Czech Technical University in Prague

Faculty of Mechanical Engineering

Department of Instrumentation and Control Engineering

BACHELOR THESIS

Pneumatic components automatic motion detection using computer vision

Automatická detekce pohybu pneumatických součástí pomocí počítačového vidění

AUTHOR: Pavel Paranin

BRANCH OF STUDY: Information and Automation Technology

SUPERVISOR: Ing. Matouš Cejnek

PRAHA 2020

Declaration

I hereby declare I have written this bachelor thesis independently and quoted all the Sources of information used in accordance with methodological instructions on ethical Principles for writing an academic thesis.

In Prague, July 2020

Pavel Paranin

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Paranin** Jméno: **Pavel** Osobní číslo: **453591**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídicí techniky**

Studijní program: **Strojírenství**

Studijní obor: **Informační a automatizační technika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Automatická identifikace pohybu lineárních pohonů pomocí strojového vidění

Název bakalářské práce anglicky:

Automatic identification of linear actuator movement via machine vision

Pokyny pro vypracování:

Cílem této práce je vytvořit program pomocí knihovny OpenCV v jazyce Python, který dokáže z dlouhého video záznamu identifikovat pohybující se lineární pohony a světelnou indikaci. Vhodnost zvolených metod by měla být v práci obhájena proti alternativním řešením. Konkrétně by program měl:

- Identifikovat v záznamu obálky jednotlivých pohonů a potenciální světelnou indikaci
- Identifikovat vektor směru pohybu jednotlivých pohybů v jejich obálkách.
- Po identifikaci zaznamenávat pohyb pohonu a změny stavu světelné indikace pro případné opravy modelu

Seznam doporučené literatury:

Bradski, Gary, and Adrian Kaehler. Learning OpenCV: Computer vision with the OpenCV library. " O'Reilly Media, Inc.", 2008.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Matouš Cejnek, U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **30.04.2020** Termín odevzdání bakalářské práce: **27.08.2020**

Platnost zadání bakalářské práce: ____________

Ing. Matouš Cejnek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Michael Valášek, DrSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání	Podpis studenta

Abstrakt

Cílem této práce je návrh praktického řešení pozorování stavu pneumatických prvků během pracovní fáze. V práci je popsána metoda detekování následujících parametrů: cesta zdvihu, obálka celkového zdvihu, směr pohybu a počet běhu pro jednotlivé aktuátory. Zároveň práce popisuje způsob detekce polohy a stavu kontrolních indikátorů na panelu PLC.

Práce se skládá z několika částí: úvod je určen pro teoreticky popis a lepší pochopení problémů. Ve druhé části je krátký popis existujících nebo soudobě používaných alternativách. Následující kapitoly se věnuji teoretickým základům navržené metody, analýze projektové architektury a detailní dokumentace všech algoritmu, použitých pro detekce nutných parametrů, uvedených v prvním odstavce.

Klíčová slova

Pneumatický píst, automatická detekce, počítačové vidění.

Abstract

The aim of this work is to present a practical solution for the problem of observing the state of pneumatic components during the working phase. So, in this work is described a method to detect such parameters as trajectory, boundaries, direction of movement and the number of cycles for pneumatic cylinders. It will also be considered how it is possible to determine the state and position of the control indicator lights on the PLC panel.

This work consists of several parts. The introduction part is devoted to theoretical and general issues of the problem. Second part describes existing or developing techniques, the essence of which is the same as of this work. Further parts are dedicated to the theoretical foundations of my method, analysis of the project architecture and detailed documentation of all processes occurring during the program execution.

Keywords

Pneumatic cylinder, automatic detection, computer vision

Contents

1 Introduction

1.1 Problem statement

Many modern industries tend to automate their manufacturing processes and to reduce the number of manual workers involved in the operation sequence. In many cases such automation requires linear motion. A pneumatic cylinder, which converts the energy of compressed air into linear motion, belongs to the simplest and cost-effective ways to accomplish this.

Pneumatic positioning has numerous advantages, including great productivity and low cost of ownership. Also, pneumatic systems are relatively easy and economical to repair.

Still, as all other systems, pneumatic motors are not totally error-proof, which means that at some point a system can fail or cause any error that can break the operation sequence. Such errors may be very costly to a manufacturer; thus, they should be eliminated as quickly as possible. One of the problems is to localize the problem and to understand its cause.

To be able to look after the automated process permanently, some monitoring system should be used. Requirements for such system would be:

- Non-stop observing of the automated process
- Ability to identify individual components
- Instant error detection
- Ability to localize the problem

Purpose of this thesis is to contribute into developing such a system.

1.2 Defining the term pneumatic cylinder

1.2.1 Pneumatic cylinder working principle

Pneumatic cylinder is a mechanical device, which converts compressed air into linear motion and forces a piston to move in a desired direction.

Pneumatic cylinder can be either single acting or double acting.

Principle of a single-acting cylinder is in supplying air only from one side of the cylinder; thus, air pressure moves the piston only in one direction. Returning movement of a cylinder is performed by a mechanical spring, when air supplying is stopped.

Double-acting cylinder does not have any spring for returning movement, but instead it has two ports, which supply air for both sides of the motor. Expanding and retraction of a motor is controlled purely by airflow.

Base components of a typical double-acting pneumatic cylinder are displayed in Figure 1: cap-end port (A), tie rod (B), rod-end port (C), piston (D), barrel (E), and piston rod (F) [1].

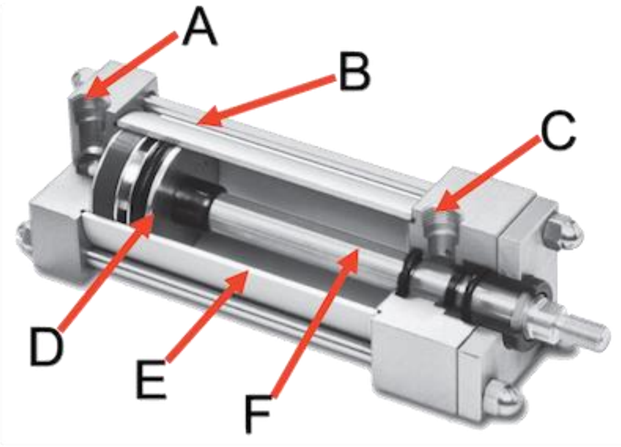


Figure 1: Standard components of a pneumatic cylinder [1].

1.2.2 Detection of cylinder position

For proper integration of pneumatic cylinders into an automated system, signals of the position status should be provided to the controller in order to build the processing flow. Sensors serve this purpose and control, whether the piston reached one of its end points during operation.

The most commonly used type of sensors are magnetic proximity sensors, which detect the magnetic field of a magnet integrated in the cylinder piston [2]. When a magnet in the piston reaches one of the sensors mounted on the cylinder, that sensor is activated and sends an „ON“ signal to a controller. Once the magnet moves out of the sensor's sight, the sensor is turned off and no signal is sent. Sensors can be mounted on multiple places on a cylinder. For example, on one side of the cylinder detects, whether the piston is retracted, and on the opposite end to detect extraction. Examples of different proximity sensors are shown in Figure 2

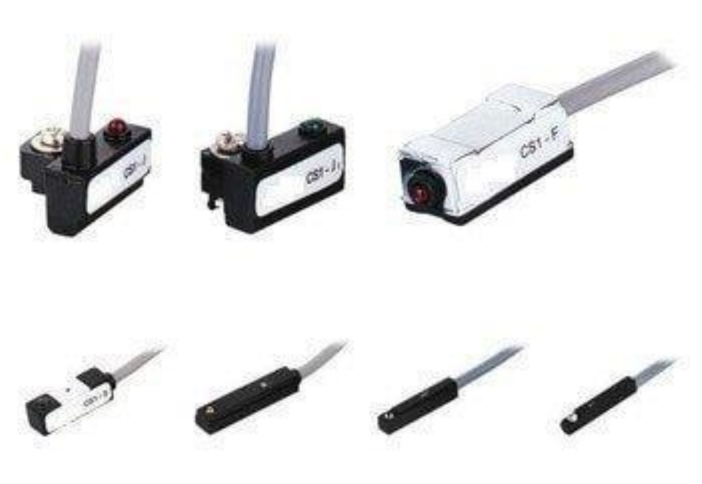


Figure 2: Different pneumatic cylinder proximity sensors [2].

Manufacturing processes are built on a principle of end position detection. Some operations are dependent on others and can be started only if some processes reached certain status. Paraphrasing the previous statement to the context of this work, some processes can start only if a motor reached some position, which is detected by a sensor.

Like any mechanical components, pneumatic cylinders are also susceptible to destruction and can fail. If this happens during the manufacturing process and is not fixed in time, such fail can lead to further errors in the sequence and repairing can cost much more than fixing only one failed cylinder.

2 State of the art

2.1 Tag Template Matching

A good example of a system for process monitoring is Camera Aided Fault Diagnostic and Isolation (CAFDI), which is still in development by another student of CTU Jiří Kubica. His solution utilizes computer vision as the core instrument for detecting errors and tracking states of observed components. It is capable of recognizing the position of cylinder pistons and states of indicator lights on the control panel.

In order to be able to differentiate required components from surroundings, this system uses Fudical Marker Systems and Tag Template Matching as main algorithms. General idea of both methods is to take some unique marker, called tag, and to stick it a place of significance, which should be detected and tracked. Once such tags are in place, system can easily detect them, since it is explicitly trained to do so. Examples of tags are shown in Figure 3.

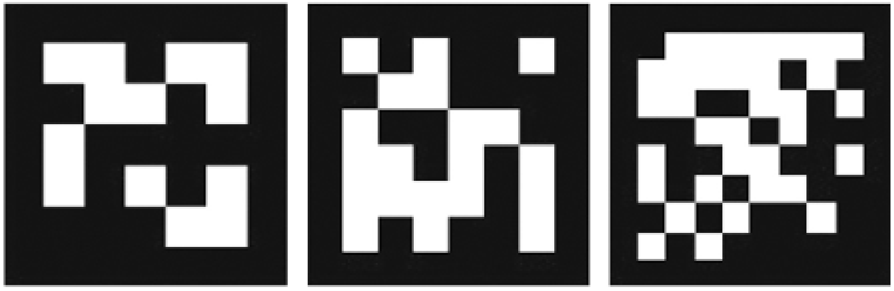


Figure 3: Example of tags. [3]

In order to detect a marker on a given picture, we need to provide to the algorithm the template image. Then by sliding it across the picture, we calculate a metric representing how "good" or "bad" the match is. We store the result of comparison for each position to a matrix, from which we can then retrieve the position with the highest value [4].

The main advantage of such system is the simplicity of usage and that it can be used in areas with poor lighting and unstable surfaces. Sufficiently strong shaking should not interfere the tag searching process.

But those tags bring not only advantageous features, but they also cause some disadvantages. Element, which should be tracked should always have a tag next to it, and if the tag is a simple sticker, it can be occasionally detached from the surface, and the process will be corrupted.

2.2 Convolutional neural network

One of the most advanced algorithms would be convolutional neural networks. Its general idea is to first teach the algorithm to recognize some patterns on the image [5]. For example, if an algorithm is fed with a large number of pictures of dogs, it will be capable of recognizing dogs on other pictures. Nevertheless, if we try to use it for cat recognition, it will fall, as no cat photos were provided before as a learning material.

In comparison to a normal neural network the convolution one takes not a vector of inputs, but its input is 3-dimensional: width, height and depth. For example, an input image can have a resolution of 32 pixels wide, 32 pixels height and 3 color levels of depth.

Convolutional network can be described as a sequence of layers, and each layer transforms one volume of activations to another through a differentiable function. Three main types of layers are used in the architecture: Convolutional Layer, Pooling Layer, and Fully Connected Layer [6]. Figure 4 shows an example architecture of a simple convolutional network.

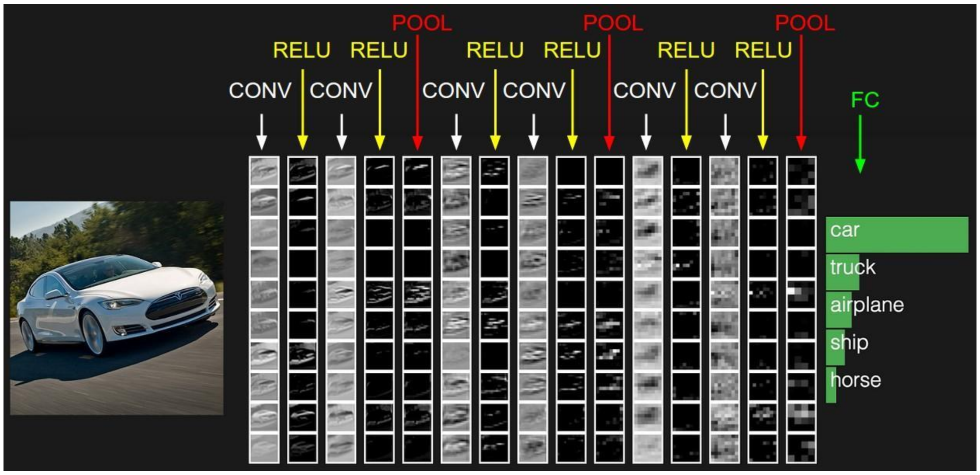


Figure 4: The activations of an example ConvNet architecture.

