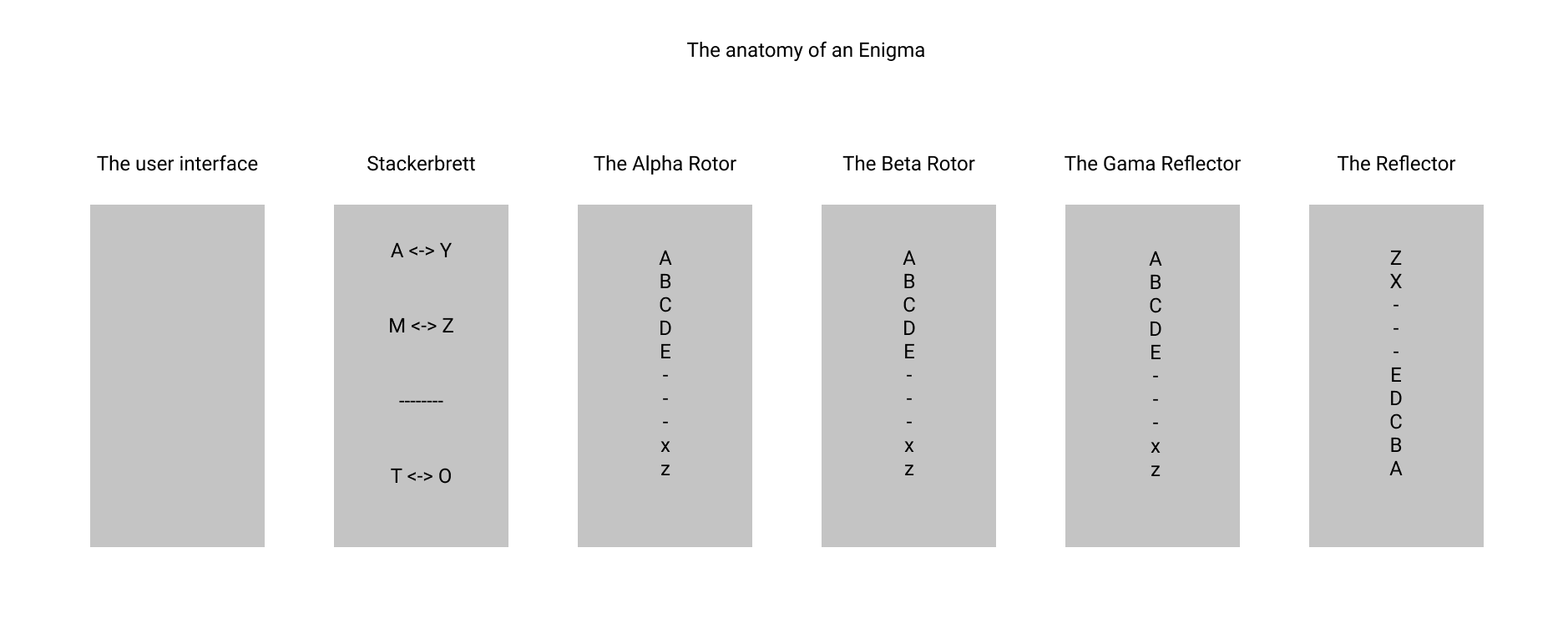
ENIGMA CRYPTOGRAPHIC ALGORITHM

The 1939–1944 was a dark period in the history of humanity. The second World War killed 60 millions of people. But it could have taken more than 5 years if the Allies didn’t crack one of the most powerful machines built up by Nazy Germany — Enigma.

Enigma — one of the most powerful encryption device built in that period. Created by [Arthur Scherbius](https://en.wikipedia.org/wiki/Arthur_Scherbius) and [Richard Ritter](https://de.wikipedia.org/wiki/Ernst_Richard_Ritter), at first the navy of Germany didn’t pay any attention to it’s invention. But in 1926 the navy started to build its own Enigmas. And that gave the German army a big advantage in this bloody war.

Enigma had a very sophisticated method of encrypting and decrypting messages. It took a lot of time and brain power to break it. But the most interesting fact is that it is based on one of the most vulnerable algorithm of encryption — **Caesar cipher**.

Basically an enigma is built up from 4 main parts - A user interface, a *Steckerbrett,* 3 rotor (could be more) and a reflector. Each part is doing a particular task, but put all together they can encrypt a message in such a way that it could be impossible to decrypt without knowing the initial settings of the enigma.

