Ukrainian Catholic University

Faculty of Applied Sciences

Business Analytics & Computer Science Programmes

Image Processing

Linear Algebra final project report

Authors:

Veronika Romanko

Bohdan Darmits

15 May 2019



Abstract

The images we see on internet pages and the photos we take with our mobile phone they all are digital images. It is possible to represent them using matrices. We will introduce how linear algebra can be applied in image processing, in particular in image filtering.

* **Introduction**

Firstly, we tried to implement image filters. Secondly we were looking for arias where it can be used. Finally, we decided to implement contour detecting.

* **Problem setting**

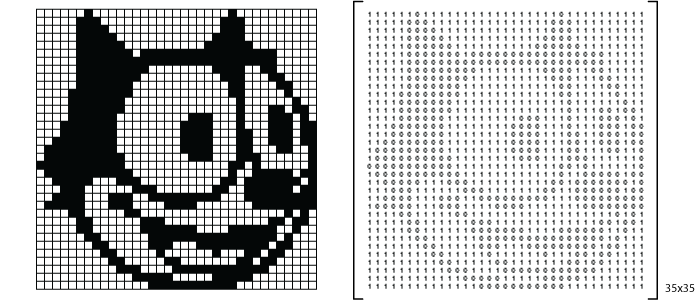
We decided not to use outside libraries. That is why it took a lot of time to get in touch with all the algorithms. All our project is written on pure java script. And all of algorithms have linear algebra on them.

**2.1 Image as a matrix**

If we zoom a lot any black and white picture, we will have something like that :



Image is a grid of small squares, called pixels. Each this piece can take only one color at a time. If assign 1 to the white color an 0 to black, the black & white picture will be represented as a matrix of 1 and 0 :



We can present the grayscale images by very similar way. But in this case the color has to be the number between 0 (black) and 255 (white) as the presentation of the intensity. For didactic purpose it is used a RGB color system. It means that each color specifies the variety of Red Green and Blue. Each color varies from 0 to 255. In RGB system, the pixel is presented as a tri-dimensional vector (r, g, b) where r, g and b are integer numbers from 0 to 255. Usually this tri-dimensional vector is stored as a single integer, using the following mapping function:

*v = f(r, g, b) = r\*65536 + g\*256 + b*

,where 65536 = + b.

Also it is possible to get the numerical value from the integer :

*r = v / 65536*

*g = (v % 65536) / 256*

*b = v % 256*

, where % is an operator to get the reminder of the integer division and / is referring to the integer division operator.

**2.2 Filtering images**

From the point of view of linear algebra, filters are applied to each pixel of the matrix using the filter function. The input of this function can be just a pixel like the adjustment of brightness, or a submatrix of pixels like the blur, where the order of the submatrix will depend on the blur ratio.

Let's consider the matrix M, as the matrix associated to a full color image:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M = |  | |  |  |  |  | | --- | --- | --- | --- | | *p11* | *p12* | ⋯ | *p1n* | | *p21* | *p22* | ⋯ | *p2n* | | ⋮ | ⋮ | ⋱ | ⋮ | | *pm1* | *pm2* | ⋯ | *pmn* | |  |

Here, pij is the pixel in the position (i, j), which is represented as the vector:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  | | --- | | *r* | | *g* | | *b* | |  |

In the simplest case, when the filter needs only a pixel as input, the function can be a linear transformation, that transforms a tridimensional vector in other words into another tridimensional vector, or not.

When it's a linear transformation, the transformation can be represented as a 3x3 matrix T, where:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | | --- | | *r'* | | *g'* | | *b'* | |  | = T • |  | |  | | --- | | *r* | | *g* | | *b* | |  |

**2.2.1 Sepia effect**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T = |  | |  |  |  | | --- | --- | --- | | *0.393* | *0.769* | *0.189* | | *0.349* | *0.686* | *0.168* | | *0.272* | *0.534* | *0.131* | |  |

The numbers inside the matrix T mentioned before, give the images, the reddish-brown color of early 20th century monochrome photographs.

