FRANKFURT
UNIVERSITY
OF APPLIED SCIENCES

IMPLEMENTATION OF AN EUCLIDEAN COLOR FILTER FOR LEARNING API

Ms. Deepali Ashok Betkar
deepali.betkar@stud.fra-uas.de

Abstract—**This paper introduces the implementation of a simple and efficient color filter algorithm for the Learning Api of the Frankfurt University of Applied Sciences.**
Any radius and color center can be transferred to the color filter. The algorithm then checks for each individual pixel of an image, whether the color value of the pixel is within this range. The Euclidean distance, i.e. the distance between the color value of the pixel and the specified color space, must be calculated. If the color value of the pixel lies within the specified color space, the pixel retains its color value. If the pixel is outside the color space, it is assigned the RGB value (0,0,0), which corresponds to the color black. After all pixels have been processed by the algorithm, you get a filtered image as output.
Since the filtering is based on the Euclidean distance, this project is about the implementation of a Euclidean Color Filter.

Keywords— Euclidean Color Filter, Implementation, Learning Api

I. INTRODUCTION

The project was carried out as part of the Software Engineering module of the Frankfurt University of Applied Sciences. The goal of this project was the implementation of a Euclidean Color Filter for the Learning API, which is continuously supplemented by project work from students. The Learning Api consists of different modules for machine learning and image processing. This project uses the interface "IPipeLineModule" of the Learning Api.

The function of a Euclidean Color Filter is to filter out a specific color spectrum from an image. Therefore, in the implementation of the filter, a radius and a color center (RGB-value) must be specified. Then the algorithm calculates for each pixel of the image the Euclidean distance to the specific color center. If the distance is within the specific radius, the pixel keeps its RGB-value, if the distance is outside the radius, the RGB-value of the pixel is changed to black (0,0,0). After all pixels have been processed, you get the filtered image as output [1]. The

calculation of the Euclidean distance is the most important point for the implementation of the filter

II. METHOD

This section deals with the methodological competencies which were necessary in advance to implement the algorithm. Since the algorithm is in principle simple, these were only a few things.

On the one hand, basics about the RGB color space had to be worked out, on the other hand theory about the Euclidean Distance had to be appropriated.

It also shows the basic Architecture of this Project

A. RGB color Space

The RGB color space describes a color by the additive mixed portions of the colors red, green, and blue.
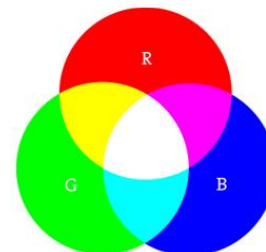

Figure 1: RGB Color Space

An RGB value is always represented 3 dimensional. The three additive colors can take a value from 0 to 100%. The portion of each color is digitally stored with a certain number of bits (color depth). By default, Visual studio works with 8 bits for each color channel. Thus, each color channel can represent 2^8 bit, i.e. an intensity from 0 to 255. There are two special color cases: if all color channels have the value 0, this results in the color black, but if all values are 255, the color white is obtained.

In total, with the three basic colors it is possible to use $2^8 * 2^8 * 2^8$ (= ~16.8 million) different color states [4].

B. Euclidean Distance

When implementing a Euclidean Color Filter, the Euclidean distance has to be calculated. The Euclidean distance generally describes the distance between two points and is described with formula (1).

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + \ldots + (q_n - p_n)^2} \qquad (1)$$

The dimension of the Euclidean space is called n.
Since an RGB value is 3-dimensional, formula (2) is used for the calculation in this implementation [2].

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + (q_3 - p_3)^2} \qquad (2)$$

According to formula (2), the Euclidean distance between the RGB color space of each individual pixel and an arbitrarily defined RGB value of the color center is calculated. For Example: Pixel-RGB-value = 101, 90, 58 and the Center-RGB-value = 220, 200, 51, inserted in (2) we get

$$d = \sqrt{(101 - 220)^2 + (90 - 200)^2 + (58 - 51)^2} = 162{,}2$$

There are three maximum cases where the distance is easy to determine. If both points have the values 0, 0, 0 (black) one gets the distance 0. If both points have the values 255,255,255 (white), the maximum possible distance of 441,67l is obtained. If one point has the value 255, 0, 0 (red) and the other point has the value 0, 255, 0 (green), then the length 360.62 is obtained.

