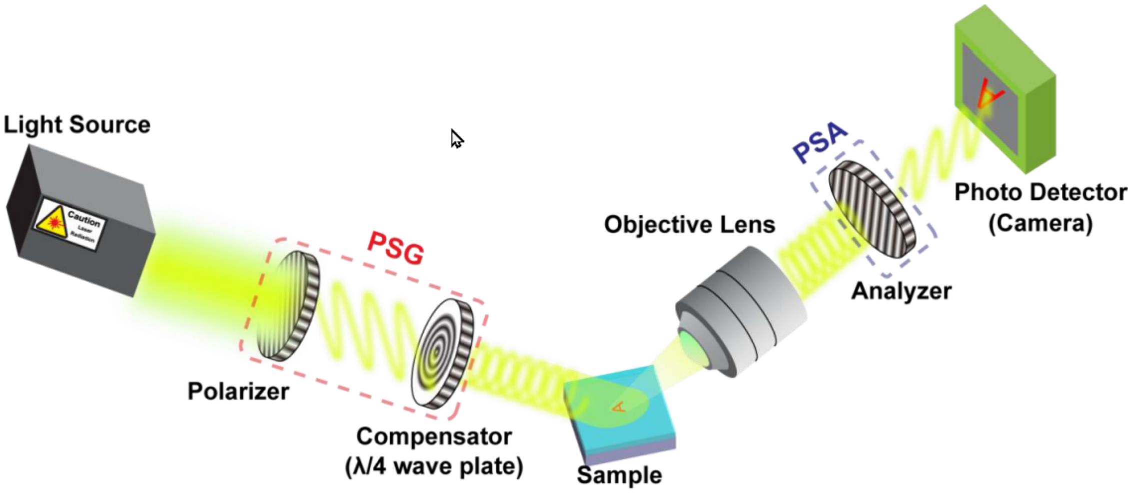
Image segmentation in ellipsometric maps

Project presented as TFM by Antonio González Delgado for KSchool. Data powered by and for Accurion GmbH

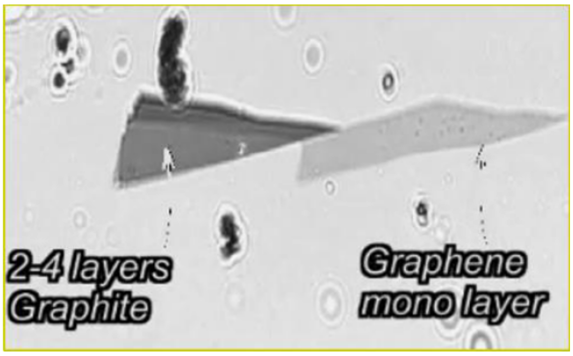
What are ellipsometric maps

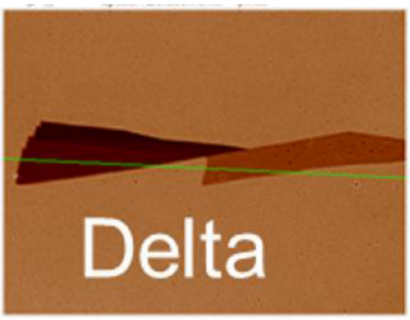
Ellipsometry is an optical method for the characterization of ultrathin films. It is based on the fact that the polarization state of light might change when a light beam is reflected from a surface. If the surface is covered by a thin film (or a stack of films), the entire optical system of film and substrate influences the change in polarization. It is therefore possible to deduce information about the film properties, especially the film thickness.

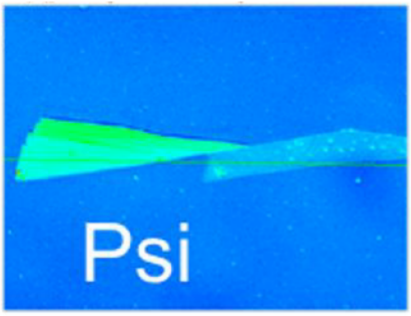


The basic components of an ellipsometer are: a light source, some optical components like polarizers and retarders and a detector. By controlling the polarization state of the light beam hitting the sample and to analysing the changes in the polarization of the reflected beam, the method obtains the so-called ellipsometric angles: Delta and Psi. They are calculated as a function of the rotation angle of the optical component, the intensity at the detector and the wavelength and angle of incidence of the incoming light. These angles can be then transferred by optical modelling into a number of physical parameters like layer thickness or optical properties.

By using imaging technology, it is possible to extend the classical ellipsometer to a new form of visualization tool or a microscope with extreme sensitivity to thin films. **Ellipsometric maps** are the result of imaging ellipsometry: images where Delta and Psi have been calculated for each pixel, adding the advantage of spatial resolution to traditional ellipsometry. Secondly, the microscope's polarization capabilities and its operation at an oblique AOI provide dramatic contrast enhancement for surface structures that only feature marginal differences in their optical properties. For example, monoatomic or monomolecular steps of ultrathin films.

