Elementary statistical methods of cryptography

Auteur : Etienne, Elodie
Promoteur(s) : Haesbroeck, Gentiane
Faculté : Faculté des Sciences
Diplôme : Master en sciences mathématiques, à finalité didactique
Année académique : 2018-2019
URI/URL : https://drive.google.com/drive/folders/1tPog2Yvfex2z_Sy2Xg5mvjS5kcQDZGbr?usp=sharing; http://hdl.handle.net/2268.2/6978

University of Liège

Master's thesis

Elementary statistical methods of cryptography

Author:
Elodie Etienne

Supervisor:
Pr. Gentiane Haesbroeck

A thesis submitted in fulfillment of the requirements for the Master's degree in mathematics

Department of mathematics

Faculty of Sciences

Academic year 2018-2019

The art of writing secret messages – intelligible to those who are in possession of the key and unintelligible to all others – has been studied for centuries. The usefulness of such messages, especially in time of war, is obvious; on the other hand, their solution may be a matter of great importance to those from whom the key is concealed. But the romance connected with the subject, the not uncommon desire to discover a secret, and the implied challenge to the ingenuity of all from who it is hidden have attracted to the subject the attention of many to whom its utility is a matter of indifference.

— Abraham Sinkov, *Mathematical Recreations & Essays*

Acknowledgements

I would like to thank my promotor Pr. Gentiane Haesbroeck who accepted to help me follow a project that was not her area of expertise. I thank her for her support and professional guidance. She consistently allowed this thesis to be my own work but steered me in the right direction with her valuable ideas and remarks whenever I needed it.

I would also like to acknowledge Pr. Yvik Swan as the second reader of this thesis and I am gratefully indebted to his comments regarding my English writing.

I would like to thank all members of the jury for agreeing to read the manuscript and to participate in the defense of this thesis.

I would also like to thank my dear friend Tom Demaret for his help concerning the beginning of Chapter 4 and for his support for years.

Finally, I must express my gratitude to my family for providing me with unfailing support and continuous encouragements throughout my years of study.

Contents

Introduction **1**

1 Encrypting a message **3**
1.1 Introduction . . . 3
1.2 Definitions . . . 4
1.3 Different types of ciphers . . . 5
1.3.1 Substitution cipher . . . 5
1.3.2 Permutation/Transposition cipher . . . 6
1.3.3 Vigenère Cipher . . . 8
1.4 Enciphered machines during the Second World War . . . 9
1.4.1 The Enigma machine . . . 9
1.4.2 The Lorenz machine . . . 12

2 Decrypting a message **16**
2.1 Chi-squared test . . . 16
2.2 Substitution cipher . . . 17
2.2.1 Step by step frequency analysis . . . 17
2.2.2 Statistical method for frequency analysis . . . 22
2.3 Permutation/Transposition cipher . . . 24
2.3.1 Known period . . . 26
2.3.2 Unknown period . . . 27
2.4 The Vigenère cipher . . . 27
2.4.1 Known period . . . 28
2.4.2 Unknown period . . . 30
2.5 The Enigma machine . . . 35
2.6 The Lorenz machine . . . 38

3 Alan Mathison Turing **40**
3.1 Biography . . . 40
3.2 The Applications of Probability to Cryptography . . . 42

4 Decryption using Markov Chain Monte Carlo **48**
4.1 Markov Chain: definitions . . . 48
4.2 General algorithm . . . 49
4.3 Matrix of bigrams . . . 50
4.3.1 Transition matrix for different English texts . . . 51
4.3.2 Transition matrix for different languages . . . 51
4.4 Adaptation of the Metropolis Algorithm for decrypting a substitution ciphertext 52
4.4.1 Plausibility function . . . 52
4.4.2 Algorithm . . . 53

4.4.3 Implementation . . . 54
4.4.4 Results . . . 54
4.5 Adaptation of the Metropolis Algorithm for decrypting a permutation/transposition ciphertext . . . 60
4.5.1 Plausibility function . . . 60
4.5.2 Slide moves . . . 60
4.5.3 Algorithm . . . 61
4.5.4 Implementation . . . 62
4.5.5 Results . . . 62
4.5.6 Finding the unknown period of the transposition cipher . . . 65

5 Comparisons based on simulations **67**
5.1 Data generation . . . 67
5.2 Performance measures of the techniques . . . 68
5.3 Simulations . . . 70
5.3.1 Vigenère cipher . . . 70
5.3.2 Substitution cipher . . . 74
5.3.3 Transposition cipher . . . 74

Summary **76**

Appendix A R code **A1**

Appendix B Lorenz addition **B1**

Introduction

The word *cipher* refers to the processes of encrypting and decrypting messages. The first cipher was a substitution of hieroglyphs where hieroglyphs were replaced by others (used[1] about 1900 B.C). Over the years, the mechanism has grown in sophistication especially when military secrecy has been involved.

While *cryptography* is the study of constructing ciphers, *cryptanalysis* is the study of breaking them. The first techniques for breaking ciphers were developed in the 9^{th} century by an Arab mathematician named Al-Kindi and were based on statistical inference. Over the years, the methods have changed but they have always been based on statistics.

I really became interested in cryptography when I saw Morten Tyldum's movie *The Imitation Game* (2014). This film tells the incredible and true story of Alan Turing and the people of Bletchley Park who, during the Second World War, cracked, among others, the reputedly unbreakable German encryption device, the Enigma machine.

After visiting Bletchley Park on February 2018 and after some research, I found out that Turing wrote articles about statistical methods that can break ciphers. I therefore investigated if it could be a subject for a master's thesis and outlined the topic myself.

Thus, this master's thesis will be devoted to the study of a subject that combines probability, statistics and cryptography. At the beginning I wanted to describe the contribution of statistics and statisticians during the wars, with a particular focus on the Second World War and Alan Turing's contribution. We extended the subject to statistical methods allowing to decrypt messages where a brute attack will be unfeasible. Nowadays, the ciphers presented in this thesis are no longer in use, except for the playful, educational or historical context, because they are breakable. However, the statistics behind them are still in use which justifies this work.

The goal of this thesis is to break ciphers in use before 1940 (called *pre-modern ciphers*) such as substitution cipher, transposition cipher and Vigenère cipher. In *transposition cipher* the positions of the letters in the text are just changed and in *substitution cipher*, letters are replaced by others. A particular case of substitution cipher is the *Caesar cipher* where each letter is shifted by a fixed number of places in the alphabet. In *Vigenère cipher*, several Caesar ciphers are used to encrypt the text according to a repeating keyword.

Additional ciphers used in the Second World War will be presented and the methods for breaking them will be partly presented.

The main references of this work will be: Sarkar's *On some connections between statistics and cryptography* (see [29]), Turing's *The Applications of Probability to Cryptography* (see [32]), Diaconis' *The Markov Chain Monte Carlo Revolution* (see [8]), Connor's *Simulation and Solving*

[1] According to D. Kahn (see [19]).

Substitution Codes (see [5]) and Chen and Rosenthal's *Decrypting classical cipher text using Markov chain Monte Carlo* (see [4]).

The first chapter will describe encryption techniques used in pre-modern cryptography and in the Second World War.

The second chapter will present methods for breaking classical ciphers introduced in the first chapter with a brief presentation of the cryptanalysis needed to break wartime ciphers.

The third chapter will present a bayesian method introduced by Turing in a wartime paper for breaking Vigenère cipher. Before that, a brief biography will be presented.

The fourth chapter will investigate the use of Markov Chain Monte Carlo theory to attack substitution cipher and transposition cipher.

The fifth and last chapter will measure the performance of some techniques presented in the previous chapters based on simulations.

My contribution to this work was to implement on my own all the ciphers[2], methods and algorithms presented and to illustrate all the concepts with the same text throughout this thesis.

As I spent six months in Southampton on an Erasmus stay, I decided to write this work in English.

[2]Except the Enigma machine and the Lorenz machine.

Chapter 1

Encrypting a message

This chapter will present some pre-modern techniques to encrypt messages. It will also describe two enciphered machines used in the Second World War. It will follow Palash Sarkar's article (see [29]), some definitions will be taken from Michel Rigo's lectures notes (see [27]) and some historical notes will be based on James Grime's references: [13] to [16].

1.1 Introduction

Throughout the following chapters, we will encrypt and decrypt the same text as illustrations.

We will use the R software; the code can be found in Appendix A.

We chose the lyrics of a song by the Irish group U2 written in 2013: *Ordinary Love*. However, we didn't repeat the chorus and didn't abbreviate some words. This is the text:

The sea wants to kiss the golden shore

The sunlight warms your skin

All the beauty that is been lost before wants to find us again

I cannot fight you anymore it is you I am fighting for

The sea throws rock together

But time leaves us polished stones

We cannot fall any further

If we cannot feel ordinary love

And we cannot reach any higher

If we cannot deal with ordinary love

Birds fly high in the summer sky

And rest on the breeze

The same wind will take care of you and I

We will build our house in the trees

Your heart is on my sleeve

Did you put there with a magic marker

For years I would believe

That the world could not wash it away

Removing line breaks, using uppercase letters and allowing 60 (arbitrary choice) characters per line, the text becomes

```
THE SEA WANTS TO KISS THE GOLDEN SHORE THE SUNLIGHT WARMS YO
UR SKIN ALL THE BEAUTY THAT IS BEEN LOST BEFORE WANTS TO FIN
D US AGAIN I CANNOT FIGHT YOU ANYMORE IT IS YOU I AM FIGHTIN
G FOR THE SEA THROWS ROCK TOGETHER BUT TIME LEAVES US POLISH
ED STONES WE CANNOT FALL ANY FURTHER IF WE CANNOT FEEL ORDIN
ARY LOVE AND WE CANNOT REACH ANY HIGHER IF WE CANNOT DEAL WI
TH ORDINARY LOVE BIRDS FLY HIGH IN THE SUMMER SKY AND REST O
N THE BREEZE THE SAME WIND WILL TAKE CARE OF YOU AND I WE WI
LL BUILD OUR HOUSE IN THE TREES YOUR HEART IS ON MY SLEEVE D
ID YOU PUT THERE WITH A MAGIC MARKER FOR YEARS I WOULD BELIE
VE THAT THE WORLD COULD NOT WASH IT AWAY.
```

1.2 Definitions

Definition 1.2.0.1. A *message* is a sequence of symbols over some fixed alphabet Σ containing N symbols.

There exists a bijection between the alphabet Σ and the set $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$. For example, figure 1.1 shows the the bijection between the usual alphabet where the space character is added (it will be denoted by "_" when it has to be seen) and the set $\mathbb{Z}_{27}$.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
13	14	15	16	17	18	19	20	21	22	23	24	25	26

Figure 1.1: The alphabet of 27 elements

In what follows we will use this alphabet. We will sometimes refer to the elements of the alphabet as *letters* even if the space symbol is not a letter in the common sense.

In order to encrypt and decrypt a text, there is a pair of algorithms that take a key and convert a text, called the *plaintext* to an encrypted text, called *ciphertext*, and back.

For each key k there exists an encryption rule e_k such that, for each text t,

$$e_k : t \longmapsto e_k(t)$$

and there exists an decryption rule d_k such that, for each encrypted text c,

$$d_k : c \longmapsto d_k(c) \qquad \text{and} \qquad \text{for all text } t,\ d_k(e_k(t)) = t.$$

Let $\mathcal{K}$ denote the set of all keys, the *key space*.

The domain of the functions e_k ($\forall k \in \mathcal{K}$) is denoted as $\mathcal{P}$. By definition, $\mathcal{P}$ is the range of the functions d_k ($\forall k \in \mathcal{K}$).

The range of the functions e_k ($\forall k \in \mathcal{K}$) is denoted as $\mathcal{C}$. By definition, $\mathcal{C}$ is the domain of the functions d_k ($\forall k \in \mathcal{K}$).

The word *cipher* refers to the algorithms of encryption and decryption and the corresponding cryptosystem is the triplet $(\mathcal{P}, \mathcal{C}, \mathcal{K})$.

A cryptosystem is said to be *mono-alphabetic* when the encryption rule e_k applies to a single symbol of the alphabet of N elements (or $\mathbb{Z}_N$) at a time and the same function is applied to each element of $\mathcal{P}$. A cryptosystem is said to be *poly-alphabetic* when the encryption rule e_k applies to m symbols of the alphabet of N elements (or $\mathbb{Z}_N$) at the same time. In such a cipher, a single letter can be encrypted by various letters depending on where it is in the message.

1.3 Different types of ciphers

1.3.1 Substitution cipher

A first easy way to encrypt a text is to use a *substitution cipher* which is a kind of mono-alphabetic cipher.

In the case of the alphabet Σ, the cryptosystem is $(\mathcal{P}, \mathcal{C}, \mathcal{K})$ where $\mathcal{P} = \mathcal{C} = \mathbb{Z}_N$ and $\mathcal{K}$ is the set of all the permutations of $\{0, 1, \ldots, N-1\}$ and contains $N!$ elements. If π denotes a permutation of the set Σ, then for the key $k = \pi$, we have

$$e_k(i) = \pi(i) \ \forall i \in \mathcal{P} \qquad \text{and} \qquad d_k(j) = \pi^{-1}(j) \ \forall j \in \mathcal{C}.$$

Example 1.3.1.1. Consider the alphabet defined in section 1.2 and the message to be sent is `THE_SEA_WANTS_TO_KISS`. If the permutation is

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 \\ 8 & 6 & 13 & 1 & 10 & 22 & 17 & 7 & 21 & 3 & 18 & 14 & 4 & 5 & 9 & 26 & 2 & 20 & 12 & 24 & 11 & 15 & 16 & 19 & 23 & 0 & 25 \end{pmatrix},$$

the enciphered message is `YHKZMKIZQIFYMZYJZSVMM`.

The whole chosen text is encrypted as:

```
YHKZMKIZQIFYMZYJZSVMMZYHKZRJOBKFZMHJUKZYHKZMLFOVRHYZQIUEMZXJ
LUZMSVFZIOOZYHKZGKILYXZYHIYZVMZGKKFZOJMYZGKWJUKZQIFYMZYJZWVF
BZLMZIRIVFZVZNIFFJYZWVRHYZXJLZIFXEJUKZVYZVMZXJLZVZIEZWVRHYVF
RZWJUZYHKZMKIZYHUJQMZUJNSZYJRKYHKUZGLYZYVEKZOKIPKMZLMZ_JOVMH
KBZMYJFKMZQKZNIFFJYZWIOOZIFXZWLUYHKUZVWZQKZNIFFJYZWKKOZJUBVF
IUXZOJPKZIFBZQKZNIFFJYZUKINHZIFXZHVRHKUZVWZQKZNIFFJYZBKIOZQV
YHZJUBVFIUXZOJPKZGVUBMZWOXZHVRHZVFZYHKZMLEEKUZMSXZIFBZUKMYZJ
FZYHKZGUKKAKZYHKZMIEKZQVFBZQVOOZYISKZNIUKZJWZXJLZIFBZVZQKZQV
OOZGLVOBZJLUZHJLMKZVFZYHKZYUKKMZXJLUZHKIUYZVMZJFZEXZMOKKPKZB
VBZXJLZ_LYZYHKUKZQVYHZIZEIRVNZEIUSKUZWJUZXKIUMZVZQJLOBZGKOVK
PKZYHIYZYHKZQJUOBZNJLOBZFJYZQIMHZVYZIQIX
```

An easy substitution cipher is the *Caesar cipher with shift* k where each letter is replaced by the letter which is k positions further in the alphabet[1]. The cryptosystem is thus $(\mathcal{P}, \mathcal{C}, \mathcal{K})$ with $\mathcal{P} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_{27} =$ and $\forall k \in \mathcal{K}$,

$$e_k(i) = i + k \pmod{27} \ \forall i \in \mathcal{P} \qquad \text{and} \qquad d_k(j) = j - k \pmod{27} \ \forall j \in \mathcal{C}.$$

It was named after Julius Caesar who described the process in the Gallic Wars.

Example 1.3.1.2. As in example 1.3.1.1, consider the message `THE_SEA_WANTS_TO_KISS`. If a shift of 20 is used, each symbol in the message becomes the symbol 20 positions further in the alphabet.

Plaintext	T	H	E	_	S	E	A	_	W	A	N	T	S	_	T	O	_	K	I	S	S
	19	7	4	26	18	4	0	26	22	0	13	19	18	26	19	14	26	10	8	18	18
Shift	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
	12	0	24	19	11	24	20	19	15	20	6	12	11	19	12	7	19	3	1	11	11
Ciphertext	M	A	Y	T	L	Y	U	T	P	U	G	M	L	T	M	H	T	D	B	L	L

The message is encrypted as `MAYTLYUTPUGMLTMHTDBLL`.

The whole chosen text is encrypted as:

```
MAYTLYUTPUGMLTMHTDBLLTMAYT_HEXYGTLAHKYTMAYTLNGEB_AMTPUKFLTRH
NKTLDBGTUEETMAYTVYUNMRTMAUMTBLTVYYGTEHLMTVYZHKYTPUGMLTMHTZBG
XTNLTU_UBGTBTWUGGHMTZB_AMTRHNTUGRFHKYTBMTBLTRHNTBTUFTZB_AMBG
_TZHKTMAYTLYUTMAKHPLTKHWDTMH_YMAYKTVNMTMBFYTEYUOYLTNLTIHEBLA
YXTLMHGYLTPYTWUGGHMTZUEETUGRTZNKMAYKTBZTPYTWUGGHMTZYYETHKXBG
UKRTEHOYTUGXTPYTWUGGHMTKYUWATUGRTAB_AYKTBZTPYTWUGGHMTXYUETPB
MATHKXBGUKRTEHOYTVBKXLTZERTAB_ATBGTMAYTLNFFYKTLDRTUGXTKYLMTH
GTMAYTVKYYSYTMAYTLUFYTPBGXTPBEETMUDYTWUKYTHZTRHNTUGXTBTPYTPB
EETVNBEXTHNKTAHNLYTBGTMAYTMKYYLTRHNKTAYUKMTBLTHGTFRTLEYYOYTX
BXTRHNTINMTMAYKYTPBMATUTFU_BWTFUKDYKTZHKTRYUKLTBTPHNEXTVYEBY
OYTMAUMTMAYTPHKEXTWHNEXTGHMTPULATBMTUPUR
```

1.3.2 Permutation/Transposition cipher

The *permutation cipher* or *transposition cipher* is a kind of *poly-alphabetic* cipher: a cipher based on multiple substitution alphabets. For an integer m, called the period, we first divide the text into m columns. In other words, we create a matrix with m columns and whose first line is the first m characters of the text, the second line is characters $m+1$ until $2m$, etc. Notice that the last row of the matrix can be incomplete if the length of the text is not a multiple of m. In this case, following [27], random symbols are added at the end of the message to have a length which is a multiple of m. When having the message split into rows, a permutation π of $\{1, \ldots, m\}$ is applied to each row $a_1 a_2 \ldots a_m$ leading to $a_{\pi(1)} a_{\pi(2)} \cdots a_{\pi(m)}$. Thus, if we divide the message into blocks of m letters, the encrypted message is the message where the m columns have been shifted using the permutation π.

In the case of the alphabet Σ, the cryptosystem is $(\mathcal{P}, \mathcal{C}, \mathcal{K})$ where $\mathcal{P} = \mathcal{C} = (\mathbb{Z}_N)^m$ and $\mathcal{K}$ is the set of all the permutations of $\{1, \ldots, m\}$ and contains $m!$ elements. If π denotes a

[1] Using classical modular arithmetic.

permutation of the set $\{1, \ldots, m\}$, then for the key $k = \pi$,

$$e_k(a_1, \cdots, a_m) = (a_{\pi(1)}, \ldots, a_{\pi(m)}).\ \forall (a_1, \cdots, a_m) \in \mathcal{P}$$

and

$$d_k(b_1, \cdots, b_m) = (b_{\pi^{-1}(1)}, \cdots, b_{\pi^{-1}(m)})\ \forall (b_1, \cdots, b_m) \in \mathcal{C}.$$

Example 1.3.2.1. Suppose the message to be sent is `THE SEA WANTS TO KISS THE GOLD`.

The first step is to divide the message into blocks of period m. In this example, we consider a period length of 6, i.e., $m = 6$ and the length of our message is a multiple of 6. It could be not the case[2].

The message split into blocks is:

T	H	E	_	S	E
A	_	W	A	N	T
S	_	T	O	_	K
I	S	S	_	T	H
E	_	G	O	L	D

Arbitrarily choosing a permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 3 & 5 & 6 & 2 \end{pmatrix},$$

leads to

H	E	E	T	_	S
_	T	W	A	A	N
_	K	T	S	O	_
S	H	S	I	_	T
_	D	G	E	O	L

The columns are shifted according to the permutation π.

The encrypted text is thus `HEET_S_TWAAN_KTSO_SHSI_T_DGEOL` and the whole chosen text is encrypted as:

```
HEET_S_TWAAN_KTSO_SHSI_T_DGEOLNO_ESHEE_RTHSIU_NLHATG_WMOSR_Y
RI_USK__ANLLHEET_BUTTAY_ASTH_IB_E_ENOBSLT_F_OEREA_NWTSON_TFI
_AUDS_AIIGN_COA_NN_HFTIG__YTOUNRYAMO_IIET___YSOU_FAIM_GNHITI
__FGORHEET_S_OTAHRSC_WRO_ETKOGHBETR_TM_UTI_VLEEAS__EUSOHLPIS
DO_ESTEESN_WCOA_NN_LFTALAFN_Y_RRTUHEIEF__WCOA_NN_LFTEEONR_DI
ROYA_LED_VANWAE__CNRONT_AACEH_YG_NHIEFRH_IWAE__CNDONT_AILE_W
HD_TORN_AIRYOBVLE_RFDIS_YG_LHI_TIHN_EM_HSUEKRM_S__AYNDEOSRT_
__TNHEREEBEZTSH_E_MIEA_WDL_NWI_ETLAKC_A_REFU_OYOAIN_D_WIE__W
LI_LBUDR_LOUHEO_USIHN__T_ETERE_RYSOUHTE_ARINS__OMLY__SEDVEE_
DU_IYOPTU_T_EWRHE_T_HI_AA_GMICARRMKEFYO_R_AIRES_WDO_ULBEE_LI
EA_VTH__TTHEO_RWLDO_UCLDOATN_WH__SIT
```

[2]If the length of the message is not a multiple of the period, we should add random letters as previously mentioned.

1.3.3 Vigenère Cipher

The Vigenère cipher was originally invented by Giovan Battista Bellaso in 1553 book *La cifra del. Sig. Giovan Battista Bellaso*, but was attributed to Blaise de Vigenère (1523–1596) in the 19^{th} century.

In the Vigenère cipher, one has to choose a keyword (the key), a sequence of symbols from Σ, whose length m is called the *period of the cipher*. The cryptosystem is $(\mathcal{P}, \mathcal{C}, \mathcal{K})$ where $\mathcal{P} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}_N)^m$ and with the key $k = (k_1, \ldots, k_m)$, the encryption rule for a message $(a_1, \cdots, a_m) \in \mathcal{P}$ is

$$e_k(a_1, \ldots, a_m) = (a_1 + k_1, \ldots, a_m + k_m) \bmod \mathrm{N}$$

and the decryption rule for a message $(b_1, \cdots b_m) \in \mathcal{C}$ is

$$d_k(b_1 \ldots b_m) = (b_1 - k_1, \ldots, b_m - k_m) \bmod \mathrm{N}.$$

In order to encrypt a message longer than m characters, one has to divide the message into blocks of m letters. If the length of the message is $rm + s$ with $0 \leq s < m$, the message is divided into r blocks of m letters and a block of s letters. The encryption of the message $a_1 \cdots a_{rm+s}$ $(0 \leq s < m)$ is done block by block :

$\forall i \in \{0, \ldots, r-1\}$, the block $a_{im+1} \cdots a_{(i+1)m}$ becomes $(a_{im+1} + k_1) \cdots (a_{(i+1)m} + k_m)$ and the last block $a_1 \cdots a_s$ becomes $(a_1 + k_1) \cdots (a_s + k_s)$.

Similarly, the decryption of an encrypted message $b_1 \cdots b_{rm+s}$ $(0 \leq s < m)$ is done block by block :

$\forall i \in \{0, \ldots, r-1\}$, the block $(b_{im+1} \cdots b_{(i+1)m})$ becomes $(b_{im+1} - k_1) \cdots b_{(i+1)m} - k_m$ and the last block $b_1 \ldots b_s$ becomes $(b_1 - k_1) \cdots (b_s - k_s)$.

This cipher is a *poly-alphabetic* because a number of mono-alphabetic Caesar ciphers with different shifts are used sequentially and are then cyclically repeated.

An easy way to encrypt a message with the Vigenère cipher is to use the table, named *tabula recta* presented in Table 1.1, where the i^{th} line represents the i^{th} symbol shifted with different shifts starting with a shift of 0 and ending with a shift of 26.

Example 1.3.3.1. Suppose the alphabet $\{A, B, \ldots, Z, \_\}$ is used and the chosen keyword is "ORDINARY" (hence a period of 8) and that the message to be encrypted is `THE_SEA_WANTS_TO_KISS`.

Plaintext	T	H	E	$\overline{26}$	S	E	A	$\overline{26}$	W	A	N	T	S	$\overline{26}$	T	O	$\overline{26}$	K	I	S	S
	19	7	4	$\overline{26}$	18	4	0	$\overline{26}$	22	0	13	19	18	$\overline{26}$	19	14	$\overline{26}$	10	8	18	18
Keystream	O	R	D	I	N	A	R	Y	O	R	D	I	N	A	R	Y	O	R	D	I	N
	14	17	3	8	13	0	17	24	14	17	3	8	13	0	17	24	14	17	3	8	13
Ciphertext	G	Y	H	H	E	E	R	X	J	R	Q	A	E		J	L	N	A	L		E
	6	24	7	7	4	4	17	23	9	17	16	0	4	$\overline{26}$	9	11	13	0	11	$\overline{26}$	4

Thus, the Vigenère cipher with the keyword "ORDINARY" encrypts the message `THE_SEA_WANTS_TO_KISS` as `GYHHEERXJRQAE_JLNAL_E`.

The whole chosen text is encrypted, using the same keyword "ORDINARY", as:

```
GYHHEERXJRQAE_JLNAL_E_JESQJWYDVKNIKWDEQQVVC_GNBFUYWHIAHJFQAW
GRQPYZQHNLBXGYHHOERRGOCAUAJXWICJREDXZEVAMBVCBHHHIADQFQWWMFZK
RQX_MAXYWDCQMCRKAEWHSIXEGQAWG_RKLCRZR_ZQNZVHKOKXWQDUMFZDVJLV
T_WLEQWPR_IBOQWPDOMPNHRKX_JLUVWPRRQZHJCAVMVXZVDCRSQRFQSWYIIE
SUC_FODBFQZMMCRKAEWHSABINRQFMFKOGYHZMIWXJVCKNNDLGQIMRLQLEULV
NROXZEYMMADANMHHPADKBJCZRATENRQFMHZDVVUHVFQTSQFI_NEQNUHIY_MF
GYCWDDZKOHAHYOLBNSLZQSQCZOCPVGYXWDCAUEQPHCPMD_IHLQDVQ_HBFJCW
__JESQEZREPBNJKMMSRJSQZQ_DQTWBOHFAABNTDZR_ECNORBMADANZCDR_MF
ZBCJGIBANEXZMHERFVCQ__JESQWZREIXLEXZMHVYEJCQE_EKNCAHELVBIVCL
VDQVBKCXGTQQVVUMMWZQVQDHZAXFQQPIDKVONWRZMYVYEICQMWERZUCJRLZB
IVCAUAJXGYHHIOHIRQFWGLUXAEWHIAIENZWHNWRV
```

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

Table 1.1: Tabula recta

1.4 Enciphered machines during the Second World War

1.4.1 The Enigma machine

The Enigma machine was created by German engineer Arthur Scherbius. It was originally developed for commercial use but was modified for German military during WWII to encipher short tactical messages. It is an electromechanical machine similar to a typewriter (see Figure 1.2). It consists of a keyboard and a lightboard with twenty-six letters each (the space character is not taken into account in this machine), a battery, three rotors and in the most sophisticated versions of the machine, an additional plugboard. The plugboard is a kind of old telephone

switchboard. Basically, when a letter is pressed, another letter is lighted up on the lightboard. Further versions were developed, some of them had four rotors, named the German Naval M4. Unlike any other code machine created before, in Enigma, a same letter pressed twice is going to turn into two different letters, so a pair of letters never becomes a pair in the encrypted message; this is why it was thought unbreakable.

Figure 1.2: The Enigma machine with three rotors.

There are three rotors: a fast rotor, a middle rotor and a slow rotor. Inside them, there is a criss-cross of wires. The rotors are set in motion each time a letter is pressed. Each rotor has twenty-six starting positions, and the movements of the three rotors are interdependent: when the first rotor completes a rotation, the next rotor shifts one gear. When the second rotor completes a rotation as well, the third rotor shifts one gear too. Moreover, the order of the rotors can be changed. There is also a battery that is connected to the bulbs to light up the enciphered letters and because the wires are inside the rotors, each time the rotors turn, the battery connects to a different bulb and this moving part makes the resulting letter change.

There is also a reflector (see Figure 1.3), a disk with 26 contact points where each of them is wired to another, so the rotor makes 13 letter pairs. The reflector transforms the letter obtained by the third rotor to another one, in order that, if the initial settings are known, when typing the ciphertext, we will find the original message. Without it, if the letter obtained by the third rotor was directly resent to the three rotors, the result would be the letter that was originally pressed, i.e., no encryption. The reflector is fixed and unchanged.

This is the basic Enigma machine, called the commercial Enigma. However, for the army, a plugboard was added (see Figure 1.4). There are six wires, each of them connects two letters into a pair. The plugboard swaps over the two letters that are connected.

Figure 1.5 presents the way a letter is encrypted in the army Enigma.

The Enigma messages were usually written down in groups of 5 letters, the Navy used 4-letter groups.

To decode a message, an Enigma machine is needed but the exact settings, i.e., the initial positions of the rotors have to be known. These settings were based on a daily key which was changed at midnight every day during the war.

