



EPITA

November 4, 2024

OCR Project report n°1

Student team :
Caroline DELIERE
Fanette SAURY
Jans GUILLOPÉ
Lise SUZANNE

Contents

1	Introduction	4
1.1	The project	4
1.2	The team members	4
1.2.1	Fanette SAURY (Project Lead)	4
1.2.2	Caroline DELIERE	4
1.2.3	Jans GUILLOPE	5
1.2.4	Lise SUZANNE	5
2	Implementation strategy	6
2.1	Task repartition	6
2.2	Progress rate	6
2.3	Project Pipeline	7
2.3.1	Required first deliveries	7
3	Image pretreatment (Jans & Fanette)	8
3.1	Grayscale (Jans)	8
3.2	Binarization (Jans)	8
3.3	Inverting pixels (Jans)	9
3.4	Manual rotation (Fanette)	9
3.4.1	Coordinate transformation	9
3.4.2	Bilinear interpolation	10
3.4.3	Source image pixel boundaries	10
4	Detection (Jans)	11
4.1	Character detection	11
4.2	Grid detection	11
4.3	Word list detection	12
4.4	Word detection	12
5	Segmentation (Lise)	13
5.1	Creating a new surface	13
5.2	Converting to PNG	14
6	Neural network (Caroline)	15
6.1	Approach of the neural network	15
6.2	Not Xor proof of concept	15
6.3	Forward propagation	16
6.4	Backward propagation	17
7	Word search solver algorithm (Lise)	18
7.1	The solver function	18
7.2	Transforming a file into a grid	19
7.3	Merging the two	19

8	User Interface (Fanette)	20
8.1	Setting Up the Development Environment	20
8.2	Initialization of the Main Window	20
8.3	A splash screen...	20
8.4	Chosen structures	21
8.5	Layout	21
8.6	File manipulation and image display	21
8.7	User interactions	22
8.8	Conclusion	22
9	Conclusion	23

1 Introduction

1.1 The project

This document presents the achievement rate of the Organized Chaotic Results team for the first 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 pursued 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 is the repartition of the tasks for this first of 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			
GUI		Principal		
Solver				Principal

Table 1: Task division for first 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	25	25
Rotation	50	50
Segmentation	100	100
Detection	75	75
Neural network	30	30
GUI	40	60
Solver	100	100

Table 2: Progress rates for first delivery (in %)

2.3 Project Pipeline

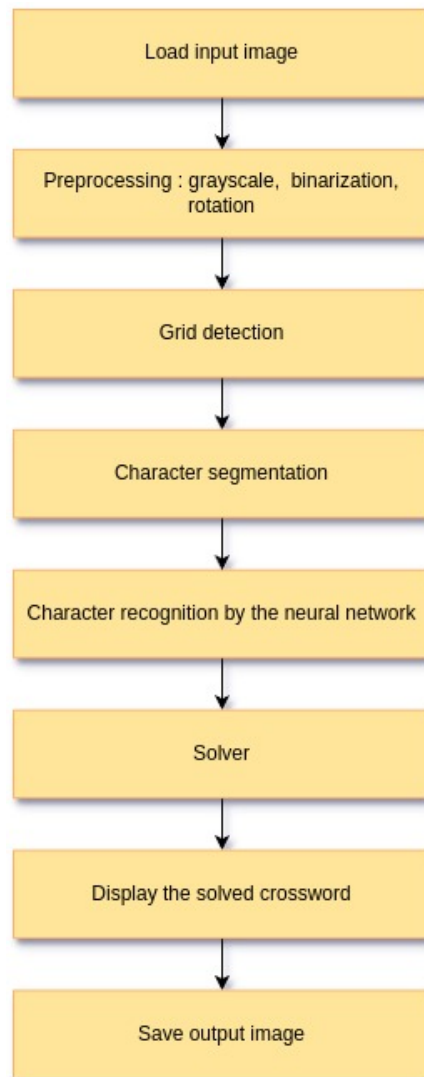


Figure 1: The OCR pipeline

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.
- Crop 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 $A \cdot B + \overline{A} \cdot \overline{B}$.

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)

Grayscaleing 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 is achieved by applying a threshold to distinguish pixels above the threshold (white) from those below it (black). Binarization is used to simplify images, enabling other algorithms to detect and recognize characters, reconstruct grids, and identify words.

In this project, we implemented three binarization algorithms: fixed binarization, Otsu thresholding, and adaptive thresholding. We initially focus on fixed binarization, which uses a constant threshold value of 127 to classify pixels. The other methods will be applied to more complex grids requiring more robust preprocessing.

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.

```

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 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) Binarized

```

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 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) Inverted

Figure 3: Binarized and inverted images

3.4 Manual rotation (Fanette)

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

3.4.1 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.

3.4.2 Bilinear interpolation

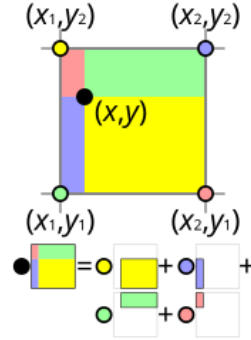


Figure 4: Bilinear transformation. Source: Wikipedia

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).

3.4.3 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.

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 5: 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

5 Segmentation (Lise)

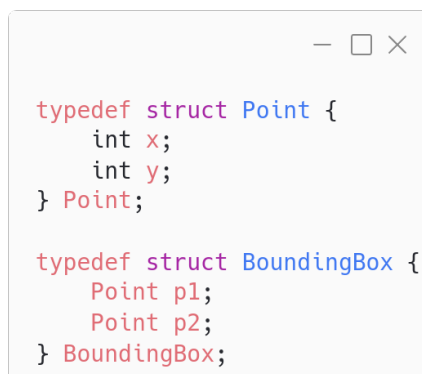
The segmentation heavily relies on the detection part. It is used when an area has been defined after the character detection.

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 6: 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.

```

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 7: 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 a function, which takes a surface, in the `segmentation.c` file.

6 Neural network (Caroline)

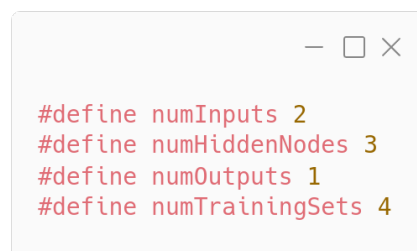
6.1 Approach of the neural network

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.

6.2 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 8: 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 9: 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.

6.3 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.

6.4 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`.

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 10: 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.

7.2 Transforming a file 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.

7.3 Merging the two

Finally, we implemented a main function that took care of distributing the different elements. After checking the validity of the arguments, we create a grid using the `ReadFile` function. Then, we parse our word and modify it in place to have only upper case letters. At the end, we call the `Solver` function with the different parameters we found. And that is it !

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 11.



Figure 11: Over time, GTK has evolved to incorporate various libraries, also developed by the GTK team.

Source: <https://www.gtk.org/docs/architecture/>

8.2 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.

8.3 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.



Figure 12: Splash screen v1

8.4 Chosen structures

For loading and manipulation, `GdkPixbuf`, `SDL_Surface` 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.

8.5 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).

8.6 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`.

8.7 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.)

8.8 Conclusion

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

9 Conclusion

The outcome of the first phase of development is very encouraging. We have been able to manage all the components simultaneously and even anticipate the more graphical tasks for the final delivery. We are proud of the functional software that connects a significant portion of the components developed by the team. The solver is complete. The proof of concept for the neural network is encouraging. The preprocessing requires further investment, but it already allows us to analyse initial grids. The segmentation and detection were completed in a short but controlled timeframe, resulting in a very positive outcome.