EPITA

December 10, 2024

OCR Project report n°2

Student team :
Caroline Deliere
Fanette Saury
Jans Guillopé
Lise Suzanne

Contents

1 Introduction

1.1 The project

This document presents the achievement rate of the Organized Chaotic Results team for the second and final OCR project defense. Our team of four EPITA 2nd-year students embarked on a semester-long project to develop a fully automated word search solver using OCR technology, implemented in the C programming language.

This report will dive into various aspects of the project, including challenges faced and features implemented. A detailed code analysis of each feature is provided in its respective section.

1.2 The team members

1.2.1 Fanette SAURY (Project Lead)

I am Fanette, the team leader for this project!
Last year, I also led our S2 project, which taught me a lot about integrating everyone's work habits to collaborate as effectively as possible. I am excited to bring what I learned into this new project, Moreover I feel fortunate to already know the other talented members well.
Over the summer, I assisted a data engineering project where we set up a complete ETL pipeline setup, which has driven my enthusiasm to go all-in on the OCR project. I was particularly eager to work on the UI, having been fascinated by the back-end and front-end development I saw some friends from the 2026 promotion pursed in their ING1 TIGER and PING projects.
Ultimately, I believe this project will be both an enriching learning experience and a rewarding team journey. I am proud of the intermediary deliverables we have produced so far and look forward to the next development period!

1.2.2 Caroline DELIERE

I entered EPITA in 2023 with the motivation to become an engineer in healthcare. I am convinced that combining computer science and health can lead to amazing progress. I would like to study mental disorders, as I am personally affected by a mental health condition, and to research the brain using computers with a team of fellow computer engineers and doctors.
This project is a really interesting exercise because of the neural networks and image recognition aspects it involves.
Teamwork is one of the most important elements in projects, as in life; on our own, it is difficult to progress, as we do not have feedback on what we are.

1.2.3 Jans GUILLOPE

I discovered programming at a very young age. My father, starting in elementary school, gave me the extraordinary ability to write text and transform it into images using Python's Turtle module. I quickly became passionate about programming and started using Scratch, where I created many different types of games. By the time I reached high school, I had already chosen my specialties, focusing on Mathematics and Computer Science. Thus, I mainly learned to code in Python.

Choosing EPITA was, in my opinion, the best decision I could have made. Not only do we learn how to program, but we also develop the skills necessary to become the engineers of tomorrow, particularly through projects and teamwork, which are very formative experiences. I have great ambitions and plan to put in a lot of effort in this project to learn as much as possible, both in programming and in group management.

1.2.4 Lise SUZANNE

Logic is something that has interested me for as long as I can remember. When I first started coding, I knew right away that it was something I would love. Coding gave me a way to use my logic skills to make things work, and I quickly got hooked on projects where I could actually see the results, like creating simple games or working with images. There is something really satisfying about watching your code come to life on the screen. This OCR project has me really excited because it brings together so many interesting elements such as the straightforward logic of building a solver, the various aspects of processing images, and the challenge of neural networks.
I am looking forward to seeing how far we can take this project together as a group !

2 Implementation strategy

The OCR project is scheduled to take place over the duration of the first semester, but spans a period of three months.

2.1 Task repartition

Below are the repartition of the tasks for the two deliveries.
Principal : Led the task.
Secondary : Assisted the principal in his task.

	Caroline	Fanette	Jans	Lise
Image pretreatment			Principal	
Rotation		Principal		
Segmentation			Secondary	Principal
Detection			Principal	Secondary
Neural network	Principal			Secondary
GUI		Principal		
Solver				Principal

Table 1: Task repartition for first delivery

As Caroline became sick shortly after the first delivery, Lise became responsible of the neural network.

	Fanette	Jans	Lise
Image pretreatment		Principal	
Rotation	Principal		
Segmentation		Secondary	Principal
Detection		Principal	Secondary
Neural network			Principal
GUI	Principal		
Solver			Principal

Table 2: Task repartition for final delivery

2.2 Progress rate

Below is the actual progress rate of each task to this day, compared to the expected progress to this day when the team analyzed the given project pipeline.

Task	Expected	Actual
Image pretreatment	100	90
Rotation	100	70
Segmentation	100	100
Detection	100	75
Neural network	100	66
GUI	100	90
Solver	100	100

Table 3: Progress rates for final delivery (in %)

2.3 Project Pipeline

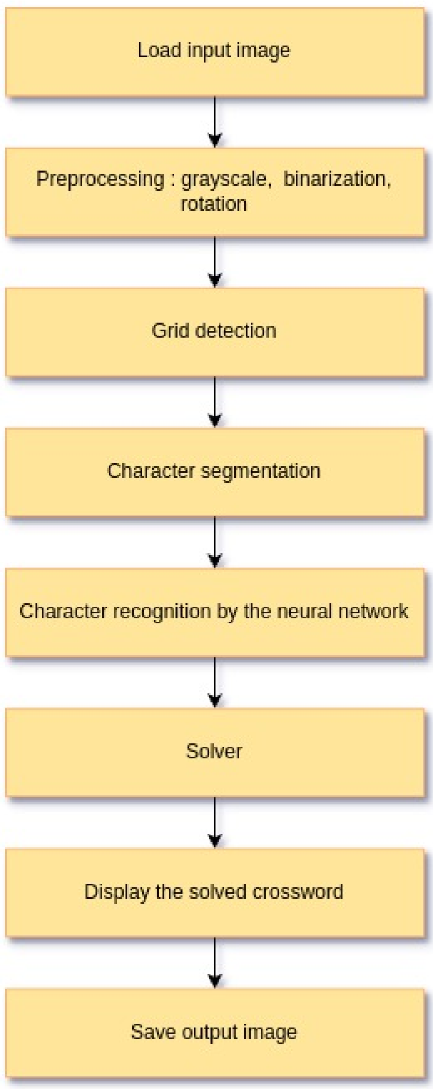

Load input image
Preprocessing : grayscale, binarization, rotation
Grid detection
Character segmentation
Character recognition by the neural network
Solver
Display the solved crossword
Save output image

Figure 1: The OCR pipeline

2.3.1 Required first deliveries

- Load an image and remove colors.
- Manually rotate the image.
- Detect the position of the grid, the word list, the letters in the grid, the words in the list, the letters in the words.
- Segment the image (save each letter as an image).
- Implement a CLI solver algorithm.
- Provide a proof of concept of a neural network capable of learning the logical function $A \cdot B + \overline{A} \cdot \overline{B}$.

2.3.2 Required final deliveries

- Complete pretreatment with automatic rotation.
- Provide a functional neural network.
- Reconstruction of the grid.
- Reconstruction of the word list.
- Resolution of the grid.
- Display of the grid.
- Save of the result.
- A GUI able to showcase all of those elements.

2.3.3 Optional deliveries

- Word Search generator from word list.
- Website.

3 Image pretreatment (Jans & Fanette)

To solve word searches, the neural network's pipeline begins with grayscale conversion to simplify the image data. Next, binarization creates a clear distinction between the dark and light areas, facilitating the identification of the puzzle's structure. Finally, color inversion may be applied, and any necessary rotation for OCR.

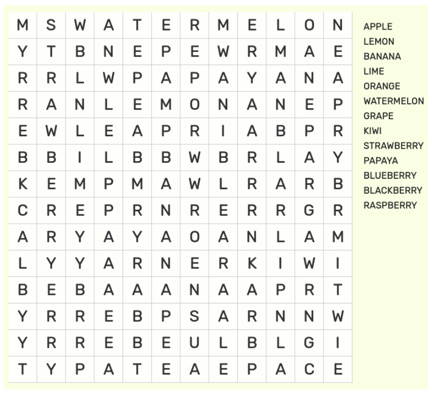
3.1 Grayscale (Jans)

Grayscaling an image involves removing color information, effectively transforming the image into shades of gray. This process is achieved by setting the values of the red (R), green (G), and blue (B) channels to a single, computed grayscale value.

For our application, a simple average-based grayscale conversion is effective. This method balances the lightest and darkest channel values, which is beneficial for later image processing tasks. The average-based grayscale conversion applies the following formula:

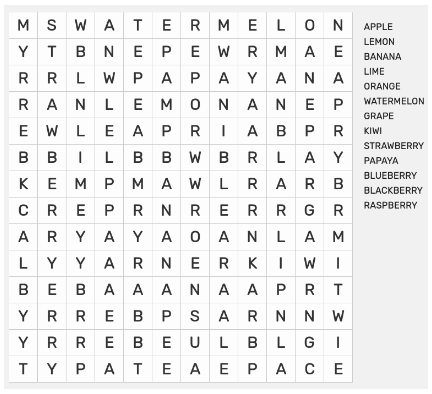
$$Gray = \frac{min(R,G,B) + max(R,G,B)}{2}$$

This method is both simple and computationally efficient, making it ideal for quick grayscale transformations.



M	S	W	A	T	E	R	M	E	L	O	N	APPLE
Y	T	B	N	E	P	E	W	R	M	A	E	LEMON
R	R	L	W	P	A	P	A	Y	A	N	A	BANANA
R	A	N	L	E	M	O	N	A	N	E	P	LIME
E	W	L	E	A	P	R	I	A	B	P	R	ORANGE
B	B	I	L	B	B	W	B	R	L	A	Y	WATERMELON
K	E	M	P	M	A	W	L	R	A	R	B	GRAPE
C	R	E	P	R	N	R	E	R	R	G	R	KIWI
A	R	Y	A	Y	A	O	A	N	L	A	M	STRAWBERRY
L	Y	Y	A	R	N	E	R	K	I	W	I	PAPAYA
B	E	B	A	A	A	N	A	A	P	R	T	BLUEBERRY
Y	R	R	E	B	P	S	A	R	N	N	W	BLACKBERRY
Y	R	R	E	B	E	U	L	B	L	G	I	RASPBERRY
T	Y	P	A	T	E	A	E	P	A	C	E	

(a) RGB



M	S	W	A	T	E	R	M	E	L	O	N	APPLE
Y	T	B	N	E	P	E	W	R	M	A	E	LEMON
R	R	L	W	P	A	P	A	Y	A	N	A	BANANA
R	A	N	L	E	M	O	N	A	N	E	P	LIME
E	W	L	E	A	P	R	I	A	B	P	R	ORANGE
B	B	I	L	B	B	W	B	R	L	A	Y	WATERMELON
K	E	M	P	M	A	W	L	R	A	R	B	GRAPE
C	R	E	P	R	N	R	E	R	R	G	R	KIWI
A	R	Y	A	Y	A	O	A	N	L	A	M	STRAWBERRY
L	Y	Y	A	R	N	E	R	K	I	W	I	PAPAYA
B	E	B	A	A	A	N	A	A	P	R	T	BLUEBERRY
Y	R	R	E	B	P	S	A	R	N	N	W	BLACKBERRY
Y	R	R	E	B	E	U	L	B	L	G	I	RASPBERRY
T	Y	P	A	T	E	A	E	P	A	C	E	

(b) Grayscale

Figure 2: RGB and Grayscale images

3.2 Binarization (Jans)

Binarization is the process of converting a grayscale image into a binary image, where each pixel is assigned one of two values—black or white. This transformation is achieved by applying a threshold to the image, distinguishing pixels above the threshold (white) from those below it (black). Binarization simplifies image content, making it easier for subsequent algorithms to detect and recognize characters, reconstruct grids, and identify words.

In this project, various binarization algorithms were explored and implemented. While some algorithms provided initial results, they had limitations in handling diverse scenarios. Ultimately, we implemented a robust algorithm known as Sauvola's thresholding, which proved to be highly effective.

3.2.1 Fixed Thresholding

Fixed thresholding is the simplest binarization method. A constant threshold value, typically 127, is applied across the entire image. Pixels with intensity values greater than or equal to the threshold are classified as white, while others are classified as black.

This method is computationally efficient but struggles with images having non-uniform lighting or complex backgrounds. Despite these limitations, fixed thresholding served as a baseline for evaluating the performance of more advanced techniques.

3.2.2 Otsu Thresholding

Otsu's method is a global binarization technique that automatically determines an optimal threshold value by minimizing intra-class variance. It assumes a bimodal histogram for the pixel intensities and calculates the threshold that best separates the two peaks.