Figure 1.3: The reflector

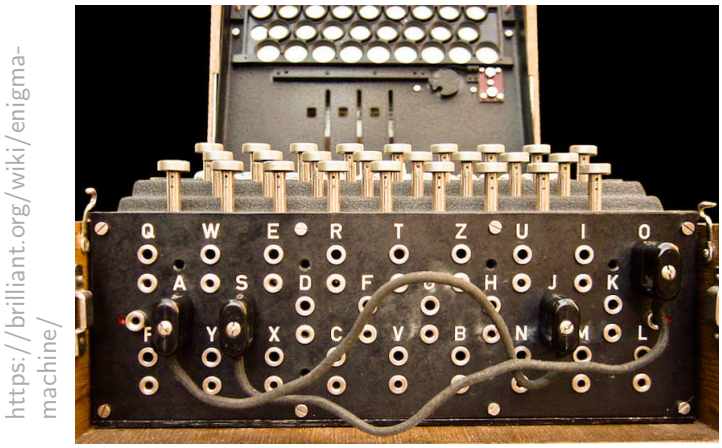


Figure 1.4: The plugboard.

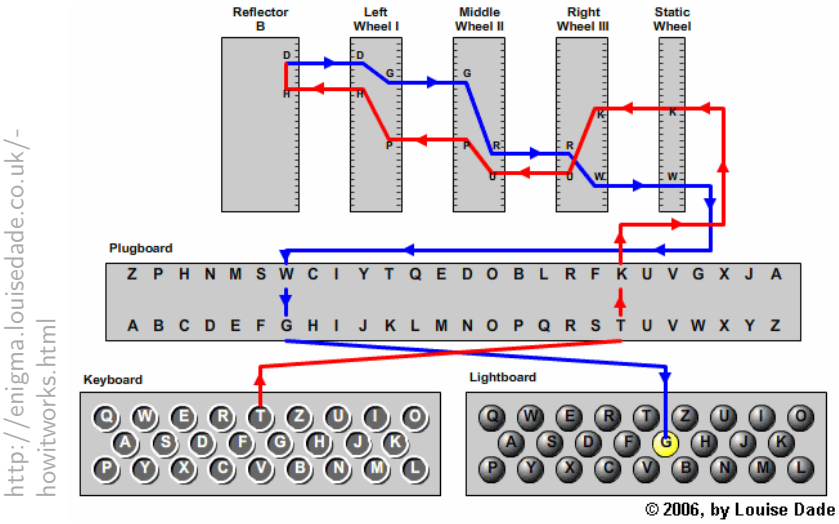


Figure 1.5: The way a letter is enciphered.

A first natural question is how many possible settings does the army Enigma contain? In other words, how many codes can it produce?
Considering the rotors only, there are $3 \times 2 \times 1$, i.e., 6 possible ways to place them in any order.

Each of them having twenty-six starting positions, there are thus 26^3, i.e., 17,576 initial starting positions. Furthermore, when the extra plugboard is added, 100,391,791,500 combinations can be made. Indeed, there are $\frac{26!}{14!6!2^6} = 100,391,791,500$ ways of choosing 6 pairs out of 26 elements. In total, there are $6 \times 17,576 \times 100,391,791,500 = 10,586,916,764,424,000$ ways to set the Enigma machine, i.e., more than ten quadrillion possibilities.

In 1938, the Germans added two more rotors to the three existing ones: one could choose three rotors out of five to set the machine and four wires were added to the plugboard in order to increase the pairs of letters that can be created. Thus, for this new Enigma machine, considering the rotors only, there are now $5 \times 4 \times 3$, i.e., 60 possible ways to place them and the combinations of the plugboard increases to $\frac{26!}{6!10!2^{10}} = 150,738,274,937,250$. It means that after 1938, the Enigma had 158,962,555,217,826,360,000 possible settings, more than 158 quintillion possibilities. So if one setting could be checked per minute for twenty-four hours every day and seven days every week, it would take[3] about 302 trillion of years to test all possible settings.

Example 1.4.1.1. Using an internet simulator[4] of the Enigma with 3 rotors to encrypt the message "The sea wants to kiss", i.e., without spaces, `THESEAWANTSTOKISS`, we obtain the encrypted message written in blocks of 4 letters: `OPCN RWQX VAJE PJZD U`.

1.4.2 The Lorenz machine

The Lorenz SZ ("Schlüselzusatz") cipher (see Figure 1.6), called sometimes the Tunny machine or Vernam cipher, was created in 1917 by an American engineer called Gilbert Vernam. It was used by Hitler himself and the top level of the Nazi party (the High Command and the German Army Field Marshals) in the second half of 1940 to send long messages (several thousands of characters) and it was even more complex than Enigma could be.

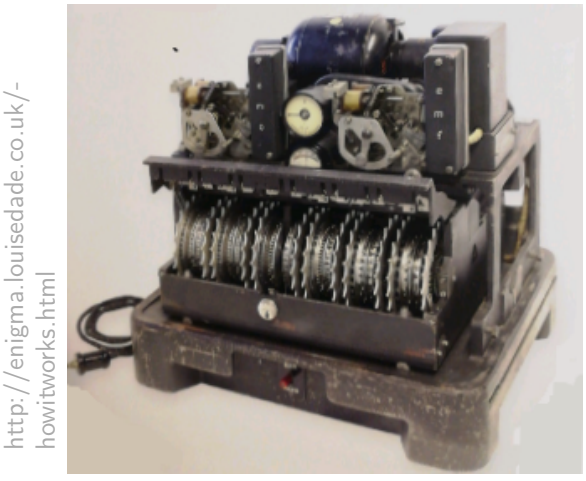


Figure 1.6: The Lorenz machine.

In this machine, the international teleprinter code (or Baudot code) (see Figure 1.7) is used: each letter is represented as a 5-bit code with electrical impulses, called *channels*, each of which

[3] $\frac{158,962,555,217,826,360,000}{60\text{x}24\text{x}365} = 302,440,173,549,897.95$

[4] http://enigma.louisedade.co.uk/enigma.html

can be either "on" (written as a cross) or "off"(written as a dot).

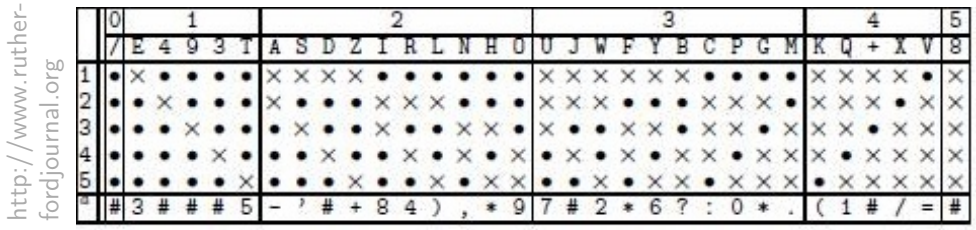


Figure 1.7: The teleprinter code.

In addition to the plaintext, extra letters are automatically generated and added, with a technique called the *modulo-2 addition*. Then, if the same extra letters are added to the encrypted message using the same technique, the original message can be read.

In other words, denoting by O the original message, by K the additional characters and by E the enciphered text, we have

$$O + K = E$$

and

$$E + K = O.$$

Let's take an easy example of a single letter to be sent.

Example 1.4.2.1. For simplicity, consider the message consisting of the single letter C, which is $[\cdot \times \times \times \cdot]$ using the international teleprinter code. Suppose a single letter, the key, is generated and added to the message. If this letter is M, which is $[\cdot \cdot \times \times \times]$ using the international teleprinter code, the encrypted message is obtained by adding together the original message C and the key M using the modulo-2 addition based on the Boolean "exclusive or" (XOR) function: if two symbols are the same, the resulting symbol is a dot, otherwise, it is a cross.

The original letter C	$\cdot$	$\times$	$\times$	$\times$	$\cdot$
The key letter M	$\cdot$	$\cdot$	$\times$	$\times$	$\times$
The coded letter	$\cdot$	$\times$	$\cdot$	$\cdot$	$\times$

The code is thus the letter L.

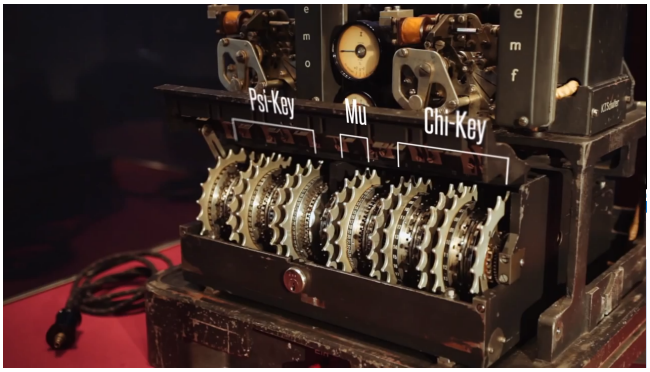
Furthermore, we have

The coded letter L	$\cdot$	$\times$	$\cdot$	$\cdot$	$\times$
The key M	$\cdot$	$\cdot$	$\times$	$\times$	$\times$
The original letter C	$\cdot$	$\times$	$\times$	$\times$	$\cdot$

As expected, the original message is found[5].

The Lorenz machine consists of twelve wheels (see Figure 1.8). The two wheels in the middle, the *motor* or Mu-wheels, dictate the movement of the five wheels on the left, the Psi-wheels. They were designed to introduce an apparent randomness into the key. There are also five wheels on the right, the Chi-wheels, that follow a regular movement pattern and are set in motion after each letter has been enciphered.

[5]More examples of this addition of letters can be found in Appendix B.



[15]

Figure 1.8: The wheels of the Lorenz machine and the keys they produce.

On the outside of the wheels there are pins, called also "cams", that can be set to "on" or "off" in order to generate a pulse or not. Altogether, there are 501 pins. They can be seen easily in Figure 1.9.

https://www.3ders.org/articles/20170306-3d-printing-aids-reconstruction-of-lorenz-cipher-machine-used-by-germans-during-wwii.html

Figure 1.9: The pins of the Lorenz machine either to "on" or "off".

The table below shows the number of pins for each wheel.

Wheel name	A	B	C	D	E	F	G	H	I	J	K	L
	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5	μ_{37}	μ_{61}	χ_1	χ_2	χ_3	χ_4	χ_5
Number of pins	43	47	51	53	59	37	61	41	31	29	26	23

Actually, the machine applies two keys to the message, named the Chi-key (produced by the Chi-wheels) and the Psi-key (produced by the Psi-wheels), so it enciphers the message twice.

The idea of the machine was to used a paper tape of the key, a technique called the one-time pad system, and if the extra characters were chosen randomly, it would be unbreakable. However, they are generated by a machine; they are thus *pseudo-random*. This choice made the Lorenz machine actually breakable.

The same natural question as for the Enigma machine is asked: how many possible settings does the Lorenz machine contain?

Considering the pins only, there are

$$\begin{aligned} 2^{501} &= 6,546,781,215,792,283,740,026,379,393,655,198,304,433,\\ &\quad 284,092,086,129,578,966,582,736,192,267,592,809,349,\\ &\quad 109,766,540,184,651,808,314,301,773,368,255,120,142,\\ &\quad 018,434,513,091,770,786,106,657,055,178,752\\ &\simeq 6.5 \times 10^{150} \end{aligned}$$

ways to set them.

Moreover, the number of starting positions the twelve wheels have is the product of the number of pins they contain, i.e.,

$$\begin{aligned} 43 \times 47 \times 51 \times 53 \times 59 \times 37 \times 61 \times 41 \times 31 \times 29 \times 26 \times 23 &= 16,033,955,073,056,318,658\\ &\simeq 1.6 \times 10^{19}, \end{aligned}$$

which is roughly 16 quintillion possible combinations.

In total, there are $2^{501} \times 16,033,955,073,056,318,658$, i.e.,

$$\begin{aligned} &104,970,795,887,142,501,519,944,408,859,713,937,438,238,568,341,\\ &584,154,526,205,632,598,745,732,639,647,278,021,173,163,831,071,\\ &764,896,225,159,592,365,198,842,461,226,688,733,330,753,486,243,\\ &770,471,723,522,422,795,262,754,816\\ \simeq &10^{170} \end{aligned}$$

possible settings.

Comparing to Enigma, there are 10^{149} times more possible settings in the Lorenz machine.

Example 1.4.2.2. Using an simulator[6] of the Lorenz machine to encrypt the message "The sea wants to kiss", i.e., without spaces, `THE_SEA_WANTS_TO_KISS`, we obtain the encrypted message: `ZTXEFCDCQL3FJVUG9MTKE`.

[6] https://lorenz.virtualcolossus.co.uk/LorenzSZ/

Chapter 2

Decrypting a message

This chapter will present some techniques to decrypt the ciphers introduced in Chapter 1, keeping all the previous notations. It will be based on Palash Sarkar's article (see [29]), on Johansson's lectures notes (see [18]), on Practical Cryptography's references (see [21] to [23]) and on James Grime's references: [13] to [15]. All the Figures in this chapter will be obtained using the R code in Appendix A. Furthermore, all the letters will be replaced by their uppercase equivalents and the non alphabetic characters will be converted to spaces.

2.1 Chi-squared test

First, a quick reminder of the Chi-squared test will be made before applying it to the cryptography context. The following will be based on Guillaume Mijoule's lectures notes (see [25]), even if it will be adapted for the discrete context.

The Chi-squared test is a test that allows to decide whether an independent and identically distributed sample $X_1, X_2, \ldots, X_n$ taking values in $\{a_1, a_2, \ldots a_k\}$ (with k being an integer or infinity) follows a given discrete distribution F. The two hypotheses of interest are:

$$H_0 : X_1, X_2, \ldots, X_n \sim F$$

$$H_1 : X_1, X_2, \ldots, X_n \nsim F$$

Let p_j denote $\mathbb{P}_F[X_1 = a_j]$ and let

$$O_j = \sum_{i=1}^{n} \mathbf{1}_{\{X_i = a_j\}}$$

the random variable corresponding to observed number of X_i equal to a_j.

The idea is to look at the quantity

$$T = \sum_{j=1}^{k} \frac{(O_j - np_j)^2}{np_j}$$

where np_j represents the expected number of random variables taking the value a_j out of n repetitions. If the sample distribution actually follows the distribution F, then the Chi-squared

statistic will be close to 0. The higher the statistic is, the less it is plausible that the distribution of the X_i follows the given distribution F. Notice that, when considering observed data, we obtain an observed value T_{obs} of the test statistic T.

More formally, the Chi-squared test is presented below:

Theorem 2.1.0.1 (Chi-squared Pearson test)**.** *With the notations introduced before, we have that under the null hypothesis H_0, the test statistic*

$$T = \sum_{j=1}^{k} \frac{(O_j - np_j)^2}{np_j},$$

asymptotically follows a Chi-squared distribution with $k-1$ degrees of freedom.

The rejection region associated to the test at a level of significance α ($\alpha \in]0,1[$) is

$$]q_{k-1,1-\alpha}, +\infty[$$

where $q_{k-1,1-\alpha}$ is the $(1-\alpha)$-quantile from a Chi-squared distribution with $k-1$ degrees of freedom.

For our context, the statistic is

$$T = \sum_{i=0}^{26} \frac{(C_i - E_i)^2}{E_i}$$

where C_i is the number of occurrences of symbol i in a given text, using symbols and numbers interchangeably, and E_i is the expected count of symbol i in English plaintext.

The Chi-squared approximation relies on the assumption that the counts are approximately normally distributed. It is generally accepted by the scientific community that when all the expected counts are greater than 5 the Chi-squared approximation tends to be reasonable.

2.2 Substitution cipher

In substitution cipher, a permutation π of $\{0, 1, \ldots, 26\}$ is used and the overall frequencies of letters do not change. In other words, if the letter E is the most used letter in the original message, $\pi(E)$ will be the letter with the highest frequency in the encrypted message. That is the key point of the *frequency analysis* first developed by Al-Kindi, an Arab mathematician and philosopher. It is a method that studies the statistical regularities of language, such as the frequencies of letters in the ciphertext, to discover the permutation π that has been used. For instance, in English, it is well known that E is the most frequent letter, followed by T and then A.

2.2.1 Step by step frequency analysis

The frequency analysis allows to know the permutation for the most frequently used letters; for the remaining letters, it won't be very useful because the distribution of the letters becomes more uniform for the less used letters and it will require a very long message to see the small differences. After considering single letters, frequency analysis of bigrams or trigrams (more

generally n-grams[1]) can be used to attack the message. At the end, the remaining letters can usually be guessed.

In order to do this analysis, an arbitrary English text is chosen: Jane Austen's novel *Pride and Prejudice*[2] which is a 658,549-character English text written in 1813. Later, we will compare the frequencies of letters of different texts.

Table 2.1 shows the frequencies obtained for each letter, Figure 2.1 is a barplot of frequencies of letters and Figure 2.2 is a letters' cloud. As expected for an English text, the space character is the most common symbol followed by the letter E and then the letter T. Moreover, the less used letters are Q, X, Z and J.

Furthermore, Figures 2.3, 2.4, 2.5 and 2.6 present by means of partial[3] words' clouds the most common bigrams and trigrams in English (based on Jane Austen's *Pride and Prejudice*). While Figures 2.3 and 2.4 consider bigrams and trigrams formed by two letters (excluding the space character), Figures 2.5 and 2.6 allow bigrams and trigrams formed by letters and the space character. From these figures, as expected, it can be seen that THE is the most common trigrams, thus finding the bigrams TH and HE as the most common bigrams is not surprising. It was also expected that the word AND will appear as a common trigram. More surprisingly, the trigram HER is also quite common but HIM for example is not, but as we will see later, it appears quite often in other English texts as well.

In the following, spaces will be taken into account in n-grams.

A	0.0634	F	0.0184	K	0.0047	P	0.0125	U	0.0225	Z	0.0013
B	0.0136	G	0.0153	L	0.0331	Q	0.0010	V	0.0088	_	0.1867
C	0.0212	H	0.0507	M	0.0226	R	0.0485	W	0.0180		
D	0.0338	I	0.0582	N	0.0581	S	0.0504	X	0.0012		
E	0.1043	J	0.0014	O	0.0602	T	0.0703	Y	0.0197		

Table 2.1: Frequencies of letters in Austen's text

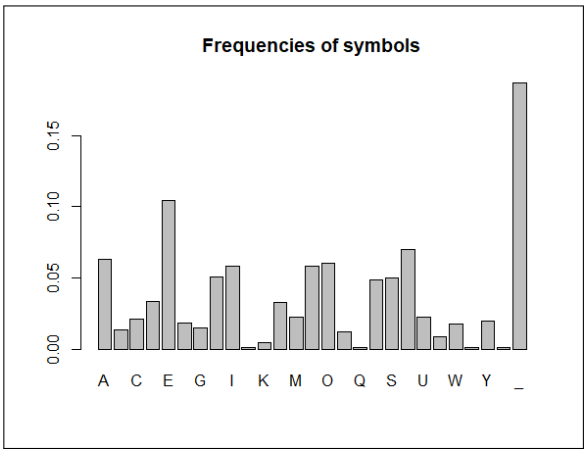


Figure 2.1: Frequencies of the different letters in Austen's text

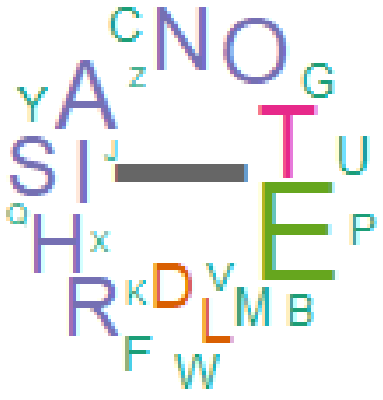


Figure 2.2: Letters' cloud in Austen's text

[1] A *n-gram* is a sequence of n consecutive letters.
[2] This novel can be found in the project Gutenberg (gutenberg.org).
[3] "Partial" because not all the bigrams or trigrams can be seen in the figure.

Figure 2.3: Partial bigrams' cloud in Austen's text

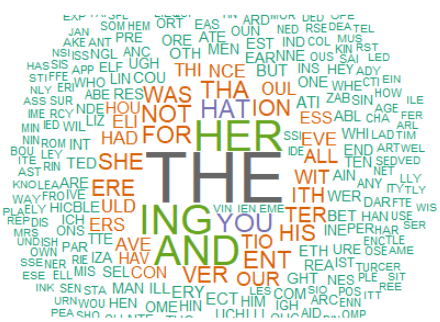


Figure 2.4: Partial trigrams' cloud in Austen's text

Figure 2.5: Partial bigrams' cloud in Austen's text

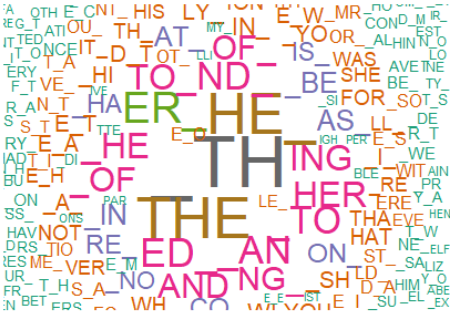


Figure 2.6: Partial trigrams' cloud in Austen's text

Example 2.2.1.1. Suppose the ciphertext is the chosen text encrypted using a Caesar shift cipher with shift 20 (see Example 1.3.1.2).

```
MAYTLYUTPUGMLTMHTDBLLTMAYT_HEXYGTLAHKYTMAYTLNGEB_AMTPUKFLTRH
NKTLDBGTUEETMAYTVYUNMRTMAUMTBLTVYYGTEHLMTVYZHKYTPUGMLTMHTZBG
XTNLTU_UBGTBTWUGGHMTZB_AMTRHNTUGRFHKYTBMTBLTRHNTBTUFTZB_AMBG
_TZHKTMAYTLYUTMAKHPLTKHWDTMH_YMAYKTVNMTMBFYTEYUOYLTNLTIHEBLA
YXTLMHGYLTPYTWUGGHMTZUEETUGRTZNKMAYKTBZTPYTWUGGHMTZYYETHKXBG
UKRTEHOYTUGXTPYTWUGGHMTKYUWATUGRTAB_AYKTBZTPYTWUGGHMTXYUETPB
MATHKXBGUKRTEHOYTVBKXLTZERTAB_ATBGTMAYTLNFFYKTLDRTUGXTKYLMTH
GTMAYTVKYYSYTMAYTLUFYTPBGXTPBEETMUDYTWUKYTHZTRHNTUGXTBTPYTPB
EETVNBEXTHNKTAHNLYTBGTMAYTMKYYLTRHNKTAYUKMTBLTHGTFRTLEYYOYTX
BXTRHNTINMTMAYKYTPBMATUTFU_BWTFUKDYKTZHKTRYUKLTBTPHNEXTVYEBY
OYTMAUMTMAYTPHKEXTWHNEXTGHMTPULATBMTUPUR
```

However, the shift is usually unknown and the goal is to find it. Even if a brute force can be applied (testing all the 27 possibles keys), the purpose is to develop a more efficient method. Thus, the first step is to look at the numbers of occurrences of single letters in the ciphertext. Such results are summarised in two ways: Figure 2.7 is a barplot of frequencies of letters and 2.8 is a letters' cloud.

Such data suggest that the space symbol is encrypted by "T". Indeed, the most common symbol in the ciphertext is T. Thus, the encryption rule should be $e_k(26) = 19$.

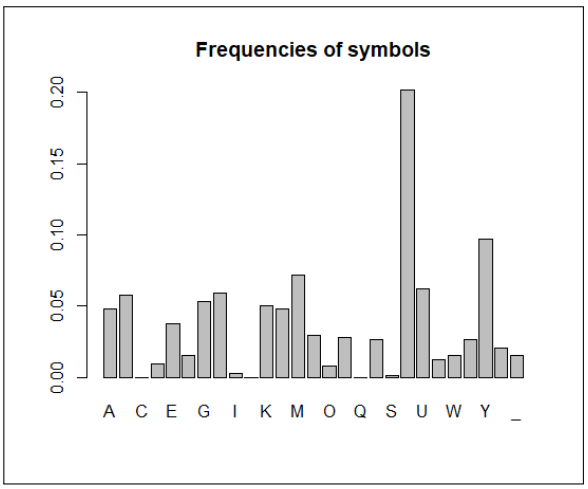


Figure 2.7: Frequencies of the different letters in the ciphertext

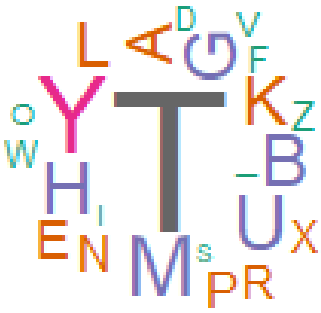


Figure 2.8: Letters' cloud of the ciphertext

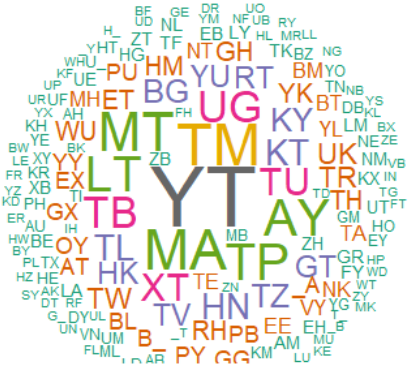


Figure 2.9: Partial bigrams' cloud of the ciphertext

Figure 2.10: Partial trigrams' cloud of the ciphertext

This guess is confirmed when looking at the bigrams (see Figure 2.9). Indeed, the bigram YT is the most common leading us to guess that E should be encrypted as Y because $\_E$ is the most common bigram in English. Thus we should have $e_k(4) = 24$.

Once again, our guess is confirmed because the most common trigram in the ciphertext is TMA (see 2.10) while it is $\_TH$ in English plaintext, i.e., $e_k(7) = 0$.

Knowing that, we can easily derived the Caesar shift used. Using the decryption rule, $d_k(j) = j - k \pmod{27}$ $\forall j \in \mathcal{C}$, we find that k should be 20 if we supposed that $e_k(26) = 19, e_k(4) = 24$ and $e_k(7) = 0$.

Frequency analysis of different English texts

In the previous section, we used a reference text to find the frequencies of letters in English. A natural question is "Are the frequencies of letters roughly the same in different English texts?". The answer seems to be yes. Indeed, three English texts were compared: Jane Austen's *Pride and Prejudice* (1813), the *English Constitution* by Walter Bagehot (1865) and Shakespeare's famous *Romeo and Juliet* (1597). The comparison of bigrams in different texts will be done in Chapter 4 through a matrix called *matrix of bigrams* and it is therefore not done here.

Figures 2.11, 2.12 and 2.13 present the frequency analysis of letters for each text.

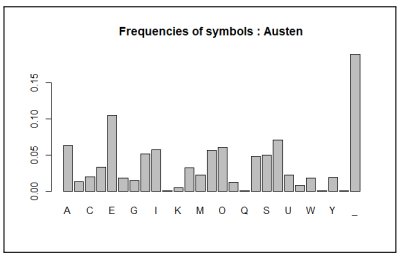


Figure 2.11: Frequencies of letters in Austen's text

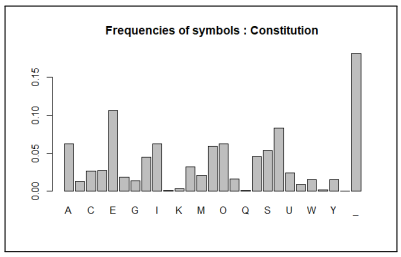


Figure 2.12: Frequencies of letters in the English Constitution

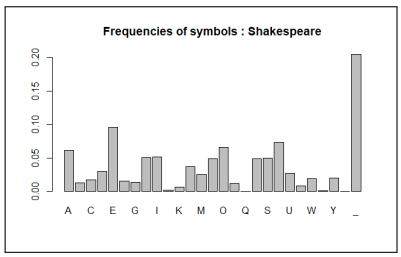


Figure 2.13: Frequencies of letters in Shakespeare's text

As we can notice, the character frequencies are roughly the same between the three different texts. For consistency, we use a Chi-squared test to test whether or not the frequencies of letters in the English Constitution and in Shakespeare's text are the same as in Austen's test. We find a p-value of 1 for both tests which strongly doesn't reject the hypotheses of the same distribution of frequencies.

Furthermore, Figures 2.14, 2.15 and 2.16 present the frequency analysis of trigrams by means of words' clouds.

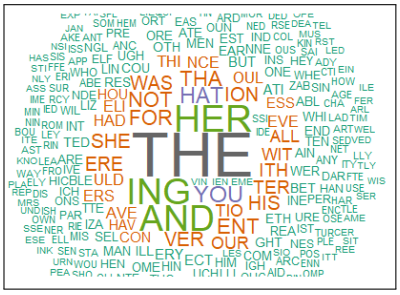


Figure 2.14: Partial trigrams' cloud in Austen's text

Figure 2.15: Partial trigrams' cloud in the English Constitution

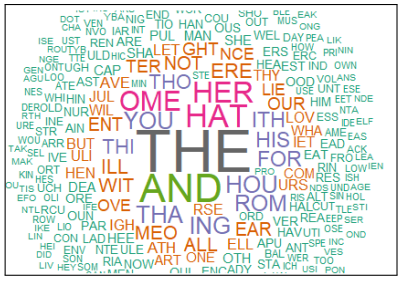


Figure 2.16: Partial trigrams' cloud in Shakespeare's text

It can be seen that the trigram THE is the most common trigram for all of the three texts and the trigrams AND, HER and ING are also quite common, even if their frequencies depend on the text.

Frequency analysis of different languages

In order to see if the language has an impact on frequency analysis, we compare three texts from different languages: Jane Austen's *Pride and Prejudice* (1813) (English text), Victor Hugo's *Les Misérables* (1862) (French text) and Francesco Domenico's *Amelia Calani* (1862) (Italian text).

To do so, all the special characters such as "é", "è", "ô", etc are replaced by their uppercase equivalents "E", "O", etc.

Figures 2.17, 2.18 and 2.19 present the frequency analysis for each text.

Regardless of the language, the space character is the most frequent symbol followed by the letter E. However, their frequencies depend on the language. As we could expect, the frequency analysis is based on a language and is useless if the ciphertext is from another language. For

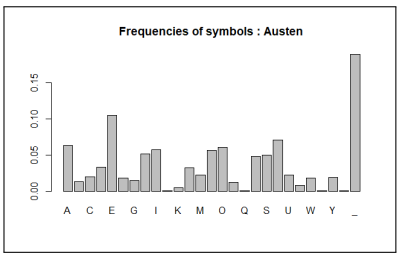


Figure 2.17: Frequencies of letters in Austen's text

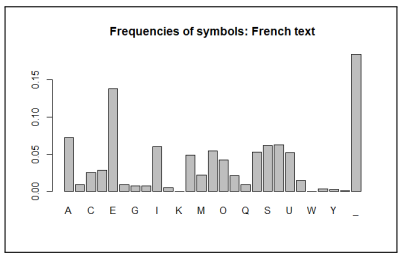


Figure 2.18: Frequencies of Hugo's text

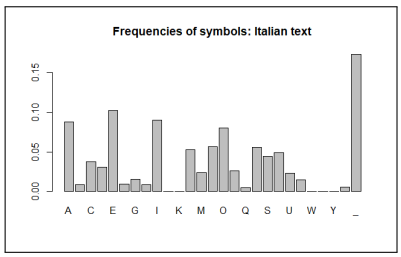


Figure 2.19: Frequencies of letters of Domenico's text

consistency, we use a Chi-squared test to test whether or not the frequencies of letters in Hugo's text and in Domenico's text are the same as in Austen's test.

