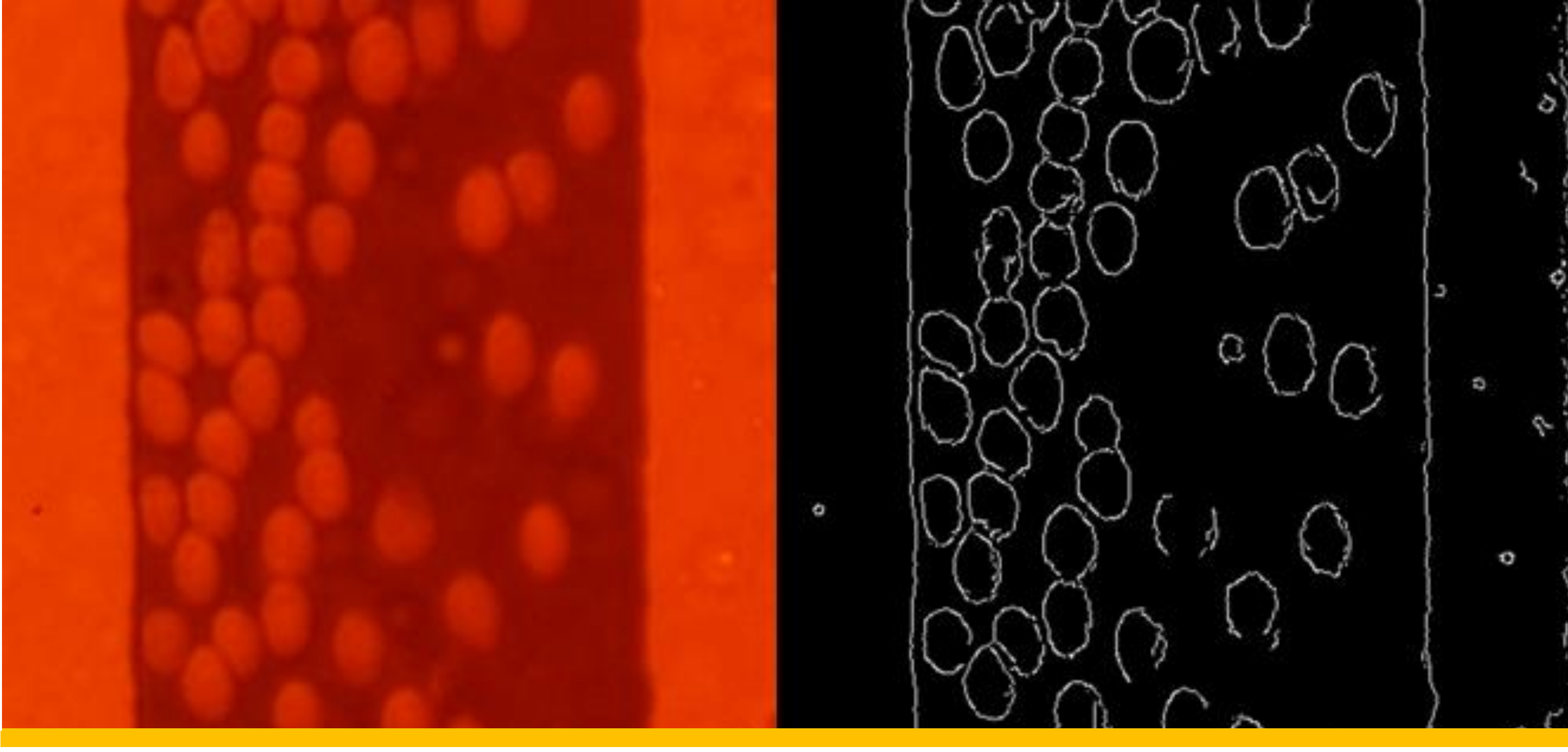


Industry Project 2

Measuring real-world object dimensions in images with OpenCV

Degree:	B.Sc. Industrial Engineering
Author:	Dominique Peytrignet
Supervisor:	Prof. Dr. Cédric Bessire
Expert:	Matt Stark
Date:	03.04.2023

Bern University of Applied Sciences

Technik und Informatik
Wirtschaftsingenieurswesen

Abstract

The ability to let a computer understand images and videos to fulfil a specific task, is called computer vision. Computer vision applications are used in many industries such as in automotive to detect and classify road signs, in manufacturing to visually control the quality of the produced product.

A well known problem in computer vision tasks is to accurately determine the real-world dimensions of the object of interest in the image. Solving this problem can be very helpful in the field of medical imaging, where a diagnosis for a patient can be drawn based on the shape and volume of their Red Blood Cells (RBC) . Current methods to determine the volume the RBC require high investment costs and lots of time. A image based processing would improve this process and safe time.

Therefore in this project, the goal was to create a script which is capable of measuring the x- and y-dimensions of objects in an image, measure the accuracy of it and identify what factors may have a negative influence on the accuracy. A further objective was, looking forward to the Bachelor Thesis, to evaluate methods on how the volume and the shape of RBCs can be estimated, based on two-dimensional images.

To achieve these goals, first a script has been developed to measure a selection of objects (Jenga piece, coins etc.) in an image, mainly using the Python library OpenCV. To detect the objects, the image first was put into grayscale colours and blurred, after that their edges and their contours were detected. After the contours were detected a rectangle was fitted on to them. The script takes the left most object in the image as a reference, to know what the ratio between the pixels and the desired metric is. This ratio is then applied to the other objects to know their dimensions. The results show that the measured values can differ up to 10% from the real world values. Higher deviations can be caused due to parallax error, depth differences or bad lightning conditions.

As a second step, images were provided where RBCs are flowing through a channel. The image was taken from a video sequence. The goal was to do a first measurement and determine their width and height, by applying the same methodology as with the selection of objects. As a reference length the width of the channel was taken. Because the real world values of the RBCs, in the image, couldn't be determined, the measured value couldn't be compared. But based on an healthy human the results are reasonable and show that it is possible to detect single RBC and measure their width and height.

In a third step, a literature research was conducted to find out methodologies on how the volume of an object can be estimated, based on a single or multiple 2D images. Further some method were presented on how the object be visualized, containing the 3rd dimension. The result showed, that the most promising methodology, for this task, is to apply the same approach as presented by Graikos et. al. where they use a depth estimation- and segmentation-network to estimate the volume of food portions.

As a future work, namely the Bachelor Thesis, it has to be evaluated whether this methodology can be applied to the RBC images. A difficulty would be trying to track the RBC in the video sequence, as it would be required to build the depth estimation network. As an alternative, further methodologies could be evaluated which have other solutions to estimate the volume of RBCs.

Table of contents

j

1 Introduction

1.1 Initial Situation and relevance

Measuring the lengths of objects in images is a crucial task in a variety of applications, including manufacturing, quality control, medical imaging, and scientific research. Accurate and efficient measurement of lengths in images can provide valuable information for making decisions, improving processes, and gaining insights into different situations.

However, the task of measuring lengths and dimensions in images is not trivial, and it presents several challenges. One of the primary challenges is the need for accurate calibration of the image capture device to ensure that measurements are consistent and repeatable. Additionally, objects in images may appear distorted due to perspective, lens distortion, and other factors, making it difficult to accurately measure their lengths.

Having a program which is capable to automatically detect and measure the dimensions of various items, based on an image / live-video, could improve automatization of various processes in industries.

1.2 Project objectives

The goal for this Industry Project 2 is to create a python script which is capable to measure the length and width of various objects shown on an image and test the accuracy of the results with multiple measurements.

To achieve this goal, following questions have been formulated to be examined:

- What methods are available to measure the x- and y-dimensions of an object in an image and how do they work?
- Which factors have a negative influence on the result?
- What are additional shapes that can be measured besides rectangles only?
- Also looking forward for the Bachelor Thesis: how can the volume of an object be extracted from an image and visualized?

1.3 Intended / expected results, deliverables and recommendations

The result for the Industry Project 2 should be a Python script which is able to detect and measure the dimensions of various objects based on a 2D image.

Also some key factors should be presented which have a negative impact on the measurement results. In addition, different image processing methods should be evaluated on how the dimensions can be measured in images.

2 Theory

2.1 OpenCV

OpenCV, or Open Source Computer Vision Library, is a popular open-source computer vision and machine learning software library. It was first developed by Intel in the late 1990s, and it is now maintained by the OpenCV community.

OpenCV is designed to help developers create applications that can analyse and understand digital images and videos. It provides a comprehensive set of tools and algorithms for image processing, object detection and recognition, facial recognition, machine learning, and more. Further there are also pre-trained models which can be used for these tasks.

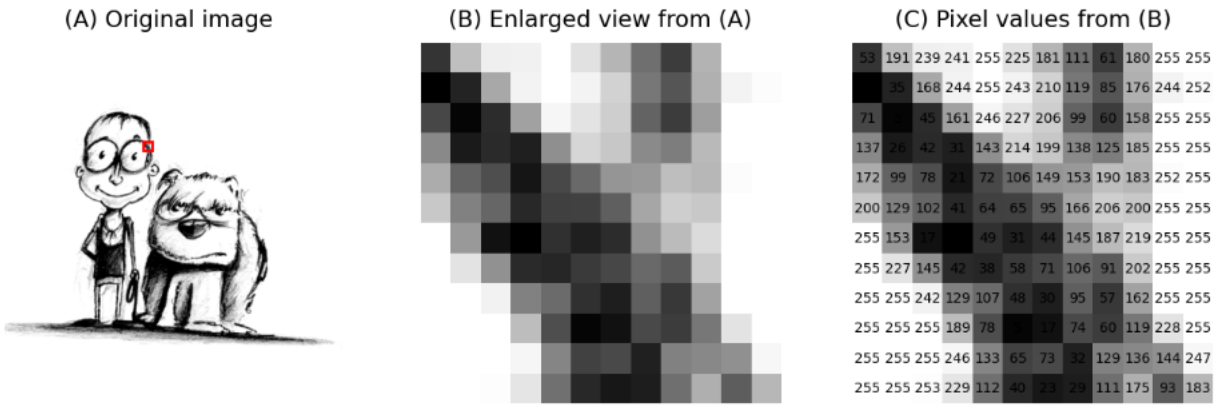
The library is written in C++ and can be used with C++, Python, Java, and other programming languages. It is cross-platform and can be used on various operating systems, including Windows, Linux, macOS, Android, and iOS. [1]

2.2 Image Theory

To understand the basics of image processing, one first has to understand what an image actually is and what components its consists of.

An image is a two-dimensional representation of the three-dimensional real world captured by a camera or other sensing technologies. Images can also be created using graphical software and don't necessarily have to represent the real world. For this project, only images which are taken from a camera are considered.

Images are composed of pixels which are basically small square elements stored in an array of numbers, normally ranging from 0 to 255 (1 Byte) in a grey-scale image, which represent the brightness of the pixel, see figure below. Each pixel can be identified by its row and column position. [2]



53	191	239	241	255	225	181	111	61	180	255	255
	35	168	244	255	243	210	119	85	176	244	252
71	5	45	161	246	227	206	99	60	158	255	255
137	26	42	31	143	214	199	138	125	185	255	255
172	99	78	21	72	106	149	153	190	183	252	255
200	129	102	41	64	65	95	166	206	200	255	255
255	153	17		49	31	44	145	187	219	255	255
255	227	145	42	38	58	71	106	91	202	255	255
255	255	242	129	107	48	30	95	57	162	255	255
255	255	255	189	78	5	17	74	60	119	228	255
255	255	255	246	133	65	73	32	129	136	144	247
255	255	253	229	112	40	23	29	111	175	93	183

When wanting to encode coloured images, the informations are usually represented by a multidimensional matrix, consisting of three channels: A red, green and a blue channel (RGB) see figure below.

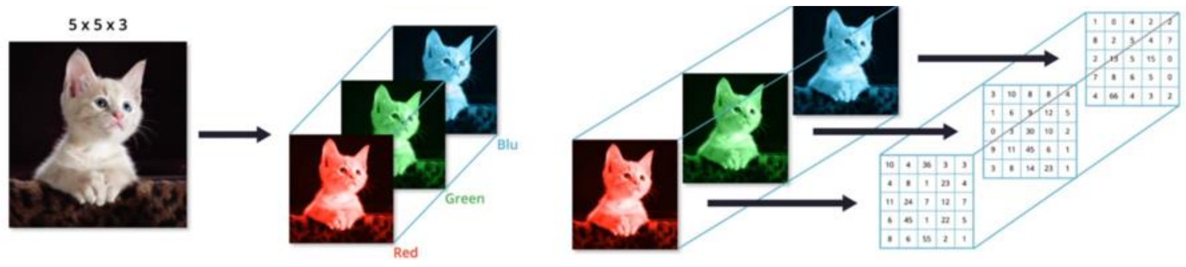


Figure 1: RGB Channels, Source: [3]

Each channel consists the informations for the brightness for either red green or blue and when combining all three channels, the multidimensional matrix is the digitized representation of the original image. [3]

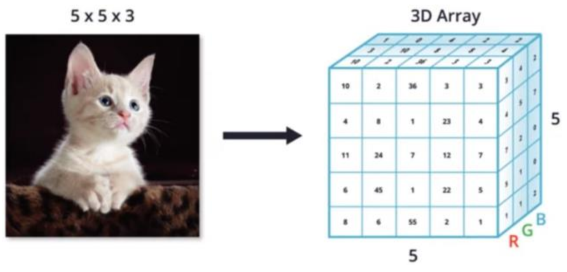


Figure 2: RGB Channels combined as a multi-dimensional matrix, Source: [3]

Image can be stored in various formats such as JPEG, PNG, BMP or TIFF. Each of these formats have advantages and disadvantages and the choice of the format depends on the application and the intended usage of the image.

2.3 Image Processing

Image processing involves the use of mathematical algorithms and techniques to enhance, analyse and manipulate digital images, i.e. manipulating the arrays which represent the pixels. Some of the important techniques used in image processing are filtering and object detection, which are of great importance for this project and are shortly described in this section.