C. Architecture

The algorithm was implemented in the .NET standard 2.0 framework. It is also based on the interface "IPipeLineModule" of the Learning Api, which has a 3 dimensional double array as input and output [3]. Since the input of a color filter is an image, it must be converted from a bitmap to a 3 dimensional array. The loading and the conversion of the image takes place in a UnitTest, because it was required like this. After the parameters for the radius and the color center have been specified, an image can finally be loaded. After the conversion into the 3-dimensional array, the algorithm is executed in the main class "EuclideanFilterModule". The main class has only one method, the "Run" method, which belongs to the interface. The "Run" method uses the "CalcDistance" class to calculate the Euclidean distance. There is also the class "GetAndSetPixels", which is needed to get the current color value of a pixel and to set a certain color value for the pixels.

To see the effect of the filter after executing the algorithm, the 3-dimensional double array have to be converted back into a bitmap and saved. These steps also take place in the UnitTest. The block diagram in figure 2 shows the architecture in a clear way.
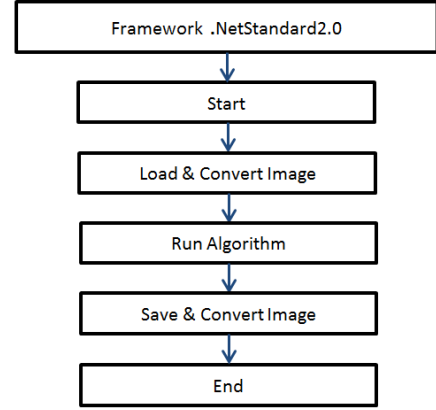

Figure 2: Architecture of the Algorithm

III. RESULTS

The Results section describes the implementation of the algorithm.

A. Load & Convert Image

Since the default for the interface "IPipeLineModule" provides an input and output as double, the image must be converted from the data type bitmap to double [„].
This is done in the "ConvertFromBitmapTo3dArray" method in the UnitTest class "EuclideanFilterModuleTester".

```csharp
public static double[,,] ConvertFromBitmapTo3dArray(Bitmap bitmap)
{
    int imgWidth = bitmap.Width;
    int imgHeight = bitmap.Height;

    //0 -> R, 1 -> G, 2 -> B
    double[,,] imageArray = new double[imgWidth, imgHeight , 3];

    for (int i = 0; i < imgWidth; i++)
    {
        for (int j = 0; j < imgHeight; j++)
        {
            Color color = bitmap.GetPixel(x: i, y: j);
            imageArray[i, j, 0] = color.R;
            imageArray[i, j, 1] = color.G;
            imageArray[i, j, 2] = color.B;
        }
    }

    return imageArray;
```

Figure 3: Code Snippet from ConvertFromBitmapTo3dArray Method

The input of this method is an arbitrary image (bitmap), which is passed by the Load method. First the image size of the image is determined. (e.g. 1024*768 pixel). From this a double[„] variable with the image size is created. In addition, each individual pixel has three RGB color values. After the loops have run through, there is information about the RGB color value for each pixel.

[0, 0, 0]	101
[0, 0, 1]	90
[0, 0, 2]	58
[0, 1, 0]	102
[0, 1, 1]	89
[0, 1, 2]	57
[0, 2, 0]	106
[0, 2, 1]	95
[0, 2, 2]	65
[0, 3, 0]	110
[0, 3, 1]	95
[0, 3, 2]	66
[0, 4, 0]	107
[0, 4, 1]	91
[0, 4, 2]	65

Figure 4 shows a screenshot during the debugging process and illustrates this process. After going through this method, the input was converted from a bitmap to a 3-dimensional double array. The imageArray data can then be passed to the algorithm. How exactly this happens is explained in detail in the chapter Unit Test.

Figure 4: Debugging of variable imageArray

B. Implementation of the EuclideanFilterModule – Main Method

The main method with the name "EuclideanFilterModule" executes the actual filtering process. It accesses two more small helper classes. The diagram in fig. 5 shows an overview of all used classes.
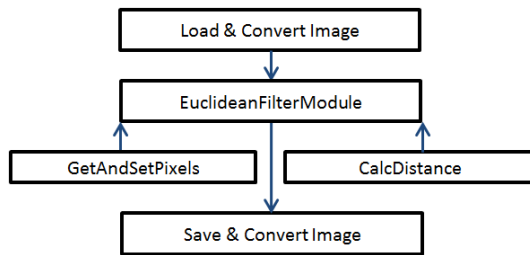

Figure 5: Overview of the classes.

