
Development of defect detection tool on ultrasound testing images

Internship

Presented by

Youness El Houssaini

Born in Ouled Teima

University of Strasbourg

Faculty: UFR mathematics and informatics

Major: scientific computing and mathematics of information

Examiner: Dr. Christophe Prud'homme

Freiburg Im Breisgau

August 2019

Confidentiality

The present work entitled "**Development of an automatic tool for defect detection on ultrasonic testing images**" contains confidential data of the Fraunhofer Institute for Mechanics of Materials IWM.

The work may only be inspected by the first and second supervisors and is available for authorized members of the selection board.

A publication and duplication of the work is not permitted - even in excerpts.

An inspection of the work by unauthorized persons requires the explicit approval of the author and the Fraunhofer Institute for Mechanics of Materials IWM.

Abstract

Within the scope of the present work, a defect detection tool has been developed in order to assess the formed defects occurred during the manufacturing process of forged components. This tool is based on data derived out of available Ultrasonic Testing images. The tool defines position of defects and their shapes in order to map the mechanism of fatigue damage. A 3D reconstruction of the distribution of defects is also proposed within this work. For this purpose, some approximations have been made from an expertise point of view. For instance, only severe damages have been treated within this work. Also, the accuracy of the detection tool is based on some key parameters e.g. the color threshold and the extrema of each defect.

The detection tool is developed using Python coding language with its image processing package Opencv for image pre-processing amongst others. Also, some functions from the lecture of Dr. Vigon, Vincent “Signal and graph”, were adapted or/and further developed. In order to make this work usable and accessible to anyone a GUI was also made based on PyQt5 package.

In this work it is shown that the detection tool gives better results when the UT images contains separated single defects and loses its accuracy to some extent when the defects are crossing each other in the given image. This crossing phenomenon is due to including defects of inner layers down the component and not only the outer surface.

Foreword

This report is part of the study program at the University of Strasbourg in the UFR Mathematics and informatics within the specialization of scientific computing and mathematics of information, CSMI. The work is considered as a continuation of the “Semester Project” taught by Dr. Céline Caldini Queiros. Thanks to whom this work had a fundamental start with her priceless advices. This work is considered as an internship that is an opportunity to bring into practice all the theoretical lectures along the first year of the Master. It took place at the Fraunhofer Institute for Mechanics of Materials IWM from the 1st June 2019 to the 1st August 2019 (For two months full time). The topic was accepted as a semester project and then as internship thanks to the agreement of the study program (CSMI) manager Prof. Dr. Prud’homme Christophe, to whom all my gratitude. He is also the examiner of this work.

First and foremost I would like to thank the Fraunhofer Institute for Mechanics of Materials IWM that gave me this opportunity of evolving in both academic and professional scale. In addition, I would like to express my special thanks to my supervising professors, Dr. Vigon Vincent and Dr. Yannick Privat for the exemplary support by the University of Strasbourg. Also, my full gratitude goes to my project manager and supervisor Dr. Igor Varfolomeev, M.sc. Aydin Ali and Dipl.-Ing. Dittmann Florian for their enduring support and helpful insight.

Furthermore, my thanks to all employees in the project team for the helpfulness and friendliness shown to me during my work. Thanks to whom the working conditions and the atmosphere were extremely positive.

Contents

| | |
|---------------------------------|----|
| 1. The project | 9 |
| 1.1 Introduction | 9 |
| 1.1.1 Context | 9 |
| 1.1.2 The task | 9 |
| 1.1.3 Methodology | 10 |
| 1.2 Preliminary steps | 11 |
| 1.2.1 UT image | 11 |
| 1.2.2 Image pre-processing | 12 |
| 1.2.3 Color space | 12 |
| 1.2.4 Image thresholding | 13 |
| 2. Image processing | 15 |
| 2.1 Defects collection | 16 |
| 2.1.1 Algorithm principle | 16 |
| 2.1.2 Results | 16 |
| 2.2 Data cleaning | 18 |
| 2.2.1 Artifacts | 18 |
| 2.2.2 Noise | 20 |
| 3. Segmentation | 22 |
| 3.1 Problem | 22 |
| 3.2 Case study: Finding extrema | 23 |
| 3.2.1 Defect mapping | 23 |
| 3.2.2 Mapping gradient | 25 |
| 3.2.3 Gradient minima | 28 |
| 3.3 Sub-defects detection | 31 |
| 3.3.1 Watershed algorithm | 32 |
| 3.3.2 K-means | 35 |
| 4. Covariance error ellipse: | 37 |
| 4.1 Necessary hypothesis | 37 |

| | | |
|-------|---|----|
| 4.2 | Confidence ellipse features | 39 |
| 4.2.1 | A geometric interpretation of the covariance matrix | 39 |
| 4.2.2 | The width, the length and the angle | 41 |
| 4.3 | Case study | 43 |
| 4.3.1 | 2D data | 43 |
| 4.3.2 | 3D data | 44 |
| 5. | Assignment of 3D coordinates | 46 |
| 5.1 | Methodology | 46 |
| 5.2 | The Python code | 47 |
| 5.3 | Assumption | 49 |
| 6. | GUI development | 49 |
| 6.1 | Introduction to GUI | 50 |
| 6.1.1 | Pre-requirement | 50 |
| 6.1.2 | GUI environments | 52 |
| 6.2 | GUI development | 54 |
| 6.2.1 | Overview | 54 |
| 6.2.2 | GUI options | 55 |
| 7. | Discussion: | 56 |
| 7.1 | Overview | 56 |
| 7.1.1 | Ontology | 56 |
| 7.1.2 | Performance | 58 |
| 7.1.3 | Validation | 58 |
| 7.2 | Enhancements | 61 |
| 7.2.1 | The key parameters | 61 |
| 7.2.2 | Other paths | 62 |
| 8. | Summary and outlook | 63 |
| 9. | Sources | 64 |
| | Appendices | 67 |

I. Nomenclature

| Symbols | Meaning | Unit |
|----------|--|-------|
| l | length of the component | mm |
| w | Width of the component | mm |
| h | height of the component | mm |
| r | Radius of a disc-defect | MPa |
| XY | The horizontal plan of the component | — |
| XZ | The vertical plan of the component | — |
| (i, j) | Coordinate of each pixel | — |
| n | Number of pixels in a single defect | — |
| S | The scale of the covariance confidence ellipse | — |
| a | The semi-major axis of an ellipse/ellipsoid | mm |
| b | The semi-manor axis of an ellipse/ellipsoid | mm |
| c | The semi-manor axis of an ellipse/ellipsoid | mm |
| α | The tilt of the defect | rad |
| r^2 | The coefficient of determination | — |

II. Abbreviation

| Abbreviation | Meaning |
|--------------|---|
| UT | <u>U</u> ltrasonic <u>T</u> esting |
| OpenCV | <u>O</u> pen source <u>C</u> omputer <u>V</u> ision |
| RGB | <u>R</u> ed <u>G</u> reen <u>B</u> lue |
| HSV | <u>H</u> ue <u>S</u> aturation <u>V</u> alue |
| LCF | <u>L</u> ow <u>c</u> ycle <u>f</u> atigue |
| maxVa | <u>M</u> aximum <u>V</u> alue |

1. The project

1.1 Introduction

This sub chapter explains the motivation of this work. Also we are going to seize the concrete description of the task with the definition of the goals to be achieved as well as the required tools.

1.1.1 Context

In mechanical engineering, the safe design of components is one of the most fundamental tasks, as their failure can have serious consequences. Known cases of damage in the recent past, such as for example, the catastrophic ICE accident in 1998 in Eschede and the tanker accidents of the tankers Erika 1999 and Prestige 2002 show that there is considerable need for research on the safe design of components. [1] Material fatigue plays a very important role in dynamically stressed components.

In fact, during the manufacturing process of forged components, the formation of defects is unavoidable. The failure rate of structural components is significantly influenced by these defects which can be detected by non-destructive methods. The aim of the overall research project is the optimization of the lifetime prediction of structural components by the consideration of nucleation process of forging defects. For this purpose low cycle fatigue (LCF) testing until fracture is carried out at specimens which are extracted from forged components in a way that a defect is positioned in the center. The fracture surfaces are then examined by means of scanning electron microscope (SEM) and energy dispersive X-ray spectroscopy (EDX). Within this procedure the defects can be recognized on the fracture surface. This test method provides images for probable defects as well as the chemical composition. The evaluation of these defects in the images is currently carried out manually and is very time-consuming, which shows the need of automation.

1.1.2 The task

The scope of the project is to establish an automatic tool for the detection as well as the classification of defects in fracture surfaces using UT images. This will have to be done in an iterative workflow including image pre-processing, definition of defects features and output

data processing. The programming language will be Python. As result, the tool should be able to detect defects in UT images and define its size and shape as well as the coordinates of its position so that a 3D reconstruction can be estimated.

1.1.3 Methodology

The work has been conducted within a group of four members communicating the results of each other regularly via an internal network. In the process of developing the detection tool, iterative and incremental framework for project management was essential. Based on the Scrum approach the following workflow was made in order to manage meeting the time schedule objectives (see Figure 1).

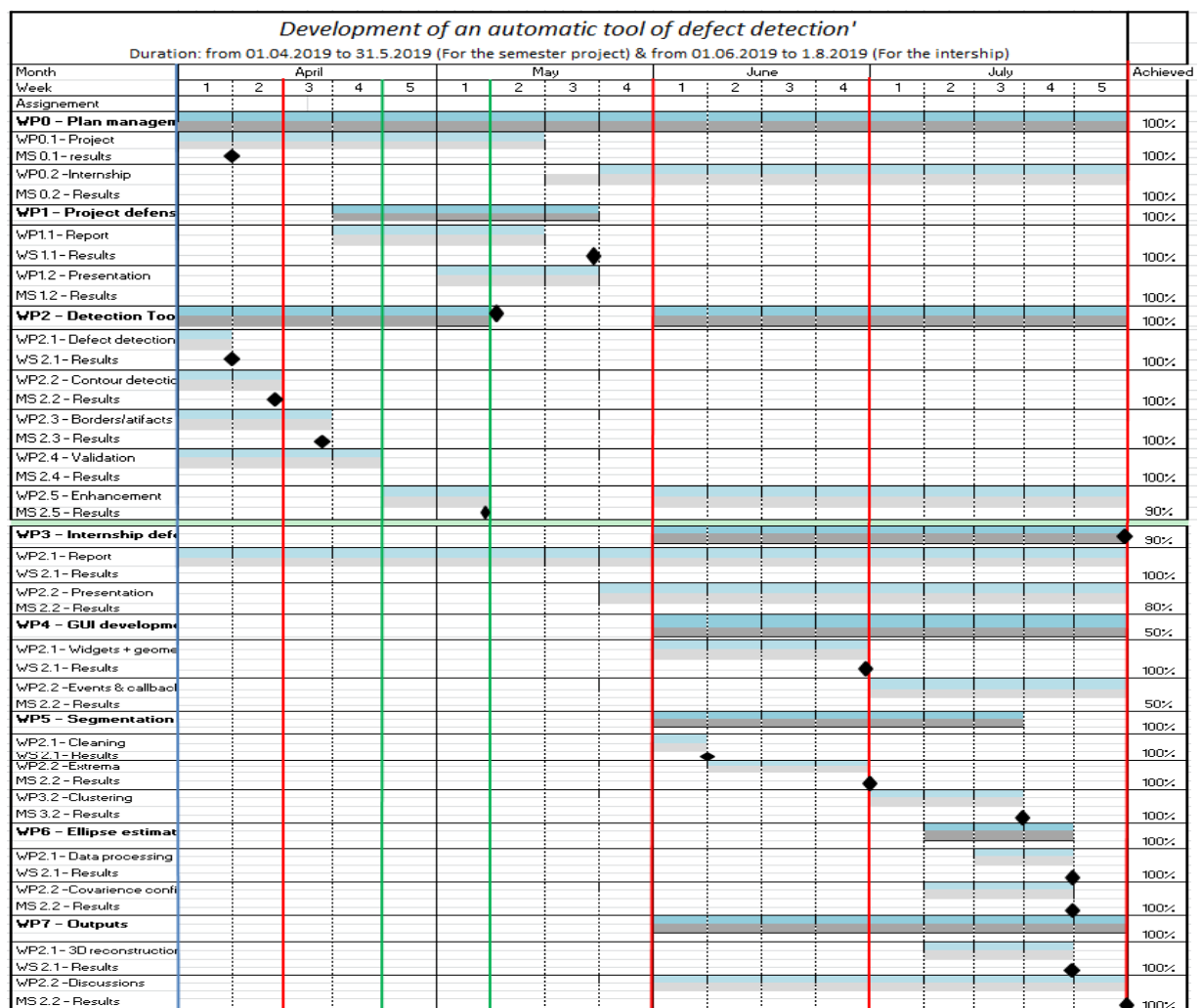


Figure 1 Screenshot of the project management plan.

This method defines "a flexible, holistic product development strategy where a development team works as a unit to reach a common goal", challenges assumptions of the "traditional,

sequential approach"[2] to product development, and enables teams to self-organize by encouraging physical co-location or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines involved.

1.2 Preliminary steps

In this sub chapter all obstacles in the process of developing the automatic detection of defects are explained. After selecting a suitable solution, the programmed tool is implemented in python coding language and presented step by step in the following chapters. In particular, this section presents encountered problematics from the pre-processing of the UT images until the output data extraction.

1.2.1 UT image

It's identical to ultrasound imaging (sonography) which uses high-frequency sound waves to view inside the human body. UT imaging technique does not have only medical application but also is used in material science. It is used to model the internal material structure in order to assess the lifetime prediction of structural components by the consideration of nucleation process of forging defects. Its aim is often to show potential probable defects to exclude unsafe components. For a better insight, an example of UT image is presented below (see Figure 2).

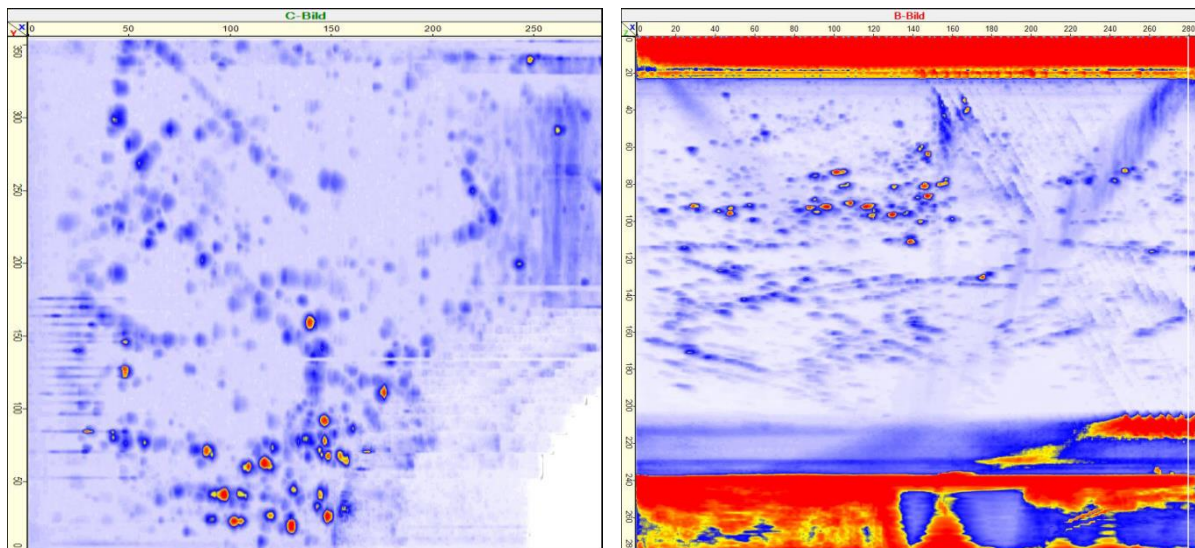
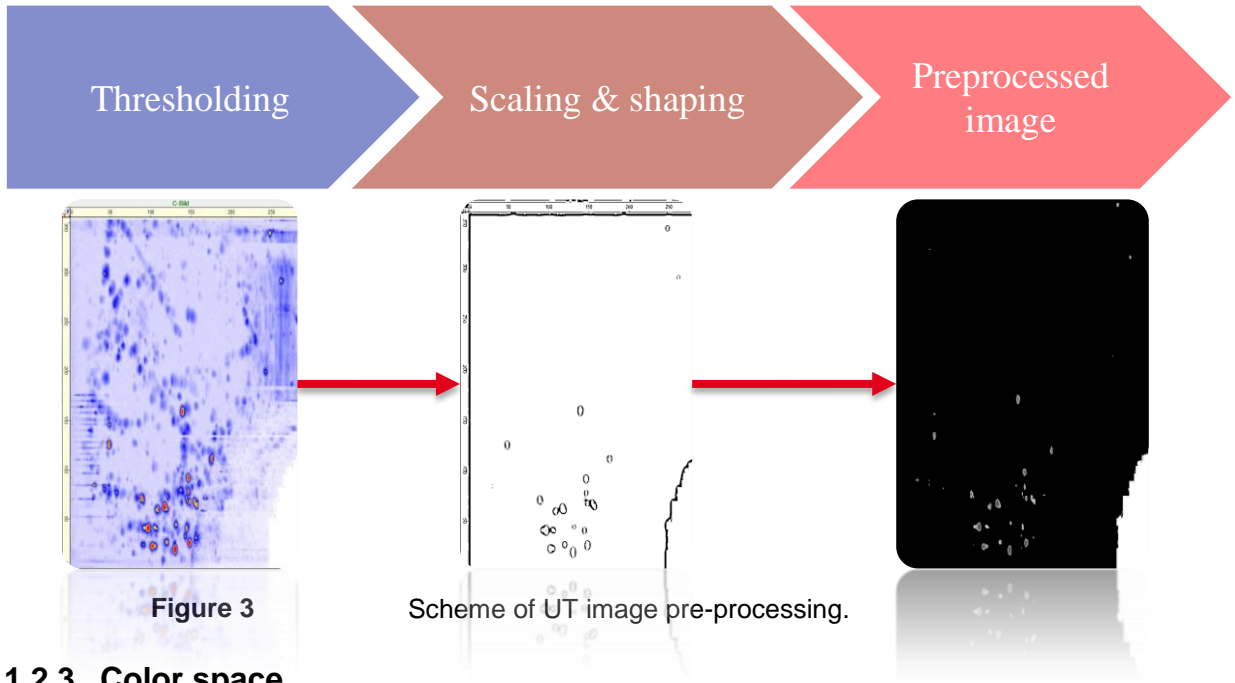


Figure 2 UT images of the project tested component, right the XY plan and XZ at left [18].

1.2.2 Image pre-processing

The UT image preprocessing is carried out mainly using OpenCV. It is a library of programming functions mainly aimed at real-time computer vision. The following chart illustrates the different steps before having the output image ready for data extraction (see Figure 3).



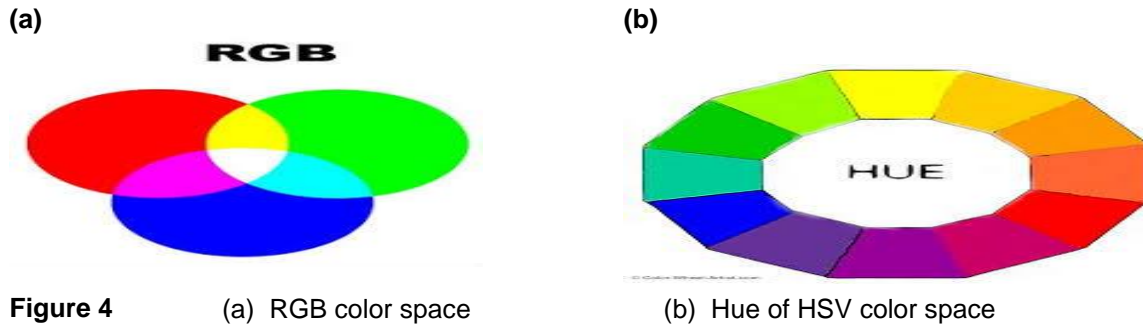
1.2.3 Color space

In order to read the UT image and reproduce it in a digital form a color space has to be chosen first. Since "color space" identifies a particular combination of the color model and the mapping function, the word is often used informally to identify a color model. A "color model" is an abstract mathematical model describing the way colors can be represented as tuples of numbers (e.g. triples in RGB or quadruples in CMYK). When defining a color space, the usual reference standard is the CIELAB or CIEXYZ color spaces, which were specifically designed to encompass all colors the average human can see. [3]

Why HSV?

HSV are alternative representations of the RGB color model, designed in the 1970s by computer graphics researchers to more closely align with the way human vision perceives color-making attributes. In these models, colors of each hue are arranged in a radial slice, around a central axis of neutral colors which ranges from black at the bottom to white at the top. This "color model" allows a better manipulation of the colors using a single parameter

called “Hue” (see Figure 4b). This is a key parameter in the UT image thresholding in the contrary to other “color models” when more than parameter is required. The RGB for instance is a combination of three parameters or colors to determine a new resulting color (see figure 4a).



Once the UT image is converted to HSV color model the yellow and red spots can be easily manipulated according to a certain value of the hue called “Threshold”.

1.2.4 Image thresholding

The OpenCV library offers many thresholding functions which will be defined and compared in this section.

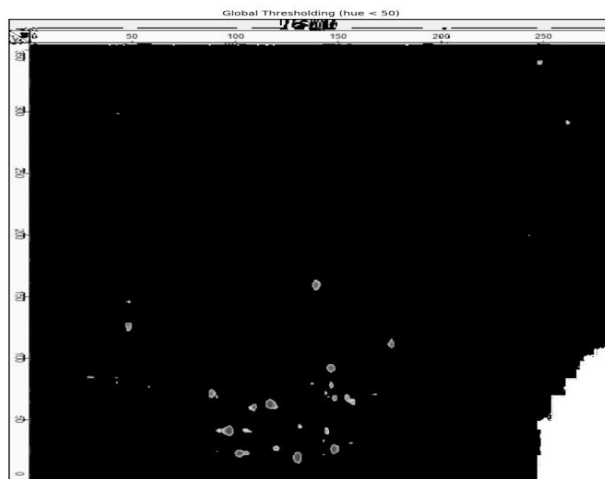


Figure 5 Obtained UT image after simple thresholding

Simple thresholding:

Here, the matter is straight forward. If pixel value is greater than a threshold value, it is assigned one value (Black in our case), else it is assigned another value (no change in our case). The function used is `cv.threshold`. First argument is the source image, which should be a grayscale image.

Second argument is the threshold value which is used to classify the pixel values. Third argument is the `maxVal` which represents the value to be given if pixel value is more than (sometimes less than) the threshold value. [4]

The figure above (See figure 5) is a result of the mentioned method above and the used code is in the appendices chapter. Color space conversion is the translation of the representation of a color from one basis to another one. This typically occurs in the context of converting an image that is represented in one color space to another color space, the goal being to make the translated image look as similar as possible to the original.

Adaptive thresholding:

In the previous section, we used a global value as threshold value. But it may not be good in all the conditions where image has different lighting conditions in different areas. In that case, we go for adaptive thresholding. In this, the algorithm calculates the threshold for small regions of the image. So we get different thresholds for different regions of the same image and it gives us better results for images with varying illumination.

It has three ‘special’ input parameters and only one output argument.

Adaptive Method - It decides how thresholding value is calculated.

- `cv.ADAPTIVE_THRESH_MEAN_C` : threshold value is the mean of neighbourhood area.
- `cv.ADAPTIVE_THRESH_GAUSSIAN_C` : threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.

