Graphical Analysis - Literature Study

Rafael De Smet

December 06, 2016

Contents

1 Introduction

In this paper I will be discussing several graphical analysis and text recognition algorithms. These algorithms will return data which are used to base the music generation on. Not all algorithms are used in the code, but are discussed for the sake of completeness.

2 Image Processing Algorithms

2.1 Edge Detection

The concept of edge detection is pretty straightforward. The algorithms try to detect all the edges in an image. Edges are the contours of objects and shapes in the image. Edge detection algorithms all use what are called convolution kernels. A kernel in image processing is a small matrix used to apply effects to an image, such as blurring and outlining. Here we will see kernels used for edge detection only. Listed below are six of the best and most used algorithms.

- Sobel
- Frei-Chen
- Prewitt
- Roberts Cross
- LoG
- Scharr

2.1.1 Convolution kernel

Convolution is the technique of multiplying together two arrays of different size but of the same dimension. An array of dimension two simply is a matrix. When working with images the pixels are represented as a (2D) matrix and the kernel is also a 2D matrix. The kernel is a small matrix that we will multiply with the image matrix to perform the convolution. This kernel matrix is different for each edge detection algorithm. We will see examples

of different matrices later in this paper.

The idea of these algorithms is to create a new image that shows the edges of the original image. Each pixel of the original image is added to its local neighbours, weighted by the kernel. This produces a new image. The result of the matrix multiplication results in new pixel values that denote the edges of the image. Mathematically we can write the convolution as follows, with O the output image, I the input image and K the kernel. $I(i, j)$ means the pixel on the i^{th} row and the j^{th} column.

$$O(i,j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i+k-1, j+l-1)K(k,l) \tag{1}$$

An example will clarify the previous.

I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}	I_{18}	I_{19}
I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	I_{26}	I_{27}	I_{28}	I_{29}
I_{31}	I_{32}	I_{33}	I_{34}	I_{35}	I_{36}	I_{37}	I_{38}	I_{39}
I_{41}	I_{42}	I_{43}	I_{44}	I_{45}	I_{46}	I_{47}	I_{48}	I_{49}
I_{51}	I_{52}	I_{53}	I_{54}	I_{55}	I_{56}	I_{57}	I_{58}	I_{59}
I_{61}	I_{62}	I_{63}	I_{64}	I_{65}	I_{66}	I_{67}	I_{68}	I_{69}

K_{11}	K_{12}	K_{13}
K_{21}	K_{22}	K_{23}

Figure 1: A pixel matrix and a kernel matrix

Using the kernel matrix we will compute every new pixel of the output image, by sliding the kernel matrix over the original pixels. Each kernel position corresponds to a single output pixel, the value of which is calculated by equation (1).

In our example, the value of the bottom right pixel in the output image will be found as follows [2]:

$$O_{57} = I_{57}K_{11} + I_{58}K_{12} + I_{59}K_{13} + I_{67}K_{21} + I_{68}K_{22} + I_{69}K_{23} \tag{2}$$

In figure 2 you can see a painting by Picasso. We will use this image to see how well the different edge detection algorithms work.

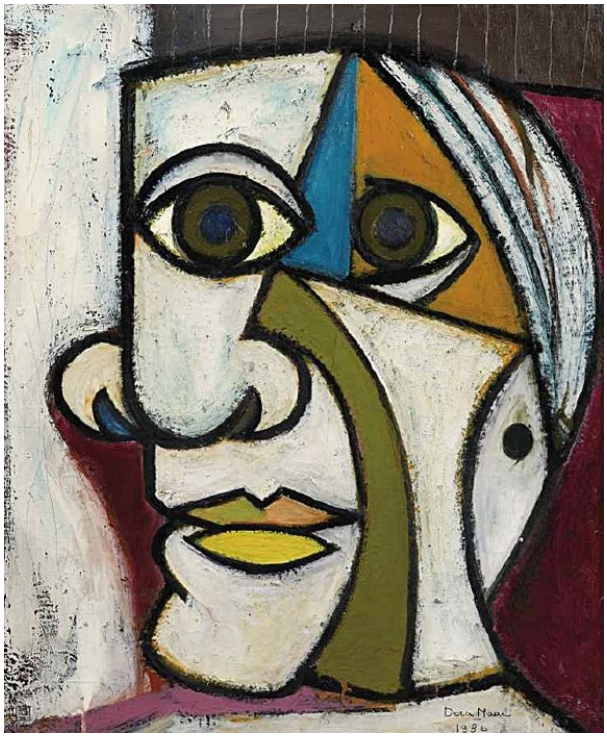

Figure 2: Picasso

2.1.2 Sobel

The Sobel algorithm performs a 2D spatial gradient measurement and defines regions of 'high spatial frequency' or edges. It uses two 3x3 kernels, one for the horizontal edges and one for the vertical edges. These two kernels are applied consecutively and the results are combined to define all the edges. In figure 3 you can see the result of the Sobel filter.

$$\text{The horizontal kernel: } \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix} \text{ and the vertical kernel: } \begin{vmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{vmatrix}$$

Figure 3: Sobel filter applied to figure 2

2.1.3 Frei-Chen

The Frei-Chen algorithm also uses 3x3 kernels, but this time there are nine different convolution kernels. The four first matrices, G1, G2, G3, G4, are used for edges, the next four are used for lines and G_9 is used to compute averages. G_9 attenuates the impact of the computations from the other matrices. Figure 4 shows the result of the Frei-Chen filter.

$$G_1 = \frac{1}{2\sqrt{2}}\begin{vmatrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & 0 \\ -1 & -\sqrt{2} & -1 \end{vmatrix} \quad G_2 = \frac{1}{2\sqrt{2}}\begin{vmatrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 1 & 0 & -1 \end{vmatrix} \quad G_3 = \frac{1}{2\sqrt{2}}\begin{vmatrix} 0 & -1 & \sqrt{2} \\ 1 & 0 & -1 \\ -\sqrt{2} & 1 & 0 \end{vmatrix}$$

$$G_4 = \frac{1}{2\sqrt{2}}\begin{vmatrix} \sqrt{2} & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & -\sqrt{2} \end{vmatrix} \quad G_5 = \frac{1}{2}\begin{vmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{vmatrix} \quad G_6 = \frac{1}{2}\begin{vmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{vmatrix}$$

$$G_7 = \frac{1}{6}\begin{vmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{vmatrix} \qquad G_8 = \frac{1}{6}\begin{vmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{vmatrix} \qquad G_9 = \frac{1}{3}\begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix}$$

Figure 4: Frei-Chen filter applied to figure 2

2.1.4 Prewitt

This algorithm is very similar to the Sobel algorithm. Again, two kernels are used, one for the horizontal and one for the vertical edges. In this case the kernels are basic convolution filters of the following form.

$$\text{Horizontal filter} = \begin{vmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{vmatrix} \qquad \text{Vertical filter} = \begin{vmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{vmatrix}$$

Figure 5: Prewitt filter applied to figure 2

2.1.5 Roberts Cross

This algorithm uses even simpler kernels than Prewitt does. This time we use two 2x2 kernels. These kernels correspond to the edges running at 45° to the pixel grid, one for each of the two perpendicular orientations. Figure 6 shows the result of the Roberts Cross filter.

$$\text{Horizontal filter} = \begin{vmatrix} 1 & 0 \\ 0 & -1 \end{vmatrix} \qquad \text{Vertical filter} = \begin{vmatrix} 0 & 1 \\ -1 & 0 \end{vmatrix}$$

2.1.6 LoG

This algorithm first applies Gaussian filtering[1] to the image and then the Laplacian method for edge detection[2], hence the name "Laplacian of Gaus-

[1]The Gaussian filter is used to blur images and remove noise and detail.

[2]The Laplacian method highlights regions in the image of rapid intensity change, so it is useful for edge detection.

