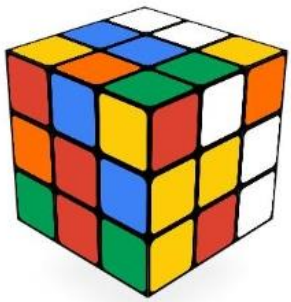
Rubik's Cube Detector

Author: Luca Banzato
Course: Image Data Mining
Professor: Piergiorgio Lanza
January 2017

Summary

1. Introduction

The main goal of this work is to recognize a Rubik's cube in space even if it's rotated or has some particular perspective. This program has been realized in **C++** and is called **RubiksCubeDetector** and uses several functions, described later, to achieve its goal.
Generally speaking this program tries to check if there is a Rubik's cube but if fails the user can change perspective in order to have a better result or, in some cases, a result.
As it is possible to notice from this image:



Rubik's cube "face" is made of nine squares, they can be nine of the same color or they can be different; however color is not considered in this approach. What is important to recognize are squares: if the algorithm finds a certain number of squares with, more or less, the same area, then the conclusion is that there might be a Rubik's cube in that image.
The biggest problem with this project was that for humans, to see a Rubik's cube is in a particolar position in an image is not a really big matter, even if the cube is seen from a different point of view; however the program does not know that and in order to achieve the goal mathematical notions regarding perspective are mandatory.
Another problem is **noise**: an image can contain a certain amount of noise which can interfere with the cube detection; this problem will be described later in a better way but for now it's important to know that it can be reduced using a Gaussian filter.
This technical note is divided in the following arguments:

2. Acronyms and abbreviations

During this technical note the following abbreviations and acronyms are used:

- **GF**: Gaussian filter
- **GLPF**: Gaussian low pass filter
- **R**: Range (dynamic)
- **RC**: Rubik's cube
- **OpenCV**: Open Source Computer Vision library

3. Bibliography

OpenCV documentation: http://docs.opencv.com

OpenCV question on RC: http://answers.opencv.org/question/25754/detect-and-analyze-a-rubik-cube/

Documentation on findContourns from OpenCV: http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/find_contours/find_contours.html

Documentation on Hough transform: http://stackoverflow.com/questions/3444634/search-for-lines-with-a-small-range-of-angles-in-opencv

Documentation on function *atan2*: http://www.cplusplus.com/reference/cmath/atan2/

Documentation on Gaussian smoothing image: http://nip-saga.blogspot.it/2015/04/digital-image-processing-with-c-Image-Smoothing-Gaussian-filter.html

Documentation on Fourier transform: http://docs.opencv.org/2.4/doc/tutorials/core/discrete_fourier_transform/discrete_fourier_transform.html

Documentation on mouse click: http://stackoverflow.com/questions/24586013/opencv-storing-the-coordinates-of-the-points-clicked-by-mouse-left-button

Documentation on mouse click and moves: http://opencv-srf.blogspot.it/2011/11/mouse-events.html

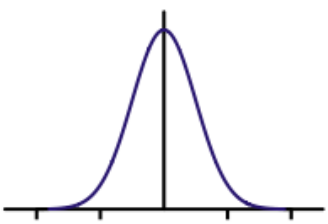
General documentation: https://it.wikipedia.org

4. Technical assumptions

This paragraph is dedicated to summarize all the formulas and mathematical concepts that were used for the project:

1) **Canny algorithm**: the Canny edge detector algorithm is an edge detector that uses a multi-stage algorithm to detect a wide range of edges in images. Among the edge detection methods developed so far, Canny algorithm is one of the most strictly defined methods that provides a good and reliable detection.

2) **Gaussian filter**: a Gaussian filter is a filter whose response can be described as a Gaussian function or an approximation of it.



Gaussian filter are applied during the Canny algorithm or to remove noise from an image. Mathematically speaking a Gaussian filter modifies the input signal by convolution with a Gaussian function and this transformation in also called **Weierstrass transform**.
3) **Log transformation**: the log transformation uses a logarithmic function in order to make the image more bright and the edges sharper.

4) **Homography**: the homography transformation is a popular technique used worldwide. It's based on quite complex geometric and mathematical concepts like **homogeneous coordinates** and **projective planes**. The real benefit of using the homography is the **homogeneous representation of straight lines**.

5) **Color**: during the work it wasn't given particulare care about colour at all although is knew that a RC have six different colours (one per face). This assumption semplified a lot the work, but the trade off was a lower precision in detecting the RC.

