FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Dubský

Optical analysis of pellet car damages

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	RNDr. Elena Šikudová, Ph.D.
Study programme:	Computer Science
Study branch:	IPSS

Prague 2019

First of all, I’d like to thank my thesis supervisor, Ms. RNDr. Elena Šikudová, Ph.D., who had willingly helped me to write this thesis.
I’d also like to thank PaulWurth, which assigned this work and allowed me to present it as my bachelor thesis.

Title: Optical analysis of pellet car damages

Author: Jan Dubský

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Elena Šikudová, Ph.D., Department of Software and Computer Science Education

Abstract: During iron processing, pelletizing is one of the necessary steps. Iron ore pellets are burned on pellet burning line, which consists of individual pellet cars. Due to constant temperature changes, pellet cars get damaged.

This thesis focuses on an optical analysis of pellet car damages. The algorithm presented can find individual ribs of pellet car and perform analysis of their shape, position and look. Such analysis can provide basic information for damage evaluation, same as the material for further research of pellet car damage causes.

Keywords: **Image processing**, **Damage**, **Steelmaking**, **Pellet car**

Contents

Chapter 1

Introduction

1.1 Iron processing and steelmaking

A modern world without steel and iron is almost unimaginable. Both steel and iron are widely used in many industry branches including engineering, building industry, automotive etc. Nowadays about 98% of world mined iron ore is used to produce steel[1].

Naturally, iron exists in the form of iron ores – for example magnetite, hematite, goethite, limonite, and siderite[1]. First of all, we have to extract raw iron from mined ores. Those processes are generally called beneficiation and include techniques like grinding, gravity separating or froth floating[1]. At the end of beneficiation, we get so-called fines, which is fine iron dust, containing significantly fewer impurities.

Now, we would like to put our fines to blast furnace, but we have to deal with one more problem. Namely, we need to ensure enough oxygen all over the blast furnace, to achieve the desired chemical reaction. In this part, there are two major solutions. Firstly, iron fines are mixed with various additional materials and water and formed into pellets[1][2]. When formed, pellets are burned on a pellet burning line, to create hard spheres. Putting pellets into a blast furnace, they touch each other only by very small part of its surface, producing enough space for air. Secondly, iron fines are mixed with coke and limestone and burned on a sinter burning line[3]. Due to burning, blend solidify into a porous material, which is then crushed into irregularly shaped ”stones”. Irregularity in stone shapes again produces enough space for air. Both pellets and sinter are then cooled and transported to blast furnace.

We have pellets or sinter, so we can put them to the top blast furnace[4]. As material slowly falls down the blast furnace, the desired chemical reaction takes place. To supply oxygen for chemical reaction, air (sometimes oxygen enriched) is blown to bottom of the blast furnace. Products of our reaction are pig iron (crude iron) and so-called slag, which are taken from the bottom of the blast furnace[5]. Slag is a waste product of iron processing, which is further used as cement ingredient, to improve its durability[6]. Our desired product, pig iron is either formed into so-called pig ingots and transported, or directly poured into steelmaking ladle or furnace and processed into steel.

Here, we finally get to steelmaking description. The main difference between

[1]Iron pellets are small balls, typically with diameter from 6–16 mm.

pig iron and steel is the amount of carbon in an alloy. Pig iron is quite carbon-rich – about 4%[5]. In the case of steel, we need to lower carbon content below 1.5%[7]. The most widely used method is called Basic oxygen steelmaking. About 70% of world steel production is produced using this method[7]. In basic oxygen steelmaking, molten pig iron is poured to so-called ladle. Oxygen is blown through molten pig iron, lowering carbon content of alloy and producing low-carbon steel[8]. Steel can be further enhanced by techniques as hardening and tempering or by alloying with different metals, to reach required mechanical properties[7]. There is also a different method of steelmaking, called Electric arc steelmaking, but this method is mostly used to produce steel from iron scrap, so I decided not to describe it in this thesis.

1.2 Problem description

The topic of this thesis will be linked with iron pelletizing and sintering. In the previous section, I spoke about forming and processing of pellets or sinter. But what I haven't described broadly is the construction of pellet/sinter burning line.

Both pellets and sinter are burned on pellet burning line and sinter burning line respectively. Those lines look very similar, so I will further refer only pellet burning line and pellet car. Everything I will describe later also applies to sinter cars.

Pellet burning line is, in fact, a traveling grate, passing through a kiln. This traveling grate consists of multiple pellet cars, slowly moving along a railway as a conveyor belt. Typically, each pellet car has from 2 to 4 transverse lines or longitudinal cast ribs (see Figure 1.1). On those ribs, pellets or sinter lie, while passing through a kiln.

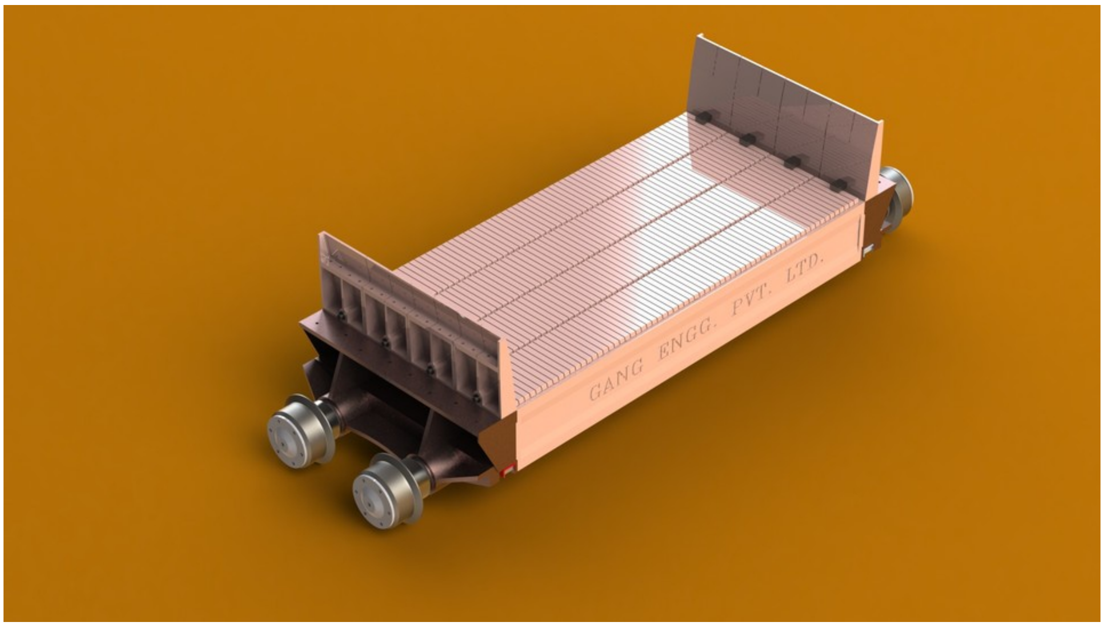


Figure 1.1: Pellet car or Sinter car respectively[9].

Although temperatures inside kiln aren't as high as in blast furnace, they still reach hundreds of degrees Celsius. Such temperatures, together with continual warming and cooling of pellet cars cause various damage of cast iron ribs. Common damage are including rib cracks, misplaced ribs, holes in rib and missing ribs, which broke and fell off pellet car. Those pellet car damage leads to iron material losses, reducing the overall productivity of pellet burning line. So pellet cars have to be regularly controlled and replaced in case, they are seriously damaged.

In the current setup, all of that damage is observed and evaluated by human. If a supervisor considers overall damage of traveling grate serious enough, the whole iron production line is stopped and damaged pellet cars are replaced by new ones. During this process, all pellet cars have to be visually controlled by a repairman, which takes time and does not provide objective damage evaluation. Moreover, it is not possible to track damage of pellet car over time, producing data for research on pellet car damage causes. Overall, the current way of pellet burning line monitoring is not satisfactory at all.

1.3 Proposed solution

As current setup is far not ideal, automatic optical monitoring of pellet car damage can be used as a better alternative. A monitoring system should be able to trace the state of pellet car over time, supplying data to evaluate current burning line damage. Further, it should produce data, which can be stored and later used for a research on pellet car damage causes.

An automatic monitoring system will, of course, require a few modifications of a pellet burning line. First of all, optical cameras will be installed above the pellet burning line in place, before pellets are poured to pellet cars[2]. Maybe you noticed I wrote cameras. That is, because pellet car is as wide, that one camera would not be able to capture it whole and stay reasonably close to pellet burning line at once. Secondly, to provide sufficient light level to optical cameras, led flash belt will be installed above the burning line. Last but not least, cameras have to be able to capture each pellet burning line, so they need to know, when is pellet car below them. Customer stated, that his pellet burning line control software knows those data and can trigger cameras to capture each pellet car.

All those modifications are expected to be implemented by the customer. Each pellet car photo will be passed to the detection algorithm, which will find individual ribs in an image. More precisely, because pellet car is quite long, not each pellet car, but each line of ribs in a pellet car will be captured and passed to an algorithm separately.

1.4 Aim of the thesis

Expecting setup as described in the previous subsection, my task was to invent an algorithm, which will analyze images of pellet burning line. In each rib line, all ribs present should be found, providing basic information, like rib shape, count and distances between them. Moreover, the algorithm should be able to identify basic rib damages listed in Subsection 1.2. The whole analysis should execute in

[2]This is the only place, where pellet car ribs are visible because pellet burning line construction does not allow access bottom part of the conveyor belt.

real time, with a reasonable amount of resources - we have preliminarily spoken about 4 CPU cores dedicated to pellet car analysis. Given burning line speed, an image of one rib line will be produced each 4 seconds approximately. Having 4 CPU cores and 4 cameras, whole algorithm execution should fit 4 seconds on a FullHD image (1920x1080px).

1.5 Test data

At the time of writing this thesis, pellet burning line modifications are not complete yet. So for testing purposes, I received two short (about 1.5 min) videos of pellet burning line. From those videos, I extracted 27 test images, which I used for algorithm testing. But because those two videos were taken on unmodified pellet burning line, extracted test images have much lower quality, than final setup should produce. For example, a pellet car is not uniformly illuminated (see Figure 1.2). On the other hand, having imperfect test images will force me to develop a robust algorithm, which will be able to work correctly even with such input.

