Report - First Defense

Chaman

October 2021

BÉGUÉ Augustin | BELVAL Anthony | HARIGNORDOQUY Mathis

Table of Contents

1 Introduction

1.1 Group Presentation

1.1.1 Augustin BÉGUÉ

I started to be interested at IT and especially coding at around 12. I have started doing it in my free time after school and it gradually became my passion and my main activity. I am starting to accumulate a lot of experience in this field, especially with TypeScript and C# two languages I have used a lot for my personal projects and in my professional experiences. I have never used C before Epita, but I have already done some Image Processing before, so I am starting the project with a good feeling.

1.1.2 Anthony BELVAL

If I had to give a single adjective to describe myself it would be curious. Logic and mathematics are my areas of predilection. Computer science is for me the perfect association between these two areas. I'm very excited to start this project, especially the neural network because I am very interested in AI and I think I want to specialize in AI later in Epita.

1.1.3 Mathis HARIGNORDOQUY

Ever since I remember I always have been surrounded by objects such as puzzles that require logical thinking in order to be solved. Among these puzzles are obviously the Rubik's cube as well as the Sudoku. Having to design a program capable of solving Sudoku is an interesting task knowing my enthusiasm for this types of puzzles. It is with interest that I approach this project, especially since the C language is a new language to learn for me.

1.2 Task Repartition

We've decided to use a "chronological" task repartition because we felt this was the best thing to do since we have to create a single program with a very defined workflow. This task repartition limits the interaction between the pieces of code of different members and thus makes things faster for all of us.

	Augustin B.	Anthony B.	Mathis H.
Image Loading			
Color Removal			
Pre-processing			
Grid detection			
Retrieval of the digits			
Character recognition			
Reconstruction of the grid			
Grid resolution			
Display of the solved grid			
Saving the solved grid			
Graphical User Interface			

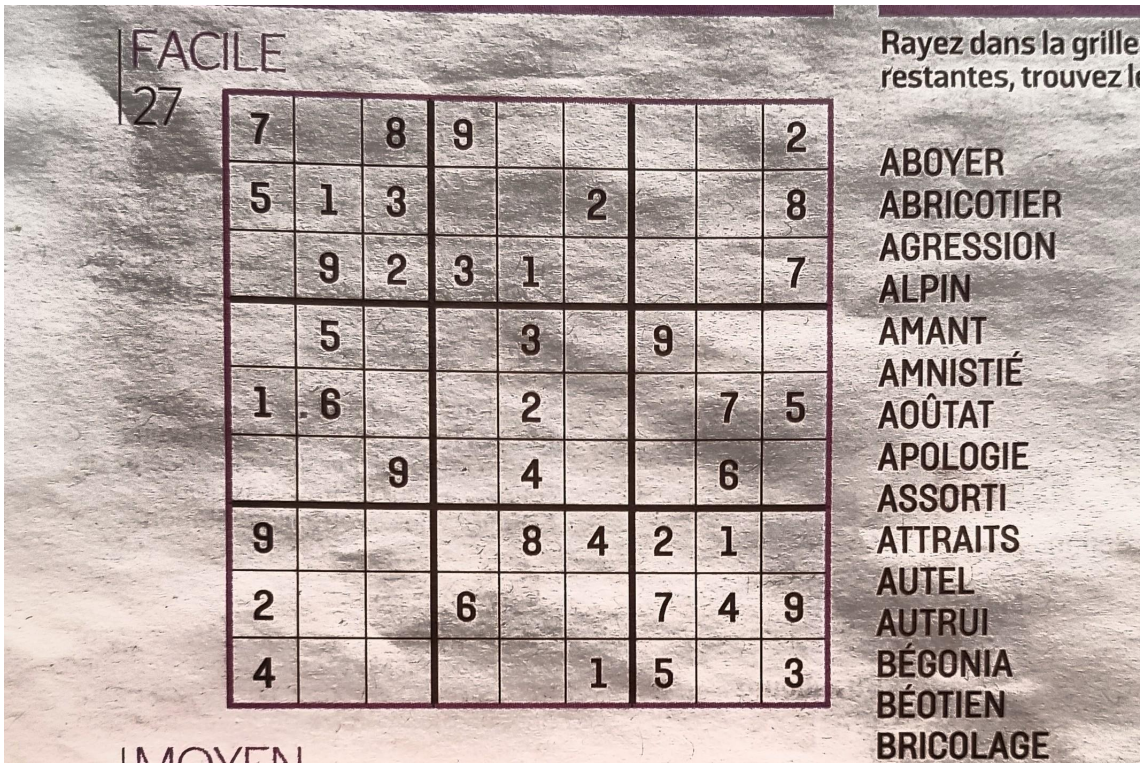
1.3 Progress of the Project

We are extremely happy with our current progress, especially considering that we are only 3 in our group. The only part that are left to be finished are the tasks that were not mendatory for the first defense.

	Progress		Progress
Image Loading	100%	NN Weight Saving & Loading	50%
Color Removal	100%	Reconstruction of the grid	0%
Pre-processing	60%	Grid resolution	100%
Grid detection	70%	Display of the solved grid	10%
Retrieval of the digits	10%	Saving the solved grid	100%
Character recognition	40%	Graphical User Interface	0%

2 Image Processing

Throughout the Image Processing and Image Splitting parts, this image will be used as a reference. This is because this is the one that is the most clearly showing the performance and working of our algorithms.



Reference: Image 4

2.1 Color Adjustments

Our goal in this part of the processing is to modify the images to make their colors even to facilitate the next parts of the processing.

2.1.1 Grayscale

We start the processing of the image by converting it to grayscale. The formula used is the following:

$$color = 0.4 * red + 0.35 * green + 0.25 * blue \tag{1}$$

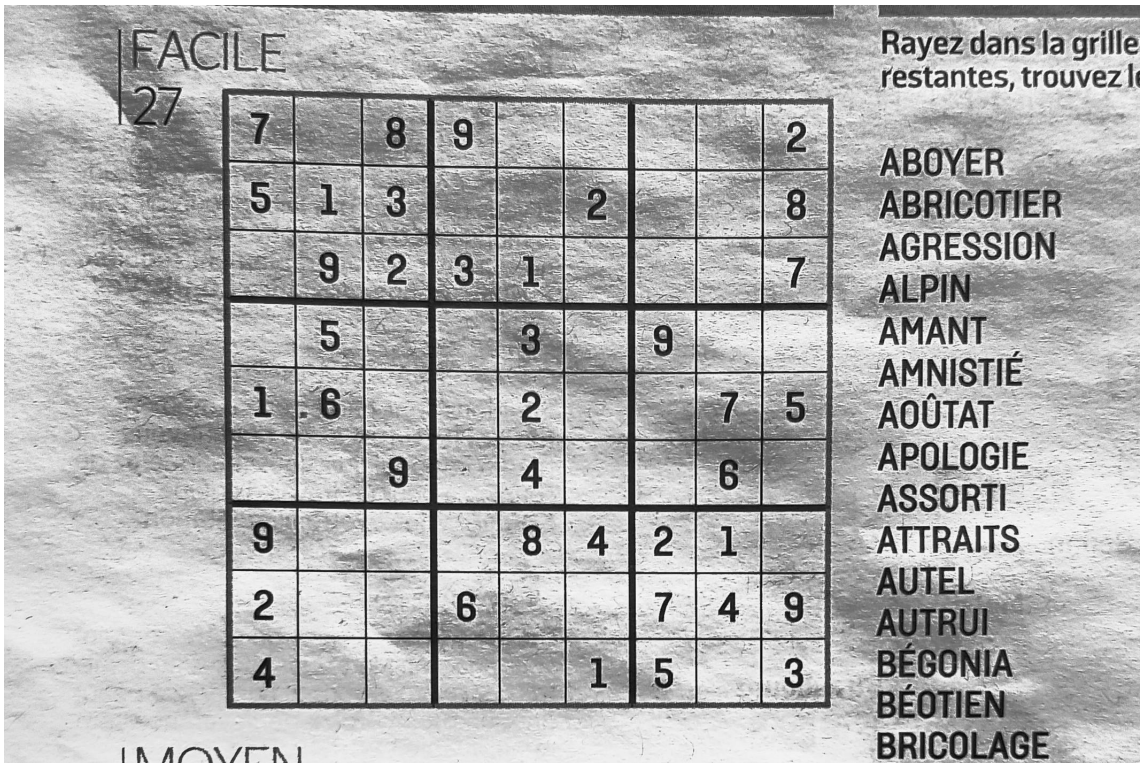


Image 4 - Grayscale Conversion

2.1.2 Gaussian Smoothing

The next step of the process is the Gaussian smoothing. We use it to smooth the noise contained in the image and even out the colors to make further processing easier. The process of this filter is straightforward. For each pixel of the image, we compute a new value from the sum of the product of neighbouring pixels and a Gaussian matrix. This matrix is generated using a Gaussian function, which is of the following form:

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

We use a kernel of size $10 * 10$ and a sigma of value $\sigma = 6$.