Finally, talking about the convolutional neural networks, this approach will only work, if it will be taught on an incredibly large dataset of photos of pneumatic cylinders. Furthermore, if a cylinder meant for detection will have some unusual piston or something will be attached to it, algorithm will probably fail.

2.3 Color-based Object Detection

An object is detected much easier, if it has some recognizable features. Those features can be for example either a shape or a color. Identifying a color on an image is a primitive task, which can be done by comparing each pixel of an image to a searched color, and if color on a pixel is close to a desired color, it is marked as 1 on a created mask matrix. If colors are different, it is signed as 0 on the mask [7].

The simplest method to detect, whether colors are similar, is to have a range of values around the values of the searched color layers. For example, if the searched color is (100, 100, 100) any and the range is 10, any color from (90, 90, 90) to (110, 110, 110) would be considered as a similar color. If the range is too small, object detection can be very sensitive to lighting of the picture, if the range is too big, it can detect undesirable colors as well as the searched ones.

Another disadvantage of this approach is that it is not general enough and even slightest change of the color can ruin the detection.

In conclusion, this method can be used in situations, where the color of detection and place lighting always remains constant.

3 Proposed method

3.1 Idea and implementation of the proposed approach

The purpose of this thesis is to explore, whether another computer vision algorithm can be proposed as an alternative to Tag Template Matching, described in the previous chapter. Scope of the thesis is to only test it on a prerecorded video and try to extract some useful information about motors and indicator lights, such as motor positions and their trajectory, position and possibly state of indicators (if latter is required for further processing steps).

Method of elements detection, which will be described in detail in the following chapters, does not need any markers or any additional equipment. It is based on an algorithm of motion detection based on comparison of each frame of the detection process to the initial frame, which is taken at the start.

This solution exposes to a user wide variety of presets to calibrate the algorithm to the point of perfect precision in detection. Furthermore, this approach is capable to detect any number of non-intersecting elements with only one camera.

The core advantage of the method used is that it is easy to setup and maintain: all the user needs to do is simply to set the cameras and to tune required parameters for optimal detection. In addition, no additional equipment should be used except cameras and a computer with graphical interface. Algorithm is resistant to small vibrations and noises; element of significance can be placed on a non-homogeneous background. In some cases, program can properly work even in places with poor lighting, of course if the corresponding parameter is tuned properly.

The biggest disadvantage of the approach is that every motion detected in the recorded area is considered either as a motor or an indicator light, since no metric of differentiating motors from any other objects is provided to the algorithm. Furthermore, if the vibration of the camera is relatively strong, it corrupts the whole detection process.

Best conditions for using proposed method are stable surface for camera attachment, sufficiently lit room and no overlapping elements in the recorded area.

3.2 Definition of computer vision

3.2.1 The term in general

Computer vision is the field, which is focused on helping computers to see and to retrieve the information from the image data. The goal is to be able to make the computer understand digital images.

This field can be called a subfield of artificial intelligence and machine learning, since it may involve usage of general machine learning algorithms.

Specter of computer vision applications is wide: when we search for an image, search engine uses CV to find info about given picture and to find similar pictures, facial-recognition algorithms are used, when somebody unlocks the phone, medicine utilizes its powers to predict various types of cancer [8], self-driving cars rely heavily on it to make sense of their surroundings. Deep learning algorithms help self-driving cars to recognize people, other cars and many different objects so that the optimal path could be built.

Modern CV has its limitations. It does a great job at classifying images and localizing objects in photos, when algorithms are trained on large data sets. However, at the core of those algorithms is still a principle of pixel matching of patterns. They have no understanding of what is going on in the image.

For example, human can use their knowledge of the world or specifically knowledge of the manufacturing process to tell, whether some pneumatic motors are used in the sequence, whether a motor moves at any moment or if any error occurred during the working phase. To be able to extract the same information from the scene computers need to be either taught to recognize necessary objects or they should use another advanced algorithm of information extraction.

3.2.2 The way computers "see"

The way computers see pictures is purely in numbers. For a computer any gray image is just a matrix of numbers and a colored image is a 3-layered set of matrices. Numbers in such matrices have values from 0 to 255, where 0 represents no color for a given layer and 255 is the highest color density. For a gray image 0 represents black color and 255 is white.

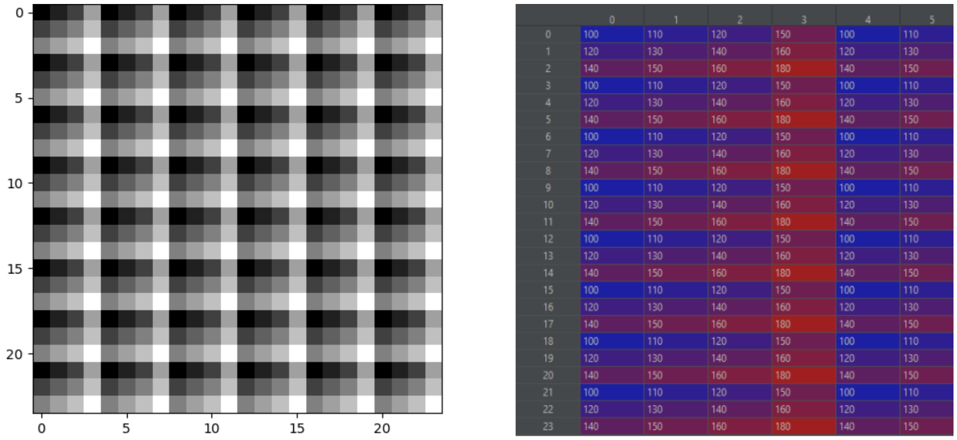
3.2.3 The way computers detect a movement

In order to detect either a motor or an indicator light on the current frame at any time, the algorithm that can detect a motion should be built. Motion in the given context is simply processed through several steps difference between the initial captured frame and any further.

3.2.3.1 Definition and representation of a frame

Every video has a frame rate characteristic, which tells how many pictures are shown one after another per unit of time. Therefore, frame is just an image, which needs to be followed by other frames in order to create a video.

Frame is represented as a matrix of numbers, where each value is the density of a color. Opposed to the normal coordinate system with y-axis going up, on a frame the y coordinate rises in the opposite direction. For example, if we take a gray image and describe it in numbers, it will look as shown in Figure 5. As it can be seen, every rectangle is just a number in a range 0-255, x-coordinate goes from left to right and y-coordinate is from top to bottom.



Gray image

	0	1	2	3	4	5
0	100	110	120	150	100	110
1	120	130	140	160	120	130
2	140	150	160	180	140	150
3	100	110	120	150	100	110
4	120	130	140	160	120	130
5	140	150	160	180	140	150
6	100	110	120	150	100	110
7	120	130	140	160	120	130
8	140	150	160	180	140	150
9	100	110	120	150	100	110
10	120	130	140	160	120	130
11	140	150	160	180	140	150
12	100	110	120	150	100	110
13	120	130	140	160	120	130
14	140	150	160	180	140	150
15	100	110	120	150	100	110
16	120	130	140	160	120	130
17	140	150	160	180	140	150
18	100	110	120	150	100	110
19	120	130	140	160	120	130
20	140	150	160	180	140	150
21	100	110	120	150	100	110
22	120	130	140	160	120	130
23	140	150	160	180	140	150

Matrix of values

Figure 5: Gray image and its representation in numbers.

3.2.3.2 Why frame should be gray scaled and blurred

Frame processing starts from gray scaling the frame, since in our case colors would not have any impact on the detection. Gray scaling helps the algorithm to be partially independent on the lighting of the scenery that is recorded. In addition, reducing the number of color-layers helps in increasing computational speed, since only one layer of the current frame should be compared.

Since every digital camera sensor has tiny variations, no two frames can be 100% identical. That said, we need to account for this and apply Gaussian smoothing to average pixel intensities. This helps smooth out high frequency noise.

As it can be seen in Figure 6, the original image is noisy and is colorful. After the processing only shades of gray remain, and noise is greatly smoothed.

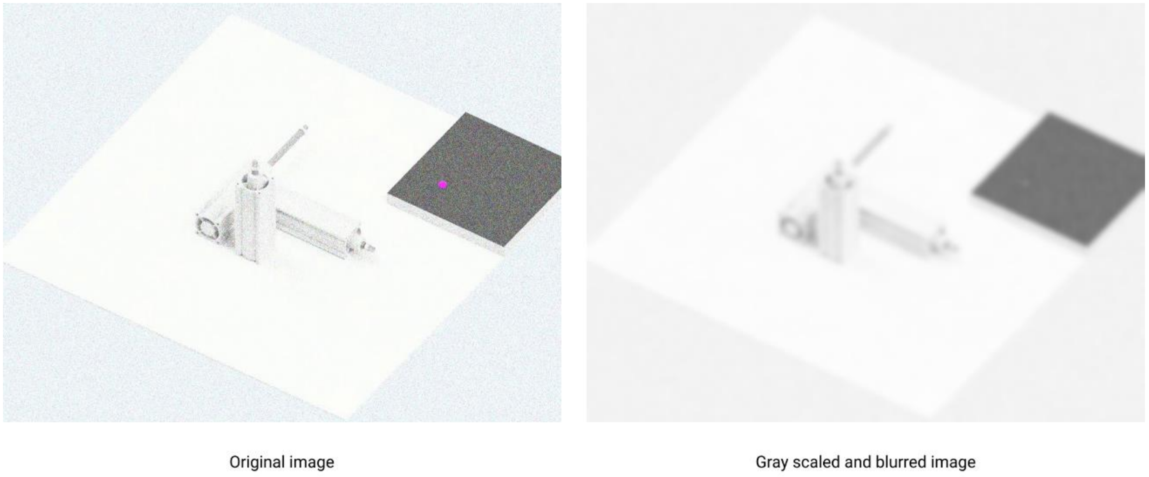


Figure 6: Result of gray scaling and blurring.

3.2.3.3 Matrix difference (frame delta)

The whole idea of the project is based on comparing frames or finding the difference between the first frame, which is taken as a reference frame, and any further. Therefore, we just need to calculate the per-element absolute difference between two matrices. Absolute value is required, because the only thing that should be extracted from the comparison is the fact of difference. Difference is found with the equation

$$delta = |initial\_frame - current\_frame|$$

All frames will have the same resolution, since the same camera captures them; therefore, no transforming is required to accomplish the matrix difference.

An example of frame delta can be seen in Figure 7. As it can be seen, areas with no detected motion are represented with black color and regions with the detected motion are clearly lighter.

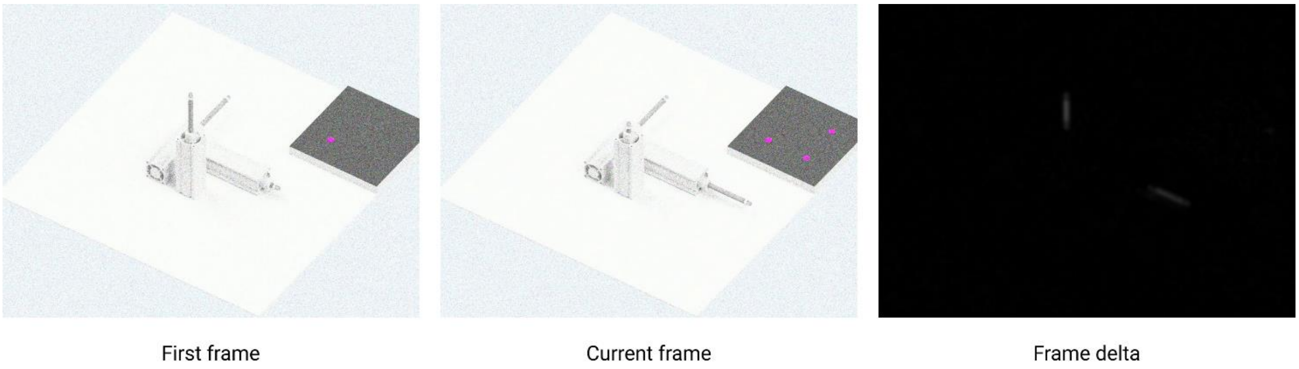


Figure 7: Eexample of frame delta. The difference between the initial frame and the current one.

