OCR - Second Report

Gaspard Jolly, Robin Paulik, Luca Sarrubi, Tom L'Hotellier

December 9, 2023

Contents

1 Group members and Organization

1.1 Tom L'Hotellier

Before studying at EPITA, I never did anything related to code or programming. It was a real surprise for me to discover how interesting was programming and it became a pleasure really fast. I was a bit worried before the begining of the year because I heard that C was trickier to learn that C but overall, I am quite comfortable with C programming now and I find some new concepts such as memory management or multi-threading, very interesting to learn about. Very quickly from the beginning of this project, I became aware of how interesting it is and how, task that can seem simple, are in the end more complex that we think about - all the pre-processing procedure for instance.

1.2 Gaspard Jolly

My name is Gaspard Jolly, in highschool when I choose to study math and digital sciences, I thought that I was just going to try those computer things that I saw on internet and maybe it would interest me. Since then I really liked programming, either when I developped a website with a friend in highschool or with the programming of the video game we did last year. This new project allows me to tackle areas that I haven't studied in the past and I've found image analysis very interesting, for example.

1.3 Luca Sarubbi

Hi! I'm Luca Sarubbi. Some day hopefully a low-level software engineer. although it all started from enjoying Minecraft's game engine through custom Java plugins, I then got introduced to lower level languages such as C or Rust. My contribution to the project will mostly be on the AI in order to gain some experience as AIs are heavily used in robotics and IoT.

1.4 Robin Paulik

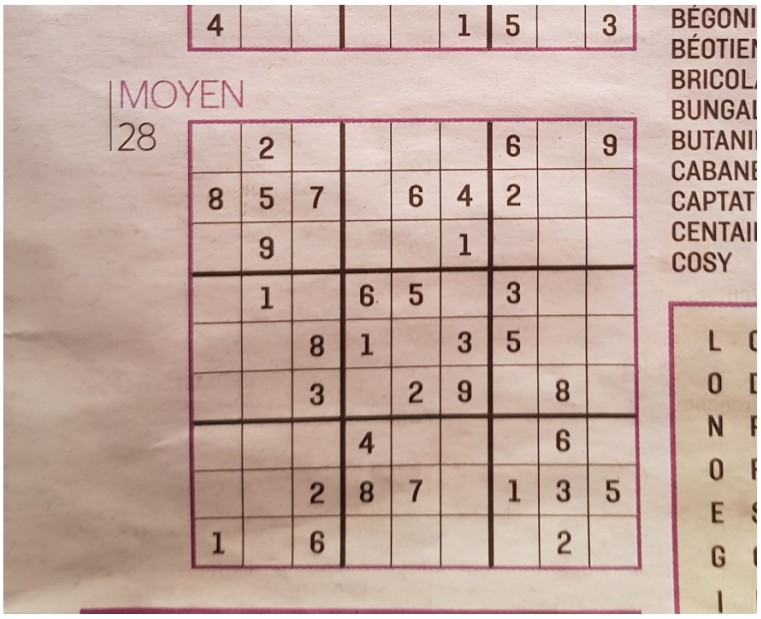
Image processing is something that has intrigued me for some time. I just really like the simple idea of modidying images through formulas. However, I've never had the courage to dig deeply into it. I think this project is a good way to start and learn about ocr, that I will probably really like discovering. I also really enjoy puzzles and enigmas, so the idea of solving sudokus in this way is particularly appealing to me. This project will also greatly improve my skills in the C language.

2 Pre - Processing

2.1 Introduction

We used multiples images to test our pre-processing, but in this report, we are going to show you the process on one single image. You can find other images in the repo of the project.

To be begin with the treatment of the image, let us firstly see the image we are going to work with.



This image contains all the shapes of colors that the grid had at the moment of the picture. Like that we cannot detect the grid because of all the nuances. We are now going to use different filters so that we can get a grid as readable as posible. The vast majority of this work had been done using SDL, using a lot of his tools, like the "SDL_Surface" , "SDL_Format" or "SDL_GetRGB()".

1. 1. Grayscale Filter
2. 2. Contrast Filter
3. 3. Median Filter
4. 4. Medium Filter
5. 5. Adaptative Threshold Filter
6. 6. Smoothing Filter
7. 7. Inversion Filter

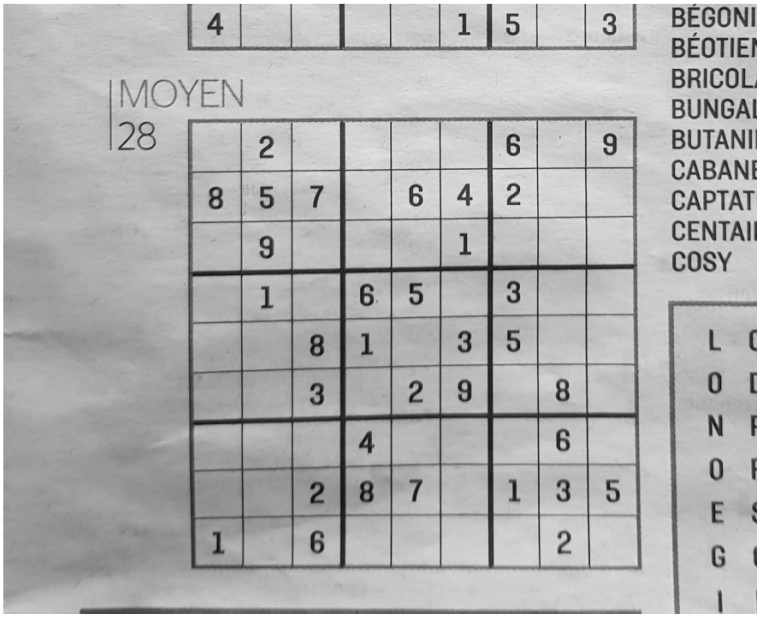
2.2 Grayscale

The first step is to change the image into a grayscale image, that is to say a picture that only contains shapes of gray.

For that we are basically going to go throught every single pixels of the image, taking the values of the RGB of each of the pixels using "SDL_GetRGB()" and apply a basic formula:

$$\text{Red Shape} = \text{Red Shape} * 0.3$$
$$\text{Green Shape} = \text{Green Shape} * 0.59$$
$$\text{Blue Shape} = \text{Blue Shape} * 0.11$$

Here is the result after applying the grayscale filter:



2.3 Standardisation of lighting

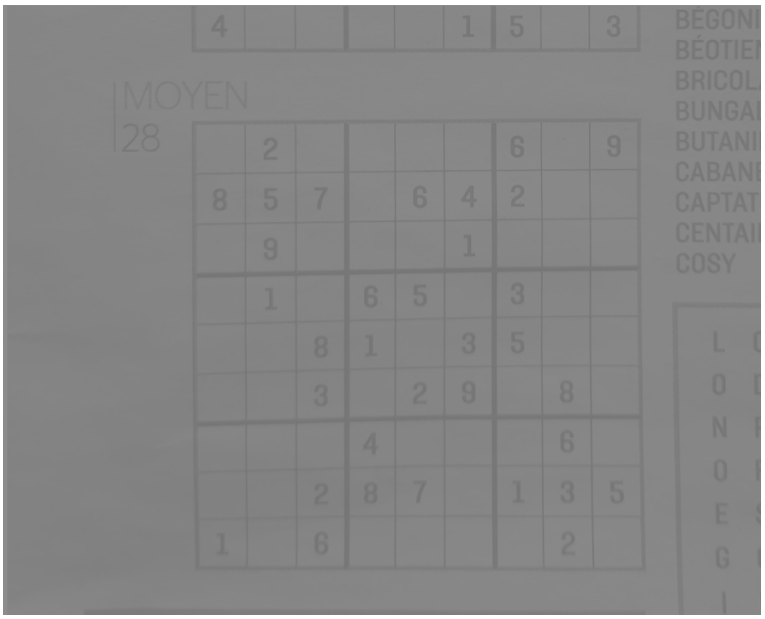
We are now using a contrast filter to help us better distinguish the writing from the distracting elements in the image.

To do this we define a factor, here we choose 2 for our example, we go throught each of the pixels of the image and then we use the following formula on each of the RGB of the pixel we are currently on.

$$\text{value} = (\text{value} - 128) * \text{factor} + 128$$

Then we check if the value that we get is between 0 and 255, if not then we set the value to the nearest bound.

We get the following image:



2.4 Noise Reduction (Median Filter)

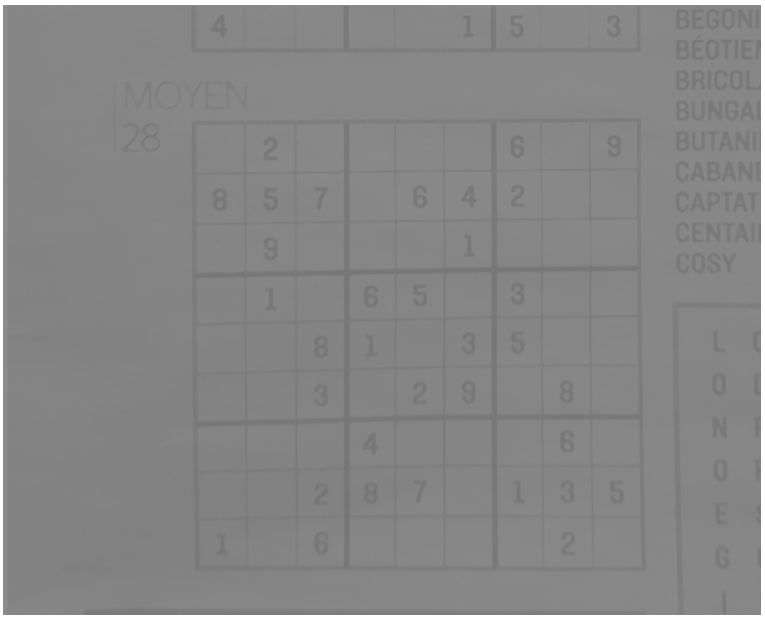
We are now going to get rid of the noise on the picture, which will give us a clearer image that we will be able to put into a threshold filter at the end of the proces.