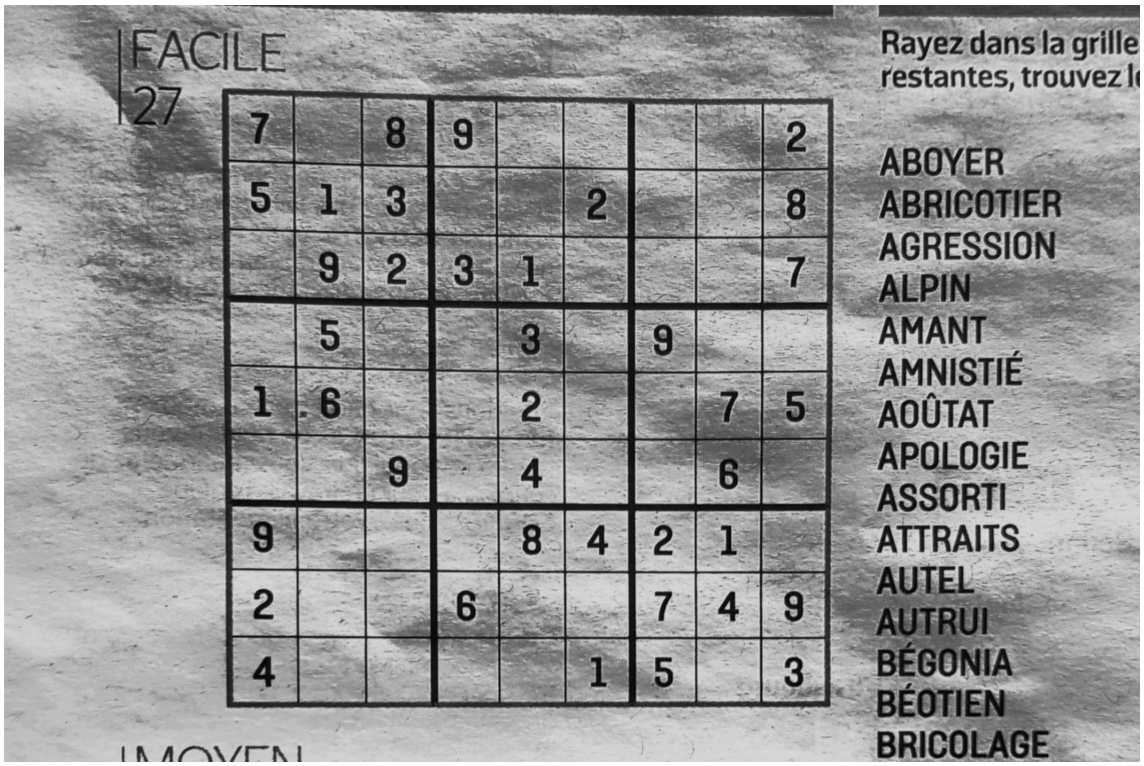


Image 4 - Gaussian Smoothing

2.1.3 Contrast & Gamma

The tweaking of the Gamma of the image allows us to accentuate the gap between the bright and dark parts of the image. The used formula is the following:

$$color = \frac{color^{\frac{\gamma}{128}}}{255} \tag{3}$$

With $\gamma = 512$.

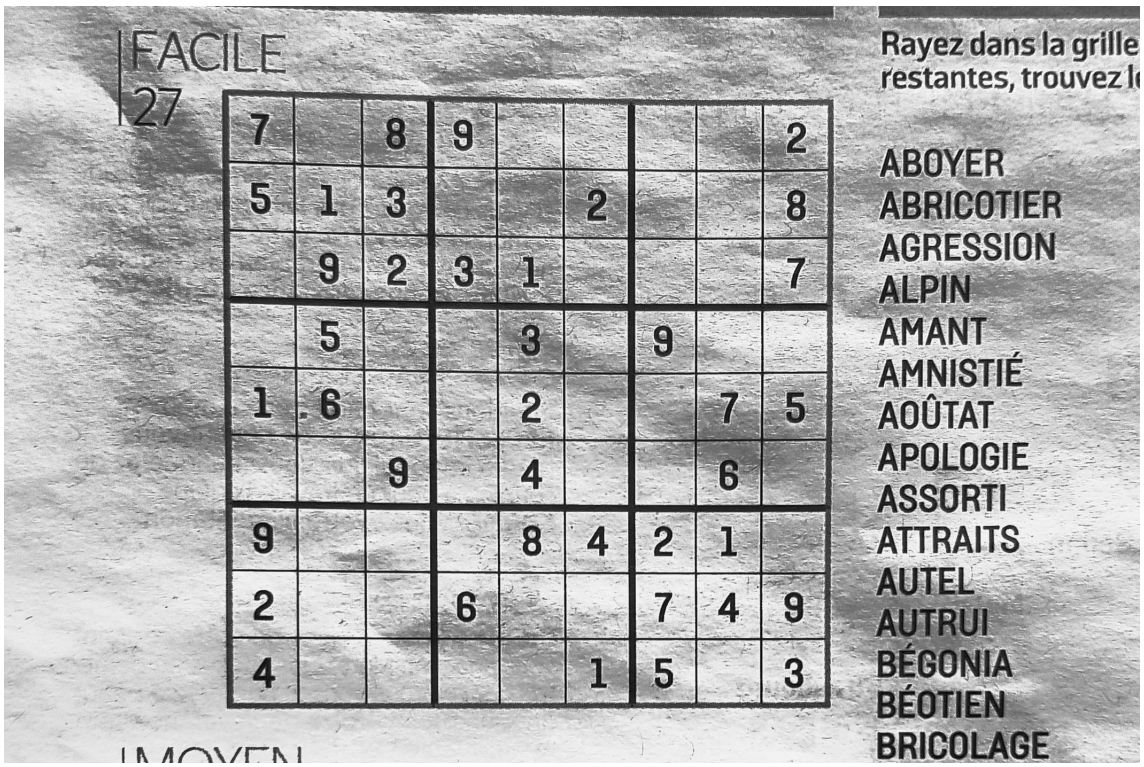


Image 4 - Gamma Adjustment

2.2 Noise Removal

The goal of this part is to remove the noise contained in the images to make the grid and character recognition easier.

2.2.1 Dilation & Erosion

The Dilation ($\oplus$) and Erosion ($\ominus$) operators are used for noise removal. We use them here to remove the bulk noise of the images and sharpen the lines and numbers of the grid. These operators are often used concurrently. We begin by dilating the image, which enlarge the details of the image and thus makes the features like lines and characters clearer.