Another common transformations are the one where the resulting pixel is obtained by adding a 3x1 matrix (tridimensional vector) to the original pixel:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | | --- | | *r'* | | *g'* | | *b'* | |  | = V + |  | |  | | --- | | *r* | | *g* | | *b* | |  |

**2.2.2 Grayscale conversion**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T = |  | |  |  |  | | --- | --- | --- | | *1/3* | *1/3* | *1/3* | | *1/3* | *1/3* | *1/3* | | *1/3* | *1/3* | *1/3* | |  |

The components of each new pixel is obtained by calculating the average of the three components.

There are not linear transformations in these examples. Examples of transformation are:

**Red channel adjustment**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V = |  | |  | | --- | | *f* | | *0* | | *0* | |  |

**Green channel adjustment**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V = |  | |  | | --- | | *0* | | *f* | | *0* | |  |

**Blue channel adjustment**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V = |  | |  | | --- | | *0* | | *0* | | *f* | |  |

**Brightness adjustment**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V = |  | |  | | --- | | *f* | | *f* | | *f* | |  |

f is a number that depends on the grade of adjustment the person wants to apply, and it can be a positive or negative number, usually ranging from -150 to 150.

Some other transformations can be obtained using a combination of the previous two transformations:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | | --- | | *r'* | | *g'* | | *b'* | |  | = T • |  | |  | | --- | | *r* | | *g* | | *b* | |  | + V |

**2.2.3 Negative**

This is the case of finding the **negative of an image** (color inversion), where each new pixel's component is obtained by subtracting the actual value from 255. The matrices are:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T = |  | |  |  |  | | --- | --- | --- | | *-1* | *0* | *0* | | *0* | *-1* | *0* | | *0* | *0* | *-1* | |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V = |  | |  | | --- | | *255* | | *255* | | *255* | |  |

**2.2.4 Contrast**

For the **contrast adjustment**, the operation is a little more complicated:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | | --- | | *r'* | | *g'* | | *b'* | |  | = V1 + T • |  | |  | | --- | | *r* | | *g* | | *b* | |  | + V2 |

Where

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V1 = |  | |  | | --- | | *-128* | | *-128* | | *-128* | |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T = |  | |  |  |  | | --- | --- | --- | | *f* | *0* | *0* | | *0* | *f* | *0* | | *0* | *0* | *f* | |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V2 = |  | |  | | --- | | *128* | | *128* | | *128* | |  |

In this case f is computed using the formula: f = (259 \* (value + 255)) / (255 \* (259 - value), where value is the grade of adjustment, usually ranging from -100 to 100.

**2.2.5 Gamma**

For **gamma correction**, we need more than adding and multiplying matrices, we need the exponentiation operator. The filter can be computed using the following formula:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | | --- | | *r'* | | *g'* | | *b'* | |  | = 255 • |  | |  | | --- | | *(r/255)1/f* | | *(g/255)1/f* | | *(b/255)1/f* | |  |

The factor f is a number ranging between 0 and 10, but without reaching the number 0.

The **blur**, as mentioned before, needs a submatrix of pixels as input, where the order of the submatrix will depend on the ratio of the blur. The primary idea is that the components of every output pixel are computed as the average of the corresponding component using 2.2.7 Finding contours

**3 Contour finding**

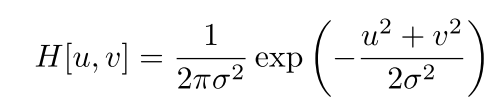
We have implemented contour finding. For that we used the Canny edge detection algorithm. It can be distributed in 5 parts:

1. Applying Gaussian filter to smooth the image in order to remove the noise
2. Finding the intensity gradients of the image
3. Applying non-maximum suppression to get rid of spurious response to edge detection
4. Applying double threshold to determine potential edges
5. Tracking edge by hysteresis.

all the pixels around it.

