Interative thesis combinging analyis, calcualtion and representation

February 7, 2020

Contents

1 Introduction

1.1 Motivation

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

presentations and prepare reports, working in teams and having enough financial support for research activities, it also imposed some new problems needed to be solved. One of the biggest challenges was the continuous increase of sensor date in always shorter intervals. With the increasing industrialization and automatisation of sensor production and testing, the quantity high quality data increased dramatically. Luck as I was, I was not the only one facing problems with the increasing amount of data needed to be analyzed. In the beginning of each analysis, the specific task was not fully defined. It was rather an exploratory research to gain an idea about possible goals. In this case the the traditional way of creating software algorithms for well defined tasks, to take over the heavy lifting, did not fit too well. I rather had the need for efficient tools to reduce time consuming parts of the data analysis and a platform to combine these 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 an efficient solutions with a steep learning curve.

Until then my, and of some others in the lab, standard procedure of working with our 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 gaining speed, 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 the analysis. Luckily I was not the only one facing such problems and a project from the Python community gained more and more interest. Articles like [Unp14] pointed me to a new way of dealing with data and using the Python programming language. Regarding the fact, that Python was already a well established programming language, the introduction of the "IPtyhon notebook ecosystem" was making the use of Python for in an scientific work flow very attractive. As mentioned in this article: [Dav13]

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

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

Besides my work for our industrial partner I also did fundamental research about semiconducting metaloxide gas sensors. Based on great research done before my time on gas sensors, my focus was now on numerical calculations of the gas sensors. Typically a theoretical model of a gas sensor is developed and the predictions are compared with experimental results. When publishing the results the peer review system assures, that the publications are well written, documented the on a solid and proven basis. But when reading such papers, I had the experience, that unfortunately many of them presented good results, but 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. This limitation can be a reason why a direct comparison model with experimental data from other sensors is often not done. 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.

Also with the results I gained while my Phd. thesis I was facing the same problem. Others under-

stood 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. Facing this problem more and more, I decided not to ignore it anymore. I made the choice to try to combine representation, description and algorithm in one unified way. I decided to orient myself back to my "educational roots" and use the powerful ecosystem around the "Jupyter notebooks" to calculate, represent and describe the results for my thesis for this task. This is why my thesis is written with the large focus on suppling a introduction to this excellent toolbox.

On the one side this presented work will allow more people to gain insides in the sensing properties of semiconducting metal oxide sensors (SMOX). I hope that on the other side, this his thesis will also be a useful introduction into Python, specially IPython notebooks, for scientific work.

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

I typically finish my introductions to Python by letting the students run their first commands. Typically for other introductions found around the globe, this is a program which outputs "Hello World.". For Python I prefer to execute some other commands with boils almost down to the philosophical essence on how "instructions" should be.

[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 SMOX based gas sensors

This section list a collection of high quality publications which cover all relevant information about semiconductor based gas sensors. This thesis is structured this way, that I will try to provide the required background in detail at the relevant places. So reading these publications now is not necessary to follow this thesis. This section is therefore a good point to come back if a more detailed view on the subject is desired. Especially when this thesis is used to apply the presented concepts to individual research topics.

When solving numerical problems it typical to introduction some assumptions, which are only valid under specific boundary conditions. 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. With my presented work I do not claim to calculate all aspects of SMOX gas sensors, but present a tool which helps to understand specific aspects. The way of presenting this knowledge should lead/motivate others to adapt the presented work to individual other cases with completely different boundary conditions.

1.3 Jupyter Notebooks

"The Zen of Python" might not always be the primary directive of each developer, but the Python community consists most probably of many people how would consider the latter points as important. So did also the inventors of the IPython and Jupyter. A quick search will reveal multiple sources in the world wide web giving a detailed picture about what Notebooks are and how Jupyter in connected with them. Here I will not try to give an general overview about this tool and rather stick to the phrase "Learning by doing.". By explaining topic related parts this notebooks will guide an interested reader to the point where:

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

is gained.

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

The default format of representing anything in a notebook in 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 plaintext 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. The plain text will reveal the way it was created, and the execution of the cell with `CTRL+ENTER` will reveal its Markdown formatted representation. One other handy feature of the Jupyter ecosystem I use is the ability to transform notebooks into multiple other formats. Just to name some: HTML, WORD (DOC, DOCX), Latex. The tool `nbconvert` is used internally to convert the Markdown formatted representation into other formats. For this thesis the default option "Export as PDF" under the File option generates a Latex based PDF file. Mastering-markdown is a web page, where I found some hints on how to format my notebook. For instance I gained the ability to make block quotes from this page based on this example:

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!

1.3.1 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 is additionally a presentation I use for my lectures to guide students into the world of Python and here one example of it's usage.

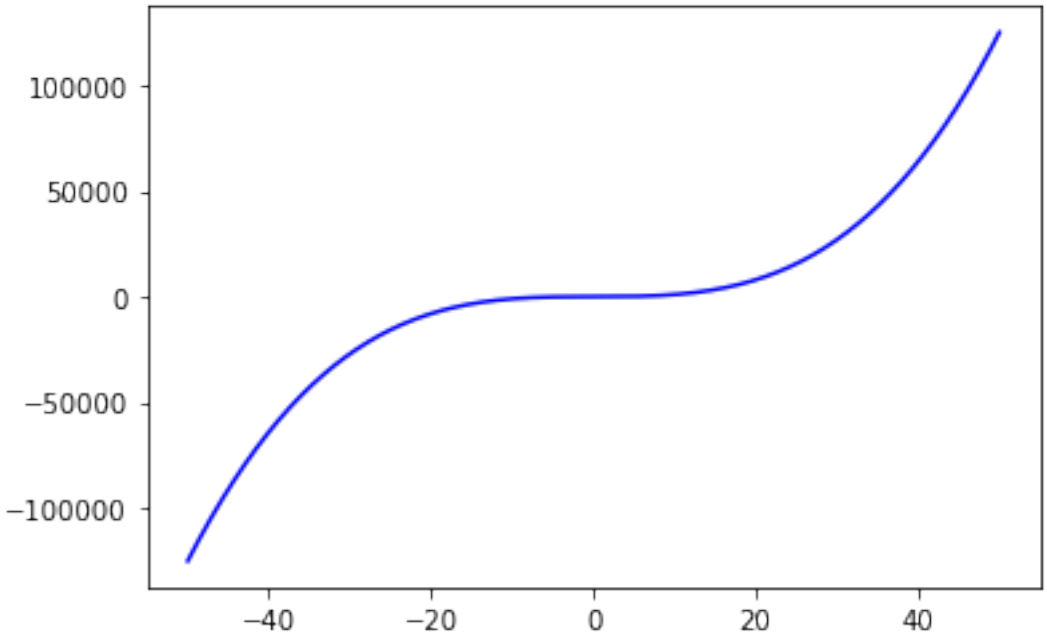
1.3.2 Example Notebook - Sneak preview

Besides the first example `import this`, here is a very basic example which should prepare exited reader on whats coming next.

"Simple is better than complex." The Jupyter environment is equipped with "magic commands", which are not part of the base programming language (i.e. Python), but rather a helper instruction to simplify common tasks. Magic commands always start with `%` and are followed with an instruction. I will demonstrate in this example the use of the `%pylab inline` instruction. This modifies the current programming space to become a lab nicely equipped for scientific work tasks. For instance a chemistry lab is commonly equipped with a balance, a water tab and a fire extinguisher, and in this case a "pylab" is (besindes many others) equipped with a data handing, a plotting and a calculation tool. 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])

[5]:
```
%pylab inline

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

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

#plot it
fig = plt.plot(x,y,'b')
```

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



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

- click on the code box
- edit the field
- 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

2 Conclusion

Since the motivation for this "interactive" thesis should be clear now, I would like to come in the next section now to my actual research topic: "Numerical calculation of semiconducting metal oxide (SMOX) based gas sensors". In the next chapter I will demonstrate how theoretical numerical calculations are used to for chemical sensors are used to better understand experimental results. In the following chapter I will demonstrate how such knowledge could be used to improve the performance of a sensor regarding it's sensitivity and selectivity.

Follow this link to come to the next section.

3 About the PDF-Version of this work

This notebook was not intended to be used as a printed hard copy or as a PDF. The provided PDF servers just as an low level representation of the original work. It should give potentially interested readers an easy entry point to Python (or any other programming language) supported science. Many of the implemented features in these notebooks like interactive widgets, animated data representations and live code examples will not work in the PDF-version. Only a static snapshot of the mutable representation can be represented outside the notebook, at best. The benefits of interacting with the presented work in a notebook should motivate the reader to use the notebook.

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

3.1 Equations

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

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

To refer to this equation a reference can be added which will look like this (Second derivative). The underlying mechanisms will link the reference with the equation, print the label, and add an hyperlink, to be able to 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 will evolving improvements of Jupyter is is just a temporal flaw. 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 eqaution in the next code block.

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

3.2 Tables

With the numerical calculations performed in this thesis, data will need to be represented also. One way of doing this is by plotting the the data. This is also suitable for a static PDF-Version of this thesis, which often is useful/necessary to have. This feature is well implemented and the transfer from Notebook to PDF version works well. Another way to display data are tables. Similar to the well known, omnipresent tool for working with tables, Excel, the Python universe has its own, but similar, tool. It's called Pandas. As a primer I will give here a very short introduction

to Pandas. What sheets are for Excel, are Dataframes for Pandas. Here a simple example how to create a Dataframe in analogy to the previous programming example:

[6]:
```
%pylab
import pandas

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

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

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

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

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

3.2.1 Before the patch

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

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

3.2.2 The patch

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

pandas.set_option('display.notebook_repr_html', True)

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

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

3.2.3 Prettier tables after the patch

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

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

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

4 Bibliography section

References

[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

[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

Numerical calculations of SMOx based gas sensors with Python

February 7, 2020

Contents

1 Abstract

The investigation of thick film semiconducting metaloxide gas sensors (SMOX) offers for a scientist many secrets to reveal. Hidden in the mystery of semiconductors most process are not available by simple measuring techniques. The general procedure of investigating a material follows the following lines: A measuring technique is applied and based on the state-of-the-art understanding of the transduction mechanism the results are interpreted. The risk of getting to wrong

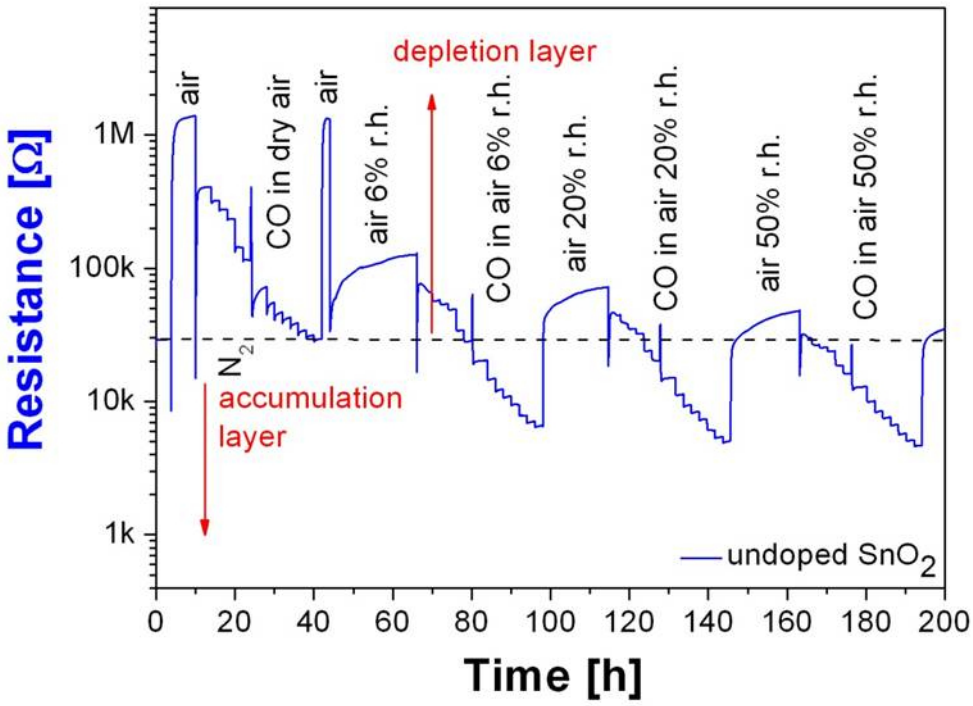
conclusions are fairly high since the amount of assumption in the logical conclusion-path are quit high.

A major key figure measured is often the resistance of the sensor. Therefore one important elements in this logical path of conclusion is how electrical properties of the semiconductor influence the resistance of the sensor. Further more the how (relative) changes of the properties result in changes in the resistance. With this knowledge also other processes related to chemical surface reaction my be understood better.

2 Motivation

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

Nevertheless, the assumption the existence of an depletion layer is not always valid. Recent experimental results have shown, that even under atmospheric conditions which are common in real live, the dominant impact of oxygen may be gone. It could be shown, that under conditions of 50% r.h. and low concentrations of CO (>1 ppm) in synthetic air, the depletion layer was gone and a accumulation layer manifested. The following figure shows the experimental results from such a measurement. In the beginning of the experiment a SnO_2 based gas sensor was exposed to pure nitrogen. For this sensor it can be assumed, that under this condition no band bending is present. This means, that the resistance under nitrogen corresponds to the flat band situation. An increase of the resistance indicates the presence of a depletion layer, where a decrease of the resistance would indicate the presence of an accumulation layer. In the following figure we clearly see, that a conduction band switch is present. A detailed description of the finding can be found here [BRW15]



With the absence of the depletion layer also most simplifications are not valid anymore. Mainly the validity of the Schottky and Boltzmann approximation are not given anymore. Facing those facts the equations to describe the transduction mechanism for a specific electrical band configurations needed a more general descriptions, which includes a depletion and accumulation layer controlled transduction mechanism.

To describe such a semiconductor the set of differential equations is available in literature. Since the previously used simplifications for such problems are not valid in the generalized case, finding an analytically solution exceeded by far my intellectual capabilities.This is why a I chose to go a different. Find a numerical solution for the problem by using computational power

While working in the field of SMOX allready some years, I was used to describe transduction processes by assigning different parts of an analytical solution to properties of the sensor. With a numerical solution this is not possible anymore and this was certainly a drawback of this method. But on the other side the relation between an effect and a intrinsic property can be studied also in detail, when numerically solving the equations for multiple values of the intrinsic property. The influence of may be studied and an insight about the principles can be gained. Therefore the goal was first to break the problem of describing a SMOX sensor into smaller parts discreet parts and second trying to solve each of it individually.

In the upcoming chapters I will describe the different part and how they have been simplified, solved individually and combined again. Each part will also hold a section where 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, different equations, many of which are described in [sec:current models] such as the shape dependent Poisson equation or the combination between the Poisson equation, the electro-neutrality equation and the geometry dependent electrical current path, must be solved. In most cases, this is an extensive mathematical effort and therefore, the numerical computing environments Python will be used to derive numerical solutions for equations, which cannot be solved analytically, as shown below. Depending on the grains size the charge trapping at the surface has different impact on the potential and charge distribution inside the grain. For large grains compared to their Debye length (L_D), a charge transfer at the surface may be leave the bulk unaffected. In contrast to large grains, relative small grains may be affected through the whole grain (Figure [fig:Potential-for-spheriacal,small_1ld]).

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

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

In the simplest case of symmetric contacts on opposite sites, a graphical representation of the results from a numerical simulation of one grain with a small depletion layer is shown in (Figure 3). The resulting total resistance of one grain as a function of the surface potential can then be used as an approximation for the resistance of a full multi-grain - sensor. (Figure [fig:grain_simulation_small_depl]). The relation between surface potential change and resistance change may also be obtained experimentally as described in the previous paragraph with the Kelvin-Probe-Method (REF).By comparing the Kelvin-Probe-Data with the results from the numerical simulation a better understanding of the influence of the grain size on the performance can be gained. The fitting of the numerical parameters of the simulation to experimental data or generally fitting of data with an appropriate function can easily be performed in the presented environment.

3.2 Semiconductor properties of the SMOX grains

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

- upscalable
- highly reproducible
- low cost

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

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

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

3.3 Choice of geometric model

Both favorable preparation methods mentioned in the latter paragraph have been investigated under a REM (Figure [fig:REM_grain_SMOX]). The 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.

Picture of a typical SMOX-Sensor with the grains visible

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

3.4 Poisson's equation

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

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

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

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

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

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

It will be assumed, that the reactions take place a the surface of the crystal and bulk diffusion will be neglected. Even if there are reports of oxygen bulk diffusion for certain materials (Quelle), this is not the general behavior and does not apply apply for SnO_2. Since all surface sites will be accessible by the gas, the solution of [eq:poisson_eq]should have a rotational symmetry. In the case of multiple grains with grain-grain contacts this assumptions needs to be validated again.

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

$(\frac{1}{r})\frac{d}{dr}\frac{rd\phi(r)}{dr} = -\frac{\rho(r)}{\epsilon\epsilon_0}$

Description of the coordinate system of one spherical symmetric SMOX grain....

3.5 Charge density

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

To summarize the findings it can be said, that the resistance drops below the level obtained under inert conditions in nitrogen, which means that the material is getting more conductive that it is initially. The root cause of the higher conductivity is the increased number of free charge carrier in the conduction band. In contrast to a depletion layer controlled transduction mechanism a accumulation layer is formed while charges are pushed into the conduction band.

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

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

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

Then the Fermi-Dirac distibution $f(E)$ can be expressed with the Boltzmann distribution b(E):

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

Based on the findings, that the flat band situation is reached in application relevant conditions, the Boltzmann approximation i not valid anymore. Operando Kelvin Probe (see: [subsec: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 upcomming calculations I will calculate and show relative error, when using the Boltzmann approximation for a relevant range of energy bendings of the conduction band.

[BHW11] and its validity for the case of an accumulation layer will be shown in the upcoming calculations.

To unify all transduction mechanisms into on numerical calculation the Fermi-Dirac distribution, without further simplification, is used to calculate the charge distributions.

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

Ferimi-Dirac:

$$f(E) = \frac{1}{exp(\frac{E-E_{Fermi}}{k_B T}) + 1} = \frac{1}{exp(\frac{E-E_C+E_C-E_{Fermi}}{k_B T}) + 1} = \frac{1}{exp(\frac{E_C-E_{Fermi}}{k_B T}) * exp(\frac{E-E_C}{k_B T}) + 1} \tag{3}$$

The density of states is given by [#Sze2007]for and electron in the conduction band with the effective mass m^{*}as followed: DERIVE NUMERICALLY!

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

With the occupation probability $f(E)$ and the density of states $N(E)$ the number of charges in the conduction band can be calculated:

$$n(E_C) = \int_{E_C}^{inf} N_{E_C}(E) * f(E)\, dE \tag{5}$$

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

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

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

Equation [eq:integral_n] was solved numerically and compared with the results obtained with the common approximations($m_e^* = 0.3m_e$ for SnO_2 [BD05] was chosen .

Previous publications prove, that 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 now the free charge carrier density ρ from Poisson's equation.

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

$$\frac{\rho(E)}{e} = n(E_{C_b}) - n(E) = n_b - n(E) \tag{6}$$

Equation $n(E_C)$ becomes then:

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

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) becomes:

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

The goal will be to calculate how potential at the surface is electrically screen by the remaining positive charges in side the grain. An important parameter for such calculations is the Debye length. In cases where the Boltzmann approximation is valid, the Debye length can be approximated with the following formula. In this case the given length is the distance needed to screen a potential V until its value reaches $\frac{V}{e}$

$$L_D = \sqrt{\frac{\epsilon\epsilon_0 k_B T}{n_b e^2}} \tag{7}$$

With the definition of the Debye length I can now transform the relevant variables of the calculation.

- 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. As I will show in the next part of the thesis, the numerical calculation is also be possible with the initial spherical Poisson equation (Poisson spherical (V)). Therefore the material specific parameters need to be given to the algorithm. The downside would be, that for every new material with any parameter changing, the calculation would need to be redone. The benefit of the latter derived equation is that it is valid for multiple combinations of intrinsic parameters. The solution would only depend on three parameters:

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

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

- Typical grainsizes reach from 0.1 to 100 L_D
- Typical temperatures are in the range of 100°C to 400°C
- The doping level n_b range from $10^{19} \frac{1}{m^3}$ 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] will now be used to numerically solve the derived equations. The odesolvers of scipy solve first order ODEs, or systems of first order ODES. To solve a second order ODE, we must convert it by changes of variables to a system of first order ODES.

Equation (Unitless Poisson equation) is an ODE of second order, so it first had to be converted by changes of variables to a system of first order ODES.

Practically a functions needs to be written, which gets as input an array of values and returns returns an array of the derived values:

EXPLAIN HOW OED SOLVER WORKS

$$derive\_func\left(V^*, \frac{dV^*}{dr^*}\right) \longrightarrow \frac{dV^*}{dr^*}, \frac{dV^{*2}}{dr^{*2}} \tag{8}$$

The 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 (Unitless Poisson equation).

$$\frac{1}{r^{*2}}\frac{d}{dr^*}r^{*2}\frac{dV^*}{dr^*} = 1 - n^*(V^*) = \frac{2r^*}{r^{*2}}\frac{dV^*}{dr^*} + \frac{r^{*2}}{r^{*2}}\frac{dV^*}{dr^{*2}} = \frac{2}{r^*}\frac{dV^*}{dr^*} + \frac{dV^*}{dr^{*2}} = 1 - n^*(V^*) \tag{9}$$

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

The odesolver needs beside the derive-function additional parameters. Namely a set of initial start values for V^* and $\frac{dV^*}{dr^*}$ and boundaries between the solver should calcualte the solution. Since we first want to calculate the shape of the conduction band for different surface potentials V^*, the initial parameter is already defined V^* . Also the boundaries should be the boundaries of the grain, r^* between 0 and the grainsize R^*.

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

If such a function can be defined, the solver is able to solve the system of equation interactively starting from a given initial condition.

3.7 Constants

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

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

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

[2]:
```
from scipy import constants as sciConst
class Constant:
    def __init__(self):
        self.K0 = sciConst.convert_temperature(0,'C','K')
        self.kB=sciConst.k
        self.EPSILON_0 = sciConst.epsilon_0
        self.E_CHARGE = sciConst.elementary_charge
        self.h = sciConst.h
        self.MASS_E = sciConst.electron_mass
        self.NA= sciConst.N_A
        self.VOL_mol = 22.4
        self.mole_per_l = self.NA/self.VOL_mol

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

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

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

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


CONST = Constant()
```

Once the basic constants are defined, creating also a simplified numerical representation of the investigated semiconducting material. The specific properties of the semiconductor will result in individual properties as the charge distribution. As explained in the theoretical section, the charge distribution n, depends on the position of the conduction band. Again start with setting up a numerical python lab, which will output all results "inline" with this document. As shown in the introduction this is done by using the magic command `%pylab inline`

[3]:
```
%pylab inline
```

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

3.8 Materials

3.8.1 Solving integrals numerically

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

From my experience, many students and researches excellently trained to define the set of equations describing their problems mathematically. Seconds, the evaluation of the individual equations with multiple variables imposes (in most cases) no problem. 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.

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 (I continuously fail at solving integral) and the simplifications often reduce the solution to specific boundary conditions (as for example the Boltzmann approximation).

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

So if the elements to be integrated can evaluated for each point between the boundaries of the integral, not much is in the way to solve the integral numerically. Here a simple example of solving: $\int_3^5 x^5 dx$

The analytical solution solution is: $\left[1/6x^6\right]_3^5 = 1/6 * 5^6 - 1/6 * 3^6 \backsimeq 2482.67$

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

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

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

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

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

[4]:
```
from scipy.integrate import quad
```

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

num_sol, num_error = quad(f,3,5)
ana_sol = 1/6*(5**6-3**6)

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

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

In this case, the numerical and the analytical solution result in the same results. How about a more complex problem? Let's look at the "Normal distribution":

$$f(x) = \frac{1}{\sqrt{2\Pi}} * e^{-\frac{x}{2}} \tag{10}$$

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

$$\int_{-\infty}^{\infty} f(x)dx = 1 \tag{11}$$

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

[5]:
```
def f(x):
    return 1/(2*pi)**0.5*e**(-(x**2/2))

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

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

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

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

Also here a numerical solution is in line with the expected result from the analytic solution.

There 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 quad function does not only give back the numerical solution, but also an estimate of the absolute error in the result.

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

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

3.8.2 Numerical description of the semiconductor

Helper functions for semiconductor calculations Michael Hübner (page 50) derived in his thesis a way to approximate the energetic difference between the the level of the Fermi-Level and the conduction band (in a flat band situation) depending of the temperature T and concentration of defects in the bulk of the grain n_b and the effective mass of electrons inside the semiconductor m_e^*. Both m_e^* and n_b are difficult to measure experimentally. This thesis will initially start with reasonable values found in literature to derive the required set of equations describing the transduction.

For a the later deep analysis those two parameters part of a set of parameters to be screen for the analysis of their influence on the overall performance of a sensing material.

Based on Michael Hübners work now possible to derive a value for the energetic distance between Fermi level and conduction band $\Delta E_F E_C$.

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

[6]:
```
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 Mich Thesis Seite 50

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


    kT = calc_kT(T_C)

    NC = calc_eff_density_of_states(T_C,mass_e_eff_factor)

    ED1C_eV = 0.034
    ED2C_eV = 0.140

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

    poly_params = (c,t1, t2, t3)


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

```
T_C = 300
ND = 9e21
mass_e_eff_factor =0.3

EDCF_eV = calc_EDCF_by_temp(T_C, ND, mass_e_eff_factor)
```

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

From literature the definition of the charge carrier density is known. It is the integral of the product of density of states g_C in the conduction band and occupation probability f_F. Typically the integral is solved analytically by introducing some simplifications. This is possible in cases, where the Boltzmann approximation is valid. In such cases the integrand can be simplified so that the integral can be solved. By additionally solving the integral numerically the solutions can be compared. As shown earlier in this thesis, the numerical solution of the integral by using the `quad` function is easy achieve.

1.2 The n_0 and p_0 Equations

The n_0 and p_0 Equations

We will assume initially that the Fermi level remains within the bandgap energy.

The equation for the thermal-equilibrium concentration of electrons may be found by integrating Equation 1 over the conduction band energy, or

$$n_0 = \int g_c(E) f_F(E) dE \tag{3}$$

The lower limit of integration is E_C and the upper limit of integration should be the top of the allowed conduction band energy.

However, since the Fermi probability function rapidly approaches zero with increasing energy as indicated in Figure1a, we can take the upper limit of integration to be infinity.

We are assuming that the Fermi energy is within the forbidden-energy bandgap.

For electrons in the conduction band, we have $E \gg E_c$.

If $(E_c - E_F) \gg kT$ then $(E - E_F) \gg kT$, so that the Fermi probability function reduces to the Boltzmann approximation, which is

$$f_F(E) = \frac{1}{1 + \exp \dfrac{E - E_F}{kT}} \approx \exp \frac{-(E - E_F)}{kT} \tag{4}$$

Applying the Boltzmann approximation to Equation (3), the thermal-equilibrium density of electrons in the conduction band is found from

$$n_0 = \int_{E_c}^{\infty} \frac{4\pi (2m_n^*)^{3/2}}{h^3} \sqrt{E - E_c} \exp\left[\frac{-(E - E_F)}{kT}\right] dE \tag{5}$$

The integral of Equation (5) may be solved more easily by making a change of variable :

$$\eta = \frac{E - E_c}{kT} \tag{6}$$

Then Equation (5) becomes

$$n_0 = \frac{4\pi (2m_n^* kT)^{3/2}}{h^3} \exp\left[\frac{-(E_c - E_F)}{kT}\right] \int_0^{\infty} \eta^{1/2} \exp(-\eta) d\eta \tag{7}$$

The integral is the gamma function, with a value of

$$\int_0^{\infty} \eta^{1/2} \exp(-\eta) d\eta = \frac{1}{2}\sqrt{\eta} \tag{8}$$

Then Equation (7) becomes

$$n_0 = 2\left(\frac{2\pi m_n^* kT}{h^2}\right)^{3/2} \exp\left[\frac{-(E_c - E_F)}{kT}\right] \tag{9}$$

We may define a parameter N_c as

$$N_c = 2\left(\frac{2\pi m_n^* kT}{h^2}\right)^{3/2} \tag{10}$$

The value of n_0

Thermal-equilibrium electron concentration in the conduction band

$$n_0 = N_c \exp\left[\frac{-(E_c - E_F)}{kT}\right] \tag{11}$$

The parameter N_c is called the **effective density of states function in the conduction band**.

If $m_n^* = m_0$ then the value of the effective density of states function at $T = 300$ K is $N_c = 2.5 \times 10^{19}$ cm^{-3}, which is the order of magnitude of N_c for most semiconductors.

If the effective mass of the electron is larger or smaller than m_0 then the value of the effective density of states function changes accordingly, but is still of the same order of magnitude.

How to derive the charge density (PDF copy, source unknown, sorry)

[1211]:
```
from scipy.integrate import quad
from scipy.interpolate import interp1d
import scipy
from functools import lru_cache
import numpy as np
import pandas as pd