We find p-values which are less than 0.001, which strongly rejects the hypothesis of the same distribution of frequencies.

Furthermore, Figures 2.20, 2.21 and 2.22 present the frequency analysis of trigrams by means of words' clouds.

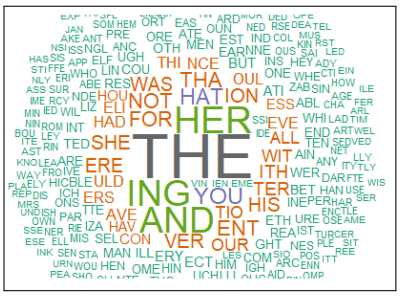


Figure 2.20: Partial trigrams' cloud in Austen's text

Figure 2.21: Partial trigrams' cloud in Hugo's text

Figure 2.22: Partial trigrams' cloud in Domenico's text

As naturally expected the most common trigrams in different languages are different.

2.2.2 Statistical method for frequency analysis

The technique presented in the previous section (see Example 2.2.1.1) requires human contribution to check the plausibility of a guess. The following technique, suggested in [21], does not ask for such contribution; instead it uses statistical method to find the key.

The idea is to decipher the text with each 27 possible keys and to compute the Chi-squared statistic and to choose the key whose corresponding Chi-squared statistic is the smallest.

The R code can be found in Appendix A.

Example 2.2.2.1. The same encrypted text as Example 2.2.1.1 is used.

Table 2.2 shows the plaintext and the Chi-squared statistics obtained when using the 27 possible keys. Furthermore, Figure 2.23 summarized the Chi-squared statistics obtained when using the 27 possible keys with a plot.

Decryption key	Plaintext	Chi-squared statistic
0	MAYTLYUTPUGMLTMHTDBL ···	1659.84
1	L_XSKXTSOTFLKSLGSCAK ···	7275.30
2	KZWRJWSRNSEKJRKFRB_J ···	3719.27
3	JYVQIVRQMRDJIQJEQAZI ···	32153.75
4	IXUPHUQPLQCIHPIDP_YH ···	6529.40
5	HWTOGTPOKPBHGOHCOZXG ···	2695.03
6	GVSNFSONJOAGFNGBNYWF ···	2016.04
7	FURMERNMIN_FEMFAMXVE ···	2133.72
8	ETQLDQMLHMZEDLE_LWUD ···	9322.22
9	DSPKCPLKGLYDCKDZKVTC ···	7831.29
10	CROJBOKJFKXCBJCYJUSB ···	21927.43
11	BQNIANJIEJWBAIBXITRA ···	5773.23
12	APMH_MIHDIVA_HAWHSQ_ ···	4314.59
13	_OLGZLHGCHU_ZG_VGRPZ ···	3295.65
14	ZNKFYKGFBGTZYFZUFQOY ···	6554.41
15	YMJEXJFEAFSYXEYTEPNX ···	7366.82
16	XLIDWIED_ERXWDXSDOMW ···	3943.84
17	WKHCVHDCZDQWVCWRCNLV ···	4578.45
18	VJGBUGCBYCPVUBVQBMKU ···	6490.48
19	UIFATFBAXBOUTAUPALJT ···	3210.66
20	THE_SEA_WANTS_TO_KIS ···	**20.79**
21	SGDZRD_ZV_MSRZSNZJHR ···	20683.90
22	RFCYQCZYUZLRQYRMYIGQ ···	5754.09
23	QEBXPBYXTYKQPXQLXHFP ···	249470.73
24	PDAWOAXWSXJPOWPKWGEO ···	5388.88
25	OCZNZVRVIONOJFDNNOCZ ···	5148.09
26	NBZUMZVUQVHNMUNIUECM ···	6281.78

Table 2.2: Chi-squared statistic using different keys

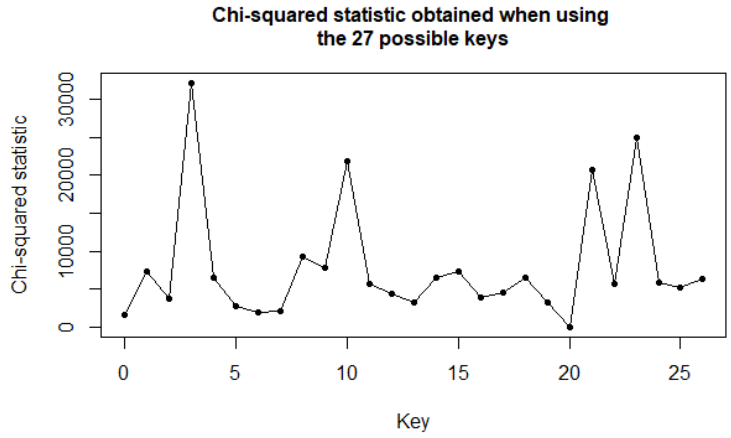


Figure 2.23: Chi-squared statistic using different keys

As we can see, the key used should be 20 because the corresponding Chi-squared statistic is the lowest and so small compared to others, which means that the distribution of frequencies of letters using the decryption key 20 follows the best the distribution of frequencies of letters in English.

The approach presented here simply uses the *Chi-squared distance* as a measure of goodness of fit between the distribution of an i.i.d sample $X_1, X_2, \ldots, X_n$ and the target distribution F. However, we can go one step further and use the Chi-squared test presented in section 2.1.

In order to have a reasonable Chi-squared approximation, all the expected counts should be greater than 5. However, with 27 categories, these are the expected counts on a 640-character text:

A	B	C	D	E	F	G	H	I	J	K	L	M
40.35	8.80	13.04	21.59	67.15	11.62	9.71	32.96	36.63	0.85	3.11	20.89	14.29

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
36.48	38.76	7.97	0.61	31.27	32.05	45.14	14.51	5.54	11.90	0.81	12.31	0.91

_
120.77

In order to use the Chi-squared test, we decide to decrease the number of categories by putting together the less used letters J, K, Q,X and Z in a new category named "Less used letters", noted "LUL", leading to a new alphabet of 23 elements. These are the new expected counts:

A	B	C	D	E	F	G	H	I	L	M	N
40.35	8.80	13.04	21.59	67.15	11.62	9.71	32.96	36.63	20.89	14.29	36.48

O	P	R	S	T	U	V	W	Y	_	"LUL"
38.76	7.97	31.27	32.05	45.14	14.51	5.54	11.90	12.31	120.77	6.28

The results of the Chi-squared test applied for each of the possible keys are presented in Table 2.3.

As we can see, the null hypothesis that the letters frequencies of the obtained plaintexts are distributed as in English is rejected at a level of significance of 5% for all keys except the 20^{th} where it is, clearly, not rejected. For consistency, the plot of the Chi-squared statistics for the 27 different keys obtained using the alphabet of 23 symbols[4] is represented in Figure 2.24. Furthermore, the Chi-squared distribution with 22 degrees of freedom is represented in Figure 2.25 and Figure 2.26 is a zoom of Figure 2.24 where the value of the 0.95-quantile of a Chi-squared distribution with 22 degrees of freedom is shown. We do not reject the null hypothesis if the statistic obtained is less than this quantile.

For general substitution cipher, a frequency analysis can be done as well. However, since the encryption function is different for all letters, this process would lead to an incomplete decryption. Nevertheless, Chapter 4 will give a method to handle this issue using Markov Chain Monte Carlo theory.

2.3 Permutation/Transposition cipher

The first step to decrypt an encrypted message using the permutation cipher is to find its period m, and then rewrite the message in form of a table with m columns. The second is to find the key, i.e., the permutation of $\{1, \ldots, m\}$ that was used. The first subsection will assume that the period is known and the second will show how to actually find the period.

[4]The alphabet obtained when the less used letters are putting together.

Decryption key	Plaintext	Chi-squared statistic	p-value
0	MAYTLYUTPUGMLTMHTDBL ···	1399.90	< 0.001
1	L_XSKXTSOTFLKSLGSCAK ···	4111.10	< 0.001
2	KZWRJWSRNSEKJRKFRB_J ···	3002.04	< 0.001
3	JYVQIVRQMRDJIQJEQAZI ···	9708.36	< 0.001
4	IXUPHUQPLQCIHPIDP_YH ···	3709.10	< 0.001
5	HWTOGTPOKPBHGOHCOZXG ···	1600.89	< 0.001
6	GVSNFSONJOAGFNGBNYWF ···	1262.57	< 0.001
7	FURMERNMIN_FEMFAMXVE ···	1807.50	< 0.001
8	ETQLDQMLHMZEDLE_LWUD ···	3687.10	< 0.001
9	DSPKCPLKGLYDCKDZKVTC ···	6292.18	< 0.001
10	CROJBOKJFKXCBJCYJUSB ···	7515.67	< 0.001
11	BQNIANJIEJWBAIBXITRA ···	2677.51	< 0.001
12	APMH_MIHDIVA_HAWHSQ_ ···	2074.47	< 0.001
13	_OLGZLHGCHU_ZG_VGRPZ ···	2558.67	< 0.001
14	ZNKFYKGFBGTZYFZUFQOY ···	5549.16	< 0.001
15	YMJEXJFEAFSYXEYTEPNX ···	3772.49	< 0.001
16	XLIDWIED_ERXWDXSDOMW ···	2035.34	< 0.001
17	WKHCVHDCZDQWVCWRCNLV ···	3439.73	< 0.001
18	VJGBUGCBYCPVUBVQBMKU ···	4467.70	< 0.001
19	UIFATFBAXBOUTAUPALJT ···	1737.81	< 0.001
20	THE_SEA_WANTS_TO_KIS ···	15.92	**0.82**
21	SGDZRD_ZV_MSRZSNZJHR ···	6918.00	< 0.001
22	RFCYQCZYUZLRQYRMYIGQ ···	3477.07	< 0.001
23	QEBXPBYXTYKQPXQLXHFP ···	8595.23	< 0.001
24	PDAWOAXWSXJPOWPKWGEO ···	4491.92	< 0.001
25	OCZNZVRVIONOJFDNNOCZ ···	3781.40	< 0.001
26	NBZUMZVUQVHNMUNIUECM ···	2721.99	< 0.001

Table 2.3: Chi-squared statistic using different keys

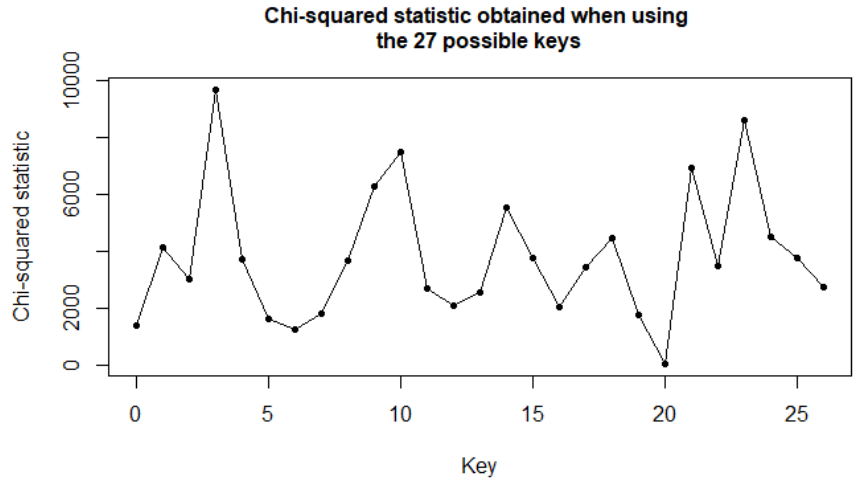


Figure 2.24: Chi-squared statistic using different keys and bringing together the less used letters

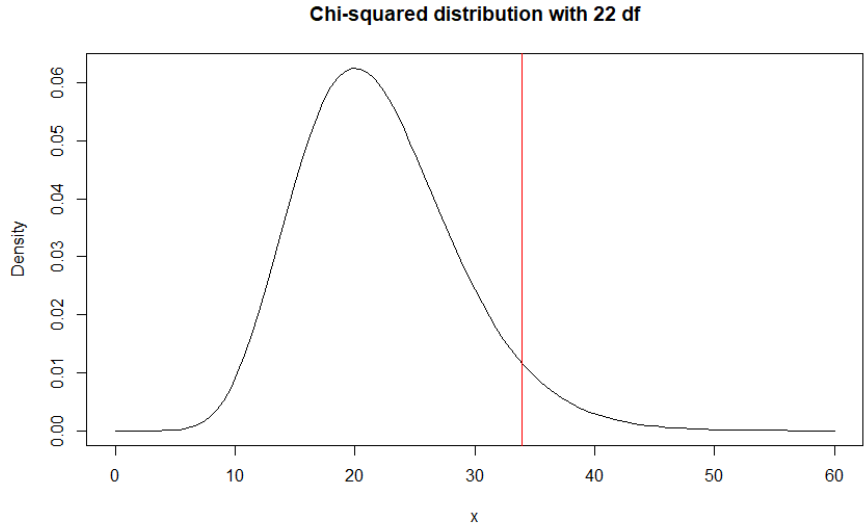


Figure 2.25: Chi-squared distribution with 22 degrees of freedom

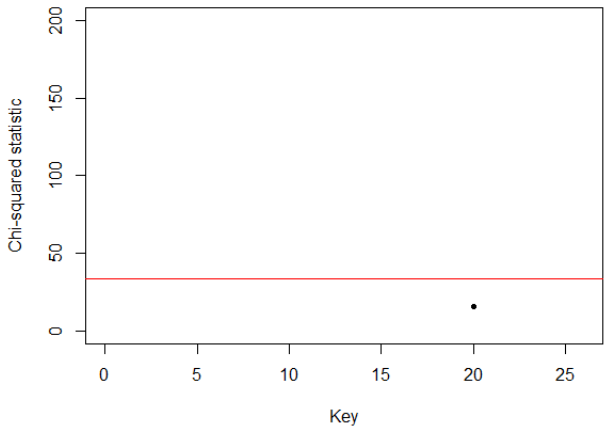


Figure 2.26: Zoom of Figure 2.24

2.3.1 Known period

Frequency analysis of bigrams or trigrams

In the case of a permutation cipher, when arranging the message into m columns, the original message can be recovered by finding the permutation π that was applied. Recall that the letters in a particular row of size m are shifted to encrypt the message. It means that if the bigram "TH" was present in the original message, the letter "T" and the letter "H" will be present as well in the encrypted row but maybe not just one after another. By analysing the distribution of bigrams or trigrams, we can find what the permutation π was. If our assumptions lead to very unlikely pairs like "ZZ", maybe they are not correct and should be corrected.

Example 2.3.1.1. Suppose that the encrypted message is `HEET_S_TWAAN_KTSO_SHSI_T_DGEOL` and that the length m is known to be 6.

So the first step is to split the text into blocks of size 6.

H	E	E	T	_	S
_	T	W	A	A	N
_	K	T	S	O	_
S	H	S	I	_	T
_	D	G	E	O	L

In an encrypted message using permutation cipher, the columns are just mixed. A first way to find the permutation used is to look at the 6!=720 possible permutations. This raw method can be done for short permutations. Otherwise, we need to attack the problem in a different way, looking at the plausible column shifts.

Looking at the first row, because "TH" is the most common bigram (without taking the space symbol into account) in English, a guess is that the inverse of 4 (under π) is immediately followed by the inverse of 1 (under π), i.e., the fourth column must be followed by the first column in the decrypted text. Furthermore, because "THE" is the most common trigram the inverse of 1 under π is either followed by the inverse of 2 (under π) or by the inverse of 3 (under π). Notice that if we use space-bigram, we know that "E_" is the most common, thus the inverse of 5 (under π) follows the inverse of 2 (under π) or the inverse of 3 (under π). After such considerations, a raw force can be applied. However, doing this asks to try a huge amount of possible permutations and it is practically unfeasible by hand. Nevertheless, Chapter 4 will present a method using Markov Chain Monte Carlo to find the permutation. Therefore, we leave the decryption of such cipher for this chapter.

2.3.2 Unknown period

Finding the period of a ciphertext encrypted with a permutation cipher is quite difficult because there is no change in the frequencies of letters. Indeed, the letters are just moved from a place to another in the message but are not encrypted with another letter. Actually, looking at bigrams can be useful. This will be done in Chapter 4 using Monte Carlo Markov Chain.

2.4 The Vigenère cipher

The first step to decrypt an encrypted message using the Vigenère cipher is to find its period m, and then rewrite the message in form of a table with m columns. The second is to find the key that was used. The first subsection will assume that the period is known and the second will show how to actually find the period. Although the frequency analysis of letters or bigrams or even trigrams will be useless because the Vigenère cipher is a poly-alphabetic cipher, we will use the frequencies of letters in each column. Furthermore, if the period is unknown, three methods will be presented to guess what it could be. The first one will be based on frequencies of letters in each column and the two others will be based on two statistical methods: Kasiski's method which searches repetitive n-grams and Friedman's index of coincidence which is based on the probability that two random letters in the text are the same.

2.4.1 Known period

Combination of frequency analysis and an algebraic approach

If the period is known to be m, we first divide the ciphertext into m columns. In other words, we create a matrix with m columns whose first line is the first m characters of the text, the second line is characters $m+1$ until $2m$, etc. Notice that the last row of the matrix can be incomplete but it is not a problem for the use we will make of this matrix. More formally, if the the ciphertext is $b_1 \dots b_N$ (hence a length n) then, we write the ciphertext in rows of length m, and we get the following array[5]:

$$\begin{array}{cccc} b_1 & b_2 & \cdots & b_m \\ b_{m+1} & b_{m+2} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{N-1} & b_n. & & \end{array}$$

Then, we look at the frequencies of letters within a column. This should help to find the key used for each of the m columns[6]. As we can expect, the more characters there are within a column, the better this method will be. Chapter 5 will compare the efficiency of this method based on different lengths of text.

Example 2.4.1.1. Suppose we have the following 640-character ciphertext and we know that the length of the cipher is 8.

```
GYHHEERXJRQAE_JLNAL_E_JESQJWYDVKNIKWDEQQVVC_GNBFUYWHIAHJFQAW
GRQPYZQHNLBXGYHHOERRGOCAUAJXWICJREDXZEVAMBVCBHHHIADQFQWWMFZK
RQX_MAXYWDCQMCRKAEWHSIXEGQAWG_RKLCRZR_ZQNZVHKOKXWQDUMFZDVJLV
T_WLEQWPR_IBOQWPDOMPNHRKX_JLUVWPRRQZHJCAVMVXZVDCRSQRFQSWYIIE
SUC_FODBFQZMMCRKAEWHSABINRQFMFKOGYHZMIWXJVCKNNDLGQIMRLQLEULV
NROXZEYMMADANMHHPADKBJCZRATENRQFMHZDVVUHVFQTSQFI_NEQNUHIY_MF
GYCWDDZKOHAHYOLBNSLZQSQCZOCPVGYXWDCAUEQPHCPMD_IHLQDVQ_HBFJCW
__JESQEZREPBNJKMMSRJSQZQ_DQTWBOHFAABNTDZR_ECNORBMADANZCDR_MF
ZBCJGIBANEXZMHERFVCQ__JESQWZREIXLEXZMHVYEJCQE_EKNCAHELVBIVCL
VDQVBKCXGTQQVVUMMWZQVQDHZAXFQQPIDKVONWRZMYVYEICQMWERZUCJRLZB
IVCAUAJXGYHHIOHIRQFWGLUXAEWHIAIENZWHNWRV
```

We split the text into 8 columns by the process mentioned above; the first few lines of the split text are:

G	Y	H	H	E	E	R	X
J	R	Q	A	E	_	J	L
N	A	L	_	E	_	J	E
S	Q	J	W	Y	D	V	K
...	...	...	...	...	...	...	...

Then we look at the frequencies of letters within each column. Figures 2.27 to 2.34 present the results for each column.

In the following paragraphs, symbols and numbers are used interchangeably as in Figure 1.1.

[5]Suppose that $n = 1 \bmod m$

[6]Recall that each column was encrypted using one Caesar shift cipher.

Analysing the frequencies of symbols from the ciphertext, one can easily derive that, within a column, the most frequent symbol in the encrypted message should be the space symbol (i.e., letter 26) in the plaintext. In fact, the most common letter for each column is easily seen in Figures 2.27 to 2.34:

- Letter "N" for column 1, i.e., letter 13;
- Letter "Q" for column 2, i.e., letter 16;
- Letter "C" for column 3, i.e., letter 2;
- Letter "H" for column 4, i.e., letter 7;
- Letter "M" for column 5, i.e., letter 12;
- Letter "_" for column 6, i.e., letter 26;
- Letter "Q" for column 7, i.e., letter 16;
- Letter "X" for column 8, i.e., letter 23;

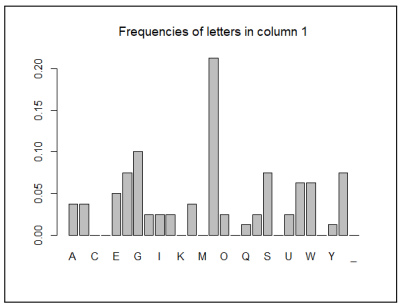


Figure 2.27: Frequencies of letters in column 1

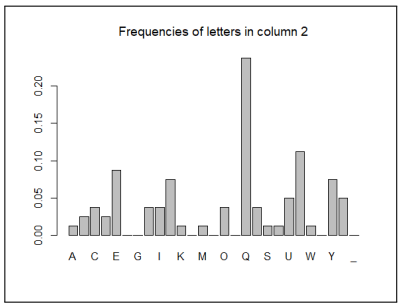


Figure 2.28: Frequencies of letters in column 2

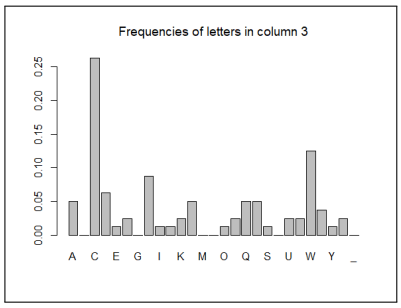


Figure 2.29: Frequencies of letters in column 3

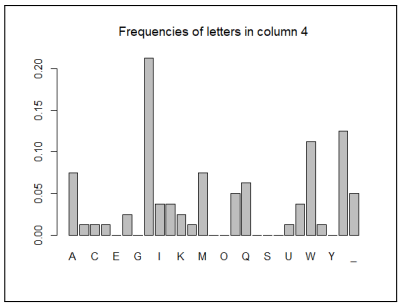


Figure 2.30: Frequencies of letters in column 4

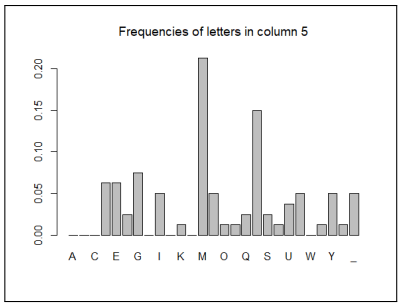


Figure 2.31: Frequencies of letters in column 5

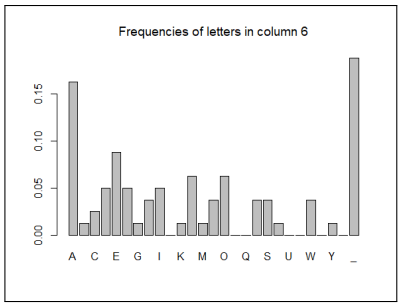


Figure 2.32: Frequencies of letters in column 6

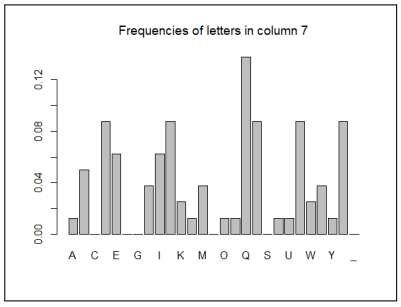


Figure 2.33: Frequencies of letters in column 7

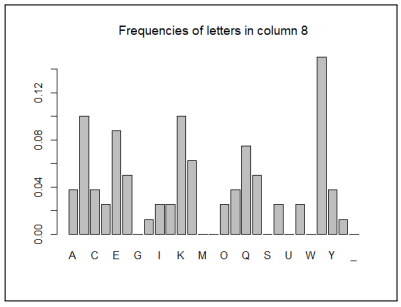


Figure 2.34: Frequencies of letters in column 8

If $k = (k_1, \ldots, k_8)$ denotes the keyword of the cipher, we can find it:

- $k_1 = 13 - 26 \bmod 27$, i.e., letter “O”;
- $k_2 = 16 - 26 \bmod 27$, i.e., letter “R”;
- $k_3 = 2 - 26 \bmod 27$, i.e., letter “D”;
- $k_4 = 7 - 26 \bmod 27$, i.e., letter “I”;
- $k_5 = 12 - 26 \bmod 27$, i.e., letter “N”;
- $k_6 = 26 - 26 \bmod 27$, i.e., letter “A”;
- $k_7 = 16 - 26 \bmod 27$, i.e., letter “R”;
- $k_8 = 23 - 26 \bmod 27$, i.e., letter “Y”;

which is the correct keyword.

Statistical method for frequency analysis

Another method is to apply, to each column[7], the statistical method presented in section 2.2.2 using the Chi-squared statistic. This method lead to the correct keyword "ORDINARY" (see R code provided in Appendix A for the implementation).

2.4.2 Unknown period

When the period is unknown, there are two things to do:

- we have to find what the period could be;
- we need to check whether or not it is the actual period that was used.

Frequency analysis

A first method is to guess what could be the length m of the cipher and to rearrange the encrypted message into m columns. The idea is to look at the frequency of letters within each column: if the guess is correct, the frequencies of letters will be similar to what is expected for the underlying language; if not, the distribution will tend to be uniform.

The code provided in Appendix A plots frequencies of letters for every possible period m from 1 to 15 (see Figures 2.35 to 2.49). To obtain these figures, we first split the ciphertext into m columns and we compute the frequencies of letters for each column. Because each column is enciphered using a single Caesar shift, we suggest to shift the vector of frequencies of each column in order that the letter with the highest frequency will be decrypted as the space character (as presented in Example 2.2.1.1). Indeed, if the period is correct, the actual shift should be found by that method. After that, for each period, the frequencies of letters are obtained by taking the average vector (sum of all of the m vectors of frequencies divided by m).

As we can see in Figures 2.35 to 2.49, only the period of 8 provides frequencies of letters similar to English. Notice that one can expect the frequencies of letters with a period of 16 to

[7]Each column was encrypted using one Caesar shift cipher.