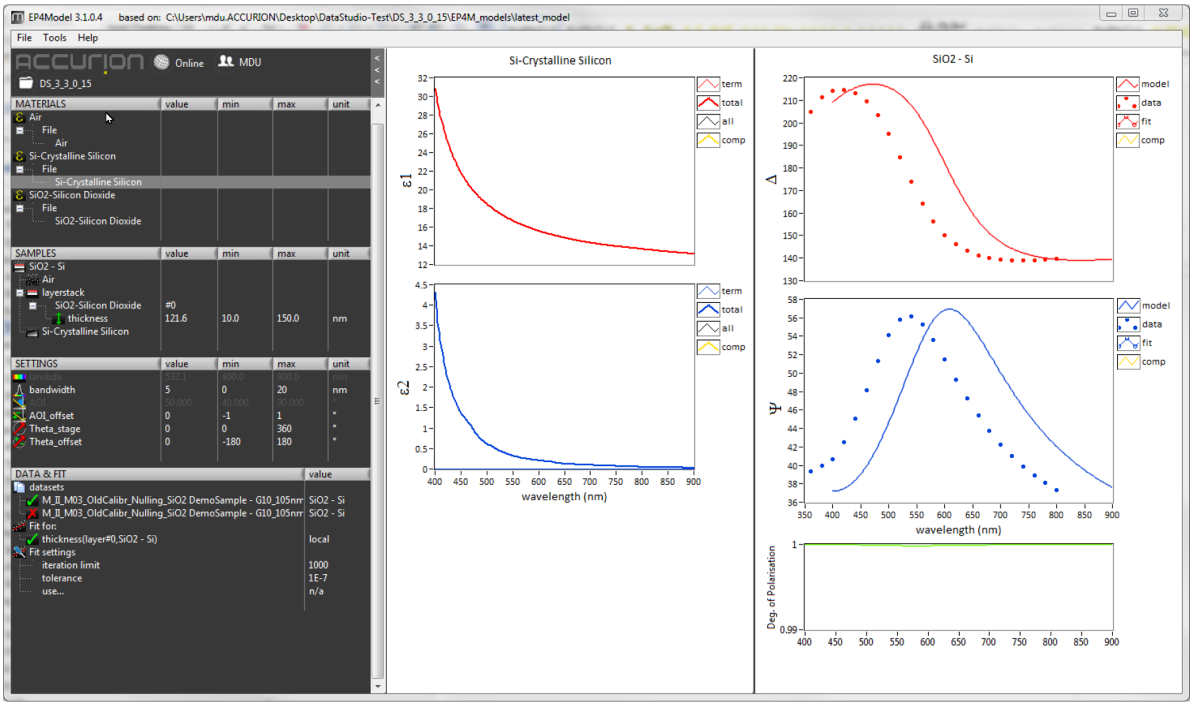






Motivation of the project

One of the major challenges of ellipsometry is the translation of the ellipsometric parameters Δ and Ψ (i.e. the sample's ellipsometric fingerprint) to its physical quantities of interest, i.e. a layer thickness or an index of refraction. Ellipsometry is an indirect technique of surface characterization that uses a numerical model of the sample for calculation of the expected values of Δ and Ψ. The physical sample properties of interest are floating parameters of this model and their final values are estimated by finding the best match of the numerical model and the experimental data. So despite ellipsometric maps can be considered as images in many ways, it is still necessary to extract the numeric value of their pixels in order to transform them into something with more physical meaning.



EP4Model 3.1.0.4 based on: C:\Users\mdu.ACCURION\Desktop\DataStudio-Test\DS_3_3_0_15\EP4M_models\latest_model

DS_3_3_0_15

MATERIALS	value	min	max	unit
Air				
File				
Air				
Si-Crystalline Silicon				
File				
Si-Crystalline Silicon				
SiO2-Silicon Dioxide				
File				
SiO2-Silicon Dioxide				

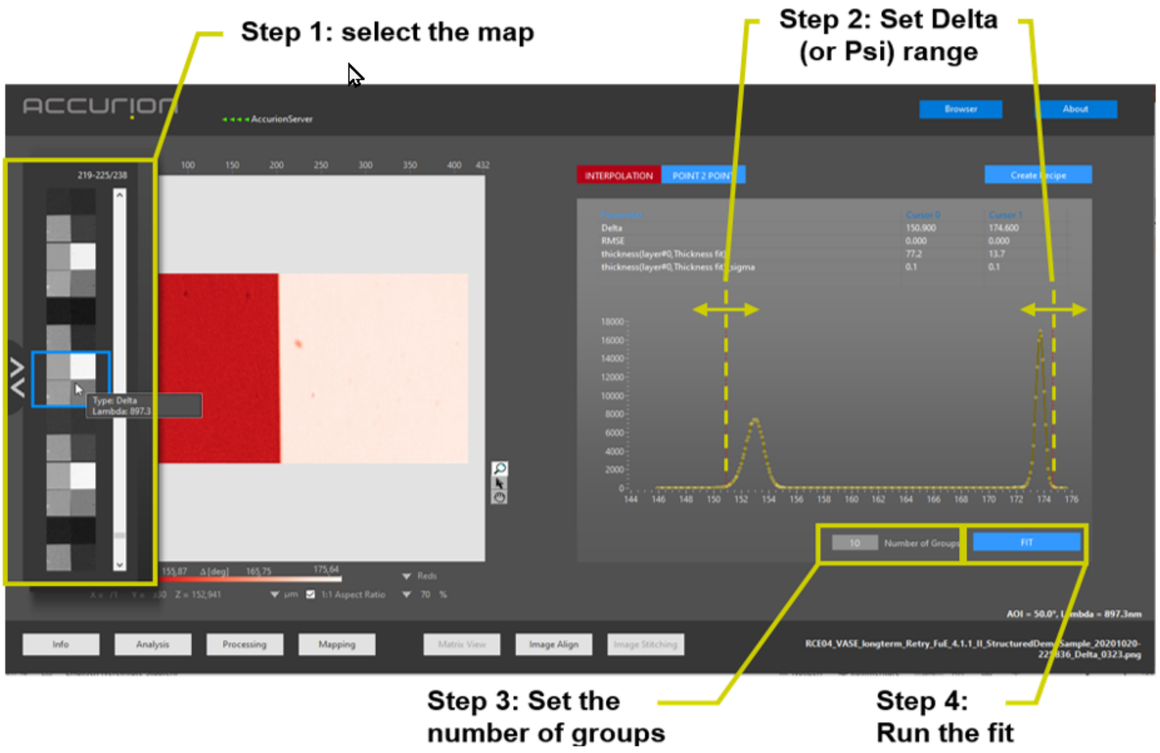
SAMPLES	value	min	max	unit
SiO2 - Si				
Air				
layerstack				
SiO2-Silicon Dioxide	#0			
thickness	121.6	10.0	150.0	nm
Si-Crystalline Silicon				