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


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

        self.nb, self.nb_err = self.n(0)
        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):
        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 = 2.14e24
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} eV')
material = Material(T_C,ND, DIFF_EF_EC_evolt=EDCF_eV)
```

```
For SnO2 at 300°C with a defect concentration of 2.14e+24 1/m³, the value of
EDCF_eV is 0.04739832440692029 eV
```

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

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

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

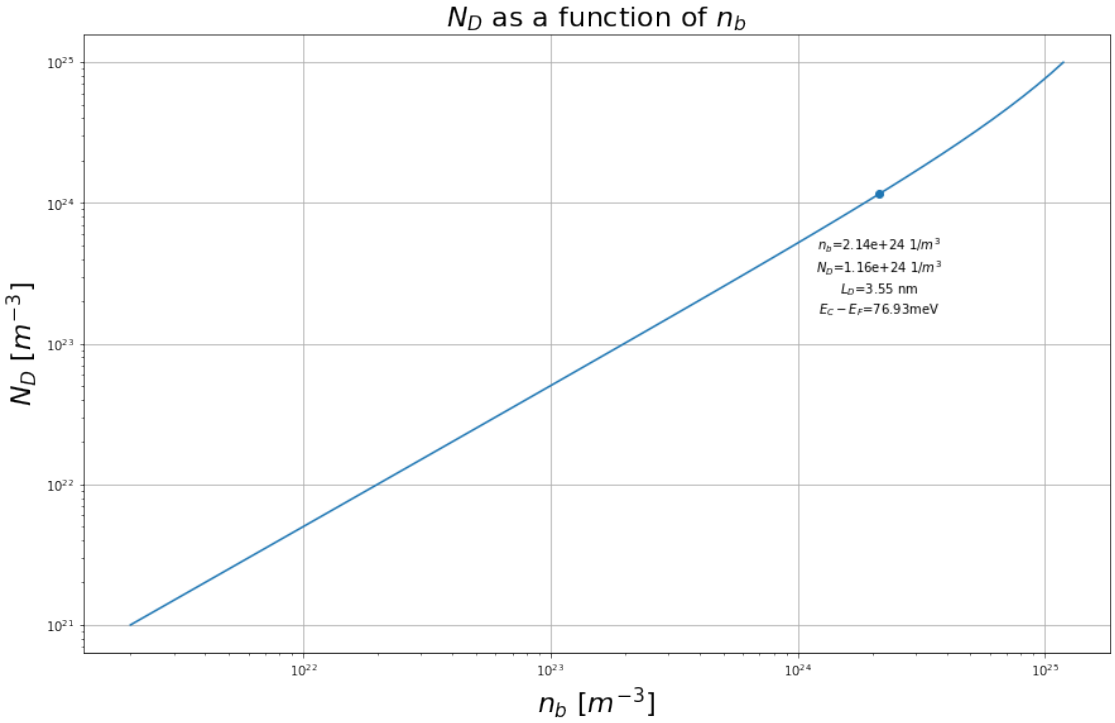
function says:

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

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

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

[1212]:
```
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' )

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

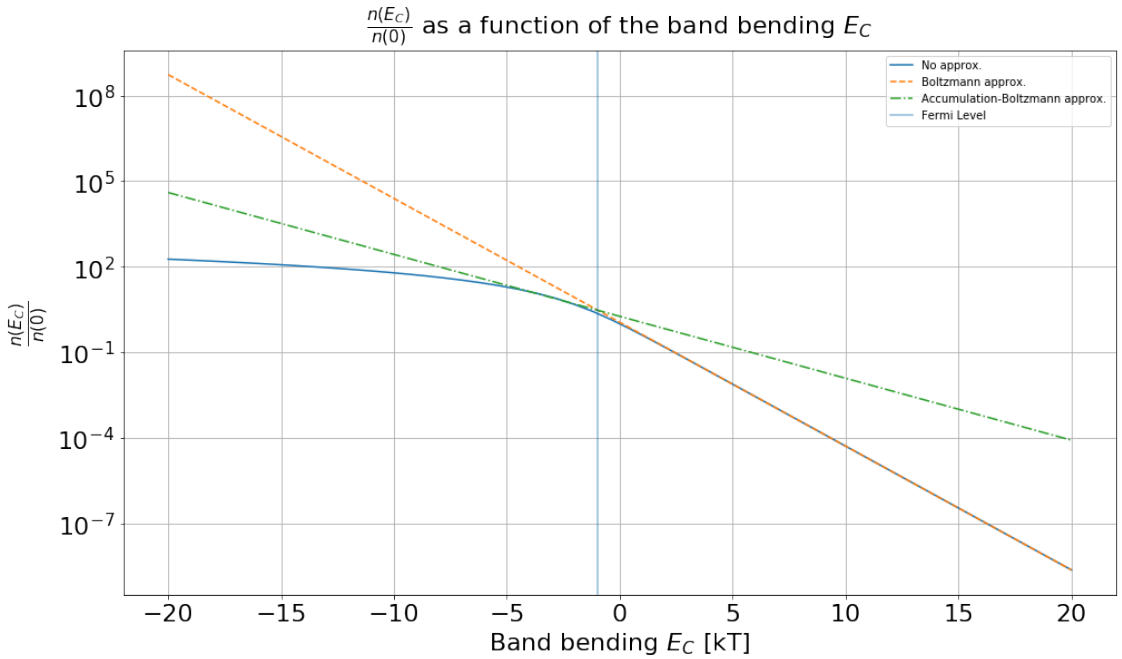


Free charge carrier conc. using the Boltzmann approximation Besides the full numerical solution, also the solutions derived from the Boltzmann approximations need to be defined. As described in [REFERENCE] this breaks down to the following functions:

[1213]:
```
def boltzmann_acc(material, E_c):
    return np.exp(-(E_c+material.Diff_EF_EC)/(material.kT*2))

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

def densitiy_of_states(self,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(self,E_c):

    return boltzmann(material,E_c)*material.NC

def n_boltzmann_acc(self,E_c):

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

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

[1214]:
```
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(T_C,ND)
fig = plot_material_char(material)
```



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

3.8.3 Numerical description of the semiconductor grains

In this section we define a SMOX grain. We approximate the grain as a sphere composed out of a material we previously defined. For one grain, the Poisson equation with spherical symmetry is solved. The transfer of the findings from a material to an actual grain are important for multiple reasons. On the one side the ratio of the available surface sites to react with the semiconductor and its bulk size play an important role. Very small grains may have relatively high concentration of surface sites but lack of electrons needed for the reaction a t the surface. So a grain may get fully depleted which may have significant an influence on the overall conduction. On the other side, the conduction path through the grain differers depending on the free charge carrier concentration. Those two relevant properties can only be analyzed, if the transfer from a material to an actual grain is solved.

To solve the Poisson equation, we will need to supply the solver with the initial values. Tow values need to be supplies. The goal of the solver is to find the shape of the conduction band inside the grain. This depends on the position at the surface. 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 curve at the surface. With these two parameters, the solver iterates from the starting condition stepwise though the grain and calculates for each step new values based on the previous iteration.

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

[1232]:
```
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 = self.R*(1.0-np.logspace(0,3,num=rPoints)/1e3) #calcualtion␣
↪points from surface to center (not lnear spaced)
        self.rs = np.linspace(self.R/1000, self.R, rPoints)


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

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

            if y[1]:
                if abs(y[1])<0.0001:
                    return 0
            return y[1]
        hit_ground.terminal = True

        #this is the solver
        #data = solve_ivp(self.deriv_E_E_dot,(r_LD[-1],r_LD[0]), ␣
↪[E_init_kT,E_dot_init_kt], t_eval=r_LD[::-1], events=hit_ground, method =␣
↪'Radau',max_step=max(r_LD/1000))
        #see the docstring why I chose the metohd BDF
        data = solve_ivp(self.deriv_E_E_dot,(r_LD[-1],r_LD[0]), ␣
↪[E_init_kT,E_dot_init_kt], t_eval=r_LD[::-1], events=hit_ground, method =␣
↪'BDF')
```

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

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

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



        return r,v, v_dot, data


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

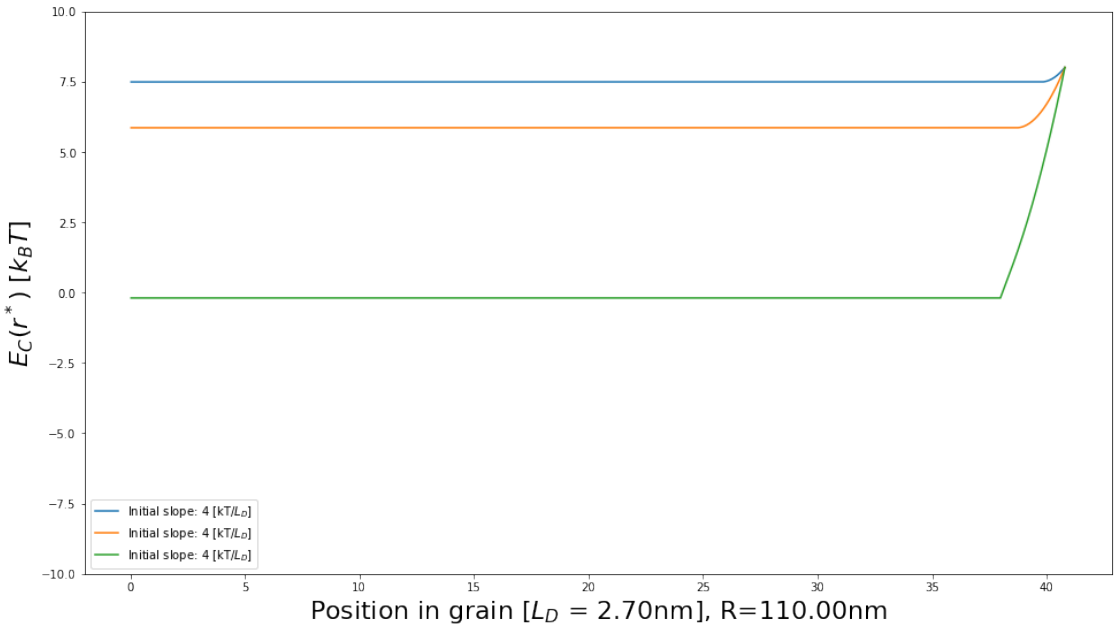
[1233]:
```
#define a grain with a specific material
def create_grain(grainsize, T_C, ND):
    mass_e_eff_factor =0.3

    material = Material(T_C,ND)
    grain = Grain(grainsize_radius=grainsize,material=material)
    return grain

#Check the influence of LD

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


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

```
2.5852950260293164
45.47592892276805
```

In the following graph an example is given, where the initial value of V^* is fixed, an different values of the surface slope are used to solve the equation.

In the previous cells we already initialized the `material` and `grain` object with reasonable parameters. First we initialize the grain with a material and calculate the solutions of the Poisson equation for multiple start parameters.

[1234]:
```
T_C = 300
ND = 2.14e24
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=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 result.

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

$$\frac{dV^*}{dr^{*2}} = 1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*} \tag{12}$$

This equation can be transformed into following form:

$$\int_0^R \frac{dV^*}{dr^{*2}} dr^* = \left[\frac{dV^*}{dr^*}\right]_0^R = \left.\frac{dV^*}{dr^*}\right|_R = \int_0^R 1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*} dr^* \qquad \text{(Surface slope)}$$

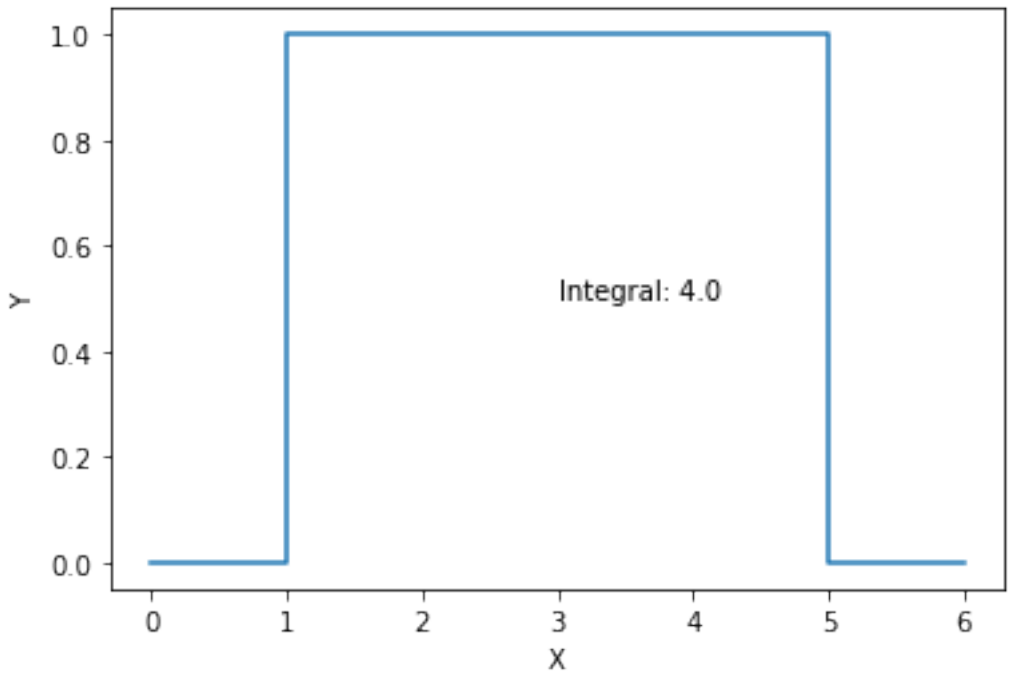
With this relation we can check each solution. For each starting condition the solution should should also be valid for this equation. The right side will again be evaluated numerically. The expression $n^*(V^*)$ can be calculated for each V^* with the function defined in `class material`. From the solver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

> 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`

[1235]:
```
x = [0,1,1,2,3,4,5,5,6]
y = [0,0,1,1,1,1,1,0,0]
fig, axe = subplots()
axe.plot(x,y)
axe.set_xlabel('X')
axe.set_ylabel('Y')
numerical_integral = np.trapz(y,x)
axe.text(3, 0.5, f'Integral: {numerical_integral}')
```

[1235]:
```
Text(3, 0.5, 'Integral: 4.0')
```



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

$$\left.\frac{dV^*}{dr^*}\right|_{R^*} - \int_0^{R^*} \left(1 - n^*(V^*) - \frac{2}{r^*}\frac{dV^*}{dr^*}\right) dr^* = 0. \tag{13}$$

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

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

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

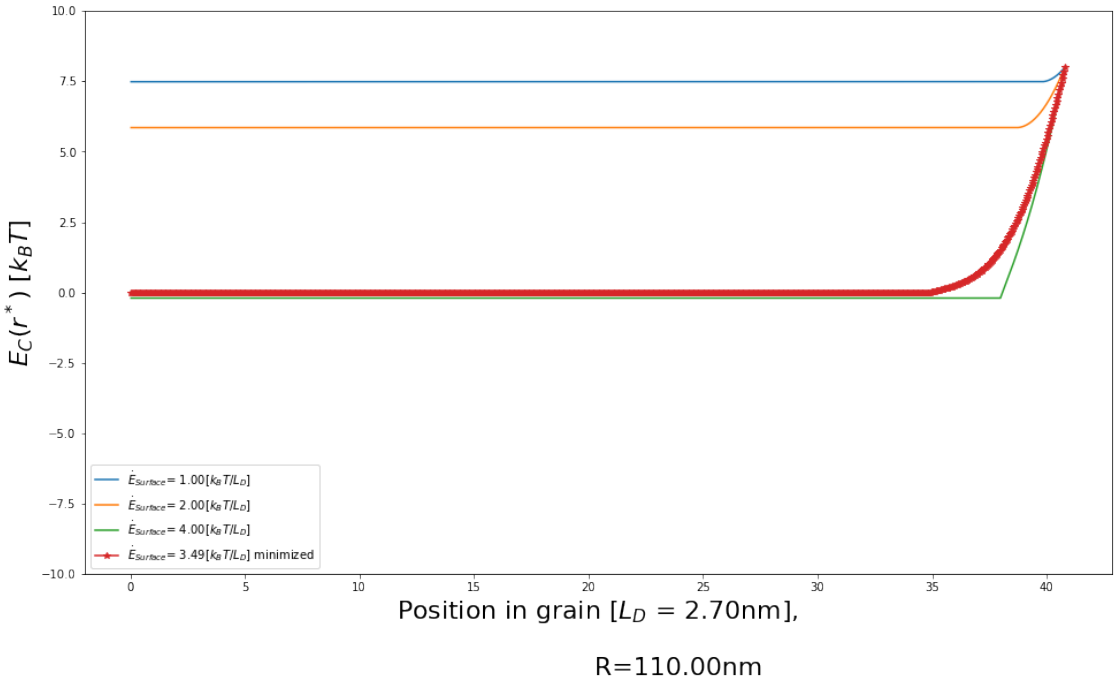
[1342]:
```
from scipy.optimize import minimize_scalar,bracket