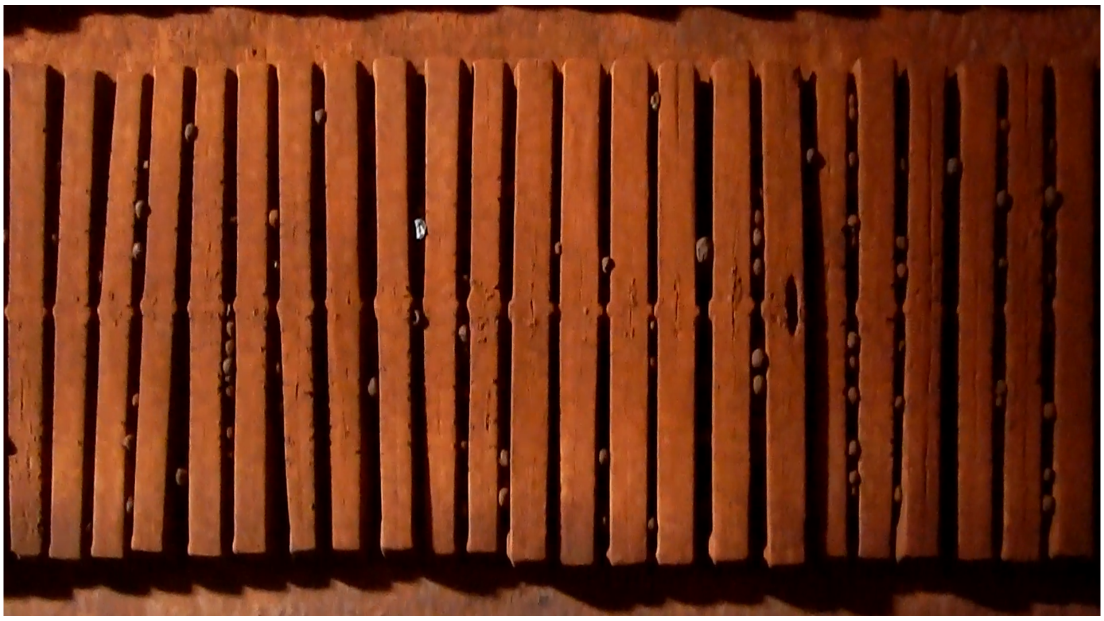


Figure 1.2: Example of a test image.

Chapter 2

Image processing algorithms

First of all, let's introduce some image processing basics - algorithm, which would be presented in an introductory course of image processing. All of those algorithms are widely implemented in many image processing libraries.

2.1 Thresholding

Thresholding[10, 11] is one of the most common segmentation algorithms used in image processing. The basic idea of thresholding is applying inequality comparison of a threshold value (given by a user) and value of each pixel followed by changing of pixel value depending on the comparison result.

Thresholding is very often applied only on greyscale image. There are variants of color image thresholding, but they are still based on thresholding of each color separately (i.e. greyscale thresholding). Common usage of thresholding is an identification of foreground and background areas of an image based on different illumination of objects in foreground and background.

In this thesis, only binary, grayscale thresholding will be used. A product of binary thresholding, as its name would suggest, is a binary image, with nonzero pixels where the value was higher than threshold value and zeros where it was less or equal than threshold value.

2.1.1 Otsu thresholding

The biggest problem of thresholding is the requirement to set a threshold value. Very often, we need to analyze scenes, the threshold value of which depends on light conditions, an image was taken under. For example, we have to process outdoor photos of the object taken in different daytimes. Taking such photo at noon, there will be much more light, than taking the same photo at nightfall. For such set of photos, we can't say universal threshold value, because it simply doesn't exist. Here comes Otsu thresholding[10, 12, 13].

Otsu thresholding expects, there is foreground (lighter part) and background (darker part) in the image. This type of image is in literature referred as bimodal image and its histogram will have two significant peaks (one for foreground and second for background color). Otsu threshold calculates a histogram of the image and finds such threshold value, that sum of weighted[1]variance of foreground and background class is minimal. After that, thresholding is applied to input image with the found threshold value.

Compared to thresholding, Otsu thresholding executes a bit slower due to histogram calculation and threshold value finding. On the other hand, there is no need to set threshold value and foreground objects in an image are found automatically and more or less independently on light conditions in the image[2].

2.2 Filtering algorithms

Whenever we take a photo, there inevitably appears inaccuracies in it, altogether called image noise. Image noise is a general term for the presence of pixels with completely random values. Noise has a wide range of causes from digital chip inaccuracies or damage to discrete (quantum) nature of light and electric charge. Though noise can be reduced by upgrading of image capture technology, it can never be removed at all. Meaning we have to expect noise in the image and deal with it because many image processing algorithms are very sensitive to noise. For this purpose, filtering algorithms[10, 14, 15, 16] have been invented.

2.2.1 Gaussian blur

Gaussian blur[14] is an arbitrary sized convolutional kernel, which calculates pixel value as the average of its neighborhood weighted by Gaussian function of distance. Because noise is made of outlier values of individual pixels, they have only negligible effect on the average result. The resulting image is noticeably smoother, than the original one (see Figure 2.1).

Unfortunately, as Gaussian blur removes noise, it removes edges in the image too. Edge is an area with bright pixels on one side and dark pixels on another one. Simple averaging cause them to blend together, resulting in a smooth color transition instead of sudden color jump (see Figure 2.1).

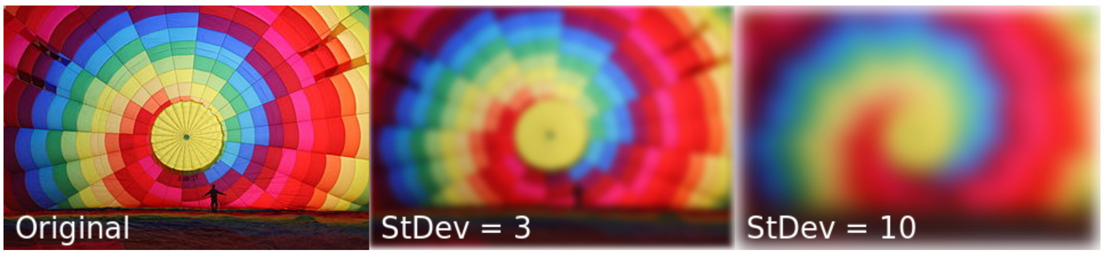


Figure 2.1: Example of Gaussian blurred image of stained glass. [17]

2.2.2 Bilateral filtering

Bilateral filter[14] algorithm is a very effective modification of Gaussian filter for noise removal while keeping edges sharp. Gaussian filter calculates the average of

[1]The variance of each class is weighted by the number of pixels it contains.

[2] More or less, because there are many factors affecting the Otsu threshold result. For example, having a glossy object, there exists the limit light level, when Otsu starts identifying light reflection as foreground and rest of object as background.

nearby pixels weighted by Gaussian function of distance. Bilateral filter weights pixels in the same way as Gaussian filter, but adds additional weight criterion, namely similarity to the center pixel (pixel, which value is calculated).

So in Bilateral filter, only nearby pixels with similar values are averaged resulting in removing noise and preserving sharp edges. The disadvantage of this algorithm is, it's higher computational difficulty and therefore a bit slower execution compared to other filtering algorithms.

Take a look at Figure 2.2 and compare it with Gaussian blur in Figure 2.1. As you can see, Bilateral filter is much better in edge preserving.

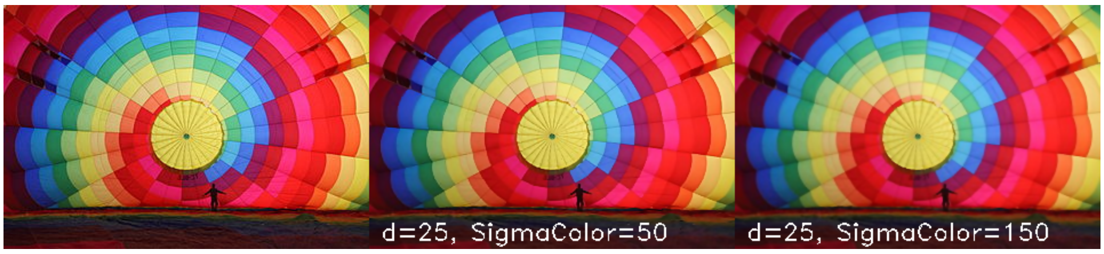


Figure 2.2: Example of Bilateral filtered image of stained glass [17]

2.3 Contrast enhancement algorithms

Very often, when taking an image, there are colors (understand shades of gray), which are in an image a lot and then colors, which almost aren't present there. In histogram point of view: histogram has significant peaks.

In normal photography, we want this behavior, because we take photography to capture reality as precisely as possible. But having those histogram peaks in image processing makes us problems for many reasons.

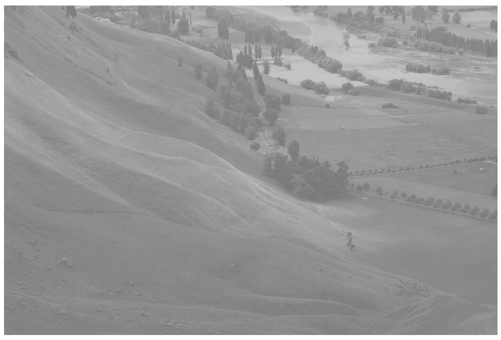
First of all, we naturally expect that having pixel value range [0-255] means, that approximately 25% of pixels are below 64, half of them below 128, etc., so those peaks confuse our intuition. But even if we deal with our intuition, another image processing algorithms does not expect such color distribution and would not work properly with such image. For example, we have already seen Otsu thresholding, which looks for two significant histogram peaks and those peaks are not expected to be next to each other. In such a case, their in-class variance would be low and Otsu would not try to split those two peaks by threshold value.

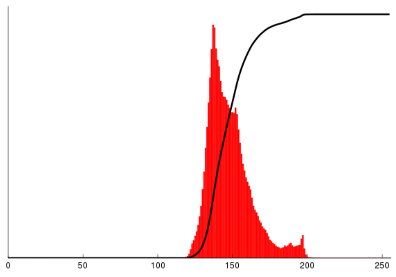
To deal with ununiform value distribution, we can use histogram equalization algorithms.

2.3.1 Histogram equalization

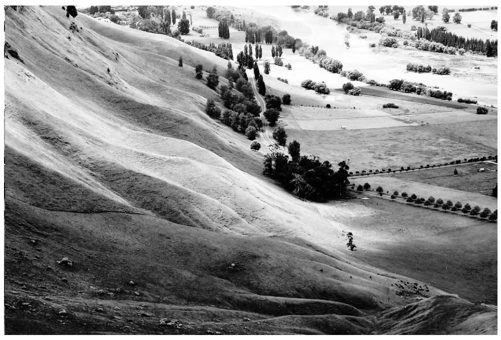
Histogram equalization[18], as its name suggests equalizes image histogram. First of all, the histogram of an input image is calculated. Then colors are remapped, producing an approximately rectangular histogram. To reach such effect, several values with only a few pixels are mapped to one color and on the other hand, neighbor values with many pixels are spread to an interval of values.

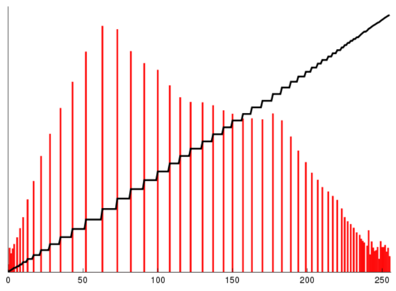
After histogram equalization, the cumulative function of histogram should be much closer to the linear function, than before and contrast of image should be much better (see histogram (2.3d) and compare it to the original histogram in (2.3b)). So it should be (approximately) true, that the half of pixels have their value below half of the available pixel value range. Overall, the output image is much more suitable for future image processing (see Figure 2.3).


(a) Original image[19]



(b) Original image histogram[20]


(c) Equalized image[21]



(d) Equalized image histogram[22]

Figure 2.3: Exhibition of histogram equalization

2.3.2 Adaptive histogram equalization

As we have seen before, histogram equalization helps us to spread image values over a whole possible value range. But histogram equalization will not work well for all kinds of images. For example, having an indoor photo of a person in front of a window. In such case, the majority of pixels will be extremely light, so histogram equalization will try to spread their values. After histogram equalization, our person (darker pixels) will be even worse to recognize then he or she was before equalization, as his or her pixels will be remapped to a smaller range of values.