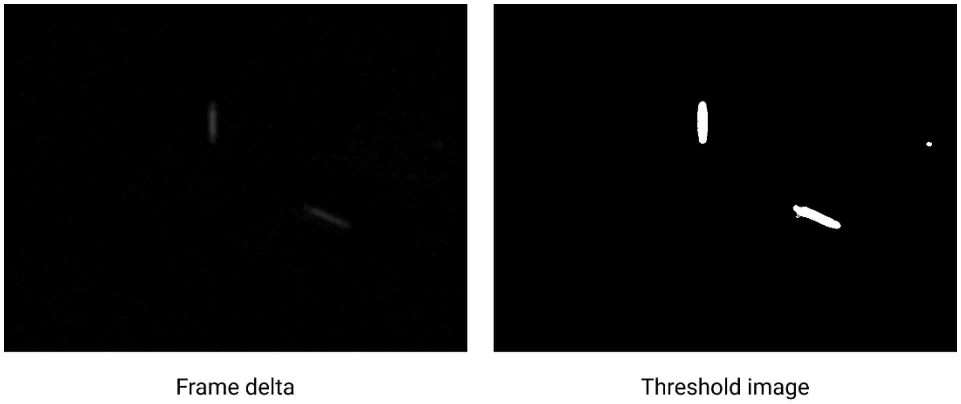
3.2.3.4 What is threshold

To properly process the frame delta between the initial and any other frame, we should know what to consider a difference. For example, motors can have a shadow on a surface underneath them. This shadow cannot be removed by gray scaling or blurring; since it is a big and a pretty visible part of a video (sunny day shadow can be taken as an example)

Since such shadow would be barely but still visible on a surface, it would have worse visible difference on a frame delta comparing to a moving actuator, which has sharp borders.

Simplest example of a threshold is a binary threshold. For every pixel, the same threshold value is applied. If the pixel value is smaller than the threshold, it is set to 0, otherwise it is set to a maximum value [9].

Many different threshold variations exist; each of them can prove useful in different situations. For example, simple binary threshold may be not good in cases if an image has different lighting conditions for different areas. In such situations, an adaptive threshold can be used. Principle of those thresholds is in determining the threshold for a pixel based on a small region around it. Result of the binary thresholding is demonstrated in Figure 8.



Frame delta

Threshold image

Figure 8: Binary thresholding.

Considering that for further steps we want to take only motions with more visible changes (e.g. moving actuator, but not the shadow), we need to use a threshold. For the purpose of this work, it was decided to choose the binary one, because it is the simplest. If any other user would like to change it to an adaptive threshold, this can be done by rewriting a single line of code.

3.2.3.5 What is erosion and dilation

Sometimes thresholding can produce unwanted noises, if the threshold value is not set correctly. To partially eliminate the influence of the unproper setup on the motion detection, applying additional morphological operations is necessary [10].

Morphological operations can be described as a shape-based image processing. They apply some structuring element to an input image and generate an output image [11]. The most basic operations are Erosion and Dilation. Their main abilities are noise removal, isolation of individual elements and joining disparate elements in an image and finding of intensity bumps or holes in an image.

To understand the principle of those operations is useful to know the term “kernel”, which is simply a window of size M x N. This kernel is usually shifted across the picture and values hold in the kernel are compared to values, placed under it in a matrix [12].

The basic idea of an erosion is to erode the boundaries of foreground object. This is done by applying some 2d kernel to an image by sliding it pixel by pixel and discarding all values under the kernel, if not some pixels have a value of 0. Otherwise, if all values of pixels under the kernel are 1, those pixels remain. In our case, it is useful for removing small white noises on a threshold image.

Since we do not want to lose any information of a motion, we want to restore the sizes of important objects, which were eroded during the corresponding step. For that purpose, we can use another morphological operation called dilation, which is just opposite of erosion. Here, pixels under the kernel are set to “1” if at least one pixel under the kernel is “1”. It increases the size of foreground objects. Since noise is gone in the previous step, we can safely increase the area of detected motions.

Result of both morphological steps can be seen in Figure 9.

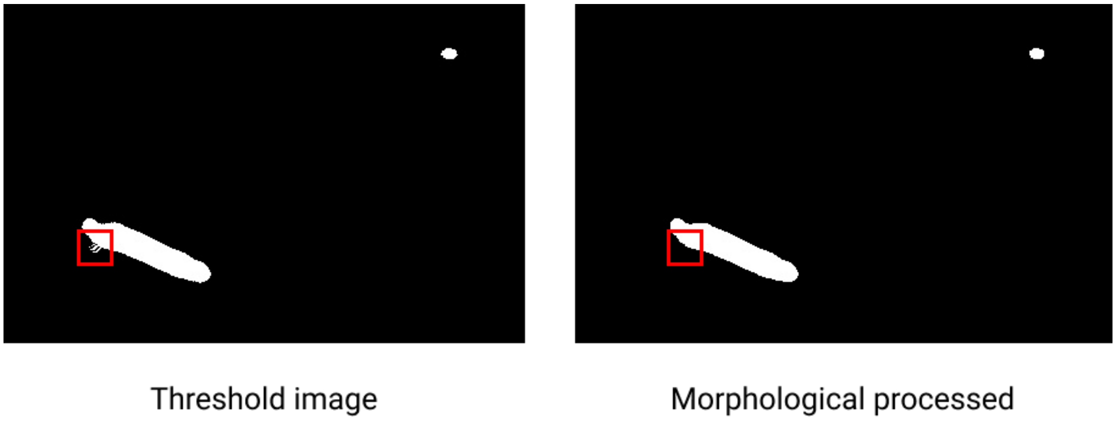
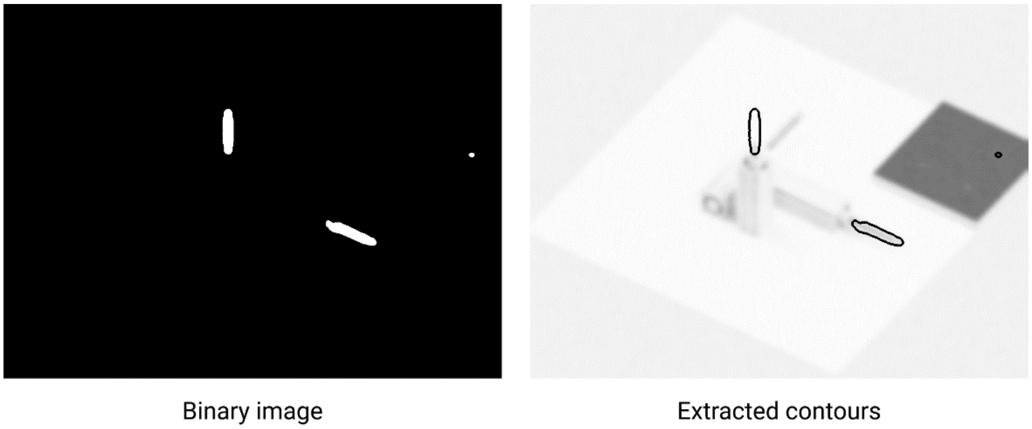


Figure 9: Example of noise removal from the image by morphological processing.

3.2.3.6 What are contours and how they are found

Once the motion is emphasized and morphologically processed, we can retrieve its position in the image. To do so we need to find the white object on a black background. In other words, we need to extract contours of objects from the image.

Contour can be described as a curve joining all the continuous points (along the boundary), having same color or intensity [13]. Contours found on an image are shown in Figure 10. The function retrieves contours from the binary image using the algorithm [14]. Contours are represented as a list of points indicating the position of a white region. This is exactly, what we were looking for.



Binary image

Extracted contours

Figure 10: Example of extracted contours.

After this final step of motion detection, we know exactly the placement of performed motions.

4 Implementation / architecture

4.1 Used terminology

Rectangle – when any difference between frames is detected, we frame the contour of the difference with the rectangle.

Overall boundaries – rectangle representing the boundaries between which some motor was moving so far. Is built as a compilation of all detected rectangles.

Investigation runs – number of motor runs, during which we collect data about the motor movement. Once the number of runs exceeds those investigation runs, statistics such as trajectory and overall boundaries are stopped collecting.

Statistics evaluator / evaluator – program, which evaluates generated by the algorithm statistics.

4.2 Current state of the project and ways of improvement

Since the purpose of this thesis is purely exploratory, current state of the projects is not fully functional and production ready. At the moment developed program can take an input video, extract the information about motors and indicator lights out of it and save those statistics into a text or a JSON file. If described method of motion detection would prove itself useful, many useful functions can be added:

1. **Detecting motions on a video stream:** currently algorithm can work only with prerecorded videos, which is not very useful in practice, since the requirement is to detect errors in the real time.
2. **Sockets:** in case if the proposed algorithm and the process of generated statistics evaluation are run as different programs, connection between them is required. Such connection can be established via network sockets. A socket can be described as an endpoint of a two-way communication link between two programs running on the network [15].
3. **Flushing the output data to another process:** currently statistics about detected motions are saved only after the video ends. If we were to detect motions on a video-stream, no statistics would be saved or sent to a statistics evaluator. To be able to build proper continuous communication between the algorithm and the evaluator, we need to send those statistics periodically during the process of detection. For example, every 5 seconds the algorithm generates statistics in a form of json structure and sends them to the evaluator via sockets.
4. **Cleaning the flushed data to preserve memory usage:** since a video-stream can be a very long process, many statistics can be generated during that time. Moreover, more statistics are

stored in memory, slower the algorithm works. To prevent such behavior, we would need to clean the data, which were flushed to the evaluator. Those data will not be further used and can be safely cleaned.

5. **Testing:** since programmers are a human being, their code is not error-proof, and it needs to be checked not only visually, but programmatically as well. To be sure, that every piece of code works, as the developer intended, at least unit tests covering the code should be written. Such tests could greatly uncover the undesired behavior or code imperfections. For testing purposes standard python library **unittest** can be used [16].

4.3 Language, libraries and code principles

4.3.1 Python and libraries

Since the main purpose of the thesis is purely investigation, instruments should be as easy and ready to use as possible. Thus, as a programming language I chose Python, which is though slow, but very functional language. It can be used in lots of directions and has many useful libraries, which can be used for computer vision, efficient computations etc.

Specifically, for this project I used single third party python library, which was ported from C++: **OpenCV**. As the developing team writes about OpenCV (Open Source Computer Vision Library), it is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products [17]. Python OpenCV is built on top of C++ OpenCV and uses a library for efficient computations **numpy** [18].

4.3.2 OOP paradigms used in the project

To make the code readable and reusable I used different OOP techniques [19] such as:

- Using objects for representing different layers of the process: **MotionDetector**, which is the main aggregator of all motions, **Motion**, which represents single motion and has several parameters corresponding to each motion and functions for motion processing. In addition, some enumerators were used, which are simple data structures, which represent a set of names holding some unique values [20].

- **Abstraction**. End user should have as easy way of handling the process as possible, thus a user works only with a class, which exposes just key functions to a user and hides another functions, classes and modules, which do things to which user should not have any access.
- **Encapsulation**. Even though Python does not have explicit protection of object members, we can simulate it just by using the convention by prefixing the name of a member by a single underscore "_", which should signalize to the end user, that the member must be used only inside the object [21].
- **Inheritance**. This principle allows classes inherit some behaviors or properties of their parent classes.

4.3.3 Project documentation

Probably one of the most important things in projects of such size and complexity is the documentation. A well-documented code not only increases the readability to the creator, but also simplifies the understanding process for any other programmer, who would like to contribute to the development. Thus, every class and every method are documented not only by this thesis, but also directly in the code.

4.3.4 The reason of pneumatic cylinder flow simulation

Since the thesis is written during the outbreak of the COVID-19 and the quarantine, it is not possible to test the algorithm on the real-world videos and was forced to come up to any other way. The best solution for such problem was to simulate the motors' flow using moving 3D models.

To simulate the process, I used **Autodesk Inventor** software, which has useful tools for making those elements move and for creating videos out of them. I took the motor 3d models from the **Festo** website. Indicator lights on a PLC panel were simulated as cylinders appearing from the black box.

To make the process look more natural and not as a render non-homogeneous background was used with some random picture of a factory. In addition, some distortion was applied to video to make it look as if the process was to be going on in some factory with shaking floor.

4.4 Setting up the project before start

As the first step, Python interpreter should be installed in order to run the program.

Secondly, install the project dependencies by navigating to the project's root folder and running the command **pip install -r requirements.txt**

To begin working with the project in its current state several parameters can be specified:

- **input_path:** path to an input video, on which the algorithm should be applied
- **output_path:** path to the output result. This is the demonstration of how the algorithm detects motions
- **recorded_runs:** this parameter specifies the number of runs, which should be done before the algorithm can extract some information about the process. For example, we cannot be sure that the extracted after one run trajectory of a motor would be correct, since it can be distorted by different noises, vibrations or motor can start running not from the end position. The more such runs, the better the result. At the same time, big number of runs is not required. Recommended number is 4 so that the motor could reach both endpoints at least twice.