SETTINGS	value	min	max	unit
lambda	532.1	400.0	900.0	nm
bandwidth	5	0	20	nm
AOI	50.000	40.000	90.000	°
AOI_offset	0	-1	1	°
Theta_stage	0	0	360	°
Theta_offset	0	-180	180	°

DATA & FIT	value
datasets	
✓ M_II_M03_OldCalibr_Nulling_SiO2 DemoSample - G10_105nm	SiO2 - Si
✗ M_II_M03_OldCalibr_Nulling_SiO2 DemoSample - G10_105nm	SiO2 - Si
Fit for:	
✓ thickness(layer#0,SiO2 - Si)	local
Fit settings	
iteration limit	1000
tolerance	1E-7
use...	n/a

A typical ellipsometry measurement will have several data points taken at different wavelengths. The larger the number of points, the more accurately the model can be fitted. So when it comes to ellipsometric maps, it is easy to end up having to handle large sets of maps, which then need to be passed to the model software in order to retrieve the fitted parameter and build one single map with a meaningful parameter in its pixels. Currently Accurion's software provides two ways to handle this task:

1. Fitting "point-to-point": in this approach the software creates a vector of delta-psi values for each pixel in the map, creating the delta-psi vs. wavelength curves which are fitted in the model software. This method is the most accurate, but it is also way too slow. For example, for a measurement with 50 wavelengths and maps of 400x300 pixels the software will have to fit 120.000 vectors of 100 elements each, split them in the two curves of 50 elements each, fit the model and retrieve the fitted parameter for that pixel-vector. Assuming a duration of 0.1s per pixel-vector (which would be a fast fit!), the combined fit time for the entire map would be 12000 s approx. 3.5 hours!. This will scale up fast depending on the size of the maps, the number of wavelengths and the complexity of the model.
2. Fitting by interpolation: this mode is a very fast mapping mode that may be used for very simple mapping tasks that feature only a single fit parameter. It is based on the idea, that – for some applications – one may translate a single-wavelength Delta or Psi value directly into a physical quantity such as a layer thickness or a refractive index. By definition, the interpolation mode requires exactly one fit parameter in the modelling recipe in order to calculate the one-on-one relation between Delta or Psi and the fit parameter. This mode is suitable only for some application cases, as as mentioned, but also requires some intervention from the user, who has to: find out whether the sample is suitable for this mode or not, and in case yes, which parameter -delta or psi- is going to be used; then select the region of the histogram of that map that the software is going to use to build a look-up table with the calculated values after the fitting.



Step 1: select the map

Step 2: Set Delta (or Psi) range

Step 3: Set the number of groups

Step 4: Run the fit

Parameter	Cursor 0	Cursor 1
Delta	150.900	174.600
RMSE	0.000	0.000
thickness(layer#0,Thickness fit)	77.2	13.7
thickness(layer#0,Thickness fit)_sigma	0.1	0.1

Number of Groups: 10

AOI = 50.0°, Lambda = 897.3nm

RCE04_VASE_longterm_Retry_FuE_4.1.1_II_StructuredDemoSample_20201020-225836_Delta_0323.png

The two options for fitting complete maps have major drawbacks: the very fast interpolation method is very limited in its application and the very accurate point-to-point method is most often too slow. Currently the only alternative is that the user scrolls through the map stack, selects any region containing structures or objects and crop it, making a faster point-to-point fitting possible as there will be much fewer pixels to fit. This alone is already time consuming and challenging, as the objects can have a differentiated contrast from the background or from other objects only at some wavelengths, so at some maps it might be difficult to find interesting objects or sections in one glance.

The aim of this project is to explore the possibilities of clustering algorithms, which are usually applied to images, to automatize the analysis of large sets of maps. The segmentation of ellipsometric maps could be used to:

- help the user to find interesting objects or regions in the maps
- extract the numeric values of the segmented areas, creating the delta-psi vs. wavelength curves automatically
- provide a way to get a coarse fit of a whole map fast, applicable to all kind of samples and map sizes, with minimal intervention from the user

This way the fitting of segmented maps would be a good balance between the interpolation and point-to-point methods in terms of time required and accuracy of the result, while still being applicable to all kind of samples. It would also reduce the work and attention required from the user and help him in the decision making when it comes to choosing which parts of the map are most interesting for further analysis.