Otsu’s binarization thresholding

In global thresholding, we used an arbitrary value for threshold value. So, how can we know a value we selected is good or not? The answer is: “trial and error method”. But consider a bimodal image (In simple words, bimodal image is an image whose histogram has two peaks). For that image, we can approximately take a value in the middle of those peaks as threshold value. That is what Otsu binarization does. So in simple words, it automatically calculates a threshold value from image histogram for a bimodal image. For images which are not bimodal, binarisation won’t be accurate. [4]

Adopted solution

In the following figure a comparison of the obtained images after using the three thresholding types mentioned above (see Figure 6). For better resolution, please check the appendices chapter.

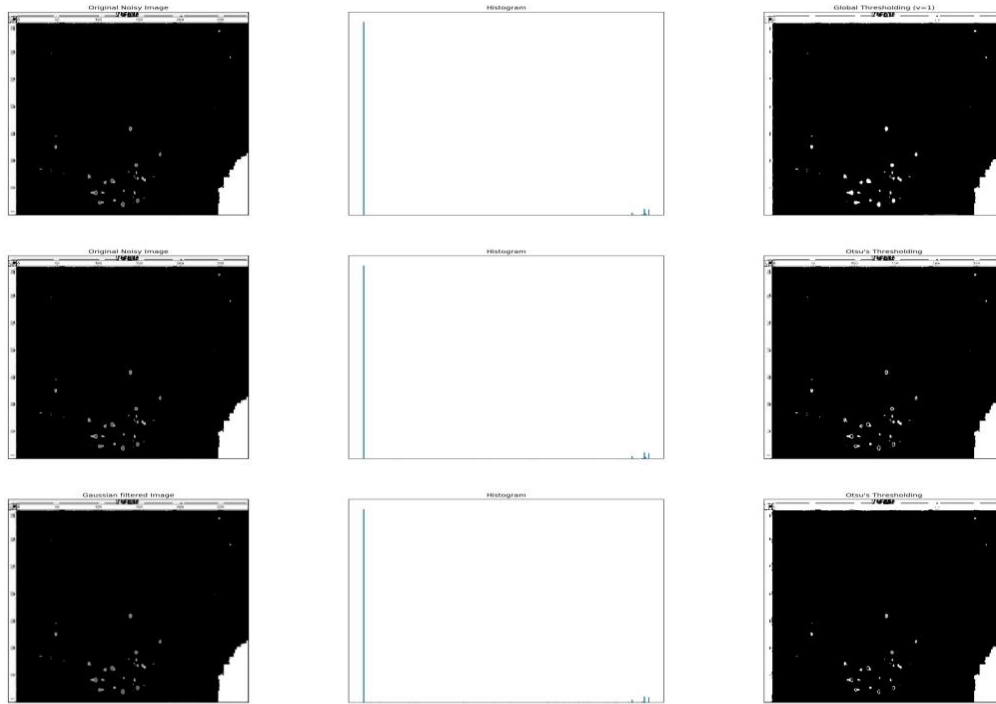


Figure 6 Original noisy image in the left and the color distribution in the middle and the thresholding output image on right side.

Given that the shown UT images doesn't have only two peaks and also they are colored images so the varying illumination is not the main factor. Therefore, the chosen thresholding is the global thresholding. Once the threshold is defined based on the color code of the HSV model it is applicable to all similar UT images.

2. Image processing

In the previous chapter, the pre-processed UT image is already performed and ready for processing. In this chapter, the aim is to extract information such as the center and the shape of defects. This is performed using an algorithm explained below. Furthermore, the obtained data of the extracted defects are cleaned in a way to get rid of noise and artifacts.

2.1 Defects collection

2.1.1 Algorithm principle

The principle of the detection algorithm is based on the chosen threshold or maxval which is the highest value of the Grayscale processed image considered as a defect. This can be summarized as follows:

If “the pixel value “ \leq “maxval”, then the pixel is considered as a defect.

This line is repeated iteratively in all neighbors of each pixel (i,j) within a single connected defect until the condition is not valid any more. Here is an illustrative chart of the defect collecting algorithm (see figure 7).

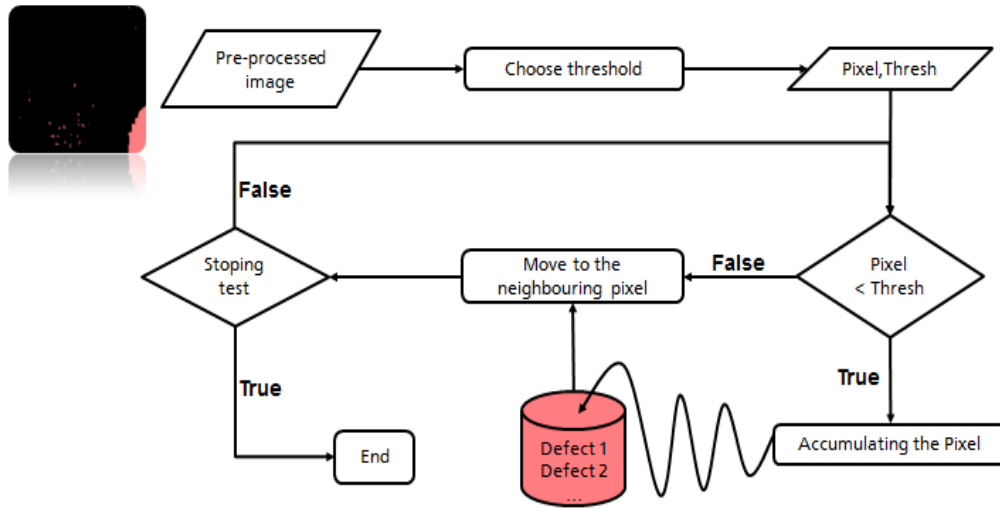


Figure 7 The algorithm flow chart of the image data extraction

The center of each defect is then calculated from the formula:

$\frac{\sum_{i,j}^n (i,j))}{n}$, n : number of defects and considered as a barycenter of all pixel. The shape of defects is considered in this chapter as a disc for simplification $\pi * r^2$. In the next chapters (chapter 3), the defects are approximated by ellipses and ellipsoids for respectively 2D and 3D data. For more details about the code, please check the appendices chapters.

2.1.2 Results

For the visualization, only the centers of defects are considered and thus compared with the validation data. This data is obtained manually by zooming into the defect and selecting its shape. The result of the validation should not be 100% identical to the validation data but at least satisfying.

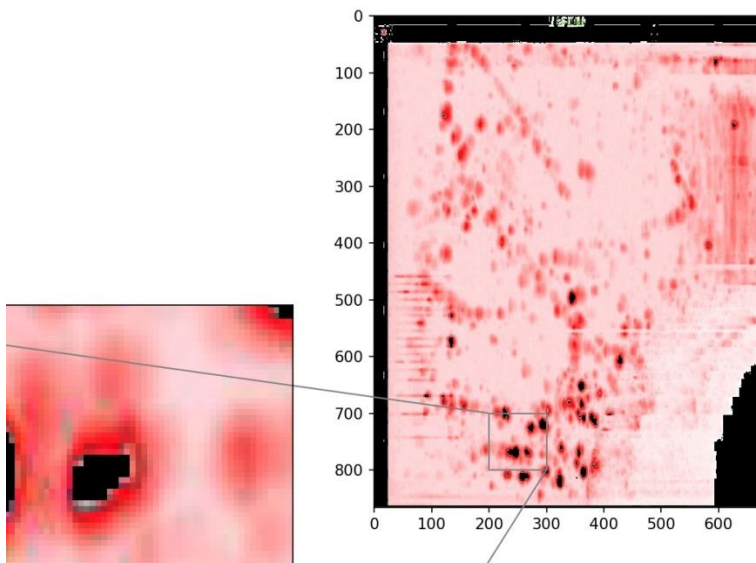


Figure 8 The image of the extracted data with a zoom to a single defect match.

Here is a representation of the obtained results for both XY and XZ 2D data.

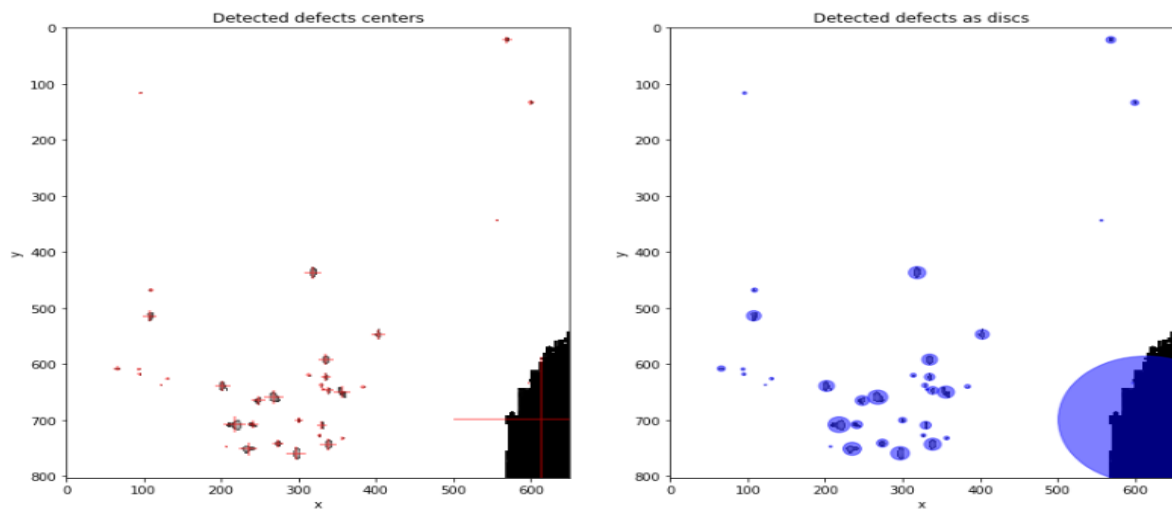


Figure 9 Detected XY defects centers in red and their shapes in blue.

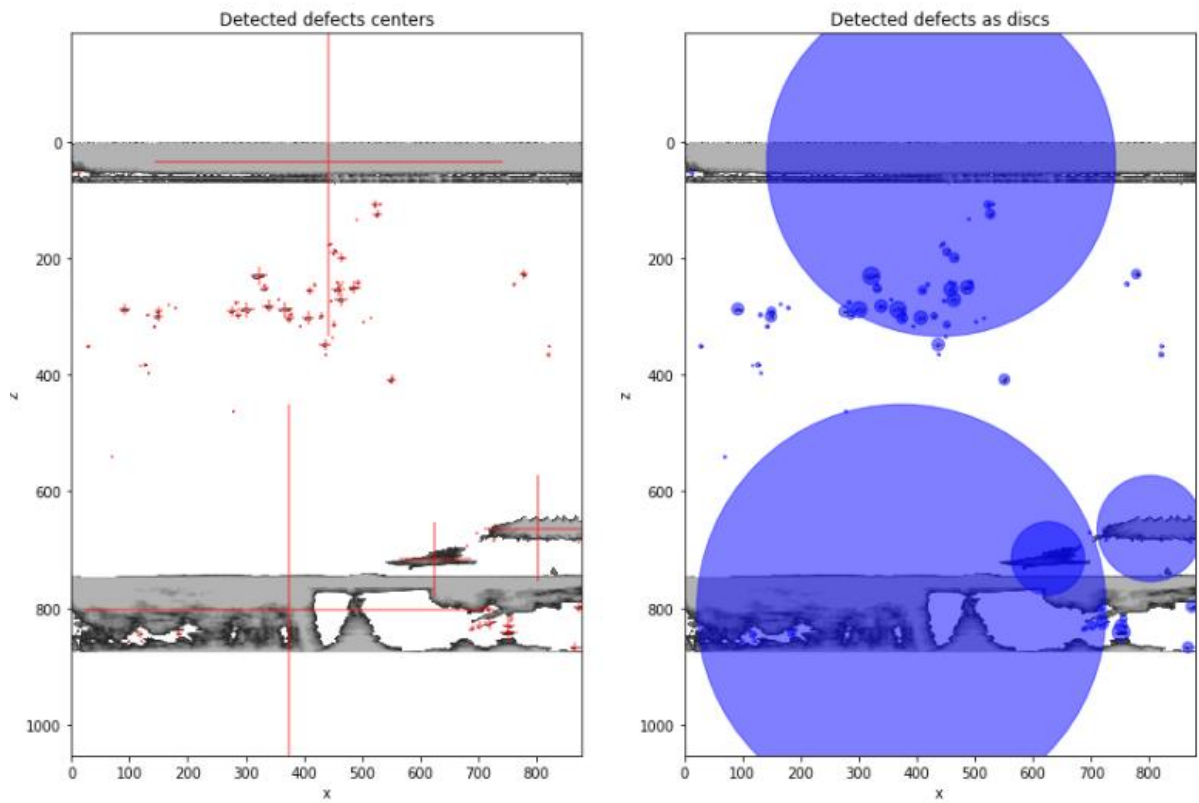


Figure 10 Detected XZ defects centers in red and their shapes in blue.

The defects contain a lot of false data due to artifacts and some noisy spots. This is processed in the next sub section.

2.2 Data cleaning

In this section we show how we clean the 2D data of XY and XZ UT images and make it ready to process. This cleaning step is made regarding two phenomena:

- ✚ Artifacts
- ✚ Noise defects

2.2.1 Artifacts

Some of the UT images contain artifacts or artefacts which could be the source of any error in the perception or representation of the real image, reproduced by the involved equipment. The phenomenon is presented in the following figure.

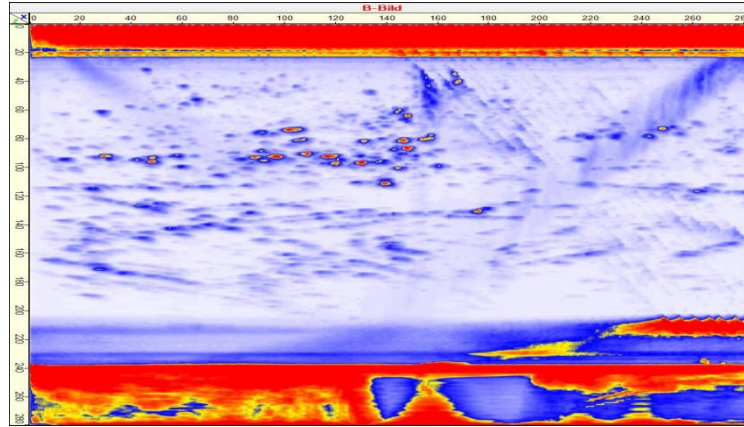


Figure 11 XZ UT image containing artifacts [18].

Those artifacts inducing false data are caused by the shape of the component under test as well as the UT process itself. In fact, the artifacts existing in the upper part of the UT image are merely due to the UT process. In the other hand, the artifacts in the bottom can be mainly due to the shape of the component. Therefore, the real shape of the component (given in $\text{mm } l \times w \times h$) is a necessary input from the user to get rid of such data.

The artifact is removed in the processing step using the “matplotlib.path” library and its function “contains_points”. It takes four coordinates or more of a polygon and returns nothing but the inside of the polygon.

Another way is also proposed in the tool is mentioning the limits of the artifacts within which there would be only defects. For instance, the input would be in that case of the figure above given as an interval [30,200]. Also, the artifacts can be avoided by putting a condition on the defects with exceeding a certain number of pixels as shown in the python code below.

```
def defects_output(img,x, y, size,alldefects,upper_intensity):

    alldefect = alldefects.reshape((alldefects.size//2,2))#i,j

    alldefect = alldefect.tolist()
    centers = np.stack((x, y), axis=-1)
    mean_intensity = []

    output = pd.DataFrame([[[]],[[]], [],[[]]].T

    drop=[]
    for i,j in enumerate(size):
        ieme_defect = pd.DataFrame(alldefect[0:j])
        i_max = np.max(ieme_defect[0])
        Intensity = get_defect_intensity(img,ieme_defect)
        mean_intensity = np.mean(Intensity)
        if mean_intensity > upper_intensity or j > 3*upper_intensity or i_max < z[0] or i_max > z[1]:
            drop.append(i)

        ith_line = pd.DataFrame([[centers[i,:].round(2), size[i],mean_intensity.round(2),ieme_defect]],
                                index = np.array([i]))
        output = output.append(ith_line)
        alldefect = alldefect[j:]

    output.columns = ['Centers in (x,y)', 'Area','Average intensity', 'the ith defect in (i,j)']
    assert(np.array_equal(np.array([output['the ith defect in (i,j)'][i].size//2 for i in range(len(size))]),size))

    output['Area']=output['Area'].astype(int)

    output = clean_defects(output,drop)
    print len(drop) , 'defects were cleared '

    return output #i,j
```

2.2.2 Noise

There is another way to consider false data as defects, that is taking into accounts tiny defects with high 'value' or brightness in the HSV color space scale. The following figure gives an idea about such noise.

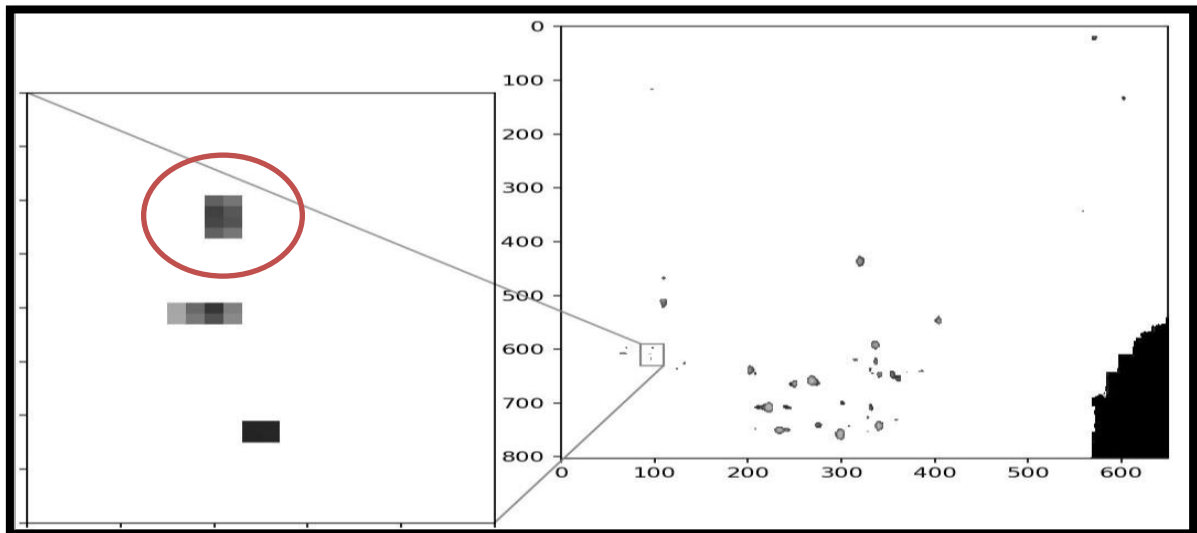


Figure 12 A zoom into an area with noise signal in XY processed UT image.

In the XY UT image there are only 2 defects as one can see in the following figure.



When looking to the grayscale matrix of such noise one can see the following intensity.

```
defect
array([[ 97., 205.],
       [ 98., 205.],
       [ 98., 204.],
       [ 97., 204.]])

get_defect_intensity(imgray,defect)
array([218, 217, 217, 217], dtype=uint8)
```

Figure 13 Grayscale intensity of a noise defect.

The average of the intensity over the whole noise is around ~217. The same work was done for every defect including noise data and the upper threshold of such noise was confirmed.

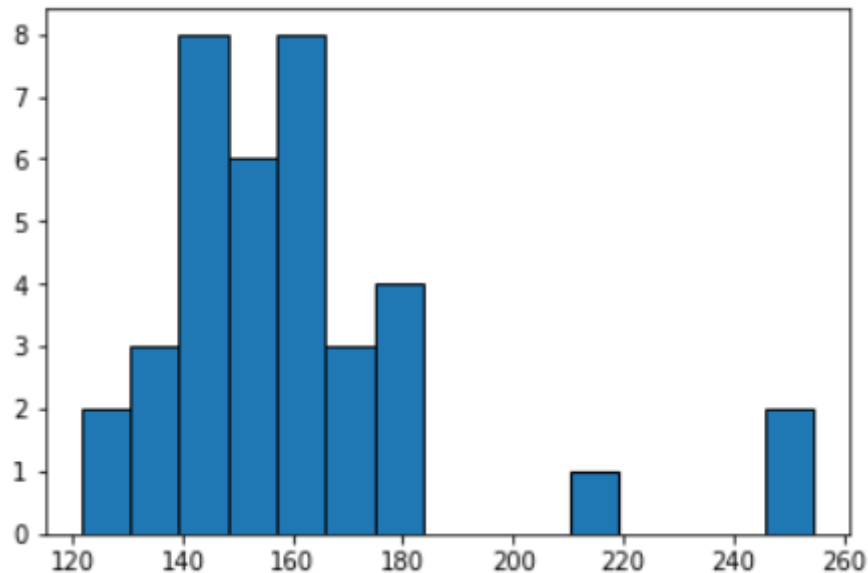


Figure 14 Histogram of frequency of the average defects intensity.

The histogram above of the detected defects with their average intensity is a reliable justification of such threshold. One can see that all intensities are distributed in a range lower than ~217. The chosen threshold is a parameter for the user that is set by default on 200 but still can be modified if needed.

3. Segmentation

The detection of defects so far was made based on a thresholding approach as presented in the previous chapters. In this chapter we are going to separate defects considered as single defects and yet they are either overlapped or neighbor defects. The segmentation was implemented using two methods:

- Watershed algorithm
- K-means algorithm

3.1 Problem

In the UT image example below one can count 4 defects but it is considered from an expert point of view that there are at least 6 defects:

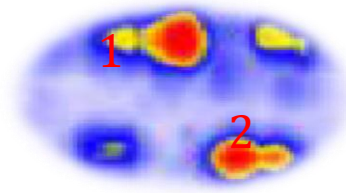


Figure 15 An UT image example with labeled multiple case study defects.

The question is then how to separate defect 1 and defect 2 for example into several defects. For that purpose two methods were explored within this work, namely the Watershed method and the k-means method. Both methods require as input knowing the number of sub defects of each defect to segment.

In that purpose we consider each defect as a map $f : E * F \rightarrow G$
 $(i, j) \rightarrow f(i, j)$

Where $E * F = \{0, 1, 2 \dots N\} * \{0, 1, 2 \dots N\}$ is the coordinate set of each pixel and $G = [0, 255]$ is the intensity of the defect (grayscale).

This mapping can be represented as an (N, N) matrix M .

Question 1:

How to get this (N, N) matrix M ?

Once the defect matrix is determined, the number of sub defects is then the number of extrema. For that purpose, the gradient of M should be calculated.

Question 2:

Which gradient operator to use? Which direction?

After determining the gradient of each defect matrix M , we should calculate the number of minima of such matrix. In fact, the ideal is to look when the gradient is equal to 0 but numerically that is not possible, so the minimum could be an acceptable approximation.

Question 3:

How to find the local minima?

3.2 Case study: Finding extrema

In that sub section we are going to answer to the previous using an illustrative example as shown in the figure below. Recall as well that we will often use the word extrema to refer to both minimums and maximums.

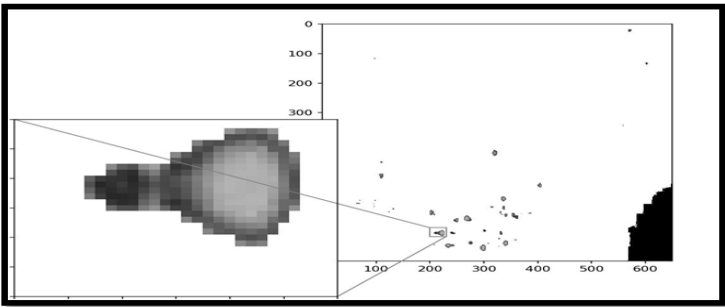


Figure 16 Zoom into an illustrative defect (number 27).