Additionally, some constants can be set for better detection if needed:

- **Threshold value for motion detection**: this is one of the most crucial parameters for the detection, since it determines how visible should the motion be to be detected. For example, if the background has the color almost identical to some motor color, we would like to have such threshold lower, so it could detect such difference. Another example is the shadow of a motor on a surface where the motor is placed: it is less visible than the motor itself, thus we would want to make the constant bigger, so it would detect only the motor. Represented as **THRESHOLD_VALUE.**

- **Window size for new or existing motions detection:** since the algorithm needs to detect motions on each frame on the video, we should know whether detected motion already exists or is new. This constant lets the algorithm know how close the detected motion's mass center should be to any existing motion mass center or borders to be considered as the same existing motion. If the distance from each existing motion is bigger than this constant, consider this detected motion as new. This parameter should be set in case of different camera's definitions or distance from camera to the recorded environment. Represented as **DETECT_MOTION_WINDOW_SIZE.**
- How many rectangles for any given motion should be detected before algorithm should decide, whether the motion is a motor or an indicator light. The same constant is required for determining the orientation of a motor. Represented as **RECTANGLES_FOR_TYPE_DETECTION.**
- **Amount of border points:** since we want to determine the trajectory of a motor for further processing steps, we need to know the endpoint on both sides of the motor run. We cannot rely on single extreme on each side, since distortions or vibrations can affect it. Thus, we want to take the weighted value from several runs. This constant determines how many values we want to store for taking the median value from. Represented as **AMOUNT_OF_BORDER_POINTS.**
- **Number of detected rectangles to detect direction:** since we want to be able to detect a direction of a motor, which would not be affected by noises or vibrations, we need to once again take weighted value of several last mass centers. Taking a median value would help to eliminate outliers from the process of direction detection. The bigger value of this constant – the better, but bigger value will

result in less detected directions, which can lead to data loss. Represented as **RECTANGLES_FOR_ORIENTATION_DETECTION.**

- **Window size of pixels for detecting movement change of a motor:** the direction of a motor's movement should be determined just for a moving motor. Considering that video can be noisy and vibrating, we should detect only movement changes that are bigger than specified window of pixels. For example, imagine that the motor is static at the current frame, but due to the vibration, some small movements are detected for this motor. For example, for given 5 frames the motor moved for 2-5 pixels each frame. We can define the constant so, that it states that the motor needs to move at least 10 pixels to be threaten as moving for a given time. The more the value is – the better direction will be detected. Taking to big number is not recommended, since it can lead to loss of data. *This value is dependent on a camera's definition.* Represented as **DIRECTION_WINDOW_SIZE__PIXELS.**
- **Window size of directions to detect a single run:** this number defines how many directions should be taken for it to be able, whether the motor reached the endpoint. The more the value is – the lesser is the noises and vibrations influence on detecting the endpoint. This constant depends on a previous constant, since if previous value is too big, a single run would not contain many detected directions, and this value should be decreased. This value should be reasonable, since if the window is too big it could "see" both endpoints at the same time, which would lead to no detecting any endpoint at all. Represented as **ENDPOINT_WINDOW_SIZE__ELEMENTS.**

4.5 Starting the project

If all presets stated in the previous step are finished, user can start the project simply as a Python script by navigating to the projects folder writing in the console **python**

src/motion_detection.py. User can also specify parameter **input_path, output_path** and **recorded_runs,** which were described in the previous chapter.

4.6 Possible user flow improvements

To make the project as easy to use and setup as possible, several additional features can be added:

- **Graphical user interface** for better constant setting experience and for providing a way to specify input and output path for videos.
- **Containerizing the project or converting it into an application** so that the only thing a user would need to do is to install the project and run, without a routine of installing the python interpreter and dependencies.
- **Automatic constants adapting** depending on the surrounding conditions. This would require a test run so that the program could adapt.

5 Documentation and specification

5.1 Project structure with all models

In term of files and directories project has following structure:

```
pneumatic_detection/
│
├── data/
│
├── src/
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── constants.py
│   │   ├── paths.py
│   ├── __init__.py
│   ├── models.py
│   ├── motion_detection.py
│   └── video_utils.py
│
└── requirements.txt
```

- **pneumatic_detection:** root directory, where all project files are situated
- **data:** directory where all generated or input videos may be situated in case of such preset
- **src:** directory with the source code
  - **utils:** package where user can set default constants and paths of processed/generated videos
    - **__init__.py:** file that is required for directories to be packages. Inside the file are all imports, which will be reused in other project files. It is required so that if the user needs something from the **constants.py** file, the only path is required to specify is the **src.utils** instead of **src.utils.constants.** It also hides some things from the **paths.py** file, so that they would not be available for the end user. Using this file user can import all constants from the constants file and only input and output paths from the paths file.
    - **constants.py:** inside this file are all constants, which need to be tuned for proper project run.
    - **paths.py:** contains the default input path of a video that will be processed and an output path of a video that will be generated after program run.

    - **models.py:** inside this module are all data structures required for motion detection. More about this package is described above
    - **motion_detection.py:** main file of the project. It starts the project, keeps track of the current processed frame, and commands to output the statistics and other things that will be described below.
    - **video_utils.py:** has useful functions for frame preprocessing and data structures for keeping the input video metadata.
- **requirements.txt:** standard file containing names of all required packages. It is required for simplifying the process for the end user, so that the user do not need manual installation of all the dependencies of the project.

5.1.1 motion_detection.py

This module should be started in order to run the program. Inside the module is created a motion detector object and a loop, which iterates through each frame of an input video. Inside the loop, the frame is first read, and then it is gray scaled and sent to the motion detector. In addition, for each frame some information is printed on the output video, such as motion type, its trajectory etc.

After the last frame of the video is processed, all statistics are printed to the `statistics.txt` file. In the future instead of file, writing can be flushing data to the further process via sockets.

This module uses objects and functions from **video_utils.py** and **models.py** files.

5.1.2 video_utils.py

Inside this module is a data structure **VideosMetadata** to contain an object of input video, an object of output video, info about first frame represented as a matrix of values, which size if width*height of the input video and an amount of frames of the input video. Number of frames is required for the loop inside main module.

This module also contains a function to grayscale a frame and function to extract all required metadata from the input video. Latter function returns **VideosMetadata.**

All functions or classes of this module are exposed to the user.

5.1.3 models.py

This module contains main data structures required for motions detection. Main class and the only one that is exposed to the user is **MotionDetector.** This is the only class, which should be used by the end user, since it has all required for motion detection methods.

To initialize the objects of this class user needs to pass two arguments: **first_frame**, which will be compared to all further frames, and **max_amount_of_recorded_runs**, which specifies, how many runs should be made before motion parameters detection.

As an additional implicit property, this class stores a list of detected motions: **detected_motions.**

This class exposes several methods for a user:

- **detect_motions_on_frame:** this is the main function, which starts the process of detecting motions on any frame that is passed as an argument. Frame is represented as matrix of values. Frame should be grayscaled before passing into the function.
- **draw_motions_on_frame:** function, which takes an information of each motion and on each frame, draws its id, boundaries, trajectory and type.
- **generate_statistics:** as the name implies generates statistics of all motions tracked by motion detector. Calls the same function for each motion.
- **draw_plots:** generates plots of steps for each motor. Since we cannot be sure, whether is the motor expanding or not (I forgot the antonym), graphs show the distance motor traveled from the point that was detected at the start, thus it is not the same as step graph.

This class also has some private helper methods, which should not be exposed to the user and should only be used by other methods:

- **_update_all_existing_motions_with_zero_placeholder:** this is the first step, which is called in **detect_motions_on_frame** public function. Purpose of this method is to update moving history for all existing motions placing 0 placeholder. This step should be made in order to update the motion's history even if the motor is in the same position on the current frame as on the first frame. This method calls the same but public method for each motion.
- **_detect_and_update_existing_motions_or_create_new:** this is the second step performed in **detect_motions_on_frame** method. This private method starts the detection of motions on the current frame. Two arguments should be provided: **gray_frame**, which is the current frame, and **frame_index**, which indicates the number of the current frame and which is used for motion's history. It performs two steps:
    - find rectangles of detected motions and their mass centers
    - for each detected pair rectangle-mass center determine, whether detected pair corresponds to an existing motion or a new motion is found.
- **_get_rectangles_mass_centers:** finds and returns a list of rectangle-mass center pairs of all found motions on the current frame. Takes single argument: **gray_frame** – current grayscaled frame.
- **_get_rectangles:** finds rectangles (boundary boxes) of all detected motions. More details on this function will be provided in the following chapter. Takes the matrix of the current frame as an argument.
- **_create_new_motion:** if no existing motion is detected, new motion should be created. Creates a new instance of **Motion** class, and then saves it to the existing motions list. Every new motion is created with an id of the length of existing motions. Takes three arguments:
- **rectangle:** rectangle of the first detected motion
- **mass_center:** of the current detected rectangle
- **frame_index:** number of the current frame. Required for the motion history.

- **_draw_rectangles_on_frame** and **_draw_trajectory_on_frame:** when the process of motion detection is finished, these functions draw found characteristics on the current frame.

Second important class is **Motion,** which represents a detected motion. This class stores all motion properties and has all required methods for motion processing. This class should not be exposed to the end user, since user should not work with any single motion separately, because it can create unpredictable results and anomalies in resulting data.

The user should not create instances of this class manually and such functionality should be left on **MotionDetector** class, which creates a new instance of **Motion** class each time it calculates that detected motion is new. The instance is created inside the method **_create_new_motion** of the **MotionDetector** class.

To initialize the instance 4 arguments should be provided:

- **record_id: int,** this parameter is crucial for further processing, since it would provide the information about which motor/blinker is dysfunctional or any other useful information to the end user or observer.
- **initial_rectangle: List[int],** this list represents the boundary box of the first detected difference between the initial frame and current frame for given motion. We need this information to initialize the borders for the motion. Those borders will be used to keep track of the boundaries inside which the motor can move.
- **initial_mass_center: Tuple[int]** this argument is required to keep track of the distance that the motor would travel during the moving periods. All further mass centers will be compared to this one, and the distance between them will be calculated.
- **motion_history: List[float]** if a motion wasn’t moving till the current frame, we need to mark it in the history by passing a n umber of zeros that is equal to the number of frames, which were prior to the current frame. So, when we initialize a new motion, we simply pass a list of zeros.

Each class instance has many parameters, which are required for processing:

- **record_id: int,** is filled by the passed **record_id** argument
- **initial_mass_center: Tuple[int],** is filled by the passed **initial_mass_center** argument
- **current_mass_center: Tuple[int],** this number defies, how far is the position of the mass center for the current frame
- **motion_history: List[float],** is initialized with **motion_history** argument. This parameter is required for statistics, for direction detecting and for defining motion orientation. Stores values of the distance between the current mass center and the initial mass center.
- **amount_of_runs: int,** when a new instance is created, this parameter is initialized with a value of 1, since the instance is created when 1 run is detected.
- **current_median_position: float,** represents a value of distance from the initial mass center at the current frame. Initialized with 0 value, since when new instance is created, no distance is traveled from the starting point.
- **current_median_mass_center: Tuple[int],** represents the median position value of last mass centers. Median position should be calculated to ignore the anomalies caused be vibrations and video noises.
- **median_position_history: List[float],** stores a list of median positions of last mass centers. Required only for plot drawing.
- **last_detected_endpoint: Direction,** represents what was the last reached endpoint by the motor. Tells the user, whether motor reached an end position of any directions from the list: [LEFT, RIGHT, UP, DOWN].
- **most_left_point, most_right_points, most_up_points, most_down_points: Set[Tuple[int]],** those parameters are needed for trajectory calculation. This is a set of distinct values of mass centers, which stores an amount of points determined be the **AMOUNT_OF_BORDER_POINTS** constant. For vertical

motions it keeps track of most left and right points, most up and down points are not update. For horizontal motions, it is vice versa.

Those values are initialized with some values, which cannot be possible for any screen resolution. Default is **(-100000, -100000)** for most right and down (closer the point to the lower edge is, bigger the coordinate is) points, **(100000, 100000)** for left and up.