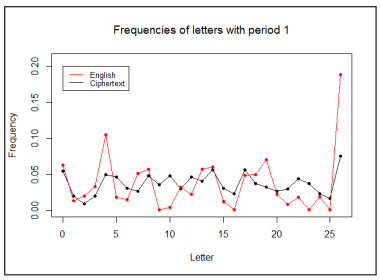


Figure 2.35: Frequencies of letters with a period of 1

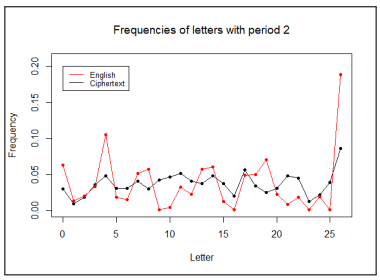


Figure 2.36: Frequencies of letters with a period of 2

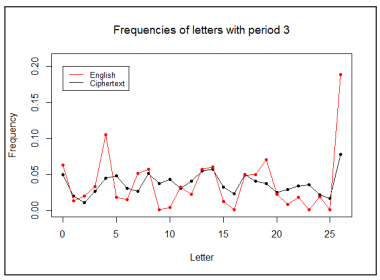


Figure 2.37: Frequencies of letters with a period of 3

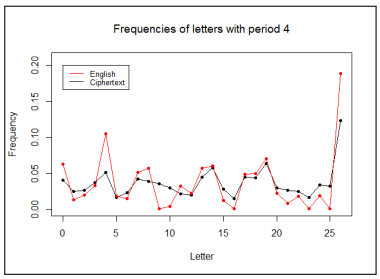


Figure 2.38: Frequencies of letters with a period of 4

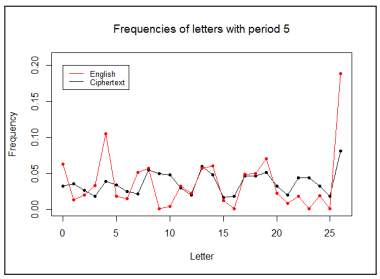


Figure 2.39: Frequencies of letters with a period of 5

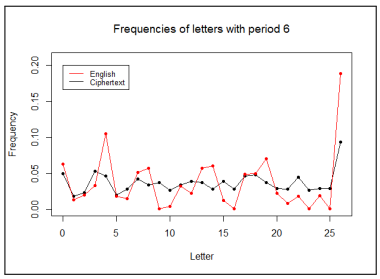


Figure 2.40: Frequencies of letters with a period of 6

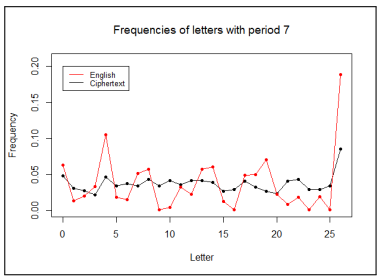


Figure 2.41: Frequencies of letters with a period of 7

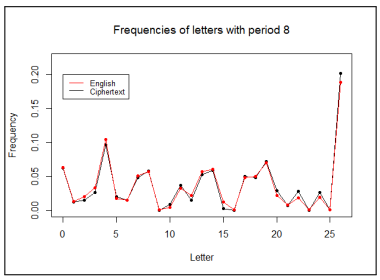


Figure 2.42: Frequencies of letters with a period of 8

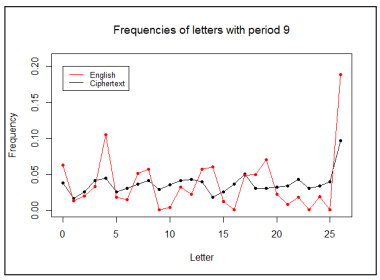


Figure 2.43: Frequencies of letters with a period of 9

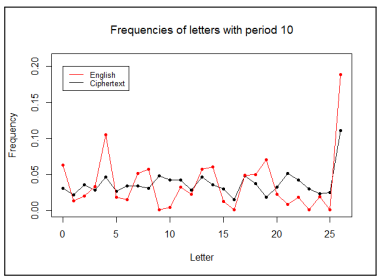


Figure 2.44: Frequencies of letters with a period of 10

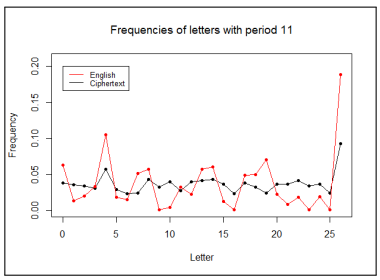


Figure 2.45: Frequencies of letters with a period of 11

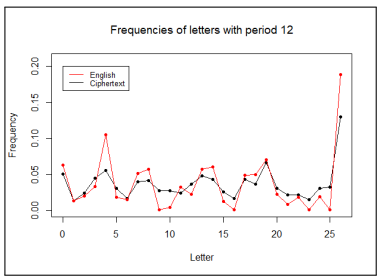


Figure 2.46: Frequencies of letters with a period of 12

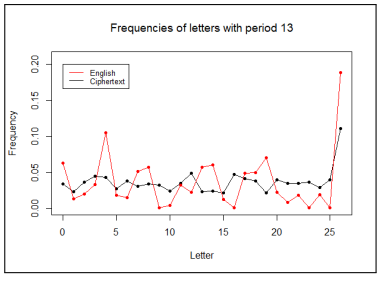


Figure 2.47: Frequencies of letters with a period of 13

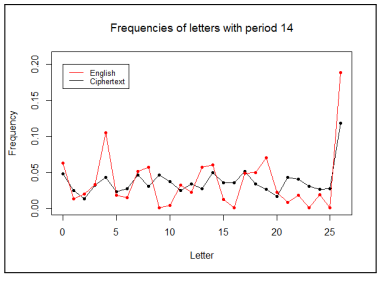


Figure 2.48: Frequencies of letters with a period of 14

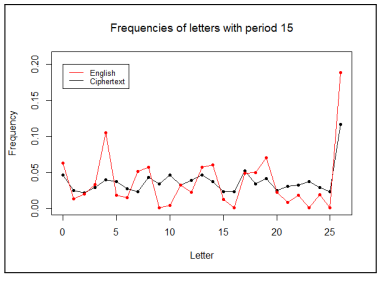


Figure 2.49: Frequencies of letters with a period of 15

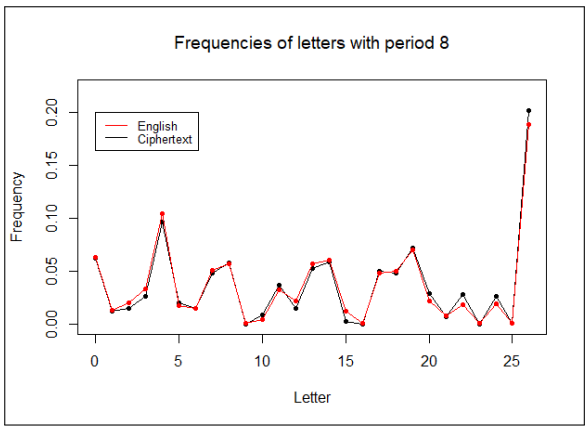


Figure 2.50: Frequencies of letters with a period of 8

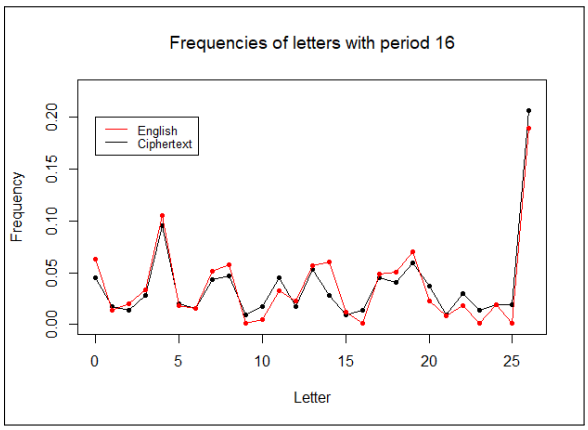


Figure 2.51: Frequencies of letters with a period of 16

be similar to English. However, in Figure 2.51, we see that the period of 8 is more similar to English than the period of 16 is.

Kasiski's method

Another method to find the period of the Vigenère cipher was invented by Friedrich W. Kasiski and relies on the following remark (see [18]): if two or more identical segments of text are repeated in the ciphertext, they can be identical segments of plaintext that were enciphered using the same segment of the key. In this case, the gap length between these two segments is congruent modulo m, where m is the period of the cipher. Thus, the value of m will be found by taking a divisor of the gcd of all these gap lengths.

However, it could be due to chance that consecutive characters are repeated in the ciphertext. Thus, following Turing's discussion about n-grams (see Turing's paper [31]), in order to implement Kasiski's technique, we start the research of identical consecutive characters with 7-grams. This technique is implemented in the R code provided in Appendix A.

Example 2.4.2.1. Assume that we have the following ciphertext:

```
GYHHEERXJRQAE_JLNAL_E_JESQJWYDVKNIKWDEQQVVC_GNBFUYWHIAHJFQAW
GRQPYZQHNLBXGYHHOERRGOCAUAJXWICJREDXZEVAMBVCBHHHIADQFQWWMFZK
RQX_MAXYWDCQMCRKAEWHSIXEGQAWG_RKLCRZR_ZQNZVHKOKXWQDUMFZDVJLV
T_WLEQWPR_IBOQWPDOMPNHRKX_JLUVWPRRQZHJCAVMVXZVDCRSQRFQSWYIIE
SUC_FODBFQZMMCRKAEWHSABINRQFMFKOGYHZMIWXJVCKNNDLGQIMRLQLEULV
NROXZEYMMADANMHHPADKBJCZRATENRQFMHZDVVUHVFQTSQFI_NEQNUHIY_MF
GYCWDDZKOHAHYOLBNSLZQSQCZOCPVGYXWDCAUEQPHCPMD_IHLQDVQ_HBFJCW
__JESQEZREPBNJKMMSRJSQZQ_DQTWBOHFAABNTDZR_ECNORBMADANZCDR_MF
ZBCJGIBANEXZMHERFVCQ__JESQWZREIXLEXZMHVYEJCQE_EKNCAHELVBIVCL
VDQVBKCXGTQQVVUMMWZQVQDHZAXFQQPIDKVONWRZMYVYEICQMWERZUCJRLZB
IVCAUAJXGYHHIOHIRQFWGLUXAEWHIAIENZWHNWRV
```

Looking at the largest n-grams (beginning with 7-grams), we found that there is one actual 9-gram: `MCRKAEWHS`. The gap length of the two occurrences (in red in the text below) of the

9-gram is 120, and the divisors are {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120}. Notice that the actual period (i.e., 8) is one of the divisors.

Friedman's index of coincidence

In this section we present how the index of coincidence, invented by William F. Friedman, can be used to find the period of a Vigenère cipher. The main references for this section are [16] and [3].

The *index of coincidence* (IC) is an estimate of the probability that two randomly selected letters from a text, noted as $\mathcal{T}$, are identical. If n_k denotes the number of occurrences of the k^{th} letter of the alphabet of N elements in the ciphertext of length n, then, naturally,

$$IC(\mathcal{T}) = \sum_{k=0}^{N-1} \frac{n_k(n_k-1)}{n(n-1)}.$$

Let p_k denote the frequency of the k^{th} letter of the alphabet. Then, if n is large,

$$IC(\mathcal{T}) \simeq \sum_{k=0}^{N-1} p_k^2.$$

In an purely random text, where the distribution of letters is uniform, we get

$$\sum_{k=0}^{26} \left(\frac{1}{27}\right)^2 = \frac{1}{27} \simeq 0.037.$$

On the other hand, in a truly English text, which is Jane Austen's *Pride and Prejudice*, we find an IC of 0.078. We find similar IC's for other English texts[8], so in the following, we will set 0.078 as the reference IC for English texts.

First Method

As suggested in [16], the IC can be used to determine the length m of the keyword used in the Vigenère cipher.

First assume that the ciphertext of n characters is divided into m (unknown) columns.

The IC can be computed as the sum of the probability that two randomly selected letters are in the same column and are identical and the probability that two randomly selected letters are in different columns and are identical.

The probability that two letters are from the same column is $\frac{1}{m}$. The probability of two letters from the same column being identical is approximately the same as for the standard English, i.e., 0.078 because both letters are enciphered using the same Caesar cipher. Finally, the probability that two randomly selected letters are from the same column and are identical is $\frac{1}{m} \times 0.078$, assuming independence.

[8] 0.078 for the *English Constitution* by Walter Bagehot and 0.08 for Shakespeare's *Romeo and Juliet*

The probability that two letters are in different columns is $1 - \frac{1}{m}$. The probability of two letters in different columns being identical is approximately the same as for a purely random text, i.e., $\frac{1}{27} \simeq 0.037$ because they are enciphered using different Caesar ciphers. Finally, the probability that two randomly selected letters are in different columns and are identical is $\left(1 - \frac{1}{m}\right) \times 0.037$.

In conclusion,

$$IC \simeq \frac{1}{m} \times 0.078 + \left(1 - \frac{1}{m}\right) \times 0.037,$$

i.e.,

$$IC \simeq \frac{0.041}{m} + 0.037 \tag{2.1}$$

which yields

$$m \simeq \frac{0.041}{IC - 0.037}. \tag{2.2}$$

Alternatively, there exist pre-tabulated tables, based on equation (2.1), to determine the length of the keyword. An example of such table is represented in Table 2.4 :

Period	1	2	3	4	5	6	7	8	9	$\cdots$	∞
IC	0.078	0.0575	0.0507	0.0473	0.0452	0.0438	0.0429	0.0421	0.0416	$\cdots$	0.037

Table 2.4: Table showing the IC of a text based on different periods

Example 2.4.2.2. Consider the ciphertext given in Example 2.4.1.1. If we compute the actual IC, we find 0.0412. Using formula (2.2), we find that the period should approximately be 8.3919 and Table 2.4 suggests a period of 8 or 9.

Combining Kasiski's method and the method based on IC can help to find what was the actual period.

Second Method

The IC does not change when we use a substitution cipher like the Caesar cipher. Since the Vigenère cipher is just multiple Caesar ciphers, we can compute the IC for each column encrypted using a single Caesar cipher and take the average. That is the main idea of the method presented in [3] to find the period of the Vigenère cipher.

To check if the period of the Vigenère cipher can be m, one has to divide the ciphertext into m columns, to consider the m strings given by these m columns and to compute the IC for each of the string. In order to obtain the IC for the entire ciphertext, we compute the mean of the IC's obtained for each column. The closer it is to 0.078 the more likely the period is correct. On the contrary, an IC near 0.037 leads to suspect that the text is a random succession of characters.

More formally, if the ciphertext $\mathcal{T}$ is $b_1 \dots b_n$ (hence a text of length n), the columns are obtained by writing down the ciphertext in rows of length m (here suppose that $n = 1 \bmod m$):

$$\begin{array}{cccc} b_1 & b_2 & \cdots & b_m \\ b_{m+1} & b_{m+2} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{N-1} & b_n. & & \end{array}$$

If $C_1, C_2, \ldots, C_m$ denote the m columns, we compute the IC for each column and we take the average:

$$IC(\mathcal{T}) = \frac{1}{m} \sum_{i=1}^{m} IC(C_i).$$

In order to find which period is the least unlikely to be right, we have to compute all average IC's for each possible period and to keep the period which IC is the closest to 0.078.

Example 2.4.2.3. Consider the ciphertext given in Example 2.4.1.1. The average IC's of the text for the periods from 1 to 17 are represented in Figure 2.52. As we can see, the correct period is found (recall that the keyword "ORDINARY" was used; hence a period of 8). However, as we should expect, all the multiples of the period will have an IC close to 0.078. The problem is that sometimes, even if the period is m, the IC of a multiple of m will be the closest to 0.078. Actually, this is not a big problem because, for example, if the actual keyword is "ORDINARY" and the algorithm suggests that the period is 16 (2 times the actual keyword), the suggested keyword will be "ORDINARYORDINARY" (2 times the actual keyword) and the message can be decrypted without any problem with that keyword.

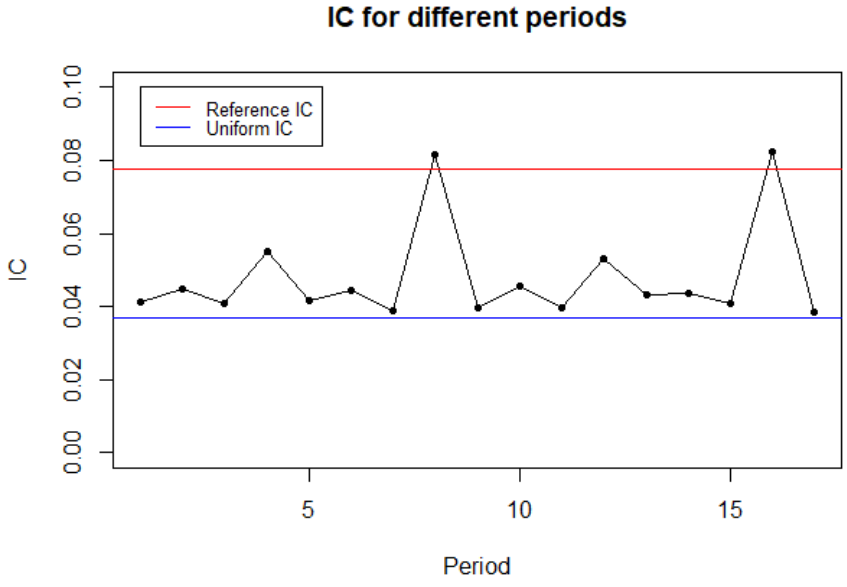


Figure 2.52: Average IC for periods 1 to 17.

2.5 The Enigma machine

Despite all the possible settings the Enigma machine contained, it was broken. It means that the machine must had a flaw. As previously mentioned, a double-letter never becomes a double-letter and most important, a letter never becomes itself. So what the codebreakers did when they had an Enigma message, was to try to guess a word or a sentence that could

appear in such a message (i.e., the standard form of the message). For example, the six O'clock morning weather report always contained the word "WETTERBERICHT"[9] and all messages finished with "HEIL HITLER". So when they had a guess, they checked that any letter from the guess did not match a letter from the encrypted message.

Example 2.5.0.1. This example is based on [16]. Consider an encrypted message with the following sequence of letters `JXATQBGGYWCRYBGDT` among others. Maybe the word "WETTERBERICHT" is in this sequence.

Attempt 1

W E T (T) E R B E R I C H T
J X A (T) Q B G G Y W C R Y B G D T

This is impossible because a T becomes a T.

Attempt 2

W E (T) T E R B E R I C H T
J X A (T) Q B G G Y W C R Y B G D T

This is impossible because a T becomes a T .

Attempt 3

W E T T E R B E R I C H T
J X A T Q B G G Y W C R Y B G D T

Everything looks good with this guess.

Attempt 4

W E T T E R B E (R) I C H T
J X A T Q B G G Y W C (R) Y B G D T

This is impossible because an R becomes an R.

Attempt 5

W E T T E R B E R I C H (T)
J X A T Q B G G Y W C R Y B G D (T)

This is impossible because a T becomes a T.

Moreover, the settings of the plugboard, i.e., the pairs of letters which were connected had to be deduced as well.

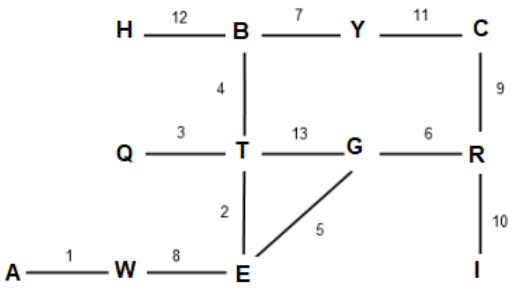
Once again, a guess had to be made to find a pair, and the work was to check if the proposal led to contradictions.

Example 2.5.0.2. Given *Attempt 3*, a correct proposal is that "WETTERBERICHT" is coded as `ATQBGGYWCRYBG`

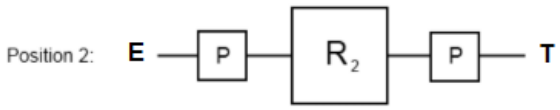
[9] "Weather report" in German

So in particular[10], for position 2, when the letter E is pressed, the enciphered letter is T.

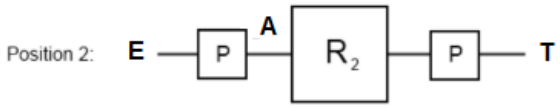
We can also deduce the pairs in the other positions. We get the menu:



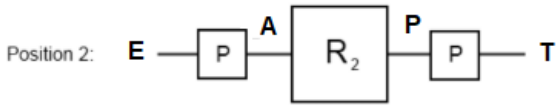
Furthermore, the i^{th} encrypted letter is the result of the plugboard, followed by the results of the three rotors and the reflector in the i^{th} position, followed by the result of the plugboard.



Suppose E is connected to A on the plugboard, so we get:



Because the wiring of the rotors is known, the letter that comes out from the rotors can be found. Let's say that A comes out as P.



That means that P is connected to T on the plugboard because the resulting letter was T. So the first deduction is the pair (P, T).

When repeating this procedure again and again, all the pairs will be found or a problem will appear such as a letter that is supposed to be connected to two different letters on the plugboard. When it's the second case, all the deductions made before are wrong as well, as Alan Turing realized.

That is how the Enigma was cracked. To deduce the day's settings, Alan Turing and George Welchman developed a machine, called the Bombe machine (see Figure 2.53), first designed by a Polish cryptologist named Marian Rejewski. This machine could decode messages from earlier versions of the Enigma machine. The development made by the British eliminated the incorrect possibilities with the process mentioned above, using electrical circuits.

[10]

W	E	T	T	E	R	B	E	R	I	C	H	T
A	T	Q	B	G	G	Y	W	C	R	Y	B	G

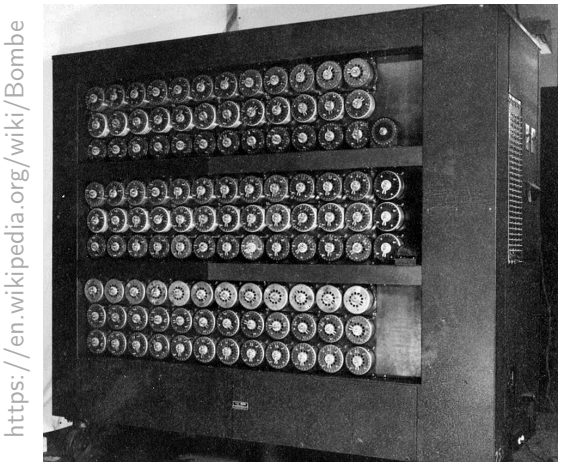


Figure 2.53: The Bombe machine

The machine was created to simulate thirty-six Enigma machine at the same time. There were three drums for each Enigma simulated, representing the three rotors. The top line of the drums simulated the fast rotor, the middle line simulated the middle rotor and the bottom line simulated the slow rotor. Each turn of a drum represented a rotor position. Thanks to the electrical circuits that were used, all the combinations of the rotor can be tested in about twenty minutes. If a given configuration didn't lead to contradictions, the machine stopped and a possible configuration of the machine could be tested. If it was the correct one, the encrypted messages can be decrypted, if not, the Bombe machine was restarted.

2.6 The Lorenz machine

Although Lorenz machine was thought unbreakable, some codebreakers actually broke it due to a terrible mistake from the Nazis. In August 1941, an operator was sending a message of 4,000 characters from Athens to Vienna and he indicated the wheels starting positions as he was supposed to do. However, the message was not received so the operator sent it again with the same settings; the indicator was transmitted once again but he abbreviated some of the 4,000 characters. By doing this, he gave two copies of the same message, with some differences, using the same key, so, when adding these codes together, it resulted that the keys "cancelled out" and the two original messages added together are displayed. Indeed, if the plus sign denotes the modulo-2 addition, we have

$$\begin{aligned}\text{Result} =& \text{Code}_1 + \text{Code}_2\\ =& \text{Code}_1 + (\text{Message}_2 + \text{key})\\ =& (\text{Code}_1 + \text{key}) + \text{Message}_2 \text{ by modulo-2 addition rules,}\\ =& \text{Message}_1 + \text{Message}_2\end{aligned}$$

Furthermore, if the first message can be found, the second will be as well because

$$\text{Result} + \text{Message}_1 = \text{Message}_2.$$

Indeed, using the XOR rule, the addition of the same letter with itself gives $[\cdot\cdot\cdot\cdot]$, which is

denoted as a "\" and the addition of any letter with "\" is the letter itself. We thus have

$$\begin{aligned}\text{Result} + \text{Message}_1 =& \text{Message}_1 + \text{Message}_2 + \text{Message}_1\\ =& \text{Message}_1 + \text{Message}_1 + \text{Message}_2\\ =& \text{Message}_2\end{aligned}$$

Example 2.6.0.1. This example is based on [15] and illustrates the fact that the addition of the Result message and the first original message gives the second original message. Suppose that the first words of the two messages are two 6-letter cities in United Kingdom. Suppose that when adding together these two encrypted cities, the word is "RSEZLS".

Suppose that we know that the first city was "LONDON". The addition, using the teleprinter code, of "RSEZLS" and "LONDON", is "OXFORD":

R	S	E	Z	L	S
L	O	N	D	O	N
O	X	F	O	R	D

It could be due to chance that "OXFORD" is found, but it is very unlikely.

If the guess for the first message is incorrect, the second message should be gibberish. Suppose now that we don't know the first city was "LONDON" and thought it might be "ALFORD". The addition, using the teleprinter code, of "RSEZLS" and "ALFORD", is "DQNDON", which is not a UK city.

R	S	E	Z	L	S
A	L	F	O	R	D
D	Q	N	D	O	N

This is how John Tiltman, Bletchley Park's chief cryptanalyst, recovered the original message and found the key. He tried a guess in one of the message, added on to the composite and saw if it gave a plausible word. When he found the key, he gave it to Bill Tutte, a Cambridge chemistry graduate, who deduced how the Lorenz machine worked, i.e., how the keys were created, without ever having seen one. At the beginning, messages were deciphered by hand but after 1943, it was not possible anymore due to German changes into the machine. However, Tommy Flowers, an electronics engineer, built the Colossus machine. It could read paper tape at a rate of 5,000 characters per second.

Actually in Bletchley Park there were two separate divisions working to decode the messages enciphered by the Lorenz machine: the Newmanry and the Testery. The Newmanry try to remove the Chi-key using statistical methods and the Testery remove the Psi-key using linguistic methods. Contrary to Enigma that had been broken using how it actually works, the Lorenz machine would never had been cracked if the sender officer had done his job as he was supposed to do. Fortunately, he did not.

Chapter 3

Alan Mathison Turing

> Sometimes it is the people no one can imagine anything of who do the things no one can imagine.
>
> *Alan Turing*

This chapter will present a bayesian method to break Vigenère cipher introduced in Chapter 1. This method was introduced by Alan Turing in a wartime paper *The Applications of Probability to Cryptography* (see [32]). The bayesian concepts will be briefly explained and some points related to the decryption of the Vigenère cipher will be summarised. This will be done thanks to Zabell's paper (see [37]). Before that, Turing's biography will be shortly described for general knowledge. It is based on Andrew Hodges' work (see [17]), Turing's main biographer.

Figure 3.1: Turing Statue at Bletchley Park

3.1 Biography

Alan Mathison Turing OBE[1] was born on 23 June 1912 in Paddington, London and is the second child of his family. He went to the English Public School, the Sherbone School from 1926 to 1931. In this school he met Christopher Morcom, a year ahead of him. Turing fell

[1]Order of the British Empire, order of chivalry

in love with him and Christopher became his best friend but this companionship ended when Christopher died of bovine tuberculosis in February 1930.

After Turing left the Sherbone School, he went to the King's College in the University of Cambridge and obtained his degree in 1934. After that, he wrote a Fellowship dissertation on central limit theorem in King's College in 1934-1935 and was elected Fellow of King's in 1935. He obtained a Smith's Prize[2] in 1936 for his work in probability theory. In 1936, he developed the idea of the Turing machine and the Universal machine that became the basis of the theory of computation and computability.

The same year he began a Ph.D (under the supervision of Alonzo Church) at Princeton University in logic, algebra and number theory and graduated in 1938. He was offered a temporary post in Princeton but preferred to come back in Cambridge.

From September 1939 to 1945, he secretly worked for the Government Code and Cypher School (GC&CS) at Bletchley Park, Britain's codebreaking centre during the Second World War. He was first assigned to the German Enigma cipher machine and he, helped by W. G. Welchman, created a decryption machine, the Bombe, to decode the Enigma (see Figure 2.53 in Section 2.5). At the end of 1939, he managed to break the U-boat Enigma machine and in 1941, the regular decryption of these messages began which helped the Allies in the Battle of the Atlantic. However, in 1942, the U-boat Enigma machine was complexified and all the previous work was useless but codebreakers managed to break it again. In 1943 he became the Chief Anglo-American crypto-consultant. At Bletchley Park, he met Joan Clarke and was engaged to her, before he retracted confessing her his homosexuality.

After the war, he worked in the National Physical Laboratory in London where he designed the Automatic Computing Engine (ACE). Meanwhile, he continued to work for Government Communications Headquarters (GCHQ)[3] until 1948.

In October 1948, he went to the University of Manchester where he was offered a post of Deputy Director of the computing laboratory.

Two years after, he created the Turing test for machine intelligence, which determines if a machine can exhibit intelligent behaviour equivalent to or, indistinguishable from, of a human.

In July 1951, he was elected to Fellowship of the Royal Society[4], the oldest national scientific institution in the world, and worked on non-linear theory of biological growth.

On 31 March 1952, he was arrested as a homosexual. He was condemned for indecent assault and had to decide between imprisonment or chemical castration with injections of oestrogen to neutralise his libido during one year. He chose the latter and continued his work on biology and physics until 7 June 1954 when he committed suicide by cyanide poisoning in Wilmslow, Cheshire.

It was not until mid-1970s that all the military secrecy around Bletchley Park was revealed and the world heard about all the men and women who worked during the war without anyone knowing it. However, all the papers they wrote at Bletchley Park were still classified until 2010's.

Finally, in 2009, more than fifty years after Turing's death, then Prime Minister Gordon

[2]"Two prizes awarded annually to two research students in mathematics and theoretical physics at the University of Cambridge from 1769. Following the reorganization in 1998, it is now awarded under the name of Smith-Knight Prize and Rayleigh-Knight Prize." from [36]

[3]The post-war successor to Bletchley Park

[4]The President, Council and Fellows of the Royal Society of London for Improving Natural Knowledge

Brown made an official public apology and said: "It is no exaggeration to say that, without his outstanding contribution, the history of World War Two could well have been very different. He truly was one of those individuals we can point to whose unique contribution helped to turn the tide of war. [...] So on behalf of the British government, and all those who live freely thanks to Alan's work I am very proud to say: we're sorry, you deserved so much better."[5] Four years after, Queen Elisabeth II granted him a posthumous royal pardon.

3.2 The Applications of Probability to Cryptography

This paper, written[6] in September 1941 and released more than seventy years after, in 2012, was probably Turing's major paper contribution in probability and statistics. It shows how probability analysis, more especially Bayes' Theorem, can be used to solve cryptanalytical problems.

In the first part of this paper, the main point is what Turing called *The Factor Principle*. In order to introduce this concept, he recalls as follows the definitions of *probability* and *odds*:

- the *probability of an event on certain evidence* is the proportion of cases in which that event may be expected to happen given that evidence.

  It is what is called now conditional probability of an event of interest A given an evidence E regarding it, i.e., $\mathbb{P}(A|E)$.

- The *odds of an event happening* or *odds in favour of an event* is the ratio $Od = \dfrac{p}{1-p}$ where p is the probability of it happening. The phraseology is *odds of* p *to* $1-p$, noted $p : 1-p$. Alternatively, one can use *odds of* x *to* y *on*, noted $x : y$, or *odds of* y *to* x *against*, noted $y : x$, where $x \geq y$ and $\dfrac{x}{x+y} = p$.

The probability given some evidence is considered because, at Bletchley Park, there was prior information available: the type of the message, its possible content, the sender, the receiver, ...

The *Factor Principle*, when considering an hypothesis, named *theory* and an evidence, named *data*, is the following:

$$\text{A posteriori odds of the theory} = \text{A priori odds of the theory} \times \frac{\text{Probability of the data being fulfilled if the theory is true}}{\text{Probability of the data being fulfilled if the theory is false}}$$

where $\dfrac{\text{Probability of the data being fulfilled if the theory is true}}{\text{Probability of the data being fulfilled if the theory is false}}$ is called *the factor for the theory on account of the data.*

Actually, if we denote $Od(H)$ the odds in favour of H, calculated with the corresponding probability $P(H)$, and $Od(H|E)$, the odds in favour of H given evidence E, calculated with the corresponding probability $P(H|E)$, the factor in favour of H provided by evidence E is defined as:

$$F(H|E) = \frac{Od(H|E)}{Od(H)}.$$

[5] http://webarchive.nationalarchives.gov.uk/20091005104048/http://www.number10.gov.uk/Page20571

[6] This document is undated but there is evidence for supposing such a date. see ZABELL (2012) [37]

This factor measures how much the odds on are modified in account of the evidence. Zabell explains that the factor principle, is just the *odds* (or *odds ratio*) form of Bayes' theorem. Let H_0 and H_1 denote two hypotheses of interest and E a form of evidence such that $\mathbb{P}(H_0) \neq 0$. $\mathbb{P}(H_1) \neq 0$ and $\mathbb{P}(E) \neq 0$. The *odds* version of Bayes' theorem is

$$\frac{\mathbb{P}(H_1|E)}{\mathbb{P}(H_0|E)} = \frac{\mathbb{P}(E|H_1)}{\mathbb{P}(E|H_0)}.\frac{\mathbb{P}(H_1)}{\mathbb{P}(H_0)},$$

that is, the posterior/final odds for H_1 versus H_0 given E equals the likelihood ratio times the initial/prior odds. Turing called the likelihood ratio, *the factor in favour of a hypothesis* H_1, provided by evidence E, omitting voluntarily, the Bayes qualification.