2.3.1 Gaussian Blur

The Gaussian blur is a type of image filter that is used in image processing to smooth an image and reducing the noise in it. The filter, also called kernel, is based on the gaussian function, which describes a normal distribution function and is convolving through the image and calculating new values for each of the pixels. [4]

To convolve an image means that the kernel is positioned over each pixel in the image, and the values of the kernel are multiplied with the corresponding pixel values in the image. The products are then summed to produce a single value, which is assigned to the corresponding pixel in the output image. This process is repeated for every pixel in the image, producing a new image that has been transformed by the kernel. The result of the convolution depends on the properties of the kernel, such as its size, shape, and weights. The gaussian blur is useful for pre-processing the image and to later detect objects for example. [5]

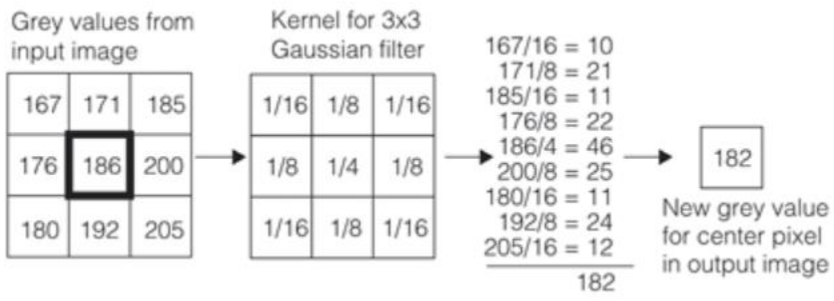


Figure 3: Gaussian Kernel, Source : [4]

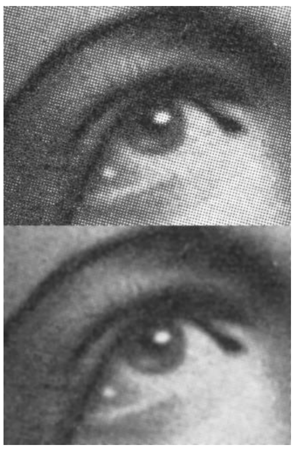


Figure 4: Effect of gaussian blur, Source: Wikipedia

2.3.2 Edge detection

Detecting edges in an image is an important task to detect objects and ultimately be able to measure their dimensions.

Edges in images are characterized by changes in pixel intensity and there are multiple algorithms to detect them, two of the most popular are " Sobel edge detection" and "Canny edge detection", which both have their advantages and are available on OpenCV. [6]

The canny edge detection algorithm is a multi-step process involving some important concepts:

In a first step the gaussian blur, described previously, is applied on the image. This is done to reduce the noise in the image and detect the edge easier.

Next, the algorithm calculates the gradient of the image using the Sobel operator. The Sobel operator is a kernel that is used to calculate the gradient of an image by convolving it with the image. The gradient represents the rate of change of the intensity of the image at each pixel.

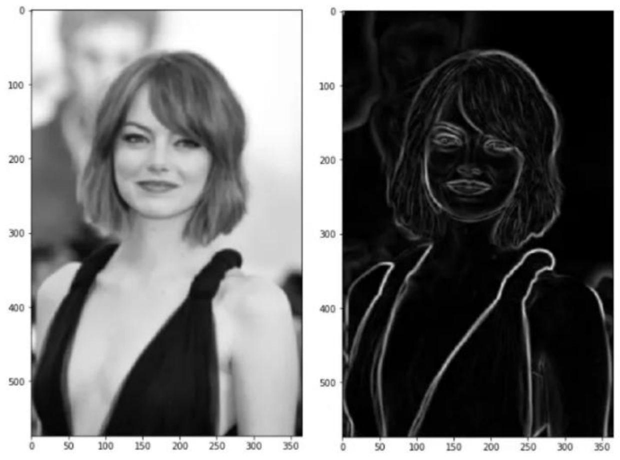


Figure 5: Left: Blurred imag; Right: Gradient intensity, Source: TowardsDataScience.com

The third step is non-maximum suppression. This step is used to thin out the edges detected in the previous step. It works by iterating over each pixel in the image and checking if it is a local maximum in the direction of the gradient. If it is not a local maximum, then it is set to zero. This step ensures that only the most significant edges are retained.

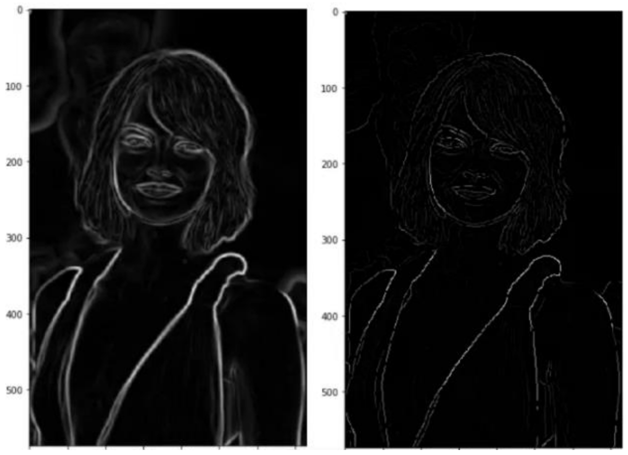


Figure 6: Effect of non-maximum suppression, Source: TowardsDataScience.com

The fourth step in the Canny edge detection algorithm is double thresholding. This step is used to separate the strong edges from the weak edges. The algorithm defines two thresholds: in OpenCV specifically maxVal and minVal. Pixels with a gradient intensity above the high threshold are considered strong edges, while pixels with a gradient intensity below the low threshold are considered non-edges. Pixels with a gradient magnitude between the high and low thresholds are considered weak edges.

The last step is edge tracking by hysteresis. This step is used to link weak edges with strong edges. The algorithm does this by iterating over each weak edge and checking if it is connected to a strong edge. If it is connected to a strong edge, then it is considered a part of an edge. If it is not connected to a strong edge, then it is discarded.
[7] [8]

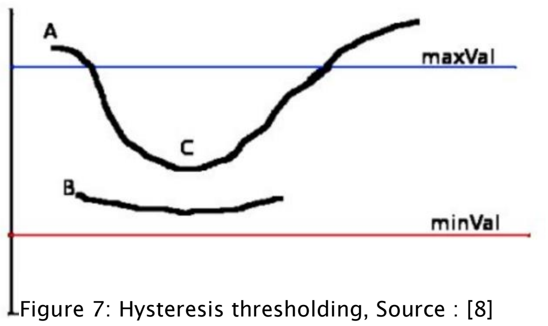


Figure 7: Hysteresis thresholding, Source : [8]

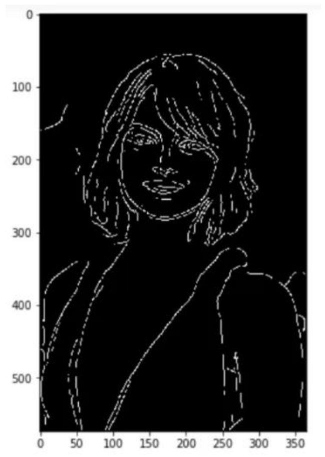


Figure 8: Final result of edge detection,
Source: TowardsDataScience.com

3 Research Design

For the Industry Project 2 a literature research is conducted about the underlying technologies and methods which are used to be able to detect objects on images and measure their dimensions. Also a research will be conducted to find out what options exist to detect various other shapes besides rectangles only, e.g. circles, polygons etc.

For the measurements, a variations of objects will be selected and their dimensions measured with a Python script, i.e. their height and width using the OpenCV library. The results will then be compared with the real-world dimensions of the objects and based on this, various factors should be identified which negatively influence the accuracy.

Further with a view to the bachelor thesis, a further literature research is made on how the information about the 3rd dimension can be extracted from an 2D image, i.e. how the volume of an object can be calculated. This will later lay the foundation for the bachelor thesis.

3.1 Processing methodology

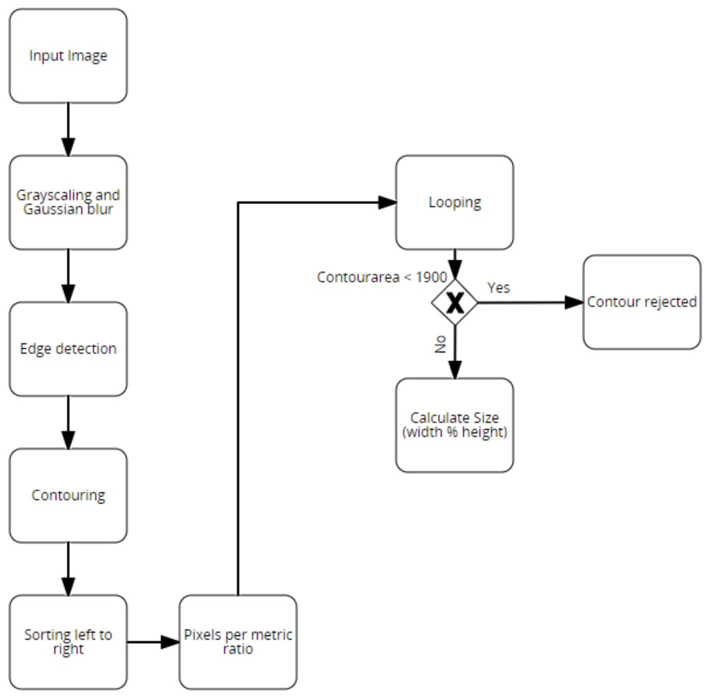


Figure 9: Methodology flowchart, Own figure

In the diagram above, the different steps to process the image and ultimately measure the size of the object is shown.

First the colour of the input image is changed to grayscale, so that the other processing steps are easier to do. Also a gaussian blur is carried out to remove noise in the image.

As next steps, the canny edge detection is performed to detect the edges, which are basically multiple small pixels. With the help of contouring, the edges can be outlined, so that they are connected and the area of a polygon can be calculated.

The results of this will be a list of multiple contours. To get the contours of the reference object , i.e. the first position of the list, this list has to be sorted in python.

To now the real world values in the image, a calibration has to be done, so the computer knows how large one pixel in the real world is. For this the pixel per metric ratio has to be calculated.
To calculate the pixel per metric ratio, a rectangle box is fitted on the contours. Together with the known real world value of the object and the pixel length the ratio can be calculated as following:

$$Ratio = \frac{Width\ of\ Rectangle\ (Pixels)}{Width\ of\ object\ (Metric)} \qquad (1)$$

To calculate now the size of another object in a desired metric with a given ratio, the width or the height of the rectangle has to be divided by the Ratio.

The script repeats these calculations in a loop for all the contours. If the contour areas are too small, they are being ignored.

This process is then being repeated with a 90° rotated image, so that another object is used as a reference.

3.2 Object selection

For the measurements a variation of different objects have been selected, as shown in the figure below. In this specific example the reference object would the 5CHF coin, which is at the left most position.

These objects have been selected because they are fast and accurate to measure in the real world and show enough contrast between the background. Also some different colours have been chosen to see if it has an impact on the result.

The requirement for these measurements are that all the objects are placed on the same plane, because if they are not the computer can differentiate the depth difference of the objects without using other technologies like sensors.

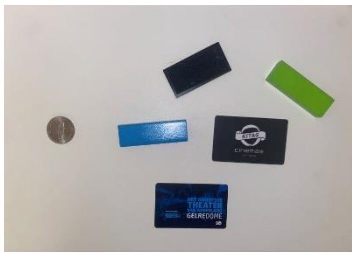


Figure 10: Image with objects, Own figure

Object	Real-world dimensions
5CHF Coin	31.34mm x 31.34mm
2CHF Coin	27.30mm x 27.30mm
Jenga Piece Green	24.67mm x 74.30mm
Jenga Piece Blue	24.25mm x 74.87mm
Black sticker badge	34.80mm x 69.78mm
Black Card (KITAG)	53.88mm x 85.46mm
Blue Card	53.88mm x 85.46mm

Table 1: Selection of objects

4 Results of measured objects

3 tests have been carried out with 3 different images, as shown below. These images have been taken with the camera lens of an iPhone 11. For the tests, the same parameters have been chosen for the preprocessing of the images, which include blurring and detecting the edges.

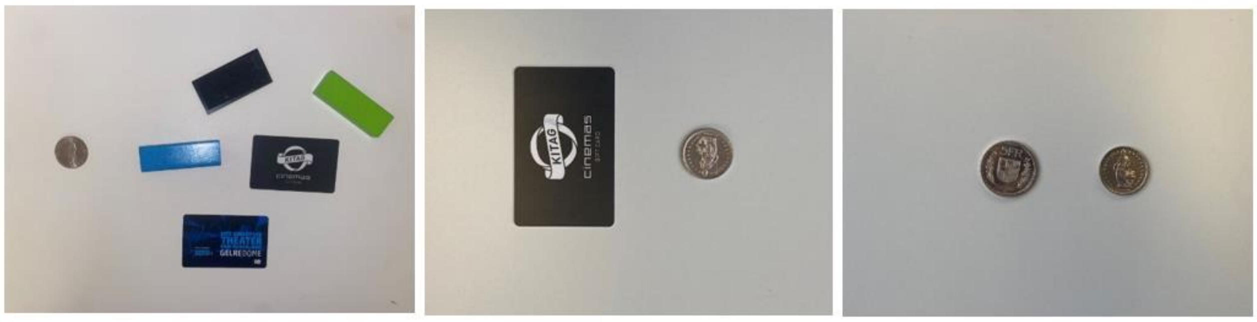


Figure 11: From left to right: Test Images 1, 2, 3, own figure

4.1 Test 1

For the first test, the image will be rotated 90° after every measurement cycle, so that once the coin is the reference object and then the blue card, the green jenga and at last the black sticker. This methodology has been chosen to see, if the length of the reference object an influence on the results has.

4.1.1 Test 1 original image

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_1.jpg	5CHF Coin	31.34 x 31.34	31.34 (Reference)	30.87 / 1.5%
	Blue Jenga	24.25 x 74.87	24.78 / 2.1%	76.03 / 1.5%
	Blue Card	53.88 x 85.46	49.63 / 8.6%	80.50 / 6.2%
	Black Card	53.88 x 85.46	51.13 / 5.4%	81.32 / 5.1%
	Black Sticker	34.80 x 69.78	35.78 / 2.7%	69.66 /0.2%
	Green Jenga	24.67 x 74.30	29.35 / 15.9%	75.45 / 1.5%

Table 2: Measurements Test_1.jpg

4.1.2 Test 1 90˚ rotated

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_1_90.jpg	5CHF Coin	31.34 x 31.34	33.51 / 6.5%	34.02 / 7.9%
	Blue Jenga	24.25 x 74.87	26.91 / 9.9%	82.54 / 9.3%
	Blue Card	53.88 x 85.46	53.88 (Reference)	87.13 / 1.9%
	Black Card	53.88 x 85.46	55.50 / 2.9%	88.29 / 3.2%
	Black Sticker	34.80 x 69.78	38.85 / 10.4%	75.62 / 7.7%
	Green Jenga	24.67 x 74.30	31.86 / 22.6%	81.91 / 9.3%

Table 3: Measurements Test_1_90.jpg

4.1.3 Test 1 180˚ rotated

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_1_180.jpg	5CHF Coin	31.34 x 31.34	30.86 / 1.6%	30.39 / 3.1%
	Blue Jenga	24.25 x 74.87	24.41 / 0.7%	74.87 / 0%
	Blue Card	53.88 x 85.46	48.87 / 10.3%	79.03 / 8.1%
	Black Card	53.88 x 85.46	50.34 / 7%	80.08 / 6.7%
	Black Sticker	34.80 x 69.78	35.24 / 1.2%	68.59 / 1.7%
	Green Jenga	24.67 x 74.30	28.90 / 14.6 %	74.30 (Reference)

Table 4: Measurements Test_1_180.jpg

4.1.4 Test 1 270˚ rotated

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_1_270.jpg	5CHF Coin	31.34 x 31.34	30.92 / 1.4%	31.40 / 0.2%
	Blue Jenga	24.25 x 74.87	24.83 / 2.3%	76.16 / 1.7%
	Blue Card	53.88 x 85.46	49.71 / 8.4%	80.64 / 6%
	Black Card	53.88 x 85.46	51.22 / 5.2%	81.47 / 4.9%
	Black Sticker	34.80 x 69.78	35.85 / 2.9%	69.78 (Reference)
	Green Jenga	24.67 x 74.30	29.40 / 16.1%	75.59 / 1.7%

Table 5: Measurements Test_1_270.jpg

4.2 Test 2

For the second test, the black card and a 2 CHF coin have been chosen to be measured. This time the distance from the reference object is much smaller than the first test. The goal was to see if the distance of an object from the reference object has an influence on the results.