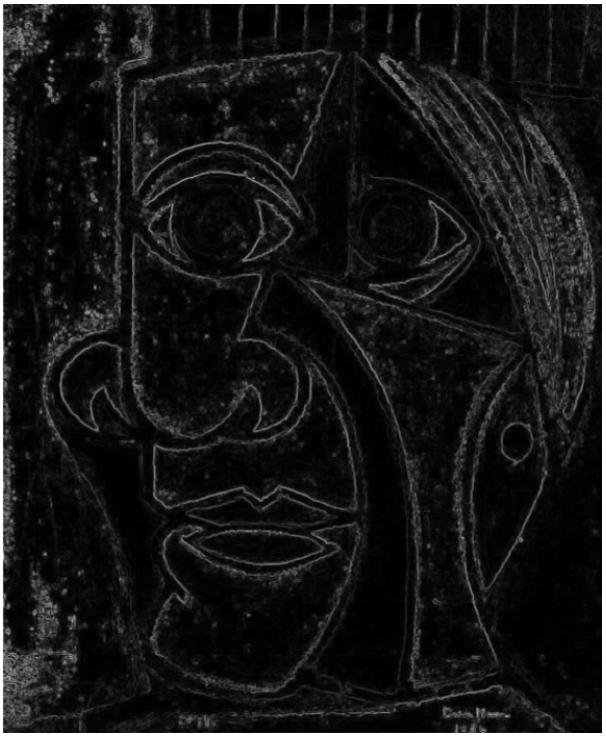

Figure 6: Roberts Cross filter applied to figure 2

sian" (LoG). The edge points of an image are detected by finding the zero crossings of the 2nd derivative of the image intensity. Because the 2nd derivative is very sensitive to noise, which could give us bad results, the Gaussian filter is used to clear the noise from the image. In figure 7, you can find the result of this filter.

The R library OpenImageR[3] uses the following LoG mask.

$$\text{LoG mask} = \begin{vmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{vmatrix}$$

[3]https://cran.r-project.org/web/packages/OpenImageR/OpenImageR.pdf

Figure 7: LoG filter applied to figure 2

2.1.7 Scharr

The last algorithm is again an extension of the Sobel algorithm. This operator improves rotational invariance, which means that if the image is rotated, the operator should define the same edges. Sobel can have difficulties with this aspect. There are many Scharr kernels, some even 5x5, but the following are most frequently used. Figure 8 shows the results.

$$\text{Horizontal filter} = \begin{vmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{vmatrix} \qquad \text{Vertical filter} = \begin{vmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{vmatrix}$$

Figure 8: Scharr filter applied to figure 2

2.2 Noise Removal

Every edge detection algorithm was based on a specific filter. Using this same idea of applying filters to images we can detect and remove typical noise, such as additive 'salt and pepper' and Gaussian noise. Analogous to the previous section, I will discuss several noise removal algorithms and the filters used. The first image of figure 9 is the image I will be using to discuss the results of the algorithms [14].

In this section I will be discussing the effects of the filters on two kinds of noise. In order to show the effects, I added two types of noises to the source image. The first is the 'salt and pepper' noise, the second image of figure 9. 'Salt and pepper' noise can be seen as sparsely occuring white and black pixels on the image. This means that some pixel values are maximized (white) and some are minimized (black). The second is the Gaussian noise, which can be seen in the third image in figure 9. Gaussian noise is statistical

noise, which means the values it can take are Gaussian-distributed.

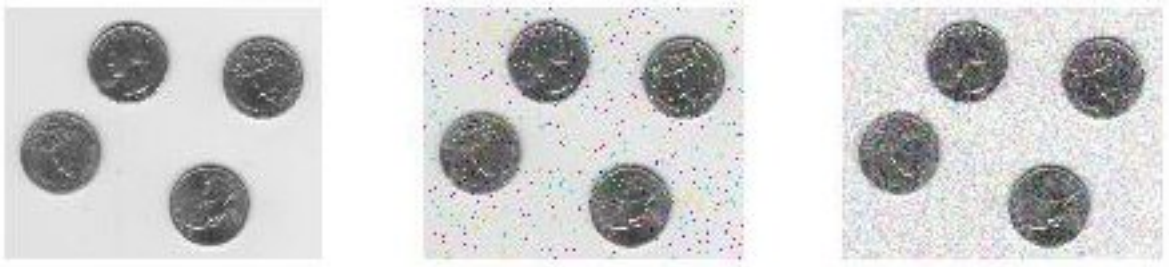

Figure 9: Original image, 'salt and pepper' noise, Gaussian noise

2.2.1 Mean filtering

This is the simplest linear filter and operates by giving weight w_K to all the pixels in the neighbourhood of a certain pixel. Let's say we have a $N \times M$ neighbourhood, then we define the weight as follows: $w_K = \frac{1}{NM}$. The filter used in this technique is a 3×3 filter, as shown below.

$$\text{Mean filter: } \begin{bmatrix} 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \end{bmatrix}$$

Figure 10 shows the results of this filter. We can see that the filter is reasonably effective at removing the Gaussian noise (third image), but at the expense of losing detail in the edges of the image. The filter is not effective for the removal of the 'salt and pepper' noise. This is caused by a large deviation of the noise values from the typical values in the neighbourhood. This means that the average value is significantly influenced. This deviation is still very visible in the result.

The main disadvantages of mean filtering are (a) it is not robust to large noise deviations (also called outliers) and (b) the edges in the images will be blurred. This suggests we should look at a more robust (to statistical outliers) filter.

Figure 10: Original image with mean filter, 'salt and pepper' noise with mean filter, Gaussian noise with mean filter

2.2.2 Median filtering

The mean filtering used the mean of a $N \times M$ neighbourhood as the weight. The median filtering uses the statistical median of the neighbourhood and is more suited because it preserves edges better while eliminating noise. Using the median is more robust to statistical outliers and does not create new unrealistic pixel values. The disadvantage of this method is the high computational cost. To calculate the median of values, requires an ordering of those values.

In figure 11 you can see the results of this filter. It is clear that the median filter is very effective at removing the 'salt and pepper' noise. The removal of the Gaussian noise still is at the expense of a slight degradation in image quality.

Figure 11: Original image with median filter, 'salt and pepper' noise with median filter, Gaussian noise with median filter

2.2.3 Rank filtering

The median filter is nothing more than a special case of another filter type, the rank filter [14]. This filter follows four common steps.

- Define neighbourhood of the target pixel ($N \times N$).
- Rank the pixels of the neighbourhood in ascending order.
- Choose the rank pixel of the filter.
- Set the filtered value to be equal to the value of the chosen rank pixel.

A common practice is to choose the minimum and maximum values of the pixels in the defined neighbourhood. Unsurprisingly, these filters are called minimum and maximum filters. In figure 12 you can see the results. The effects on the 'salt and pepper' noise is not so good. Because it selects the maximum and minimun values, the high values in the image are amplified. The effects on the Gaussian noise are better. The noise is largely removed, but at the cost of image detail.