However, Otsu's method is sensitive to changes in image luminosity. If the image contains uneven lighting or lacks a clear bimodal distribution, the algorithm's performance can degrade significantly, leading to poor binarization results.

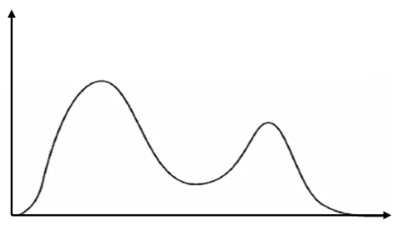


Figure 3: Example of bimodal distribution

3.2.3 Adaptive Thresholding

Adaptive thresholding addresses the limitations of global methods by computing a threshold value for each pixel based on its local neighborhood. This allows the threshold to adapt to variations in lighting and background complexity.

For example, in one test case, an image with a yellow background was successfully binarized, transforming the background into the foreground. By adjusting the reference points dynamically, adaptive thresholding can handle cases where global methods fail, making it suitable for complex and non-uniform images.

3.2.4 Sauvola Thresholding

Sauvola's thresholding is an advanced adaptive binarization technique that calculates a unique threshold for each pixel based on the mean and standard deviation of intensities within a local window. It is specifically designed to handle noisy images and significant changes in luminosity.

To optimize performance, Sauvola's method leverages integral images, which enable efficient computation of local mean and standard deviation values. While the algorithm has higher computational complexity compared to simpler methods, its reliability makes it a superior choice in challenging scenarios. Even in images with significant noise or uneven lighting, Sauvola's method consistently produces high-quality binarization results.

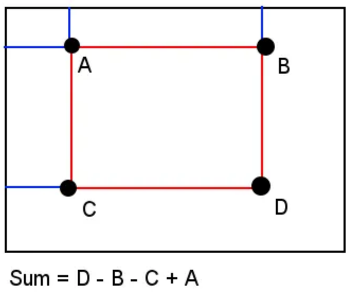
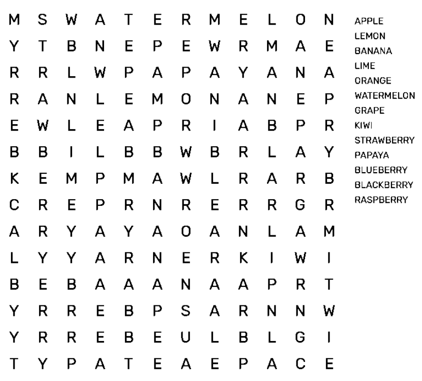


Figure 4: Sum of a region of integral images

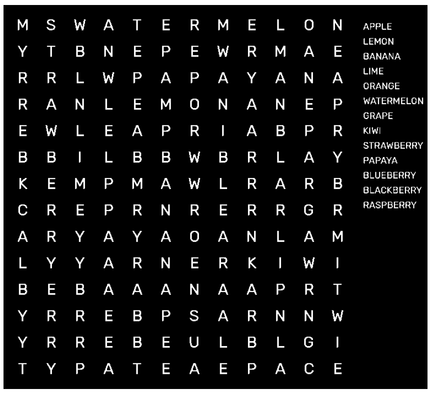
3.3 Inverting pixels (Jans)

Pixel inversion is a process that reverses the color values in an image, effectively creating a "negative" of the original. Inversion simply flips black pixels to white and white pixels to black.

Pixel inversion is used to enhance contrast, emphasize certain image features, and prepare images for specific processing techniques, especially in OCR for the neural networks.



(a) Binarized



(b) Inverted

Figure 5: Binarized and inverted images

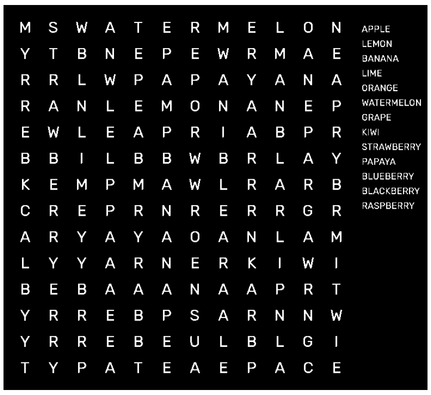
3.4 Noise removal (Jans)

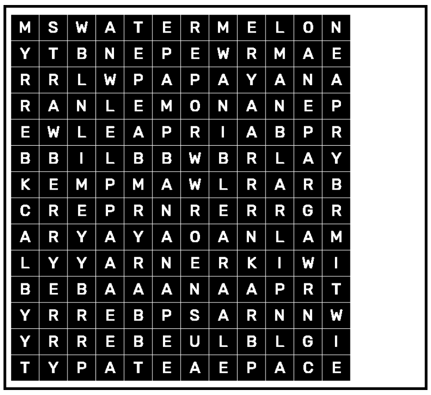
Our noise removal algorithm uses defined thresholds to eliminate pixel clusters that are either excessively large or too small. The process begins with identifying pixel clusters. Initially, we employed a recursive flood-fill approach to analyze each pixel in the foreground (white pixels). However, we have since adopted a sequential labeling algorithm, which efficiently identifies all pixel clusters in two passes through the image.

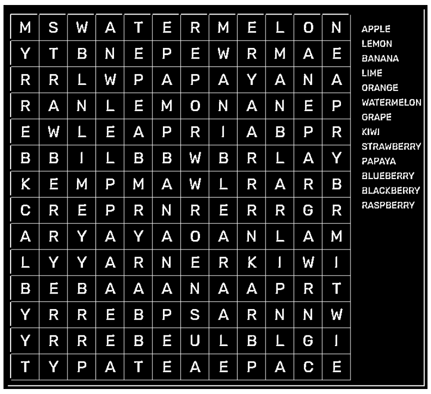
This method addresses potential labeling conflicts that arise when two clusters meet. To resolve these conflicts, we use an equivalence table to store equivalent labels, which are then corrected during the second traversal of the image.

3.5 Binarization results

There is still some residual noise present; however, it is significantly reduced compared to earlier stages. This improvement allows for the analysis of pixel cluster appearance frequency, which can then be utilized to compute the grid.


(a) Average


(b) Global


(c) Adaptative

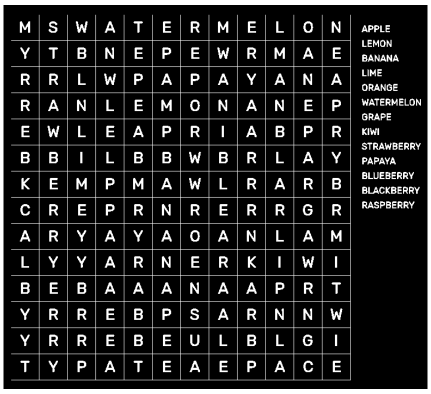
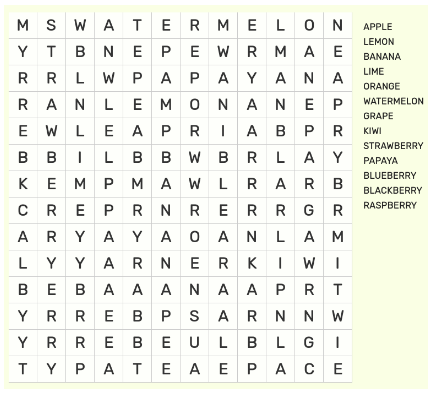
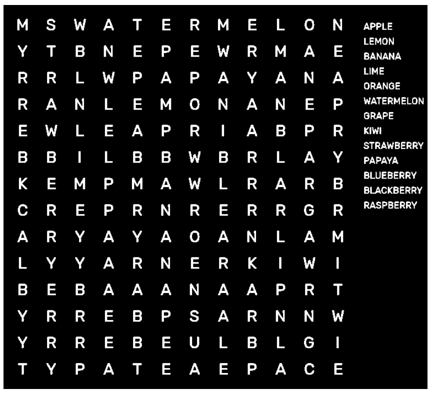

(d) Sauvola

Figure 6: Binarization algorithm comparison on level_1_image_1

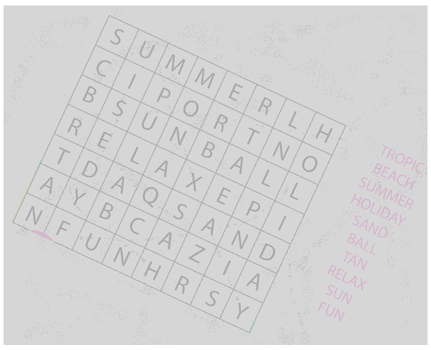


(a) Plain

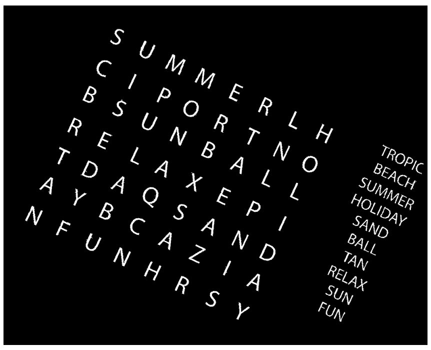


(b) Processed

Figure 7: Sauvola processing and filtering on level_1_image_1

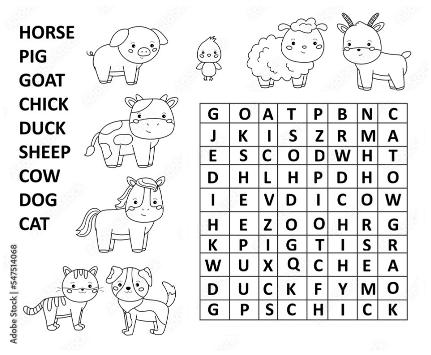


(a) Plain

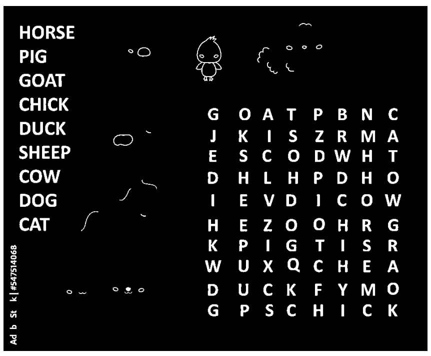


(b) Processed

Figure 8: Sauvola processing and filtering on level_1_image_2



(a) Plain



(b) Processed

Figure 9: Sauvola processing and filtering on level_4_image_2

3.6 Manual rotation (Fanette)

For the chosen image rotation algorithm, the main steps involve calculating transformed coordinates, ensuring bounds, and performing bilinear interpolation.

3.6.1 Coordinate transformation

To rotate each pixel in the destination image back to its source location, we apply an inverse rotation matrix. If the rotation angle is θ (in radians), the source coordinates $(x_{\text{src}}, y_{\text{src}})$ for each destination pixel (x, y) are calculated as:

$$x_{\text{src}} = (x - n_{\text{cx}})\cos(-\theta) - (y - n_{\text{cy}})\sin(-\theta) + c_x$$

$$y_{\text{src}} = (x - n_{\text{cx}})\sin(-\theta) + (y - n_{\text{cy}})\cos(-\theta) + c_y$$

where n_{cx} and n_{cy} are the center coordinates of the destination image, and c_x and c_y are the center coordinates of the source image. This mapping reverses the rotation, ensuring that each pixel in the rotated image aligns with its corresponding source position.

3.6.2 Bilinear interpolation

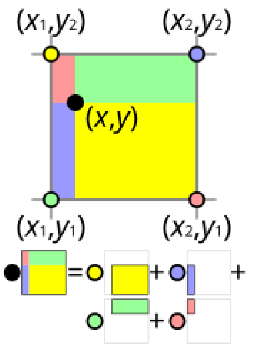


Figure 10: Bilinear transformation. Source: Wikipedia

To create an image with the least possible artifacts, bilinear interpolation is used. For each calculated position $(x_{\text{src}}, y_{\text{src}})$, the pixel values are interpolated based on the four neighbor source pixels. If $(x_{\text{src-int}}, y_{\text{src-int}})$ is the integer part of $(x_{\text{src}}, y_{\text{src}})$, and (dx, dy) are the fractional parts, the interpolated pixel value P is given by:

$$P = (1 - dx)(1 - dy) \cdot p_1 + dx(1 - dy) \cdot p_2 + (1 - dx)dy \cdot p_3 + dxdy \cdot p_4$$

where p_1, p_2, p_3, and p_4 are the intensities of the four nearest pixels (top-left, top-right, bottom-left, and bottom-right).