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 differenc␣
↪is the error to be minimized
    res = abs((dV-vdot[-1]))

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

def find_best_E_dot_init(E_init_kT, grain,debug = False, bounds = None):
    #if bounds are given as a hint for the minimize algorithm, then the␣
↪method='Bounded' can be used
    #In this case the algorithm will search in the biven interval
    if bounds:
        res = minimize_scalar(min_vdot,args=(grain, E_init_kT, debug), method =␣
↪'Bounded', bounds = bounds)
    else:
        res = minimize_scalar(min_vdot,args=(grain, E_init_kT, debug))
    #res = bracket(min_vdot,xa=bounds[1], xb=bounds[0],args=(grain, E_init_kT,␣
↪debug))
    return res
```

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

[1345]:
```
#grain = Grain(100e-9,material)


T_C = 300
ND = 2.14e24
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)
res
E_dot_init_kT = res.x
E_dot_init = grain.material.kT_to_J(E_dot_init_kT)
r,v, v_dot, data = grain.solve_with_values(E_init, E_dot_init)
axe.plot(r,v, '-*', label='$\dotE_{Surface}$'+ f'= {E_dot_init_kT:.2f}[$k_BT$/
 ↪$L_D$] minimized')

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



3.8.4 Additional relevant paramters

From the above solution some additional results may be calcuated with too much of computational effort. For instance the total number of free charge carries inside the grain. From the shape charge distribution inside the grain, a simple integral of the values over the volume of the sphere will reveal this value. The difference of this value to the value in the flat band situation should then be the number of charges (electrons) trapped at the surface $N^{-}_{Surface}$. The projection of the concentration of donors N_D inside the grain onto the surface

[1353]:
```
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)

    ser['all_c_at_flatband'] = all_c_at_flatband
    ser['charges_trapped_at_surface'] = charges_at_surface
    ser['surface_vacancies_projection'] = grain.material.ND*((grain.R+0.
↪1*1e-9)**3 - grain.R**3)*4/3*pi




    #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'] = -grain.material.J_to_kT(grain.material.Diff_EF_EC)

    return ser
```

```
calc_charges = calc_sum_of_charges(grain, r, v)
pd.Series(calc_charges)
```

[1353]:
```
R
1.1e-07
grain_vol
5.57528e-21
grain_surface
1.52053e-13
n                                  [3.696182155339544e+24,
3.696182155339544e+24,...
all_c_at_flatband
20631.2
charges_trapped_at_surface
5093.45
surface_vacancies_projection
32.569
temp
300
mass_eff
2.73282e-31
ND
2.14e+24
EPSILON
9.86
nb
3.70048e+24
E_Fermi
-0.95967
dtype: object
```

3.8.5 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. For the beginning I will define one material and calculate the resulting conduction band bending for multiple surface potential. Additionally I would like to investigate the influence of different grain sizes on the sensing performance.

Those results will lead in a second step to an understanding of the relation between surface reaction, resistance change and grain size. But for now lets concentrate on generating data for further analysis.

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

import pandas and creates a `Dataframe`, where all our results will be stored.

[1354]:
```
import pandas as pd
#The results will be saved in a DataFrame
dF_calc = pd.DataFrame()
```

Next, we need to create the numerical representation of the material.

[1438]:
```
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()
    ser_temp['Einit_kT'] = E_init_kT
    ser_temp['E_dot_init_kT'] = res.x
    ser_temp['res'] = res.fun


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

    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(-10,11))):
        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
```

[1418]:
```
# create a list of the tuples with (T_C, ND, Grainradius) to be alter␣
 ↪caluculated in parallel
params = []
```

```
T_C = 300
ND_ref = 1e21
for ND in [ND_ref, ND_ref*1e1, ND_ref*1e2, ND_ref*1e3]:
    for grainsize in [ 6.25e-9, 12.5e-9, 25e-9,50e-9,100e-9]:
        params.append((T_C, ND, grainsize))

print(params)

#using multiple processors of the system to calculate the solutions in parallel.
#All solution is are returned in a list, which needs then to be combined again␣
 ↪(see next cell)
from multiprocessing import Pool
with Pool(8) as p:
    all_res_list = p.starmap(calc_solution_by_parameters, params)
    pass
```

```
[(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)]
```

[1419]:
```
dF_calc = pd.concat(all_res_list)
dF_calc.index = range(len(dF_calc))
```

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

[1420]:
```
dF_calc.to_hdf('results.h5', 'raw', mode='w')
```

```
/usr/lib/python3.8/site-packages/pandas/core/generic.py:2530:
PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed,key->block1_values] [items->['n',
'r', 'v', 'v_dot']]

  pytables.to_hdf(path_or_buf, key, self, **kwargs)
```

[1421]:
```
calc_dF_all = pd.read_hdf('results.h5', 'raw')
```

[1422]:
```
def create_grain_from_data(dF):
    if type(dF)==pd.Series:
        dF = pd.DataFrame([dF])
```

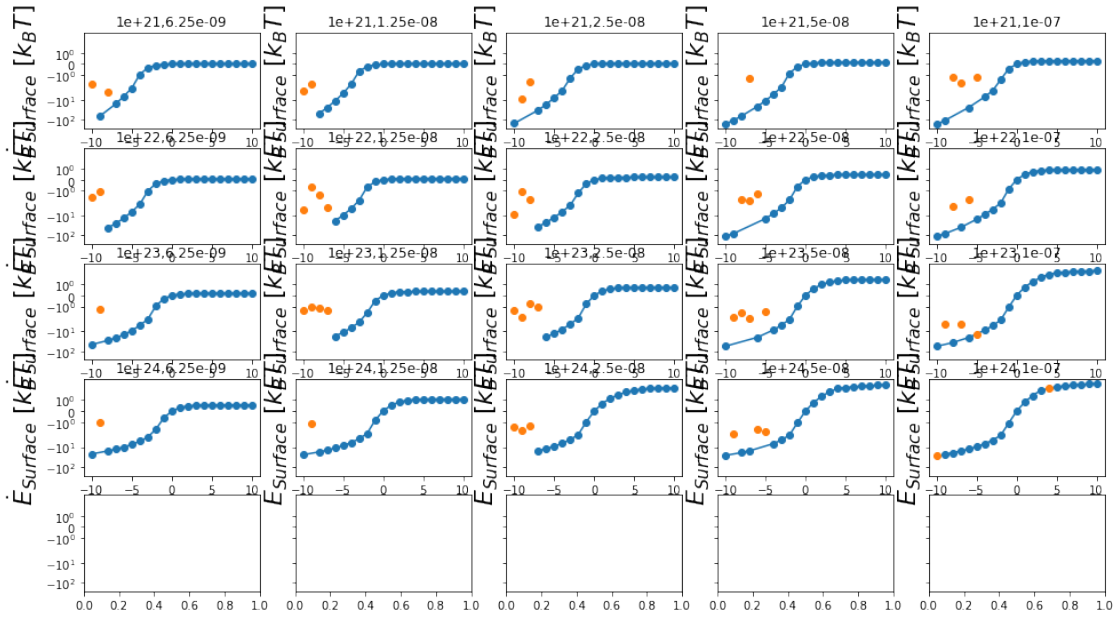
```
    if len(dF['temp'].unique())==1:
        T_C = dF['temp'].unique()[0]
    else:
        raise Exception('Multiple paramters for one grain are invalid.')

    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
```

3.8.6 Estimate correct solutions of incorrectly found minima

Sometimes the function to minimize the function does not work was expected. In most cases the searching algorithm may find a local minimum and miss the global minimum. Generally this problem could be solved by suppling additional hints to the minimization algorithm, where to look for the minimum. To do so we could find the the points, where the minimization result is too high. 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 the 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 result from the minimalization, we can distinguish good and bad results easily.

[1423]:
```
gs = calc_dF_all.groupby(['ND','R'])
fig, axe = subplots(len(gs)//5+1,5, figsize=(16,9), sharey=True)

calc_dF_all['E_dot_init_kT_estimation'] = None
for ax_i, ((ND,R), g) in enumerate(gs):
```

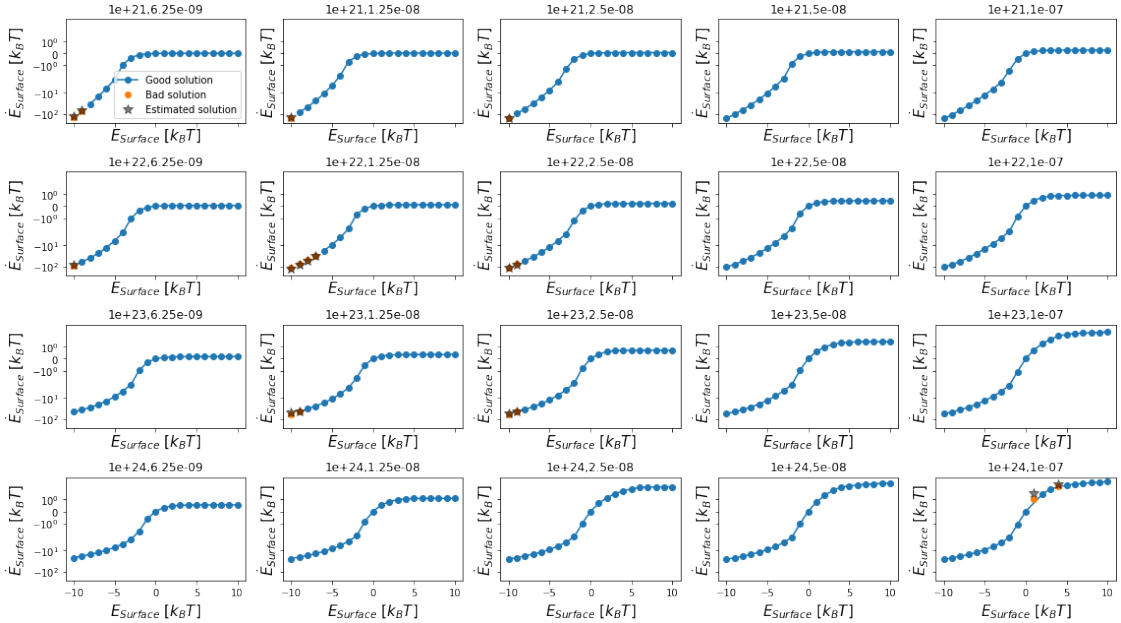
```
    #select the axe
    axe = fig.axes[ax_i]

    #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-')
    axe.plot(g_bad['Einit_kT'], g_bad['E_dot_init_kT'],'o')

    #some plotting sugar
    axe.set_yscale('symlog')
    axe.set_title(f'{ND},{R}')
    axe.set_xlabel(r'$E_{Surface}$ [$k_BT$]', fontsize =22)
    axe.set_ylabel(r'$\dot{E}_{Surface}$ [$k_BT$]', fontsize =22)
```



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 might should lye. As humans can, so does Python. For this problem a simple polynomial fit of the good points is enough. Once the parameters of the polynom are fitted, a estimation of the correct solution of the failed points can be calculated. 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 hold-

ing 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 alter.