4.2.1 Test 2 original image

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_2.jpg	Black Card	53.88 x 85.46	53.88 (Reference)	85.83 / 0.4%
	2CHF Coin	27.30 x 27.30	28.43 / 4.0%	27.35 / 0.2%

Table 6: Measurements Test_2.jpg

4.2.2 Test 2 180° rotated

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_2_180.jpg	Black Card	53.88 x 85.46	52.53 / 2.6%	82.90 / 3.1%
	2CHF Coin	27.30 x 27.30	27.30 (Reference)	26.24 / 4.0%

Table 7: Measurements Test_2_180.jpg

4.3 Test 3

4.3.1 Test 3 original image

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_3.jpg	5CHF Coin	31.34 x 31.34	31.34 (Reference)	30.00 / 4.5%
	2CHF Coin	27.30 x 27.30	27.02	25.94 / 5.2%

Table 8: Measuremens Test_3.jpg

4.3.2 Test 3 180° rotated

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_3_180.jpg	5CHF Coin	31.34 x 31.34	31.35 / 0%	30.27 / 3.5%
	2CHF Coin	27.30 x 27.30	27.30 (Reference)	25.95 / 5.2%

Table 9: Measurements Test_3_180.jpg

4.4 Interpretation of test results

The results show that the accuracy of the measurements differ up to 10% from the real-world dimensions of the object, when the edge detection was optimal. When the edge detection was not optimal for example for the green Jenga piece or other reasons, see next chapter, the differences were higher. What also can be seen is that the accuracy of Test 2 and 3 better are than Test 1. The assumption from the author for this difference is, that the measured object are much closer to each other and therefore the parallax effect is decreased compared to the image from Test 1.

4.5 Limits and causes of inaccuracy.

4.5.1 Viewing from the side

In the test image 1, the side area of the green Jenga piece can be seen very good. This is due to the angle of the camera lens relative to the Jenga piece, which is creating a parallax error. If the camera viewing field would be in line with the edges of the piece, the error would be minimized.
This error caused the effect that the canny edge detection didn't capture the top edge of the piece rather it took the bottom edge, as seen in the figure below. That is why the width of the green Jenga piece was measured too inaccurate in the measurements.

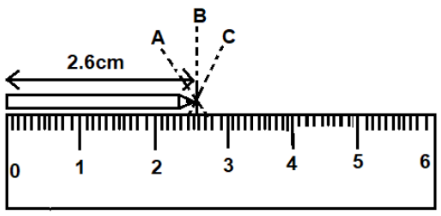


Figure 12: Example of a parallax error,
Source: Vedantu.com

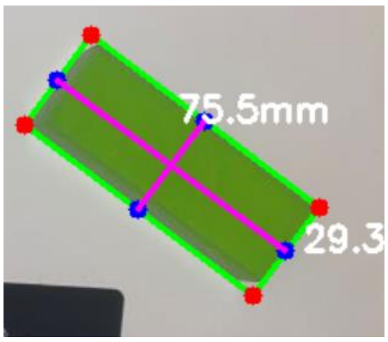


Figure 13: Cutout of measurements,
Own figure

4.5.2 Depth difference

Objects that are nearer to the camera lens appear bigger than they actually are. Even though the chosen objects are laid out on an plane surface, the height difference could negatively influence the measurement results. In the test measurements performed, the two Jenga pieces where the highest objects and therefore nearest to the camera lens, but there were no significant negative influence In the results.

To show the effect better, a photo was captured where the blue card was taken as the reference object and the blue Jenga piece was put vertically up, so it stands higher.

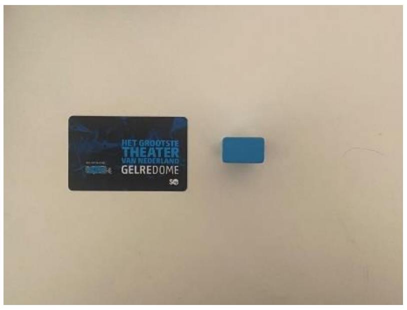


Figure 14: Test_4.jpg Depth Difference, Own figure

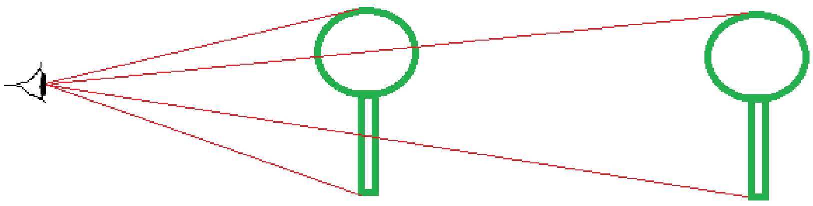


Figure 15: Visualization on the effect of different distances to the camera lens

Image	Object	Real-world dimensions [mm]	Width [mm] / % Difference	Height [mm] / % Difference
Test_4.jpg	Blue Card	53.88 x 85.46	54.01 / 0.2%	85.46 (Reference)
	Blue Jenga	24.25 x 14.45	30.20 / 19.7%	18.83 / 23.3%

Table 10: Measurements Test_4.jpg

The results show significantly percentage difference to the real-world dimensions of the Jenga piece (higher than 10%) .

4.5.3 Shadows

In some images the light condition were not optimal and shadows were visible, like in the figure below from the Test_2.jpg image. This caused that the box was fitted on the edge of the shadow instead of the coin, leading to an inaccurate measurement. Better lightning conditions and optimize preprocessing would minimize this error.

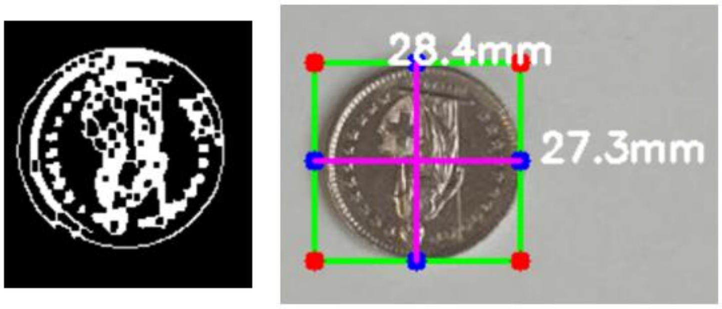


Figure 16: Shadow interpreted as edge, Own figure

4.6 ArUco Markers

In some situations it might be not possible to have a reference object at the left most position or any other objects whose dimensions are known. For this problem, an alternative solution would be, to place an ArUco marker with known width & height on the table, where the other objects are placed and let the program detect the marker and calculate the pixel per metric ratio with its known width or height. (figure below)

An ArUco markers are a type of two-dimensional barcode, which consists of a square or rectangular pattern of black and white cells, which are designed to be easily recognized by the computer vision program. They are often used as reference markers, to track the position and orientation of objects in the camera's field of view. [9]

Other applications may be using augmented reality to anchor virtual objects to real-world locations. By detecting the markers in a live camera feed, the computer can overlay the virtual objects on top of the real world in a way that appears to be anchored to the physical environment.

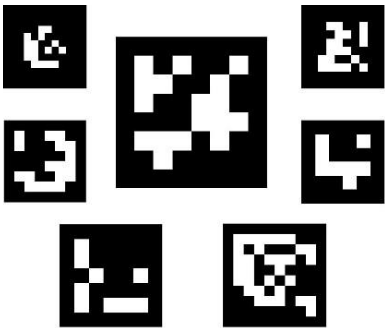


Figure 17: Examples of ArUco Markers, Source: [9]

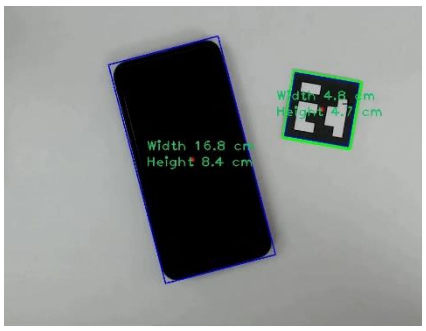


Figure 18: Object measuring with ArUcoMarker, Source: Pysource

5 2D-Measurement of red blood cells

5.1 Red blood cells shape

A red blood cell has a distinctive biconcave disc shape, which means it is flattened in the middle and thicker at the edge. The cell of an average healthy human is about 7-8 micrometers in diameter and 2 micrometers in thickness at the thickest point. The biconcave shape provides a large surface area-to-volume ratio, which allows for efficient gas exchange of oxygen and carbon dioxide between the cell and its surrounding environment. The shape also allows the cell to be flexible, which enables it to travel through small blood vessels and capillaries. The red blood cell is essentially a membrane-bound sac filled with hemoglobin, a protein that binds to oxygen and transports it throughout the body. [10]

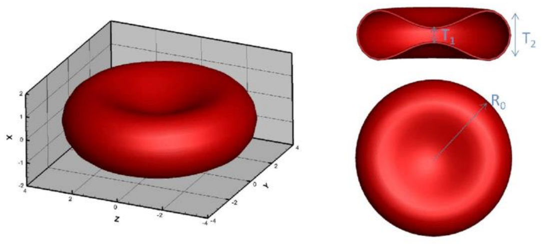


Figure 19: Geometry of red blood cells, Source: [10]

5.2 Processing Methodology for red blood cells images

Microscopy images of red blood cells (RBC) have been provided for this project, which shows cells passing through a channel with a width of 50µm. The frames were taken from a video sequence.

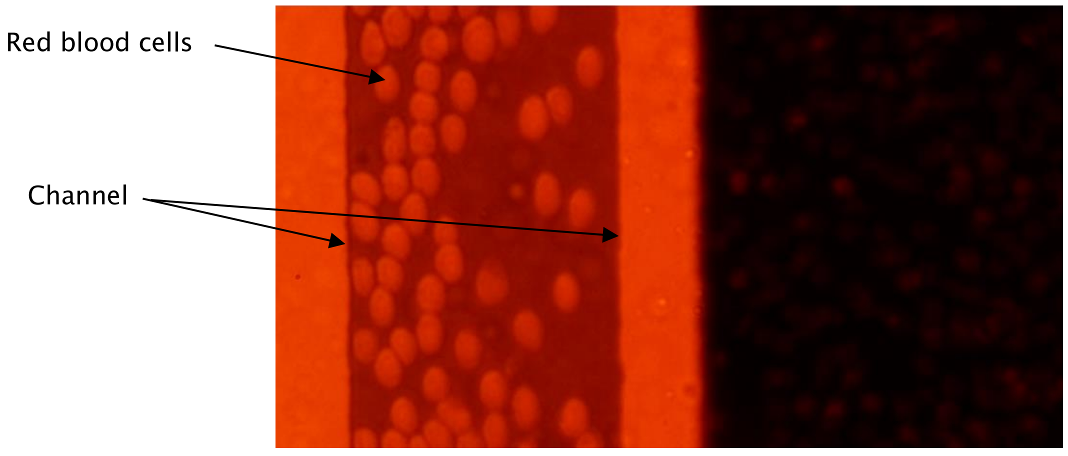


Figure 20: Screenshot from video of red blood cells passing through a channel, Source: BFH

The method used for measuring the objects (Coin, Jenga etc.) should now be applied to measure the dimensions of the RBCs in the image.

Because in this image there is no reference object at the left most position, another reference with known dimensions has to be chosen, to calculate the pixel to metric ratio. The channel with known width of 50µm is well suited for this task.

To measure the pixel length of the channel, i.e. the width of the channel, the computer needs to know where the edged of this channel are located. Because the channel is represented as two straight lines, the probabilistic Hough transform algorithm has been applied, to detect them. The algorithm gives as an output, the two extremes of line, namely the starting point and the end point (x0, y0, x1, y1). [11]

5.3 Probabilistic Hough Transform

The Probabilistic Hough Transform (PHT) is a computer vision algorithm used to detect lines or other geometric shapes in an image. The algorithm is an extension of the standard Hough Transform and was introduced to address some of the limitations of the original algorithm.

In the standard Hough Transform, each point in an image generates a sinusoidal curve in a parameter space that corresponds to the parameters of the line that passes through that point, see figure below. The intersection of curves in this parameter space indicates the parameters of the line in the image. However, this approach can be computationally expensive and is sensitive to noise and missing data. [12]

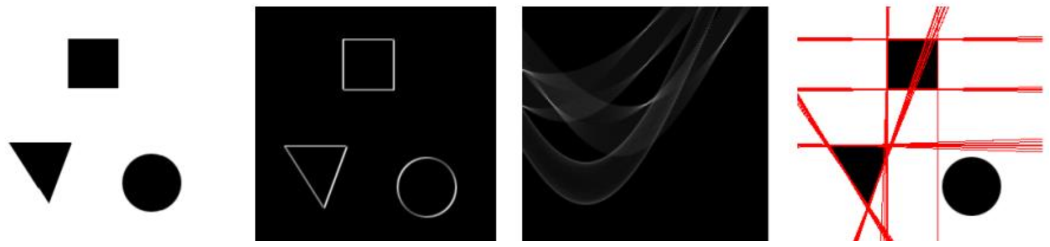


Figure 21: Standard hough transform. From left to right: input image, edged image, parameter space and overlay of detected lines, Source: [12]

The Probabilistic Hough Transform, on the other hand, uses a random subset of points in the image to estimate the parameters of the line. Specifically, it first randomly selects two points from the image and computes the parameters of the line that passes through them. It then checks if other points in the image lie on this line within a certain tolerance. If enough points are found, the algorithm considers this a valid line segment and returns the endpoints of the line. If not, the process is repeated with a new random subset of points.

By only considering a random subset of points in the image, the PHT is less sensitive to noise and can be much faster than the standard Hough Transform. However, it may not be as accurate in detecting lines in images with low contrast or with a high degree of curvature. [13]

5.4 Results

5.4.1 Line detection

With the described probabilistic Hough transform algorithm, it was possible to identify some straight lines in the edged image, red in the figure below. Because the edges of the channel aren't precisely straight, due to abrasions or the roughness of the surface, the detected lines don't cover the whole length of the channel, rather multiple smaller lines are detected.

