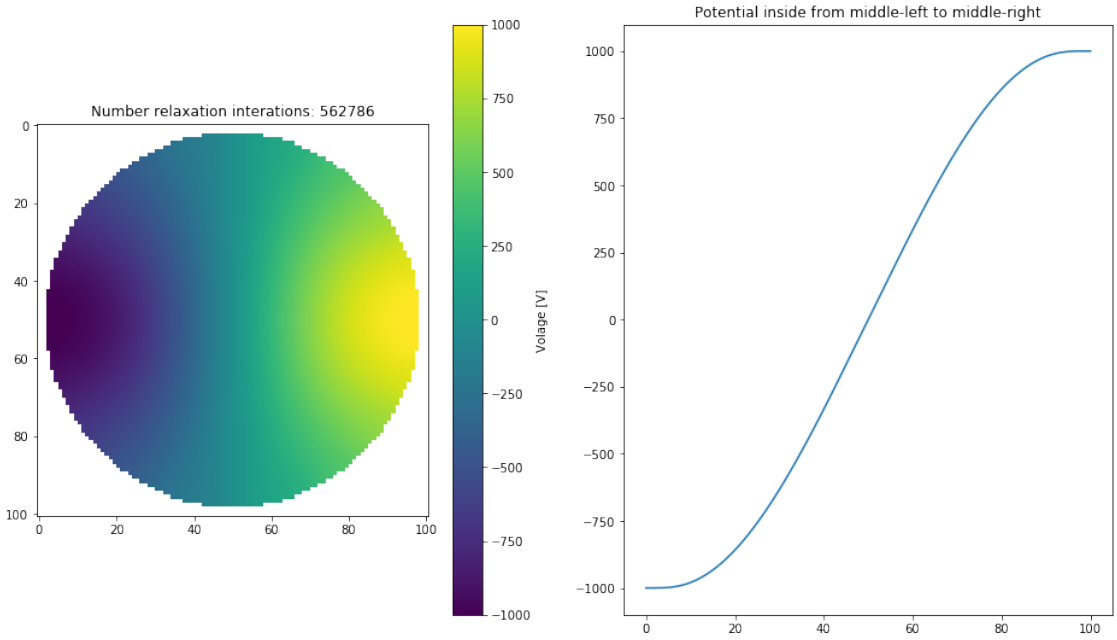
[63]:
```
from scipy import signal


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]

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()
```



The video can then be loaded again and played back in the notebook. For the PDF-Version this is a link to a online hosted version of the video. Video on GitHub

[1]:
```
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 traversing the center slice 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 current 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 contrast to the resistance at the flatband situation, the resistance for a specific band bending 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.

[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.

[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
```

[5]:
```
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.')
```

```
Duration of the calculation: 51.81 min.
```

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

Again, those results are saved in a local file and re-imported as a sanity check.

[3]:
```
calc_dF_sol_with_rel_res_change.to_hdf('numerical_sol.h5','raw')
```

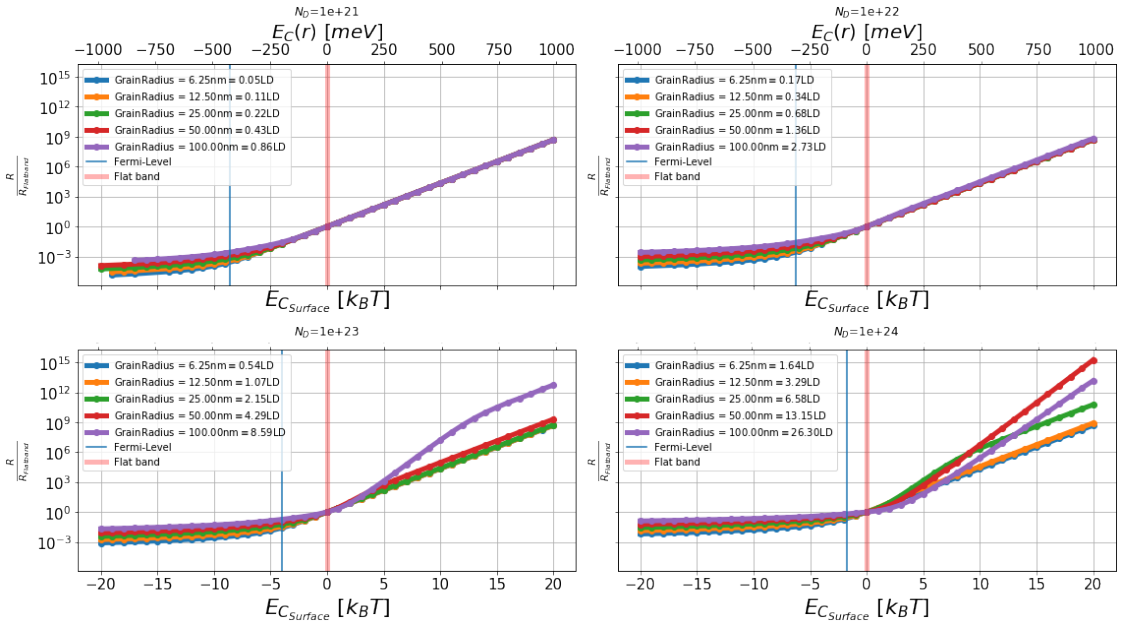
[168]:
```
calc_dF_sol = pd.read_hdf('numerical_sol.h5','raw')
```