**3.1 Gaussian filtering**

We used Gaussian filter to get rid of the noise. It’s algorithm is applied with a Gaussian kernel. The kernel size depends on the expected blurring effect. The equation for a Gaussian filter kernel is given by :



That are the function of generating kernel and applying Gaussian blur :

**function** generateKernel(sigma, size) {  
 **var** kernel = [];  
 **var** EulerNumber = 2.718;  
 **for** (**var** y = -(size - 1) / 2, k = 0; k < size; y++, k++) {  
 kernel[k] = [];  
 **for** (**var** x = -(size - 1) / 2, m = 0; m < size; x++, m++) { //creates kernel round to 3 decimal places  
 kernel[k][m] = 1 / (2 \* Math.PI \* Math.pow(sigma, 2)) \* Math.pow(EulerNumber, -(Math.pow(Math.abs(x), 2) + Math.pow(Math.abs(y), 2)) / (2 \* Math.pow(sigma, 2)));  
 }  
 }  
  
 **var** normalize = 1 / SumArray(kernel); //normalize the kernel, makes its sum equals to 1  
  
 **for** (**var** i = 0; i < kernel.length; i++) {  
 **for** (**var** j = 0; j < kernel[i].length; j++) {  
 kernel[i][j] = Math.round(normalize \* kernel[i][j] \* 1000) / 1000;  
 }  
 }  
 **return** kernel;  
}  
  
**function** gaussianBlur(canvas, sigma, size) {  
 **var** context = canvas.getContext('2d');  
  
 **var** imageDataCopy = context.getImageData(0, canvas.height / 2, canvas.width, canvas.height / 2);  
 **var** kernel = generateKernel(sigma, size);  
  
 RunImage(canvas, size, **function** (current, neighbors) {  
 **var** r\_ = 0, g\_ = 0, b\_ = 0, pixel;  
 **for** (**var** i = 0; i < size; i++) {  
 **for** (**var** j = 0; j < size; j++) {  
 pixel = getPixel(neighbors[i][j], imageDataCopy); //we multiply the pixel valueue by the kernel  
 r\_ += pixel.r \* kernel[i][j];  
 g\_ += pixel.g \* kernel[i][j];  
 b\_ += pixel.b \* kernel[i][j];  
 }  
 }  
  
 setPixel(current, {r: r\_, g: g\_, b: b\_}, imageDataCopy);  
 });  
  
 context.putImageData(imageDataCopy, 0, 600);  
}

**3.2 Gradient**

We implemented gradient calculations to detect the edge intensity and direction by calculating the gradient of the image, by using edge detection operators. The edge detection operator returns a value for the first derivative in the horizontal direction (G*x*) and the vertical direction (G*y*). Below is how filters are applied to the image and how to get the intensity and edge direction matrices :

**function** roundInFourDirrections(degree) { //we round degree to vertical, horizontal, diagonals  
 **var** degree = degree < 0 ? degree + 180 : degree;  
 **if** ((degree >= 0 && degree <= 22.5) || (degree > 157.5 && degree <= 180)) {  
 **return** 0;  
 } **else if** (degree > 22.5 && degree <= 67.5) {  
 **return** 45;  
 } **else if** (degree > 67.5 && degree <= 112.5) {  
 **return** 90;  
 } **else if** (degree > 112.5 && degree <= 157.5) {  
 **return** 135;  
 }  
};

function gradient(canvas, op) {  
 var context = canvas.getContext('2d');  
 var imageData = context.getImageData(0, canvas.height / 2, canvas.width, canvas.height / 2);  
 var imageDataCopy = context.getImageData(0, canvas.height / 2, canvas.width, canvas.height / 2);  
 var dirrection\_map = [];  
 var gradient\_map = [];  
  
 var SOBEL\_X\_FILTER = [  
 [-1, 0, 1],  
 [-2, 0, 2],  
 [-1, 0, 1]  
 ];  
  
 var SOBEL\_Y\_FILTER = [  
 [1, 2, 1],  
 [0, 0, 0],  
 [-1, -2, -1]  
 ];  
  
 var ROBERTS\_X\_FILTER = [  
 [1, 0],  
 [0, -1]  
 ];  
  
 var ROBERTS\_Y\_FILTER = [  
 [0, 1],  
 [-1, 0]  
 ];  
  
 var PREWITT\_X\_FILTER = [  
 [-1, 0, 1],  
 [-1, 0, 1],  
 [-1, 0, 1]  
 ];  
  
 var PREWITT\_Y\_FILTER = [  
 [-1, -1, -1],  
 [0, 0, 0],  
 [1, 1, 1]  
 ];  
  
 var OPERATORS = {  
 'sobel': { x: SOBEL\_X\_FILTER, y: SOBEL\_Y\_FILTER, len: SOBEL\_X\_FILTER.length},  
 'roberts': { x: ROBERTS\_X\_FILTER, y: ROBERTS\_Y\_FILTER, len: ROBERTS\_Y\_FILTER.length},  
 'prewitt': { x: PREWITT\_X\_FILTER, y: PREWITT\_Y\_FILTER, len: PREWITT\_Y\_FILTER.length}  
 };  
  