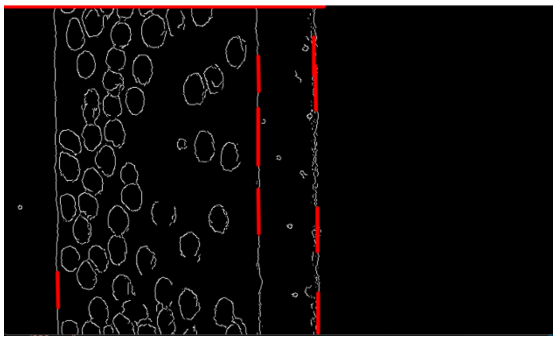


Figure 22: Detected lines on edged RBC image, Own figure

To be able to measure the width of the channel, minimum one line on each edge (left and right) has to be detected. The diameter in pixels can then be calculated by subtracting the x-coordinates of the two lines (See Jupyter Notebook).

5.4.2 Measurements

After the pixel per metric ratio, has been calculated, the same process has been performed as with the objects. The contours of the red blood cells have been identified and rectangles were fitted around them, to measure the width and height.

The measurement show that not every blood cell could be identified accurately. The reason for this is that in the edged image, the contours of the blood cells have multiple stops and are therefore not interpreted as a whole cell. Another reason is some of the blood cells are too near each other and may be recognized as one.

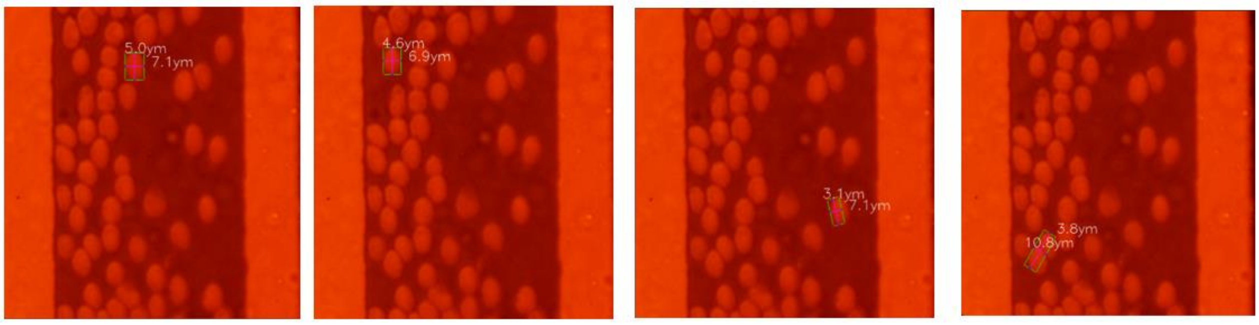


Figure 23: Measurement with two good examples and two poor, Own figure

However the cells that could be identified accurately show good results in the measurement.
In the first image in the figure above, the script calculated the dimensions of the red blood cells with a width of 5.0µm and a height of 7.1µm. Based on the average red blood cell diameter of 7-8 µm [14] of a healthy human and that the blood cells get stretched due to the velocity of passing through the channel, the results are reasonable. To be able to validate the measurement, the diameter of each cells would have to be known, or the median diameter of all the cells taken from the human.

6 Methods for volume extraction of objects in 2D Images

6.1 Volume estimation using multiple images with different views

One way to estimate the volumetric parameters of an object, is using multiple images, which show the object in different perspectives and angles. This method is an extension of the methodology applied for this Industry Project 2, described in chapter 4 and is commonly used in applications like estimating the calories intake of food portions for example.

By taking a minimum of two pictures of the object of desire and putting a reference object of known length next to it, it is possible to approximately calculate the volume of some generic shaped objects, like buckets, cups, books, bottle etc. [15]

Take for example the blue Jenga from the tests. With the two images, which show the piece in different views, it is possible to calculate the volume of the object.

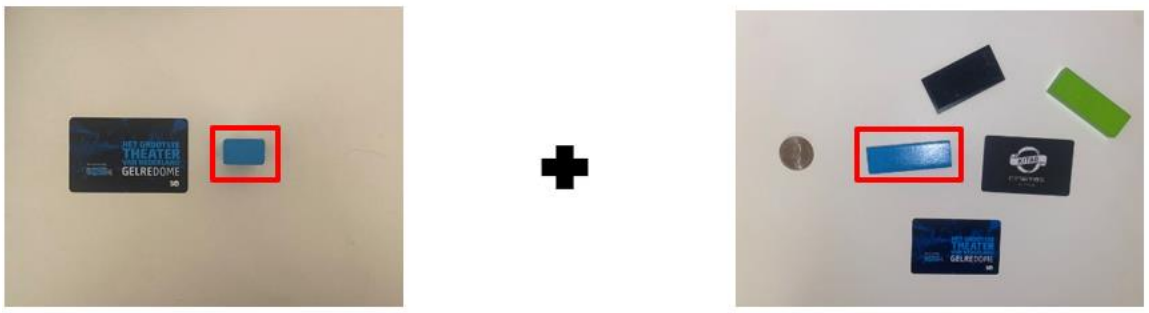


Figure 24: Methodology of using two images to calculate the volume, Own figure

A requirement of this method is, that the measurements of the different views are accurate enough and potential error are eliminated.

For the purpose of measuring the volume of a red blood cell this process is not very applicable, as it would require to have another microscopy image portraying the cells from a different angle, preferably from the side, in an 90° angle. Because the channel has only a thickness of 10µm, installing a microscope to capture an image, would be nearly impossible. This methodology can therefore be written off due to the equipment limitations.

6.2 Machine Learning

Heliyon presented a method to use a deep-learning algorithm, namely convolutional neural network (CNN), to be able to estimate the red wine volume filled in wine glasses.

They created a database with various images which show filled wine glasses in different angles and distances to the camera. They have also used different glasses. To each of the image, the filled in volume of the wine is attached, which range from 50ml to 300ml.

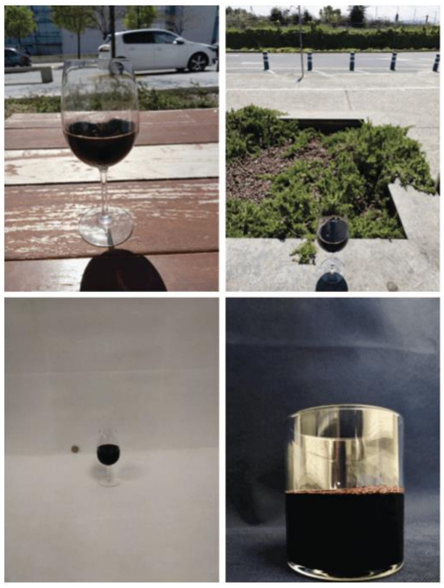


Figure 25: Sample images of database, Source: [16]

By training the model they were able to estimate the volume with no mean absolute error (MAE) higher than 10%. The following images show the MAEs for each amount of wine filled in the glasses.
[16]

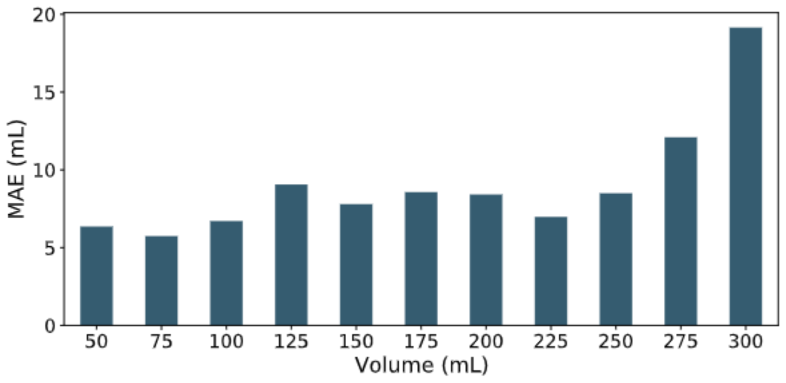


Figure 26: Results of the model with different wine volumes, Source: [16]

If applied to the RBC image, this methodology would only be able to, like the previous subchapter, estimate the volume of the cell and not gather more information about its shape. Therefore no real visualization could be made with this method. Further, it is technically very challenging to manually measure the volume of one single blood cell with medical instruments and draw the link to the cell position in the image taken.

6.3 Molecular Depth estimation

An extension of the method, shown by *Helion* [16] to predict the wine volume, is the usage of molecular depth-estimation networks, presented by *Graikos* et al.[17], to estimate food volume from a single input image.

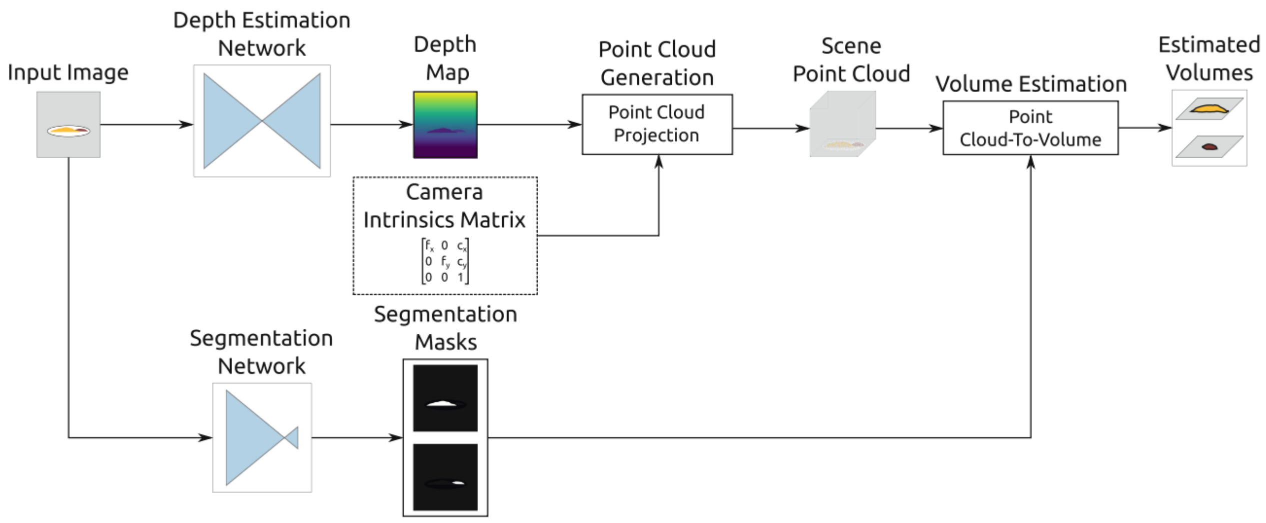


Figure 27: Proposed System Architecture, Source : [17]

The authors propose a framework that combines a depth prediction network with a food segmentation network to estimate the volume of individual food items in the image.
The depth-estimation network is trained on image sequences, 3 consecutive frames of a video, which the authors have collected from a database, which consists of over 50 hours of food video material. With the help of the captured frames, in which each of those frames the camera has another position relative to the food, it is possible, together with the intrinsic parameters of the camera, to estimate the depth map of the image scene, without having a reference object.

Together with the food segmentation, which was also trained with a neural network, they are able to project each pixel to a corresponding point in a 3D space (Point cloud representation).

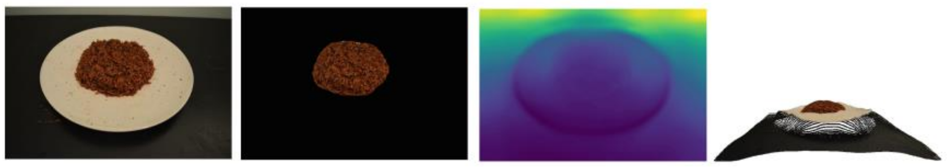


Figure 28: From left to right: Input Image, Segmentation mask, Depth estimation map, Point cloud generation; Source: [17]

This methodology of estimating the volume would match very good with the case of the RBCs, because it does not only estimates the volume, it also takes the topographical information into consideration.

Problems that could occur when applying this methodetology is the fact that, to have accurate performance of the model, enough prelabelled training data of the RBCs has to be available. Further in the research paper, in each frame of the videos, the food is shown from different angles from the camera. In case of this industry project , it would not be the camera that is changing the position, rather it is the RBC. And because the cells are moving really fast through the channel, it would be challenging being able to track them across multiple frames.

6.4 Information about the deformation

Another way to estimate the volume of a RBC could be, taking their biomechanical properties [18] into consideration. While the RBC are passing through the channel in high velocity, they are being stretched in one axis and are exposed to stress in that direction.

Assuming having a perfectly shaped RBC as a synthetic model, which has the geometrical properties of a healthy human blood cell. It would be possible to estimate the volume of the stretched model based on the biomechanical properties of the RBC and the surface area, which can be estimated with the script developed in this project.

The difficulty of this method would be in knowing of how much stress the RBCs are exposed to, when passing through the channel. Also the assumption would have to be made that not every cell is exposed to the same amount of stress. Measuring these differences would be very challenging with limited equipment.

6.5 Visualization of extracted volume

6.5.1 Surface plotting

One method of visualizing a red blood cell shaped volumetric object, is using popular libraries like matplotlib, see figure below. For this, some mathematical parameters have to be defined to be able to plot the surface. There exist also other libraries, like mayavi or plotly. [19]

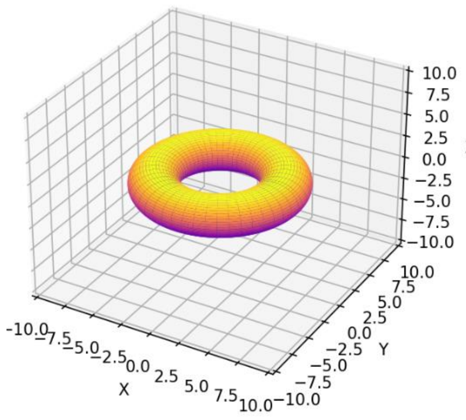


Figure 29: Torus Visualization using matplotlib, Source: Own figure Torus_example.py

Because these mathematical models don't have any unevenness in their shapes, the difficulty of this method would be, translating the data gathered from the 2D-image to generate a new 3D visualization, which show the deformation of the actual RBC in the real world.

6.5.2 CAD

An other possibility would be, creating a three-dimensional model with the help of the pyautocad library. With this librabry it would possible to create an interface between python and the AutoCAD program, to create 3D visualizations of the RBC. [20]

Another advantage would be, that it is easier to deform after the creation since it is not tied to a mathematical function, rather the shape of the RBC can be deformed freely using the computer mouse.