The main method has a constructor to which two parameters are passed: "Color center" and "float radius". These two parameters must be specified before running the algorithm. For "center", the Argb value of the color center must be selected.

If, for example, one wants to filter a yellowish color portion, the values should be approximately Argb = 255, 220, 200, 51. The first value stands for alpha, which determines the transparency value. This value should always be 255, as this corresponds to full coverage. The following values determined the red-, green- and blue-value. To determine the color value of a pixel, different freeware can be used, e.g. ImageJ. An appropriate radius, on the other hand, can only be found by trying it out. If the radius is too large or too small, the image will not be filtered properly. In order to observe the change of the parameters in almost real time, there is another project with a Winforms application in addition to this algorithm. In the Winforms application it is possible to change all four parameters and directly observe the change in the image. This application uses the NuGet package of this algorithm.

As already described in the introduction, the algorithm was implemented with the interface IPipeLineModule of the Learning Api. The only method of this interface has the name double[,,] Run. Figure 6 illustrates this relationship.

```
public class EuclideanFilterModule : IPipelineModule<double[,,], double[,,]>
{
    public double[,,] Run(double[,,] data, IContext ctx)
    {
        if (data == null)
        {
            throw new ArgumentNullException();
        }
    }
```

Figure 6: Interface IPipeLineModule and Method Run

The parameters of the interface IPipeLineModule show the data type double[,,] as input (TIN) and output (TOUT). Accordingly, the method has as output double[,,] Run and as input double[,,,] data. The input "data" is passed by an expansion method from the method "ConvertFromBitmapTo3DArray" through the variable double[,,] imageArray. IContext is another interface that in turn can access other interfaces such as IDataDescriptor. However, these functions are not useful for the filter, so this field can be ignored.

The decision which rgb value the pixel adopts is made in the if/else statement from fig. 7.

```
float distance = CalcDistance.ComputeEuclideanDistance(color, Center);

if (distance <= Radius)
{
    GetAndSetPixels.SetPixel(result, row:i, col:j, color);
}
else
{
    GetAndSetPixels.SetPixel(result, row:i, col:j, Color.Black);
}
```

Figure 7: if-else statement for filtering

At this point in the code it is decided whether the color value of the pixel is preserved or changed to black (rgb-value 0,0,0). Both helper classes are required for the decision.

C. CalcDistance class

The helper class GetAndSetPixels consists of the methods GetPixel and SetPixel.
The method GetPixel is needed to get the color value of the current pixel.

```
public static float ComputeEuclideanDistance(Color pixelColor, Color center)
{
    float r2 = (float) Math.Pow(pixelColor.R - center.R, 2);
    float g2 = (float) Math.Pow(pixelColor.G - center.G, 2);
    float b2 = (float) Math.Pow(pixelColor.B - center.B, 2);

    return (float) Math.Sqrt(r2 + g2 + b2);
}
```

Figure 8: Method to calculate the Euclidean distance

Figure 8 shows that the Euclidean distance is calculated according to formula (2) from the chapter "Method". The individual color values of the current pixel and the center

color values are subtracted. The order of the subtraction is irrelevant, since the values are squared. The maximum value that can be returned is 441.671. With the returned value of the distance, it is possible to check the else/if statement from figure 7.

D. GetAndSetPixels class

The helper class GetAndSetPixels consists of the methods GetPixel and SetPixel.
The method GetPixel is needed to get the color value of the current pixel.

```csharp
public static Color GetPixel(double[,] imageArray, int row, int col)
{
    double r = imageArray[row, col, 0];
    double g = imageArray[row, col, 1];
    double b = imageArray[row, col, 2];

    return Color.FromArgb(red: (int)r, green: (int)g, blue: (int)b);
}
```
Figure 9: Method "GetPixel" from GetAndSetPixel-class

Figure 9 shows the method GetPixel. For this method, the double[,,] array "data" and the current pixel value are passed. This makes it possible to extract the RGB colors for each pixel and then compare them with the center in the CalcDistance class.
The method SetPixel assigns an RGB value to the current pixel.

```csharp
public static void SetPixel(double[,] imageArray, int row, int col, Color color)
{
    imageArray[row, col, 0] = color.R;
    imageArray[row, col, 1] = color.G;
    imageArray[row, col, 2] = color.B;
}
```
Figure 10: Method "SetPixel" from GetAndSetPixel-class

Figure 10 shows the method SetPixel. If the distance is inside the radius, the color values of the current pixel are stored unchanged in double[,,] result. If the distance is outside the radius, the RGB value is set to 0 (black) and stored in double[,,] result. After this method has been performed for each pixel, the filtered image is stored as double[,,] in the variable result.