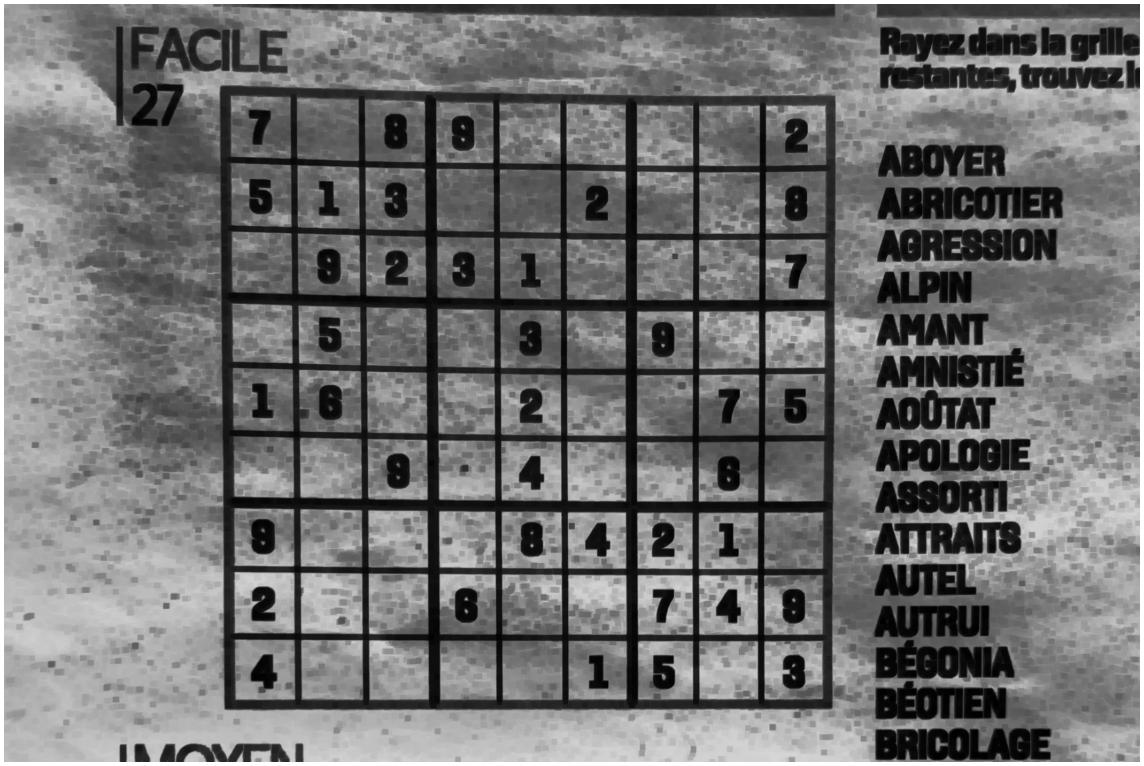


Image 4 - Dilation

We then erode it, to remove the added width to the detail and restore their original size. The result is an image with the same principal features, but with less noise and a more even background.

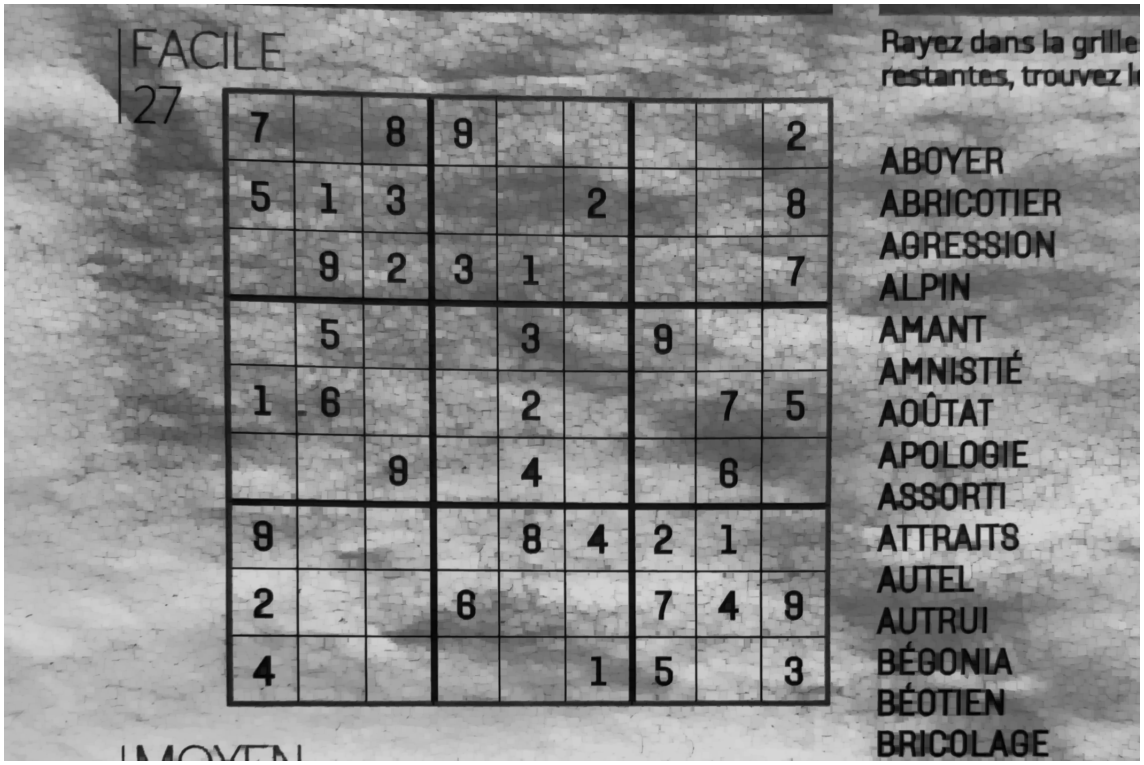


Image 4 - Erosion

These two operations enable us to prepare the image for the next step by giving us images with an even background and sharp features on top of it.

2.2.2 Thresholding

Thresholding is used to binarize the images (make the colors 0 or 255) in order to only keep the features we want on the image. The algorithm we decided to use is Otsu's method, named after Nobuyuki Otsu. It allows us to perform an efficient automatic thresholding on images with an even background. This method uses the variance of an histogram filled with the number of pixels at each value (from 0 to 255) to find a threshold value located between the colors of the foreground and of the background of the image.

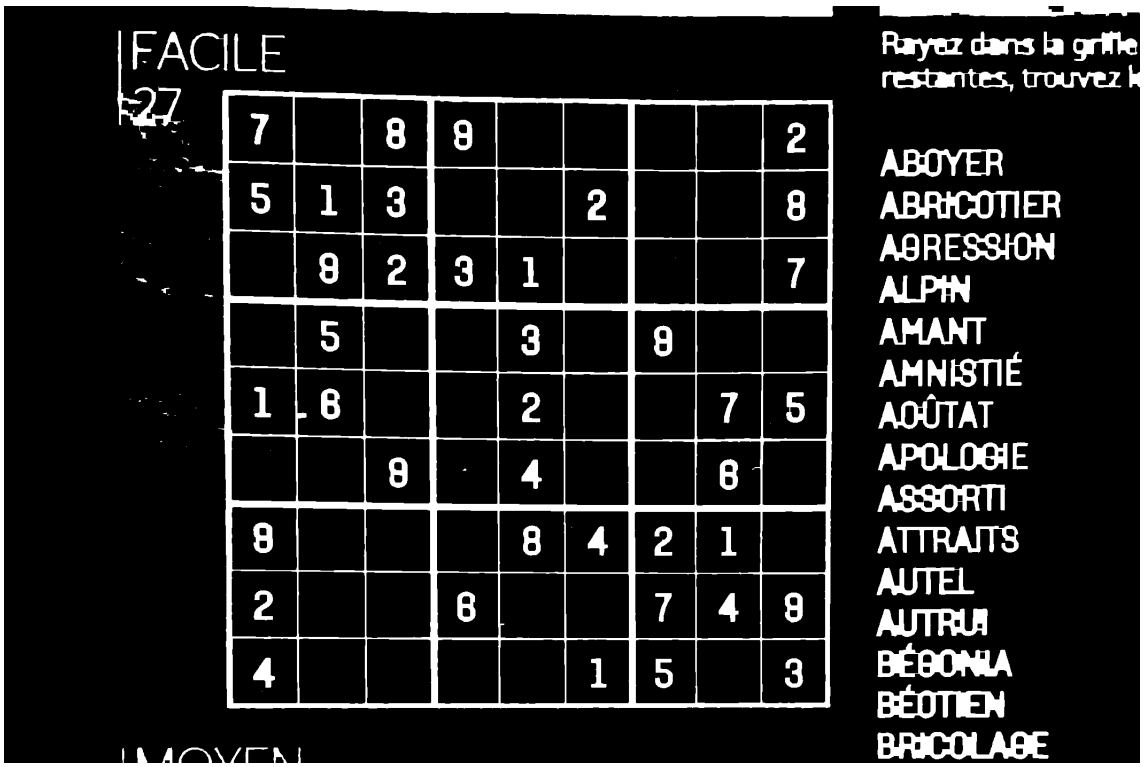


Image 4 - Otsu's Thresholding

As you can see, the result for simple images is very successful.
We also implemented a Sauvola thresholding. This is an adaptive threshold used for images where the background is not even at all. While it is not implemented in our main pipeline yet, it allows us to process more difficult images where the lightning condition is poor or the background is constitued of multiple colors.