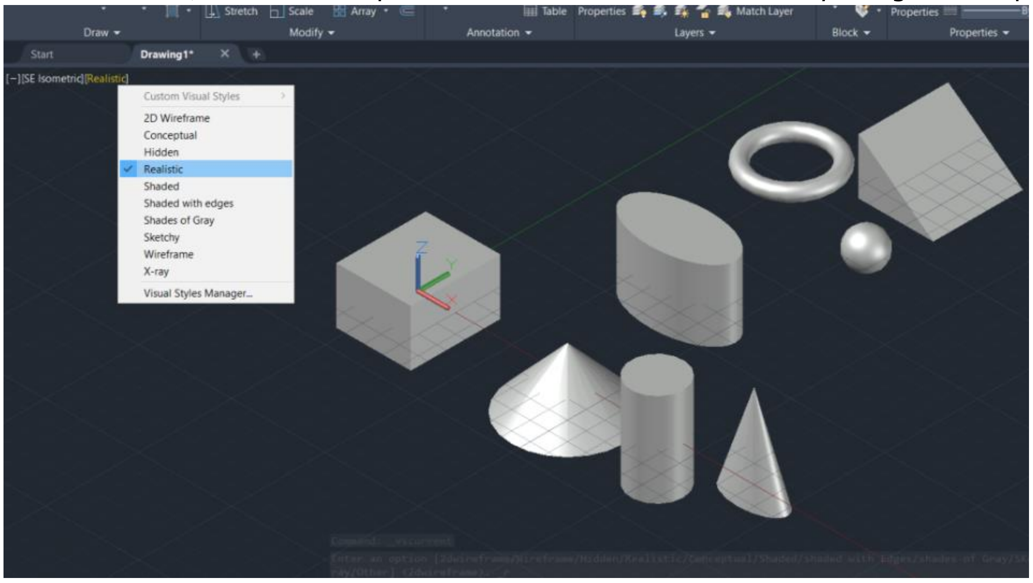
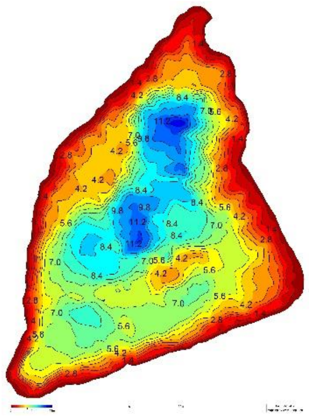


Figure 30: Pyautocad various objects, Source: [13]

Also for this method, the difficulty would be to translate the data gathered from the image so that an 3D CAD file can be created.

6.5.3 2D topographical image

As a last method, the idea of a topographical could be an elegant way to visualize the depth and therefore volumetric information of the RBCs in the images, similar to the topographical maps used in the field of cartography to show the depth of oceans, seas, valleys etc. This method would make it easier to visualize the gathered information from the images, as no 3D visualization has to be generated, rather the gained information can be directly converted to the depth map.



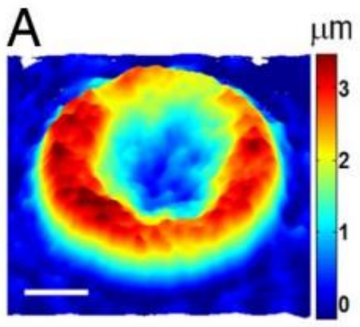


Figure 31: Topographical image of red blood cell, Source: [14]

Figure 32: Depth map of waters, Source: asv-braunschweig.de

7 Discussion

7.1 Created Python scripts

For this project several Python scripts have been created for each topic covered. The files can be accessed via the GitHub repository: *https://github.com/DominiquePeytrignet/BTHE* or in the Appendix.

The notebook "Object_Measurements" covers all the measurement taken with the selected objects.

The notebook "Red_Blood_Cell" deals with the RBC image provided by the supervisors. It covers the topic about calculating the pixel per metric ratio, measuring the size of the RBC, creating a cropped out image of a single cell and an optimized detection method. This notebook creates four images in the folder "Created_Images".

With the "Alternative_shapes" notebook, some possibilities were demonstrated on what alternative shapes can be identified. Methods for detecting or drawing circles, polygon and ellipses on objects are applied in the script.

The last script "Torus_example" shows an example of how the python library "matplotlib" can be used to draw 3-dimensional surfaces. In this example the shape of a torus is demonstrated and shows that it would be possible to also draw a red blood cell, which has an biconcave disk shape.

7.2 Answering the research questions

- What methods are available to measure the x and y dimensions of an object in an image and how do they work?

The main methodology that was applied in this project, for measuring the x and y dimension, is the usage of an reference length of an object with known size. For the measurements with the various objects, the width or the height of the left most object was chosen to be the reference length. For the image with the RBCs, the channel width was known and therefore taken as reference length.

As an alternative methodology to use the left most object as reference, the concept of using ArUcoMarkers was introduced. This makes it easier for the camera to identify the marker.

-Which factor have a positive or negative influence on the result?

It is of great importance that the light conditions are optimal, when taking the photos, so that the Canny algorithm doesn't interpret a shadow as an edge. Also to improve the edge detection the contrast between the object and the background should be as high as possible, i.e. one shouldn't put a white object on an white table.

Further it is important that the camera angle is in alignment with the edges of the object, so that the measuring error from the parallax effect is minimized. To achieve this, it is better to not put too many objects in one image, because this will put them further away from the image center.

As a last point, the object to be measured should have the same distance to the camera as the reference object. Because the camera can't distinguish between the depth difference of the two objects without more information.

-What are additional shapes that can be measured besides rectangles only?

In the Jupyter Notebook the possibility of measuring circles and polygons with various amount of edges are demonstrated. Further a method is presented on how an ellipse can be fitted to a detected rectangle, which can be useful for the RBC image.

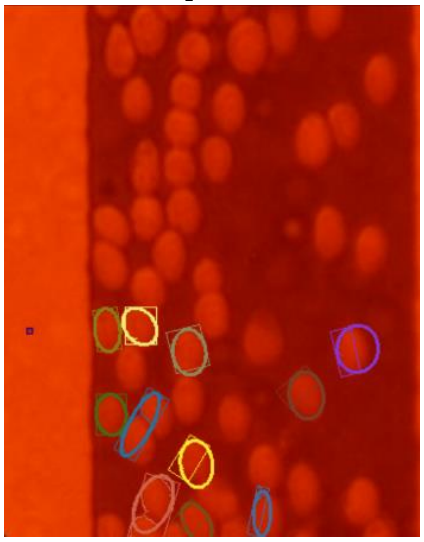


Figure 33: Fitted ellipses on rectangles, Own figure

-How can the volume of an object be extracted from an image and visualized?
Some methods were presented in Chapter 6 on how the volume of an object can be estimated and additionally on how the object can be visualized as a three-dimensional representation.

The methods of using two or more images with a big enough angle difference between them, can't be applied to them RBC image, because there are no resources available at the BFH site.

The most promising methodology, for extracting the three-dimensional information from the RBC images, are the examples of estimating the food and wine volumes. With the molecular depth network it would be possible to create a point cloud of the blood cell, as a visualization, and estimate the volume. However a challenge would be to create the database, which needs 3 consecutive frames of a video, which would require an algorithm that can track the position of the RBC in the video sequence.

Another optimistic possibility is the idea of taking the deformation of the RBC into consideration and derive the volume from the forces applied to the cell. The only difficulty would be actually measuring the forces which are applied to a single blood cell.

As a visualization choice, the depth map would probably be the best suitable, as it can be directly applied to the two-dimensional image and doesn't need a 3D representation.

7.3 Retrospective of goals

The goal for this Industry Project 2 was to create a python script which is capable to measure the length and width of various objects shown on an image and test the accuracy of the results with multiple measurements. As a result two list should be presented, one with the factors, which have a negative impact on the results and one which shows different methods to measure two-dimensional properties of objects.

These goals and results are achieved. In chapter 4 the test with the various items were performed and the accuracy measured. Further the major causes of inaccuracy are explained. In chapter 5 the measurement on red blood cell was performed. The accuracy couldn't be measured because the ground truth dimensions of the cells are not known. But the results seem plausible.

As an addition and as a look-out for the following Bachelor Thesis, methods for estimating the volume of objects in two-dimensional image are presented. These methods will serve as a research foundation for the thesis.

7.4 Potential

In the Jupyter Notebook file "Red_Blood_Cell", a method was presented on how the RBC can be detected more efficiently. The result of the local maxima method are decent, because not every cell was identified perfectly, some cells were interpreted as multiple maximas. Further it also detects maximas which aren't cells, i.e. the channel walls.

A better solution to detect the RBCs on images or eventually live tracking on the video sequences could be, training a customized Haar-cascade classifier on a self-developed dataset. The dataset would consist of various positive and negative examples of red blood cells. The good examples would have to be manually cut out from the video sequences, that are available at the BFH site.

This would be very time consuming as for a good Haar-cascade classifier with decent accuracy, approximately 1000 positive images and 10'000 negative are required, when building a classifier which can classify different objects [21]. Because the features of a different object have much more variation than that of a RBC, the amount of images required would probably be much less. Still the question remains, what images should be used for the negative examples.

7.5 Critical points

Some points which are important to mention:

- In the search for alternative methods to measure the x- and y-dimensions of objects in image, methods of using distance sensors or stereo images are not considered, because the equipment and time resources are not available.

- The RBC measurements are based on the assumption the surfaces of the red blood cells are in right angle to the camera view. There is the small possibility that the side edge of the RBC disk is facing the camera view and therefore falsify the results. Although the probability are very low because the channel thickness (z-direction) is very thin, approximately 10µm.

- The idea of creating a connection between python and AutoCAD to create RBC models, hasn't been tested.

- To have a statistical more significant results for the measurement test, typically more measurements would have to be performed. Therefore the results should be taken with a grain of salt.

8 Conclusion , recommandations, future works

In this project a measurement test was performed on a selection of objects, where the width and height were compared to the their real-world dimensions. The results showed that the measurements can differ up to 10% from the real-world values. Higher deviation can be caused due to parallax errors, non-optimal lightning conditions or depth difference between the objects.

Next the same methodology, as with the various object, was applied to the Red blood cell image. First the new pixel per metric ratio was calculated and then the width and height of the cell was measured. The results show that the measured dimensions of the cells are reasonable when comparing them to other healthy humans.

For future work, namely the Bachelor Thesis, the goal would be to find a way to estimate the volume of a single RBC and create an appealing 3D visualization. For this, first research has been conducted during this project. The most promising method, in the view of the author, is the usage of monocular-depth estimations , because it is capable of creating a three-dimensional visualization together with a volume estimation.

The author recommends therefore for the Bachelor Thesis as a first step, diving deeper into this topic to see if the same method can be a applied to the RBC video sequence. As a second step a further research could be made with analyses other methodologies to estimate the volume of the cells.

9 Contents of figures

10 Contents of tables

11 Bibliography

[1] N. Mahamkali and V. Ayyasamy, “OpenCV for Computer Vision Applications,” Mar. 2015.

[2] “Images & pixels — Introduction to Bioimage Analysis.” https://bioimagebook.github.io/chapters/1-concepts/1-images_and_pixels/images_and_pixels.html (accessed Feb. 26, 2023).

[3] “Machine Learning - Going Furthur with CNN Part 2,” *DEV Community*, Mar. 17, 2020. https://dev.to/sandeepbalachandran/machine-learning-going-furthur-with-cnn-part-2-41km (accessed Feb. 26, 2023).

[4] “CVL Tools Vision Guide - Gaussian Convolution - Documentation | Cognex.” https://support.cognex.com/docs/cvl_900/web/EN/cvl_vision_tools/Content/Topics/VisionTools/Gaussian_Convolution.htm (accessed Feb. 26, 2023).

[5] R. C. Gonzalez and R. E. Woods, *Digital image processing*. New York, NY: Pearson, 2018.

[6] “Edge Detection Using OpenCV | LearnOpenCV #,” Jun. 10, 2021. https://learnopencv.com/edge-detection-using-opencv/ (accessed Feb. 27, 2023).

[7] Y. Lee, “Literature review on edge detection methods”.

[8] “OpenCV: Canny Edge Detection.” https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html (accessed Feb. 27, 2023).

[9] “OpenCV: Detection of ArUco Markers.” https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html (accessed Mar. 12, 2023).

[10] J. F. Hoffman, “Biconcave shape of human red-blood-cell ghosts relies on density differences between the rim and dimple of the ghost’s plasma membrane,” *Proc. Natl. Acad. Sci.*, vol. 113, no. 51, pp. 14847–14851, Dec. 2016, doi: 10.1073/pnas.1615452113.

[11] “OpenCV: Hough Line Transform.” https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html (accessed Mar. 15, 2023).

[12] I. Macdonald, “Probabilistic Hough Transform”.

[13] “ijcsit20140503325.pdf.” Accessed: Feb. 27, 2023. [Online]. Available: https://ijcsit.com/docs/Volume%205/vol5issue03/ijcsit20140503325.pdf

[14] M. Kinnunen, A. Kauppila, A. Karmenyan, and R. Myllylä, “Effect of the size and shape of a red blood cell on elastic light scattering properties at the single-cell level,” *Biomed. Opt. Express*, vol. 2, no. 7, pp. 1803–1814, Jun. 2011, doi: 10.1364/BOE.2.001803.

[15] A. S. Parihar, M. Gupta, V. Sikka, and G. Kaur, “Dimensional analysis of objects in a 2D image,” in *2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, Jul. 2017, pp. 1–7. doi: 10.1109/ICCCNT.2017.8203937.

[16] “Artificial intelligence to estimate wine volume from single-view images | Elsevier Enhanced Reader.” https://reader.elsevier.com/reader/sd/pii/S240584402201845X?token=23012F6EABC3FDC65B414A479FB632ADA2DD1F5517A3FDCE59FD67E5AAF6CC2B3882BDE55EA6C28C761A76BA8AE0EDE4&originRegion=eu-west-1&originCreation=20230319164442 (accessed Mar. 19, 2023).

[17] “Graikos et al. - 2020 - Single Image-Based Food Volume Estimation Using Mo.pdf.” Accessed: Mar. 19, 2023. [Online]. Available: https://protein-h2020.eu/wp-content/uploads/2020/09/Single-Image-Based-Food-Volume-Estimation-Using-Monocular-Depth-Prediction-Networks.pdf

[18] G. Tomaiuolo, “Biomechanical properties of red blood cells in health and disease towards microfluidics,” *Biomicrofluidics*, vol. 8, no. 5, p. 051501, Sep. 2014, doi: 10.1063/1.4895755.

[19] “3D surface (colormap) — Matplotlib 3.7.1 documentation.” https://matplotlib.org/stable/gallery/mplot3d/surface3d.html (accessed Mar. 17, 2023).

[20] T. Sawant, “Solid AutoCAD objects in pyautocad (Python),” *SCDA*, Sep. 23, 2021. https://www.supplychaindataanalytics.com/solid-objects-in-autocad-using-pyautocad-python/ (accessed Mar. 17, 2023).

