Report on Cryptographic Mission Generator

Mathieu Chedas and Benoit Dajoux

May 28, 2024

Contents

Introduction

We would like to thank you for your attention to this report, which describes the work carried out as part of the project you proposed to us. This project, focused on the world of cryptography and education, aimed to model the encryption techniques of Enigma and Sphinx machines, in order to create a cryptographic mission generator based on these two systems. This experience was an enriching opportunity for us to put into practice the skills acquired during our first two years at ISIMA, while discovering new aspects of computer development.

This report is structured in two parts, following the chronological order of our work. The first part is dedicated to the discovery and modeling of the Enigma and Sphinx machines. We will present each machine, as well as the steps of their modeling. We will also detail the process of generating enigmas in LaTeX for these two machines.

The second part deals with the creation of the website. We will describe the different enigma generation pages for Enigma and Sphinx, as well as the general structure of the site. We will discuss the challenges we encountered, the solutions we implemented, and the technological tools we used.

1 Discovery and Modeling of Enigma and Sphinx Machines

In this part, you will find a quick presentation of the two machines, followed by detailed explanations on the functioning of their models, both developed in Python. We chose this language because it is close to Javascript, which we subsequently used for enigma generation on our web page, thus facilitating the conversion of our programs from one language to another.

1.1 Enigma Machine

1.1.1 Machine Presentation

The Enigma machine is a famous electromechanical cipher machine invented by the German engineer Arthur Scherbius in the 1920s. Initially designed for commercial applications, it was quickly adopted by the German armed forces and became a crucial element of Nazi Germany's military communications during World War II. Thanks to its complexity and multiple interchangeable rotors, Enigma could generate billions of encryption combinations, making its messages almost impossible to decipher without the correct key.

However, despite its formidable reputation, the Enigma machine was vulnerable to the relentless efforts of Allied cryptanalysts. The first to achieve this monumental feat was Marian Rejewski, a brilliant Polish mathematician and cryptanalyst. In 1932, Rejewski and his colleagues at the Polish Cipher Bureau managed to reconstruct the internal workings of the machine and create replicas, thus paving the way for the systematic deciphering of German messages.

When the war broke out, the Poles shared their discoveries with the British and the French. At Bletchley Park, the British decryption center, brilliant minds like Alan Turing continued this crucial work. Turing, a mathematical genius, invented an electromechanical machine called the Bombe. This revolutionary invention significantly automated and accelerated the process of deciphering messages encrypted by Enigma.

The success of the Enigma decipherment had major consequences for the outcome of World War II. By allowing the Allies to read enemy secret communications, they could anticipate and thwart German military movements. This technological advance was a decisive factor that contributed significantly to the Allied victory and shortened the duration of the conflict, thus saving countless lives.

Figure 1: Photograph of the Enigma machine

1.1.2 Machine Modeling

Encryption

Enigma encryption takes place in 2 major steps: choosing the orientation of the rotor(s), then encrypting the message.

- The first step consists of choosing an initial orientation for the rotor(s); the rotor(s) are turned, and once this is done, we move on to the second step.
- The second step consists of encrypting the message; we take the first letter of our message and follow the connections, which gives us the "encrypted" letter. We then turn the inner rotor one notch counter-clockwise, then look at the second letter, and so on until the end of our message.

Here is the algorithm used:

Algorithm 1: Enigma Encryption

Data: A message of size n, internal and external connection tables, orientation
Result: The encrypted message of size n, the encryption key

```
1  begin
2      internalConnections=generateInternRandomConnexion()
3      externalConnections=generateExternRandomConnexion()
4      orientation=random(0,26)
5      encryptedMessage=""
6      for i ← 0 to n − 1 do
7          letter = search(internalConnections,externalConnections,message[i],i,orientation)
8          encryptedMessage += letter
9      end
10     return
11     encryptedMessage;
12 end
```

The "search" function returns the "encrypted" letter, in other words, the letter that corresponds in the machine.

Algorithm Explanation:
To understand the algorithm, here is a representation of the machine we created; on the 2-rotor version, 2 red notches would appear.

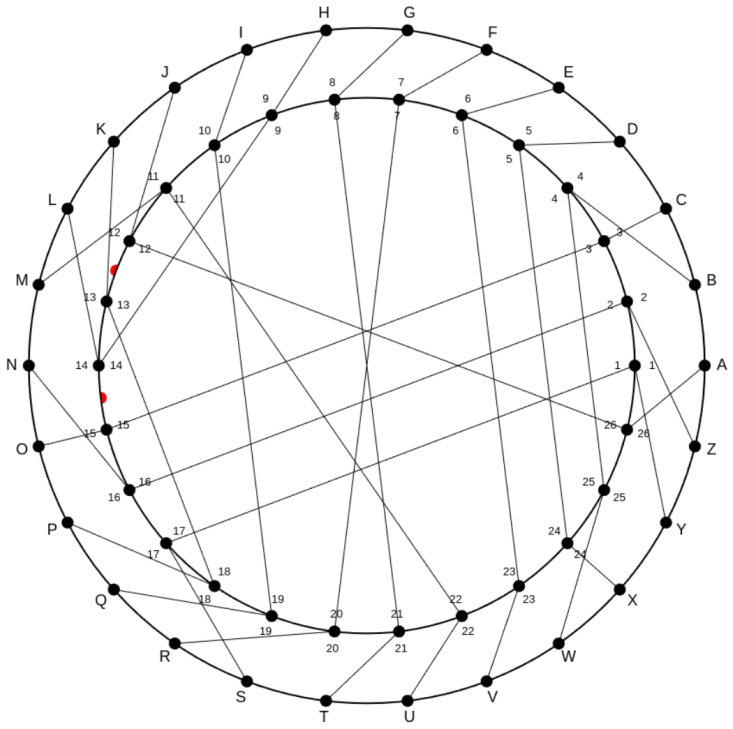


Figure 2: Enigma wheel image

We first implemented two generation functions, which return the internal and external connection tables. These tables follow specific rules.

- External connections: an outer point can only connect to the inner circle point directly opposite it, or to a shift of two positions to its left or right.
- Internal connections: an inner circle point can only connect to another inner circle point that is not in its "neighborhood"; it must go to a point with at least 4 neighbors between itself and the other point.

Once this is done, we iterate through the entire message, and for each letter, we find the corresponding one on the circle, while applying a rotation of i to the inner circle, and we add the letter to the new message. Once the iteration is finished, we return the encrypted message. To avoid "syntax" errors, a function was created to transform the message into "formatted" mode, all in uppercase, without special characters and without accents, for example: "hello world"
$\rightarrow$ "$HELLOWORLD$"

If we are on the 2-rotor version, we look at the 2 available notches on the wheel, and as soon as they are aligned, we turn the outer rotor one notch in the same direction as the inner rotor, then continue the encryption as before. The algorithm remains largely the same, except that the search function takes as arguments the position of the two notches and the initial orientation of the second rotor. To illustrate this encryption technique, here is an example below, with a 2-rotor version wheel, set in the initial position for the message "BONJOUR".

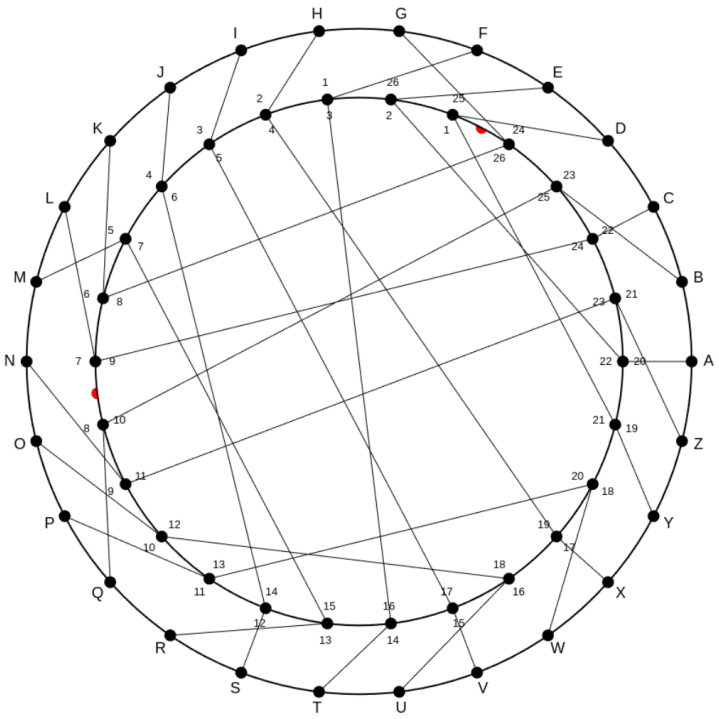


Figure 3: Oriented Enigma wheel image

Here the message "BONJOUR" encrypts to "QCGXHBH"; B is indeed connected to Q, then if we turn the inner circle one notch counter-clockwise, O is indeed connected to C, etc...

Decryption

Decryption, if all information, connections, and orientations are available, is essentially the same as encryption. Indeed, as seen in the example, if B is connected to Q, then Q is connected to B, so we use the same algorithm to decrypt a message.

Calculation of Minimum Required Indices

A function had to be created to provide the minimum required index for there to be only one solution to the enigma. This function works as follows:

- We send the first

  i letters of the plaintext message (starting with 1, then 2,...)

- If for this number of letters there are several possible messages, we send one more letter.

This cycle is repeated until there is only one possible solution.

Auxiliary Functions

```
def generate_intern_random_connexions():
def generate_extern_random_connexions():
```

Functions that generate the connections.

```
def solutions():
```

Function that finds the minimum required index for the solution.

```
def normaliser_chaine():
```

Function that formats the message, removing special characters and converting everything to uppercase.

1.2 Sphinx Machine

1.2.1 Machine Presentation

The Sphinx machine is a pocket encryption machine, developed around 1930 by the Société des Codes Télégraphiques Georges Lugagne in Marseille. It can only be used for letter encryption but is nonetheless interesting. It consists of a metal frame with ten tracks, each containing two movable strips with mixed alphabets. The 20 transposed alphabets are identified by a number, printed in red, and are always used in pairs, the upper alphabet representing the plaintext and the lower alphabet used to represent the ciphertext.