2.3 Image Rotation

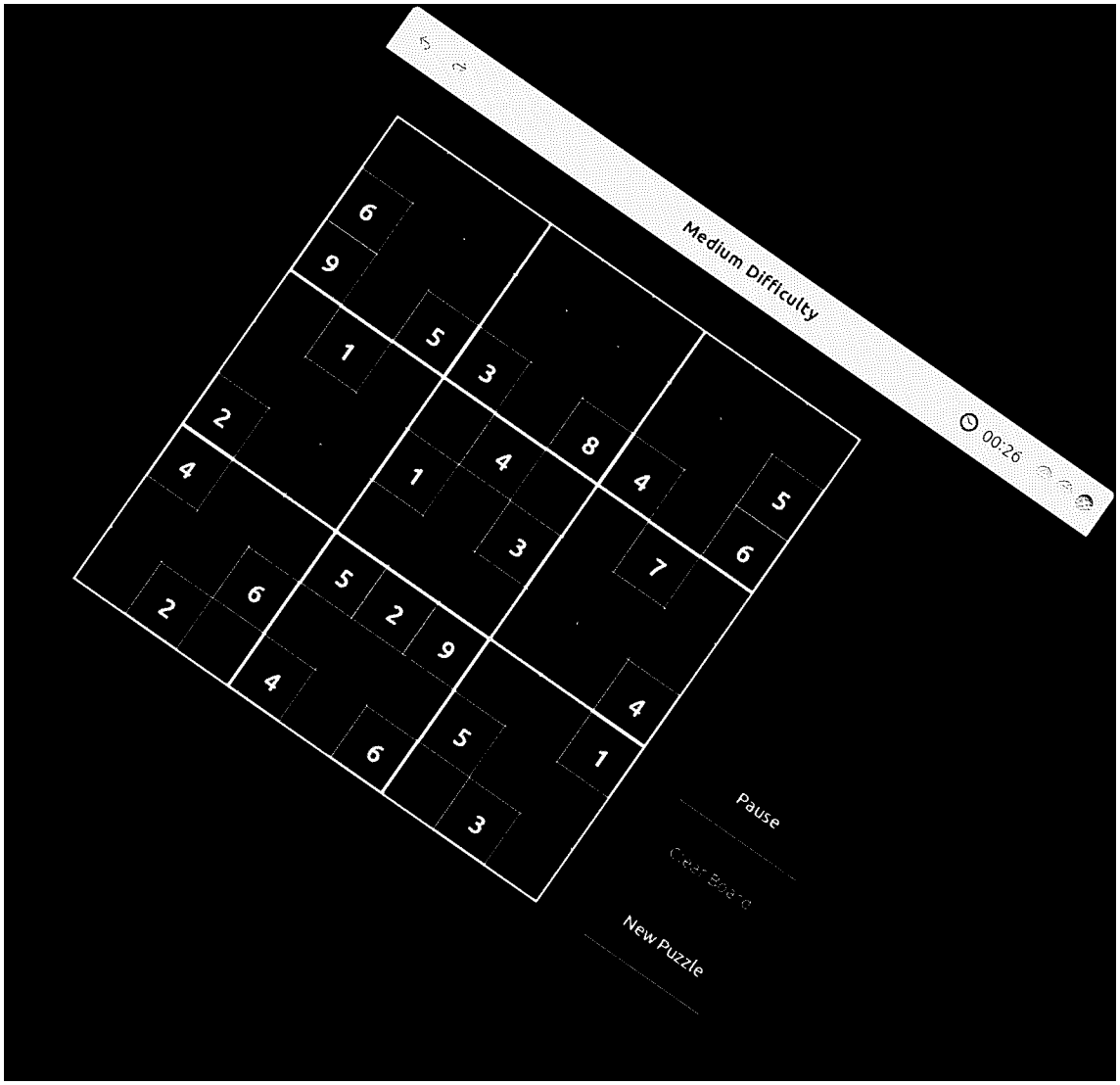


Image 5 - Before Rotating

To rotate the image, we decided to use a rotation matrix. This method is fast and accurate, giving us results without jagged edges and allowing us to do rotations with very accurate angles. The rotation matrix expands to the following formulas to find the coordinates of the rotated pixels (x, y) by an angle θ on the original image (x', y'):

$$x' = x\cos\theta - y\sin\theta \tag{4}$$

$$y' = x\sin\theta + y\cos\theta \tag{5}$$

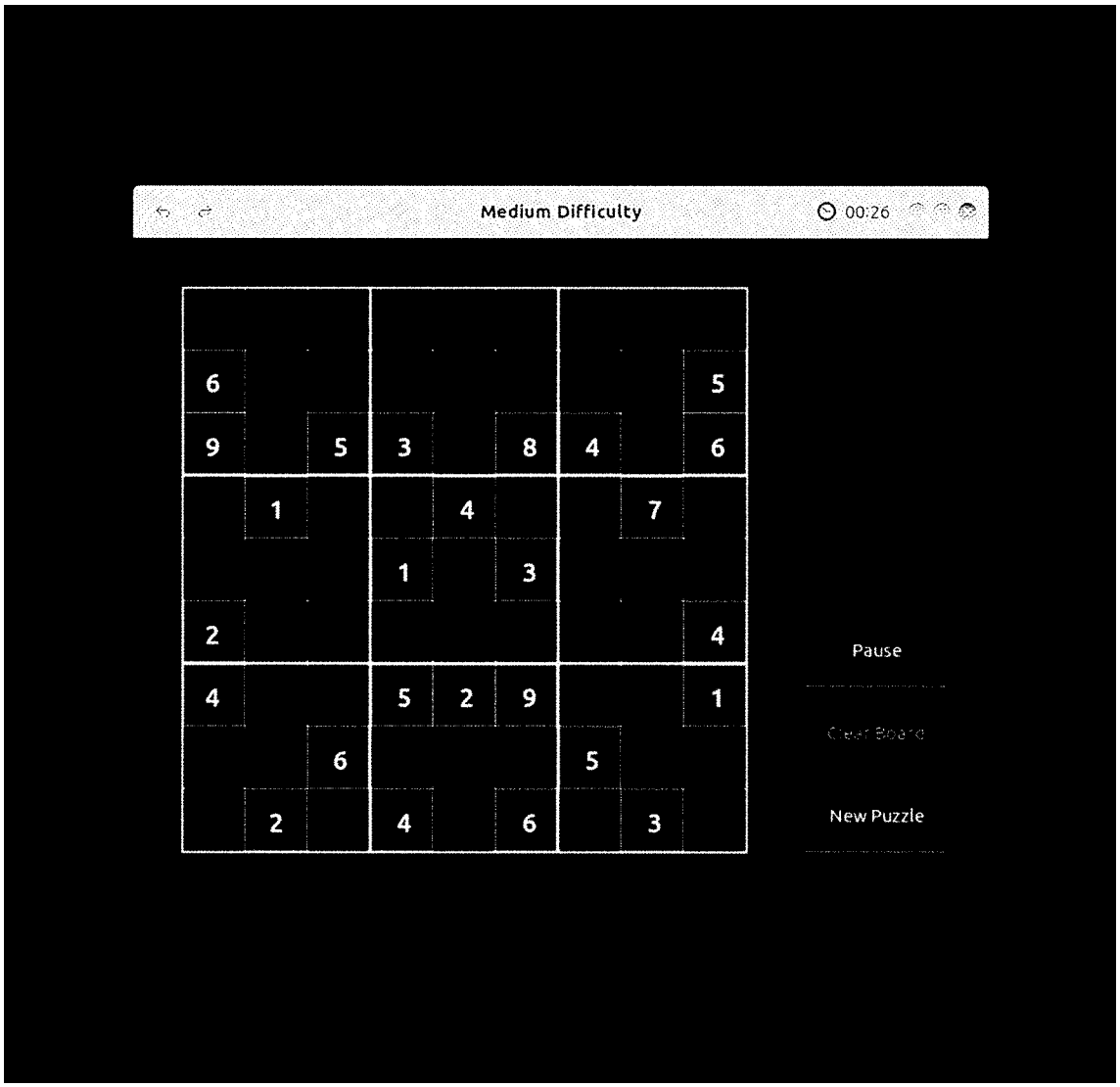


Image 5 - After Rotating by 35°

2.4 Edge Detection

For the Edge Detection part of the Image Processing, we've decided to use the Canny Edge Detection algorithm. It is made of the three following parts.