 RunImage(canvas, 3, function (current, neighbors) {  
 var x\_ = 0, y\_ = 0;  
 var pixel = new Pixel(current, imageDataCopy.width, imageDataCopy.height);  
 if (!pixel.isBorder()) {  
 for (var i = 0; i < OPERATORS[op].len; i++) {  
 for (var j = 0; j < OPERATORS[op].len; j++) {  
 x\_ += imageData.data[neighbors[i][j]] \* OPERATORS[op]["x"][i][j];  
 y\_ += imageData.data[neighbors[i][j]] \* OPERATORS[op]["y"][i][j];  
 }  
 }  
 }  
 dirrection\_map[current] = roundInFourDirrections(Math.atan2(y\_, x\_) \* (180 / Math.PI));  
 gradient\_map[current] = Math.round(Math.sqrt(x\_ \* x\_ + y\_ \* y\_));  
 setPixel(current, gradient\_map[current], imageDataCopy);  
 });  
  
 context.putImageData(imageDataCopy, 0, 600);  
  
 return { dirrection\_map: dirrection\_map, gradient\_map: gradient\_map };  
}

**3.3 Non-maximum Suppression**

The final image should have thin edges. We have to perform non-maximum suppression. So, program goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions. Non-maximum-suppression algorithm is: create a matrix initialized to 0 of the same size of the original gradient intensity matrix, Identify the edge direction based on the angle value from the angle matrix, checking if the pixel in the same direction has a higher intensity than the pixel that is currently processed; returning the image processed with the non-max suppression algorithm. Below is an example of it:

function getPixelNeighbors(dir) {  
 var degrees = {  
 0: [{ x: 1, y: 2 }, { x: 1, y: 0 }],  
 45: [{ x: 0, y: 2 }, { x: 2, y: 0 }],  
 90: [{ x: 0, y: 1 }, { x: 2, y: 1 }],  
 135: [{ x: 0, y: 0 }, {x: 2, y: 2 }]  
 };  
 return degrees[dir];  
}  
  
function nonMaximumSuppress(canvas, dirMap, gradMap) {  
 var context = canvas.getContext('2d');  
 var imageDataCopy = context.getImageData(0, canvas.height / 2, canvas.width, canvas.height);  
 RunImage(canvas, 3, function(current, neighbors) {  
 var pixNeighbors = getPixelNeighbors(dirMap[current]);  
 var pix1 = gradMap[neighbors[pixNeighbors[0].x][pixNeighbors[0].y]];  
 var pix2 = gradMap[neighbors[pixNeighbors[1].x][pixNeighbors[1].y]];  
 if (pix1 > gradMap[current] ||  
 pix2 > gradMap[current] ||  
 (pix2 === gradMap[current] &&  
 pix1 < gradMap[current])) {  
 setPixel(current, 0, imageDataCopy);  
 }  
 });  
 context.putImageData(imageDataCopy, 0, 600);  
}

4 **Conclusions**

In this project, we tested different filtering methods. Our aim was to make a good website, where we can use image filters and find object contours. We used a pure java script. So there are no outside libraries used. All the functions use linear algebra in their implementation. We have made an interactive web page for testing filters and for finding contours of objects. We have ran this filters with different images. So, the result of each of them is showed below.



sepia



grayscale



negative



brightness



threshold



gradient



mirror



saturation



Gaussian



sharpen



laplacian



prewittHorizontal



prewittVertical



highpass



red



green



blue

4 References

1. http://blog.kleinproject.org/?p=588
2. archives.math.utk.edu/ICTCM/VOL22/S014/paper.pdf

[3] pdfs.semanticscholar.org/4f3f/1cce5541a6dfec75fe7afa844ecbbbd8bb0f.pdf