[1453]:
```
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']<1]
        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'{ND},{R}')
        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']
        g_fit_params = np.polyfit(x,y,5, w=w)
        fit_func = np.poly1d(g_fit_params)
        g_bad_correct_y  = fit_func(g_bad['Einit_kT'])
        axe.plot(g_bad['Einit_kT'], g_bad_correct_y, '*k', markersize=10,␣
↪alpha=0.5, label='Estimated solution')
        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)
```



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

3.8.7 Recalculating the failed solutions with a little help ...

[1439]:
```
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 polynomial fit
    estim = ser_temp['E_dot_init_kT_estimation']
    #Since the correct solution sould be in proximity of this solution, an␣
 ↪interval is created from this value (here +-10%)
    bounds = sorted((estim*0.9, estim*1.1))
    ser_new = solve_grain_for_E_init_kT(E_init_kT,grain, debug = False, bounds =␣
 ↪bounds)
    ser_new.name = ser_temp.name
    ser_temp.update(ser_new)
    return ser_temp

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

[1448]:
```
index_to_recalc = list(calc_dF_all[calc_dF_all['res']>1].
 ↪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)
```

[1443]:
```
for s in new_sers:
    calc_dF_all.loc[s.name] = s
```

[1444]:
```
calc_dF_all.to_hdf('results.h5', 'corr', mode='a')
```

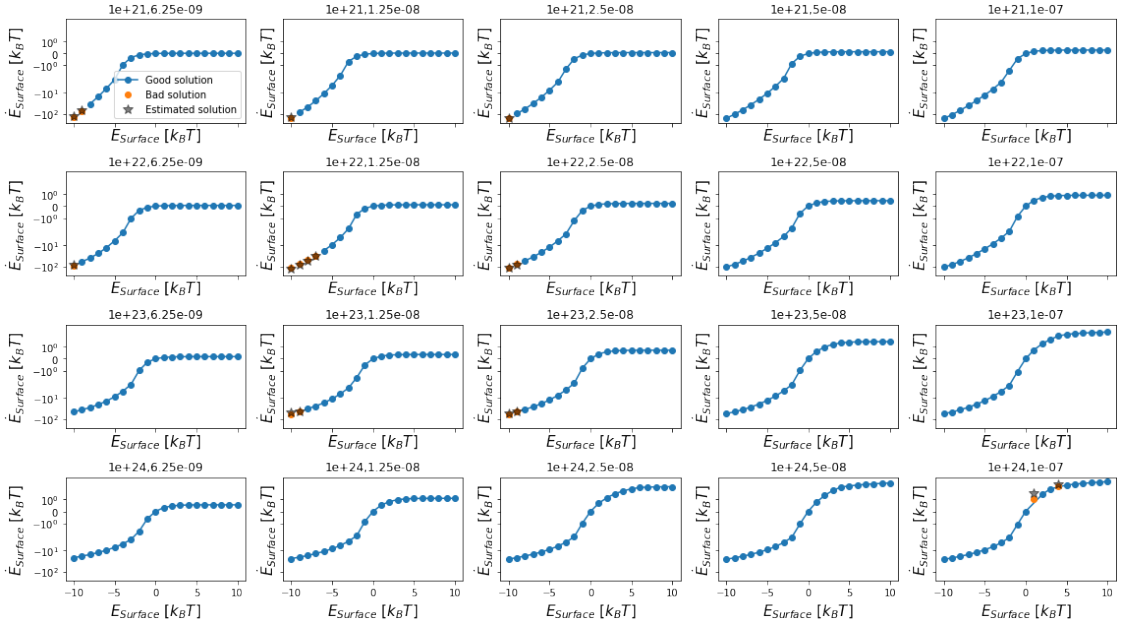
```
/usr/lib/python3.8/site-packages/pandas/core/generic.py:2530:
PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed,key->block1_values] [items->['n',
'r', 'v', 'v_dot', 'E_dot_init_kT_estimation']]

  pytables.to_hdf(path_or_buf, key, self, **kwargs)
```

3.8.8 Checking the solutions

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

[1454]:
```
calc_dF_all = pd.read_hdf('results.h5', 'corr')
print(len(calc_dF_all[calc_dF_all['res']>1]))
check_solutions(calc_dF_all)
print(f'The maximal "error" of the minimization is :{calc_dF_all["res"].max()}')
```

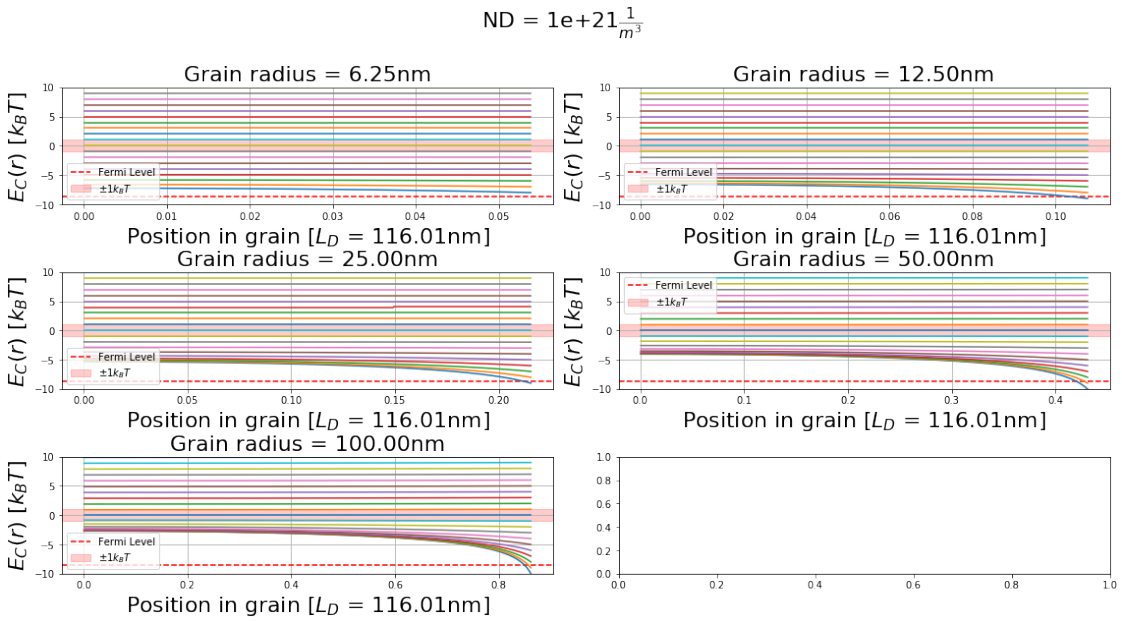
```
17
The maximal "error" of the minimization is :14990.140443668204
```



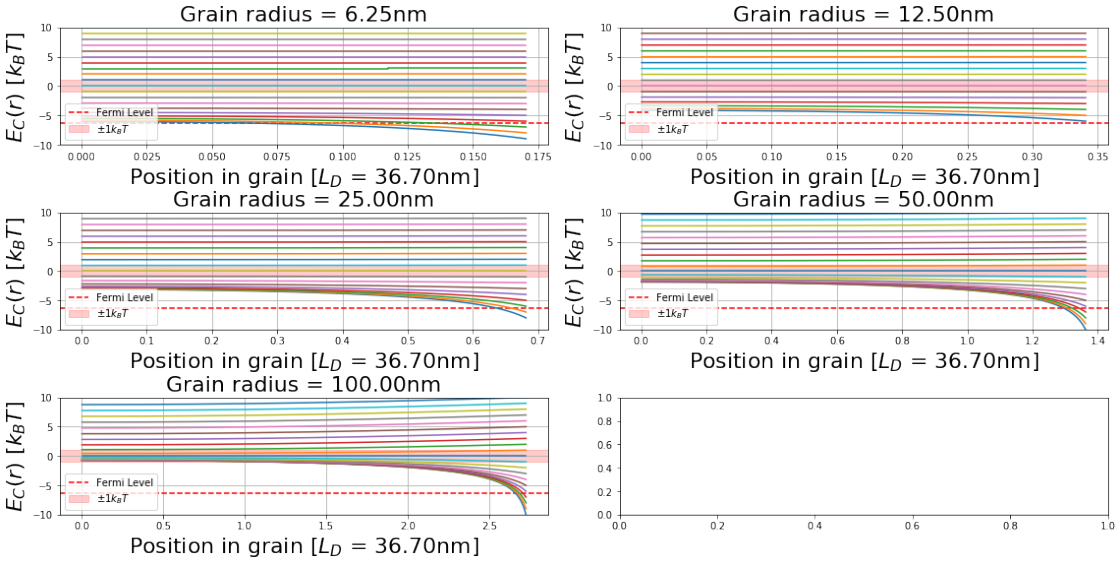
3.9 Shape of the potential drop inside the grain

[1457]:
```
for ND, calc_dF in calc_dF_all.groupby('ND'):
    fig, axes= subplots(3,2,figsize = (16,9), sharex=False)
    fig.suptitle(f'ND = {ND}'+ r'$\frac{1}{m^3}$', fontsize = 22)
    #fig.suptitle(f'$\\num{{{2e-10}}}$', fontsize = 22)

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

        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.set_ylim(-10,10)
        axe.axhspan(-1,+1,color='r', alpha=0.2, label='$\pm 1 k_BT$')

        for vinit, ser_temp in calc_dF_grainsize.iterrows():
            if ser_temp['res']>5:
                continue

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

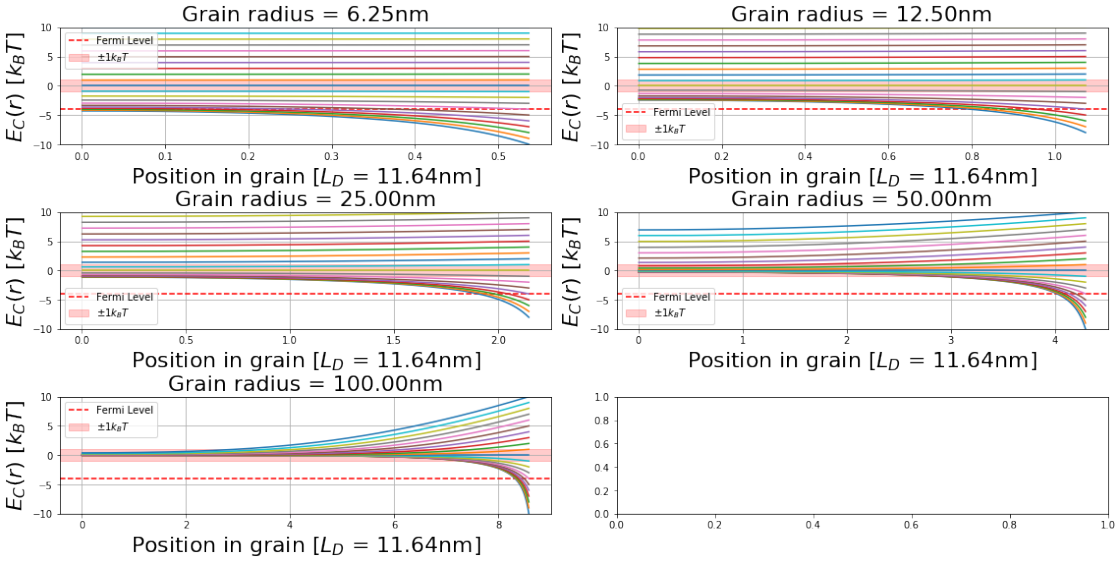
```
            axe.set_title(f'Grain radius = {grain.R*1e9:.2f}nm', fontsize=22)

            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)
        #axe.set_xlim(0,40)
    fig.tight_layout()
    fig.subplots_adjust(top=.85)
    close()
    display(fig)
    for i in range(5):print()
```

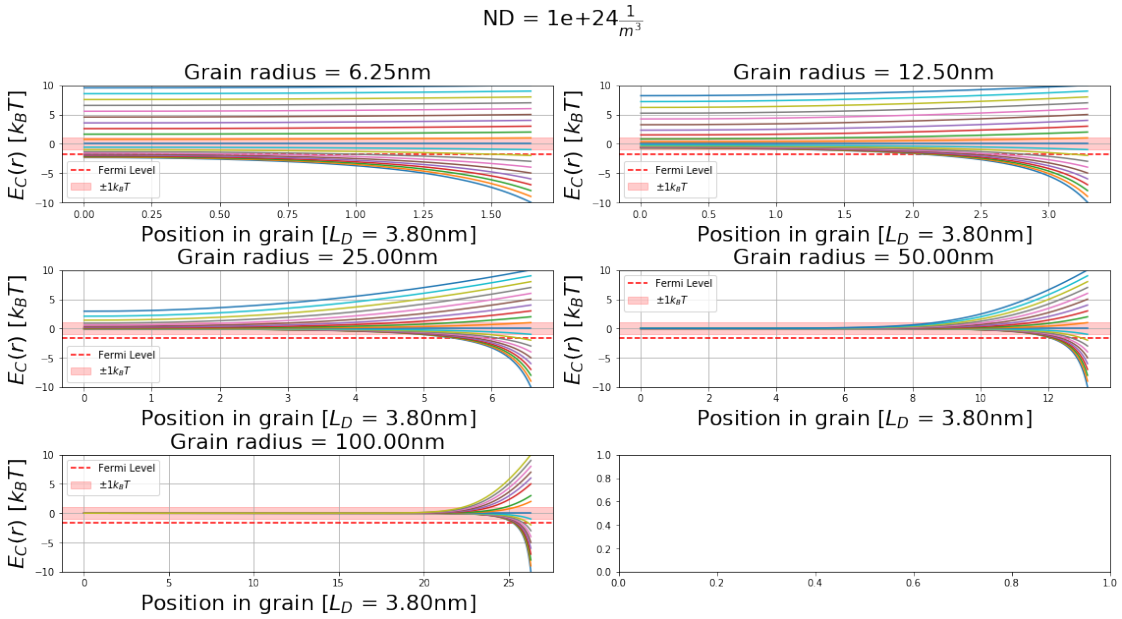


ND = 1e+22 $\frac{1}{m^3}$



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





4 Summary

In this notebook the flowowing steps have been accomplished:

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

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

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

In the next notebook this calculated date will be used derive the total resistance of a grain under different conditions. The anisotropic charge carrier distribution inside the grain has a mayor in-

fluence 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.

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.

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

[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

From SMOx-grains to resistance

February 7, 2020

Contents

1 Review

In the last notebook the semiconductor part of the SMOX grains was addressed. This included the numerical calculation the charge carrier density as as function of the conduction band bending. Additionally the Poisson equation for spherical grains was solved. The grain results are saved to file and can now be here used again. Additionally a Python module was created. In this module all the parts we need to recycle from the previous notebook are merged together. By merging the relevant classes and functions into on python file, the command: `from part2 import *` will execute all the commands in this file and add them to the main namespace. By this the classes `material` and `grain` will again be available for further evaluations.