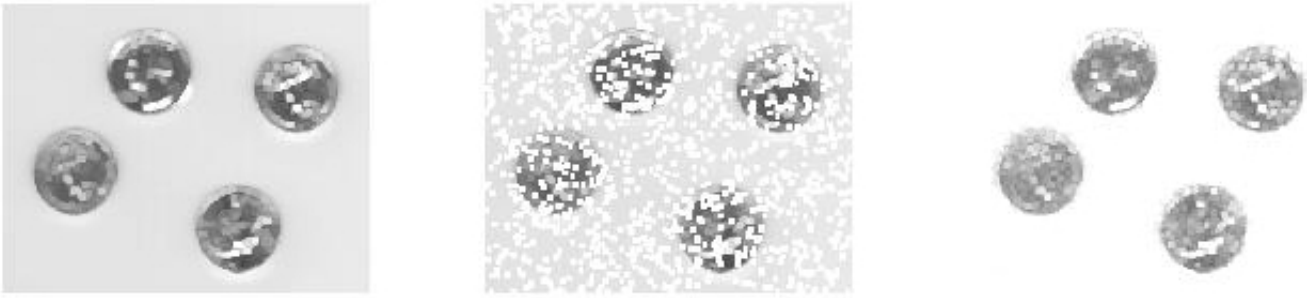

Figure 12: Original image with rank filter, 'salt and pepper' noise with rank filter, Gaussian noise with rank filter

2.2.4 Gaussian filtering

This technique differs from the other three I discussed. The filter used is a discrete kernel derived from the continuous 2D Gaussian function defined as follows [14]:

$$f(x,y) = \frac{1}{2\pi\sigma^2} exp(-\frac{x^2+y^2}{2\sigma^2}) \tag{3}$$

Applying the Gaussian filter has the effect of smoothing the image. The degree of smoothing is controlled by the choice of the standard deviation σ. The Fourier transform of the Gaussian function is a Gaussian function itself. This means that it is very convenient for the frequency-domain analysis of filters, but that discussion is out of scope for this paper.

In figure 13 you can see the results. In all cases, the smoothing effect causes the edges to fade a little, but it also removes the noise to a certain degree. For this result I used a Gaussian filter with $\sigma = 2$. This filter gives the best result of all filters, with the median filter a close second choice.

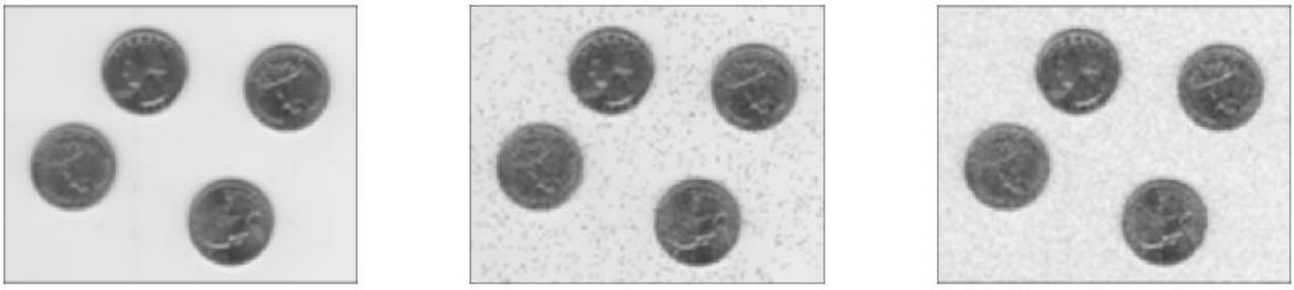

Figure 13: Original image with Gaussian filter ($\sigma = 2$), 'salt and pepper' noise with Gaussian filter, Gaussian noise with Gaussian filter

2.3 Color Analysis

Only edge detection algorithms are not sufficient to get enough data from the image to form a musical interpretation. We will need some form of color analysis as well. This section describes some algorithms and methods to obtain information about the colors of the image.

2.3.1 RGB - HSV - CMYK

To represent color on the computer the RGB representation is the most commonly used. Each pixel is described using three values, the amount of red (R), green (G) and blue (B). Using these values we can count how many pixels in the image are dominantly red, green or blue. When we're going to translate the image data into musical patterns, we can match each color to another kind of music (happy, sad, etc...)

HSV, sometimes called HSB, is another representation of a pixel. This time we use the hue (color, denoted by H), the saturation (S) and the brightness or value (B or V) to describe the pixel. Where the RGB model consists of three values indicating the amounts of a certain color each pixel has, the HSV model consists of three independent components to form a color.

Hue First we have the hue, which is the color. The color is represented using a circle with all the colors on it. The value of this component is the degrees of the angle we have to make on the circle to get this color.

Saturation The saturation indicates the fullness of the color. This value is expressed in a percentage, with 0% a gray, flat color and 100% a full, rich color.

Value of Brightness The value or brightness indicates the amount of light of the color and is also expressed in a percentage, 0% is black and 100% is white.

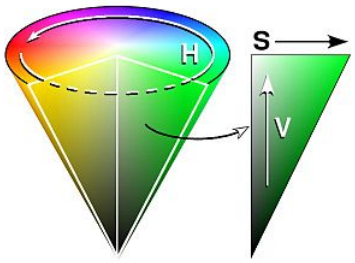

Figure 14: The HSV color representation

CMYK is another representation of color, based on the mixing of four colors, cyan (C), magenta (M), yellow (Y) and key (K, black). This model is used frequently with printing, but not very interesting for this paper.

2.3.2 Black And White Balance

During the image analysis we will convert the image to a gray scale image, so we can determine the amount of white and black in it. Using the RGB model, this is an easy process. Black in the RGB model is (0,0,0) and white is (255,255,255). So we can simply count the amount of pixels that are very close to those two values and we know the amount of black and white pixels in the image. Analogous to this we can count every gray pixel in the image, by finding every pixel where the RGB values are exactly the same.

2.3.3 Color Quantization (Image Segmentation)

Sometimes it can be easier to work with images that are partitioned in simpler regions. This technique is called color quantification or image segmentation. There are many different ways to implement this technique. One such approach is using the k-means cluster algorithm. This algorithm attempts to

partition a data set into k clusters. The data set contains the RGB values of each pixel. The disadvantage of this algorithm is that we don't know what the right value for k is. So we have to perform this algorithm a couple of times, each time with a different k k to see which value gives us a good result.

A second technique is the X-means algorithm which is an improvement of the k-means algorithm. The X-means algorithm can determine the number of clusters by itself, using the Bayesian information criterion (BIC). BIC is a selection criterion for clustermodels. Based on the total number of clusters found by X-means, we can generate parts of the piece of music.

To illustrate this technique, figure 15 shows an image of a mandrill. Using the k-means algorithm we can break this image into four color regions [13]. This means $k = 4$. The result of this segmentation can be found in figure 16. We have found four dominant colors: red, lightblue, gray and darkgray (almost black). We can get a better understanding of color quantization by visualizing our image in the color space. In figure 17 you can see the four dominant colors resulting from the algorithm.

Figure 15: Mandrill image

Figure 16: Mandrill image segmented

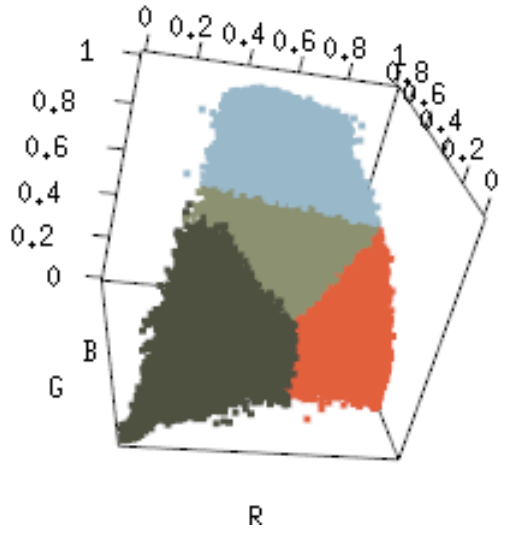

Figure 17: Segmented mandrill image in color space

2.4 Image Hashing

It is also possible to get the hash value of an image. The hash is a hexadecimal number and is another representation of the image. This is used to compare

images.

2.4.1 Average Hashing

The first hash method is the average hash of the image. This algorithm contains five steps.

1. Convert the image to grayscale.
2. Reduce the size of the image, to reduce the number of computations.
3. Average the resulting colors. For an 8x8 image, 64 will be averaged.
4. Compute the bits of the hash value by comparing if each color value is above or below the mean.
5. Construct the hash.