This filter is a bit different from the last one. We are still going throught all the pixels, except the border ones. This can be explained because we are going to use a kernel on all the other pixels. A kernel is a matrix (here a 3x3 grid) that looks like this:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The kernel will take the shape of the pixel and his 8 neighboors and put them in an array. Then we just sort the array in the ascending order and take the median of the latest.

We then change the values of RGB of the pixel we are currently on to the values of the median we took. Here is the result:



This filter will be improve for the next defense. Indeed we are not taking into account the border pixels, to resolve this problem we will use a padding method si that those pixels will also be pasted in the median filter.

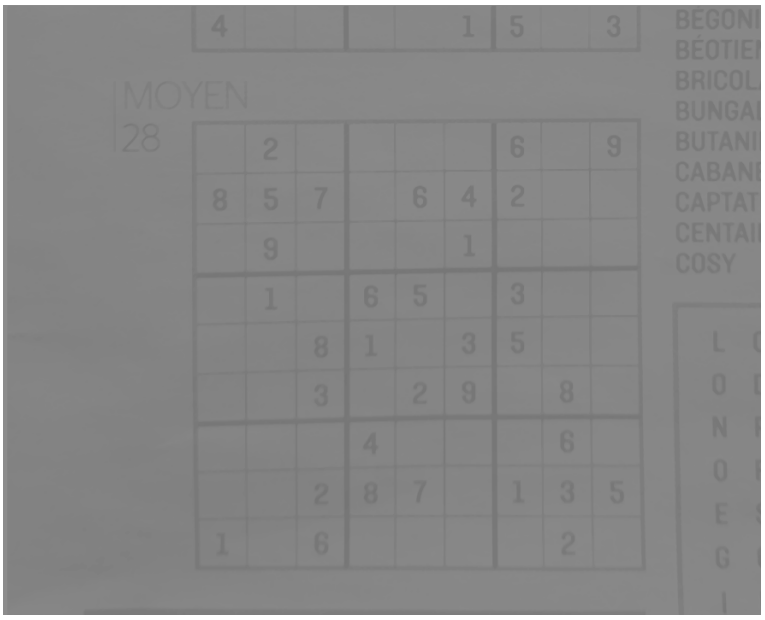
2.5 Reducing parasites (Medium Filter)

This step is used to blur the image and get rid of unnecessary details on the image that could complicate the detection of the grid.

As we did with the last filter, we are going to take a kernel, but this time we are going to use the Gauss' blur, which means that we are going to use a particular kernel that will use the pascal triangle:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

We then get the next image:



2.6 Adaptive Threshold (Binarisation)

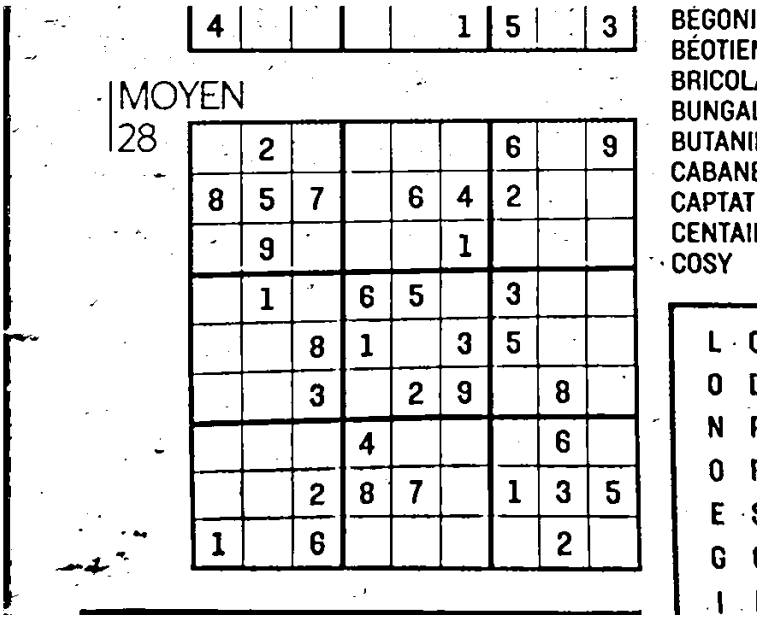
For this part, we are going to pass the image into a filter which will reduce the shapes of the image to only black or white. For this purpose we are going to use the principle of the Otsu algorithme. We have to calculate a threshold for the image, in our example we are going to use a particular threshold that we have determined. For the next defense, we will have a function that calculate the threshold of the image that we put in the filter.

As we did with the latest filters, we are going to cut the images in subimages, using kernel.

Then using the threshold that we found we are going to go throught each of the pixels, except the ones on the borders, calculate the average of his neighboors and then use the following formula:

$$\begin{aligned} \text{If } (\text{actual} - \text{average}) > \text{threshold} &\Rightarrow \text{white pixel} \\ \text{Else} &\Rightarrow \text{black pixel} \end{aligned}$$

Then we get the following image:

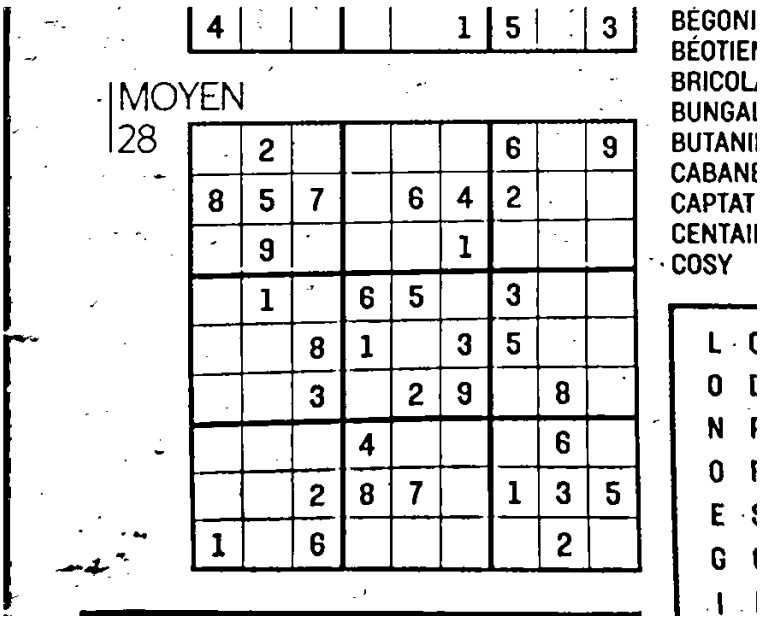


2.7 Smoothing

Now that we have a clean image with nothing more than white and black pixels, we are going to thinken up the lines to make it easier to detect the soduku lines.

For that, to begin with we create a new surface that is a copy of the current image, where we are going to change the pixels that need to be changed, because if we used the smoothing filter on the image without using a copy, we would get a totally black image. Then we go throught each of the pixels (again except the border ones) and if there is a black pixels in the neighboors of the current pixel we change this one into a black pixel (if it was not already) in the copied surface. Then we transfer the copied surface into the initial one.

We get the following result:

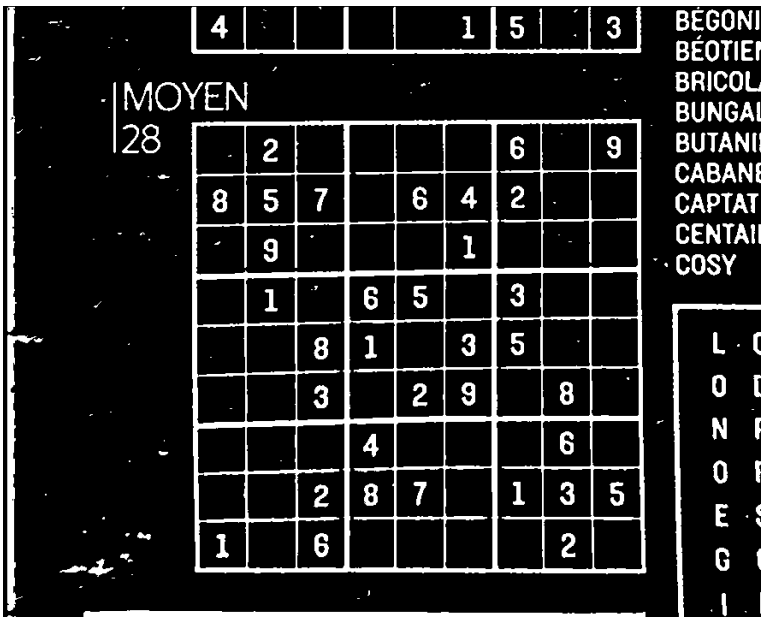


2.8 Inversion

Here we just basically inverse the values of each of the pixels on the image using this formula:

$$\text{value} = 255 - \text{value} \tag{1}$$

We finally get this image after using each of the filters



2.9 Conclusion

We hence get an image that is now usable for the grid detection as well as recognition with the neural network.

3 Image manipulation

3.1 Introduction

After pre-processing and before sending the grid to detection and splitting, there is still an important step to do: rotation.

Since the detection process needs the grid to be perfectly straight, rotation should work for any angle.

