Report - Final Defense

Chaman

December 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			

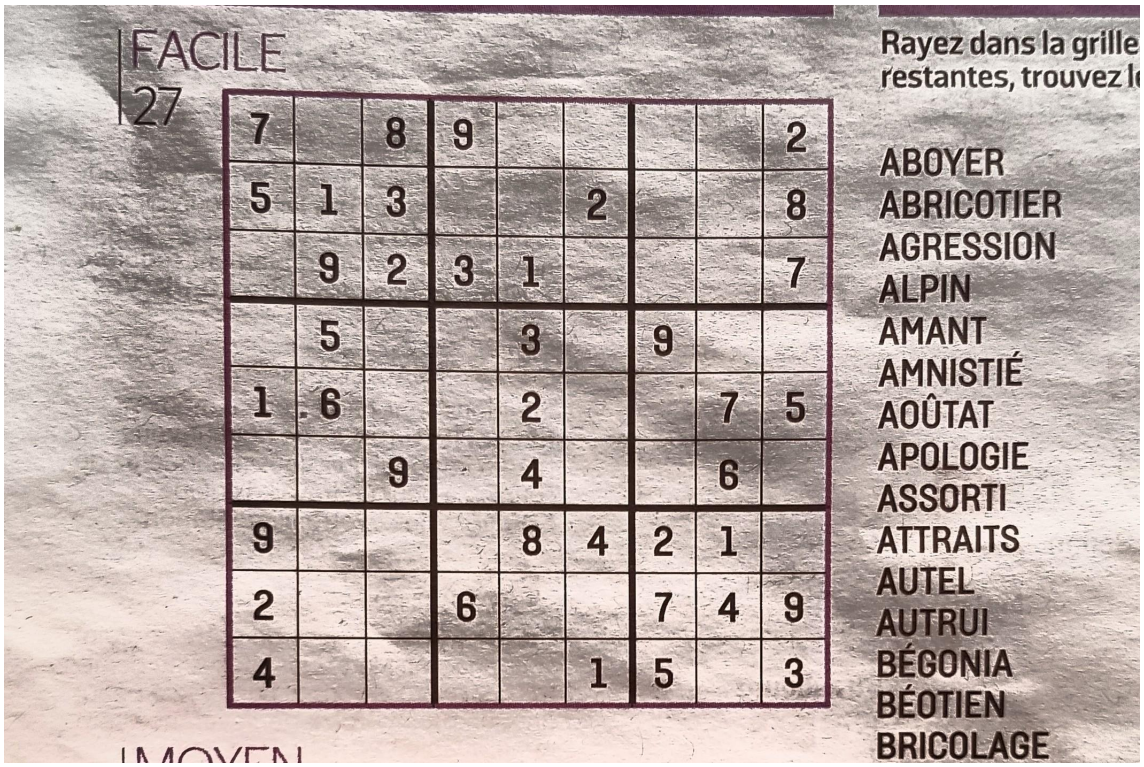
1.3 Progress of the Project

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	100%
Color Removal	100%	Reconstruction of the grid	100%
Pre-processing	100%	Grid resolution	100%
Grid detection	100%	Display of the solved grid	100%
Retrieval of the digits	100%	Saving the solved grid	100%
Character recognition	100%	Graphical User Interface	100%

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:

$$color = 0.4 * red + 0.35 * green + 0.25 * blue$$

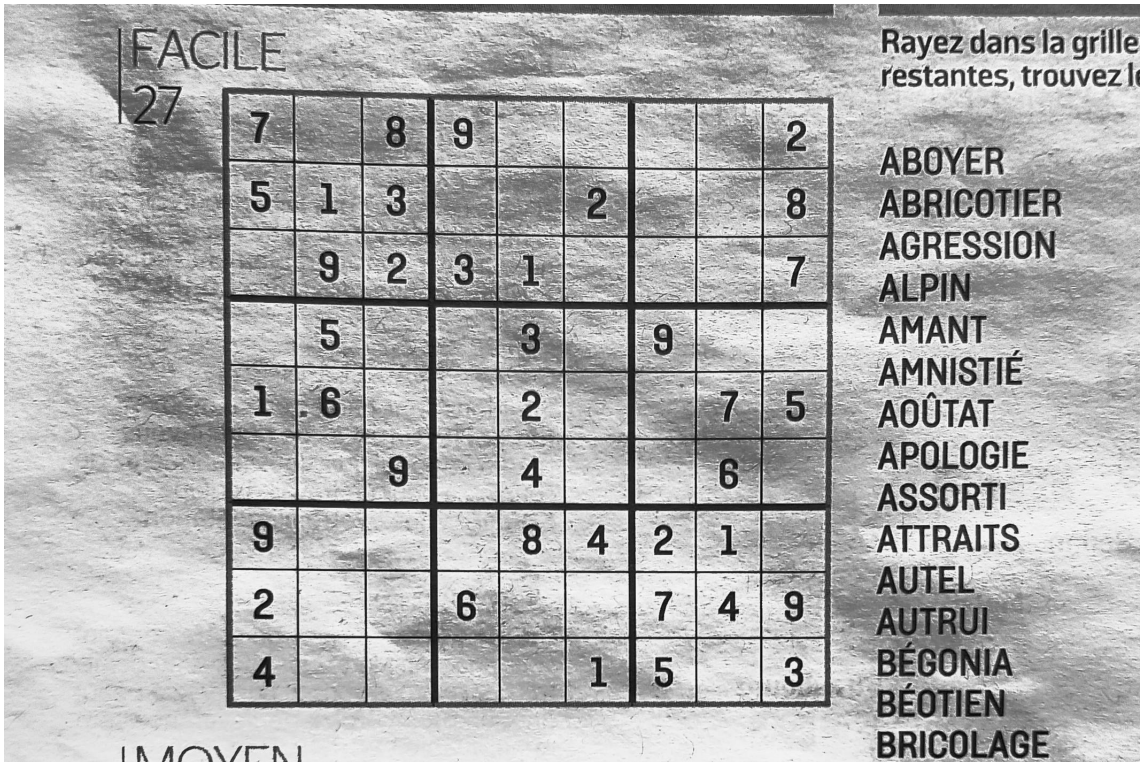


Image 4 - Grayscale Conversion

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:

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

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

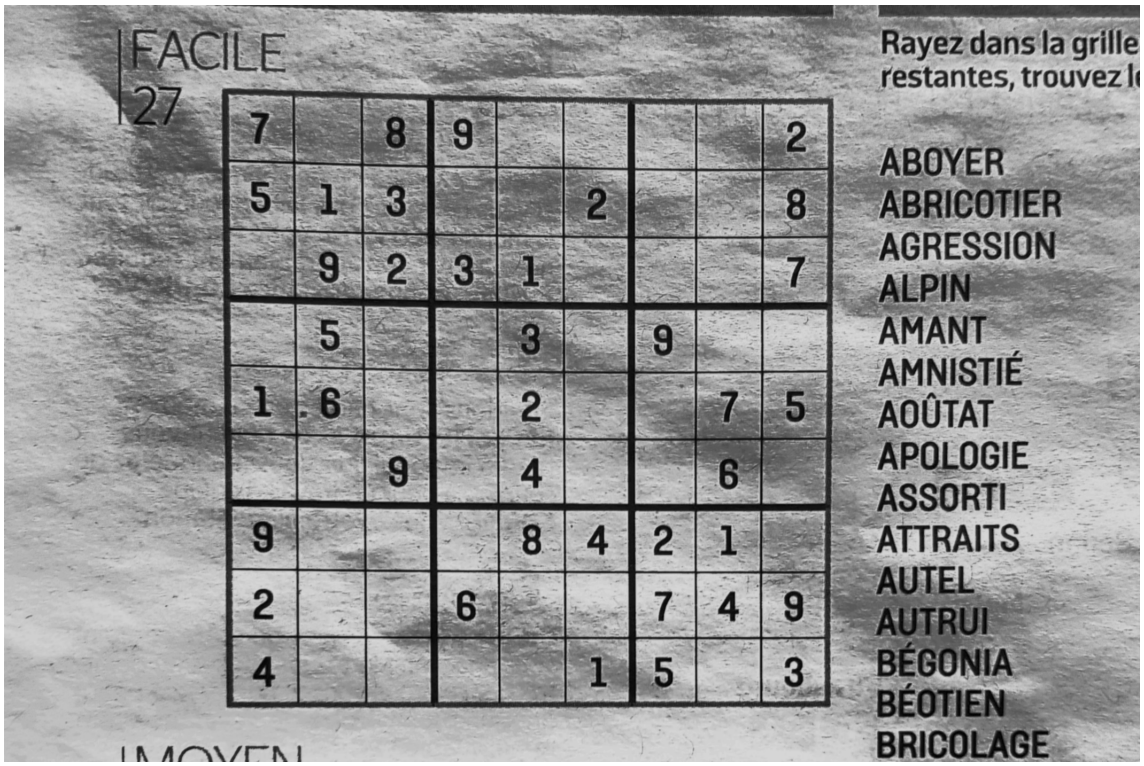


Image 4 - Gaussian Smoothing

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:

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

With $\gamma = 512$.

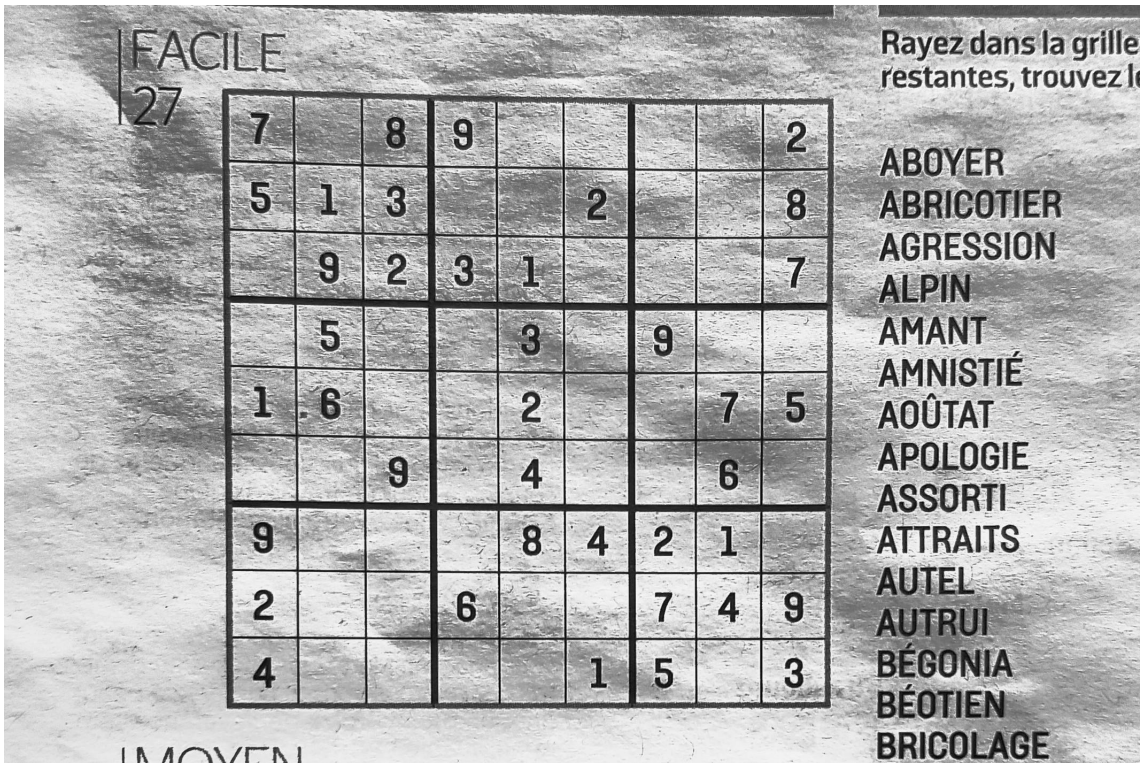


Image 4 - Gamma Adjustment

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.

