Antti Alamäki

Implementation of OpenCV in a Machine Vision System

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree programme in Electrical and Automation Engineering

Bachelors thesis

11.11.2017

Tekijä(t) Otsikko Sivumäärä Aika	Antti Alamäki OpenCV:n käyttöönotto osana konenäköjärjestelmää 45 sivua + 9 liitettä 20.11.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Sähkö- ja automaatiotekniikan tutkinto-ohjelma
Suuntautumisvaihtoehto	Automaatiotekniikka
Ohjaaja(t)	Lehtori Jari Savolainen

Tässä insinöörityössä perehdytään OpenCV:n käyttöön osana konenäköjärjestelmää ja pilottihanketta. Työssä selvitettiin tosielämän käyttötapausten ja ongelmatilanteiden kautta OpenCV:n suorituskykyä sekä muuntautumiskykyä Python- ja C++ toteutuksien avulla erilaisissa ympäristöissä osana konenäköjärjestelmää.

Työssä toteutettiin teknologiavertailu ja valinta eri käyttöympäristöjen sekä ohjelmistoversioiden välillä asiakkaiden vaatimukset ja tarpeet huomioiden. Nämä valinnat kyettiin perustelemaan toteuttamalla osana työtä toteuttamalla kolmeen eri käyttötapaukseen perustuvat ohjelmistoversiot kummallakin valituista kielistä. Ohjelmoinnin tarkoituksena oli myöskin pullokaulojen ja ongelmakohtien jäljittäminen sekä C++:n Python-laajennuksen testaus, näissä tehtävissä onnistuttiin kiitettävästi. Osana teknisiä valintaperusteita ja pullonkaulojen kartoitusta myös suoritettiin suorituskykyanalyysi eri ohjelmointikielillä tehtyjen toteutusten välillä joiden perusteella voitiin tehdä alustavia johtopäätöksiä suorituskyvyn suhteen. Testien ja ohjelmoinnin tulosten perusteella tehtyjä valintoja sekä suorituskykyanalyysin tuloksia esitellään tässä insinöörityössä salassapitosopimuksen sallimissa rajoissa.

Työn viimeisenä osana toteutettiin sensorijärjestelmän kalibrointiin käytettävä pienimuotoinen ohjelmakoodi Pythonilla jonka toimivuutta testattiin tehdasympäristössä. Testien avulla kyettiin varmistamaan kalibrointikoodin ongelmaton toiminta sekä soveltuvuus valittuun käyttötarkoitukseen ja ympäristöihin.

Avainsanat	OpenCV, konenäkö, Python, C++, suorituskykyanalyysi

Author(s) Title Number of Pages Date	Antti Alamäki Implementation of OpenCV in a Machine Vision System 45 pages + 9 appendices 20 November 2017
Degree	Bachelor of Engineering
Degree Programme	Degree programme in Electrical and Automation Engineering
Specialisation option	Automation Technology
Instructor(s)	Jari Savolainen, Senior Lecturer

In this thesis work, the general goal was to research and implement OpenCV into a machine vision as a part of a pilot study. The work was carried out by programming solutions for pre-selected real-world use-cases and problems using Python and C++ programming languages in different environments and setups as a part of a machine vision system.

A central part of this thesis work was a technology review and selection between different versions of software while taking into account client needs and requirements. These selections were further proven by programming three use-cases in both of the pre-selected programming languages. The second goal of the programming was to detect any complications or bottlenecks in the implementations, these tests were all successfuly executed and proven. As a part of the detection for bottlenecks and performance issues, all the three use-cases were benchmarked, on basis of these benchmarks it was possible to make some preliminary assumptions on the performance of the scripts. The principles of the technology selection and programming along with the benchmarking results are presented as a part of this thesis within permitations of the NDA.

As the last part of the work, a small calibration script was created in Python for calibrating the sensor system. The functionality of this script was tested in a real factory environment. With these tests, it was possible to verify the flawless functioning and suitability of the script in the selected usage and environments.

Keywords	OpenCV, machine vision, Python, C++, benchmarking

Contents

Abbreviations

CMOS	Complementary Metal Oxide Semiconductor. Technology commonly used in microprocessors. In machine vision systems used in form of a CMOS sensor.
Windows CE	Windows Community Edition. Edition of Microsoft Windows commonly used in industrial applications.
WHL	Python Wheel. File format commonly used for Python extensions.
PIP	Python Installation Package. Installation and packaging appliance for Python extensions.
ROI	Region of Interest. An area inside an image, used for inspection of a specific feature or set of features in machine vision appliances.
API	Application Programming Interface. Programming interface which enables communication and calls between different software applications.
JSON	JavaScript Object Notation. Data transfer format which is used for serialization of data, resembles the structure of a JavaScript object.

1 Introduction

In this thesis work, the general goal was to research and implement OpenCV into a machine vision system using Python and C++. The work was done as a part of a pilot study for capabilities of OpenCV under different environments and setups. The demand for a feasible solution is high on both client and developer / supplier side.

The work consisted of four adjacent parts. The first part was technology research, selection and setup of development and release environments while considering the client requirements within the limits of available technologies. Second part was arbitrary research and programming with different technologies and methods of implementation. The motive for the second phase was to give proof to clients about the capabilities of the new upcoming technology and to detect and tackle any obstacles on the way of adapting such technology. The tests were partially based of real-life use-cases, presentation of which is restricted with NDA. Therefore, these use-cases will be handled verbally with compensatory pictures that resemble the real use-case within limitations of the NDA agreement.

Third part of the thesis work was benchmarking the different technologies to detect frauds in performance between OpenCV implementation using Python and OpenCV using C++. One of the initial objectives was to test extending C++ with Python so that the OpenCV image processing implementation would be done using combination of OpenCV and Python and the Python script would be called from within C++. This approach, if successful and feasible performance-wise, would minimize integrators' need to touch the C++ source which in turn greatly reduces development, integration and maintenance costs.

The last part of the thesis work was integrating the developed scripts and algorithms into the actual machine vision system. No actual integration was performed, instead a calibrator script was written for calibrating the system. In this thesis work, it is handled as one use-case but is technically distinct from the others as this script was not benchmarked and was written only in Python.

2 About the Company and the Project

This thesis work was done for FocalSpec, a company specialized in machine vision systems and sensors for quality control in industrial environments. The company was founded in 2009 under VTT for research and development of line confocal sensors and scanners. Usually in the world of machine vision, lasers are used for measuring 3D dimensions, usually the Z-resolution of lasers is not enough for measuring for example surface roughness. The disadvantage of a laser technology in general is that multiple lasers are required for the purpose which complicates the setup and configuration and increases the expenses. Another main obstacle with lasers is limitation with materials; they are not suitable for example for transparent materials.

2.1 Technology Provided by FocalSpec

According to the company website [1.], the technology provided by FocalSpec is Line Confocal Imaging which provides an approach to the previously mentioned main shortfalls of existing technologies. Instead of using lasers, the system is based on an indirect light source (transmitter) and a receiver. In the system, white light emitting from a light source on the sensors' transmitter side is split into a continuous spectrum with thousands of wavelengths. Each wavelength is focused on a plane with certain distance from the sensor, the dominating wavelength is then reflected back to the sensors' receiver. On basis of this, CMOS array is formed, from which by means of software a 2D and 3D grayscale image is created. The basic idea of the system is displayed in figure 1.

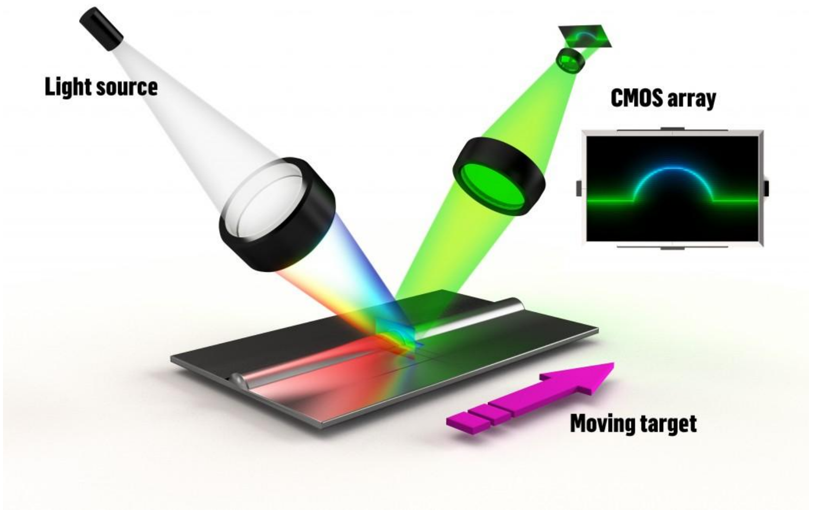


Figure 1. Illustration of the technology by FocalSpec from the FocalSpec website

Advantages of this sort of technology is that the system is much simpler in means of configuration than existing laser-based systems. Also, due to the nature of this technology, the system is capable of handling virtually any material and transparency or reflectance are not an issue.

3 Review of Available Technologies and Tools

In this project, the platform being used was Windows 10 Enterprise. Most of the client hardware is Windows-based with Windows 10 having a major share. Some of the clients may also run embedded Windows CE systems but these are not a major part and were not included in this thesis work.

Guidelines for the development were to adapt the latest possible software and technologies. With this, some precautions had to be taken as not all clients may be able to run for example the latest OpenCV distribution due to system limitations, this issue was tackled within the thesis work. At the time of this thesis work, the latest version of Python was 3.6.x and latest version of OpenCV was 3.x. It must be noted that to run the latest OpenCV with Python, the Python had to also be the latest 3.6.x series; older versions are not supported.

Other option for usage was to use older 2.4.x series version of OpenCV along with older Python.

3.1 Introduction to OpenCV

According Wikipedia [2.] and the OpenCV website [3.], OpenCV is an open source (BSD licence) library of functions, primarily focused on real time computer vision systems. The project was initially launched by Intel in 1999 with the first version out in 2000. The project was later supported by Willow Garage and is currently maintained by Itseez with the latest version being OpenCV 3 at the time of this thesis work.

The platform was originally written platform-independent using C++. The primary interface is still C++ though it maintains also an older, les comprehensive C interface as well. OpenCV has bindings (wrappers) available for MATLAB, Java and Python but also unofficial PHP wrappers are available. The bindings work in a way that the OpenCV is initiated and called within another language using wrapper functions that map to the functions in OpenCV core [4.]. Advantages of this approach is that the same core functionality is available for all supported languages without the need to rewrite the core for each language separately.

This provides a robust interface with the benefits of optimized performance from C++ and thus reduces delays caused by the actual language from which the OpenCV functions are called. This performance and the benchmarking results with conclusions will be described in the chapter 7 of this thesis about benchmarking the use-cases.

3.2 Differences Between Latest and Previous OpenCV and Python

During research, some major differences were detected in the approach of implementation of OpenCV between the latest and the previous version series. Also, Python syntax has seen some major changes in the latest version. These details were matters that had to be considered from client point of view as updating existing scripts written for older Python and OpenCV versions must be updated to the latest syntax which might be an obstacle for some clients.

Good example of a syntax change is for example the regular print statement in Python; in older versions, the print statement was written in format "**print 'test'**", the latest version requires parenthesis in the form "**print('test')**". Unfortunately changes on this level require a lot of manual work and testing to ensure full compatibility, this should be noted in implementation costs.

3.3 Windows Platform and Tools

As mentioned in the beginning of this chapter, the platform was Windows 10. For this platform, the best selection for C++ development is the Microsoft Visual Studio. As the platform was 64bit, also Visual Studio was selected the 64bit version. Selection of the platform and tools on this part was straight forward as the differences between 2015 and 2017 versions of Visual Studio are rather minimal from developer point of view so it was decided to go with the 2017 version.

4 Setup of Tools and Environments

This chapter consists of explanation of setting up the development, test and release environments. The chapter will also cover any issues and obstacles that were identified during the installation process along with analysis on possible impact on client.

4.1 Setup of Python and Extensions

The version of Python used in this work was Python 3.6 with the currently latest 3.3 version of OpenCV. During the research phase, different versions of OpenCV and Python were investigated but due to practical reasons it was decided to go with Python 3.6. Primary issue with Python 3.6 is that there is no official OpenCV support for any Python 3.x version [5.]. This makes the installation slightly tricky even though unofficial WHL files are available for the purpose.

Beside the Python 3.6, also Matplotlib and numpy (Numerical Python) were installed as the latest version of OpenCV heavily resides on numpy and a lot of the OpenCV functions using Python expect a numpy array as an input.

Table 1. Installed software for Python and OpenCV

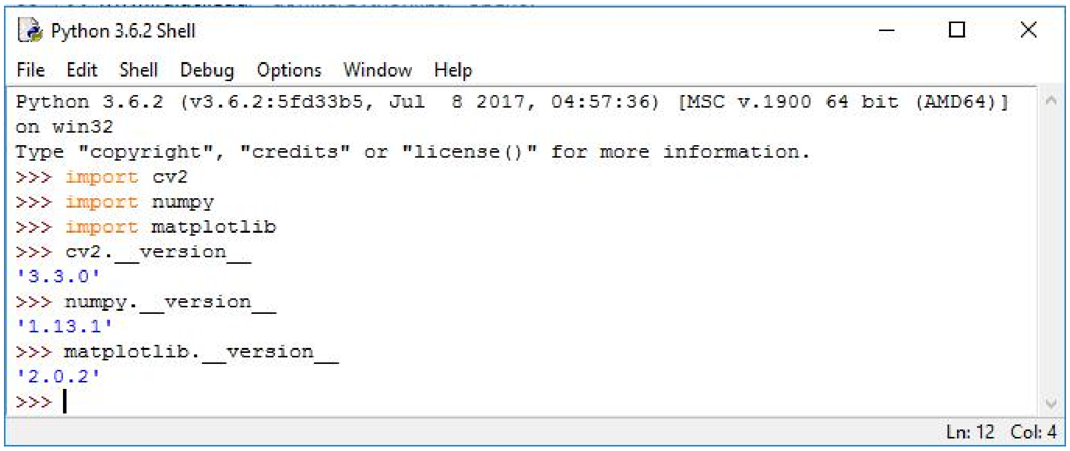
Software	Version	Source
Python	3.6	https://www.python.org/downloads/release/python-360
OpenCV	3.3.0	http://www.lfd.uci.edu/~gohlke/pythonlibs/#opencv
Matplotlib	2.0.2	http://www.lfd.uci.edu/~gohlke/pythonlibs/#matplotlib
Numpy	1.13.1	http://www .lfd.uci.edu/~gohlke/pythonlibs/#numpy

The installation of OpenCV, Matplotlib and Numpy was done via Pythons' package manager "pip". In the installation, the specific whl files were picked to ensure that the correct version of the applications got installed. By default, the Matplotlib that would have been installed was 2.1.1 but for compatibility reasons, older 2.0.2 was chosen. The installation commands are presented in the figure 2, the list of installed extensions is shown in the table 1.

```
pip install opencv_python-3.3.0+contrib-cp36-cp36m-win_amd64.whl

pip install numpy-1.13.1+mkl-cp36-cp36m-win_amd64.whl

pip install matplotlib-2.0.2-cp36-cp36m-win_amd64.whl
```

Figure 2. Installation commands for Python extensions

To test that that the setup was working as expected, Python provides a way to show the version of extensions. The extensions can be displayed in the manner presented in figure 3 by entering the commands in the Python shell.