This odds version is easily proved using Bayes' theorem:

$$\mathbb{P}(H|E) = \frac{\mathbb{P}(E|H).\mathbb{P}(H)}{\mathbb{P}(E)} \tag{3.1}$$

where H and E are two events such as $\mathbb{P}(H) \neq 0$ and $\mathbb{P}(E) \neq 0$.

The definition of *deciban* that will follow was introduced by Turing because, when the evidence consists of independent parts, the factor principle involves many products and the wish was to replace these products by additions in order to compute them more easily (by hand during the WWII).

Indeed, when the evidence E consists of independent parts $E_1, E_2, \ldots, E_n$, the overall likelihood of E for a theory H, is the product of the individual likelihoods of E_j, that is,

$$\mathbb{P}(E_1 \cap E_2 \cap \ldots \cap E_n|H) = \mathbb{P}(E_1|H).\mathbb{P}(E_2|H).\cdots.\mathbb{P}(E_n|H)$$

as Zabell explains.

Thus,

$$\frac{\mathbb{P}(H_1|E)}{\mathbb{P}(H_0|E)} = \frac{\mathbb{P}(H_1)}{\mathbb{P}(H_0)}.\prod_{j=1}^{n}\frac{\mathbb{P}(E_j|H_1)}{\mathbb{P}(E_j|H_0)}.$$

Furthermore, because Bayes factors are a method to quantify the (relative) weight of evidence of two competing hypotheses, Turing introduces the notion of *deciban* for the units in which weight of evidence is measured.

The *deciban* or *decibanage in favour of the theory* is the logarithm of the factor $\frac{\mathbb{P}(H_1|E)}{\mathbb{P}(H_0|E)}$, taken to the base $10^{\frac{1}{10}}$, that is, 10 times the logarithm base 10 of the factor.

A *ban* is the logarithm base 10 of the factor.

According to a Bletchley Park 1944 Cryptographic Dictionary[7], a *ban* is a "fundamental scoring unit for the odds on, or probability factor of, one of a series of hypotheses which, in order that multiplication may be replaced by addition, are expressed in logarithms. One ban thus represents an odds of 10 to 1 in favour, and as this is too large a unit for most practical purposes decibans and centibans are normally employed instead." In others words, a ban is to the base 10 what a bit is to the base 2. However, the terminology of a *bit* was created by Shannon and Tukey, eight years after Turing invented the ban. The reason for the name of "ban" was the huge amount of sheets that were printed in Banbury in England, on which

[7]See [28]

weight of evidence were entered in decibans for a process called *Banburismus*. In a few words, this process determined the probability of the Enigma settings based on sequential conditional probabilities. Actually, I.J. Good, Turing's assistant noticed that instead of using the deciban, using the *half-deciban*, 20 times the logarithm base 10 of the factor $\frac{\mathbb{P}(H_1|E)}{\mathbb{P}(H_0|E)}$, and rounded to the nearest integer will save time in writing and arithmetic because most values would become single digit. As he said: "This must have saved half the time of the work on Banburismus!".

After having introduced the previous definitions, Turing gives an example of an encrypted message using the Vigenère cipher and a period of 10. However we decided to use the example presented in section 1.3.3 where the chosen text *Ordinary Love* was encrypted with a Vigenère cipher and a period of 8. Although Turing does not consider the space character in his paper because of the standard form of military message during the Second World War (no space character), we decide to "extend" his paper by taking into account the space character, for consistency with the previous chapters. In the following, we will alternate between the consideration of the example (using letters available in the example) and the general case (using general notations for letters: α, β, etc).

The example of the Vigenère ciphertext written out in its correct period is presented below:

G	Y	H	H	E	E	R	X
J	R	Q	A	E	_	J	L
N	A	L	_	E	_	J	E
S	Q	J	W	Y	D	V	K
N	I	K	W	D	E	Q	Q
V	V	C	_	G	N	B	F
U	Y	W	H	I	A	H	J
F	Q	A	W	G	R	Q	P
Y	Z	Q	H	N	L	B	X.

The goal is to find the key, i.e., the shift used for each column, thanks to the factor principle. First, he computes for each column the odds in favour of the key, say β, using the factor principle and only takes into account the evidence given by a single letter, say α, from that column, i.e.,

$$\begin{aligned}\text{A posteriori odds in favour of key } \beta &= \text{A priori odds in favour of key } \beta \\ &\times \frac{\text{Probability of getting } \alpha \text{ in cipher if key is } \beta}{\text{Probability of getting } \alpha \text{ in cipher if key is not } \beta}.\end{aligned}$$

For each column, the key with the highest a posteriori odds is supposed to be the key that was used. In the following the letters and the numbers are again used interchangeably.

The probability that the key is β may be taken as $\frac{1}{27}$, thus the a priori odds in favour of key β is $\frac{\frac{1}{27}}{1-\frac{1}{27}} = \frac{1}{26}$.

For example, for the first column of the table when considering the key B, we have

$$\begin{aligned}\text{A posteriori odds in favour of key } B &= \text{A priori odds in favour of key } B \\ &\times \frac{\text{Probability of getting } G \text{ in cipher if key is } B}{\text{Probability of getting } G \text{ in cipher if key is not } B}.\end{aligned}$$

For the first column, if we test the key B, the probability of having a letter G in the cipher with the key B is the probability of having F as plaintext (what Turing defines to be *in the*

clear), i.e., as plaintext, which is the frequency of the letter F in plain language noted p_F. If the key is not B, any character other than F can be in the clear (i.e., 26 choices) and the probability is $\frac{1-p_F}{26}$.

More generally, the probability of having a letter α in the cipher with the key β is just the probability of having $\alpha-\beta$ in the clear which is the frequency of the letter $\alpha-\beta$ in plain language noted $p_{\alpha-\beta}$. If the key is not β, any other letter can be in the clear (i.e., 26 choices) and the probability is $\frac{1-p_{\alpha-\beta}}{26}$.

Thus, given the letter α, the odds in favour of the key β for a given column, i.e., the posterior odds of the theory are

$$\begin{aligned}&\text{A priori odds in favour of key } \beta \times \frac{\text{Probability of getting } \alpha \text{ in cipher if key is } \beta}{\text{Probability of getting } \alpha \text{ in cipher if key is not } \beta}.\\ =&\frac{1}{26}\frac{26p_{\alpha-\beta}}{1-p_{\alpha-\beta}}\end{aligned}$$

After having considered the evidence given by a single letter, we take the evidence given by the whole column and if we assume that the evidence of one letter is independent of another, the a posteriori odds in favour of key β

$$\frac{1}{26}\prod_i \frac{26p_{\alpha_i-\beta}}{1-p_{\alpha_i-\beta}}$$

where $\alpha_1, \alpha_2, \ldots$ is the series of letters in the that column.

The easiest way to compute this product is to create a table of the factors $\frac{26p_\gamma}{1-p_\gamma}$ in half-decibans, taken to the nearest integer, i.e., $20\log\left(\frac{26p_\gamma}{1-p_\gamma}\right)$.

However, if we must do it by hand, two improvements can be done:

- add columns, beginning in the right side of the sheet, containing multiples of the values in half-decibans, called *multiple-columns*;
- add a second copy of this table underneath the first.

Furthermore, a transparent "gadget" can be prepared to speed the calculations: a sheet of paper with the letters displayed vertically and a hole is put next to each letter that appears in the column of the message. If a letter appears n times, the hole will be placed in the corresponding multiple-column, i.e., in the n^{th} multiple-column (beginning in the right side). The figure 3.2 (p. 47) shows

- the table for scoring a Vigenère in units of half-decibans where the frequencies of letters based on Jane Austen's text *Pride and Prejudice* are used,
- the gadget built for the example that is considered here,
- the combination of the table and the gadget, using the key B.

For example, the first column is `GJNSNVUFY`. If we use the key B, i.e., a Caesar shift of 1, the decrypted column is `FIMRMUTEX` and the scores are -6 for F, 4 for I, -9 for the two M, 3 for R, -4 for U, 6 for T, 10 for E and -30 for X, that is, in total, -26.

These calculations can be done for each of the 27 possible keys. Table 3.1 presents the results for each column and for each possible keys.

	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8
A	-55	-106	-115	**27**	**14**	**60**	-148	-145
B	-26	-101	-76	-104	-53	-103	-45	-52
C	-19	-74	-47	-73	-21	-101	**36**	-46
D	-67	-29	**33**	-18	-60	-88	-48	-25
E	-108	-44	-77	-37	-3	-38	-28	-9
F	-18	-38	-20	-40	**9**	-39	-84	**3**
G	**18**	-54	-117	-144	-81	-89	-70	-52
H	-24	-78	-70	-6	-31	-50	-105	-120
I	-108	-67	**6**	**71**	-185	-90	-28	-63
J	-57	-27	-29	-66	-27	-8	-2	-41
K	-33	-41	-6	-93	-128	-98	**16**	-13
L	-99	-65	-81	-143	-26	-62	-121	-30
M	-79	-61	-31	-73	-49	**15**	-98	-10
N	-18	**3**	-137	-115	**0**	**6**	-79	-98
O	**21**	-55	-59	-6	-27	-23	-16	-109
P	-108	-117	-59	**20**	-163	-145	-43	-74
Q	-42	-21	-15	-53	-16	-73	-20	-38
R	-92	**41**	-6	-88	-71	-102	**59**	-46
S	-24	-81	-104	-88	-23	-12	-105	-35
T	-19	-95	-25	-22	-57	-36	-110	-6
U	-65	-68	-112	-16	-17	-40	-126	-70
V	**10**	-33	-60	-35	-83	-107	-46	-88
W	-58	-30	-67	**14**	-84	-77	-28	-120
X	-88	-69	**6**	**52**	-32	-1	-57	-15
Y	-138	**1**	-39	-152	-86	-46	-6	**50**
Z	-37	-6	-27	-129	-34	-45	-52	-58
_	-56	-77	-55	-76	-57	**0**	-41	-79

Table 3.1: Table of scores for different columns

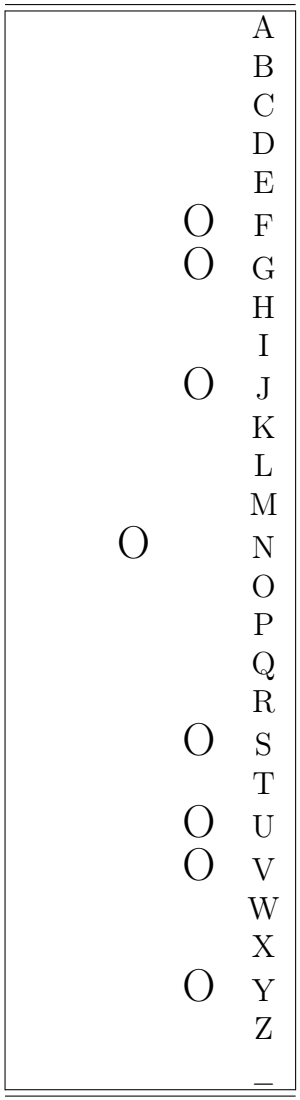
The highest score a possible key has, the more probable it is.

Taking the highest score for each column, one gets the keyword "ORDIAARY". However, the actual keyword was "ORDINARY". Nevertheless, if we decrypt the ciphertext using the keyword, we found:
`THE_EEA_WANTE_TO_KISE_THE_GOYDEN_SHODE_THE_SGNLIGHT_IARMS_YOGR_SKIN_NLL_`,
i.e., written in columns,

```
T H E _ E E A _
W A N T E _ T O
_ K I S E _ T H
E _ G O Y D E N
_ S H O D E _ T
H E _ S G N L I
G H T _ I A R M
S _ Y O G R _ S
K I N _ N L L _
```

The errors can be easily fixed with human intervention by considering, for the fifth column, the remaining possible best keys F or N.

24	19	15	10	5	A
-44	-35	-26	-18	-9	B
-27	-21	-16	-11	-5	C
-4	-3	-3	-2	-1	D
48	39	29	19	10	E
-32	-25	-19	-13	-6	F
-40	-32	-24	-16	-8	G
15	12	9	6	3	H
20	16	12	8	4	I
-146	-117	-88	-59	-29	J
-90	-72	-54	-36	-18	K
-6	-5	-3	-2	-1	L
-23	-18	-14	-9	-5	M
20	16	12	8	4	N
22	18	13	9	4	O
-48	-39	-29	-19	-10	P
-161	-129	-96	-64	-32	Q
13	10	8	5	3	R
14	11	8	5	3	S
30	24	18	12	6	T
-22	-18	-13	-9	-4	U
-64	-51	-39	-26	-13	V
-31	-25	-18	-12	-6	W
-148	-119	-89	-59	-30	X
-29	-23	-18	-12	-6	Y
-143	-115	-86	-57	-29	Z
78	63	47	31	16	_



24	19	15	10	5	A
-44	-35	-26	-18	-9	B
-27	-21	-16	-11	-5	C
-4	-3	-3	-2	-1	D
48	39	29	19	10	E
-32	-25	-19	-13	-6	F
-40	-32	-24	-16	-8	G
15	12	9	6	3	H
20	16	12	8	4	I
-146	-117	-88	-59	-29	J
-90	-72	-54	-36	-18	K
-6	-5	-3	-2	-1	L
-23	-18	-14	-9	-5	M
20	16	12	8	4	N
22	18	13	9	4	O
-48	-39	-29	-19	-10	P
-161	-129	-96	-64	-32	Q
13	10	8	5	3	R
14	11	8	5	3	S
30	24	18	12	6	T
-22	-18	-13	-9	-4	U
-64	-51	-39	-26	-13	V
-31	-25	-18	-12	-6	W
-148	-119	-89	-59	-30	X
-29	-23	-18	-12	-6	Y
-143	-115	-86	-57	-29	Z

78	63	47	31	16	_
24	19	15	10	5	A
-44	-35	-26	-18	-9	B
-27	-21	-16	-11	-5	C
-4	-3	-3	-2	-1	D
48	39	29	19	(10)	E
-32	-25	-19	-13	(-6)	F
-40	-32	-24	-16	-8	G
15	12	9	6	3	H
20	16	12	8	(4)	I
-146	-117	-88	-59	-29	J
-90	-72	-54	-36	-18	K
-6	-5	-3	-2	-1	L
-23	-18	-14	(-9)	-5	M
20	16	12	8	4	N
22	18	13	9	4	O
-48	-39	-29	-19	-10	P
-161	-129	-96	-64	-32	Q
13	10	8	5	(3)	R
14	11	8	5	3	S
30	24	18	12	(6)	T
-22	-18	-13	-9	(-4)	U
-64	-51	-39	-26	-13	V
-31	-25	-18	-12	-6	W
-148	-119	-89	-59	(-30)	X
-29	-23	-18	-12	-6	Y
-143	-115	-86	-57	-29	Z

78	63	47	31	16	_

Figure 3.2: Table for scoring a Viginère in units of half-decibans and the gadget for the first column in the encrypted message.

Chapter 4

Decryption using Markov Chain Monte Carlo

This chapter will explain how one can use the Markov Chain Monte Carlo theory (and more especially the Metropolis-Hastings algorithm) to decode substitution and transposition ciphers. Furthermore, the Metropolis-Hastings algorithm will be used to find the unknown period of a ciphertext encrypted using transposition cipher. This chapter will be widely based on three references from Persi Diaconis, Stephen Connor and Chen and Rosenthal (see [8], [5]and [4]). Furthermore we will use papers from Dobrow and Fathi-Vajargah and Kanafchian (see [9] and [10]). The first definitions will be taken from Yvik Swan's lectures notes (see [30]).

We will focus on the decryption of a text that was encrypted using substitution or transposition ciphers. The goal will be to find the inverse permutation π that was used. In order to find it, we will look at the probability that a letter of the alphabet follows another in a given language. We will extend our alphabet to symbols such as numbers because in the following chapter we will encrypt and decrypt texts from a huge database of texts and in general, ignoring numbers would be prejudicial for the understanding of the original message.

The texts used in this chapter will be modified as follows: all the letters will be replaced by their uppercase equivalents and the non alphabetic or non numeric characters will be converted to spaces. Thus, the alphabet will contain 37 characters and will be ordered as follows:

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
1	2	3	4	5	6	7	8	9	0	_		

4.1 Markov Chain: definitions

A discrete *Markov chain* $\{X_t : t \in \mathbb{N}\}$ is a sequence of dependent random variables $X_0, X_1, \ldots, X_t, \ldots$ such that for all $t \in \mathbb{N}$, $\mathcal{L}(X_{t+1}|X_0, \ldots, X_t) = \mathcal{L}(X_{t+1}|X_t)$ i.e. at any time t the one-step-ahead probability distribution of X_{t+1} given the entire past $\sigma(X_s, 0 \leq s \leq t)$ depends only on the present state of knowledge $\sigma(X_t)$.

The state-space χ is the countable of finite set of values the chain is allowed to take. Let k (with k being an integer or infinity) denote the cardinality of the set χ.

For x and y belonging to χ, let p_{xy} denote the probability $\mathbb{P}(X_{t+1} = y|X_t = x)\forall t \in \mathbb{N}$. The probabilities $p_{xy}(x, y \in \chi)$ are called the *transition probabilities*.

Let $\boldsymbol{P}$ be the $k \times k$ matrix that collects all these probabilities with $(\boldsymbol{P})_{xy} = p_{xy}$

The chain is *irreducible* if it allows free moves all over the state-space, i.e. if no matter the starting point X_0 there is a positive probability of eventually reaching any region of the state-space.

The chain $\{X_t : t \in \mathbb{N}\}$ is *aperiodic* if $(\boldsymbol{P})_{xx} > 0 \ \forall x \in \chi$, i.e, there is a positive probability of staying at the current position.

The discrete probability distribution $\boldsymbol{s} = (s_1, s_2, \ldots, s_k)$ on a state space χ is *stationary* for the chain $\{X_t : t \in \mathbb{N}\}$ (with transition matrix $\boldsymbol{P}$) if

$$\boldsymbol{sP} = \boldsymbol{s}.$$

Thus, if we choose $X_0 \sim \boldsymbol{s}$ as the initial distribution, then the chain will always have the same distribution.

4.2 General algorithm

Given a target discrete probability distribution $\boldsymbol{s} = (s_1, s_2, \ldots, s_k)$ (with k being an integer or infinity) on a state space χ, the Metropolis-Hastings algorithm builds a Markov chain $X_0, X_1, \ldots$ (denoted $\{X_t : t \in \mathbb{N}\}$) with stationary distribution $\boldsymbol{s}$. The following theorem (see Theorem 4.2.0.1) ensures that, for large value of t, X_t is approximately distributed from $\boldsymbol{s}$.

Theorem 4.2.0.1. *If a Markov chain $\{X_t : t \in \mathbb{N}\}$ on a finite or countable state space χ is irreducible and aperiodic, with stationary distribution $\boldsymbol{s} = (s_1, s_2, \ldots, s_k)$ (with k being an integer or infinity), then for every $A \subseteq \chi$,*

$$\lim_{t \to \infty} \mathbb{P}(X_t \in A) = \sum_{x \in A} s_x.$$

Given the distribution $\boldsymbol{s}$ and some matrix $\boldsymbol{Q}$ of probabilities $\{q_{xy} : x, y \in \chi\}$ the algorithm generates a Markov chain according to:

1: Given $X_t = x \in \chi$,
2: Generate Y from the probability distribution $\{q_{xy} : y \in \chi\}$
3: Take

$$X_{t+1} = \begin{cases} Y & \text{with probability } \rho(x, Y) \\ x & \text{with probability } 1 - \rho(x, Y) \end{cases}$$

where $\rho(x, y) = \min\left(\dfrac{s_y q_{yx}}{s_x q_{xy}}, 1\right)$ is the Metropolis-Hastings acceptance probability.

If the proposal matrix $\boldsymbol{Q}$ is *symmetric*, i.e. $q_{xy} = q_{yx}$, the acceptance probability for each proposal is $\rho(x, y) = \min\left(\dfrac{s_y}{s_x}, 1\right)$.

Let U([0,1]) denote the continuous uniform distribution on the interval [0,1], the general algorithm is:

Algorithm 1 Metropolis-Hastings algorithm

1: $t = 0$
2: $X_0 = x_0$
3: **for** $t = 1$ to T **do**
4: Given $X_t = x$
5: Generate Y from the probability distribution $\{q_{xy} : y \in \chi\}$
6: Compute $\rho(x, Y) = \min\left(\frac{s_Y q_{Yx}}{s_x q_{xY}}, 1\right)$
7: Generate $u \sim U([0, 1])$
8: **if** $u \leq \rho(x, Y)$ **then**
9: $X_{t+1} = Y$
10: **else**
11: $X_{t+1} = x$
12: **end if**
13: **end for**

4.3 Matrix of bigrams

The matrix of bigrams is a matrix whose element (i, j) records the number of times the j^{th} symbol of the alphabet follows the i^{th} symbol in a given text. In order to compare matrices of bigrams for different texts, we create a transition matrix whose element (i, j) is the probability that the j^{th} symbol of the alphabet follows the i^{th} symbol in a given text. This is simply the matrix of bigrams modified to be a probability matrix. The transition matrix obtained using Austen's text is represented in Figure 4.1.

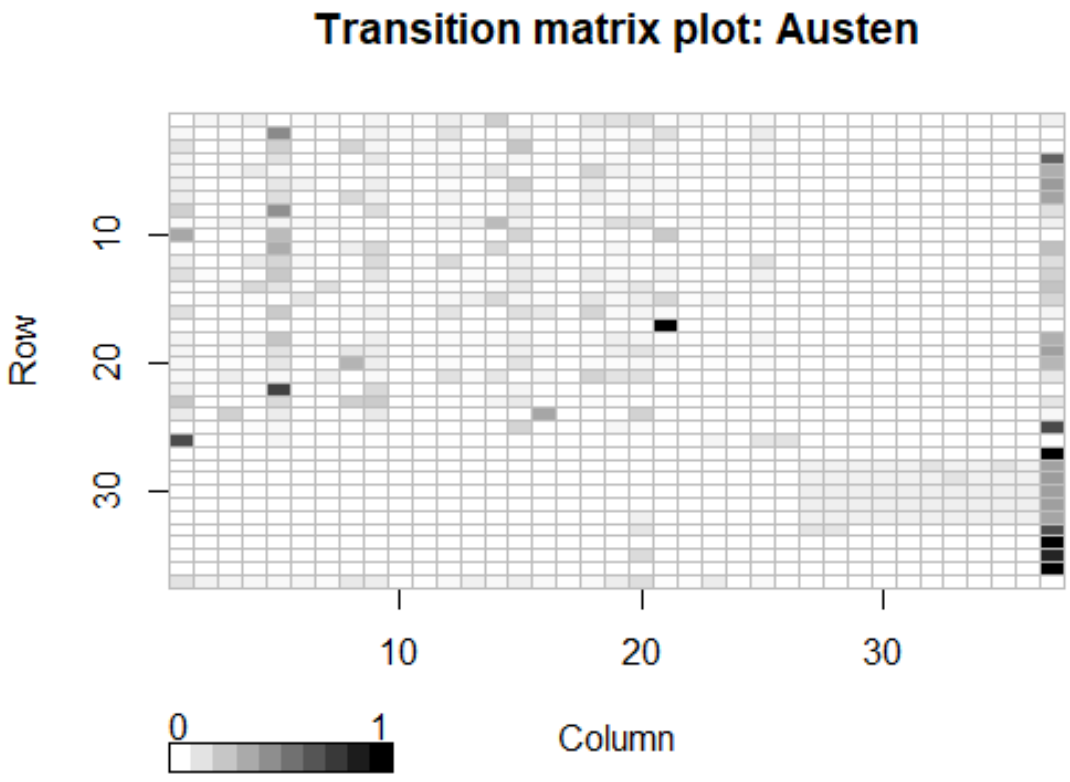


Figure 4.1: The transition matrix of the reference text

From Figure 4.1, it can be seen that the most common column (i.e., the most "colored" column) is the last one, corresponding to the space character. Then, the second most common column is the fifth, corresponding to the letter E. This is not surprising when considering the frequency analysis of English done in section 2.2.1. We found zero for elements of the matrix corresponding to letters following numbers and for elements of the matrix corresponding to numbers following letters. Indeed, in a text with spaces, no number directly follows a letter and no letter directly follows a number.

The algorithm presented in the next section will be based on the computed matrix of bigrams. This suggests to compare the transition matrices of different texts to see whether or not they are different. Thus we compare three English texts and three texts from different languages. We only present the transition matrices of different texts but we won't implement the algorithm using these different texts because these comparisons were widely studied in [5].

4.3.1 Transition matrix for different English texts

The texts are the same as those used for the frequency analysis comparison (see section 2.2.1): Jane Austen's *Pride and Prejudice* (1813), the *English Constitution* by Walter Bagehot (1865) and Shakespeare's *Romeo and Juliet* (1597).

Figures 4.2 and 4.3 present the results.

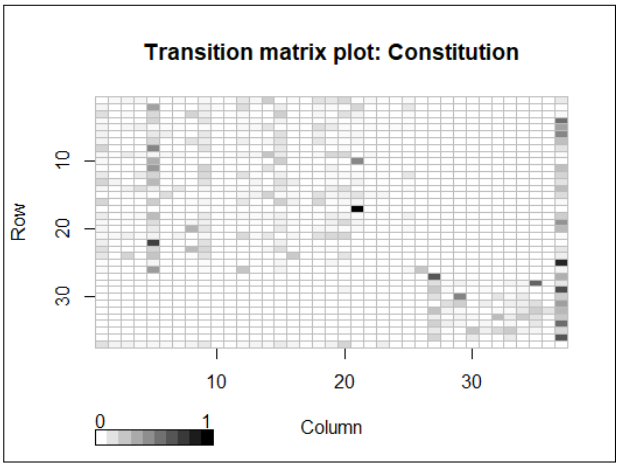


Figure 4.2: Transition matrix for the English Constitution

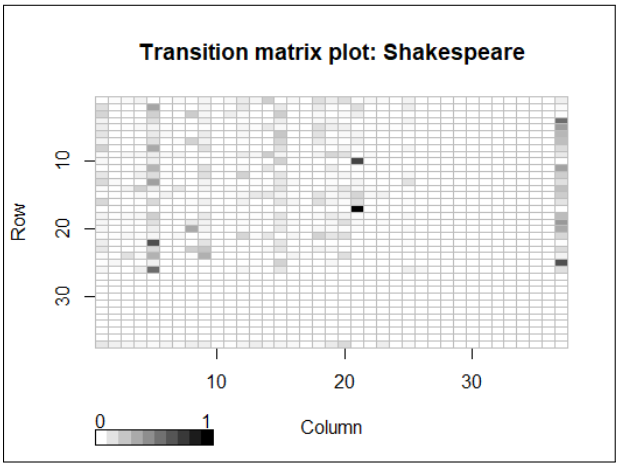


Figure 4.3: Transition matrix of Shakespeare's text

As we can see, except the use of numbers in Austen's text and in the English Constitution but not in Shakespeare's text, the brief analysis of Austen's matrix done previously, can be applied to the two new matrices. We must not use a reference text without any numbers (like Shakespeare's text) for the decryption process if we think that the encrypted message can contain numbers.

4.3.2 Transition matrix for different languages

We compare the transitions matrices of three different texts in different languages: Jane Austen's *Pride and Prejudice* (1813) (English text), Victor Hugo's *Les Misérables* (1862) (French text) and Francesco Domenico's *Amelia Calani* (1862) (Italian text). We modify the texts in the same way as before.

Figures 4.4 and 4.5 represent the transition matrices of the two non English texts.

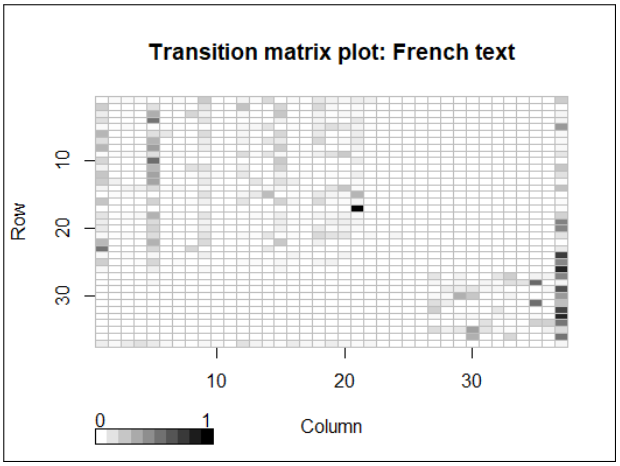


Figure 4.4: Transition matrix for the French text

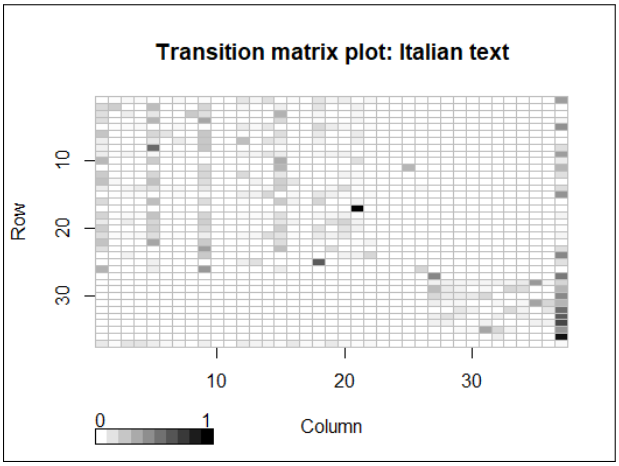


Figure 4.5: Transition matrix for the Italian text

We can see some differences between the matrices, where some bigrams are more likely to occur depending on the used language. It is fairly consistent with the frequency analysis done in section 2.2.1. The columns of most probable elements correspond to the most common symbols.

4.4 Adaptation of the Metropolis Algorithm for decrypting a substitution ciphertext

In this adaptation, the state space χ consists of all the possible keys corresponding to all the possible permutations, i.e., 37! elements.

4.4.1 Plausibility function

Let M denote a matrix of bigrams where M_{ij} records the number of occurrences of the pair (i, j) in the reference text and $\hat{M}_{ij}$ denote the number of occurrences of the pair (i, j) in the decrypted text using permutation $\hat{\pi}$. To avoid problems of zeroes, we first add one to each element of the matrices M and $\hat{M}$.