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:

For those 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 Rad is small compared to the Debye length ($Rad < 5 * L_D$). In this case the resistance changes, due to the depletion layer controlled conduction, similar for all grains. This can be explained by the fact, that these grains are fully depleted. Without a separated depletion layer, possible volume effects of the different grain sizes do not exist. For those small grains, the effect of the different grain size has an impact on the sensing properties only when the conduction band at the surface approaches the Fermi level. As seen in the last notebook, the dimensions of the accumulation layer are much smaller compared to the depletion layer. Therefore the grains are never getting "fully accumulated" and the grain size has again an influence on the transduction mechanism.

4.4.2 Large grains:

For large grains ($Rad > 5 * L_D$), which are not fully depleted, two effects may overlay each other. First, the effect of a high resistive surface layer increases the resistance and second, the low resistive bulk volume shrinks. Those two effects add up and increase in some cases the slope of the resistance change more than for fully depleted grains. The small grains, which are fully depleted, do not show this effect. 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 one of smaller grains.

4.4.3 Flat band situation:

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

4.5 Conclusion

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 data corresponding to this grain size will reveal "hidden" properties as the Debye length or doping level N_D. As an additional input for a deeper understanding of 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

Comparing numerical results with experimental data

March 20, 2020

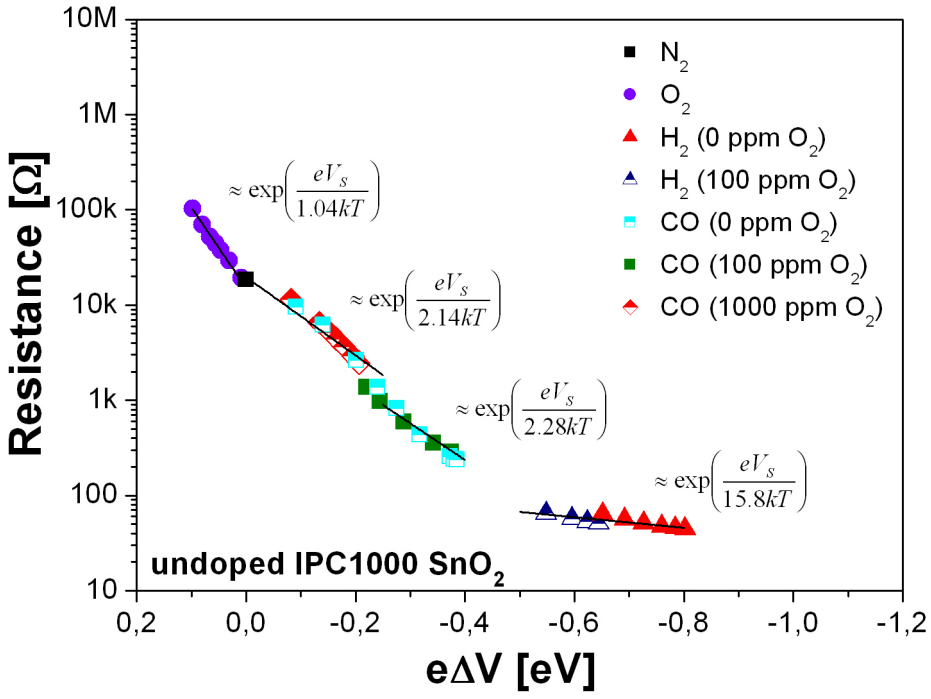
Contents

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 dataset of the graphical representation below originates from the Phd. thesis of Julia Rebholz: [Reb16].

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 simulation.