Challenges presented by ellipsometric maps

- Accurion's ellipsometric maps are saved as .png files which can be displayed normally in any OS with any software reading .png image format. However, the arrays of binary information contained in those images are not directly readable by any traditional image handling libraries in python. They need to be decoded into readable arrays of float32 numbers, which will be then mono-channel images with real delta or psi values any python library can deal with.
- Because of the method used to collect the data and build the ellipsometric maps, there will be empty pixels with NaN instead of a delta-psi value. This NaN pixels must be removed before the image array can be processed, as they are a no-go for many libraries and methods involving math calculations. The values replacing the NaNs should be as similar as possible to their surrounding, in order to introduce as less noise as possible in the maps.
- The datasets will always contain a variable number of individual maps which have either delta or psi values in their pixels. Despite these delta and psi maps should be paired, as each measured wavelength produces one delta and one psi map, there is no good a-priori way to know whether a map is delta or psi or at which wavelength they were collected. Only one measurement method includes "Delta" and "Psi" in the file naming, but for other methods the nomenclature will be different. The "metadata" relating wavelengths, map types and file names is only retrievable from a single .dat file which comes along the .png files.

Methodology

Given the nature of the data and the goal of the project, it was necessary to create a data treatment tool which is flexible enough to be used with any dataset from any sample or measurement type. It should maximize the information which is automatically given to the user, helping in the analysis and decision making while giving the tools to extract meaningful data.

The chosen approach was to pack all methods for data pre-processing and then build a class which can instantiate a folder containing all .png files and one .dat file corresponding to one ellipsometry measurement. By doing so, the class will automatically pre-process all maps, making them ready for their treatment. The class will also have all necessary attributes and methods built-in, including segmentation algorithms, plotting and numeric data extraction methods. All of this will work in the background. The front-end is an interactive interface built with the **streamlit** library, in which the user just needs to enter the path to the folder containing the measurement files and start playing with the data. The class module is called *clustering_elliMaps.py* and the pre-processing methods are packed in the module *astroclean.py* (there are several methods although only one of them is used in this project; the rest were built during tests to get to the optimal results and could still be used in the future in different approaches to this problem). Both modules are in the folder package: *elliPack*. For quick reference, a list of attributes and methods of the class can be found in the file *class_description.md*.

Despite we can refer to our data as images and they can be treated like single-channel (or "gray") images in some ways, the data type and the need to keep the precision of the decimal part of the numbers made the use of typical image-handling libraries such as **cv2** or **scikit-image** almost impossible (despite all the time and effort spent trying to go that way). The final goal is to extract some numeric data and for that our data can be handled as traditional **numpy** arrays and the segmentation with K-Means proved to work just the way it was needed.

1. Pre-processing data

After instantiating a folder path to the ***LambdaVarEllimaps*** class, it will automatically locate and read the .dat file inside using **pandas** and **os** libraries, in order to get the list of wavelengths measured, identify the files for the delta and psi maps and match them. This creates a list of files where delta and psi maps are shuffled: a delta map is always followed by its corresponding psi map for that wavelength.

Then the load method of the class is applied iterating over the file list. This load method first uses the *imread* module from the ***nanofilm package*** which transforms the raw maps from binary to readable **numpy** arrays of float32 type. Then it uses the **astropy** library to convolve a Gaussian 9x9 kernel over the maps and replace the NaNs by the mean value of the kernel at the centre pixel. This library was chosen because of its good performance, as it is specially designed to remove bad data like the NaNs from images and being able to handle float data type. More information about how this convolution and NaN removal is applied can be found here and here. Despite the good performance of this method for removing the bad pixels, it has some limitations: if there are NaN areas in the input image larger than the kernel, the interpolation will fail or produce "outliers" introducing noise. This is more remarkable if the NaN large areas are located at the edge of the images. This is intrinsic to the method, as for those cases there is no good way of setting the boundaries for the convolve function so that it does not introduce noise in some way. For our case, it is still possible to manually change the kernel size used by the load method, but the boundary settings were left as the defaults used by *astropy's* convolve function: fill with zeros.

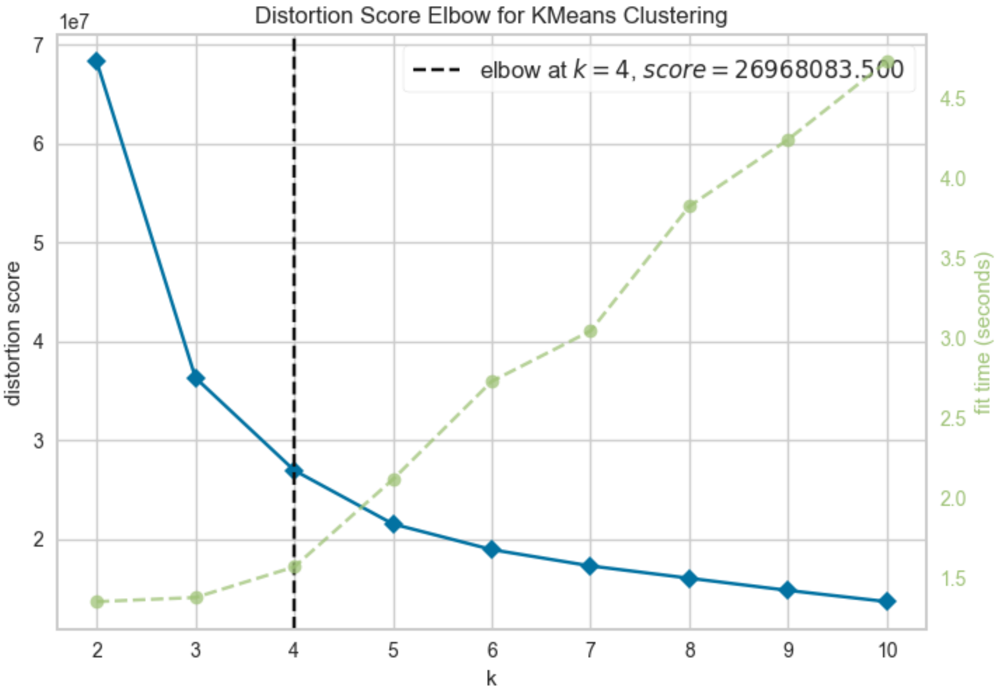
The load method also set some important class attributes which will be used later. It piles the processed maps into a stack using *numpy.dstack*. The dimensions of this stack will be (rows, cols, nWL*2), being nWL the number of wavelengths of the instantiated measurement. Basically this process transforms the whole dataset into an image of (nWL*2) *"color channels"*. Then reshapes the stack to have it the right dimensions to be passed later to the clustering algorithm (rows*cols, nWL*2), storing the stacks in two different attributes. The dimensions are also stored as attributes as they are used later on the code for proper indexing of some other objects.