5. Algorithm description

In this paragraph is explained in a more detailed way the algorithm implementation and the techniques used during the work.

5.1. Algorithm implementation

The program takes an arbitrary number of images provided as arguments when is launched as input in which there might or might not be a RC. It is possible to scroll the pictures loaded by pressing any key of the keyboard, except for "ESC", which ends the program.
For each image, the function *findSquares*, representing the key point of the algorthm, is called. First of all it provides a way to remove noise from the current image using both a **GF** and a **GLPF** as follows:

```
GaussianBlur( image, timg, Size( 3, 3 ), 0, 0 );
Mat outMagt = lowpassFilter(timg);
```

In the project a **3x3** gaussian mask is used to reduce noise. After appliying the blurry effect to the image, a Lowpass Gaussian Filter is applied to enhance the smoothness. All the conversions are done in the method *lowpassFilter*, which converts the picture in the Frequency Domain (obtaining it's **spectrum**) for better performances and applies a Low-Pass Gaussian Filter; At the end the process is reversed, in order to obtain again an image in the Space Domain. The next step consists of separating the **RGB** layers in order to obtain three images, one for each layer. After that the **Canny edge detector** algorithm is applied: this reveals and shows edges in the image, of course the number of edges detected depends on the thresholding level and to avoid complications an adaptive thresholding was used.

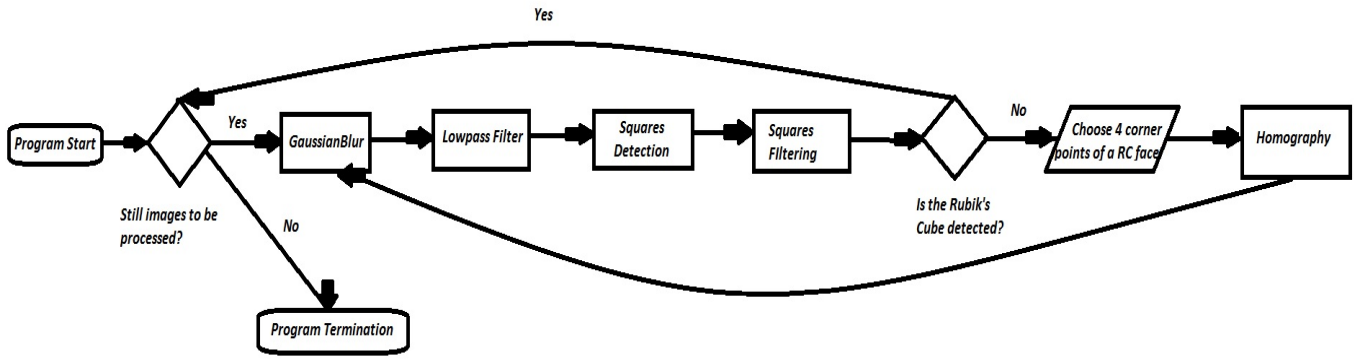
```
Canny(gray0, gray, 0, thresh, 5);
```

In order to improve the quality of the detected edges the function *dilate* is used, a morphological operator that unites two near segments within a certain range.

```
dilate(gray, gray, Mat(), Point(-1,-1));
```

The edges discovered through Canny are then stored in a vector. In the introduction it was explained that the final goal is to know if there's a RC in the current image and also that a RC face is made of nine little squares similar to each other.
Having said that it is possible to look after edges which are more or less "squares" or, in a better way, similar to squares; This task is accomplished through some mathematical formula applied to the previous edges: edges which have a cosine near 0 can be excluded due to the fact they're representing a very stretched square or in the worst scenario a line, which do not correspond to something which was looked for. Another control done is to check if all the lines which compose the square are nearly equal in length, in order to avoid rectangles.
The remaining edges are those with the shape of a parallelepiped and they are put into the structure called *MapArea*. This structure was realized to do the following task: starting with the set of edges previously described if there is a certain number of parallelepiped with similar area, then it is possible to assume that there might be a RC in the current image.
If a certain number of parallelepiped is not reached then the RC is considered as not detected and it may be not present in the image or there might be a perspective problem. In the latter case the help of the user is required in order to detect a cube. It is given the possibility to select four points by clicking them with the left button of the mouse in a specific order (Top-Left, Top-Right, Bottom-Left, Bottom-Right), corresponding to the vertex of a face of a RC. A **homography** function is applied to remove the perspective and after that the same procedure previously described is applied and a different window is opened, allowing the user to see the result.
In both of the cases the edges that should be part of the RC are drawn on the current image; also a string is printed (in terminal), saying if there is or not a RC.