2.4.2 Difference Hash

We can also compute the dHash of an image. It is similar to the average hash, but now the difference between adjacent pixels is also considered in the computation.

2.4.3 Perceptive Hash

This method uses the discrete cosine transform (DCT) and compares the pixels based on the frequencies, given by the DCT, instead of the color values. DCT is a similar technique as Fourier analysis, it expresses a finite sequence of points, in this case the pixels, in terms of a sum of cosine functions.

2.4.4 Use Of The Hash

One application of these methods is the recognition of images. Search engines like Google use this technique to search similar images to any image you provide to the search engine.

In figure 21 you can see the results of applying the three types of hash to three images of a tree, figures 18, 19 and 20. It is clear that the average hash produces similar results for every tree, while there is more difference in the results of the difference hash and the perceptive hash.

Figure 18: Tree 1

Figure 19: Tree 2

	Average Hash	Difference Hash	Perceptive Hash
Tree 1	ffcf8303030187ff	00007c00b852ec7c	7f9c2185534b98f1
Tree 2	ffff8f070707dfdf	000000e804f8f800	4796a9695a6929d6
Tree 3	ffdf8f8787078708	d6878788e300f8f8	87a569535b9899c9

Figure 21: Table of hash results

Figure 20: Tree 3

2.5 Entropy

Still another method we can use to get information from an image is to calculate the entropy. Entropy is a measure of the amount of disorder in the image.

If all the pixels of the image have the same level of a specific parameter, the entropy is zero for that parameter. This means there is not a lot of information (almost none) to be gained from the image. An image of a blue sky has a very low entropy cause we only see blue.

When all the pixels in the image are different, the entropy of the image is maximum and there is a lot of information to get from it. As the entropy describes how much (dis)order there is, we will eventually use this to determine how much (dis)order there is in the generated music.

To get the right entropy information about the pixels, we use a histogram that shows the count of the distinct pixel values. The entropy of an image H is defined as:

$$H = -\sum_{k=0}^{M-1} p_k log_2(p_k) \tag{4}$$

where M is the number of unique pixel values and p_k is the count of each pixel level, i.e. how many times this pixel level occurs in the image.

To illustrate the concept of entropy, I have provided two examples. In figure 22 you can see the same painting as I used in the discussion of the edge detection algorithms. The left part of this image shows the original painting. It has quite a lot of information, there is a face with several distinguishable areas, lots of different colours and many edges. The middle part of the image shows the grayscale version of the painting. This is used to calculate the entropy [12]. The right part of the image shows the entropy for every pixel, based on the 10x10 neighbouring pixels. Red indicates a high degree of information while blue indicates a low degree.

In figure 23 you can see the results of the algorithm applied to an image with very little information. The left part of the image shows the original image, which is a blue and red image. The middle part again shows the grayscale version of the original. The right part again shows the entropy for every pixel. As expected, we only have a high degree of information on the edge between the blue and the red parts.

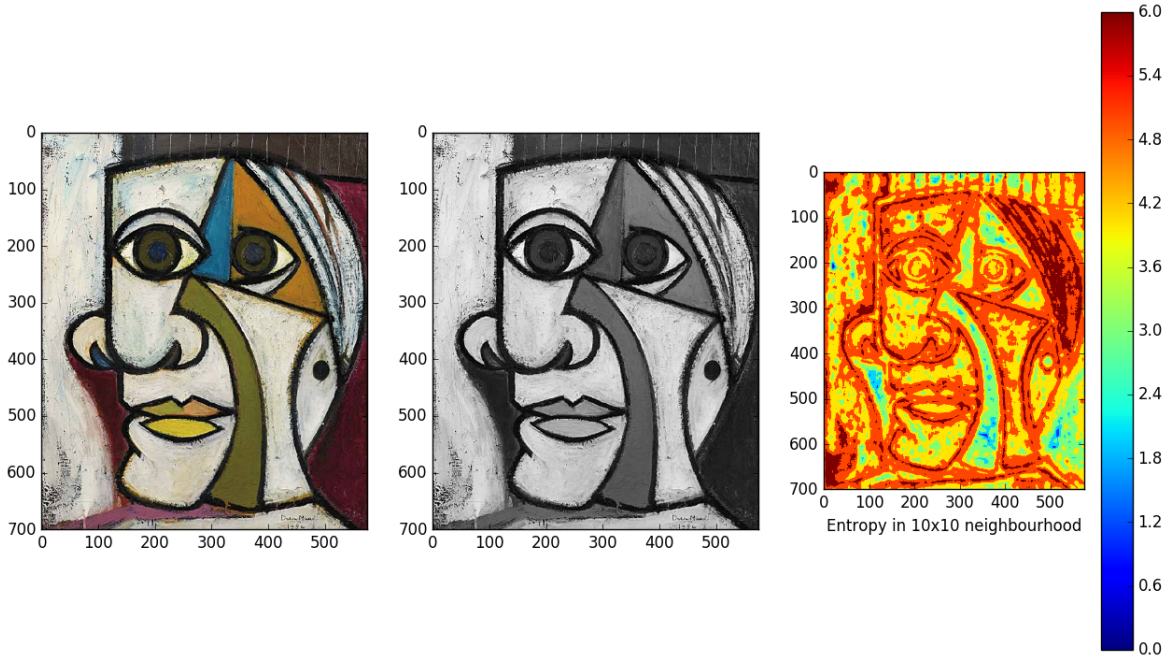

Figure 22: Entropy of the Picasso, figure 2

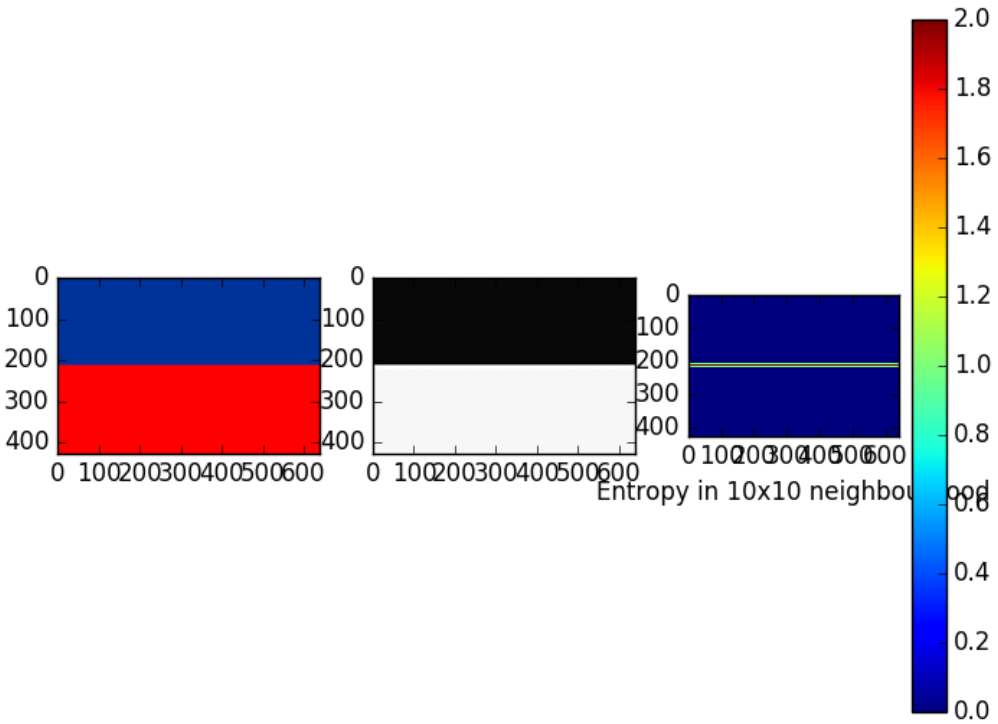

Figure 23: Entropy of a blue and red image

3 Text Processing Algorithms

To conclude this paper, I will discuss text processing algorithms because it can be a nice addition to the purely graphical algorithms discussed so far. In text processing, text in an image is recognized and its meaning extracted. Eventually we could use this information for sentiment analysis.