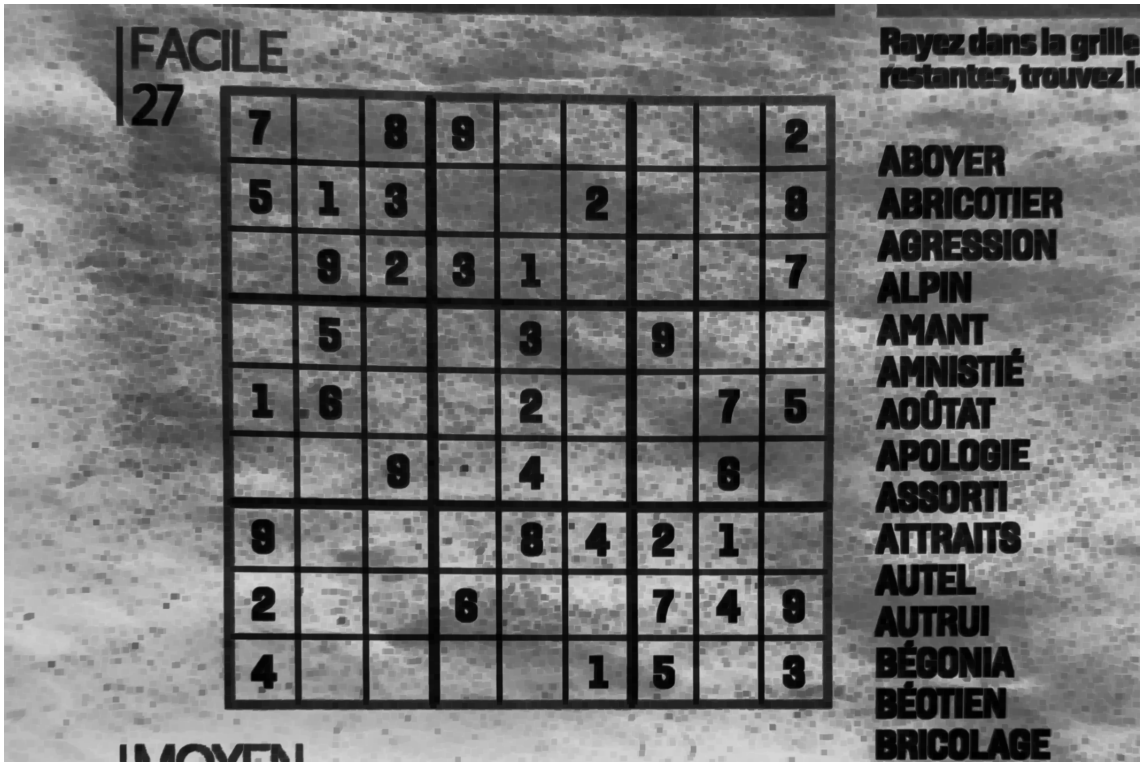


Image 4 - Dilation

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.

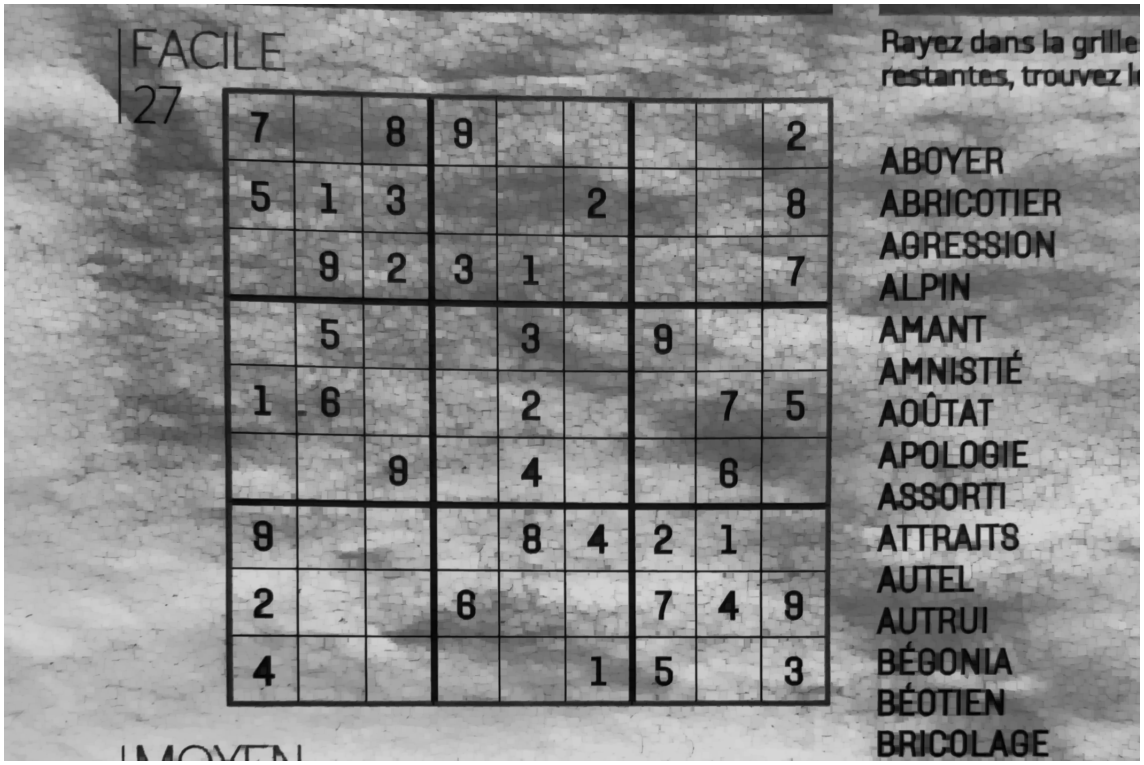


Image 4 - Erosion

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 Otsu's 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.

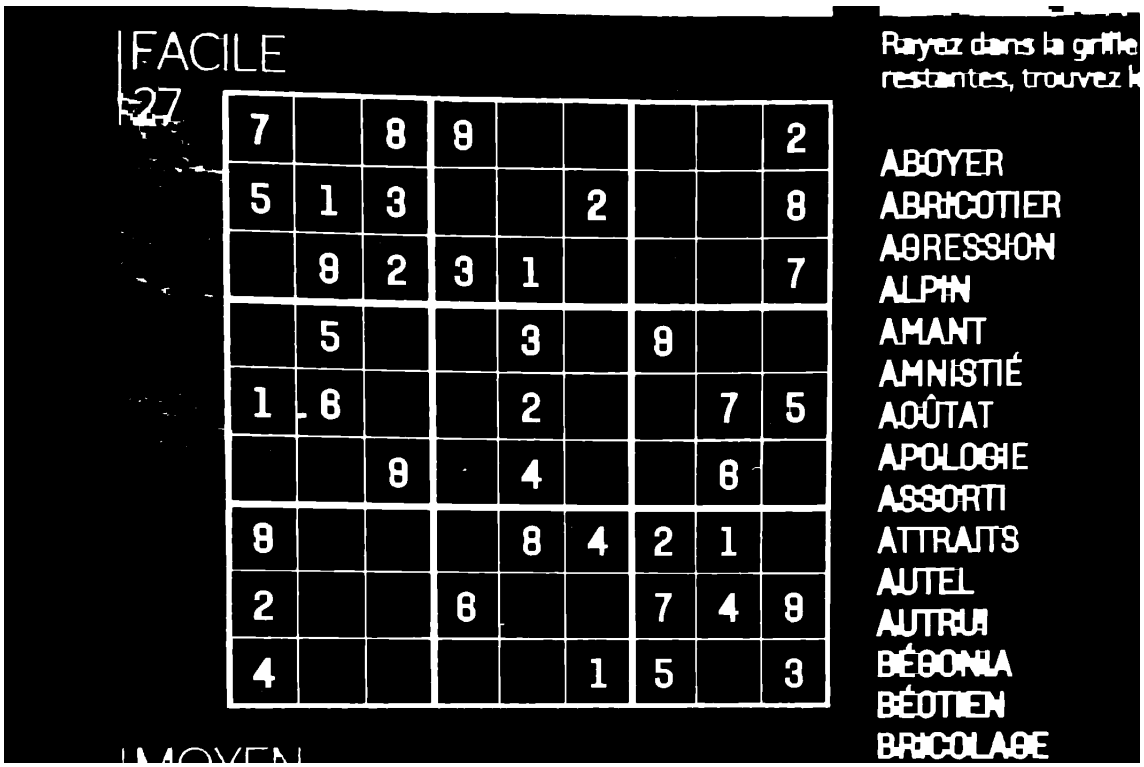


Image 4 - Otsu's Thresholding

As you can see, the result for simple images is very successful.

2.2.3 Sauvola Thresholding

We also implemented a Sauvola thresholding. This is an adaptive threshold used for images where the background is not even at all. It allows us to process more difficult images where the lightning condition is poor or the background is constitued of multiple colors. This filter computes the standard deviation and the local mean on the whole image, and then applies the following formula to get a threshold value for every pixel:

$$threshold_{i,j} = localMean_{i,j} * \left(1 + k * \frac{stdDeviation_{i,j}}{R-1}\right)$$

with $k \in [0.2, 0, 5]$ and $R = 128$

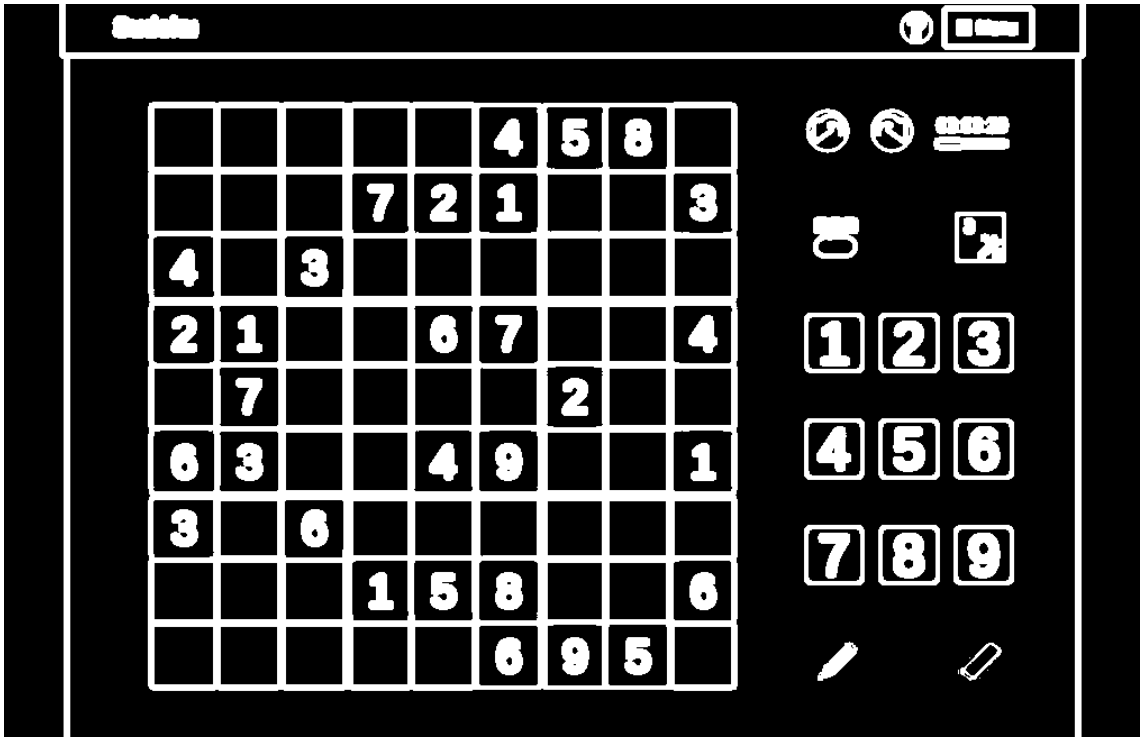


Image 4 - Sauvola Thresholding

As you can see, we were able to extract the lines and the digits of the image. The only drawback is that it gives bad results on noisy images. This is why we have to use a combination of the two thresholding algorithms.