Given a possible permutation $\hat{\pi}$, its *plausibility function* is defined as

$$Pl(\hat{\pi}) = \prod_{i,j} (M_{ij})^{\hat{M}_{ij}} \tag{4.1}$$

Intuitively, the best guess is any permutation $\hat{\pi}$ with the highest plausibility, because in this case as Chen and Rosenthal said, "pair frequencies in the decrypted message (using the possible permutation $\hat{\pi}$) match those of the reference text".

We can derive a probability distribution $\boldsymbol{s}$ proportional to the plausibility by defining

$$s_{\hat{\pi}} = \frac{Pl(\hat{\pi})}{\sum_{\hat{\pi}'} Pl(\hat{\pi}')} \tag{4.2}$$

where the denominator contains 37! terms representing all the possible permutations.

The goal is to sample from the distribution $\boldsymbol{s}$. Fortunately, we do not need to know the denominator of (4.2) thanks to the Metropolis-Hastings algorithm previously presented. Instead only the ratios of two probabilities like (4.2) are needed. Indeed, given a matrix $\boldsymbol{Q}$ of probabilities, a current state π_1 and a proposal π_2, the algorithm requires the computation of the acceptance probability

$$\rho(\pi_1, \pi_2) = \min\left(\frac{s_{\pi_2} q_{\pi_2\pi_1}}{s_{\pi_1} q_{\pi_1\pi_2}}, 1\right),$$

which is simply given by

$$\min\left(\frac{Pl(\pi_2) q_{\pi_2\pi_1}}{Pl(\pi_1) q_{\pi_1\pi_2}}, 1\right)$$

4.4.2 Algorithm

The algorithm runs as follows:

- Start with an initial guess $\hat{\pi}$ for the decoding permutation;
- Compute the plausibility of $\hat{\pi}$ on the encrypted text: $Pl(\hat{\pi})$;
- Repeat the following steps for a sufficient number of iterations:
  - Switch randomly the values $\hat{\pi}$ assigns to two symbols; the new permutation is denoted $\hat{\pi}'$;
  - Compute the plausibility of $\hat{\pi}'$: $Pl(\hat{\pi}')$.
    - ➔ If $Pl(\hat{\pi}') > Pl(\hat{\pi})$, keep $\hat{\pi}'$
    - ➔ Else, keep $\hat{\pi}'$ with probability $\dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$ and keep $\hat{\pi}$ with probability $1 - \dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$.

For a given permutation $\hat{\pi}$, the probability distribution $\{q_{\hat{\pi}\hat{\pi}'} : \hat{\pi}' \in \chi\}$ is defined as follows:

$$q_{\hat{\pi}\hat{\pi}'} = \begin{cases} \dfrac{1}{37^2} & \text{if the transition from } \hat{\pi} \text{ to } \hat{\pi}' \text{ can be made by randomly swapping two values} \\ 0 & \text{else.} \end{cases}$$

The proposal matrix $\boldsymbol{Q}$ of probabilities $\{q_{\hat{\pi}\hat{\pi}'} : \hat{\pi}, \hat{\pi}' \in \chi\}$ is symmetric, hence the simplification for the acceptance probability.

Furthermore, a new proposal $\hat{\pi}'$ with smaller plausibility than $\hat{\pi}$ can still be accepted with probability $\dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$. As explained in [3], this "allows to escape a local maximum by temporarily accepting a worse solution".

Given a reference text, we construct the matrix of bigrams. Then, if $U(\{1, 2, \ldots, 37\})$ denotes the discrete uniform distribution on integers 1 to 37, the algorithm is presented in the following page (see Algorithm 2).

As Chen and Rosenthal explained, "intuitively, after many iterations, the algorithm is likely to be at a decryption key which gives decryption text pair frequencies close to those of the reference text, and is thus more likely to be correct". They also proved that the above algorithm will converge to solutions with maximal plausibility functions. The proof is based on Markov chain law of large numbers which requires the Markov chain to be irreducible and aperiodic. In our case, $\{X_t : t = 0, \ldots, T\}$ is the sequence of decryption keys produced by the algorithm using

Algorithm 2

$t = 0$
$X_0 = \hat{\pi}_0$
for $t = 1$ to T **do**
    Given $X_t = \hat{\pi}$
    Generate $i, j \sim U(\{1, 2, \ldots, 37\})$
    Generate $\hat{\pi}'$ by swapping characters with positions i, j on the last key $\hat{\pi}$
    Compute $\rho(\hat{\pi}, \hat{\pi}') = \dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$
    Generate $u \sim U([0, 1])$
    if $u \leq \rho$ **then**
        $X_{t+1} = \hat{\pi}'$
    else
        $X_{t+1} = \hat{\pi}$
    end if
end for

the plausibility function. The chain is irreducible because every permutation can be obtained from any other by a series of swaps and every proposed swap has a positive probability to be accepted since $Pl(\pi) > 0$ (we added one to elements of the matrix M) for all decryption keys π. Furthermore, since the proposal of swapping a letter with itself is allowed, the Markov chain is aperiodic.

4.4.3 Implementation

There are three parts in the R code provided in Appendix A.

The first part consists of the construction of the matrix of bigrams based on Jane Austen's text.

The second part is the encryption process of a substitution cipher as presented in section 1.3.1. However, in this chapter, we consider an alphabet of 37 elements as said earlier.

The final part of the R code is the decryption using the Metropolis-Hastings algorithm. In order to choose the starting decryption key, as suggested by [5], the most frequent character of the encrypted text is mapped with the most frequent character in the reference text, the second most common symbol of the ciphertext is mapped with the second most common symbol of the reference text, etc. Furthermore, we compute the plausibility on the log-scale to avoid numerical errors.

4.4.4 Results

The results obtained when running the algorihtm are presented in this section. We encrypt three texts: the chosen text corresponding to the song *Ordinary love* (see Chapter 1.1), the first chapter of Austen's *Pride and Prejudice* and a French text about the song *Ordinary Love*.

The *accuracy*, also called *efficiency*, is based on the number of letters correctly revealed. A letter is *correctly revealed* if its first occurrence in the decrypted text is the same as in the

plaintext. Then, the *accuracy* is defined as $\frac{m_s}{n_s}$ where m_s is the number of letters correctly revealed and n_s is the number of different letters in the plaintext (which may not be 37).

The choice of this definition will be explained in Chapter 5.

Ordinary Love

When we encipher the chosen text *Ordinary love* with the permutation cipher, we find:

```
M_PAGPLAIL9MGAMRAQ5GGAM_PAZRWBP9AG_RYPAM_PAGX9W5Z_MAILYHGAFRXYAG
Q59ALWWAM_PAJPLXMFAM_LMA5GAJPP9AWRGMAJP6RYPAIL9MGAMRA659BAXGALZL
59A5ATL99RMA65Z_MAFRXAL9FHRYPA5MA5GAFRXA5ALHA65Z_M59ZA6RYAM_PAGP
LAM_YRIGAYRTQAMRZPM_PYAJXMAM5HPAWPL3PGAXGAORW5G_PBAGMR9PGAIPATL9
9RMA6LWWAL9FA6XYM_PYA56AIPATL99RMA6PPWARYB59LYFAWR3PAL9BAIPATL99
RMAYPLT_AL9FA_5Z_PYA56AIPATL99RMABPLWAI5M_ARYB59LYFAWR3PAJ5YBGA6
WFA_5Z_A59AM_PAGXHHPYAGQFAL9BAYPGMAR9AM_PAJYPPCPAM_PAGLHPAI59BAI
5WWAMLQPATLYPAR6AFRXAL9BA5AIPAI5WWAJX5WBARXYA_RXGPA59AM_PAMYPPGA
FRXYA_PLYMA5GAR9AHFAGWPP3PAB5BAFRXAOXMAM_PYPAI5M_ALAHLZ5TAHLYQPY
A6RYAFPLYGA5AIRXWBAJPW5P3PAM_LMAM_PAIRYWBATRXWBA9RMAILG_A5MALILF
```

The algorithm was run 10,000 times and these are the results:

Iteration 1:

```
TRE_SEA_UANTS_TO_PISS_TRE_GODMEN_SROHE_TRE_SLNDIGRT_UAHWS_COLH_S
PIN_ADD_TRE_BEALTC_TRAT_IS_BEEN_DOST_BEYOHE_UANTS_TO_YINM_LS_AGA
IN_I_FANNOT_YIGRT_COL_ANCWOHE_IT_IS_COL_I_AW_YIGRTING_YOH_TRE_SE
A_TRHOUS_HOFP_TOGETREH_BLT_TIWE_DEAVES_LS_KODISREM_STONES_UE_FAN
NOT_YADD_ANC_YLHTREH_IY_UE_FANNOT_YEED_OHMINAHC_DOVE_ANM_UE_FANN
OT_HEAFR_ANC_RIGREH_IY_UE_FANNOT_MEAD_UITR_OHMINAHC_DOVE_BIHMS_Y
DC_RIGR_IN_TRE_SLWWEH_SPC_ANM_HEST_ON_TRE_BHEEZE_TRE_SAWE_UINM_U
IDD_TAPE_FAHE_OY_COL_ANM_I_UE_UIDD_BLIDM_OLH_ROLSE_IN_TRE_THEES_
COLH_REAHT_IS_ON_WC_SDEEVE_MIM_COL_KLT_TREHE_UITR_A_WAGIF_WAHPEH
_YOH_CEAHS_I_UOLDM_BEDIEVE_TRAT_TRE_UOHDM_FOLDM_NOT_UASR_IT_AUAC
```

Log-plausibility: 7438.335

Accuracy: 0.5

Iteration 100:

```
TRE_SEA_BANTS_TO_PISS_TRE_GODMEN_SROYE_TRE_SLNDIGRT_BAYCS_WOLY_S
PIN_ADD_TRE_UEALTW_TRAT_IS_UEEN_DOST_UEHOYE_BANTS_TO_HINM_LS_AGA
IN_I_FANNOT_HIGRT_WOL_ANWCOYE_IT_IS_WOL_I_AC_HIGRTING_HOY_TRE_SE
A_TRYOBS_YOFP_TOGETREY_ULT_TICE_DEAVES_LS_JODISREM_STONES_BE_FAN
NOT_HADD_ANW_HLYTREY_IH_BE_FANNOT_HEED_OYMINAYW_DOVE_ANM_BE_FANN
OT_YEAFR_ANW_RIGREY_IH_BE_FANNOT_MEAD_BITR_OYMINAYW_DOVE_UIYMS_H
DW_RIGR_IN_TRE_SLCCEY_SPW_ANM_YEST_ON_TRE_UYEEZE_TRE_SACE_BINM_B
IDD_TAPE_FAYE_OH_WOL_ANM_I_BE_BIDD_ULIDM_OLY_ROLSE_IN_TRE_TYEES_
WOLY_REAYT_IS_ON_CW_SDEEVE_MIM_WOL_JLT_TREYE_BITR_A_CAGIF_CAYPEY
_HOY_WEAYS_I_BOLDM_UEDIEVE_TRAT_TRE_BOYDM_FOLDM_NOT_BASR_IT_ABAW
```

Log-plausibility: 7557.887

Accuracy: 0.4583

Iteration 200:

```
TRE_SEA_BANTS_TO_UISS_TRE_GODMEN_SROYE_TRE_SLNDIGRT_BAYCS_WOLY_S
UIN_ADD_TRE_PEALTW_TRAT_IS_PEEN_DOST_PEFOYE_BANTS_TO_FINM_LS_AGA
IN_I_HANNOT_FIGRT_WOL_ANWCOYE_IT_IS_WOL_I_AC_FIGRTING_FOY_TRE_SE
A_TRYOBS_YOHU_TOGETREY_PLT_TICE_DEAVES_LS_JODISREM_STONES_BE_HAN
NOT_FADD_ANW_FLYTREY_IF_BE_HANNOT_FEED_OYMINAYW_DOVE_ANM_BE_HANN
OT_YEAHR_ANW_RIGREY_IF_BE_HANNOT_MEAD_BITR_OYMINAYW_DOVE_PIYMS_F
DW_RIGR_IN_TRE_SLCCEY_SUW_ANM_YEST_ON_TRE_PYEEZE_TRE_SACE_BINM_B
IDD_TAUE_HAYE_OF_WOL_ANM_I_BE_BIDD_PLIDM_OLY_ROLSE_IN_TRE_TYEES_
WOLY_REAYT_IS_ON_CW_SDEEVE_MIM_WOL_JLT_TREYE_BITR_A_CAGIH_CAYUEY
_FOY_WEAYS_I_BOLDM_PEDIEVE_TRAT_TRE_BOYDM_HOLDM_NOT_BASR_IT_ABAW
```

Log-plausibility: 7576.744

Accuracy: 0.5

Iteration 500:

```
TLE_SEA_HANTS_TO_UISS_TLE_GOMDEN_SLOYE_TLE_SRNMIGLT_HAYPS_WORY_S
UIN_AMM_TLE_BEARTW_TLAT_IS_BEEN_MOST_BEFOYE_HANTS_TO_FIND_RS_AGA
IN_I_CANNOT_FIGLT_WOR_ANWPOYE_IT_IS_WOR_I_AP_FIGLTING_FOY_TLE_SE
A_TLYOHS_YOCU_TOGETLEY_BRT_TIPE_MEAVES_RS_JOMISLED_STONES_HE_CAN
NOT_FAMM_ANW_FRYTLEY_IF_HE_CANNOT_FEEM_OYDINAYW_MOVE_AND_HE_CANN
OT_YEACL_ANW_LIGLEY_IF_HE_CANNOT_DEAM_HITL_OYDINAYW_MOVE_BIYDS_F
MW_LIGL_IN_TLE_SRPPEY_SUW_AND_YEST_ON_TLE_BYEEKE_TLE_SAPE_HIND_H
IMM_TAUE_CAYE_OF_WOR_AND_I_HE_HIMM_BRIMD_ORY_LORSE_IN_TLE_TYEES_
WORY_LEAYT_IS_ON_PW_SMEEVE_DID_WOR_JRT_TLEYE_HITL_A_PAGIC_PAYUEY
_FOY_WEAYS_I_HORMD_BEMIEVE_TLAT_TLE_HOYMD_CORMD_NOT_HASL_IT_AHAW
```

Log-plausibility: 7645.48

Accuracy:0.5833

Iteration 1,000:

```
THE_SEA_PANTS_TO_KISS_THE_GOFDEN_SHORE_THE_SUNFIGHT_PARLS_WOUR_S
KIN_AFF_THE_BEAUTW_THAT_IS_BEEN_FOST_BEMORE_PANTS_TO_MIND_US_AGA
IN_I_CANNOT_MIGHT_WOU_ANWLORE_IT_IS_WOU_I_AL_MIGHTING_MOR_THE_SE
A_THROPS_ROCK_TOGETHER_BUT_TILE_FEAVES_US_JOFISHED_STONES_PE_CAN
NOT_MAFF_ANW_MURTHER_IM_PE_CANNOT_MEEF_ORDINARW_FOVE_AND_PE_CANN
OT_REACH_ANW_HIGHER_IM_PE_CANNOT_DEAF_PITH_ORDINARW_FOVE_BIRDS_M
FW_HIGH_IN_THE_SULLER_SKW_AND_REST_ON_THE_BREEYE_THE_SALE_PIND_P
IFF_TAKE_CARE_OM_WOU_AND_I_PE_PIFF_BUIFD_OUR_HOUSE_IN_THE_TREES_
WOUR_HEART_IS_ON_LW_SFEEVE_DID_WOU_JUT_THERE_PITH_A_LAGIC_LARKER
_MOR_WEARS_I_POUFD_BEFIEVE_THAT_THE_PORFD_COUFD_NOT_PASH_IT_APAW
```

Log-plausibility: 7865.221

Accuracy: 0.7083

Iteration 1,500:

```
THE_SEA_PANTS_TO_KISS_THE_GOLDEN_SHORE_THE_SUNLIGHT_PARMS_YOUR_S
KIN_ALL_THE_BEAUTY_THAT_IS_BEEN_LOST_BEFORE_PANTS_TO_FIND_US_AGA
IN_I_CANNOT_FIGHT_YOU_ANYMORE_IT_IS_YOU_I_AM_FIGHTING_FOR_THE_SE
A_THROPS_ROCK_TOGETHER_BUT_TIME_LEAVES_US_JOLISHED_STONES_PE_CAN
NOT_FALL_ANY_FURTHER_IF_PE_CANNOT_FEEL_ORDINARY_LOVE_AND_PE_CANN
OT_REACH_ANY_HIGHER_IF_PE_CANNOT_DEAL_PITH_ORDINARY_LOVE_BIRDS_F
LY_HIGH_IN_THE_SUMMER_SKY_AND_REST_ON_THE_BREEWE_THE_SAME_PIND_P
ILL_TAKE_CARE_OF_YOU_AND_I_PE_PILL_BUILD_OUR_HOUSE_IN_THE_TREES_
YOUR_HEART_IS_ON_MY_SLEEVE_DID_YOU_JUT_THERE_PITH_A_MAGIC_MARKER
_FOR_YEARS_I_POULD_BELIEVE_THAT_THE_PORLD_COULD_NOT_PASH_IT_APAY
```

Log-plausibility: 7952.558

Accuracy: 0.875

Iteration 2,000:

```
THE_SEA_WANTS_TO_KISS_THE_GOLDEN_SHORE_THE_SUNLIGHT_WARMS_YOUR_S
KIN_ALL_THE_BEAUTY_THAT_IS_BEEN_LOST_BEFORE_WANTS_TO_FIND_US_AGA
IN_I_CANNOT_FIGHT_YOU_ANYMORE_IT_IS_YOU_I_AM_FIGHTING_FOR_THE_SE
A_THROWS_ROCK_TOGETHER_BUT_TIME_LEAVES_US_POLISHED_STONES_WE_CAN
NOT_FALL_ANY_FURTHER_IF_WE_CANNOT_FEEL_ORDINARY_LOVE_AND_WE_CANN
OT_REACH_ANY_HIGHER_IF_WE_CANNOT_DEAL_WITH_ORDINARY_LOVE_BIRDS_F
LY_HIGH_IN_THE_SUMMER_SKY_AND_REST_ON_THE_BREEXE_THE_SAME_WIND_W
ILL_TAKE_CARE_OF_YOU_AND_I_WE_WILL_BUILD_OUR_HOUSE_IN_THE_TREES_
YOUR_HEART_IS_ON_MY_SLEEVE_DID_YOU_PUT_THERE_WITH_A_MAGIC_MARKER
_FOR_YEARS_I_WOULD_BELIEVE_THAT_THE_WORLD_COULD_NOT_WASH_IT_AWAY
```

Log-plausibility: 7986.475

Accuracy: 0.95833

As we can see, this algorithm is quite efficient. Choosing our first guess using frequencies of letter leads to 50 % of characters correctly matched. Indeed, the space character, the letter E, T, A, ... are correctly placed in the message. However this is not surprising given the discussion we did in Chapter 2 about frequency analysis. Furthermore, at iteration 100, even if the log-plausibility is higher than before, the efficiency regresses. Indeed, the log-plausibility is just an indicator of how English-like the text is. Furthermore, at iteration 200, the problem was fixed and we have again 50% of characters correctly matched. Moreover, at iteration 1,500, 87,5% of the symbols are correctly assigned. Furthermore, at iteration 2,000, only the letter "X" is misplaced. Indeed, the word "BREEXE" should be "BREEZE". However, the algorithm won't be able to handle the problem even if 10,000 iterations are allowed. The reason this won't be fixed is the transition matrix used. Indeed, in Austen's text there is no bigram "EZ", thus when the algorithm tries the letter Z instead of X, the corresponding log-plausibility decreases because there is no such bigram in Austen's text and the proposal is accepted with less probability.

It has to be emphasised, that only 2,000 iterations (actually less than 2,000, but we do not print all the iterations of course) are needed to decrypt the message (forgetting that one letter is misplaced), while a rough force would need to test 1.37637×10^{43} possible permutations.

Chapter 1 of Austen's text

If the first 468 characters of Austen's text are encrypted using substitution cipher and the same algorithm is applied to decrypt the corresponding ciphertext, only 2,100 iterations are needed to correctly decrypt it. However, it not so surprising because the reference matrix used in the Metropolis-Hastings algorithm was based on Austen's text.

French text

To emphasise on the importance of the used language in the message, we try to decrypt an encrypted French text using the reference matrix based on Austen's text. The French text is about the song *Ordinary Love* and the convention used in section 2.2.1 about French texts are kept:

```
ORDINARY_LOVE_EST_UNE_CHANSON_DU_GROUPE_U2_CE_SINGLE_ECRIT_EN
_LHONNEUR_DE_NELSON_MANDELA_FAIT_PARTIE_DE_LA_BANDE_ORIGINALE
_DU_FILM_MANDELA_UN_LONG_CHEMIN_VERS_LA_LIBERTE_2013_LA_CHANS
ON_REMPORTE_NOTAMMENT_LE_GOLDEN_GLOBE_DE_LA_MEILLEURE_CHANSON
_ORIGINALE_A_LA_71EME_CEREMONIE_DES_GOLDENS_GLOBES_ELLE_EST_N
OMMEE_A_LOSCAR_DE_LA_MEILLEUR_CHANSON_ORIGINALE_LORS_DE_LA_86
_EME_CEREMONIE_DES_OSCARS_LA_POCHETTE_DU_SINGLE_ORDINARY_LOVE
_EST_UNE_PEINTURE_DE_NELSON_MANDELA_FAITE_PAR_OLIVIER_JEFFERS
```

This is the encrypted text:

```
RYB59LYFAWR3PAPGMAX9PAT_L9GR9ABXAZYRXOPAXUATPAG59ZWPAPTY5MAP9
AW_R99PXYABPA9PWGR9AHL9BPWLA6L5MAOLYM5PABPAWLAJL9BPARY5Z59LWP
ABXA65WHAHL9BPWLAX9AWR9ZAT_PH59A3PYGAWLAW5JPYMPAUVN2AWLAT_L9G
R9AYPHORYMPA9RMLHHP9MAWPAZRWBP9AZWRJPABPAWLAHP5WWPXYPAT_L9GR9
ARY5Z59LWPALAWLASNPHPATPYPHR95PABPGAZRWBP9GAZWRJPGAPWWPAPGMA9
RHHPPALAWRGTLYABPAWLAHP5WWPXYAT_L9GR9ARY5Z59LWPAWRYGABPAWLA87
APHPATPYPHR95PABPGARGTLYGAWLAORT_PMMPABXAG59ZWPARYB59LYFAWR3P
APGMAX9PAOP59MXYPABPA9PWGR9AHL9BPWLA6L5MPAOLYARW535PYAEP66PYG
```

The algorithm was run 10,000 times and these are the results:

Iteration 1:

```
INRHTONP_AIBE_ESL_MTE_UYOTSIT_RM_CNIMWE_MK_UE_SHTCAE_EUNHL_ET
_AYITTEMN_RE_TEASIT_DOTREAO_FOHL_WONLHE_RE_AO_GOTRE_INHCHTOAE
_RM_FHAD_DOTREAO_MT_AITC_UYEDHT_BENS_AO_AHGENLE_KXVQ_AO_UYOTS
IT_NEDWINLE_TILODDETL_AE_CIARET_CAIGE_RE_AO_DEHAAEMNE_UYOTSIT
_INHCHTOAE_O_AO_JVEDE_UENEDITHE_RES_CIARETS_CAIGES_EAAE_ESL_T
IDDEE_O_AISUON_RE_AO_DEHAAEMN_UYOTSIT_INHCHTOAE_AINS_RE_AO_21
_EDE_UENEDITHE_RES_ISUONS_AO_WIUYELLE_RM_SHTCAE_INRHTONP_AIBE
_ESL_MTE_WEHTLMNE_RE_TEASIT_DOTREAO_FOHLE_WON_IAHBHEN_ZEFFENS
```

Log-plausibility: 6161.484

Accuracy: 0.138

Iteration 100:

```
IARNTOAP_HIBE_ESL_FTE_UYOTSIT_RF_GAIFWE_FK_UE_SNTGHE_EUANL_ET
_HYITTEFA_RE_TEHSIT_DOTREHO_CONL_WOALNE_RE_HO_MOTRE_IANGNTOHE
_RF_CNHD_DOTREHO_FT_HITG_UYEDNT_BEAS_HO_HNMEALE_K2VX_HO_UYOTS
IT_AEDWIALE_TILODDETL_HE_GIHRET_GHIME_RE_HO_DENHHEFAE_UYOTSIT
_IANGNTOHE_O_HO_JVEDE_UEAEDITNE_RES_GIHRETS_GHIMES_EHHE_ESL_T
IDDEE_O_HISUOA_RE_HO_DENHHEFA_UYOTSIT_IANGNTOHE_HIAS_RE_HO_43
_EDE_UEAEDITNE_RES_ISUOAS_HO_WIUYELLE_RF_SNTGHE_IARNTOAP_HIBE
_ESL_FTE_WENTLFAE_RE_TEHSIT_DOTREHO_CONLE_WOA_IHNBNEA_ZECCEAS
```

Log-plausibility: 6243.801

Accuracy: 0.138

Iteration 1,000:

```
ORISTARU_HOKE_END_FTE_GBATNOT_IF_WROFME_FY_GE_NSTWHE_EGRSD_ET
_HBOTTEFR_IE_TEHNOT_LATIEHA_CASD_MARDSE_IE_HA_PATIE_ORSWSTAHE
_IF_CSHL_LATIEHA_FT_HOTW_GBELST_KERN_HA_HSPERDE_Y9Z4_HA_GBATN
OT_RELMORDE_TODALLETD_HE_WOHIET_WHOPE_IE_HA_LESHHEFRE_GBATNOT
_ORSWSTAHE_A_HA_QZELE_GERELOTSE_IEN_WOHIETN_WHOPEN_EHHE_END_T
OLLEE_A_HONGAR_IE_HA_LESHHEFR_GBATNOT_ORSWSTAHE_HORN_IE_HA_JO
_ELE_GERELOTSE_IEN_ONGARN_HA_MOGBEDDE_IF_NSTWHE_ORISTARU_HOKE
_END_FTE_MESTDFRE_IE_TEHNOT_LATIEHA_CASDE_MAR_OHSKSER_VECCERN
```

Log-plausibility: 6397.845

Accuracy: 0.1724

Iteration 5,000:

```
ARHITORY_NAGE_ESD_UTE_MPOTSAT_HU_WRAUCE_UF_ME_SITWNE_EMRID_ET
_NPATTEUR_HE_TENSAT_LOTHENO_BOID_CORDIE_HE_NO_VOTHE_ARIWITONE
_HU_BINL_LOTHENO_UT_NATW_MPELIT_GERS_NO_NIVERDE_F2ZX_NO_MPOTS
AT_RELCARDE_TADOLLETD_NE_WANHET_WNAVE_HE_NO_LEINNEURE_MPOTSAT
_ARIWITONE_O_NO_3ZELE_MERELATIE_HES_WANHETS_WNAVES_ENNE_ESD_T
ALLEE_O_NASMOR_HE_NO_LEINNEUR_MPOTSAT_ARIWITONE_NARS_HE_NO_J8
_ELE_MERELATIE_HES_ASMORS_NO_CAMPEDDE_HU_SITWNE_ARHITORY_NAGE
_ESD_UTE_CEITDURE_HE_TENSAT_LOTHENO_BOIDE_COR_ANIGIER_KEBBERS
```

Log-plausibility: 6514.888

Accuracy: 0.2414

Iteration 10,000:

```
ARHINORY_TAVE_EDS_UNE_MPONDAN_HU_6RAUWE_UG_ME_DIN6TE_EMRIS_EN
_TPANNEUR_HE_NETDAN_LONHETO_FOIS_WORSIE_HE_TO_BONHE_ARI6INOTE
_HU_FITL_LONHETO_UN_TAN6_MPELIN_VERD_TO_TIBERSE_G5XK_TO_MPOND
AN_RELWARSE_NASOLLENS_TE_6ATHEN_6TABE_HE_TO_LEITTEURE_MPONDAN
_ARI6INOTE_O_TO_2XELE_MERELANIE_HED_6ATHEND_6TABED_ETTE_EDS_N
ALLEE_O_TADMOR_HE_TO_LEITTEUR_MPONDAN_ARI6INOTE_TARD_HE_TO_Q1
_ELE_MERELANIE_HED_ADMORD_TO_WAMPESSE_HU_DIN6TE_ARHINORY_TAVE
_EDS_UNE_WEINSURE_HE_NETDAN_LONHETO_FOISE_WOR_ATIVIER_JEFFERD
```

Log-plausibility:6444.355

Accuracy: 0.3793

As naturally expected, apart from the space character and the E and some common letters that are correctly matched, all the proposal plaintexts are gibberish. This is not surprising that some letters are correctly matched because French language shares similarities with English language for the most common letters as showed in Chapter 2. Furthermore the algorithm won't be able to match more letters than 37%. Even at the last iteration (iteration 10,000), the proposal plaintext doesn't mean anything in French. Another thing to notice is that the log-plausibilities are lower than they were for the song *Ordinary Love*; it shows that the text seems less English.

4.5 Adaptation of the Metropolis Algorithm for decrypting a permutation/transposition ciphertext

For transposition cipher, given a period of m, the goal is to find the permutation of $\{1, \ldots, m\}$ that was used. The adaptation is quite similar to the case of a transposition ciphertext. One might think that choosing a random permutation of $\{1, \ldots, m\}$ and making proposals by randomly swapping two elements of the permutation (as we did for substitution cipher) will be sufficient to decrypt the text. However, we tested this method and it was unsuccessful. We therefore follow the idea of slide moves (see section 4.5.2) introduced in [4].

4.5.1 Plausibility function

Given a possible permutation $\hat{\pi}$, its plausibility function is the same as the one defined in section 4.4.1, i.e.,

$$Pl(\hat{\pi}) = \prod_{i,j} (M_{ij})^{\hat{M}_{ij}}$$

where $M(i,j)$ records the number of occurrences of the pair (i,j) in the reference text and $\hat{M}(i,j)$ denotes the number of times the pair (i,j) appears in the decrypted text using permutation $\hat{\pi}$.

4.5.2 Slide moves

Suppose that the period of the cipher is m and that the current permutation is $\hat{\pi}$. The idea is to select a random contiguous sequence of decryption positions and to insert it in another

position. More formally, the new proposal is to take a block of n decryption positions from position k_1 and to move it k_2 positions further. If needed, the operation is cyclical modulo m.