E. Convert Image & Save

As double[,,] it is not possible to view the filtered image. Therefore it is necessary, to convert the data of the 3 dimensional double array back to the data type bitmap. Figure 11 shows the method in which this is done.

```csharp
public static Bitmap ConvertFrom3dArrayToBitmap(double[,,] imageArray)
{
    int imgWidth = imageArray.GetLength(dimension: 0);
    int imgHeight = imageArray.GetLength(dimension: 1);

    //0 -> R, 1 -> G, 2 -> B
    Bitmap bitmap = new Bitmap(imgWidth, imgHeight);

    for (int i = 0; i < imgWidth; i++)
    {
        for (int j = 0; j < imgHeight; j++)
        {
            Color color = GetAndSetPixels.GetPixel(imageArray, row: i, col: j);
            bitmap.SetPixel(x: i, y: j, color);
        }
    }

    return bitmap;
}
```
Figure 11: Method "SetPixel" from GetAndSetPixel-class

The input for this method are the filtered color information of all individual pixels from double[,,] result of the EuclideanFilterModule.
The RGB values are stored for each pixel with the "GetPixel" method in the variable color. It is then stored in the variable bitmap with (bitmap).SetPixel. The bitmap.SetPixel method has the same function as the SetPixel method from the GetAndSetPixel class.

F. UnitTests "EuclideanFilterModuleTester"

The entire algorithm with loading and saving an image is executed with the UnitTest "Run".
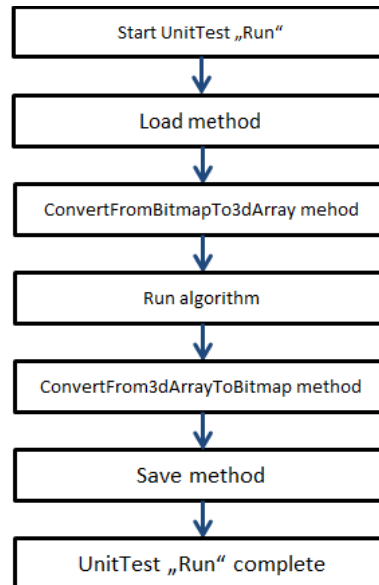Figure 12 shows the exact procedure of this UnitTest.

Figure 12: Architecture of the main UnitTest

All steps of figure 12 are already described in more detail in the points 3A to 3E.
The algorithm has also implemented an extension method. This adds the implemented Color Filter to the Learning Api. This makes it possible to execute the algorithm directly from the Learning Api. Figure 13 shows how this step is implemented.

```
[TestMethod]
public void RunApi()
{
    LearningApi api = new LearningApi();
    EuclideanFilterModule module = new EuclideanFilterModule(Color.FromArgb( a

    api.UseActionModule<double[,,], double[,,]>( moduleFunction: (input, ctx) =>
    {
        string baseDirectory = AppDomain.CurrentDomain.BaseDirectory;
        string path = Path.Combine(baseDirectory, "TestPictures\\3.jpg");
        double[,,] data = Load(path);
        return data;
    });

    api.AddModule(module);

    double[,,] output = api.Run() as double[,,];
```
Figure 13: Code Snippet from "Run"-UnitTest

In the method api.UseActionModule an image is loaded and converted with the method ConvertFromBitmapTo3dArray. The converted double[,,] is returned as variable "data".
With "api.AddModule(module)" the Color Filter is added to the Learning Api.
The code "api.Run() as double[,,]" passes the variable "double[,,] data" into the "Run" method of the filter (see fig. 6) [3].
The filtered "data" is stored in the variable "double[,,] output". Then a bitmap is created with the method "ConvertFrom3dArrayInBitmap" (see point E) and saved afterwards.
Figures 14 and 15 show an example run with this UnitTest.

Figure 14: Original image (before running the algorithm)

Figure 15: Filtered image (after running the algorithm)

In this example, it was tried to retain the yellowish color portion. The values used for the color center in this example are Argb = 255, 220, 200, 51. The radius was 150.0.