2. Pixelwise segmentation using K-Means

One of the goals this project is aiming is a typical problem of unsupervised ML, being image segmentation a typical example of the use of data clustering algorithms. This have been addressed using **K-Means** algorithm from **scikit-learn**, which is one of the most commonly used in image segmentation. Other clustering algorithms like **DBSCAN** were also tested but the results were really poor compared to K-Means.

The application of **K-Means** to image segmentation is quite straight forward. The image must be reshaped to (rows*cols, dim3) and pass it to the algorithm along with the number of clusters we want it to divide the pixels into, named *k*. By default, the algorithm uses the *k-means++* initialization method, which selects initial cluster centers for the clustering in a smart way to speed up convergence. After fitting the K-Means model, it is possible to retrieve the cluster centres and the labels of the clusters elements, so that a segmented image can be rebuilt after reshaping again to the original dimensions. This is all done automatically with the class method ***clusterize()*** which takes only the desired *k* as argument and applies it to the reshaped image stack already stored as class attribute. As mentioned above, we can consider our data as a multichannel image of nWL*2 "colors". But it is necessary to access the pixel values and coordinates of each "color" individually after reshaping back, which conditions the data handling done after the segmentation.

The main limitation when using the K-Means algorithm is to know the optimal number of clusters into which the data can be grouped with best results and performance. This is usually done with the *Elbow* method, which helps to select the optimal number of clusters by fitting the model with a range of values for *k*. If the line chart has an inflection point (an "elbow"), it is an indication that the underlying model at this point fits best. This has been implemented using the **Yellowbrick** library, which provides a good and easy way of visualizing this clustering scoring curves. It also provides an estimation of the processing time required for each *k* considered and finds the "elbow" which likely corresponds to the optimal value of *k*. The class methods ***getEstimation()*** and ***getEstimation2()*** will produce the *yellowbrick* visualizers using the scoring methods of distortion and Calinski-Harabasz respectively. The distortion method computes the sum of squared distances from each point to its assigned centre, while Calinski-Harabasz computes the ratio of dispersion between and within clusters. Both estimators work with a *k* range between 2 and 10, as a larger range would increasing the processing time of the estimation more than it was desirable. Once run, the estimator figures are saved as .jpg files in the same instantiated folder, to that they can retrieved and consulted later without having to run the front-end or the class method again.



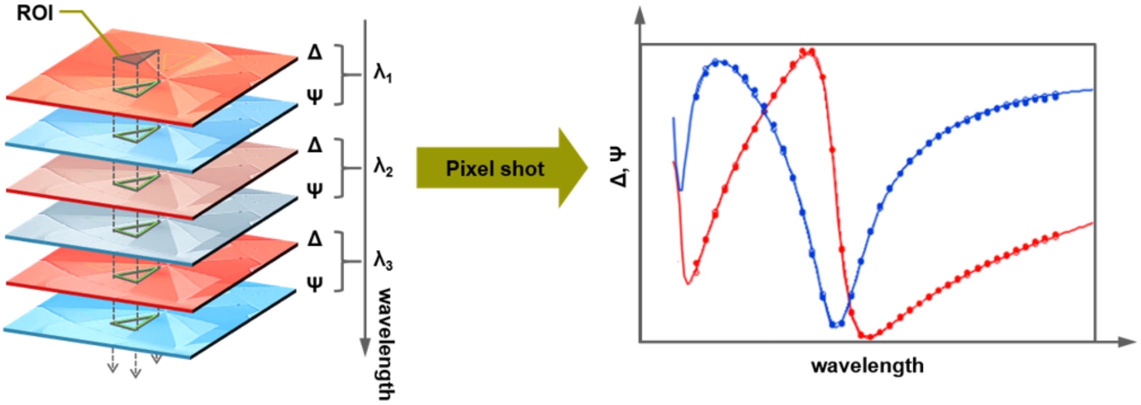
However, the "elbow "method does not work well if the data is not very clustered or if the data values are quite similar to each other. In this cases, the result will be a quite smooth curve with an unclear inflection point. Also, the automatic finding of the "elbow" provided by *yellowbrick* is sometimes wrong, or at least counterintuitive for a human looking at the curve. Especially in the Calinski-Harabasz metric curve. Both metric visualizers have been implemented for the user to have a **help** when deciding the optimal number of clusters, but the outcome should be considered as a hint and not as a rule, as due to the nature of the data it is many times difficult to find an inflection point in the curves.