- **mass_center_history: List[Tuple[int]],** keeps track of all mass centers of detected motion contours. It is required for statistics and direction detection. Is initialized with an empty list, since no rectangles were detected except the initial one.
- **motion_direction_history: List[Direction],** this list represents the history of which directions the motor travels. Direction describes the vector in which the motor traveled for the current frame comparing to previous frames. This is required for endpoint detection.
- **trajectory_from: Point,** represents the abstract starting point of the moving trajectory. This value should not be the actual starting point of the trajectory, but some point, which is one of two main endpoints.
- **trajectory_to: Point,** this point is the opposite of the **trajectory_from**. Same abstract point, which is located on the other side of the direction.
- **orientation: Orientation,** represents, which direction is main for any given motor. Motor can be either vertical or horizontal. This value is determined by the rectangle of the motor. If the difference between left and right boundaries is bigger than between lower and upper, motor has horizontal motion. And vice versa. This is an enumeration value. Default value is Orientation.NOT_DEFINED.
- **motion_type: MotionType,** each detected motion can be either a motor or a blinker. This parameter holds exactly this information. This parameter is determined by the mass centers of a motion: if a motion has some detected mass

centers but most of them are null, that means that the motion is a blinker. Default value is MotionType.NOT_DEFINED.

- **trajectory_is_defined: bool,** is a simple flag, which tells, whether for a given motion method calculating a trajectory has already been run and the trajectory is defined. Required to detect the trajectory only once for each motion to reduce the amount of calculations.
- **most_left_edge, most_right_edge, most_up_edge, most_down_edge: int,** these values are required to keep track of the boundaries of the overall motion's movement. Each time the new rectangle, which has any boundary exceeding any of those values, is detected, exceeded boundary is updated.

Methods of the **Motion** class will be described in the next chapters.

This module also has several helper enumerations and functions:

- **Direction: enum,** represents the direction of the motor's movement. Have basic directions: **left, right, down, up**, and mixes of those basic directions: **left_up, left_down, right_up, right_down.** Each direction has an integer value corresponding to it. Directions have following values: **left = 2, right = 3, up = 5, down = 6**, **left_up = 10, left_down = 14, right_up = 15, right_down = 21.** Those integer values are not random and are specified for basic direction detection simplification. Basic directions have rising prime numbers and mixes have a multiplication of those prime numbers. So if the user would want to determine whether the direction is **up**, all the user need to do is to check whether the direction can be divided by the value of **up = 5.** For example: 5 % 5 = 0, 10 % 5 = 0, 15 % 5 = 0 and no other value would give 0 from such an operation. Aside from the method, which determines basic direction, this class also has a method, which compares two directions and returns true if they are the same, and a method, which returns the opposite direction.
- **Orientation: enum,** holds the values describing which directions can the motor have. Holds 3 values: **vertical, horizontal** and **not_defined.** Last one is the

default value for all motions, which orientation is not defined for the current frame.

- **MotionType: enum,** is used to store values, which describe whether a motion is a motor, a blinker or whether it is not defined for the current frame.
- **_find_distance: function,** takes two mass centers represented as points **p** and **q** and calculates the Euclidian distance between them:

$$d(p,q) = d(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Class diagram for all described classes is shown in Figure 11.

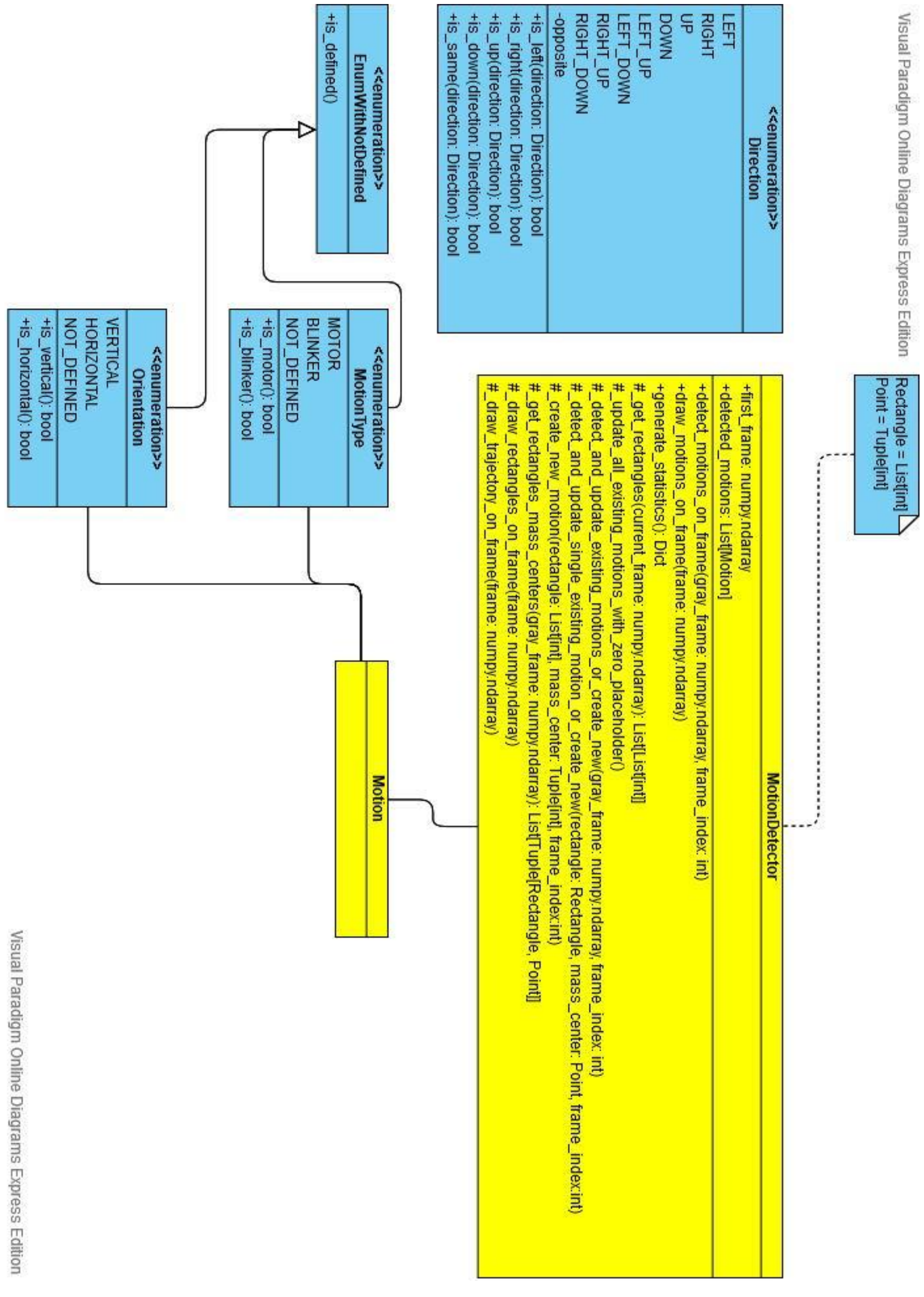

Visual Paradigm Online Diagrams Express Edition
<<enumeration>>
Direction
LEFT
RIGHT
UP
DOWN
LEFT_UP
LEFT_DOWN
RIGHT_UP
RIGHT_DOWN
-opposite
+is_left(direction: Direction): bool
+is_right(direction: Direction): bool
+is_up(direction: Direction): bool
+is_down(direction: Direction): bool
+is_same(direction: Direction): bool
<<enumeration>>
EnumWithNotDefined
+is_defined()
Rectangle = List[int]
Point = Tuple[int]
MotionDetector
+first_frame: numpy.ndarray
+detected_motions: List[Motion]
+detect_motions_on_frame(gray_frame: numpy.ndarray, frame_index: int)
+draw_motions_on_frame(frame: numpy.ndarray)
+generate_statistics(): Dict
#_get_rectangles(current_frame: numpy.ndarray): List[List[int]]
#_update_all_existing_motions_with_zero_placeholder()
#_detect_and_update_existing_motions_or_create_new(gray_frame: numpy.ndarray, frame_index: int)
#_detect_and_update_single_existing_motion_or_create_new(rectangle: Rectangle, mass_center: Point, frame_index:int)
#_create_new_motion(rectangle: List[int], mass_center: Tuple[int], frame_index:int)
#_get_rectangles_mass_centers(gray_frame: numpy.ndarray): List[Tuple[Rectangle, Point]]
#_draw_rectangles_on_frame(frame: numpy.ndarray)
#_draw_trajectory_on_frame(frame: numpy.ndarray)
<<enumeration>>
MotionType
MOTOR
BLINKER
NOT_DEFINED
+is_motor(): bool
+is_blinker(): bool
<<enumeration>>
Orientation
VERTICAL
HORIZONTAL
NOT_DEFINED
+is_vertical(): bool
+is_horizontal(): bool
Motion
Visual Paradigm Online Diagrams Express Edition

Figure 11: Project class diagram.

5.2 Flow of a single cycle of motions detection

1. The process starts with reading the frame and converting it to gray shades.
2. The process of finding motions starts with the **MotionDetector's** method **detect_motions_on_frame**
3. Inside this method, two other methods are called: **_update_all_existing_motions_with_zero_placeholder** and **_detect_and_update_existing_motions_or_create_new.** The former one is needed to update all existing motions with a zero value in their movement history. This step is performed to ensure that not only the history of moving on the current frame motors will be updated, but the history of motors, which are on the same position as on the first frame, as well. Latter step detects motions and either updates existing or creates new ones.
4. Inside **_detect_and_update_existing_motions_or_create_new** the first thing made is getting rectangles (boundaries) and their mass centers of detected motions. If no motions are found, no rectangles are found as well, which would mean, that no further steps are performed. However, if at least one motion and its rectangle is found, we proceed to the next step.
5. To each detected trio (rectangle, mass center, and frame index) the class applies the method **_detect_and_update_single_existing_motion_or_create_new**, which purpose is to compare given mass center to all existing motions' mass centers. If a motion, which is close enough to the passed as a parameter mass center is found, this motion is update. Otherwise, new motion is created.
6. If any existing motion was detected on the current frame, we need to update the information about it. This step is made by calling **update_record_data** method of a motion, which is represented by **Motion** class instance, which was detected. This method updates the motion in several steps by calling following methods of **Motion** instances:
   6.1. **_update_motion_history**, which only takes a mass center of detected motion and index of the current frame as parameters. This function simply updates the mass center history, current mass center and replaces a zero value in motion history, which was add to this

history in the second step, with a real calculated value of a distance between the initial mass center and the current one.

6.2. **_detect_motion_type**, which determines whether the motion is a motor or a blinker. If the motion is a blinker, we do not need to perform further steps, since the only information we need to know about the blinker is the position of the blinker, which should not change, since it is not a moving object and its id. If the motion is a motor, the process continues.

6.3. If the amount of runs of the given motor does not exceed the maximal amount of runs predefined for updating motion's parameters, method _**update_edges** should be run. This method expands the overall boundaries pf the motion if one or more boundaries of the current rectangle exceed the overall motion boundaries.

6.4. If orientation of the motion is not defined, it should be defined first by calling **_detect_orientation** method, since the information about the orientation is required for next steps. Orientation can be defined only if required amount of mass centers is present in the motion's history. If orientation is already defined, we can proceed to further steps.

6.5. If number of runs does not exceed the amount of runs predefined for information extraction, next step is to call the method **_update_border_points**, which updates border points of the motor. If the orientation is vertical, lower and upper points are updated. If the motion is horizontal, most left and most right points are updated.

6.6. Next step is to update the direction where the motor has traveled in the last frames. This can be done by **_detect_and_save_direction**, which is required for the following step.

6.7. To know whether a motor made a whole run from one endpoint to another, we need to check whether any of endpoint was reached. This is performed by the method **_detect_and_save_endpoint**.

6.8. If a motor reached predefined number of runs required for information extraction, we can calculate the trajectory of movement of the motor.

7. When all parameters of all motions were updated, we can draw them on a frame for a better visual representation for an observer. This is done by calling the method **_draw_rectangles_on_frame**, which takes a matrix of the current frame as an argument.

This method for each existing motion draws an overall rectangle of boundaries, mass center of the current rectangle and a text of an id of a motion and its type. Next step is to display a trajectory of all motor movements. This is done by calling the method **_draw_trajectory_on_frame**, which simply draws the straight line between the starting point of the trajectory to the end point.

8. When the video processing is finished, statistics of all found motions need to be generated by calling the public method **generate_statistics.** Those statistics can be then sent to the further process or saved to a file.

Full cycle is shown below in Figure 12.

