# From SMOx-grains to resistance

December 13, 2019

## 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.

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

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

[2]:
```
from part2 import *
```

## 2 Load the results

[3]:
```
calc_dF = pd.read_hdf('results.h5', 'raw')
calc_dF.index = range(len(calc_dF))
```

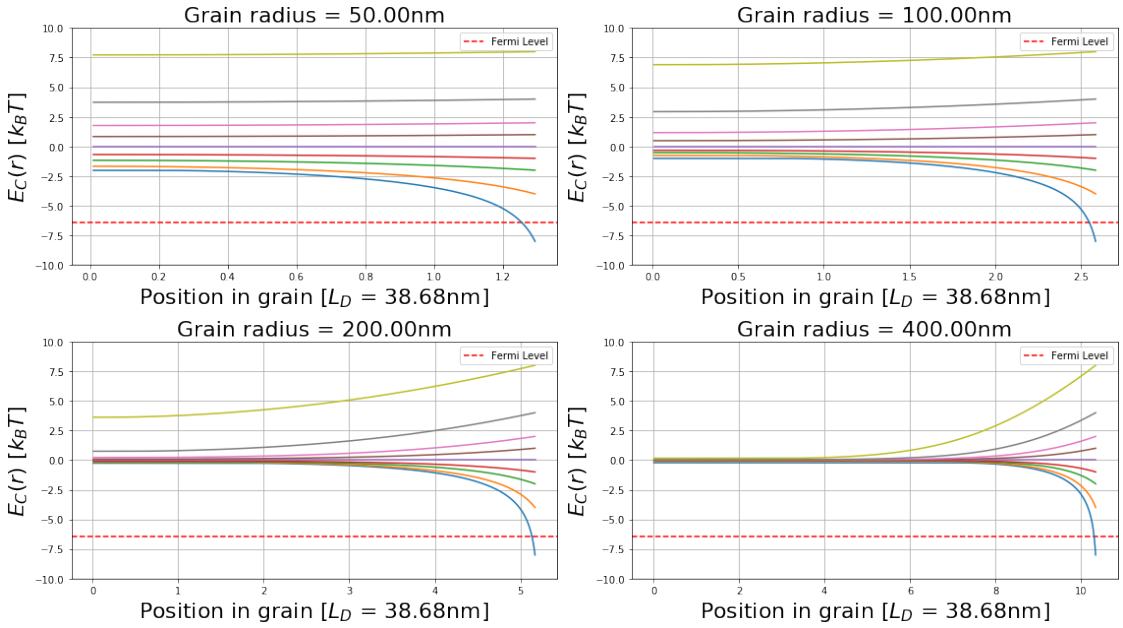
In the following code block, we define a function, that will initialize a new `grain` object from the saved data from the previous notebook. All required data is given in the imported table. The helper function will initialize the `grain` with the corresponding `material` and return it.

[4]:
```
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.')


    EDCF_eV = calc_EDCF_by_temp(T_C, ND, mass_e_eff_factor)

    material = Material(T_C,DIFF_EF_EC_evolt=EDCF_eV)
    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.

[5]:
```
fig, axes= subplots(2,2,figsize = (16,9))
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)

    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()
```




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 $\mu_n$ 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:

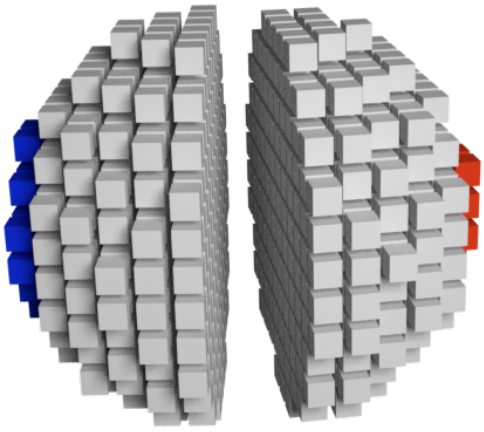
$$Conductivity = \sigma = q * n * \mu_n \tag{2}$$

The relation between resistivity $\rho$ 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 $\sigma$ 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:

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

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

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

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

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

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

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

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

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.

[19]:
```
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.