The anatomy of an Enigma

Also one of the main features of enigma is that for correct decryption the user needs the initial settings of the enigma when the message was encrypted. If one setting is different, then the whole task becomes impossible.

So let’s begin.

The User Interface

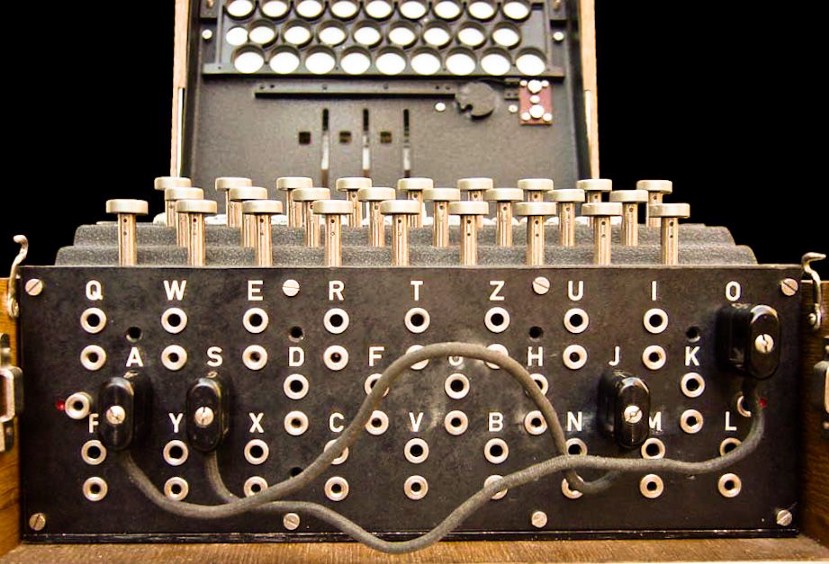


The enigma keyboard.

The user interface is very simple — it is represented by a keyboard of 26 letters. It is used to insert the input in the machine. But also, the pressed keys activate a mechanism that changes the state of the first rotors, rotating it with one unit, thus changing the encryption key after every press of the a certain button.

**The Steckerbrett**

https://miro.medium.com/max/60/1*PoFzu8Af3iQP81OpLw_gYg.jpeg?q=20

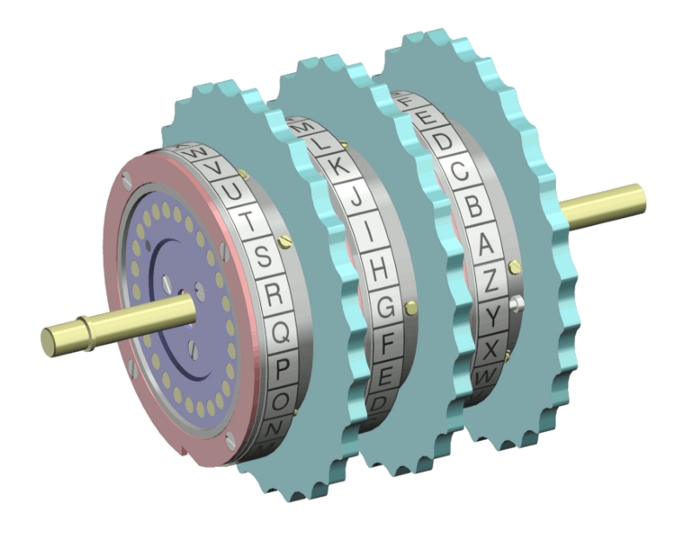


The Steckerbrett

The stackerbrett is a tableau of sockets is an another mechanism that that was created to make the decryption mechanism harder. But actually it was the weakest part of the machine. It is a mechanism that connects 2 letters one from the input layer and another from the output layer. That encodes theinput letter as the output layer connected to it, without using the next parts of the machine. However, even if such a letter is pressed the rotors are turned anyway.

**The rotors**

https://miro.medium.com/max/60/1*bt2Wr4hTPlAKaH-Pvb4zHQ.png?q=20



The enigma rotors.