3.1 Pre-processing

Text processing starts with recognizing the individual characters, this is called Optical Character Recognization (OCR). To increase the effectiveness of the OCR algorithms we can pre-process the images. Several techniques can be used.

Layout analysis Often called 'zoning', this technique identifies columns, paragraphs, sections, etc as distinct blocks.

Character isolation If you want to apply OCR to every character separately (per-character OCR), this technique breaks groups of multiple characters into single letters.

Binarisation Converts the image from color to grayscale. It is a simple way to separate the text from the background of the image. Most OCR algorithms work only on grayscale images since these images are easier to work with.

3.2 Character Recognition

3.2.1 Pattern Recognition

One of the first techniques developed is pattern recognition, sometimes called matrix matching. This technique involves comparing an image of text to a previously stored glyph[4] on a pixel-by-pixel basis. This only works if the glyph and the input text are in the same font, same scale and if the input text can be isolated from the rest of the image.

[4] A matrix representation of the symbol, intended to represent a readable character.

This is very similar to classification in supervised machine learning . We have some learning data (the stored glyph) and we use this to get the correct results from unknown data.

3.2.2 Feature Extraction

This techinque uses 'features' as glyphs, for example, lines, closed loops and line intersection. Using these features instead of the complete glyph reduces dimensionality and makes the recognition computationally more efficient.

An example of an algorithm used is the k-nearest neighbour classification algorithm or k-NN. It is used to compare image features with stored glyph features and chooses the nearest match. Labeled training data is needed, since it is a supervised learning algorithm. Important to notice is that the feature vectors are in a multidimensional feature space, each with a class label. Storing the vectors in a multidimensional space allows us to calculate the distances between vectors. This distance will be used to choose the best classification. A new instance (in this case a character to recognise) is classified by assigning the label of the feature vector which is most frequent among the k training samples nearest to the test sample [16].

The Euclidian distance is commonly used to determine the distance between two continous samples. If the variables are discrete, which is the case in text classification, the overlap metric or Hamming distance is used. This metric is the number of positions at which the symbols to compare are different. In figure 24 you can see an example of this metric. The distance between 100 and 011 is 3, which is the number of sides of the cube you have to follow to get from 100 to 011 [17].

In figure 25 you can see an example of the k-NN algorithm. The green circle is the new instance. To determine if it should be classified as a blue square or as a red triangle, we look at the number of occurences of the possible classifications in the circle. The solid line circle is the space created by the algorithm with $k = 3$. Now the green circle is seen as a red triangle, because there are two red triangles and only one blue square. If $k = 5$, the space created by k-NN is the dashed line circle. Now the algorithm would classify the green circle as a blue square because there are two red triangles and three blue squares in the circle.

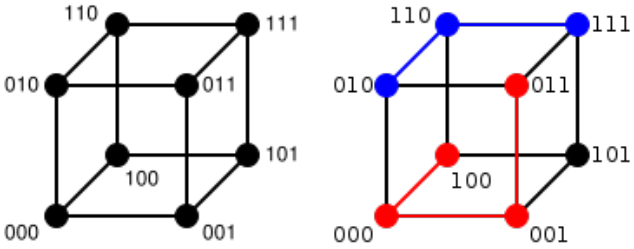

Figure 24: Example of the Hamming metric

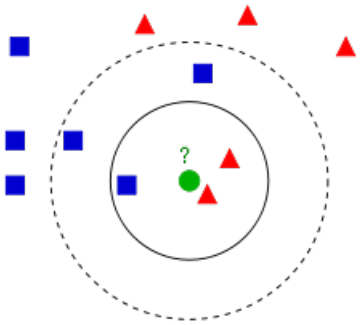

Figure 25: Example of the k-NN algorithm

4 Conclusion

Eventually the ArtToMusic application will extract useful data from all the algorithms discussed in this paper. In this section I will run several algorithms on several paintings, each in a completely different style. At the end we will have a collection of parameters we can use to generate music. I will use the painting of Picasso, figure 2, a painting of Bruegel, figure 30 and a painting of Mondriaan, figure 35.

These three paintings are very different. The Bruegel is very busy and intricate, so we expect results that represent this, for example a high entropy of the edge detection result. The Picasso is a face so it's not a busy painting, but with a wide range of colours we expect a high value for k in the image segmentation algorithm. The Mondriaan, figure 35 consists of big squares of solid colours which suggets a low amount of edges and a low entropy in general.

4.1 Picasso

4.1.1 Edge Detection

Let's look at the results of the edge detection algorithms for the Picasso. The resulting images can be found in section 2.1. We want to get as much information as possible from the result of the edge detection. When we look at the entropy of each resulting image, it is clear that the results of the algorithms are very close to each other. In table 26 you can see the entropies of the columns of each resulting image. In this case we will choose the algorithm with the highest entropy, the LoG algorithm. The resulting output image is figure 7.

	Sobel	**Frei Chen**	**Prewitt**	**Roberts Cross**	**LoG**	**Scharr**
Entropy	12.30425	12.30326	12.30373	12.32597	12.88781	12.30777

Figure 26: Entropies of edge detection algorithms on the Picasso

The output from this algorithm is a matrix representing the edges of the image. We need to get information about this matrix to create useful input parameters. For instance, we can count the number of distinct values

in the matrix. This gives us an idea of how many edges there are. When we round the pixel values of the output of the edge detection algorithm to three decimals we get a good value. Less than three decimals doesn't show the difference between the paintings and more than three decimals returns a number that almost represents the whole matrix. For the LoG algorithm this gives us 811 unique values. To translate this into an input parameter we can create a scale, ranging from 500 to 1500[5] and see where the calculated number is situated on this scale. A low number of edges can result in a slow piece of music and a high number of edges can result in a fast piece of music.

4.1.2 Image Segmentation

Using the technique of image segmentation we can get the amount of colors in the image. Analogous to the results of the edge detection we can create a scale that represents the possible values of k. A low k (e.g. 1 or 2) will result in a piece of music with very few notes, while a high k (e.g. 20 and upwards) will result in a busy piece of music. If we run this algorithm on the same Picasso of figure 2 we get the following result. When we use $k = 15$ we get a satisfying result. Figure 27 shows the segmented Picasso painting and figure 28 shows the colors in the RGB color space. The segmented image is very close to the original image cause we were able to find sufficient colours to reconstruct the image.

4.1.3 Colours

The image segmentation not only returns the number of colours (clusters) in the painting, it also specifies what these colours are. In the table below you can see the 15 colours extracted from the Picasso painting, shown in figure 28. You can see the RGB value of each colour, in 255 format and in percentage, as well as the HSV value. The hexadecimal representation is also included.

[5]This is an arbitrary chosen scale.