For this first defense, the rotation works with a user input in the command line with the following structure:

$$./rotate(imagefilepath)(angle)$$

When executed, this command will save the rotated image in a new file, in the 'Rotation' directory.

```
tom@tom-ThinkPad-P15v-Gen-2i:~/OCR/Rotation$ ./rotate sudoku_1.jpeg 28
```

3.2 Rotation

Let's now dive into the implementation of this rotation program.

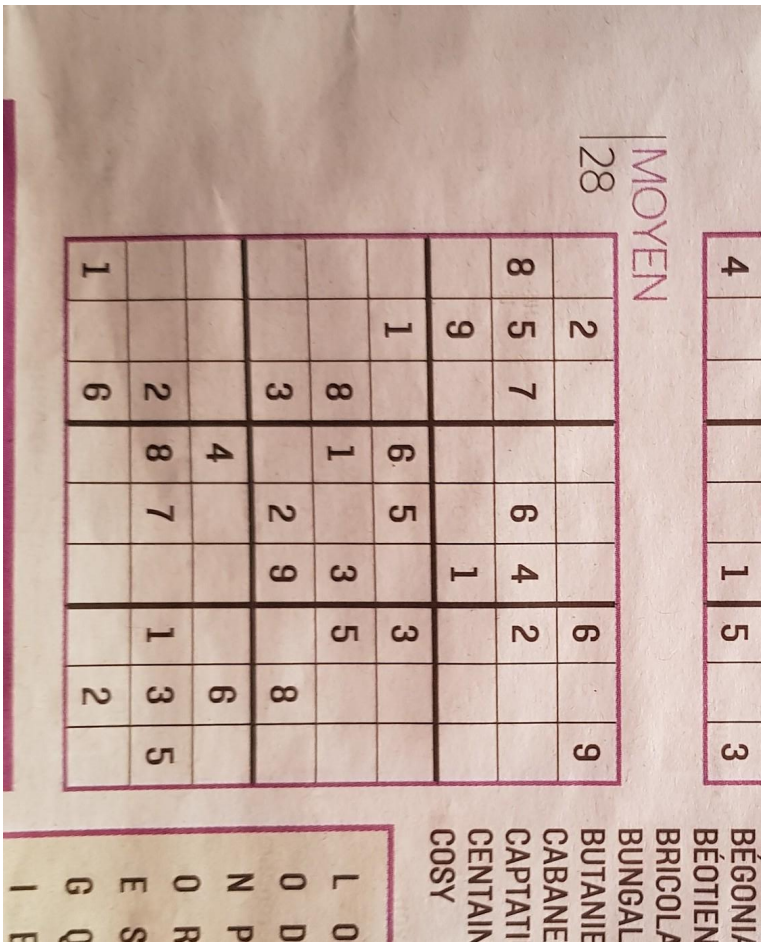
First of all, there are two very different methods to rotate the image.

The first one is used when the angle given by the user is a multiple of 90.

When that is the case, we call an auxiliary function called `rotation90()` the number of times equal to the input divided by 90. For example, if the input is 270, we call the function three times. This method is the simplest one because rotating an image by 90 degrees to the right can be done with matrix manipulation using the following formula.

$$\text{rotatedpixel}[y][\text{height} - 1 - x] = \text{pixel}[x][y] \tag{2}$$

The dimension of the new image have obviously been invereted before. This is the result for a 90 degree rotation.



The second one now, is way trickier and makes the program work for any angle.

Rotating an image by an angle that is not a multiple of 90 is not possible right from the hook, as you cannot save an image that is tilted. To tackle that problem, we hence had to modify the dimension of the new image in order to add some space for the image to rotate correctly and not to lose any pixels. Now for the actual rotation, we obviously had to use trigonometry. The formula was difficult to find but once we got it, it was a quite straightforward implementation.

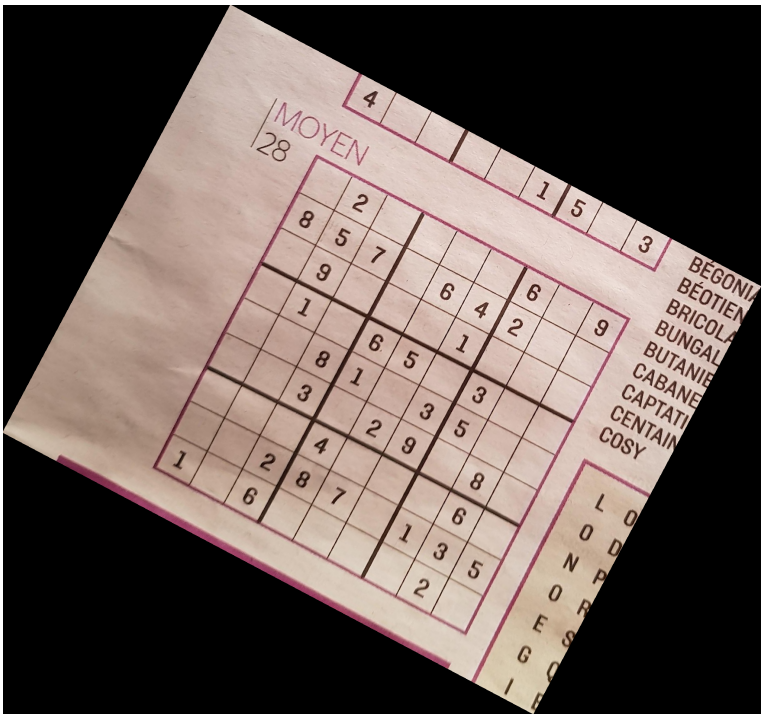
Here is the formula:

$$\text{sourceX} = (x - \frac{\text{width}}{2}) \cdot \cos(\text{angle}) - (y - \frac{\text{height}}{2}) \cdot \sin(\text{angle}) + \text{cX} \tag{3}$$

$$\text{sourceY} = (x - \frac{\text{width}}{2}) \cdot \sin(\text{angle}) + (y - \frac{\text{height}}{2}) \cdot \cos(\text{angle}) + \text{cY} \tag{4}$$

$$\text{rotatedpixel}[x][y] = \text{pixel}[sourceY][sourceX] \tag{5}$$

For instance, for a 27 degree rotation, we would get the following image:


MOYEN
28
BÉGONI
BÉOTIEN
BRICOLA
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY

4 Grid Recognition

4.1 Edge Detection

4.1.1 Sobel Operator

The Sobel operator is used for edge detection in digital image processing. It involves convolution with two 3x3 kernels to calculate the gradient in the x and y directions. The Sobel operator helps identify edges in an image.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

X Kernel (X Gradient G_x)

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Y Kernel (Y Gradient G_y)

For each pixel, the gradient's norm will be as following :

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2} \tag{6}$$

And the gradient direction, that will be used for the next step :

$$\mathbf{\Theta} = \operatorname{atan}(\mathbf{G_y}, \mathbf{G_x}) \tag{7}$$

4.1.2 Non-Maxima Removal

After applying the Sobel operator, non-maxima suppression is used to thin the edges. This step ensures that only local maxima in the gradient direction are retained while the rest of the points are set to zero.

4.1.3 Double Thresholding

Double thresholding is applied to the magnitude of the gradient. Pixels with a magnitude above a high threshold are considered strong edges, while those below a low threshold are considered non-edges. Pixels with a magnitude between the two thresholds are considered potential edges.

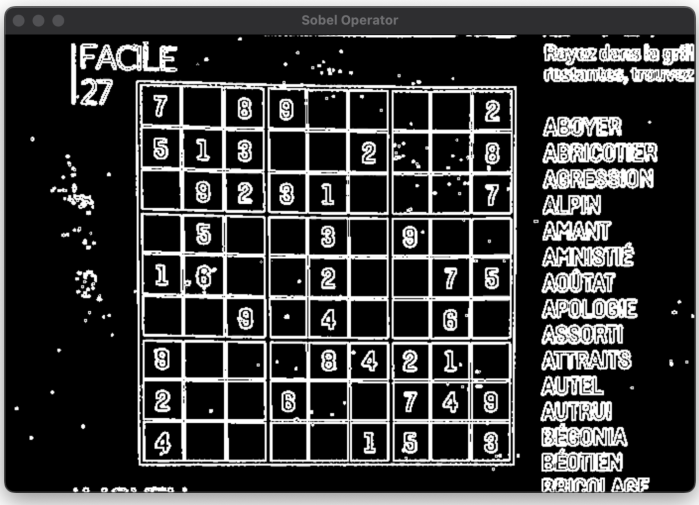


Figure 1: Image after Sobel Operator

4.1.4 Hysteresis Analysis

Hysteresis analysis is used to connect weak edge pixels to strong edge pixels. It helps to trace continuous edges by considering the connectivity between pixels. If a weak edge pixel is connected to a strong edge pixel, it is considered part of the edge; otherwise, it is removed. Once this step is done, the edge detection is fully done and it is now possible to go onto the next step.

Figure 2: Image after the Edgde Detection Process

4.2 Hough Transform

The Hough transform is used to detect lines in an image. It creates an accumulator array of two dimensions, ρ and θ, to find the most prominent lines. The Hough transform is

commonly used in line detection, thus it was extremely adapted for the project.