2.3 Image Rotation

2.3.1 Manual Rotation

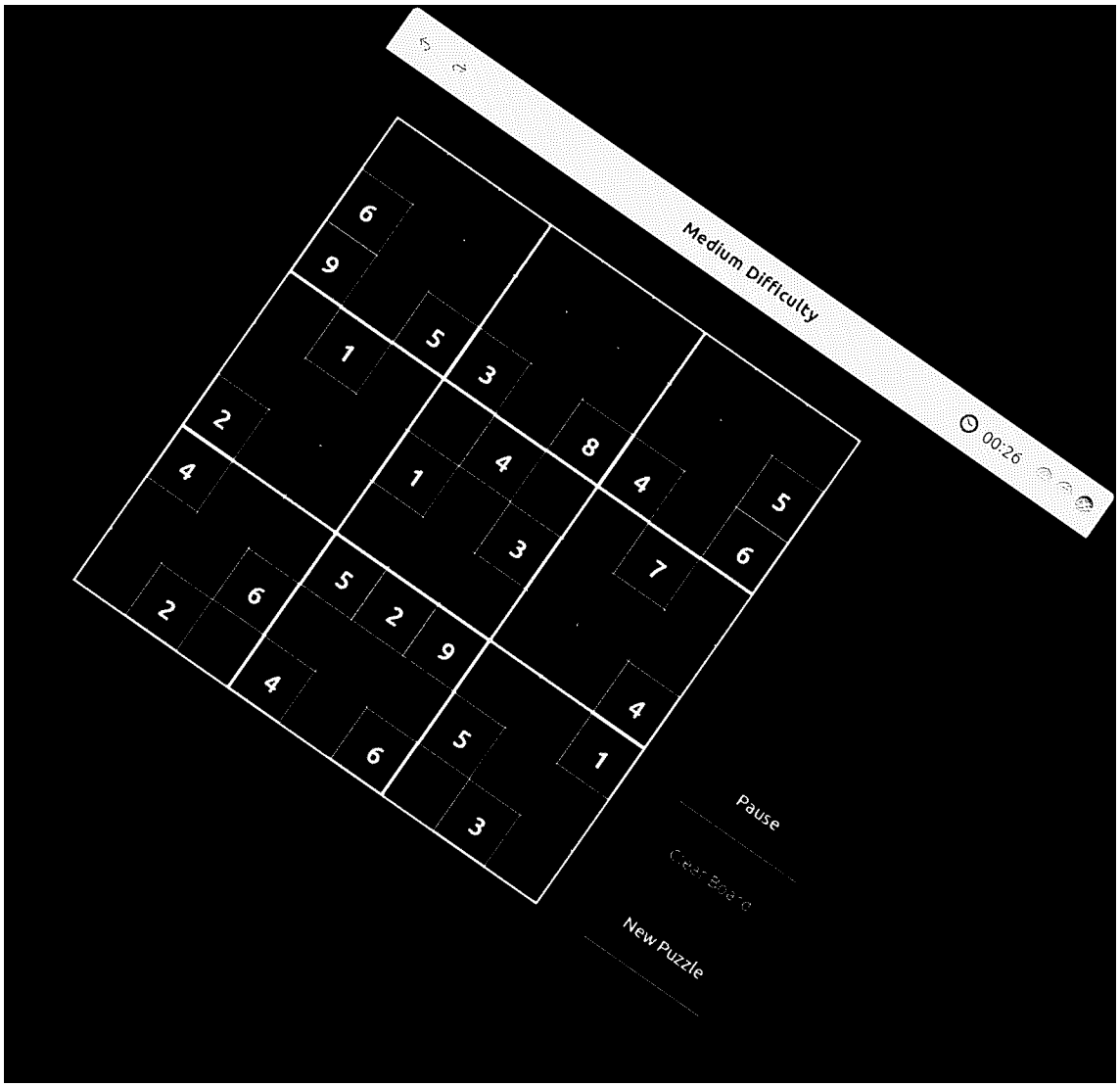


Image 5 - Before Rotating

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'):

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

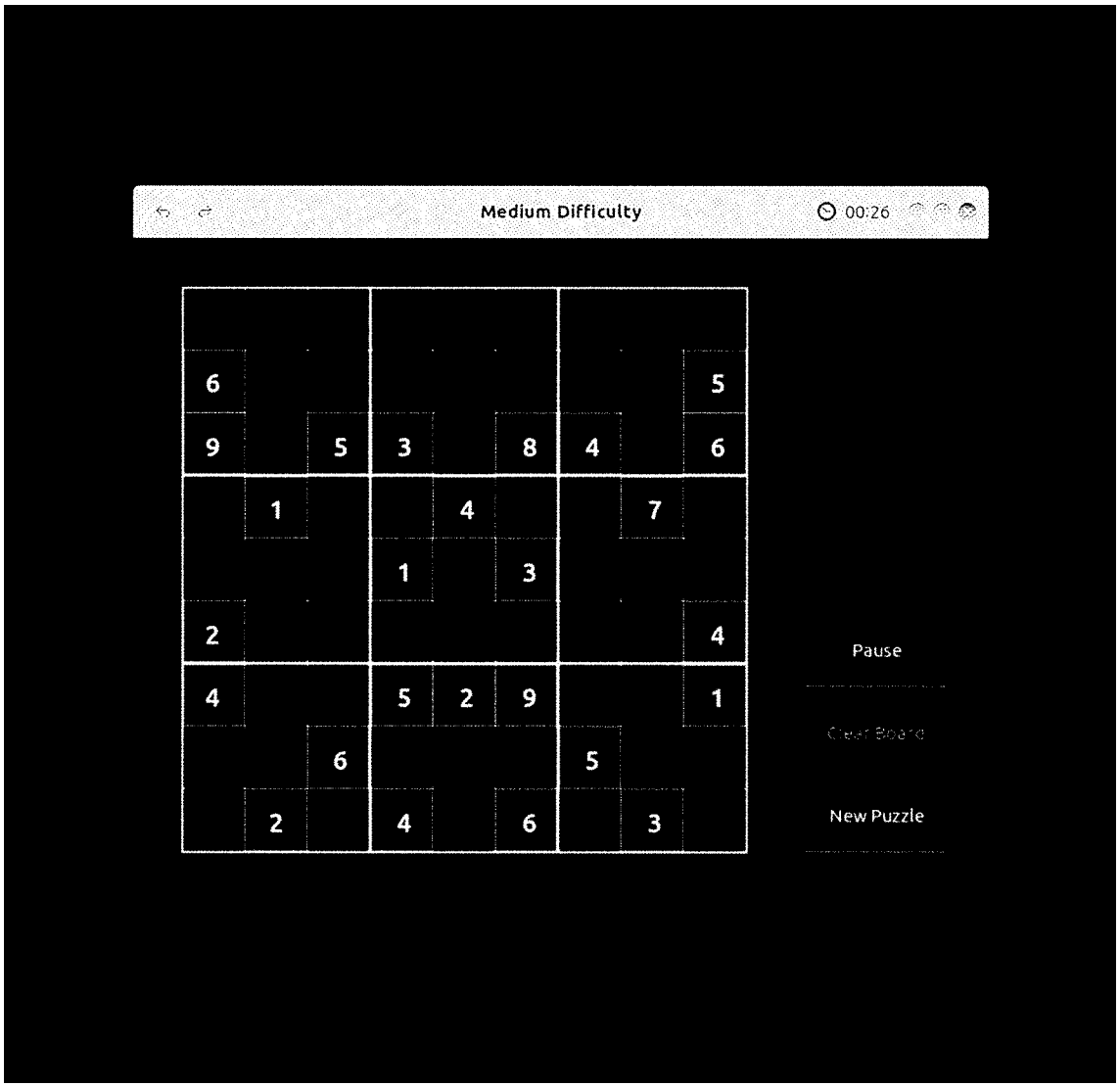


Image 5 - After Rotating by 35°

2.3.2 Automatic Rotation

For the automatic rotation, we use the data produced by the Hough Transform algorithm described in this paper. Please refer to it to learn about the accumulator we're about to talk about. We run trough the Hough Accumulator and count the votes for every theta angle found. If the most voted angle is above $|2\,\mathrm{deg}|$ and under $|45\,\mathrm{deg}|$, we apply a manual rotation with this angle. Otherwise, we run trough the accumulator again until we find a valid angle to rotate with, until a certain threshold.

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:

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

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:

$$G = \sqrt{(G_x)^2 + (G_y)^2} \quad (3)$$

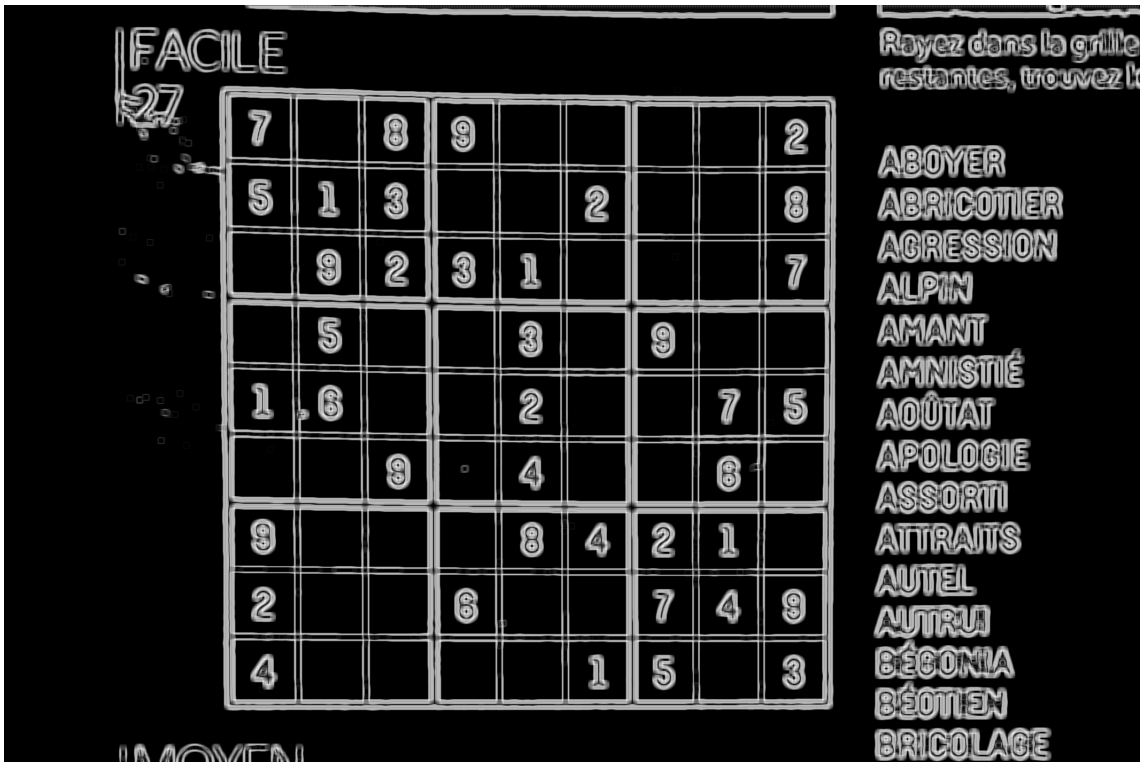


Image 4 - Magnitude Gradient using Sobel Filter

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.