Figure 27: The Picasso segmented in colours

Hex	**R**	**G**	**B**	**H**	**S**	**V**
#0D0D0A	13 (5%)	13 (5%)	10 (4%)	60°	23%	5%
#695757	105 (41%)	87 (34%)	87 (34%)	0°	17%	41%
#B5B3A6	181 (71%)	179 (70%)	166 (65%)	52°	8%	71%
#D6D4C9	214 (84%)	212 (83%)	201 (79%)	52°	6%	84%
#C7C7B8	199 (78%)	199 (78%)	184 (72%)	60°	8%	78%
#E6E6DB	230 (90%)	230 (90%)	219 (86%)	60°	5%	90%
#7D756E	125 (49%)	117 (46%)	110 (43%)	28°	12%	49%
#363029	54 (21%)	48 (19%)	41 (16%)	32°	24%	21%
#AD802B	173 (68%)	128 (50%)	43 (17%)	39°	75%	68%
#696129	105 (41%)	97 (38%)	41 (16%)	52°	61%	41%
#471A1F	71 (28%)	26 (10%)	31 (12%)	354°	63%	28%
#216682	33 (13%)	102 (40%)	130 (51%)	198°	75%	51%
#AB998A	171 (67%)	153 (60%)	138 (54%)	28°	19%	67%
#1F1F1A	31 (12%)	31 (12%)	26 (10%)	60°	16%	12%
#4A423D	74 (29%)	66 (26%)	61 (24%)	24°	18%	29%

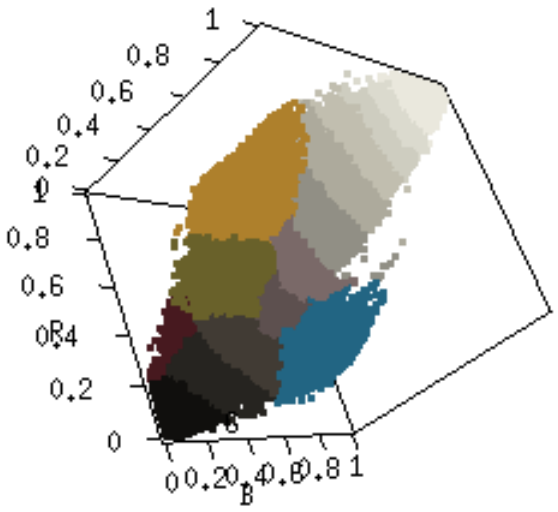

Figure 28: Colours of the Picasso

4.1.4 Noise Removal

Let's look at the results of the noise removal agorithms for the Picasso. As discussed in section 2.3 we have four algorithms. In order for these algorithms to work we have to convert the images to gray scale images. To use these algorithms in a meaningfull way, we can apply one of them on the image before the edge detection step to improve the results of the edge detection algorithm.

In figure 29 you can see the results. The median filter gives us the clearest result, as expected cause this is the most statistical robust filter of the four discussed here. Since the Gaussian filter also smooths the image, we won't be using this to detect the edges, even though the noise is removed. The median and the rank filter are not adequate enough to compete with the other filters.

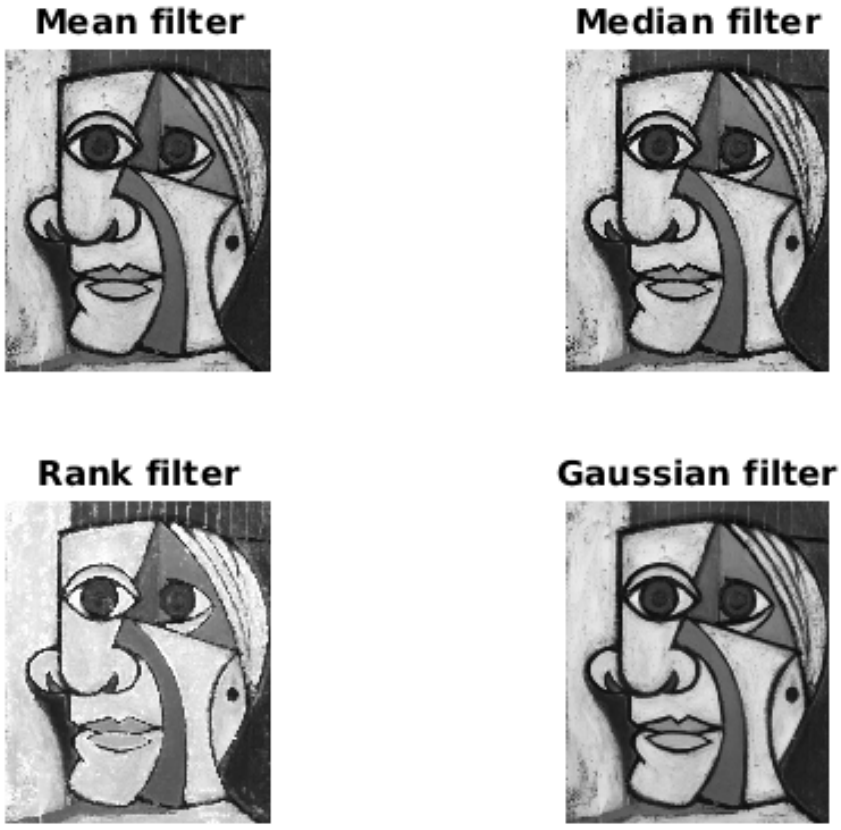

Figure 29: Results of noise removal algorithms on the Picasso

4.2 Bruegel

As explained above, we use a painting of Bruegel, figure 30, because we expect results from the algorithms that indicate a busy, high tempo piece of music.

4.2.1 Edge Detection

Analogous to the Picasso, I applied the six edge detection algorithms to the Bruegel painting. In table 31 you can see the results. As with the Picasso, the LoG algorithm has the highest entropy. We find 848 unique values, so this would give a similar tempo as the Picasso.

Figure 30: The Bruegel

	Sobel	**Frei Chen**	**Prewitt**	**Roberts Cross**	**LoG**	**Scharr**
Entropy	12.56623	12.56724	12.57051	12.59074	13.01124	12.5687

Figure 31: Entropies of edge detection algorithms on the Bruegel

4.2.2 Image Segmentation

When we apply the image segmentation algorithm to the painting of Bruegel, we can see that it takes $k = 20$ clusters to classify enough colours. This is a big number, more than we needed with the Picasso, so we expect a busy piece of music to be generated based on this parameter. In figure 32 you can see the segmented version of this painting. Since our k is so high, there is not so much difference between the original painting and this version. In figure 33 you can see the colours extracted.

Figure 32: The Bruegel segmented in colours

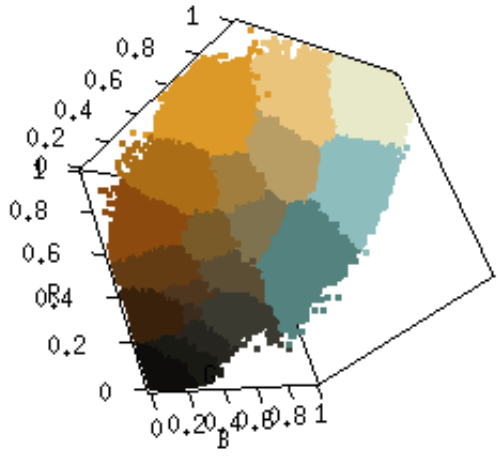

Figure 33: Colours of the Bruegel

4.2.3 Colours

Analogous to the Picasso painting, below are the 20 colours of the Bruegel painting.

Hex	R	G	B	H	S	V
#B17817	177 (69%)	120 (44%)	23 (9%)	38°	87%	69%
#E3AD57	227 (89%)	173 (68%)	87 (34%)	37°	62%	89%
#B3A173	179 (70%)	161 (63%)	115 (45%)	44°	36%	70%
#A88542	168 (66%)	133 (52%)	66 (26%)	40°	61%	66%
#695733	105 (41%)	87 (34%)	51 (20%)	40°	51%	41%
#47736B	71 (28%)	115 (45%)	107 (42%)	170°	38%	45%
#DB961C	219 (86%)	150 (59%)	28 (11%)	38°	87%	86%
#4D402E	77 (30%)	64 (25%)	46 (18%)	36°	40%	30%
#241F1A	36 (14%)	31 (12%)	26 (10%)	30°	28%	14%
#4A2E14	74 (29%)	46 (18%)	20 (8%)	29°	73%	29%
#12120F	18 (7%)	18 (7%)	15 (6%)	60°	17%	7%
#8C4D12	140 (55%)	77 (30%)	18 (7%)	30°	87%	55%
#F7F7C7	247 (95%)	247 (97%)	199 (78%)	60°	19%	97%
#362E24	54 (21%)	46 (18%)	36 (14%)	34°	33%	21%
#6BA1A1	107 (42%)	161 (63%)	161 (63%)	180°	34%	63%
#876930	135 (53%)	105 (41%)	48 (19%)	40°	64%	53%
#A6D1CF	166 (65%)	209 (82%)	207 (81%)	72°	100%	82%
#EDCC85	237 (93%)	204 (80%)	133 (52%)	42°	44%	93%
#827857	130 (51%)	120 (47%)	87 (34%)	46°	33%	51%
#664017	102 (40%)	64 (25%)	23 (9%)	32°	77%	40%

4.2.4 Noise Removal

When we apply the four noise removal algorithms to the Bruegel we get the result in figure 34. We can draw the same conclusion as with the Picasso. The median filter the best to work with. Gaussian filter is a close second, but the smoothing of the image causes us the disregard it, thinking of the edge detection algorithms.