The lines are expressed in polar coordinates and can be represented like this :

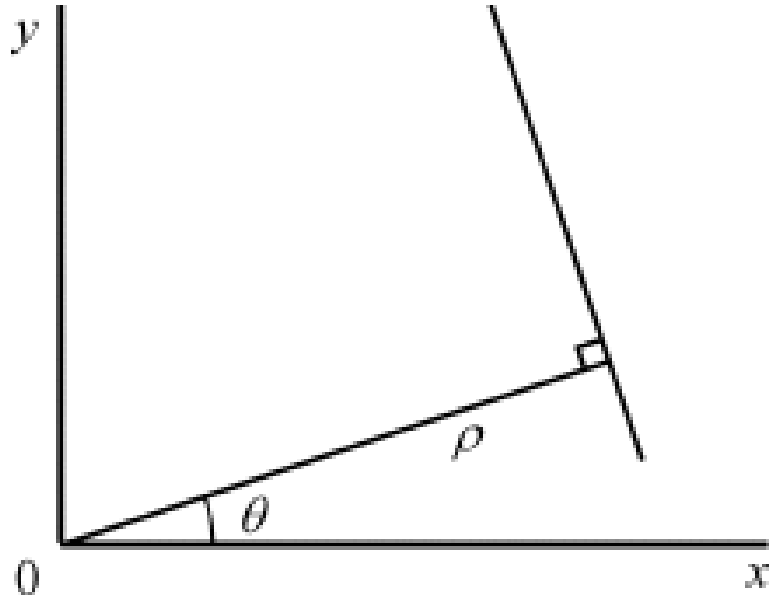


Figure 3: The line expressed with ρ and θ

Thus, since we are in polar coordinates, to help for our computations, we can express x and y in cartesian coordinates as functions of ρ and θ:

$$x = \rho \cdot \cos(\theta)$$
$$y = \rho \cdot \sin(\theta)$$

We can also deduce the other related couples to trace our lines, with d a distance large enough , in this case the diagonal:

For vertical lines:

$$x_1 = x_0 + d \cdot \cos(\theta)$$
$$y_1 = y_0 + d \cdot \sin(\theta)$$

For horizontal lines:

$$x_2 = x_0 - d \cdot \cos(\theta)$$
$$y_2 = y_0 - d \cdot \sin(\theta)$$

Once we have our lines, we have to refine them to only keep ten of them, that are in fact the grid lines. To do this, we detect lines that are similar enough and we merge them

by only keeping the average of these two lines. Once this process is done, we have this result and our grid is isolated:

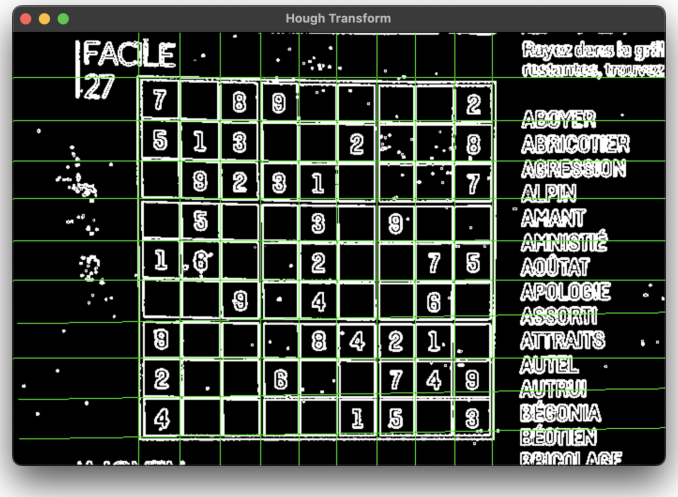


Figure 4: Image after Hough Transform

Since we tested with many images, we realized Hough was not performing that well on every image. The reason for that is that it had many parameters, including thresholds and limited number of detected lines. To fix that we had to rethink the way we did it. Moreover for some images, some edges not from the sudoku were detetected and there were no solution to only keep 10 lines. We decided to change it and to keep every predominant line, even not from the grid. It was then an issue for the grid detection. In practical way, we also lowered the maximum treshold.

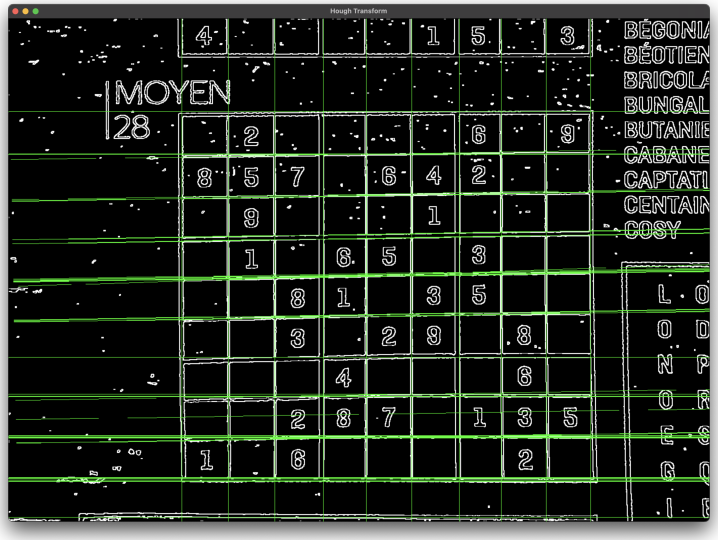


Figure 5: Fixed Hough Transform

Even if we are missing some edges like in this image, the detection still can be done. We just have to get a minimal number of grid lines.

5 Automatic Rotation

The rotation was already implemented, so that was really missing what the right angle. To get it we needed to use the canny edge algorithm and the hough transform. However the rotation had per effect to change the width and the height of the image. The downside was that it lowered the proportion the sudoku had on the image thus the Hough Transform was not performing well.

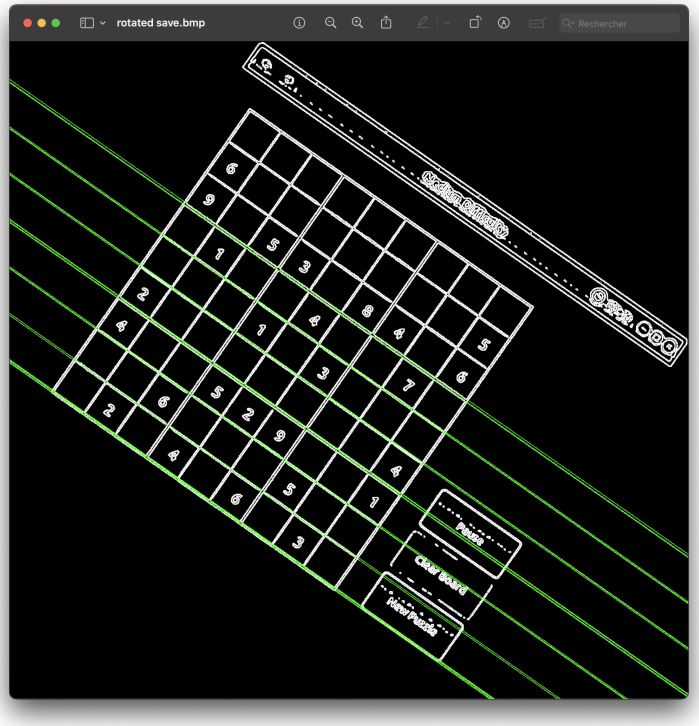


Figure 6: Image 5 with 55 degree angle before rotation

To fix that, we chose to recenter it and then crop the image. It lowered the sudoku's render however, it was still performing very well for the detection's angle. To get the angle with hough, we just implemented an histogram with every angle from range 0 to 180. The predominant one, that is the one with the most occurences was the right angle. Then we just had to apply the rotate function, previously made, with that angle and we are done for the automatic rotation

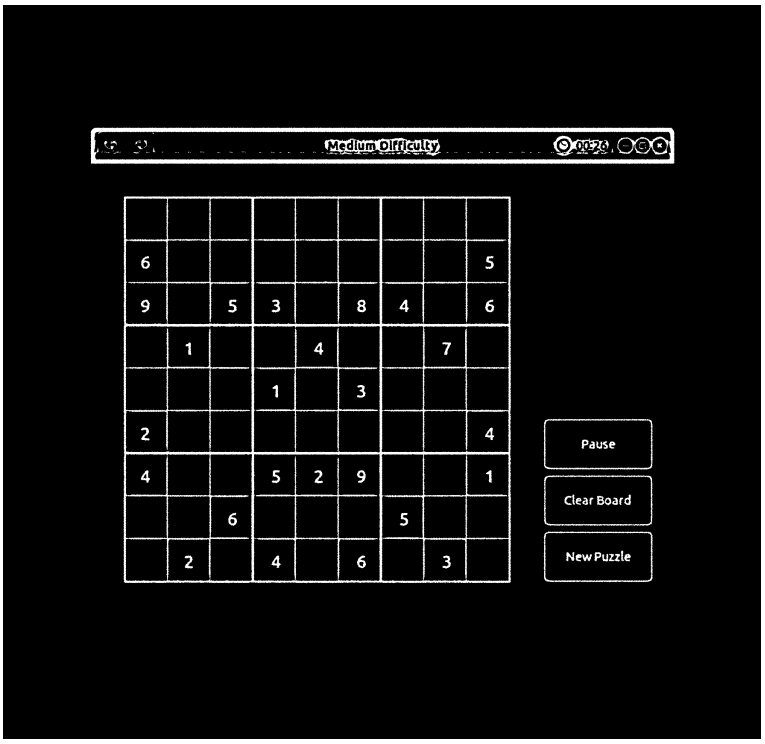


Figure 7: Same image rotated and center/adjusted

6 Cell Division

6.0.1 Grid Detection

Once the lines are traced on the grid with Hough, we have to identify the intersection between those lines. We first detect the four vertices of the grid. Then we just extract the part of the surface where the grid is located and render it as a bitmap. Since we modified the Hough Process, we had to redo the detection process. This time it was not as easy as modifying one or two parameters, since the whole algorithm was not fitting at all no more. From now on we detect the grid by finding the biggest square detected. To do that, for every vertical line, we go through every horizontal line at the intersecting it, and we see if there is a couple for which they can form a square. Now we just keep the biggest one. This imply still having a good hough detection, if not, the detection would be impossible.