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> import cv2
>>> import numpy
>>> import matplotlib
>>> cv2.__version__
'3.3.0'
>>> numpy.__version__
'1.13.1'
>>> matplotlib.__version__
'2.0.2'
>>> 
```

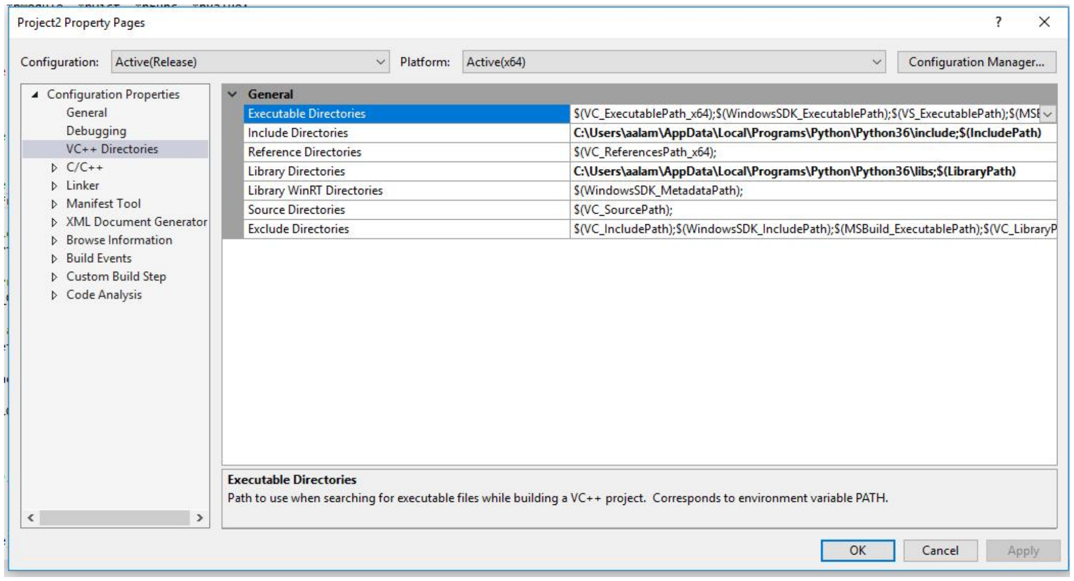
Figure 3. Testing that the Python extensions are correctly initiated

4.2 Configuration of Visual Studio 2017

As one of the goals in this thesis work was to test extending Python and how the script would operate, the C++ compiler (Visual Studio 2017) was setup to run Python scripts. To start a project, a new project was setup in Visual Studio 2017 (File → New Project → Visual C++ → General → Empty Project).

4.2.1 Python Support

After setting up the project, the python extensions and libraries (located under the Python installation folder) had to be added to the project as Visual Studio 2017 would not include these in the project by default. This was done by opening the Property pages for the project and updating the settings in the way demonstrated at Figure 4.



Configuration: Active(Release) Platform: Active(x64)

General

Executable Directories	$(VC_ExecutablePath_x64);$(WindowsSDK_ExecutablePath);$(VS_ExecutablePath);$(MSE
Include Directories	**C:\Users\aalam\AppData\Local\Programs\Python\Python36\include;$(IncludePath)**
Reference Directories	$(VC_ReferencesPath_x64);
Library Directories	**C:\Users\aalam\AppData\Local\Programs\Python\Python36\libs;$(LibraryPath)**
Library WinRT Directories	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);
Exclude Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);$(MSBuild_ExecutablePath);$(VC_LibraryP

Executable Directories
Path to use when searching for executable files while building a VC++ project. Corresponds to environment variable PATH.

Figure 4. Configuration of Python on Visual Studio

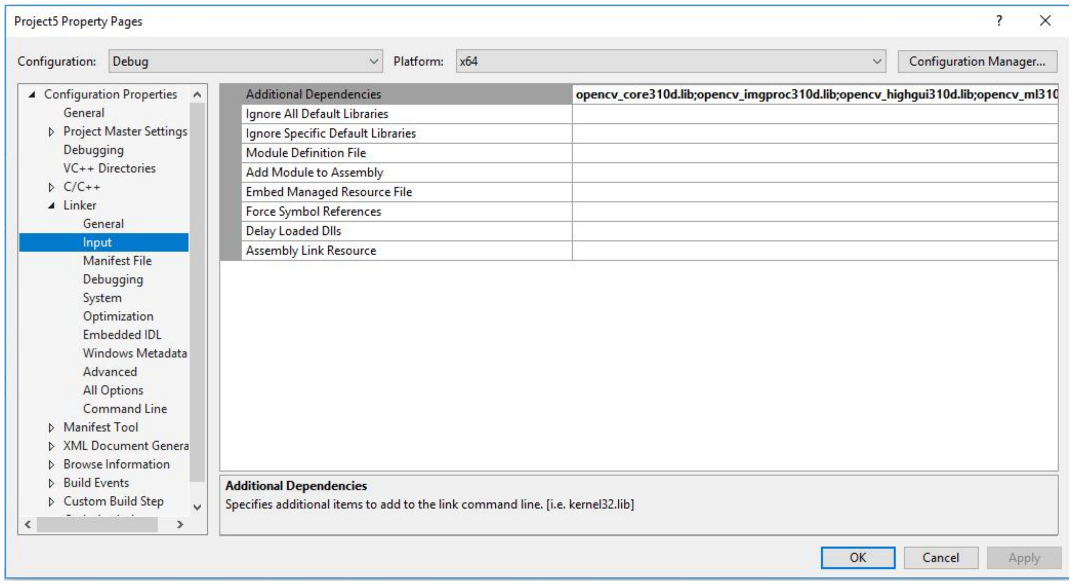
There were some complications in the installation; when running a test code to ensure that the Python support was working the way expected, the system gave an error about the python.h not being loaded. After some research, this proved to be a result of mismatch in the 32bit vs. 64bit environments; the Visual Studio was 32bit whereas Python was 64bit, therefore the libraries were not able to load correctly. This was corrected by adding a 64bit version of the header file after which the test code executed successfully.

4.2.2 OpenCV

As one of the initiatives of the work was to test the capabilities of C++ and OpenCV combo, OpenCV support had to be added. The 3.1 version of OpenCV on Visual Studio was installed using the NuGet package manager. While this approach has some disadvantages and obstacles, it is the simplest way of installation and ensures the best compliance with the projects. The main disadvantage is that the setup of OpenCV using NuGet requires some manual settings and OpenCV must be installed for each project separately. On the other hand, this resolution allows running different versions of OpenCV and other tools in different projects. In a client project, in real test and production environment, this might prove crucial even though the aim is to support primarily the latest versions of Python and OpenCV.

The settings that had to be done manually involve adding the Python libraries to the VC++ directories under Property Pages for each of projects. The path to modify was the same Library path as presented in the above illustration for setting up Python. Additionally, after some experimenting, it was found out that setting up also OpenCV was necessary to modify the linker paths; for some reason, NuGet package manager doesn't execute these changes automatically. Possibly because due to a NuGet bug, the Python libraries were still missing after the NuGet Installation of OpenCV. For this reason, the libraries had to be manually copied from the OpenCV installation that had been done for the standalone Python implementation. The linker paths are available thru the Property pages for the project under the Linker → Inputs tab, the setup is shown in figure 5.

It is important to note that the libraries for development and release compilations are different; the development libraries are marked for example opencv_core310d.dll whereas release version is opencv_core310.dll. They must be setup separately for product and development compilation as the development libraries contain some arbitrary code for debugging and they will not work properly seaminglessly for the release compilation. This arbitrary code also unnecessarily increases size of the built executable.


Project5 Property Pages

Configuration: Debug   Platform: x64   Configuration Manager...

Additional Dependencies	opencv_core310d.lib;opencv_imgproc310d.lib;opencv_highgui310d.lib;opencv_ml310
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line. [i.e. kernel32.lib]

Figure 5. Setup of OpenCV libraries in linker

4.2.3 Other Tools

The last tool that was setup at the initial stage was setting up a JSON writer/parser for serializing the result data from the script. For test purposes and ease of use, RapidJSON was chosen even though the company in principle uses another library. Also, due to the type of project that was being used in the setup, there were some issues while installing the parser via NuGet. The primary issue was that the packages had not been compiled to support an empty C++ project, instead they were meant for a Windows Desktop project. After some google search, a suitable version for the parser was found and installed.

4.3 Initial Tests with Python, OpenCV and Visual Studio

4.3.1 Python and Visual Studio

A tiny snippet of code was written with C++ and Python to test the functionality of the Python and Visual studio combo. The test initiated a couple of errors, the first of which was a missing python36.dll error. This was due to a bug in the Python installation and was fixed by copying the appropriate dll (dynamic link library) to C:/Windows/System32 folder.

Another issue that was identified at this stage was a bug in the way the OpenCV library was loaded thru the import statement in Python. The bug was due to a change that was implemented in the OpenCV 3.x, for a reason or other the import statement for OpenCV library failed. This was fixed by editing the cv.py file that comes with the whl (pythonwheel) file. This file defines the library and how it will be loaded.

4.3.2 Visual Studio and OpenCV

The testing of OpenCV and Visual Studio combo was conducted by a minimal test script, the code compiled without errors. However, when running the program file for the project, it failed with an issue that the system could not find the OpenCV dll files. While the solution of the issue might not be feasible in a client environment, the issue was solved by copying the dll's to the Windows system32 folder; a more correct way would have to add their location to the Windows path.

5 Principles and Theory of Image Handling

OpenCV provides with a large variety of functions for image handling and processing. While same goals can be achieved in number of ways, there are differences in performance and memory usage between the functions. In this project, the raw images produced by the sensor system can be up to gigabytes in size which sets a challenge for example for image loading times.

5.1 Loading the Image – Region of Interest (ROI)

One of the first things to do is to define, which part of the image is of interest. In OpenCV, this can be done by extracting a part of the master image using coordinate points, this method is called region of interest (ROI). Beside saving machine resources, this way also simplifies the code in a lot of cases as focus can be put on only a certain part of the image thus not having to deal with other areas of the bitmap.

Another thing with loading the images is that OpenCV provides several parameters to use when loading the images. In the FocalSpec sensor system, the produced 2D images are all 8bit grayscale images. Therefore, when loading the image, it makes no sense to load it as a colour image; instead, it will be loaded using a OpenCV parameter IMREAD_GRAYSCALE to load the image directly as a grayscale image. Listing 1 is an example of using the parameter in Python.

```
image = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
```

Listing 1. Example of image loading in OpenCV

5.2 Image Pre-processing

Pre-processing the images prior to feature detection has some impact on the performance and in some cases, impacts the feature detection. Among these methods are thresholding the image, applying filtering on the image to remove noise and as the last, methods of edge detection and image transformation. The guiding target in image pre-processing is to reduce the amount of results produced by the actual feature detection.

Image pre-processing is a slightly delicate subject and should be handled with thought because the worst-case scenario is a performance hit instead of performance gain. Applications and needs for image pre-processing vary greatly case by case and experience has proven that in many cases, the best results are achieved not by a single method of pre-processing but instead by a combination of different techniques.

While OpenCV provides a vast amount of different image processing tools, this chapter presents in theory the ones that were tested and used during this thesis.

5.2.1 Smoothing Images and Noise Removal

One of the first ways to reduce the amount of results during feature detection is to apply some filtering. According to the OpenCV website [6.] among the available filters, are the following:

- **Gaussian Blur** which uses the Gaussian Kernel for blurring operations
- **Bilateral filtering** is good for removing noise in the images while retaining the sharpness of edges, but disadvantage is that it is relatively slow compared to other filters
- **Median Blur** takes median of all the pixels under kernel area and central element is replaced with this median value; highly effective against salt-and-pepper noise in the images
- **Averaging** takes the average of all the pixels under kernel area and replaces the central element with its value; this is done by either the OpenCV function blur() or boxFilter() depending on the use-case

The upside of filtering is that it will reduce results from the feature detection. Downside on the other hand is that filtering will always impact the sharpness of the image and specially with Gaussian Blur this might become an issue. Filtering was tested in all the use-cases but eventually it was used only in the calibration application together with Canny edge detection, this application is further described in the chapter 6.4 of this thesis.

The primary reason for this resolution was that accuracy plays a critical role in this application and any filtering will impact sharpness of the image. It is possible to deal with excess feature detection results in other ways, for example filtering by size was used in all the cases that were handled during this thesis work. Instead of blurring the image, features were mostly detected from the raw image and other techniques like ellipse fitting and image moments were used for finding the centre of mass, angle of distortion and other things whenever required.

5.2.2 Thresholding

Thresholding is a method for altering the colours in an image. According to OpenCV tutorials [7.], typically, in computer vision the rule that applies is that if image is altered easier to be read by human, it will be easier to handle also by the algorithm.

The OpenCV threshold() function takes as input the image source/bitstream, threshold minimum value, maximum value and threshold style. On basis of these values, the function loops thru the image pixel by pixel and if the color in a pixel is larger than the thtreshold minimum value, the function will change the color of the pixel into that. If it is something else, color will be changed to something different.

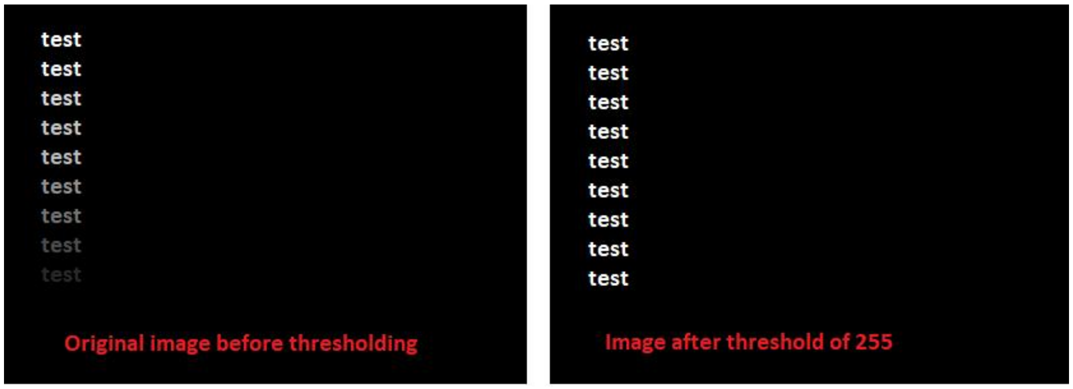


Figure 6. Thresholding an image – part 1

The illustration in figure 6 demonstrates what would happen if threshold was set to 0 with maximum value of 255. Value 255 stands for white color and as a result, the color of all the texts would be converted to white color.

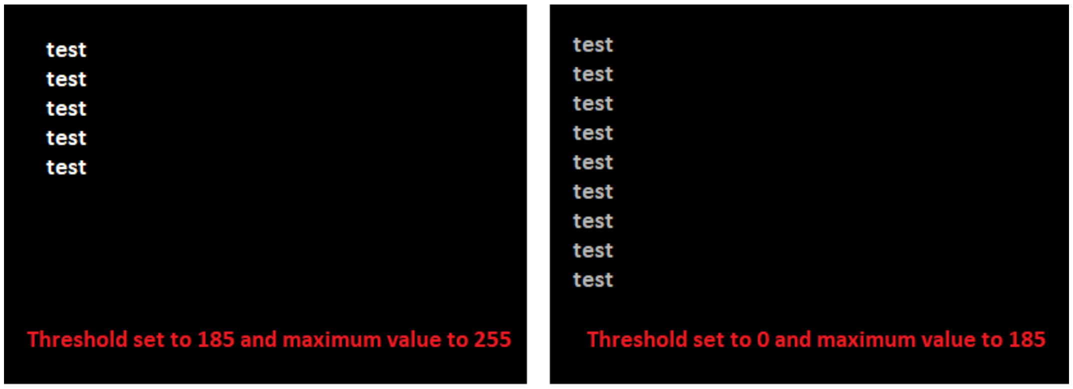

test
test
test
test
test
Threshold set to 185 and maximum value to 255
test
test
test
test
test
test
test
test
test
Threshold set to 0 and maximum value to 185

Figure 7. Thresholding an image - part 2

The first part of the illustration in figure 7 shows what would happen if threshold was set to 185 with maximum value 255; this would result that some of the text would disappear because their color is less than 185 and the rest would be replaced by 255 aka the white color. The second part of the figure 7 illustrates what would happen if threshold was set to 0 and maximum value to 185; the colour of all the texts in the image would be replaced by the colour 185.

OpenCV supports a number of methods for thresholding and the correct one should be selected on a case basis. The most commonly used method is binary thresholding and its inverse function. Other styles are threshold to zero along with its inverse function and threshold to trunc.

In all cases, providing a static threshold value might not be feasible. For example in a case where the lighting conditions change, a static threshold will result in alterations of the colour scheme, this will apply to both RGB and grayscale images. For this purpose, OpenCV provides a function called apativeThreshold() in which the user defines, by which means the threshold value is calculated. The available values are:

- **ADAPTIVE_THRESH_MEAN_C** threshold value is the mean of neighborhood area
- **ADAPTIVE_THRESH_GAUSSIAN_**C threshold value is the weighted sum of neighborhood values where weights are a gaussian window

Besides the above, apativeThreshold() also takes a second optional parameter which is the neighboring block size.

After extensive testing in each of the use-cases, it was decided that traditional thresholding was needed in only in the second use-case described in the paragraph 6.2

of this thesis. Some tests were made also with adaptiveThreshold() but it was decided that this was not eventually not required.

5.2.3 Edge Detection Methods

During this thesis work, also edge detection was tested to find edges around and within regions of interest. In OpenCV, a popular method for detecting edges is Canny edge detection, function canny [8.] which takes as parameters the threshold minimum and maximum values and kernel size. Canny should always be applied with threshold and filtering (blurring) of image as edge detection is sensitive to noise in the picture; good way for blurring is the blur() function described in chapter 5.2.2 of this thesis.

Canny edge detection proved a good solution in detection of single edges and was applied in the detection of edges for calibration, as described in the chapter 6.4 of this thesis about calibration of measurement applications.

5.3 Data Extraction

Beside image pre-processing, also the way the data is extracted from the image has impact on the memory usage. OpenCV provides multiple ways for data extraction, the most important of which are findContours and Hough Line Transform [9.]. Hough Line Transform is applicable for example when extracting continuous edges detected by Canny. On the other hand, if Canny detected only line fragments, the way for combining them together is thru findContours function. Both methods were tested and eventually findContours was used in all the use-cases.

Technically speaking, contours are boundaries of shapes with the same pixel intensity. The findContours() function takes several parameters which are defined on a case basis. The result output of findContours() is a vector of coordinate points for the edges of each contour feature. If wrongly set, this array can be much larger than necessary which in turn will impact performance. This matter received special attention along with other performance-related issues in this thesis work.

5.4 Ellipse Fitting and Detecting Contour Middle Points

One way to detect the centre of mass inside contours found from the images is ellipse fitting [10.]. This method calculates the ellipse that fits (in a least-squares sense) a set of 2D points best of all. To detect the set of 2D points inside which the ellipse is fitted, the typical way is to first draw a minimum area rectangle around the contour. Inside this minimum area rectangle, the ellipse is then fitted to get for example the centre point of the contour. The rectangle used for fitting can be either a direct rectangle using the OpenCV boundingRectangle() function or rotated rectangle using the OpenCV minAreaRect(), which one is used is dependent on the use case.

The following illustration in figure 8, taken from OpenCV documentation [11.], demonstrates the usage of rotated rectangle and the fitEllipse() function. The first part of the image shows the outer boundaries of the contour in green and the rotated rectangle in red. The second part of the illustration demonstrates fitting the ellipse inside the rotated rectangle.

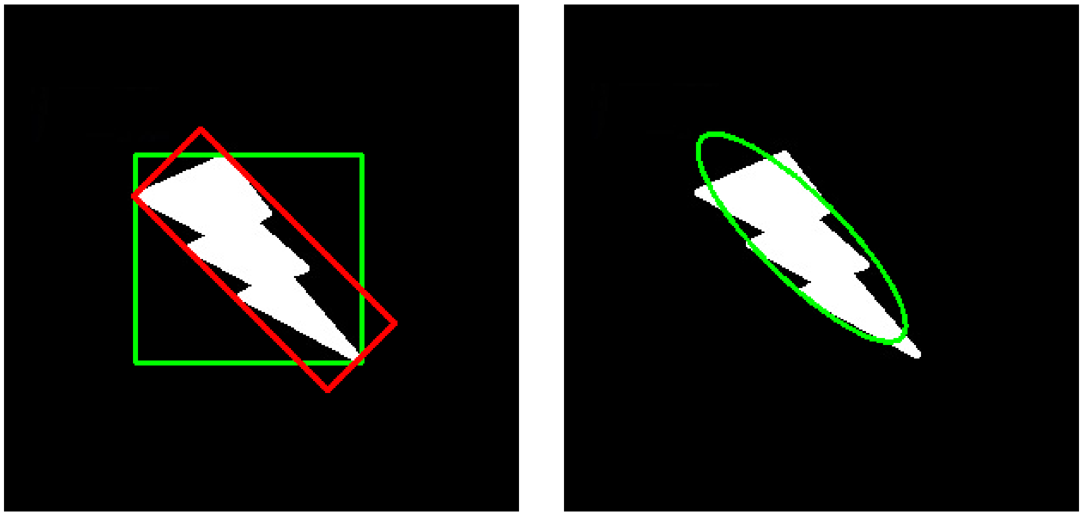


Figure 8. Rotated rectangle and fitEllipse

Another method for detecting the contour middle point is using the OpenCV moments() function; this was tested in one of the use cases. The function computes moments, up to the third order, of a vector shape or a rasterized shape and returns spatial moments, central moments and normalized central moments. The centre of mass can thus be calculated on their basis.

5.5 Methods for Extending C++ with Python

One of the key objectives in this thesis work was testing the extending of Python with C++. The de facto method for this is to use the Python API which is incorporated in the C++ application by including the Python.h file. The Python C++ extension provides with access to most aspects of the Python run-time system. Advantages of directly extending Python from within C++ are that by this way, the built-in object types and C++ library functions and system calls are accessible from within Python.

6 Use-cases Using Python and C++

As one of the primary initiatives of this thesis work was to detect obstacles and compare the performance of OpenCV with Python and C++, some use-cases were tested with. These use-cases totalled in three cases for the current use. This chapter will introduce the use cases in practise. All the use cases were implemented in both C++ and Python but as the methods are identical for both, this chapter will primarily focus on the Python versions. The complete codes with comments are available in the appendixes 1-9 of this thesis.

6.1 The First Use-case

The initiative in this first use case had three initiatives. The first was to introduce ourselves to the use of regions of interest, second was to find a coloured (black) area within an image using the fitEllipse function. The third part of this use-case was testing how the Python API for C++ works.

In this use-case, a dummy image was used instead of something from the camera and sensor system as the initiative was more focused on the embedding rather than real life feature detection.

6.1.1 Python: Detecting a Black Area Inside an Image

The first task, as in any of the use-cases, was to define the region of interest; the script both in C++ and Python was designed in a manner that the ROI can be input as parameter to the script; this method applied to all the use-cases handled during this thesis work. In the test case, the ROI should roughly outline the black area in the image. The ROI was extracted in 8bit grayscale to save memory. The original image and the extracted region of interest are presented in the following figure 9.

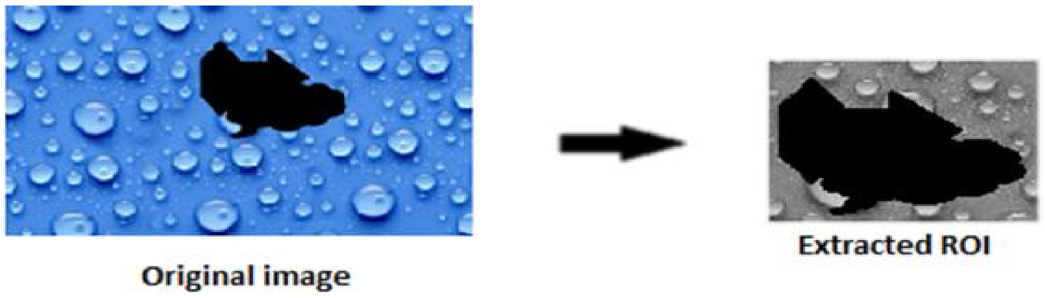


Figure 9. Test image in 1st use case

The first thing in all the use-cases is loading the image and defining the region of interest, an example of this is shown in the listing 2. The images are always loaded in 8bit grayscale mode to save memory. After this, the arguments are split into a tuple and finally the ROI is extracted from the image on their basis.