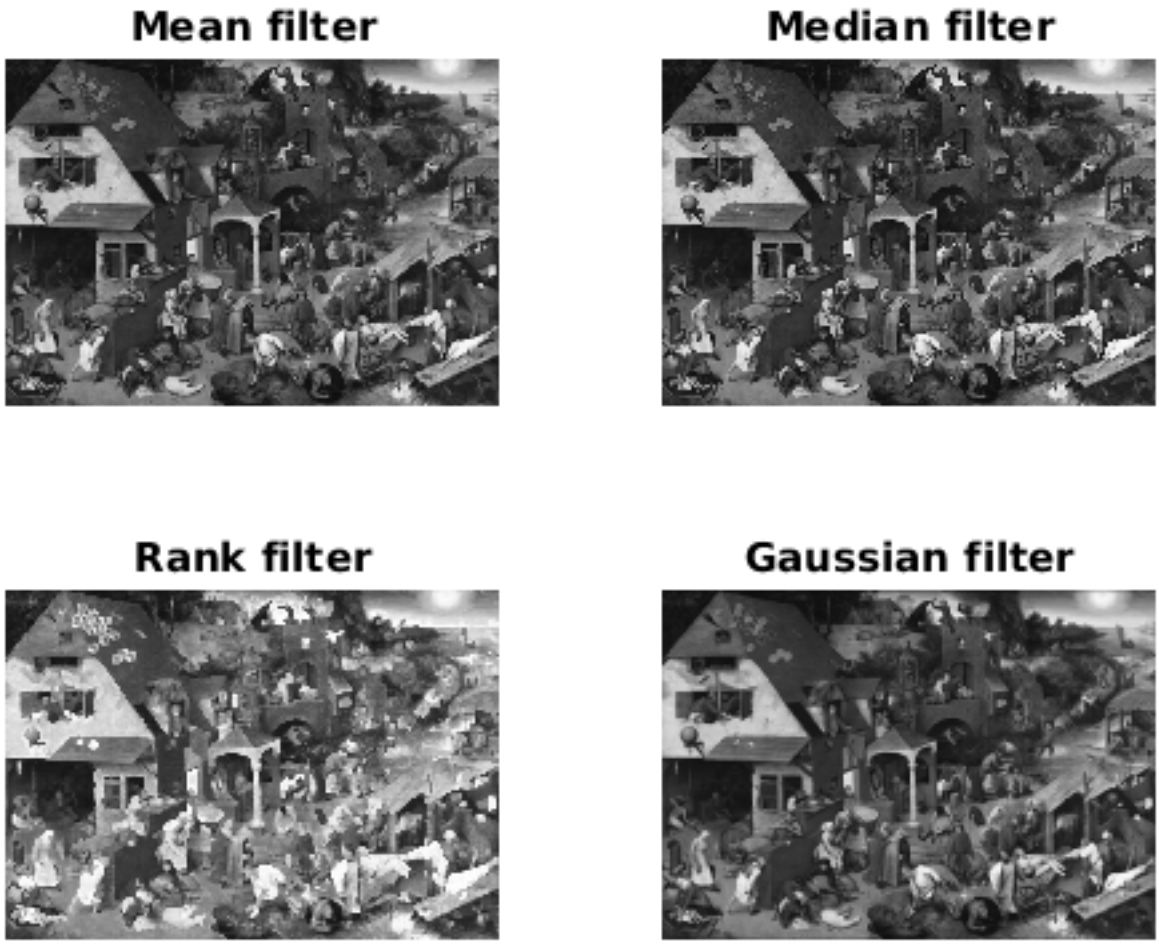

Figure 34: Results of noise removal algorithms on the Bruegel

4.3 Mondriaan

To conlude the comparison, a painting of Mondriaan is used, figure 35. Mondriaan's paintings were famous for its straight lines and full colours. We expect results that indicate a piece of music with little notes and not so difficult or busy rhythms.

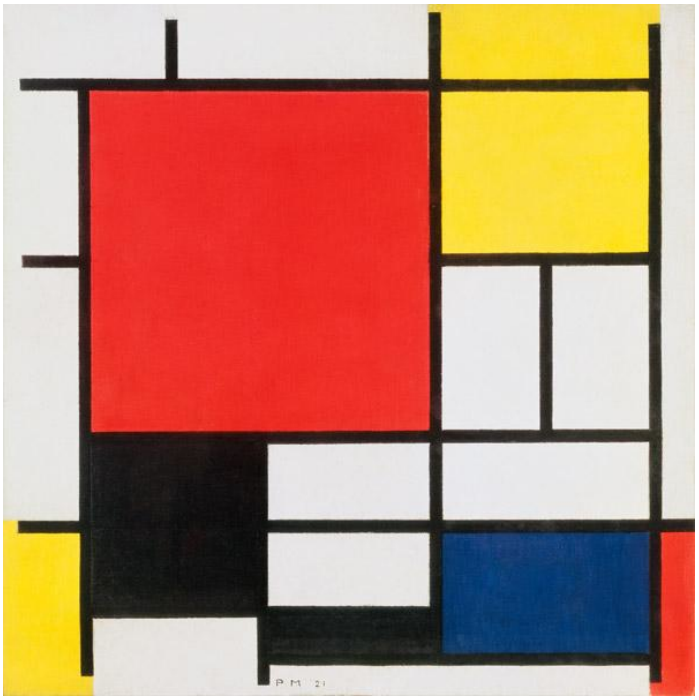

Figure 35: The Mondriaan

4.3.1 Edge Detection

Analogous to the Picasso and the Bruegel, the six edge detection algorithms were applied to the Mondriaan painting. In table 36 you can see the results. As with the Picasso and the Bruegel, the LoG algorithm has the highest entropy. We find 806 unique values, so this would again give a similar tempo as the Picasso and the Bruegel. This result is unexpected because with the naked eye we see very little edges on the Mondriaan painting and would expect a lower number of unique values than the two previous paintings. The overall entropy is lower in this painting, which was to be expected, but

the difference between the entropies of the Mondriaan and the Picasso is less than expected.

	Sobel	**Frei Chen**	**Prewitt**	**Roberts Cross**	**LoG**	**Scharr**
Entropy	11.0401	11.03431	11.03016	11.02275	12.77017	11.05094

Figure 36: Entropies of edge detection algorithms on the Mondriaan

4.3.2 Image Segmentation

When we apply the image segmentation algorithm to the painting of Mondriaan, we can see that it takes $k = 6$ clusters. Since it is a small number, our expectations of a simple piece of music are confirmed (for now). In figure 37 you can see the segmented version of this painting. Even though we have a low k, we don't see any difference between the original and the segmented version. This is because the original was already very segmented, all the colours are separated from each other by black lines. This makes it easier to segment it and results in less colours. In figure 38 you can see the colours extracted.