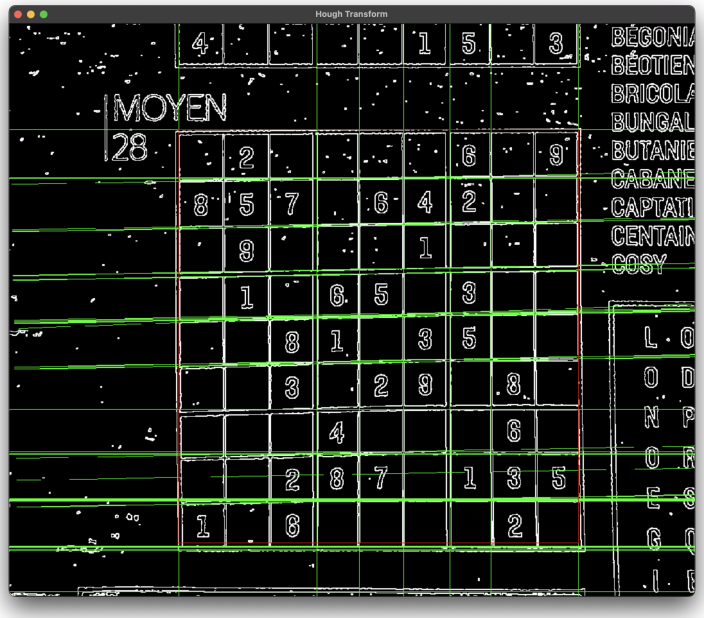


Figure 8: grid detected in red

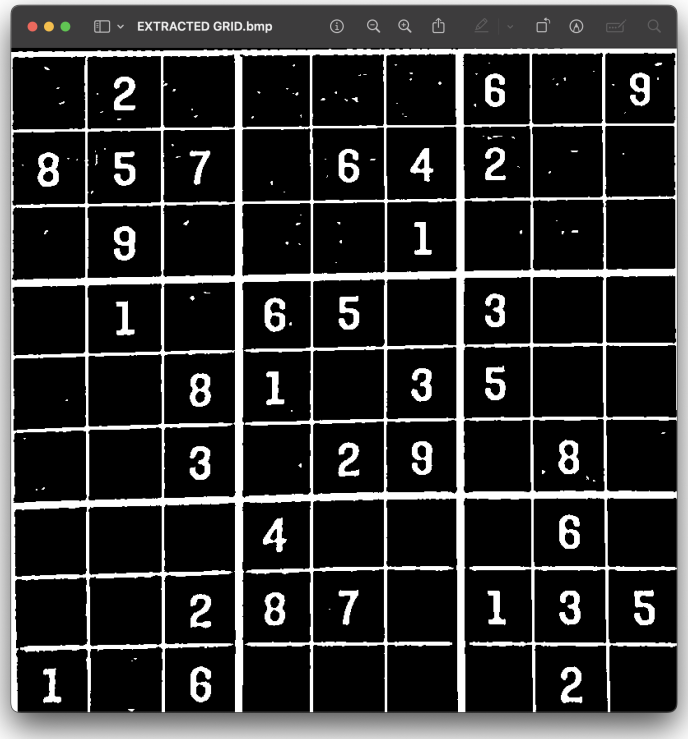


Figure 9: the extracted grid

6.0.2 Cell Division

We then repeat the same process for each intersection, and as a result, we have our 81 grid cells rendered as bitmaps.

Each intersection is represented by a red pixel in figure 4. Every cell is rendered like that :

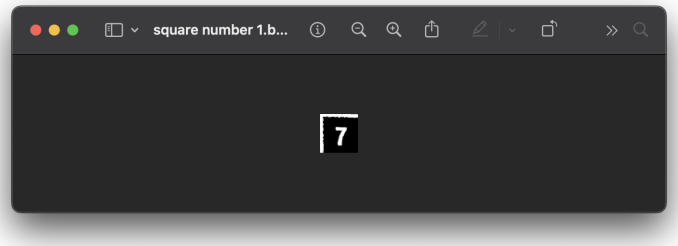


Figure 10: Example of a grid cell

6.0.3 Cell Cleaning

Then, the last step is to remove the lines, with a simple function. If we detect a non black pixel in the edges, then we remove recursively every non black adjacent pixel.

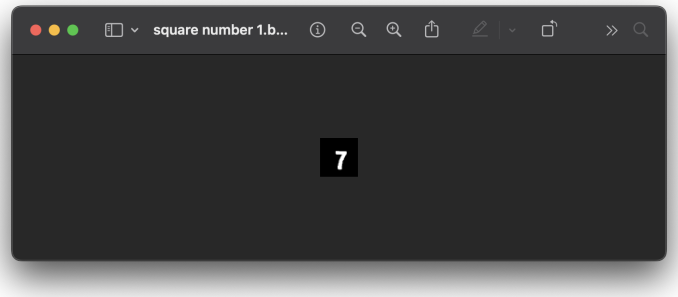


Figure 11: Cleaned grid cell

7 Solver

Despite all the fancy technology used in this project the Sudoku still needs to be solved.

7.0.1 Backtracking

In order to solve the Sudoku, we use the classic backtracking algorithm. It is a brute-force technique that consists in trying all the possibilities until a valid solution is found. If no solutions are found then the Sudoku grid is deemed unsolvable.

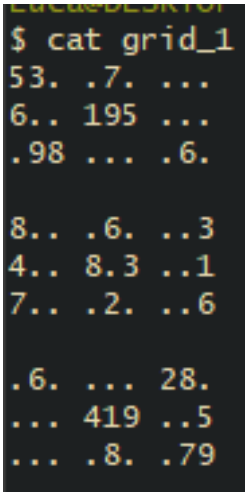


Figure 12: unsolved Sudoku before being fed to the solver program

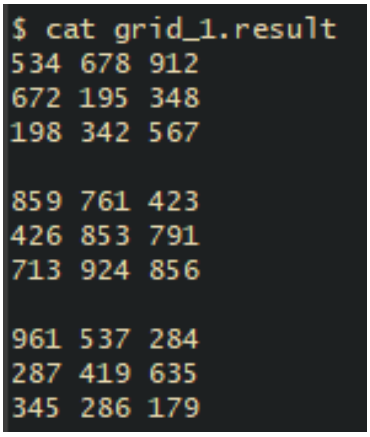


Figure 13: solved Sudoku under the same format as the previous figure

8 Character Recognition

8.1 Global Structure

We have chosen to implement a classic structure used for character recognition. It consists (at least for this state of the project) of one 784 neurons wide input layer, a 30 neurons hidden layer as well as a 10 neurons output layer where each value represents the value of its index (for instance: an output layer of [0.0100, 0.0024, 0.9923, 0.0001, ..., 0.1254, 0.0065] means that the AI think the value is a 2 as C arrays are 0-indexed). As for training the Stochastic Gradient Descent (SGD) on the MNIST dataset freely available at http://yann.lecun.com/exdb/mnist/ is used.

8.1.1 Feedforward

The activation function used is the sigmoid function although this is subject to change.

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1 + e^{-x}}$$

8.1.2 BackPropagation

As the sigmoid function is used during the feedforward process, its derivative is used during the backpropagation.

$$\text{Sigmoid prime: } \sigma(x)' = x * (1 - x)$$

8.2 Image training

in order to train IAs need a dataset so decided to use the MNIST public database as a rich dataset of labeled handwritten digits.

8.2.1 Inputs

The MNIST database is made with images of 28 by 28 pixels ranging from 0 to 255 saved under the idx3-ubyte format and all the labels are saved under the idx1-ubyte format. So a separate loader has been made to load data from the separate files, map all pixels from the range [0, 255] to [0, 1] and pack all images with their corresponding labels inside a designated data structure. Which is why we choose 28 * 28 neurons as our input layer.

8.2.2 Training process

Our Stochastic Gradient Descent (SGD) provides 3 hyper parameters:

- The number of epochs
- The mini batch size
- The learning rate (often specified as 'eta')

We have tried many different values for each of them and have found that we can obtain 95.58% accuracy on the test dataset of the MNIST database with 30 epochs, a batch size of 100 and 3.0 for the learning rate. The training took 67 seconds on an AMD Ryzen 5 1500x.