Adaptive histogram equalization[23] is a modification of histogram equalization suitable for images with different light levels in different parts of the image. Instead of equalizing the whole image at once, the input image is divided into small blocks (8x8 pixels for example) and each block is equalized independently. In other words, local area histogram is calculated for each block and used to equalize that block instead of equalizing the whole image at once. Consequently, only a small area of the image is equalized at once and is not affected by the presence of big light or dark area on the other side of the image as it would be in case of histogram equalization. This way, contrast of the image can be enhanced all over the image and not only in a dominant part of it.

The main disadvantage of this algorithm is its tendency to amplify image noise, especially in flat (monochromatic) areas of an image. Noise, an outlier value, will not have sufficient weight to influence global histogram but will have significant weight in a local histogram. To reduce this effect, an image filtering algorithm should be applied before adaptive histogram equalization.

2.3.3 Clip limited adaptive histogram equalization

Clip limited adaptive histogram equalization[23] (CLAHE) is based on adaptive histogram equalization and additionally deals with a problem of amplifying image noise. Compared to adaptive histogram equalization, CLAHE takes one more parameter called contrast limit. This method also splits image to small blocks but is there is a pixel in a block, which exceeds contrast limit, it's clipped and uniformly distributed over the whole histogram. Contrast limit ensures that noise in a block will not affect a local histogram. This way, CLAHE avoids noise amplification and keeps advantages of adaptive histogram equalization when the contrast limit is set correctly.

2.4 Distance transform

The distance transform[24, 25] is a binary image processing algorithm, setting value of each nonzero pixel in an original image to its distance from the closest zero pixel. Typical usage of distance transform is finding of thresholded object center, but it's as well used as a stage of more complex algorithms. For example, Stroke Width Transform algorithm for text detection in image uses the distance transform.

2.5 Connected components

Very often, we have a binary image, obtained for example by thresholding, and we need to find all objects in it. To do so, we can use the connected components algorithm[10, 26]. Connected components is a clusterization algorithm for binary images.

White (=nonzero) areas of an input image are considered as objects and black (=zero) pixels as background. Using 4-fold of 8-fold connectivity, this algorithm finds all independent objects and marks them with unique numbers (usually in range $\{1..n\}$). Resulting clusters can be easily separated by applying the equality operator for each ID[3].

2.6 Edge detection algorithm

Edge detection[10, 27] is one of fundamental feature analysis used in image processing. In general, an edge is considered to be each place in an image, where

[3] Very often there is an extension of connected components algorithm returning a bounding rectangle. This does not seem to be important, but can be used for speeding up your algorithm, because you can limit your view just to this bounding rectangle and work with much smaller image consequently.

the color changes significantly. To find such places, we can use several attitudes, which are described below.

2.6.1 Sobel operator

Sobel operator[28] is convolutional kernel used for calculating of image first derivative in X or Y direction. As we all know, the derivative has high absolute value in areas, where function value changes significantly. Those are edges we are looking for.

The basic idea of Sobel is comparing of a small neighborhood around each pixel and using it to approximate derivative by the difference. Specifically, Sobel uses 3x3 neighborhood centered on the pixel, the value of which is calculated and calculates the difference of pixels on its opposite sides.

Approximating of the derivative by the difference in local neighborhood can produce noticeable inaccuracies, especially in noisy images. Further processing of Sobel image is usually necessary, to eliminate those false edges.

2.6.2 Laplace operator

Laplace operator[29, 30] is another 3x3 convolutional kernel, used for local approximation of the second derivative. Laplace operator is defined as $Lap(f) = \frac{d^2f}{dx^2} + \frac{d^2f}{dy^2}$. As you can see from its definition, unlike Sobel, which have individual operators for X and Y direction, there is only one version Laplace operator approximating X and Y direction at once.

Now, we should take a look at the behavior of the second derivative at edges. An edge is a place, where the first derivative has its local maximum, implying zero value second derivative in that place. So edges in an image can be found by comparing a Laplacian image to zero.

Unfortunately, it's true, that edge pixels will have zero value in Laplacian image, but it isn't true, that all zero values in Laplacian image are edges. For example, having a smooth surface, its Laplacian value will be zero too. Consequently, the Laplacian image used for edge detection has to be filtered in purpose of removing false positives.

2.6.3 Canny edge detection

Canny edge detection[10, 31] is a more sophisticated algorithm for detecting edges in an image.

In the first step of the algorithm, image noise is removed using Gaussian blur. When most of the noise is removed, Sobel operators in X and Y direction are used and their results are used to calculate gradient direction and magnitude. In the image, we have only horizontal, vertical and two diagonal directions, so the algorithm can round gradient direction to 45° because higher precision of gradient direction couldn't be used anyway. Please note, that gradient is the direction of function value growth, implying edge should be perpendicular to gradient direction.

In a real image, the edge can be more than one pixel thin. If we want to find thin edges, we need to filter out pixels with lower gradient magnitude, that their neighbors. To do so, the algorithm looks in the direction of gradient and if the current pixel gradient magnitude is not local maximum, it is suppressed to zero. This removal technique ensures, we always take only edge pixel with the highest magnitude, producing thin edges.

To decide, which potential edge is really edge and which is just noise, the algorithm takes two more values as its input - minVal and maxVal respectively. Pixels with gradient magnitude higher than maxVal are sure to be edges. Analogously, algorithm discards all potential edge pixels having magnitude below minVal. Pixels in the range $[minVal, maxVal]$ will be accepted as an edge if they are connected by an edge to a pixel with magnitude above maxVal (remember connected components in 2.5). Consequently, only significant edges in the image are found.

Canny edge detection is a common algorithm used for finding of real edges (see Figure 2.4), unlike Sobel and Laplace operator, which are more often used as a stage of more complex algorithm. The only disadvantage of Canny detection is, edges found are not continuous, so Canny edge detector can't be used for image segmentation. However, there is no basic edge detection algorithm producing closed edges.

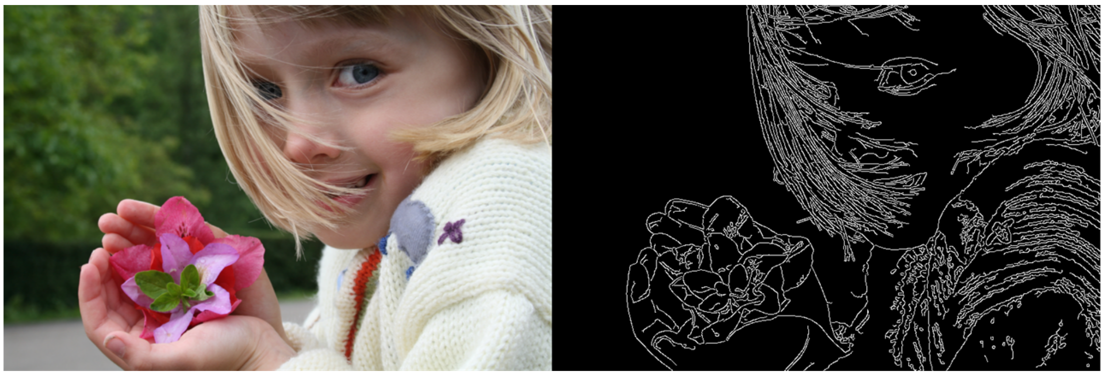


Figure 2.4: Example of Canny edge detection [32]

2.7 Hough line transform

Hough lines transform[10, 33, 34] as its name advice is an algorithm for finding lines in a binary image.

First of all, we need to think about a line description. A line can be mathematically described in many forms, including parametric equation $y = k * x + q$. The small disadvantage of the parametric equation is, it can't describe the vertical line, because such line has all values of y for one value of x and, thus such line can't be described as a function from X to Y. To be able to describe all lines, including vertical lines, we can choose a different form of description. Such form is called line polar coordinates and describes line by equation $\rho = x * cos(\theta) + y * sin(\theta)$, where ρ is the distance of the line from origin and θ is the angle of X-axis and distance perpendicular line (line perpendicular to line described and containing point $[0, 0]$). This form is used to describe the line in the Hough line transform algorithm.

At the beginning of the algorithm, the 2D accumulator of ρ and θ values is allocated. Precisions of the accumulator in both dimensions are input parameters of the Hough line transform given by a user. The algorithm is based on point voting for all lines it can lie on. Every white (=nonzero) point of an image is taken, all possible lines this point can lie on are found and belonging accumulator field (ρ, θ) is incremented. When all points vote for their lines, all pairs of ρ and θ with more votes than threshold value are returned as lines found. See the result of Hough line transform result in Figure 2.5.

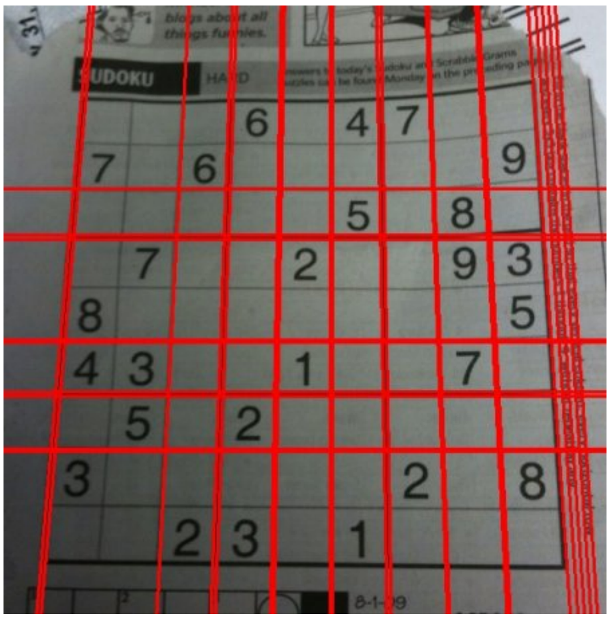


Figure 2.5: Example of Hough lines result[34]

After a short analysis of the algorithm, we can conclude, that accumulator size affects algorithm duration the same as quality of the result in a significant way. When a user sets a low resolution of the accumulator, algorithms execution will be fast, but with inaccurate result. On the other hand, too high resolution means higher memory requirements same as much more possible lines, found for each point, resulting in significantly longer execution time.

Generally, the Hough line transform is a robust algorithm for line finding with very good results. A small disadvantage is, that it's quite time and memory consuming if higher result precision is required.

Chapter 3

Description of an algorithm

Now, when all basic image processing algorithms were introduced, we can describe the rib finding algorithm itself.

3.1 Expected input image parameters

First of all, I will list properties of an input image, algorithm expects and counts on. All of those expectations are based on the dealings with the customer and are very easy to ensure by camera installation above pellet car belt as described in 1.3.

Those parameters are:

1. The image is taken from upwards of pellet car.
2. Exactly one rib grid must be present in the image.
3. Ribs must be in upside down direction.
4. Ribs can't be cropped in a vertical direction.
5. Ribs are expected all over the width of the image.
6. The image can contain pellet car (i.e. not ribs) at top or bottom.
7. The image must be uniformly illuminated.
8. There are expected no shadows in the image, except those cast by ribs.