3.2.1 Defect mapping

The corresponding matrix of the figure shown above is as follows:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 118 | 139 | 127 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 169 | 176 | 168 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 122 | 162 | 168 | 158 | 150 | 160 | 169 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 157 | 164 | 147 | 131 | 124 | 120 | 163 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 123 | 168 | 168 | 143 | 120 | 102 | 97 | 102 | 144 | 162 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 181 | 186 | 152 | 116 | 99 | 85 | 76 | 99 | 119 | 167 | 0 |
| 6 | 0 | 0 | 151 | 178 | 180 | 131 | 111 | 154 | 192 | 190 | 164 | 125 | 91 | 85 | 83 | 79 | 82 | 101 | 150 | 112 |
| 7 | 0 | 0 | 201 | 210 | 207 | 155 | 136 | 191 | 207 | 191 | 156 | 123 | 90 | 78 | 77 | 77 | 75 | 107 | 149 | 149 |
| 8 | 132 | 193 | 215 | 215 | 211 | 172 | 148 | 177 | 186 | 171 | 140 | 113 | 87 | 82 | 79 | 75 | 77 | 98 | 138 | 168 |
| 9 | 155 | 204 | 212 | 202 | 208 | 186 | 164 | 182 | 186 | 168 | 137 | 110 | 82 | 76 | 79 | 81 | 78 | 93 | 132 | 168 |
| 10 | 174 | 214 | 207 | 195 | 210 | 197 | 173 | 184 | 177 | 163 | 133 | 104 | 82 | 75 | 78 | 81 | 79 | 85 | 129 | 173 |
| 11 | 157 | 196 | 203 | 204 | 216 | 188 | 160 | 182 | 173 | 156 | 130 | 106 | 87 | 77 | 77 | 79 | 80 | 82 | 133 | 176 |
| 12 | 0 | 0 | 173 | 203 | 209 | 157 | 131 | 176 | 192 | 175 | 143 | 112 | 91 | 80 | 77 | 77 | 80 | 87 | 141 | 172 |
| 13 | 0 | 0 | 118 | 162 | 173 | 116 | 96 | 155 | 203 | 193 | 162 | 122 | 94 | 83 | 80 | 77 | 82 | 92 | 149 | 156 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 170 | 187 | 182 | 148 | 111 | 93 | 87 | 79 | 90 | 109 | 155 | 140 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 121 | 166 | 188 | 160 | 124 | 106 | 96 | 86 | 103 | 122 | 161 | 130 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 171 | 183 | 147 | 120 | 117 | 117 | 107 | 161 | 162 | 0 |
| 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 127 | 171 | 176 | 160 | 143 | 128 | 152 | 166 | 135 | 0 | 0 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 177 | 179 | 171 | 178 | 150 | 0 | 0 |
| 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 106 | 133 | 149 | 140 | 107 | 0 | 0 |

Figure 17 Matrix (19, 19) of the illustrative defect.

Here is a plot of the above matrix:

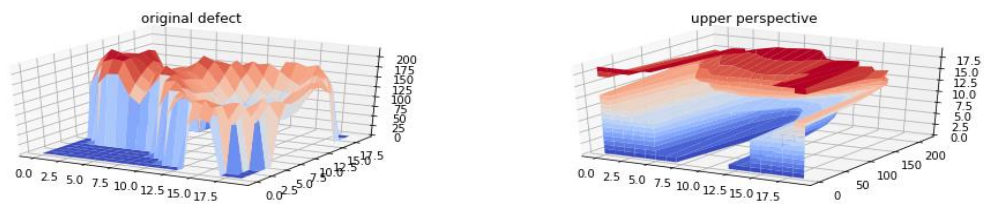


Figure 18 Not smoothed intensity of the illustrative.

It would be possible to calculate the number of extrema by only applying a minima/maxima function but here in this example one can see that is not possible. Both a minima and a maxima should exist in that example, that is why a calculus of gradient is necessary even if time consuming.

Beside, one can see that the mapping is not smooth enough to be derived. In that context a smoothing function was included in the defect matrix python code for better results as shown in the snippet bellow.

```
def defect_matrix(img, defect, Gaussianshape):

    defect = np.array(defect)

    i_min = np.min(defect[:,0]) #starting line index
    i_max = np.max(defect[:,0]+1) #last line index (+1 for python)
    j_min = np.min(defect[:,1]) #starting column index
    j_max = np.max(defect[:,1]+1) #last column index (+1 for python)

    defect_shape = img[i_min:i_max, j_min:j_max].shape

    dim = np.max(defect_shape)
    matrix = np.zeros((dim, dim))

    assert (img[i_min:i_max, j_min:j_max][img[i_min:i_max, j_min:j_max] != 0.].size
            == get_defect_intensity(img, defect).size)

    matrix[0:defect_shape[0], 0:defect_shape[1]] = img[i_min:i_max, j_min:j_max].copy()

    matrix = cv2.GaussianBlur(matrix, (Gaussianshape, Gaussianshape), 0) #smoothing

    return matrix
```

The parameter 'Gaussianshape' is the shape of the window to which the matrix is convoluted in order to smooth it. That is a key parameter for obtaining the exact number of extrema and thus good segmentation.

Smoothing:

A good mask to smooth an image is the Gaussian mask: a discretization of the Gaussian density. This produces a moving-average, where the close pixels are more important than the far ones. [Vincent Vigon]

When the Gaussian filter is applied to each defect the distribution of the matrix intensity is more regular giving more importance to centralized pixels.

There two kind of smoothing:

- ✓ Strong smoothing:

The detection is robust against the noise, but the contours are thick.

- ✓ Weak smoothing:

It has sensibility to the noise but fine localization of the contour.

The matrix intensity after smoothing with a (5, 5) Gaussian filters looks like the following:

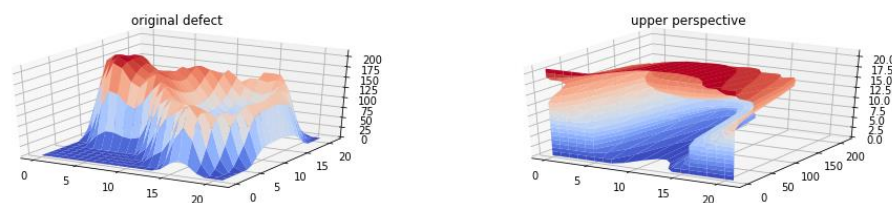


Figure 19 (5, 5) smoothed intensity of the illustrative defect.

3.2.2 Mapping gradient

Here is a resume of the most used gradient operators for contour detection according to the “research gate” discussion [8]. After choosing the Sobel operator based on the following comparison and also recommendation from the lecture of Prof. Vincent Vigon.

Which operator to use:

A) First Order Derivative Edge Detection.

Generally, the first order derivative operators are very sensitive to noise and produce thicker edges.

- ✓ Roberts filtering:

Diagonal edge gradients are susceptible to fluctuations. Gives no information about edge orientation and works best with binary images.

- ✓ Prewitt filter:

The Prewitt operator is a discrete differentiation operator which functions similar to the Sobel operator, by computing the gradient for the image intensity function. Make use of

the maximum directional gradient. As compared to Sobel, the Prewitt masks are simpler to implement but are very sensitive to noise.

✓ Sobel filter:

Detects edges are where the gradient magnitude is high. This makes the Sobel edge detector more sensitive to diagonal edge than horizontal and vertical edges.

Sobel and Prewitt methods are very effectively providing good edge maps.

- ✓ Frei-Chen method: Frei and Chen have adapted the Sobel's model and proposed a pair of isotropic operator which makes $K = \sqrt{2}$. This makes the gradient for horizontal, vertical, and diagonal edges the same at the edge center. The isotropic smoothed weighting operator proposed by Frei and Chen can easily pick up subtle edge detail and produce thinner edge lines, but it also increase the possibility of erroneously detect noise as real edge points.
- ✓ Kirsch compass kernel: Is a non-linear edge detector that finds the maximum edge strength in a few predetermined directions. Kirsch edge operators have been proposed for image segmentation of mammographic images.
- ✓ Robinson compass filtering: Similar to the Kirsch masks, with mask coefficients of 0, 1, and 2.

Kirsch and Robinson methods require more time for calculation and their results are not better than the ones produced by Sobel and Prewitt methods.

B) 2nd Order Derivative Edge Detection.

If there is a significant spatial change in the second derivative, an edge is detected. 2nd Order Derivative operators are more sophisticated methods towards automatized edge detection, however, still very noise-sensitive.

As differentiation amplifies noise, smoothing is suggested prior to applying the Laplacians. In that context, typical examples of 2nd order derivative edge detection are the Difference of Gaussian (DOG) and the Laplacian of Gaussian (LoG) (e.g. the Marr - Hildreth method).

Which directions to choose:

The chosen operator is the so called Sobel operator with its double precision. The gradient is then calculated for the following directions:

- ‘North→South’:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- ‘West→East’:

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & -2 \\ -1 & 0 & -1 \end{bmatrix}$$

- ‘North-west→South-east’:

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -2 & -1 \end{bmatrix}$$

- ‘South-west→North-East’:

$$\begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}$$

The other four directions are giving the same results as they are the reverse of those directions. In fact, the last 4 are just the opposite of the first 4: for instance, the scalar product along the North direction $(\nabla f, (0,1)) = \frac{\partial f}{\partial y}$ the opposite of the scalar product along the South direction $(\nabla f, (0,-1)) = -\frac{\partial f}{\partial y}$.

Gradient magnitude:

The operator uses four 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives.

$$G_{(*,*)} = (\nabla f, (*,*)) = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} * M$$

$\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$ is the sobel operator corresponding to the direction $(*,*)$.

M is the matrix of the defect image.

All the partial derivatives of the chosen directions have to be taken into consideration in order to find the local minima. Therefore, the objective function is then proportional to the norm of the gradient, as shown below:

$$\text{Magnitude} = \text{coef} * \sqrt{G_{(1,0)}^2 + G_{(0,1)}^2 + G_{(1,1)}^2 + G_{(1,-1)}^2}$$

The python code for that is given below as snippet:

```
def getCompassOperators(kind):
    res=np.empty([8,3,3])

    if kind=="kirsch":
        conv_mask = np.array([5, 5, 5,-3,-3,-3,-3,-3])
    elif kind=="robinson2":
        conv_mask = np.array([1,2,1,0,-1,-2,-1,0])
    elif kind=="robinson1":
        conv_mask = np.array([1,1,1,0,-1,-1,-1,0])

    """Discrete Laplacian 4"""
    elif kind=="laplace1":
        conv_mask = np.array([0,1,0,1,0,1,0,1])

    """Discrete Laplacian 8"""
    elif kind=="laplace2":
        conv_mask = np.array([1,1,1,1,1,1,1,1])

    """Robinson Laplacian (to take with you in a desert island)"""
    elif kind=="laplace3":
        conv_mask = np.array([1,-2,1,-2,1,-2,1,-2])

    for i in range(8):
        oneDir=np.zeros(9)
        oneDir[[0,1,2,5,8,7,6,3]]=conv_mask
        "to turn the coef"
        conv_mask=np.concatenate([conv_mask[1:],conv_mask[0:1]])
        res[i,:,:]=oneDir.reshape([3,3])

        if kind=="laplace1":
            res[i,1,1] = -4

        """Discrete Laplacian 8"""
        elif kind=="laplace2":
            res[i,1,1] = -8

        """Robinson Laplacian (to take with you in a desert island)"""
        elif kind=="laplace3":
            res[i,1,1] = 4

    return res
```

The convolution is made using a “Scipy “library with its package: signal (sg).

```
operators=getCompassOperators("robinson2")
all_grad = np.abs(np.array([sg.convolve(case, operators[i,:,:], "valid") for i in range(4)]))

Norm = np.sqrt(all_grad[0,:,:]**2 + all_grad[1,:,:]**2+all_grad[2,:,:]**2+all_grad[3,:,:]**2)
```

3.2.3 Gradient minima

In this sub section we are going to start looking at trying to find relative minimums and relative maximums of the intensity matrix determined above. That can be reached as well by finding the local minimums in the Gradient matrix obtained by the displayed python code above. Here is the obtained Gradient matrix:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|----|---------|---------|---------|---------|---------|--------|---------|---------|---------|---------|--------|---------|--------|--------|--------|--------|---------|---------|
| 0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 298.84 | 636.30 | 889.92 | 963.19 | 750.43 | 153.03 | 741.13 | 1010.66 | 413.96 |
| 1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 624.84 | 940.88 | 920.21 | 613.75 | 238.74 | 272.86 | 214.04 | 935.31 | 742.40 |
| 2 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 301.29 | 652.25 | 960.98 | 729.08 | 57.04 | 289.81 | 332.31 | 323.64 | 342.66 | 477.28 | 994.19 |
| 3 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 682.23 | 1045.37 | 818.40 | 251.44 | 304.18 | 346.11 | 306.04 | 230.32 | 328.26 | 170.07 | 895.66 |
| 4 | 369.87 | 736.16 | 996.32 | 965.62 | 822.35 | 756.48 | 900.39 | 1099.91 | 841.68 | 109.28 | 390.97 | 379.90 | 259.76 | 172.28 | 114.85 | 269.14 | 414.37 | 522.66 |
| 5 | 788.68 | 1154.92 | 1206.64 | 1142.02 | 1002.65 | 927.50 | 1093.82 | 932.38 | 371.33 | 339.20 | 440.83 | 302.87 | 144.34 | 77.20 | 56.25 | 194.09 | 419.13 | 224.19 |
| 6 | 1193.45 | 1010.76 | 258.26 | 354.14 | 451.39 | 240.08 | 389.98 | 52.65 | 267.68 | 390.30 | 382.36 | 232.56 | 62.32 | 32.92 | 26.94 | 151.17 | 400.59 | 248.37 |
| 7 | 1007.78 | 550.02 | 9.59 | 228.95 | 370.04 | 124.84 | 251.02 | 112.12 | 307.94 | 371.22 | 343.64 | 214.71 | 49.03 | 13.64 | 20.74 | 127.70 | 370.51 | 370.29 |
| 8 | 352.96 | 18.06 | 64.92 | 108.94 | 295.12 | 117.33 | 132.46 | 83.35 | 275.97 | 343.00 | 313.27 | 188.72 | 38.88 | 11.40 | 16.85 | 75.51 | 322.96 | 452.43 |
| 9 | 257.94 | 48.87 | 22.89 | 57.20 | 260.67 | 48.97 | 75.88 | 129.85 | 270.88 | 329.87 | 290.82 | 177.84 | 32.80 | 27.24 | 7.62 | 39.67 | 304.36 | 501.13 |
| 10 | 988.17 | 551.37 | 107.01 | 127.71 | 372.72 | 183.47 | 157.31 | 109.37 | 272.35 | 334.44 | 284.67 | 179.69 | 60.61 | 10.00 | 17.15 | 31.30 | 318.78 | 523.32 |
| 11 | 1128.22 | 992.55 | 379.44 | 369.25 | 547.40 | 339.29 | 373.19 | 91.88 | 326.50 | 396.06 | 332.35 | 201.00 | 81.17 | 19.90 | 16.37 | 57.93 | 356.51 | 475.64 |
| 12 | 653.02 | 1062.23 | 1159.00 | 1135.24 | 1011.74 | 894.85 | 947.64 | 530.52 | 172.31 | 387.82 | 418.63 | 276.44 | 128.83 | 66.14 | 42.10 | 123.28 | 388.53 | 353.12 |
| 13 | 289.04 | 627.64 | 893.15 | 895.84 | 751.82 | 698.79 | 1048.50 | 948.13 | 176.82 | 240.09 | 438.85 | 340.91 | 186.54 | 121.74 | 88.15 | 198.00 | 392.19 | 201.17 |
| 14 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 652.54 | 1025.28 | 896.86 | 372.73 | 341.34 | 383.94 | 239.26 | 199.15 | 172.13 | 289.47 | 383.41 | 247.71 |
| 15 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 296.39 | 643.33 | 1029.64 | 833.19 | 84.78 | 353.82 | 347.66 | 293.96 | 265.66 | 352.70 | 250.06 | 691.41 |
| 16 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 667.80 | 1025.92 | 988.22 | 595.94 | 190.29 | 351.82 | 357.88 | 278.32 | 312.13 | 1020.09 |
| 17 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 311.09 | 667.80 | 934.12 | 1002.05 | 753.05 | 80.98 | 32.37 | 110.50 | 801.70 | 932.58 |

Figure 20 Gradient magnitude matrix (17, 17) of the illustrative defect.

And here its 3d plot to have an idea about its smoothness:

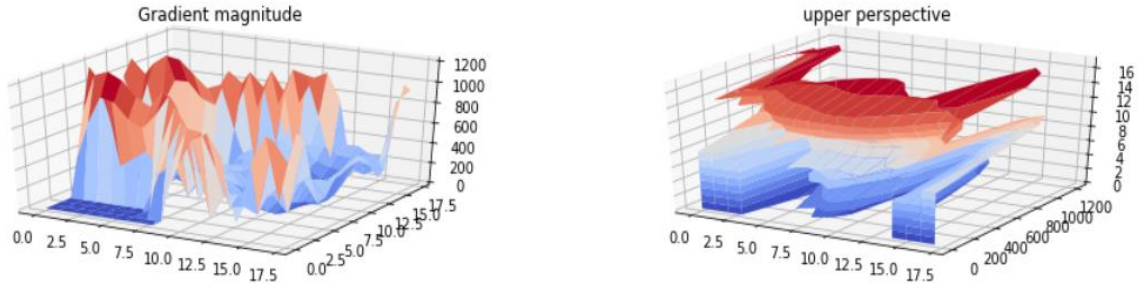


Figure 21 Gradient magnitude matrix (17, 17) of the illustrative defect.

Smoothing:

One can observe in the graph of the gradient magnitude that it's not smooth enough for better minima extraction. Therefore, a smoothing kernel of (9, 9) is convolved to this matrix before starting the minima extraction as explained below.

Maxima detection:

Let's denote by G the rolling window over the gradient magnitude M_G , which we can obtain by the convolution between the image and one of the previous operator explained in the previous sub section (i.e. Sobel). Then let's note:

$$G(i^*, j^*) = \begin{bmatrix} G(i-1, j-1) & G(i-1, j) & G(i-1, j+1) \\ G(i, j-1) & G(i, j) & G(i, j+1) \\ G(i+1, j-1) & G(i+1, j) & G(i+1, j+1) \end{bmatrix}$$

We consider the pixel (i, j) is a minimum when: all its neighbors are greater than $G(i, j)$. The same definition is applied to a rolling window of $(2p+1) \times (2p+1)$ kernel.

$$G(i^*, j^*) = \begin{bmatrix} G(i-p+1, j-p+1) & \cdots & G(i-p+1, j+p-1) \\ \vdots & G(i, j) & \vdots \\ G(i+p-1, j-p+1) & \cdots & G(i+p-1, j+p-1) \end{bmatrix}$$

The rolling window is obtained by the following python code:

```
def rolling_windows_img(a, kshape):
    outShape=(a.shape[0]-kshape[0]+1,)+(a.shape[1]-kshape[1]+1,)+kshape
    outStrides=a.strides+a.strides
    return np.lib.stride_tricks.as_strided(a, shape=outShape, strides=outStrides)
```

This function produces a rolling window which is just a view on the data. It is then used in the minima extraction function as shown below:

| | | | | | | | | |
|------------------|---|----|---|---|---|---|---|---|
| | 8 | 10 | 9 | 8 | 6 | 5 | 3 | 3 |
| Number of | 4 | 4 | 4 | 4 | 5 | 3 | 3 | 3 |
| extrema | 2 | 2 | 2 | 2 | 3 | 1 | 3 | 2 |
| | 1 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |

Table 1 Summary of number of extrema according to smoothing and local window kernels.

It is obvious that the dimension of both the smoothing and the local rolling window kernels are very crucial to define the number of extrema and thus the number of sub defects.

Question:

How to find the best combination?

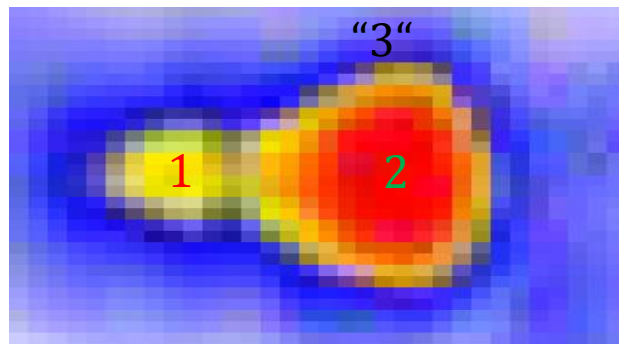


Figure 23 Labeled study case defect.

If it is only for this case of defect shown above, one can say it's 2 to 3 sub defects when considering a possible overlapping defect in yellow with the one in red. Thus we can choose the combination in green on the table above because it gives right approximation independently of the smoothing kernel. However the problem is more complex. We got in the same UT image different kind of defects with different size which influence the local rolling window choice. Besides, we have another UT image to match with a different resolution which influences the smoothing window dimension.

This conflict is going to be discussed furthermore in the discussion chapter.

3.3 Sub-defects detection

In this sub chapter we are going to see the two explored methods for segmentation of defects in UT images used within this work. At first, the watershed method is explained and then implemented in python language. Then the k-means method is introduced and used to solve the segmentation problem.

3.3.1 Watershed algorithm

The watershed is a classical algorithm used for segmentation, that is, for separating different objects in an image.

Principle:

Starting from user-defined markers or local extrema, the watershed algorithm treats pixels values as a local topography (elevation). The algorithm floods basins from the markers, until basins attributed to different markers meet on watershed lines. In many cases, markers are chosen as local minima of the image, from which basins are flooded.

In the example below (see Figure below), two overlapping circles are to be separated (a). To do so, one computes an image that is the distance to the background (b). The maxima of this distance (i.e., the minima of the opposite of the distance) are chosen as markers, and the flooding of basins from such markers separates the two circles along a watershed line(c). [5]

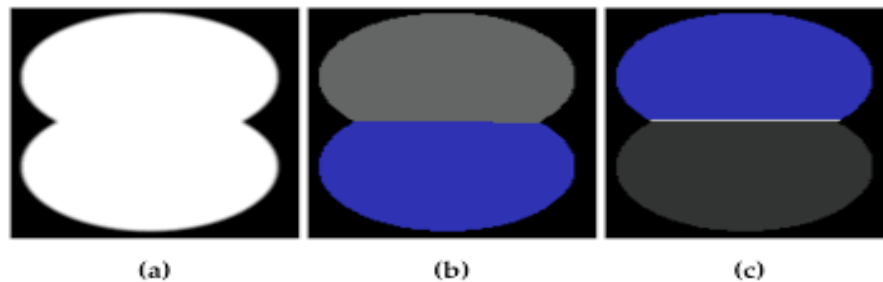


Figure 24

(a) Overlapping circles

(b) Different markers meeting on the watershed line

(c) Separated circles along the watershed line

In order to implement this algorithm in our situation, the position of the so called markers which have to be extrema are necessary as shown in the figure below.