[2]:
```
%pylab inline
```

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

[3]:
```
from part2 import *
```

2 Load the results

[4]:
```
calc_dF = pd.read_hdf('results.h5', 'corr')
calc_dF.index = range(len(calc_dF))
calc_dF = calc_dF[calc_dF['res']<4]
```

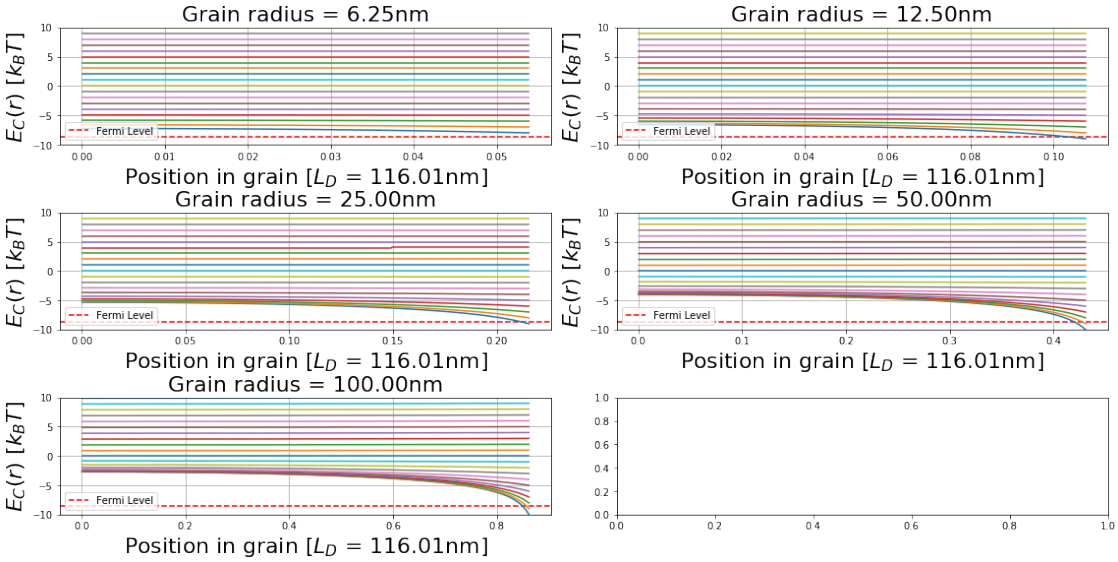
In the following code block, we define a function, that will initialize a new `grain` object from the saved data from the previous notebook. All required data is given in the imported table. The helper function will initialize the `grain` with the corresponding `material` and return it.

[5]:
```
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
```

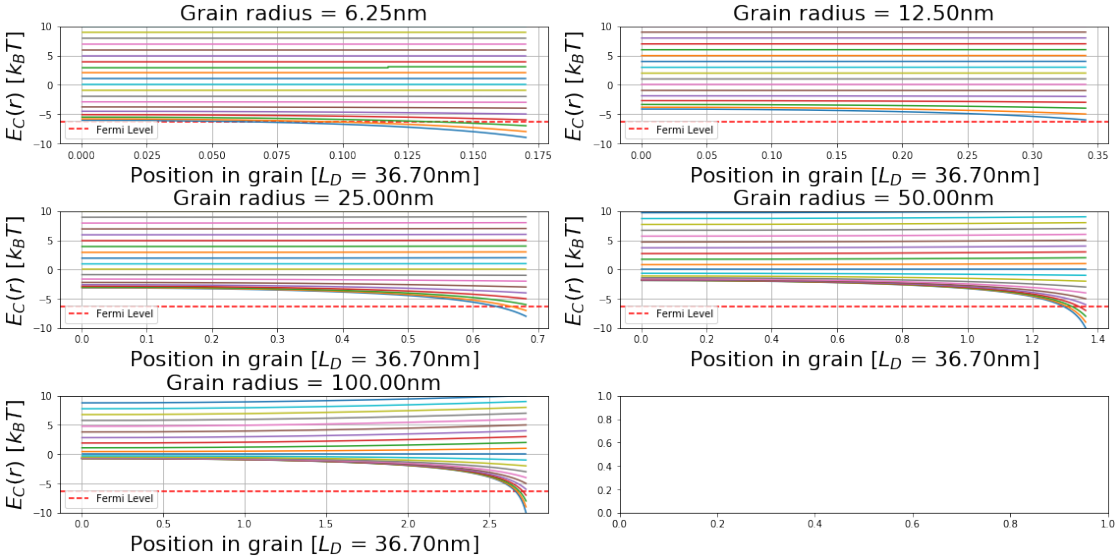
With the helper function initializing a grain from a saved dataset, we can again represent the results from the previous notebook. Such a representation will be helpful for a better understanding of the needed steps to calculate the total resistance of a grain.

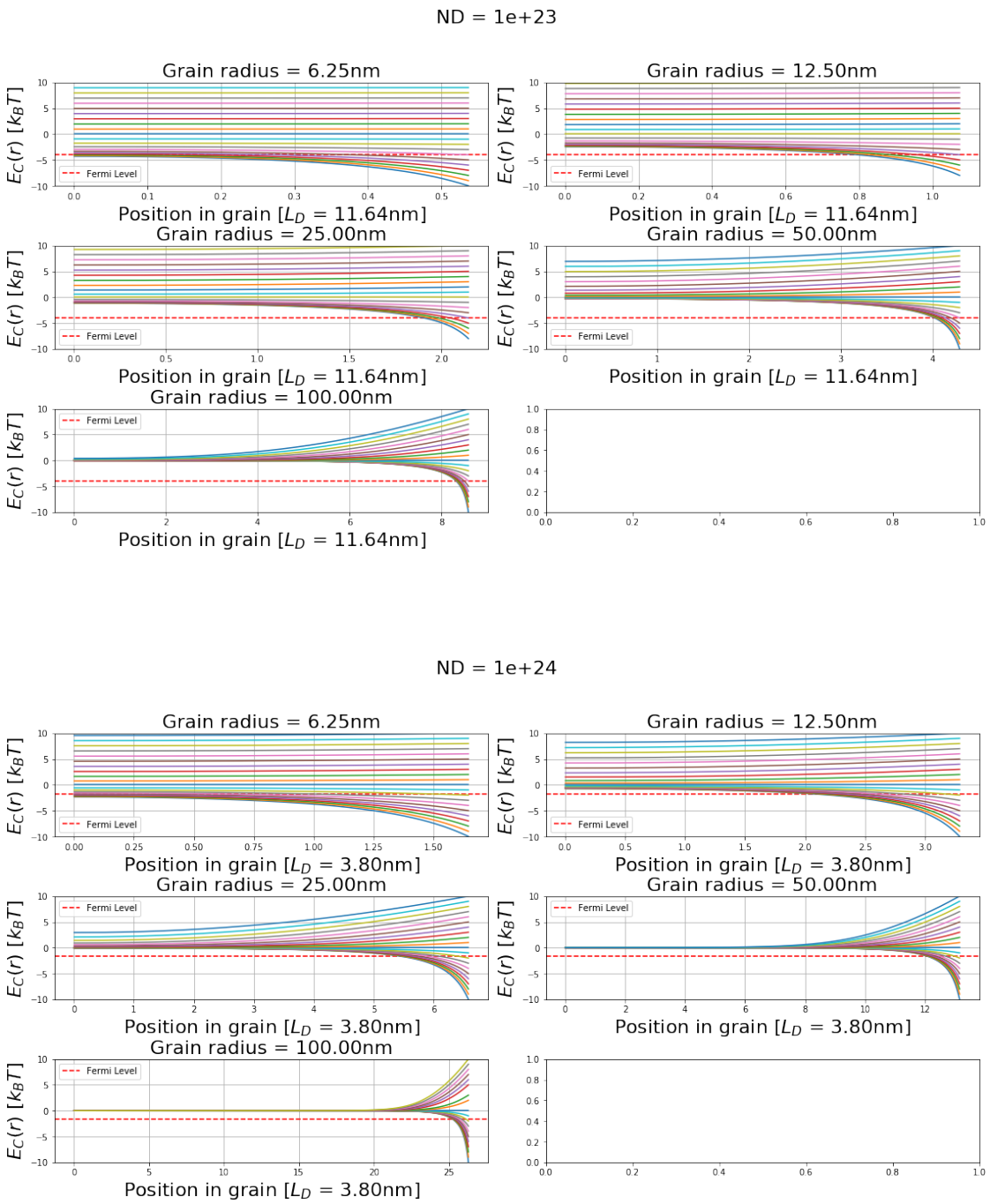
[6]:
```
for ND, calc_dF_ND in calc_dF.groupby('ND'):
    fig, axes= subplots(3,2,figsize = (16,9))
    fig.suptitle(f'ND = {ND}', fontsize = 22)
    for ax_i, (R, calc_dF_grainsize) in enumerate(calc_dF_ND.groupby('R')):
        axe = fig.axes[ax_i]

        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.set_ylim(-10,10)

        for vinit, ser_temp in calc_dF_grainsize.iterrows():

            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.tight_layout()
    fig.subplots_adjust(top=.85)
```

ND = 1e+21



ND = 1e+22





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.

3 From charge distribution to resistance

3.1 The "numerical" grain

With the previous tools and calculation it is now possible to assign each point inside the grain a certain charge density. 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 in the conduction band and μ_n the mobility of the electrons. For the case of and p-type semiconductor like SnO_2 with $n >> p$, the conductivity can be simplified to the following equation:

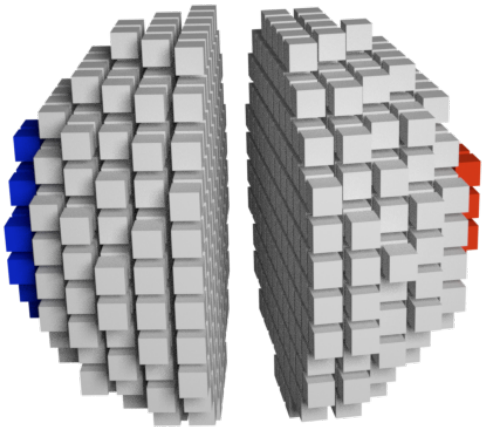
$$Conductivity = \sigma = q * n * \mu_n \tag{2}$$

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

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

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

To derive the 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 proportional to the gradient of the potential. Therefore the potential distribution inside the grain needs to be known. To do this, the grain is represented by a numerical model by slicing it into equal distributed cubes of the same size. Each cube will have a defined conductivity and potential. The colored areas in the picture indicate the areas, where a bias potential will be applied to generate a virtual electrical field.



To simulate the spread of the bias potential areas inside the grain, a technique called relaxation will be used. The general idea is to guess an initial potential distribution, and then, based on the laws of physics, iteratively correct this guess. The correction is done by re-calculating each time the potential U_0 of one center-"cube" based on potential U_i and conductivity σ of the direct neighbors. By doing this for each "cube", the potential distribution will more and more apply to the physical solution. When approaching to the solution, the overall changes in the potential of each cube will get smaller and smaller. In an ideal case it will not change anymore. In this case the potential of eachcube will be just as it should be to fulfill the laws of physics. To understand how this process can be supported by the means of modern matrix operations, I will shorty derive how U_0 is calculated from the surrounding U_i. In a second step matrix convolution will be used to solve the problem efficiently. First we will need to combine Ohm's law and Kirchhoff's first law:

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

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

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

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

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

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

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

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

With A the cube face area and l the distance between the cube centers, which have no relevance when using a model where all cubes are equal.

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

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

[7]:
```python
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='linear')
    v_int = interpolate.interp1d(r*grain.material.LD, v, kind='linear')

    return n_int, v_int
```

As mentioned earlier, the positions of the applied potential will only be the cubes on the far left and right, as indicated in the picture. 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 by using this simplification is to be able to represent the potential inside the grain 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. First we create now the data structure of the grain for further simulations. The $N \times N$ cubes will be represented by a numpy array.

[8]:
```python
def pos_to_r_o(xi,yi,grain, d):
    '''
    By passing the xi and yi indices, the grain and one array, the position (r)
    inside the grain is return
    '''
    dx = 2*grain.R/d.shape[0]
    cx = d.shape[0]//2
    cy = d.shape[1]//2
    ri = ((xi-cx)**2+(yi-cy)**2)**0.5
    return(ri*dx)




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

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


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

    ri = (((xi-cx))**2+((yi-cy))**2)**0.5
    if False:
        rx = -grain.R+xi*cube_size#+0.5*cube_size
        ry = -grain.R+yi*cube_size#+0.5*cube_size

        r = (rx**2+ry**2)**0.5
    else:
        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)
    nx = ny = 1+2*int(round((grain.R/cube_size)))
    #print(nx,ny)

    #initalize the data for the volatege (d_v) and conductivity (d_cond) with␣
 ↪zeros
    d_v = np.zeros((nx,ny))
    d_cond = np.zeros((nx,ny))

    #an additional mask for values outside the grain will be created
    d_mask = np.zeros((nx,ny))


    for xi in range(d_cond.shape[0]):
        for yi in range(d_cond.shape[1]):
```

```
            r = float(pos_to_r(xi,yi,grain, cube_size, d_v))
            try:
                #if r is outside the grain, n_int(r) will fail and got the␣
↪except part below
                # otherwise the conductivity will be saved in units of nb

                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
```

3.2 Precalc the numerical grains for all conditions

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

[14]:
```
d_cond_plots = []
for i, ser in calc_dF.iterrows():
    print(f'Initalized {i+1} of {len(calc_dF)}.', 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/25)
    d_cond_plot = d_cond.copy()
    d_cond_plot[np.where(d_mask==0)]=None
    d_cond_plots.append(d_cond_plot)
calc_dF.loc[:, 'd_cond'] = d_cond_plots
```

```
Initalized 420 of 403.
```

[15]:
```
calc_dF.to_hdf('precalc_dF.h5','raw', mode= 'w')
```

[16]:
```
calc_dF = pd.read_hdf('precalc_dF.h5','raw')
```

[17]:
```
%matplotlib inline
```

[18]:
```
def plot_grain_states(calc_dF_grainsize, vmax=None, vmin=None):
    fig, axes = subplots(3,7, figsize = (16,9))
```

```
    grain = create_grain_from_data(calc_dF_grainsize)

    for ax_i, (vinit, ser) in enumerate(calc_dF_grainsize.iterrows()):
        axe = fig.axes[ax_i]

        Einit_kT = ser['Einit_kT']

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

        d_cond_plot = 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(np.log(d_cond_plot), interpolation='bicubic',
                        extent=(-grain.R*1e9, grain.R*1e9, -grain.R*1e9, grain.
 ↪R*1e9),
                        vmax=vmax, 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=9e21):
    R = GrainRadius/1e9
    calc_dF_grainsize = calc_dF.groupby(['ND','R']).get_group((ND,R))
    max_n = np.log(calc_dF_grainsize['d_cond'].apply(lambda r:np.nanmax(r))).
 ↪max()
    min_n = np.log(calc_dF_grainsize['d_cond'].apply(lambda r:np.nanmin(r))).
 ↪min()

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


use_interactive_controls = True

if use_interactive_controls:
    from ipywidgets import interact, interactive, fixed, interact_manual
    import ipywidgets as widgets
    grainsizes = list(calc_dF['R'].unique())
    interact(plot_conductivity, GrainRadius=np.
 ↪array(grainsizes)*1e9,ND=list(calc_dF.groupby(['ND']).groups.keys()),␣
 ↪text='Select a grainsize:');
```

```
else:
    GrainRadius = 200

    for ND in list(calc_dF.groupby(['ND']).groups.keys()):
        plot_conductivity(GrainRadius, ND)
```

```
interactive(children=(Dropdown(description='GrainRadius', options=(6.25, 12.5, 25.0, 50.0, 100.0
```

3.3 Relaxation

3.3.1 Concolute2d:

Equation (3.3.1) 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. The function needs two parameters as inputs. The first is the matrix itself, while the second is the description of the convolution operation. In a very short description, this is What the algorithm will do:

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

It would be out of the scope to dig deeper into the details of convolutions, but this working a bit with the following example should reveal the main concept.