For a better idea of a typical input image, take a look at Figure 3.1.

3.2 Subroutines

The algorithm itself frequently uses several subroutines. Because it would be quite complicated and confusing, to explain those subroutines during algorithm description itself, I will rather explain them now and just reference them in the description section. Maybe their purpose will not be clear now, but I believe ideas behind them are going to clarify later in the course of algorithm description.

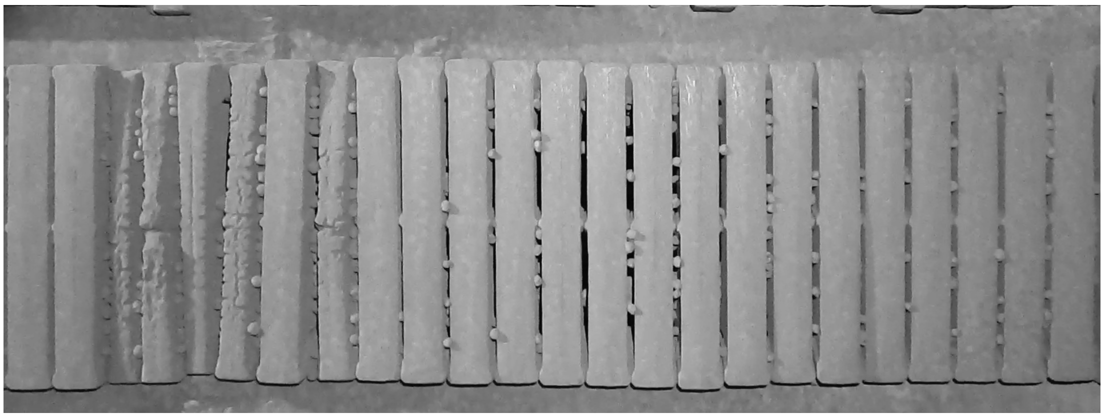


Figure 3.1: Typical input image of the algorithm.

3.2.1 Extract objects

This function takes a binary image and minimal size of the object in pixels as its parameters. Using connected components function, all components are extracted from the input image, but only those bigger than minimal object size are taken and returned as a list of binary masks. Smaller components are discarded. Purpose of this function is clearing a binary image from small objects. In an algorithm, we will often use this function to remove pellet thresholds from the binary image.

3.2.2 Remove holes in mask

Another function based on connected components but using them in another way. Parameters of this function are input binary image and a number called maximal hole size. The function finds all holes in a binary image (using connected components on inverted image) and removes those, which are smaller than maximal hole size. The return value of this function is the binary image without holes. We will use this function to make rib thresholds continuous.

3.2.3 Rib angle

Detecting of rib angle is an important part of this algorithm. The function accepts a binary mask of one rib and using Hough lines calculates its angle.

First of all, Laplace filter is applied to rib mask to get just borders of rib mask. Without this step, Hough lines would see lines everywhere, because every point of rib mask would be considered as a point of a line. This step ensures only rib edge pixels will be used for rib angle detection. Laplacian could be substituted by distance transform and condition ≥ 1.5 for each pixel, but this would cost 3x3 kernel convolution and extra condition for each pixel. Laplace operator costs just 3x3 kernel convolution and gives the same result.

When having just edge pixels of rib, Hough lines can be used. We get plenty of lines found, but each line was voted by different number of edge points. Real rib angle should be correctly calculated as the average of lines angles weighted by the number of votes for a line. Unfortunately, here comes a small problem with the implementation of Hough lines I use as part of OpenCV library - it discards all votes when Hough lines function returns. To simulate weighted average, this function takes most voted 10 lines (returned lines are ordered by the number of votes) and calculate the arithmetic average[1]. The average angle of those 10 lines is then returned and considered as rib angle.

3.2.4 Line matched filter[2]

Having a threshold image of ribs, there are very often inaccuracies. For example darker parts of rib, which have been removed by thresholding, or rib cracks which were detected as background. We need to remove such inaccuracies of a threshold to determine ribs and spaces in an image. The typical solution to this problem would be using morphological closing. But it is not possible here, because this operation would connect all ribs into one and discard spaces between them. Instead of closing, we will use scan line in direction of rib, which will go along the whole mask, to reconstruct rib, but keep ribs separate.

This function takes three parameters: binary image, a binary mask of rib fragment and coefficient from (0,1) interval. Moreover, rib fragment is expected to be present in the binary image too. So typically, 1st argument is a threshold image of pellet car and 2nd argument is one connected component, extracted from that threshold image (i.e. a fragment of a rib).

First of all, the rib fragment angle is calculated using the Rib angle function described above in section 3.2.3. Line with the same angle as detected fragment direction in drawn and shifted through the whole width of rib fragment pixel by pixel. In each position, all binary image (1st parameter) pixels covered by line are taken and compared to the total number of line pixels. If this ratio is sufficient (bigger than given coefficient parameter), the whole line is added to result.

For a better understanding of this algorithm, take a look at Figure 3.2a, where you can see Line matched filter applied on one rib fragment. Red object is the binary mask of rib fragment (2nd argument of this function), black and white is the original image (1st argument of function), which includes rib fragment (red area) too. Green area is added by the matched filter - these are lines, which covered a sufficient amount of binary image pixels. The result of the Line matched filter for this rib is then shown in Figure 3.2b. This way, discontinuous rib masks are line by line connected, without blending together with the neighboring rib.

3.3 Algorithm description

Now, when we have explained all algorithms and subroutines used, we can finally start the description of the algorithm itself. Our algorithm will take two arguments - a greyscale input image and set of its parameters (about 20 values). Those are values like expected count of ribs in the image, a number of pixels per millimeter (scale of the image), minimal size of rib and many other.

[1]Line vertical deviation threshold (10°) is used to ensure only vertical lines affect angle average.

[2] Name matched filter is often used in the signal analysis. In image processing, this is not casual algorithm.

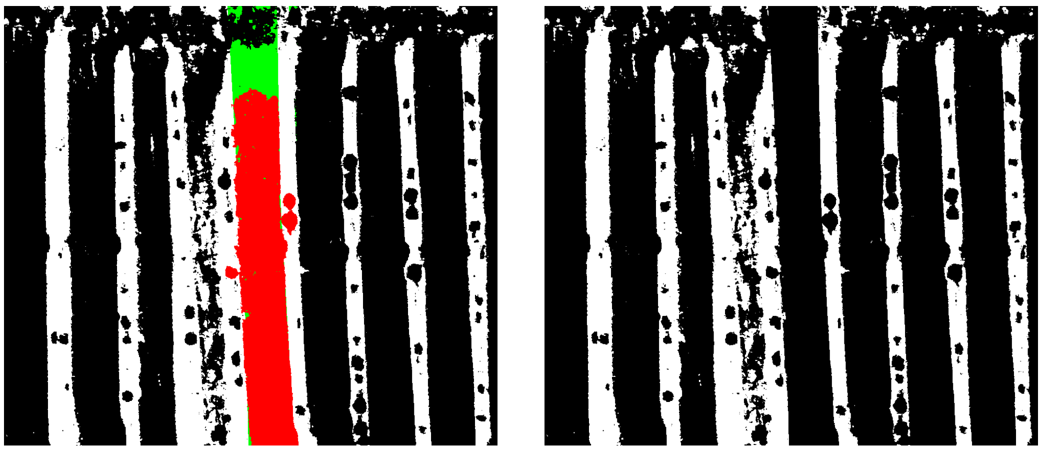

(a) Color visualization (b) Output

Figure 3.2: Illustration of Line matched filter

3.3.1 Input preprocessing

First of all, an input image is bilateral filtered to remove noise, but keep edges sharp. Keeping edges sharp in important for postprocessing part of the algorithm, where edge detection will be used. To use whole grayscale color range and improve light conditions among a whole image, the bilateral filter is followed by clip limited histogram equalization (CLAHE).

3.3.2 Thresholding

Now, we can use the Otsu threshold to extract ribs and remove the majority of background (see Figure 3.3). Of course, some parts of ribs will be thresholded too, causing ribs to split to several fragments. On the other hand, many pellets and background areas will be visible in the threshold image. We will deal with all of those inaccuracies of threshold image in a few following steps.

Figure 3.3: Result of thresholding of output image.

3.3.3 Finding of the area covered by ribs

An original input image can contain pellet car (area without ribs) in upper and bottom part[3]. Those parts are not interesting for us. They are present only because the camera installed above the pellet car belt captures a higher image, than ribs are. So in this part, we will cut off those areas, to achieve the image, where only ribs are present.

When we take a look at the thresholded image (see Figure 3.3), ribs are very well visible for human, because they are split by vertical spaces. Even in a situation, where we have a lot of noise and pellets in threshold image, those spaces are still very clear to see for us. So our algorithm could look for image rows, where spaces are well visible. Equally, it could look for rows, where ribs are well visible. Ribs are exactly as clear as spaces, but they have two advantages over spaces – they are wider and we know, how wide should they approximately be. So, the algorithm will take a look at each image row and look for abscissa, which is not thresholded and is sufficiently long (let's say 50% of undamaged rib width). This way, the algorithm can go line by line and look for a sufficient amount of ribs (let's say 65% of expected rib count in the image).

Sure there will be rows without ribs, where our condition will be satisfied too, but there are not too much of such rows and their occurrences are random. Typically, there will be an unthresholded part of pellet car with small thresholded areas inside (take a look at top of Figure 3.3). To prevent such row to be understood as rib area, we can require fulfilling rib count condition in several consecutive rows (let's say 10 rows), before we say, that there are ribs in a row. This condition will be already sufficient for correct finding of rows containing ribs.

So the final algorithm stage will scan line by line from upside down until it finds rib area begin and then do the same algorithm from the bottom line of an image. This way, we will cut out top and bottom of an image (see Figure 3.4), where no ribs are and rest of the analysis will be performed on the image, which contains only ribs.

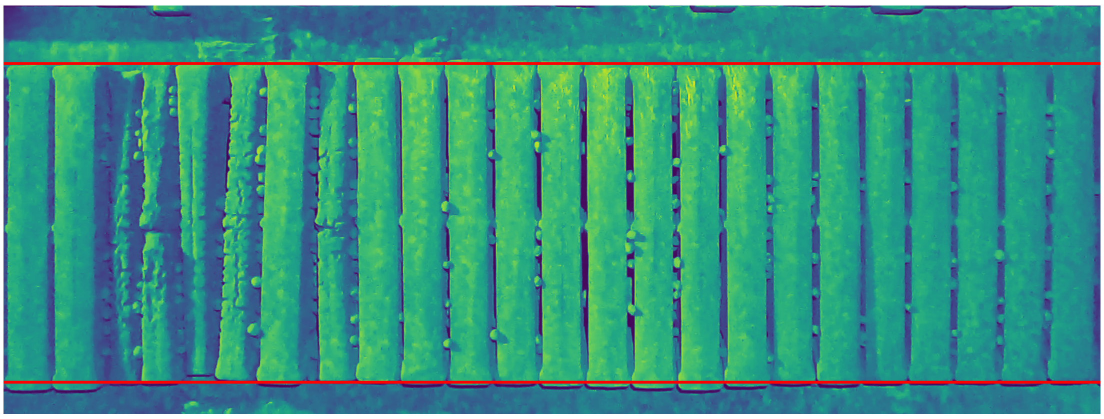


Figure 3.4: Vertically cropped image

[3]Just to remind: The whole width of an image is expected to be covered by ribs, but ribs do not have to fill whole image in the vertical direction.