3.6.3 Source image pixel boundaries

To avoid accessing pixels outside the source image boundaries, a condition ensures that $(x_{\text{src}}, y_{\text{src}})$ remains within the valid pixel range. Specifically, $0 \leq x_{\text{src}} < \text{width}_{\text{src}} - 1$ and $0 \leq y_{\text{src}} < \text{height}_{\text{src}} - 1$ prevent out-of-bounds errors. This boundary check is critical for safe interpolation near the edges of the source image.

These formulas collectively handle rotation, interpolation, and boundary conditions, ensuring an accurate and smooth rotated output image.

3.7 Automatic rotation (Fanette)

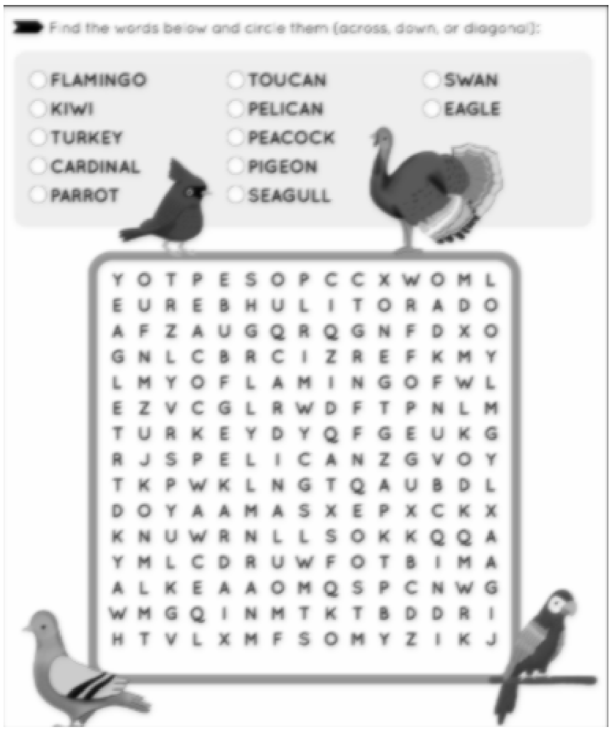
The automatic grid rotation relies on detecting the primary angle of the input image using the Hough Transform. This method identifies dominant lines in the grid and calculates the most prevalent angle. Correcting the grid's orientation ensures that subsequent steps, such as edge detection and word identification, are performed accurately.

3.7.1 Canny edge detection algorithm

This algorithm effectively identifies edges in the input image, simplifying the Hough Transform's task of detecting straight lines. The implementation of the Canny algorithm is fully functional and helps visualize the grid structure. However, its potential is limited without the assembled pipeline.

Step 1 : Grayscale Conversion The first step in the Canny edge detection process is to convert the input image to grayscale. This simplifies the image by reducing it to a single intensity channel, with pixel values ranging from 0 to 255. By focusing on intensity rather than color, the algorithm can process the image more efficiently.

Step 2 : Gaussian Blur Once the image is converted to grayscale, Gaussian blur is applied to reduce noise. Noise in an image can interfere with edge detection by creating false edges. The Gaussian filter smooths the image by averaging pixel values in a point-spread area, which helps reduce high-frequency noise. The sigma, inferred in the mask, is of about 1.4.



Find the words below and circle them (across, down, or diagonal):

FLAMINGO	TOUCAN	SWAN
KIWI	PELICAN	EAGLE
TURKEY	PEACOCK	
CARDINAL	PIGEON	
PARROT	SEAGULL	

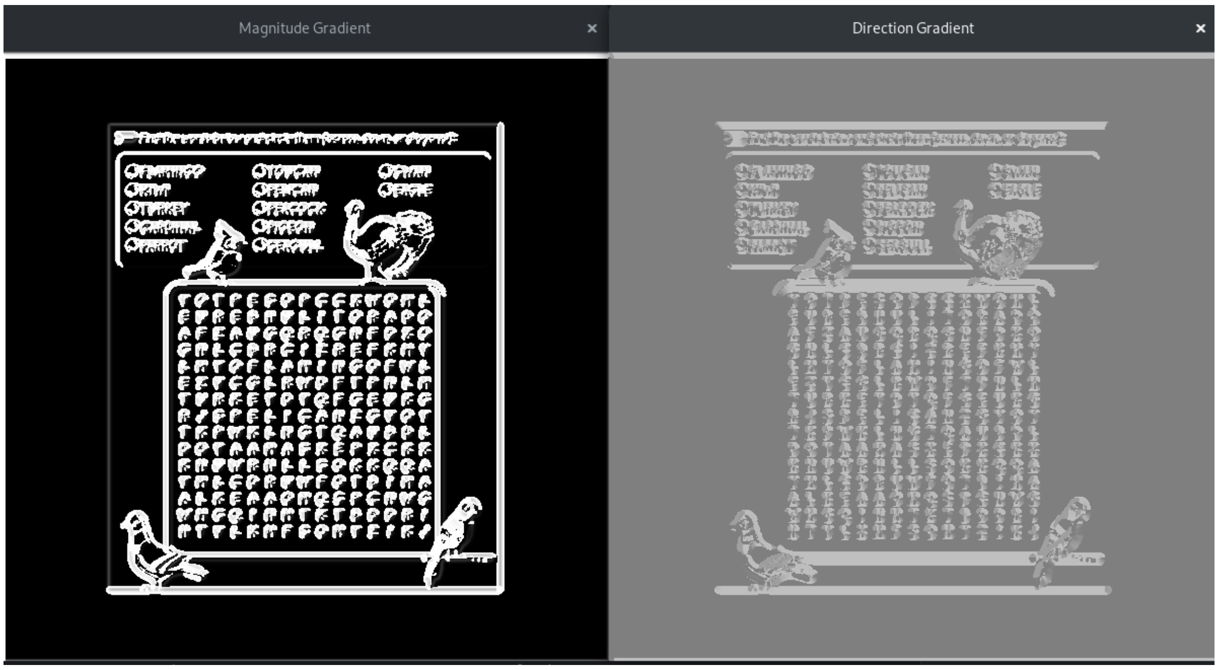
```
Y O T P E S O P C C X W O M L
E U R E B H U L I T O R A D O
A F Z A U G Q R Q G N F D X O
G N L C B R C I Z R E F K M Y
L M Y O F L A M I N G O F W L
E Z V C G L R W D F T P N L M
T U R K E Y D Y Q F G E U K G
R J S P E L I C A N Z G V O Y
T K P W K L N G T Q A U B D L
D O Y A A M A S X E P X C K X
K N U W R N L L S O K K Q Q A
Y M L C D R U W F O T B I M A
A L K E A A O M Q S P C N W G
W M G Q I N M T K T B D D R I
H T V L X M F S O M Y Z I K J
```

Step 3 : Sobel filters The "intersity gradients" of the image are determined using Sobel filters, which compute changes in pixel intensity along the horizontal (G_x) and vertical (G_y) directions. An edge is identified where there is a significant change in intensity, as these changes correspond to the boundaries of objects in the image. The Sobel filters for G_x and G_y are defined as follows:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Then we calculate the magnitude and angle of the directional gradients:

$$|G| = \sqrt{G_x^2 + G_y^2}, \quad \angle G = \arctan\left(\frac{G_y}{G_x}\right)$$



The gradient magnitude highlights areas with significant intensity changes, while the gradient direction represents the orientation of those changes. This step outputs a gradient magnitude image and a direction image, which are crucial for detecting edges in subsequent steps.

Step 4 : Non-maxima suppression The gradient magnitude image often contains thick edges, which need to be thinned for better detection. Non-maximum suppression preserves only the pixels that form the local maxima in the gradient direction. We chose non-maxima suppression without interpolation, which requires us to divide the 3x3 grid of neighboring pixels into 8 sections. Those are described on the figure below.

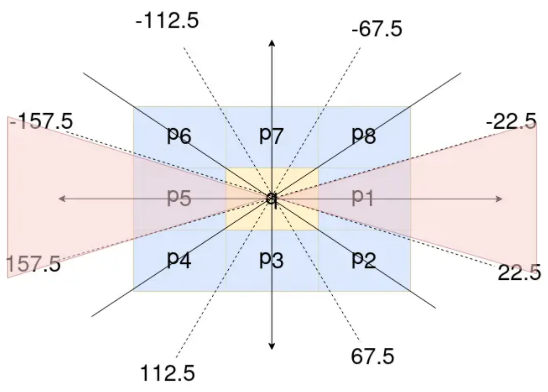
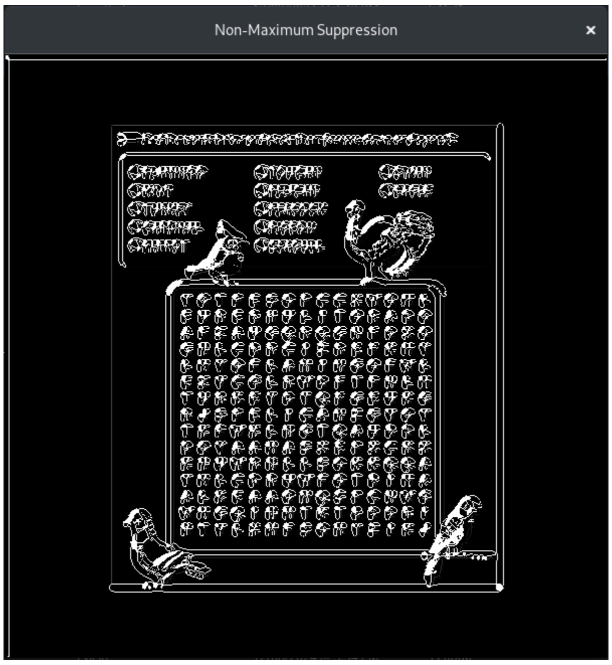


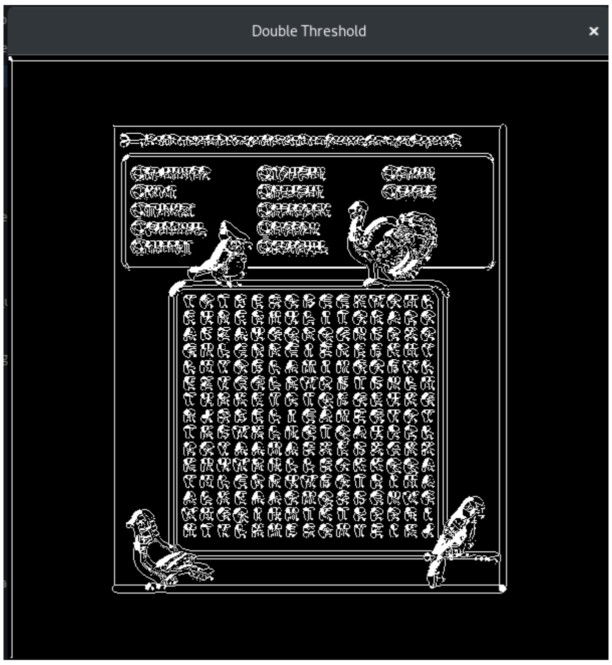
Figure 11: Non maxima pixel suppression directions. Source: Anil K. Shrestha



Step 5 : Double Thresholding Double thresholding classifies pixels into three categories based on their intensity relative to the high and low thresholds. We chose a low threshold = 0.3 and high threshold = 0.7 after trial and error.