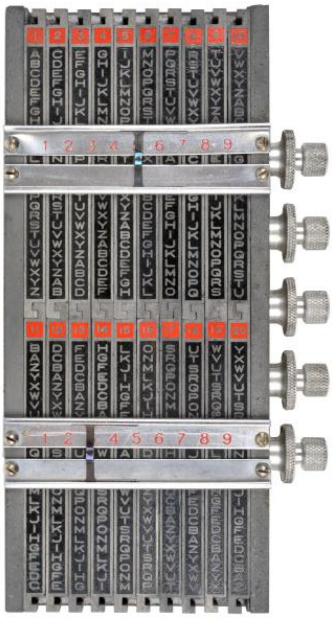


Figure 4: Photograph of the Sphinx machine

1.2.2 Machine Modeling

Encryption

Sphinx encryption consists of 2 major steps: placing the strips and encrypting the message.

The first step is to place the strips arbitrarily or thoughtfully into the machine. The order of placing the strips corresponds to the encryption key. The 20 strips, which are always the same, are visible in appendix.

Once the strips are placed, the desired message must be encrypted. To do this, for each letter of the message, you must place the letter of interest in the designated slot/notch on the upper part of the machine using the top strips, and read the resulting letter in the corresponding lower slot/notch. Below is the Sphinx encryption algorithm we used.

Algorithm 2: Sphinx Encryption

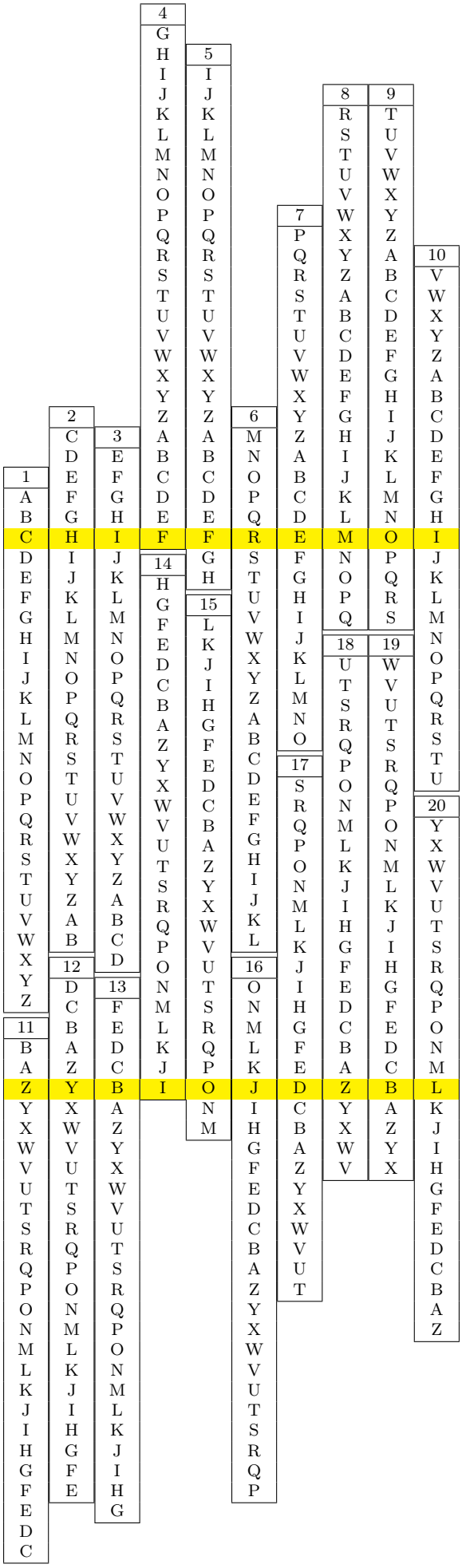
Data: A message of size n
Result: The encrypted message of size n, the encryption key

```
1  begin
2      upperStrips=shuffle(generateUpperStrip)
3      lowerStrips=shuffle(generateLowerStrip)
4      encryptedMessage=""
5      for i ← 0 to n − 1 do
6          letterIndex = search(upperStrips[i])
7          encryptedMessage += lowerStrips[i])
8      end
9      return
10     encryptedMessage,generateKey(upperStrips,lowerStrips);
11 end
```

Algorithm Explanation:

We first implemented two generation functions, allowing us to obtain a list containing all the upper strips and a list containing all the lower strips, with their alphabet, in the form of a list of 27 characters, the 27th character serving to indicate the strip number. Once all the strips are obtained, the encryption function randomly shuffles these two strip lists, the result of the shuffle corresponding to the encryption key. Then, for each character of the word given as an argument, we simply apply the encryption algorithm by looking at the position of the current letter in strip number n

To better illustrate this encryption technique, here is an example of encryption for the message `CHIFFREMOI`.


1
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
11
B A Z Y X W V U T S R Q P O N M L K J I H G F E D C
2
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
12
D C B A Z Y X W V U T S R Q P O N M L K J I H G F E
3
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
13
F E D C B A Z Y X W V U T S R Q P O N M L K J I H G
4
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
14
H G F E D C B A Z Y X W V U T S R Q P O N M L K J I
5
I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
15
L K J I H G F E D C B A Z Y X W V U T S R Q P O N M
6
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
16
O N M L K J I H G F E D C B A Z Y X W V U T S R Q P
7
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
17
S R Q P O N M L K J I H G F E D C B A Z Y X W V U T
8
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
18
U T S R Q P O N M L K J I H G F E D C B A Z Y X W V
9
T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
19
W V U T S R Q P O N M L K J I H G F E D C B A Z Y X
10
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
20
Y X W V U T S R Q P O N M L K J I H G F E D C B A Z

Example :

In the example above, the plaintext `CHIFFREMOI` encrypts to `ZYBIOJDZBL` with this strip configuration. The two yellow lines represent the slots/notches of the Sphinx machine. This strip configuration can be represented by the key (1,11) (2,12) (3,13) (4,14) (5,15) (6,16) (7,17) (8,18) (9,19) (10,20).
The key is constructed as follows: we will represent each pair of strips by their number, and we will write each pair of strips in the order in which they are positioned in the machine (here the pair (3, 13) is the third pair of strips from the left).

Decryption

The decryption function uses the same principle: we organize the strip tables according to the provided key, then we repeat the process described previously. The only difference lies in the inversion of the roles of the upper and lower strips. We will search for the current letter of the encrypted message in the list of lower strips, and we will look for its correspondence in the list of upper strips.

Calculation of Indices to Provide

In order to generate enigmas, it was necessary to implement a function allowing the calculation of the best indices to give to someone so that they could deduce a unique key for message decryption. The principle allowing us to calculate the best solution will be as follows. We will have 2 different calculation models that we will call with different parameters, and each will return a list of indices to give to the user, sufficient to deduce a unique key. We will then compare each of these lists of indices to choose the best one, which will be the one with the lowest score. This score can be calculated with the formula:

$$\text{Score} = L \times 1 + C \times 1.75$$

where:

L] is the number of letters in the index list. C] is the number of strip pairs.

Now that we know how to choose the best list of indices, let's look in more detail at these two calculation methods to understand how these lists are created.

Calculation Method N°1

For this method, based on a certain number of given letters (corresponding to the first letters of the plaintext message), we will create a list for each pair `(plaintext message letter, ciphertext message letter)` of the strip pairs that could allow this letter pair to appear on the machine. To do this, we will use the method that consists of looking at the table of 20 aligned strips, and if we find the two letters of the letter pair at the same position in two different strips, then these two strips will form a potential pair, and will therefore be added to the list. These lists can be quite long (see example), so it can quickly become difficult for a user to find the solution to the enigma using this principle. This is why when generating the enigma, we will ensure, while indicating it to the user, that the numbers of the strips placed on the top of the machine are necessarily between 1 and 10, and that the numbers of the strips placed at the bottom are between 11 and 20.

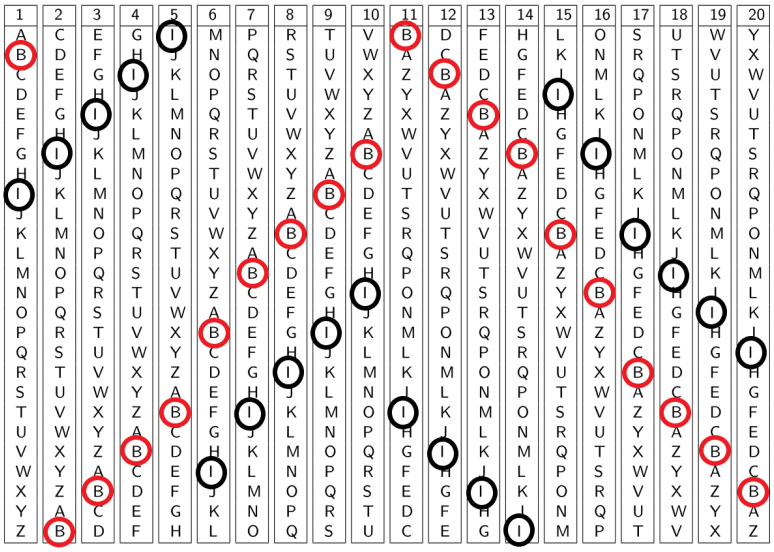


Figure 5: Creation of couple lists

Example :

Based on the figure above, let's find the list of possible strip pairs for the letter pair `(B, I)`. We will identify all present letters B and I. By looking at their alignment, they will allow us to deduce the following list of pairs: (5,11) (4,12) (3,13) (2,10) (2,14) (16,10) (16,14) (1,9) (17,15) (17,8) (18,7) (10,16) (9,6) (8,17) (7,5) (11,5) (7,18) (11,18) (12,4) (12,19) (13,3) (13,20) (14,3). Adding our condition on the upper and lower strips, we can eliminate all pairs whose first digit is greater than 10 (the upper strips, represented by the first digit of the pairs, are between 1 and 10) and all pairs whose second digit is less than 11 (the lower strips, represented by the second digit of the pairs, are between 11 and 20). We can also eliminate all pairs having two strips between 1 and 10 or two strips between 11 and 20.
We will finally obtain the following list: (5,11) (4,12) (3,13) (2,14) (10,16) (8,17) (7,18).

After having drawn up the list of possible strip pairs for each plaintext letter provided, we will try to deduce valid strip pairs. To do this, we will first check if a list of pairs contains only one element. We then distinguish two cases.

Case N°1:

One of the lists of pairs contains only one element, then this strip pair is necessarily valid. We will then add it to the list of discovered strip pairs. Each time we find a strip pair, we will go through the lists of possible strip pairs and delete the pairs containing a number already part of the final key found.