```python
# Load the image in grayscale
image = cv2.imread(image, cv2.IMREAD_GRAYSCALE)

roi = roi.strip('\'')

# create ROI from arguments and crop image
a, b, c, d = (int(x) for x in roi.split(','))
image = image[a: b, c: d]
```

Listing 2. Image load and extracting the ROI

The same operation in C++ is shown in the listing 3. The code loads the image in grayscale, splits the parameter by commas into an array and defines a Rect object with the values. Finally it uses the the Rect object to extract the ROI.

```
Mat image;

image = imread(argv[1], IMREAD_GRAYSCALE); // Read the file

if (!image.data) {
    cout << "Could not open or find the image" << std::endl;
    return -1;
}

// Split the roi argument into an array
string arg2 = argv[2];
std::stringstream iss(arg2);

// Roi is a string separated by commas so add dummy char for ,
char c;
int a[4];
iss >> a[0];

for (int i=1; i < 8; i++) {
    iss >> c >> a[i];
}

// Define the rectangle and crop the image
Rect rec(a[0], a[1], a[2], a[3]);

Mat roi = image(rec);
```

Listing 3. Loading image in C++ and OpenCV

The next thing is finding the contour. In this case, thresholding was not needed as the aim was to search for a pure black area on an 8bit grayscale image, therefore findContours() can be applied directly. In the findContours(), parameters RETR_LIST and CHAIN_APPROX_SIMPLE were used. The first parameter defines how the contours are fetched, RETR_LIST means that they are fetched as a list with no parent-child relationships [12.].

The second parameter defines which points are fetched. In this case, all the points around the contour are not needed so therefore to save memory, the code uses CHAIN_APPROX_SIMPLE which removes all the redundant points and compresses the contour. The result consists only of the corner points of the contour, this is demonstrated at the following listing 4.

```
# find contours in the image without tresholds
cnts = cv2.findContours(image, cv2.RETR_LIST,
                        cv2.CHAIN_APPROX_SIMPLE)
cnts = cnts[0]
```

Listing 4. Example of findContours()

After this, as shown in listing 5, the code loops thru the contours. In the code, len(c) is needed to make sure that the contour is a continuous line and not a line fragment. While looping thru the contours and find the bounding rectangle around the contour. The dimensions of the bounding rectangle are used to filter the contours as there are only certain ones that are of interest.

```python
# loop thru the contours
for c in cnts:
    if len(c) > 8:

        x, y, w, h = cv2.boundingRect(c)

        if w < 220:
```

Listing 5. Usage of boundingRect()

If the contour width fits the requirements, after the filtering the centre of the contour is calculated using moments as shown in listing 6. In reality this was not needed for anything in this case and was only done for test purposes.

```python
# compute the centre of the contour
M = cv2.moments(c)
cX = float(M["m10"] / M["m00"])
cY = float(M["m01"] / M["m00"])
```

Listing 6. Using moments to calculate centre of mass

Finally, as shown in listing 7, minAreaRect() is used to find the minimum area rectangle around the contour, the function returns a rotated rectangle which are converted to points in a numpy array and further used for the OpenCV fitEllipse() function. Finally the results are printed to the image using drawContours() and ellipse() functions. In the drawContours(), the (255,0,0) stands for a white color and the last parameter is the width

```python
# use minAreaRect to find the minimum rotated rectangle and
# convert the result to points
box = cv2.minAreaRect(c)
box = cv2.boxPoints(box)
box = np.array(box, dtype = "float")
box = perspective.order_points(box)

ellipse = cv2.fitEllipse(c)
cv2.drawContours(image,[box.astype("int")],-1, (255, 0, 0), 1)
```

Listing 7. Drawing the contour and ellipse

of the plot. To distinguish the drawings, the ellipse is printed in black (0, 255, 0) with 2px wide print. The final plot results are shown in the next figure 10.

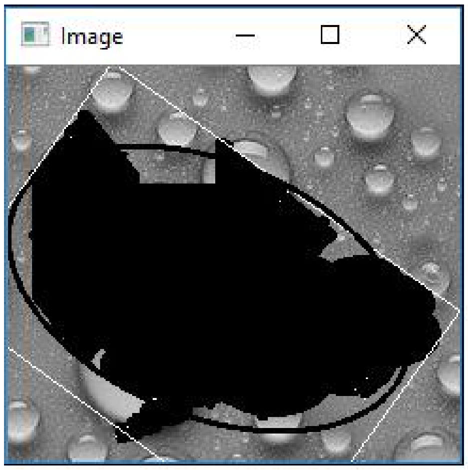


Figure 10 Output of the detected regions

6.1.2 C++: Testing the Python API

The second part of the first use-case was testing the extending of Python on C++. The actual methods for extending the Python in C++ are explained in the technical overview in the chapter 5.5 of this thesis work, this chapter will focus on the actual experiments and . As a general comment for the code, this was a test only to see if this works and did not put much effort on splitting and beautifying the code so the whole set of code is written under the main().

As seen in the code in the listing 8, the first thing on the list of tasks is to initiate the Python objects and check that all the necessary arguments for the C++ code are present (if not, exit the code). If successful, decode the parameters so they can be passed to the Python script.

```c
int main(int argc, char * argv[]) {

    PyObject * pName, * pModule, * pDict, * pFunc;
    PyObject * pArgs, * pValue, * t;

    int i;

    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile functionname
                        [args]\n");
        return 1;
    }

    Py_Initialize();

    // Decode the parameter and import
    pName = PyUnicode_DecodeFSDefault("detect_shapes");
    pModule = PyImport_Import(pName);
    Py_DECREF(pName);
```

Listing 8. Python initialization

If the paramenters are present, they are added to a string and some error checking is performed to verify that the parameters actually exist. This part worked fine all the way, also no issues with adding the parameters to the tuple. When this is done, the Python script will run and eventually the results are fetched from Python. This is demonstrated the following listing 9.

```c
// Push the two arguments (image name and roi) into a string
pFunc = PyObject_GetAttrString(pModule, argv[2]);

// Check that the function exists and if not, catch the error
if (pFunc && PyCallable_Check(pFunc)) {
    pArgs = PyTuple_New(argc - 3);
    for (i = 0; i < argc - 3; ++i) {
        pValue = PyBytes_FromString(argv[i + 3]);
        if (!pValue) {
            Py_DECREF(pArgs);
            Py_DECREF(pModule);
            fprintf(stderr, "Cannot convert argument\n");
            return 1;
        }

        // Add the arguments to tuple to be passed into Python
        PyTuple_SetItem(pArgs, i, pValue);
    }

// Fetch the return value
pValue = PyObject_CallObject(pFunc, pArgs);
```

Listing 9. Sample of returning arguments from Python

The result handling from Python is done in the way displayed at listing 10. The code just checks the value of pValue and if not null, stores the results from Python into a wharc_t from which it can be further parsed using the JSON parser of choice. In theory this is very simple, but practise proved unfortunately otherwise.

```
// If call successfull print results else fail
if (pValue != NULL) {
    const wchar_t * json = PyBytes_AsString(pValue);
}
else {
    Py_DECREF(pFunc);
    Py_DECREF(pModule);
    PyErr_Print();
    fprintf(stderr, "Call failed\n");
    return 1;
}
```

Listing 10. Reading arguments from Python

Returning the values from Python proved tricky with issues due to the data type of the return value. According to the Python API documentation, it should be possible to fetch any type of data from the Python script. After extensive tests, this proved not to be the case, the code compiled without errors but the code crashed with TypeError, meaning that the data type was incompatible.

This issue got solved but for some reason, if the return value from Python to C++ was tuple data, trying to parse the result with PyArg_ParseTuple() returns only data pointers but not the actual data. The type error continued with all datatypes except strings so it was decided to serialize the data in Python using JSON and pass the JSON string to C++. Despite extensive research, still more research is required to confirm whether these issues were due to the specific Python version in use or due to the Windows 10. Eventually in the course of this thesis work, the only way to safely return data was by serializing it into a string in the format of JSON in Python and then returning and deserializing the data in C++. This proved to be a feasible solution also for client setups and does not compromise the performance of the application.

6.1.3 Conclusions on the first use-case

The Python part worked perfectly as expected. However, when deciding the further course of action with this kind of extending, also concerns about integration costs and stability of the resolution in a client environment play a decisive role. Due to these factors, it was decided that despite the advantages of directly extending Python in C++, it is for the moment too unstable for production use. Upon this resolution, research on this part was stalled at the current state of the project.

Due to these issues, it was decided to change the approach so that the Python script writes results into a text file in JSON format. The Python script would then be launched with an execute call from within C++ and eventually the results in C++ would be read from the Python script's output file. This file must be either a physical file or database entry because reading from memory will not work due to the nature of Python; when the Python script exits, any memory allocations will be wiped at the instant.

6.2 The Second use-case

In the second use-case, the initiative was to detect the position of certain objects in a picture recorded by the sensor system. This included detection of the distance between two pins and detection of distance from the end of a pin to its base. The work was done in both C++ and Python for benchmarking purposes, the benchmarking process along with results and analysis will be covered in the chapter 7 of this thesis work.

Due to limitations by NDA, the original images from the sensor system will not be presented in this thesis work.

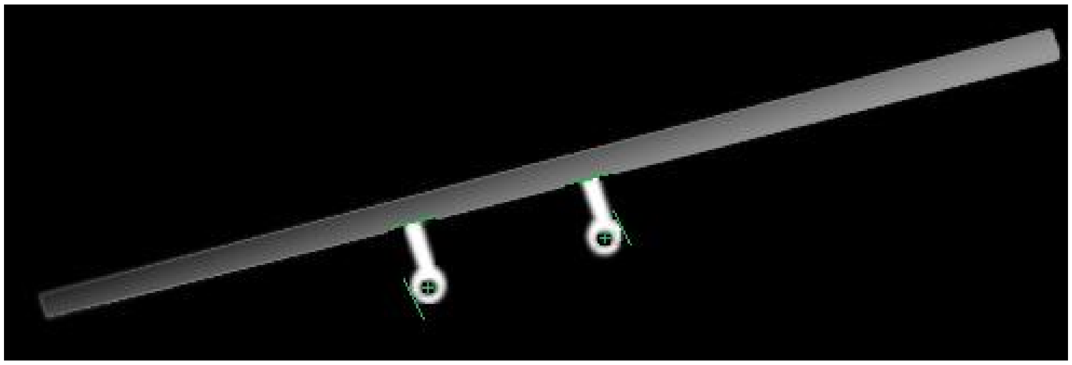


Figure 11. Points of measure in second use case

To illustrate what was measured, the points of measurement are presented in green in the above figure 11. The purpose was to measure the distance between the external edges of the pins and distance from the centre point of the hole in the pins to their baseline. One requirement for the measure is that the algorithm must work even if the position and angle of the measured artefact changes.

6.2.1 Preparations

After analysing the picture, first the region of interest is defined from within which all the measurements are done. The parameters for the script are the image source and the region of interest in coordinate points, extracting the region of interest is done identically to the way described in the beginning of previous chapter 6.1.1.

6.2.2 Contours

In this picture inside the region of interest, there are two different primary colours. The pins in the original image are shown as pure white whereas the hole in the middle of the pins is showing as black. In the solution for this problem, it is needed to find to find the xy-position of each of the holes in the image and the position of the white pin itself. Therefore, two distinct feature detections with findContours() function required for the task, one with threshold and one without.

The following code sample in listing 11 checks for the black holes in the pins, as a result the centre points for the holes in both pins are stored in a numpy (numerical python) array. Before storing the centre points, some filtering by size is required to define which contours are of interest, the len() at line 7 and the w > 100 at line 11 are for this purpose.

If the filter criteria are met, the code first calculates the minimum area rectangle for the contour, stores the values in a numpy array and uses the fitEllipse function to detect the centre of mass which will be the xy-coordinates for the centre of the hole in each pin.

```
cnts2 = cv2.findContours(image, cv2.RETR_LIST,
                         cv2.CHAIN_APPROX_NONE)

cnts2 = cnts2[0]

for i in cnts2:

    if len(i) > 8:
         x, y, w, h = cv2.boundingRect(i)

         # Filter contours by size
         if w > 100:

             # Fetch minimum area rectangle around
             # the contours and dump the results
             # into a numpy array
             box = cv2.minAreaRect(i)
             box = cv2.boxPoints(box)
             box = np.array(box, dtype = "float")
             cv2.drawContours(image, [box.astype
                             ("int")], - 1, (255,
                             0, 0), 1)

             # Use fitEllipse to fit an ellipse
             # inside the contour, dump the result
             # into a numpy array and fetch the
             # center point into a tuple
             ellipse = cv2.fitEllipse(i)
             (center, r, t) = ellipse

             if center1 == 0:
                 (center1, center2) = center, 0
             else :
                 (center1, center2) = center1,
                                      Center
```

Listing 11. Checking the size of contour, detecting the dimensions and ellipse fitting

To detect the white pins themselves, a second round of findContours() is implemented with a thresholded image as the source. The threshold statement used in this case was the following, it will change all colours above 127 (greyish colour) into 255 (pure white). A sample of the thresholding used in this use-case is presented in listing 12.