8.3 Saving and Loading

In order to avoid having to train the AI each time the process is stopped and restarted, it is need to create a way to save and write all its weights and biases to a file. For this task we made a simple plain text custom file format: The first value n is always an integer and it specifies the number of layers the AI has. The value is the followed by n integers n_i, with i being the index of the layer, that specify the amount of neurons of each layer. those values are enough to deduce the number of weights and biases that the AI will contains so the file will then contain $\sum_{k=1}^{n-1} n_k$ biases and $\sum_{k=0}^{n-2}(n_k * n_{k+1})$ weights. The format is also in plain text because it enables the use for debugging purposes.

```
3 784 30 10 -1.301793 0.005424 -1.792563 -1.847729 -1.258441 0.395521 -0.446648 0.654379
-0.768026 -0.538065 -0.734470 -0.117964 -0.785015 0.083502 -1.097018 -0.368624 -1.045627
-0.485394 -1.122907 -1.330792 -0.419250 0.905501 -0.339100 -0.867621 -1.016356 -1.262261
-1.024899 -1.351308 -1.823998 -1.575985 -1.729217 -0.196558 -0.482185 -1.939564 -1.124072
-0.056885 -0.814465 -1.905381 -2.604399 -1.033782 0.000367 0.005699 -0.001646 0.007046 -
0.006823 -0.008372 -0.004663 0.008230 0.003898 0.004666 -0.003394 -0.004656 -0.006440 -
0.008799 0.004298 0.007853 -0.003793 0.002093 -0.002412 0.003629 0.001976 0.005369
0.006363 0.000446 0.005270 0.004762 0.008840 0.005610 0.008280 -0.008640 0.008762 0.006892
0.000709 0.006998 -0.001942 0.008499 0.006359 0.007066 -0.004208 0.003199 0.010210
0.006950 -0.007940 -0.009253 -0.014933 -0.000974 -0.001586 -0.012056 0.005217 0.001227
0.009439 0.008799 -0.002935 0.009977 -0.001768 0.001824 0.007995 0.002579 -0.002270 -
0.001639 0.001600 -0.008791 -0.002448 0.010134 -0.032619 -0.054488 -0.041746 -0.091187 -
0.128778 -0.160518 -0.206360 -0.208832 -0.183828 -0.128409 -0.180009 -0.077561 -0.040836
-0.020736 -0.012311 0.000772 0.008615 -0.000293 -0.005533 0.001563 0.008089 -0.009590 -
0.008276 0.008222 0.009312 -0.032424 -0.025816 -0.060433 -0.034472 -0.133796 -0.217373 -
0.261102 -0.078227 -0.037873 -0.016649 -0.069150 -0.034517 -0.031898 -0.264413 -0.158502
-0.204738 -0.356070 -0.239219 -0.061261 0.007084 -0.004133 0.010375 0.009016 0.004314 -
0.007327 -0.008036 0.008166 -0.026495 -0.118276 -0.134776 -0.045249 -0.083213 -0.106096 -
0.098539 0.001099 -0.160585 -0.244843 -0.300797 0.195786 -0.290279 -0.153622 0.010702 -
0.334157 -0.322236 -0.285692 -0.149635 0.041110 0.013514 -0.004188 -0.008384 0.005394 -
0.008146 -0.005346 -0.001073 -0.016316 -0.140879 -0.288739 -0.218045 -0.213568 -0.256679
```

Figure 14: Initial values of saved AI trained on the MNIST dataset (95% accuracy)

8.4 Note on performance

Because of the nature of neural networks and the Stochastic Gradient Descent algorithm, AIs require huge amounts of computing power, especially about matrix multiplications. In order to try to gain performance our AI code base follows a no-malloc after startup policy where any function used during training or normal usage (such as network_SGD() or network_evaluate()) do not allocate any data on the heap as to avoid any potential memory fragmentation, memory leaks or use-after-free bugs. Another step taken to help performance is the use of a BLAS library (in our case openblas). BLAS (or Basic Linear Algebra Subprograms) libraries provide heavily optimized functions to handle matrix such as sgemm() and dgemm(). Our AI can seamlessly compile with and without linking to the library so it can be trained on a more powerful machine that has a BLAS library and then used or trained on smaller dataset on another computer.

8.5 Affine Transformations

According to Google search, affine means 'allowing for or preserving parallel relationships'. So affine transformations allow to move, scale, sheer and rotate images while preserving the parallel relations between lines of the image. As such understanding and implementing them has proven very usefull.

They are used after each cell has been extracted from the sudoku grid in order to prepare the images for the AI.

8.5.1 Translations

We fist center each cell by calculation the average position of white pixels $p = (p_x, p_y)$

$$offset_x = \frac{\sum_{k=0}^{n} p_x}{\sum_{k=0}^{n} 1}$$

and

$$offset_y = \frac{\sum_{k=0}^{n} p_y}{\sum_{k=0}^{n} 1}$$

for each pixel $p = (p_x, p_y)$ in the input image, we can map it to a pixel $p' = (p'_x, p'_y)$ on the output image:

$$\begin{bmatrix} p'_x \\ p'_y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & -offset_x \\ 0 & 1 & -offset_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{8}$$

8.5.2 Zooming on a point

As affine transformations are linear maps, rotations and scales are always centered around a point which is the origin.

So in order to zoom or rotate on a point, we need to first translate the selected point to the origin of the space and after having applied the transformation(s) we translate the pixel back to the original position.

For example, a rotation θ around center point $c = (c_x, c_y)$:

$$\begin{bmatrix} p`_x \\ p`_y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{9}$$

$$\begin{bmatrix} p`_x \\ p`_y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & -c_x \cdot \cos\theta + c_y \cdot \sin\theta \\ \sin\theta & \cos\theta & -c_x \cdot \sin\theta - c_y \cdot \cos\theta \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} p`_x \\ p`_y \\ z \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & -c_x \cdot \cos\theta + c_y \cdot \sin\theta + c_x \\ \sin\theta & \cos\theta & -c_x \cdot \sin\theta - c_y \cdot \cos\theta + c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{11}$$

While this method works perfectly and is more efficient than doing 3 different transformations, It has a problem is the zoom factor is bigger than zero.

Here, we apply a zoom of factor 2 on this picture and we then offset by -10 along the x ans y axis.

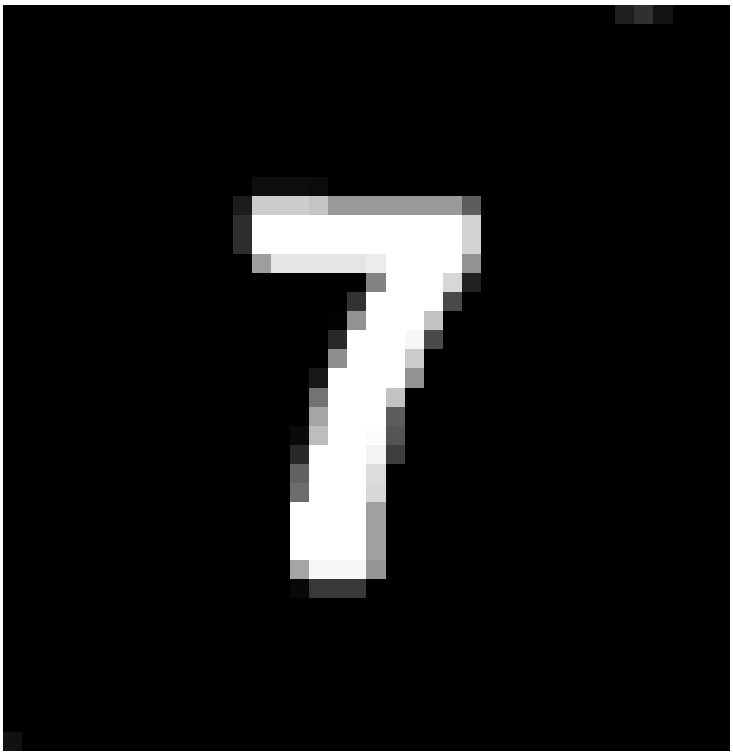


Figure 15: initial picture of a seven extracted from a Sudoku grid

As you can see, since we apply the transformation from the input to the output, if the mapped input size is bigger than the original (before it is truncated) we loose in density.

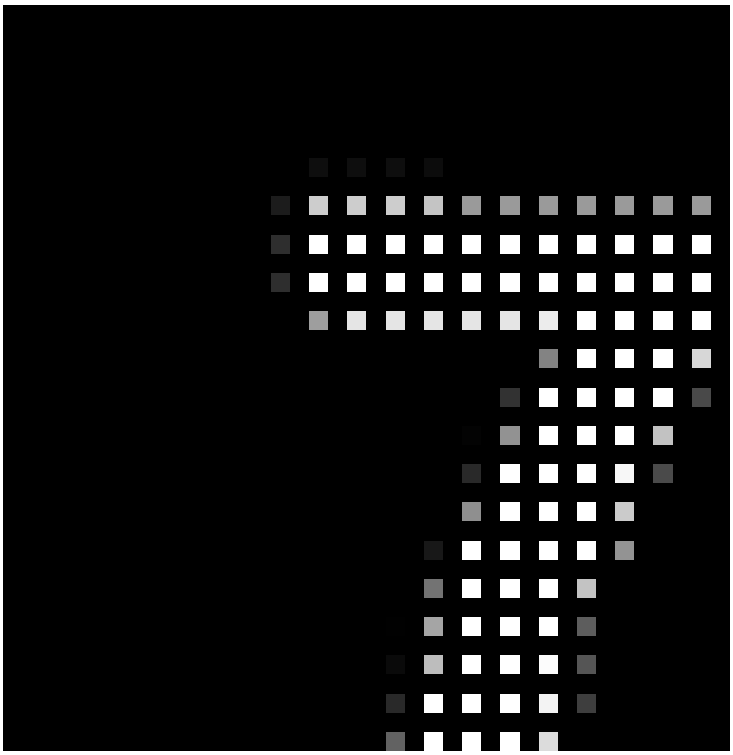


Figure 16: Figure 15 after the zoom and translation

To remedy this problem we apply the transformation the other way around: instead of mapping each input pixel to its output space, we map each output pixel into the input space. This way, we are sure that all output pixels are filled.

$$f(p) = p' \tag{12}$$
$$Ap = p' \tag{13}$$
$$p = A^{-1}p' \tag{14}$$
$$p = f^{-1}(p') \tag{15}$$