3.3.4 Removing cracks in threshold image

Processing vertically cropped image, we know, there are only 3 types of objects in the image: ribs, pellets, and spaces. Spaces are thresholded, pellets are not and parts of ribs can be thresholded, thought the majority of their area is unthresholded.

First of all, we will find all fragments of ribs. Fragments of ribs are unthresholded areas, which were ribs in the cropped image. The reason, why I use term fragments of ribs instead of ribs is, that rib can be split into more fragments by thresholding. To make this clear, in the majority of cases rib stays connected and forms one fragment in threshold image, but there are exceptions, like the rib, which is cracked in middle (for more information see 4.3.3).

To find fragments of ribs, we will use function Extract objects (see 3.2.1) with minimal size parameter about 25% of undamaged rib area. So only big connected parts of threshold image will be taken, ignoring all individual pellets. A very common case in this phase of the algorithm is, that one fragment is formed by multiple ribs. This is caused by pellets between ribs, which connect neighboring rib into one connected component. For now, this is not important and we will deal with this fact later.

Our extracted components are connected, though they aren't continuous. Typically, there are deep cracks in ribs, forming something like gulfs or lakes in threshold image (see green areas in Figure 3.5). Because thresholded cracks interfere deep into a rib, but takes just small area of it, Line matched filter function (see 3.2.4) applied on each fragment will remove them, resulting in continuous rib fragments (see Figure 3.5).

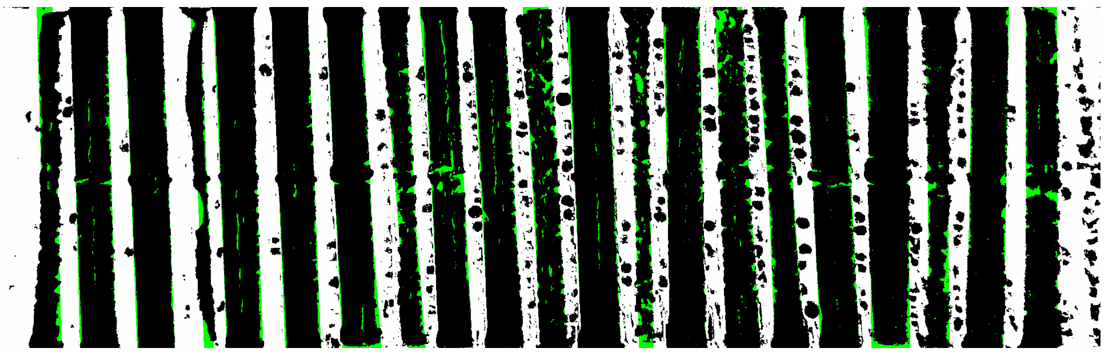


Figure 3.5: Rib masks reconstructed by the Line matched filter.

As stated above, in this step, we use the Line matched filter for multiple fragments (multiple ribs) at once, since they are connected into one component. Rib angles of each rib in a component can be slightly different, resulting in detected angle as the average of all ribs forming one component. Therefore, quite a high Line matched filter coefficient must be used (about 0.9), not to connect rib fragments together, due to angle detection inaccuracies.

3.3.5 Removing vertical rib cracks in threshold image

Thresholded cracks have been removed, but there can be vertically oriented cracks in the middle of rib, which were not removed by matched filter due to insufficient match. Those thresholded cracks are now just small, bounded holes in rib, much smaller than spaces in between ribs. Applying of function Remove holes in mask (see 3.2.2) with sufficient small maximal hole area parameter will remove rest of cracks in ribs without removing spaces in between ribs.

3.3.6 Disconnecting pellets from ribs

We still have a bunch of rib fragments connected by pellets into one component or pellets, which are connected to fragment and therefore not removed. To get rid of at least some of them, we can use the distance transform followed by thresholding with a small threshold value.

This operation will remove just a small percentage of continuous rib mask compared to pellets, where significant percentage of area is removed. Majority of pellets connected to ribs will be disconnected by this operation.

3.3.7 Inverting image

For following two steps, we will use the inverted binary image, so ribs and spaces will have inverted values in a binary image. Consequently, each algorithm or function applied to the inverted image will understand spaces and ribs and vice versa.

3.3.8 Removing disconnected pellets

From an inverted point of view, pellets disconnected from ribs are now just small holes inside rib (space in the cropped image). To remove them, we will use function Remove holes in mask (see 3.2.2) applied on this inverted image.

3.3.9 Removing rest of pellets

In the previous step, we have removed most of the pellets, but not all of them. Residual pellets are still random spread in spaces in the image and connected to ribs.

We have already been in a similar situation before. Then, we have had straight ribs with random cracks thresholded out from it. When you think about it, spaces are more or less straight and in the inverted image, they look exactly the same, as ribs did before. So using of Line matched filter (see 3.2.4) function on the inverted image should solve our problem and remove rest of pellets in the image (see Figure 3.6).

Maybe, you could ask why have we not used Line matched filter already instead distance transform, thresholding and removing holes. Problem is, our ribs are not complete yet. We have continuous rib fragment without holes, but still, big parts of ribs are thresholded out because of shadows, or because they are just dirty and therefore darker (see 3rd rib in Figure 3.6). So we need to use Line matched filter with quite a big coefficient (about 0.8) not to remove parts of ribs, but to remove just pellets. For this reason, we need an image with a minimum of pellets, otherwise matched filter would not be able to match space and remove pellets in it. That is why we had to use the distance transform before.

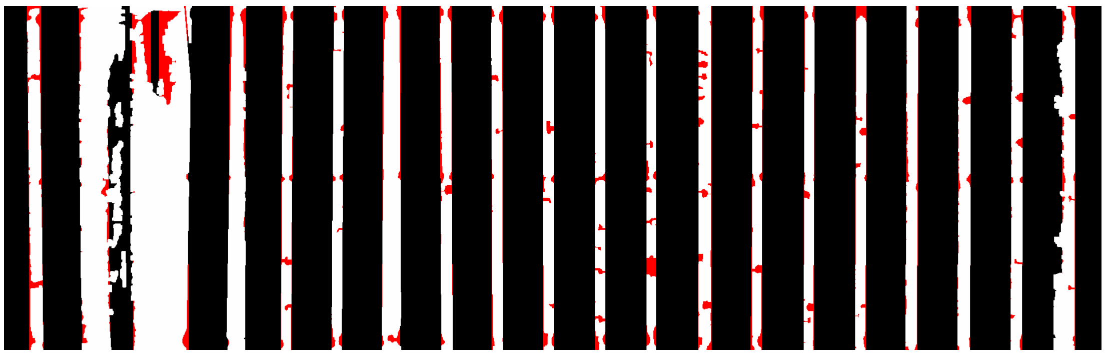


Figure 3.6: All pellets removed by Line matched filter.

3.3.10 Reconstructing ribs

We finally have the image only with spaces and ribs. Our ribs are incomplete, but what is important, pellets are completely removed from the image. So we will invert previously inverted image back and finally find all of the ribs in the image.

Function Extract objects (see 3.2.1) will again find us all rib fragments. This time, we already know, that each rib fragment belongs exactly to one rib. That is because there are no pellets, which could connect multiple ribs into one fragment. On the other hand, one rib can be still represented as multiple fragments with holes between them. To connect fragments into ribs, we will again use Line matched filter (see 3.2.4) function. Because pellets are already out, we can finally afford to use much lower coefficient (let's say 0.4), therefore repairing parts of ribs, we were not able to repair before (see Figure 3.7).

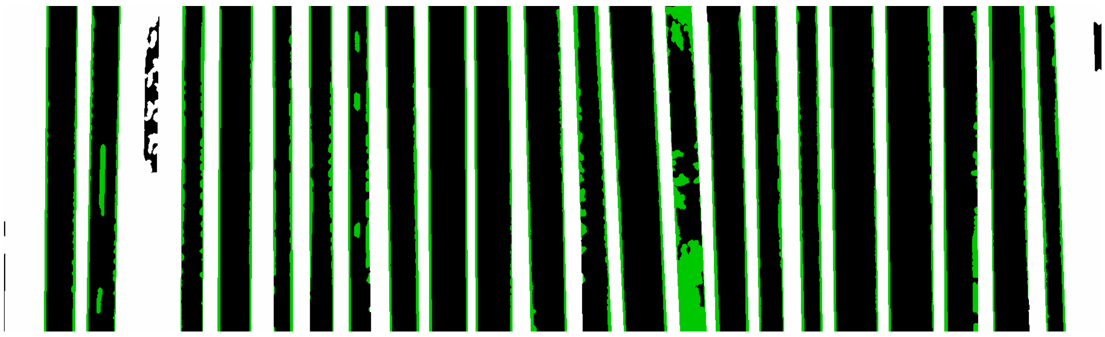


Figure 3.7: Reconstructed rib masks

3.3.11 Improving rib masks

The whole rib is sure to be inside our mask, but due to Line matched filter, our masks have straight edges and are more or less rectangular. But we would like to have much better rib shape descriptions than this.

To achieve this, we use Otsu thresholding again but with a small modification. We will Otsu threshold only our ribs (their shapes from the previous step), with rest of the image set to zero. Because the majority of the thresholded image is black, only truly dark parts (i.e. spaces round rib) will be removed, but the whole rib will be kept unthresholded.

On the other hand, there can still be small, dirty areas of ribs, which will be thresholded even by this thresholding. To produce a continuous mask, we will use function Remove holes in mask (see 3.2.2) to repair those threshold inaccuracies. Now we finally have masks which describe real rib shape.

3.3.12 Postprocessing, further analysis

Now, when we have rib masks, we have ended rib finding itself. Our output is a list of rib masks, each representing one rib in the original image.

However customer required further outcomes of the algorithm, than only rib binary masks. For example, one of the requirements was the ability to store data about each pellet car and the possibility to track it over time. As we all can see, binary masks are not the most saving data format to store, same as comparing binary masks is not the most effective way how to track pellet car changes over time. For this reason, the rib finding part is followed by several statistical algorithms, producing compact and intuitive representation.

List of those statistical algorithm follows:

- First of all, principal component analysis (PCA) is performed on each rib mask, producing values as rib width, height, area, the center of mass, etc.
- Using rib width and center of mass, the width of spaces between ribs is calculated as the horizontal distance of centers of mass minus half of width of each rib.
- Rib angle is calculated for each rib, using Rib angle (see 3.2.3) function.
- Even though the average width of each rib had been found by PCA, minimum width of each rib is found too. The reason is that rib falling down off pellet car will be shaded by neighboring ribs, resulting in narrowing of rib mask in the area, where it is cracked.
- Last of all, cracks in ribs are found as another measure of rib damage. Canny edge filter on masked rib image is used and the amount of crack pixels found is compared to rib total area, producing rib damage index.

Results of all operations described above are returned as the output of this algorithm. To make all those data (including rib masks) more user readable, I have created an overview image (see Figure 3.8).