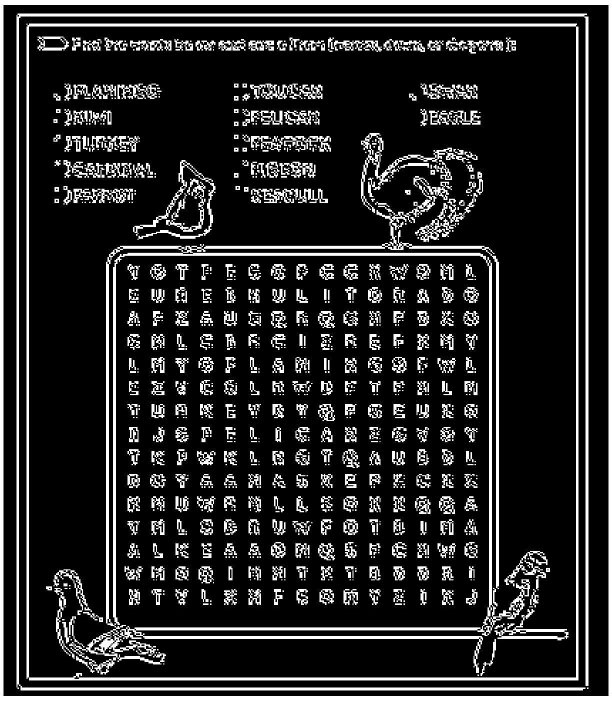
- **Strong edges:** Pixels with intensity values greater than the high threshold. These are highly likely to represent actual edges and are retained as part of the final edge map.
- **Weak edges:** Pixels with intensity values between the high and low thresholds. These may or may not represent actual edges. Their classification depends on their connectivity to strong edges, which is addressed in the next step (edge tracking by hysteresis).
- **Non-edges:** Pixels with intensity values below the low threshold. These are considered to be background or noise and are discarded by setting their values to zero.

This categorization simplifies the edge detection process by focusing on strong and weak edges while eliminating low-intensity edges.



Step 6 : Edge tracking by Hysteresis Now that we have determined what the strong edges and weak edges are, we need to determine which weak edges are actual edges. To do this, we perform an edge tracking algorithm. Weak edges that are connected to strong edges will be actual/real edges. Weak edges that are not connected to strong edges will be removed.

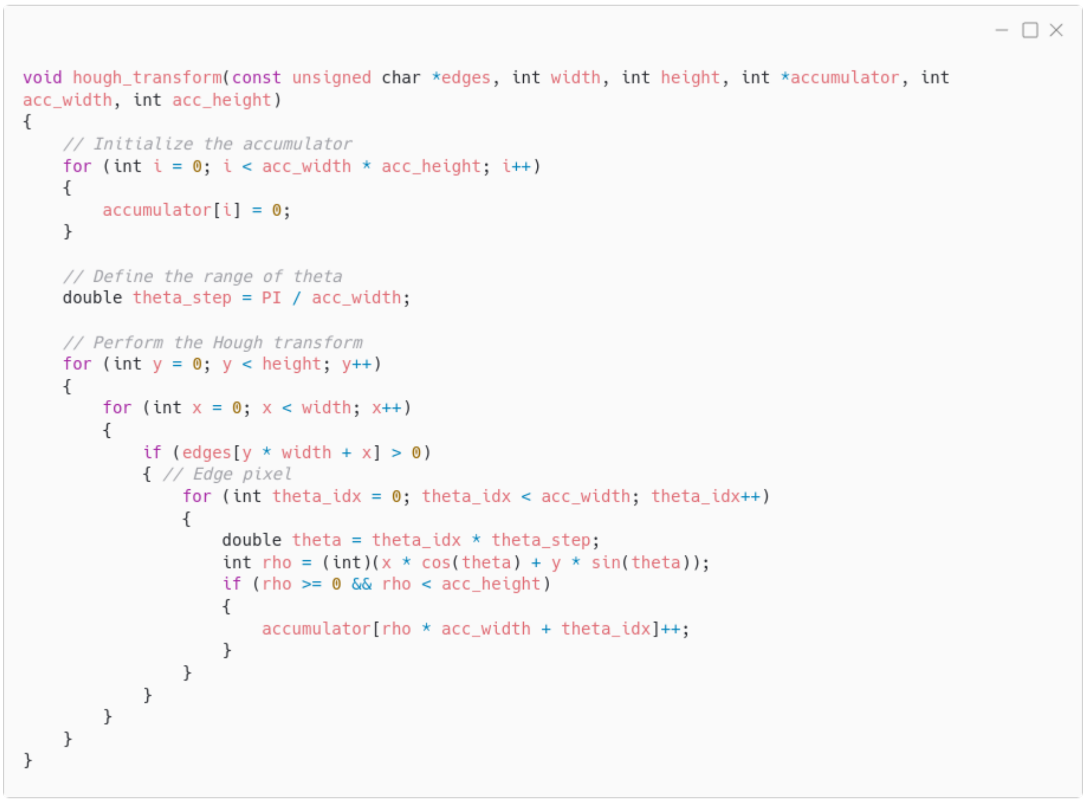
Final result The algorithm works on all levels and images. Here the assembled Canny edge algorithm on the example image!



3.7.2 Hough Transform

After applying the Canny edge detection algorithm to highlight the edges in the image, the Hough Transform is used to identify straight lines that represent structural elements, such as text baselines or page edges. The Canny algorithm outputs a binary edge map, which serves as input to the Hough Transform. The transform works by mapping edge points into a parameter space, where clusters of points correspond to straight lines in the image.

Using this method, the dominant lines in the document are detected, and their angles are analyzed to compute the skew of the image. This skew angle is essential for correcting the orientation of the grid in the image. Below is the code.



```c
void hough_transform(const unsigned char *edges, int width, int height, int *accumulator, int
acc_width, int acc_height)
{
    // Initialize the accumulator
    for (int i = 0; i < acc_width * acc_height; i++)
    {
        accumulator[i] = 0;
    }

    // Define the range of theta
    double theta_step = PI / acc_width;

    // Perform the Hough transform
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            if (edges[y * width + x] > 0)
            { // Edge pixel
                for (int theta_idx = 0; theta_idx < acc_width; theta_idx++)
                {
                    double theta = theta_idx * theta_step;
                    int rho = (int)(x * cos(theta) + y * sin(theta));
                    if (rho >= 0 && rho < acc_height)
                    {
                        accumulator[rho * acc_width + theta_idx]++;
                    }
                }
            }
        }
    }
}
```

Figure 12: Hough transformation function

3.7.3 Finding the angle

Using this method, the dominant lines in the document are detected, and their angles are analyzed to compute the skew of the image. This skew angle is essential for correcting the orientation of the document. The computed angle is then passed to the rotation algorithm, detailed in the previous section, where bilinear interpolation is used to apply the necessary rotation. However, I didn't have time to spend enough energy on the testing of the automatic rotation..

4 Detection (Jans)

In this project, detection involves identifying the boundaries of characters, the grid, individual words, and the word list. This step is crucial for enabling accurate segmentation and optimal reconstruction of the matrix that represents the grid.

4.1 Character detection

The first step in the detection process is character detection, which is the most critical task, as all subsequent detection activities rely on its accuracy. Successfully identifying individual characters is essential for establishing the boundaries of words and the grid structure. However, this task can be complicated by the presence of noise, which may obscure or distort characters, leading to potential inaccuracies in detection. Therefore, implementing robust techniques to enhance character visibility and mitigate noise is vital for achieving reliable results in the overall detection process. This emphasizes the importance of pretreatment in preparing the image for accurate character recognition.

To accurately detect characters on a fully preprocessed image, the following steps are implemented:

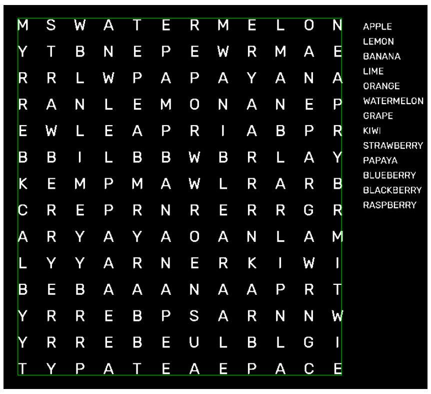
1. **Iterate through all pixels** in the image
2. **If a white pixel is found:**
   - Check if this pixel is already included in a bounding box of a character; if so, skip to the next pixel.
   - If the pixel is not part of an existing bounding box, use the flood fill algorithm to delineate the bounding box of the detected character.
3. **Save all resulting bounding boxes** in an array for further processing

4.2 Grid detection

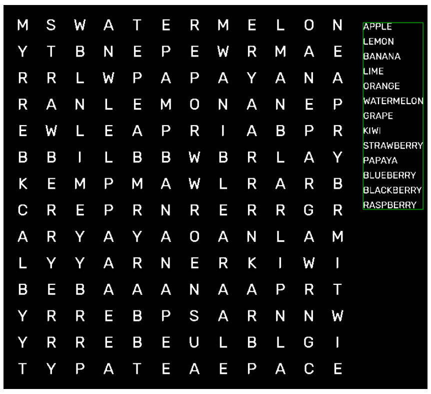
The second step in the detection process is detecting the grid which require the most computations involving histograms, array conversions and thresholding techniques. Below is an outline of the general algorithm used for grid detection:

1. **Retrieve all characters** using the previous detection algorithm
2. **Compute histograms:**
   - For each index i, calculate the histogram representing the number of characters that have at least one pixel aligned with the x-axis at position i
   - Similarly, compute a histogram for the y-axis, representing the number of characters that have at least one pixel aligned with the y-axis at position j
3. **Calculate the most probable number of columns and rows** based on the computed histograms
4. **Sort all character bounding boxes** first by their Y-coordinate and then by their X-coordinate

5. **Search for probable characters** for each pair of matrix indexes
6. **Determine the top-left and bottom-right corners** of the identified matrix to obtain the bounding box of the grid



(a) Grid detection



(b) Word list detection

Figure 13: Detection algorithms

4.3 Word list detection

The third step in the detection process is detecting the list of words. Below is an outline of the general algorithm used for detecting the list of words:

1. **Remove all grid characters** from the set of detected characters
2. **Identify remaining characters**, which correspond to the list box characters
3. **Obtain the bounding box** for the list of words based on the remaining characters

4.4 Word detection

The final step in the detection process is identifying each word within the list of words. The following outlines the general algorithm used for this detection:

1. **Filter character boxes:** create an array of bounding boxes that excludes characters found within the grid bounding box
2. **Create a histogram** for the bounding boxes of the words to analyze their distribution
3. **Count the number of non-zero entries** in the histogram to determine the total number of words
4. **Construct bounding boxes** for each word
    - For each word, find and aggregate its character bounding boxes based on their positions
    - Create a bounding box for each complete word based on the first and last character bounding boxes

4.5 Final results

Our character detection algorithm occasionally identifies noise, which we attempt to filter out using fixed thresholds for cluster size and pixel count. Despite extensive efforts to mitigate noise, none of the subsequent algorithms successfully resolved the issue. Below is a detailed description of two of the unsuccessful attempts that were fully implemented and tested:

1. **Dilation method**
   - **Approach**
     We applied morphological dilation to the image to merge closely positioned characters, effectively forming clusters representing words. The bounding boxes of these merged clusters were then analyzed to detect the words. The cluster appearance frequency algorithm was subsequently applied to the remaining clusters to identify the grid structure.
   - **Outcome**
     While the concept appeared robust, the presence of noise interfered with the detection of the grid. The irregularities introduced by noise rendered the cluster appearance frequency algorithm ineffective.
2. **Average Distance Between Vertices**
   - **Approach**
     Clusters were represented by their center points (vertices), and the closest distances between all vertices were calculated. Points that were too far from others were removed as outliers.
   - **Outcome**
     Noise within the grid led to false detections of "closest elements," significantly affecting the accuracy of the algorithm. The method failed to reliably isolate valid grid elements due to this interference.

In our exploration of character detection, we developed and tested numerous algorithms, each demonstrating effectiveness in specific cases but falling short in others. The primary challenge was finding a single solution capable of handling all scenarios reliably. Our current detection algorithm performs well on fully preprocessed and denoised images, proving its potential under optimal conditions.

By integrating the strengths of our failed attempts, we could have designed a highly robust detection algorithm capable of addressing a broader range of cases. However, the time constraints of the OCR project limited our ability to implement and refine these combined approaches. This highlights the potential for future development to improve the algorithm's versatility and overall performance.

5 Segmentation (Lise)

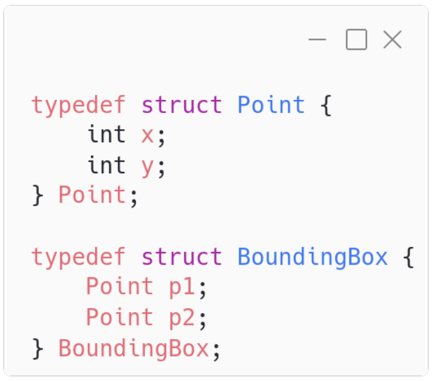
The segmentation heavily relies on the detection part. It is used when an area has been defined after the character detection.