Besides the "Run" UnitTest, which executes the filter with a selected image, there is another UnitTest called "GenerateImageAndRunApi(). The difference to the previous UnitTest is that it creates 2x2 pixel images with known color values.

```
[TestMethod]
public void GenerateImageAndRunApi()
{
    LearningApi api = new LearningApi();
    EuclideanFilterModule module = new EuclideanFilterModule(Color.FromArgb(255, 255, 0, 0), 1.0f);
    Bitmap bitmap = new Bitmap(2, 2);

    bitmap.SetPixel(0, 0, Color.FromArgb(255, 0, 0));
    bitmap.SetPixel(0, 1, Color.FromArgb(255, 0, 0));
    bitmap.SetPixel(1, 0, Color.FromArgb(0, 255, 0));
    bitmap.SetPixel(1, 1, Color.FromArgb(0, 0, 255));

    double[,,] data = ConvertFromBitmapTo3dArray(bitmap);

    api.UseActionModule<double[,,], double[,,]>((input, ctx) => { return data; });
```
Figure 16: Code Snippet from "GenerateImageAndRunApi()"-UnitTest

In figure 16 one can see that a new bitmap with imagewidth and imageheight = 2 is created. The created instance of EuclideanFilterModule has as color center only the color red and the radius is 1. Then the pixels are set. The first two pixels also have the color red. Then the algorithm is executed as described in the UnitTest "Run".

```
Assert.AreEqual(Color.FromArgb(255, 0, 0), outputAsBitmap.GetPixel(0, 0));
Assert.AreEqual(Color.FromArgb(255, 0, 0), outputAsBitmap.GetPixel(0, 1));
Assert.AreEqual(Color.FromArgb(0, 0, 0), outputAsBitmap.GetPixel(1, 0));
Assert.AreEqual(Color.FromArgb(0, 0, 0), outputAsBitmap.GetPixel(1, 1));
```
Figure 17: Code Snippet from "GenerateImageAndRunApi()"-UnitTest

Figure 17 shows the asserts of this UnitTest. It is checked whether the color red of the first two pixels has been preserved and whether the last two pixels have been set to black. When calculating the Euclidean distance, the distance for the color red is 0, which is within radius 1. Accordingly, this color is preserved, while the other two pixels have a distance of 255 and thus become black.

G. UnitTests "CalcDistanceTests"

There are three other unit tests that check the correct functioning of the Euclidean distance calculation.

```
[TestMethod]
public void CalcDistance_Color1IsBlackColor2IsWhite_ReturnsMaxValue()
{
    Color color1 = Color.Black;
    Color color2 = Color.White;

    float actual = CalcDistance.ComputeEuclideanDistance(color1, color2);
    Assert.AreEqual(441.67295593006370984949881708f, actual, 0.01f);
}
```
Figure 18: UnitTest "Color1IsBlackColor2IsWhite_ReturnsMaxValue()"

Figure 18 shows the first test method. Two colors are generated there. One is white (255,255,255) and the other is

black (0,0,0). The Euclidean distance is calculated as follows:

$$d = \sqrt{(0-255)^2 + (0-255)^2 + (0-255)^2} = 441,67$$

The assert then compares whether the value matches.

```
[TestMethod]
public void CalcDistance_Color1IsBlackColor2IsBlack_ReturnZero()
{
    Color color1 = Color.Black;
    Color color2 = Color.Black;

    float actual = CalcDistance.ComputeEuclideanDistance(color1, color2);
    Assert.AreEqual(0f, actual, 0.01f);
}
```

Figure 19: UnitTest "Color1IsBlackColor2IsBlack_ReturnZero()"

Figure 19 shows the second test method. The principle remains unchanged. Now both colors are white (0,0,0). Accordingly, the euclidean distance should be

$$d = \sqrt{(0-0)^2 + (0-0)^2 + (0-0)^2} = 0$$

```
[TestMethod]
public void CalcDistance_Color1IsBlueColor2IsRed_ReturnsMaxValue()
{
    Color color1 = Color.Red;
    Color color2 = Color.Blue;

    float actual = CalcDistance.ComputeEuclideanDistance(color1, color2);
    Assert.AreEqual(360.6244584f, actual, 0.01f);
}
```

Figure 20: UnitTest "Color1IsBlueColor2IsRed_ReturnsMaxValue()"

Figure 19 shows the last UnitTest of the calculation of the Euclidean distance. In this one color is red and one color is blue. Accordingly, the value for this calculation should be

$$d = \sqrt{(255-0)^2 + (0-0)^2 + (255-0)^2} = 360,62$$

H. UnitTests "EuclideanColorTests"

There are four unit tests that check the correct functioning of the Euclidean distance calculation by using different formulas and logic regarding Euclidean Color Distance Formula as a base.

```
[TestMethod]
⊘ | 0 references
public void Test_EuclideanFormula()
{
    var p1 = new Point(5, 4);
    var p2 = new Point(4, 1);

    Assert.AreEqual(3.1622776601683795, Distance.EuclideanFormula(p1, p2));
}
```