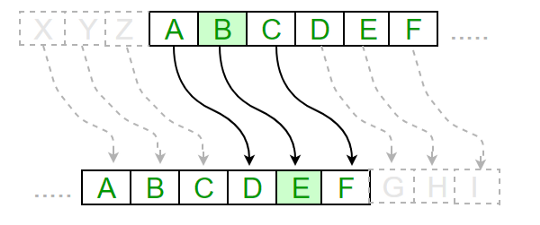
The rotors are special gears with 26 pins. Every pin is related to an English letter. When an electric signal (in mechanical machine) comes to a rotor, it will be passed to the next rotor at the same place. At positions (0, 0, 0) every letter will be encoded as itself. But at state (1, 0, 0) the letter A will be encoded as B, B as C and so on. But wait, why?

The answer lies in the caesar cipher algorithm of encryption. Let’s take a look on how it works.

Firstly we need an ordered alphabet → A B C D E F G H …

Now, to encrypt we need a key K that we will be used to encrypt our message. K is a positive integer and it encodes every letter as the letter that is after K steps after the initial letter. In the image below the K = 3.

https://miro.medium.com/max/60/1*VTWBF_xrygGGaRy2aL0E3g.png?q=20



Caesar cipher when K = 3

In the case showed above A will be replaced with D, B with E, C with F and so on.

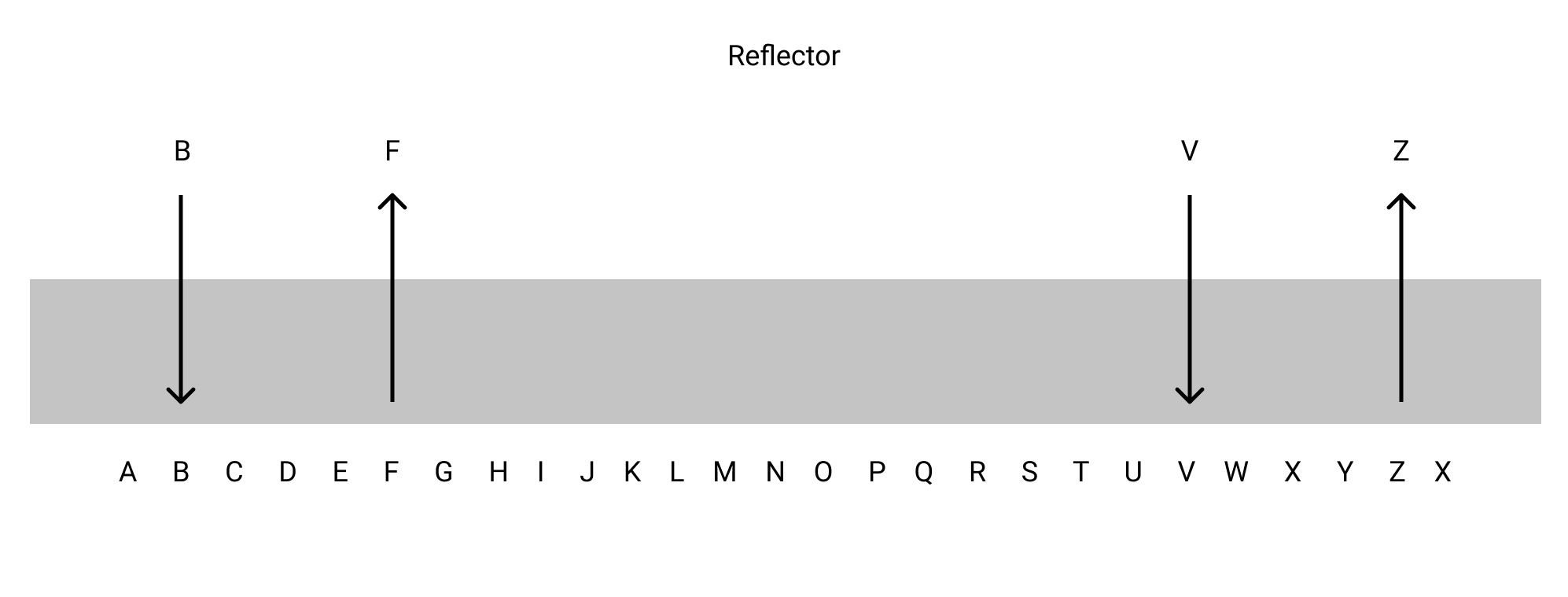
In such way each rotor encrypts the message. The variable K in permutations in the algorithm is actually the number of turns from the 0 — state.

But that’s not the end, after every 26-th rotation of the first rotor, the second also does a rotation.