```
ret, tresh = cv2.threshold(image, 127, 255, 0)
```

Listing 12. Sample of thresholding in OpenCV

After applying a similar filtering as in the first findContours() round, the code uses a standard Python function for detecting the extreme points of the contour from a tuple of points. This method was selected because it is by far the easiest and least memory-consuming way for achieving the result.

```
# Get the extreme points of the contours and add them to a
# tuple
if isinstance(res1, int):
    (res1, res2, extTop1, extBot2) = getPoints(c), 0,

 tuple(c[c[: , : , 1].argmax()][0]), 0

    cent1 = np.array([float(center1[0]), float(center1[1])])
else :
    (res1, res2, extTop1, extBot2) = res1, getPoints(c),

 extTop1, tuple(c[c[: , : , 1].argmin()][0])
    cent2 = np.array([float(center2[0]), float(center2[1])])
```

Listing 13. Calculating the top left and top right points of the bounding rectangle

6.2.3 Calculating the Distances

In the snippet of code in listing 13, a custom function getPoints() is called. The function, demonstrated in listing 14, will return the Euclidean distance between the top right and the top left extreme points of the contour. The function first uses minAreaRect() function to fetch the minimum rectangle around the contour after which the system detects the corner points and sorts them. Finally, the function calculates the midpoint between the top left and top right points and returns the coordinates.

```
box = cv2.minAreaRect(c) box = cv2.boxPoints(box)
box = np.array(box, dtype = "float")
xSorted = box[np.argsort(box[: , 0]), : ]
leftMost = xSorted[: 2, : ]
rightMost = xSorted[2: , : ]
leftMost = leftMost[np.argsort(leftMost[: , 1]), : ]

(tl, bl) = leftMost D = dist.cdist(tl[np.newaxis], rightMost,
                                    "euclidean")[0]
(br, tr) = rightMost[np.argsort(D)[::-1], : ]

x = float((br[0] + tr[0]) / 2)
y = float((br[1] + tr[1]) / 2)

return np.array([x, y])
```

Listing 14. Calculating the Euclidean distance

This calculation is required to detect the midpoint of the edges of the pin at the position where the pin meets its base, due to the form of the pins this will be the shortest distance from the centre of the hole in the pin to its base. The following figure 12 describes what the function is trying to achieve.

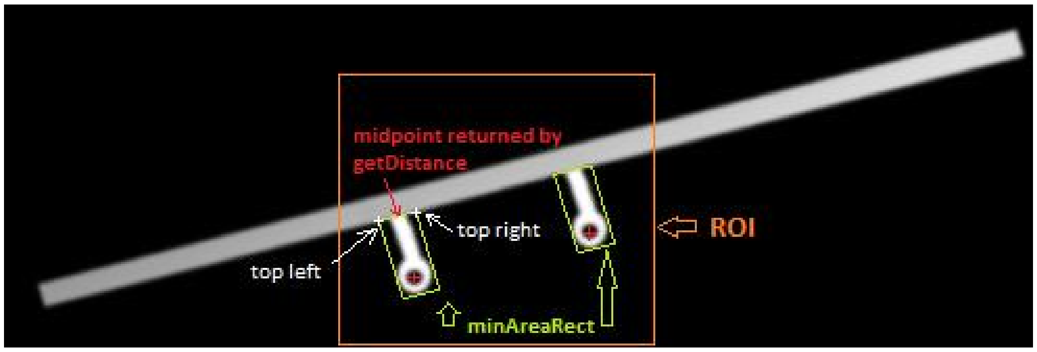


Figure 12 Output of getDistance

Finally, as displayed in the listing 15, the data is stored in a numpy array and the OpenCV norm() function was used to compute the required measures between the coordinate points.

```
    points3 = np.array([extTop1[0], extTop1[1]])
    points4 = np.array([extBot2[0], extBot2[1]])

    data["distance_1"] = cv2.norm(res1, cent1)
    data["distance_2"] = cv2.norm(res2, cent2)
    data["distance_between_objects"] = cv2.norm(points3, points4
```

Listing 15. Using OpenCV norm() function

6.2.4 Conclusions

The use-case required a lot of testing and attempts because one of the requirements was to get the distances with subpixel accuracy. This was slightly tricky because not all the OpenCV functions accept a floating point number as the entry. If floating point numbers are not used, some rounding will appear in between the values. The issue was much bigger in the C++ implementation than in the Python due to the nature of C++ as a heavily typified programming language.

Another challenge was to actually define the point against which the distance from the centre of the hole to the base of the pin is measured but this was sorted with some logic.

6.3 The Third use-case

In the third use-case, the objective was to detect the position of certain features and their distance from one other. The image that was used is presented in the following figure, the image itself is slightly manipulated due to issues in displaying it with the correct aspect ratio. It appears that even though the OpenCV namedWindow() function has a parameter WINDOW_KEEPRATIO to maintain aspect ratio, this function does not work correctly under windows environment and therefore the image appeared twisted on the screen when displayed using OpenCV.

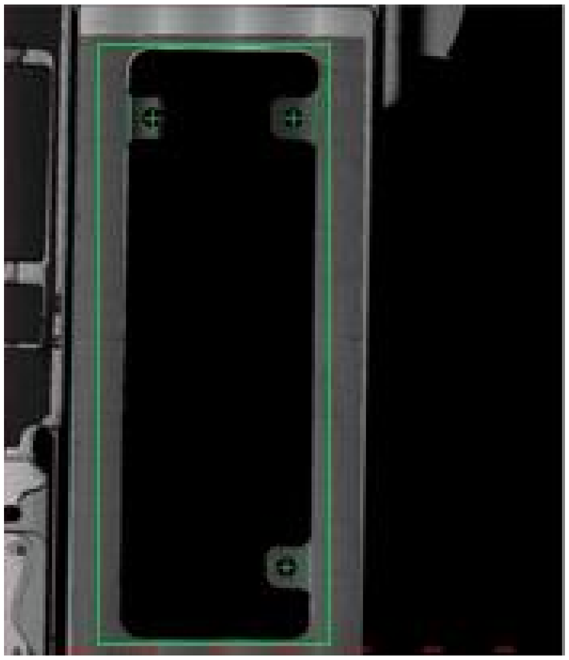


Figure 13. Image for the third use-case

From the image, the goal was to detect the distance between the holes marked with crosshair and their distance to the base of each of the features. The hole centres are presented in figure 13 along with the baseline which is presented by the short green line on either side of the hole. The developed algorithm should be such that it would take into account possible skewing of the artifact when it is recorded by the camera system, aka the object will not always be in the same position when measured.

6.3.1 Preparations

First task was to define the region of interest, there were two options with this. First option was to find three distinct regions of interest, one for each of the features. The second option would be to go with a single region of interest that contains all the three features.

The image was very large in size with width of roughly 14000px and height of around 1500px so memory consumption-wise it would have made sence to use three regions of interest. However, the main colour inside the single region of interest is black and the image is a 8bit grayscale image so a single larger region of interest will not have a huge performance impact. Therefore, to simplify the algorithm, decision was made to use a single region of interest. The region of interest is displayed as the large green rectangle in the image in figure 13.

6.3.2 Feature Detection

In this picture, there are total four contours that were of interest. Three of the contours are one for each of the holes and one large one is the solid black area inside the region of interest. The large contour is used to define the baseline from each of the hole centres to the base.

After extracting the region of interest, the features are fetched using the OpenCV findContours() function. In this case, thresholding is not required as the color marking the holes in the three features is completely black, same with the main color in the large contour. As a parameter, CHAIN_APPROX_NONE is needed, this is slightly memory consuming but this was done because all the points in the contour are required for future actions.

```
for x in roi.split(','))
    image = image[a: b, c: d]

    # Get centers  for contours; this will fetch the centres
    # for the black circles inside the ROI, no tresholding
    # needed as the color is pure black
    cnts2 = cv2.findContours(image, cv2.RETR_LIST,
                             cv2.CHAIN_APPROX_NONE)
    cnts2 = cnts2[0]
```

Listing 16. Finding the contours

After applying findContours(), the system loops thru the identified contour features. Some filtering is applied by the size of the contour to filter out the ones that are of no interest. The width of the contour is fetched using the OpenCV minAreaRect function, after which the code stores the centres of the contours into a pre-defined tuple. The centres equal to the centres of the holes and this can be done without ellipse fitting because the holes are an even shape (pure circle). Ellipse fitting would be used to detect the centre of mass of the contour but as the contours in this case are circle, the centre of mass is equal to the centre of the mininum area rectangle. At this point, the code also stores the possible skew angle of the contour, the angle is found from the output of the minAreaRect function.

After this operation, the code as shown in listing 17, calculates a point which is 500 coordinate points from the centre of contour in the direction provided by the angle from the minimum area rectangle. 500 pixels was selected to make sure that an intersection occurs. This value is then stored in a pre-defined tuple.

```
box = cv2.minAreaRect(i)

if box[1][0] > 100:

    if box[1][0] < 500:

        centers.insert(z, box[0])
        angle = box[2]

        # Calculate coordinates for the endpoint of the line
        # from centre of the hole
        line_x = box[0][0] + length * math.cos(angle *
                 math.pi / 180.0)
        line_y = box[0][1] + length * math.sin(angle *
                 math.pi / 180.0)

        points.insert(z, (line_x, line_y)) z = z + 1
```

Listing 17. Getting the other end point for vector

Eventually, the code detects also the large black area visible in figure 13 inside the region of interest. The corners of this black area are stored in a tuple using the OpenCV boxPoints function like shown in the sample of listing 18.

```
# Check for the large black area inside the roi
if box[1][0] > 2000:
    box2 = cv2.boxPoints(box)
    angle = box[2]
```

Listing 18. Bounding rectangle by the boxPoints() function

Once this operation is complete and the necessary values for the rest of the calculations are done, the code defines vectors from the centre of each of the holes to the previously calculated coordinate points from the centre point. For this purpose, the code introduces a function line() where the vector definition is computed, a sample of this method is shown in the listing 19. The code also computes a vector across the left and right edge of the large contour separately according to the ways shown in the following figure 14.

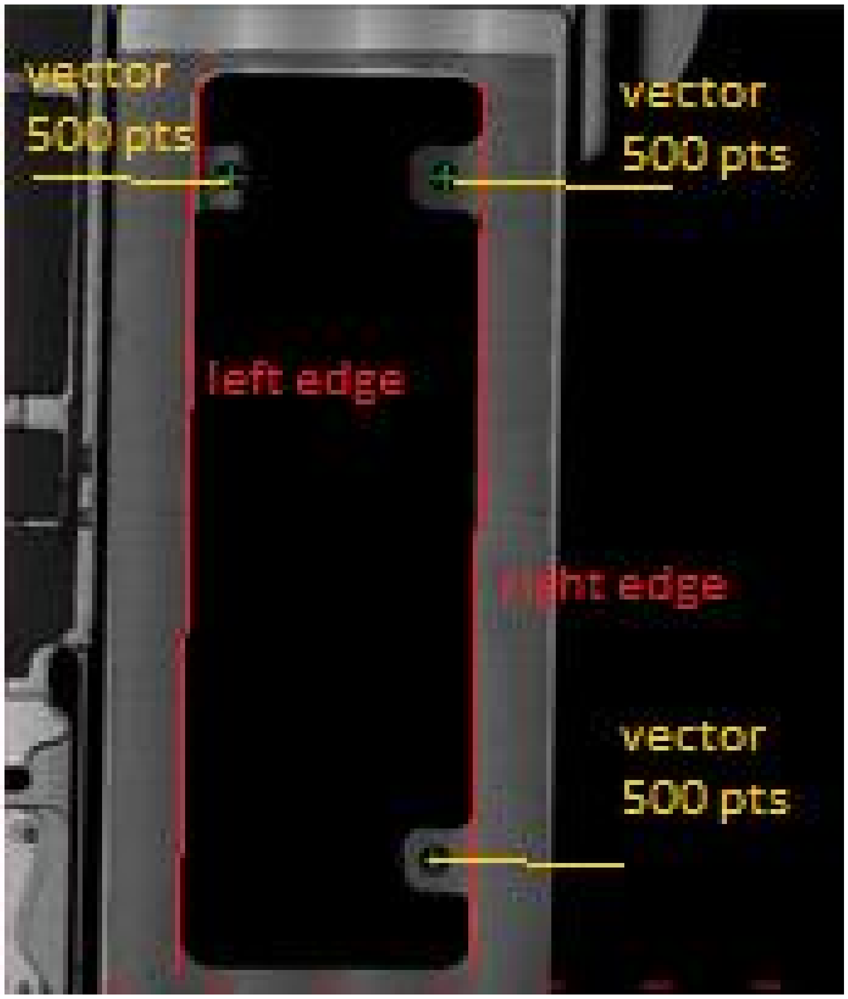


Figure 14 outline of the vectors in the image

```
# Define the equation for a line between points
def line(point1, point2):
    a = (point1[1] - point2[1])
    b = (point2[0] - point1[0])
    c = (point1[0] * point2[1] - point2[0] * point1[1])

    return a, b, -c  # return the coefs of line equation
```

Listing 19. Getting the line equations

After the line equation, the system makes a call to a function that computes the intersection point between the left/right edge and the vector from centre point of each of the holes. For this, there is a custom function intersect() which is shown in the next listing 20.

```
# Define the function for intersection
def intersect(line1, line2):

    # Get the main determinant from matrix[A1 B1, A2 B2]
    D = line1[0] * line2[1] - line1[1] * line2[0]

    # Dx from matrix[C1 B1, C2 B2]
    Dx = line1[2] * line2[1] - line1[1] * line2[2]

    # Dy from matrix[A1 C1, A2 C2]
    Dy = line1[0] * line2[2] - line1[2] * line2[0]

    if D != 0:

        # Calculate x and y
        x = Dx / D
        y = Dy / D
        return x, y

    else :
        return False
```

Listing 20. Function for calculating the intersection of two vectors

The results are then stored in a tuple which is written to JSON and stored in a file for future use. The complete Python and C++ source codes with comments are available in the appendix 5 and 6 of this thesis.

6.3.3 Conclusions

The case in this one involved a lot of thinking because of the requirements (skewing of the picture etc). The code was made as dynamic as possible and will take basically any amount of holes. Even more customization could have been made to increase the performance. Also the code still has small flaws, for example if the artifact is more than 90 degrees skewed, the program will fail. This was not taken into account in this case because in general it is correct to assume that if a artifact arrives so much skewed, the error is most likely elsewhere in the system.

Another point for customization could have been to make the code so dynamic that all the skew angles and intersection points would be passed dynamic to the appropriate line and intersection functions. However this was not within the scope of this use-case.

6.4 Fourth Use-case: Calibrator Using Python

The fourth and last use-case approaches a common issue in any sensor and machine vision system, the issue is called calibration. Calibration in this case means setting the lenses and other equipment in an aligned order so that they will give correct results. For ease of use and for consistency of results, it is good to have an automated system to handle the task.

The initiative in this use-case was to find the skew angle and distance in coordinate points for calibration error within the y-axis. To make the calibration as accurate as possible, the system returns the distance and angle with subpixel accuracy.

The code for the calibrator application was made only in Python as there was no need for benchmarking between the two approaches.

6.4.1 Feature search on the image

In this use-case, the images used for the tasks feature a standard. As seen in the example image in the figure 16, the black lines do not continue aligned. This means a calibration error which should be detected.

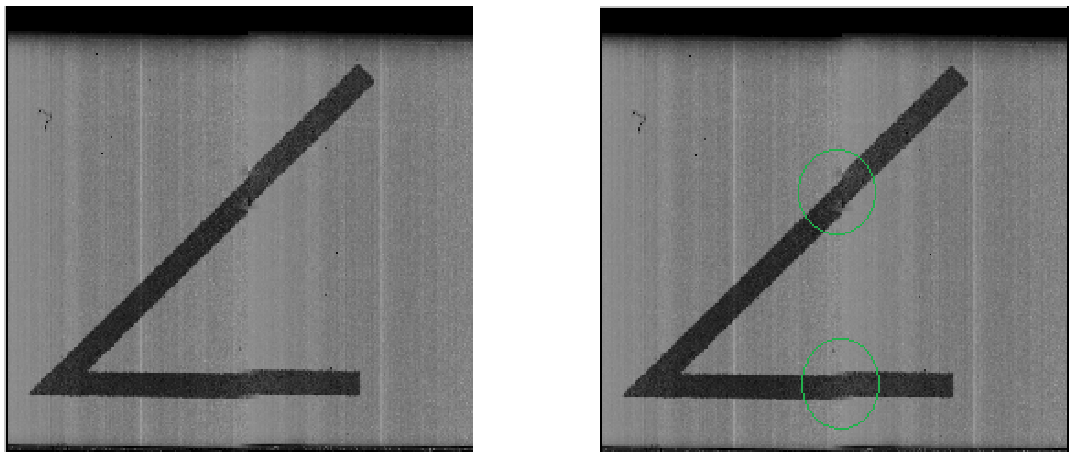


Figure 15. Sample image for calibration

6.4.2 Preparations

In this case, a couple of alternate approaches were considered to get the calibration error. Two of the approaches involved using two distinct regions of interest in different sections of the bottom horizontal black line shown in the image. From these two different regions of interest, it was possible to get either the extreme points of the contour or alternatively use Canny edge detection to find the top position of the black line in each of the regions of interest. The third option would have been to use a single region of interest along with Harris Corner Detection to detect the point where the alignment of the shape fails.

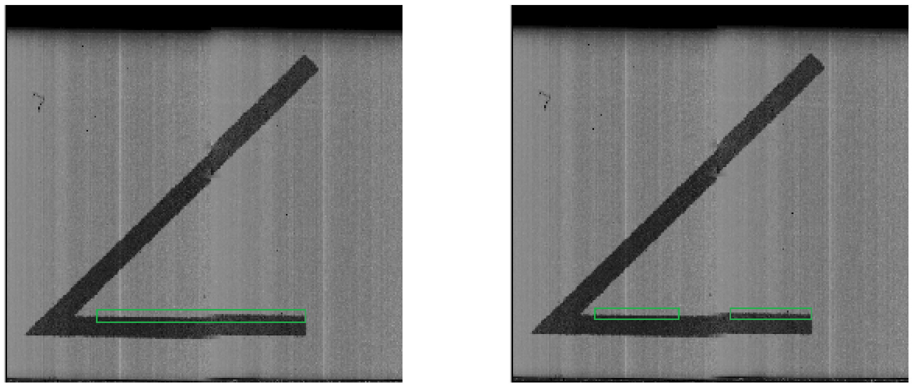


Figure 16. Approach with one or two ROI

In this case, the approach with two regions of interest was taken along with Canny edge detection and ellipse fitting.