Figure 21: UnitTest "Test_EuclideanFormula()"

Figure 21 shows the UnitTest of the calculation of the Euclidean formula with two different points i.e. pixel from image. In this two random pixel points from the image are selected to check the Euclidean formula. Accordingly, the value for this calculation should be

$$Euclidean = \sqrt{((5-4)^2 + (4-1)^2)} = 3.1622776601683$$

```
[TestMethod]
⊘ | 0 references
public void Test_EuclideanSimilarity()
{
    var p1 = new Point(5, 4);
    var p2 = new Point(4, 1);

    Assert.AreEqual(0.2402530733520421, Distance.EuclideanSimilarity(p1, p2));
}
```

Figure 22: UnitTest "Test_EuclideanSimilarity()"

Figure 22 shows the UnitTest for checking similarity between two selected point or pixel from image. The Euclidean similarity can be check with its formula. Accordingly, the euclidean similarity should be

$$EuclideanSimilarity = (1/(1 + EuclideanDistance(P1, P2))$$
$$= 1/(1 + 3.1622\ldots) = 0.2402530733520$$

```
[TestMethod]
⊘ | 0 references
public void Test_ColorMatching()
{
    var current = new Color();
    var match = new Color();

    Assert.AreEqual(0.0f, Distance.ColorMatching(current, match));
}
```

Figure 23: UnitTest "Test_ColorMatching()"

Figure 23 shows the UnitTest for checking the color matching between current selected color and match standard color. The color matching can be check with Euclidean distance formula.

$$d = \sqrt{(255-255)^2 + (0-0)^2 + (0-0)^2} = 0$$

For example, if two selected colors are same i.e. match color and current color both are Red, then it will give result as 0.

```
[TestMethod]
⊘ | 0 references
public void Test_FindNearestColor()
{
    var map = new Color[1];
    var current = new Color();

    Assert.AreEqual(0.0f, Distance.FindNearestColor(map, current));
}
```

Figure 24: UnitTest "Test_FindNearestColor()"

Figure 24 shows the UnitTest to find the nearest color to the current color from various match color. The color finding can be also check with the same principle as Euclidean distance as,

$$d = \sqrt{(0-0)^2 + (0-0)^2 + (255-255)^2} = 0$$

For example, if selected color is same Blue, the nearest color for the Blue is another blue which having minimum euclidean distance from selected current color.

IV. DISCUSSION

The goal of the project was the implementation of a Euclidean Color Filter algorithm for the Learning Api. Almost every image processing program offers a Euclidean Color filter. Most of them were realized with Python. There are only a few libraries like aforgenet.com that offer a solution for NetStandard. With this algorithm a new possible library is usable. Especially in connection with the Learning API, which already supports further image processing features, it is quite possible that this algorithm will be used by users.

The goal of a Euclidean Color Filter is to create a certain color range. It is also possible to remove unwanted areas of an image. Based on that, one can say, that the implementation was very successful.

The five-unit tests ensured that the algorithm worked reliably and quickly. The UnitTests "Run" checked the full scope of the code. After loading and converting the image into a 3-dimensional double array, the algorithm is applied to it. The output is then converted back into a bitmap and saved.

In summary, it can be said that the project was carried out successfully according to the specifications.

The next step for the future is to extend the filter to include moving images such as gif or webm files. Also, it would be great to see some more image processing filters within the Learning Api, so that library will become more powerful and useful.

V. REFERENCES

[1] Aorgenet.com. (2006-2013). Retrieved on 22. 12 2018 at aforgenet.com:
http://www.aforgenet.com/framework/docs/html/67fa83b5-dede-8d3a-8d3b-b7a6b9859538.htmI.

[2] Claus, S. (2015). *massmatics.de*. Retrieved on 22. 12 2018 at https://www.massmatics.de/merkzettel/#!155:Abstand_zwischen_zwei_Punkten

[3] Dobric, D. *https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi*. Retrieved on 27. 12 2018 at https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi

[4] Reas, B. F. (2003). *processing.org*. Retrieved on 22. 12 2018 at https://processing.org/tutorials/color/