Algorithm sketch



5.2. Algorithms used in the project

A) Canny Edge Detector

The Canny Algorithm is an Edge Detector acting following these steps:

1) it reveals edges along the X axis. This first order derivative is done through Gauss filter.
2) it reveals edges along the Y axis. This first order derivative is done through Gauss filter;
3) the gradient that unites the directional derivatives X and Y is normalized;
4) it uses a double thresholding; it takes a grayscale image and returns a binary image to improve quality avoiding false edges (false positives) if the thresholding level is too low; if the thresholding level is too high there's the presence of false edges known as false negatives.
5) thinning using interpolation to find the pixels where the norms of gradient are local maximum; during the thinning operator a connectivity analysis to detect and link edges is done.

An adaptive threshold is used in the program to get better results.

B) Log transformation

The log transformation can be expressed with the following formula:

$$s = c\ln(1 + r)\ for\ each\ r > 0$$

There's the addition of the constant **1** to the logarithm to avoid the natural logarithm of 0, which corresponds to color black. The **c** value is a constant that considers the **dynamic range** (**R**) of the image; it can be expressed by the following formula:

$$c = \frac{L - 1}{\ln R}$$

This technique is used mostly for dark images, to make them brighter.

C) Gaussian Filter

Regarding the Gaussian filter mentioned before, an image $g(x, y)$ can be modeled, considered, as follows:

$$g(x, y) = f(x, y) + \eta(x, y)$$

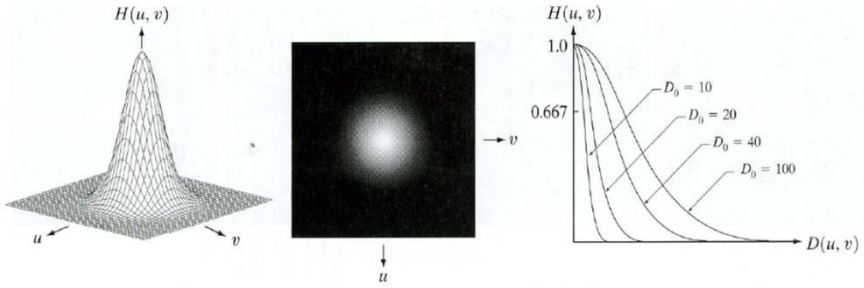
In this case $f(x, y)$ represents the image and $\eta(x, y)$ is the induced noise. The noise can be expressed through a Gaussian curve but it's distribution is always a random one; if the noise is added to the image sooner or later it will stabilize and the image will be more sharp. This technique is used for: low quality image, to improve their quality of course; image with illumination problems and space related images.

D) Gaussian Low-Pass Filter

It's used to eliminate the noise from a specific image. It can be expressed as follows:

$$Gaussian\ low\ pass\ filter = H(u) = e^{\frac{-\frac{1}{2}u^2}{{u_c}^2}}$$

The frequence u_c represents the standard deviation of the gaussian filter. A 3D representation of the GLPF is shown in the next image:



a b c

FIGURE 4.17 (a) Perspective plot of a GLPF transfer function. (b) Filter displayed as an image. (c) Filter radial cross sections for various values of D_0.

E) Homography

The **omographic transformation** is an invertible mapping from points defined in P^2->P^2 ; three points called x_1, x_2 and x_3 lie on the same straight line if and only if $h(x_1)$, $h(x_2)$ and $h(x_3)$ lie on that line.
A homography is a linear transformation based on three homogeneous vectors, these vectors are represented by a **3x3** non singular matrix **x' = Hx.** To explain that in a better way, the matrix can be represented as follows:

$$\begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

During the affine transformation (perspective invariance with eight degrees of freedom) ther're the following parameters $h_{31} = 0$, $h_{32} = 0$ and $h_{33} = 1$.
The real benefit of using the homography is the **homogeneous representation of straight lines**. A straight line in a plane can be represented with the following formula:

$$ax + by + c = 0$$

Of course, different choices in parameters a, b and c lead to different straight lines; so a straight line can be represented through a vector (a, b, c) but the correspondence between lines and vector isn't univocal: the straight lines **ax + by + c = 0** and **(ka)x + (kb)y + kc = 0** are exactly the same for every $k \neq 0$. Any class of vector that respects this relation of equivalence is defined as **homogeneous vectors**.
The homogeneous point representation says us that: a point **x = (x, y)** lies on the line **l = (a, b, c)** if and only if **ax + by + c = 0**. This point can be represented as a tridimensional vector in this way:

$$(x, y, 1)$$

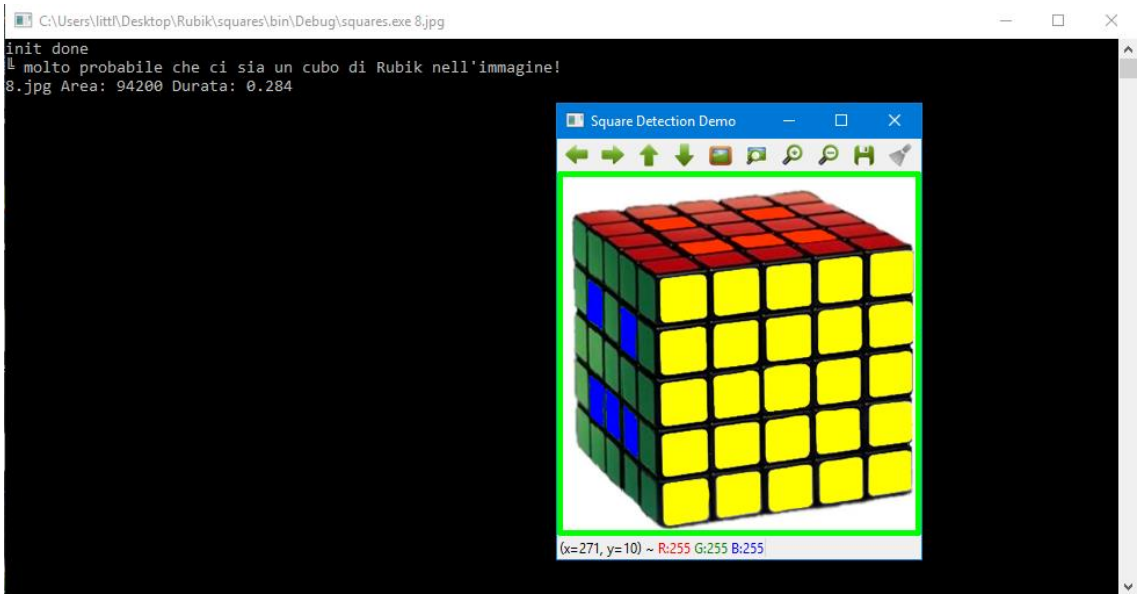
If a point is expressed through a homogeneous vector and that vector's third value is 1, then that point is defined as a **real point**. If the third element is 0, then it is spoken of **ideal points** or **points to infinite**.

6. Tests and obtained results

Several tests were conducted in order to find cases in which the algorithm was producing good results or something strange happened. All of the tests consist in passing a specific image as input for the program, show the result and put in evidence specific problems.