```
# Create two ROIs and crop image
a, b, c, d = 550, 50, 2200, 600
e, f, g, h = 550, 50, 700, 600

roi1 = image[a: a + b, c: c + d]
roi2 = image[e: e + f, g: g + h]

# Function calls
dist1ellipse, dist1angle = calcDistance(roi1)
dist2ellipse, dist2angle = calcDistance(roi2)
```

Listing 21. Defining the ROI's and function calls

After the regions of interest are defined in the fashion shown in the above listing 21, the code calls for a custom function calcDistance() which returns the position of the edge of the black area in pixels.

6.4.3 Feature Detection

Canny as a function uses convolution to detect the edges and thus works the best after some noise reduction from the image. After some testing, the correct function for blurring seemed to be the OpenCV blur() function. This function and Gaussian Blur were tested with different kernel sizes 1x1, 3x3, 5x5 and 7x7.The first three produced the same outcome whereas the 7x7 blurred the image too much. Some thresholding was also tested with different kernel sizes in the blurring with no impact in the outcome. In OpenCV, Canny is implemented in the fashion demonstrated at the following listing 22.

```
# Blur some of the noise out of the image, kernel 3x3
    blur = cv2.blur(roi, (3, 3), 0)

    # Use Canny to detect the edges
    edges = cv2.Canny(blur, 50, 200, 1)
```

Listing 22. Example of Canny usage

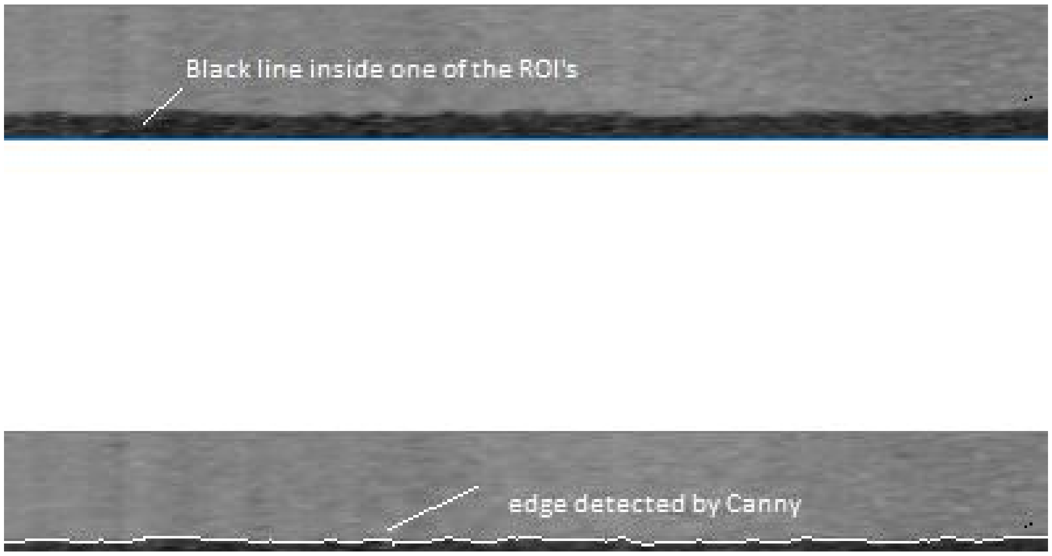


Figure 17. Edge detection results by Canny

As seen in the above illustration in figure 17 of the original region of interest and the Canny output, the edge detected is not a continuous line. Therefore, the appropriate method to extract the edges is by findContours(). Another way would have been via Hough Line Transform, but that works only for solid lines.

Results of the findContours are filtered by size and position so that further operations are applied only to the contours that fit the criteria. For the filtering, a minimum area rectangle is fitted around the contour after which the system uses fitEllipse() function to fit an ellipse inside the rectangle. From the output of fitEllipse(), the system returns the centre of mass for the contour along with the angle of the fitted ellipse. The following figure 18 is a screenshot of the output of ellipse fitting function.

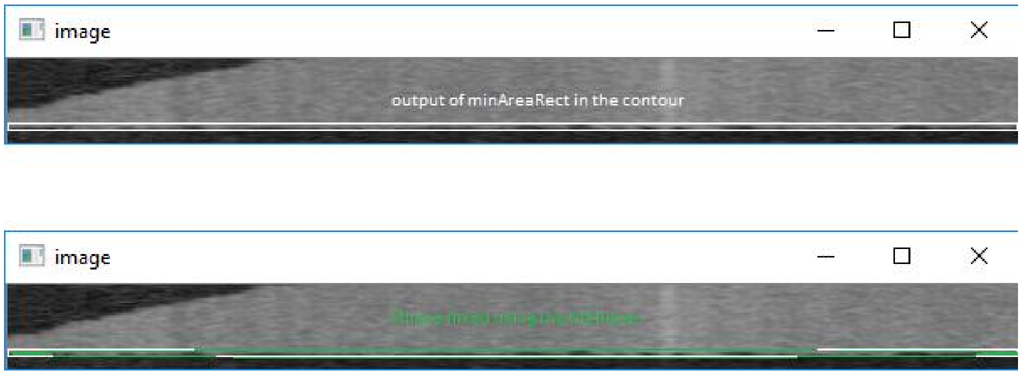
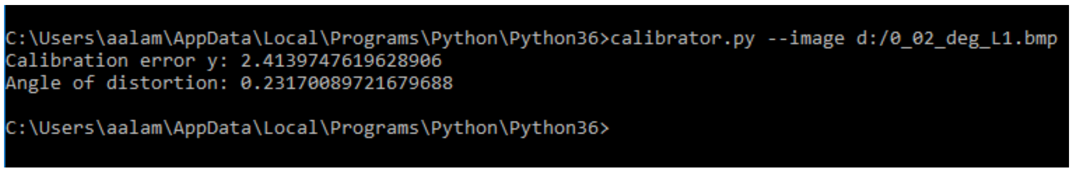


Figure 18. Output of minAreaRect and fitEllipse functions

Eventually, the computed results are returned and displayed on-screen. The output in the figure 19 is the outcome of calculations on the sample image presented in figure 15.

```python
# Function calls
dist1ellipse, dist1angle = calcDistance(roi1)
dist2ellipse, dist2angle = calcDistance(roi2)

# Print the calibration error on y - axis
print("Calibration error y:", dist2ellipse[0][1] -
      dist1ellipse[0][1])

# Print the average angle of distortion
print("Angle of distortion:", ((dist1angle - 90) +
     (dist2angle - 90)) / 2)
```

Listing 23. Calculations and output of the calibrator



```
C:\Users\aalam\AppData\Local\Programs\Python\Python36>calibrator.py --image d:/0_02_deg_L1.bmp
Calibration error y: 2.4139747619628906
Angle of distortion: 0.23170089721679688

C:\Users\aalam\AppData\Local\Programs\Python\Python36>
```

Figure 19. Detected calibration error and angle of distortion

7 Benchmarking the Scripts

The initiative in benchmarking was to find differences in performance between OpenCV using Python and similar application using C++. The benchmarks were performed for the second and third use-cases presented in the chapters 5 and 6 of this thesis.

The benchmark results are available in graphical form in the appendixes 8 and 9.

7.1 Executing the Benchmarks

In a windows environment, there are multiple ways for continuously running an executable or a script file, one way is the windows tasks and the other a batch script. In this case, the latter method was chosen for platform-independency reasons, also the batch script is relatively simple to maintain.

The benchmark was executed for both warm and cold cache in both Python and C++. The batch script for Python and C++ are identical with the only difference being the script call and the file output name. In both cases, the script will first loop ten times and run the executable or script with five second intervals to simulate a cold cache and then ten times with one second interval to simulate a warm cache, this is demonstrated at the following sample of the script in the listing 24.

```
@
echo off set loop = 0: loop
if "%loop%" == "1"@
echo Running batch benchmark at 1 second interval(cold cache)
     >> benchmark_results_c Project4 d: /roi.bmp
     500,350,1000,500
Timeout / t 5 set / a loop = % loop % +1
if "%loop%" == "10"
set loop2 = 0
if "%loop%" == "10"
goto loop2 goto loop: loop2
if "%loop2%" == "1"@
echo Running batch benchmark at 5 second interval(warm cache)
     >> benchmark_results_c Project4 d: /roi.bmp
     500,350,1000,500
timeout / t 1 set / a loop2 = % loop2 % +1
if "%loop2%" == "10"
goto next goto loop2: next echo Benchmark finished
```

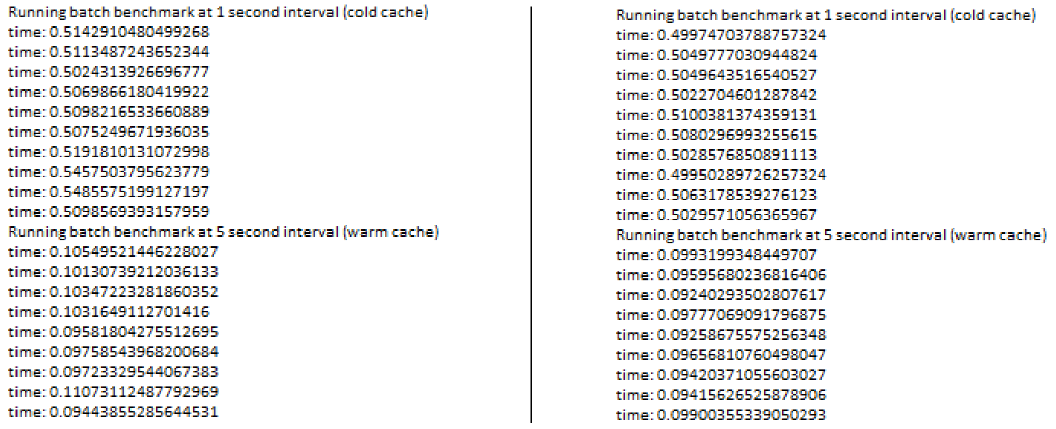
Listing 24. Sample of the benchmark script

7.2 Benchmark Results and Analysis

In the following, there is some analysis on the performance of the different scripts. As a note, all the benchmarks were executed multiple times at different hours to get a realistic impression on the performance. Despite this, the variations in the benchmark results at different times were very slight and they should be treated with some caution because also any background processes running on the system have an impact on performance, especially considering that the tests were made on a Windows platform.

7.2.1 The Second Use-case

In the second use case, not a great deal of effort was put in optimization of the code. The benchmark results for the Python script looked like what is shown in the figure 20. The first column is taken with the image load and the second one with it included.

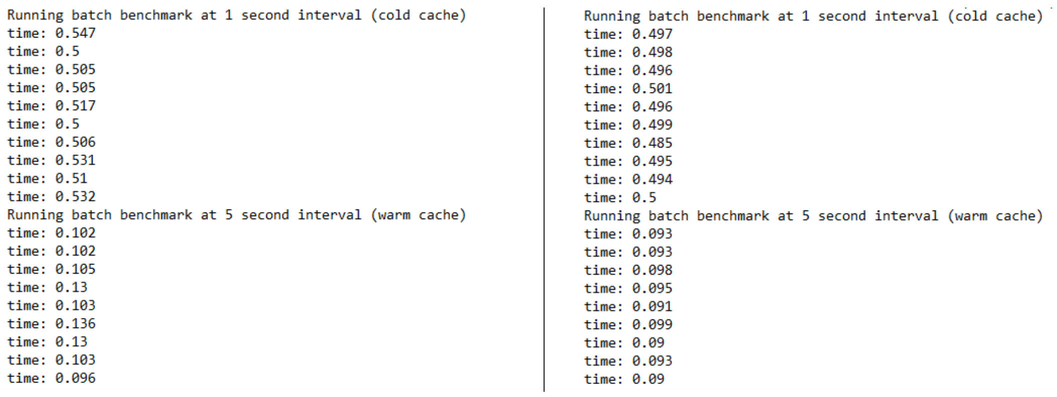


```
Running batch benchmark at 1 second interval (cold cache)
time: 0.5142910480499268
time: 0.5113487243652344
time: 0.5024313926696777
time: 0.5069866180419922
time: 0.5098216533660889
time: 0.5075249671936035
time: 0.5191810131072998
time: 0.5457503795623779
time: 0.5485575199127197
time: 0.5098569393157959
Running batch benchmark at 5 second interval (warm cache)
time: 0.10549521446228027
time: 0.10130739212036133
time: 0.10347223281860352
time: 0.1031649112701416
time: 0.09581804275512695
time: 0.09758543968200684
time: 0.09723329544067383
time: 0.11073112487792969
time: 0.09443855285644531
```

```
Running batch benchmark at 1 second interval (cold cache)
time: 0.49974703788757324
time: 0.5049777030944824
time: 0.504964351654052７
time: 0.5022704601287842
time: 0.5100381374359131
time: 0.5080296993255615
time: 0.502857685089111３
time: 0.49950289726257324
time: 0.5063178539276123
time: 0.5029571056365967
Running batch benchmark at 5 second interval (warm cache)
time: 0.0993199348449707
time: 0.09595680236816406
time: 0.09240293502807617
time: 0.09777069091796875
time: 0.09258675575256348
time: 0.09656810760498047
time: 0.09420371055603027
time: 0.09415626525878906
time: 0.09900355339050293
```

Figure 20. Benchmark results with Python

The C++ benchmarks were executed with the release version of the use case because in the debug version, there is some extra code which the complier adds to the executa-ble.



```
Running batch benchmark at 1 second interval (cold cache)
time: 0.547
time: 0.5
time: 0.505
time: 0.505
time: 0.517
time: 0.5
time: 0.506
time: 0.531
time: 0.51
time: 0.532
Running batch benchmark at 5 second interval (warm cache)
time: 0.102
time: 0.102
time: 0.105
time: 0.13
time: 0.103
time: 0.136
time: 0.13
time: 0.103
time: 0.096
```

```
Running batch benchmark at 1 second interval (cold cache)
time: 0.497
time: 0.498
time: 0.496
time: 0.501
time: 0.496
time: 0.499
time: 0.485
time: 0.495
time: 0.494
time: 0.5
Running batch benchmark at 5 second interval (warm cache)
time: 0.093
time: 0.093
time: 0.098
time: 0.095
time: 0.091
time: 0.099
time: 0.09
time: 0.093
time: 0.09
```

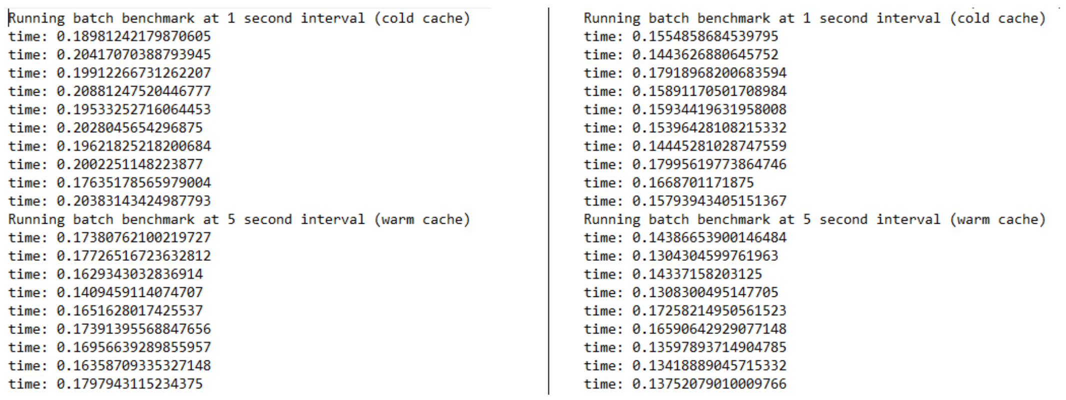
Figure 21. Benchmark results with C++

The benchmark results shown in figure 21 follow quite much the expected pattern. Due to the relatively small size of the image (around 2000x1000px), not even the image load had much impact on the performance. Considering that OpenCV is written in C++ and Python provides only a wrapper for the functions, there should not be much difference in performance. In this use-case, very little calculation takes place in Python itself, so the main overhead should come from the image load time.

Looking at the results between the warm versus cold cache, both Python and C++ seem to react to them with similar pattern. With warm cache, the execution time is much faster which supports the theory about the image load increasing the execution time significantly. With warm cache, the image is preloaded into memory which is clearly visible in the results on both implementations.

7.2.2 The Third Use-case

In the third use-case, much more effort was put on the performance than in the second use-case and this is clearly visible in the benchmark results. In this case, while the resolution of the image remained the same as in the second use-case, the picture that was used was much larger in size (around 14000x1500px). The following illustration demonstrates the performance on Python with both image load included and disincluded.



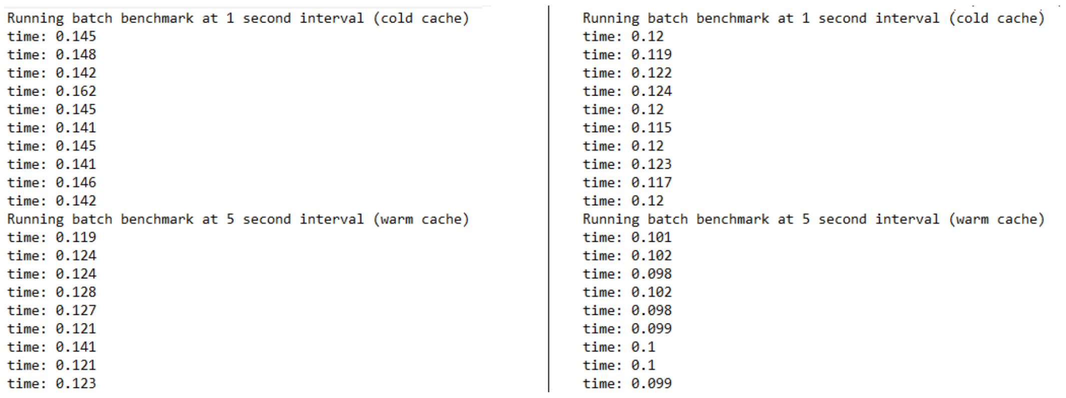
```
Running batch benchmark at 1 second interval (cold cache)
time: 0.18981242179870605
time: 0.20417070388793945
time: 0.19912266731262207
time: 0.20881247520446777
time: 0.19533252716064453
time: 0.2028045654296875
time: 0.19621825218200684
time: 0.2002251148223877
time: 0.17635178565979004
time: 0.20383143424987793
Running batch benchmark at 5 second interval (warm cache)
time: 0.17380762100219727
time: 0.17726516723632812
time: 0.1629343032836914
time: 0.1409459114074707
time: 0.1651628017425537
time: 0.17391395568847656
time: 0.16956639289855957
time: 0.16358709335327148
time: 0.1797943115234375
```