[21] B. P. Le, H. Jeon, N. Truong, and J. Hak, “Applying the Haar-cascade Algorithm for Detecting Safety Equipment in Safety Management Systems for Multiple Working Environments,” *Electronics*, vol. 8, p. 1079, Sep. 2019, doi: 10.3390/electronics8101079.

12 Declaration of authorship

I hereby certify that I composed this work completely unaided, and without the use of any other sources or resources other than those specified in the bibliography. All text sections not of my authorship are cited as quotations and accompanied by an exact reference to their origin.

Bern, 03.04.23

[illegible]

____
City, Date   Dominique Peytrignet

13 Version control

Version	Date	Description	Author
0.1	02.03.2022	Document created	Dominique Peytrignet
0.2	03.04.2023	Document finished	Dominique Peytrignet

14 Appendix

14.1 Object_Measurements.ipynb

Measure width and height of some objects using reference object

Import dependencies

In [106…
```
from scipy.spatial import distance as dist
from imutils import perspective
from imutils import contours
from matplotlib import pyplot as plt

import numpy as np
import imutils
import cv2
```

Function to compute midpoint from two points

In [107…
```
def midpoint(ptA, ptB):
        return ((ptA[0] + ptB[0]) * 0.5, (ptA[1] + ptB[1]) * 0.5)
```

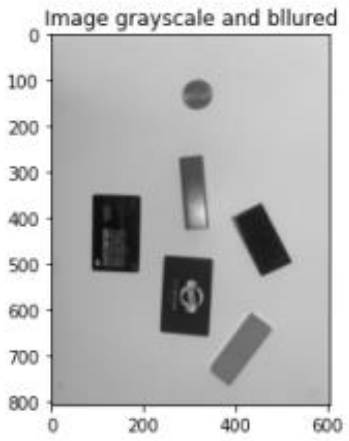
Define width of left most object in image and image-path

In [108…
```
width =  53.88 # change when new reference object
image_path = "Images/Test_1_90.jpg"
```

Load and blur image slightly

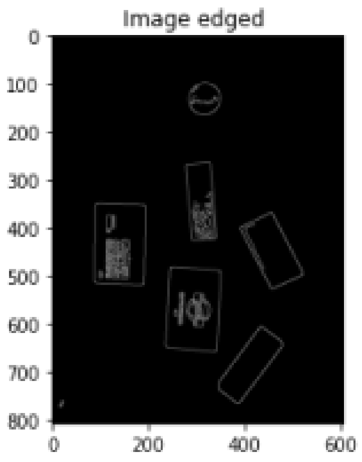
In [109…
```
image = cv2.imread(image_path)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (5, 5), 0)
```

In [110…
```
gray_show = cv2.cvtColor(gray, cv2.COLOR_GRAY2RGB)
plt.imshow(gray_show)
plt.title("Image grayscale and bllured")
plt.show()
```



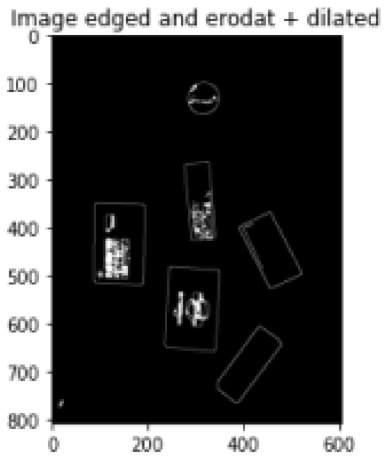
Edge detection and dilation and erotion to close gaps between object edges

In [111…
```
edged = cv2.Canny(gray, 50, 100)
edged_show = cv2.cvtColor(edged, cv2.COLOR_GRAY2RGB)
plt.imshow(edged_show)
plt.title("Image edged ")
plt.show()
```



In [112…
```
edged = cv2.dilate(edged, None, iterations=1)
edged = cv2.erode(edged, None, iterations=1)
```

In [113…
```
edged_show = cv2.cvtColor(edged, cv2.COLOR_GRAY2RGB)
plt.imshow(edged_show)
plt.title("Image edged and erodat + dilated")
plt.show()
```



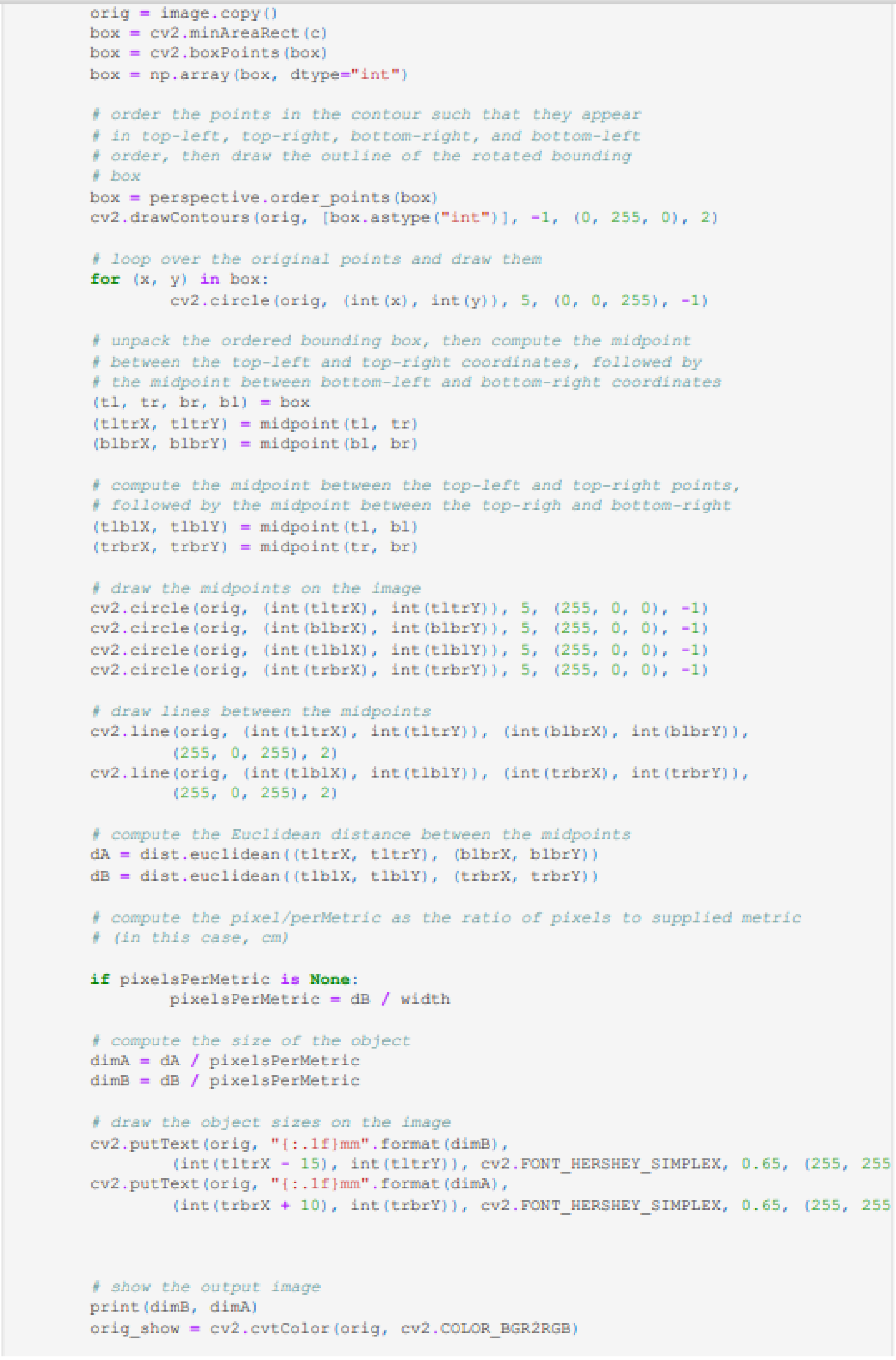
Find contours in edged image

In [114…
```
cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
```

Sort contours from left ro right so that left most object is reference object

In [115…
```
(cnts, _) = contours.sort_contours(cnts)
pixelsPerMetric = None
```

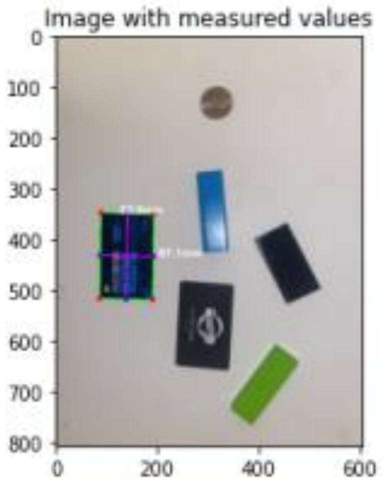
In [116…
```
cv2.imshow("Edged", edged)
cv2.waitKey(0)
cv2.destroyAllWindows()
for c in cnts:
        # if the contour is not sufficiently large, ignore it
        if cv2.contourArea(c) < 1900:
                continue

        # compute the rotated bounding box of the contour
```



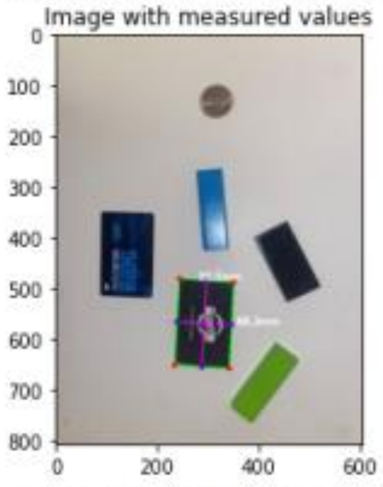
```
orig = image.copy()
box = cv2.minAreaRect(c)
box = cv2.boxPoints(box)
box = np.array(box, dtype="int")

# order the points in the contour such that they appear
# in top-left, top-right, bottom-right, and bottom-left
# order, then draw the outline of the rotated bounding
# box
box = perspective.order_points(box)
cv2.drawContours(orig, [box.astype("int")], -1, (0, 255, 0), 2)

# loop over the original points and draw them
for (x, y) in box:
        cv2.circle(orig, (int(x), int(y)), 5, (0, 0, 255), -1)

# unpack the ordered bounding box, then compute the midpoint
# between the top-left and top-right coordinates, followed by
# the midpoint between bottom-left and bottom-right coordinates
(tl, tr, br, bl) = box
(tltrX, tltrY) = midpoint(tl, tr)
(blbrX, blbrY) = midpoint(bl, br)

# compute the midpoint between the top-left and top-right points,
# followed by the midpoint between the top-righ and bottom-right
(tlblX, tlblY) = midpoint(tl, bl)
(trbrX, trbrY) = midpoint(tr, br)

# draw the midpoints on the image
cv2.circle(orig, (int(tltrX), int(tltrY)), 5, (255, 0, 0), -1)
cv2.circle(orig, (int(blbrX), int(blbrY)), 5, (255, 0, 0), -1)
cv2.circle(orig, (int(tlblX), int(tlblY)), 5, (255, 0, 0), -1)
cv2.circle(orig, (int(trbrX), int(trbrY)), 5, (255, 0, 0), -1)

# draw lines between the midpoints
cv2.line(orig, (int(tltrX), int(tltrY)), (int(blbrX), int(blbrY)),
        (255, 0, 255), 2)
cv2.line(orig, (int(tlblX), int(tlblY)), (int(trbrX), int(trbrY)),
        (255, 0, 255), 2)

# compute the Euclidean distance between the midpoints
dA = dist.euclidean((tltrX, tltrY), (blbrX, blbrY))
dB = dist.euclidean((tlblX, tlblY), (trbrX, trbrY))

# compute the pixel/perMetric as the ratio of pixels to supplied metric
# (in this case, cm)

if pixelsPerMetric is None:
        pixelsPerMetric = dB / width

# compute the size of the object
dimA = dA / pixelsPerMetric
dimB = dB / pixelsPerMetric

# draw the object sizes on the image
cv2.putText(orig, "{:.1f}mm".format(dimB),
        (int(tltrX - 15), int(tltrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255
cv2.putText(orig, "{:.1f}mm".format(dimA),
        (int(trbrX + 10), int(trbrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255



# show the output image
print(dimB, dimA)
orig_show = cv2.cvtColor(orig, cv2.COLOR_BGR2RGB)
```

```
        plt.imshow(orig_show)
        plt.title("Image with measured values")
        plt.show()

        cv2.imshow("Image", orig)


        cv2.waitKey(0)


cv2.destroyAllWindows()
```

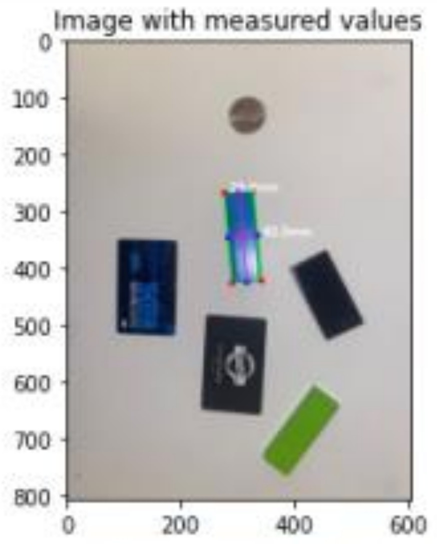
```
53.88 87.12965403601484
```



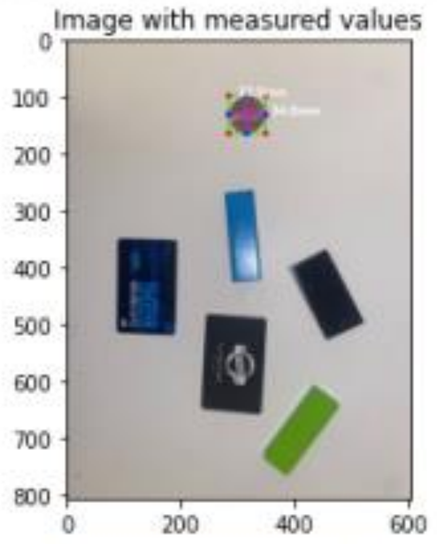
```
55.50289562773611 88.28706056927709
```



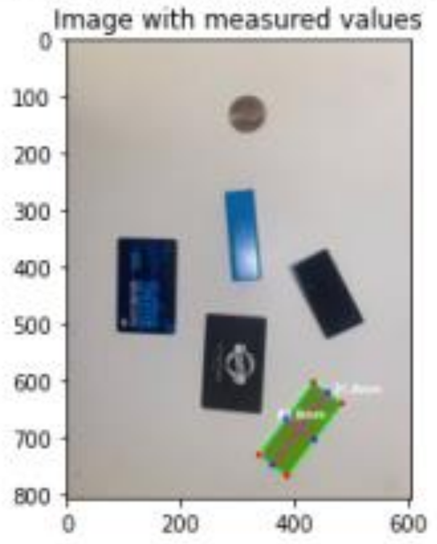
```
26.90637877313119 82.53896541717818
```



```
33.50773935184458 34.023243034180645
```



```
81.91360000721394 31.858169275759515
```



```
75.62150866446135 38.84876929964136
```