[23]:
```
from scipy import signal

U = np.array([[1,2,3],[1,2,3],[1,2,3]])
print('U:')
print(U)
print()

n = np.array([[1,1,1],[10,10,10],[100,100,100]])
print('n:')
print(n)
print()

print('U*n')
print(U*n)
print()

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

```
signal.convolve2d(U*n, conv, boundary='fill', mode='same', fillvalue=0)
```

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

n:
[[  1   1   1]
 [ 10  10  10]
 [100 100 100]]

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

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

[23]:
```
array([[ 12,  24,  32],
       [121, 242, 323],
       [210, 420, 230]])
```

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

[24]:
```
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
```

When 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 (Φ) to the grain, the total resistance Ω could be calculated by Ohm's low: $\Omega = \frac{\Delta\Phi}{Current}$, where $\Delta\Phi = V$ 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.

$$j = \lim_{A\to 0} \frac{I_A}{A} = \lim_{d\to 0} \frac{I_A}{d * d} = \sigma E \tag{13}$$

$$E = \frac{V}{d} = \frac{\Delta\Phi}{d} \tag{14}$$

$$I_A = \sigma * V * d \tag{15}$$

which will be the current going through one cube

[25]:
```
def calc_current_center(d_v, d_cond, d_mask, cube_size, grain):
    #center_pos = d_v.shape[0]//2
    #center_pos = 20
    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
```

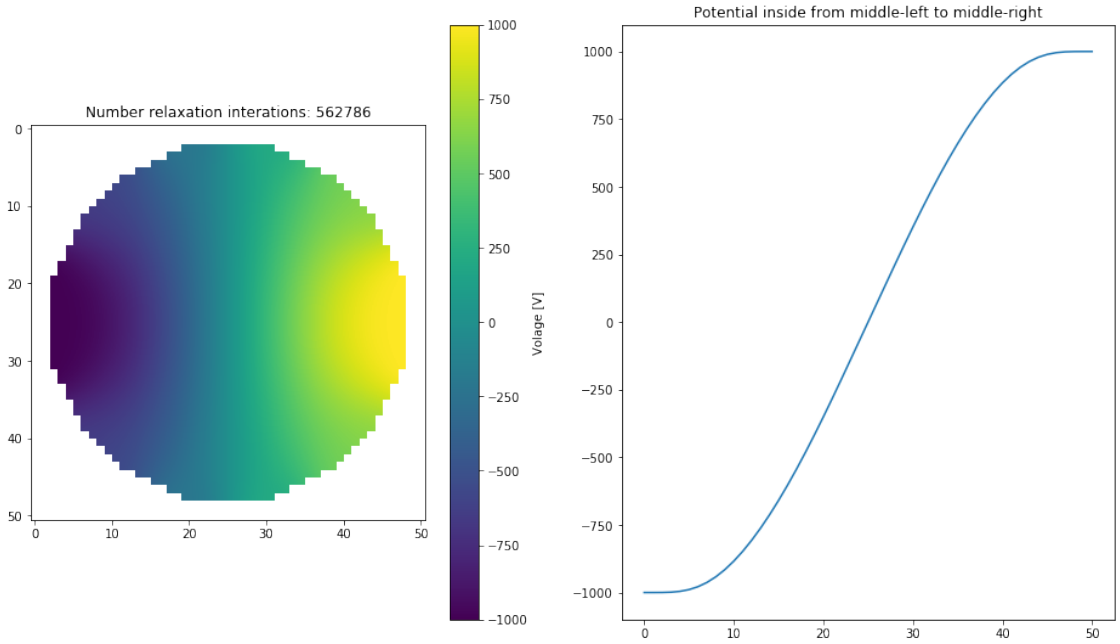
[26]:
```
calc_dF['current'] = None
```

4 See how a single solution evolves

The interactive environment of Jupyter will be needed here. The following command activates it. (For some unknown reasons, you might need to execute this command 2-3 times to get activated)

[57]:
```
import matplotlib.animation as animation

c_dF = calc_dF.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/25
ser['cube_size'] = cube_size
grain = create_grain_from_data(ser)

if cube_size=='LD':
    cube_size_value = grain.material.LD/2
else:
    cube_size_value = cube_size

d_v, d_cond, d_mask = create_numerical_grain_matrix( grain,␣
 ↪ser,cube_size=cube_size_value)
ns = 1
def update(frame):
    axe.clear()
    axe_v.clear()
    n = 5
    n = conv_runs[frame]
    global d_v
    global ns
    ns+=n
    axe.set_title(f'Number relaxation interations: {ns}')
    axe_v.set_title('Potential inside from middle-left to middle-right')

    d_v, _, _ = solve_relaxation(d_v = d_v , d_cond=d_cond, d_mask=d_mask, n=n)
```

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


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

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

    return img


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

axe = axes[0]

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

axe_v = axes[1]

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

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

plt.show()
```



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

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

[58]:
```
from IPython.display import Video
display(Video('./im.mp4'))
```

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

4.0.1 Precalcualtion of all conditions

[59]:
```
def calc_conv_by_ser(ser):
    cube_size = ser['cube_size']
    vinit = ser.name

    grain = create_grain_from_data(ser)
    if cube_size=='LD':
        cube_size_value = grain.material.LD/100
    else:
        cube_size_value = 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)
    #print(center_current_tot)
    #plot_num_grain(d_v, d_cond, d_mask)
    #plot_voltage_1d(d_v)
    #plot_center_current(r, center_current)
    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
```

[120]:
```
start_t = time.time()
from multiprocessing import Pool
#for vinit, ser in calc_dF.iterrows():
all_solutions = []
for cube_size in ['LD']:
    c_dF = calc_dF.copy()
    # cubesize defined by LD
    c_dF['cube_size'] = cube_size

    # cubesize defined by radius
    c_dF['cube_size'] = c_dF['R']/25

    if False:
        for i, ser in c_dF.iterrows():
            print(ser)
            all_solutions.append(calc_conv_by_ser(ser))
            print('Done')
            print()
    else:
        ser_list = []
        for i, ser in c_dF.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)
print(time.time()-start_t)
```

```
686.3965079784393
```

[179]:
```
calc_dF_sol.to_hdf('numerical_sol.h5','raw',␣
 ↪mode='w',complevel=9,complib='bzip2')
```

```
/usr/lib/python3.8/site-packages/pandas/core/generic.py:2530:
PerformanceWarning:
```

```
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed,key->block1_values] [items->['n',
'r', 'v', 'v_dot', 'd_cond', 'd_v', 'd_mask']]

  pytables.to_hdf(path_or_buf, key, self, **kwargs)
```

[64]:
```
calc_dF_sol = pd.read_hdf('numerical_sol.h5','raw')
```

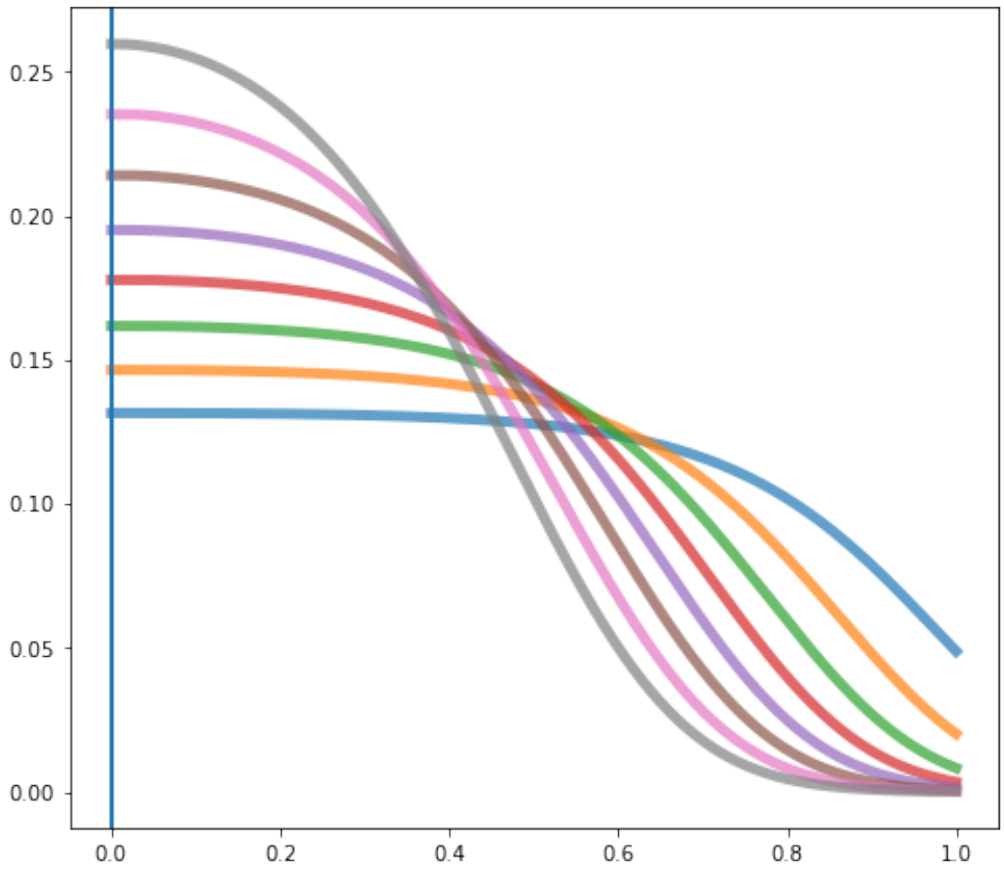
5 check the integral for the charges

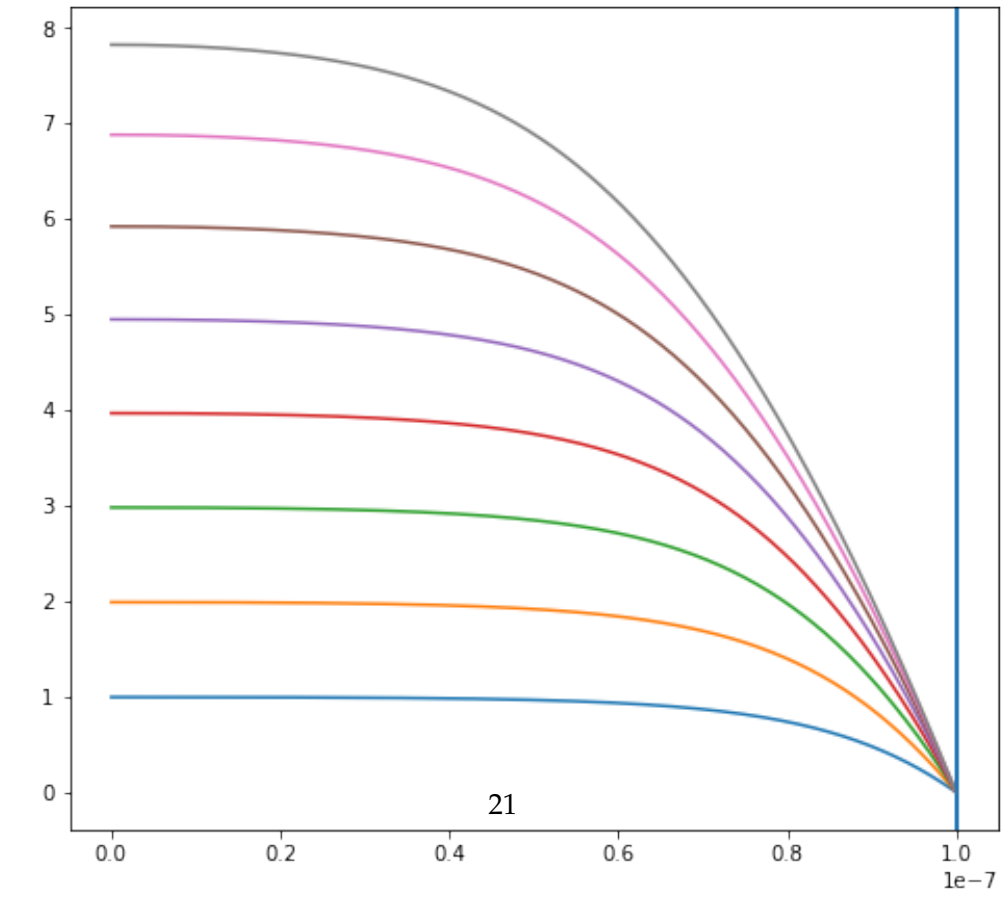
[65]:
```
%matplotlib inline
```

[66]:
```
index = 8
ser = calc_dF_sol.iloc[index]


fig, axes = subplots(2, figsize=(8,16))
for v in [1,2,3,4,5,6,7,8]:
    try:
        ser = calc_dF_sol[(calc_dF_sol['R']==100e-9) &␣
 ↪(calc_dF_sol['ND']==1e+23) & (calc_dF_sol['Einit_kT']==v)].iloc[0]
    except:
        continue
    grain = create_grain_from_data(ser)


    #print(alle,surf, round((surf-alle)/alle*100,2))

    axe = axes[0]
    #axe.set_yscale('log')
    n_avg = np.trapz(ser['n'], ser['r'])
    #n_avg = ser['all_charges']
    print(v,n_avg)
    #print(ser)
    axe.plot(ser['r']*grain.material.LD/ser['R'], np.array(ser['n'])/n_avg,␣
 ↪linewidth=5, alpha=0.7)
    #axe.axhline(grain.material.n(0)[0], c='k', linestyle='--')
    axe.axvline(ser['R'])

    axe = axes[1]
    axe.plot(ser['r']*grain.material.LD, v-np.array(ser['v']))
    #axe.set_ylim(-8,8)
    axe.axvline(ser['R'])
    #ser
```

```
1 1.5021657752341905e+24
```

```
2 1.340593351501356e+24
3 1.2036159250637978e+24
4 1.0814610889734379e+24
5 9.677215620595219e+23
6 8.583617474730477e+23
7 7.503639913508516e+23
8 6.420063673281262e+23
```