Example:

In this example, we have 3 letter pairs, so we are in the case where the algorithm is called with 3 letters from the plaintext message. We then notice that the pair `(E,U)` has only one element in its list of strip pairs (impossible in the first step, we assume there have been intermediate treatments). Thus, as this pair is the only one in the list, it necessarily corresponds to an element of the key. We will therefore be able to eliminate the other strip pairs containing either an 8 or a 12, because they cannot exist (each strip is present exactly once). Thus, we

can observe on the right, the lists of possible pairs after deduction.

A ⟶ D	S ⟶ B	E ⟶ U	A ⟶ D	S ⟶ B	E ⟶ U
6 ⟶ 12	5 ⟶ 18	8 ⟶ 12	2 ⟶ 19	5 ⟶ 18	8 ⟶ 12
4 ⟶ 14	7 ⟶ 12		4 ⟶ 14		
8 ⟶ 11	8 ⟶ 13				
2 ⟶ 19					

Example of valid strip pair deduction

Case N°2:

All lists of pairs contain at least two elements. We will then add to the list of discovered strip pairs, a strip pair by reading it directly from the key. We will also add this strip pair to the list of given indices. To choose the strip pair to give, we iterate through all the previously compiled lists of possible pairs, and we count the strip numbers that appear most often. The pair whose numbers appear most often will then be selected, as it will allow the elimination of the most possibilities.

Example:

In this example, we have 3 letter pairs, as before. We notice that all pairs have at least two elements in their lists. We will therefore have to count how many times each number appears.

Suppose the key is as follows: (4,14) (8,12) (5,18) ...
We then have the following results: (4,14) = 2, (8,12) = 6, (5,18) = 2
The key pair with the highest score is (8,12), so we will add it to the list of indices, and consider it found by adding it to the initially empty list of the deduced final key. We can then make the same deductions as in the previous example.

A ⟶ D	S ⟶ B	E ⟶ U	A ⟶ D	S ⟶ B	E ⟶ U
6 ⟶ 12	5 ⟶ 18	8 ⟶ 12	2 ⟶ 19	5 ⟶ 18	8 ⟶ 12
4 ⟶ 14	7 ⟶ 12	1⟶ 17	4 ⟶ 14		
8 ⟶ 11	8 ⟶ 13	3 ⟶ 16			
2 ⟶ 19					

Example of valid strip pair deduction

We will thus repeat these 2 operations until all the strip pairs of the key have been deduced, with the indices stored in the list of indices.
This first calculation method will be executed 10 times, by giving one to 10 letters as a parameter at the beginning.

Calculation Method N°2

The second calculation method will use the same reasoning as the first, except that instead of directly providing all letters, then the number of strip pairs necessary for deduction, we will provide

n letters then m strip pairs, where n and m are values passed as parameters. Thus, we will call this function with different values of

n and m to calculate different lists of possible indices. This method can be described by the following algorithm:

Algorithm 3: Index List Calculation Method 2

Data: Two integers n and m, the encryption key, the two strip lists, the plaintext and ciphertext messages
Result: indexList, deducedKey

```
1  begin
2      indexList = []
3      deducedKey = []
4      ListOfCouples = []
5      currentIndex = 0
6      while length(deducedKey) ≠ length(key) do
7          for i ← 0 to n − 1 do
8              currentIndex = i
9              deduceCoupleList(message[currentIndex])
10         end
11         for i ← 0 to m − 1 do
12             deduceKey(ListOfCouples)
13             giveKey(key, deducedKey)
14         end
15     end
16     return indexList;
17 end
```

Ultimately, these two calculation methods allow us to obtain satisfactory results. The indices provided are often balanced and minimal.

Auxiliary Functions

We found it interesting to implement certain generic functionalities thanks to simple functions, thus allowing the reusability of their code. Here they are briefly presented.

```
def int[][] generate_upper_strip():
def int[][] generate_lower_strip():
```

These two functions allow to generate respectively strips 1 to 10 and strips 11 to 20, which allows manipulating these lists without fearing that side effects could harm the functioning of other functions. If we want to retrieve the original strip lists, we just need to use these two functions.

```
def int [] generate_key(int [][] up, int [][] down):
```

This function will take 2 lists of strips as arguments and will return the key corresponding to these lists. It will return, in the form of a list of tuples, the strip pairs in order of their positioning.

```
def (string, int[]) encrypt_with_key(string word, int [] key):
```

This function will take a word to encrypt and a key as arguments, and will encrypt the word according to the key. It will be useful when implementing the option allowing the user to generate an enigma from a file.

```
def (int [][], int [][]) organize_strip(int [] key, int [][]
up_strip_list, int [][] down_strip_list):
```

Finally, this last utility function will allow us to arrange the strips in a specific order, which can be useful for the `encrypt_with_key` function, but also for the decryption function.

1.3 Generation of Enigmas in Latex

1.3.1 Enigma

An enigma is presented as follows. It contains the wheel on the first page, then on the back a descriptive text, which gives the encrypted message, the beginning of the solution, as well as the instructions to follow to successfully decrypt it. This enigma will be generated by manipulating character strings; a part will be "hardcoded" as it will constantly be the same, and data such as connections, orientation, and messages will be added in the middle of these character strings. For the wheel drawing, we used the provided model, adding the necessary modifications to manage the wheel's orientation.

```
def generate_latex():
```

This function will generate the latex file of the enigma with the correct connections and messages.

```
def generate_latex_solution():
```

This function will generate the latex file of the solution with the correct wheel rotation.

1.3.2 Sphinx

A Sphinx enigma is presented as follows. It contains an explanatory text, summarizing the machine's operation, and indicating the message to be decrypted, as well as various clues that can help in solving the enigma. This text is also followed by an example to better illustrate the machine's operation (see appendix).

This enigma will be generated using different functions, each processing a part of the enigma, and all operating on the same principle. They will call the functions described above, and perform string manipulation, each time with a main character string, to which text will be added, which will finally be written to a file. The details of these functions are presented below.

```
def void generate_tex_strip():
```

This function will allow generating the representation of the 20 strips of the Sphinx machine. Each being represented using a LaTeX table.

```
def void generate_tex_example(string message, string
encrypted_message, int[] key):
```

This function will generate the example of the Sphinx machine's operation, which can be found above. It will be called with an encrypted message, a plaintext message, and a key as arguments. These elements are hardcoded in the form of lists, and will depend on the number of strips the user has chosen for generating their enigma (see strip number configuration).

```
def generate_tex_solution(string message, string encrypted_message,
int[] key):
```

Finally, this last function will allow generating the enigma solution file. This will consist of generating a succession of representations similar to that displayed as an example in the enigma file (see above). Each strip will be represented by a LaTeXtable, but the difficulty will lie in managing the offset between each strip. For this, before each table, we will create a certain number of spaces with the LaTeXcommand `\phantomA`. Please note, LaTeXcan have particular behaviors with the management of floating elements when it comes to spacing on the page. The loop that allowed us to manage the spacing of the tables is shown below.

```
1 for i in range(0, len(up_strip_list[elem[0]]) - 1):
2 if up_strip_list[elem[0]][i] == message[count]:
3 indent = i
4 gap = 13 - indent
5 for j in range(26 + gap*2):
6 temp_chain += "\phantom{A} \\ \n"
7 temp_chain += "\hline"
```

Explanations:

We will execute this loop for each pair of strips. First, we will go through the upper strip and look for the position of the letter we are interested in (the one that should be in the Sphinx machine's slot).
From its position, we will determine if it is before or after the middle of the strip, and its distance from the middle of the strip. Then we will add as much space as necessary (`\phantomA` corresponds to a space of one uppercase letter), so that the letter we want to highlight is in the middle of the figure.

1.3.3 Sphinx Enigma Strip Number Configuration

Once the modeling was valid for 10 strips, it became necessary to make the number of strips configurable before moving on to website creation. Indeed, it was easier to modify code in a language we mastered better.
Configuring the strips can be useful for simplifying the enigma, especially for children, as there are fewer possible strip pairs, which makes the key easier to find. Thus, when the number of strips is 10, the Sphinx machine will use the 20 alphabets present in the physical version of the machine, but when this number is different from 10, we have chosen to randomly select

2n alphabets, where n corresponds to the chosen number of strips. The configuration was not difficult to operate, the change essentially consisting of replacing `10` with the variable `NUMBER_OF_STRIP`. The biggest change was the addition of a special case during the creation of the strip lists. It was necessary to add a condition randomly shuffling the strip list, and retrieving the first `NUMBER_OF_STRIP` if the chosen number of strips was different from 10.

2 Website Creation

We will now present the second part of the project, the creation of the cryptographic mission generator website. You will find a general presentation of the Enigma and Sphinx pages, as well as detailed information on the techniques used for their creation.

But before talking about the site, there is a function to take into account because it allows us to solve a problem related to a specificity of javascript:

```
Number.prototype.mod = function(n) {
var m = (( this % n) + n) % n;
return m < 0 ? m + Math.abs(n) : m;
};
```

Indeed, this function is used everywhere in the code because Javascript "allows" negative modulo operations, which would have changed the entire algorithm implemented in Python for the encryption functions.

This function therefore returns a number between 0 and n and is used as follows:

parseInt(value).mod(n). It allowed us to replace the "

2.1 Enigma puzzle generation page

Upon arriving at the Enigma puzzle generation page, there is a button to start the puzzle generation. However, the user can choose to generate a custom puzzle that they have stored in a text file, with a precise syntax:

- 1 line for external connections, 1 for internal connections, 1 for initial orientation, and the personalized clear message for the 1-rotor version
- before the clear message, 1 line must be added for the second rotor's orientation, 1 for the inner notch's location, and 1 line for the outer notch

This file import allows for use in an educational activity, so that all participants have the same puzzle to facilitate correction.

Here is an example of the file syntax:

```
A:25,B:3,C:4,D:6,E:7,F:5,G:9,H:8,I:11,...

1:21,2:19,3:14,4:16,5:17,6:20,7:23,8:0,...

19

JAIMELEBASKET
```

The ":" are used to indicate pairs, one element on the left and the other on the right, and the "," are used to separate the pairs.

When the user clicks the "Start mission" button, the data will be generated in the background using the previously generated Python functions that we were able to translate into Javascript.

The puzzle will then appear with an animated message explaining the mission's principle, the encrypted message, the beginning of the solution, and instructions for decrypting the message.

Figure 6: site message

Then, below the message, there is a reminder of the ciphertext where the letters are aligned with those in the letter input section directly below, to facilitate reading.

Figure 7: site input

Finally, below all of that, there is the wheel with arrows to rotate the rotor(s) and a button that allows displaying the solution to the puzzle if one has not succeeded.