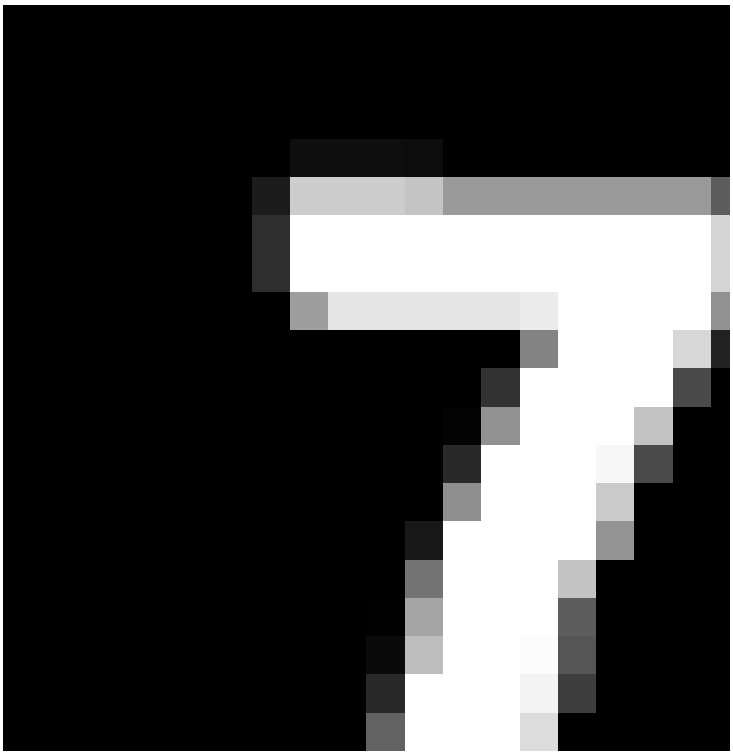


Figure 17: Figure 15 after the zoom and translation using the inverse function

8.6 training dataset

We've upgraded our dataset from the MNIST database alone to the MNIST database and more than a thousand fonts fetched from the Google font API.

8.7 Detection

As stated earlier the AI outputs a float array of length 10 corresponding to the confidence of the AI in the guessed number.

While we previously took the largest value, we now require it to be at least of 0.7 and to have at least 0.5 points difference with the second highest value.

These measures enable us to be more prudent as while undetected values may be found by the solver, if a value is wrongfully detected, the solver wont be able to solve the Sudoku grid.

9 User Interface

9.1 Introduction

The user interface is one of the major part of the project as it is the conveyor from the program to the user. Our goal was to make an UI very easy to use for whoever wants to use the app, and so to make it usable by most people, even if they don't know anything about programming. We are very proud of the result and how we could implement that and so, let's dive in the core of the subject very quickly.

9.2 General Principle

9.2.1 Glade

For the first defense, we started doing a bit of the UI using GTK, but, knowing nothing about it, we didn't know about glade and so we soon realized that it would be very hard to do something beautifull without using it. The mini-project was also a factor in using glade since, it showed what we could do with it.

Glade was very easy to learn especially because of the extended GTK documentation. Placing and moving GTK widgets on the canvas and using container was easy, however somewhat frustrating as some behaviours can be a bit weird but overall it is a really awsome tool.

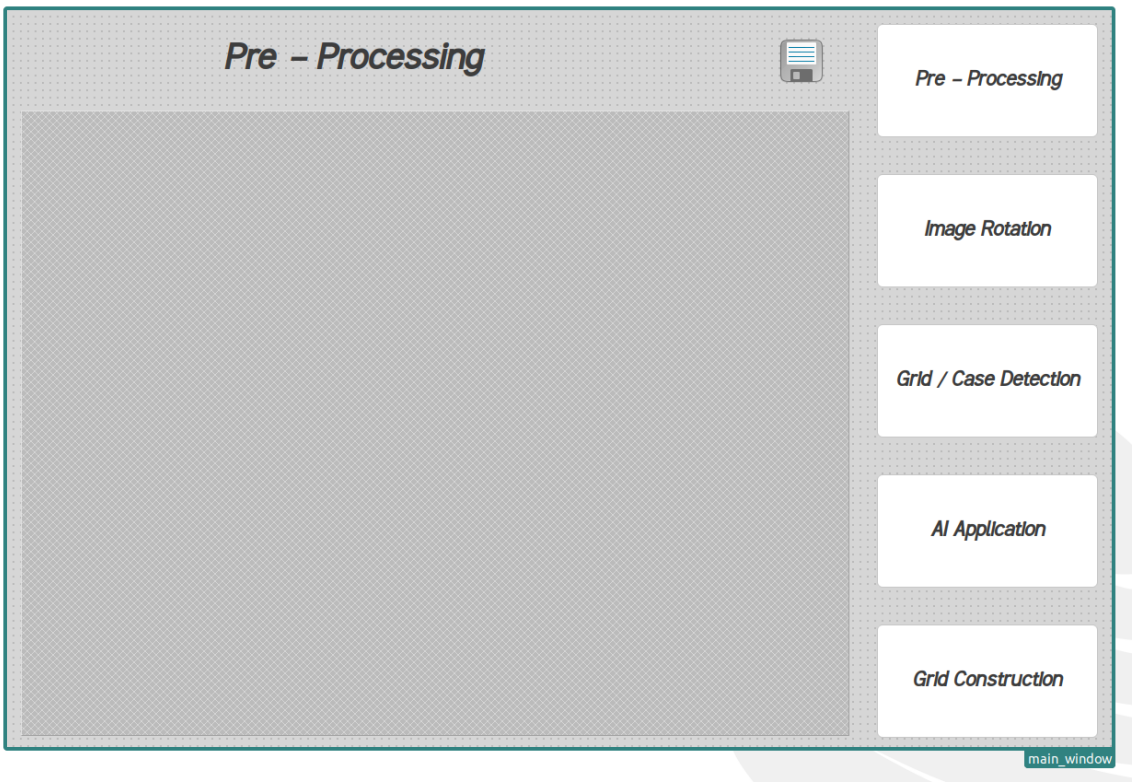


Figure 18: Solver menu in glade

It is while doing glade that we had the idea of doing 3 separates menus: the main menu, the solver menu and the functionalities menu.

We are then able to save what we did on the glade app as a glade file that is XML code and then, we can use a Gtk builder to call this file in our C code as follow:

```
//Create a builder
GtkBuilder *builder =
gtk_builder_new_from_file("glade/solver_menu.glade");

//Get a widget from it
GtkWidget *main_window =
GTK_WIDGET(gtk_builder_get_object(builder, "main_window"));
```

9.2.2 Using CSS

Only using glade can give you a menu but it will not give you something beautiful as it is not stylized. In order to do that, we can use CSS and link it with our glade file and our Gtk Widgets in the C file. The personalization potential is quite limited as plenty of CSS properties aren't available on Gtk Widgets, but it can still give a good style to the UI as you can change the background color, the text color, the borders of the buttons and of the components, and a bit more.

```
#main_window {
    background-color: #011627;
}


/* MAIN MENU */

#ocr_text {
    color: #7e57c2;
}

#authors_text {
    color: #FFFFFF;
}

#solver_button, #func_button {
    background: #0e293f;
    border: 1px solid #7e57c2;
    color: #FFFFFF;
}

#solver_button:hover, #func_button:hover {
    border: 3px solid #FFFFFF;
}
```

Figure 19: CSS file for the main menu

To link CSS to a file is also pretty straighforward and we created a simple function applyCSS(widget, provider) to apply CSS to each and every component of the UI. This function is very usefull as we will use it when we click a button and change the style of the button.

9.2.3 Code

Now that every menu was done in glade and each had its style thanks to CSS, we were able to implement it in C. The TP we had on GTK helped lot in order to understand how to connect buttons to functions and initialize a GTK window. The code is organized that way:

Each menu has its loading function that works like that:

1. The builder is created from the associated glade file
2. The CSS provider for the menu is created to be then applied to each widget
3. We get each widget from the builder using its name set in glade
4. We get the image from the path and we put it in a box
5. We put all the widgets in the Application structure
6. The buttons are connected to their functions
7. All the widgets are shown

What we soon realized is that we cannot pass multiple parameters to a button function. To avoid that problem, we created the following app structure that is used whatever the menu:

```
typedef enum State
{
    START,
    PRE_PROCESSING,
    ROTATION,
    DETECTION,
    AI,
    CONSTRUCTION
} State;

typedef struct Widgets
{
    GtkWidget *main_window;
    GtkWidget *pre_process_button;
    GtkWidget *image_rotation_button;
    GtkWidget *construction_button;
    GtkWidget *detection_button;
    GtkWidget *ai_button;
    GtkWidget *save_button;
    GtkWidget *main_title;
    GtkWidget *image;
    GtkWidget *image_box;
    GtkWidget *manual_text;
```

```
    GtkWidget *manual_holder;
    GtkWidget *manual_button;
    GtkWidget *black_and_white_button;
    GtkWidget *canny_button;
    GtkWidget *median_button;
    GtkWidget *threshold_button;
    GtkWidget *inversion_button;
    GtkWidget *automatic_text;
    GtkWidget *automatic_button;
    GtkWidget *pre_process_menu;
    SDL_Surface *image_surface;
} Widgets;

typedef struct Application
{
    State state;
    Widgets widgets;
} Application;
```

We also created the following Loader structure that will be used to load any menu when the button is clicked in the main menu:

```
typedef struct Loader
{
    GtkWidget *window;
    char* image_path;
} Loader;
```

Now let's talk about how we link the buttons to the right functions in the project.

First of all, every member of the group created a general function of what he did that could be called in the UI. To transfer it from his folder to the UI, we used header files and include in the makefile using "inc.mk" files. To summarize, everyone has a file that has the main functions that we need to use, and this file is called kind of like a library.