[119]:
```
def pos_to_r(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[:,0] = -v
    d_v[:,-1] = +v
    d_v = d_v*d_mask
    return d_v

def create_numerical_grain_matrix( grain, ser,size_n=100,):
    #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 = size_n

    #initalize the data for the volatege (d_v) and conductivity (d_cond) with␣
 ↪zeros
    d_v = np.zeros((2*nx+1,2*ny+1))
    d_cond = np.zeros((2*nx+1,2*ny+1))

    #an additional mask for values outside the grain will be created
    d_mask = np.zeros((2*nx+1,2*ny+1))
```

```
    for xi in range(d_cond.shape[0]):
        for yi in range(d_cond.shape[1]):
            r = float(pos_to_r(xi,yi,grain, d_cond))
            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


def pos_to_r2(xi,yi,grain, cube_size):
    '''
    By passing the xi and yi indices, the grain and one array, the position (r)
    inside the grain is return
    '''



    rx = -grain.R+xi*cube_size
    ry = -grain.R+yi*cube_size

    r = (rx**2+ry**2)**0.5
    return r

def r_to_pos(r, grain, cube_size):
    return int(ceil((r+grain.R)/cube_size))


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 = r_to_pos(grain.R, grain, cube_size)
```

```
    #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_r2(xi,yi,grain, cube_size))
            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.

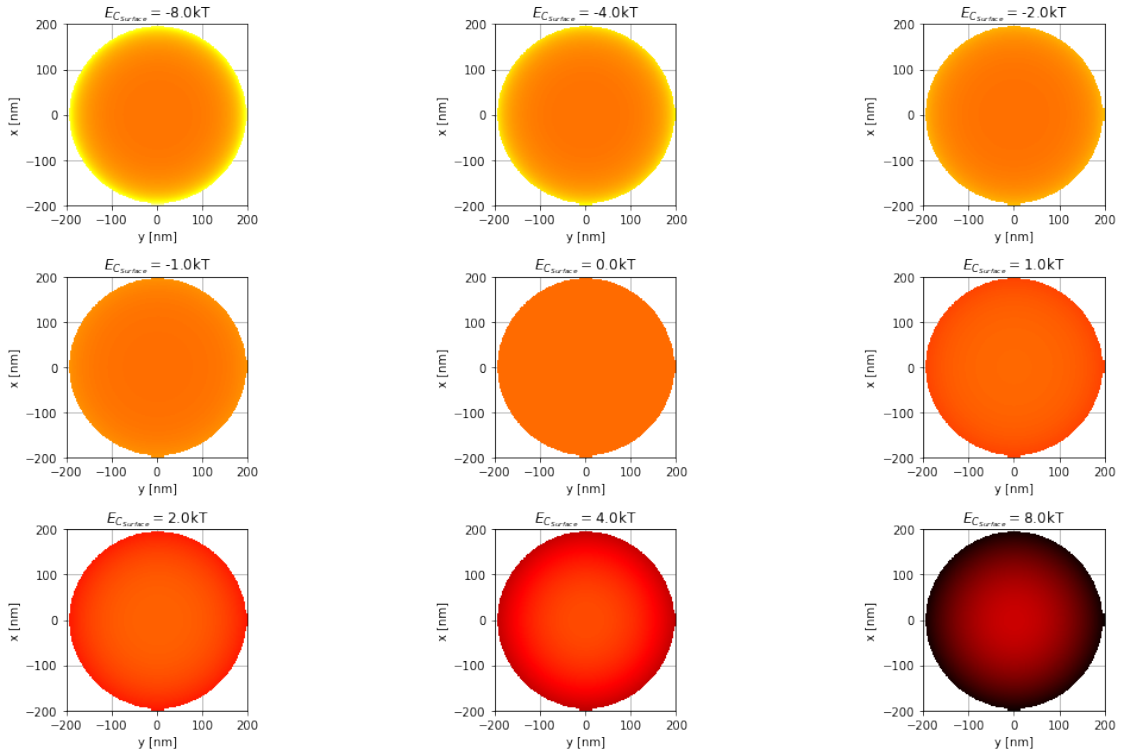
[121]:
```
d_cond_plots = []
for i, ser in calc_dF.iterrows():
    print(f'Initalized {i+1} of {len(calc_dF)}.', end='\r')
    grain = create_grain_from_data(ser)
    d_v, d_cond,d_mask = create_numerical_grain_matrix( grain,␣
↪ser,cube_size=1e-9)
    d_cond_plot = d_cond.copy()
    d_cond_plot[np.where(d_mask==0)]=None
    d_cond_plots.append(d_cond_plot)
calc_dF.loc[:, 'd_cond'] = d_cond_plots
```

```
Initalized 36 of 36.
```

[122]:
```
%matplotlib inline
```

[127]:
```
def plot_grain_states(calc_dF_grainsize, vmax=None, vmin=None):
    fig, axes = subplots(3,3, 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()



def plot_currents(GrainRadius):
    R = GrainRadius/1e9
    calc_dF_grainsize = calc_dF.groupby('R').get_group(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 = False

if use_interactive_controls:
    grainsizes = list(calc_dF['R'].unique())
    interact(plot_currents, GrainRadius=np.array(grainsizes)*1e9, text='Select a␣
 ↪grainsize:');
else:
    GrainRadius = 200
```

```
plot_currents(GrainRadius)
```




## 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.

[128]:
```
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]]
```

[128]:
```
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.