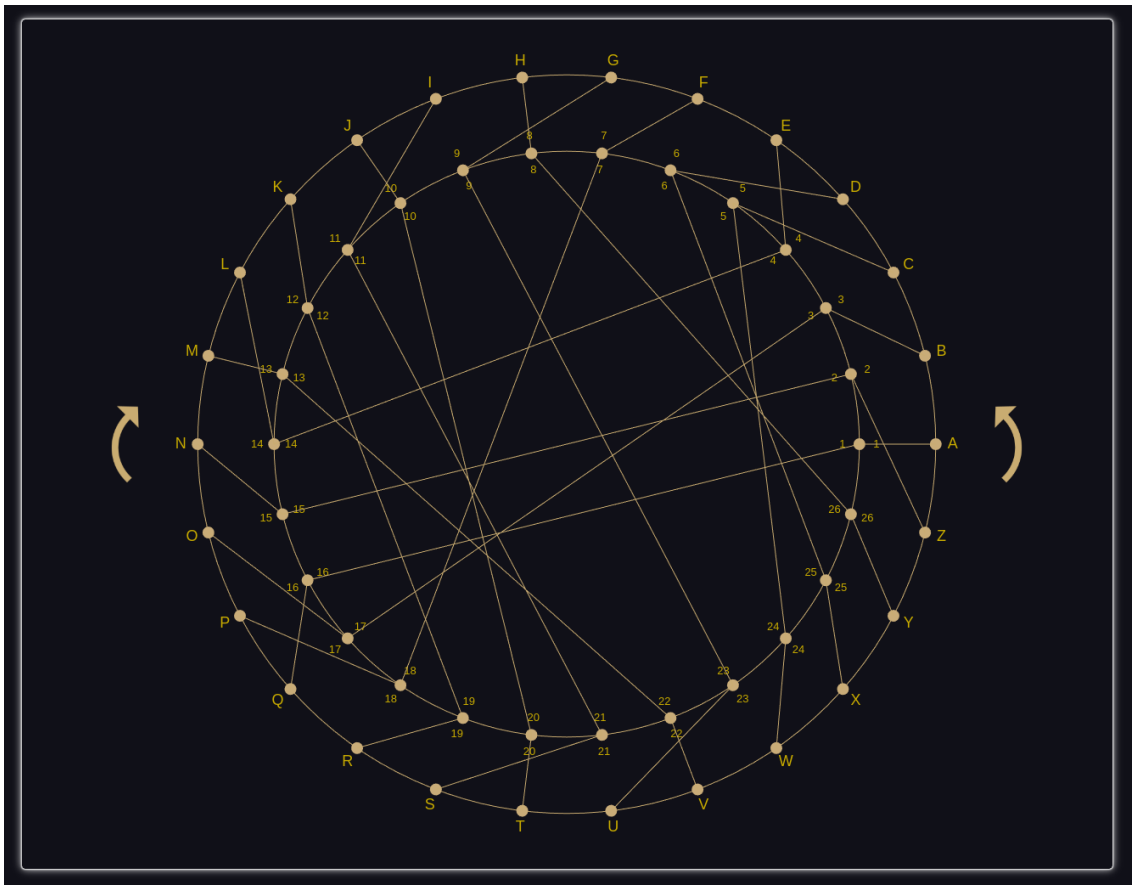


Figure 8: site wheel

Also, in the top right, there is an "options" button that brings up a menu to load the file seen above, but also allows downloading the text file to reload the same puzzle later.

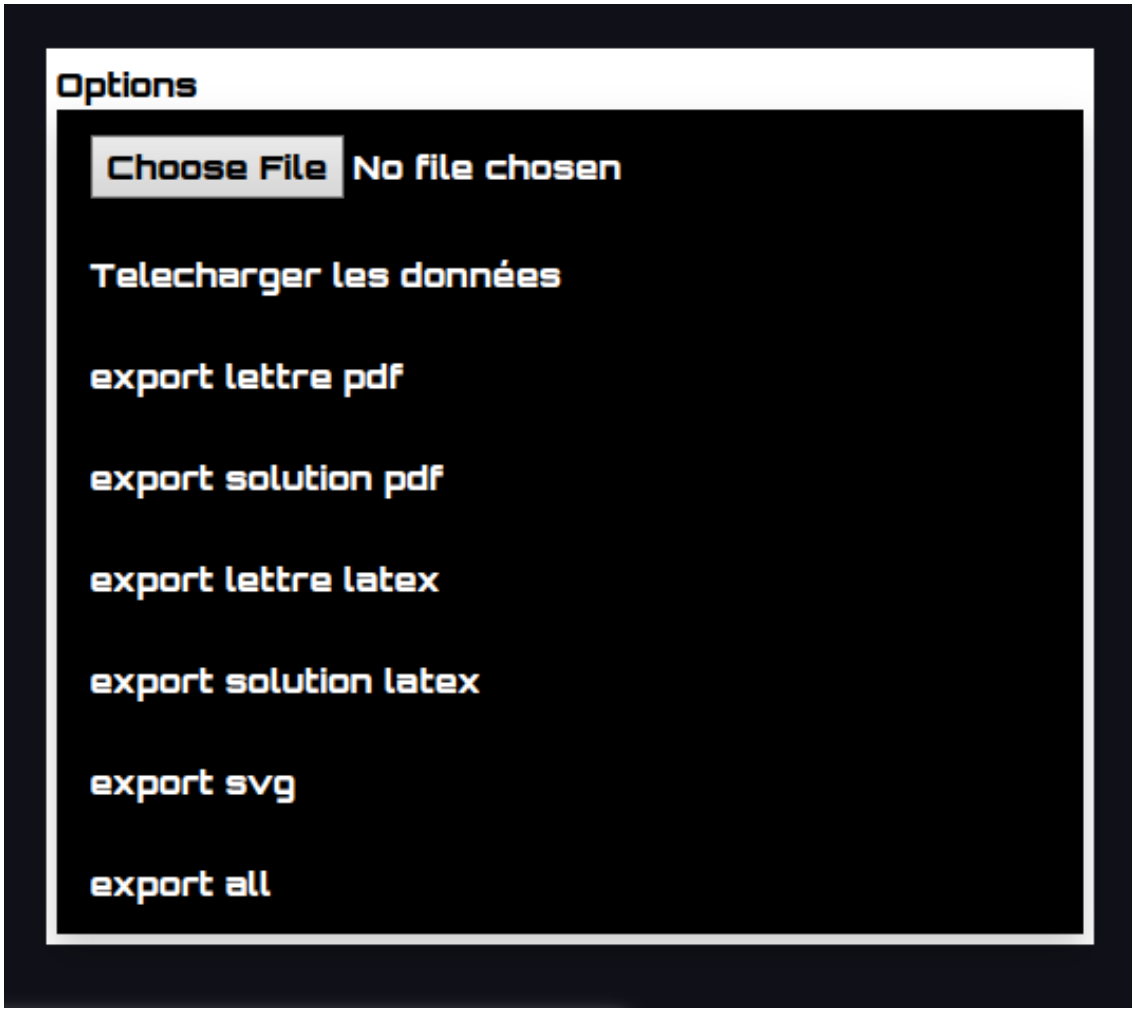


Figure 9: enigma options

There are buttons to download the puzzle in LaTeX format (the same as in Python) but also directly in PDF format thanks to jsPdf, a Javascript module that allows creating PDF files. It is a fairly simple module to use; you can add text, images, and place them wherever you want by specifying the desired coordinates. You can also draw geometric shapes yourself using jsPdf.

Here is an example of creating a document with jsPdf:

```
const doc = new jsPDF();
var today = new Date();
var dateString = "Le " + today.toLocaleDateString("fr-FR") + " ,    Aubi re";
var message1 = "A qui de droit, j'ai con u ma version de la machine Enigma. Je
    vous l'accorde, ce n'est";
var message2 = "pas tr s ressemblant, mais, promis, elle marche presque comme la
    vraie! Pour chiffrer un message,";
var message3 = "il faut tourner la roue interieur d'un cran dans le sens inverse
    des aiguilles d'une montre apr s";
var message4 = "chaque lettre."
var message5 = "Voici mon message:";
var message6 = "La solution:";
var sol = start_letters;
for(let i=0;i

doc.addImage(svgDATA,"SVG",5,40,200,200);
doc.addPage();

doc.setFontSize(12);
doc.text(dateString, 10, 30);
doc.text(message1, 10, 40);
doc.text(message2, 10, 50);
doc.text(message3, 10, 60);
doc.text(message4, 10, 70);
doc.text(message5, 10, 80);
doc.text(complete_encrypted_word, 10, 90);
doc.text(message6, 10, 105);
doc.text(sol, 10, 115);
doc.save("lettre_enigma.pdf");
```

This is the pattern used for the 1-rotor Enigma letter; the "svgDATA" is a screenshot of the wheel in SVG format, which is added to the PDF document.

The options menu also allows downloading the puzzle solution in LaTeX format (still the same as in Python) as well as in PDF format (again with jsPdf).

There is also a button to download the wheel image in SVG format, to recreate the wheel with a wooden board and a laser cutter, which allows solving the puzzle manually and not in front of a computer.

Finally, there is a button to download everything; all files are inserted into a zip file, thanks to the jsZip module. This module allows integrating various files, whether created with jsPdf or simply LaTeX files like those seen above.

Here is an example of jsZip usage:

```
zip = new JSZip();
boolZip=true;
exportLettrePDF();
exportSolutionPDF();
exportLettreLatex();
exportSolutionLatex();
exportSVG();

zip.generateAsync({ type: "blob" }).then(function(content) {
    // Create a link for downloading the zip
    a.href = URL.createObjectURL(content);
    a.download = "enigma.zip"; // Name of the zip file to download
    a.click();
});
boolZip=false;
```

Here, the boolean "boolZip" is set to true, which means that in the export...() function calls, the functions will not launch the download of files one by one; they will add them to the zip, and then the zip will be downloaded.

For the display on the site, we had to add several functions to display the message with the correct messages in the right places, as well as one of the most important functions, which is to draw the wheel. It works with a canvas and is redrawn with each call to the function, with the "rotation" passed as a parameter.

The arrows to the left and right of the wheel decrement or increment the wheel's orientation by 1, relative to its position before clicking the arrow.

The wheel is positioned and its size varies so that the wheel and the message input are visible on the same page without having to scroll up and down each time.