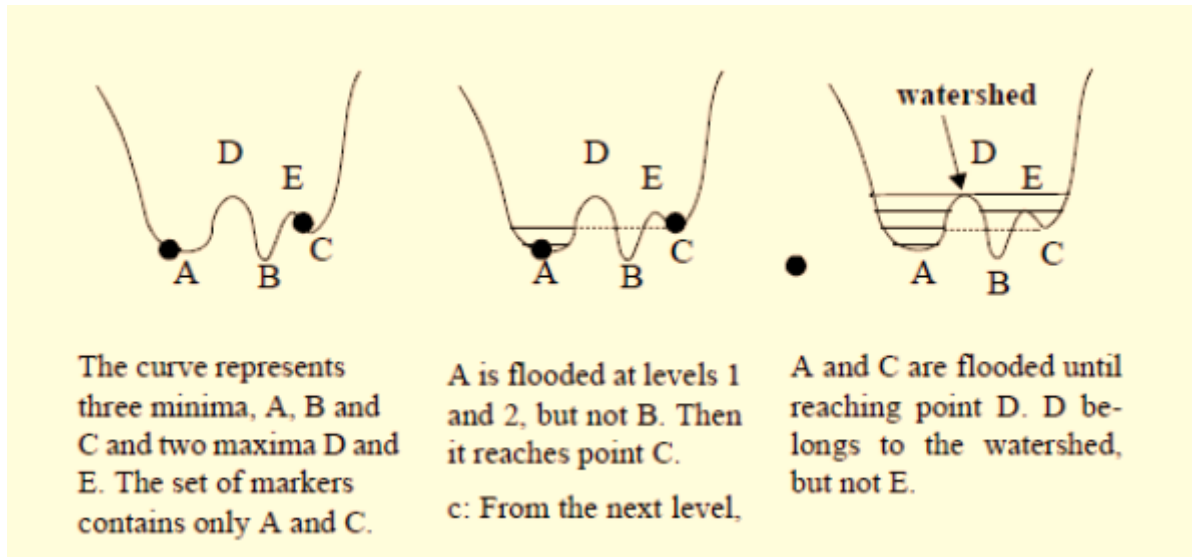


Figure 25

Watershed principle from [10].

Implementation:

Here is a draft of how the watershed algorithm is implemented:

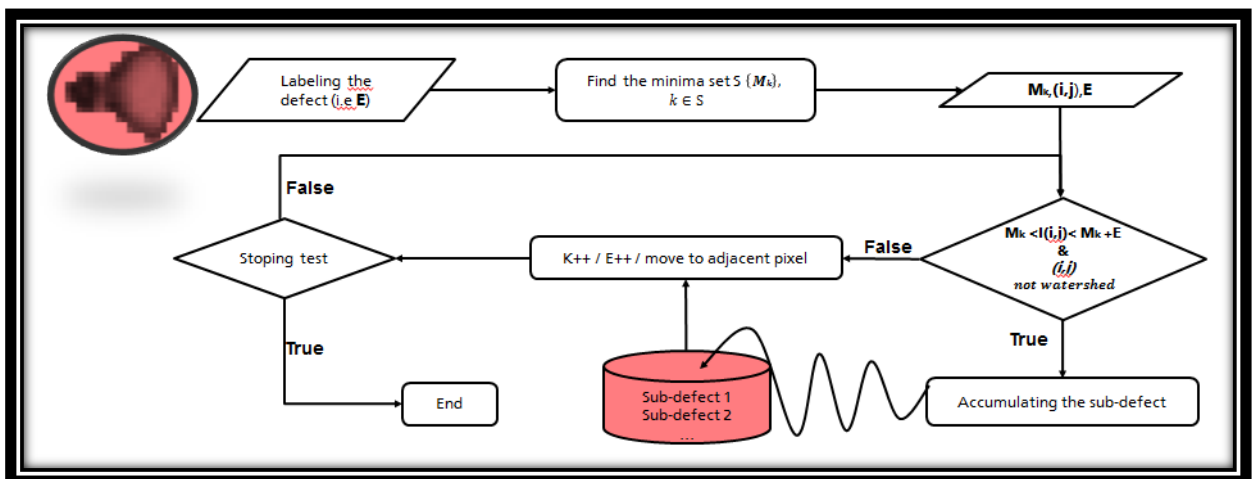


Figure 26

A draft of the implemented watershed algorithm.

The first step of the implementation is labeling. This has as purpose accelerating the inter pixel movement and also define the watershed thickness. Afterwards, the coming step is to define the markers or the minima before starting the watershed algorithm. In our case study, the separation is shown below.

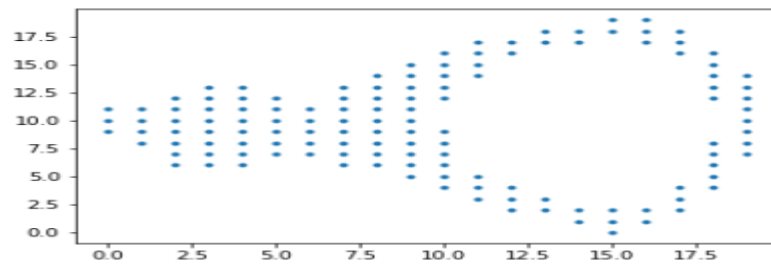


Figure 27 Defect separation with the watershed algorithm and the labeled defect at the left.

Limitation:

The implemented watershed method gives good results when the defect intensity is fluctuating showing clear local minima. In our case we have a defect with a minimum and a maximum which does not allow better separation as illustrated bellow.

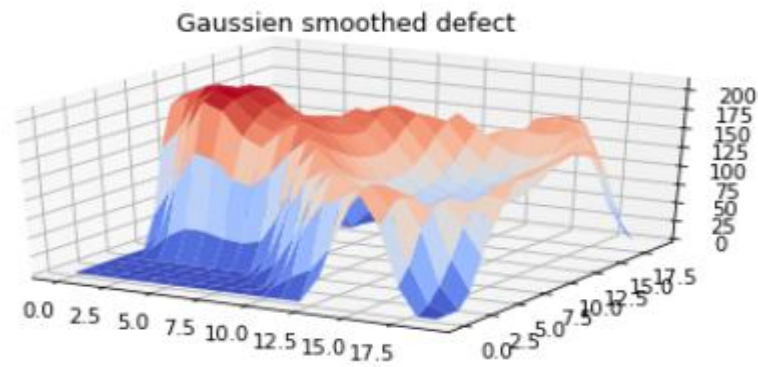


Figure 28 Smoothed defect intensity.

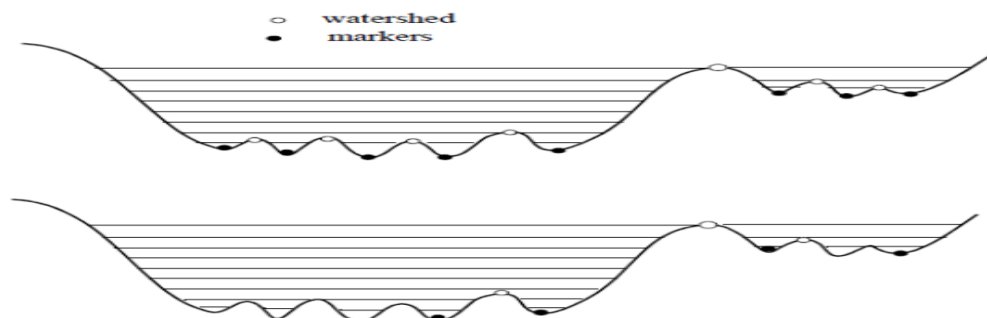


Figure 29 Example of how the result depends on the choice of markers [11].

For such reason another method is used for segmentation, namely the k-means algorithm detailed in the sub section below.

3.3.2 K-means

K-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. For better understanding one can check the link [12] from which the method principle was taken.

Principle:

K-means clustering is a type of unsupervised learning, which is used when you have unlabeled data (i.e., data without defined categories or groups). The goal of this algorithm is to find groups in the data, with the number of groups represented by the variable K . The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. Data points are clustered based on feature similarity. The results of the K-means clustering algorithm are:

1. The centroids of the K clusters, which can be used to label new data
2. Labels for the training data (each data point is assigned to a single cluster)

Rather than defining groups before looking at the data, clustering allows you to find and analyze the groups that have formed organically. The "Choosing K " section below describes how the number of groups can be determined.

Each centroid of a cluster is a collection of feature values which define the resulting groups. Examining the centroid feature weights can be used to qualitatively interpret what kind of group each cluster represents.

Python code:

In this work the code was made using the 'Scikit-learn' library with its package 'Sklearn.cluster' from which 'KMeans' is imported. The results of the K-means clustering algorithm are obtained in two steps:

- The centroids of the K clusters:

```
"""creation of the algo"""
kmeans=KMeans(n_clusters=2)
"""computation of centroids"""
kmeans.fit(pts)
"""here is the result"""
kmeans.cluster_centers_

array([[ 5.96,  9.76],
       [14.8 ,  9.89]])
```

The number of clusters is given by the number of extrema as explained in the previous sub chapter. For the computation of the centroids the defect is considered to be a cloud of points where it is 0 when there no defect and 1 when the defect is there.

- Labels for the training data

The labels for each cluster are obtained by the mean of the function “kmeans.predict (pts)”. The final result would be divided into two labels ‘0’ for one sub-defect and ‘1’ for the other sub-defect as shown in the figure below.

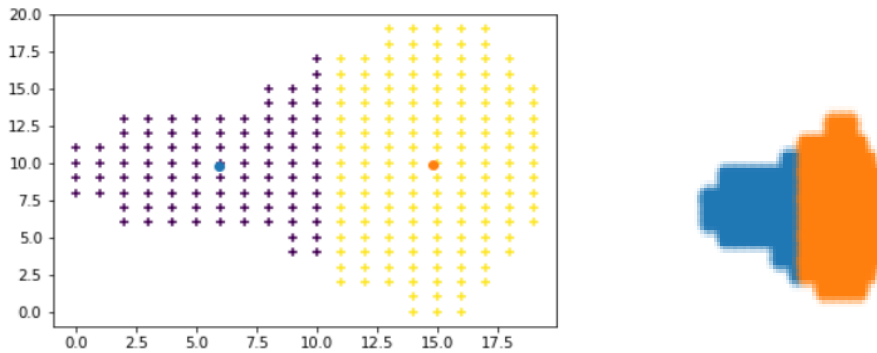


Figure 15 Clustered 2D data of the case study defect.

Once the algorithm is applied on the whole 2D data UT image the results are visualized in the figure below.

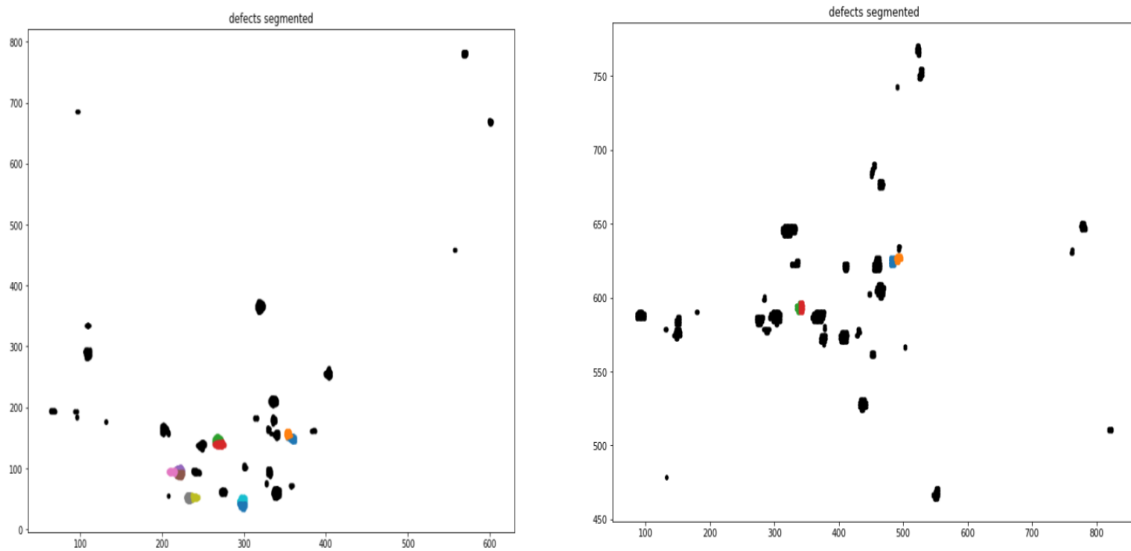


Figure 31 (a) Segmented 2D XY defects. (b) Segmented 2D XZ defects.

In this figure one can see that some defects are over segmented due to the shapes of the smoothing window kernel and the local minima window kernel. This over segmentation is going to be discussed furthermore into the discussion chapter.

The sub-defects are approximated to ellipse shapes in order to have an idea on their distribution so it can be possible to match the 2D data of each plan with the other. The obtained results are shown below.

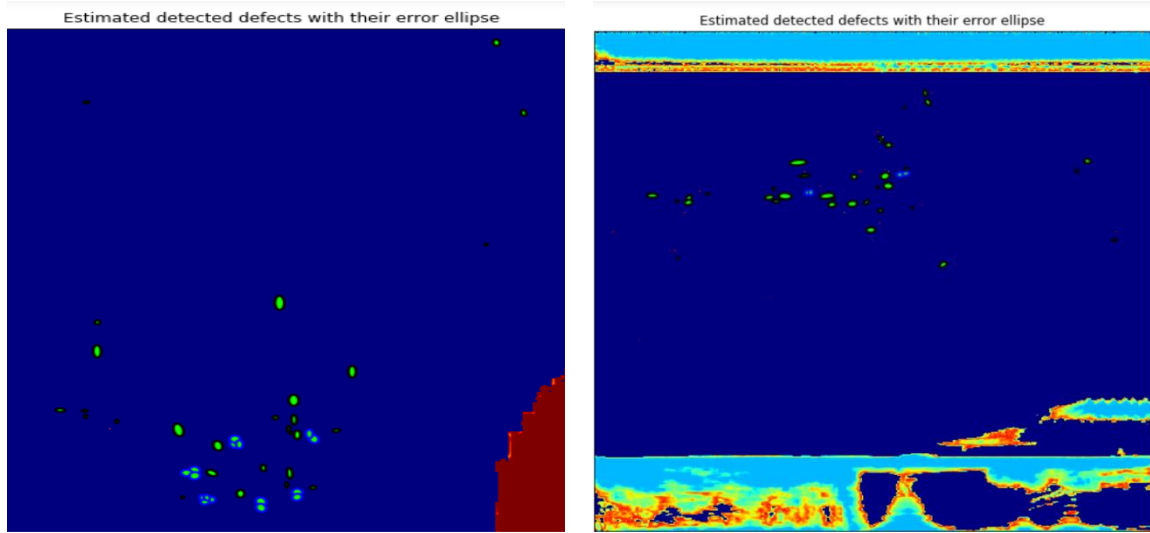


Figure 32 (a) Estimated ellipses to 2D XY defects. (b) Estimated ellipses to 2D XZ defects.

The green ellipses are the not segmented defects, the blue ones are the segmented and the red ones are either artifacts or noise. The red spots are cleaned in previous steps but kept here to have an overview on the changes made on the UT images.

This estimation of ellipses is explained in the next chapter by the mean of two defect samples.

4. Covariance error ellipse:

This section is devoted to estimating each defect with an ellipse called covariance error ellipse. It is also explained how to obtain confidence ellipses for different confidence values (e.g. 90% confidence interval). At the end of the section a python code will be shown in part and more details on the report appendix. For more theoretical explanations one can consult the following link [13].

4.1 Necessary hypothesis

The 2D data extracted from the UT images are smoothed with a Gaussian filter. This means that the coordinates of each pixel in every defect (x-values and the y-values and/or the z-values) are normally distributed. This condition is not only necessary for removing noise from a data set, allowing important patterns to stand out (i.e. extrema) but also for the determination of the scale 'S' of the error ellipse represented by the following equation:

$$2D: \left(\frac{X}{\sigma_x}\right)^2 + \left(\frac{Y}{\sigma_y}\right)^2 = S$$

$$3D: \left(\frac{X}{\sigma_x}\right)^2 + \left(\frac{Y}{\sigma_y}\right)^2 + \left(\frac{Z}{\sigma_z}\right)^2 = S$$

When assuming that X, Y and/or Z are independent data samples normally distributed, then we can deduce that the sum of their square is distributed to a Chi-square distribution. According to Wikipedia, in probability theory and statistics, the chi-square distribution (also chi-squared or χ^2 -distribution) with k degrees of freedom is the distribution of a sum of the squares of k independent standard normal random variables.

Therefore, we can easily obtain the probability that the above sum, and thus S equals a specific value by calculating the Chi-Square likelihood. In fact, since we are interested in a confidence interval, we are looking for the probability that S is less than or equal to a specific value which can easily be obtained using the cumulative Chi-Square distribution. As shown in the following figure from Wikipedia which is also depicted in a table in this link [14].

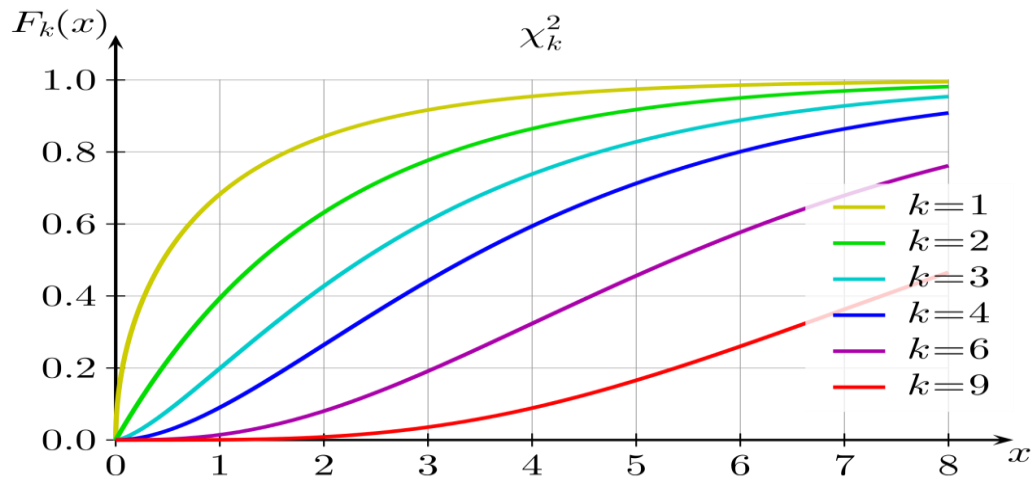


Figure 33 Cumulative distribution functions for six k degrees of freedom.

The areas given across the top are the areas to the right of the critical value. To look up an area on the left, subtract it from one, and then look it up (i.e.: 0.05 on the left is 0.95 on the right).

| df | 0.995 | 0.99 | 0.975 | 0.95 | 0.90 | 0.10 | 0.05 | 0.025 | 0.01 | 0.005 |
|----|-------|-------|-------|-------|-------|-------|-------|-------|--------|--------|
| 1 | --- | --- | 0.001 | 0.004 | 0.016 | 2.706 | 3.841 | 5.024 | 6.635 | 7.879 |
| 2 | 0.010 | 0.020 | 0.051 | 0.103 | 0.211 | 4.605 | 5.991 | 7.378 | 9.210 | 10.597 |
| 3 | 0.072 | 0.115 | 0.216 | 0.352 | 0.584 | 6.251 | 7.815 | 9.348 | 11.345 | 12.838 |

Table 2 Probability table of χ^2 -distribution with k = 1, 2, 3 degrees of freedom.

A Chi-Square distribution is defined in terms of ‘degrees of freedom’, which represent the number of unknowns. In our case there are two unknowns, and therefore two degrees of

freedom. For example, using this probability table we can easily find that, in the 2-degrees of freedom case: $P(s < 5.991) = 1 - 0.05 = 0.95$

Therefore, a 95% confidence interval corresponds to $S=5.991$. In other words, 95% of the data will fall inside the ellipse defined as:

$$2D: \left(\frac{X}{\sigma_x}\right)^2 + \left(\frac{Y}{\sigma_y}\right)^2 = 5.991$$

In this work, several confidence intervals were used namely 90% for a scale $S = 4.605$ and 80% for $S = 3.22$ value.

Later on, a three degree of freedom corresponding value is used for estimating the defects in 3D scale. In fact, as shown in the table the following values were used as illustrative examples, namely 90% for a scale $S = 6.251$ and 95% for $S = 7.815$ value.

Therefore, a 95% confidence interval corresponds to $S=7.815$. In other words, 95% of the data will fall inside the ellipse defined as:

$$3D: \left(\frac{X}{\sigma_x}\right)^2 + \left(\frac{Y}{\sigma_y}\right)^2 + \left(\frac{Z}{\sigma_z}\right)^2 = 7.815$$

4.2 Confidence ellipse features

In this section we start with generic introduction the covariance matrix and what's the meaning of its values. After understanding the meaning we move how to calculate such matrix in order to get the confidence ellipse features.

4.2.1 A geometric interpretation of the covariance matrix

In this sub chapter one can understand the geometric interpretation of the covariance and the meaning of the standard deviation in that context. We consider for illustration a 2D feature space as shown below. The variance or the square of the standard deviation σ^2 can be used to explain the spread of the data in the directions parallel to the axes of the feature space.

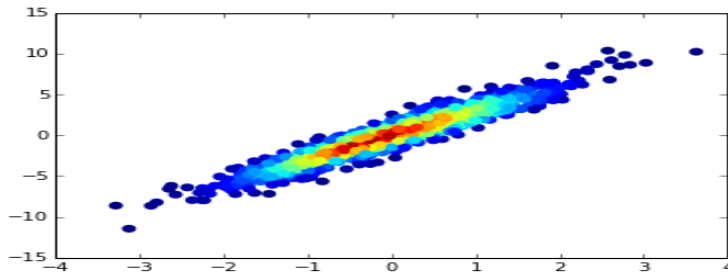


Figure 34 Illustrative example of 2D data defect space.

For this data, we could calculate the variance $\sigma_x^2 = \sigma_{(x,x)}$ in the x-direction and the variance $\sigma_y^2 = \sigma_{(y,y)}$ in the y-direction using the formula: $\sigma_{(x,x)} = E[(x - E(x)) * (x - E(x))]$

However, the horizontal spread and the vertical spread of the data does not explain the clear diagonal correlation. The figure above clearly shows that on average, if the x-value of a data point increases, then also the y-value increases, resulting in a positive correlation. This correlation can be captured by extending the notion of variance to what is called the ‘covariance’ of the data:

$$\sigma_{(x,y)} = E[(x - E(x)) * (y - E(y))]$$

For 2D data, we thus obtain $\sigma_{(x,x)}$, $\sigma_{(y,y)}$, $\sigma_{(x,y)}$ and $\sigma_{(y,x)}$. These four values can be summarized in a matrix, called the covariance matrix:

$$\Sigma = \begin{bmatrix} \sigma_{(x,x)} & \sigma_{(y,x)} \\ \sigma_{(x,y)} & \sigma_{(y,y)} \end{bmatrix}$$

If x is positively correlated with y, y is also positively correlated with x. In other words, we can state that $\sigma_{(x,y)} = \sigma_{(y,x)}$. Therefore, the covariance matrix is always a symmetric matrix with the variances on its diagonal and the covariance off-diagonal. Two-dimensional normally distributed data is explained completely by its mean and its $2 * 2$ covariance matrix. Similarly, a $3 * 3$ covariance matrix is used to capture the spread of three-dimensional data, and a $N * N$ covariance matrix captures the spread of N-dimensional data.