Examples 4.5.2.1. Suppose that the period is known to be 13 and the current proposal key is the identity permutation, i.e.,

$$\hat{\pi} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix},$$

which will be simply denoted by its second row (0 1 2 3 4 5 6 7 8 9 10 11 12) for these examples.

If $n = 2$, $k_1 = 3$ and $k_2 = 4$, the new proposal will be (0 1 4 5 6 7 2 3 8 9 10 11 12). The block "2 3" corresponding to two elements starting from the third element, is moved 4 times to the right.

If $n = 3$, $k_1 = 5$ and $k_2 = 8$, the new proposal will be (0 1 4 5 6 2 3 7 8 9 10 11 12). The block "4 5 6" corresponding to three elements starting from position 5, is moved 8 positions further using cyclical modulo m operations.

The reasons for introducing these new moves are the better proposals they allow. Indeed, with swap moves, changing a decryption key which gives a decrypted text "UNLIGHTS" to the correct decryption requires at least 7 swap moves. With slide moves, it only requires one move. Furthermore, suppose we encrypt our chosen text *Ordinary Love* using transposition cipher and a period of 13 and suppose that the first thirteen letters of the encrypted message are decrypted as `THE_WANTS_SEA` instead of `THE_SEA_WANTS`. Using slides moves, the part "WANTS_" can be moved to the left of "SEA" in just one move.

4.5.3 Algorithm

The algorithm runs as follows:

- Start with an initial guess $\hat{\pi}$ for the decoding permutation;
- Compute the plausibility of $\hat{\pi}$ on the encrypted text: $Pl(\hat{\pi})$.
- Repeat the following steps for a sufficient number of iterations:
  - Randomly select contiguous sequence of decryption positions and insert it at another position; the new permutation is denoted $\hat{\pi}'$;
  - Compute the plausibility of $\hat{\pi}'$: $Pl(\hat{\pi}')$.
    - ➔ If $Pl(\hat{\pi}') > Pl(\hat{\pi})$, keep $\hat{\pi}'$
    - ➔ Else, keep $\hat{\pi}'$ with probability $\dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$ and keep $\hat{\pi}$ with probability $1 - \dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$.

The transition from $\hat{\pi}$ to $\hat{\pi}'$ is made by randomly selecting a contiguous sequence of decryption positions and inserting it at another position, so the proposal is symmetric, hence the simplification for the acceptance probability.

Furthermore, as previously mentioned for substitution cipher, a new proposal $\hat{\pi}'$ with smaller plausibility than $\hat{\pi}$ can still be accepted with probability $\dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$ in order to escape a local maximum.

Given a reference text, we construct the matrix of bigrams. Then, the algorithm is:

Algorithm 3

$t = 0$
$X_0 = \hat{\pi}_0$
for $t = 1$ to T **do**
    Given $X_t = \hat{\pi}$
    Generate $n \sim U(\{1, 2, \ldots, m-1\})$
    $k_1 \sim U(\{1, 2, \ldots, m-n+1\})$
    $k_2 \sim U(\{1, 2, \ldots, m-n\})$
    Generate $\hat{\pi}'$ by slide moves, in the last key $\hat{\pi}$, a block of n decryption positions from position k_1, k_2 positions further
    Compute $\rho(\hat{\pi}, \hat{\pi}') = \dfrac{Pl(\hat{\pi}')}{Pl(\hat{\pi})}$
    Generate $u \sim U([0, 1])$
    if $u \leq \rho(\hat{\pi}, \hat{\pi}')$ **then**
        $X_{t+1} = \hat{\pi}'$
    else
        $X_{t+1} = \hat{\pi}$
    end if
end for

Chen and Rosenthal also proved that the above algorithm will converge to solutions with maximal plausibility functions. It is still based on the irreducibility and the aperiodicity of the created chain. Let $\{X_t : t = 0, \ldots, T\}$ denote the sequence of decryption keys produced by the algorithm using the plausibility function. The chain is irreducible because every permutation can be obtained from any other by a series of slide moves and every proposed slide move has a positive probability to be accepted since $Pl(\pi) > 0$ (we added one to elements of the matrix M) for all decryption keys π. Furthermore, since the proposal of keeping the same decryption key is allowed (given n and k_1, taking $k_2 = m - n$ leads to the same decryption key), the Markov chain is aperiodic.

4.5.4 Implementation

There are three parts in the R code provided in Appendix A.

The first part consists of the construction of the matrix of bigrams based on Jane Austen's text.

The second part is the encryption process of a transposition cipher as presented in section 1.3.2. However, in this chapter, we consider an alphabet of 37 elements, as said earlier.

The final part of the R code is the decryption using the Metropolis-Hastings algorithm. The starting decryption is the identity permutation, which leads the decrypted text to be the same as the ciphertext. Furthermore, the plausibility function is implemented in the same way as for the substitution cipher.

4.5.5 Results

The results obtained when running the algorithm are presented in this section. We only try to decrypt the chosen text corresponding to the song *Ordinary love*.

The accuracy is based on the proportion of letters correctly positioned in one period. The following definitions are from Chen and Rosenthal (see [4]). "A letter is said to be *correctly positioned* if it has the same neighbours in the decrypted text as in the plaintext. We do not count the letters in the start and end positions as they only have one neighbour." The accuracy is defined as $\frac{m_t}{n_t}$ where m_t is the number of letters correctly placed and n_t is the period of the cipher minus 2 (because the letters in the start and end positions are not taken into account).

The choice of this definition will be explained in Chapter 5.

Ordinary Love

When we encipher the chosen text *Ordinary love* with the permutation cipher, we find:

```
WTH__AEAESSTN_ET_SSITO_K_HHEODS_NOL_EGRITH_LNUGE_STHUSAMOY_RRKSW_
HBNAT_LE_ELI_TSUYAHT_T__AITEENSOL_EF_BBSOR_TNA_E_WOTAIID_SUGNN_FA
TIICONN__GA_FNOTYA_UY_ROHMY__T_SIOII_EUTGA_HGIIM_F_NS_O__EHERTTFA
CTRWOR_KOOSH_UTEHB_RTTIEG_SSELEVA___EMU_OOIDEHSLNSPTNTSWAC_N__EEO
_RALYNAFLT_FUEAE_W_F_RNIHC_DO_LEEOTIFNRENAYVOL_RD_NAORW_NNATEEC__
HHC__YNIHEAAGCN_F_EWAIO_RNI__EW_LTDOATHLEDN_YROI_ARVYIIDLF__RGSBH
_M_NEHTSIM_HUNRRSA_YD_EKE_ERTOHT___ENSBSEZ__EHAE_TEMLAIDLIW_NK_WT
FO_AO_E_CUREYEIADW_I_NL__WO__U_DLUBHILRT_UE_NIHST_OEREESUOY_EA_RH
__TINO_M_SSRYDOEVID__EUELYRWPTEHTEUI___IMHAGAMC_A_T__AKRROFYER_RE
DE__LUO_ILWSBTHEEAHT_VE_ITODWRC_DUO_L_L__O_HSAITAWNT7AAMQCUOY6KWN
```

The algorithm was run 20,000 times and these are the results:

Iteration 1:

```
WTH__AEAESSTN_ET_SSITO_K_HHEODS_NOL_EGRITH_LNUGE_STHUSAMOY_RRKSW_
HBNAT_LE_ELI_TSUYAHT_T__AITEENSOL_EF_BBSOR_TNA_E_WOTAIID_SUGNN_FA
TIICONN__GA_FNOTYA_UY_ROHMY__T_SIOII_EUTGA_HGIIM_F_NS_O__EHERTTFA
CTRWOR_KOOSH_UTEHB_RTTIEG_SSELEVA___EMU_OOIDEHSLNSPTNTSWAC_N__EEO
_RALYNAFLT_FUEAE_W_F_RNIHC_DO_LEEOTIFNRENAYVOL_RD_NAORW_NNATEEC__
HHC__YNIHEAAGCN_F_EWAIO_RNI__EW_LTDOATHLEDN_YROI_ARVYIIDLF__RGSBH
_M_NEHTSIM_HUNRRSA_YD_EKE_ERTOHT___ENSBSEZ__EHAE_TEMLAIDLIW_NK_WT
FO_AO_E_CUREYEIADW_I_NL__WO__U_DLUBHILRT_UE_NIHST_OEREESUOY_EA_RH
__TINO_M_SSRYDOEVID__EUELYRWPTEHTEUI___IMHAGAMC_A_T__AKRROFYER_RE
DE__LUO_ILWSBTHEEAHT_VE_ITODWRC_DUO_L_L__O_HSAITAWNT7AAMQCUOY6KWN
```

Log-Plausibility: 6873.838

Accuracy: 0

Iteration 100:

```
TTE_HANSSEA_WE_O_TTH_KISS_EGLDOOR_EN_SHTTE_HGH_SUNLISWRMAR_KS_YOU
BI_ANE_ELL_THSATYU_I__THATEBENE_BF_LOSTOOE_R_T_WANTSIFNDIGAN_US_A
I__CI_FGANNOTOH_YTYMROU_AN_EIT_OUI_IS_YG_M_AIN_FIGHT_FR_OEATTHE_S
THOWRK_OS_ROCTGTHET_IER_BUSM_LE_U_EAVESOPLIOSTNSHED_TE_WSNO_E_CAN
RFLLAFUT_ANY_AHR_E_CNIF_WEDNT_OORIFEEL_NNRYA_AD_LOVER_E_WT_ECANNO
HAH_CIGEANY_HNRIF_ANO_WE_C_TDE_THOAL_WIERINDOV_ARY_LIBRDI_HGS_FLY
MHIN_SUM_THE_RE_SRD_EKY_ANRS_OT_BEN_THEEEE_ZAM_THE_SAWNDI_TK_WILL
OECA__YURE_OFI_NDA_WL_I_WE_LBU_URHILD_O_OSEUHET_IN_TERESE_HA_YOUR
_R_ITMYSS_ON_OLEVE_YUE_DIDW_UTPE_I_THERMT_AHC_A_MAGIARERKYER_FOR_
ESI___BLWOULDHIVEE_TE_THATD_ORWUL_LD_CO_NT_OITAWASH_AWYMAON6KUCQ7
```

Log-Plausibility: 7507.119

Accuracy: 0.2727

Iteration 200:

```
TSTHEA_WANE_SE__TISS_THO_KE_GON_SHORLDET_THUNLIGHE_SSKWA_YOUR_RMS
BEINL_THE__ALS_AUTHAT_ITY_EFBELOST_BEN_O_ORANTS_TE_WINFIUS_AGAND_
IG_INNOT_F_CAORHTU_ANYM_YO_IE_IS_YOUIT_G__AIGHTINM_F_TFOHE_SEAR_T
TOHR_ROCK_OWSTIGER_BUT_THES_MEAVES_U_LEONPOHED_STLIST_ES_CANNO_WE
RTFAANY_FULL_ANHEF_WE_CR_IDINOEEL_ORT_FNDNALOVE_ARY_RE_WANNOT_E_C
HEACNY_HIGH_ANOR_WE_CANIF__OT_L_WITHDEAE_RDRY_LOVINAIGBI_FLY_HRDS
MMH_THE_SUIN_REERY_AND__SKREST_THE_B_ONE_EZHE_SAME_TAKWIWILL_TND_
OUE_E_OF_YCARIL_AI_WE_WND__HL_LD_OURBUI_TOUIN_THESE_EAREYOUR_HES_
_SRT_ON_MY_ISOULE_DID_YEVEWI_PTHERE_UT_MATHMAGIC__A_ARRKFOR_YEER_
ELS_OULD_BI_WHEIETHAT_TVE_D__WD_COULORL_ANOASH_ITT_WA6WAUCQ7ONYMK
```

Log-Plausibility: 7665.866

Accuracy: 0.4545

Iteration 500:

```
NTSHE_SEA_WATHE_TO_KISS_T_RE_OLDEN_SHOGHT_HE_SUNLIGT_SKARMS_YOURW
_BEN_ALL_THEIIS_UTY_THAT_ABEFEEN_LOST_BTO_RE_WANTS_OAININD_US_AGF
FIGI_CANNOT__MORT_YOU_ANYHU_I_IT_IS_YOENG_AM_FIGHTI_A_TOR_THE_SEF
_TOROWS_ROCKH_TIETHER_BUTGUS_E_LEAVES_MTONOLISHED_SPOT_S_WE_CANNE
URTALL_ANY_FFCANER_IF_WE_HRDIOT_FEEL_ONANDARY_LOVE_N_REWE_CANNOT_
GHECH_ANY_HIANNO_IF_WE_CARH_O_DEAL_WITTVE_DINARY_LORHIGIRDS_FLY_B
UMM_IN_THE_SH_RER_SKY_ANDEBRET_ON_THE_SME_ZE_THE_SAETAKIND_WILL_W
YOU_CARE_OF_EWILAND_I_WE__R_H_BUILD_OULE_TUSE_IN_THOHEAEES_YOUR_R
Y_ST_IS_ON_MRYOUEEVE_DID_L_WIPUT_THERE__MAH_A_MAGICTEARKER_FOR_YR
BEL_I_WOULD_STHEEVE_THAT_ILD_WORLD_COU_T_AOT_WASH_INNA6AYMKUCQ7OW
```

Log-plausibility: 7787.988

Accuracy: 0.7273

Iteration 1,000:

```
E_SEA_WANTSHTO_KISS_THE_T_LDEN_SHORE_OGE_SUNLIGHT_HTRMS_YOUR_SKAW
_ALL_THE_BENITY_THAT_IS_UAEN_LOST_BEFEBE_WANTS_TO_ROND_US_AGAINIF
_CANNOT_FIGI__YOU_ANYMORTHIT_IS_YOU_I_EM_FIGHTING_A_R_THE_SEA_TOF
OWS_ROCK_TORHTHER_BUT_TIEG_LEAVES_US_EMLISHED_STONOP_WE_CANNOT_SE
LL_ANY_FURTAFR_IF_WE_CANEHT_FEEL_ORDIONRY_LOVE_ANDANE_CANNOT_REW_
H_ANY_HIGHECAIF_WE_CANNO_RDEAL_WITH_O_TINARY_LOVE_DRRDS_FLY_HIGIB
IN_THE_SUMM_H_SKY_AND_RERE_ON_THE_BRETSE_THE_SAME_ZEND_WILL_TAKIW
CARE_OF_YOU_END_I_WE_WILA_BUILD_OUR_H_LSE_IN_THE_TUOES_YOUR_HEAER
_IS_ON_MY_STREVE_DID_YOUELUT_THERE_WIP__A_MAGIC_MAHTER_FOR_YEARKR
I_WOULD_BEL_SVE_THAT_THEEIORLD_COULD_W_T_WASH_IT_AONYMKUCQ70NA6AW
```

Log-plausibility: 7823.08

Accuracy: 0.8182

Iteration 8,409:

```
THE_SEA_WANTS_TO_KISS_THE_GOLDEN_SHORE_THE_SUNLIGHT_WARMS_YOUR_SK
IN_ALL_THE_BEAUTY_THAT_IS_BEEN_LOST_BEFORE_WANTS_TO_FIND_US_AGAIN
_I_CANNOT_FIGHT_YOU_ANYMORE_IT_IS_YOU_I_AM_FIGHTING_FOR_THE_SEA_T
HROWS_ROCK_TOGETHER_BUT_TIME_LEAVES_US_POLISHED_STONES_WE_CANNOT_
FALL_ANY_FURTHER_IF_WE_CANNOT_FEEL_ORDINARY_LOVE_AND_WE_CANNOT_RE
ACH_ANY_HIGHER_IF_WE_CANNOT_DEAL_WITH_ORDINARY_LOVE_BIRDS_FLY_HIG
H_IN_THE_SUMMER_SKY_AND_REST_ON_THE_BREEZE_THE_SAME_WIND_WILL_TAK
E_CARE_OF_YOU_AND_I_WE_WILL_BUILD_OUR_HOUSE_IN_THE_TREES_YOUR_HEA
RT_IS_ON_MY_SLEEVE_DID_YOU_PUT_THERE_WITH_A_MAGIC_MARKER_FOR_YEAR
S_I_WOULD_BELIEVE_THAT_THE_WORLD_COULD_NOT_WASH_IT_AWAYMKUCQ70NA6
```

Log-plausibility: 8002.556

Accuracy: 1

As we can see, this algorithm is fairly efficient. Although there are 6,227,020,800 possible permutations when using a period of 13, it requires only 8,409 iterations to find the correct decryption key.

Although we started with the identity permutation and a corresponding accuracy of 0, after 100 iterations, more than 27% of the letters are correctly positioned and after 200 iterations, more than 45% are correctly positioned. At iteration 500, almost three quarters of the letters are correctly placed. At iteration 1,000, 81 % of the letters are correctly placed and this doesn't change until iteration 8,409, where the text is correctly decrypted. The log-plausibility corresponding to iteration 1,000 is so "high" that any proposal made by the algorithm is rejected. It is only after 7,409 iterations that it is eventually accepted. Notice that the last ten letters of the decrypted text at iteration 8,409 are gibberish because they correspond to random letters added when the chosen text was encrypted with a period of 13.

4.5.6 Finding the unknown period of the transposition cipher

In section 2.3.2, we said that Markov Chain Monte Carlo would help to find the period. Indeed, the algorithm introduced by Chen and Rosenthal is also useful for that particular problem. If for each period m, the entire algorithm is run a number of times and the plausibility

of the decryption key that is returned by our algorithm is memorized as Pl_m, then the period is the value of m that leads to the highest plausibility, i.e. $\mathrm{argmax}_m \mathrm{Pl}_m$. Indeed, the highest plausibility a text has, the more similar it is to English plaintext.

There are two ways of proceeding. We can run the algorithm 20,000 times (say) for each possible period and we find the correct period and the decryption key that was probably used (the key with the highest plausibility for that given period). However, this requires a lot of iterations even for incorrect periods. Instead, we can run the algorithm a smaller number of times (say 100), as suggested in [4], and we find the most probable period and after that, for this period we run the algorithm 20,000 times. This is the latest method we chose because with 100 iterations, the implemented algorithm gave the correct period each time it was run.

Consider the text encrypted with a period of 13 in the previous section. However, suppose that only the ciphertext is available and that the period is unknown.

The algorithm was run 100 times for possible periods between 3 and 20. The results are presented in Figure 4.6. As we can see, the period with the highest log-plausibility is the period that was used to encrypt the text.

Period	Log-plausibility	Period	Log-plausibility
3	7007.393	12	7027.081
4	7055.353	**13**	**7641.886**
5	6965.011	14	7090.355
6	7072.576	15	7062.406
7	7028.015	16	7101.992
8	7105.245	17	7078.943
9	7091.891	18	7084.924
10	7058.260	19	7119.204
11	7102.140	20	7109.103

Figure 4.6: Log-plausibility of the returned decryption key for each tested period

Chapter 5

Comparisons based on simulations

In this chapter, we will measure the performance of some techniques presented in the previous chapters based on simulations. The presentation of the results will be based on [4]. To the best of our knowledge, a few articles present the comparison of such techniques. We will therefore try to compare them using systematically statistical tools.

First, we will compare the performance of the three techniques presented to break a Vigenère cipher: the combination of frequency analysis and an algebraic approach (see section 2.4.1), the statistical method for frequency analysis based on the Chi-squared statistic (see section 2.4.1) and Turing's method (see section 3.2).

Then we will measure the performance of the Markov Chain Monte Carlo's methods presented in Chapter 4.

We will also measure the computing time of our runs.

The program, written in R software (see Appendix A), will be run on an Asus VivoBook with the following system configuration:

Processor	Intel(R) Core(TM) i7-8550 CPU @ 1.80GHz 1.99 GHz
Memory	8.00 Go
OS version	Windows 10 1803 17134.706
Compiler	R i386 3.5.1

Table 5.1: System configuration of the machine running the attacks.

5.1 Data generation

To test the accuracy of the algorithms presented in the previous chapter, we use the five following texts[1]: Austen's *Pride and Prejudice* (1813), the English Constitution by Walter Bagehot (1865), Shakespeare's *Romeo and Juliet* (1594), Lewis Carroll's *Alice's Adventures in Wonderland* (1865) and Thomas Longueville's *The Curious Case of Lady Purbeck* (1909).

For Vigenère cipher, the alphabet consists of the 26 letters from A to Z and the space character (alphabet of 27 elements) and for substitution and transposition ciphers, we add numbers as we did in Chapter 4 (alphabet of 37 elements).

[1] Available in the project Gutenberg (gutenberg.org).

Out of all these texts[2], we select a random portion of consecutive characters that will be the message to encrypt. In this chapter, we consider messages of length 50, 100, 200, 500 and 1,000 as suggested in [4].

In order to randomly encrypt the selected text using the different ciphers considered in this chapter, we proceed as follows:

- for Vigenère cipher, we choose an arbitrary number m between 1 and 15 that will be the period used to encrypt the text. Then we choose an arbitrary[3] m-character keyword and encrypt the selected text using this keyword.
- for substitution cipher, we choose a random permutation of the alphabet of 37 elements and we encrypt the text using this permutation;
- for transposition cipher, we choose a random number m from 2 to 15 that will be period used to encrypt the text. Then, we choose a random permutation of the set $\{1, \ldots, m\}$ and we encrypt the text using this permutation.

5.2 Performance measures of the techniques

In order to measure the performance of these techniques, we repeat the random selection, the encryption and the decryption processes 100 times. This choice is based on [4].

We keep the choice made in Chapter 4 for the number of iterations of the Metropolis-Hastings algorithm: 10,000 iterations for substitution cipher and 20,000 iterations for transposition cipher. It was suggested in several references including [4] and [20]. The algorithm returns whichever decryption key from whichever iteration which gave the largest log-plausibility. However, as suggested in [4] and in [20], the best would be to run the algorithm several times (say 3 times) and to return whichever decryption key from whichever iteration from whichever run which gave the largest log-plausibility. We do not follow this suggestion as it would require larger computing time[4] but we know that we should. Indeed, this proposal would increase the performance of the Metropolis-Hastings algorithm which should not stay into a local maximum.

For all the techniques, a *successful run* is a run of the algorithm where the ciphertext is correctly decrypted.

Vigenère cipher

The accuracy is naturally defined as $\frac{n}{m}$ where n is the number of letters of the keyword that are correctly decrypted and m is the period of the cipher, i.e., the length of the keyword.

We add the following definition: a decryption whose corresponding accuracy is lower than 0.5 is said to be *bad*.

[2]Representing respectively 1,657,101 characters when numbers are not taken into account and 1,659,011 characters when they are.

[3]Each letter of the keyword is randomly chosen from the alphabet of 27 elements.

[4]When we did the simulations for the Metropolis-Hastings algorithm without the suggestion, the computer ran during approximately 66 hours for substitution cipher and 18 hours for transposition cipher.

Substitution cipher

A first natural definition for the accuracy of a decryption could be the proportion of letters correctly decrypted, i.e., $\frac{n}{37}$ where n is the number of letters that are correctly decrypted. However, this definition would take into account letters that are not in the plaintext. Thus, a letter incorrectly decrypted would diminish the accuracy of the decryption even if it doesn't appear in the plaintext. For example, suppose that the original plaintext that was enciphered does not contain the letter J and the letter Z. If the the permutation found with the Metropolis-Hastings algorithm is correct except for the 10^{th} and the 26^{th} elements (corresponding to letters J and Z), the accuracy would be $\frac{35}{37} \simeq 0.9459$ even if the entire text is correctly decrypted.

Thus, we take the definition from Chen and Rosenthal (see [4]) where the accuracy is based on the number of letters correctly revealed. A letter is *correctly revealed* if its first occurrence in the decrypted text is the same as in the plaintext. Then, the *accuracy* is defined as $\frac{m_s}{n_s}$ where m_s is the number of letters correctly revealed and n_s is the number of different letters in the plaintext (which may not be 37).

We could have taken a definition in order to weight the value of the letters depending on their frequency within the plaintext. However, in this case, the accuracy would depend on the plaintext.

We add the following definition: a decryption whose corresponding accuracy is lower than 0.5 is said to be *bad*.

Transposition cipher

If the period of two is used, the encrypted text is either encrypted, either the plaintext itself. Thus, we define the efficiency for a period of 2 to be 1 if the text is correctly decrypted and to be 0 if not.

For other periods, the accuracy of a decryption could be defined as $\frac{n}{m}$ where n is the number of elements of the permutation that are correctly found and m is the period of the cipher. However, this definition would not be satisfactory. Indeed, suppose that the period of the cipher is 7 and a string of plaintext is `THE_SEA`. If the algorithm decrypts the corresponding encrypted string as `ATS_EHE`, the corresponding accuracy would be $\frac{1}{7} \simeq 0.0.1429$. However, the decrypted string `SEA_THE` yields the same accuracy although it is clearly better than the proposal `ATS_EHE`.

Thus, we follow Chen and Rosenthal's definition of accuracy based on the proportion of letters correctly positioned in one period. "A letter is said to be *correctly positioned* if it has the same neighbours in the decrypted text as in the plaintext. We do not count the letters in the start and end positions as they only have one neighbour." The *accuracy* is then defined as $\frac{m_t}{n_t}$ where m_t is the number of letters correctly positioned and n_t is the period of the cipher minus 2 (because the letters in the start and end positions are not taken into account).

We add the following definition: a decryption whose corresponding accuracy is lower than 0.5 is said to be *bad*.

5.3 Simulations

5.3.1 Vigenère cipher

In order to compare the three techniques, we compare the efficiency of these techniques for different lengths of ciphertext and the time spent on the decryption. Tables 5.2 to 5.6 and Figures 5.1 to 5.8 present the results for different lengths of text.

	Average Accuracy	Standard Error	No. of successful runs (/100)	Average Duration (in seconds)
Freq. analysis	0.3868	0.0232	9	0.00619
Chi-squared	0.624	0.0290	30	1.68322
Turing	0.6724	0.0273	32	0.48018

Table 5.2: Results of attacks on Vigenère cipher with a length of text of 50.

	Average Accuracy	Standard Error	No. of successful runs (/100)	Average Duration (in seconds)
Freq. analysis	0.5388	0.0257	20	0.00904
Chi-squared	0.7958	0.0200	40	3.36159
Turing	0.8494	0.0169	50	0.58891

Table 5.3: Results of attacks on Vigenère cipher with a length of text of 100.

	Average Accuracy	Standard Error	No. of successful runs (/100)	Average Duration (in seconds)
Freq. analysis	0.6685	0.0217	20	0.00529
Chi-squared	0.9347	0.0093	59	6.61961
Turing	0.9792	0.0043	79	0.82294

Table 5.4: Results of attacks on Vigenère cipher with a length of text of 200.

	Average Accuracy	Standard Error	No. of successful runs (/100)	Average Duration (in seconds)
Freq. analysis	0.8263	0.0155	30	0.00344
Chi-squared	1	0	100	22.89028
Turing	1	0	100	1.41674

Table 5.5: Results of attacks on Vigenère cipher with a length of text of 500.

	Average Accuracy	Standard Error	No. of successful runs (/100)	Average Duration (in seconds)
Freq. analysis	0.9219	0.0095	53	0.00144
Chi-squared	1	0	100	58.51794
Turing	1	0	100	2.61304

Table 5.6: Results of attacks on Vigenère cipher with a length of text of 1,000.