2.4.1 Sobel Filtering

The first part of the canny edge filtering is the creation of a gradient representing the direction of the edges of the image, called intensity gradient. To do that, we start by creating a gradient in x (G_x) and a gradient in y (G_y), by using the corresponding Sobel operators:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * Image \quad (6) \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * Image \quad (7)$$

In these equations, the $*$ operator represents the *Convolution* operator. This operator iterates trough an image and sums the product of the values of the kernel and the corresponding pixels of the image.
We then compute the magnitude gradient, which represent the presence of an edge on the image:

$$G = \sqrt{(G_x)^2 + (G_y)^2} \quad (8)$$

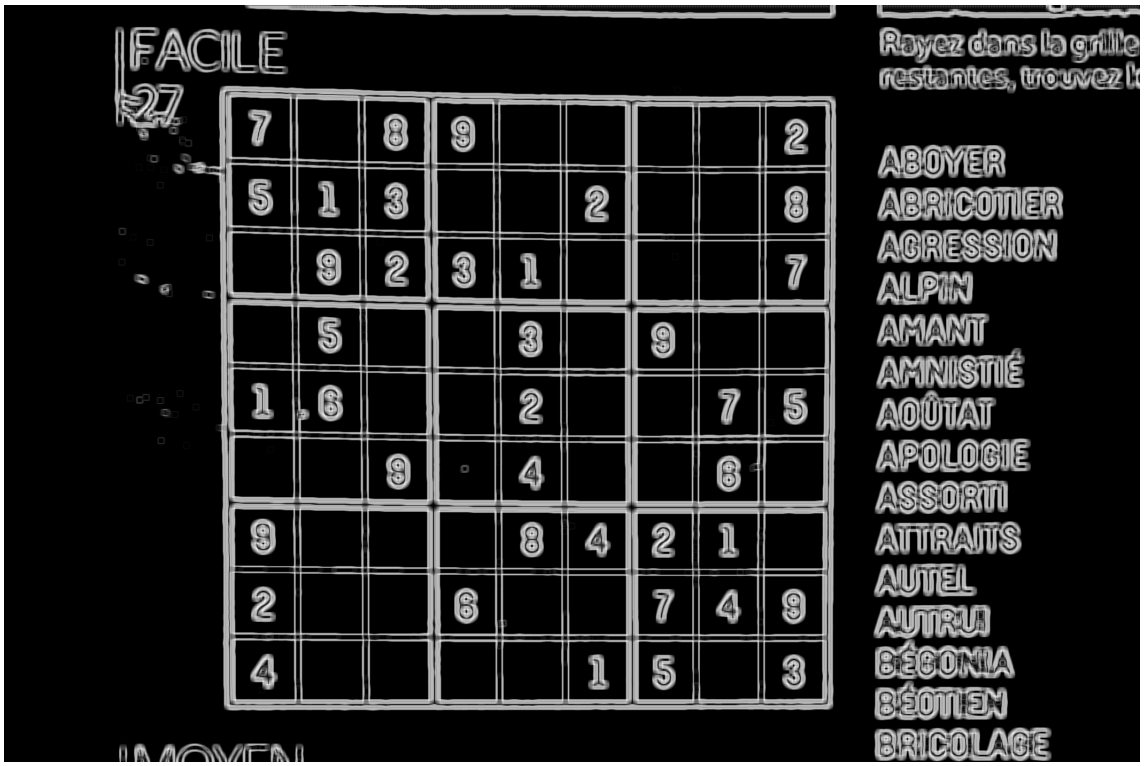


Image 4 - Magnitude Gradient using Sobel Filter

2.4.2 Non Maximal Suppression & Hysterisis Analysis

These two algorithm are used to thin the edges found in the Magnitude Gradient. They are long and not very interesting, so we will not go trough their process to keep the length of this report down.

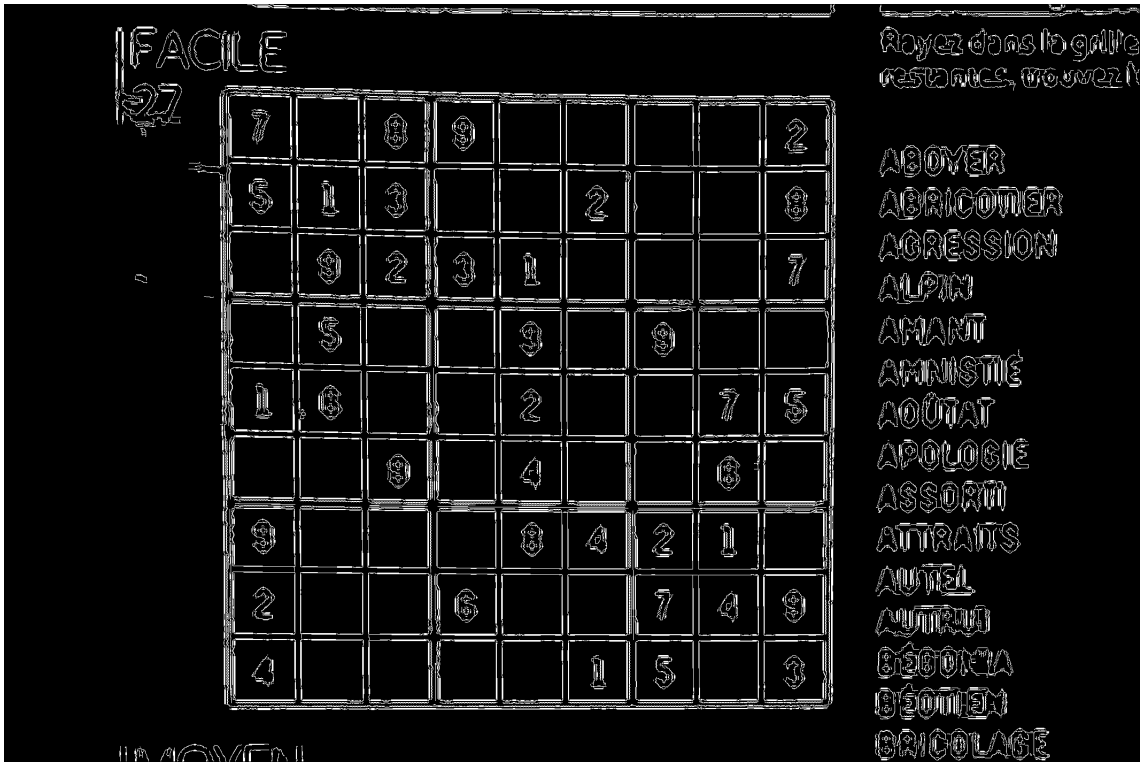


Image 4 - Canny Edge Filter

The result of the canny edge detection is an image where every possible edge of the image is represented by a white pixel. This image can then be used with line detection algorithm to find the coordinates of these edges.

3 Image Splitting

Here is the pipeline we decided to use to split the image into the cells of the grid of the Sudoku:

- Line Detection using Hough Transform
- Square Detection from the line's coordinates
- Selection of the square most likely to contain the grid
- Cropping of the image following this square
- Splitting of the image in 9x9 parts to get the cells

3.1 Line Detection

To detect and retreive the coordinates of the line on the image, we are using the Hough Transform algorithm. This algorithm maps edge points from a filtered image on the Hough Space to represent lines. In the Cartesian space (the edge image's space), the lines are represented in the form of this equation:

$$y = ax + b \tag{9}$$

For the purpose of the algorithm, each white point on the edge image is considered as a line. Each line on the edge image creates a point on the Hough Space, of coordinates (ρ, θ), with:

$$\rho = x * cos(\theta) + y * sin(\theta) \tag{10}$$

and θ going from $-90\,\text{deg}$ to $90\,\text{deg}$.
This formula is used to cast votes in an accumulator. Each white point of the edge image creates a vote in the same region as the other points located on the same lines. These votes are then used to find the most dominant lines on the Hough Space by finding local maximums for each regions that are above an arbitrary threshold (70% of the maximum number of votes in our case).
The mentioned accumulator creates the following image when plotted and normalized. The local maximums that we have to find in order to have the dominant lines are represented as the whites zones where the gray curves all meet.