As we saw earlier, the covariance matrix is always a symmetric matrix thus we can represent the covariance matrix by its eigenvectors and eigenvalues (Spectral theorem):

$$\Sigma \vec{v} = \lambda \vec{v}$$

With \vec{v} is an eigenvector of Σ , and λ is the corresponding eigenvalue.

This can be represented efficiently using matrix notation:

$$\Sigma V = V L$$

With V is the matrix whose columns are the eigenvectors of Σ , and L is the diagonal matrix whose non-zero elements are the corresponding eigenvalues.

This means that we can represent the covariance matrix as a function of its eigenvectors and eigenvalues:

$$\Sigma = V L V^{-1}$$

This equation is called the ‘Eigen decomposition’ of the covariance matrix and can be obtained using a Singular Value Decomposition algorithm. Whereas the eigenvectors represent the directions of the largest variance of the data, the eigenvalues represent the magnitude of this variance in those directions. In other words, V represents a rotation matrix, while \sqrt{L} represents a scaling matrix. The covariance matrix can thus be decomposed further as:

$$\Sigma = R S R^{-1}$$

With $R = V$ is a rotation matrix and $S = \sqrt{L}$ is a scaling matrix.

In this sub chapter we showed that the covariance matrix of observed data is directly related to a linear transformation of white, uncorrelated data. This linear transformation is completely defined by the eigenvectors and eigenvalues of the data. While the eigenvectors represent the rotation matrix, the eigenvalues correspond to the square of the scaling factor in each dimension.

4.2.2 The width, the length and the angle

The center of the estimated ellipse is the barycenter of the defect and thus calculated already in the previous chapters. Here in this sub chapter we consider $(0, 0)$ and $(0, 0, 0)$ are the origins of respectively the 2D and 3D covariance confidence ellipses.

In general, the equation of an axis-aligned ellipse with a major axis of length $2a$ and a minor axis of length $2b$, centered at the origin, is defined by the following equation:

$$\text{2D:} \quad \left(\frac{X}{a}\right)^2 + \left(\frac{Y}{b}\right)^2 = 1$$

$$\text{3D:} \quad \left(\frac{X}{a}\right)^2 + \left(\frac{Y}{b}\right)^2 + \left(\frac{Z}{c}\right)^2 = 1$$

In our case, the lengths of the axes are defined by the standard deviations σ_x and σ_y or/and σ_z of the data such that the equation of the error ellipse becomes:

$$\text{2D:} \quad \left(\frac{X}{\sigma_x}\right)^2 + \left(\frac{Y}{\sigma_y}\right)^2 = S$$

$$\text{3D:} \quad \left(\frac{X}{\sigma_x}\right)^2 + \left(\frac{Y}{\sigma_y}\right)^2 + \left(\frac{Z}{\sigma_z}\right)^2 = S$$

Where S defines the scale of the ellipse and is explained in the previous sub chapter.

The error ellipse shown by the figure below can therefore be drawn as an ellipse with a major axis length equal to $2\sigma_x\sqrt{S}$ and the minor axis length to $2\sigma_y\sqrt{S}$ or/and $2\sigma_z\sqrt{S}$.

In other words, whereas we calculated the standard deviations σ_x and σ_y or/and σ_z parallel to the x-axis and y-axis or/and z-axis earlier, we now need to calculate these variances parallel to what will become the major and minor axis of the confidence ellipse. The directions in which these variances need to be calculated are illustrated by a pink and a green arrow in the figure below.

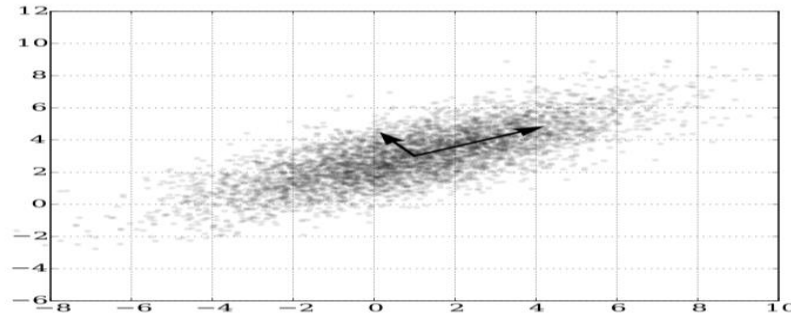


Figure 35 Illustrative error ellipse with direction axis in green and pink.

These directions are actually the directions in which the data varies the most, and are defined by the covariance matrix. The covariance matrix can be considered as a matrix that linearly transformed some original data to obtain the currently observed data. In a previous sub section about eigenvectors and eigenvalues we showed that the direction vectors along such a linear transformation are the eigenvectors of the transformation matrix. Indeed, the vectors shown by pink and green arrows in the figure above are the eigenvectors of the covariance matrix of the data, whereas the length of the vectors corresponds to the eigenvalues.

The eigenvalues therefore represent the spread of the data in the direction of the eigenvectors. In other words, the eigenvalues represent the variance of the data in the direction of the eigenvectors. In the case of axis aligned error ellipses, i.e. when the covariance equals zero, the eigenvalues equal the variances of the covariance matrix and the eigenvectors are equal to the definition of the x-axis and y-axis or/and z-axis . In the case of arbitrary correlated data, the eigenvectors represent the direction of the largest spread of the data, whereas the eigenvalues define how large this spread really is.

Thus, the 'S'% confidence ellipse can be defined similarly to the axis-aligned case, with the major axis of length $2\sqrt{\lambda_1 * S}$ and the minor axis of length $2\sqrt{\lambda_2 * S}$ or/and $2\sqrt{\lambda_3 * S}$, where λ_1 and λ_2 or/and λ_3 represent the eigenvalues of the covariance matrix.

To obtain the orientation of the ellipse, we simply calculate the angle of the largest eigenvector towards the x-axis:

$$\alpha = \arctan\left(\frac{V_1(y)}{V_1(x)}\right)$$

Given that V_1 is the eigenvector of the covariance matrix. It corresponds to the largest eigenvalue.

Based on the minor and major axis lengths and the angle α between the major axis and the x-axis, it becomes trivial to plot the confidence ellipse.

4.3 Case study

In this sub section, several examples will be shown for both 2D and 3D data set. The results are outputs based on previous treatments consisting mainly on:

- ✚ Image processing
- ✚ Data cleaning
- ✚ Defects recognition
- ✚ Extrema detection
- ✚ Segmentation

4.3.1 2D data

The presented cases below are samples from the 2D data from the (x,y) UT image. After detecting the defects matrixes with several extrema, the segmentation is made by the mean of the k-means clustering method. The sub defects are estimated by the mean of their covariance confidence ellipses for different scale values as explained in earlier sections and illustrated in the figure below.

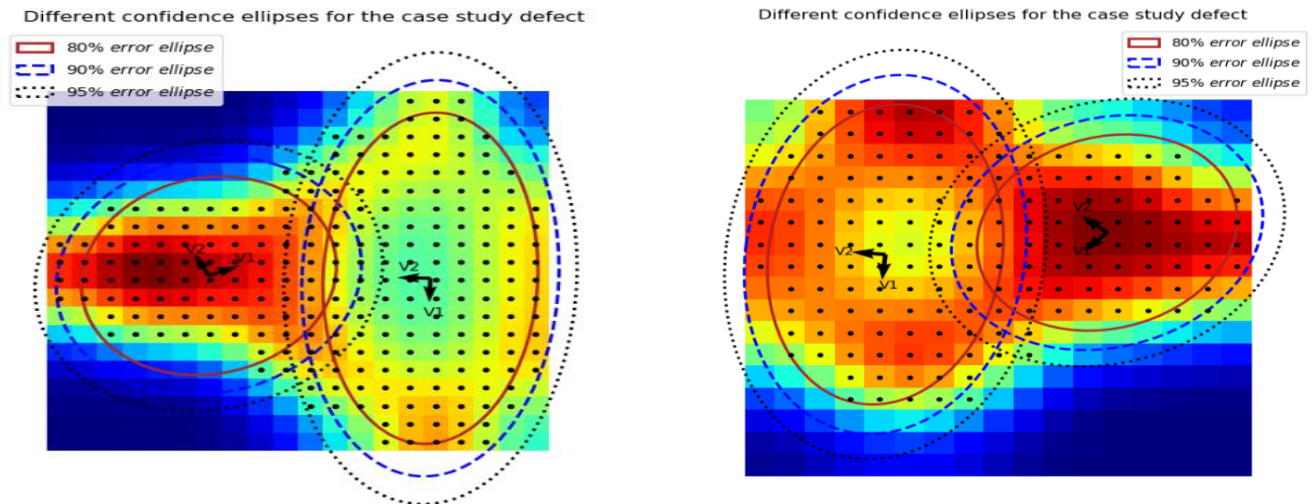


Figure 36 Defect number 27 on the left and number 34 on the right from (x,y) data.

The python code used for doing such estimation is done mainly using two functions, the first one for features calculation 'ellipse_features'.

```
def plot_ellipse(ax, centroid, angle, stand_dev_x, stand_dev_y, defect, fill = False, edgecolor = 'black'):
    ax.imshow(np.flip(imgtosuperimpose, 0), cmap='jet')

    x = centroid[0] #x coordinate
    y = centroid[1] #y coordinate

    e = patches.Ellipse((x, y), stand_dev_y, stand_dev_x, angle=angle, linewidth=2, fill=fill, edgecolor=edgecolor)

    ax.add_patch(e)
    ax.set_title('Estimated detected defects with their error ellipse');

    ax.add_artist(e)
    e.set_alpha(0.9)
    e.set_facecolor([0.1, 1, 0]) #green

    return ax
```

And the second one to make figures called 'plot_ellipse' as shown in the following snippet.

```
def ellipse_features(defect, s_error):
    defect = defect.T #defect should be an array of shape (2,...)!!
    cov = np.cov(defect)
    w, v = LA.eig(cov) #eigen values, eigen vectors

    index = np.argmax(w) #the corresponding value to the biggest eigen value

    if np.abs(v[0, index]) == 0:
        angle = np.sign(v[1, index]) * 90.
    else:
        angle = np.arctan(v[1, index] / np.abs(v[0, index]))
        angle = math.degrees(angle)

    stand_dev_x = np.sqrt(np.float(s_error) * w[0]) #Standard deviation on the direction of eigen vector of position 0
    stand_dev_y = np.sqrt(np.float(s_error) * w[1]) #Standard deviation on the direction of eigen vector of position 1

    return angle, stand_dev_x, stand_dev_y
```

As explained earlier the angle is given by the highest eigenvalue index. In order to avoid division by null, in case of $\pm 90^\circ$, such case was treated separately. The standard deviation is actually the eigenvalue but in the code the names of the half-length and the half-width were misused.

4.3.2 3D data

The figure below is obtained after the following steps were accomplished:

- ✚ Image processing
- ✚ Data cleaning
- ✚ Defects recognition
- ✚ Extrema detection
- ✚ Segmentation
- ✚ 3D Assignment

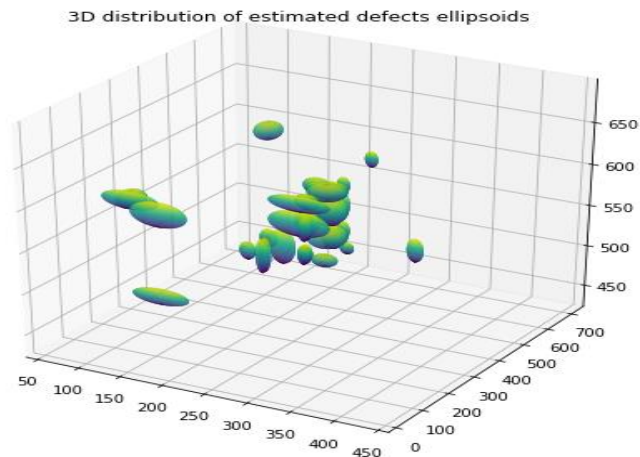


Figure 37 Estimated ellipsoids to the matching defects.

To obtain such figure the following snippets shows the used python code.

```
def one_ellipsoid_3D(ax, Ellipsoid_features, i, scale = 40):

    x0, y0, z0 = Ellipsoid_features[0]
    sig_x, sig_y, sig_z = Ellipsoid_features[1]

    coefs = (sig_x[i]**2, sig_y[i]**2, sig_z[i]**2) # Coefficients in  $a^2/c x^2 + a^2/c y^2 + a^2/c z^2 = 1$ 
    # Radii corresponding to the coefficients:
    rx, ry, rz = scale/np.sqrt(coefs)

    # Set of all spherical angles:
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)

    # Cartesian coordinates that correspond to the spherical angles:
    # (this is the equation of an ellipsoid):
    x = rx * np.outer(np.cos(u), np.sin(v)) + x0[i]
    y = ry * np.outer(np.sin(u), np.sin(v)) + y0[i]
    z = rz * np.outer(np.ones_like(u), np.cos(v)) + z0[i]

    # Plot:
    ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap='viridis', edgecolor='none', alpha=0.4)
```

The inputs are obviously the ellipsoid features (centroids, angle, length, widths). Besides there is a parameter called scale explained in the previous sub section. Here the value 40 is chosen only for illustration.

```
def all_ellipsoids_3D(Ellipsoid_features, scale):

    number_of_defects = Ellipsoid_features[0].shape[1]

    fig = plt.figure(figsize=(8,8)) # Square figure
    ax = fig.add_subplot(111, projection='3d')

    for i in range(number_of_defects):
        one_ellipsoid_3D(ax, Ellipsoid_features, i, scale)

    x, y, z = Ellipsoid_features[0]

    ax.set_xlim([np.min(x)-50, np.max(x)+50])
    ax.set_ylim([np.min(y)-50, np.max(y)+50])
    ax.set_zlim([np.min(z)-50, np.max(z)+50])
    ax.set_title('3D distribution of estimated defects ellipsoids')

    plt.savefig('fig')
    plt.show()
```

The code above is a loop for all the segmented defects which is time consuming and need to be optimized.

5. Assignment of 3D coordinates

In this section, the methodology towards the 3D reconstruction of segmented defects is explained. Also a python code is introduced as an assignment script.

5.1 Methodology

The input of this step is based on the results of the previous chapters so the accuracy of the 3D assignment is dependent to their precision. In previous chapters we explained how to calculate the standard deviation of each defect over its major and minor axis and also its tilt. In fact those values are used to estimate the center position of each defect with a standard deviation over x-axis considered as an error gap for both (x,y) and (x,z) 2D data. The figure below pictures the defect as an ellipse as presented in previous chapters and illustrate the standard deviation over x.

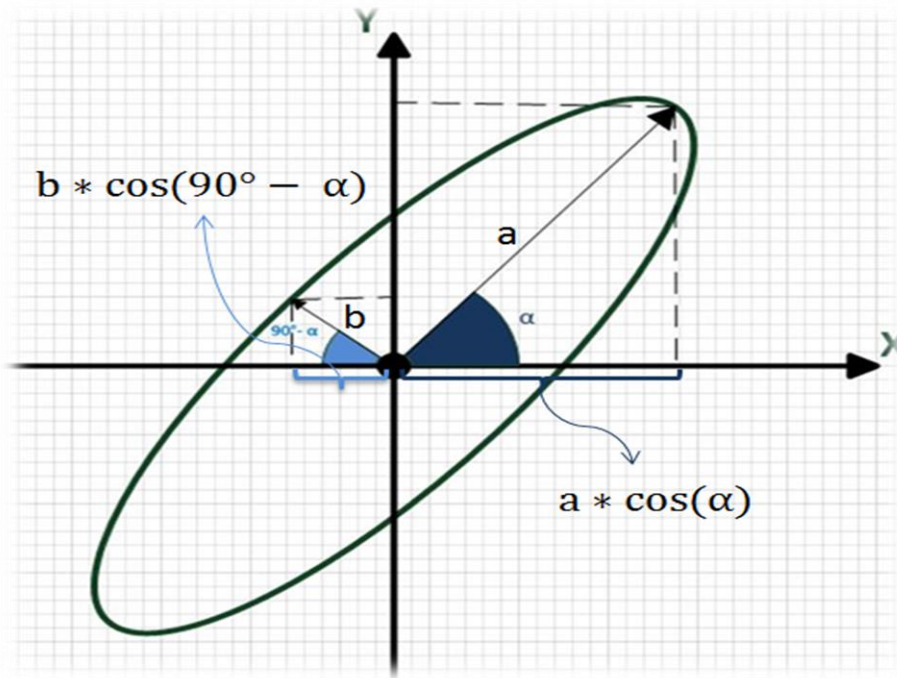


Figure 38 Sketch of the standard deviation over x calculation.

As shown in the figure above one can obtain the standard deviation as follows:

$$\sigma_{x-xy}, \sigma_{x-xz} = \text{Max} ((a * \cos(\alpha)), (b * \cos(90^\circ - \alpha)))$$

Given that:
$$\begin{cases} a = \sqrt{\lambda_1 * S} \\ b = \sqrt{\lambda_2 * S} \end{cases}, \text{ and } \alpha \text{ is the tilt angle of the ellipse/defect}$$

$\lambda_{1,2}, S$ and α are explained in the previous chapters.

Once σ_{x-xy} and σ_{x-xz} were calculated, one can have a list of defects centroids with an assigned error gap for both the (x, y) and (x, z) 2D data. Then the 2D data with the highest value of standard deviation is chosen as reference. The other set of 2D data centroids are verified to be matching inside the interval $[-coef * \sigma_x, +coef * \sigma_x]$ which has 68.2% as confidence interval for $coef = 1$. In fact, this is an assumption presented in the previous chapter to assume that the defect pixels are distributed normally [15].

5.2 The Python code

In this sub section we are going to see an example of 3D assignment using a python script and followed by a visualization of the output 3D reconstruction.

- The first step is to acquire the inputs as: Centroids, a, b and α .
- Secondly the standard deviation over x is calculated throughout the following code:

```
def projection_x(Column):
    feat = list_array(Column)

    x_a = np.cos(np.radians(feat[0])) * np.maximum(feat[1], feat[2]) #a,b
    x_b = np.cos(np.radians(90 - np.abs(feat[0]))) * np.minimum(feat[1], feat[2])

    x = np.maximum(x_a, x_b)

    return x
```

- Then the assignment step is performed using the following code:

```

def assignement(features_xy,centroids_xy,features_xz,centroids_xz,coef=1): #if xy.size > xz.size

    '''xz plan'''
    x_z = projection_x(features_xz)#projection on x axis of xz plan
    xz = list_array(centroids_xz)#centers of defects
    s_dev_xz = list_array(features_xz)#(44,3)array of ellipse features

    '''xy plan'''
    x_y = projection_x(features_xy)
    xy = list_array(centroids_xy)
    s_dev_xy = list_array(features_xy)

    '''matching indexes'''
    coef = coef
    index_xyz = []

    if xy[0].size < xz[0].size :
        xy,xz = xz,xy

    for i,x_xy in enumerate(xy[0].values):
        for k,x_xz in enumerate(xz[0].values):
            if ((x_xy >= (x_xz - coef*x_z[k])) and (x_xy <= (x_xz + coef*x_z[k]))):
                index_xyz.append([i,k])
    print 'There',len(index_xyz),'matching defects'

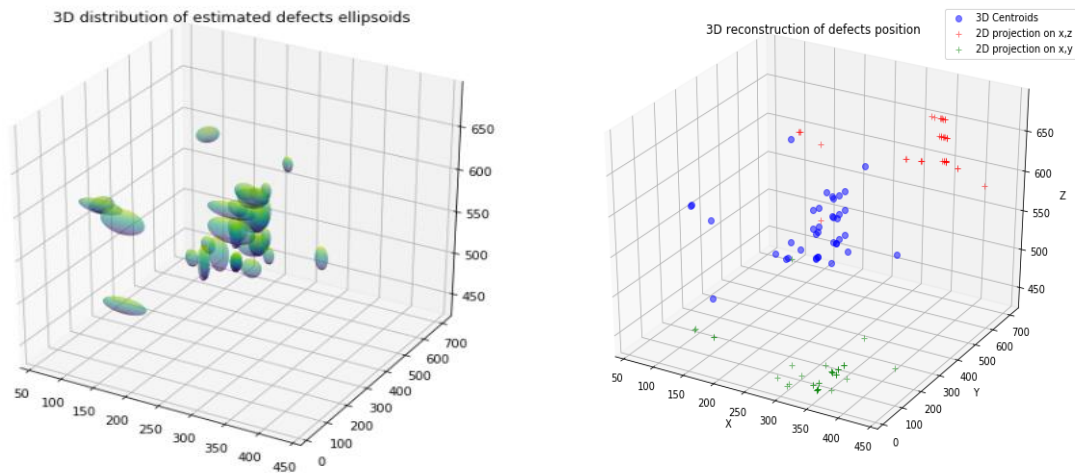
    index_xyz = np.array(index_xyz)

    '''matching coordinates'''
    x = np.array(xy[0])
    x = x[index_xyz[:,0]]
    y = np.array(xy[1])
    y = y[index_xyz[:,0]]
    z = np.array(xz[1])
    z = z[index_xyz[:,1]]
    '''matching stand deviations'''
    sig_x = np.array(s_dev_xy[1])
    sig_x = sig_x[index_xyz[:,0]]
    sig_y = np.array(s_dev_xy[2])
    sig_y = sig_y[index_xyz[:,0]]
    sig_z = np.array(s_dev_xy[1])
    sig_z = sig_z[index_xyz[:,1]]
    '''matching angles'''
    angle_xy = np.array(s_dev_xy[0])
    angle_xy = angle_xy[index_xyz[:,0]]
    angle_xz = np.array(s_dev_xy[0])
    angle_xz = angle_xz [index_xyz[:,1]]

    return np.array([np.array([x,y,z]),np.array([sig_x,sig_y,sig_z]),np.array([angle_xy,angle_xz])])

```

The output is then 3D coordinates with the ellipsoid features $a, b, c, \alpha_{xy,xz}$. Here is a 3D reconstruction of the 2D data set:



5.3 Assumption

In that case example, represented in the figure above, the result can be resumed in the following tables:

| Number of XY data | Number of XZ data | Matching points | |
|----------------------|----------------------|--------------------|--------------------|
| | | XY as reference | XZ as reference |
| 44 | 38 | 39 | 19 |

Table 3 The number of matching defects.

In order to obtain the figure above with more matching defects as seen in the table XY 2D data centroids were chosen as reference. This means that they were compared to the confidence interval of the XY 2D centroids with regard to the standard deviation over x. The standard deviation corresponds to the projection of the XZ a, or b on x-axis (see previous sub section).

In that context, the choice of the reference 2D data set needs to be determined and is considered as an assumption in this work. In fact the 2D data with the highest number of defects is assumed to be the pivot or the reference of the assignment algorithm.

6. GUI development

The whole code was written so far using Python Version 3.6.4. In this chapter it is included in as GUI callbacks and the way to do so is been explained.

6.1 Introduction to GUI

6.1.1 Pre-requirement

In order to make sure that all packages are compatible with the python code script, a virtual environment is used. Also, virtual environments provide a secluded environment with no conflicts with system programs, and they can be easily reproduced on any other system.

Virtual environment

Step 1:

To create a virtual environment, a folder containing our project script should be created first. Then, one should find the location of the Python version (3.6.4) installation on the computer. The following command gives the path:

\$ which python 3.6.4

Step 2:

After taking a note of the location (e.g. /c/Anaconda3/python) one should open a terminal in the project folder and run the following command:

\$ virtualenv -p /location/of /python3.6 myvenv

In order to open a terminal in Windows system, one can use “Git bash”. In fact, “Git Bash” for Windows is a package that is comprised of two parts:

Git: It is a version control system (VCS) which tracks the file changes, commonly used for programming in a team setting.

Bash: It is a UNIX shell command line interface commonly used in different Linux machines.