6.1 – Fake positive in an image

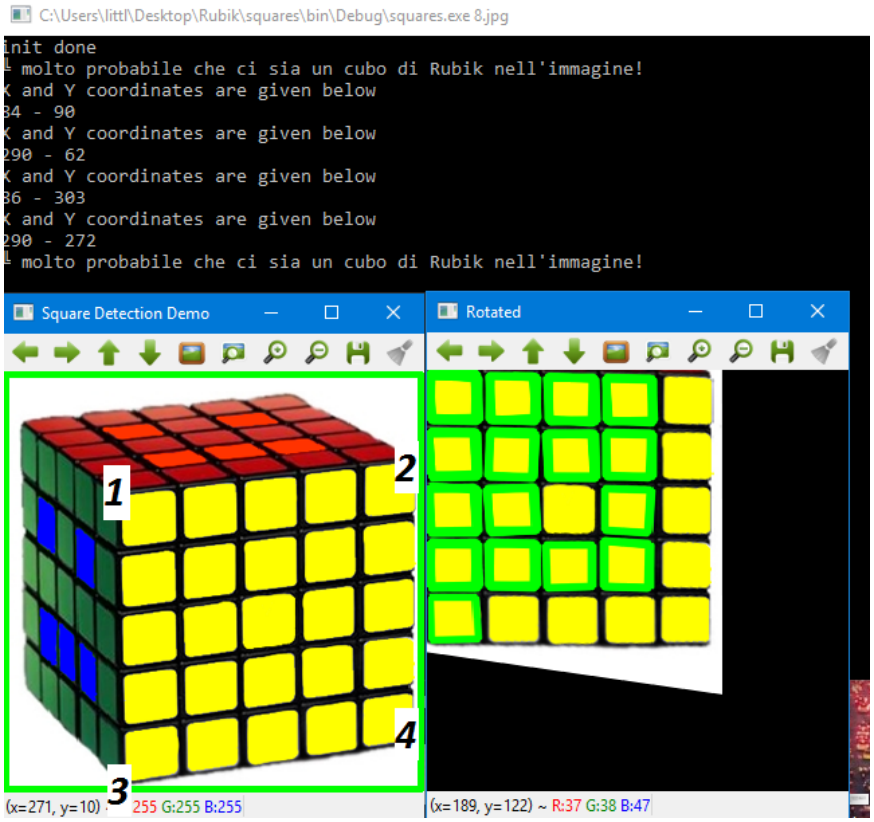
This test was realized by passing in input the following synthetized picture **(8.jpg in the data set)**, representing a Rubik's Cube with all the square edges rounded.



This image leads the program to suppose the presence of a RC, however only the border of the image is put in evidence thus reaching to the conclusion the detected edges are completely wrong. This is the result of the combination of many factors:

1. During the image processing phase, a **0-filled** border is added to the picture in order to outcome the problem of the Gaussian kernel unable to be applied to the border pixels. The consequence is the Canny edge detector will find a certain number of squares regarding the black border, due to the different threshold levels applied.
2. After the image processing phase, the result image is blurred and the starting picture had the little squares not so defined.

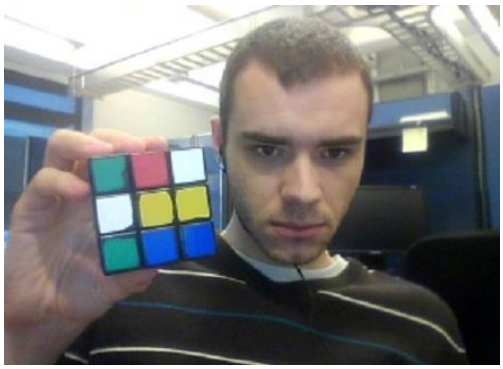
At the end the set of edges which can represent a RC is composed only by the ones regarding the black border and they're printed. A temporary solution is to use the perspective removal approach described in the previous chapters. By choosing four points representing the corners of a RC face, it is possible to set a face perpendicular in the point of view of the user.



This exaple reveals some of the limits of this program, expecially the fact the program sometimes needs some human advise to find a better solution to the one proposed.

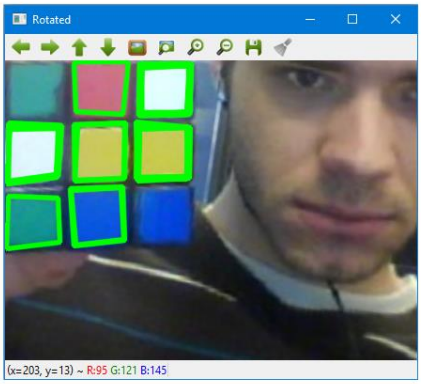
6.2 – The size of the rotated cube

Another problematic image is the picture below (**5.jpg in the data set)**. The edges of the RC aren't sharp as before and some edges are not complete.

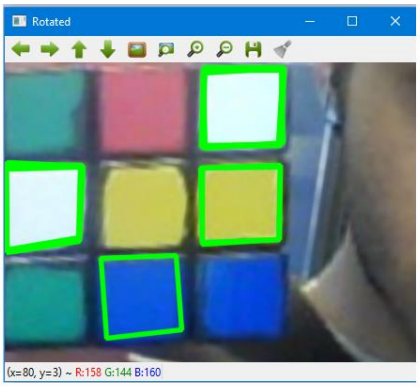


The homographic approach used in the previous test comes again at hand, but there's the necessity to point out that the homography transformation need a reasonable width and height for the new picture without perspective and this allows to get a benefit alongside a problem.
The benefit is the fact it is possible to enlarge a particular zone of a picture without doing any supplementar operations, thus allowing an easier detection of the RC in the image.
However it must be taken into account the original image size and expecially the Rubik's Cube size inside it. Giving a fixed dimension, as it was done in the project (the RC face has size **200x200**) may create problems. A trivial example is done by changing the size to **400x400** and **600x600**.