0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.2
0.4
0.6
0.8
1.0


8
7
6
5
4
3
2
1
0
0.0
0.2
0.4
0.6
0.8
1.0
1e−7

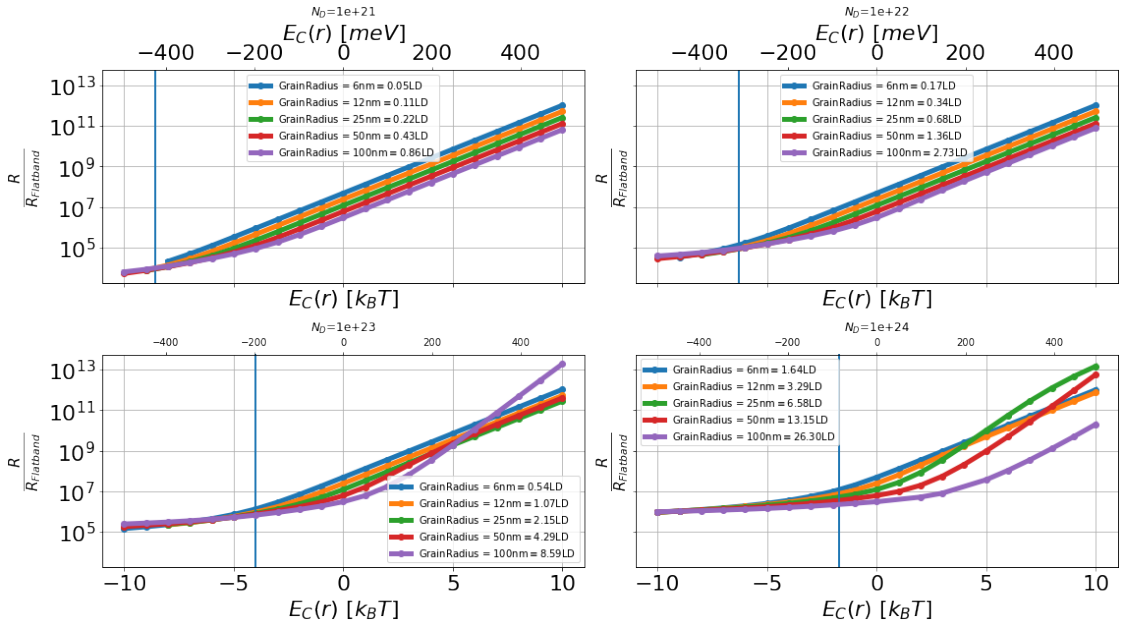
These results are in line with the results presented from this publication:

related publication

5.0.1 Comparing with experimental data

[67]:
```
calc_dF_size = calc_dF_sol
calc_dF_size = calc_dF_sol
fig, axes = subplots(len(calc_dF_size['ND'].unique())//2,2,figsize = (16,9),␣
 ↪sharey=True, sharex=True)
if len(fig.axes)==1:
    axes = [axes]
for ax_i,(ND,calc_dF_n) in enumerate(calc_dF_size.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])
        #print((grain.material.LD)*1e9)
        #print(calc_dF_grainsize['d_v'].iloc[0].shape)
        axe.axvline(-grain.material.J_to_kT(grain.material.Diff_EF_EC))
        s = calc_dF_grainsize['d_v'].iloc[0].shape[0]
        #print(calc_dF_grainsize['cube_size_value'].iloc[0]*1e9)
        #print()

        flat_band = calc_dF_grainsize[calc_dF_grainsize['Einit_kT']==0].
 ↪iloc[0]['current']
        #res = flat_band/calc_dF_grainsize['current']
        res = 2000/calc_dF_grainsize['current']


        #res = 1/calc_dF_grainsize['current']
        v = calc_dF_grainsize['Einit_kT']
        rel_size = grain.R/grain.material.LD
        axe.plot(v, res, 'o-', label = f'GrainRadius = {R*1e9:.
 ↪0f}nm$\equiv${rel_size:.2f}LD', linewidth=5)


        axe.set_yscale('log')

        axe.set_ylabel(r'$\frac{R}{R_{Flatband}}$', fontsize =22)
        axe.set_xlabel('$E_C(r)$ [$k_BT$]', fontsize =22)
```

```
        axe.tick_params(axis='both', which='major', labelsize=22)

        axe.grid(b=True)
    axe_up.plot(v*CONST.J_to_eV(grain.material.kT)*1000, res,alpha=0)
    if ax_i in [0,1]:
        axe_up.set_xlabel('$E_C(r)$ [$meV$]', fontsize =22)
        axe_up.tick_params(axis='both', which='major', labelsize=22)


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



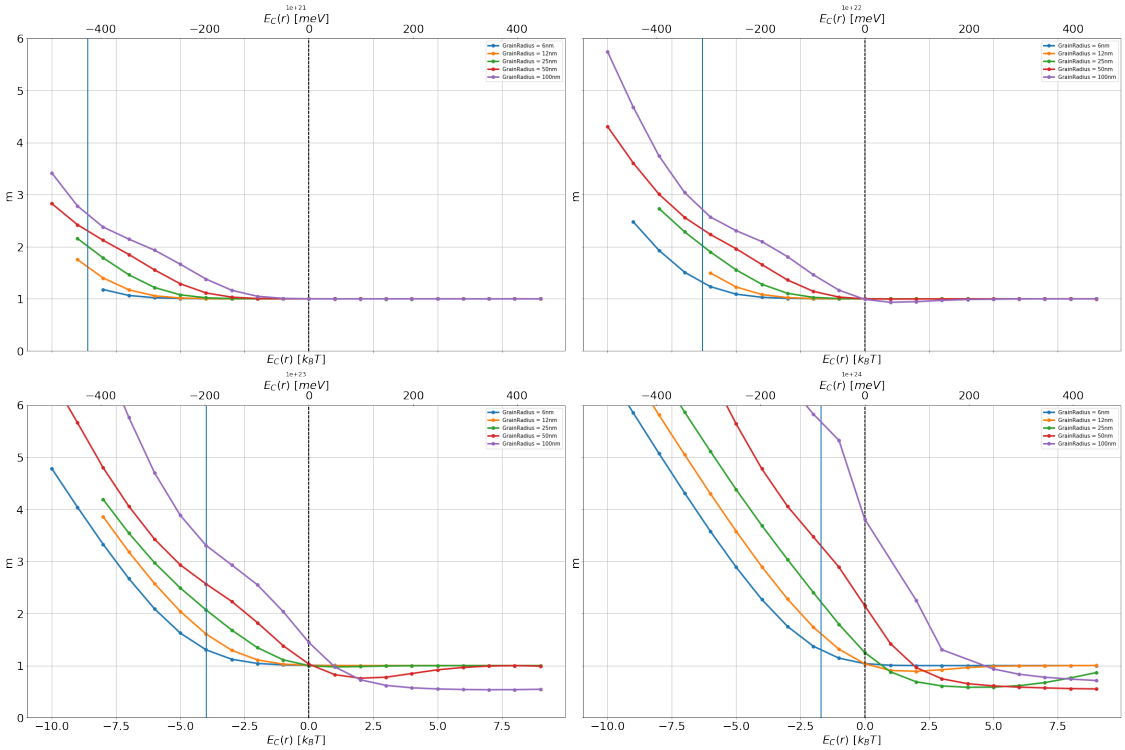
[170]:
```
110/2/3.6
```

[170]:
```
15.277777777777777
```

5.0.2 The 'm' factor

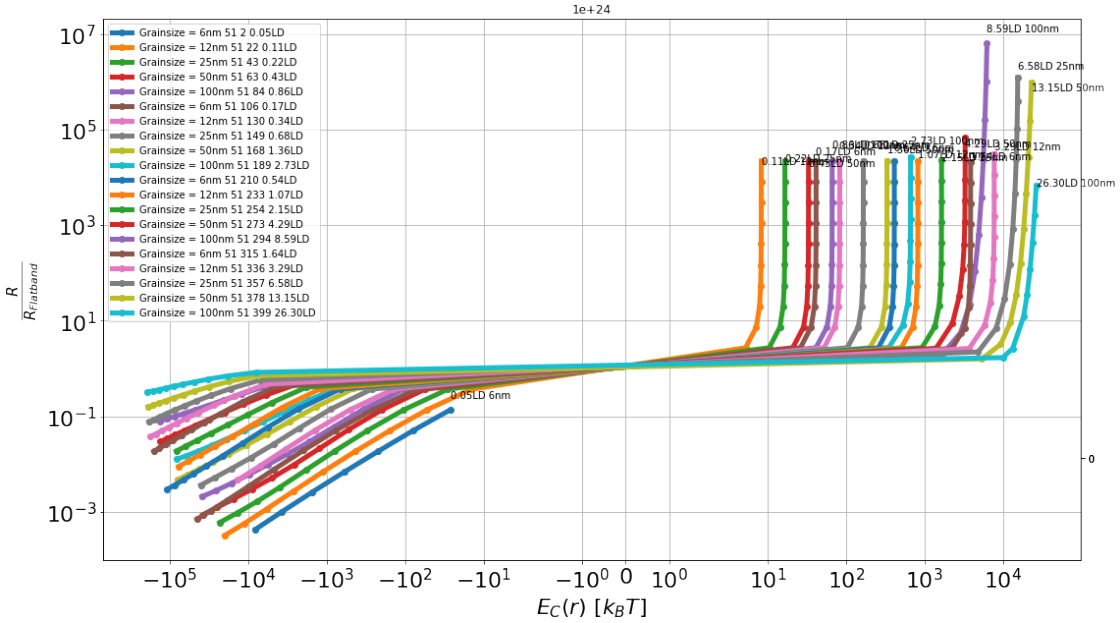
[68]:
```
calc_dF_n_all_ND = calc_dF_sol

fig, axes = subplots(len(calc_dF_size['ND'].unique())//2,2,figsize = (30,20),␣
 ↪sharey=True, sharex=True)

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

    axe = fig.axes[ax_i]
```

```
    axe.set_title(ND)

    axe_up = axe.twiny()
    for ax_i, (R,calc_dF_grainsize) in enumerate(calc_dF_n.groupby('R')):
        if R==50e-9:
            pass
        else:
            pass
        grain = create_grain_from_data(calc_dF_grainsize.iloc[0])
        axe.axvline(-grain.material.J_to_kT(grain.material.Diff_EF_EC))
        axe.axvline(linestyle='--', color='k')

        flat_band = calc_dF_grainsize[calc_dF_grainsize['Einit_kT']==0].
↪iloc[0]['current']
        res = flat_band/calc_dF_grainsize['current']
        res = 1/calc_dF_grainsize['current']
        log_res = np.log([float(r) for r in res.values])
        v = calc_dF_grainsize['Einit_kT']
        m = np.diff(log_res)/np.diff(v)

        sc = calc_dF_grainsize['charges_trapped_at_surface']
        #print(sc,(grain.material.LD**(2/3)*surf))
        surf = R**2*4*pi
        #surf_ratio = (grain.material.nb**(2/3)*surf)/sc
        surf_ratio = sc/surf
        v = calc_dF_grainsize['Einit_kT']
        axe.plot(v[0:-1], 1/m,'o-', label = f'GrainRadius = {R*1e9:.0f}nm',␣
↪linewidth=3)
        #axe.plot(v, res, 'o-', label = f'{size_n} {R}', linewidth=5)
        #axe.set_yscale('log')

        axe.tick_params(axis='both', which='major', labelsize=22)

        axe.set_ylabel('m', fontsize  = 22)
        axe.set_xlabel('$E_C(r)$ [$k_BT$]', fontsize =22)
        axe.grid(b=True)
        axe.set_ylim(0,6)
    axe_up.plot(v[0:-1]*CONST.J_to_eV(grain.material.kT)*1000, 1/m,alpha=0)
    axe_up.set_xlabel('$E_C(r)$ [$meV$]', fontsize =22)
    axe_up.tick_params(axis='both', which='major', labelsize=22)

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



[178]:
```
calc_dF_size = calc_dF_sol
calc_dF_size = calc_dF_sol
#fig, axes = subplots(len(calc_dF_size['ND'].unique()),1,figsize = (16,9),␣
 ↪sharey=True, sharex=True)
if len(fig.axes)==1:
    axes = [axes]
fig, axe = subplots(1,1,figsize = (16,9), sharey=True, sharex=True)
for ax_i,(size_n,calc_dF_n) in enumerate(calc_dF_size.groupby('ND')):

    #axe = axes[ax_i]
    axe.set_title(size_n)
    axe2= axe.twinx()
    for ax_i, (R,calc_dF_grainsize) in enumerate(calc_dF_n.groupby('R')):
        flat_band = calc_dF_grainsize[calc_dF_grainsize['Einit_kT']==0].
 ↪iloc[0]['current']

        calc_dF_grainsize =␣
 ↪calc_dF_grainsize[calc_dF_grainsize['charges_trapped_at_surface'].abs()>0.01]

        grain = create_grain_from_data(calc_dF_grainsize.iloc[0])
        #print((grain.material.LD)*1e9)
        #print(calc_dF_grainsize['d_v'].iloc[0].shape)
        s = calc_dF_grainsize['d_v'].iloc[0].shape[0]
```

```
        #print(calc_dF_grainsize['cube_size_value'].iloc[0]*1e9)
        #print()


        res = flat_band/calc_dF_grainsize['current']
        #res = 2000/calc_dF_grainsize['current']
        sc = calc_dF_grainsize['charges_trapped_at_surface']
        #print(sc,(grain.material.LD**(2/3)*surf))
        #surf = R**2*4*pi
        surf = calc_dF_grainsize['grain_surface']
        #surf_ratio = (grain.material.nb**(2/3)*surf)/sc
        surf_ratio = sc/surf/1e12

        sc = calc_dF_grainsize['charges_trapped_at_surface']

        #res = 1/calc_dF_grainsize['current']
        v = calc_dF_grainsize['Einit_kT']
        rel_size = grain.R/grain.material.LD
        #axe.plot(v, res, 'o-', label = f'Grainsize = {R*1e9:.0f}nm {s}␣
↪{calc_dF_grainsize.iloc[0].name} {rel_size:.2f}LD', linewidth=5)
        axe.plot(surf_ratio, res, 'o-', label = f'Grainsize = {R*1e9:.0f}nm {s}␣
↪{calc_dF_grainsize.iloc[0].name} {rel_size:.2f}LD', linewidth=5)
        idxmax = res.idxmax()
        axe.text(surf_ratio.loc[idxmax], res.loc[idxmax]*random.randint(50,200)/
↪100, f'{rel_size:.2f}LD {R*1e9:.0f}nm')
        #axe2.plot(v,surf_ratio,  'o-', linewidth=1,label = f'Grainsize = {R*1e9:
↪.0f}nm {s} {calc_dF_grainsize.iloc[0].name} {rel_size:.2f}LD')


        axe.set_yscale('log')
        axe2.set_yscale('symlog')
        axe.set_xscale('symlog')
        #axe.set_xlim(1e12,1e17)
        axe.set_ylabel(r'$\frac{R}{R_{Flatband}}$', fontsize =22)
        axe.set_xlabel('$E_C(r)$ [$k_BT$]', fontsize =22)

        axe.tick_params(axis='both', which='major', labelsize=22)

        axe.grid(b=True)
        #axe2.legend()

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



6 Bibliography section

References

Comparing numerical results with experimental data

February 7, 2020

Contents

1 Bibliography section

References