```
Running batch benchmark at 1 second interval (cold cache)
time: 0.1554858684539795
time: 0.1443626880645752
time: 0.17918968200683594
time: 0.15891170501708984
time: 0.15934419631958008
time: 0.15396428108215332
time: 0.14445281028747559
time: 0.17995619773864746
time: 0.1668701171875
time: 0.15793943405151367
Running batch benchmark at 5 second interval (warm cache)
time: 0.14386653900146484
time: 0.1304304599761963
time: 0.14337158203125
time: 0.1308300495147705
time: 0.17258214950561523
time: 0.16590642929077148
time: 0.13597893714904785
time: 0.13418889045715332
time: 0.13752079010009766
```

Figure 22. Benchmark results using Python

Looking at the results displayed in figure 22, the performance of the script is on a very high level, this can be judged by a much faster execution time as compared to the second use-case where not nearly as much computing was performed in Python. Surprisingly, the difference between warm and cold cache with and without the image load is very slight.

For a comparison, the next figure 23 demonstrates the performance on C++. As can be seen, the C++ implementation has a slight advantage in performance over the version in Python.



```
Running batch benchmark at 1 second interval (cold cache)
time: 0.145
time: 0.148
time: 0.142
time: 0.162
time: 0.145
time: 0.141
time: 0.145
time: 0.141
time: 0.146
time: 0.142
Running batch benchmark at 5 second interval (warm cache)
time: 0.119
time: 0.124
time: 0.124
time: 0.128
time: 0.127
time: 0.121
time: 0.141
time: 0.121
time: 0.123
```

```
Running batch benchmark at 1 second interval (cold cache)
time: 0.12
time: 0.119
time: 0.122
time: 0.124
time: 0.12
time: 0.115
time: 0.12
time: 0.123
time: 0.117
time: 0.12
Running batch benchmark at 5 second interval (warm cache)
time: 0.101
time: 0.102
time: 0.098
time: 0.102
time: 0.098
time: 0.099
time: 0.1
time: 0.1
time: 0.099
```

Figure 23 Benchmark results using C++

Surprisingly, in C++ warming the cache seems to have slightly more impact on the performance with both image load and without included. In theory, considering the nature of C++, the assumption would have been that C++ is faster which in this use-case doesn't seem to be the case. The conclusion was confirmed by multiple test runs in different environments.

7.3 Overall Conclusions on the Benchmarks

The performance of C++ and Python seem to have very slight differences as proven by both use-cases. While this is a smallish sample of all tests and does not in any way resemble a real-life production environment, the results do give some insight on what can be expected in a real-life case. On the other hand, looking at the performance, even though the differences are very slight between the programming languages and some consideration must be given over the results, even the small differences might in some real-life cases prove to be a bottleneck for production.

Time will tell whether the performance is good enough or not but looking at the prerequisites between the second and the third use-case, the meaning of optimization is clearly proven. Even the image size does not have relatively as much importance as optimization – in the third use-case, the image was many times larger than in the send one and yet despite the heavier calculations the third one was multiple times faster.

8 References

1. FocalSpec. Technology [online]. FocalSpec Oy. 2017.
   URL: https://www.focalspec.com/tehcnology. Accessed 21 September 2017.

2. Wikipedia.org. OpenCV [online]. Wikipedia. October 2017.
   URL: https://en.wikipedia.org/wiki/OpenCV. Accessed 22 October 2017.

3. OpenCV.org. About [online]. OpenCV community. 2017.
   URL: http://www.opencv.org/about.html. Accessed 27 September 2017.

4. OpenCV.org. How OpenCV-Python Bindings Works [online]. OpenCV community. September 2017.
   URL: http://docs.opencv.org/3.2.0/da/d49/tutorial_py_bindings_basics.html. Accessed 30 September 2017.

5. Opencv.org. Package list. [online]. OpenCV Python community. September 2017.
   URL: https://pypi.python.org/pypi/opencv-python/3.3.0.10. Accessed 30 September 2017.

6. OpenCV.org. Smoothing images [online]. OpenCV development team. November 2014.
   URL: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_filtering/py_filtering.html#filtering. Accessed 5 October 2017.

7. Mallick S. OpenCV Threshold (Python , C++) [online]. Big Vision LLC. November 2015.
   URL: https://www.learnopencv.com/opencv-threshold-python-cpp. Accessed 5 October 2017.

8. OpenCV.org. Canny Edge Detection [online]. OpenCV development team. 2017.
   URL: https://docs.opencv.org/3.1.0/da/d22/tutorial_py_canny.html. Accessed 6 October 2017.

9. OpenCV.org. Hough Line Transform [online]. OpenCV development team. November 2014.
URL: https://docs.opencv.org/3.0/beta/doc/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html. Accessed 20 October 2017.

10. OpenCV.org. Structural analysis and shape descriptors [online]. OpenCV development team. November 2014.
11. URL: https://docs.opencv.org/3.0-beta/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#fitellipse. Accessed 10 October 2017.

12. OpenCV.org. Contour Features [online]. OpenCV development team. 2017.
URL: https://docs.opencv.org/3.1.0/dd/d49/tutorial_py_contour_features.html. Accessed 11 October 2017.

13. OpenCV documentation. Contours: Getting Started [online]. OpenCV development team. 2017.
URL: https://docs.opencv.org/3.1.0/d4/d73/tutorial_py_contours_begin.html. Accessed 11 October 2017.

C++ source for the first use-case

```
# include < Python.h >

int main(int argc, char * argv[]) {

    PyObject * pName, * pModule, * pDict, * pFunc;
    PyObject * pArgs, * pValue, * t;
    int i;
    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize(); // Decode the parameter and import
    pName = PyUnicode_DecodeFSDefault("detect_shapes");
    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) { // Push the arguments (image name and roi) into a string

        // Check that the function exists and if not, catch the error
        pFunc = PyObject_GetAttrString(pModule, argv[2]);

        if (pFunc && PyCallable_Check(pFunc)) {

            pArgs = PyTuple_New(argc - 3);

            for (i = 0; i < argc - 3; ++i) {
                pValue = PyBytes_FromString(argv[i + 3]);

                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }

                // Add the arguments to tuple to be passed into Python
                PyTuple_SetItem(pArgs, i, pValue);
            }

            pValue = PyObject_CallObject(pFunc, pArgs); // Fetch the return value
            Py_DECREF(pArgs); // If call successfull print results else fail

            if (pValue != NULL) {
                const wchar_t * json = PyBytes_AsString(pValue);
            } else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr, "Call failed\n");
                return 1;
            }
        }
        else {
            if (PyErr_Occurred()) PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    } else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
        return 1;
    }
    if (Py_FinalizeEx() < 0) {
        return 120;
    }
    return 0;
}

```

Python source for the first use-case

```python
import the necessary packages# from pyimagesearch.shapedetector
import ShapeDetector
import argparse
import imutils
import cv2
import cv
import numpy as np
import json

def shapedetector(image, roi):

    # load the image and resize it to a smaller factor so that
    # the shapes can be approximated better
    image = image.decode("utf-8")
    image = image.strip('\'')
    image = cv2.imread(image, cv2.IMREAD_GRAYSCALE)

    roi = roi.decode("utf-8")
    roi = roi.strip('\'')

    # create ROI from arguments and crop image
    a, b, c, d = (int(x) for x in roi.split(','))

    image = image[a: b, c: d]

    # find contours in the image without tresholds
    cnts = cv2.findContours(image, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
    cnts = cnts[0]

    if imutils.is_cv2()

    else cnts[1]

        # loop over the contours
        for c in cnts:

            if len(c) > 5:

                x, y, w, h = cv2.boundingRect(c)

                if w < 220:
                    if w > 10:

                        # compute the center of the contour
                        M = cv2.moments(c)
                        cX = int(M["m10"] / M["m00"])
                        cY = int(M["m01"] / M["m00"])

data = {}
data['x'] = x
data['y'] = y
data['cX'] =
cX data['cY'] = cY

json_data = json.dumps(data) return bytes(str(json_data), 'utf-8')

```

C++ source for the second use-case

```
#include < opencv2 / opencv.hpp >
#include < iostream >
#include < string >
#include < ctime >
#include < fstream >
#include "packages/rapidjson.beta.1.0.1/build/native/rapidjson/include/rapidjson/document.h"
#include "packages/rapidjson.beta.1.0.1/build/native/rapidjson/include/rapidjson/prettywriter.h"
#include "packages/rapidjson.beta.1.0.1/build/native/rapidjson/include/rapidjson/writer.h"
#include "packages/rapidjson.beta.1.0.1/build/native/rapidjson/include/rapidjson/stringbuffer.h"

using namespace cv;
using namespace rapidjson;
using namespace std;

vector < vector < Point > > contours; // RNG rng(12345);
vector < Vec4i > hierarchy;

Mat roi;

vector < Point2f > distance;
vector < float > calculateDistances(vector < Point2f > , vector < Point2f > );
vector < Point2f > get_contours1(Mat);
vector < Point2f > get_contours2(Mat);

int main(int argc, char * * argv) {

    std::clock_t start;
    double duration;

    // Check amount of parameters, return error if wrong
    if (argc != 3) {
        cout << " Usage: ImageToLoadAndDisplay roi (roi in the format x-start,x-height,y-start,y-height) " << endl;

            return -1;
        }

        // Establish the Mat object and load the image in grayscale
        Mat image;

        image = imread(argv[1], IMREAD_GRAYSCALE); // Read the file

        if (!image.data) {
            cout << "Could not open or find the image" << std::endl;
            return -1;
        }
        start = std::clock(); // Split the roi argument into an array
        string arg2 = argv[2];
        std::istringstream iss(arg2);

        char c; // dummy character for the colon
        int a[4];
        iss >> a[0];

        for (int i = 1; i < 8; i++) iss >> c >> a[i];
            Rect rec(a[0], a[1], a[2], a[3]);
            Mat roi = image(rec);
            vector < Point2f > distance1 = get_contours1(roi);
            vector < Point2f > distance2 = get_contours2(roi);
            vector < float > result = calculateDistances(distance1, distance2);

            Document d; // Null

            d.SetObject();
            Document::AllocatorType & allocator = d.GetAllocator();
            d.AddMember("Distance between the pins", result[0], allocator);
            d.AddMember("Distance between 1st pin and base", result[1], allocator);
            d.AddMember("Distance between 2nd pin and base", result[2], allocator);

            StringBuffer strbuf;
            PrettyWriter < StringBuffer > writer(strbuf);

            d.Accept(writer);
            ofstream file;

            file.open("script_output");
            file << strbuf.GetString();
            file.close();
            duration = (std::clock() - start) / (double) CLOCKS_PER_SEC;
            file.open("benchmark_results_c", ios_base::app);
            file << "time: " << duration << '\n';
            file.close();
            return 0;
        }

        // Get the contours with white area; this is done with
        // threshold, could have perhaps been done with inverse function as well
        vector < Point2f > get_contours1(Mat roi) {

        vector < Point2f > distance;
        Mat img_bw;

        threshold(roi, // source image
            img_bw, // destination image
            127, // threhold val.
            255, // max. val
            0); // binary

        findContours(img_bw, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE)
;
        vector < vector < Point > > contours_poly(contours.size());
        vector < float > radius(contours.size());
        vector < Rect > boundRect(contours.size());

        // Format the Mat object "drawing" by creating an empty array
        Mat drawing = Mat::zeros(img_bw.size(), img_bw.type());

        float res1 = 0;
        float res2 = 0;

        vector < float > v; // Loop thru contours

        for (size_t i = 0; i < contours.size(); i++) {

            // Approximate the polygonal curve, 3 was enough as a
            // precision but can be adjusted
            approxPolyDP(Mat(contours[i]), contours_poly[i], 3, true);

            // Calculate the bounding rectangle
            boundRect[i] = boundingRect(Mat(contours_poly[i]));

            size_t count = contours[i].size();

            if (count < 5) continue;

                   // Declare a new mat object and convert the contour to points
                   Mat points;
                   Mat(contours[i]).convertTo(points, CV_32F);

                   // Use fitEllipse to get the xy position and size of the contour
                   RotatedRect box = fitEllipse(points);

                   // Check that the size of the bounding rectangle meets the
                   // minimum requirements, we don't want to find everything
                   if (box.size.height > 100 && box.size.width < 700) {

                       // Get the extreme points for each of the contours; not
                       // doing the storing in the most elegant way, this code
                       // is pretty static because we have only two contours of
                       // interest in this case
                       if (distance.size() == 0) {
                           Point extBot = * max_element(contours[i].begin(), contours
                                          [i].end(), [](const Point & lhs,
                               const Point & rhs) {
                               return lhs.y < rhs.y;
                           });
                           Point extRight1 = * max_element(contours[i].begin(),
                                             Contours[i].end(), [](const Point
                                             & lhs,
                               const Point & rhs) {
                               return lhs.x < rhs.x;
                           });
                           distance.push_back(extBot);
                           distance.push_back(extRight1);
                       }
                       else {
                           Point extTop = * min_element(contours[i].begin(), contours
                                          [i].end(), [](const Point & lhs,
                               const Point & rhs) {
                               return lhs.y < rhs.y;
                           });

                           Point extRight2 = * max_element(contours[i].begin(),
                                             contours[i].end(), [](const Point
                                             & lhs,
                               const Point & rhs) {
                               return lhs.x < rhs.x;
                           });

                           distance.push_back(extTop);
                           distance.push_back(extRight2);
                       }
                   }
               }
               return distance;
           }

           // Get the center point of the black ellipses in the picture, this is
```

```
        // done without thresholds
        vector < Point2f > get_contours2(Mat roi) {
        vector < vector < Point > > contours_poly(contours.size());
        vector < Rect > boundRect(contours.size());

        findContours(roi, contours, hierarchy, RETR_LIST, CHAIN_APPROX_NONE);

        vector < Point2f > center1; // Loop thru contours

        for (size_t i = 0; i < contours.size(); i++) {

            // Approximate the polygonal curve, 3 was enough as a
            // precision but can be adjusted
            approxPolyDP(Mat(contours[i]), contours_poly[i], 3, true);
            boundRect[i] = boundingRect(Mat(contours_poly[i]));

            size_t count = contours[i].size();

            if (count < 5) {
                continue;
            }

            // Check the size of the bounding rectangle, there are other
            // black areas as well which we are not interested in
            if ((boundRect[i].width < 200) && (boundRect[i].width > 100)) {

                // Declare a new mat object and convert the contour to points
                Mat points;
                Mat(contours[i]).convertTo(points, CV_32F);

                RotatedRect box = fitEllipse(points);

                // Use fitEllipse to get the xy position and size of the contour
                    center1.push_back(box.center);
            }
        }

        return center1;
    }

    // Calculate the distance between countour points of interest,
    // return a vector with results
    vector < float > calculateDistances(vector < Point2f > distance, vector <
                                        Point2f > center1) {
    vector < float > result;

    result.push_back(sqrt(pow((distance[0].x - distance[2].x), 2) +
                     pow((distance[0].y - distance[2].y), 2)));
    result.push_back(sqrt(pow((distance[1].x - center1[0].x), 2) +
                     pow((distance[1].y - center1[0].y), 2)));
    result.push_back(sqrt(pow((distance[3].x - center1[1].x), 2) +
                     pow((distance[3].y - center1[1].y), 2)));
        return result;
    }
```

Python source for the second use-case

```
import the necessary packages# from pyimagesearch.shapedetector
import ShapeDetector
import argparse
import imutils from imutils
import perspective from imutils
import contours
import cv2
import cv
import numpy as np
import json from decimal
import *
import math
import time from scipy.spatial
import distance as dist

def shapedetector(image, roi):

    # load the image and resize it to a smaller factor so that
    # the shapes can be approximated better
    image = image.strip('\'')
    image = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
    t0 = time.time()# roi = roi.decode("utf-8")
    roi = roi.strip('\'')