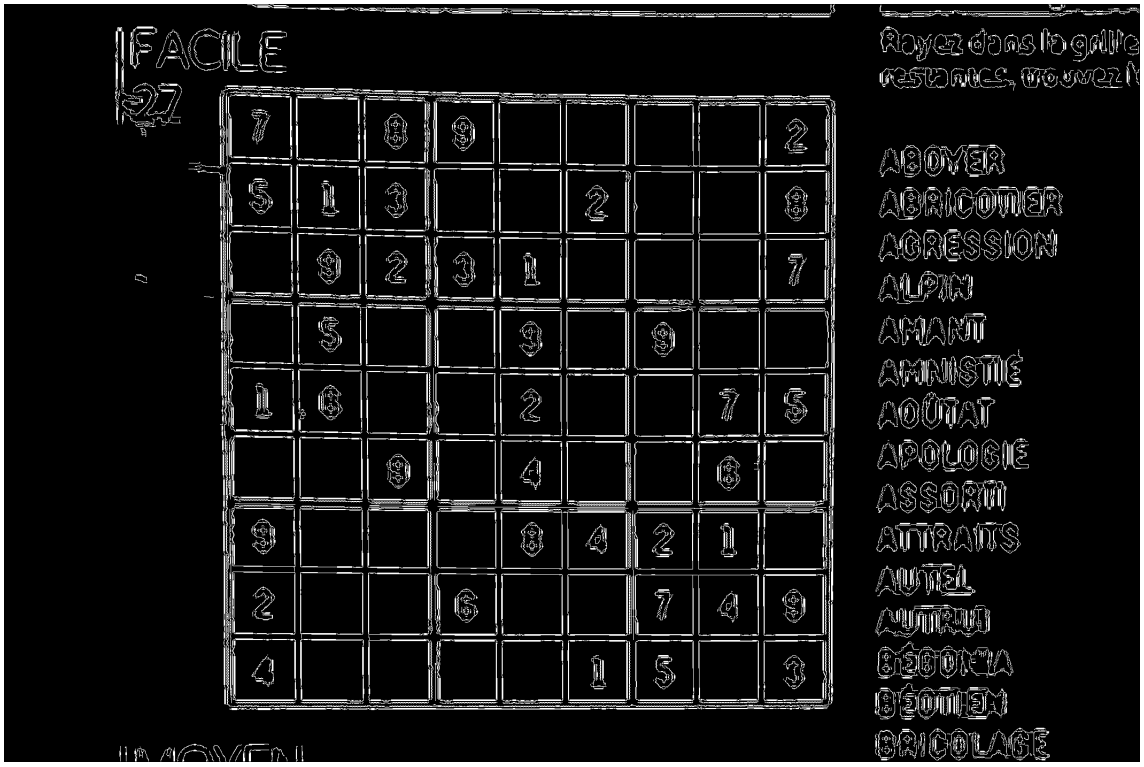


Image 4 - Canny Edge Filter

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:

$$y = ax + b$$

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:

$$\rho = x * cos(\theta) + y * sin(\theta)$$

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.

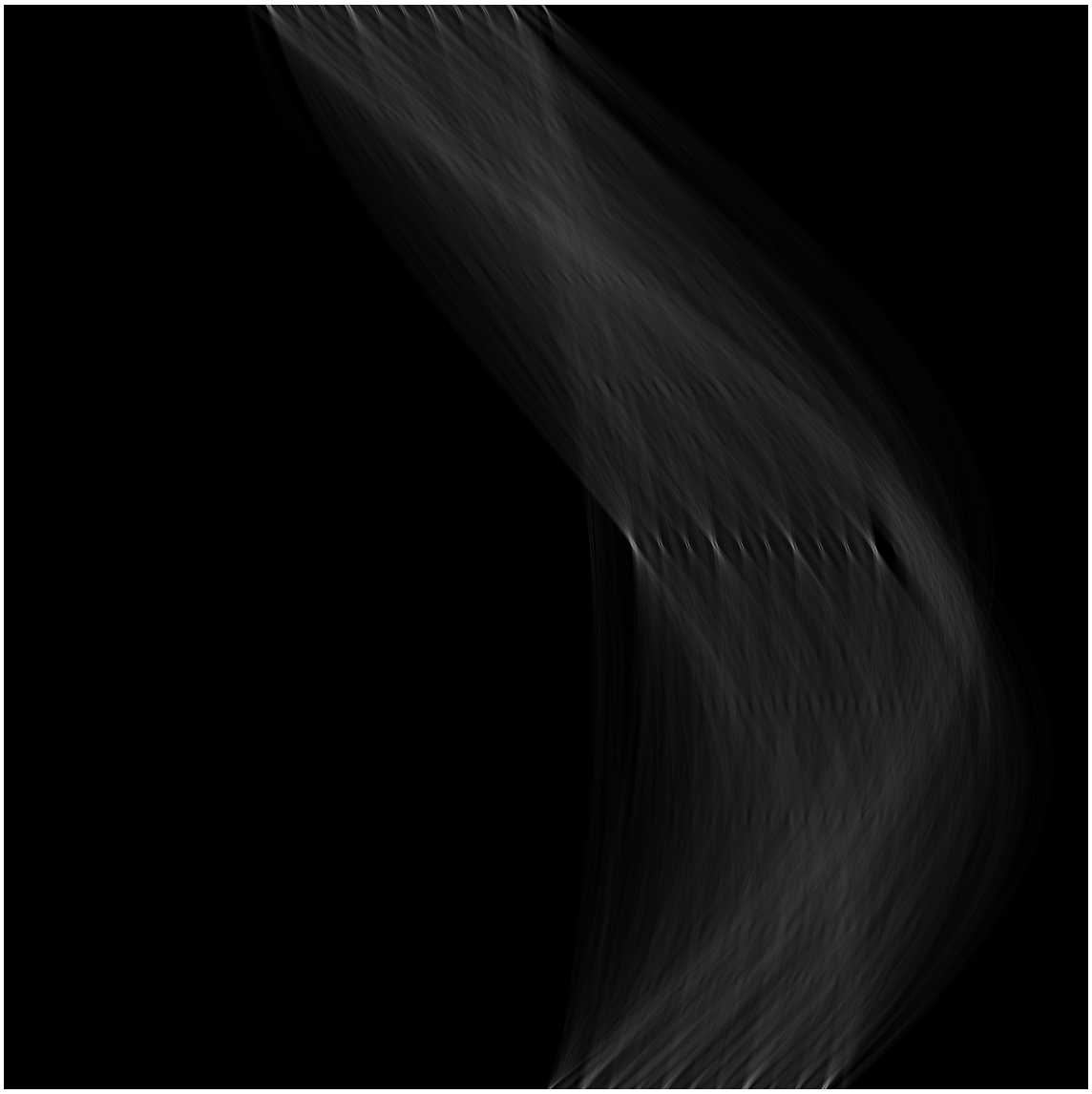


Image 4 - Votes of the Hough Space's accumulator plotted

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:

$$x0 = \rho * \cos\theta$$

$$y0 = \rho * \sin\theta$$

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

$$x1 = x0 + d * (-\sin\theta)$$

$$y1 = x0 + d * \cos\theta$$

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.

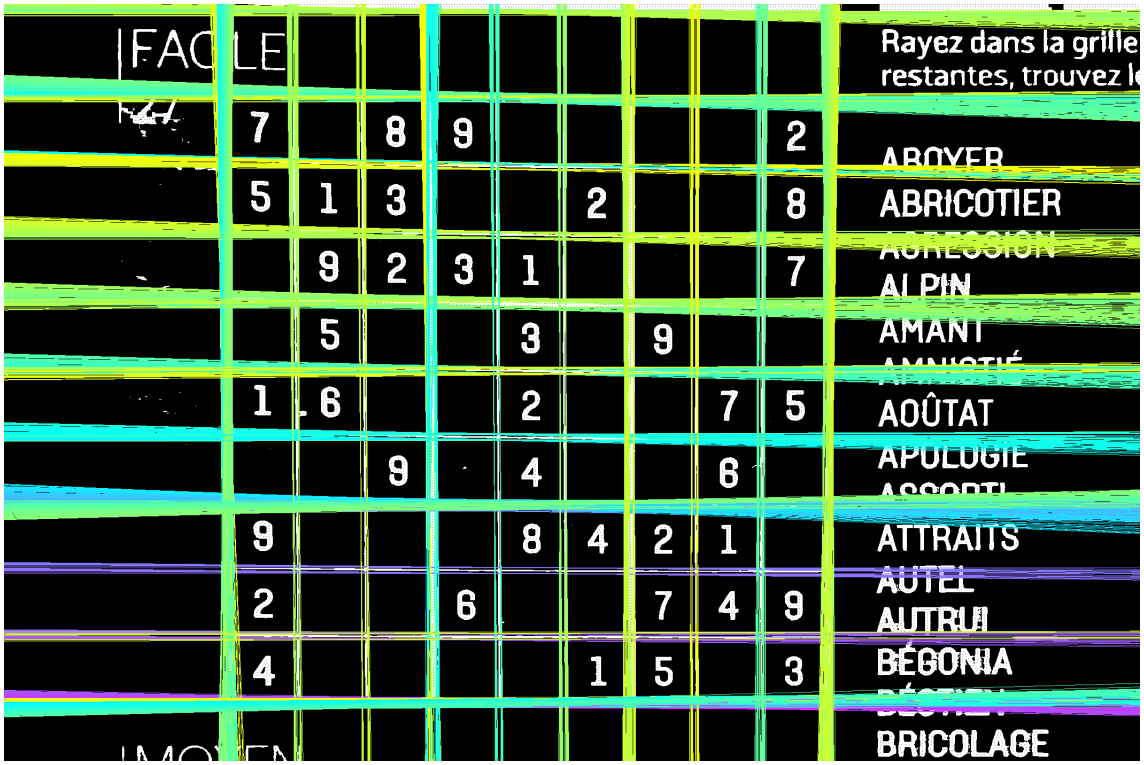


Image 4 - Hough Transform lines converted to Cartesian coordinates and drawn on the image

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.

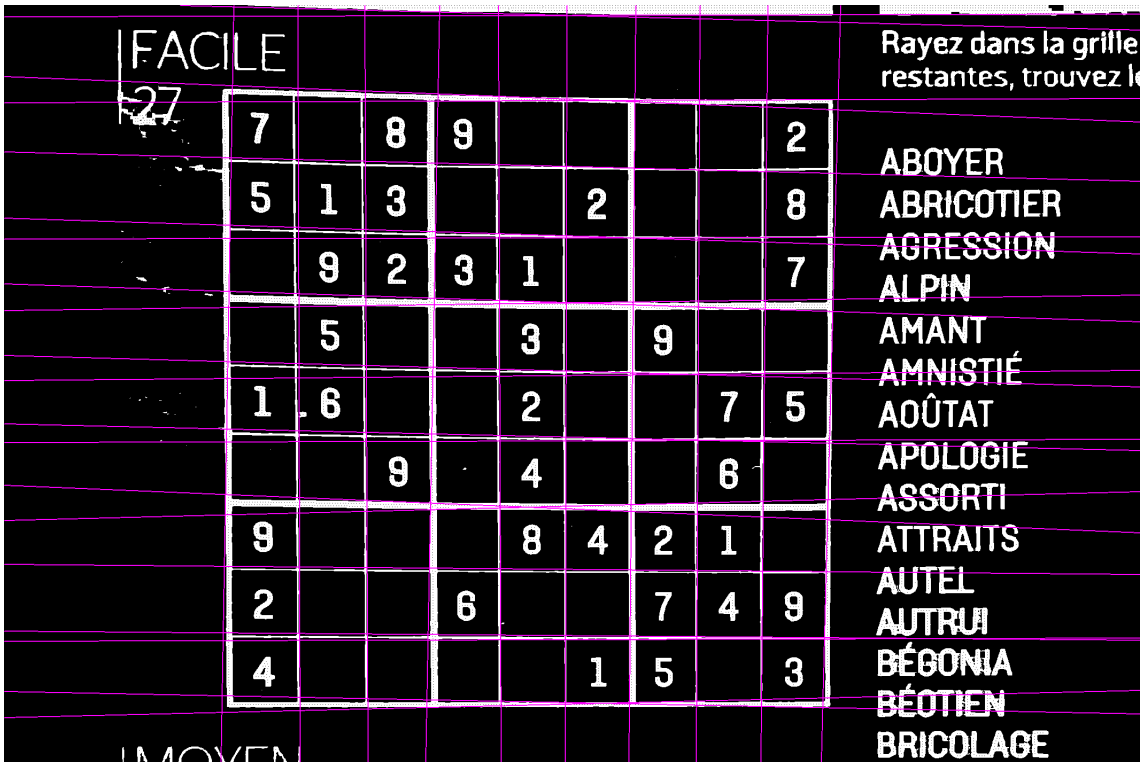


Image 4 - Reduction of the number of edges detected by averaging

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.