3. Extracting numerical data out of maps with "clustershot"

One of the goals of the project is to extract the numerical data out of the segmented map stack to build the delta-psi vs. wavelength curves which can later be fitted using Accurion's model software. This is done with the ***clustershot()*** method, which is called automatically along with ***clusterize()***. This method picks the first map of the stack and then uses *numpy.unique* and *numpy.where* to extract the desired information (coordinates of all pixels of the cluster, delta and psi values of that cluster at each map) by iterating first over the cluster list and then over the maps of the stack with *for* loops. The data are appended to lists which are stored as the class attributes *self.delta_shot*, *self.psi_shot* and *self.cluster_coordinates*.

The delta and psi "shots" combined with the wavelength values are the data needed by the model software to find the desired parameter (thickness or optical properties of the surfaces visualized in the maps). The fitted parameter could be then combined with the cluster_coordinates to build a single image with relevant information. Currently it is not possible to communicate directly the model software with any external apps, so this can't be done automatically. But the class method ***exportDFs()*** provides a way to export the data into two .dat files, one containing the *clustershot* data and the other containing the cluster coordinates. This way the *clustershot* data can be loaded manually to the model software for the fitting and then use the cluster coordinates to build the new map after retrieving the fitted parameter from the model.

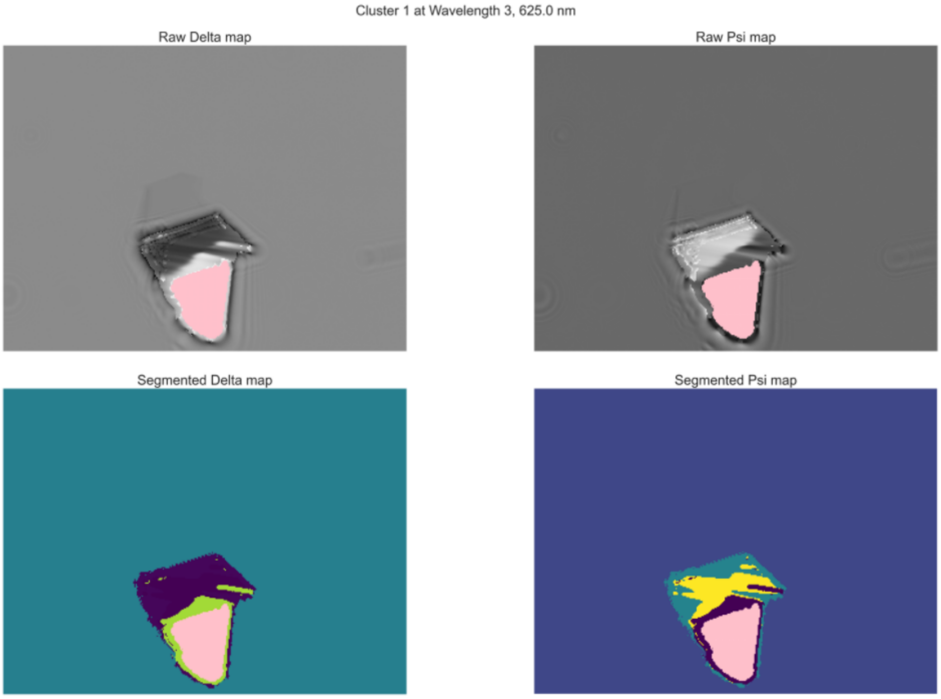
This *clustershot* method on the segmented maps comes to replace and improve with automatization the manual method which is currently available in Accurion's software: the user could load a map dataset into the software, scroll through it looking for interesting objects, manually draw one or more regions of interest (or ROIs) and then perform a so-called *pixel-shot*, which would extract the delta and psi curves by averaging all pixels inside the drawn ROIs.