    # create ROI from arguments and crop image
    a, b, c, d = (int(x)

    for x in roi.split(','))
        image = image[a: b, c: d]

    # Get centers for contours; this will fetch the centers for the black circles
    # inside the ROI, no tresholding needed as looking for black color only
    cnts2 = cv2.findContours(image, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)

    cnts2 = cnts2[0]

    if imutils.is_cv2()
    else cnts2[1]

    (center1, center2) = 0, 0

    for i in cnts2:
        if len(i) > 8:

            x, y, w, h = cv2.boundingRect(i)

            # Filter contours by size
            if w < 400:

                if w > 100:

                # Fetch minimum area rectangle around the contours and dump results
                # to a numpy array
                box = cv2.minAreaRect(i)
                box = cv2.boxPoints(box)
                box = np.array(box, dtype = "float")

```

```python
            # Use fitEllipse to fit an ellipse inside the contour,
            # dump the result to numpy array and fetch the center
            # point into A tuple
            ellipse = cv2.fitEllipse(i)
            (center, r, t) = ellipse

            if center1 == 0:
                (center1, center2) = center, 0
            else :
                (center1, center2) = center1, center

            # Apply some tresholds to fetch the white areas inside the ROI
            ret, tresh = cv2.threshold(image, 127, 255, 0)
            cnts = cv2.findContours(tresh, cv2.RETR_EXTERNAL,
                                    cv2.CHAIN_APPROX_SIMPLE)
            cnts = cnts[0]

            if imutils.is_cv2()
            else cnts[1]

            # Establish a new blank tuple to store the extreme points of the
            # contour
            (res1, res2, extTop1, extBot2) = 0, 0, 0, 0

            # loop over the contours
            for c in cnts:
                if len(c) > 8:

                    x, y, w, h = cv2.boundingRect(c)

                    if w > 400:

                        # Get the extreme points of the contours and add them to
                        # a tuple
                       if isinstance(res1, int):

                           (res1, res2, extTop1, extBot2) = getPoints(c), 0,
                                                            tuple(c[c[: , : , 1]
                                                            .argmax()][0]), 0
                           cent1 = np.array([float(center1[0]),
                                             float(center1[1])])
                       else :
                            (res1, res2, extTop1, extBot2) = res1, getPoints(c),
                                                             extTop1, tuple(c[c
                                                             [: , : , 1].argmin()
                                                             ][0])
                            cent2 = np.array([float(center2[0]),
                                              float(center2[1])])

    data = {}

    points3 = np.array([extTop1[0], extTop1[1]])
    points4 = np.array([extBot2[0], extBot2[1]])
    data["distance_1"] = cv2.norm(res1, cent1)
    data["distance_2"] = cv2.norm(res2, cent2)
    data["distance_between_objects"] = cv2.norm(points3, points4)

    t1 = time.time() total = t1 - t0

    filewrite('time: {}\n'.format(total), 'benchmark_results_python', 'a')

    return json.dumps(str(data))

```

```
# Calculate mininum rotated rectangle around the contour, order the points
# and get the midpoint between the top right and bottom right points
def getPoints(c):

    box = cv2.minAreaRect(c)
    box = cv2.boxPoints(box)
    box = np.array(box, dtype = "float")
    xSorted = box[np.argsort(box[: , 0]), : ]
    leftMost = xSorted[: 2, : ]
    rightMost = xSorted[2: , : ]
    leftMost = leftMost[np.argsort(leftMost[: , 1]), : ]

    (tl, bl) = leftMost D = dist.cdist(tl[np.newaxis], rightMost, "euclidean")[0]
    (br, tr) = rightMost[np.argsort(D)[::-1], : ]

    x = float((br[0] + tr[0]) / 2)
    y = float((br[1] + tr[1]) / 2)

return np.array([x, y])

# Write the results to file
def filewrite(json, filename, writemode):

    f = open(filename, writemode)
    F1.write(json) f.close()
    return 0

ap = argparse.ArgumentParser() ap.add_argument("-i", " --image",
                                               help = "path to the image file")
ap.add_argument("-r", "--roi", help = "region of interest,
                format 'x,y,width,height'")
args = vars(ap.parse_args())

json = shapedetector(args["image"], args["roi"]);
filewrite(json, 'script_output', 'w')

```

C++ source for the third use-case

```
#include < opencv2 / opencv.hpp >
#include < iostream >
#include < string >
#include < ctime >
#include < fstream >
#include "rapidjson/document.h"
#include "rapidjson/prettywriter.h"
#include "rapidjson/writer.h"
#include "rapidjson/stringbuffer.h"
#include < cmath > using namespace cv;

using namespace rapidjson;
using namespace std;

vector < vector < Point > > contours; // RNG rng(12345);
vector < Vec4i > hierarchy;

Mat roi;

vector < float > calculateDistances(vector < Point2f > );
vector < Point2f > get_contours(Mat);
vector < float > lineDraw(float, float, float, float);
vector < float > intersect(vector < float > , vector < float > );

int main(int argc, char * * argv) { // Initialize the timer

    std::clock_t start;
    double duration; // Check amount of parameters, return error if wrong

    if (argc != 3) {
        cout << " Usage: ImageToLoadAndDisplay roi (roi in the format x-start,x-
                  height,y-start,y-height) " << endl;
        return -1;
    }

    // Establish the Mat object and load the image in grayscale
    Mat image;
    image = imread(argv[1], IMREAD_GRAYSCALE); // Read the file

    if (!image.data) {
        cout << "Could not open or find the image" << std::endl;
        return -1;
    }

    start = std::clock(); // Split the roi argument into an array
    string arg2 = argv[2];
    std::istringstream iss(arg2);
    char c; // dummy character for the colon
    int a[4];
    iss >> a[0];

    for (int i = 1; i < 8; i++) iss >> c >> a[i];
        Rect rec(a[0], a[1], a[2], a[3]);
        Mat roi = image(rec);
        vector < float > distances = calculateDistances(get_contours(roi));
        Document d; // Null
        d.SetObject();
        Document::AllocatorType & allocator = d.GetAllocator();

        d.AddMember("Distance between the pins", result[0], allocator);
        d.AddMember("Distance between 1st pin and base", result[1], allocator);
        d.AddMember("Distance between 2nd pin and base", result[2], allocator);

        StringBuffer strbuf;
        PrettyWriter < StringBuffer > writer(strbuf);
        d.Accept(writer);
        ofstream file;
        file.open("script_output");
        file << strbuf.GetString();
        file.close();
        duration = (std::clock() - start) / (double) CLOCKS_PER_SEC;

        // Benchmark results to a file
        file.open("benchmark_results_c", ios_base::app);
        file << "time: " << duration << '\n';
        file.close();

        return 0;
    }

    // Get the center point of the black ellipses in the picture, this is done
    // without thresholds
    vector < Point2f > get_contours(Mat roi) {
    findContours(roi, contours, RETR_LIST, CHAIN_APPROX_SIMPLE);

    // Bunch of variable definitions
    vector < RotatedRect > minRect(contours.size());

    RNG rng(12345);
    Mat drawing;

    vector < Point2f > centers;
    vector < Point2f > intersects;
    vector < float > angles;
    vector < float > dists1, orig1;
    vector < float > dists2, orig2;
    vector < float > isect_point, isect_point_orig;

    CvPoint2D32f points[4];

   int  length  =  500;
   float line_endpoint_x;
   float line_endpoint_y;
   double pi = atan(1) * 4;
   int x = 0;
   vector < Point2f > distances; // Loop thru contours

   for (size_t i = 0; i < contours.size(); i++) {

       // Use minAreaRect to fetch the minimum enclosing rectangle around
       // the contour
       minRect[i] = minAreaRect(Mat(contours[i]));
       size_t count = contours[i].size();

       // Make sure that we are actually inspecting a closed contour and not
       // a line
       if (count < 5) {
           continue;
       }

```

```
        // Get the large black area inside the ROI, map corners to a vector
        // with cvBoxPoints
        if (minRect[i].size.width > 2000) {
           Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255),
                                 rng.uniform(0, 255));

           cvBoxPoints(minRect[i], points);
         }

         // Some magic to filter out unwanted contours
         if ((minRect[i].size.width > 100) && (minRect[i].size.width < 500)) {
           centers.push_back(minRect[i].center);
           angles.push_back(minRect[i].angle);
         }
     }

     // Loop thru the hole centers and calculate endpoint of a line from that spot
     // to 500px away

     // Max value of m is dynamic so it will take any amount of hole centers
     for (int m = 0; m < centers.size(); m++) {

         // Calculate the line endpoints; take into account the angle (possible
         // skewing of the object)
         line_endpoint_x = centers[m].x + length * cos(angles[m] * pi / 180.0);
         line_endpoint_y = centers[m].y + length * sin(angles[m] * pi / 180.0);

         orig1 = lineDraw(centers[m].x, centers[m].y, line_endpoint_x,
                          line_endpoint_y);

         if (x == 0 || x == 1) {

             // Two first holes are on the right edge so intersection occurs on
             // the right side baseline
             Orig = lineDraw(points[3].x, points[3].y, points[2].x, points[2].y);
         }
         else {

             // Third hole is on the left edge so intersection occurs on the left
             // side baseline
             orig = lineDraw(points[0].x, points[0].y, points[1].x, points[1].y);
         }

         x++;

         // Stuff results to a vector
         dists1 = orig1;
         dists1.insert(dists1.end(), orig1.begin(), orig1.end());
         dists2 = orig;
         dists2.insert(dists2.end(), orig.begin(), orig.end());
         isect_point_orig = intersect(dists1, dists2);
         isect_point = isect_point_orig;
         isect_point.insert(isect_point.end(), isect_point_orig.begin(),
                            isect_point_orig.end());
         isect_point[1] *= -1;
         isect_point[0] *= -1;
         intersects.push_back(Point2f(isect_point[0], isect_point[1]));
     }

     // Combine results to a single vector
     distances.insert(distances.end(), centers.begin(), centers.end());
     distances.insert(distances.end(), intersects.begin(), intersects.end());

```

```
  return distances;
}

// Calculate intersection between base and line from the hole centerpoints
vector < float > intersect(vector < float > line1, vector < float > line2) {
    vector < float > points;
    float x;
    float y;
    float D = (line1[0] * line2[1]) - (line1[1]) * (line2[0]); // Main determinant
    float Dx = (line1[2] * line2[1]) - (line1[1] * line2[2]); // Get the Dx
    float Dy = (line1[0] * line2[2]) - (line1[2] * line2[0]); // Get the Dy

    // Check if the lines are ever intersecting; if they are, calculate x and
    // y positions
    if (D != 0) {
        x = Dx / D;
        y = Dy / D;
    }

    // TODO: Could apply a little bit of error handling with try-catch if the
    // intersection fails, left it out because in real life the error means a
    // problem elsewhere (issue with ROI etc)
    points.push_back(x);
    points.push_back(y);

    return points;
}

// Construct functions for the lines to feed them for the intersect function
vector < float > lineDraw(float point1_x, float point1_y, float point2_x,
        float point2_y) {
    vector < float > points;
    float a = point1_y - point2_y;
    float b = point2_x - point1_x;
    float c = ((point1_x * point2_y) - (point2_x * point1_y));
    points.push_back(a);
    points.push_back(b);
    points.push_back(c);

    return points;
}

// Calculate the distance between holes and distances from holes to the base
// This is constructed dynamic so will work for any size of a vector of points
vector < float > calculateDistances(vector < Point2f > points) {
    vector < float > result;

    for (int i = 0; i < points.size(); i++) { // Loop thru indices

        if (i < points.size() / 2) {

            // Calculate distance from center points to the base
            result.push_back(sqrt(pow((points[i].x - points[i + 3].x), 2)
                                + pow((points[i].y - points[i + 3].y), 2)));
        }
        else {

            // Calculate distances between the hole center points
            if (i < points.size() - 1) {
                result.push_back(sqrt(pow((points[i].x - points[i + 1].x), 2) +
                                    pow((points[i].y - points[i + 1].y), 2)));
            }
            else {
                result.push_back(sqrt(pow((points[i].x - points[(points.size() / 2)]
                                      .x),2) + pow((points[i].y - points[i –
                                       (points.size() / 2) + 1].y), 2)));
            }
        }
    }

    return result;
}
```

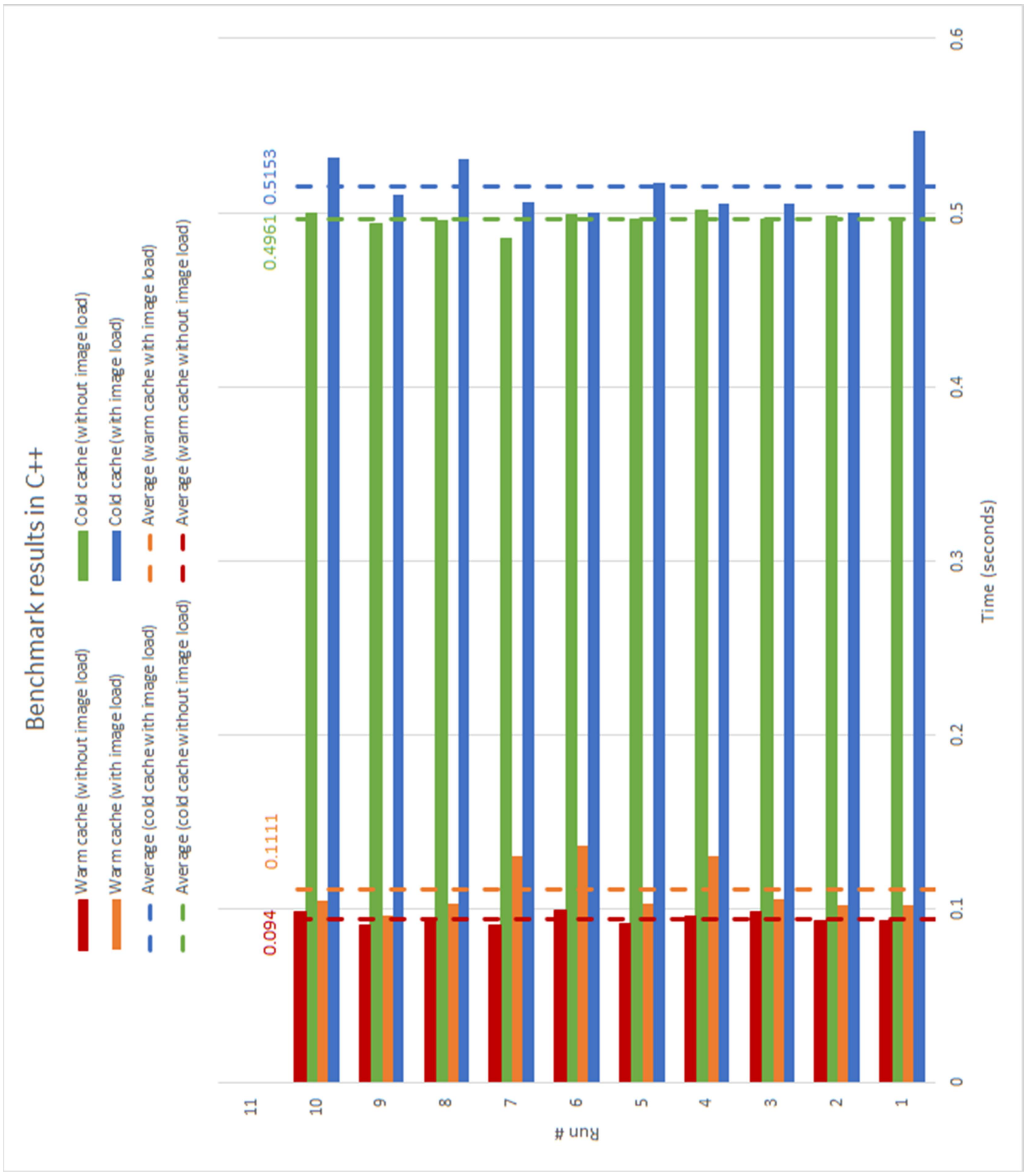
Python source for the third use-case

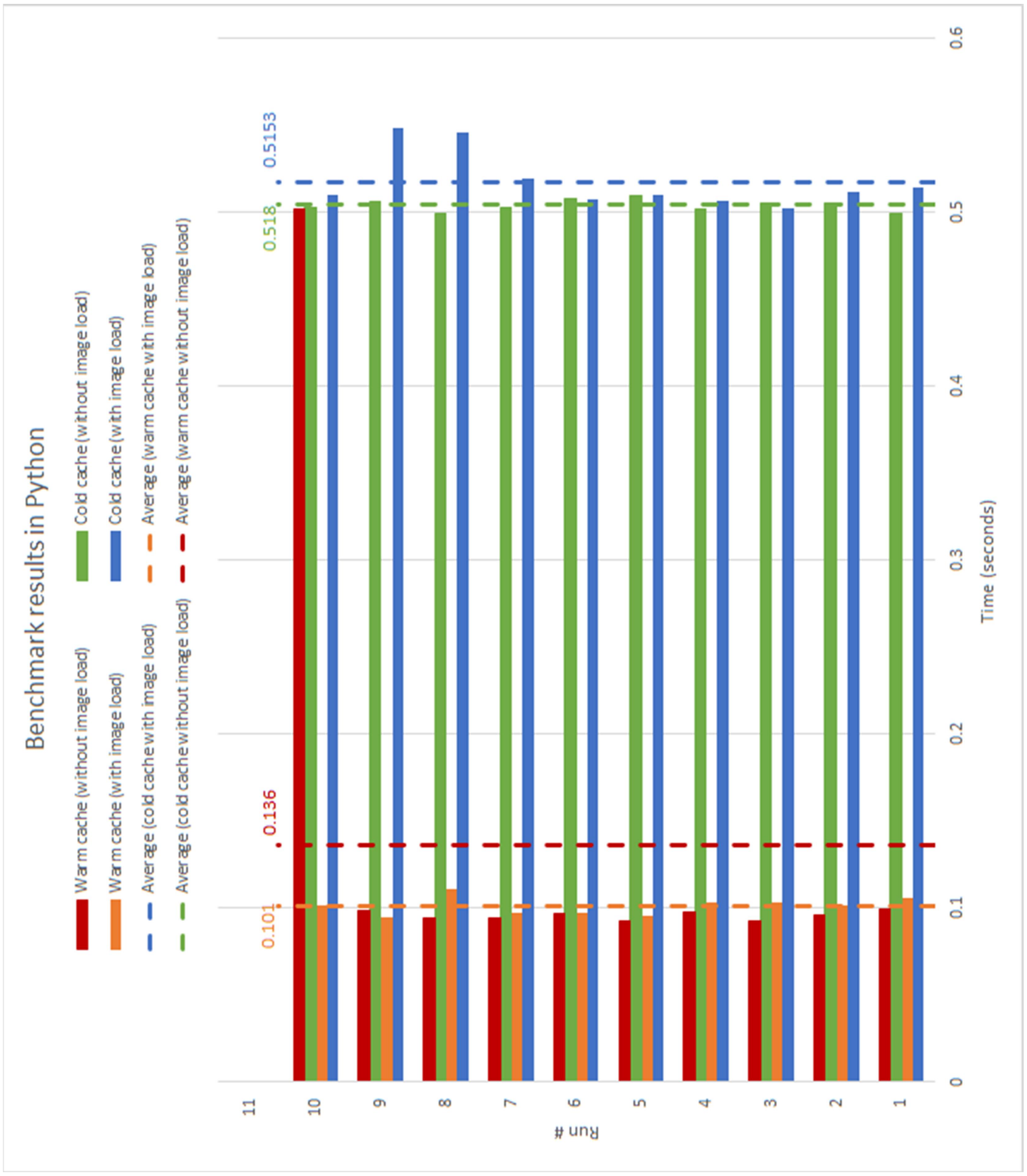
```python
# Import  the  necessary  packages
import argparse
import imutils from imutils
import contours
import cv2
import cv
import numpy as np
import json
import math
import time

def shapedetector(image, roi):