This figure shows the dependency of the resistance on the band bending changes for the undoped SnO_2 nanopower based gas sensor. $e\Delta = 0$ is denoted as the situation in N_2

2 Fitting experimental data to numerical results

2.1 Importing experimental and numerical data

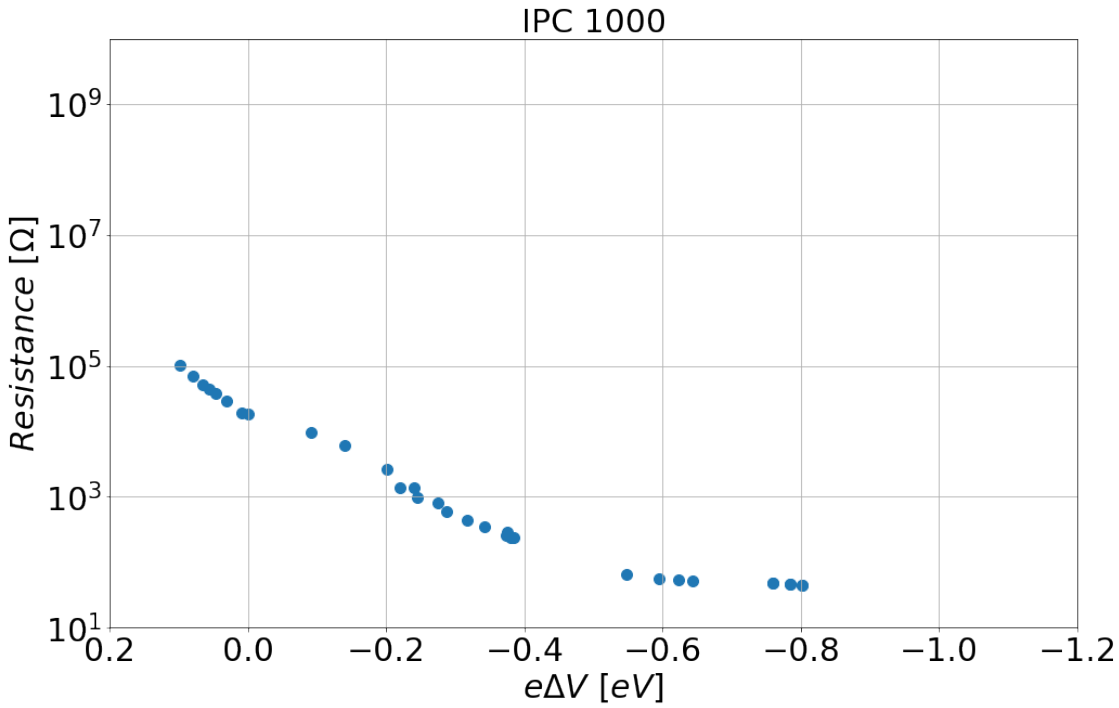
The experimental data is saved in an excel file, which will be loaded by using the tools provided by `pandas`.

[1]:
```
#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

[2]:
```
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('$e\Delta V$ $[eV]$', fontsize = labelsize)
    axe.tick_params(axis='both', which='both', labelsize=labelsize)
    axe.grid()