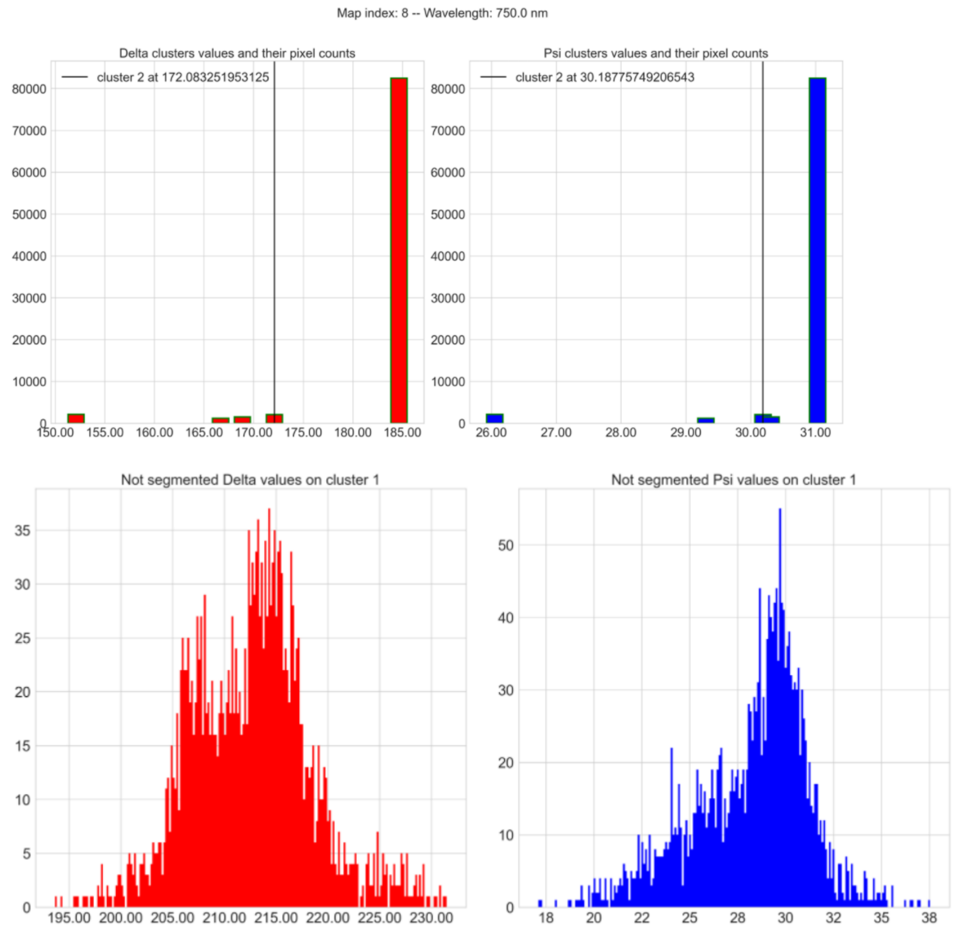
Plotting data

Some plotting functions have been integrated in the class in order to visualize the maps and the extracted data in a meaningful way for the user. All plotting methods of the class have been built using **Matplotlib**.

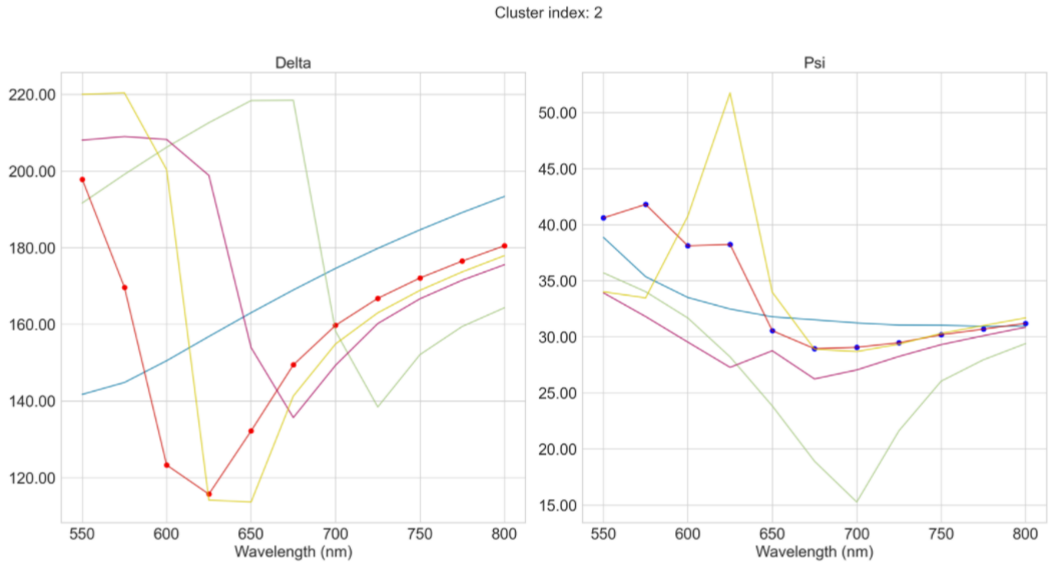
The image plot functions are ***self.plotDeltaPsi(idxSelector)***, ***self.plotSegmentedDeltaPsi(idxSelector)***, ***self.plotClusterOverMaps(C_Selector, idxSelector)***. They allow to visualize the raw delta and psi maps at selected wavelength, the segmented delta and psi map at selected wavelength and all pixels of a chosen cluster overlaying the raw and segmented maps at selected wavelength. The last plot can be especially useful as it allows to see where exactly the selected cluster is located in your raw map, which might not always be obvious when only looking at them.



There are two other plots which will show useful information to the user. ***self.plotBarSegmentedMap(C_Selector, idxSelector)*** will show the relative size of the clusters are and how close their values are. If some clusters are overlapping or very close, it could be an indication that the chosen number of clusters was not totally correct. ***self.plotClusterRawValues(C_Selector, idxSelector)*** will show the histogram of raw values of the pixels belonging to the selected cluster. If the raw values of the cluster have a wide distribution and more than one peak, it could be an indication that there are smaller groups inside the cluster that were "unified" in the segmentation process. The user might want to re-run the segmentation with a larger *k* or study that cluster more in deep in Accurion's software.

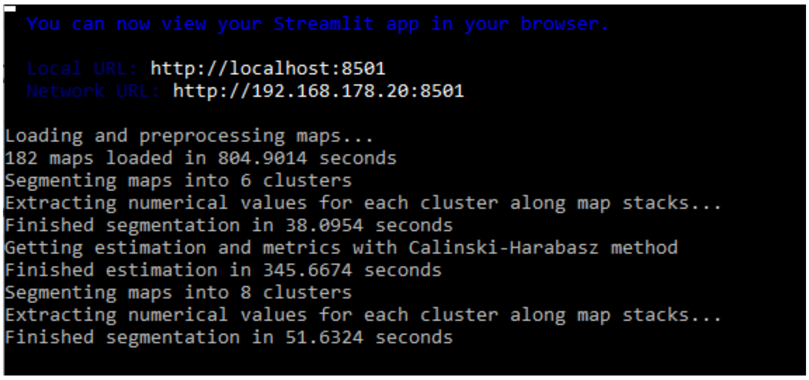


There is one more plot available which will show the user all clustershots available and the one selected: ***self.plotAllShots(C_Selector)***. This will give a trained user, who knows what values and shape of curves to expect, information about what materials could be found inside the selected cluster.