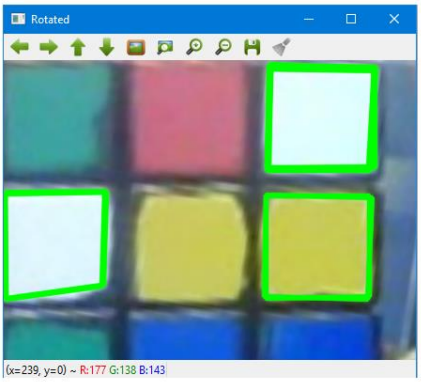
This test shows that increasing the size of a little cube leads to an accuracy loss, ignoring the fact the RC is detected or not. This is due to the increase of the noise with the image zoom.



200x200



300x300

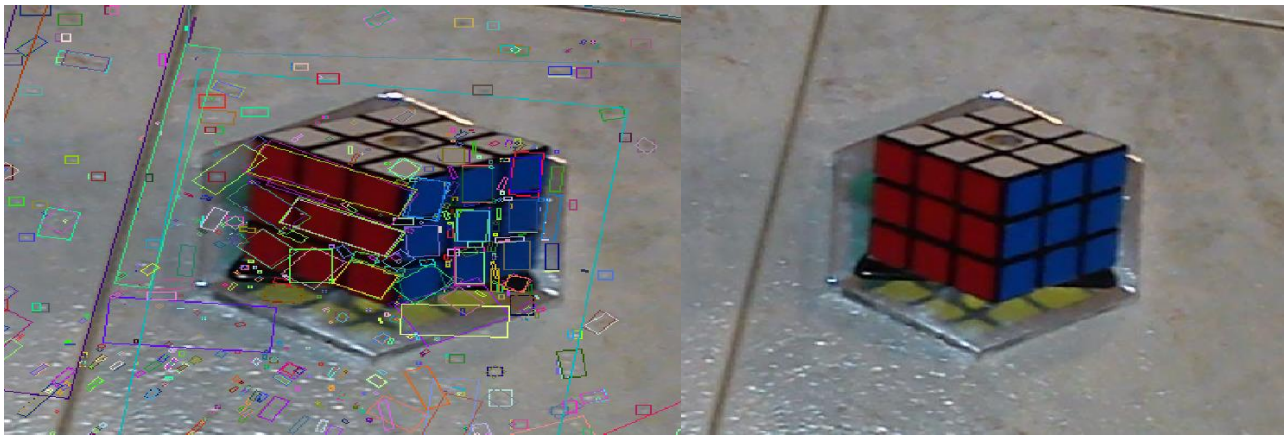


400x400

The same problem happens in the reverse situation, where the image and the RC size are significantly huge and some factors don't allow us to detect the cube. Reducing the image size again introduces noise alongside losing informations.