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);
```



This figure represents the experimental data as show before, but plotted using the Python environment. The dependency of the resistance on the band bending changes is shown for the undoped SnO_2 nanopower based gas sensor. $q\Delta V = 0$ is denoted as the situation in N_2.

2.3 From R_{V_S} to ΔR_{V_S}

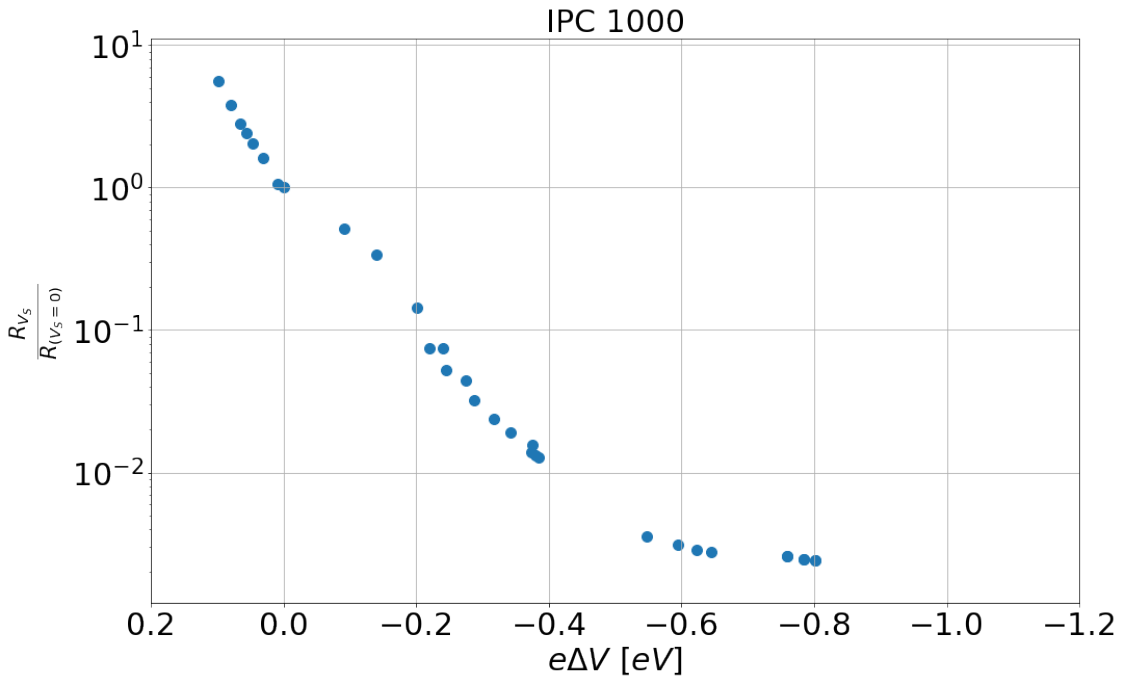
In the experimental dataset the value at $0qV_s$ represent the data points measured under nitrogen. Therefore $\Delta R_{V_S} = \frac{R_{V_S}}{R_0}$ is calculated by :

- First: derive the resistance R_0 under nitrogen
- Second: divide all resistance values R_{V_S} by R_0

[49]:
```
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);
axe.set_title(sens, fontsize = 30)
```

[49]: `Text(0.5, 1.0, 'IPC 1000')`



The resistances have been normalized to the resistance at $q\Delta = 0$

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 of qV_s as 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 values of qV_s from the experimental data. Once this is done, the different between the numerical model and the exp. data can be calculated and evaluated.

[50]:
```
#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 "IPC1000" is known to be in average radius of 55nm (diameter of 110nm) [DRRM+99]. 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 into account

[51]:
```
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()
best_fits_error_dF = pd.DataFrame({'error':sum_of_squares_grainsize})

T_temp = [i[0] for i in best_fits_error_dF.index]
R_temp = [i[1] for i in best_fits_error_dF.index]
ND_temp = [i[2] for i in best_fits_error_dF.index]
best_fits_error_dF['T'] = T_temp
best_fits_error_dF['Radius'] = R_temp
best_fits_error_dF['$N_D$'] = ND_temp
best_fits_error_dF.index = range(len(best_fits_error_dF))

display(best_fits_error_dF)

caption = Latex(r'''\begin{center}
Table with the best fit parameters
\end{center}''')
display(caption)
```

	error	T	Radius	N_D
0	6.822288	300.0	5.000000e-08	1.000000e+22
1	7.665906	300.0	1.000000e-07	1.000000e+22
2	14.358742	300.0	1.000000e-07	1.000000e+21
3	18.771491	300.0	5.000000e-08	1.000000e+21
4	175.204658	300.0	5.000000e-08	1.000000e+23
5	1256.744498	300.0	1.000000e-07	1.000000e+23
6	9905.878196	300.0	5.000000e-08	1.000000e+24
7	39709.781227	300.0	1.000000e-07	1.000000e+24