Below the wheel, there is a button that displays the solution with a specific message containing the decrypted message and calls the wheel generation function with the initial wheel rotation to encrypt the first letter.

Here is the function that allowed drawing the wheel:

```
function genererRoue(rotation){
const canvas = document.getElementById("roue");
const contexte = canvas.getContext('2d');

contexte.clearRect(0, 0, canvas.width, canvas.height);

const rayonExterne = canvas.width/2.3;
const rayonInterne = canvas.width/2.9;
contexte.strokeStyle = 'rgb(201, 172, 113)';

contexte.beginPath();
contexte.arc(canvas.width / 2, canvas.height / 2, rayonExterne, 0, Math.PI * 2);
contexte.stroke();
contexte.closePath();

contexte.beginPath();
contexte.arc(canvas.width / 2, canvas.height / 2, rayonInterne, 0, Math.PI * 2);
contexte.stroke();
contexte.closePath();

const nombrePoints = 26;
const angleIncrement = (2 * Math.PI) / nombrePoints;

const pointsRouges = [];
const pointsBleus = [];
const pointsVerts = [];
for (let i = 0; i < nombrePoints; i++) {
const angle1 = -(angleIncrement * i);
const angle2 = -(angleIncrement * i) + (rotation * Math.PI/13);

    const xExterne = canvas.width / 2 + rayonExterne * Math.cos(angle1);
const yExterne = canvas.height / 2 + rayonExterne * Math.sin(angle1);

    const rayon = canvas.width/60 < 7 ? canvas.width/60 : 7;
contexte.beginPath();
contexte.arc(xExterne, yExterne, rayon, 0, Math.PI * 2);
contexte.fill();
contexte.closePath();

    pointsRouges.push({ x: xExterne, y: yExterne });
const xInterne = canvas.width / 2 + rayonInterne * Math.cos(angle1);
const yInterne = canvas.height / 2 + rayonInterne * Math.sin(angle1);
const xInterne2 = canvas.width / 2 + rayonInterne * Math.cos(angle2);
const yInterne2 = canvas.height / 2 + rayonInterne * Math.sin(angle2);
contexte.beginPath();
contexte.arc(xInterne, yInterne, rayon, 0, Math.PI * 2);
contexte.fill();
contexte.closePath();

    pointsBleus.push({ x: xInterne2, y: yInterne2 });
pointsVerts.push({ x: xInterne, y: yInterne });
```

```
52
53      contexte.font = '13px Arial';
54      contexte.fillStyle = 'rgb(201, 172, 0)';
55      contexte.textAlign = 'center';
56  contexte.textBaseline = 'middle';
57
58      const offsetX = Math.cos(angle1) * 20;
59      const offsetY = Math.sin(angle1) * 20;
60  contexte.fillText((i + 1).toString(), xInterne + offsetX, yInterne + offsetY);
61
62      const offsetX2 = Math.cos(angle2) * 20;
63  const offsetY2 = Math.sin(angle2) * 20;
64
65      contexte.fillText((i + 1).toString(), xInterne2 - offsetX2, yInterne2 -
            offsetY2);
66  const labelOffsetX = Math.cos(angle1) * 20;
67  const labelOffsetY = Math.sin(angle1) * 20;
68
69      const lettre = String.fromCharCode(65 + i);
70  contexte.font = '18px Arial';
71  contexte.fillText(lettre, xExterne + labelOffsetX, yExterne + labelOffsetY);
72
73      contexte.fillStyle = 'rgb(201, 172, 120)';
74  }
75
76  contexte.lineWidth = 1;
77  contexte.beginPath();
78
79  for (let i = 0; i < internRandomConnexion.length; i++) {
80      const nextIndex = parseInt(internRandomConnexion[i][1]-1).mod(26);
81  contexte.moveTo(pointsBleus[i].x, pointsBleus[i].y);
82  contexte.lineTo(pointsBleus[nextIndex].x, pointsBleus[nextIndex].y);
83  }
84
85  for (let i = 0; i < externRandomConnexion.length; i++) {
86      const nexIndex = parseInt(externRandomConnexion[i][1]-1).mod(26)
87      contexte.moveTo(pointsRouges[i].x, pointsRouges[i].y);
88  contexte.lineTo(pointsVerts[nexIndex].x, pointsVerts[nexIndex].y);
89  }
90
91  contexte.stroke();
92  contexte.closePath();
93  }
```

2.2 Sphinx puzzle generation page

2.2.1 Home

Upon arriving at the Sphinx puzzle generation page, there is an interface to launch the puzzle generation, as well as an interface where the user can adjust its parameters.

They can choose the number of strips they want to use (10 by default), and can also choose to generate a 100

This option can be particularly useful in an educational setting, ensuring that all students have the same puzzle, allowing for common correction and progress.

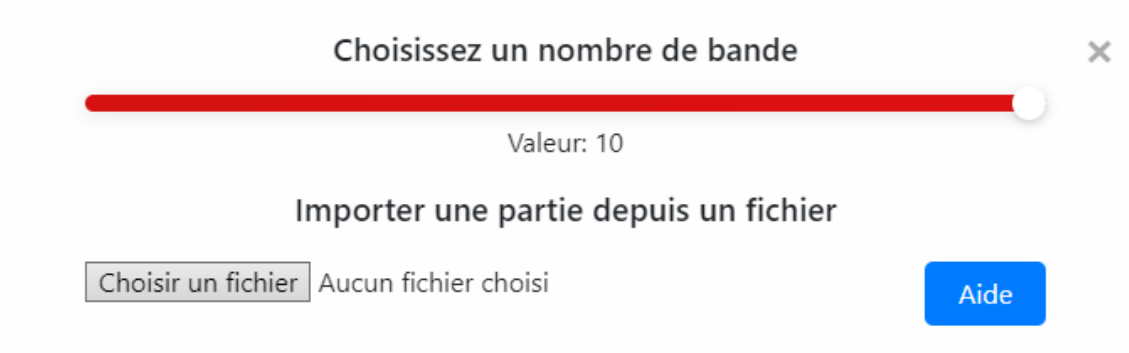


Figure 10: Sphinx Generation Options

For file import, we used the following functionalities:

```
var fileInput = document.getElementById('fileInput');
var file = fileInput.files[0];

if (!file) {
alert('Veuillez selectionner un fichier.');
return;
}

if (file.type !== 'text/plain') {
alert('Veuillez selectionner un fichier au bon format (.txt).');
return;
}

var reader = new FileReader();
reader.onload = function (event) {traitement}
```

This code allows retrieving a file that the user has uploaded using the HTML tag `<input type="file" id="fileInput" accept=".txt" autocomplete="off">`. Once this file is retrieved, its format and validity are checked before processing it to extract information.

When the user clicks the `Start Mission` button, the puzzle and its parameters will be generated in the background, using the Python functions presented previously, which we were able to translate into Javascript automatically and quite easily (using ChatGPT, which saved a lot of time).

In addition to generating the puzzle parameters, a large number of functions calculating the display dimensions based on the screen size and the puzzle are executed at the same time.

This ensures that when the user interacts with the site, everything will be fluid and already generated.

While the machine finishes the various calculations, and to add a more interactive side, an animated message is displayed, communicating to the user the various clues they will need to solve the puzzle, as well as the message to decrypt.

This animated message is in the same form as the puzzle message generated in LaTeX(see appendix).

With this information, the user can then solve the puzzle, using the graphical interface provided for this purpose.

It is composed of several parts, whose operation is described below.

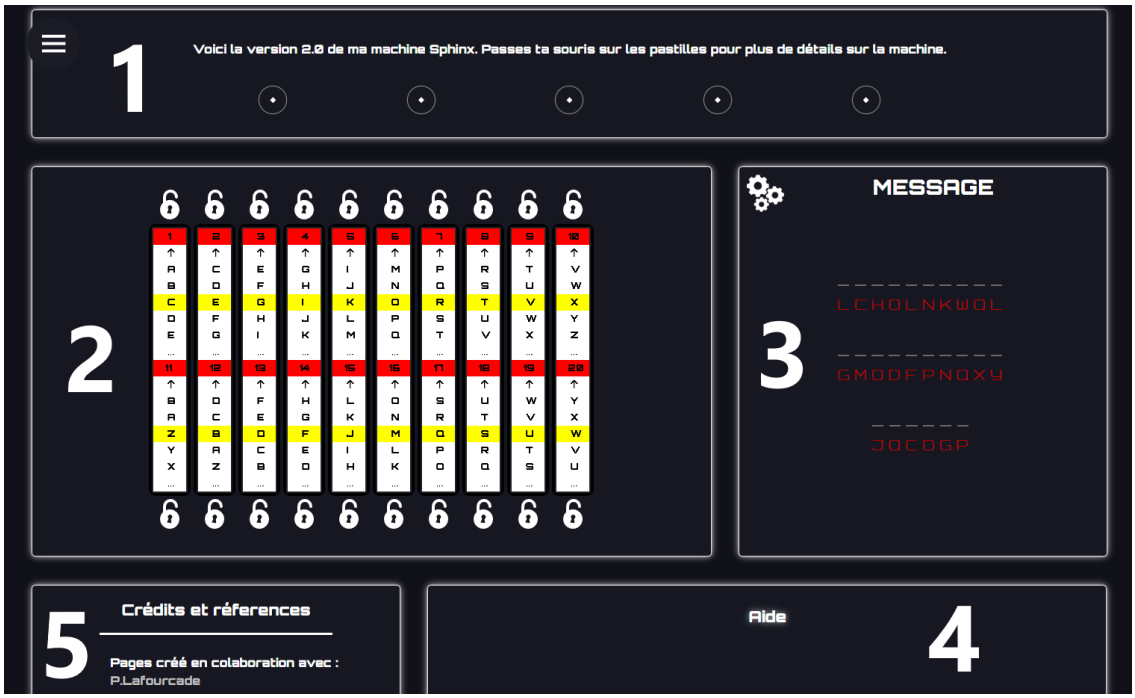


Figure 11: Overview of the Sphinx graphical interface

2.2.2 Part 1: Guide

The first part will serve as a guide, providing the user with different information on how the modeled Sphinx machine works, explaining in particular how to use this machine.

The presentation in bullet points allows for a more compact design, but also a more dynamic experience for the user.

This part is essentially built with HTML and CSS but will also use 2 JavaScript display functions: one to animate in flashing red the bubbles that the user has not yet read, and one to correctly resize the two bubbles closest to the right edge of the screen, so that the text appears correctly on the page.