It's important to make sure that “virtualenv” package is installed; if not the following command can be used:

\$ pip install virtualenv

Step 3:

The previous step creates a new virtual environment in a folder named myvenv inside the project folder. In that last step, this virtual environment is activated. This is done by running the following command:

\$ source myenv/Scripts/activate

Now if the command `python` is typed, it should pick up Python 3.6.3 from within the virtual environment. From now onward, every time we have to run a Python script or install a new module, we will first activate the virtual environment using the preceding command and run or install the module within this new virtual environment.

Using Pew for UNIX users:

Step 1:

To create a virtual environment, one should find the location of the Python version (3.6.4) installation on the computer. The following command gives the path:

`$ which python 3.6.4`

Step 2:

After taking a note of the location (e.g. `/c/Anaconda3/python`), a folder containing our project script should be created first. Then, one runs the following command:

`$ pew new myenv`

It's important to make sure that "pew" package is installed, if not the following command can be used:

`$ pip install pew`

Step 3:

The previous step creates a new virtual environment in a folder named `myenv` inside the project folder. In order to have access to the path from the child process or in other words to add python path to environment variables, the following command is useful:

`$ export PATH=$PATH: location/of/python`

In that last step, this virtual environment is activated. This is done by running the following command:

`$ pew work on myenv`

Now if the command `python` is typed, it should pick up Python 3.6.3 from within the virtual environment. From now onward, every time we have to run a Python script or install a new module, we will first activate the virtual environment using the preceding command and run or install the module within this new virtual environment.

6.1.2 GUI environments

GUI programming deals with graphical objects called widgets. Looking at a window in a typical GUI, the window may consist of buttons, text fields, sliders, and other graphical elements. Each button, slider, text field, etc. is referred to as a widget. There are also “invisible” widgets, called frames, for just holding a set of smaller widgets. A full GUI is a hierarchy of widgets, with a top level widget representing the complete window of the GUI. The geometric arrangement of widgets in parent widgets is performed by a geometry manager [7].

Tkinter:

One of the recommended GUI toolkits is “Tkinter” (pronounced tea-kay-inter) which is Python interface to Tk, the GUI toolkit for Tcl/Tk. Tcl (Tool command language), which is pronounced as tickle, is a popular scripting language in the domains of embedded applications, testing, prototyping, and GUI development. On the other hand, Tk is an open source, multiplatform widget toolkit that is used by many different languages to build GUI programs [8].

In other words, “Tkinter” is just a wrapper around a C extension that uses the Tcl/Tk libraries.



It is very important to pay attention to the Python version. In fact, the “Tkinter” interface is implemented as a Python module — Tkinter.py in Python 2.x Versions and tkinter/__init__.py in Python 3.x Versions. If you look at the source code.

Tkinter is a great tool for the programming of GUI applications in Python. The features that make Tkinter a great choice for GUI programming include the following:

- ✚ It is simple to learn (simpler than any other GUI package for Python)
- ✚ Relatively little code can produce powerful GUI applications
- ✚ Layered design ensures that it is easy to grasp
- ✚ It is portable across all operating systems
- ✚ It is easily accessible, as it comes pre-installed with the standard Python distribution.
- ✚ None of the other Python GUI toolkits have all of these features at the same time. [7]

PyQT:

Apart from its rich and versatile widgets, PyQt offers an easy to use interface designer within its distribution packages. It can spare a lot of time using the mouse and dragging widgets instead of coding any single angle or position. [16]

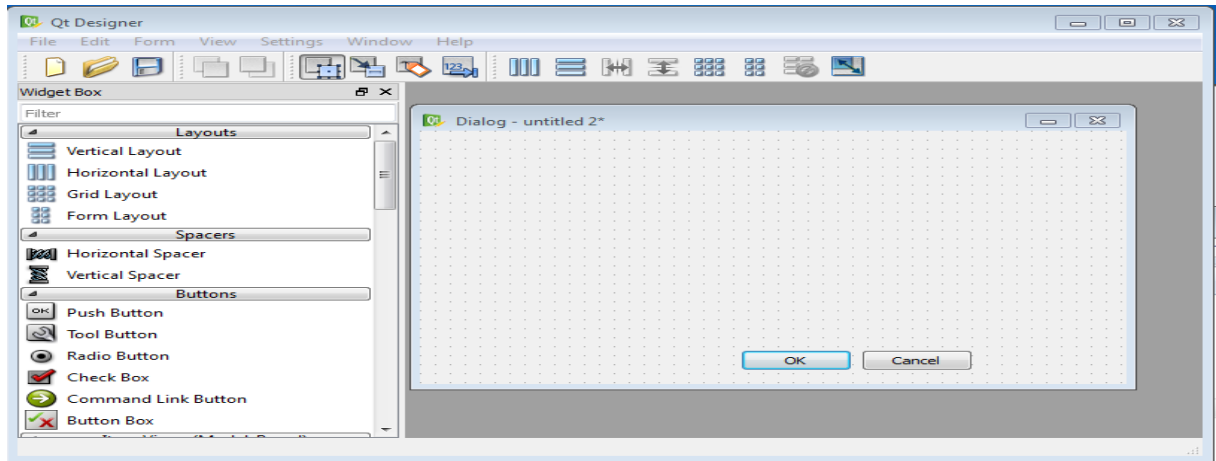


Figure 39 A screen shot of the Qt designer window.

The usage of PyQt designer can be summarized into the following steps (see figure below):

- ✚ Creating the window with dragged in widgets
- ✚ Saving the file as "Filename.ui"
- ✚ Convert the ".ui" file to python code using:
 - ✓ Pyuic5 converter located in 'scripts' of python distribution
 - import sys
 - import os
 - print(os.path.dirname(sys.executable))
 - ✓ And execute: \Location\of\Pyui5 -x \location\of\the filename.ui -o \new\name, on the shell.

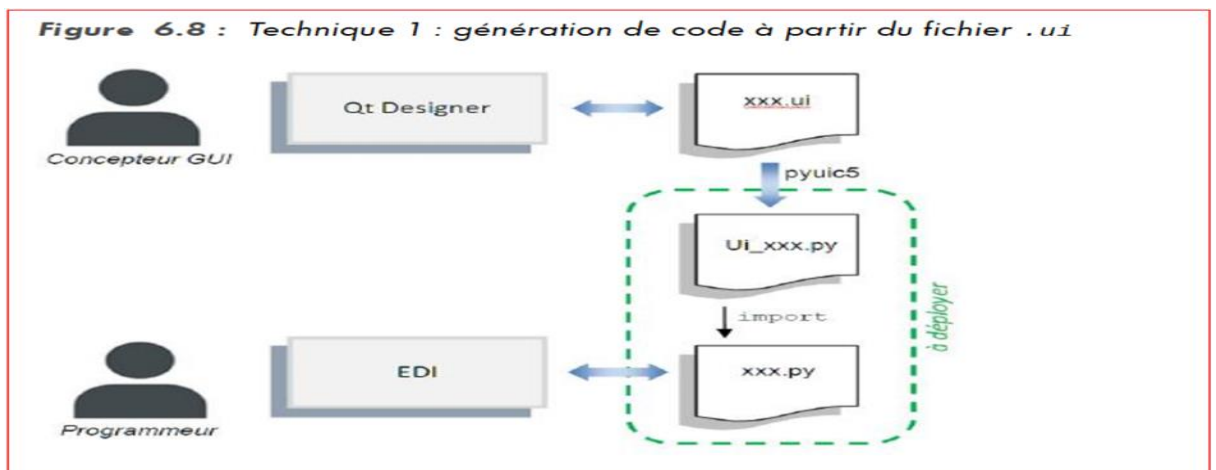


Figure 40 A screen shot of the generation of python code from ".ui" files [8].

6.2 GUI development

The chosen GUI environment is the PyQt due its rich and easy interactive designer interface.

6.2.1 Overview

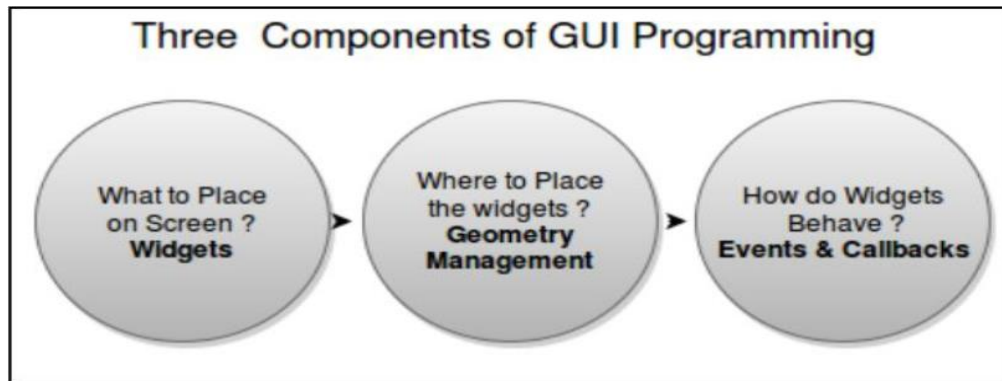


Figure 41 Screen shot of the three components of GUI programming.

According to the figure above showing the three component of GUI programming, the detection tool was developed.

Widget & geometry management:

Using the PyQt designer the design of the basic design of the tool might look like the figure below.

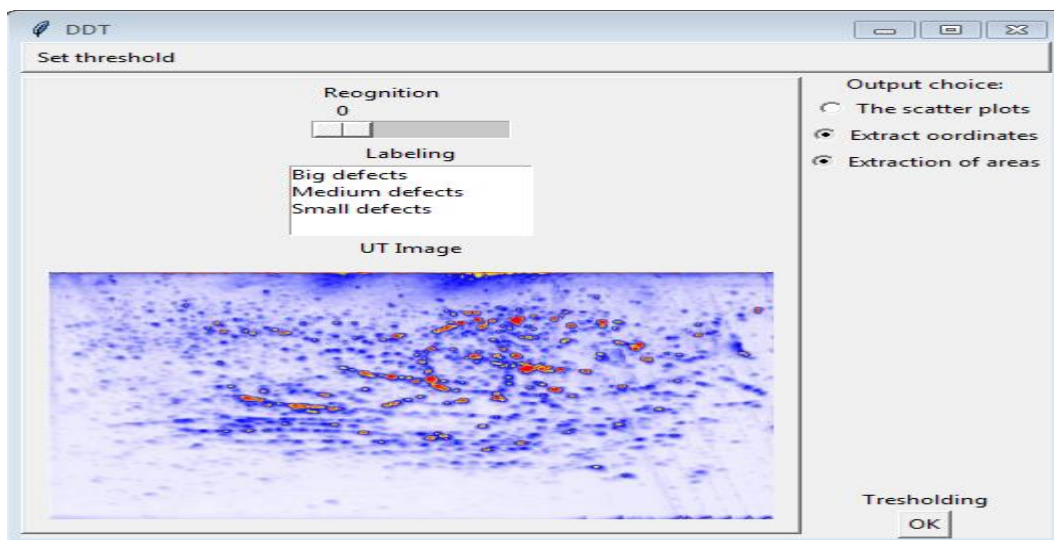


Figure 42 Screen shot of basic detection tool design.

The window contains the following widgets:

- ❖ Top-level Label Button
- ❖ Frame LabelFrame Listbox
- ❖ Menu Menubutton Message
- ❖ OptionMenu PanedWindow Radiobutton
- ❖ Scale Scrollbar Spinbox
- ❖ Text Bitmap Image

6.2.2 GUI options

After we laid out the GUI window, we need to program its behavior. The written code controls how the GUI responds to events such as button clicks, menu item selection, window resizing, or the creation of graphs and images. This programming takes the form of a set of functions, called callbacks, for the GUI figure itself and for each component.

Cropping option:

In order to make resizing of the image easier and quicker, a cropping callback was included as shown in appendices chapter (see cropping script).

After running the mentioned code the following window is obtained:

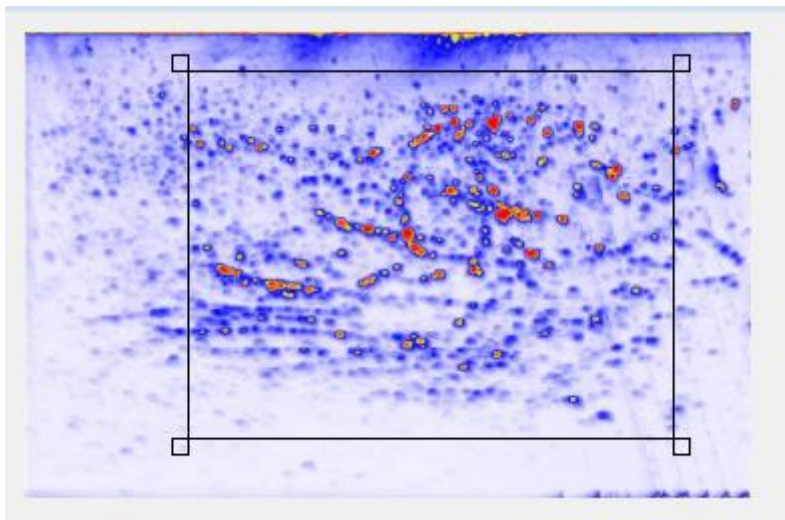


Figure 43 Screen shot of the cropping option window.

When the 'mousedown' event is triggered with dragging one of the four corners of the polygon above, one can define the borders of the detected defects. The artifacts and the extra scale of the original UT image is avoided and thus results are quicker and accurate.







Basic tool options:

The following form of code is included in the same design python code

```
def The_detection_tool():
```

```
    # giving results when button is clicked
```

The included functions were explained in the previous chapters:

-  Image processing
-  Data cleaning
-  Defects recognition
-  Smoothing and extrema detection
-  Segmentation
-  3D reconstruction

The results are given either as graphs or exported CSV files of detected defects coordinates and features.

7. Discussion:

Within this chapter we are going to discuss the different possibilities to make this defect detection tool more accurate. In that context, an overview of the whole tool is displayed below with indicating the key parameters to enhance it. Also for more friendly use the GUI is to be entailed with some pre-processing options as well as post-processing ideas.

7.1 Overview

In order to understand what to enhance one has to know the different inputs all over the defect detection process.

7.1.1 Ontology

An ontology encompasses a representation, formal naming and definition of the categories, properties and relations between the concepts, data and entities that substantiate one, many or all domains of discourse. It is a recommended way to organize the different inputs and outputs as well as the relations between the different functions of the defect detection tool class.

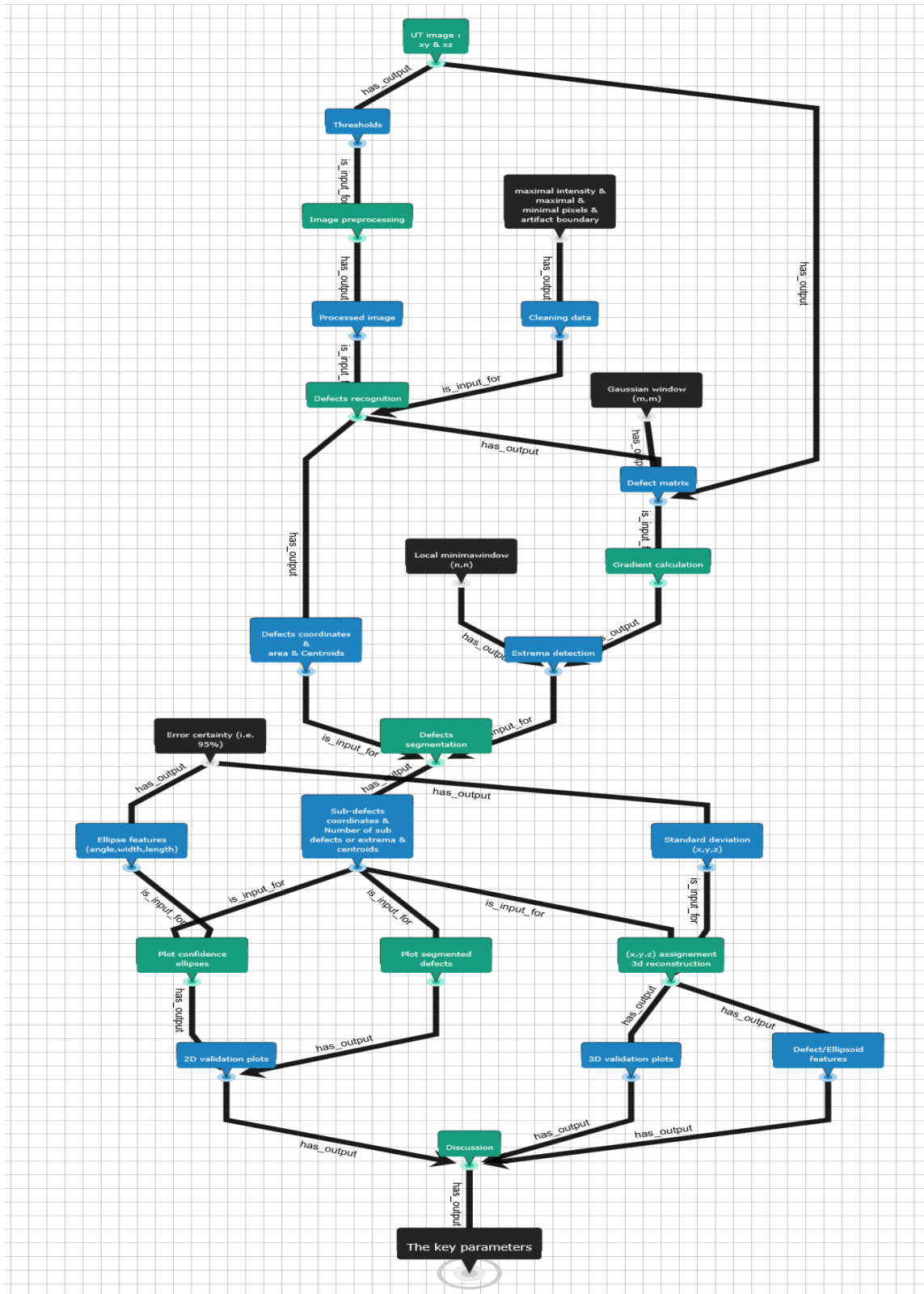


Figure 44 Ontology of the defect detection tool.

The whole process is summarized in 6 steps:

- ✚ Image processing
- ✚ Data cleaning
- ✚ Defects recognition
- ✚ Smoothing and extrema detection
- ✚ Segmentation
- ✚ 3D reconstruction

7.1.2 Performance

For each step of the mentioned process above, the following table resume the required time for the code execution.

| Steps | 1 | 2 | 3 | 4 | 5 | 6 |
|----------|----------------------|----------------------|------|----------------------|----------------------|----------|
| Time (s) | 320×10^{-3} | 183×10^{-3} | 7.98 | 590×10^{-3} | 410×10^{-3} | ~ 0 |

Table 4 Required Wall time for every step of the developed tool process.

The table above is showing where the performance enhancements can be pursued. Also it gives estimation about the required overall time for the GUI tool. However, the visualization time is not included. It is shown apart in the table below.

| Graphical outputs | 2D plots(ellipses) | 3D plots (ellipsoids) |
|-------------------|--------------------|-----------------------|
| Time (s) | 330.10^{-3} | 110 |

Table 5 Required wall time for the output visualization.

The 3D ellipsoid code has to be definitely optimized given its huge time consuming functions.

Despite of the high required time in seconds scale, the tool is extremely time-saving for an employee doing this job manually. But only if the results turn to be accurate?!

7.1.3 Validation

The validation data are obtained throughout **a whole working day** that might be **even more** if the UT images contain a more defects. The results were validated based on a linear regression with the validation data. In statistics, the coefficient of determination, denoted r^2 is the proportion of the variance in the dependent variable that is predictable from the independent variable(s).

The better the linear regression fits the tool data in comparison to the validation data, the closer the value of r^2 is to 1.

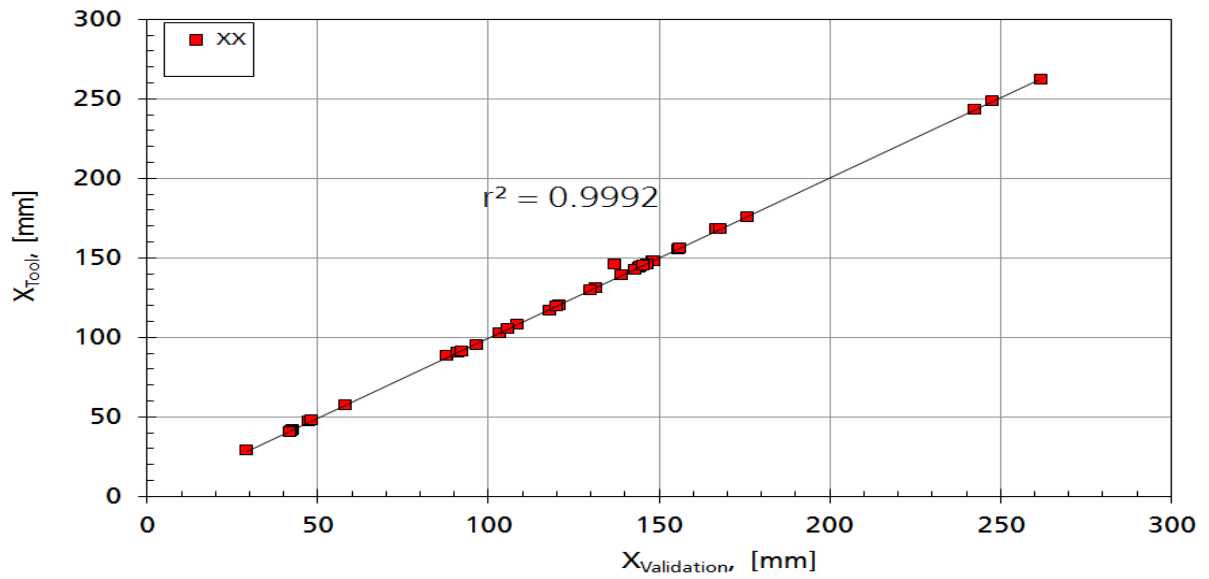


Figure 44 Linear regression of the X-axis coordinates of defects centroids.

The coefficient of determination is close to 1 so the X coordinate obtained using the tool match with those given by the validation data. Also almost all defects were detected (i.e. 38/40 defect on the validation data).

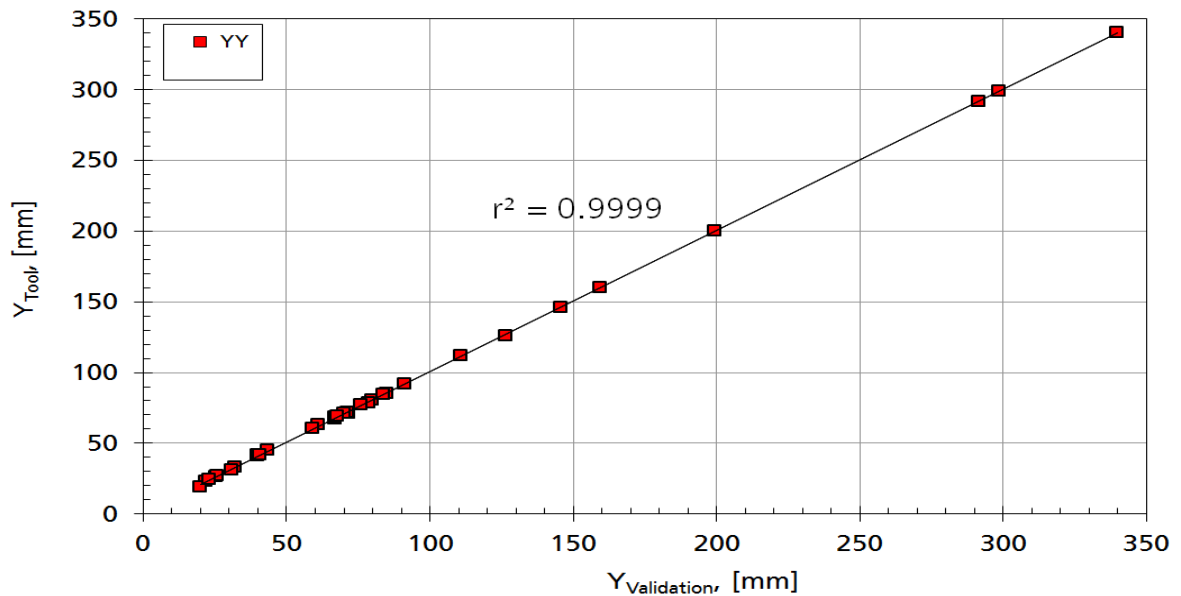


Figure 45 Linear regression of the Y-axis coordinates of defects centroids.