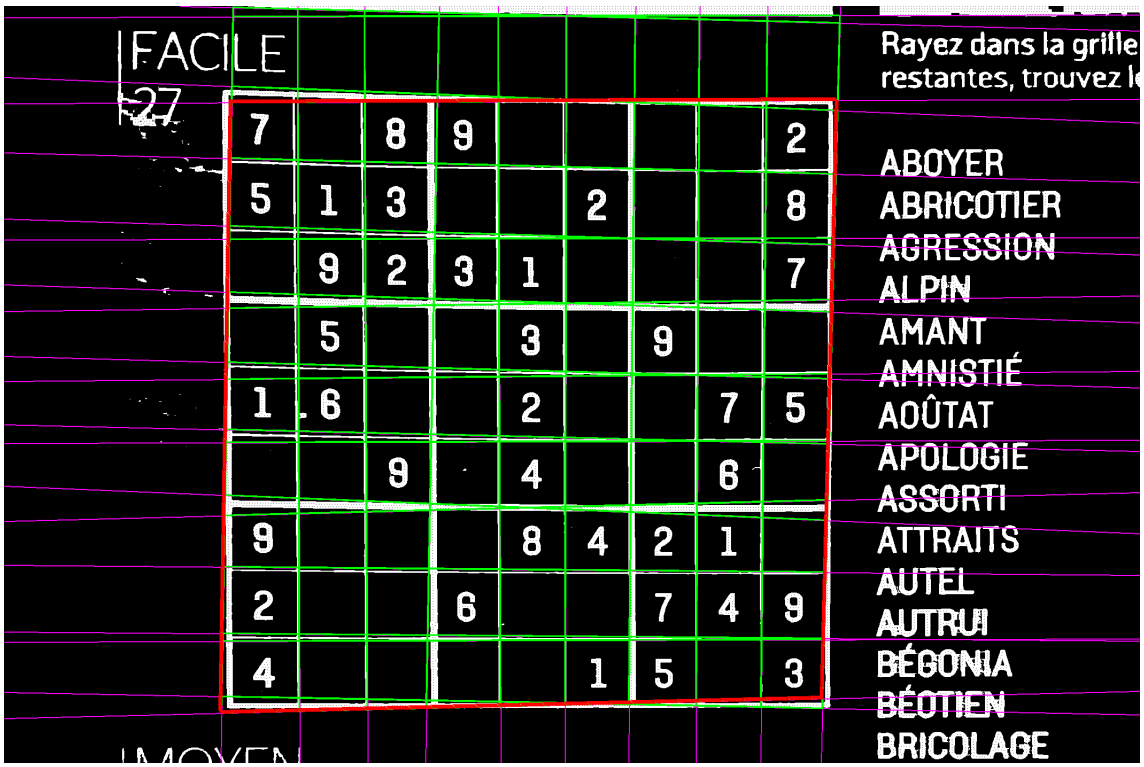


Image 4 - Square detection and selection

3.3 Perspective Transform

For some images, we needed to correct their perspective in order to be able to perform the rest of the image processing correctly. We achieved that by computing a homography matrix.
We start by computing four correspondence points tuples corresponding to the coordinates of the grid's corner before and after the perspective adjust.
With our 4 points $(p1, p'1), (p2, p'2), (p3, p'3), (p4, p'4)$, we rewrite them as 2 x 9 matrixes of the following form:

$$p_i = \begin{bmatrix} -x_i & -y_i & -i & 0 & 0 & 0 & x_i x' i & y_i x'_i & x'_i \\ 0 & 0 & 0 & -x_i & -y_i & -i & x_i x' i & y_i y'_i & y'_i \end{bmatrix}$$

We then stack them in a matrix P to compute $PH = 0$ and solve the equation leading to the relation between the coordinates before and after the perspective transform.
However, to avoid the obvious solution for all factors to be 0, we add a last row containing only 0 except the last element which is equal to one, and we set the last element of our 0 matrix to be one. We therefore get the following equation:

$$PH = R \Leftrightarrow \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 x'_1 & y_1 x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1 x'_1 & y_1 y'_1 & y'_1 \\ -x_2 & -y_2 & -2 & 0 & 0 & 0 & x_2 x'_2 & y_2 x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -2 & x_2 x'_2 & y_2 y'_2 & y'_2 \\ -x_3 & -y_3 & -3 & 0 & 0 & 0 & x_3 x'_3 & y_3 x'_3 & x'_3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -3 & x_3 x'_3 & y_3 y'_3 & y'_3 \\ -x_4 & -y_4 & -4 & 0 & 0 & 0 & x_4 x'_4 & y_4 x'_4 & x'_4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -4 & x_4 x'_4 & y_4 y'_4 & y'_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

This can then be solved with the following equation:

$$H = P^{-1} x R$$

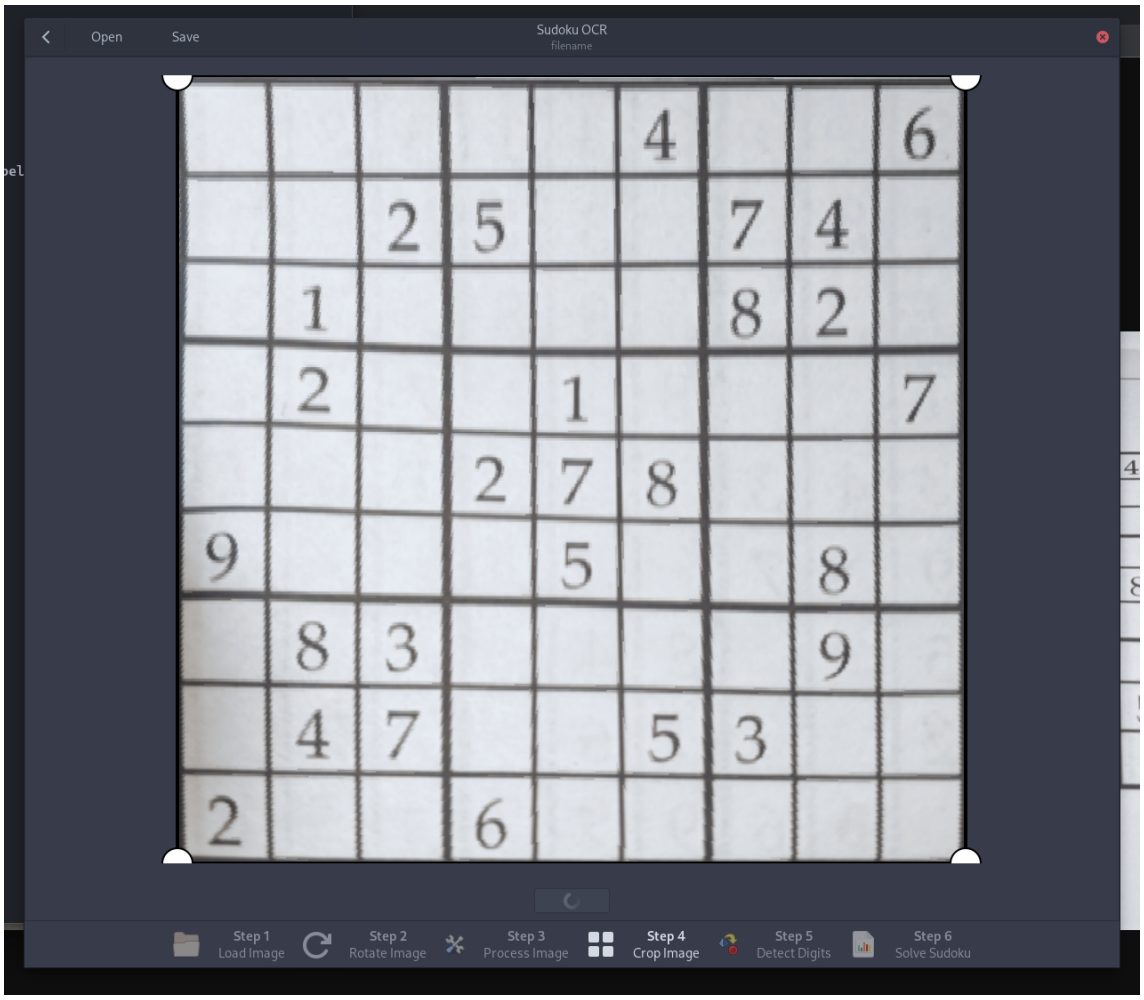
We then lay out H like so to get a transformation matrix:

$$\begin{bmatrix} h1 & h2 & h3 \\ h4 & h5 & h6 \\ h7 & h8 & h9 \end{bmatrix}$$

We then invert this matrix and compute the coordinates on the input image (x', y') for every pixel of the output image (x, y).

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h1 & h2 & h3 \\ h4 & h5 & h6 \\ h7 & h8 & h9 \end{bmatrix}^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Perspective Transform - Input Square



Perspective Transform - Result

3.4 Digits Extraction

After the correction of the perspective of the image, the digits needs to be extracted for the network to recognise them. We start by splitting the image 9 times by its height and 9 times by its width. When this is done, we obtain digits like these.

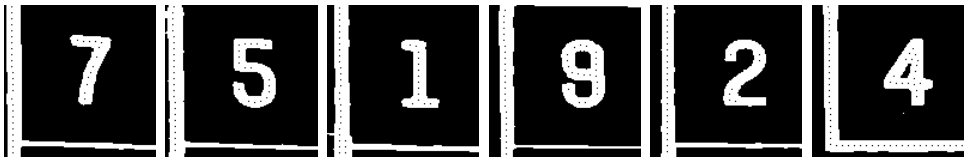


Image 4 - First column splitted in cells

The next step is to clean the borders of the image to extract only the digit. This is done by progressively looking for the most extreme coordinates of the white parts of the image. We start to look for these in a small square located at the center of the image, and we gradually increase the size of this square until the most extreme coordinate is not evolving. When this point of the process is reached, the square is only containing the whole digit and no other unwanted parts of the cell. The digits then look like these.

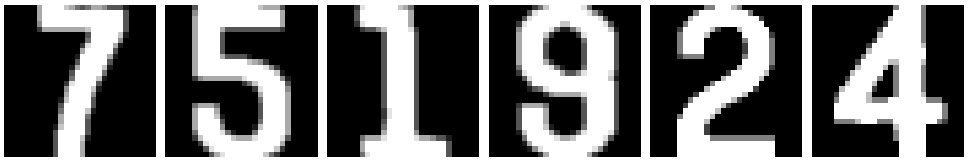


Image 4 - First column splitted in cells and extracted digits

4 Sudoku Solver

4.1 Sudoku solving

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 :

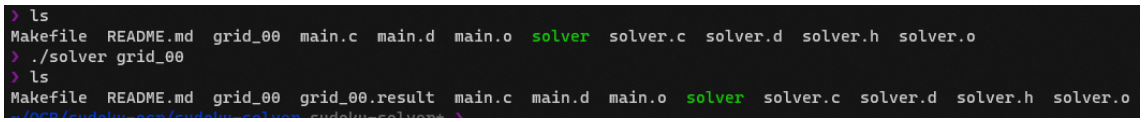
- For every empty cells in the Sudoku's grid, a number between 1 and 9 is assigned.

- 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.

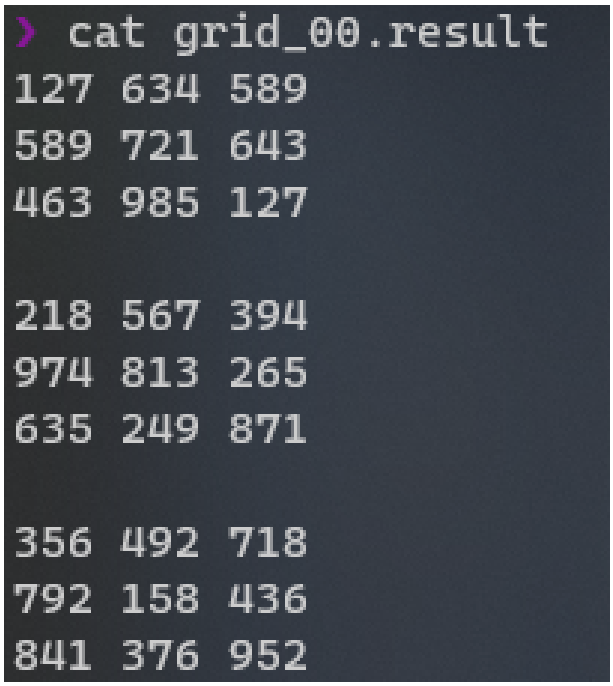
As specified in the book of specification, the Sudoku's solver algorithm is an application that runs from the command line only. The solver takes as input a file in which an incomplete grid is saved.