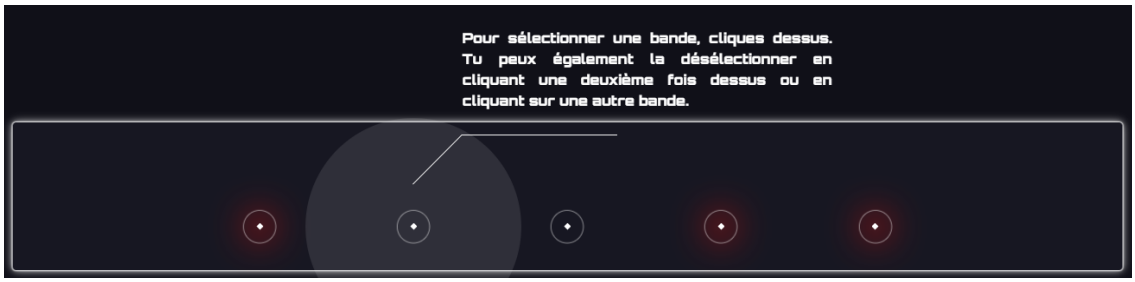


Figure 12: Part 1 of the Sphinx graphical interface

2.2.3 Part 2: Sphinx Machine

Part number 2 is the most important; it is the heart of the machine and the puzzle. It is a model of the Sphinx machine, which naturally adapts to the number of strips entered by the user. It is represented in this way, with partially visible strips, because we had difficulties making the interface compact. In an older version, the strips moved vertically physically on the page, but this made the interface much less pleasant to use. We therefore had to simulate vertical movement by modifying the content of a strip based on pressing the up and down arrow keys on the keyboard. In this model, each strip is composed of 2 independent fragments, and all fragments are interchangeable via drag and drop. Each fragment is an HTML element of type `canvas` and has an identifier with its strip number, to make processing and manipulations simpler.

To implement the drag and drop system, we used the following bootstrap: `https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js`. It implements attributes and states specific to drag and drop for many types of HTML elements. The core of drag and drop management is handled by the following functions, of which here is an overview.

```
/** @type {HTMLAnchorElement[]} */
const listItems = document.querySelectorAll('.list-group-item');
let draggedItem;
var curr_target;

/**
 * @param {HTMLElement} target
 */
const addBorder = (target) => {
    Traitement
}

/**
 * @param {HTMLElement} target
 */
const removeBorder = (target) => {
    Traitement
}

/**
 * @param {DragEvent} e
 */
const handleDragStart = (e) => {
Traitement
}

...

listItems.forEach(listItem => {
    listItem.addEventListener('dragstart', handleDragStart);
    listItem.addEventListener('dragenter', handleDragEnter);
    listItem.addEventListener('dragleave', handleDragLeave);
    listItem.addEventListener('dragover', handleDragOver);
    listItem.addEventListener('dragend', handleDragEnd);
    listItem.addEventListener('drop', handleDrop);
});
```

Explanation :

The skeleton of these functions was provided in an explanation video linked to the bootstrap, and we had to redefine and adapt them to our case.

Generally in this code, we will select all our strips and add an event listener to them for different stages of the drag and drop process, such as the beginning of a strip's capture, the

release of a strip, or the fact that the strip is being manipulated.

Depending on these different events, we will call the functions whose skeleton is defined above.

Furthermore, a strip can be selected by clicking on it, which allows it to be moved vertically using the up and down arrow keys.

The selection will be handled by the following JavaScript function.

```
all_strips.forEach(element => {
element.addEventListener('click', function (event) {
event.preventDefault();
var focused_strip = event.target;
var is_up;
if (focused_strip.classList.contains("category1")) {
is_up = true;

       } else {


            is_up = false;
        }

        var second_focused_strip;
        if (is_up) {
            second_focused_strip =
down_strip_list_drag[all_up_strip.indexOf(focused_strip)]; // Retrieves the bottom
    strip aligned with the selected strip
} else {
second_focused_strip = all_up_strip[down_strip_list_drag.indexOf(focused_strip)]; /
    / Retrieves the top strip aligned with the selected strip
}

        if (old_focused_strip === false) { // If no element is selected

focused_strip.classList.add("fancy");
            second_focused_strip.classList.add("fancy");
            old_focused_strip = focused_strip;
            selected_strip = focused_strip;
} else if ((focused_strip.id == old_focused_strip.id)) { // If we want to deselect
    the element by clicking on the same one
focused_strip.classList.remove("fancy");
second_focused_strip.classList.remove("fancy");
old_focused_strip = false;
selected_strip = -1;
} else if ((focused_strip.id.substring(9) == old_focused_strip.id.substring(9)) ||
    (second_focused_strip.id.substring(9) == old_focused_strip.id.substring(9))) {
     // If we want to deselect the element by clicking on the second strip
second_focused_strip.classList.remove("fancy");
old_focused_strip = false;
focused_strip.classList.remove("fancy");
selected_strip = -1;
} else { // If we want to change the selection
selected_strip = focused_strip;
if (old_focused_strip.classList.contains("category1") == is_up) { // If we click on
     the same strip category
if (is_up) { // If these are top strips
down_strip_list_drag[all_up_strip.indexOf(old_focused_strip)].classList.remove("
    fancy");
} else { // If these are bottom strips
all_up_strip[down_strip_list_drag.indexOf(old_focused_strip)].classList.remove("
    fancy");
}
} else { // If the last 2 clicked are not of the same category
if (is_up) { // If the clicked one is a top strip
```

```
49 all_up_strip[down_strip_list_drag.indexOf(old_focused_strip)].classList.remove("
       fancy");
50 } else { // If the clicked one is a bottom strip
51 down_strip_list_drag[all_up_strip.indexOf(old_focused_strip)].classList.remove("
       fancy");
52 }
53 }
54 old_focused_strip.classList.remove("fancy");
55 focused_strip.classList.add("fancy");
56 second_focused_strip.classList.add("fancy");
57 old_focused_strip = focused_strip;
58 }
59 });
60 });
```

Explanation :

In this code, we will first add an event listener to each strip, then we will execute the code starting from **line 3** if one of the strips is clicked.

First, we will check if the strip is on the upper part of the machine or not; this will allow us to link the clicked strip to its pair without causing an error.

Indeed, since we will need to access its pair, it is important to know which array to search in.

We will then distinguish 4 cases starting from **line 19**:

1. No strip is currently selected
2. The strip just clicked was already selected
3. The strip just clicked corresponds to the pair of the selected strip (the strip aligned with the selected strip)
4. The strip just clicked is different from the currently selected strip

For each case, we will perform a certain number of simple operations, assignments, and class modifications.

The `fancy` class will allow us to manage the highlighting of the selected strip pair.

We will then add it to the two aligned strips that the user has selected, and remove it from all other strips.

Each strip fragment also has a yellow area, corresponding to the box allowing us to encrypt and decrypt a message.

Finally, the padlocks allow the user to lock a strip, meaning to block the drag and drop functionality for that strip, and to color the number green. This allows the user to mark the strip and orient themselves if they believe it is in the correct position. To lock a strip and prevent drag and drop from working on it, we will simply add the HTML attribute `dragdrop=false`, directly provided by the bootstrap. Furthermore, to prevent it from being swapped with a strip that might have been dropped on it, we will check during the `drop` phase if both strips that we want to swap have the HTML attribute `dragdrop=true`.

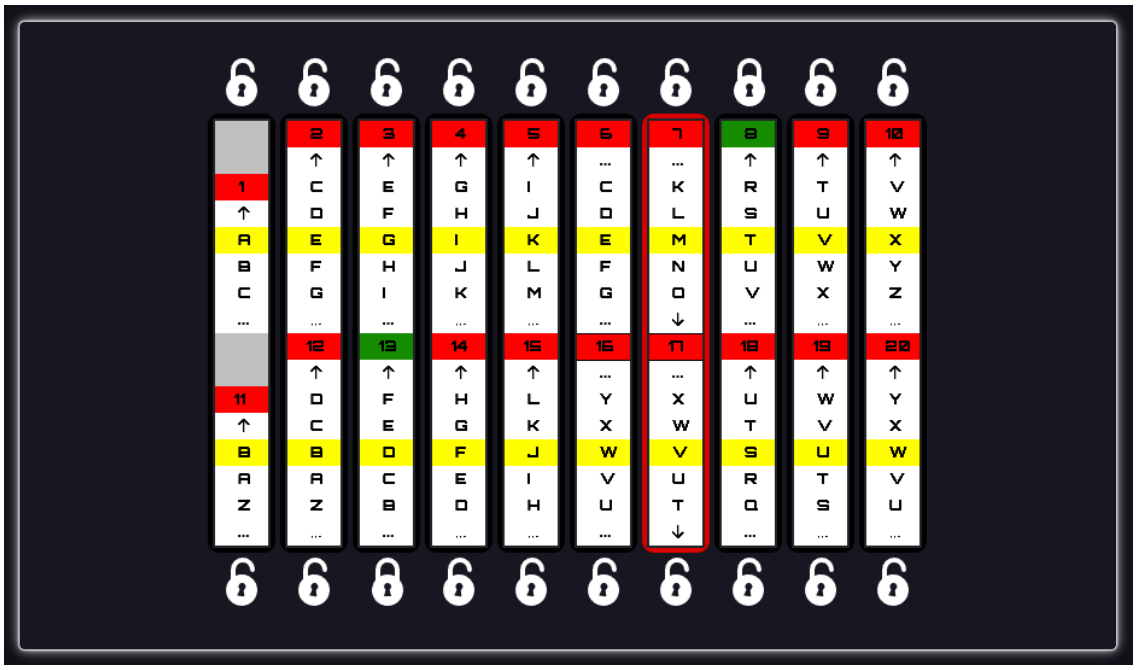


Figure 13: Part 2 of the Sphinx graphical interface

2.2.4 Part 3.1: Answer Interface

Part 3.1 contains the interface allowing the user to enter the message they are decrypting.