At the beginning, we called the associated function of the button into each callback function. This method was convenient and there wasn't any problem with it until we had to link the AI to the UI. We couldn't load the AI file when calling from a button callback function. We still don't know where the problem comes from but anyway, we had to do it differently.

We now call each function in the main function and before the window is opened. First of all, this ensures that the user won't have any problem while using the UI: if he can use it, then his sudoku can be solved. Secondly, calling extensive functions such as Hough detection when a button was clicked created a small lapse of time where the UI wasn't usable as the function was compiling. Now, even though the menu may take 2-3 seconds to launch, the UI is as fluid as possible since every function is already executed before and, the UI is only here to display the result of these.

9.3 The Menus

9.3.1 The Main Menu

The main menu is the most simple menu ever: it's the name of the project, two buttons and the name of the authors. Now about the buttons:

1. Solver Menu: This button is used to open the "Solver" window, which is the automated solving window.
2. Individual Functionalities: The individual functionality window is used to apply individual functionalities on images such as filter and rotation. It is only here for the defense and the demonstration and normally wouldn't be used by the user.

Figure 20: Main menu

9.3.2 The Solver Menu

As we just said, the solver menu is the menu handling the automatic solving of the sudoku. It has all the tools the user need to receive his solved sudoku grid in the end of the process and can be used very easily just by clicking the buttons.

Let's now talk about what each buttons does:

1. Pre-Processing: This button is used to show the user what the sudoku grid he entered looks like after the pre-processing functions are applied on it.
2. Rotation: This buttons triggers the automatic rotation of the grid, it displays the image after it has been automatically rotated by the automatic rotation function.

3. Detection: This activates the detection of the grid and the little cases, and stores them into a dedicated folder. It displays the image with the lines of the detection so that the user makes sure the process worked corectly.

4. IA button: This button doesn't display anything new to the user but it's here to show him that the IA as been enabled for his current grid.

5. Construction: Finally, the construction button solves the sudoku with the file the IA gave him and displays the newly solved grid to the user.

Style-wise, the user can see the buttons he clicked turn green, and an hover effect is present on each button of the menu to offer a greater experience. In order to do that we had to create a CSS file for each state the user is in and apply it to the widgets whenever the button is clicked.

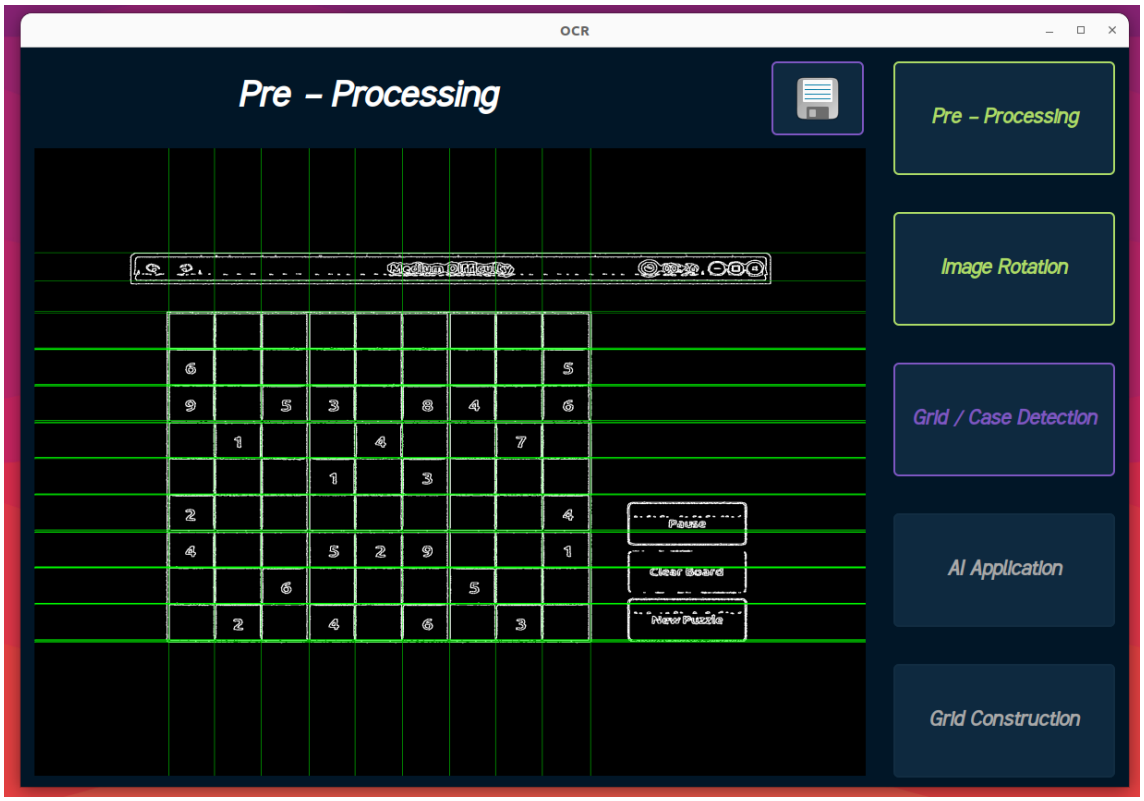


Figure 21: Solver menu

It is also important to note that the user is guided. He cannot click a button if the previous one has not been clicked. This is done by checking the states in the code.

9.3.3 The Functionalities Menu

This menu is very similar to the previous one except that this time, on the top you also have a button box that let you choose between pre-processing and rotation.

1. Pre-Processing: In this page you will have access to the majority of the filters used in the pre-processing stage. This can be usefull to detect which filter is responsible if anything goes wrong.

2. Rotation: This button puts at your disposition the option to do a manual and an automatic rotation. The manual rotation is done by entering a value in a text holder and then clicking a button.

This menu is not really convenient for the user and isn't very useful for him. But it is important for the defense to show different hidden functionalities of our program.

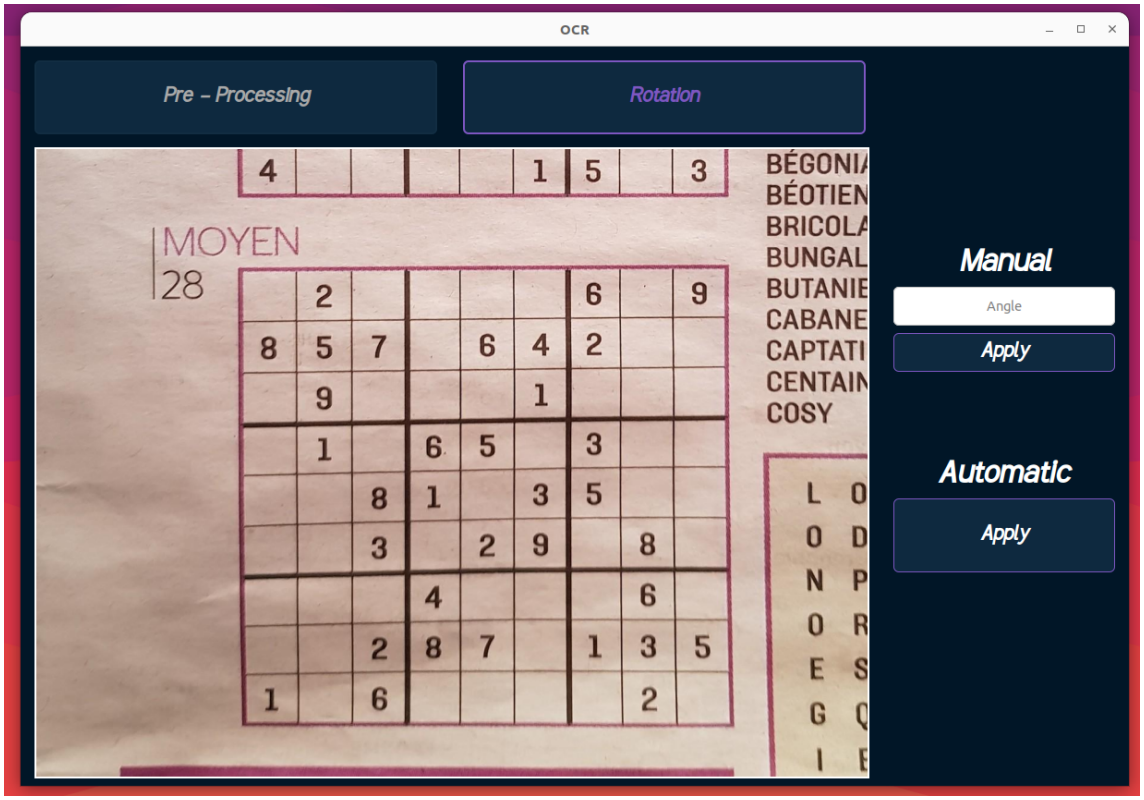


Figure 22: Functionality menu

10 Grid construction

The grid construction process was one of the last one we implemented. We were quite afraid of it, but in the end, it was moslty easy to do.

It is important to denote that this program absolutely need a complete grid in the text format to be able to work. It is intimely linked with the solver since, it takes the text file grid returned by the solver and transforms it into an image.

To do that, we first had to create and save an empty sudoku grid and a transparent image of each digit present in a sudoku, so from 1 to 9.

Then the program works as such:

1. First, each digit image, so from 1 to 9 is saved into an SDL Surface array
2. The text grid is converted into a 9 by 9 array, with each element of the array being the number corresponding to the yth column of the xth line.
3. Then the empty grid is loaded and, for each number in the array, the number is put at its corresponding coordinates in the empty grid. That is done with the BlitSurface function of SDL.
4. Finally, this grid is saved into an image and can be displayed in the UI.

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

Figure 23: Solved grid