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.

4.2 Grid Reconstruction

In order to reconstruct the solved grid, we opted for a method which is very classical. We first created sample of number images as well as a sample of an empty Sudoku grid. We chose to have an 800x800 pixels image for the empty grid and 90x90 pixels images for each digits.

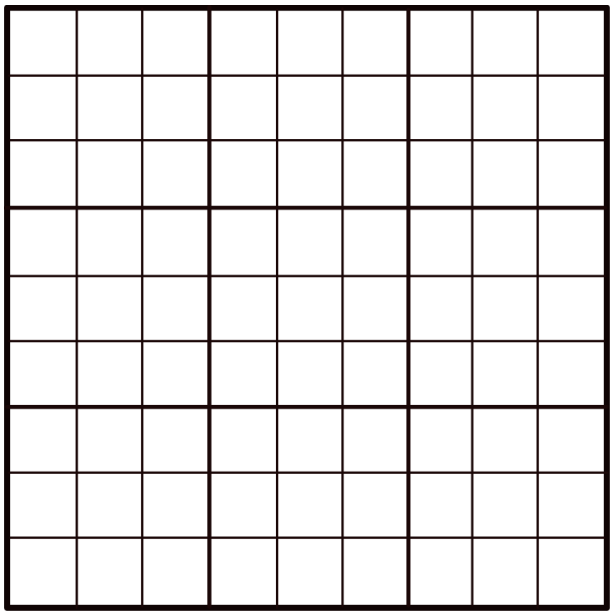


Image of the empty Sudoku's grid

The method used here consists on traversing the image of the empty grid and replacing each cells with the image of the corresponding number in the solved grid. To do so, we use two functions :
The first one replace all the pixels of an image by the ones of another image at a specific coordinates on the image that needs to be modified.
The second one allows you to browse the resolved and unresolved grid and change the color of the number on the image of the resolved grid if it was not in the initial grid.

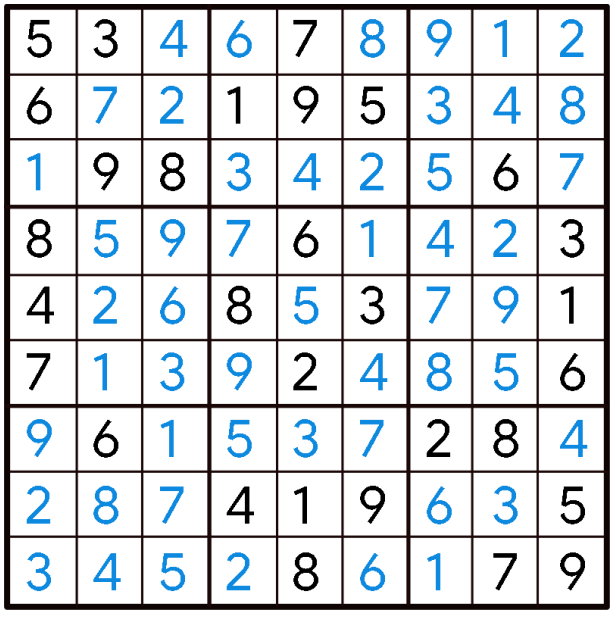


Image of the Solved Grid

5 Neural Network

5.1 XOR

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.

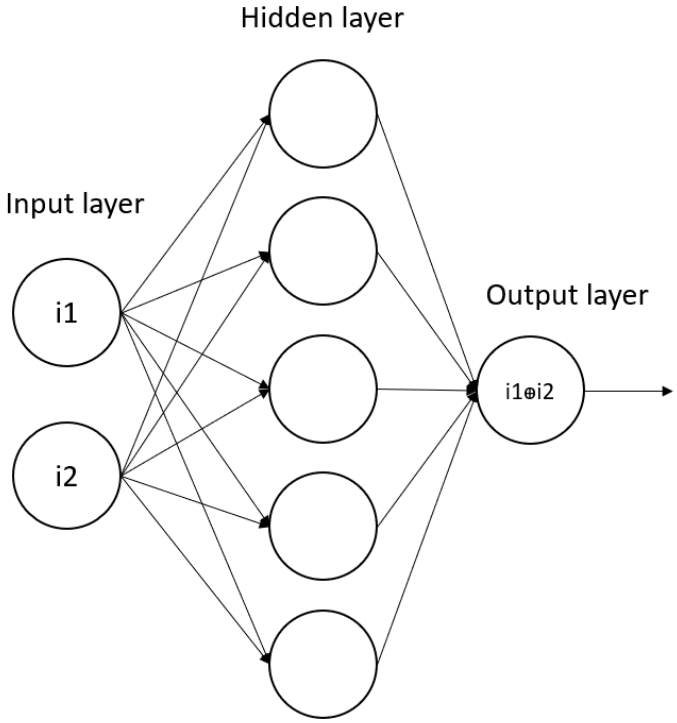
Then the practical part began. After reading the code provided in e-book, thinking of a way to imitate the Numpy library was the main issue. Creating a structure called Matrix seemed to be the best answer. In this structure are defined a lot properties on matrix such as multiplication, transpose for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

5.2 Digits Recognition

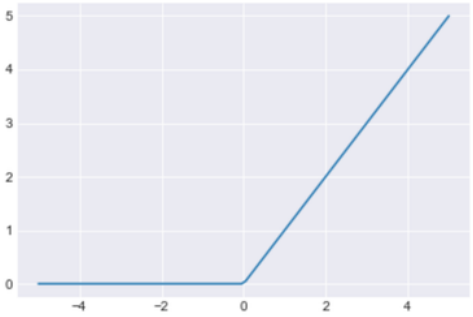
5.2.1 Training Set

To train the network we needed a large training set with multiple fonts for printed digits. So we coded a python function that generates more than 3,000 images using google fonts. With this training set the training will be difficult but will well train the network.

5.2.2 The network

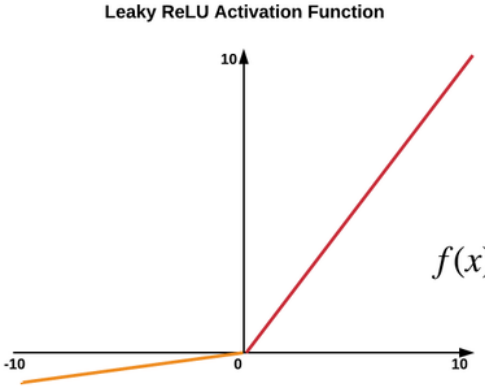
First we had to adapt the network structure to use a picture as input. Pictures are 28x28 pixels sized so the new input of the network is a matrix of size 1x784. Since all calculation done on the forward pass and back propagation are done taking into account constant parameter of the network we didn't had to change this part for now. Then the number of outputs is now 10. And of course we have done a function to load the weights and biases from a file in order to execute the network without training it before. The adaption done we first tested the network but results were very bad. So we changed the activation functions and the loss function. First the activation function of the hidden layer. Initially we choose ReLu (rectified linear activation unit) which equation is:

$$relu(x) = \begin{cases} 0 & \text{if x} < 0 \\ x & \text{if x} \geq 0 \end{cases}$$



ReLu activation function

But after some test we decided to change to another version of ReLu which is the leaky ReLu. It gives better results. Leaky ReLu equation is:



$$f(x) = \begin{cases} 0.01x & \text{for x} < 0 \\ x & \text{for x} \geq 0 \end{cases}$$

Leaky ReLu activation function

Next for the output layer we choose the softmax activation function which is very simple to implement with matrix. It distributes the value between 0 and 1 and the sum of all outputs is 1. Here is its equation:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Softmax activation function

After these changes we needed a new back propagation so we used a back propagation with the cross entropy loss function which is perfect with softmax activation function for the derivative. We also coded a function to shuffle the training set at each epoch.
Now that the network structure was done we just had find the best parameters to train the network. There are two main choices a large number of hidden node and a small number of epochs or the opposite. With our network the second choice leads to better results. In around 700 epochs and 120 hidden neurons we reached an accuracy of 90% in the training set.

6 Graphical User Interface

Our goal when designing the GUI was to give the user a simple and straightforward experience. To to this, we've chosen to have a layout with multiple pages, a simple flow and a minimal number of buttons. We've implemented a step indicator so the user isn't lost during the wait time.

6.1 Technical Details

6.1.1 Dynamic Image Resizing

A problem we found when designing the interface is that the out-of-the-box GtkImages wouldn't scale at all to the size of the window. We resolved this problem by using a GtkDrawingArea instead of a GtkImage. This method allowed us to query the available size for each image on the draw events and rescale the image properly using pixbufs. The process is straightforward. We start by querying the available size. We then convert this size to the aspect ratio of the image that needs to be displayed. Then, we convert the image to a pixbuf and we rescale this pixbuf to the computed size. We then draw the content of the pixbuf at coordinates where the image will be centered.

6.2 Live Square Selection

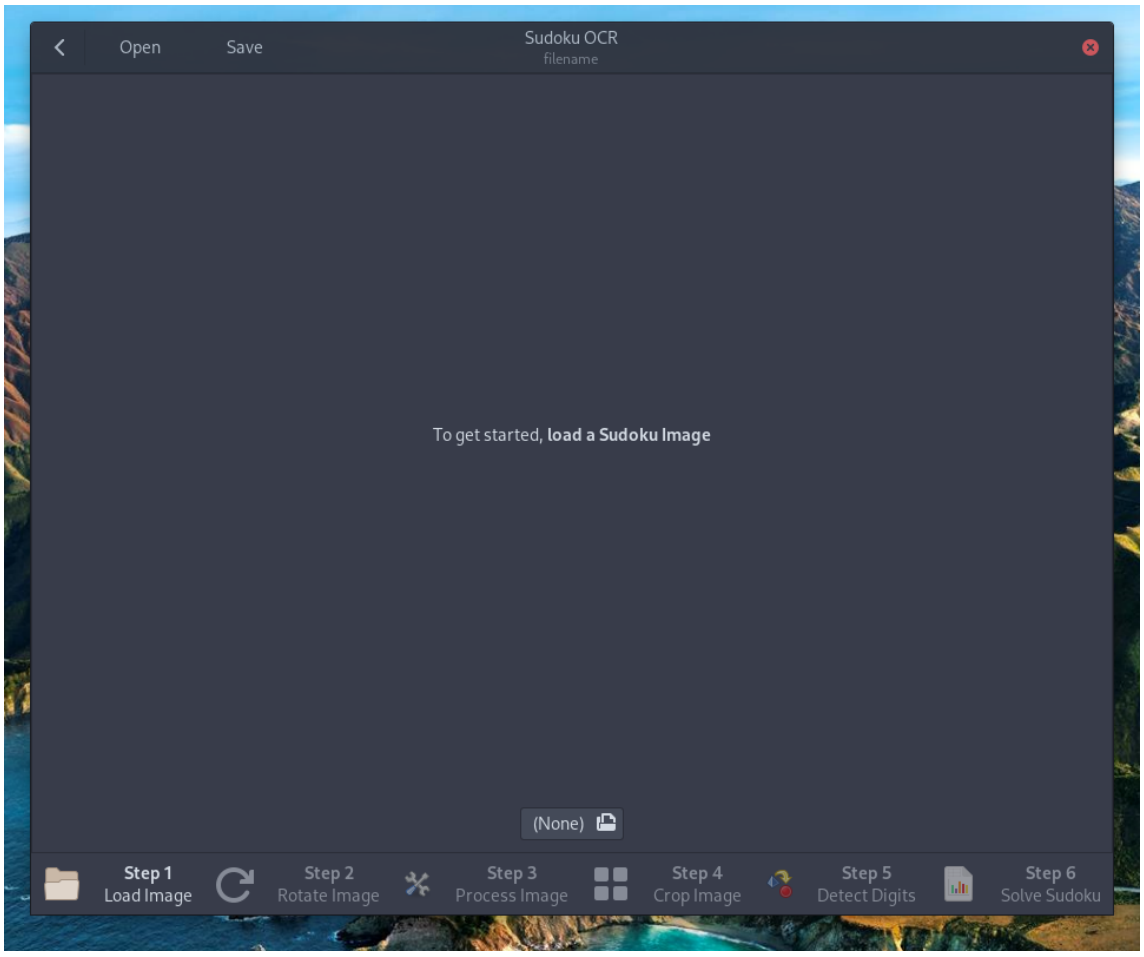
To select the square as described in step 4, we had to design a custom widget. To do that, we used a GtkDrawingArea and an EventBox. The widget starts by drawing the image as described in the previous paragraph. We then scale the square using the same

technique. After, we draw the lines of the square and the handles at the corners. We then use the GtkEventBox to detect the mouse position and make the nearest handle follow it when the user is "dragging" it.
When the user is done adjusting the square and presses the confirm button, we get the coordinates of the scaled square and convert them back to the original scale of the image using the inverse procedure from before.