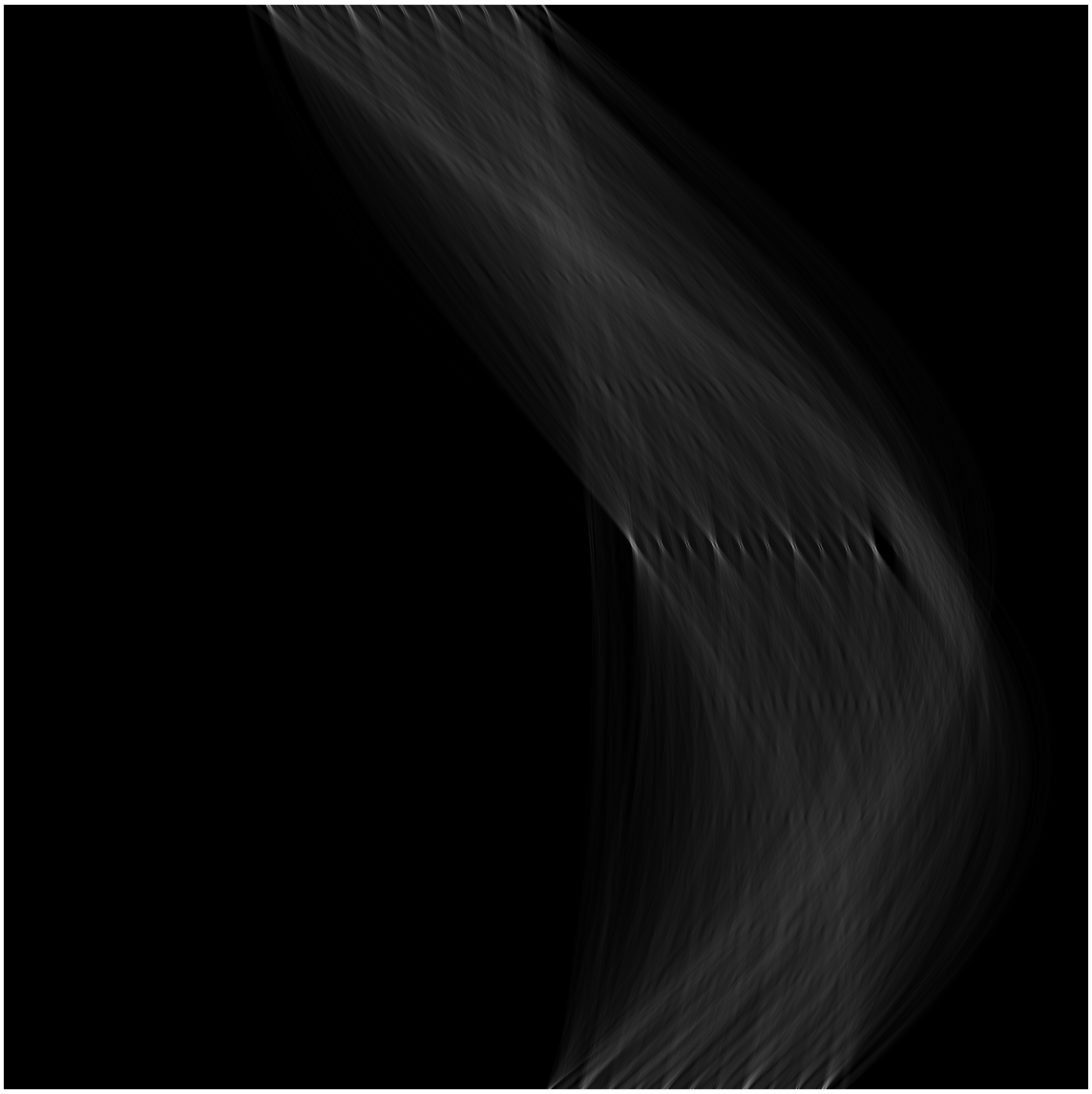


Image 4 - Votes of the Hough Space's accumulator plotted

When the local maximums of coordinates (ρ, θ) are found, the following formula can be used to find the Cartesian coordinates $(x0, y0)$ of a point located on the line represented by the maximum:

$$x0 = \rho * \cos\theta \tag{11}$$

$$y0 = \rho * \sin\theta \tag{12}$$

We can then add an arbitrary distance and an the same θ to find a second set of coordinates located on the line:

$$x1 = x0 + d * (-\sin\theta) \tag{13}$$

$$y1 = x0 + d * \cos\theta \tag{14}$$

Finally, when the coordinates of the selected lines are found, we can draw them on the image. As you can see, the results are pretty accurate. However, there is still many lines and we need to average them to only get a few dominant ones.

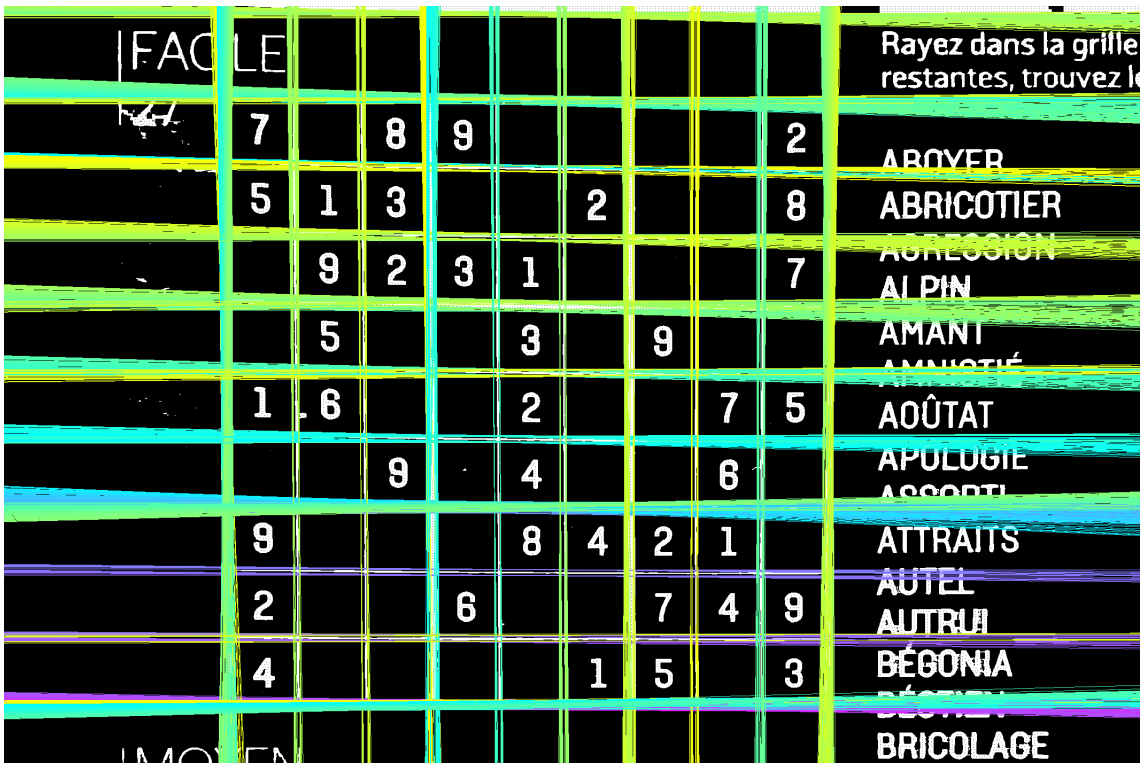


Image 4 - Hough Transform lines converted to Cartesian coordinates and drawn on the image

3.2 Square Detection & Selection

As said before, we need to average the found lines to only get the dominant ones and speed up our further calculations on them. In order to do that, we've decided to average the lines that have neighbouring coordinates.

- We begin by adding the first found line to a list
- We then iterate over the following lines
  - If the difference of the current line starting/ending point with the starting/ending point of one of the lines contained in the list is lower than an arbitrary threshold, we modify the entry of the list to the average of these two lines.
  - Otherwise, we add the current line to the list

The result of the averaging of the lines found is shown in the following image. As you can see, the only remaining lines are following the grid and some of the other dominant features of the image.