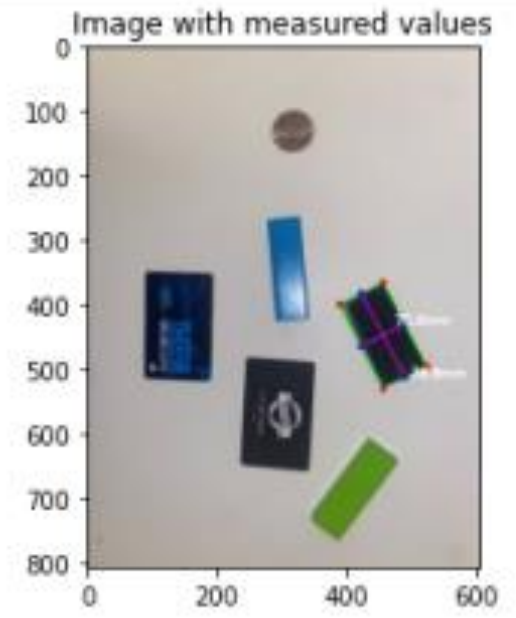


Image with measured values
0
100
200
300
400
500
600
700
800
0
200
400
600

14.2 Red_Blood_Cell.ipynb

Measuring width and height of red blood cells

Calculate PixelPerMetric Ratio

Import dependencies

```
In [276…  from scipy.spatial import distance as dist
          from matplotlib import pyplot as plt
          from imutils import perspective
          from imutils import contours
          import imutils
          import numpy as np
          import math
          import cv2
```

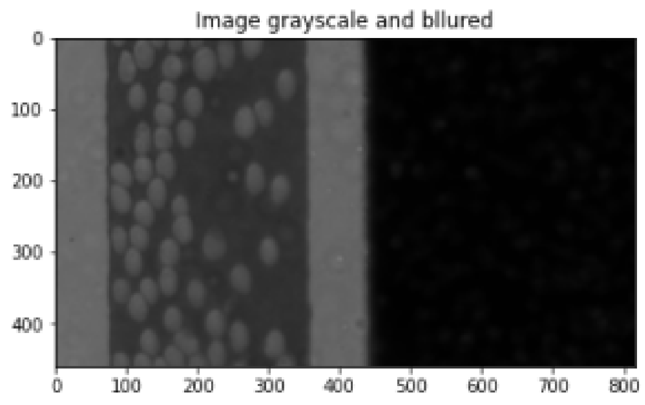
Function to compute midpoint from two points

```
In [277…  def midpoint(ptA, ptB):
                  return ((ptA[0] + ptB[0]) * 0.5, (ptA[1] + ptB[1]) * 0.5)
```

```
In [278…  image_path_red = "Images/RBC.png"
```

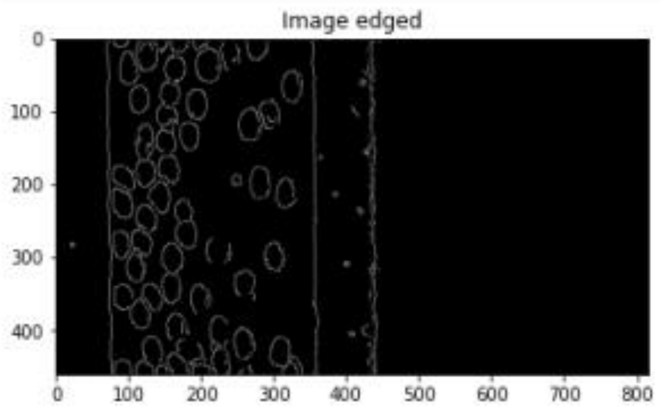
```
In [279…  image = cv2.imread(image_path_red)
          gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
          gray = cv2.GaussianBlur(gray, (5, 5), 0)
```

```
In [280…  # show in notebook
          gray_show = cv2.cvtColor(gray, cv2.COLOR_GRAY2RGB)
          plt.imshow(gray_show)
          plt.title("Image grayscale and bllured")
          plt.show()
```



Canny edge detection

```
In [281…  edged = cv2.Canny(gray, 10, 30, None, 3)
          cv2.imwrite('Created_Images/Edged_image.png', edged)

          edged_show = cv2.cvtColor(edged, cv2.COLOR_GRAY2RGB)

          plt.imshow(edged_show)
          plt.title("Image edged")
          plt.show()
```



In [282...
```
edged_BGR = cv2.cvtColor(edged, cv2.COLOR_GRAY2BGR)
```

Line detection

https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html

In [283...
```
lines = cv2.HoughLinesP(edged, 1, np.pi / 180, 10, None, 50, 10)
```

In [284...
```
if lines is not None:
        for i in range(0, len(lines)):
            l = lines[i][0]
            cv2.line(edged_BGR, (l[0], l[1]), (l[2], l[3]), (0,0,255), 3, cv2.LINE_AA)

print(lines)


cv2.imshow("Lines", edged_BGR)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
[[[  0   1 449   1]]

 [[356 223 356 143]]

 [[440 458 440 401]]

 [[436  83 437 146]]

 [[ 75 372  76 422]]

 [[434  97 434  43]]

 [[439 344 439 282]]

 [[356  70 357 120]]

 [[356 256 357 319]]]
```

9 Lines have been identified, where there is minimum one line on each side of the channel (Kanal) The channel has a known width of 50ym A line consists of two points: the starting point (x1, y1) and the end point (x2, y2)

with the starting point of the line from the left edge A1 (75, 372) and the end point A2 (76,422) it is possible to calculate the difference in the x-axis of the image with the second line from the right edge with B1 (356, 223) and B2 (356, 143). This difference will be the reference length to calculate the pixel per metric ratio of this image:

Pixel length: 356-75 =281 pixels Known Length: 50ym pixel per metric ratio: 281 / 50 = 5.62 pixels/ym

```
In [285… a = lines[1][0][0]
        b = lines[4][0][0]
        pixels = a-b
        width = 50 #ym
        pixelsPerMetric = pixels / width
        print(pixelsPerMetric)
```

```
5.62
```

Detecting and measuring Red blood cells

```
In [286… cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
```

```
In [287… for c in cnts:
            if cv2.contourArea(c) <10:
                continue

            #print(cv2.contourArea(c))
            orig = image.copy()
            box = cv2.minAreaRect(c)
            box = cv2.cv.BoxPoints(box) if imutils.is_cv2() else cv2.boxPoints(box)
            box = np.array(box, dtype="int")

            # order the points in the contour such that they appear
            # in top-left, top-right, bottom-right, and bottom-left
            # order, then draw the outline of the rotated bounding
            # box
            box = perspective.order_points(box)
            cv2.drawContours(orig, [box.astype("int")], -1, (0, 255, 0), 1)

            # loop over the original points and draw them
            for (x, y) in box:
                cv2.circle(orig, (int(x), int(y)), 1, (0, 0, 255), -1)

                # unpack the ordered bounding box, then compute the midpoint
                # between the top-left and top-right coordinates, followed by
                # the midpoint between bottom-left and bottom-right coordinates
            (tl, tr, br, bl) = box
            (tltrX, tltrY) = midpoint(tl, tr)
            (blbrX, blbrY) = midpoint(bl, br)

                # compute the midpoint between the top-left and top-right points,
                # followed by the midpoint between the top-righ and bottom-right
            (tlblX, tlblY) = midpoint(tl, bl)
            (trbrX, trbrY) = midpoint(tr, br)

                # draw the midpoints on the image
            cv2.circle(orig, (int(tltrX), int(tltrY)), 1, (255, 0, 0), -1)
            cv2.circle(orig, (int(blbrX), int(blbrY)), 1, (255, 0, 0), -1)
            cv2.circle(orig, (int(tlblX), int(tlblY)), 1, (255, 0, 0), -1)
            cv2.circle(orig, (int(trbrX), int(trbrY)), 1, (255, 0, 0), -1)

                # draw lines between the midpoints
            cv2.line(orig, (int(tltrX), int(tltrY)), (int(blbrX), int(blbrY)),
                    (255, 0, 255), 1)
            cv2.line(orig, (int(tlblX), int(tlblY)), (int(trbrX), int(trbrY)),
                    (255, 0, 255), 1)

                # compute the Euclidean distance between the midpoints
```

```
    dA = dist.euclidean((tltrX, tltrY), (blbrX, blbrY))
    dB = dist.euclidean((tlblX, tlblY), (trbrX, trbrY))

        # compute the pixel/perMetric as the ratio of pixels to supplied metric
        # (in this case, cm)

    if pixelsPerMetric is None:
        pixelsPerMetric = dB / width

        # compute the size of the object
    dimA = dA / pixelsPerMetric
    dimB = dB / pixelsPerMetric

        # draw the object sizes on the image
    cv2.putText(orig, "{:.1f}ym".format(dimB),
                (int(tltrX - 15), int(tltrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255
    cv2.putText(orig, "{:.1f}ym".format(dimA),
                (int(trbrX + 10), int(trbrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255




    cv2.imshow("Image", orig)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

Some blood cells weren't able to be detected proparly by the program, but some RBC are well detected and the with and height could be measured.

Trying to fill in the gaps by further preprocessing the edged image with dilation and morphological transformation,slightly improved the detection of single RBCs

https://docs.opencv.org/3.4/d4/d76/tutorial_js_morphological_ops.html

In [288…
```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
dilated = cv2.dilate(edged, kernel)
cnts = cv2.findContours(dilated.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
```

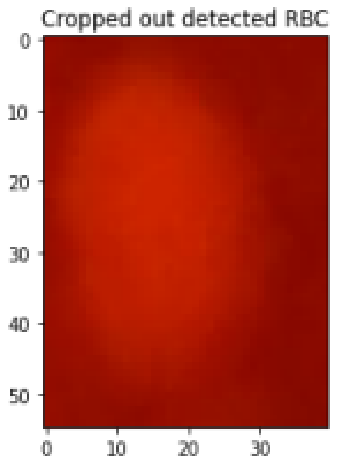
In [289…
```
for c in cnts:
    if cv2.contourArea(c) <100:
        continue

    #print(cv2.contourArea(c))
    orig = image.copy()
    orig2 = orig.copy()
    box = cv2.minAreaRect(c)
    box = cv2.boxPoints(box)
    box = np.array(box, dtype="int")


    # order the points in the contour such that they appear
    # in top-left, top-right, bottom-right, and bottom-left
    # order, then draw the outline of the rotated bounding
    # box
    box = perspective.order_points(box)
    cv2.drawContours(orig, [box.astype("int")], -1, (0, 255, 0), 1)

    # loop over the original points and draw them
    for (x, y) in box:
        cv2.circle(orig, (int(x), int(y)), 1, (0, 0, 255), -1)

        # unpack the ordered bounding box, then compute the midpoint
        # between the top-left and top-right coordinates, followed by
        # the midpoint between bottom-left and bottom-right coordinates
    (tl, tr, br, bl) = box
    (tltrX, tltrY) = midpoint(tl, tr)
    (blbrX, blbrY) = midpoint(bl, br)

        # compute the midpoint between the top-left and top-right points,
        # followed by the midpoint between the top-righ and bottom-right
    (tlblX, tlblY) = midpoint(tl, bl)
    (trbrX, trbrY) = midpoint(tr, br)

        # draw the midpoints on the image
    cv2.circle(orig, (int(tltrX), int(tltrY)), 1, (255, 0, 0), -1)
    cv2.circle(orig, (int(blbrX), int(blbrY)), 1, (255, 0, 0), -1)
    cv2.circle(orig, (int(tlblX), int(tlblY)), 1, (255, 0, 0), -1)
    cv2.circle(orig, (int(trbrX), int(trbrY)), 1, (255, 0, 0), -1)

        # draw lines between the midpoints
    cv2.line(orig, (int(tltrX), int(tltrY)), (int(blbrX), int(blbrY)),
             (255, 0, 255), 1)
    cv2.line(orig, (int(tlblX), int(tlblY)), (int(trbrX), int(trbrY)),
             (255, 0, 255), 1)

        # compute the Euclidean distance between the midpoints
    dA = dist.euclidean((tltrX, tltrY), (blbrX, blbrY))
    dB = dist.euclidean((tlblX, tlblY), (trbrX, trbrY))

        # compute the pixel/perMetric as the ratio of pixels to supplied metric
        # (in this case, cm)

    if pixelsPerMetric is None:
        pixelsPerMetric = dB / width

        # compute the size of the object
    dimA = dA / pixelsPerMetric
    dimB = dB / pixelsPerMetric

        # draw the object sizes on the image
    cv2.putText(orig, "{:.1f}ym".format(dimB),
             (int(tltrX - 15), int(tltrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255
    cv2.putText(orig, "{:.1f}ym".format(dimA),
             (int(trbrX + 10), int(trbrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255




    cv2.imshow("Image", orig)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

Cropping and adaptive thresholding

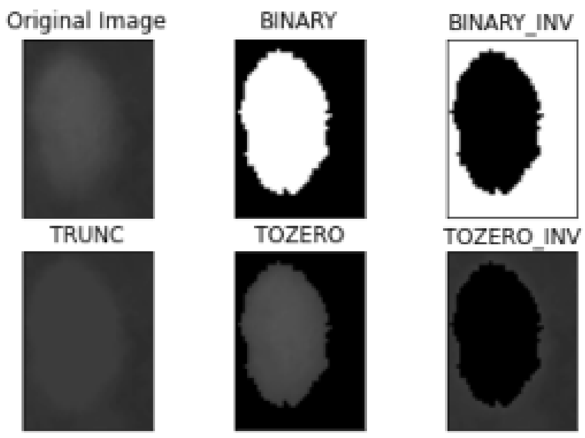
Trying to detect edges better by cropping out a well detected RBC and applying adaptive thresholding In the future an algorithm could be developed which automatically can classify which detected contours are actually single RBCs

for now this decision was made manually

```
In [290…  [X, Y, W, H] = cv2.boundingRect(cnts[4]) # 4=> manually identified contours box which ha
          a = 5
          cropped_image = image[Y:Y+H+a, X:X+W+a]
          cropped_image_show = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2RGB)
          plt.imshow(cropped_image_show)
          plt.title("Cropped out detected RBC")
          cv2.imwrite('Created_Images/cropped_RBC.png', cropped_image)

Out[290]: True
```



```
In [291…  # https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html
          import cv2 as cv
          import numpy as np
          from matplotlib import pyplot as plt
          a= 60
          img = cv.imread('Created_Images/cropped_RBC.png', cv.IMREAD_GRAYSCALE)
          assert img is not None, "file could not be read, check with os.path.exists()"
          ret,thresh1 = cv.threshold(img,a,255,cv.THRESH_BINARY)
          ret,thresh2 = cv.threshold(img,a,255,cv.THRESH_BINARY_INV)
          ret,thresh3 = cv.threshold(img,a,255,cv.THRESH_TRUNC)
          ret,thresh4 = cv.threshold(img,a,255,cv.THRESH_TOZERO)
          ret,thresh5 = cv.threshold(img,a,255,cv.THRESH_TOZERO_INV)
          titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
          images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
          for i in range(6):
              plt.subplot(2,3,i+1),plt.imshow(images[i],'gray',vmin=0,vmax=255)
              plt.title(titles[i])
              plt.xticks([]),plt.yticks([])
          plt.show()
```

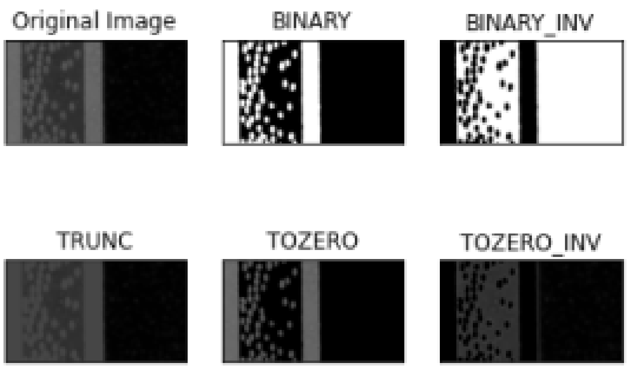


WIth the help of cropping and adaptive thresholding it was possible to focus on one single RBC. With the binary image generated,it would be theoreticaly possible to generate some sort of a topographical depth map, as explained in the report, if more informations would be available of this specific cell.