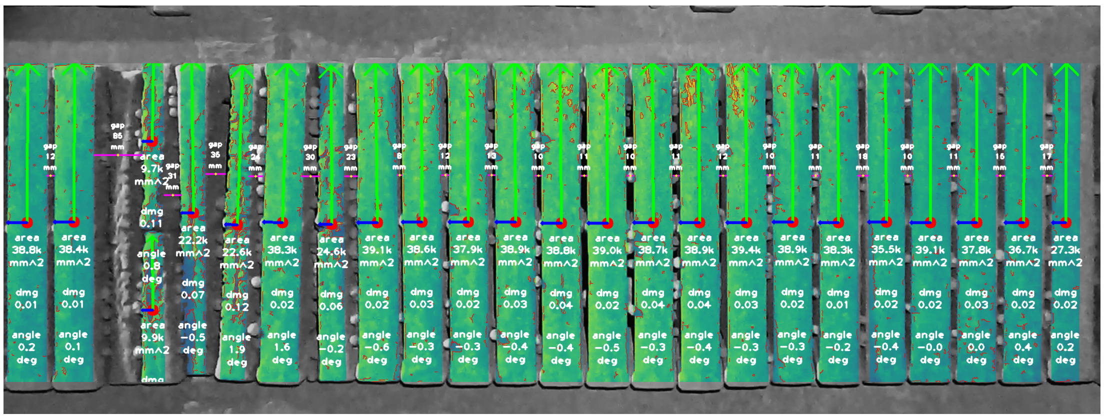


Figure 3.8: Overview image of algorithm result.

Chapter 4

Evaluation

4.1 Basic features

Some algorithm outputs are obvious and were mostly already described during the algorithm description.

Here comes the list of those features:

- Rib shape, described by binary masks.
- Rib count, as the count of rib masks, returned (except for situation described in 4.3.3).
- PCA results: area, the center of mass, width, height, etc.
- Widths of spaces between ribs, calculated using rib width and center of mass from PCA.
- Rib angles, which is calculated several times, all over algorithm execution.

4.2 Damaged ribs detection

Until now, we have spoken only about the finding of ribs in an image. But the original purpose of this algorithm was pellet car damage evaluation. So in this section, we will discuss, which damage is this algorithm able to detect and how.

4.2.1 Degraded ribs finding

First casual rib damage is degraded rib. Degraded rib lost its original shape by gradual rounding (see Figure 4.1). Degrading of ribs leads to higher iron material losses, as spaces are larger, than they should be and an iron material falls off the conveyor belt.

Let's take a look, how our algorithm behaves in case of pellet car with degraded ribs. Do you remember Otsu thresholding at the beginning of the algorithm? Otsu thresholding finds ideal threshold value to split foreground and background intensity levels. And because light goes to pellet car from upwards, the intensity of light at rib sides is much lower, than the intensity of rib tops and a pellet car frame. For the same reason, the intensity of degraded rib sides is much lower,

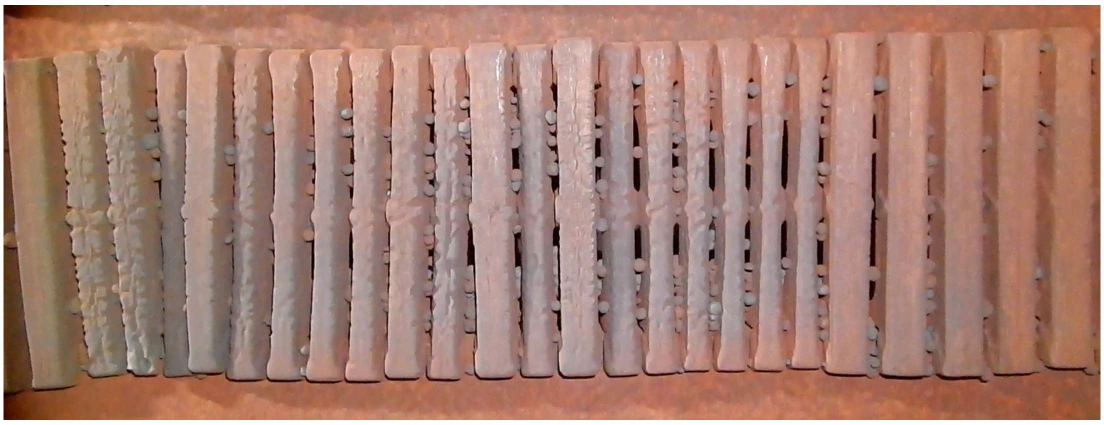


Figure 4.1: Pellet car with rounded (degraded) ribs.
.

causing Otsu threshold to remove its degraded sides too. Overall, degraded ribs are detected as ribs, which are significantly narrower, than non-degraded rib should be.

4.2.2 Pushed down ribs

Another common kind of pellet car damage is pushed down rib, meaning rib, which is located below the level of other ribs in pellet car. Such ribs are typically either cracked in the middle or bent by iron ore weight. Pushed down ribs are about to fall off pellet car soon, so they have to be detected and such pellet car must be replaced during the next service stop. Example of pushed down rib can be seen in Figure 4.2, where pushed down rib is marked by red arrow[1]. But how

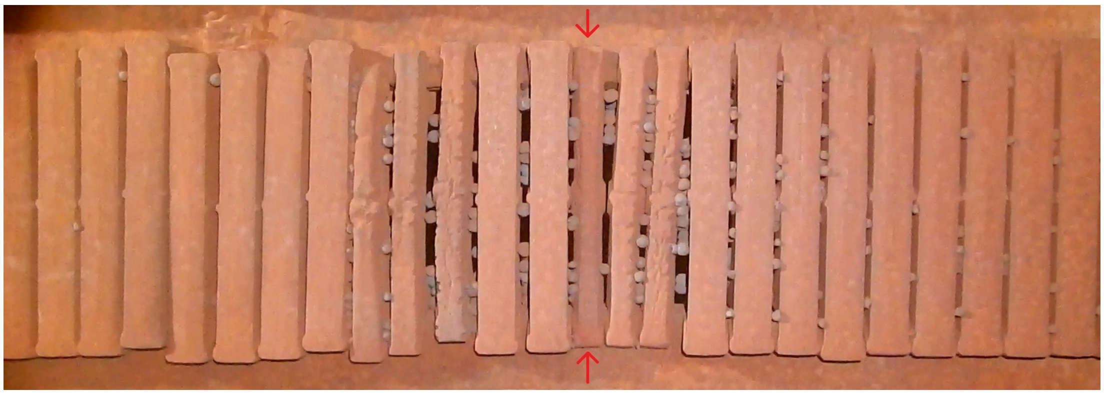


Figure 4.2: Pushed down rib.

to detect those ribs?

Same as last time, the answer is again Otsu thresholding. Pushed down rib is located below other ribs, so it is shaded by other ribs and consequently less illuminated. After Otsu thresholding, at least center part of pushed down rib does not appear in thresholded image and the rest of the rib is cleaned away as noise (pellets etc.) in further steps of the algorithm.

[1] Generally, it is quite complicated for a human to find pushed down ribs in an image. This is caused by human eye adaptability to different light levels in different areas of an image.

Therefore, pushed down ribs appear in algorithm output as the missing rib. Such behavior might sound incorrect at the beginning, but after a deeper consideration, we can conclude, that pushed down ribs, which are about to fall off, shouldn't be detected as normal ribs. So in the end, this algorithm behavior is exactly what we want.

Moreover, the algorithm is much better in finding such ribs, than human. You can see this fact in Figure 4.2, where human can barely recognize pushed down rib, but the algorithm has no problem to find it.

4.2.3 Rib cracks

Less serious, but still important rib damage are cracks in rib surface (see Figure 4.3). Those cracks are caused by thermal expansion of rib due to continual warming and cooling of pellet car. Because there could be a connection between cracks in rib surface and cracking of ribs into pieces, our algorithm should watch those damage too.

As described in 3.3.12, Canny edge detector is used and the number of edge pixels found is compared to the rib area. Resultant rib cracks index then indicates how much is rib surface covered with cracks, where higher index means more cracks.

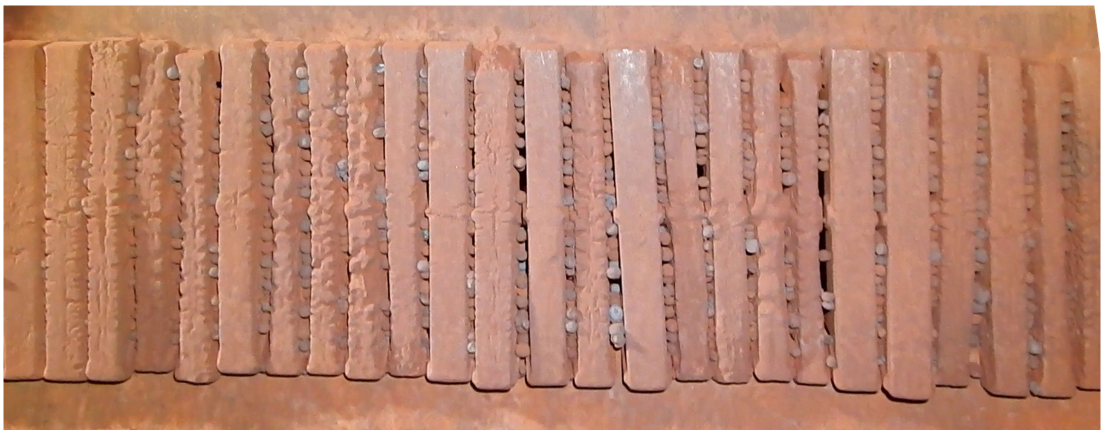


Figure 4.3: Rib cracks.

4.3 Unexpected situations

During testing of the algorithm, several surprising pellet car states appeared. Those states have never been mentioned before, so the algorithm hasn't been designed to deal with them. Very often, it isn't even obvious, how exactly should algorithm behave in case of such pellet car states.

4.3.1 Holes in ribs

In one test image, I have noticed a hole in rib. The hole makes no problem to rib detection itself, as the current algorithm removes all holes in masks after last Otsu thresholding. This step is on the one hand necessary to produce a continuous rib mask. But on the other hand, possible rib hole is unfortunately removed in this step too.

Because hole detection was never required as algorithm output, this behavior should not be problematic. If hole detection was missing some time in the future, there could be (in my opinion) implemented by another thresholding with a fixed threshold value. Such altitude would probably work because hole (the one I have seen) looks significantly darker than messy parts of ribs (it is almost black).

4.3.2 Different rib vertical position

Another state of pellet car, which can be seen in test images, but was not described by the customer is a different vertical position of ribs. Of course, pellet car ribs can't significantly move in the vertical direction due to pellet car construction. Even though, when you take a look at Figure 4.4, you can see, vertical displacement can be at least noticeable.