[153]:
```
from scipy import signal


def solve_relaxation(d_v, d_cond, d_mask):
    res_new = 1000
    for i in range(1000000):

        conv = [[0,1,0],[1,0,1],[0,1,0]]
        numerator = signal.convolve2d(d_v*d_cond, conv, boundary='fill',
                                      mode='same', fillvalue=0)
        denominator = signal.convolve2d(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

    return d_v, d_cond, d_mask
```

[157]:
```
calc_dFs.keys()
```

[157]:
```
dict_keys([100])
```

[79]:
```
calc_dF['current'] = None
```

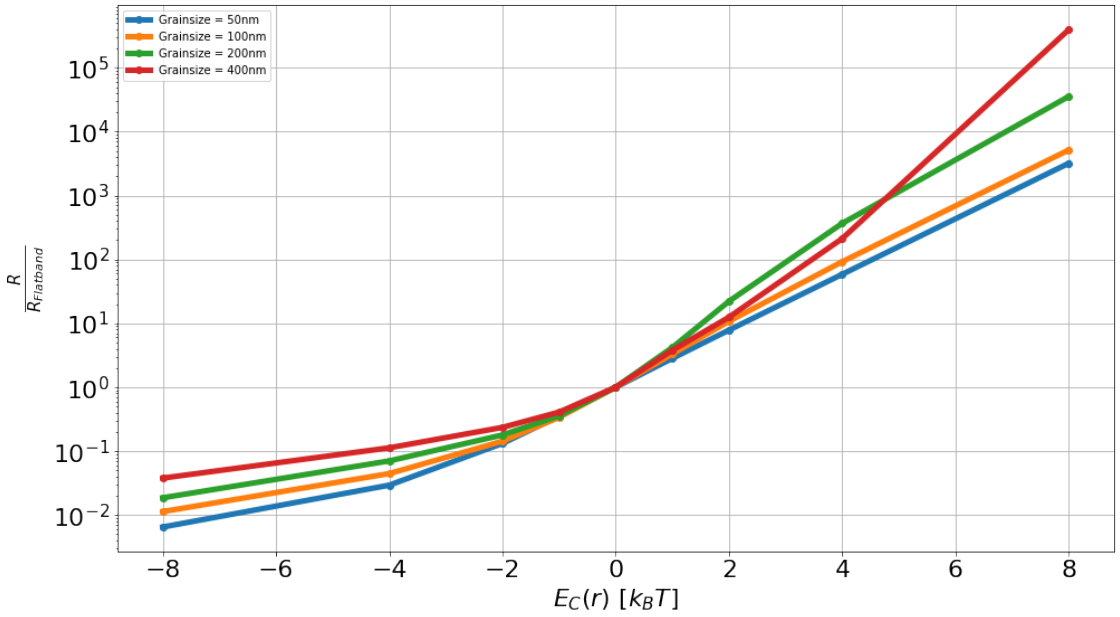
### 3.3.2 Precalcualtion of all conditions

[137]:
```
#for vinit, ser in calc_dF.iterrows():
calc_dFs = {}
for cube_size in [10e-9,5e-9]:
    c_dF = calc_dF.copy()
    for i, (ind,ser) in enumerate(c_dF.iterrows()):

        print(f'Initalized {i+1} of {len(c_dF)}.')
        vinit = ser.name

        grain = create_grain_from_data(ser)

        d_v, d_cond, d_mask = create_numerical_grain_matrix( grain,␣
↪ser,cube_size=cube_size)
        d_v = initaliz_d_v(d_v, d_mask, 1000)

        d_v, d_cond, d_mask = solve_relaxation(d_v, d_cond, d_mask)

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

        #plot_num_grain(d_v, d_cond, d_mask)
        #plot_voltage_1d(d_v)
        #plot_center_current(r, center_current)


        c_dF.loc[vinit, 'current'] = center_current_tot
    calc_dFs[cube_size] = c_dF
```

### 3.3.3 Comparing with experimental data

[161]:
```
fig, axes = subplots(1,1,figsize = (16,9), sharey=True, sharex=True)
if len(fig.axes)==1:
    axes = [axes]
for ax_i,(size_n,calc_dF_n) in enumerate(calc_dFs.items()):

    axe = axes[ax_i]
    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']
        res = flat_band/calc_dF_grainsize['current']


        #res = 1/calc_dF_grainsize['current']
```