The coefficient of determination is close to 1 so the Y coordinate obtained using the tool match with those given by the validation data. Also almost all defects were detected (i.e. 38/40 defect on the validation data).

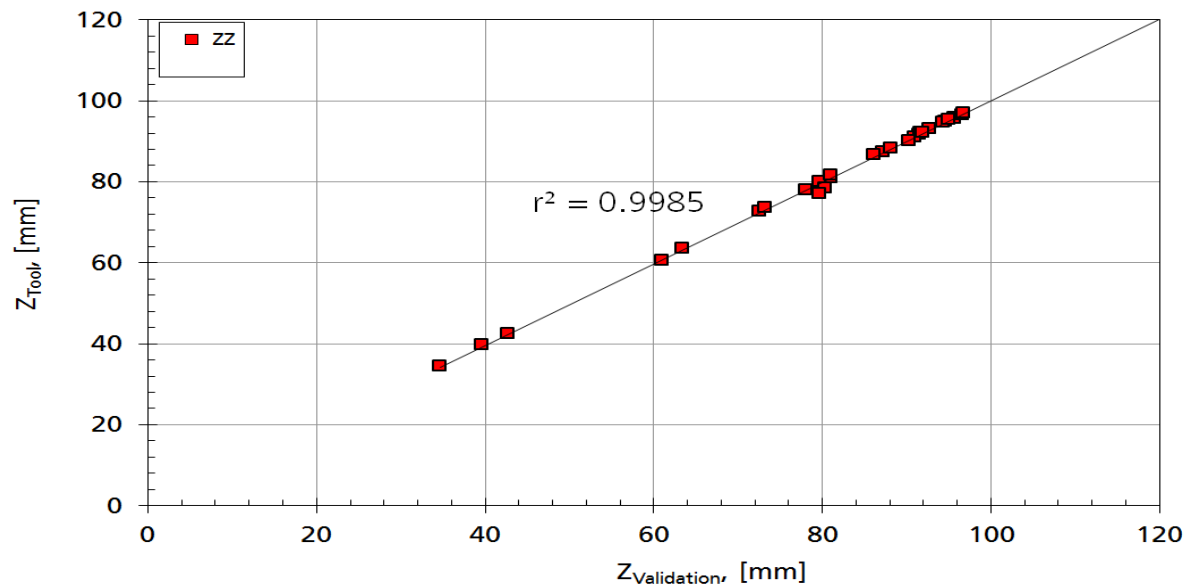


Figure 46 Linear regression of the Z-axis coordinates of defects centroids.

The coefficient of determination is less close to 1 than the other coordinates but still the X coordinates obtained using the tool match with those given by the validation data. However not all defects were detected from the XZ 2D UT image.

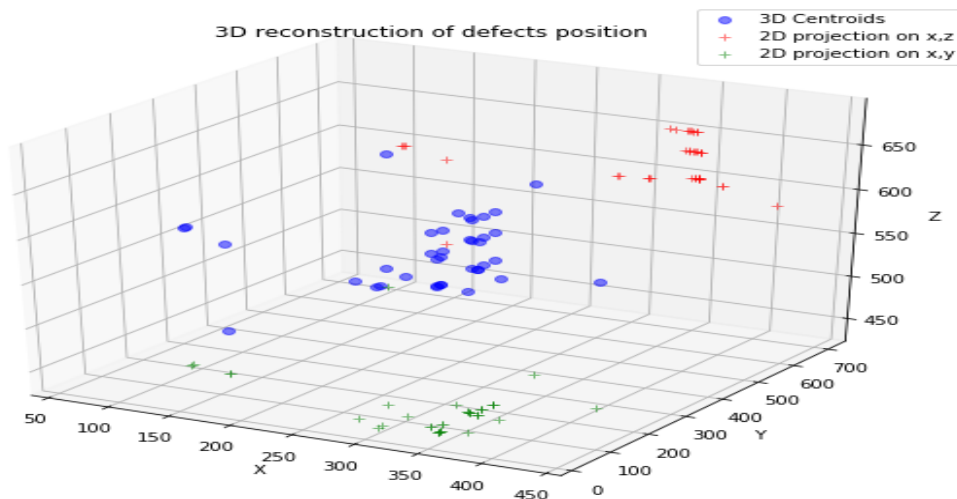


Figure 47 3D coordinates of matching defects centroids.

In the process of the 3D coordinates assignment some coordinates are not assigned and those the total number of defects is reduced. In order to have more accurate results, another 2D Utimage is required that is the YZ plan in order to have a precise overview on defects. Also the tool needs to be enhanced during the segmentation process in order to avoid under or over-segmentation as shown in the figure above.

7.2 Enhancements

Beside the performance enhancements there is what is more important that is the accuracy of the tool output. As shown in the validation sub section, the results are not 100% identical and thus the tool has to be enhanced.

7.2.1 The key parameters







The developed tool has two types of parameters: those which are predefined by the component and others which need to be optimized furthermore by the user. In order to have a better modeling of defects the parameters shown in 'Black boxes' on the ontology draft are the ones that need to be optimized.

Here is a description of each parameter and the reason why it is considered to be a key parameter:

Threshold:

The chosen Threshold according to the assigned color to defects in the UT image is used to recognize defects and label them as well to big (red & yellow), medium (yellow) and small (blue).

Hue is the color portion of the model, expressed as a number from 0 to 360 degrees:

-  Red falls between 0 and 60 degrees.
-  Yellow falls between 61 and 120 degrees.
-  Green falls between 121-180 degrees.
-  Cyan falls between 181-240 degrees.
-  Blue falls between 241-300 degrees.
-  Magenta falls between 301-360 degrees.

maximal intensity:

The maximal intensity is required to get rid of the noisy pixels and also some artifacts. This is a user parameter to be defined to clean 2D data.

The Gaussian window :

The Gaussian window or the dimension of the smoothing kernel applied to each defect. It is considered as a key parameter for better precision of the gradient calculus.

The local extrema window:

The local extrema window or the dimension of the window used as a mask to enable the extrema determination. This is absolutely to be considered as a key parameter of the detection tool.

confidence ellipse error:

Once the defect pixels are collected an estimation of their distribution is made by the mean of the confidence ellipse according to a certain chosen scale. This scale is representing the percentage of how many pixels are surrounded by the so called error ellipse. It is also a coefficient for the estimated standard deviation over x deduced and the angle of ellipse, so it is definitely a key parameter.

For better accuracy those parameters can be optimized based on parameter identification methods using the validation data.

7.2.2 Other paths



In addition to finding an optimum value of the considered key parameters one can also explore other methods of segmentation. For instance, other gradient calculation filter can be used instead of the Sobel filter. This filter is more adapted to edge detection and our goal would be rather “Blob detection”.

Blob detection

“In computer vision, blob detection methods are aimed at detecting regions in a digital image that differ in properties, such as brightness or color, compared to surrounding regions. Informally, a blob is a region of an image in which some properties are constant or approximately constant; all the points in a blob can be considered in some sense to be similar to each other.”[17]

Within this work the Laplacien of Gaussian was explored as one of the alternative methods in chapter 3. Besides, one of the recommended methods is maximally stable extremal regions (MSER).

Extremal regions in this context have two important properties that the set is closed under:

-  Continuous transformation of image coordinates. This means it is affine invariant and it doesn't matter if the image is warped or skewed.
-  Monotonic transformation of image intensities. The approach is of course sensitive to natural lighting effects as change of day light or moving shadows.

7.3 Conclusion

The developed tool presented in this work is suitable for the detection of defects when the UT images are provided with the same color intensity. It is time saving for the whole defect recognition to the 3D reconstruction final step. The principle is to use UT images for XY plan and the XZ plan in order to determine the 3D distribution of defects. For this application the mentioned tool is not 100% accurate and has to be further enhanced as it was discussed in chapter 7.

In order to optimize the detection algorithm furthermore, more precision is required regarding the extrema determination. The gradient descent method is not the most adapted method for our case, other algorithm would be efficient according to experts. Some of those recommended algorithms are for instance:

- The greedy algorithm
- The ant colony optimization algorithm
- The genetic algorithm

After validating the detection tool, the next step is to optimize its running time and reduce especially the time needed to display the 3D graphs. Also the GUI should acquire as maximum options as possible to make the usage easier and quicker. This will be the continuation of the work as a ‘Student assistant’ within the working team.

8. Sources

- [1]. H. A. Richard und M. Sander. Ermüdungsrisse: Erkennen, sicher beurteilen, vermeiden. Auflage. Berlin Heidelberg New York: Springer-Verlag. 2012.
- [2]. [https://en.wikipedia.org/wiki/Scrum_\(software_development\)#Workflow](https://en.wikipedia.org/wiki/Scrum_(software_development)#Workflow)
- [3]. https://en.wikipedia.org/wiki/Color_space
- [4]. https://opencv-pythontutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html
- [5]. https://scikit-image.org/docs/dev/auto_examples/segmentation/plot_watershed.html
- [6]. Langtangen, Hans Petter. Python Scripting for Computational Science. Springer 2008.
- [7]. Chaudhary, Bhaskar. Tkinter GUI Application Development Blueprints. 2018.
- [8]. https://www.researchgate.net/post/What_are_the_differences_in_first_order_derivative_edge_detection_algorithms_and_second_order_edge_detection_algorithms
- [9]. <http://tutorial.math.lamar.edu/Classes/CalcIII/RelativeExtrema.aspx>
- [10]. https://developer98.openinventor.com/APIS/RefManCpp/group_image_segmentation_region_growing.html
- [11]. https://developer98.openinventor.com/APIS/RefManCpp/group_image_segmentation_region_growing.html
- [12]. <https://www.datascience.com/blog/k-means-clustering>
- [13]. <https://www.visiondummy.com/2014/04/draw-error-ellipse-representing-covariance-matrix/>
- [14]. <https://people.richland.edu/james/lecture/m170/tbl-chi.html>
- [15]. https://en.wikipedia.org/wiki/Standard_deviation
- [16]. <https://www.youtube.com/watch?v=ksW59gYEI6Q>
- [17]. https://en.wikipedia.org/wiki/Blob_detection#The_Laplacian_of_Gaussian

[18]. Weber, F., Ultraschall-Untersuchung von Schmiedeteilen, Fraunhofer IZFP, Saarbrücken, 2017.

Other links

1. For colors code the following webpage was taken as a reference
 - <http://colorizer.org/>
2. To familiarize with image processing vocabulary
 - https://www.tutorialspoint.com/dip/concept_of_masks.htm
 - <http://cns-alumni.bu.edu/~slehar/fourier/fourier.html>
3. The coding is based on the package Opencv and its comprehensive documentation.
 - https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html#converting-colorspaces
 - <https://towardsdatascience.com/build-your-own-convolution-neural-network-in-5-mins-4217c2cf964f>
 - <https://www.youtube.com/watch?v=rTawFwUvnLE>
4. The comprehension of the CNN approach was reached using basically the following webpages:
 - <http://cs231n.github.io/convolutional-networks/#overview>
5. From Git command lines to GitHub/local server
 - <https://en.wikipedia.org/wiki/Git>
 - [https://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))

To emulate a bash environment on windows→

- <https://stackoverflow.com/questions/45034549/difference-between-git-gui-git-bash-git-cmd>
- <https://www.quora.com/What-is-the-difference-between-Git-GUI-Git-Bash-and-Git-CMD>

For image transformation

- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_geometric_transformations/py_geometric_transformations.html#geometric-transformations

Appendices

Thresholding python code:

```
def detect(path, lower_bound, upper_bound, lower_bound_1 = [0,0,0], upper_bound_1 = [0,0,0]):

    img = cv2.imread(path,1) # BGR color image reading and 0 flag for gray
    #img = cv2.medianBlur(img,5) #blur to make the transition smooth from color to color
    #img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) #convert to RGB for real image plot

    # converting it into Hue, saturation, value (HSV)
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    '''Threshold'''

    # define range of red color in HSV
    lower = np.array(lower_bound)
    upper = np.array(upper_bound)

    # Threshold the HSV image to get only red colors
    mask_0 = cv2.inRange(hsv, lower, upper)
    mask_1 = cv2.inRange(hsv, np.array(lower_bound_1), np.array(upper_bound_1))
    mask = mask_0 + mask_1

    # Bitwise-AND mask and original image
    res = cv2.bitwise_and(img, img, mask = mask)

    return res
```

Shaping or resizing the image:

```
def shaping(image,i,j,resol = [300,300]):

    img = image
    rows,cols = img.shape

    frame = [[0,0],[cols,0],[0,rows],[cols,rows]]
    dimension = [[0,0],[resol[1],0],[0,resol[0]],[resol[1],resol[0]]]

    pts1 = np.float32(frame)
    pts2 = np.float32(dimension)

    M = cv2.getPerspectiveTransform(pts1,pts2)
    shaped = cv2.warpPerspective(img,M,(j,i))

    return shaped
```

```

def pre_shaping(image,resol = [305,366]):

    img = image
    rows,cols = img.shape

    frame = [[0,0],[cols,0],[0,rows],[cols,rows]]
    dimension = [[0,0],[resol[1],0],[0,resol[0]],[resol[1],resol[0]]]

    pts1 = np.float32(frame)
    pts2 = np.float32(dimension)

    M = cv2.getPerspectiveTransform(pts1,pts2)
    pre_shaped = cv2.warpPerspective(img,M,(resol[1],resol[0]))

    return pre_shaped

def real_shaping(normalized_image,rows_dim,cols_dim,i = 8,j = 17):

    img = normalized_image
    rows,cols = img.shape

    frame = [[i,j],[rows,j],[i,cols],[rows,cols]]
    dimension = [[0,0],[cols_dim,0],[0,rows_dim],[cols_dim,rows_dim]]

    pts1 = np.float32(frame)
    pts2 = np.float32(dimension)

    M = cv2.getPerspectiveTransform(pts1,pts2)

    real_shape = cv2.warpPerspective(img,M,(cols_dim,rows_dim))

    return real_shape

```

Cleaning data:

```

def data_reduction(polygon,coordinates,area):
    old_dim = len(coordinates)
    p = pth.Path(polygon)
    mask = p.contains_points(coordinates)
    reduced_coord = coordinates[mask]
    reduced_area = area[mask]
    new_dim = len(reduced_coord)
    print(old_dim-new_dim,' values were reduced')
    return reduced_coord,reduced_area

'''Function call'''

area = area
coordinates = coord
polygon = np.array([[0,35],[560,35],[560,200],[0,200]])
reduced_coord, reduced_area = data_reduction(polygon,coordinates,area)

```

Finding the local minima:

```

""" this function produce a rolling window which is just a view on the data:
= a special way to move in the data. No copy is made.
"""
def rolling_windows_img(a,kshape):
    outShape=(a.shape[0]-kshape[0]+1,)+(a.shape[1]-kshape[1]+1,)+kshape
    outStrides=a.strides+a.strides
    return np.lib.stride_tricks.as_strided(a,shape=outShape,strides=outStrides)

```

```

def findLocalMin(data, kshape, minimum):
    dataRoll = rolling_windows_img(data, kshape)
    a, b = dataRoll.shape[0], dataRoll.shape[1]
    L = a*b
    W = kshape[0]*kshape[1]
    dataRoll = dataRoll.reshape(L, W)

    if minimum == True:
        armin = np.argmin(dataRoll, axis=1)
    else:
        armin = np.argmax(dataRoll, axis=1)

    res = np.array([], dtype=np.int8)
    extrema = 0
    for i, j in enumerate(armin):
        where = np.where((dataRoll[i, :] == dataRoll[i, j]))[0]

        if (j == W//2 and where.size == 1):
            coord = np.array([i//b + (kshape[0])//2, (i%a + (kshape[1])//2)])
            res = np.append(res, coord)
            extrema += 1

    res = res.reshape(res.size//2, 2)

    return res, extrema

```

Defects collection:

```

def add_neighbours_in_stack(image, a, b, stack, i, j, inComponent):
    inComponent[i, j] = True
    neighbours = [(i+1, j), (i-1, j), (i, j-1), (i, j+1)]
    #neighbours = [(i-1, j-1), (i+1, j-1), (i+1, j), (i-1, j), (i, j-1), (i, j+1), (i-1, j+1), (i+1, j+1)]
    for (k, l) in neighbours:
        if 0<=k<image.shape[0] and 0<=l<image.shape[1]:
            if (image[k, l]>a and image[k, l]<b) and not inComponent[k, l]:
                inComponent[k, l] = True
                stack.append((k, l))

    return

```

```

def connected_component_nb(image, seuil, i0, j0, inComponent):
    stack = [(i0, j0)]
    defect = []
    nb=1
    barycentre = [i0, j0]
    while len(stack)>0:
        (i, j) = stack.pop()
        nb+=1
        add_neighbours_in_stack(image, seuil, 260, stack, i, j, inComponent)

        defect.append((i, j))
        barycentre[0] += i
        barycentre[1] += j

    barycentre[0] /= nb
    barycentre[1] /= nb

    return nb-1, barycentre, defect

```

```

def all_connected_components(image, seuil):
    inComponent = np.empty(image.shape, dtype=np.bool)
    inComponent[:, :] = False # stoping test

    baryAndSizes = []
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if image[i, j]>seuil and not inComponent[i, j]:
                baryAndSizes.append(connected_component_nb(image, seuil, i, j, inComponent))

    return baryAndSizes

```

```
def defect_collection(image,thresh,plot):

    baryAndSizes = all_connected_components(image, thresh)
    number_of_defects = len(baryAndSizes)

    x = np.array([],dtype=float)
    y= np.array([],dtype=float)
    size = np.array([],dtype=np.int8)
    defects = np.array([],dtype=int)

    '''defects list'''
    for i in range(number_of_defects):#baryAndSizes = nb, barycentre,defect
        x = np.append(x, baryAndSizes[i][1][0])
        y = np.append(y, baryAndSizes[i][1][1])
        size = np.append(size, baryAndSizes[i][0])
        defects = np.append(defects, baryAndSizes[i][2])

    '''plot'''
    if plot:
        fig, ax = plt.subplots(1,3, figsize = (22,12))
        fig.suptitle('Results', fontsize=16)
        ax[0].set_xlim(0,image.shape[1])
        ax[1].set_xlim(0,image.shape[1])
        ax[2].set_xlim(0,image.shape[1])

        title = ['Detected defects centers','Detected defects as discs']
        ax[0].imshow(255-image, cmap='gray', vmin = 0, vmax = 255,)
        ax[0].scatter(y, x, marker='+', s=size, c = 'r', alpha = 0.5)
        ax[0].set_title(title[0])

        ax[0].set_xlabel('x')
        ax[0].set_ylabel('y')

        ax[1].imshow(255-image, cmap='gray', vmin = 0, vmax = 255,)
        ax[1].scatter(y, x, marker='o', s=size, c = 'b', alpha = 0.5)
        ax[1].set_title(title[1])

        ax[1].set_xlabel('x')
        ax[1].set_ylabel('y')
        fig.show()

    return y, image.shape[0]-x, size,defects #return the coordinates in x,y axis
```

Conversion from coordinates to pixels:

```
def xy_ij(xy_coord,dim_y):

    xy_coord = np.array(xy_coord)

    ij_coord = np.fliplr(xy_coord)
    ij_coord[:,0] = dim_y - ij_coord[:,0]

    return ij_coord

def ij_xy(ij_coord,dim_y):

    ij_coord = np.array(ij_coord)

    ij_coord[:,0] = dim_y - ij_coord[:,0]
    xy_coord = np.fliplr(ij_coord)

    return xy_coord
```

Get the intensity of defects coordinates:

```
def get_defect_intensity(img,defect): #pixel should np.array (x,y)

    defect = np.array(defect)
    #defect = xy_ij(defect,img.shape[0])
    i = defect[:,0]
    j = defect[:,1]
    intensity = img[i.astype(int),j.astype(int)]

    return intensity
```

Clean defects:

```
def clean_defects(pandasoutput, lines_to_delete):  
    output = pandasoutput.drop(lines_to_delete)  
    return output
```

Defect matrix:

```
def defect_matrix(img, defect, Gaussianshape):  
    defect = np.array(defect)  
  
    i_min = np.min(defect[:,0]) #starting line index  
    i_max = np.max(defect[:,0]+1) #last line index (+1 for python)  
    j_min = np.min(defect[:,1]) #starting column index  
    j_max = np.max(defect[:,1]+1) #last column index (+1 for python)  
  
    defect_shape = img[i_min:i_max, j_min:j_max].shape  
  
    dim = np.max(defect_shape)  
    matrix = np.zeros((dim, dim))  
  
    #assert (img[i_min:i_max, j_min:j_max][img[i_min:i_max, j_min:j_max] != 0.].size  
            #== get_defect_intensity(img, defect).size)  
  
    matrix[0:defect_shape[0], 0:defect_shape[1]] = img[i_min:i_max, j_min:j_max].copy()  
  
    matrix = cv2.GaussianBlur(matrix, (Gaussianshape, Gaussianshape), 0) #smoothing  
  
    return matrix
```

Number of extrema:

```
def extrema_number(local_window_dim, Mat):  
    kshape = min(local_window_dim, Mat.shape[0])  
  
    if kshape%2 == 0:  
        kshape = kshape - 1  
  
    mini, number_extrema = findLocalMin(Mat, (kshape, kshape), True)  
  
    if number_extrema == 0:  
        number_extrema = 1  
  
    return number_extrema
```

Ellipse features:

```
def ellipse_features(defect,s_error):

    defect = defect.T#defect should be an array of shape (2,...)!!

    cov = np.cov(defect)
    w, v = LA.eig(cov)#eigen values, eigen vectors

    index = np.argmax(w) #the cooresponding value to the biggest eigen value

    if np.abs(v[0,index]) == 0:
        angle = np.sign(v[1,index])*90.
    else:
        angle = np.arctan(v[1,index]/np.abs(v[0,index]))
        angle = math.degrees(angle)

    stand_dev_x= np.sqrt(np.float(s_error)*w[0])#Standard deviation on the direction of eigen vector of position 0

    stand_dev_y= np.sqrt(np.float(s_error)*w[1])#Standard deviation on the direction of eigen vector of position 1

    return angle, stand_dev_x,stand_dev_y
```