5.1 Creating a new surface

We implemented a function with the following prototype:

- `SDL_Surface *cut(SDL_Surface *surface, BoundingBox *box);`

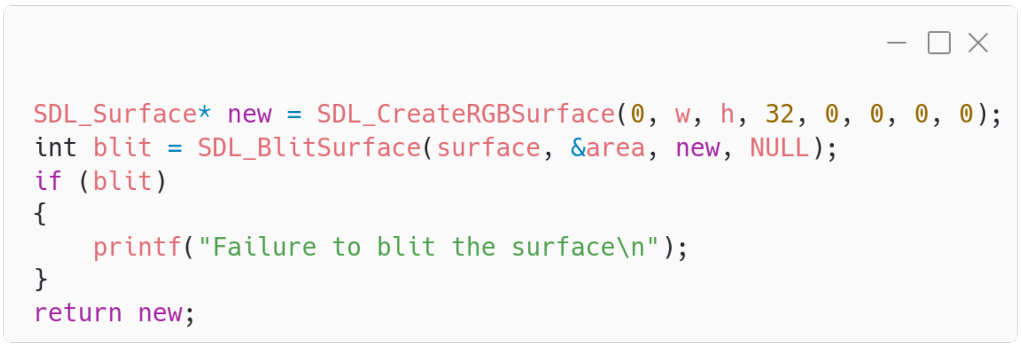
It calls the structure defined below, with `p1` being the top-left corner and `p2` the lower right corner of an area.



```
typedef struct Point {
    int x;
    int y;
} Point;

typedef struct BoundingBox {
    Point p1;
    Point p2;
} BoundingBox;
```

Figure 14: Segmentation structures

The idea is pretty simple: we take an image along with an area, and we copy said area of the image into a new image. To do so, we use two SDL functions: `SDL_CreateRGBSurface`, which creates a new surface, and `SDL_BlitSurface`, which will copy an area of a surface into another surface.



```c
SDL_Surface* new = SDL_CreateRGBSurface(0, w, h, 32, 0, 0, 0, 0);
int blit = SDL_BlitSurface(surface, &area, new, NULL);
if (blit)
{
    printf("Failure to blit the surface\n");
}
return new;
```

Figure 15: SDL_BlitSurface usage

At the end, we simply return the new surface, which now contains the area we wanted to copy.

5.2 Converting to PNG

After obtaining the new surface with the `cut` function, we need to save it as an image for later use.
This function calls cut with the specified parameters, and then saves the resulting surface in a designated folder as a PNG file. To convert the surface into a PNG file, we use the function `IMG_SavePNG`, which takes the surface and a path to where it will be stored.

5.3 Link with the neural network

5.3.1 Implemented

We implemented 2 functions in `source/neural_network/convertion.c` to be able to take a .png image and to return a list of 0s and 1s :

- `SDL_Surface *toSDL(char *path);`
- `void SDL_to_list(SDL_Surface *surface, int len, double **list);`

The `toSDL` function takes a path to an image and convert it to a surface. Then, it resizes it to a specific dimension using the functions `sdl_surface_to_gdk_pixbuf`, `resize_pixbuf` and `gdk_pixbuf_to_sdl_surface`. Finally, the function converts the format of the surface to `SDL_PIXELFORMAT_INDEX8`, which is more efficient in our case as we are working with inverted and greyscaled images.
The `SDL_to_list` function takes a surface and a list to fill. It will parse through the surface, and fill the list with either a 0 if the pixel is black, or 1 if the pixel is white.

5.3.2 Not implemented

We did not have time to implement the rest of the process, but we wanted to explain how we would have done it.

The idea was as follows:
We would convert each bounding box of the grid (respectively, of the list) to a surface using the function `cut`. Then, we would convert the surface to a list of 0s and 1s using the function `SDL_to_list` implemented in `source/neural_network/convertion.c`. This list would then be given to the neural network, which would give back the character it found. Then, we would create a file for the grid (respectively, the list) and fill it with the different characters found.

This process was thought of in order to have a smooth transition between the segmentation and the neural network, and to create the files needed afterwards for the solver.

6 Neural network (Caroline & Lise)

6.1 NXOR (Caroline)

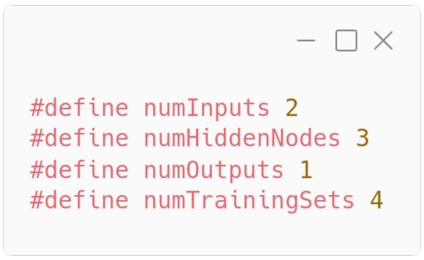
6.1.1 Approach of the neural network

When I first researched information about neural networks, there was a lot. On one hand, there was enough to understand what a neural network is, and on the other hand, too much to lose me about it. After many research sessions and a lot of time, I finally could code some things.

Firstly, a neural network is a computer system inspired by the animal (such as human) brain and nervous system. So, as in biology, we have to create some neurons, train them in different ways, and compute the error margin to adjust for the next training.

6.1.2 Not Xor proof of concept

Neurons are composed of a random weight and a random bias. They are part of a layer with hidden nodes. We defined in the code some variables such as ***numInputs*** and ***numOutputs*** (for the number of inputs and outputs a neuron has), ***numHiddenNodes*** (number of hidden nodes in the layer), and ***numTrainingSets*** (number of sets for the training part).

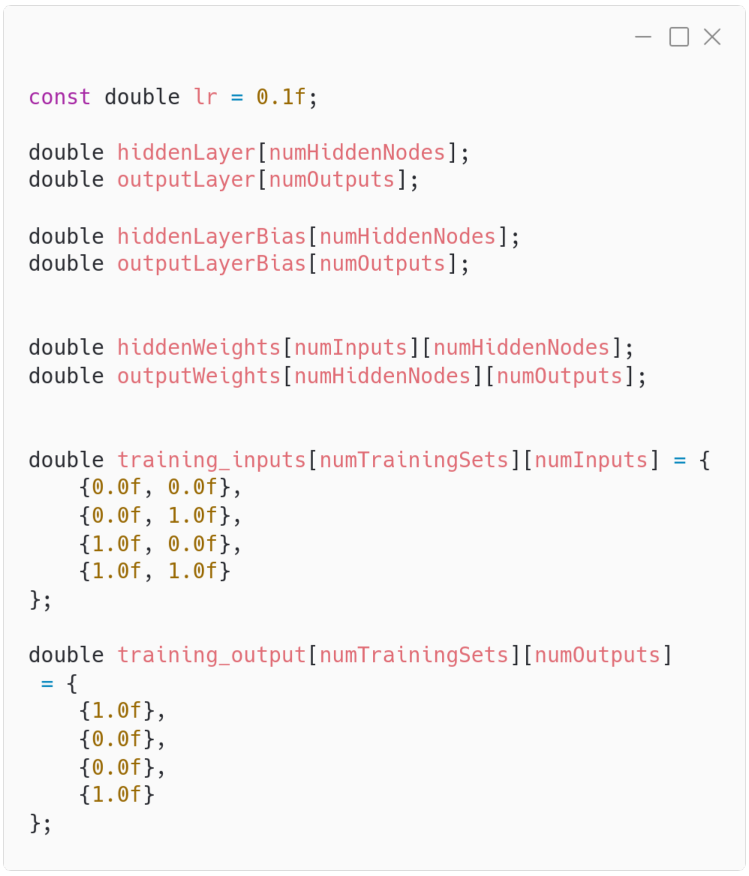


```
#define numInputs 2
#define numHiddenNodes 3
#define numOutputs 1
#define numTrainingSets 4
```

Figure 16: The main properties are macros

In the *main()* function, we define various arrays to represent neurons, along with a learning rate *lr* that plays a crucial role in optimizing the neural network. This learning rate determines the magnitude of the adjustments made to the network's weights during training, while minimizing the value of the loss function.

Let's define the elements on the image containing code below. The *training_ inputs* are used to represent all the combinations of 0 and 1. The *training_ outputs* are the expected results for all the combinations of the *training_ inputs*. In this case, they represent the **NOT XOR** operation.



```
const double lr = 0.1f;

double hiddenLayer[numHiddenNodes];
double outputLayer[numOutputs];

double hiddenLayerBias[numHiddenNodes];
double outputLayerBias[numOutputs];


double hiddenWeights[numInputs][numHiddenNodes];
double outputWeights[numHiddenNodes][numOutputs];


double training_inputs[numTrainingSets][numInputs] = {
    {0.0f, 0.0f},
    {0.0f, 1.0f},
    {1.0f, 0.0f},
    {1.0f, 1.0f}
};

double training_output[numTrainingSets][numOutputs]
 = {
    {1.0f},
    {0.0f},
    {0.0f},
    {1.0f}
};
```

Figure 17: Overview

After those lines, we initialize the *hiddenWeights*, the *outputWeights*, and the *outputLayerBias* with random values using *rand*() and RAND_MAX from the *<stdlib.h>* library.
We initialized the number of epochs of the training; here it is set to 100,000. Then, for every epoch, we shuffled the order of the training inputs.

6.1.3 Forward propagation

For every `numTrainingSets`, we first perform a forward propagation: for all `numHiddenNodes`, we compute the sum of the `training_inputs` times its respective `hiddenWeights`, adding the `hiddenLayerBias` associated with the current hidden node. Then we update the `hiddenLayer` by applying the `sigmoid()` function to the previously computed number.

Still in the forward propagation, for every `numOutputs`, we add the `hiddenLayerBias` of the current output to the sum of all `hiddenLayer` values times their respective `outputWeights` for every `numHiddenNodes`. We apply the `sigmoid()` on this number and update the `outputLayer` with it.

6.1.4 Backward propagation

Next, we perform the backward propagation, which computes the change in output weights in `deltaOutput`, an array of `numOutputs` elements. For every output, we compute the error, which is the `training_output` minus the associated `outputLayer` value. We store this error in `deltaOutput`, multiplied by `dSigmoid()` of the `outputLayer`. We then repeat a similar process for the hidden weights.

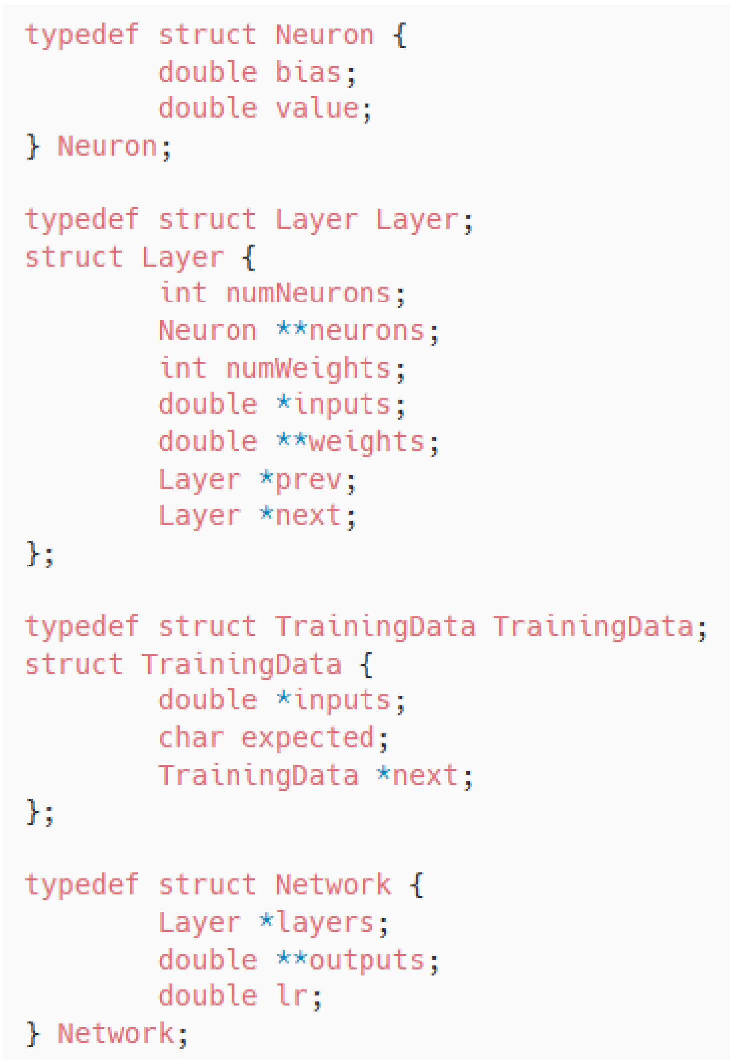
Finally, we apply the changes in the output weights. For every `numOutputs` (resp. `numHiddenNodes`), we add to the `outputLayerBias` (resp. `hiddenLayerBias`) the `deltaOutput` (resp. `deltaHidden`) times `lr`, then add the sum of all `hiddenLayer` values times the `deltaOutput` (resp. `deltaHidden`) times `lr`.