Table with the best fit parameters

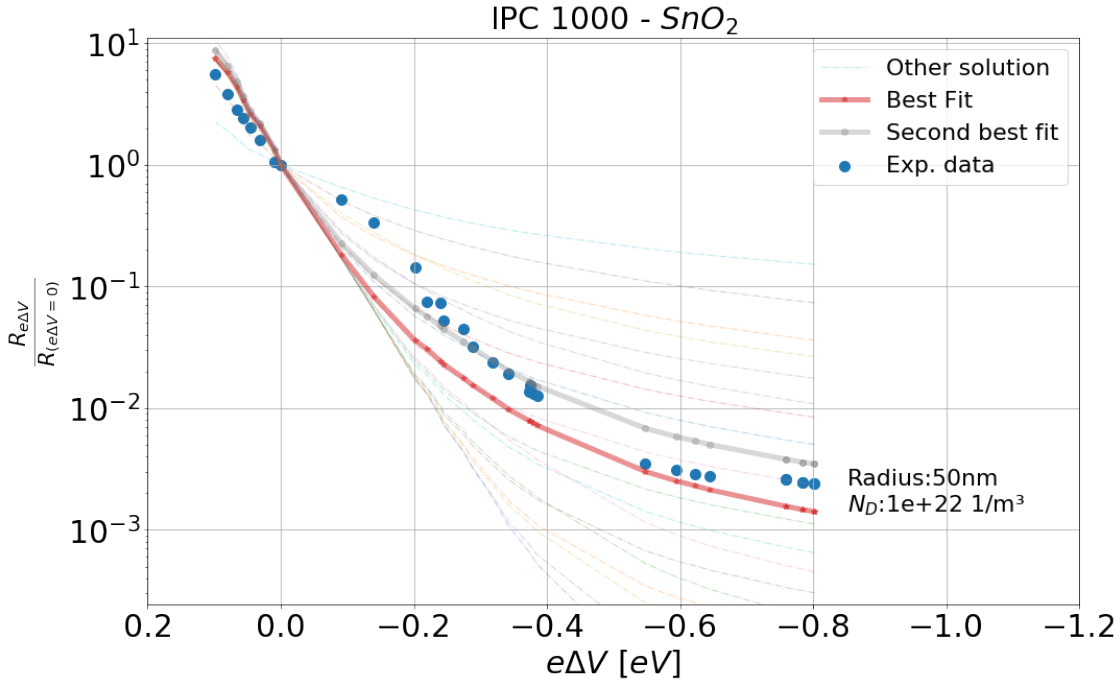
2.6 Representation of the fit

Finally the best fit results can be represented graphically.

[70]:
```
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 = 'Second best fit'
    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[0:1]:
        axe.text(last_x-0.05,last_y,
            f'Radius:{grain_tuple[1]*1e9:.0f}nm\n$N_D$:{grain_tuple[2]:.2} 1/m³',
                fontsize=22)
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)
axe.set_title(sens, fontsize = 30)
```

[70]: `Text(0.5, 1.0, 'IPC 1000 - $SnO_2$')`



Comparison of the numerical results with the experimental data points

2.7 Summary

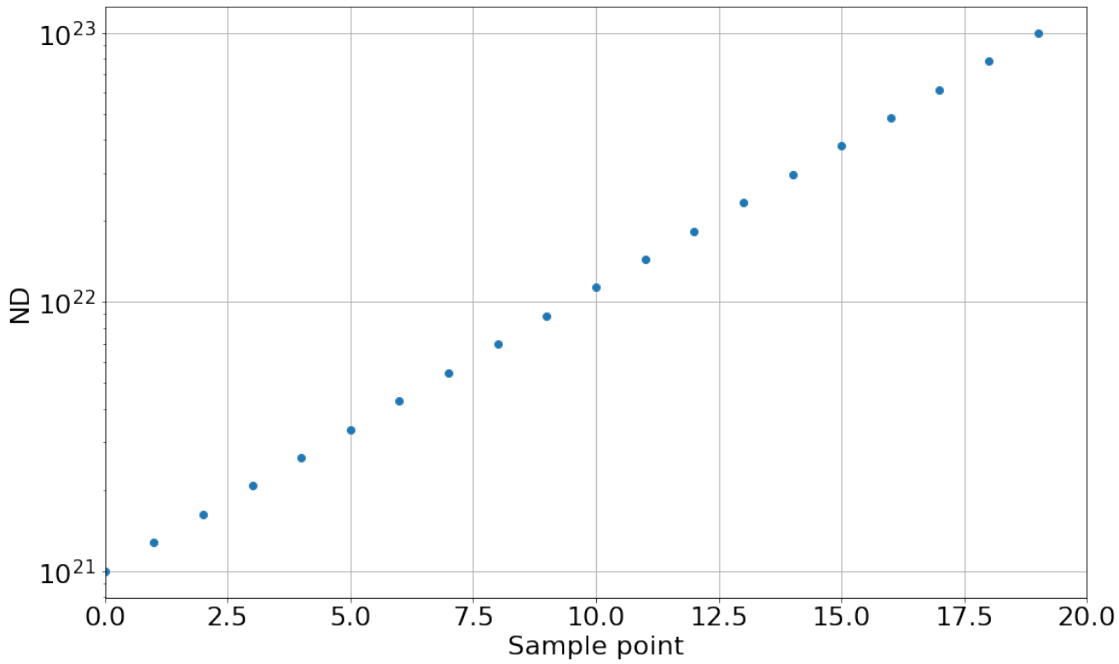
The two best fits do not fit perfectly to the experimental data. One obvious reason is the coarse screening of the grain size and and N_D. A additional simulation of grains with a radius of 55 nm and a finer screening of N_D will turn out to be helpful for a better result of the fit. On the other side 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.