It will allow aligning the letters to be decrypted with the user's answer, and allows grouping the letters of the encrypted message by packets of n, where n corresponds to the number of selected strips. It allows either n letters per line, or 2n letters per line depending on n and the screen size. To achieve this result, most of the code is in a CSS file, and the display management based on screen size is done using a simple `if` in JavaScript.

```
function manage_submit_init() {
var mult = 1;
if (document.getElementById("option").offsetWidth <= CharWidthPixels * 3 *
    NUMBER_OF_STRIP * 4) {
mult = 1
} else {
mult = 2
}

    for (var i = 0; i < Math.floor(Math.floor(complete_encrypted_word.length /
        NUMBER_OF_STRIP) / mult); i++) {
        var inputElement = document.createElement("input");
...Initialization of inputElement...
inputElements.push(inputElement);
var cryptedElement = document.createElement("p");
cryptedElement.classList.add("crypted-hint");
var firstPart = complete_encrypted_word.substring(i * mult * NUMBER_OF_STRIP, i *
    mult * NUMBER_OF_STRIP + NUMBER_OF_STRIP);
var secondPart = complete_encrypted_word.substring(i * mult * NUMBER_OF_STRIP +
    NUMBER_OF_STRIP, i * mult * NUMBER_OF_STRIP + NUMBER_OF_STRIP * mult);
...Creation of the box containing the text to display...
}

    // Special case
```

```
21      if (complete_encrypted_word.length.mod(NUMBER_OF_STRIP * mult) != 0) {
22          Traitement
23      }
24  }
```

Explanation :

This code will manage the creation of elements to display and their layout.

First, we will check whether to display n or 2n letters per line (**line 3**).

Then, we will create a certain number of elements as long as the number of letters per line is n or 2n.

When we reach the end of the word, we will need to display the remaining letters whose number is strictly less than n or 2n.

To do this, we will perform a similar treatment, adding some subtleties in terms of accessors and indices in the loop on **line 21**.

Figure 14: Part 3.1 of the Sphinx graphical interface with n letters per line

Figure 15: Part 3.1 of the Sphinx graphical interface with 2n letters per line

2.2.5 Part 3.2: Options

When clicking on the gear icon, one can also access the options section.
In this section, various options are found, such as automatic strip locking, which makes the padlocks disappear and helps a user by locking and displaying in green the strips they place correctly.
When the option is activated, we will proceed in two steps: first, we will call an initialization function for the mode, which will remove the padlocks and check for each strip if it is correctly placed.
If this is the case, we will lock it using the previously presented method.
Once this is done, with each drag and drop performed, we will perform an additional check, looking to see if the two exchanged strips are correctly placed or not.
We will also need to manage the consistency of the elements if the option is deactivated.
We will then make the padlocks reappear, while locking those whose strips have been indicated as correctly positioned by the automatic locking option. The user is free to unlock them later.

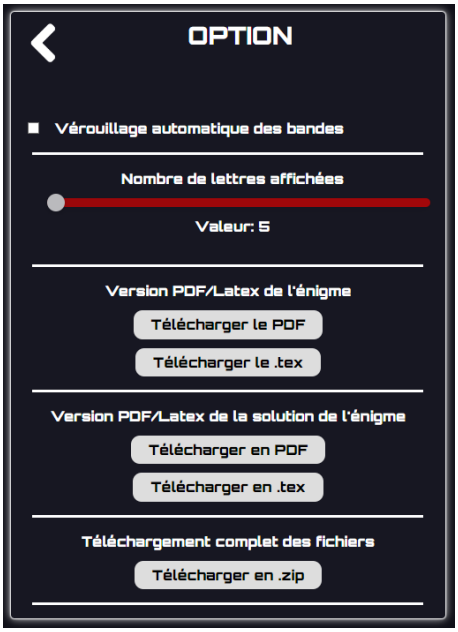


Figure 16: Part 3.2 of the Sphinx graphical interface

There is also a selector allowing the user to choose the number of letters they want to display on the screen per strip fragment. This option allows a user with a large screen to use it entirely, but is also useful for a user who might have difficulty making the connection between the compact version of the Sphinx and the version initially presented to them.
To update the number of letters, we will first wait for the user to select the desired number of letters. For this, we will add a 0.5-second delay before making the modifications. If the user has not touched the selector for 0.5 seconds, we will modify the display. This prevents the site from experiencing bugs/slowdowns due to calculations. The modification of the display will simply consist of resizing the canvas and redrawing it while respecting the consistency of locking and vertical movement.

Finally, in this section, it is possible to download the generated puzzle in different formats, PDF or LaTeX, but also to download all available representations so that the user can choose the one that suits them best. These files are generated with the same method as those presented above for the Enigma part. We were able to reuse the code already written in Python to implement LaTeX generations, but had to start from scratch to generate PDFs, as we are using a different technology.

2.2.6 Part 4: Help

Part 4 allows displaying the puzzle's solution directly on the page, helping a struggling user.

It represents the solution in the same form as the manipulable canvases, except here, a single canvas is used to represent all the strips. The user can thus read the message directly on the highlighted yellow areas, and can click on the arrows to see different parts of the solution. To draw the solution, we simply sort the strip array in the order described by the key, using the JavaScript translation of the Python functions presented above. Then, for the strip pair number n%10 of the list (where n is the current letter's position in the word), we will look for the index of the clear message's letter at position n, and draw it with the 4 surrounding letters.

We will perform this process each time the user clicks on one of the arrows to see another part of the solution, by calling the drawing function with the next/previous part of the word that was currently displayed.

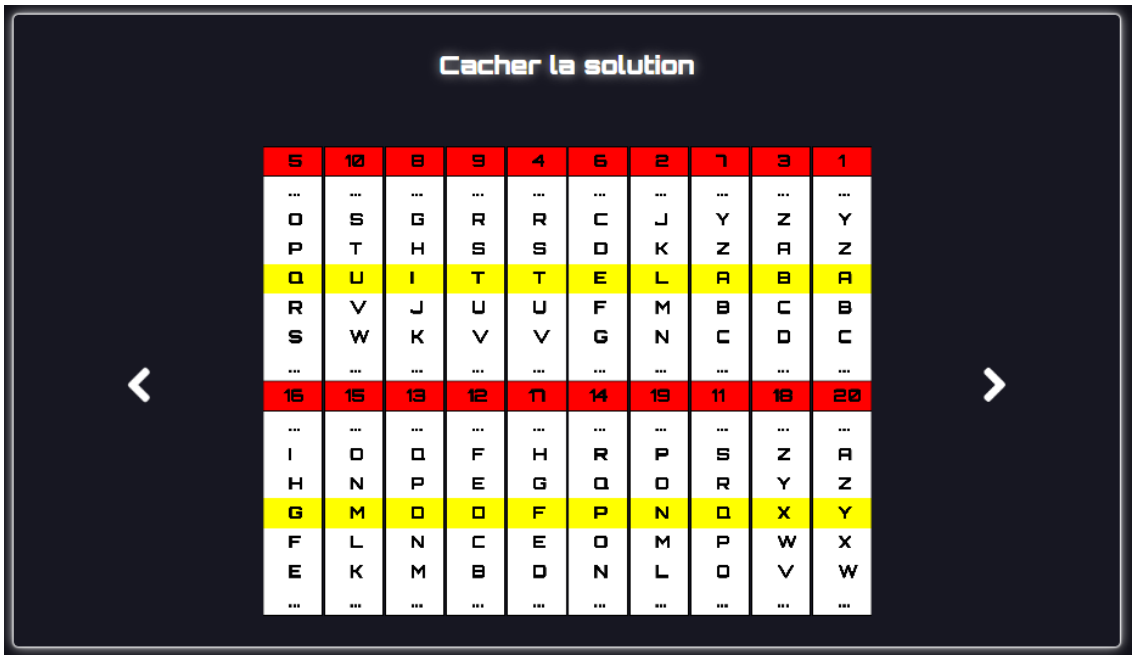


Figure 17: Part 4 of the Sphinx graphical interface

2.2.7 Part 5: Credits and References

The 5th part simply serves to mention the different people who worked on the page, and makes the design more visually appealing.
It also allows a return to the home page.

Figure 18: Part 5 of the Sphinx graphical interface

2.3 General website structure

The website is composed of 6 pages: a puzzle generation page for each machine (Enigma 1 rotor, Enigma 2 rotor, Sphinx), as well as a presentation page for Enigma 1 rotor/2 rotor and for Sphinx.

Finally, the site naturally includes a homepage where we explain our approach and the concept of the website.