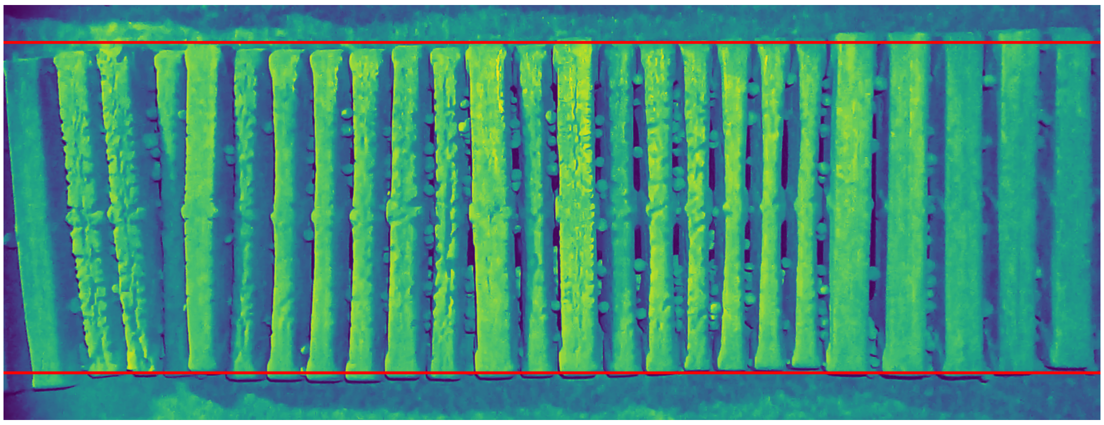


Figure 4.4: Vertically displaced ribs.

As described in section 3.3.3, all ribs are vertically cropped at the same height. Consequently, in case of vertically displaced ribs, the rib is cut on one side. Analogously, part of pellet car on the other side of the rib may be detected as a rib, despite last thresholding, which is designed to differ rib from space, not to differ pellet car from a rib.

If vertical displacement of detected masks appears to be a problem, the solution would be a bit more complicated, than in case of the previous problem. I would suggest an additional step of the algorithm located before last Otsu thresholding when the algorithm has rectangular masks of ribs. Those masks could be used to original Otsu thresholded image, where rib and spaces are well visible thus can be used to correct vertical detection. The principal of this step would be vertical cropping based just on local neighborhood followed by shifting rib mask in its direction.

In the current algorithm, vertical displacement of ribs is ignored and the necessity of vertical crop modifications will be discussed with the customer. Vertical misdetection can cause slightly incorrect rib length and area measurement, resulting for example in lower rib cracks index.

4.3.3 Broken apart rib

The most surprising pellet car state was rib cracked into two pieces, which hadn't fallen off pellet car yet (see Figure 4.5). Until testing phase, where algorithm detected only the half of rib, I have expected, that broken rib immediately falls out of pellet car. More surprisingly, despite this assumption, the algorithm was able to correctly identify cracked rib (understand to produce a correct mask of each rib piece). On the other hand, other parts of the algorithm weren't designed for cracked ribs and had to be slightly modified to produce correct results.

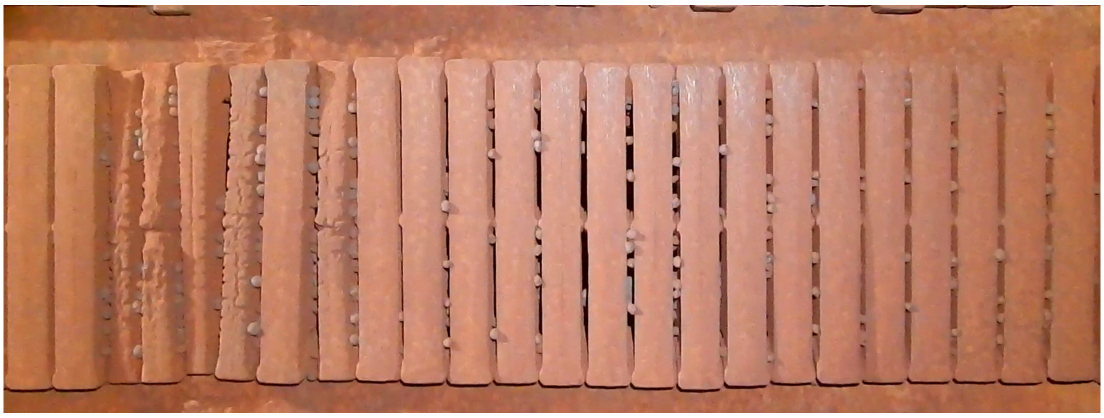


Figure 4.5: Pellet car with a broken rib.

There are two possible behaviors of the algorithm, in case of a broken rib. The first option is to find two half-sized masks, which will be returned as two independent ribs. The second option is to look for broken ribs and connect masks of parts into one rib mask, producing single discontinuous rib mask. From those two options, I have chosen the first one. It is more correct as rib pieces are independent ribs despite the fact, they used to be one rib in the past. In addition, this solution is easier to implement and less time-consuming.

The first and most important modification was necessary for the last Otsu thresholding. In this stage, rectangular masked ribs are again thresholded to correctly identify rib, which is never ideally rectangular. But after thresholding, there is typically one continuous rib and many tiny components. Originally, the algorithm found the biggest component and throw all other. It worked correctly for non-broken ribs, but in case of broken rib always discarded smaller of two parts. For this reason, not only the biggest component must be accepted as the rib. Instead, all components bigger than minimal area, which is set as input of algorithm, are taken as ribs. After this modification, all parts of cracked rib were detected correctly.

Another problem came with space detection. Spaces are measured as horizontal distance of rib centers of mass, minus half of the sum of neighboring rib widths. In case of broken ribs, which horizontal distance of centers of mass is almost zero, space width was suddenly negative. Simple ignoring of all spaces with negative width was sufficient modification. The only disadvantage of this solution is, that the number of spaces does not equal to number of ribs minus one.

4.4 Incorrect detections

Even though most test images were analyzed correctly, there were a few images, which result was not 100% correct.

4.4.1 Insufficient illumination

In some test images, not all of the ribs which should be detected were actually detected. After an examination of those images, I concluded, that all of them were insufficiently illuminated. Whereas those differences in illumination were not slightly different light level over the image, but real shadows over a big part of pellet car (see Figure 4.6).

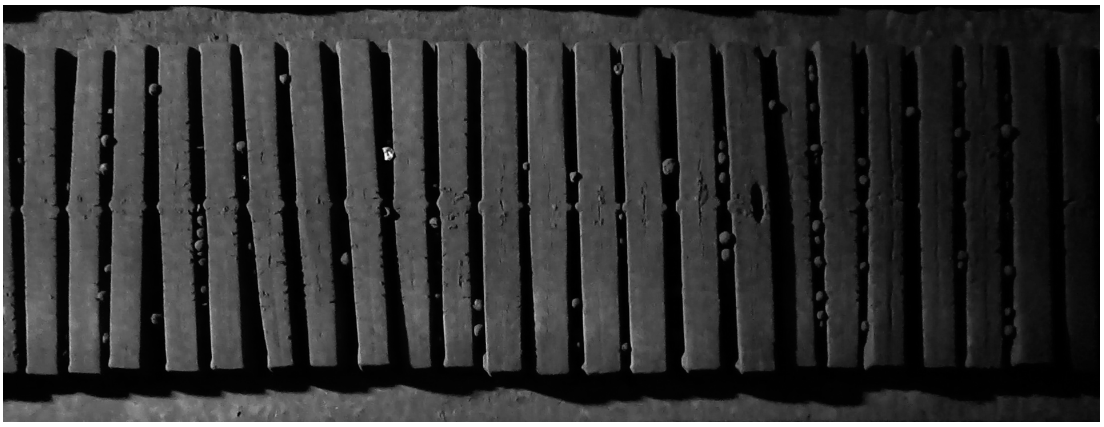


Figure 4.6: Insufficiently illuminated test image

In the final setup, there should be led belt flashlight to ensure sufficient and uniform illumination all over the image. Because the algorithm worked well on all uniformly illuminated test images, it should work in the final setup too. For this reason, there is probably no need to modify the algorithm for this reason.

4.4.2 Rib at the edge of an image

Another problem, or misdetection, which will have to be discussed with the customer is behavior of the algorithm at image vertical edges. In many test images, edge ribs are not captured whole, but there is just part of rib captured in an image.

Because the algorithm uses rib area for removing of pellets, if only narrow part of a rib is captured, it is removed as a pellet. Much bigger problems are ribs, which are sufficiently big to pass thresholding mentioned. Those ribs are correctly detected, but their masks are for example narrower, resulting in smaller rib area measured by the algorithm. Overall, behavior of the algorithm at the edge of the image is more or less unpredictable and incorrect in case, the edge goes through a rib.

There basically three simple solutions to this behavior. First, the algorithm can ignore all ribs at image edges. Let's say, that all ribs, which have the center of mass closer to the edge, than some threshold value are discarded. Another solution would be setting cameras in the factory to such position, that edge of the image will be always in space between ribs. Due to the construction of pellet cars, ribs should be more or less at the same position in all pellet cars. Yet, this solution is probably the worst of those three, as we have already seen pellet car, with shifted ribs. Last of those three solutions is based on an intention to watch whole width of pellet car. For this purpose, about 4 cameras are expected to be installed above pellet car next to each other. Because the position of cameras above pellet car belt will be known as well as their distance, those 4 images can be composed into one. This one, very wide image will then contain all ribs and its edge will be at the edge of pellet car, which is at a fixed horizontal position. In my personal opinion, the third solution is the easiest one and unlike other solutions ensures, that each rib will be measured exactly once.

Chapter 5

Implementation

In this section, I would like to shortly discuss possible algorithm implementation and present my reference implementation, which was used to evaluate test image results.

5.1 Before coding

First of all, we should choose libraries, we will use and programming language. I decided to for OpenCV library. OpenCV is an open source image processing library[35], which is free for academic same as commercial usage. It supports multiple platforms: PC, Android, iOS, CUDA, and OpenCL. Several programming languages are supported by OpenCV including C++, Python, Matlab, Java, C# and many others[36].

When I had my image processing library chosen, I had to decide, which programming language to use. Because I was at the beginning of my research and I expected a lot of testing, I took Python. Honestly, it was a great choice. Later, I tried OpenCV for Java and C++ as well and none of them was as easy to use as Python version. In Java, I even had a problem to find current documentation of some of the basic functions. Overall, I was very satisfied with Python OpenCV, which cooperates with numpy and together make a very powerful, comfortable and easy-to-use setup.

As a development environment, I used Jupyter Notebook. For those, who do not know it, Jupyter is an open source web application[37], which allows you to mix code and markdown text, so it's quite easy to take your project, including images and graphs, add some description and generate nice pdf progress report. But the biggest advantage of Jupyter Notebook over another IDE was its ability to keep Python program global state. In Jupyter, you have cells of code. Dividing of code into cells is fully up to you and you can run each cell individually. But when execution of cell modifies Python program global state (define a function, modify global variable, etc.), this state is kept for all other cells, which run after it. For me, this was a great advantage, which allowed me, for example, to load test images only once at the beginning of my workday. This feature saved me plenty of time, as I didn't have to run the whole program every time, I wanted to modify a single constant or single line of code. Moreover, I was not forced to write efficient research code and could focus only on the purpose of the code. Honestly, I can recommend Jupyter Notebook a perfect research tool.

5.2 Implementation

As you could see in Chapter 3 Description of an algorithm, the algorithm has a pipeline structure and is relatively short (about 10 stages). So I implemented the algorithm as a single function, which contains the whole pipeline and just calls subroutines and library functions.