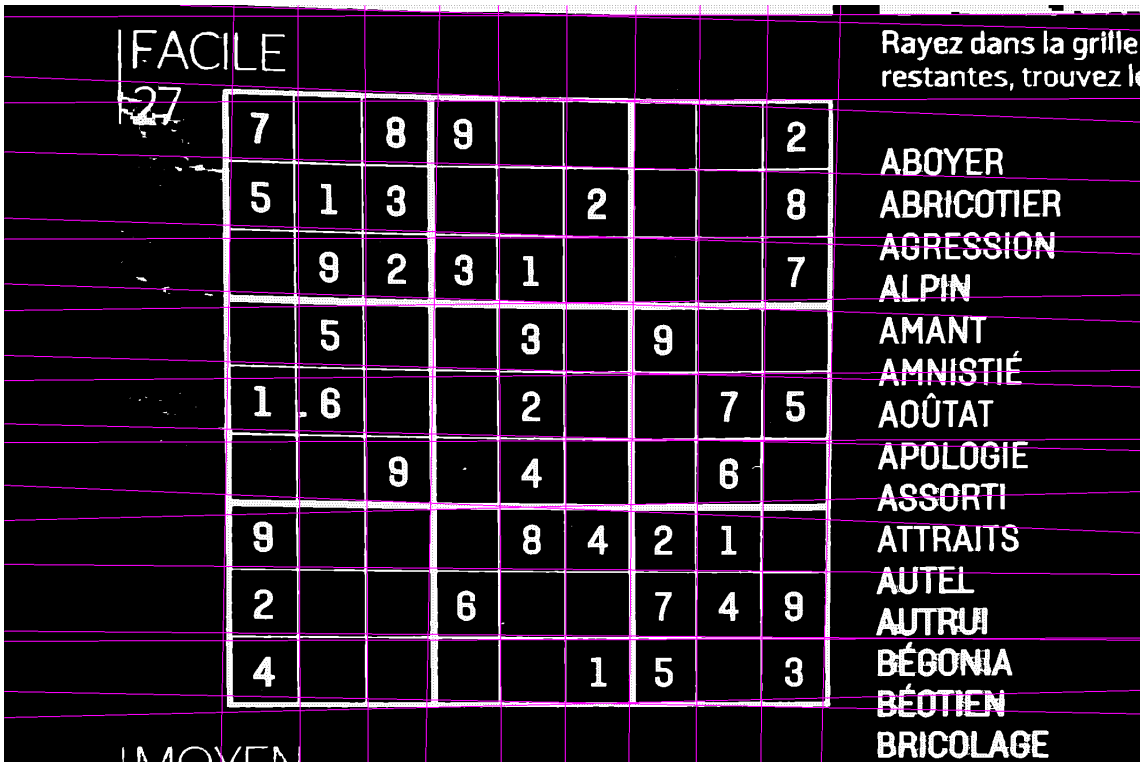


Image 4 - Reduction of the number of edges detected by averaging

The next step is the square detection. To perform this task, we've created a straightforward algorithm. While it could probably be improved, it is fairly efficient and can be reimplemented with multithreading to speed things up.

- We start by iterating over all the remaining lines
  - For the selected line, we find all the intersecting lines.
  - If the intersecting line is not the selected line, we continue
- If the number of intersecting lines found before refinding the first one is 4, we consider that the set of lines found is a square

Every found square can be seen with green edges on the following Image.
We then need to select the correct square which follows the outer edge of the Sudoku's grid.
To do that, we compute a factor for each found square, which is composed of:

- The squareness: the difference between the smallest and the longest edge of the square (lower is better)
- The area (higher is better)

We then select the square with the best factor, and consider it is the delimitation of the grid. This square is shown as the red square on the following image.

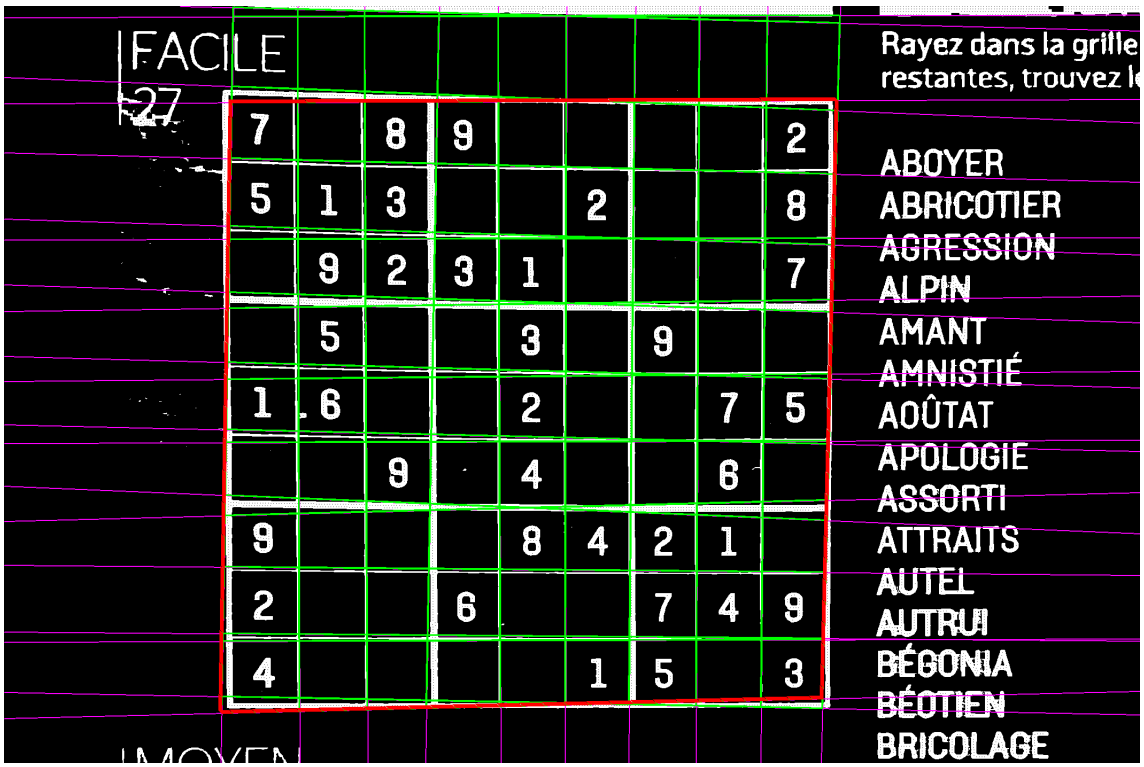


Image 4 - Square detection and selection

3.3 Square Cropping & Grid Splitting

These remaining part are trivial. To make the image cropping follow the found square, we find the closest corner of the square to the origin, and extract the pixels located from the corner's coordinates to the corner's coordinates added to the length of the longest edge of the square.
The following image is the result of the cropping made following the found square.

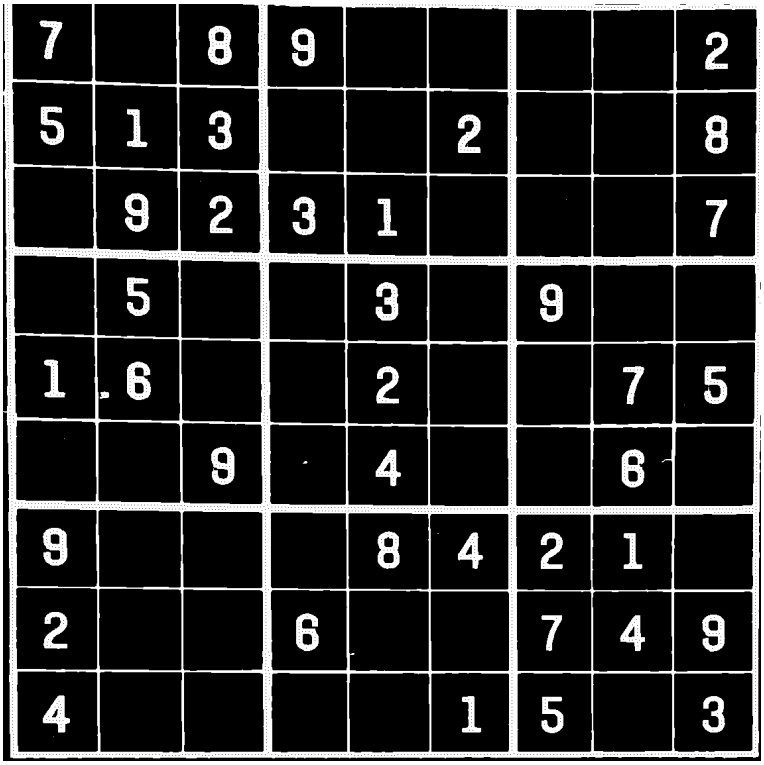


Image 4 - Cropped

Finally, we split the image in 9 parts on the x axis and 9 parts on the y axis to extract the grid cells. The following images are the extracted numbers contained in the first column of the cropped image.

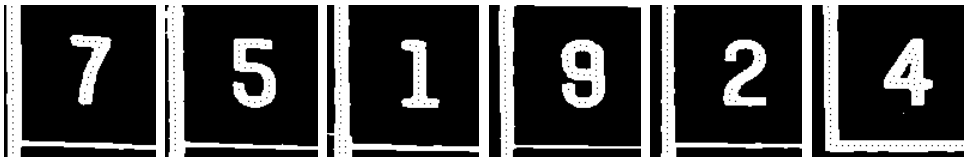


Image 4 - First column splitted in cells

4 Sudoku Solver

Although we had already studied Sudoku solvers last year during a practical work, the design of the the solving algorithm was simpler than expected. As a result the theoretical and mathematical aspect has been exempted from us.

Like other backtraking problems, Sudoku can be solved one by one by assigning numbers between 1 and 9 to empty cells. If the assignment doesn't lead to a solution, then another number if tested in the current cell of the grid.
Here a simpler method is used. Indeed all the possibilities are generated and tested as so :

- For every empty cells in the Sudoku's grid, a number between 1 and 9 is assigned.