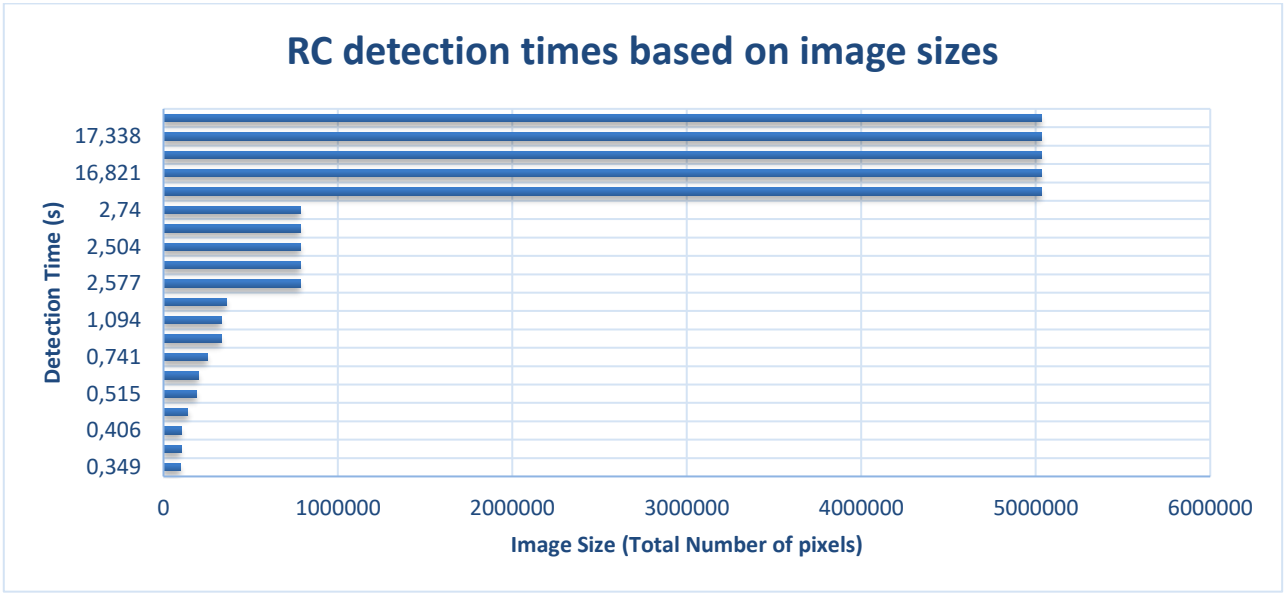
6.3 – A failed attempt to detect the RC

Alongside the working project there's a previous attempt **(inside file squares01.cpp)** to detect the RC inside an image. The algorithm uses a different approach to overcome the perspective problem, focusing on the detection of the lines which form the RC contours instead of the edges surrounding the squares.
A gamma function is applied to the image to enhance the response of the black lines surrounding the squares of the RC, followed by a first application of Canny. The **Hough** line detector is then used to detect and **draw** the lines inside the image, strengthening the squares detected by a new application of the Canny edge detector and the **minAreaRect** method**.** However the idea was left due to the difficulty to estimate which square is suitable for the Rubik's Cube.
The only positive point is the improved square detection **(below, left image)** in respect to what it is obtained with the presented project **(below, right image)**.



6.4. Some considerations about the computational times

The whole program time and space complexity depends strongly on the size of the image passed by input. Below are presented the computational times of the RC detection procedure based on the image set used.



From the data provided, it is possible to make the following considerations.

- **Real-time** applications of this project are not suitable for the algorithm, except the case in which strong assumptions are set, like a data set only composed of pictures with resolution under **400x400**, not a real situation in most of the cases.
- The computational time for a QHD (2560x1600) is around 17 seconds, and considering the increasing spead of cameras able to take photos with this resolution and higher, processing even 100 images is time consuming.
- The format format (jpg, png, bmp, …) and the compression used for the image is not as relevant as the number of the composing pixels. This is due to the algorithms used for the project, starting from the Gaussian Filter, which was left for the different size of the images the program is able to

operate with. **It is possible to remove the Gaussian Kernel to improve the computational speed, in exchange for a more noisy image.**

7. Conclusions and future works

This project focuses on solving the recognition of a RC in space even if it's rotated or is seen from a particular point of view. This problem comes along with a huge documentation because of the fact that it has been addressed several times in history. In the wanted approach, at the beginning to **Keep It Simple Stupid** (**KISS**), the concept was to find all the lines forming the RC contours.
This approach was not working at all and a relevant number of problembs were encountered. Some of them were regarding the dinstinction between the lines which weree part of the RC and those which were not. For the reasons explained in one of the **tests** described previously, the idea was discarded because it was too complex and the edge detection wasn't that good;

Widely speaking, there's room for a range of improvements, starting from the one of facing the problem in a general way.

- If the program is optimized for a specific image side, the improvements of accuracy and computational time would be better.
- Another improvement would be the removal of fake positives due to the black border introduced to the image during the Image Processing phase. A solution to this problem would be not so difficult to implement (eg check if the detected squares area is equal to the input image one). However is was not done due to the fact there were other more important points to take care of.
- Last, **the MapArea structure** created for the classification of the area sizes of the squares **should be improved or substituted with a better implementation**, like the one using the **Map** object of **C++**. In the current version, to improve the speed of the program, the **MapArea** returns the first set of squares which number is equal or greater than 10 (this value is set taking into account different levels of thresholds were used, allowing more than one contour to be detected in the same place). **The consequence is the impossibility to detect two or more RC with different sizes in the same picture.**