The algorithm takes an image and about 20 constants as its parameters. Unlike image, which is easy to pass to function, you don't want to pass 20 values as function parameters separately. So I used Python namedtuple collection to pass all constants as a single argument. In C++ or Java, I would have to write some struct/class to store those values, as in those languages, it is not possible to keep them in single collection, due to different data types.

Additionally, my algorithm implementation takes the third parameter. This algorithm is boolean, indicating whether the algorithm should produce data visualization images (those are images, you can see all over this thesis). Because drawing of images takes a lot of time, and they are quite big, in final setup, this boolean will be probably always false.

My algorithm has a single output, which is a Python dictionary collection. Again in C++ or Java, it would be class, or maybe only interface in case of Java to allow some advanced tricks, as lazy evaluation of data visualization images, etc.

5.3 Possible parallelization

Nowadays, even the worst PC has multiple CPU cores and what is probably more important, single thread performance of CPUs does not grow significantly over time. Consequently, parallelization is very often the best way how to speed up program execution. So in case of real-time pellet car analysis, we should consider possible ways of algorithm parallelization.

First of all, we could take a look at the algorithm and parallelize individual parts. One such part could be the usage of line matched filter, which is executed for each rib separately, so we can parallelly process multiple ribs. On the other hand, there is still a significant part of an algorithm, which can't be parallelized. Mostly, because of pipeline stages, which are not implemented as a parallel algorithm in OpenCV. So this way we will barely reach remarkable speed up.

Another possibility is the parallel pipeline design pattern. In this pattern, each stage of the pipeline is executed in a separate thread and those stages pass data to each other using thread-safe queues or similar data structures. Designing parallel pipeline, programmer focus on dividing an algorithm into a few stages, which all should take approximately the same time not to create a bottleneck.

To cut a long story short, using the parallel pipeline pattern for this algorithm would not be the best choice. First of all, it would be quite hard to split the algorithm into several parts, which would take all roughly the same time. Secondly, parallel pipeline usually fits use cases, where we need fast stream processing. Typical usage is TV signal decoding when TV has just a few milliseconds to decode video and it has to decode frame-by-frame in a given order (i.e. stream). In case of pellet car processing, we don't really care about latency, but only of throughput. For those reasons, the parallel pipeline is too complicated way of parallelization and therefore not suitable for this kind of algorithm.

In my opinion, the best way of parallelization is a concurrent analysis of multiple pellet cars. First of all, unlike the parallel pipeline, we don' have to modify an algorithm and find its parts with similar execution time. Instead, we have each algorithm as one part, producing an equal load of all CPU cores. Secondly, there is no synchronization overhead, because there is no synchronization needed. Each thread takes one image and set of parameters as it's input and produces set on output data and images. There are no shared data between threads. At least during algorithm execution. Sure there has to be some synchronization while writing results to database or file, but this kind of synchronization is actually trivial to implement.

In my reference implementation, I used the third method mentioned and parallelization was really trivial. I exactly only took the serial algorithm and wrapped it by for loop, which executes each the serial algorithm in one thread[1]. Because the only requirement for algorithm speed says, that it has to be able to analyze as fast, as pellet car belt moves, this solution can be used in the final setup as well.

5.4 Performance

My implementation in Python executes on a single FullHD image (1920x1080px) about 4.5s without drawing output images. Algorithm together with drawing of output images, takes about 16s. Those tests were performed in a single thread and my CPU is Intel i7-4702MQ (Haswell) with 16GB of RAM (which is far not an issue). I expect, that execution time in final setup (with better hardware) could be about 3s, which meet the requirements of the customer for real-time execution.

[1]Processes actually, as Python currently does not allow parallel thread execution due to interpreter global lock.

Chapter 6

Conclusion

At the beginning of this thesis, we had briefly introduced the modern iron production process with focus on pellet/sinter burning. We'd stated, that current way of monitoring pallet car damages is very obsolete considering nowadays computer possibilities and upcoming industry 4.0 revolution. One of possible solutions is using computer vision algorithms to analyze pellet car damage from its image. This thesis focused on inventing of such algorithm.

With given assumptions on the input image (described in 3.1 Expected input image parameters), the algorithm itself starts with threshold image and its main steps are:

1. To find the area covered by rib in the vertical direction.
2. To join rib threshold pieces, which were disconnected by thresholding.
3. To find individual ribs by removing pellets from the threshold image.
4. To produce damage indices and shape description from rib masks.

Algorithm final output consists of rib masks and damage indices, which are produced from rib masks and should somehow correspond to real rib damage.

When we had our algorithm described, we had to discuss, how it works, as its main purpose was not to find ribs in an image, but to evaluate the damage of pellet cars. As we stated in 4.2 Damaged ribs detection, all common rib damages like degraded ribs, pushed down ribs, and rib cracks are detected by this algorithm. Further, we discussed several situations, which were not expected during algorithm design and were never described by the customer. For each of them, we specified necessary changes in the algorithm and outlined possible solutions of those them. Overall, we concluded, that algorithm works and that is can be used to evaluate the damage of pellet cars.

In the last part of this thesis, I briefly introduced my reference implementation with focus on the programming language, libraries, and developer tools I used. Moreover, I discussed the advantages and disadvantages of possible parallelization methods of the algorithm. And last, but not least, we discussed algorithm performance.

List of Figures

Bibliography

[1] "Iron ore - wikipedia." https://en.wikipedia.org/wiki/Iron_ore. (Accessed on 03/03/2019).

[2] "Pelletizing - wikipedia." https://en.wikipedia.org/wiki/Pelletizing. (Accessed on 03/03/2019).

[3] "Sinter plant - wikipedia." https://en.wikipedia.org/wiki/Sinter_plant. (Accessed on 03/03/2019).

[4] "Blast furnace - wikipedia." https://en.wikipedia.org/wiki/Blast_furnace. (Accessed on 03/06/2019).

[5] "Pig iron - wikipedia." https://en.wikipedia.org/wiki/Pig_iron. (Accessed on 03/03/2019).

[6] "Slag - wikipedia." https://en.wikipedia.org/wiki/Slag. (Accessed on 03/06/2019).

[7] "Výroba oceli – wikipedie." https://cs.wikipedia.org/wiki/V%C3%BDroba_oceli. (Accessed on 03/03/2019).

[8] "Basic oxygen steelmaking - wikipedia." https://en.wikipedia.org/wiki/Basic_oxygen_steelmaking. (Accessed on 03/07/2019).

[9] J. Muni, "Pellet car — 3d cad model library — grabcad." https://grabcad.com/library/pellet-car-1, 7 2013. (Accessed on 03/03/2019).

[10] E. R. Davies, *Computer and Machine Vision: Theory, Algorithms, Practicalities.* Academic Press, 4th ed., 2012.

[11] "Basic thresholding operations — opencv 2.4.13.7 documentation." https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html. (Accessed on 09/18/2018).

[12] "Opencv: Image thresholding." https://docs.opencv.org/3.4.1/d7/d4d/tutorial_py_thresholding.html. (Accessed on 09/18/2018).

[13] "Otsu's method - wikipedia." https://en.wikipedia.org/wiki/Otsu's_method, 2018. (Accessed on 09/18/2018).

[14] "Opencv: Smoothing images." https://docs.opencv.org/3.1.0/d4/d13/tutorial_py_filtering.html. (Accessed on 09/18/2018).

[15] "Image noise - wikipedia." https://en.wikipedia.org/wiki/Image_noise. (Accessed on 09/18/2018).

[16] "Shot noise - wikipedia." https://en.wikipedia.org/wiki/Shot_noise. (Accessed on 09/18/2018).

[17] IkamusumeFan, "Cappadocia gaussian blur.svg - wikipedia." https://upload.wikimedia.org/wikipedia/commons/6/62/Cappadocia_Gaussian_Blur.svg, 7 2015. (Accessed on 01/11/2019).

[18] "Histogram equalization — opencv 2.4.13.7 documentation." https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/histogram_equalization/histogram_equalization.html. (Accessed on 12/14/2018).

[19] Konstable, "Unequalized hawkes bay nz.jpg - wikipedia." https://en.wikipedia.org/wiki/File:Unequalized_Hawkes_Bay_NZ.jpg, 6 2006. original Phillip Capper, modified by User:Konstable (Accessed on 01/11/2019).

[20] Jarekt, "Unequalized histogram.svg - wikipedia." https://en.wikipedia.org/wiki/File:Unequalized_Histogram.svg, 5 2008. (Accessed on 01/11/2019).

[21] Konstable, "Equalized hawkes bay nz.jpg - wikipedia." https://en.wikipedia.org/wiki/File:Equalized_Hawkes_Bay_NZ.jpg, 6 2006. original Phillip Capper, modified by User:Konstable (Accessed on 01/11/2019).

[22] Jarekt, "Equalized histogram.svg - wikipedia." https://en.wikipedia.org/wiki/File:Equalized_Histogram.svg, 5 2008. (Accessed on 01/11/2019).

[23] "Opencv: Histograms - 2: Histogram equalization." https://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html. (Accessed on 12/14/2018).

[24] "Opencv distance transformation." https://www.tutorialspoint.com/opencv/opencv_distance_transformation.htm. (Accessed on 12/14/2018).

[25] "Opencv: Image segmentation with distance transform and watershed algorithm." https://docs.opencv.org/3.1.0/d2/dbd/tutorial_distance_transform.html. (Accessed on 12/14/2018).

[26] "Structural analysis and shape descriptors — opencv 3.0.0-dev documentation." https://docs.opencv.org/3.0-beta/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=connectedcomponents. (Accessed on 12/14/2018).

[27] "Opencv 3 image edge detection : Sobel and laplacian - 2018." https://www.bogotobogo.com/python/OpenCV_Python/python_opencv3_Image_Gradient_Sobel_Laplacian_Derivatives_Edge_Detection.php. (Accessed on 12/14/2018).

[28] "Opencv: Sobel derivatives." https://docs.opencv.org/3.2.0/d2/d2c/tutorial_sobel_derivatives.html. (Accessed on 12/14/2018).

[29] "Laplace operator — opencv 2.4.13.7 documentation." https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html. (Accessed on 12/14/2018).

[30] "Spatial filters - laplacian/laplacian of gaussian." https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm. (Accessed on 12/14/2018).

[31] "Opencv: Canny edge detection." https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html. (Accessed on 12/14/2018).

[32] JonMcLoone, "Ääretuvastuse näide.png - wikimedia commons." https://commons.wikimedia.org/wiki/File:%C3%84%C3%A4retuvastuse_n%C3%A4ide.png, 6 2010. (Accessed on 01/11/2019).

[33] "Opencv: Hough line transform." https://docs.opencv.org/3.3.1/d3/de6/tutorial_js_houghlines.html. (Accessed on 12/14/2018).

[34] "Hough line transform — opencv 3.0.0-dev documentation." https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html. (Accessed on 12/14/2018).

[35] "Opencv library." https://opencv.org/. (Accessed on 02/27/2019).

[36] "Opencv - wikipedia." https://en.wikipedia.org/wiki/OpenCV. (Accessed on 02/27/2019).

[37] "Project jupyter — home." https://jupyter.org/. (Accessed on 02/27/2019).