3 Second iteration

With the approximate values of the correct model, the previous steps of creating a look-up table for multiple model is repeated. This steps is not shown in this work. Basically the screening of the parameters is limited to grains with a radius of 55 nm and a value of N_D between 1×10^{21} and 1×10^{23}. For the second iteration 20 values between these boundaries have been selected. The points are evently spaced on a logarithmic scale. The `numpy` function `logspace` was used for this purpose.

[7]:
```
NDs = np.logspace(21,23,20)
fig, axe = subplots(figsize=((1+5**0.5)/2*9,9))
axe.plot(NDs,'o')
axe.set_yscale('log')
```

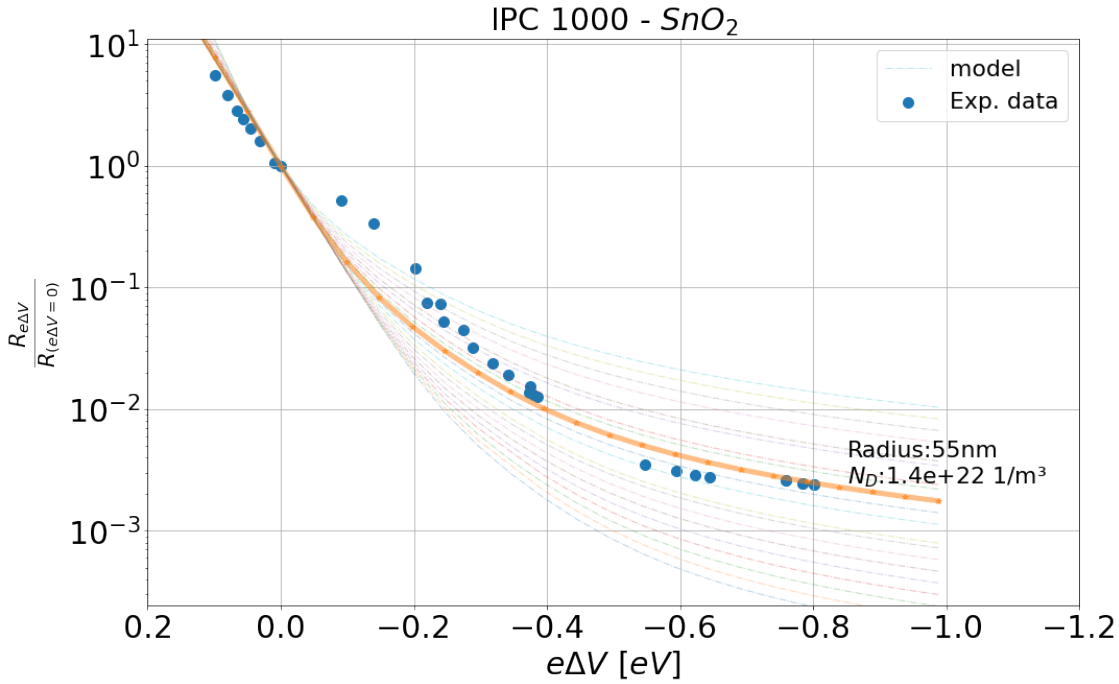
```
axe.set_ylabel('ND', fontsize = 22)
axe.set_xlabel('Sample point', fontsize = 22)
axe.set_xlim(0,20)
axe.tick_params(axis='both', which='both', labelsize=22)
axe.grid()
```



The results from the new numerical calculation are saved in a file called "numerical_sol_4_part_4_ipc1000_55nm.h5" and will be loaded in the next cell. For refitting the results the above procedure is repeated with the new numerical dataset. The required steps to fit the data and represent the results are exported in a new function called `fit_exp_with_num_data`. This function needs the exp. dataset, the numerical dataset, a title for the figure and optionally a text for the caption. With these arguments the output of the fit is represented in the notebook.