6.3 User Flow Steps

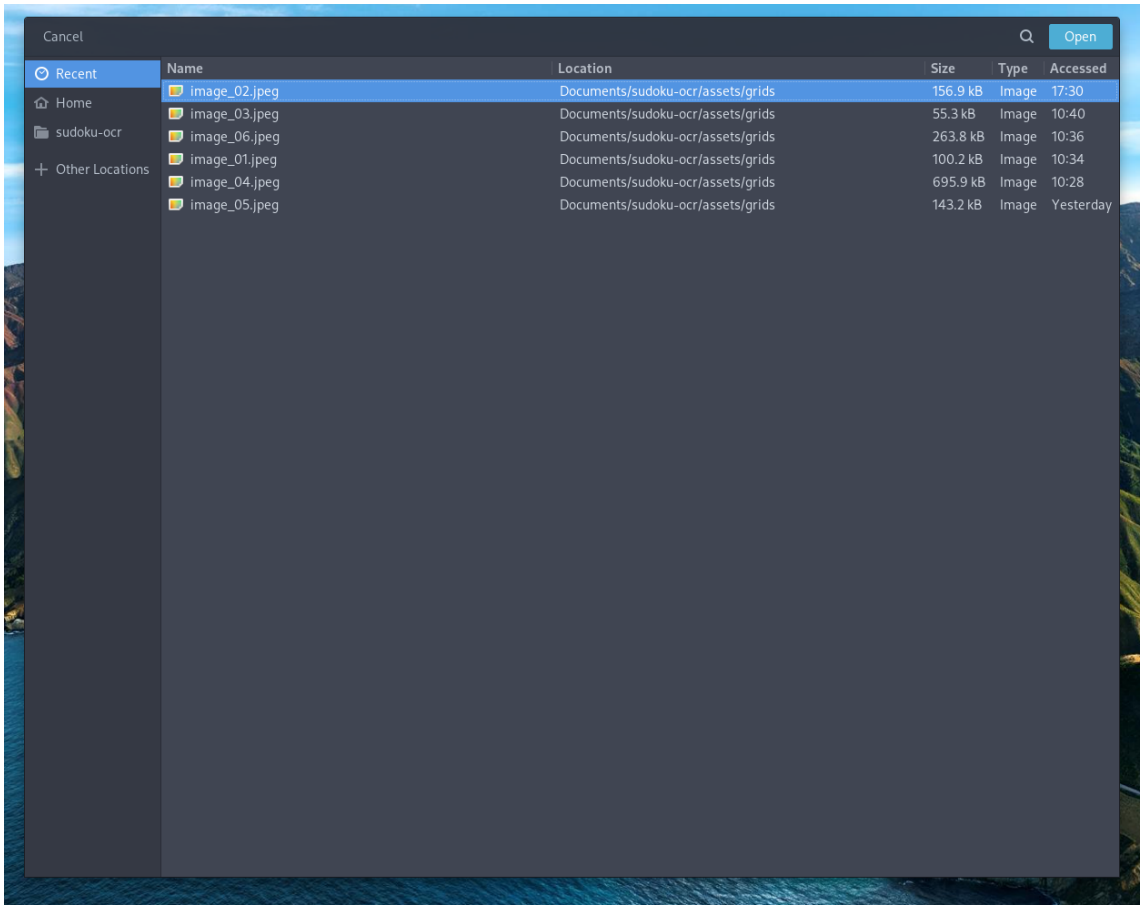
6.3.1 Step 1 - Image Loading

The first step of the process is loading the image. There is two options to do that. You can either press the open button on the top left, or press the button located at the bottom of the page.



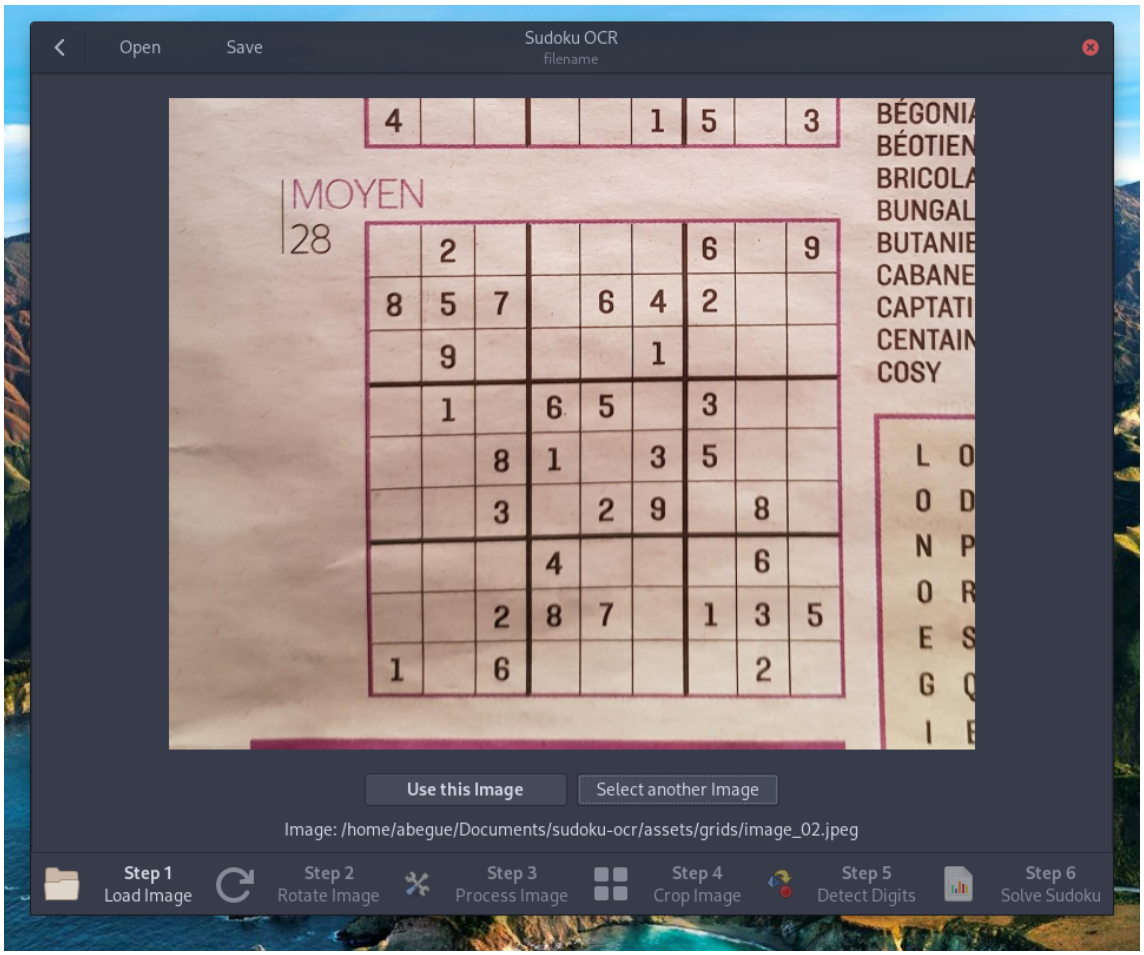
Start Page

In either cases, this will open a FileChooser widget which will allow the user to select a file.



File Chooser

After the file is chosen, a preview will be displayed with the option to choose another one. Note that at any point in the process, the user can choose to restart the process with another image by pressing the Open button in the top left corner.



File Confirmation

6.3.2 Step 2 - Image Rotation

After confirming the chosen image, the user will be asked to rotate the image. While an automatic rotation is implemented, we need the user to put the digits of the grid in the right orientation.

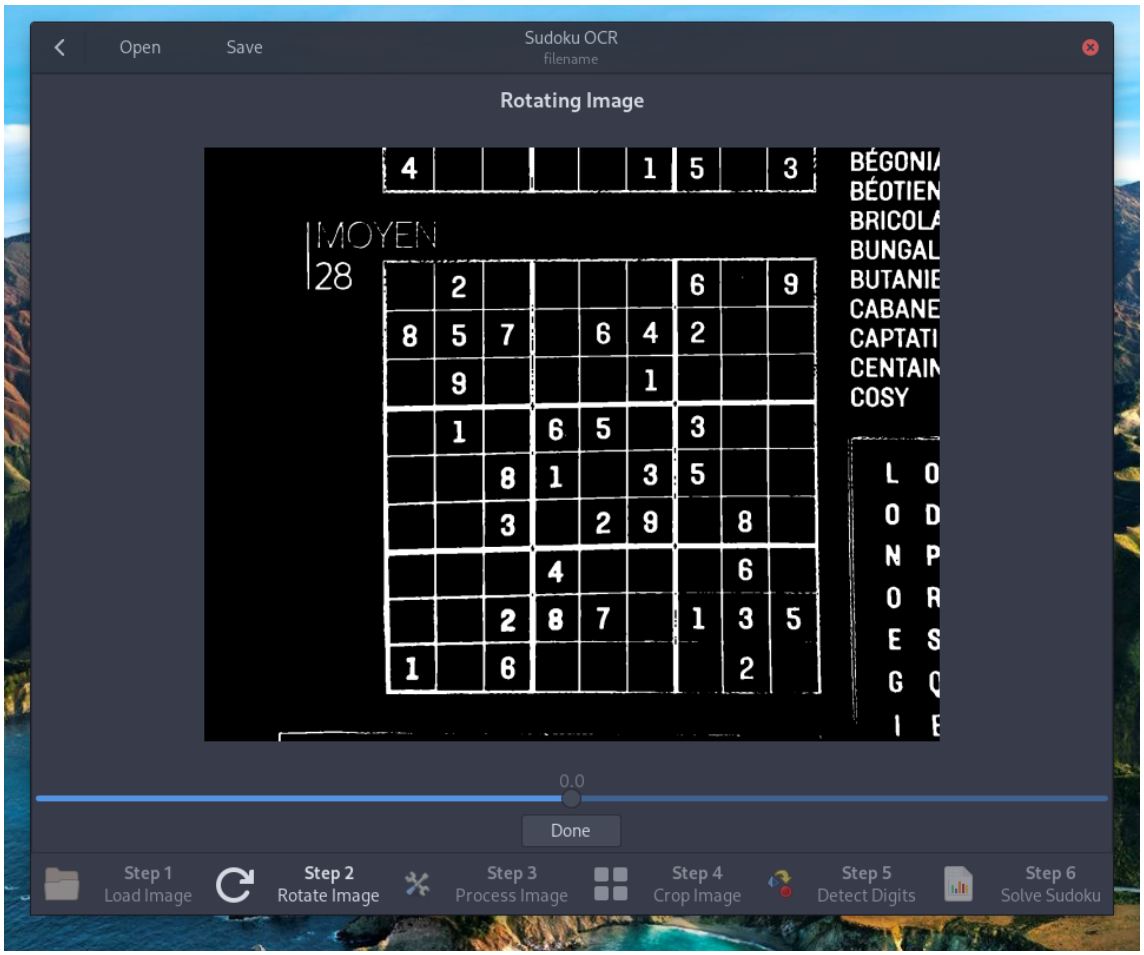
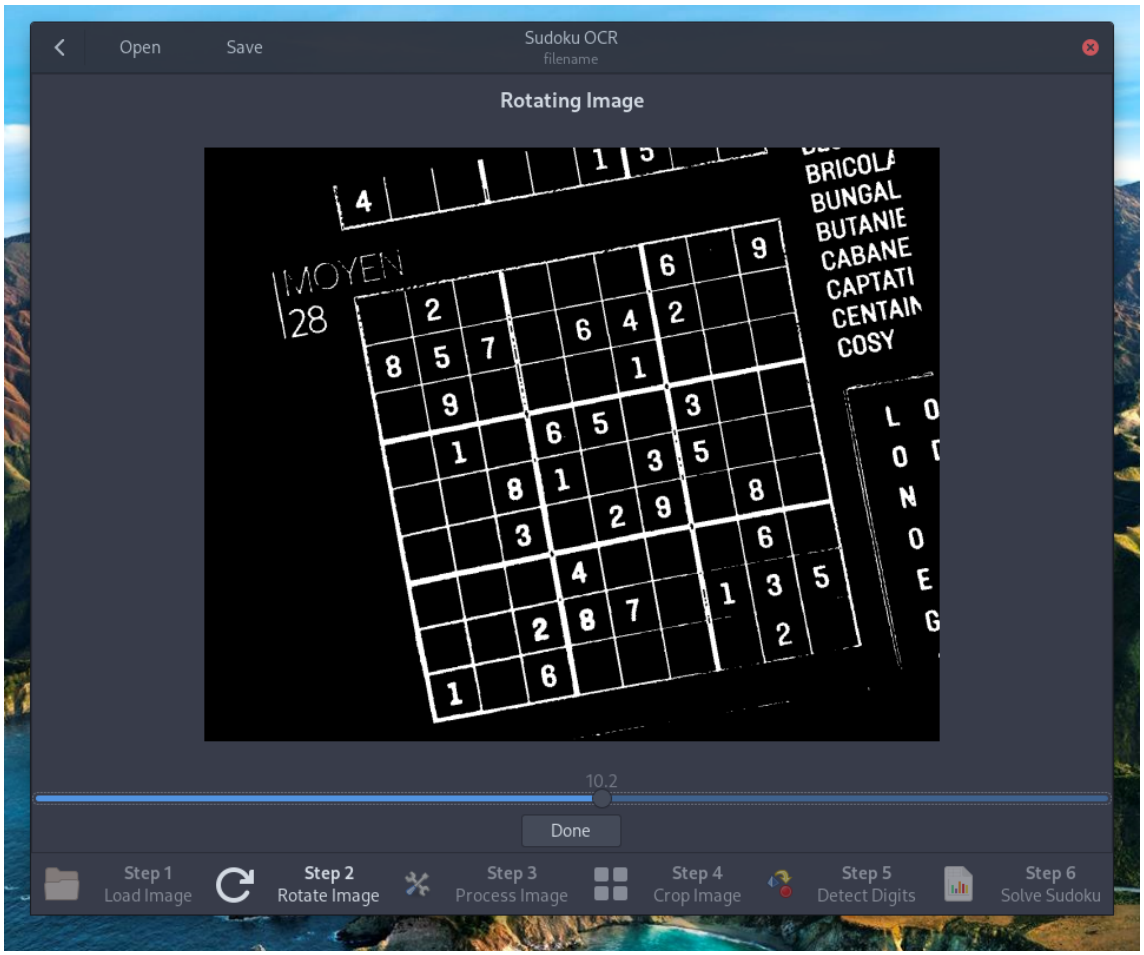