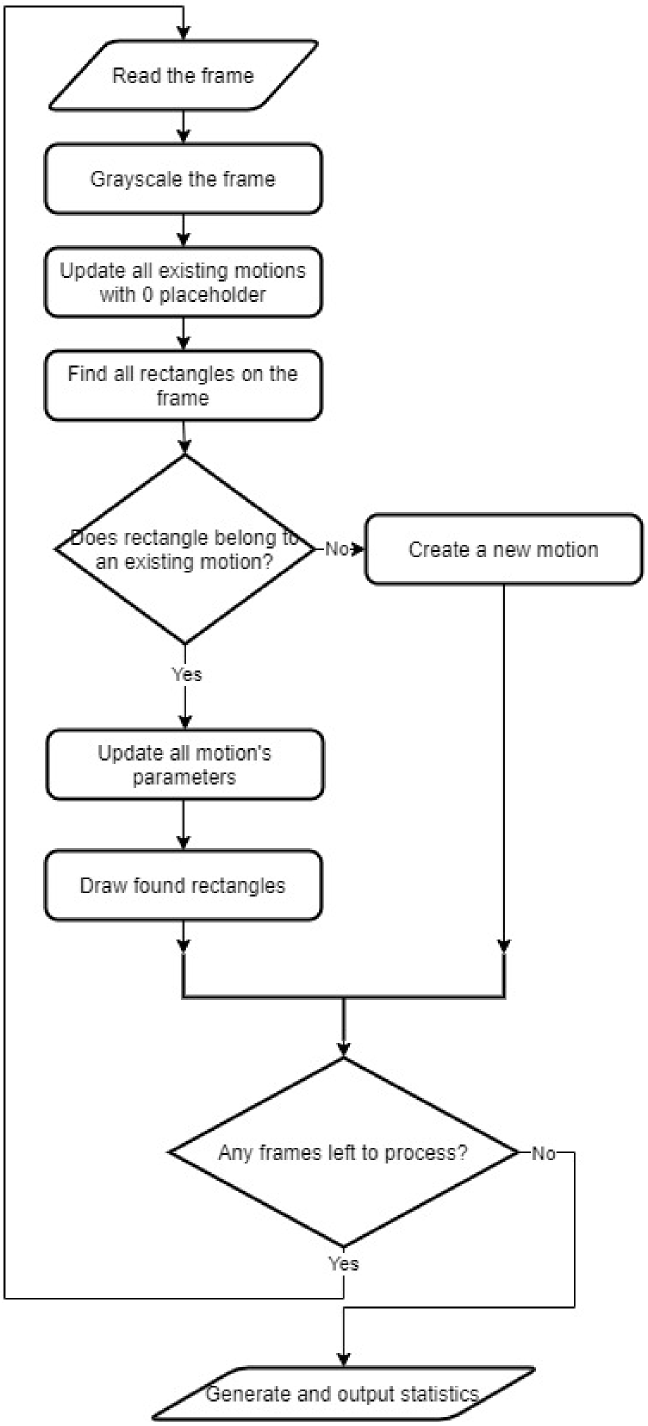

Read the frame
Grayscale the frame
Update all existing motions with 0 placeholder
Find all rectangles on the frame
Does rectangle belong to an existing motion?
No
Create a new motion
Yes
Update all motion's parameters
Draw found rectangles
Any frames left to process?
No
Yes
Generate and output statistics

Figure 12: Single cycle of motion detection.

5.3 The way to find rectangles of motions

One of the most crucial steps after determining the presence of any motion is to detect its current boundaries, which are necessary to get the current mass center of a motion and to expand the motion's overall boundaries, which would be useful to know the exact borders between which the motion can move.

```
def _get_rectangles(self, current_frame: np.ndarray) -> List[List[int]]:
    # compute the absolute difference between the current frame and
    # first frame
    frame_delta = cv2.absdiff(self.first_frame, current_frame)
    thresh = cv2.threshold(frame_delta, MotionDetector._threshold_value, 255, cv2.THRESH_BINARY)[1]
    # dilate the threshold image to fill in holes, then find contours
    # on threshold image
    kernel = np.ones((3, 3), np.uint8)
    erosion = cv2.erode(thresh, kernel, iterations=1)
    dilation = cv2.dilate(erosion, kernel, iterations=1)
    contours = cv2.findContours(dilation, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours = imutils.grab_contours(contours)
    contours = filter(lambda contour: cv2.contourArea(contour) > 5, contours)
    rectangles = list(map(cv2.boundingRect, contours))
    return rectangles
```

The first step is to calculate the absolute difference of the initial frame and the current one. This is done by a single OpenCV command **absdiff** [22] on line 2

After the absolute difference is calculated, we should specify, what we consider as a motion by calling the threshold method. This function takes the frame delta as a picture, on which the threshold will be calculated, two integers representing the minimal and the maximal borders (note that the maximal value is hardcoded to be 255, since it is the maximal possible value for any pixel value. This value represents white color). As the last argument, we should specify the type of the threshold.

As we want the found threshold image without any noises, we need to apply the erosion and dilation on the found threshold. We use the kernel 3 by 3 for those purposes. We make only a single iteration for each function, since we want to preserve smaller movements such as blinkers and we want to simplify the computations.

To extract the contours, we need to call a function **findContours** [23], which takes 3 arguments: frame, from which contours should be extracted, parameter specifying that we want to retrieve only the extreme outer contours, and which contour approximation algorithm to use. In our case, we use the algorithm, which compresses horizontal, vertical, and diagonal segments and leaves only their end points.

Once the contours are found, we simply need to check whether those contours are not simple micro noises by comparing their size to some artificial value. This step can be skipped, since we already did a good job on noise filtering in the previous steps.

The last action is to find the bounding rectangle for each found contour and return a list of such rectangles.

5.4 Processing of previously found rectangles

5.4.1 The way to determine whether a motion exists

Once we find the list of rectangles, we need to inspect, whether detected rectangles correspond to any existing motions or a new motion should be created.

To determine whether any rectangle correspond to an existing motion, we should take an assumption, that if the mass center of the found rectangle is inside the extended boundaries of the overall rectangle of any existing motion, we consider this found rectangle corresponding to that motion.

Example of a mass center is shown in Figure 13. Mass center of any rectangle can be found as follows:

$$\bar{x} = \frac{x_1 + x_2}{2}; \; \bar{y} = \frac{y_1 + y_2}{2}$$

If the detected rectangle is outside of any overall rectangle – new motions should be created.

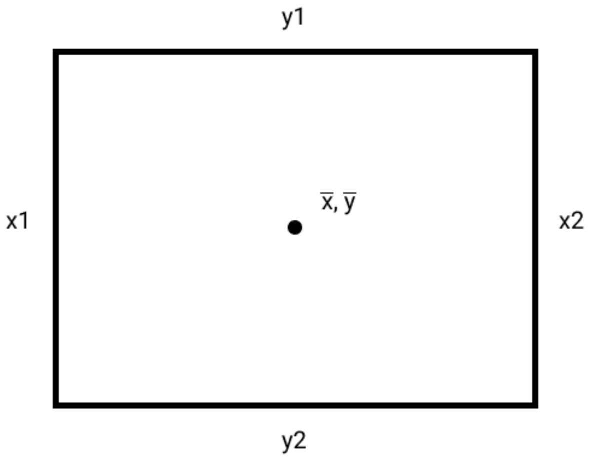


Figure 13: Representation of rectangle's mass center.

5.4.2 Creation of new motions

If the detected rectangle does not belong to any existing motion, we need to create the new one. This is done by creating an instance of Motion class and initializing it with a record id, which is equal to the length of already existing motions plus one. Initial rectangle is the detected rectangle, initial mass center is the mass center of the detected rectangle, and motion history is the list full of zeros, which length is equal to the number of passed frames.

5.4.3 Updating existing motions

Once the program defines that the detected rectangle belongs to an existing motion, we need to update that motion in order to detect such things as whether it is moving or whether it made whole run from one side of its trajectory to another and other parameters.

5.4.3.1 History

First thing to update is motion history, which includes updating motion mass center history, history of distances from the initial mass center and current mass center. Updating the history is necessary for further parameter updates.

5.4.3.2 Detecting the motion type: motor or indicator light

This property of a motion can be detected only in case if number of detected mass centers is bigger than the predefined constant **_min_amount_of_rectangles_for_type_detection.** This condition is used because we do not want such detection to be performed too early. We rather take a reasonable amount of movements, defined by a window of size of previously stated constant, from the motion history. Then if most of such movements are less than a value of another constant **_direction_window_size_pixels**, which determines whether a movement was performed, the motion is detected as a blinker, which must be static (but can have small fluctuations due to noises). Otherwise, if motion moves in distances bigger then that constant, motion is detected as a motor.

This is the last step of updating if the motion is detected to be a blinker. All latter described steps are relevant only for a motor.

For example, imagine that the value of the constant **_min_amount_of_rectangles_for_type_detection** is equal to 10; value of the constant **_direction_window_size_pixels** is equal to 2 pixels. We can say that any distance smaller than 2 is insignificant. The program should detect 10 rectangles for a motion and save their mass centers to the history of mass centers. When the history of mass centers has required 10 elements, we can filter those, which distance from the initial mass center is less than 2 pixels. If number of such mass centers is bigger than the half of the **_min_amount_of_rectangles_for_type_detection** (it means that in given example the number should be at least 6), we can say, that even though mass centers were detected, object didn't travel any distance at all or travelling was caused by the noise and is insignificant, thus the element is a blinker. Otherwise, the element can be classified as a motor.

We compare the travelled by the mass center distance to the constant **_direction_window_size_pixels** just because noises of the camera can fake the movement of an element, which results in a non-zero value of the distance between the current and the initial mass centers even for static objects such as a blinker. If we are comparing the distance to zero, we will almost always get a number of significant distance bigger than the half of a window, thus every object would be detected as a motor.

Pseudocode for this step would be:

```
If len(mass_center_history) == _min_amount_of_rectangles_for_type_detection:
    last_movements = motion_history[-_min_amount_of_rectangles_for_type_detection:]
    non_moving = last_movements.filter(movement_distance <
_direction_window_size_pixels)
    if len(non_moving) > ceil(_min_amount_of_rectangles_for_type_detection):
        motion_type = BLINKER
    else:
        motion_type = MOTOR
```

5.4.3.3 Updating the overall boundaries of any motion

When a motor is initialized, its boundaries of motion are defined by the initial rectangle. However, whenever any rectangle is detected, it can expand those boundaries.

The process of expanding is very simple: if any side of the detected rectangle exceeds any corresponding side of the current overall boundaries, update that edge of the overall boundary with the edge of the detected rectangle.

An example is illustrated in Figure 14. Overall boundaries, which are represented as the rectangle with black stroke, of some motion have the y coordinate of the upper edge **yo1** and x coordinate of the rightest edge **xo2**. Current rectangle, the one that has the red stroke, has y coordinate of the upper edge **yc1**, which is above the y coordinate of the upper edge of the overall boundaries. In addition, x coordinate of the rightest edge of the current rectangle is greater than the x coordinate of the rightest edge of the overall boundaries. Therefore, overall boundaries must be updated with the new coordinates.

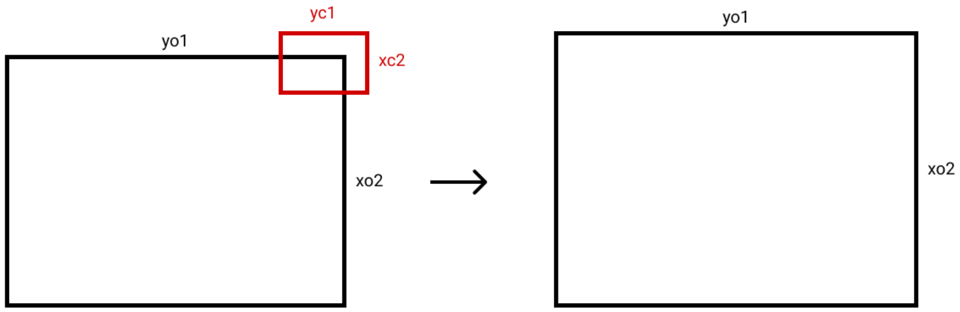


Figure 14: Overall boundaries expansion.

Part of the pseudocode to describe this algorithm can be written as following:

```
if rectangle.rightest_edge > overall_boundaries.rightest_edge:
    overall_boundaries.rightest_edge = rectangle.rightest_edge
if rectangle.upper_edge < overall_boundaries.upper_edge:
    overall_boundaries.upper_edge = rectangle.upper_edge
…
```

In the same manner, lowest and most left edges are updated.

5.4.3.4 Detecting the orientation of a motor and why it is necessary

Before we move to further processing steps, we need to detect the orientation of the motor, which can be either vertical or horizontal. This will simplify the algorithms of detecting end points and calculating the trajectory. The principle of simplification will be described in corresponding points.

The orientation is calculated by the overall boundaries of the motion: if difference between horizontal edges is bigger than the difference between vertical edges, we consider such motion as horizontal and vice versa.