[4]:
```
from IPython.display import display, Math, Latex
from fit_exp_data import fit_exp_with_num_data


calc_dF = pd.read_hdf('numerical_sol_4_part_4_ipc1000_55nm.h5','raw')

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


caption_figure = 'Optimized fit with a finer screening of the parameters'
```

```
sens = 'IPC 1000 - $SnO_2$'
bf = fit_exp_with_num_data(dF_exp, calc_dF, sens=sens, caption_figure='Best fit␣
 ↪results with finer screening of parameters.')
```



Best fit results with finer screening of parameters.

	error	T [°C]	Radius [nm]	N_D [$1/m$]
0	4.427580	300.0	55.0	1.438450e+22
1	4.782059	300.0	55.0	1.128838e+22
2	5.665853	300.0	55.0	1.832981e+22
3	5.986754	300.0	55.0	8.858668e+21
4	6.967131	300.0	55.0	2.335721e+22
5	8.322148	300.0	55.0	6.951928e+21
6	9.304737	300.0	55.0	5.455595e+21
7	10.971085	300.0	55.0	4.281332e+21
8	12.569745	300.0	55.0	3.359818e+21
9	13.993224	300.0	55.0	2.636651e+21

Table with the best fit parameters

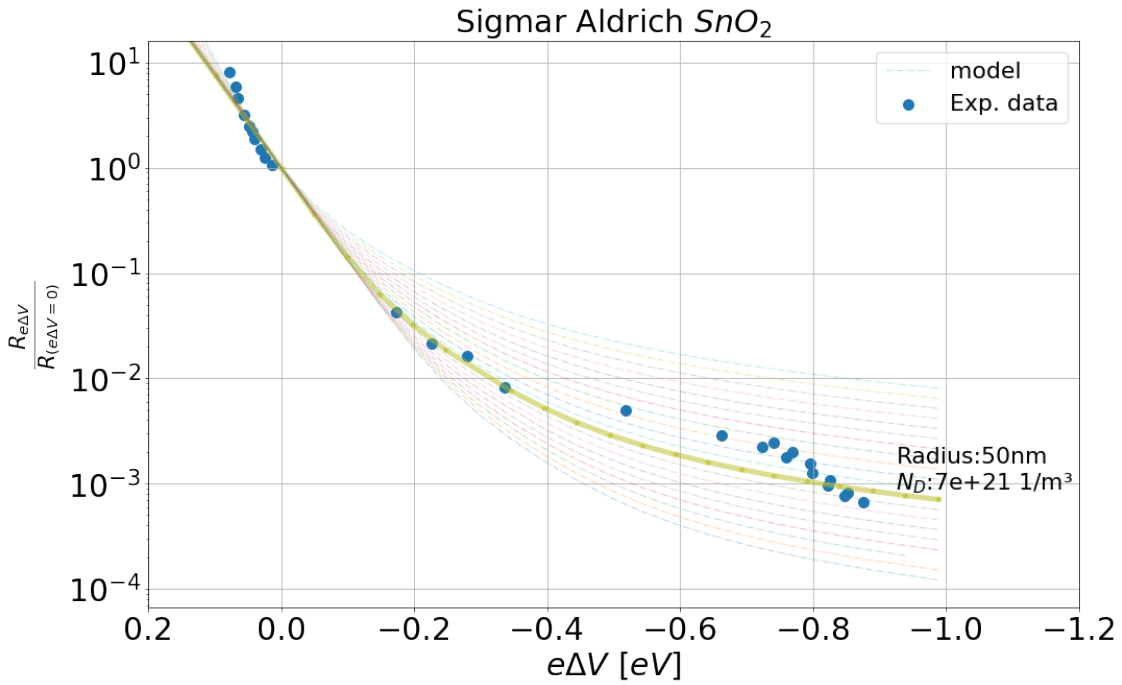
	IPC 1000 - SnO_2
E_C - E_F	0.29 eV
N_D	1.44e+22 1/m³
n_b	2.87e+22 1/m³
L_D	30.60 nm

Table with material properties of the best fitting material.

4 Results for a commercial available SnO_2

In the thesis of Julia Rebholz [Reb16] the commercially available "SnO_2" from Sigmar Aldrich was also tested. In the next part experimental results from simultaneous work function and resistance measurements are being fitted with numerical results. The average grain radius of this material is around 50nm