In the same way does and the third rotor, after every 26-th rotation of second rotor, it does a rotation, but unlikely to the first pair of rotors, the second also takes a turn when the third takes one.

**The reflector**

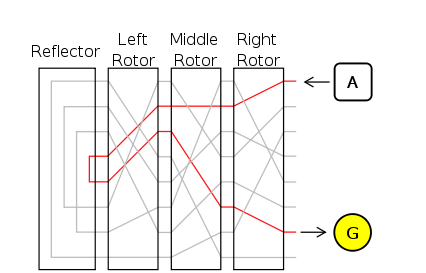
The reflector is another part of the enigma. It actually gives the opportunity to the machine to use the same settings for encryption and decryption. It works in the following way:

The structure of a reflector.

**How all this works?**

There are a lot of parts, and every one has its own principle of work, and you may be confused how it all works together. It’s easier than you could imagine so let’s start from the beginning.

Suppose we press a letter on the keyboard. If it is connected to the stackerbrett, then it will be encoded as the end letter form this connection. If not, it will go through the rotors, changing itself after every rotor. After the letter exists from the third rotor it enters in the reflector, there it is changed with it’s reflection. Then the reflection enters in the rotor number 3, then changes it, and the second and first rotors also will change it, giving the new encryption.



How enigma acutally is encripting an character.

Now let’s implement it in code.

Python code.

Firstly, we will need to import the necessary libraries and packages.

# Importing all libraries  
# We need ascii\_lowercase from string to get the english alphabet  
from string import ascii\_lowercase  
# We need json library to add the possibility for enigma to import settings from a json format  
import json

* From string — ascii\_lowercase — for generating the alphabet list.
* json — for uploading the enigma setting from a file in json format.

After that we create a class — enigma, that will represent our enigma, and the \_\_init\_\_ function for setting the enigma, where we must set the alphabet (**self.alphabet**) and the steckerbrett (**self.steckerbrett**). There are 3 methods:

* From the file — importing all settings from a json file.

def \_\_init\_\_(self, steckerbrett = {" ":" "}, settings\_file=None, alpha=None, beta=None, gama=None):  
 *''' The initial setting of enigma before the encryption '''* # Creating a list of all alphabet letters  
 self.alphabet = list(ascii\_lowercase)  
  
 '''  
 Steckerbrett is a system of sockets that connects pairs of letters that are interchanged between them,  
 without going throw all the rotors of enigma  
 '''  
 self.steckerbrett = steckerbrett  
 if settings\_file != None:  
 ''' If the setting sites is got then we load the setting from it as a json format '''  
 try:  
 # I verify if there is a such file with setting that we got  
 self.settings = json.load(open(settings\_file, 'r'))[0]  
 except IOError as e:  
 # The first enigma error - There is no such a settings file  
 print("Enigma Error 1: There is no such setting file")  
 finally:  
 # steckerbratt -> a dictionary with pairs of interchangeable pairs of letters  
 self.steckerbrett = self.settings['steckerbrett']  
 # Setting the states of rotors  
 self.alpha = self.settings['alpha']  
 self.beta = self.settings['beta']  
 self.gama = self.settings['gama']

* Manual setting.

elif alpha != None and beta != None and gama != None and steckerbrett != None:  
 ''' Setting the rotors and the steckerbrett manually '''  
 if type(steckerbrett) is not dict:  
 self.steckerbrett = {" " : " "}  
 self.alpha = alpha  
 self.beta = beta  
 self.gama = gama

* Default setting of all rotors.

else:  
 # Setting all rotors to base states and steckerbrett to have only space case  
 if type(steckerbrett) is not dict:  
 self.steckerbrett = {" " : " "}  
 rotors = [self.alpha, self.beta, self.gama]  
 for rotor in rotors:  
 if rotor == None or type(rotor) is not int or type(rotor) is not float:  
 rotor = 0  
 else:  
 rotor = rotor % 26  
 self.alpha = rotors[0]  
 self.beta = rotors[1]  
 self.gama = rotors[2]

The next thing that we must do in the \_\_init\_\_ function is to make the last changes in stackerbrett and set the reflector.