6.2 Character Recognition (Lise)

The original idea was to start from the neural network of the NXOR operation, and to transform it in order to be able to recognize letter instead. However, it appeared that the original code was not really functional, and thus we decided to rewrite all the functions.

6.2.1 Structuring the network

To better work on the neural network, and to facilitate the different operation we had to do, we implemented the following structures :



```c
typedef struct Neuron {
        double bias;
        double value;
} Neuron;

typedef struct Layer Layer;
struct Layer {
        int numNeurons;
        Neuron **neurons;
        int numWeights;
        double *inputs;
        double **weights;
        Layer *prev;
        Layer *next;
};

typedef struct TrainingData TrainingData;
struct TrainingData {
        double *inputs;
        char expected;
        TrainingData *next;
};

typedef struct Network {
        Layer *layers;
        double **outputs;
        double lr;
} Network;
```

Figure 18: Structures

Those structures are used throughout almost every files in `source/neural_network/`. They are the core of our code.
A `Neuron` has a bias and a value. The `Layer` is a structure of a double linked list, where each layer has `numNeurons Neuron`, `numWeights` inputs and a matrice of weigths. The `Network` is a structure with a pointer to the first layer, a list of outputs and a learning rate.
Finally, the `TrainingData` structure is a linked list which contains a list of 0s and 1s

representing the image to study, and the expected character.
We decided to create a specific file to store every function concerning the setup of the network. The `source/neural_network/setup.h` contains the following :

```c
#define nLayers 2
#define nInputs (dimension * dimension)
#define nNodes 50
#define nOut 26

double GetMax(double x, double y);
double RandFrom(double min, double max);
void InitWeigths(Layer *layer);
void InitBiases(Layer *layer);
void RecoverWeigths(Layer l, int rows, int cols, double mat[rows][cols]);
void RecoverBiases(Layer l, int nodes, double arr[nodes]);
void PrintData(Network net);
Neuron *CreateNeuron();
void DestroyNeuron(Neuron *n);
Layer *CreateFirstLayer(int ni, int nn);
Layer *CreateLayer(Layer *l, int nn);
void SaveLayers(Layer *first);
Layer *RecoverFirstLayer(const char *fweight, const char *fbias);
void RecoverSecondLayer(Layer *l, const char *fweight, const char *fbias);
void DestroyLayer(Layer *layer);
TrainingData *CreateData();
TrainingData *LinkData(TrainingData *d1, TrainingData *d2);
void DestroyData(TrainingData* data);
Network *CreateNet(int numLayers, int lr);
Network *RecoverNet(const char *fw1, const char *fb1,
                const char *fw2, const char *fb2);
void DestroyNet(Network *network);
```

Figure 19: setup.h

The list is pretty long, but all of them are necessary for the network to work. The main point of each of those functions can be deduced from their name.

6.2.2 Implementing the network

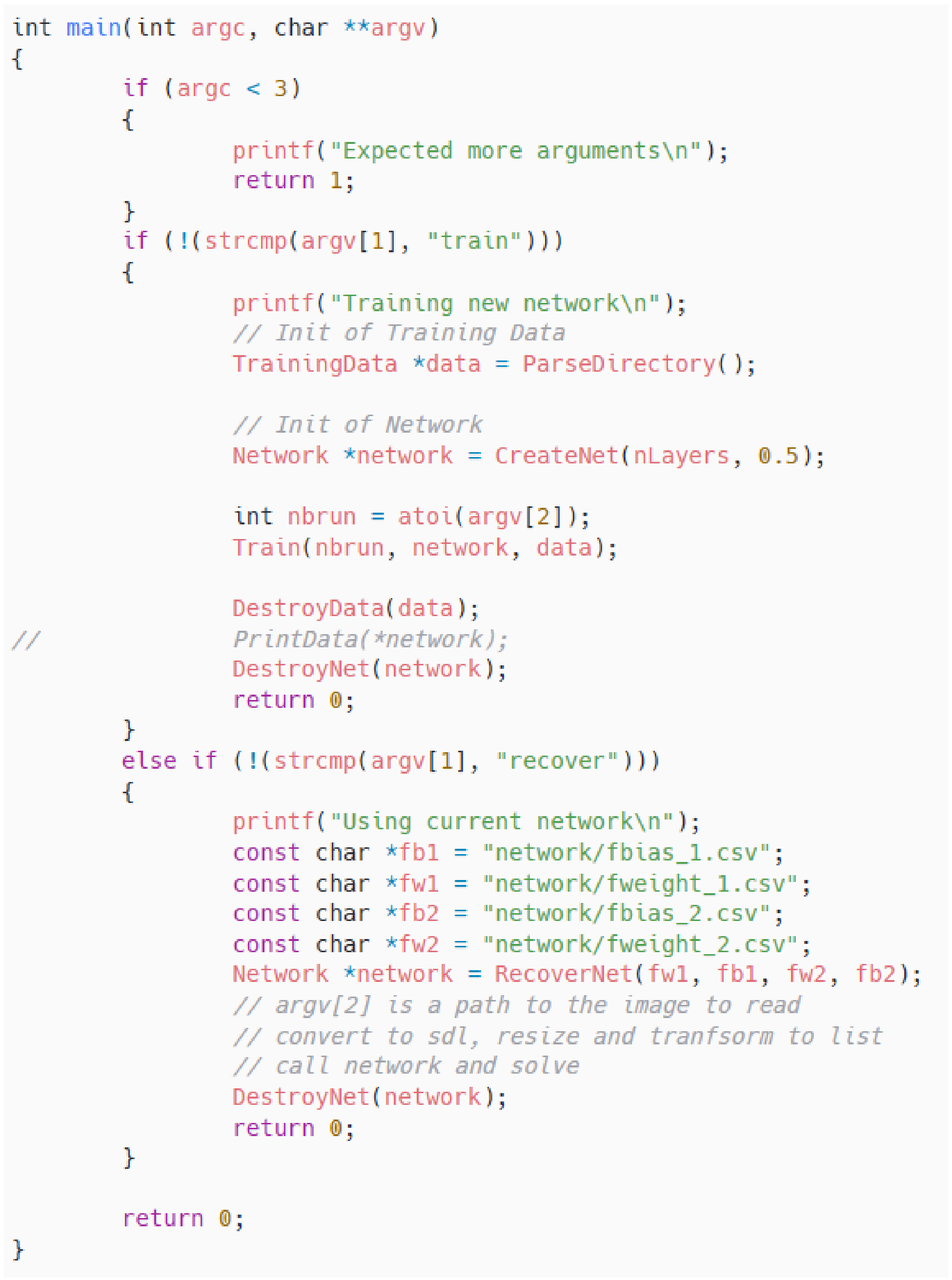
The main file of the neural network part is `source/neural_network/neural.c`. It is where we can find all the functions concerning the actual implementation of the network. You can find below the most important functions of this file.

```c
void Forward(int length, Network *net, int i);

void Backward(Network *net, TrainingData data, int i);

double Result(TrainingData *data, Network net, int run);

void Train(int nbrun, Network *net, TrainingData *data);

void Find(Network *net);
```

Figure 20: neural.h

`Train` is the function which will loop `nbrun` times over the list of `TrainingData`, calling for each of them `Forward` and `Backward`. It will be used to, well, train the network. The `Find` function is a function that will "only" call `Forward`, and not `Backward`.

The executable `neural` will refer to the following main :



```c
int main(int argc, char **argv)
{
        if (argc < 3)
        {
                printf("Expected more arguments\n");
                return 1;
        }
        if (!(strcmp(argv[1], "train")))
        {
                printf("Training new network\n");
                // Init of Training Data
                TrainingData *data = ParseDirectory();

                // Init of Network
                Network *network = CreateNet(nLayers, 0.5);

                int nbrun = atoi(argv[2]);
                Train(nbrun, network, data);

                DestroyData(data);
//              PrintData(*network);
                DestroyNet(network);
                return 0;
        }
        else if (!(strcmp(argv[1], "recover")))
        {
                printf("Using current network\n");
                const char *fb1 = "network/fbias_1.csv";
                const char *fw1 = "network/fweight_1.csv";
                const char *fb2 = "network/fbias_2.csv";
                const char *fw2 = "network/fweight_2.csv";
                Network *network = RecoverNet(fw1, fb1, fw2, fb2);
                // argv[2] is a path to the image to read
                // convert to sdl, resize and tranfsorm to list
                // call network and solve
                DestroyNet(network);
                return 0;
        }

        return 0;
}
```

Figure 21: neural.c/main

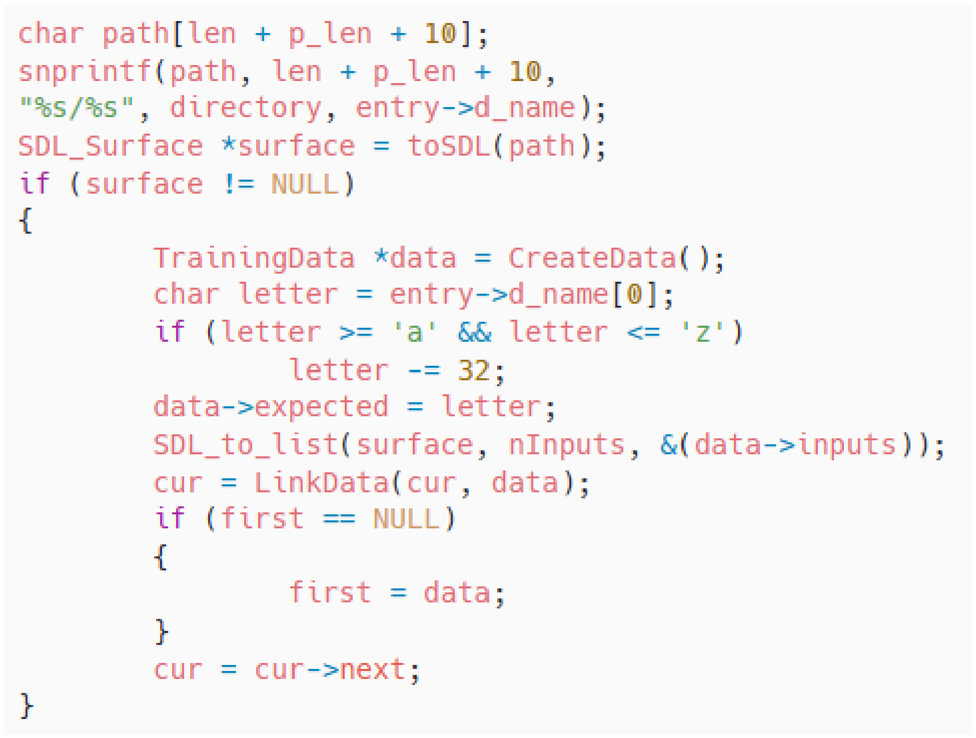
By calling `./neural train <number of trainings to do>`, it will create a new network and train it. The idea was then to save the network using csv files (as explained later). Unfortunately, we did not have the time to finish linking everything.
By calling `./neural recover <path to the image>`, we recover the previous network and use it to find out what letter is represented in the image sent in.