[67]:
```
from IPython.display import display, Math, Latex
from fit_exp_data import *


calc_dF = pd.read_hdf('numerical_sol_4_part_4_SA_50nm.h5','full')

dF_exp = pd.read_excel('Kelvin_Data.xlsx', sheet_name='SA_Benni').
 ↪sort_values(by='dV')


caption_figure = 'Optimized fit with a finer screening of the parameters'
sens = 'IPC 1001'
bf = fit_exp_with_num_data(dF_exp, calc_dF, sens='Sigmar Aldrich $SnO_2$',␣
 ↪caption_figure='Best fit results with finer screening of parameters.')
```



	error	T [°C]	Radius [nm]	N_D [$1/m$]
0	2.379963	300.0	50.0	6.951928e+21
1	2.546552	300.0	50.0	8.858668e+21
2	3.129596	300.0	50.0	5.455595e+21
3	4.300544	300.0	50.0	4.281332e+21
4	4.370771	300.0	50.0	1.128838e+22
5	5.592128	300.0	50.0	3.359818e+21
6	6.953802	300.0	50.0	2.636651e+21
7	8.195738	300.0	50.0	2.069138e+21
8	9.110344	300.0	50.0	1.438450e+22
9	9.314258	300.0	50.0	1.623777e+21

Table with the best fit parameters

	Sigmar Aldrich SnO_2
E_C - E_F	0.33 eV
N_D	6.95e+21 1/m³
n_b	1.39e+22 1/m³
L_D	44.01 nm

Table with material properties of the best fitting material.

5 Conclusion

In this last section the calculations from the numerical model have been fine tuned for different SnO_2 materials. The model's parameter of the grain radius and temperature have been fixed corresponding to the values from the experimental results. Additionally the parameter of the defect concentration N_D was screen in fine grained steps around the estimated value. The resulting numerical datapoints were fitted against the experimental results. With the best fitting numerical model otherwise hidden intrinsic properties of the semiconducting material have been computed.

It could be shown, that the numerical method to describe the relation between surface potential change ΔV_S and resistance change ΔR with respect to the grain radius R, operation temperature T and defect concentration N_D is suitable to fit experimental results and gain additional information about the sensor as:

1. Distance between the conduction band (E_C) and the Fermi level (E_F)
2. Concentration of defects in the bulk N_D
3. Concentration of free charge carriers in the conduction band in the bulk n_b
4. Debye length L_D of the semiconductor

6 Bibliography section

References

[DRRM+99] DIÉGUEZ, A. ; ROMANO-RODRÍGUEZ, A. ; MORANTE, J. R. ; KAPPLER, J. ; BÂRSAN, N. ; GÖPEL, W.: 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. In: *Sensors and Actuators, B: Chemical* 60 (1999), nov, Nr. 2, S. 125–137. http://dx.doi.org/10.1016/S0925-4005(99)00258-0. – DOI 10.1016/S0925–4005(99)00258–0. – ISSN 09254005

[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

Summary & Outlook

March 20, 2020

1 Summary

In this work a numerical model to simulate the conduction mechanism in a SMOX based gas sensor was developed. Based on freely available tools around the "Jupyter Notebook" programming environment the following steps have been demonstrated:

- solving differential equations
- solving integrals
- numerical simulation by using relaxation algorithms based on convolution operations
- fitting numerical results to experimental data

The results from the numerical model match with experimental data. Additionally this work combines the necessary elements for further research in one compact notebook. The chosen representation form of the "Jupyter notebook" unifies the explanatory descriptions, development of algorithms, data import and analysis and graphical representation. With this advantage further research about the transduction mechanism of SMOX sensors can easily be based on this work.

2 Outlook

For a better understanding of the mechanism involved in the sensing process of SMOX gas sensors, the chemical surface reactions will need to be included in this model. For now, this numerical model is able to predict otherwise not measurable properties from simultaneous work function and resistance measurements. This includes:

- the concentration of surface charges involved in the sensing process
- the influence of temperature changes on the semiconductor

Typically the changes in resistance of such a SMOX sensor is used as a signal. Often this one dimensional information of resistance changes is not enough for a detailed characterize of the chemical interaction. Nevertheless the change in resistance is a direct consequence of the changed surface potential. And again, the surface potential changes are triggered by chemical reactions at the surface. Very simplified, the information in the measured resistance (output) is the "encrypted" information of the surface reaction (input). For the "decryption" of this hidden information the output and input needs to analyzed.

To increase the amount of input and output data points, the temperature of the sensor can be modulated. A modulation of the operation temperature while measuring the resulting resistance changes is one of the modern ways to operate gas sensors. Typically the correlation between resistance (output) and chemical surface reaction (input) is analyzed with machine learning based

algorithms. With the presented model, the implications on the semiconductor can be predicted and the resistance changes resulting only from the semiconductor can be estimated. Hence the additional changes in the resistance should result from the altered surface chemistry. The numerical model may be one possibility to better understand the correlation and fine tune the operation modes.