# Making the steckerbrett interchangeable and removing these pairs from the alphabet  
for letter in list(self.steckerbrett.keys()):  
 if letter in self.alphabet:  
 self.alphabet.remove(letter)  
 self.alphabet.remove(self.steckerbrett[letter])  
 self.steckerbrett.update({self.steckerbrett[letter]:letter})  
# Setting the reflector  
self.reflector = [leter for leter in reversed(self.alphabet)]

The next function of permutation. The heart of the encryption algorithm. All rotors — alpha, beta and gamma, are deciding the Key — K for the caesar cipher.

def permutate(self, rotor):  
 *''' This function is permutatting the alphabet depending on the rotors settings '''* new\_alphabet = ''.join(self.alphabet)  
 new\_alphabet = list(new\_alphabet)  
 for iter in range(rotor):  
 new\_alphabet.insert(0, new\_alphabet[-1])  
 new\_alphabet.pop(-1)  
 return new\_alphabetdef inverse\_permutation(self, rotor):  
 *''' This function is permutatting the alphabet depending on the rotors settings on the back way '''* new\_alphabet = ''.join(self.alphabet)  
 new\_alphabet = list(new\_alphabet)  
 for iter in range(rotor):  
 new\_alphabet.append(new\_alphabet[0])  
 new\_alphabet.pop(0)  
 print(self.alphabet)  
 print(new\_alphabet)  
 return new\_alphabet

And, the most important part — the encryption process.

Firstly, we convert every letter from text to lower and convert it to a list.

def encrypt\_text(self, text):  
 *''' This function encrypts a string '''* encrypted\_text = []  
 # Text preprocessing  
 text = text.lower()  
 text.split()

After that, we are going through the list and we are processing every letter. If the letter is in the stackerbrett than we just return the value that it is connected to. And of course it will go through a process of rotating the rotors in enigma system.

for letter in text:  
 # Checking if the letter is in steckerbrett  
 if letter in self.steckerbrett:  
 # If it is, the we encrypt it as it's pair  
 encrypted\_text.append(self.steckerbrett[letter])  
 # Turning the rotors  
 self.alpha += 1  
 if self.alpha % 26 == 0:  
 self.beta += 1  
 self.alpha = 0  
 if self.beta % 26 == 0 and self.alpha % 26 != 0 and self.beta >= 25:  
 self.gama += 1  
 self.beta = 1

Else, we encrypt every letter by the settings of every rotor, we find the corresponding index of the letter in the alphabet list (of course after we remove the letters from stackerbrett). We are repeating this process with the every rotor using the output of every preceding rotor.

else:  
 # Encrypting throw rotors  
 # Letter is encrypted by first rotor  
 temp\_letter = self.permutate(self.alpha)[self.alphabet.index(letter)]  
 # Letter is encrypted by second rotor  
 temp\_letter = self.permutate(self.beta)[self.alphabet.index(temp\_letter)]  
 # Letter is encrypted by third rotor  
 temp\_letter = self.permutate(self.gama)[self.alphabet.index(temp\_letter)]

After the last rotor — gamma returns the letter, it is “reflected” by the reflector, returning the mirror of that letter.

# Reflector is returning the inverse of that letter  
temp\_letter = self.reflector[self.alphabet.index(temp\_letter)]

There we start the reversed way through the rotors, in the next order — gamma, beta, alpha.

# Back way  
# Letter is encrypted by third rotor  
temp\_letter = self.inverse\_permutation(self.gama)[self.alphabet.index(temp\_letter)]  
print("gama - {}".format(temp\_letter))  
# Letter is encrypted by second rotor  
temp\_letter = self.inverse\_permutation(self.beta)[self.alphabet.index(temp\_letter)]  
print("beta - {}".format(temp\_letter))  
# Letter is encrypted by first rotor  
temp\_letter = self.inverse\_permutation(self.alpha)[self.alphabet.index(temp\_letter)]  
print("alpha - {}".format(temp\_letter))  
encrypted\_text.append(temp\_letter)  
print(temp\_letter)

After that we just join all encrypted letters from the list, and get the encrypted text.

Now let’s see an example:

Enigma = enigma({"b":'a', ' ':' ', 'e':'z'}, alpha=5, beta=17, gama=24)

I created an enigma using manually setting, but setting it from a file also work. Now let’s encrypt a string, I chose a phrase that I got from one of my colleagues —**“there is no time”.**

print(Enigma.encrypt\_text('there is no time'))

And the result is:

iuzkz tj on itpz

When I try it backward I get the same phrase:

print(Enigma.encrypt\_text('iuzkz tj on itpz'))  
>>>there is no time

Succes, it works.