6.2.3 Training the network - Parsing directories

To train our network, we implented the TrainingData structure. It is initialized thanks to the 2 following functions :

- `TrainingData *ParseDirectory();`
- `TrainingData *ReadDirectory(const char *directory);`

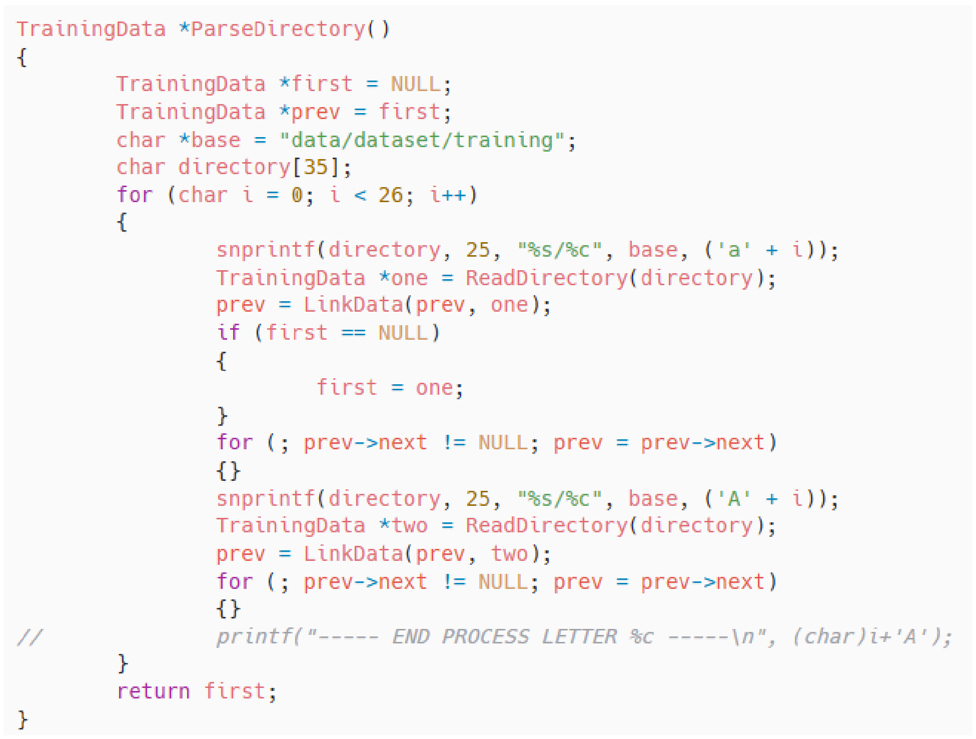
The `ReadDirectory` function parses a directory and create a TrainingData for every .png image it finds. It uses the functions `CreateData` and `LinkData` created in the file `source/neural_network/setup.c`, as well as `SDL_to_list` from `source/neural_network/convertion.c`
You can see below the part of the function where it does just that.



```c
char path[len + p_len + 10];
snprintf(path, len + p_len + 10,
"%s/%s", directory, entry->d_name);
SDL_Surface *surface = toSDL(path);
if (surface != NULL)
{
        TrainingData *data = CreateData();
        char letter = entry->d_name[0];
        if (letter >= 'a' && letter <= 'z')
                letter -= 32;
        data->expected = letter;
        SDL_to_list(surface, nInputs, &(data->inputs));
        cur = LinkData(cur, data);
        if (first == NULL)
        {
                first = data;
        }
        cur = cur->next;
}
```

Figure 22: ReadDirectory code

The `ParseDirectory` function parses all the directories stored in a base directory, and calls `ReadDirectory` for each of them.
You can check the process of the function below.



```c
TrainingData *ParseDirectory()
{
        TrainingData *first = NULL;
        TrainingData *prev = first;
        char *base = "data/dataset/training";
        char directory[35];
        for (char i = 0; i < 26; i++)
        {
                snprintf(directory, 25, "%s/%c", base, ('a' + i));
                TrainingData *one = ReadDirectory(directory);
                prev = LinkData(prev, one);
                if (first == NULL)
                {
                        first = one;
                }
                for (; prev->next != NULL; prev = prev->next)
                {}
                snprintf(directory, 25, "%s/%c", base, ('A' + i));
                TrainingData *two = ReadDirectory(directory);
                prev = LinkData(prev, two);
                for (; prev->next != NULL; prev = prev->next)
                {}
//              printf("----- END PROCESS LETTER %c -----\n", (char)i+'A');
        }
        return first;
}
```

Figure 23: ParseDirectory code

6.2.4 Saving the network - CSV

We worked on how to save the network quickly after implementing the first ligns of code. It would be useless to train a network if we could not find a way to save it for later use ! We spent a long time researching ways to do that before we thought of saving the weigths and biases in csv files. The idea was to save the weigths and biases of a `Layer` in separate files for each of the layers of the network. To do that, we implemented the following functions in `source/neural_network/csv.c`:

- `void WriteCsvWeight(const char *filename, Layer l);`
- `void WriteCsvBiases(const char *filename, Layer l);`
- `void ReadCsvWeigths(const char *filename, Layer *layer, int max_weights);`
- `void ReadCsvBiases(const char *filename, Layer *layer, int max_neurons);`

7 Word search solver algorithm (Lise)

We built the solver by working through each stage methodically.

7.1 The solver function

To find a word in a grid, we implemented a function taking a matrix, its dimensions, along with the word to find. Its prototype is the following:

- `int Solver(int row, int col, char **mat, char word[]);`

Before parsing the grid, we first check whether the word can be in the grid in terms of size. For this, we calculate the length of said word, and check if it is less or equal to the maximum between the number of rows and that of columns. If not, then the word cannot possibly be inside the grid, and thus there is no need to parse it.

If there is no problem there, then we can go on to the rest of the function. The idea is to parse the grid and to find a match with the first letter of the word. Then, we call another function that will search in all the possible directions for the word, starting from the point we found a match at. This boolean function has the following prototype:

- `int Search(int row, int col, char **mat, char word[], int r, int c, int n, int *er, int *ec);`

To iterate over the different directions, we use the following variables:

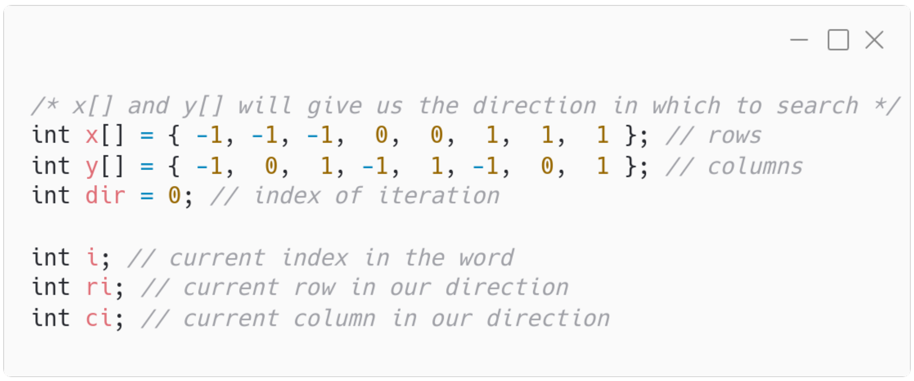

```
/* x[] and y[] will give us the direction in which to search */
int x[] = { -1, -1, -1,  0,  0,  1,  1,  1 }; // rows
int y[] = { -1,  0,  1, -1,  1, -1,  0,  1 }; // columns
int dir = 0; // index of iteration

int i; // current index in the word
int ri; // current row in our direction
int ci; // current column in our direction
```

Figure 24: Solver variables

This allows us to go in one direction depending on the value of dir. If dir is 0, then we will go through the upper left corner, while if dir is 7, we will go through the lower right corner. Then, everytime we iterate, we check whether we have a problem or not. If we are out of bounds, or if we have no match, then we stop and change our direction. If nothing stops us, then we continue searching. If we reach the end of the word, then we stop searching and return True. Else, we continue until we searched every direction. If the word was not found, then we return False.

This was the end of the first version of the solver program. Then, we modified the functions so that it would print the coordinates of the word in the grid if it was found, or "Not Found" if it was not. To do this, we used the variables er and ec to store the row and column of the last letter of the word.

Later, we modified the prototype of `Solver` to be able to recover said coordinates. This was made so that, after we reconstruct the image, we could draw a line from one point to another, by linking those coordinates to those of the bounding boxes.

- `int Solver(int row, int col, char **mat, char *word, int len, int *sr, int *sc, int *er, int *ec);`

Unfortunately, we did not have the time to implement the feature, so the coordinates are currently unused.

7.2 Transforming a file

7.2.1 Into a grid

To use our function `Solver`, we needed to transform a file containing a grid into a matrix, and to find its dimensions. To do this, we implemented the two following functions:

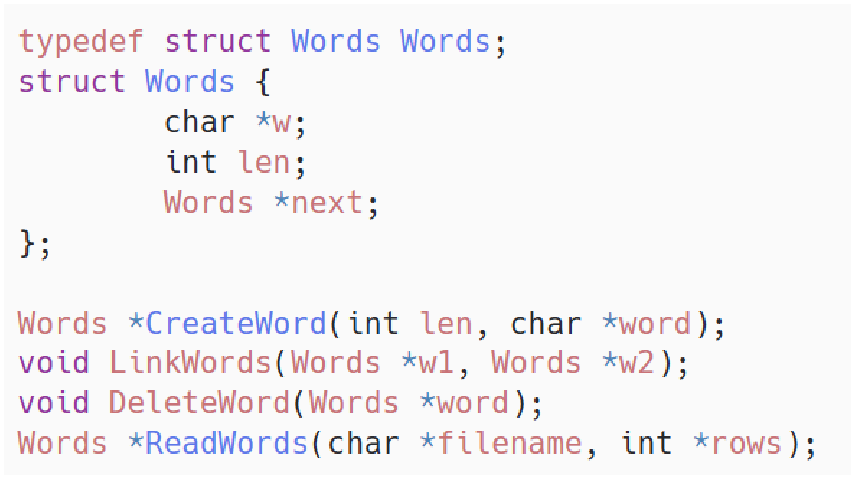
- `char *ReadLine(FILE *file, int *n);`
- `char **ReadFile(char *filename, int *row, int *col);`

The `ReadLine` function takes a file already open and reads a line one character at a time, and it returns an array containing the line read, while storing the length of said line in n.

The `ReadFile` function uses `ReadLine` to read the file opened one line at a time to find and store the number of lines (row) as well as the length of the last line (col). Then, it rewinds the file after creating a matrix. It will parse the file once again by actually storing the lines in the matrix this time, but only if each line has the same length. At the end, `ReadFile` will return the matrix created, or NULL if an error occurred.

7.2.2 Into a list of words

To make the program more efficient, we decided to make it so that the main function would take a file containing the list of words, instead of taking one word at a time. To do this, we created a structure `Words`, which allows us to make a linked list of words. Then, we implemented the function `ReadWords` that is a modified version of the `ReadFile` function. For each line of the file, `ReadWords` will call `CreateWord` and `LinkWords` to create the linked list. Then, the function will return the first element of the list.



```
typedef struct Words Words;
struct Words {
        char *w;
        int len;
        Words *next;
};

Words *CreateWord(int len, char *word);
void LinkWords(Words *w1, Words *w2);
void DeleteWord(Words *word);
Words *ReadWords(char *filename, int *rows);
```

Figure 25: addition to read.h

7.3 Merging the whole

Finally, we implemented a main function that took care of distributing the different elements.

The first version was the following :
After checking the validity of the arguments, we create a grid using the ReadFile function. Then, we parse our word, modifying it in place to only have upper case letters. Then, we call the Solver function with the different parameters we found. And that was it !

Later, we modified the main function so that, instead of taking a grid and a word, it would take a grid and a list of words. For this, we call `ReadWords` and call `Solver` for each `Words` of the linked list returned.

Additionally, as we were not sure of what the final application would be like, we decided to display on the terminal the grid and each word along with the result of its search.

8 User Interface (Fanette)

We anticipated early the needs of having a Graphical User Interface, to test together and assemble our technical tools in the pipeline.

8.1 Setting Up the Development Environment

We chose to use the GTK+ 3.0 and SDL2 libraries together. SDL2, designed specifically for low-level pixel manipulation, was the most suitable tool for preprocessing tasks. For the user interface and rotation functionality, however, GTK+ was selected for its higher-level GUI capabilities, which simplify these aspects. The architecture of GTK+ is illustrated in Figure 26.

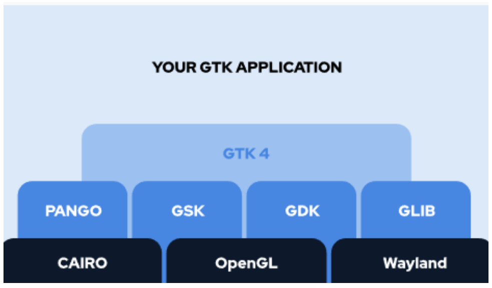


Figure 26: Over time, GTK has evolved to incorporate various libraries, also developed by the GTK team.