Image Rotation

The slider on the bottom of the page can be used to dynamically tweak the rotation amount. The interface provides a live preview of the rotation as the user moves the slider.



Example After Rotation

6.3.3 Step 3 - Image Processing

The image processing happens between each step of the process to get the next image needed for the steps. The part between the step two and four is the longest one because this is when the lines and the grid square are detected.

6.3.4 Step 4 - Image Cropping

When the grid square is detected, we offer the user the option to tweak it in case the detection was not accurate enough. While in most case the detection is good, this is helpfull when the grid is very slanted or a heavy perspective correction is needed, because the algorithm can't detect these sorts of squares.

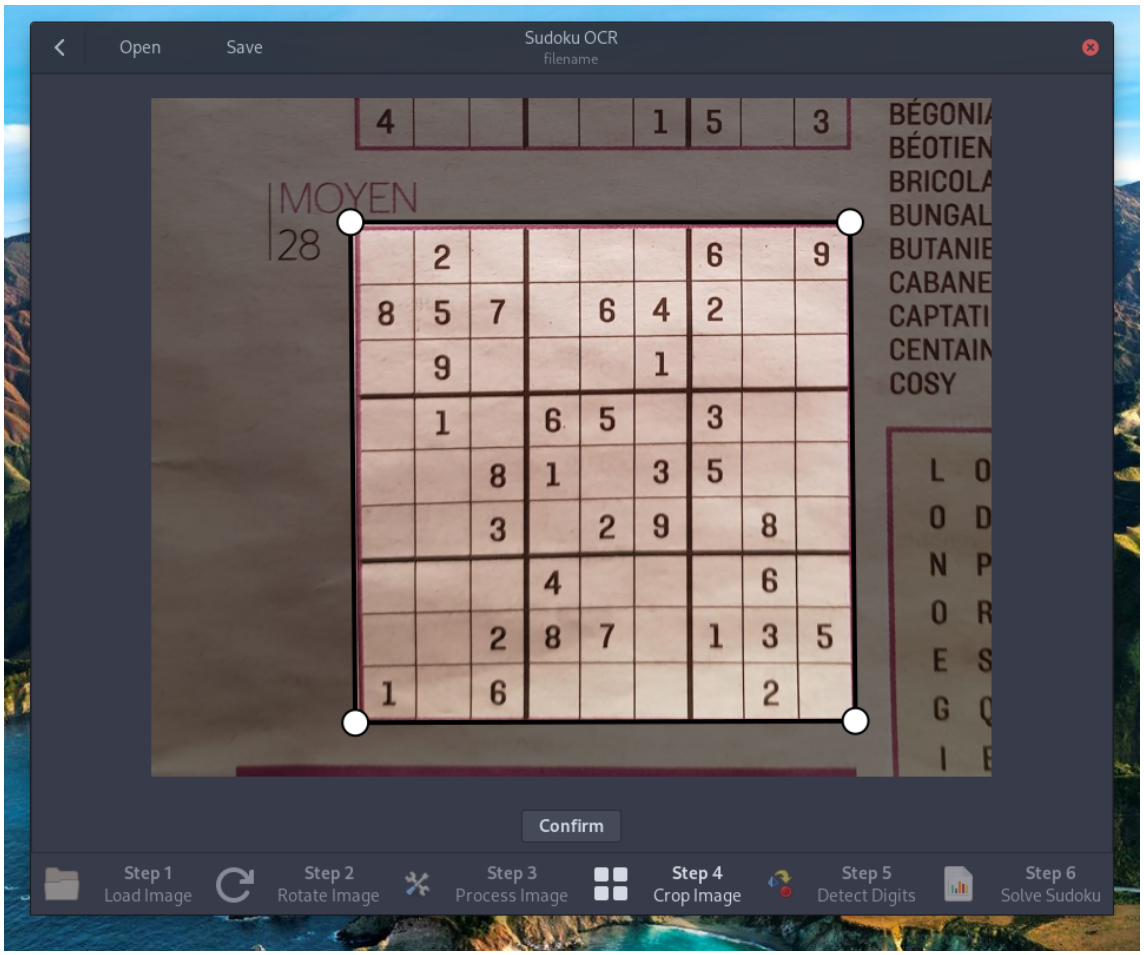


Image Cropping

6.3.5 Step 5 - Digits Detection

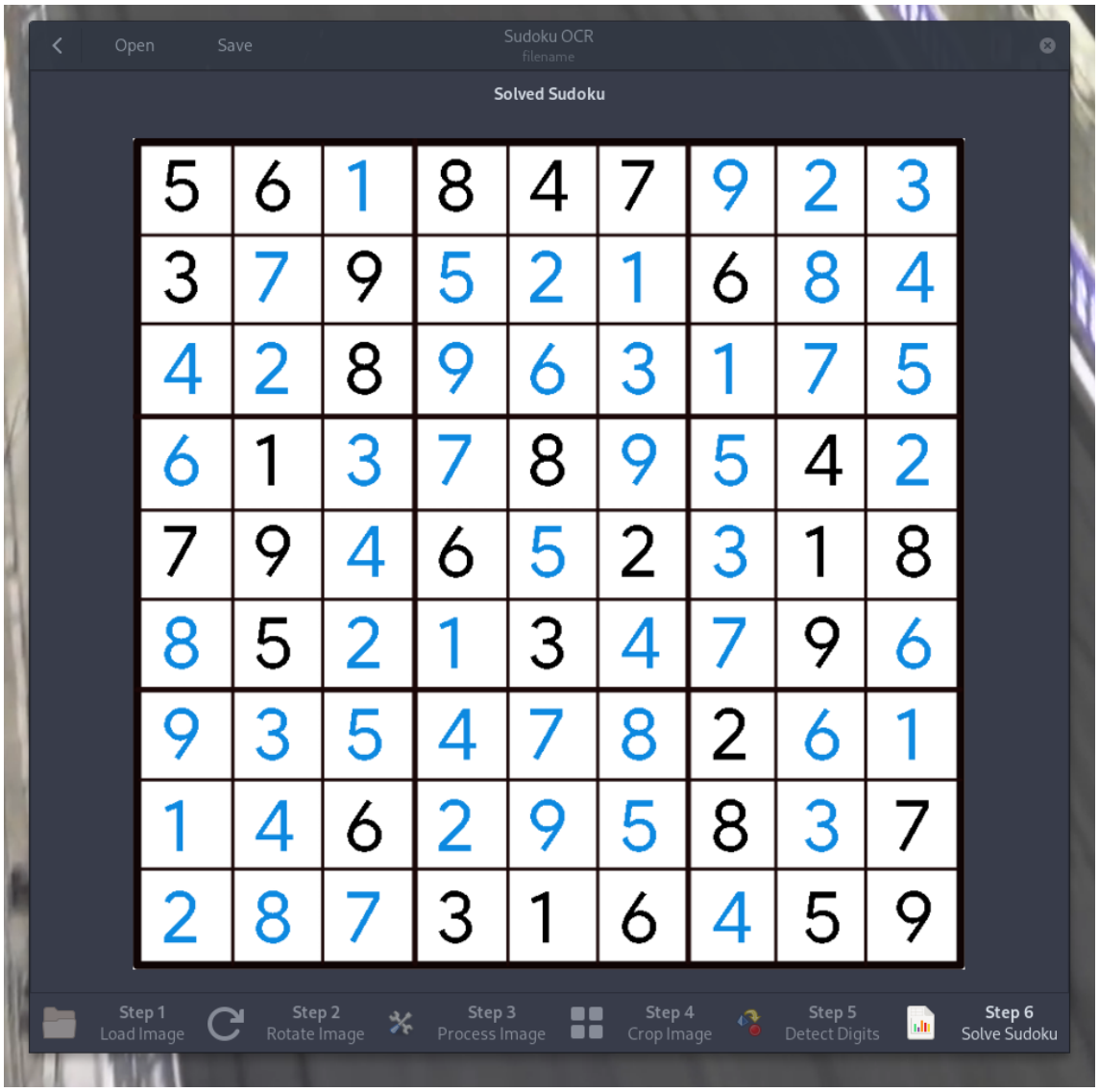
After the square is chosen and the grid is splitted and cleaned, the resulting digits are fed to the neural network to get what they represent. The result is displayed on top of the image to allow a for a correction by the user.

Detected Digits Display

By clicking a digit and then typing the replacement, a user can modify what was recognised by the neural network. This can be done on every digit as soon as they are detected. If the grid is deemed invalid, the user will be warned and able to modify the grid to correct the error making the grid invalid. The user is able to hide the grid or the image while correcting to check if the actual digits and the found ones are the same.

6.3.6 Step 6 - Sudoku Solving

After the Digit Detection is done, the Sudoku solving algorithm is called. We therefore obtain, following the application this program, a solved Sudoku grid. This grid will be saved in a file with a .result extension. Ina addition, this grid will then be reconstructed as an image.



Reconstructed Grid

7 Conclusion

After months of hard work, problem solving and more. We are so happy that our project is working. We learnt so much from this project, not only in C but also in project managing and group cohesion. This project was much more difficult than the project S2, there were a lot of specifications, hard stuff to understand but we did it. So yes, everything is done and well done.