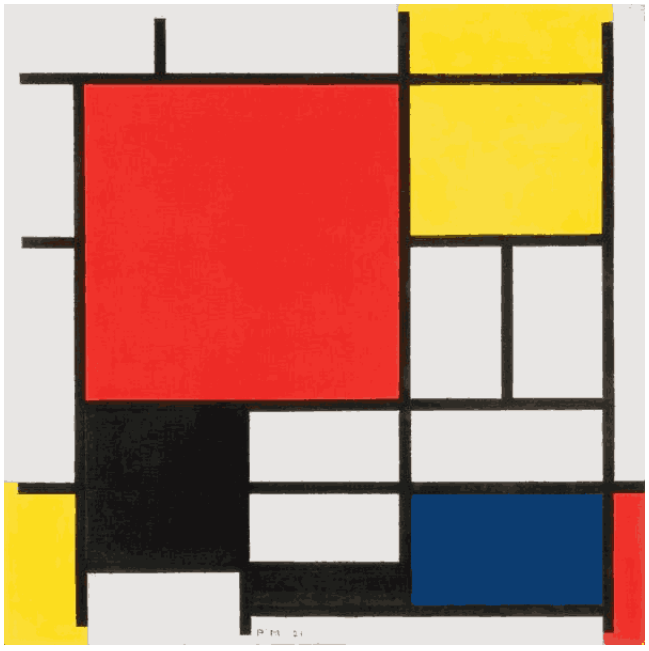

Figure 37: The Mondriaan segmented in colours

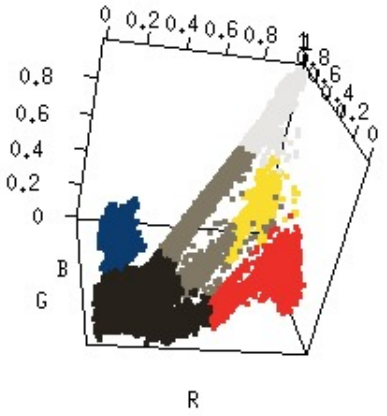

Figure 38: Colours of the Mondriaan

4.3.3 Colours

Below are the colours of the Mondriaan painting.

Hex	R	G	B	H	S	V
#FCDE30	252 (99%)	222 (87%)	48 (19%)	52°	81%	99%
#E6E6E3	230 (90%)	230 (90%)	227 (89%)	60°	1%	90%
#26211C	38 (15%)	38 13(%)	28 (11%)	30°	26%	15%
#ED2E29	237 (93%)	46 (18%)	41 (16%)	2°	83%	93%
#807863	128 (50%)	120 (47%)	99 (39%)	44°	23%	50%
#143B70	20 (8%)	59 (23%)	112 (44%)	216°	82%	44%

4.3.4 Noise Removal

When we apply the four noise removal algorithms to the Mondriaan we get the result in figure 39. Since there are a lot of squares of full colour and not so many edges in this image, the results of the noise removal algorithms are very similar. To continue the trend we again choose the median filter to work with.

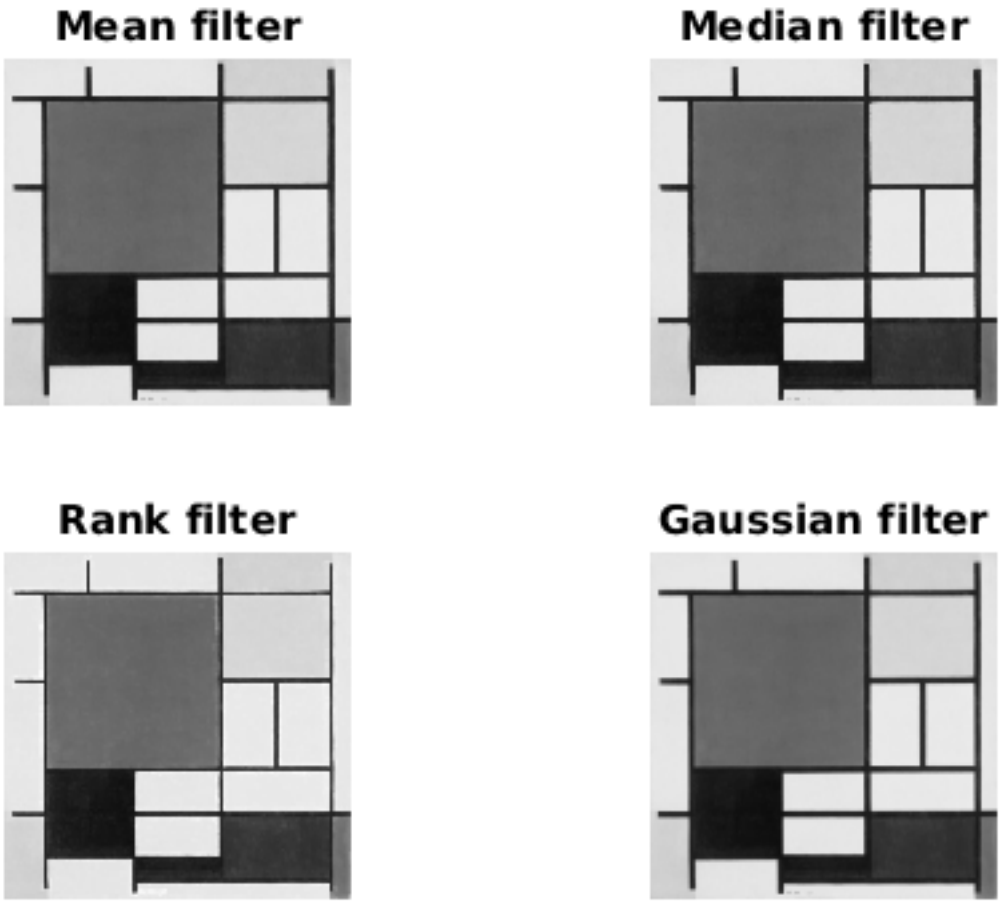

Figure 39: Results of noise removal algorithms on the Mondriaan

References

[1] Lampros Mouselimis *OpenPackageR, An Image Processing Toolkit* 2017

[2] R. Fisher, S. Perkins, A. Walker, E. Wolfart *Hypermedia Image Processing Reference* 2003

[3] Daniel Rákos' blog (`http://rastergrid.com/blog/2011/01/frei-chen-edge-detector/`) 2011

[4] University of Auckland, New Zealand, Computer Science course Computer Graphics and Image Processing *Gaussian Filtering* 2010

[5] Computação Visual e Multimédia, course on Image Processing

[6] RoboRealm blog (`http://www.roborealm.com/help/Prewitt.php`) 2017

[7] Jogn Costella's blog (`http://johncostella.com/edgedetect/`) 2012

[8] Simon Colton, Pedro Torres *Evolving Approximate Image Filters* 2009, Computational Creativity Group Department of Computing, Imperial College London

[9] Dr. Neal Krawetz' blog (`http://www.hackerfactor.com/blog/?/archives/529-Kind-of-Like-That.html`) 2013

[10] John Loomis' site (`http://www.johnloomis.org/ece563/notes/basics/entropy/entropy.html`) 1998

[11] Kevin Meurer's blog (`http://kevinmeurer.com/a-simple-guide-to-entropy-based-discretization/`) 2015

[12] Python Multimedia Codec Exercices Documentation (`https://www.hdm-stuttgart.de/~maucher/Python/MMCodecs/html/basicFunctions.html`) 2013

[13] Ryan Walker's blogpost (`https://www.r-bloggers.com/color-quantization-in-r/`) 2016

[14] C. Solomon, T. Breckon *Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab* 2011

[15] https://en.wikipedia.org/wiki/Optical_character_recognition

[16] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[17] https://en.wikipedia.org/wiki/Hamming_distance