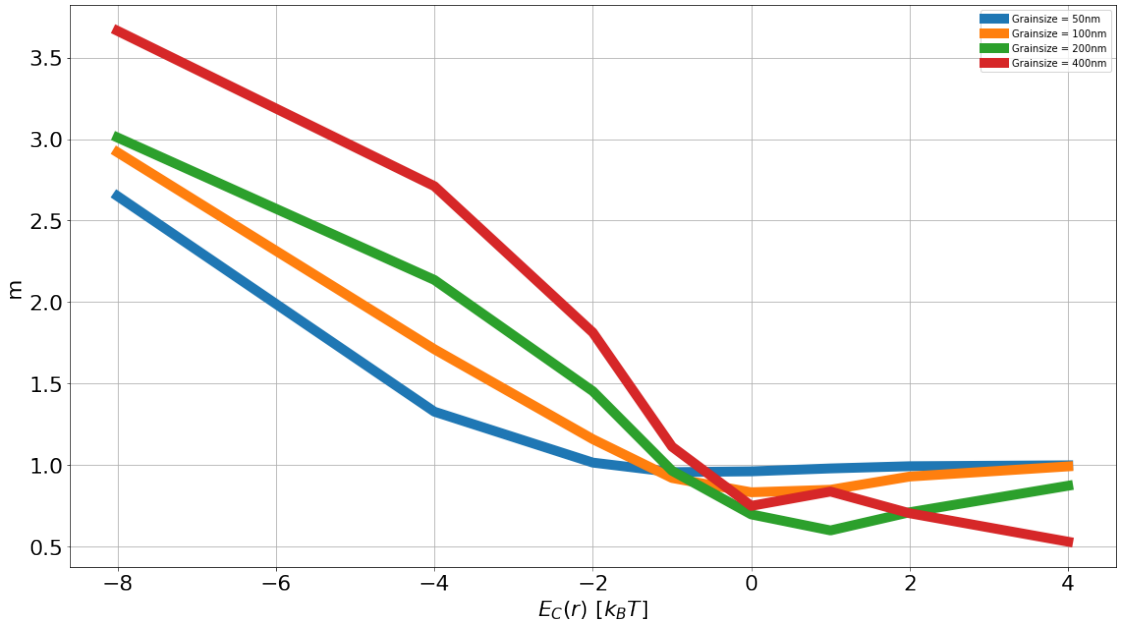
```
        v = calc_dF_grainsize['Einit_kT']
        axe.plot(v, res, 'o-', label = f'Grainsize = {R*1e9:.0f}nm', linewidth=5)


        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)

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




### 3.3.4 The 'm' factor

[165]:
```
fig, axe = subplots(1,1,figsize = (16,9), sharey=True, sharex=True)
size_n = 100
calc_dF_n = calc_dFs[size_n]



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

    v = calc_dF_grainsize['Einit_kT']
    axe.plot(v[0:-1], 1/m,'o-', label = f'Grainsize = {R*1e9:.0f}nm',␣
↪linewidth=10)
    #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)

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




[ ]:
```
def plot_num_grain(d_v, d_cond, d_mask):
    fig, axes =subplots(1,3)
    d_v_plot = d_v.copy()
```

```
    d_v_plot[np.where(d_mask==0)]=None

    axes[0].imshow(d_mask)
    axes[1].imshow(d_cond)
    axes[2].imshow(d_v_plot,interpolation= 'nearest')

def plot_voltage_1d(d_v):
    fig, axe = subplots()
    center = d_v[d_v.shape[0]//2,:]
    axe.plot(center)

def calc_current_center(d_v, d_cond, d_mask):
    center_pos = d_v.shape[0]//2
    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, d_v)) for xi in␣
 ↪range(len(center_current))])

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

def plot_center_current(r, center_current):
    fig, axe = subplots()
    axe.plot(r*1e9,center_current)
    return

#vinit, current = calc_current(d_v, d_cond, d_mask)
```

[ ]:
```
for i, ser in calc_dF.iterrows():
    print(f'Initalized {i+1} of {len(calc_dF)}.', end='\r')
    grain = create_grain_from_data(ser)
    d_v, d_cond,d_mask = create_numerical_grain_matrix( grain, ser,size_n=100)
    d_cond_plot = d_cond.copy()
    d_cond_plot[np.where(d_mask==0)]=None
    calc_dF.loc[ser.name, 'd_cond'] = [d_cond_plot]
```

[150]:
```
calc_dF.to_hdf('res.h5', 'raw')
```

```
/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', 'current']]

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

# 4 Bibliography section

## References