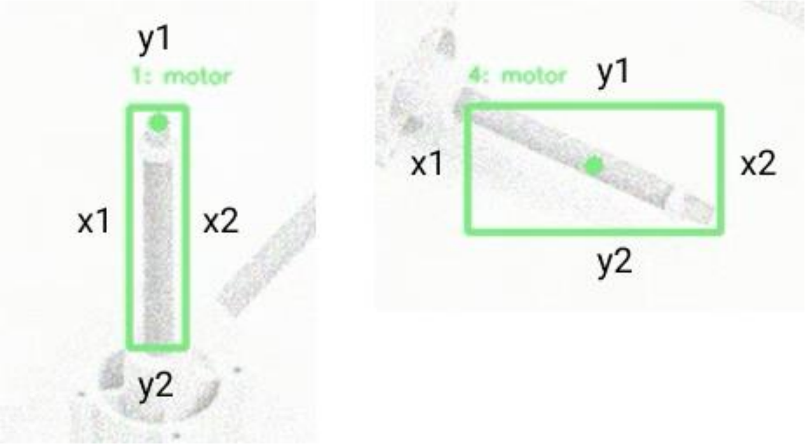


Figure 15: Two types of motors: vertical and horizontal respectively.

As can be seen in Figure 15 on the left-hand side image the distance between vertical coordinates is greater than between horizontal coordinates. That means that the motor has vertical orientation. On the right-hand side, picture distance between x2 and x1 is greater than between y1 and y2, thus this motor has a horizontal orientation.

As soon the orientation is found, it is stored in the instance's property **orientation.**

5.4.3.5 Definition of end points (border points)

Border or end point is the term to describe one of the points, that are on the very end of the motor's trajectory. Such points are required for proper resulting trajectory calculation. Those points represent the extremes of mass centers.

Those points are found during the investigation runs. Algorithm of end points collecting is based on comparing already stored end points with new detected points of mass centers.

For example, let us take the motor with horizontal orientation, which has 5 end points represented as dots in Figure 16. Those points were obtained during several runs as the points with the greatest x coordinate out of all points.

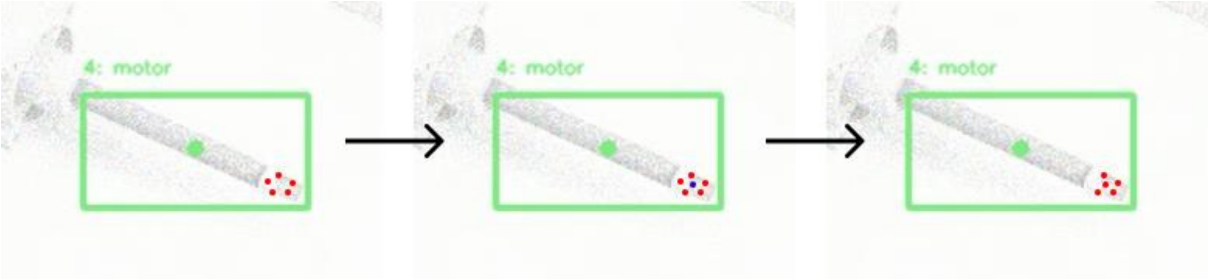


Figure 16: Process of end points updating.

During the current run, new mass center was detected, which is shown as the blue dot on the same Figure 4. Since the motor orientation is horizontal, the only coordinate we need to compare is the x coordinate. As can be seen from the picture, new detected mass center has x coordinate greater than the point with the smallest x coordinate from all endpoints on that end.

After that, we can update the end points so that the current mass center replaces the point with the lowest x coordinate.

The pseudocode for this algorithm would look as following:

```
min_of_most_right = find_min_by_x_coordinate(most_right_points)
max_of_most_left = find_max_by_x_coordinate(most_left_points)
if current_mass_center.x_coordinate > min_of_most_right.x_coordinate:
    most_right_points.add(current_mass_center)
    if len(most_right_points) > max_allowed_amount_of_end_points:
        most_right_points.remove(min_of_most_right)
if current_mass_center.x_coordinate < max_of_most_left.x_coordinate:
    most_left_points.add(current_mass_center)
    if len(most_left_points) > max_allowed_amount_of_end_points:
        most_left_points.remove(max_of_most_left)
```

Important note: when a motor is vertical, the lowest point has the greatest y coordinate, since the image has 0 y coordinate at the top and the biggest value at the bottom.

We need to store only end points of the corresponding orientation: lower and upper for vertical and left and right for horizontal, since points of the opposite orientation are not used.

To keep only distinct values, python data type **set** is used, which purpose is to store only non-similar values. End points are stored in corresponding **most_left_points, most_right_points, most_lower_points, most_upper_points** properties.

5.4.3.6 Detecting the direction of motor movement

Detecting the direction of motor’s movement helps the program to tell, whether a motor reached an end position of one of its two ends. It can also be a useful part of statistics, which would tell an observer, whether a motor stays in place and moves in the correct direction or whether it was misplaced. Unwanted misplacement can cause serious damage to the system and should be detected as soon as possible.

To detect the direction, in which the motor has travelled during some last frames, we simply need to compare the current mass center to some previous mass center. Comparing them by both coordinates would tell us, whether the motor travelled up, down, left or right.

Such an approach can be affected by outliers, which can be caused by the camera noise. For example, let us look at the mass center history, which contains points (0, 0), (10, 10), (20, 20), (15, 15), (30, 30). Obviously, either point (15, 15) is an anomaly caused by noise distortions, since motor cannot get in the opposite direction just for one frame, or such a movement is insignificant. Resulting list of directions would be UP_RIGHT, UP_RIGHT, LEFT_DOWN, UP_RIGHT, even though in reality motor should travel only in the direction UP_RIGHT.

To eliminate the influence of anomalies from the process, we do not want to compare direct values to each other, but rather take a median of a group of such points and compare it to a median of the previously detected group.

Let us compare 3 different methods of detecting the direction.

Setup

- Points = (68, 72), (69, 74), (70, 76), (55, 32), (72, 80), (73, 82), (74, 84), (71, 88)
- Since the orientation of the motor is vertical (y coordinate changes more rapidly than x coordinate), any direction in the DOWN direction is correct (DOWN, LEFT_DOWN, RIGHT_DOWN). Nevertheless, preferred detected direction is RIGHT_DOWN, since it is the true direction of movement.
- Points are shown in Figure 17. Normal points have green color. Outliers are marked as the red points.
- Window size of weighted elements = 5

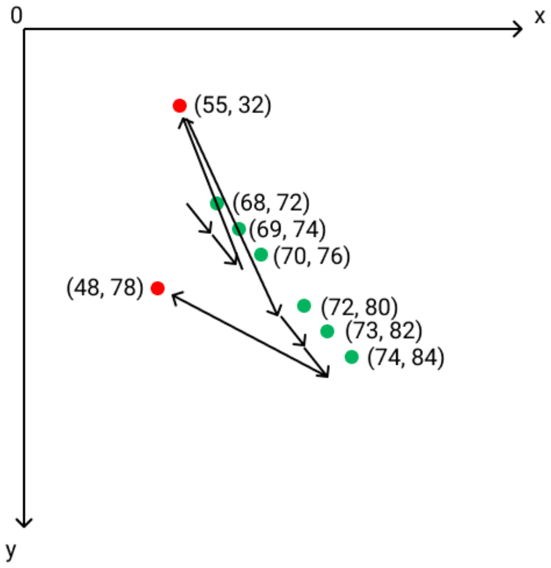


Figure 17: Outliers comparing to normal points.

Direct comparison

(69, 74) - (68, 72) = (1, 2) -> RIGHT_DOWN

(70, 76) – (69, 74) = (1, 2) -> RIGHT_DOWN

(55, 32) – (70, 76) = (-15, -44) -> LEFT_UP

(55, 32) – (72, 80) = (17, 48) -> RIGHT_DOWN

(73, 82) – (72, 80) = (1, 2) -> RIGHT_DOWN

(74, 84) – (73, 82) = (1, 2) -> RIGHT_DOWN

(48, 78) – (74, 84) = (-26, -6) -> LEFT_UP

Number of detected directions: 7

Number of correct detections: 5

Number of incorrect detections: 2

Percentage of incorrect detections: 28,6%

Median and mean comparison

First group: (68, 72), (69, 74), (70, 76), (55, 32), (72, 80)

Second group: (69, 74), (70, 76), (55, 32), (72, 80), (73, 82)

Third group: (70, 76), (55, 32), (72, 80), (73, 82), (74, 84)

Fourth group: (55, 32), (72, 80), (73, 82), (74, 84), (48, 78)

Median comparison

Table 1: Comparing median weighted points

Group	Median point	Traveled distance	Direction
First	(69, 74)	-	-
Second	(70, 76)	(1, 2)	RIGHT_DOWN
Third	(72, 80)	(2, 4)	RIGHT_DOWN
Fourth	(72, 82)	(0, 0)	-

Number of detected directions: 2

Number of correct detections: 2

Number of incorrect detections: 0

Percentage of incorrect detections: 0%

Mean comparison

Table 2: Comparing mean weighted points

Group	Mean point	Traveled distance	Direction
First	(66.8, 66.8)	-	-
Second	(67.8, 68.8)	(1, 2)	RIGHT_DOWN
Third	(68.8, 70.8)	(2, 4)	RIGHT_DOWN
Fourth	(64.4, 71.2)	(-4, 0.4)	LEFT_DOWN

Number of detected directions: 3

Number of correct detections: 3

Number of incorrect detections: 0

Percentage of incorrect detections: 0%

Conclusion

Table 3: Comparing all previous methods

Method	Detected directions	Correct detections	Incorrect detections	Percentage of incorrect
Direct	7	5	2	28.6
Median	2	2	0	0
Mean	3	3	0	0

As we can see from the table in case if data contains outliers, detecting directions by comparing points directly can lead to high percentage of errors. Nevertheless, at the same time, it has the largest number of detected detections. Therefore, it is quantity over quality.

Comparing median values of groups has the least amount of detected directions, but all detected directions are the true directions. This default method is used in the program, since it is more resistant to outliers no matter whether the outliers are close to data or far away.

Comparing mean values results in more detected directions than comparison by medians, but it detected one partially correct direction: LEFT_DOWN. This method is very sensitive to the outliers as well. If any outlier would be too far away from the correct points, it can lead to completely incorrect result.

5.4.3.7 Detecting the number of back-forth runs of a motor

Having previously detected directions, we can easily detect the fact of reaching an end position by a motor, which can tell us, whether a motor made a full cycle.

Let us imagine a situation, where motor's movement is purely horizontal. Last reached end position was on the left side and it is moving to the right. Motor's head is in the middle of its full path; if we take several last directions from the direction's history (a number of directions we take is determined by the constant **_endpoint_window_size_elements**), those directions would be purely RIGHT, since it traveled only in the right direction several last frames. Therefore, its end position remains the same, since it did not reach the opposite end position from the previous one. This is shown in Figure 18 as the first step.

When the motor's head reaches motor path's most right position, we still cannot be sure whether its head will go backwards. History contains only RIGHT directions. Therefore, last reached end position and number of runs remains the same.

After motor's head moved several times to the LEFT directions, motor directions history starts to be populated with LEFT direction. However, since number of LEFT directions is less than most of taken directions, we still do not increase the number of runs and previously detected end position.

After the moment when the motor reaches the right end position of its path, its direction history would be inconsistent and it would have not only RIGHT, but LEFT as well, and it might look like [RIGHT, RIGHT, RIGHT, LEFT, LEFT, LEFT, LEFT, LEFT]. As we can see, number of directions to the left is prevailing, which means, that the motor indeed reached its end position on the right side. Given that its previously detected end position was at the left side, we can tell that it made a whole run from left to right, thus we can increase the number of total runs by 1.

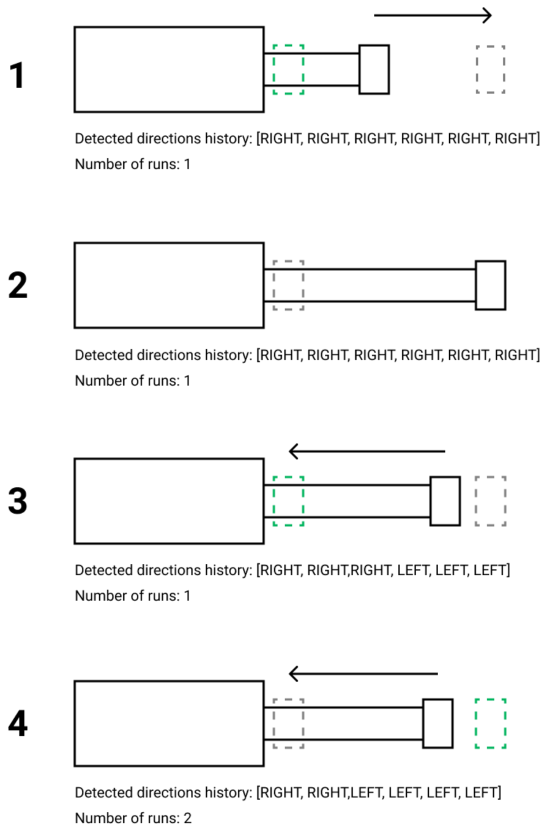

1
Detected directions history: [RIGHT, RIGHT, RIGHT, RIGHT, RIGHT, RIGHT]
Number of runs: 1
2
Detected directions history: [RIGHT, RIGHT, RIGHT, RIGHT, RIGHT, RIGHT]
Number of runs: 1
3
Detected directions history: [RIGHT, RIGHT,RIGHT, LEFT, LEFT, LEFT]
Number of runs: 1
4
Detected directions history: [RIGHT, RIGHT,LEFT, LEFT, LEFT, LEFT]
Number of runs: 2

Figure 18: Process of detecting end positions.