- Once all the empty cells are filled, the Sudoku is tested using the Sudoku's rules : every squares has to contain a single number, only the numbers from 1 to 9 can be used, each 3 by 3 squares can only contain each number from 1 to 9, each vertical column can only contain each number from 1 to 9 and each horizontal rows can contain each number from 1 to 9.

- First we check if all numbers are present only once in each lines. Then the same check is done is each rows. Finally this check is done in each 3 by 3 squares in the Sudoku's grid.

- If all those checks are correct the solved Sudoku is written in a file. Otherwise, an error message is displayed.

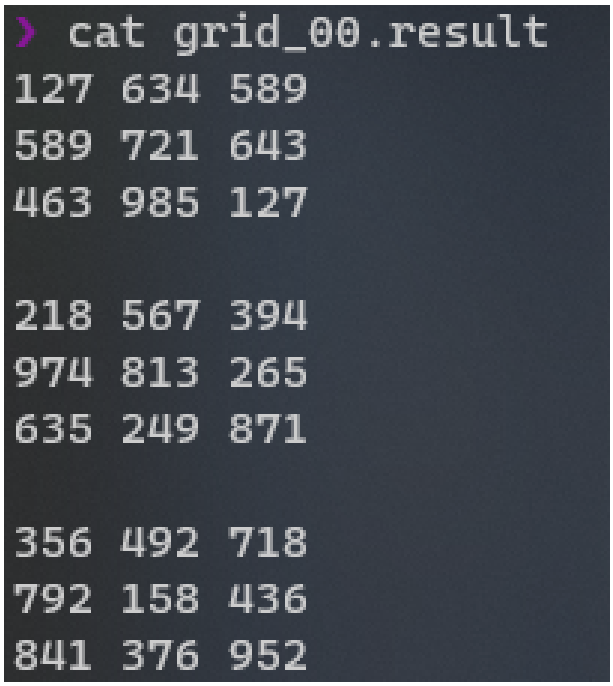
As specified in the book of specification, the Sudoku's solver algorithm is an application that runs from the command line only. The solver takes as input a file in which an incomplete grid is saved.

```
❯ ls
Makefile  README.md  grid_00  main.c  main.d  main.o  solver  solver.c  solver.d  solver.h  solver.o
❯ ./solver grid_00
❯ ls
Makefile  README.md  grid_00  grid_00.result  main.c  main.d  main.o  solver  solver.c  solver.d  solver.h  solver.o
```

Sudoku Solver Application

The algorithm solves the grid. If a solution is found and file containing the result is generated. The name of the generated file is the name if the input file with addition of the extension ".result".

The solved gird is then displayed according to a specific model.



Solved Grid Display

Newt thing is to upgrade the quality of the display by displaying an image of the solved grid instead of external data.

Moreover, displaying the new cells filled with different colors would make it possible to clearly differentiate the cells initially filled from the ones filled following the application of the program.

5 Neural Network

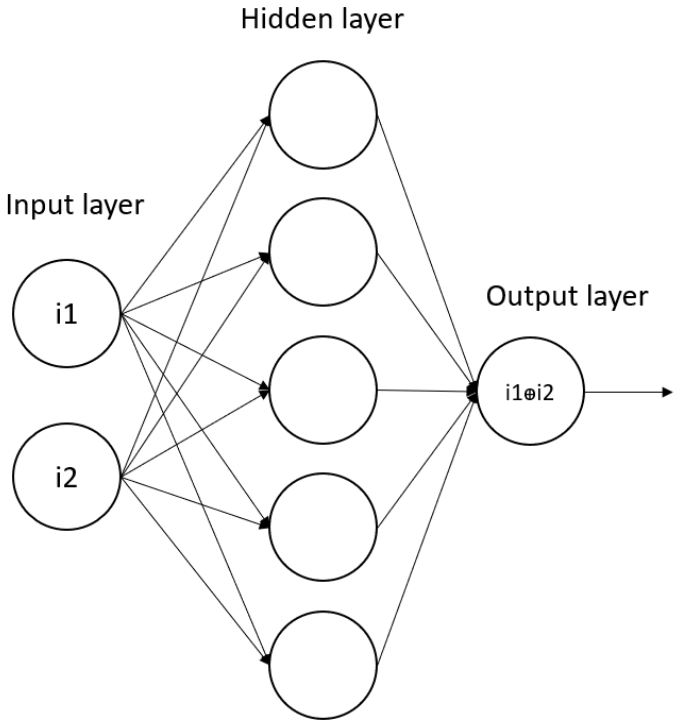
Doing the neural network for the XOR was a very interesting work. First the theoretic part, learning what is a network, what is it made of was very instructive, especially the back propagation which is the most important part of the work.

Then the practical part began. After reading the code provided in e-book, thinking of a way to imitate the Numpy library was the main issue. Creating a structure called Matrix seemed to be the best answer. In this structure are defined a lot properties on matrix such as multiplication, transpose for example. With this half was done.

```
// set the value at a given index
Matrix *m_setIndex(Matrix *a, int r, int c, double val);

// map all value of a row according to the function given in parameter
Matrix *m_map_row(Matrix *a, int r, double (*f)(double));

// map all value of a column according to the function given in parameter
Matrix *m_map_col(Matrix *a, int c, double (*f)(double));

// map all value of the matrix according to the function given in parameter
Matrix *m_map(Matrix *a, double (*f)(double));

// add matrix b to a
Matrix *m_add(Matrix *a, Matrix *b);

// substract matrix b to a
Matrix *m_subtract(Matrix *a, Matrix *b);

// the matrix resulting is the matrix where indexes of same place are multiplied (it's just named hadamard product...)
Matrix *m_hadamard(Matrix *a, Matrix *b);

// multiply a matrix by a single double
Matrix *m_scalar_mult(Matrix *a, double x);

// add a double to each value of the matrix
Matrix *m_scalar_add(Matrix *a, double x);

// return a new matrix being the product of a and b
Matrix *m_mult(Matrix *a, Matrix *b, Matrix *dest);

// return a new matrix being the transpose
Matrix *m_transpose(Matrix *a, Matrix *dest);
```

Main properties on matrices

Last thing to do but not least the network it self. The network is a translation of the code given in the e-book mixed with other things found when looking for information on how to implement a neural network. So implementing the feed forward, sigmoid function, cost function... The back propagation was a tough part but it works after a lot of research and hours of trying to understand.
The network takes different parameters, first the number of input, number of outputs, number of hidden neurons (there will always be only one hidden layer), number of epochs (kind of the number of generation of AI) and number of training tests. Setting these parameters is a crucial stage. Too much hidden neurons will decrease the accuracy, same for the number of epochs. This was a lot of testing to find the best combinations of parameters. So here is a picture of the network.



XOR network's

The final network is made of two inputs, two doubles to compare with the exclusive or, 5 hidden neurons and one output which is the result of the exclusive or then there are 1,000,000 epochs and 4 training. Here is a result of the XOR after training.

```
Result of tests after training
Result for (0, 0) (Expected: 0): 0.001188
Result for (0, 1) (Expected: 1): 0.996129
Result for (1, 0) (Expected: 1): 0.996045
Result for (1, 1) (Expected: 0): 0.005010


Accuracy of the network:
Passed 100000 out of 100000 tests. (Accuracy = 1.000000)
Total error over 100000 random tests: 0.388701
Average error per trial: 0.000004
```

result of a trained network

Next thing is to adapt the network for digit recognition and implementing the function softmax.

Another part of the network is to save the final weights and bias after learning stage. All weights and bias are saved in a specific file. As on the picture below, those data are

stored using matrices , there are the dimension of the matrix followed by all the values. Next thing to do is a function that loads the data from a file.

save

```
2
8
4.972494
-0.133161
2.764573
0.976580
4.849846
0.236541
1.281854
5.772450
-3.299796
1.785419
3.088688
1.012580
4.679147
3.212272
2.438733
5.615913
1
8
1.567621
-0.403036
-4.452655
-1.262553
-1.590042
-1.816426
-2.715700
-2.121886
8
1
-5.795848
-2.004169
-5.784204
-2.120369
6.303753Q
-4.151865
-3.811864
8.440718
1
1
0.125562
```

Save file

6 Conclusion

To conclude we feel comfortable about the project advancement. The image processing part is nearly done, we will work on the automatic rotation and improve the algorithm to enable the processing of more varying and harder image. The Sudoku solver is almost done, now let's work on displaying an image of the result instead of just the result on the console. The basic structure for a neural network is done and working in the Xor, saving bias and weights also, then the loading of a saved network will be done as well as the main neural network for the digit recognition.