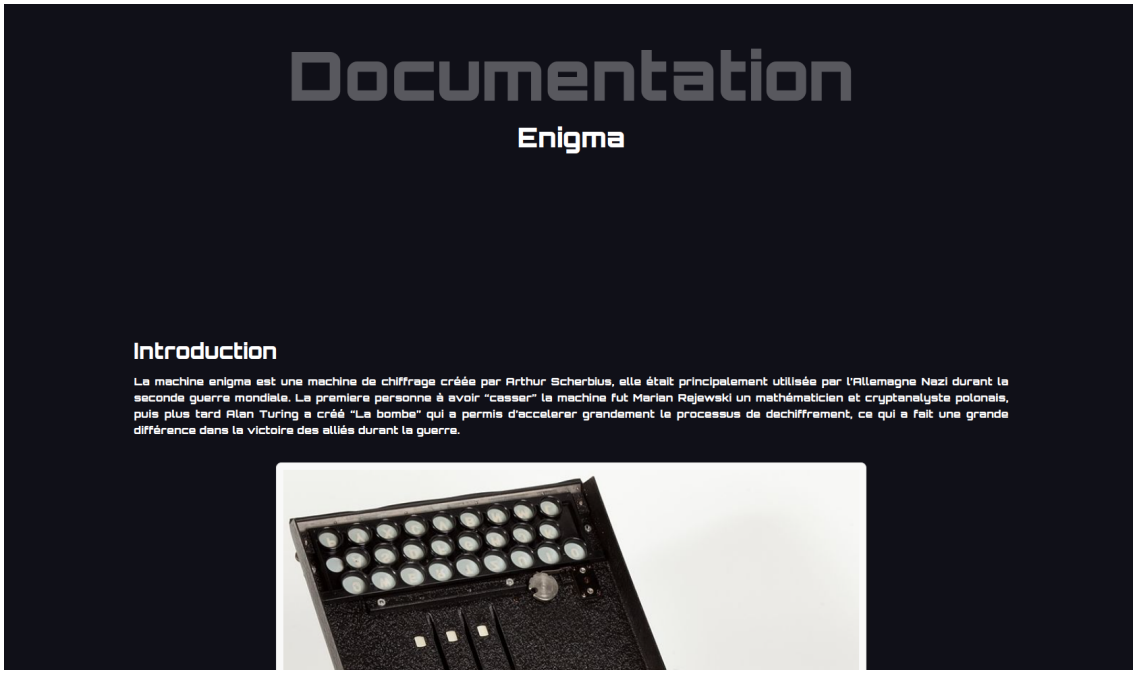


Figure 19: Enigma machine documentation page

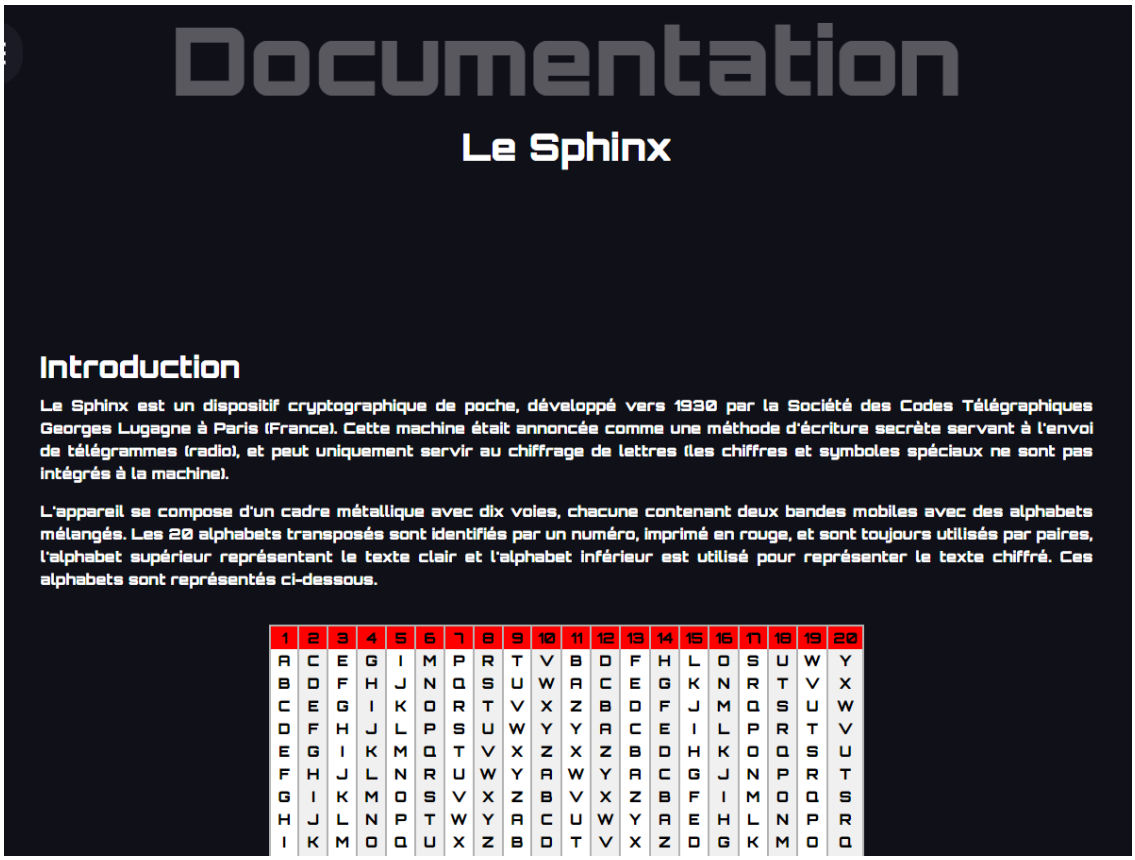


Figure 20: Sphinx machine documentation page

Conclusion

This project has been an extremely enriching and formative experience for us. It allowed us to discover and become familiar with new technologies and methodologies, while working on concrete and useful applications. It allowed us to acquire a lot of new knowledge and valuable experience in development and project management. Our regular meetings and teamwork strengthened our ability to collaborate effectively. The utilitarian and educational aspect of the project has been a constant source of motivation, giving meaning to our work.

We sincerely thank you for your guidance and support throughout this project.

Appendices

To whom it may concern,
I have found these 20 strips containing alphabets.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	C	E	G	I	M	P	R	T	V	B	D	F	H	L	O	S	U	W	Y
B	D	F	H	J	N	Q	S	U	W	A	C	E	G	K	N	R	T	V	X
C	E	G	I	K	O	R	T	V	X	Z	B	D	F	J	M	Q	S	U	W
D	F	H	J	L	P	S	U	W	Y	Y	A	C	E	I	L	P	R	T	V
E	G	I	K	M	Q	T	V	X	Z	X	Z	B	D	H	K	O	Q	S	U
F	H	J	L	N	R	U	W	Y	A	W	Y	A	C	G	J	N	P	R	T
G	I	K	M	O	S	V	X	Z	B	V	X	Z	B	F	I	M	O	Q	S
H	J	L	N	P	T	W	Y	A	C	U	W	Y	A	E	H	L	N	P	R
I	K	M	O	Q	U	X	Z	B	D	T	V	X	Z	D	G	K	M	O	Q
J	L	N	P	R	V	Y	A	C	E	S	U	W	Y	C	F	J	L	N	P
K	M	O	Q	S	W	Z	B	D	F	R	T	V	X	B	E	I	K	M	O
L	N	P	R	T	X	A	C	E	G	Q	S	U	W	A	D	H	J	L	N
M	O	Q	S	U	Y	B	D	F	H	P	R	T	V	Z	C	G	I	K	M
N	P	R	T	V	Z	C	E	G	I	O	Q	S	U	Y	B	F	H	J	L
O	Q	S	U	W	A	D	F	H	J	N	P	R	T	X	A	E	G	I	K
P	R	T	V	X	B	E	G	I	K	M	O	Q	S	W	Z	D	F	H	J
Q	S	U	W	Y	C	F	H	J	L	L	N	P	R	V	Y	C	E	G	I
R	T	V	X	Z	D	G	I	K	M	K	M	O	Q	U	X	B	D	F	H
S	U	W	Y	A	E	H	J	L	N	J	L	N	P	T	W	A	C	E	G
T	V	X	Z	B	F	I	K	M	O	I	K	M	O	S	V	Z	B	D	F
U	W	Y	A	C	G	J	L	N	P	H	J	L	N	R	U	Y	A	C	E
V	X	Z	B	D	H	K	M	O	Q	G	I	K	M	Q	T	X	Z	B	D
W	Y	A	C	E	I	L	N	P	R	F	H	J	L	P	S	W	Y	A	C
X	Z	B	D	F	J	M	O	Q	S	E	G	I	K	O	R	V	X	Z	B
Y	A	C	E	G	K	N	P	R	T	D	F	H	J	N	Q	U	W	Y	A
Z	B	D	F	H	L	O	Q	S	U	C	E	G	I	M	P	T	V	X	Z

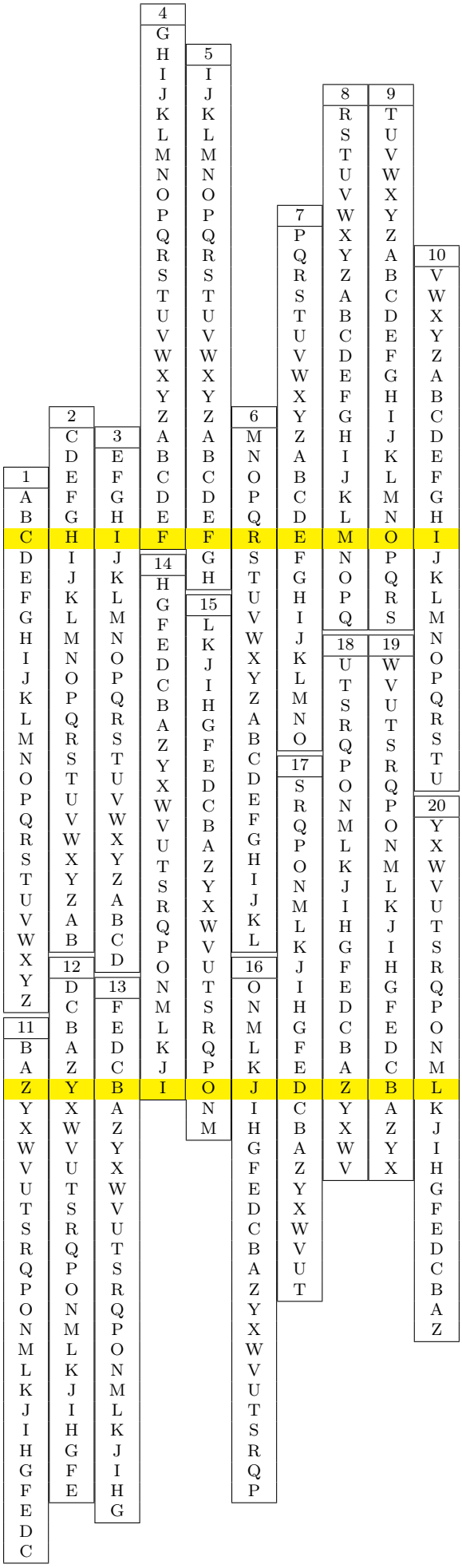
I also have a photo of the pocket encryption machine SPHINX from the Société des Codes Télégraphiques of George Lugagne in Marseille. In this photo, the plaintext at the top CHIFFREMOI is encrypted as ZYBIOJDZBL using the key (1,11) (2,12) (3,13) (4,14) (5,15) (6,16) (7,17) (8,18) (9,19) (10,20).

I also discovered an encrypted letter containing the following:

CODYEZRTEOIHUXDBQ

From my studies, I know that every message from this secret agent begins with the word LEVENTSOUF. Moreover, the key used is a permutation of strips 1 to 10 for the top part, and a permutation of strips 11 to 20 for the bottom part.

Another source gave me 4 strip combinations: (4,14) (3,20) (10,12) (1,17).


1
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
11
B A Z Y X W V U T S R Q P O N M L K J I H G F E D C
2
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
12
D C B A Z Y X W V U T S R Q P O N M L K J I H G F E
3
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
13
F E D C B A Z Y X W V U T S R Q P O N M L K J I H G
4
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
14
H G F E D C B A Z Y X W V U T S R Q P O N M L K J I
5
I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
15
L K J I H G F E D C B A Z Y X W V U T S R Q P O N M
6
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
16
O N M L K J I H G F E D C B A Z Y X W V U T S R Q P
7
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
17
S R Q P O N M L K J I H G F E D C B A Z Y X W V U T
8
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
18
U T S R Q P O N M L K J I H G F E D C B A Z Y X W V
9
T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
19
W V U T S R Q P O N M L K J I H G F E D C B A Z Y X
10
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
20
Y X W V U T S R Q P O N M L K J I H G F E D C B A Z

Will you be able to find the key to this puzzle and decrypt the message?
Good luck!