4. Interactive interface

An **streamlit** app has been built to serve as an interactive front-end for the user. After startup the user is asked for a folder path containing the measurement data. When the path is passed, the app will instantiate it as a LambdaVarEllimaps class and run all the map pre-processing automatically. This is the most time consuming part at the moment. For large sets it can take several minutes (up to 30 in the largest set tested so far) but the time consumed by the pre-processing depends strongly on the map size and the number and size of NaN areas. The estimation of the number of clusters can also take a few minutes. The segmentation itself rarely takes more than one minute. Here is an example of the time to process a large dataset:

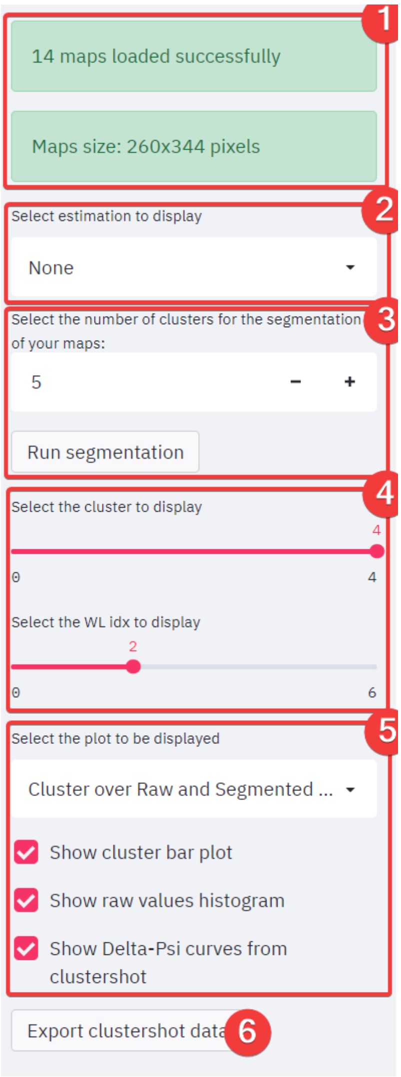


```
  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.178.20:8501

Loading and preprocessing maps...
182 maps loaded in 804.9014 seconds
Segmenting maps into 6 clusters
Extracting numerical values for each cluster along map stacks...
Finished segmentation in 38.0954 seconds
Getting estimation and metrics with Calinski-Harabasz method
Finished estimation in 345.6674 seconds
Segmenting maps into 8 clusters
Extracting numerical values for each cluster along map stacks...
Finished segmentation in 51.6324 seconds
```

A side panel provides the user with all the tools to treat and display the processed data.



14 maps loaded successfully

Maps size: 260x344 pixels

Select estimation to display: None

Select the number of clusters for the segmentation of your maps: 5

Select the cluster to display: 4 (0–4)

Select the WL idx to display: 2 (0–6)

Select the plot to be displayed: Cluster over Raw and Segmented ...

☑ Show cluster bar plot

☑ Show raw values histogram

☑ Show Delta-Psi curves from clustershot

1. Info section. Shows number of maps loaded and maps size
2. Estimator selector. Here the user can choose which estimation metric to visualize, distortion or Calinski-Harabasz. Asmentioned above, depending on the dataset size this can also take from few seconds to several minutes. After getting the estimation it is better to set the selector box back to ***None***, otherwise the rest of the app will be slowed down.
3. Number of cluster input and segmentation launcher. The user can enter the desired number of clusters in the box and then launch the segmentation of the dataset with the button **Run segmentation**. This step is normally quite fast.
4. Wavelength and cluster selectors. With these sliders the user can choose which wavelength and cluster index to display in the plots selected in the next section. The cluster selector slider will not be available until the segmentation has been run.
5. Plot type selector. In the selector box the user can choose the type of map plot to be displayed. With the chexboxes the user can choose to display any other plot with additional information about the numeric data. Again, an error will be raised if the segmentation has not been launched and a plot using segmentation outputs is selected.
6. Button to export the data in two .dat files as described above.

Summary and conclusion

It was possible to overcome the challenges presented by the data and make them suitable to be processed in python by the traditional libraries. The segmentation with K-Means algorithm can be applied to ellipsometric maps, which are processed as images of *n* channels and float32 data type. With the segmentation the number of pixel-vector in the dataset is reduced from hundreds of thousands to a handful of values. This is naturally reducing the z-resolution (the fine differences in pixel values) of the maps many times, but the result is good enough in many cases.

Compared to the interpolation method currently used in Accurion's software, this method increases the application cases, and significantly reduces the effort required for the user to explore and get insights from the data, introducing a lot of automatization and ease of use. The loss in z-resolution is compensated by the fast data exploration and the automated way to inform the user about the possible missing intra-structures.

Despite there is still room for improvements in many ways, especially concerning the performance of the pre-processing and the front-end, the main goals of the project have been fulfilled. It was proven that ML algorithms can be used also for our very especial kind of data, accelerating data exploration and handling. In the traditional trade-off between accuracy and speed when analysing the data produced from imaging ellipsometry, this method keeps being fast and sacrifices resolution only compared to the slow point-to-point method. On the other hand, it broadens the application cases for a fast method by allowing to use both delta and psi for the model fitting. It also facilitates and automatizes significantly the data exploration, which goes in the direction of what customers are currently demanding: one (or at least fewer) click solutions for analysing the complex data produced by imaging ellipsometry.