    # Load the image and resize it to a smaller factor so that the shapes can be
    #  approximated better
    image = image.strip('\'')
    image = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
    t0 = time.time()

    roi = roi.strip('\'')

    # Create ROI from arguments and crop image
    a, b, c, d = (int(x)

    for x in roi.split(','))
        image = image[a: b, c: d]

    # Apply the masks and treshold the image
    mask = np.zeros(image.shape, np.uint8)

    # Get centers  for contours; this will fetch the centers for the black circles
    # inside the ROI, no tresholding needed as the color is pure black
    cnts2 = cv2.findContours(image, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
    cnts2 = cnts2[0]

    if imutils.is_cv2()

    else cnts2[1]

    # Define some numpy arrays for storage
    centers = []
    points = []
    dist2edge = []
    dist2point = []

    z = 0
    angle = 0
    length = 500

    # Assume a line length 500 px to make sure that intersection occurs
    for i in cnts2:

        if len(i) > 8:

            # Fetch minimum area rectangle around the contours
            # and dump results to a numpy array, the array needed
            # is needed for printing only
            box = cv2.minAreaRect(i)

            if box[1][0] > 100:

                if box[1][0] < 500:

                    centers.insert(z, box[0])
                    angle = box[2]

                    # Calculate coordinates for the endpoint of the line
                    # from center of the hole
                    line_x = box[0][0] + length * math.cos(angle * math.pi / 180.0)
                    line_y = box[0][1] + length * math.sin(angle * math.pi / 180.0)

                    points.insert(z, (line_x, line_y)) z = z + 1

                # Check for the large black area inside the roi
                if box[1][0] > 2000:
                    box2 = cv2.boxPoints(box)
                    angle = box[2]

    # While taking the angle of the minimum bounding rectangle, of the large
    # contour(position of the object under the camera might change) define a line
    # from the center of the holes to the appropriate edge of the large contour
    for j in range(0, 3):

        # Inspecting 3 holes so draw 3 lines
        l1 = line([centers[j][0], centers[j][1]], [points[j][0], points[j][1]])

        # Line from the hole center
        if j == 0:

            # At the same time calculate distances between holes
            dist2point = np.append(dist2point, math.sqrt((centers[j][0] – centers
                                   [j + 1][0]) * * 2 + (centers[j][1] – centers
                                   [j + 1][1]) * * 2))
            dist2point = np.append(dist2point, math.sqrt((centers[j][0] – centers
                                   [j + 2][0]) * * 2 + (centers[j][1] – centers
                                   [j + 2][1]) * * 2))

            # Line from the right edge of the main contour
            l2 = line([box2[2][0], box2[2][1]], [box2[3][0], box2[3][1]])

        if j == 1:

            dist2point = np.append(dist2point, math.sqrt((centers[j][0] - centers
                                   [j + 1][0]) * * 2 + (centers[j][1] – centers
                                   [j + 1][1]) * * 2))


            # Line from the right edge of the main contour
            l2 = line([box2[2][0], box2[2][1]], [box2[3][0], box2[3][1]])

        if j == 2:

            # Line from the left edge of the main contour
            l2 = line([box2[0][0], box2[0][1]], [box2[1][0], box2[1][1]])

        r = intersect(l1, l2) # get the intersection point

        # calculate distance to edge and append to tuple for storage
        dist2edge = np.append(dist2edge, math.sqrt((r[0] - centers[j][0]) * * 2 +
                              (r[1] – centers[j][1]) * * 2))

        data = {}
        data["distance_to_edges"] = {}
        data["distance_between_holes"] = {}

        for q in range(0, dist2edge.size):

            data["distance_to_edges"][q] = dist2edge[q]

        for q in range(0, dist2edge.size):
            data["distance_between_holes"][q] = dist2point[q]

        t1 = time.time()
        total = t1 - t0

        filewrite('time: {}\n'.format(total), 'benchmark_results_python', 'a')

        return json.dumps(str(data))

    # Define the equation for a line between points
    def line(point1, point2):
        a = (point1[1] - point2[1])
        b = (point2[0] - point1[0])
        c = (point1[0] * point2[1] - point2[0] * point1[1])

        return a, b, -c  # return the coefs of line equation

    # Define the function for intersection
    def intersect(line1, line2):
        D = line1[0] * line2[1] - line1[1] * line2[0]

        # Get the main determinant from matrix[A1 B1, A2 B2]
        Dx = line1[2] * line2[1] - line1[1] * line2[2] # Dx from matrix[C1 B1, C2 B2]
        Dy = line1[0] * line2[2] - line1[2] * line2[0] # Dy from matrix[A1 C1, A2 C2]

        if D != 0:

            # Calculate x and y
            x = Dx / D
            y = Dy / D
            return x, y

        else :
            return False

    # Write the results to file
    def filewrite(json, filename, writemode):
        f = open(filename, writemode)
        f.write(json)
        f.close()
        return 0

    ap = argparse.ArgumentParser()
    ap.add_argument("-i", "--image", help = "path to the image file")
    ap.add_argument("-r", "-roi", help = "region of interest, format 'x,y,width,height'")

    args = vars(ap.parse_args())
    json = shapedetector(args["image"], args["roi"]);

```

Python source for the Calibrator

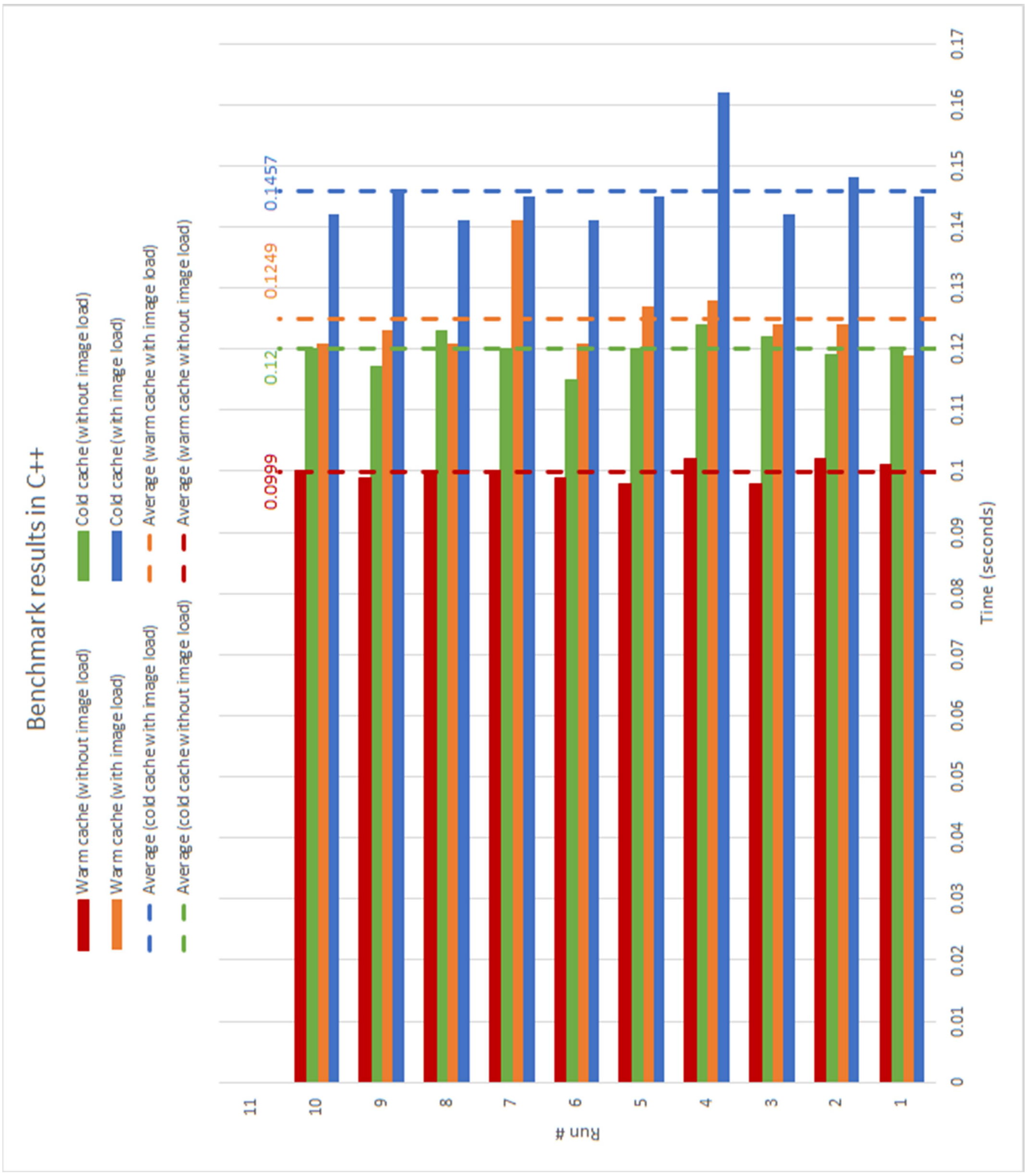
```python
# Import the necessary packages
import argparse
import imutils from imutils
import contours
import cv2
import cv
import numpy as np
import math

def calibrator(image):

    # Load the image and resize it to a smaller factor so that
    # the shapes can be approximated better
    image = image.strip('\'')
    image = cv2.imread(image, cv2.IMREAD_GRAYSCALE)

    # Create two ROIs and crop image
    a, b, c, d = 550, 50, 2200, 600
    e, f, g, h = 550, 50, 700, 600

    roi1 = image[a: a + b, c: c + d]
    roi2 = image[e: e + f, g: g + h]

    # Function calls
    dist1ellipse, dist1angle = calcDistance(roi1)
    dist2ellipse, dist2angle = calcDistance(roi2)

    # Print the calibration error on y - axis
    print("Calibration error y:", dist2ellipse[0][1] - dist1ellipse[0][1])

    # Print the average angle of distortion
    print("Angle of distortion:", ((dist1angle - 90) + (dist2angle - 90)) / 2)

def calcDistance(roi):

    # Blur some of the noise out of the image, kernel 3x3
    blur = cv2.blur(roi, (3, 3), 0)

    # Use Canny to detect the edges
    edges = cv2.Canny(blur, 50, 200, 1)

    # Find the contours from the canny output, chain_approx_simple to save memory
    cnts = cv2.findContours(edges, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    cnts = cnts[0]

    if imutils.is_cv2()

    else cnts[1]

    for i in cnts:

        if len(i) > 8:

            box = cv2.minAreaRect(i)

            if box[0][1] > 30:

                # Use fitEllipse to find the center of mass
                ellipse = cv2.fitEllipse(i)
                p = ellipse[2]

    # Return the center point and angle of distortion
    return ellipse, p

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", help = "path to the image file")
args = vars(ap.parse_args())

calibrator(args["image"]);

```

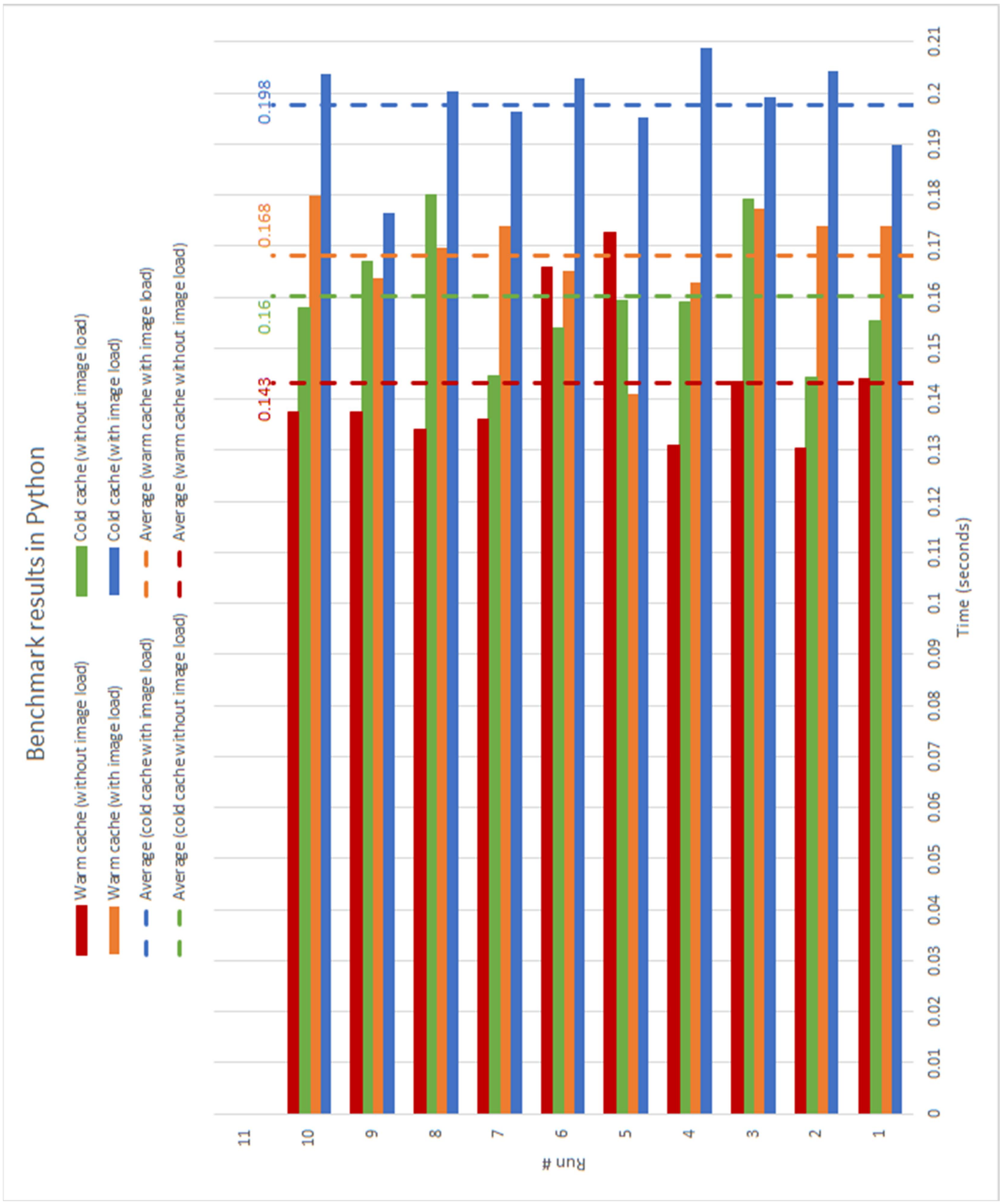
Graphs on the benchmark results for the second use-case




Benchmark results in Python
Warm cache (without image load)
Warm cache (with image load)
Average (cold cache with image load)
Average (cold cache without image load)
Cold cache (without image load)
Cold cache (with image load)
Average (warm cache with image load)
Average (warm cache without image load)
0.5153
0.518
0.136
0.101
Run #
11
10
9
8
7
6
5
4
3
2
1
0
0.1
0.2
0.3
0.4
0.5
0.6
Time (seconds)

Graphs on the benchmark results for the third use-case




Benchmark results in Python
Warm cache (without image load)
Cold cache (without image load)
Warm cache (with image load)
Cold cache (with image load)
Average (cold cache with image load)
Average (warm cache with image load)
Average (cold cache without image load)
Average (warm cache without image load)
0.198
0.168
0.16
0.143
Run #
11
10
9
8
7
6
5
4
3
2
1
Time (seconds)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0.11
0.12
0.13
0.14
0.15
0.16
0.17
0.18
0.19
0.2
0.21