Trying to use the same thresholding method to the original image and see wheter it improves the detection.

In [292…

```
a= 70
img = gray
img = cv2.GaussianBlur(img, (5, 5), 0)
assert img is not None, "file could not be read, check with os.path.exists()"
ret,thresh1 = cv.threshold(img,a,255,cv.THRESH_BINARY)
ret,thresh2 = cv.threshold(img,a,255,cv.THRESH_BINARY_INV)
ret,thresh3 = cv.threshold(img,a,255,cv.THRESH_TRUNC)
ret,thresh4 = cv.threshold(img,a,255,cv.THRESH_TOZERO)
ret,thresh5 = cv.threshold(img,a,255,cv.THRESH_TOZERO_INV)
titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
for i in range(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray',vmin=0,vmax=255)
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
cv2.imwrite('Created_Images/Binary.png', thresh1)
```



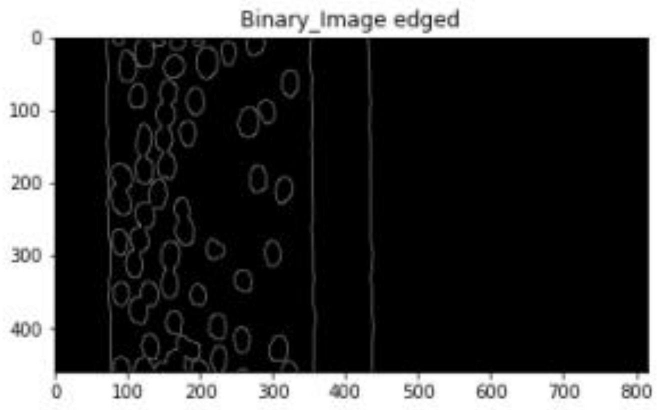
Out[292]:

```
True
```

using same methodology as before, by using canny and finding contours

In [293…

```
#Canny edge detection
binary_img = cv2.imread("Created_Images/Binary.png")
binary_edged = cv2.Canny(binary_img, 10, 30, None, 3)


binary_edged_show = cv2.cvtColor(binary_edged, cv2.COLOR_GRAY2RGB)

plt.imshow(binary_edged_show)
plt.title("Binary_Image edged")
plt.show()
```



```
In [294…  #find contours
          cnts = cv2.findContours(binary_edged.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
          cnts = imutils.grab_contours(cnts)
          # draw rectangles and measure

          for c in cnts:
              if cv2.contourArea(c) <100:
                  continue

              #print(cv2.contourArea(c))
              orig = image.copy()
              orig2 = orig.copy()
              box = cv2.minAreaRect(c)
              box = cv2.boxPoints(box)
              box = np.array(box, dtype="int")


              # order the points in the contour such that they appear
              # in top-left, top-right, bottom-right, and bottom-left
              # order, then draw the outline of the rotated bounding
              # box
              box = perspective.order_points(box)
              cv2.drawContours(orig, [box.astype("int")], -1, (0, 255, 0), 1)

              # loop over the original points and draw them
              for (x, y) in box:
                  cv2.circle(orig, (int(x), int(y)), 1, (0, 0, 255), -1)

                  # unpack the ordered bounding box, then compute the midpoint
                  # between the top-left and top-right coordinates, followed by
                  # the midpoint between bottom-left and bottom-right coordinates
              (tl, tr, br, bl) = box
              (tltrX, tltrY) = midpoint(tl, tr)
              (blbrX, blbrY) = midpoint(bl, br)

                  # compute the midpoint between the top-left and top-right points,
                  # followed by the midpoint between the top-righ and bottom-right
              (tlblX, tlblY) = midpoint(tl, bl)
              (trbrX, trbrY) = midpoint(tr, br)

                  # draw the midpoints on the image
              cv2.circle(orig, (int(tltrX), int(tltrY)), 1, (255, 0, 0), -1)
              cv2.circle(orig, (int(blbrX), int(blbrY)), 1, (255, 0, 0), -1)
              cv2.circle(orig, (int(tlblX), int(tlblY)), 1, (255, 0, 0), -1)
              cv2.circle(orig, (int(trbrX), int(trbrY)), 1, (255, 0, 0), -1)

                  # draw lines between the midpoints
              cv2.line(orig, (int(tltrX), int(tltrY)), (int(blbrX), int(blbrY)),
                          (255, 0, 255), 1)
```

```
    cv2.line(orig, (int(tlblX), int(tlblY)), (int(trbrX), int(trbrY)),
                (255, 0, 255), 1)

        # compute the Euclidean distance between the midpoints
    dA = dist.euclidean((tltrX, tltrY), (blbrX, blbrY))
    dB = dist.euclidean((tlblX, tlblY), (trbrX, trbrY))

        # compute the pixel/perMetric as the ratio of pixels to supplied metric
        # (in this case, cm)

    if pixelsPerMetric is None:
        pixelsPerMetric = dB / width

        # compute the size of the object
    dimA = dA / pixelsPerMetric
    dimB = dB / pixelsPerMetric

        # draw the object sizes on the image
    cv2.putText(orig, "{:.1f}ym".format(dimB),
                (int(tltrX - 15), int(tltrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255
    cv2.putText(orig, "{:.1f}ym".format(dimA),
                (int(trbrX + 10), int(trbrY)), cv2.FONT_HERSHEY_SIMPLEX, 0.65, (255, 255




    cv2.imshow("Image", orig)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

The result show that the script draws the rectangle a little too small, and don't cover the whole actual RBC. As an advantage though the single cell are detected more accurately

Other method to detect RBC

With the help of local maxima detection is it possible to identify the RBC much faster, and also get the coordinates of the detected dots, although in some RBC there are multiple dots.

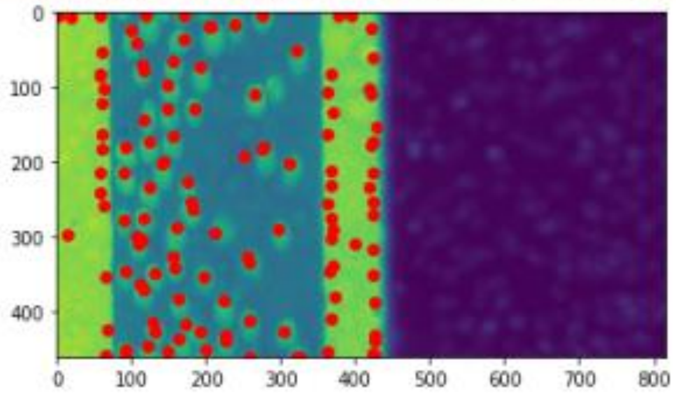
https://stackoverflow.com/questions/9111711/get-coordinates-of-local-maxima-in-2d-array-above-certain-value

In [295…

```
import numpy as np
import scipy
import scipy.ndimage as ndimage
import scipy.ndimage.filters as filters
import matplotlib.pyplot as plt


neighborhood_size = 30
threshold = 20

data = gray

data_max = filters.maximum_filter(data, neighborhood_size)
maxima = (data == data_max)
data_min = filters.minimum_filter(data, neighborhood_size)
diff = ((data_max - data_min) > threshold)
maxima[diff == 0] = 0

labeled, num_objects = ndimage.label(maxima)
```

```
xy = np.array(ndimage.center_of_mass(data, labeled, range(1, num_objects+1)))



plt.imshow(data)


plt.autoscale(False)
plt.plot(xy[:, 1], xy[:, 0], 'ro')
plt.savefig('Created_Images/localmaximas.png', bbox_inches = 'tight')
```

```
C:\Users\Domin\AppData\Local\Temp\ipykernel_4368\3505596374.py:13: DeprecationWarning: P
lease use `maximum_filter` from the `scipy.ndimage` namespace, the `scipy.ndimage.filter
s` namespace is deprecated.
  data_max = filters.maximum_filter(data, neighborhood_size)
C:\Users\Domin\AppData\Local\Temp\ipykernel_4368\3505596374.py:15: DeprecationWarning: P
lease use `minimum_filter` from the `scipy.ndimage` namespace, the `scipy.ndimage.filter
s` namespace is deprecated.
  data_min = filters.minimum_filter(data, neighborhood_size)
```



By further processing the image it would be possible to remove the detected maximas on the channel (left and right) With the coordinated of the red dots, the position of the RBC can be detected faster than going through every single contour.

14.3 Alternative_shapes.ipynb

Another question of the Industrial Project 2 is, if other shapes than rectangulars can be detected in images

Circles

The opencv module offers a function called " hough circle" which is able to detect circles on blurred grayscale images

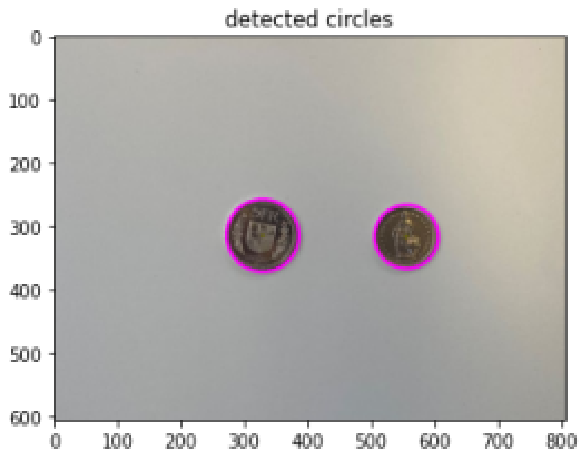
https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html

In [3]:

```python
from scipy.spatial import distance as dist
from imutils import perspective
from imutils import contours
from matplotlib import pyplot as plt

import numpy as np
import imutils
import cv2

src = cv2.imread("Images/Test_3.jpg")


gray = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
gray = cv2.medianBlur(gray, 5)
rows = gray.shape[0]
circles = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, 1, rows / 8,
                            param1=50, param2=80,
                            minRadius=1, maxRadius=100)

if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0, :]:
        center = (i[0], i[1])
        # circle center
        cv2.circle(src, center, 1, (0, 100, 100), 3)
        # circle outline
        radius = i[2]
        cv2.circle(src, center, radius, (255, 0, 255), 3)

result = cv2.circle(src, center, radius, (255, 0, 255), 3)


cv2.imshow("detected circles", src)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [4]:

```python
# show in notebook
show_result = cv2.cvtColor(result, cv2.COLOR_BGR2RGB)

plt.imshow(show_result)
plt.title("detected circles")
plt.show()
```



A limitation of this function is that it is only capable of detecting perfectly circular images.

Ellipses

Because the Red blood cells from the "RBC.png" image aren't perfectly circular, a method to detect ellipses is needed.

A way to draw ellipses is to fit them in the detected rectangles (minAreaRect) witht the "fitEllipse" method.

This method altough doesn't detect the ellipses persay rather it just fits in the rectangle. But the result show that some ellipses were fitted very well, when examing them visually.
https://docs.opencv.org/3.4/de/d62/tutorial_bounding_rotated_ellipses.html

In [12]:
```
from __future__ import print_function
import cv2
import numpy as np
import random as rng
import imutils

image_path_red = "Images/RBC.png"

image = cv2.imread(image_path_red)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (5, 5), 0)

edged = cv2.Canny(gray, 10, 30, None, 3)

cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)

minRect = [None]*len(cnts)
minEllipse = [None]*len(cnts)
for i, c in enumerate(cnts):
    if cv2.contourArea(c) <10:
        continue
    minRect[i] = cv2.minAreaRect(c)
    if c.shape[0] > 50:
        minEllipse[i] = cv2.fitEllipse(c)


drawing = np.zeros((edged.shape[0], edged.shape[1], 3), dtype=np.uint8)

for i, c in enumerate(cnts):
    if cv2.contourArea(c) <10:
            continue
        color = (rng.randint(0,256), rng.randint(0,256), rng.randint(0,256))
        # contour
        cv2.drawContours(image, cnts, i, color)
        # ellipse
        if c.shape[0] > 50:
            cv2.ellipse(image, minEllipse[i], color, 2)
        # rotated rectangle
        box = cv2.boxPoints(minRect[i])
        box = np.intp(box) #np.intp: Integer used for indexing (same as C ssize_t; normally
        cv2.drawContours(image, [box], 0, color)

        cv2.imshow("Original", image)
        cv2.waitKey(0)
cv2.destroyAllWindows()
```

Polygons

With the following code example, it is possible to detect any polygonial shapes. It is possible to adjust the threshold of how many sides the polygon should have.

In [5]:
```
# read the input image
img = cv2.imread('Images/polygons.png')

# convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# apply thresholding to convert the grayscale image to a binary image
ret,thresh = cv2.threshold(gray,50,255,0)

# find the contours
contours,hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
print("Number of contours detected:",len(contours))
for cnt in contours:
   approx = cv2.approxPolyDP(cnt, 0.01*cv2.arcLength(cnt, True), True)
   (x,y)=cnt[0,0]

   if len(approx) >= 10: #Threshold of how many sides the polygon should have
      img = cv2.drawContours(img, [approx], -1, (0,255,0), 3)
      cv2.putText(img, 'Polygon', (x, y),
cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)
cv2.imshow("Polygon", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
Number of contours detected: 3
```

These examples showed that it is possible to detect alternative shapes (circles, polygons) with OpenCV and also draw ellipses.