The knowledge of the orientation of a motor, greatly simplifies the detection of an end position, since it allows us to track only directions of the corresponding orientation. Let us look at the example, where motor has vertical orientation, it travels down and it has inconsistent data in the direction history: [RIGHT_DOWN, DOWN, LEFT_DOWN, DOWN, RIGHT_DOWN]. If no orientation were known, we would not know the right direction to detect the corner. However, since we know that the motor is vertically oriented and that last directions can be grouped as DOWN directions; we can state that the corner will be detected if the motor reaches the UP endpoint.

5.4.3.8 Calculating the result trajectory of a motor

After all investigation runs, when some statistics are no longer collected, having sets of all possible end points, we can calculate the trajectory. Our goal is to visualize the trajectory in such manner, that it would be stretched across the whole area of the overall boundaries. However, as the definition of overall boundaries says, it is nothing more than a compilation of all detected rectangles, which means that detected end points will never be situated exactly on borders. This is a consequence of the fact that all end points are mass centers of detected rectangles, from which the overall boundaries are built. Thus, if we were to calculate the trajectory from the weighted value of end points, we would result in something that is shown in Figure 19a), whereas the desired result is displayed in Figure 19b).

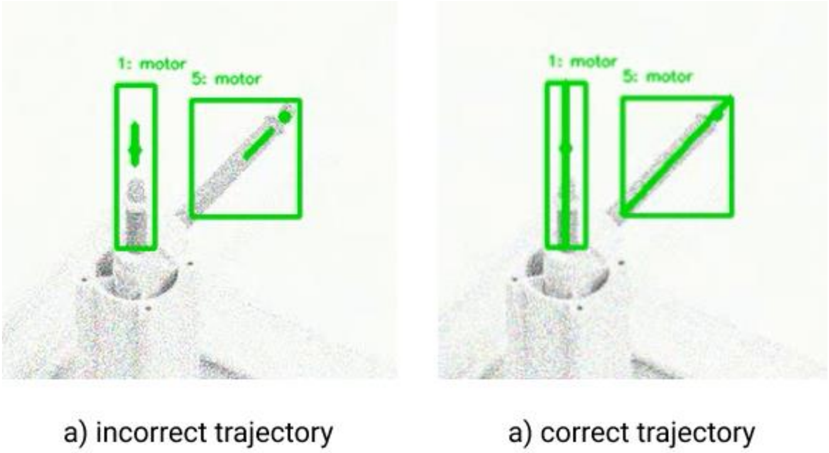


Figure 19: Examples of an incorrectly and correctly calculated trajectory.

Therefore, in order to have the correct trajectory, we need to make additional calculations. Once again, detected orientation will greatly help us to reduce the amount of computations. For example, if we know that the orientation is vertical, we can assume that the trajectory is a line with a slope greater than 45˚. Moreover, its trajectory will be bounded by the upper and lower edges of the overall boundaries. Therefore, we need to have only upper and lower endpoints.

So, having two sets of corresponding to the orientation points we first need to calculate the weighted median value of those points for each set to eliminate the influence of outliers and video noises. Once weighted points for both end point sets are calculated, we need to find the coefficients of the resulting linear equation between those weighted points. This can be done by a simple **numpy** function **polyfit** [24]**.**

Having the required coefficients of slope β and y-intercept α we can find any point of the trajectory by substituting values into the equation $y = \alpha + \beta x$.

For example, if a motor has a vertical orientation, we can calculate its trajectory by putting known value *y1* of the lower edge of the overall boundaries into the equation, resulting in the *x1* coordinate. If the same procedure is done with the upper edge with the *y1* coordinate, we would get a straight line between points *(x1, y1)* and *(x2, y2)*, which is the desired trajectory from one edge to another.

5.4.3.9 Drawing rectangles and a trajectory on a frame

Drawing the extracted information is the easiest part, since it only requires knowing the coordinates of the objects that need to be drawn on a frame. Every motion stores its overall boundaries and trajectory exactly in the form of coordinates corresponding to the size of the frame. So, all we need to do is just take those coordinates and put them into corresponding OpenCV function. We also need to specify the color of the drawn objects and its stroke width. In addition to the overall boundaries and the trajectory of a motion, we also show the current mass center, id of the object and motion type, which are also stored as a parameter inside a motion object. Trajectory can be shown only if case, if it is defined.

5.4.3.10 Extracted statistics

Either when the video is exhausted or when the time of data flushing comes, algorithm starts to generate statistics from the motion properties.

Main goal of the thesis was to develop an algorithm, which would be capable to detect motors and indicator lights; therefore, this is the most general information that is extracted:

- Overall number of detected motions
- Number of detected motors
- Number of detected indicator lights

For each individual motion, we can also extract many useful statistics. Every motion will have information about its ID and its motion type. Additionally, if motion is a motor, it will have following fields:

- Amount of detected runs
- Motion history of directions
- Trajectory
- Orientation

If motion type is indicator light, it will have a single additional field: its position on a frame.

Full example of such statistics can be found in the additional materials to this thesis.

5.5 Recommendations on video quality

To get the best result from the algorithm my recommendation is to have a camera placed on some stable surface, which is not vibrated or at least where vibrations do not cause a distortion more than couple of pixels do. In addition, it is recommended to use the cameras with good video quality to reduce the number of noise distortions. At the same time, video resolution depends on a computational power of the machine, on which the algorithm runs. Recommended video quality for weak machines is 480p. For better machines video resolution can be 1080p and higher. Recommended frame rate is 24 fps, which is optimal enough to catch even quickest events, but which is not to computationally expensive.

Algorithm in its current state was tested on a 30-second video on a machine with following characteristics:

Table 4: Characteristics of machine used for processing

OS	Microsoft Windows 10 Pro
Processor	Intel® Core™ i7-9750H
GPU	NVIDIA GeForce RTX 2060
RAM	16GB
Video resolution	1024x768
Frame rate	24 FPS

Each frame was processed for 0.016973 seconds, which is 2.6 times faster than the frame rate of the original video, thus the algorithm could potentially efficiently run during the video-stream without any delays.

6 Possible algorithm improvements

Additionally to paragraphs 4.2 and 4.6, some general improvements can be made. First, influence of an unstable surface under the camera can be eliminated by comparing each frame of the sequence to the weighted mean of some previous frames along with the current frame. This way the algorithm can be dynamically adjusted to the background.

In addition, dynamical Otsu Thresholding can be used instead of binary thresholding. Such change could potentially eliminate the need to manually set the thresholding value. More information can be found in [25].

Currently, the algorithm can detect a state of an indicator light, but such an information is not stored and is not saved in statistics. This feature can be added with couple of lines of code.

Lastly, lots of code refactoring can be made, which will involve better architecture of the project, since current architecture is too dependent on two main motion classes and it should be split into several different logical blocks and modules. Implementing principles of dependency injection would be a good addition in case, if the project needs to be more modular and different algorithms should be easily interchanged. All default constants should be moved either from the python module to a json configuration file or to an environment file. Probably the best refactoring would be converting the whole project into another programming language like C++, which could drastically improve the performance of the algorithm and which could potentially make it resolution independent. Algorithms used for motion detection and processing can be optimized as well.

7 Conclusion

Proposed in this thesis algorithm satisfies all the requirements, which were made in order to test the hypothesis that an algorithm, utilizing power of computer vision without any need in any other equipment than cameras and a computer, can be made.

It can successfully detect motors and indicator lights on a frame. Moreover, it can extract many useful statistics about the process. Another analytical algorithm to detect possible errors or fails in the manufacturing sequence can further use this data. It can also be used for basic data gathering. This algorithm can tell the observer, what are the boundaries of movement for a motor, what is the trajectory, directions in which a motor moved in any moment of time. It can tell the position of an indicator light and it can potentially recognize the state of the light.

Even though the algorithm is written in an interpreted language Python, its speed is high enough to process the video without any delays. Thus, it leaves a space for additional processing steps, which can be applied to each frame, to extract even more information.

As a further step of improvement such features as creating step graphs of motors movements or matching indicator lights to motors for better control over the process can be potentially added.

References

[1] Tameson, "Pneumatic Cylinder - How They Work," [Online]. Available: https://tameson.com/pneumatic-cylinders.html. [Accessed 29 July 2020].

[2] Tameson, "Pneumatic Cylinder Sensors - How They Work," [Online]. Available: https://tameson.com/pneumatic-cylinder-sensors.html. [Accessed 29 July 2020].

[3] R. M.-S. F. M.-C. M. M.-J. S. Garrido-Jurado, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition,* vol. 47, pp. 2280-2292, 2014.

[4] opencv dev team, "Template Matching," 31 12 2019. [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html. [Accessed 29 July 2020].

[5] И. Голиков, "Свертончная нейронная сеть, часть 1: структура, топология, функции активации и обучающее множество," 31 January 2018. [Online]. Available: https://habr.com/ru/post/348000/. [Accessed 29 July 2020].

[6] Stanford, "CS231n Convolutional Neural Networks for Visual Recognition," [Online]. Available: https://cs231n.github.io/convolutional-networks/. [Accessed 29 July 2020].

[7] H. H. A. K. a. Y. M. Mustafah, "Colour-based Object Detection and Tracking for Autonomous Quadrotor UAV," 2013.

[8] B. Dickson, "This MIT AI Predicts Breast Cancer Risk Up to 5 Years in Advance," 23 May 2019. [Online]. Available: https://www.pcmag.com/news/this-mit-ai-predicts-breast-cancer-risk-up-to-5-years-in-advance. [Accessed 29 July 2020].

[9] OpenCV, "Image Thresholding," OpenCV, [Online]. Available: https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html. [Accessed 29 July 2020].

[10] A. K, "Morphological Transformations," 25 January 2014. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html. [Accessed 29 July 2020].

[11] opencv dev team, "Eroding and Dilating," 31 December 2019. [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html. [Accessed 29 July 2020].

[12] V. Powell, "Image Kernels Explained Visually," 29 June 2018. [Online]. Available: https://setosa.io/ev/image-kernels/. [Accessed 29 July 2020].

[13] OpenCV, "Contours : Getting Started," 18 July 2020. [Online]. Available: https://docs.opencv.org/4.4.0/d4/d73/tutorial_py_contours_begin.html. [Accessed 29 July 2020].

[14] S. S. a. others, "Topological structural analysis of digitized binary images by border following," in *Computer Vision, Graphics, and Image Processing*, 1985, p. 32–46.

[15] Oracle, "What Is a Socket?," 19 July 2018. [Online]. Available: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html. [Accessed 29 July 2020].

[16] Python Software Foundation, "unittest — Unit testing framework," 2 August 2020. [Online]. Available: https://docs.python.org/3/library/unittest.html. [Accessed 2 August 2020].

[17] OpenCV, "About OpenCV," [Online]. Available: https://opencv.org/about/. [Accessed 29 July 2020].

[18] NumPy, "About Us," [Online]. Available: https://numpy.org/about/. [Accessed 29 July 2020].

[19] S. Nakov, "Object-Oriented Programming Principles (OOP)," in *Fundamentals of Computer Programming with C#*, Faber, Veliko Tarnovo, Bulgaria, 2013.

[20] Python Software Foundation, "enum — Support for enumerations," 2 August 2020. [Online]. Available: https://docs.python.org/3/library/enum.html. [Accessed 2 August 2020].

[21] Python Software Foundation, "Private Variables," 29 June 2020. [Online]. Available: https://docs.python.org/3.7/tutorial/classes.html#private-variables. [Accessed 29 July 2020].

[22] OpenCV, "Operations on Arrays," 31 December 2019. [Online]. Available: https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html#absdiff. [Accessed 29 July 2020].

[23] OpenCV, "Structural Analysis and Shape Descriptors," 2 August 2020. [Online]. Available: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a. [Accessed 2 August 2020].

[24] NumPy, "numpy.polyfit," 29 June 2020. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html. [Accessed 29 July 2020].

[25] D. A. Greensted, "Otsu Thresholding," 17 June 2010. [Online]. Available: http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html. [Accessed 29 July 2020].

Table of Figures