Defects segmentation:

```
def separation(img,case,number_extrema,s_error):

    points = np.array(case)#i,j
    pts = ij_xy(points,img.shape[0])

    '''
    plt.plot(pts[:,0],pts[:,1],"o");
    '''

    """creation of the algo"""
    kmeans=KMeans(n_clusters=number_extrema)
    """computation of centroids"""
    kmeans.fit(pts)
    """here is the result"""
    centroids = kmeans.cluster_centers_
    labels = kmeans.predict(pts)

    '''
    plt.scatter(pts[:,0],pts[:,1],c=labels,marker="+");
    for centroid in centroids:
        plt.plot(centroid[0],centroid[1],"o")
    '''

    assertion = 0
    sub_defects = []
    Features = []
    for i in range(number_extrema):
        sub_defects.append(pts[labels == i])#x,y replace with points if i,j
        Features.append([ellipse_features(sub_defects[i],s_error)])
        assertion += sub_defects[i].size//2

    assert (assertion == points.size//2)

    return centroids,sub_defects,Features #x,y
```


Defects output:

```
def defects_output(img,x, y, size,alldefects,upper_intensity):

    alldefect = alldefects.reshape((alldefects.size//2,2)) #i,j

    alldefect = alldefect.tolist()
    centers = np.stack((x, y), axis=-1)
    mean_intensity = []

    output = pd.DataFrame([[[],[], [],[]]]).T

    drop=[]
    for i,j in enumerate(size):
        ieme_defect = pd.DataFrame(alldefect[0:j])
        i_max = np.max(ieme_defect[0])
        Intensity = get_defect_intensity(img,ieme_defect)
        mean_intensity = np.mean(Intensity)
        if mean_intensity > upper_intensity or j > 3*upper_intensity or i_max < z[0] or i_max > z[1]:
            drop.append(i)

        ith_line = pd.DataFrame([[centers[i,:].round(2), size[i],mean_intensity.round(2),ieme_defect]],
                                index = np.array([i]))
        output = output.append(ith_line)
        alldefect = alldefect[j:]

    output.columns = ['Centers in (x,y)', 'Area','Average intensity', 'the ith defect in (i,j)']
    assert(np.array_equal(np.array([output['the ith defect in (i,j)'][i].size//2 for i in range(len(size))]),size))

    output['Area']=output['Area'].astype(int)

    output = clean_defects(output,drop)
    print len(drop) , 'defects were cleared '

    return output #i,j
```

Defects segmentation:

```

def defects_separation(img,output,Localshape,Gaussianshape,s_error):#dataframe output of defects after noise and art

separated_output = pd.DataFrame([]).T

for index, row in output.iterrows():

    ieme_defect = row['the ith defect in (i,j)']

    '''matrix of each defect'''

    matrix = defect_matrix(img,ieme_defect,Gaussianshape)

    '''Gradient norm'''

    operators=getCompassOperators("robinson2")
    all_grad = np.abs(np.array([sg.convolve(matrix, operators[i,:,:], "valid") for i in range(4)]))
    Norme = np.sqrt(all_grad[0,:,:]**2 + all_grad[1,:,:]**2+all_grad[2,:,:]**2+all_grad[3,:,:]**2)

    '''Extrema'''

    if matrix.shape[0] <=8:
        extrema = 1
    else:
        extrema = extrema_number(Localshape,Norme)

    '''separation and ellipse_features'''

    centroids,sub_defects,Features = separation(img,ieme_defect,extrema,s_error)

    ith_line = pd.DataFrame([[extrema,centroids.round(2),sub_defects,Features]],index = np.array([index]))
    separated_output = separated_output.append(ith_line)

separated_output.columns = ['extrema','centroids','sub_defects','Features']

separated_output['extrema']=separated_output['extrema'].astype(int)

return separated_output #x,y

```

Plot defects:

```

def plot_defects(img,output,localwindow,gaussianwindow,segmentation,s_error):

fig=plt.figure(figsize=(12,9))

output_subdefect = defects_separation(img,output,localwindow,gaussianwindow,s_error)

index = output_subdefect['extrema'][output_subdefect['extrema'] >= segmentation].index #indexes of the ouput defects
values = output_subdefect ['extrema'][output_subdefect['extrema'] >= segmentation].values # number of subdefects acc

Number_of_defects = sum(output_subdefect ['extrema'].values)

defects = output_subdefect['sub_defects'][output_subdefect['extrema'] >= segmentation]#defects column according to s
for k,i in enumerate(index):
    case = defects[i]
    seg = values[k]
    for j in range(seg):
        if seg == 1:
            plt.plot(case[j][:,0],case[j][:,1],"k.");
        else:
            plt.plot(case[j][:,0],case[j][:,1],".");

plt.title('defects segmented')
plt.show();
print 'There are ', Number_of_defects, 'total detected detected'

return output_subdefect

```

Plot ellipses:

```
def plot_ellipse(ax,centroid,angle, stand_dev_x,stand_dev_y,defect,fill = False,edgecolor = 'black'):  
  
    #fig = plt.figure(figsize=(16,16)) #decommented when drwing one ellipse  
    #ax = fig.add_subplot(111, aspect='equal')#decommented when drwing one ellipse  
    #ax.scatter(defect[:,0], defect[:,1],marker="+")  
    ax.imshow(np.flip(imgtosuperimpose, 0),cmap='jet')  
  
    x = centroid[0] #x coordinate  
    y = centroid[1] #y coordinate  
  
    e = patches.Ellipse((x, y), stand_dev_y, stand_dev_x,angle=angle, linewidth=2, fill=fill,edgecolor=edgecolor)  
  
    ax.add_patch(e)  
    ax.set_title('Estimated detected defects with their error ellipse');  
  
    ax.add_artist(e)  
    e.set_alpha(0.9)  
    e.set_facecolor([0.1,1,0])#green
```

```
def plot_all_ellipses(sub_defects,segmentation,s_error):  
  
    fig = plt.figure(figsize=(12,9))  
  
    ax = fig.add_subplot(111, aspect='equal')  
  
    index = sub_defects ['extrema'][sub_defects['extrema'] >= segmentation].index  
    #indexes of the ouput defects according to segmentation  
    values = sub_defects['extrema'][sub_defects['extrema'] >= segmentation].values  
    # number of subdefects according to segmentation  
  
    for k,i in enumerate(index):  
  
        seg = values[k]  
  
        for j in range(seg):  
            '''Ellipse features'''  
            case = sub_defects['sub_defects'][i][j]  
            centroid = sub_defects['centroids'][i][j]  
            angle, stand_dev_x,stand_dev_y = sub_defects['Features'][i][j][0];  
  
            if seg == 1:  
  
                plot_ellipse(ax,centroid,angle,2* min([stand_dev_y,stand_dev_x]),  
                             2*max([stand_dev_y,stand_dev_x]),case,True)#fill and edge color are optional arg  
  
            else:  
  
                plot_ellipse(ax,centroid,angle, 2*min([stand_dev_y,stand_dev_x]),  
                             2*max([stand_dev_y,stand_dev_x]),case,True,'blue')#fill and edge color are optional arg  
  
    return ax
```

XY/XZ defects example:

| | Centers in (x,y) | Area | Average intensity | the ith defect in (i,j) |
|----|------------------|------|-------------------|--------------------------------------|
| 0 | [589.0, 781.0] | 48 | 174.35 | 0 1 0 19 567 1 19 568 2 19 56... |
| 1 | [98.0, 686.0] | 8 | 148.75 | 0 1 0 117 95 1 117 96 2 117 97 3 ... |
| 2 | [600.0, 689.0] | 32 | 172.16 | 0 1 0 131 599 1 131 600 2 131... |
| 3 | [557.0, 459.0] | 4 | 162.75 | 0 1 0 344 557 1 344 558 2 345 55... |
| 4 | [319.0, 366.0] | 148 | 146.46 | 0 1 0 428 317 1 428 318 2 ... |
| 5 | [109.0, 335.0] | 18 | 163.67 | 0 1 0 466 109 1 466 110 2 467... |
| 6 | [108.0, 289.0] | 104 | 149.49 | 0 1 0 508 106 1 508 107 2 ... |
| 7 | [403.0, 256.0] | 94 | 141.86 | 0 1 0 538 403 1 538 404 2 539... |
| 9 | [335.0, 211.0] | 122 | 139.51 | 0 1 0 586 333 1 586 334 2 ... |
| 10 | [66.0, 195.0] | 30 | 147.53 | 0 1 0 607 63 1 607 64 2 607 6... |
| 11 | [94.0, 194.0] | 8 | 135.12 | 0 1 0 609 93 1 609 94 2 609 95 3 ... |
| 12 | [95.0, 185.0] | 8 | 162.62 | 0 1 0 617 95 1 617 96 2 618 96 3 ... |
| 13 | [335.0, 180.0] | 52 | 164.65 | 0 1 0 617 335 1 617 336 2 618... |
| 14 | [314.0, 183.0] | 16 | 157.19 | 0 1 0 619 313 1 619 314 2 619... |
| 15 | [131.0, 177.0] | 8 | 161.62 | 0 1 0 625 131 1 625 132 2 626 13... |
| 16 | [202.0, 164.0] | 120 | 152.17 | 0 1 0 631 201 1 631 202 2 ... |
| 18 | [329.0, 165.0] | 20 | 161.55 | 0 1 0 635 329 1 635 330 2 636... |
| 20 | [384.0, 163.0] | 16 | 145.25 | 0 1 0 639 385 1 639 386 2 640... |
| 21 | [356.0, 153.0] | 144 | 175.81 | 0 1 0 641 352 1 641 353 2 ... |
| 22 | [339.0, 156.0] | 60 | 151.25 | 0 1 0 643 338 1 643 339 2 643... |
| 23 | [332.0, 158.0] | 6 | 163.33 | 0 1 0 645 332 1 645 333 2 645 33... |
| 24 | [268.0, 144.0] | 200 | 131.54 | 0 1 0 649 267 1 649 268 2 ... |
| 25 | [248.0, 138.0] | 94 | 156.32 | 0 1 0 659 248 1 659 249 2 659... |
| 26 | [300.0, 103.0] | 32 | 182.72 | 0 1 0 697 299 1 697 300 2 697... |
| 27 | [218.0, 95.0] | 244 | 139.88 | 0 1 0 699 221 1 699 222 2 ... |
| 28 | [330.0, 94.0] | 60 | 175.68 | 0 1 0 703 329 1 703 330 2 703... |
| 29 | [241.0, 95.0] | 64 | 181.70 | 0 1 0 705 237 1 705 238 2 705... |
| 30 | [327.0, 76.0] | 12 | 153.42 | 0 1 0 725 327 1 725 328 2 726... |
| 31 | [357.0, 71.0] | 14 | 148.00 | 0 1 0 731 356 1 731 357 2 731... |
| 32 | [339.0, 60.0] | 144 | 130.16 | 0 1 0 735 337 1 735 338 2 ... |
| 33 | [274.0, 62.0] | 64 | 172.64 | 0 1 0 737 273 1 737 274 2 737... |
| 34 | [235.0, 52.0] | 168 | 141.92 | 0 1 0 745 230 1 745 231 2 ... |
| 35 | [207.0, 56.0] | 4 | 132.00 | 0 1 0 747 207 1 747 208 2 748 20... |
| 36 | [297.0, 44.0] | 176 | 121.63 | 0 1 0 749 296 1 749 297 2 ... |

| | Centers in (x,y) | Area | Average intensity | the ith defect in (i,j) |
|----|------------------|------|-------------------|-------------------------------------|
| 2 | [523.0, 768.0] | 32 | 166.22 | 0 1 0 105 522 1 105 523 2 105... |
| 4 | [527.0, 752.0] | 44 | 175.73 | 0 1 0 121 526 1 121 527 2 121... |
| 5 | [490.0, 743.0] | 4 | 156.25 | 0 1 0 133 490 1 133 491 2 134 49... |
| 8 | [452.0, 687.0] | 32 | 172.91 | 0 1 0 185 454 1 185 455 2 185... |
| 9 | [465.0, 677.0] | 44 | 153.84 | 0 1 0 197 462 1 197 463 2 197... |
| 10 | [778.0, 649.0] | 44 | 177.16 | 0 1 0 225 776 1 225 777 2 225... |
| 11 | [322.0, 646.0] | 148 | 140.03 | 0 1 0 227 315 1 227 316 2 ... |
| 14 | [493.0, 634.0] | 14 | 151.43 | 0 1 0 241 492 1 241 493 2 241... |
| 15 | [762.0, 632.0] | 10 | 149.30 | 0 1 0 243 762 1 243 763 2 244 76... |
| 19 | [487.0, 626.0] | 92 | 169.45 | 0 1 0 247 490 1 247 491 2 247... |
| 20 | [459.0, 623.0] | 99 | 145.65 | 0 1 0 249 458 1 249 459 2 249... |
| 21 | [332.0, 624.0] | 34 | 178.76 | 0 1 0 251 334 1 251 335 2 251... |
| 22 | [410.0, 621.0] | 36 | 173.39 | 0 1 0 253 408 1 253 409 2 253... |
| 23 | [464.0, 605.0] | 92 | 126.26 | 0 1 0 267 461 1 267 462 2 267... |
| 25 | [447.0, 603.0] | 6 | 175.00 | 0 1 0 273 447 1 273 448 2 273 44... |
| 26 | [284.0, 600.0] | 10 | 158.90 | 0 1 0 275 284 1 275 285 2 276 28... |
| 28 | [339.0, 594.0] | 78 | 144.60 | 0 1 0 279 339 1 279 340 2 279... |
| 30 | [92.0, 589.0] | 70 | 156.86 | 0 1 0 285 89 1 285 90 2 285 9... |
| 31 | [179.0, 591.0] | 6 | 165.00 | 0 1 0 285 179 1 285 180 2 285 18... |
| 32 | [301.0, 588.0] | 126 | 126.57 | 0 1 0 285 297 1 285 298 2 ... |
| 33 | [368.0, 588.0] | 132 | 131.43 | 0 1 0 285 364 1 285 365 2 ... |
| 34 | [150.0, 585.0] | 36 | 168.17 | 0 1 0 289 150 1 289 151 2 289... |
| 35 | [276.0, 585.0] | 60 | 150.93 | 0 1 0 289 272 1 289 273 2 289... |
| 36 | [378.0, 580.0] | 8 | 165.50 | 0 1 0 295 378 1 295 379 2 296 37... |
| 37 | [131.0, 579.0] | 8 | 146.12 | 0 1 0 297 130 1 297 131 2 297 13... |
| 38 | [149.0, 576.0] | 60 | 141.78 | 0 1 0 297 147 1 297 148 2 297... |
| 39 | [287.0, 578.0] | 28 | 170.04 | 0 1 0 297 283 1 297 284 2 297... |
| 40 | [430.0, 577.0] | 22 | 148.50 | 0 1 0 297 430 1 297 431 2 297... |
| 41 | [407.0, 574.0] | 90 | 124.30 | 0 1 0 299 404 1 299 405 2 299... |
| 42 | [375.0, 573.0] | 58 | 167.50 | 0 1 0 301 373 1 301 374 2 301... |
| 44 | [502.0, 567.0] | 4 | 136.25 | 0 1 0 309 502 1 309 503 2 310 50... |
| 45 | [452.0, 562.0] | 24 | 169.79 | 0 1 0 313 450 1 313 451 2 313... |
| 49 | [437.0, 528.0] | 76 | 141.55 | 0 1 0 345 433 1 345 434 2 345... |
| 53 | [821.0, 511.0] | 12 | 166.08 | 0 1 0 365 819 1 365 820 2 365... |
| 56 | [132.0, 479.0] | 4 | 152.25 | 0 1 0 397 132 1 397 133 2 398 13... |

3D reconstruction script:

```
def list_array(Column):  
  
    Column_list = np.array([val for sublist in Column for val in sublist])  
    Column_list = Column_list.reshape((Column_list.shape[0],Column_list.size/Column_list.shape[0]))  
  
    array = pd.DataFrame(Column_list)  
  
    return array
```

```
def projection_x(Column):  
  
    feat = list_array(Column)  
  
    x_a = np.cos(np.radians(feat[0]))*np.maximum(feat[1],feat[2]) #a,b  
    x_b = np.cos(np.radians(90-np.abs(feat[0])))*np.minimum(feat[1],feat[2])  
  
    x = np.maximum(x_a,x_b)  
  
    return x
```

```
def assignement(features_xy,centroids_xy,features_xz,centroids_xz,coef=1): #if xy.size > xz.size  
  
    '''xz plan'''  
    x_z = projection_x(features_xz)#projection on x axis of xz plan  
    xz = list_array(centroids_xz)#centers of defects  
    s_dev_xz = list_array(features_xz)#(44,3)array of ellipse features  
  
    '''xy plan'''  
    x_y = projection_x(features_xy)  
    xy = list_array(centroids_xy)  
    s_dev_xy = list_array(features_xy)  
  
    '''matching indexes'''  
    coef = coef  
    index_xyz = []  
  
    if xy[0].size < xz[0].size :  
        xy,xz = xz,xy  
  
    for i,x_xy in enumerate(xy[0].values):  
        for k,x_xz in enumerate(xz[0].values):  
            if ((x_xy >= (x_xz - coef*x_z[k])) and (x_xy <= (x_xz + coef*x_z[k]))):  
                index_xyz.append([i,k])  
    print 'There',len(index_xyz),'matching defects'  
  
    index_xyz = np.array(index_xyz)  
  
    '''matching coordinates'''  
    x = np.array(xy[0])  
    x = x[index_xyz[:,0]]  
    y = np.array(xy[1])  
    y = y[index_xyz[:,0]]  
    z = np.array(xz[1])  
    z = z[index_xyz[:,1]]  
    '''matching stand deviations'''  
    sig_x = np.array(s_dev_xy[1])  
    sig_x = sig_x[index_xyz[:,0]]  
    sig_y = np.array(s_dev_xy[2])  
    sig_y = sig_y[index_xyz[:,0]]  
    sig_z = np.array(s_dev_xy[1])  
    sig_z = sig_z[index_xyz[:,1]]  
    '''matching angles'''  
    angle_xy = np.array(s_dev_xy[0])  
    angle_xy = angle_xy[index_xyz[:,0]]  
    angle_xz = np.array(s_dev_xy[0])  
    angle_xz = angle_xz [index_xyz[:,1]]  
  
    return np.array([np.array([x,y,z]),np.array([sig_x,sig_y,sig_z]),np.array([angle_xy,angle_xz])])
```

3D plots with 2D projection:

```
def plot_3d_2d(x,y,z):

    #z= img_xz.shape[0]-z
    fig = plt.figure(figsize=(10,8))
    ax = fig.gca(projection='3d')

    ax.plot(x, y, z, 'bo', label='3D Centroids',alpha =0.5)
    ax.set_title('3D reconstruction of defects position')

    ax.plot(x, z, 'r+', zdir='y', zs=np.max(y)+50, label='2D projection on x,z',alpha =0.5)
    ax.plot(x, y, 'g+', zdir='z', zs=np.min(z)-50, label='2D projection on x,y',alpha =0.5)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    ax.set_xlim([np.min(x)-50, np.max(x)+50])
    ax.set_ylim([np.min(y)-50, np.max(y)+50])
    ax.set_zlim([np.min(z)-50, np.max(z)+50])
    ax.legend()
    plt.savefig('fig')

    plt.show()
```

3D ellipsoids:

```
def one_ellipsoid_3D(ax,Ellipsoid_features,i,scale =40):

    x0,y0,z0 = Ellipsoid_features[0]
    sig_x,sig_y,sig_z = Ellipsoid_features[1]

    coefs = (sig_x[i]**2, sig_y[i]**2, sig_z[i]**2) # Coefficients in a0/c x**2 + a1/c y**2 + a2/c z**2 = 1
    # Radii corresponding to the coefficients:
    rx, ry, rz = scale/np.sqrt(coefs)

    # Set of all spherical angles:
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)

    # Cartesian coordinates that correspond to the spherical angles:
    # (this is the equation of an ellipsoid):
    x = rx * np.outer(np.cos(u), np.sin(v))+x0[i]
    y = ry * np.outer(np.sin(u), np.sin(v))+y0[i]
    z = rz * np.outer(np.ones_like(u), np.cos(v))+z0[i]

    # Plot:
    ax.plot_surface(x, y, z,rstride=1, cstride=1,cmap='viridis', edgecolor='none',alpha=0.4)
```

```
def all_ellipsoids_3D(Ellipsoid_features,scale):

    number_of_defects = Ellipsoid_features[0].shape[1]

    fig = plt.figure(figsize=(8,8)) # Square figure
    ax = fig.add_subplot(111, projection='3d')

    for i in range(number_of_defects):
        one_ellipsoid_3D(ax,Ellipsoid_features,i,scale)

    x,y,z = Ellipsoid_features[0]

    ax.set_xlim([np.min(x)-50, np.max(x)+50])
    ax.set_ylim([np.min(y)-50, np.max(y)+50])
    ax.set_zlim([np.min(z)-50, np.max(z)+50])
    ax.set_title('3D distribution of estimated defects ellipsoids')

    plt.savefig('fig')
    plt.show()
```

GUI cropping script:

```

import sys

from PyQt5.QtCore import *

from PyQt5.QtGui import *

from PyQt5.QtWidgets import *

class Window(QWidget):

    def __init__(self):

        QWidget.__init__(self)

        self.image = QPixmap("image.png")

        self.resize(self.image.width(), self.image.height())

        self.largest_rect = QRect(10, 10, self.image.width(), self.image.height())

        self.clip_rect = QRect(10, 10, self.image.width(), self.image.height())

        self.dragging = None

        self.drag_offset = QPoint()

        self.handle_offsets = (

            QPoint(8, 8), QPoint(-1, 8), QPoint(8, -1), QPoint(-1, -1))

        self.show()

        self.resize(800, 600)

    def paintEvent(self, event):

        painter = QPainter()

        painter.begin(self)

        painter.drawPixmap(QRect(10, 10, self.image.width(), self.image.height()),

self.image)

        painter.drawRect(self.clip_rect)

        for i in range(4):

            painter.drawRect(self.corner(i))

        painter.setClipRect(self.clip_rect)

```



```

    painter.end()

def corner(self, number):
    if number == 0:
        return QRect(self.clip_rect.topLeft() - self.handle_offsets[0], QSize(8, 8))
    elif number == 1:
        return QRect(self.clip_rect.topRight() - self.handle_offsets[1], QSize(8, 8))
    elif number == 2:
        return QRect(self.clip_rect.bottomLeft() - self.handle_offsets[2], QSize(8, 8))
    elif number == 3:
        return QRect(self.clip_rect.bottomRight() - self.handle_offsets[3], QSize(8, 8))

def mousePressEvent(self, event):
    for i in range(4):
        rect = self.corner(i)
        if rect.contains(event.pos()):
            self.dragging = i
            self.drag_offset = rect.topLeft() - event.pos()
            break
    else:
        self.dragging = None

def mouseMoveEvent(self, event):
    if self.dragging is None:
        return
    left = self.largest_rect.left()
    right = self.largest_rect.right()

```

```

top = self.largest_rect.top()

bottom = self.largest_rect.bottom()

point = event.pos() + self.drag_offset + self.handle_offsets[self.dragging]

point.setX(max(left, min(point.x(), right)))

point.setY(max(top, min(point.y(), bottom)))

if self.dragging == 0:

    self.clip_rect.setTopLeft(point)

elif self.dragging == 1:

    self.clip_rect.setTopRight(point)

elif self.dragging == 2:

    self.clip_rect.setBottomLeft(point)

elif self.dragging == 3:

    self.clip_rect.setBottomRight(point)

self.update()

def mouseReleaseEvent(self, event):

    self.dragging = None

    currentQRect = self.currentQRubberBand.geometry()

    cropQPixmap = self.update().copy(currentQRect)

    cropQPixmap.save('output.png')

if __name__ == "__main__":

    app = QApplication(sys.argv)

    window = Window()

    window.show()

    sys.exit(app.exec_())

```