Source: `https://www.gtk.org/docs/architecture/`

8.2 Initial interface

The features of the initial interface, presented at the first defense, are described here.

8.2.1 Initialization of the Main Window

The main window is the primary interface of the application.
It is initialized with the GTK application using `gtk_application_new()`, and the main window is created with `gtk_application_window_new()`. We set properties like the window title, position, and size. Ultimately, there is a connection to the “destroy” signal to `gtk_main_quit()` to handle window close events.

8.2.2 A splash screen...

A splash screen is displayed when the application starts. This splash screen is implemented in a separate function, `show_splash_screen()`, which is called during the application’s activation. It was added for fun, inspired by professional software like GIMP.

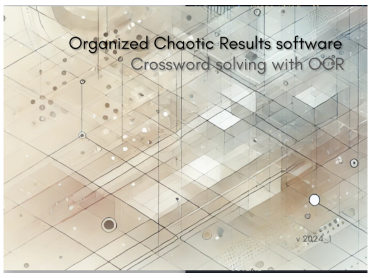

Organized Chaotic Results software
Crossword solving with OCR
v 2024_1

Figure 27: Splash screen v1

8.2.3 Chosen structures

For loading and manipulation, `GtkPixbuf`, `SDLSurface` and `GtkImage` are used together. For convenience, I wrote the following methods to simplify the conversions :

- `GdkPixbuf sdl_surface_to_gdk_pixbuf(SDL_Surface surface);`
- `SDL_Surface gdk_pixbuf_to_sdl_surface(GdkPixbuf pixbuf);`
- `GdkPixbuf image_to_pixbuf(GtkImage image);`

Essentially, the image is first loaded as a pixbuf. When the user clicks on the various image transformation buttons, it is converted to an SDL_Surface, then converted back to a pixbuf for display. Pixbufs are utilized due to their simplicity. However, I am eager to optimize the initial version of this process by researching best practices used in similar complex manipulation contexts.

8.2.4 Layout

The layout of the UI uses the GTK boxes and grid containers. A grid organizes the widgets, e.g. the image and buttons. As well, I added properties to force a specific range of resizing for each widget, padding (specifically for the image widget to allow the changing size of Pixbuf after rotation and alignment (e.g. buttons text).

8.2.5 File manipulation and image display

To load and display images, we utilize the GdkPixbuf library. First, we load an image from a file using the `gdk_pixbuf_new_from_file()` function. This function reads the image file and creates a GdkPixbuf object that represents the image in memory. Next, we resize the image to fit within the desired dimensions. To do this, we calculate the diagonal length of the original image to determine the appropriate size for the resized image, ensuring that the aspect ratio is maintained. We achieve this resizing using a auxiliary function, `resize_pixbuf()`.

After resizing, we add white borders to the image if necessary. This process is handled by the function that calls the previous function, `resize_with_borders()`, which ensures that the image fits perfectly within the specified dimensions. The final image is then created as a GtkImage widget using `gtk_image_new_from_pixbuf()`.

In addition to loading and displaying images, the final step of the pipeline involves saving an image. Currently, this is done using a simple GdkPixbuf save function with a default `.png` extension.

Both the image loading and saving processes utilize callback signals attached to the submenu buttons, which invoke GtkFileChooser.

8.2.6 User interactions

Implementing user interactions in a GTK application involves connecting signals (events) to callback functions that define the behavior of the application when those events occur. This allows us to respond to user actions such as button clicks, text entry, and other interactions. There are not only buttons, but also text boxes (designed for the user to input angle rotation values.)

8.2.7 Conclusion

Note that the function prototypes described were subject to change. We continued working on a better GUI, with a CSS stylesheet among others.

8.3 Second interface

The features for the second interface, designed for the second defense, are described here. The motivation was to rely on the fundamentals of user interface building acquired previously, to propose a more complete set of features for image processing. We removed the splash screen because i3 doesn't support such pop-ups.

8.3.1 The new GUI

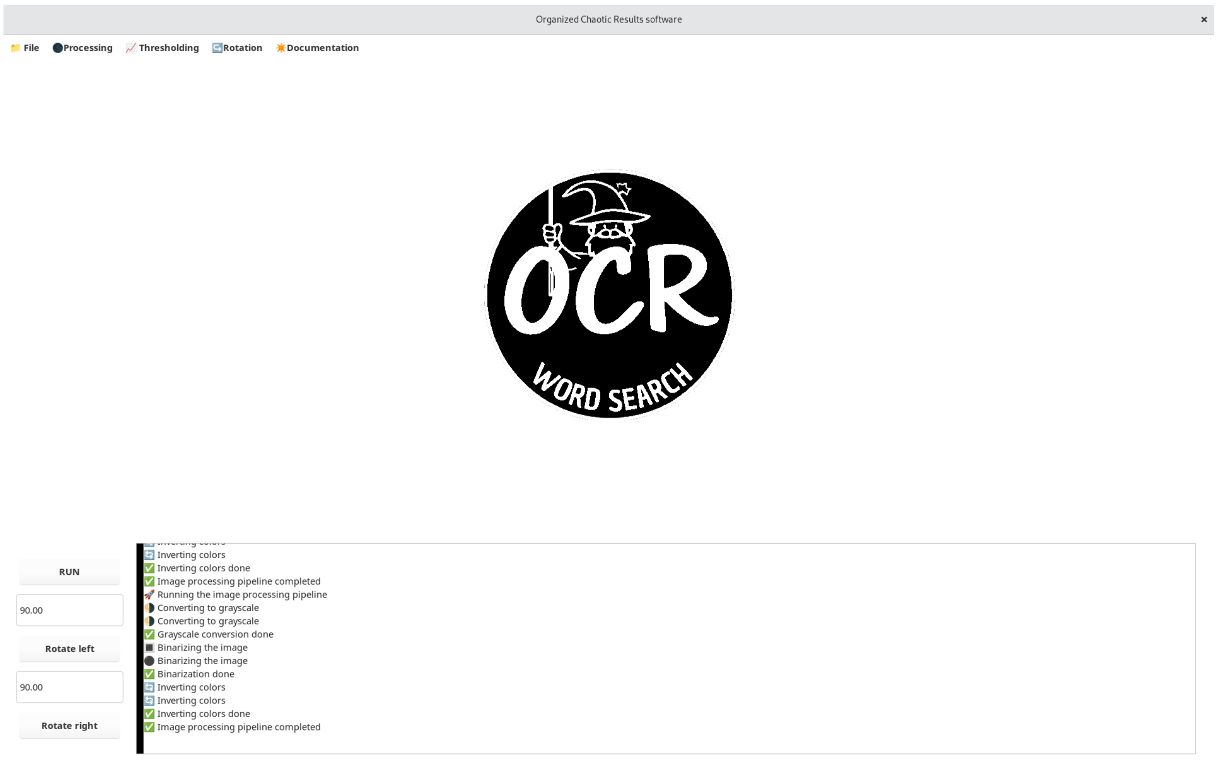
This is the new interface, designed with simplicity and user experience in mind. It offers an intuitive and visually clean layout. It presents all features to correctly process any image with various preprocessing functions. It also features documentation. (With enough time, we could have imagined including the solver inside, and also the detection.)

8.3.2 Components

The interface is held together by an XML designed thanks to Glade and on a minimal CSS.

The menubar

The buttons and *TextEntry* components The RUN button initiates a minimal preprocessing pipeline, which includes: grayscale > binarization > inverting B&W. Rotate Left and Rotate Right buttons allow users to adjust the orientation of the image in custom increments. The default angles are 90° left and right. Additionally, the adjacent input fields enable users to customize rotation angles if needed.



```
🔄 Inverting colors
✅ Inverting colors done
✅ Image processing pipeline completed
🚀 Running the image processing pipeline
🌓 Converting to grayscale
🌓 Converting to grayscale
✅ Grayscale conversion done
⬛ Binarizing the image
⚫ Binarizing the image
✅ Binarization done
🔄 Inverting colors
🔄 Inverting colors
✅ Inverting colors done
✅ Image processing pipeline completed
```

Figure 28: The new interface

The log view The log panel, located at the bottom of the interface, is a feature allowing the user to recall what were the last operations executed. They are extensive and detail each important step of executed functions. The panel provides a comprehensive step-by-step account of the preprocessing stages. Each action (e.g., grayscale conversion, binarization, inversion on the image above) is listed clearly, ensuring transparency in the process.

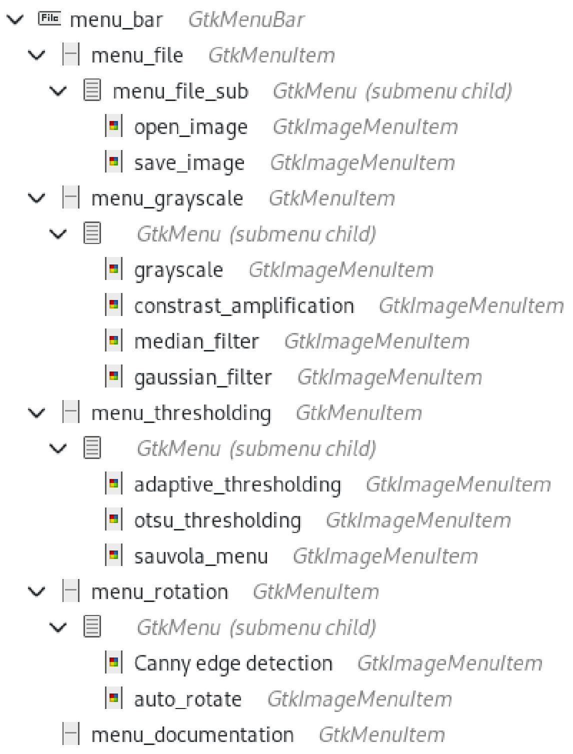
8.3.3 Successful implementations

The GUI design reflects a focus on practicality and user satisfaction. I am particularly proud of how the interface balances functionality and simplicity, ensuring that users of all technical levels can make the most of its features.

8.3.4 Challenges and Implementation Issues

Despite successful planning, the implementation faced frustrating obstacles:

Simple rotation feature removal by accident, becoming a persistent bug Initially, a simple manual rotation feature was integrated into the user interface, providing a fallback for cases where automatic rotation failed. Unfortunately, this feature was inadvertently removed during later updates, leaving the automatic rotation as the only method.



```
menu_bar  GtkMenuBar
  menu_file  GtkMenuItem
    menu_file_sub  GtkMenu (submenu child)
      open_image  GtkImageMenuItem
      save_image  GtkImageMenuItem
  menu_grayscale  GtkMenuItem
    GtkMenu (submenu child)
      grayscale  GtkImageMenuItem
      constrast_amplification  GtkImageMenuItem
      median_filter  GtkImageMenuItem
      gaussian_filter  GtkImageMenuItem
  menu_thresholding  GtkMenuItem
    GtkMenu (submenu child)
      adaptive_thresholding  GtkImageMenuItem
      otsu_thresholding  GtkImageMenuItem
      sauvola_menu  GtkImageMenuItem
  menu_rotation  GtkMenuItem
    GtkMenu (submenu child)
      Canny edge detection  GtkImageMenuItem
      auto_rotate  GtkImageMenuItem
  menu_documentation  GtkMenuItem
```

Figure 29: The menu items

For three consecutive days, I encountered critical bugs in the simple rotation logic. These bugs prevented me from demonstrating the feature during testing. Despite extensive debugging, the exact cause remains unresolved.

Pipeline Integration While each component of the OCR process, including rotation, edge detection, and word identification, works independently, they are not fully integrated into a cohesive pipeline. We didn't plan to integrate them, except as a bonus, so no delay has been taken on those.

9 Conclusion

Lise was supposed to work on linking the different parts, but due to Caroline's departure, she had to take over the critical task of managing the neural network. We decided to focus on the most important aspects of the project, neglecting the smaller but still essential deliverables.

We all dedicated most of our free time to this project, often sacrificing more sleep than was reasonable. Despite the time we spent on each of our tasks to improve their individual quality, we were unable to produce a strong final product. Unfortunately, this also means that even if we had chosen to focus less on quality and more on assembling the different parts, we still likely would not have achieved a viable outcome.

In conclusion, we all loved working on this very challenging project. However, we did not succeed in managing our time and tasks efficiently.