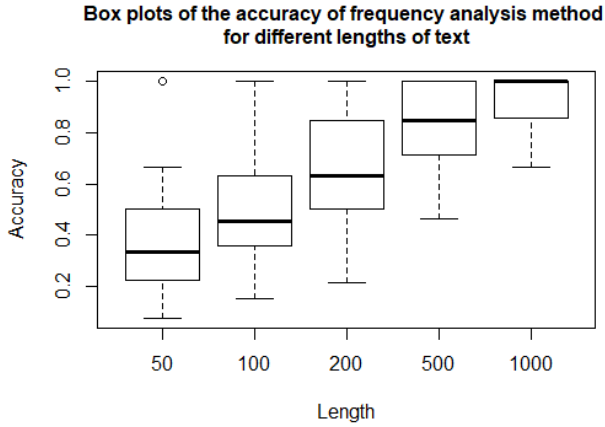


Figure 5.1: Box plots of the accuracy of decryption for freq. analysis method

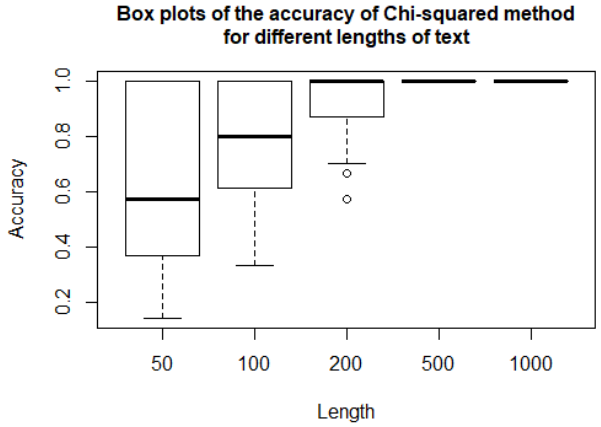


Figure 5.2: Box plots of the accuracy of decryption for Chi-squared method

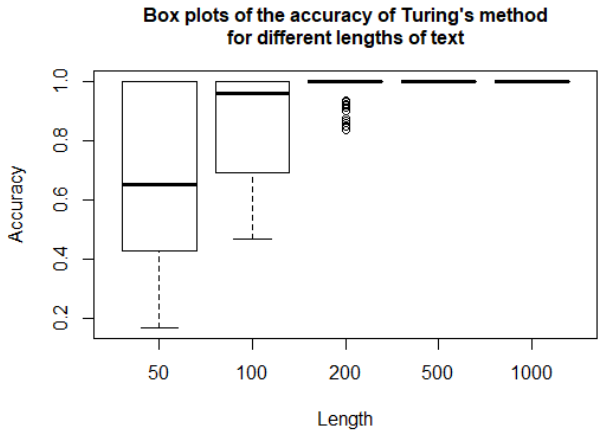
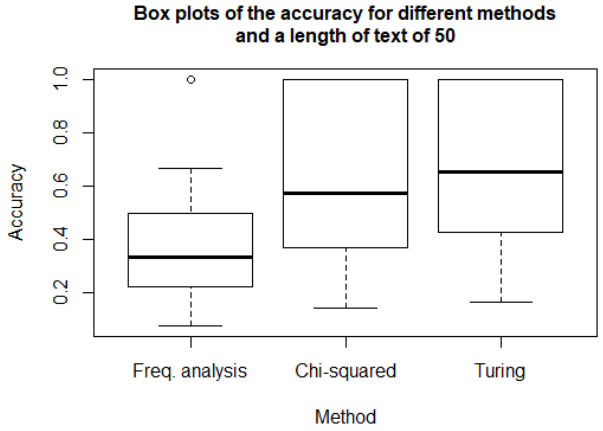
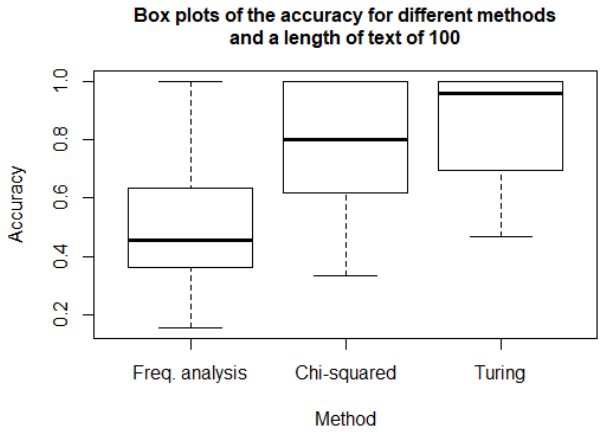


Figure 5.3: Box plots of the accuracy of decryption for Turing's method



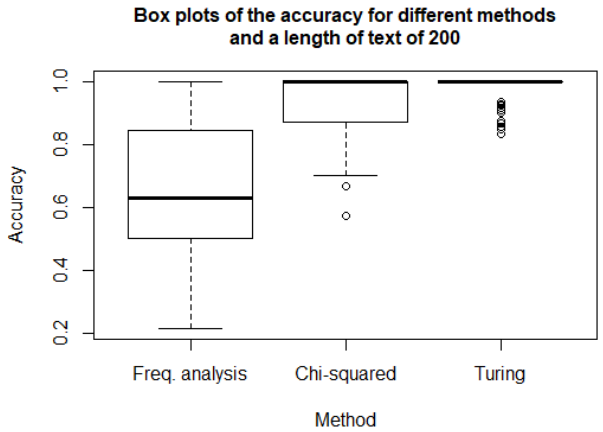
	Bad decryptions (/100)
Freq. analysis	72
Chi-squared	36
Turing	31

Figure 5.4: Box plots of the accuracy of decryption for the three methods and a length of text of 50



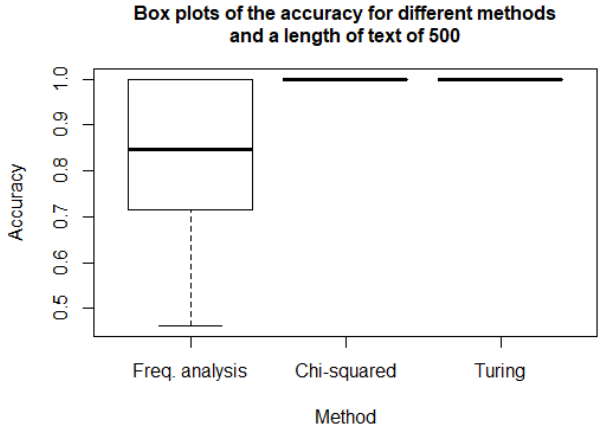
	Bad decryptions (/100)
Freq. analysis	53
Chi-squared	6
Turing	1

Figure 5.5: Box plots of the accuracy of decryption for the three methods and a length of text of 100



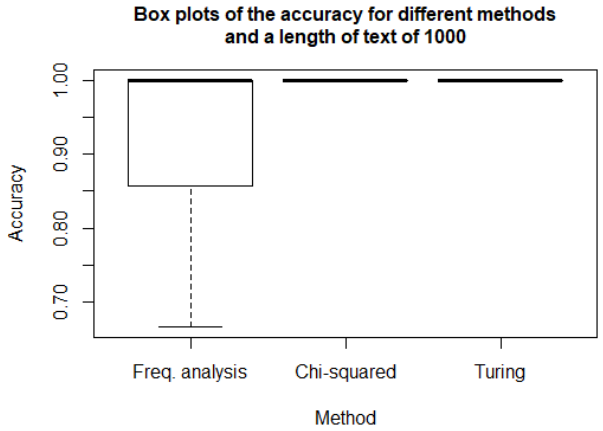
	Bad decryptions (/100)
Freq. analysis	23
Chi-squared	0
Turing	0

Figure 5.6: Box plots of the accuracy of decryption for the three methods and a length of text of 200



	Bad decryptions (/100)
Freq. analysis	1
Chi-squared	0
Turing	0

Figure 5.7: Box plots of the accuracy of decryption for the three methods and a length of text of 500



	Bad decryptions (/100)
Freq. analysis	0
Chi-squared	0
Turing	0

Figure 5.8: Box plots of the accuracy of decryption for the three methods and a length of text of 1000

As we can see, the accuracy of the techniques depends on the length of text that was used; the longer, the better.

Whatever the length of text is, Turing's technique is the most efficient in terms of accuracy, bad decryptions and computing time and it is significantly better given the standard errors. The Chi-squared method is also quite efficient but it requires much more time compared to others. For small lengths of text, the accuracy of the frequency analysis technique is so bad compared to the two other techniques. Even for larger lengths of text (except for a length of text of 1000), the frequency analysis method leads to a larger proportion of bad decryptions.

After some investigations about these results, we find that each time Turing's method fails, the two other methods fail as well and each time Chi-squared method fails the method based on frequency analysis fails too.

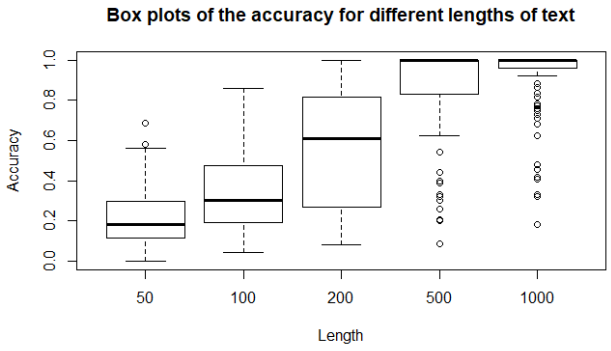
Bad decryption arise when the considered ciphertext is too small or when it corresponds to a portion of plaintext beginning in the middle of a word and ending in the middle of another.

5.3.2 Substitution cipher

In order to compare the performance of the Metropolis-Hastings algorithm for substitution cipher, we compare the efficiency of this technique for different lengths of ciphertext and the time spent on the decryption. Table 5.7 and Figure 5.9 present the results.

	Average Accuracy	Standard Error	No. of successful runs (/100)	Average Duration (in seconds)
Length: 50	0.2137	0.0142	0	301.0323
Length: 100	0.3387	0.0199	0	437.4753
Length: 200	0.5635	0.0294	7	554.0230
Length: 500	0.8694	0.0228	55	494.7783
Length: 1,000	0.9115	0.0176	52	585.7377

Table 5.7: Results of attacks on substitution cipher for different lengths of text.



	Bad decryptions (/100)
Length: 50	95
Length: 100	77
Length: 200	43
Length: 500	11
Length: 1,000	8

Figure 5.9: Box plots of the accuracy of decryption for substitution cipher

As we can see, the accuracy of the Metropolis-Hastings algorithm for substitution cipher widely depends on the length of text: the longer, the better. Furthermore, the duration time increases with the length of text because the algorithm has to measure the efficiency of the decryption on the entire text. It also depends on the successful runs through the number of iterations needed to correctly decrypt the text. Part of the bad decryptions are caused by the presence of numbers in the selected text such as date or years. Furthermore, the decryption process cannot handle monarchical ordinal like "GEORGE III". It is also the footnote numbering, leading to plaintext like "DESPATCHES38", which causes problems.

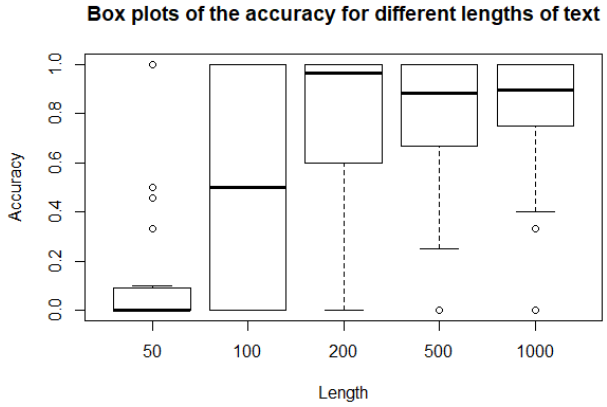
5.3.3 Transposition cipher

In order to compare the performance of the Metropolis-Hastings algorithm for transposition cipher, we compare the efficiency of this technique for different lengths of ciphertext and the time spent on the decryption. Table 5.8 and Figure 5.10 present the results.

As we can see, the accuracy of the Metropolis-Hastings algorithm for transposition cipher depends on the length of text too: the longer, the better. When the text is too small (length of 50 or 100), the process leads to a larger proportion of bad decryptions. For a length of text of 100, the algorithm leads to both good and bad results. For longer texts, the number of bad decryptions decreases significantly and the accuracy increases. Furthermore, the duration time increases with the length of text because the algorithm has to measure the efficiency of the

	Average Accuracy	Standard Error	No. of successful runs (/100)	Average Duration (in seconds)
Length: 50	0.1926	0.0376	17	61.33348
Length: 100	0.5173	0.0422	34	65.81684
Length: 200	0.7756	0.0304	42	80.81258
Length: 500	0.79699	0.0267	47	171.39540
Length: 1,000	0.8508	0.0194	49	268.63110

Table 5.8: Results of attacks on transposition cipher with different lengths of text.



	Bad decryptions (/100)
Length: 50	82
Length: 100	46
Length: 200	15
Length: 500	10
Length: 1,000	5

Figure 5.10: Box plots of the accuracy of decryption for transposition cipher

decryption on the entire text. It also depends on the successful runs through the number of iterations needed to correctly decrypt the text. Part of the bad decryptions are caused by the presence of numbers in the selected text as explained in the previous section for substitution cipher.

In conclusion, the accuracy of all the ciphers presented in this thesis depends on the length of the text. For Vigenère cipher, Turing's method is the best method out of the three presented in this thesis. The Markov chain Monte Carlo theory is quite efficient for both substitution and transposition cipher but widely depends on the allowed number of iterations (here we fixed respectively 10,000 iterations and 20,000 iterations). The bad decryptions are often caused by numbers in the selected text that cannot be correctly decrypted.

Summary

In this thesis, we successfully broke various pre-modern ciphers. All the methods were based on frequency analysis of letters or of bigrams of the ciphertext compared to a reference text. The methods presented in Chapters 2 and 3 can be easily applied and their ideas are quite natural. The fourth chapter was the most evolved and indicated the usefulness of the Markov Chain Monte Carlo theory. The last chapter measured the performance of some techniques and investigated the importance of the length of ciphertext in the decryption process. Overall, the simulations consolidated the fact that the methods presented are quite efficient.

Naturally, the more information we had about the ciphertext, the better was our decryption.

The above cipher attacks assumed that the type of cipher was known in advance. In real situation, this assumption does not hold. In this thesis we only considered three types of ciphers: mono-alphabetic cipher, transposition cipher and poly-alphabetic cipher.

Looking at frequencies of letters can be a precious clue to determine which cipher was used. Indeed, transposition cipher does not change the frequencies of letters at all because it only changes the positions of letters. Furthermore, mono-alphabetic cipher substitutes a letter with another so the distribution of letters is the same. If the distribution of letters changes, it is a poly-alphabetic cipher that was used. A Chi-squared test can be used to formally test whether the distributions of letters are the same (as presented in Chapter 2). More developments to distinguish the ciphers can be found in Kocmànek's bachelor thesis (see [20]).

Furthermore, the index of coincidence presented in section 2.4.2 can be used to determine whether the message was encrypted using a mono-alphabetic or poly-alphabetic cipher. In the case of a mono-alphabetic cipher, the IC of the ciphertext should be nearly the same as the IC for English plaintext, i.e., 0.078. For poly-alphabetic ciphers, the distribution of the frequencies would be more uniform; in the extreme case the IC will be 0.037. In conclusion, when computing the IC for the ciphertext we try to attack, the closer it is to 0.078, the more likely it has been enciphered with a mono-alphabetic cipher. On the contrary, the closer the IC is to 0.037, the more likely the cipher is poly-alphabetic.

In this thesis we did not investigate modern ciphers such as the *Data Encryption Standard* (DES) or the *Advanced Encryption Standard* (AES) which are in use nowadays. We did not discuss either about the RSA (named after its inventors Rivest, Shamir and Adleman), which is based on number theory. These systems require much more mathematical theory and methods that would have been out of the topic.

Appendix A

R code

All the relevant codes produced for this master's thesis are included in this appendix and are available (with relevant .txt files) at:
https://drive.google.com/drive/folders/1tPog2Yvfex2z_Sy2Xg5mvjS5kcQDZGbr?usp=sharing

We do not claim that these codes are the most efficient way of producing the results and the algorithms introduced in this thesis; they are included purely for completeness. The functions are briefly commented when needed.

Chapter 1

```
library(tm)
library("stringr")
library(readr)
```

Used alphabet

```
Alphabet <- c("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", " ")

Alphabet_space <- c("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", "_")

Alphabet_size <- length(Alphabet)
Alphabet_Number <- 0:(Alphabet_size-1) #A is 0, B is 1,..., Z is 25,  _ is 26.
names(Alphabet_Number) <- Alphabet #Names : Functions to get or set the names of an object.
#In order to use letters and numbers interchangeably
```

Useful functions

```
Format_Text_NP<- function(Text){

  #Remove break lines
  Text <- gsub("\n", " ", Text)
  Text <- gsub("\r", " ", Text)

  #Remove punctuation
  Text<- removePunctuation(Text)

  #Remove numbers
  Text<- removeNumbers(Text)
```

```
  #Remove double spaces
  Text <- str_squish(Text)

  #Uppercase
  Text<- toupper(Text)
  return(Text)
}
```

→ This function formats the given text by removing break lines, punctuation, numbers and double spaces. It returns the text in uppercase format.

```
Char_To_Num<- function(Text){
  Num_vect<- NULL

  for(i in 1:nchar(Text)){
    Num_vect[i]<- Alphabet_Number[substr(Text,i,i)]
    #"substr(Text,i,i)" extracts the i-th character in a character vector.
    #"Alphabet_Number[substr(Text,i,i)" : corresponding number in the alphabet of the
        character
  }

  return(Num_vect)
}
```

→ This function converts a character string into a vector of corresponding numbers.

```
Num_To_Char<- function(Num_vect){
  Text<- "" #initialize Text without characters
  for(i in 1:length(Num_vect)) Text<- paste0(Text, Alphabet[Num_vect[i]+1])
  #function "paste0": concatenates without spaces
  return(Text)
}
```

→ This function converts a vector of numbers into the corresponding character string.

```
Text_To_Matrix<- function(Text,m){
  N<- nchar(Text)

  if(N%%m!=0){
    Num_add<- NULL #It will contain the Alphabet numbers of letters that will be added at the
        end of the text

    #random choice
    Num_vect<- sample(1:Alphabet_size) #random permutation of {1, ..., Alphabet_size}.
    for(i in 1: N%%m){
      Num_add[i]<- Num_vect[i]
    }
    Char_add<- Num_To_Char(Num_add) #character string that will be added at the end of the
        text

    Text<- paste(Text, Char_add, sep= "")#paste(word1,word2, sep=""): concatenates two words
        without space
    N<- nchar(Text) #Compute the new length of the text
  }

  Mat_text<- matrix(0, nrow=N/m ,ncol=m)
  for(i in 1:(N/m)){
    for(j in 1:m ){
      Mat_text[i,j]<- substr(Text,j+m*(i-1),j+m*(i-1))
    }
  }

  return(Mat_text)
}
```

→ This function transforms the text into a matrix with m columns. If the length of the text N is a multiple of m, there will be $\frac{N}{m}$ rows. Else, this function adds random characters at the end of the text in order to have a length of text of N' which is a multiple of m. In this case, the number of rows will be $\frac{N'}{m}$. The i^{th} row contains characters $(i-1)*m+1$ to $i*m$.

```
Char_Count<- function(Text, each.letter=FALSE){
  N = nchar(Text)

  if(each.letter==TRUE){
    Vect<- rep(0, Alphabet_size)
    for(i in 1:N){
      if(is.na(Alphabet_Number[substr(Text,i,i)])== FALSE){
      Vect[Alphabet_Number[substr(Text,i,i)]+1 ] <- Vect[ Alphabet_Number[substr(Text,i,i)]+1
          ] + 1
      }
    }
    return(Vect)
  }

  return(N)
}
```

→ This function counts the total numbers of characters of a text that are from the considered alphabet. If the option `each.letter=TRUE` is added, it counts the number of occurrences of each character in a text.

```
Matrix_To_Text<- function(Mat_text){

  Text<- matrix(do.call(paste0, as.data.frame(Mat_text))) #concatenate the matrix by row
  Text<- paste(Text, collapse="") #concatenate the vector
  return(Text)
}
```

→ This function converts a matrix into a character string. If the matrix has m columns, the i^{th} row of the matrix will be characters $(i-1)*m+1$ to $i*m$ of the string.

Used texts

Reference texts

```
# Austen's text
Ref_text1 <- read_file("Austen.txt")
Ref_text1<- Format_Text_NP(Ref_text1)
Ref_size1<- nchar(Ref_text1)

# English Constitution
Ref_text2 <- read_file("Constitution.txt")
Ref_text2<- Format_Text_NP(Ref_text2)
Ref_size2<- nchar(Ref_text2) #Number of characters

# Shakespeare's text
Ref_text3<- read_file("Shakespeare.txt")
Ref_text3<- Format_Text_NP(Ref_text3)
Ref_size3<- nchar(Ref_text3) #Number of characters

# French text
Ref_text_fr<- read_file("Hugo.txt")
Ref_text_fr<- Format_Text_NP(Ref_text_fr)
Ref_size_fr<- nchar(Ref_text_fr) #Number of characters

# Italian text
Ref_text_it<- read_file("Italian.txt")
Ref_text_it<- Format_Text_NP(Ref_text_it)
Ref_size_it<- nchar(Ref_text_it) #Number of characters
```

Chosen text

```
# Song "Ordinary love"
Chosen_text <- read_file("ordinary_love2.txt")
Chosen_text<- Format_Text_NP(Chosen_text)
```

Substitution cipher

```
Substitution_Cipher <- function(Permutation, Text){
  N = nchar(Text)

  #encryption
  Ciphertext <- "" initialize Ciphertext without characters
  for(i in 1:N){
    Ciphertext <- paste0(Ciphertext , Alphabet[Permutation[Alphabet_Number[substr(Text,i,i)
        ]+1]+1])
  }

  return(Ciphertext)
}
```

→ This function encrypts a text using substitution cipher. The i^{th} symbol of the alphabet is replaced by the symbol whose number is at the i^{th} position in the shuffle vector *Permutation*.

Examples

```
Permutation <- 0:(Alphabet_size-1) #vector that contains integers from  1 to Alphabet_size
Permutation <- sample(Permutation) #random shuffle

#Quick example
Subst_ex<- Substitution_Cipher(Permutation, "THE SEA WANTS TO KISS")   #ciphertext

#Whole text "Ordinary Love"
Subst_ex_whole<- Substitution_Cipher(Permutation, Chosen_text)  #ciphertext
```

Caesar shift cipher

```
Caesar_cipher<- function(Text, n, Decrypt=FALSE){

  Num_vect <- Char_To_Num(Text)

  Shifted_num <- (( Num_vect + (if (Decrypt==TRUE) -1 else 1)*n) %% Alphabet_size)  #Replace a
      character with character n positions further in the alphabet

  Ciphertext <- Num_To_Char(Shifted_num)

  return(Ciphertext)
}
```

→ This function encrypts a text using a Caesar cipher with shift n. An option `Decrypt=TRUE` must be added to decrypt the text using a shift n.

Examples

```
#Quick example "THE SEA WANTS TO KISS"
Char_To_Num("THE SEA WANTS TO KISS") #convert the character string into numbers

Caesar_ex<-Caesar_cipher("THE SEA WANTS TO KISS",20)  #ciphertext
Char_To_Num(Caesar_ex) #corresponding numbers

#Whole text "Ordinary Love"
Caesar_ex_whole<- Caesar_cipher(Chosen_text, 20) #ciphertext
```

Permutation/Transposition cipher

```
Permutation_Cipher<- function(Text, m, see.permutation=FALSE, seed=NULL){
  N<- nchar(Text)

  Mat_text<- Text_To_Matrix(Text,m)
```

```
  N<- nrow(Mat_text)*ncol(Mat_text)

  #random permutation
  if(length(seed)!=0) set.seed(seed)
  Permutation<- sample(1:m)
  if(see.permutation == TRUE) print(Permutation)


  Mat_permutation<- matrix(0, nrow=N/m ,ncol=m)
  for(i in 1:m){
    Mat_permutation[,i]<- Mat_text[,Permutation[i]]
  }
  return(Mat_permutation)
}
```

→ This function encrypts a text using the permutation/transposition cipher. It returns the encrypted text written out as a matrix. An option `see.permutation= TRUE` must be added to see the permutation that was used. An option `seed=` must be added to set a seed before the random shuffle.

Examples

```
Permutation_ex<- Permutation_Cipher("THE SEA WANTS TO KISS THE GOLD",6, see.permutation = TRUE
    )
Matrix_To_Text(Permutation_ex)

#Whole text "Ordinary Love"
Permutation_ex_whole<- Permutation_Cipher(Chosen_text, 6, see.permutation = TRUE)
Matrix_To_Text(Permutation_ex_whole)
```

Vigenere cipher

```
Vigenere_cipher<- function(Text, Keyword, Decrypt=FALSE, WS=FALSE){
  Num_vect<- Char_To_Num(Text)

  Num_key<- Char_To_Num(Keyword)

  mod<- length(Num_vect)%%length(Num_key)
  div<- (length(Num_vect)-mod)/length(Num_key)

  Keystream<- rep(Num_key, div)
  for(i in 1:mod){
   Keystream<-  c(Keystream, Num_key[i])
  }

  Num_ciphertext<- NULL

  for(i in 1:length(Num_vect)){
    Num_ciphertext[i]<- (Num_vect[i] + (if (Decrypt==TRUE) -1 else 1)*Keystream[i])%% Alphabet
        _size

  }

  return(Num_To_Char(Num_ciphertext))
}
```

→ This function encrypts a text using the Vigenere cipher and a keyword. An option `Decrypt= TRUE` must be added to decrypt the text using the specified keyword.

Examples

```
#Quick example with keyword "ORDINARY"
Vig_ex<- Vigenere_cipher("THE SEA WANTS TO KISS", "ORDINARY")
Char_To_Num(Vig_ex)

#Whole text "Ordinary Love"
Vig_ex_whole<- Vigenere_cipher(Chosen_text, "ORDINARY")
```

Chapter 2

Librairies

```
library(DescTools)
library(stringr)
library("wordcloud")
library("ngram")
```

N-grams function

Letters

```
Frequencies_Symbols<- function(Text,  bar.plot = FALSE, words.cloud=FALSE){

     Text_count<- Char_Count(Text, each.letter=TRUE)

     Text_freq  <- Text_count/sum(Text_count) #probability vector

    if(bar.plot==TRUE) barplot(Text_freq, names=Alphabet_space, main="Frequencies of symbols")

    if(words.cloud==TRUE){
      wordcloud(words = Alphabet_space, freq = Text_freq , min.freq = 1,
                max.words=200, random.order=FALSE,random.color = FALSE, rot.per=0.35,
                colors=brewer.pal(8, "Dark2"))

    }

  return(Text_freq)
}
```

→ This function computes the frequencies of letters of a text. An option `bar.plot=TRUE` must be added to plot the corresponding barplot. An option `words.cloud=TRUE` must be added to plot the corresponding words' cloud.

N-grams

```
Most_Common_ngrams<- function(Text,n, space=FALSE, words.cloud=FALSE, mult.occ=FALSE){

  Text__<- gsub(" ", "_", Text) #The original spaces are replaced by the character "_"
  Text_s<- gsub("([A-Z])", "\\1 ", Text__) #add spaces between each letter to use functions "
      ngram"
  Text_s<- gsub("_", "_ ", Text_s) #add spaces after each symbol "_" to use functions "ngram"

  Text_ng <- ngram(Text_s, n =n)
  Text_ngrams<- get.phrasetable(Text_ng)

  Text_ngrams_mod<- gsub(" ", "", Text_ngrams$ngrams, fixed = TRUE) #No spaces between
      characters of the ngrams

  Text_ngrams_names<- Text_ngrams_mod
  Text_ngrams_freq<- Text_ngrams$freq

  if(space==FALSE){ #All the trigrams containing a space character will not be taken into
      account
    Text_ngrams_names<- NULL
    Text_ngrams_freq<- NULL

    string_rep<- paste(rep("*", n), collapse="") #n-grams containing spaces will be replaced
        by "***"

    for(i in 1:n){
      string<- paste(rep("\\S+", n-1), collapse="")
      string<- paste(c(string, "\\S"), collapse = "")
```

```
      substr(string, (i-1)*3+1,(i-1)*3+2)<- "__"
      string<- gsub("__", "_", string)

      Text_ngrams_mod <- gsub(string, string_rep, Text_ngrams_mod) #if the n-gram contains a
          space character, the ngram will be replaced by "***" and will not be taken into
          account

    }

    for(i in 1:length(Text_ngrams_mod)){ #Delete all the n-grams that contains a space
      #It will modifies the corresponding frequencies vector of the n-grams
     if(Text_ngrams_mod[i]!=string_rep){
       Text_ngrams_names<- c(Text_ngrams_names, Text_ngrams_mod[i] )
       Text_ngrams_freq<- c(Text_ngrams_freq,Text_ngrams$freq[i])
      }
    }
  }

  if(words.cloud==TRUE){

   wordcloud(words = Text_ngrams_names, freq = Text_ngrams_freq, min.freq = 1+mult.occ,
             max.words=300, random.order=FALSE,random.color = FALSE,  rot.per=0,
             colors=brewer.pal(8, "Dark2"))
  }

  return(Text_ngrams)
}
```

→ This function returns the frequencies of the n-grams available in the text. If the option `space=TRUE` is added, the space is considered as a symbol and is taken into account in the n-grams. If the option `words.cloud=TRUE`, a words' cloud of the n-grams is plotted.

Frequencies of n-grams in the reference text

Letters

```
Ref_freq1<- Frequencies_Symbols(Ref_text1,bar.plot = FALSE, words.cloud = FALSE)
barplot(Ref_freq1, names=Alphabet_space, main="Frequencies of symbols: Austen")

Ref_freq2<- Frequencies_Symbols(Ref_text2,bar.plot = FALSE, words.cloud = FALSE)
barplot(Ref_freq2, names=Alphabet_space, main="Frequencies of symbols: Constitution")

Ref_freq3<- Frequencies_Symbols(Ref_text3,bar.plot = FALSE, words.cloud = FALSE)
barplot(Ref_freq3, names=Alphabet_space, main="Frequencies of symbols: Shakespeare")

Ref_freq_fr<- Frequencies_Symbols(Ref_text_fr,bar.plot = FALSE, words.cloud = FALSE)
barplot(Ref_freq_fr, names=Alphabet_space, main="Frequencies of symbols: French text")

Ref_freq_it<- Frequencies_Symbols(Ref_text_it,bar.plot = FALSE, words.cloud = FALSE)
barplot(Ref_freq_it, names=Alphabet_space, main="Frequencies of symbols: Italian text")
```

LUL

```
LUL<- function(Ref_freq){
  lul_ref<- Ref_freq[10]+Ref_freq[11]+Ref_freq[17]+Ref_freq[24]+Ref_freq[26]#store the values
      of less used letters
  Ref_freq<- Ref_freq[-c(10,11,17,24,26)] #remove those values from the vector

  return(c(Ref_freq, lul_ref)) #add lul value
}
```

→ This function creates a vector of frequencies by putting together letters J,K,Q,X,Z in a new category named LUL.

Comparisons

```
chisq.test(Ref_size2*Ref_freq2, p=Ref_freq1) #English Constitution
chisq.test(Ref_size3*Ref_freq3, p=Ref_freq1) #Shakespeare's text
chisq.test(Ref_size_fr*Ref_freq_fr, p=Ref_freq1) #French text
chisq.test(Ref_size_it*Ref_freq_it, p=Ref_freq1) #Italian text
```

Bigrams

```
Ref_bigrams1<- Most_Common_ngrams(Ref_text1, 2, words.cloud = TRUE)
Ref_bigrams2<- Most_Common_ngrams(Ref_text2, 2, words.cloud = TRUE)
Ref_bigrams3<- Most_Common_ngrams(Ref_text3, 2, words.cloud = TRUE)
Ref_bigrams_fr<- Most_Common_ngrams(Ref_text_fr, 2, words.cloud = TRUE)
Ref_bigrams_it<- Most_Common_ngrams(Ref_text_it, 2, words.cloud = TRUE)

#with spaces
Ref_bigrams_s1<- Most_Common_ngrams(Ref_text1, 2,space = TRUE, words.cloud = TRUE)
Ref_bigrams_s2<- Most_Common_ngrams(Ref_text2, 2,space = TRUE, words.cloud = TRUE)
Ref_bigrams_s3<- Most_Common_ngrams(Ref_text3, 2,space = TRUE, words.cloud = TRUE)
Ref_bigrams_s_fr<- Most_Common_ngrams(Ref_text_fr, 2,space = TRUE, words.cloud = TRUE)
Ref_bigrams_s_it<- Most_Common_ngrams(Ref_text_it, 2,space = TRUE, words.cloud = TRUE)
```

Trigrams

```
Ref_Trigrams1<- Most_Common_ngrams(Ref_text1, 3, words.cloud = TRUE)
Ref_Trigrams2<- Most_Common_ngrams(Ref_text2, 3, words.cloud = TRUE)
Ref_Trigrams3<- Most_Common_ngrams(Ref_text3, 3, words.cloud = TRUE)
Ref_Trigrams_fr<- Most_Common_ngrams(Ref_text_fr, 3, words.cloud = TRUE)
Ref_Trigrams_it<- Most_Common_ngrams(Ref_text_it, 3, words.cloud = TRUE)

#with spaces
Ref_Trigrams_s1<- Most_Common_ngrams(Ref_text1, 3,space = TRUE, words.cloud = TRUE)
Ref_Trigrams_s2<- Most_Common_ngrams(Ref_text2, 3,space = TRUE, words.cloud = TRUE)
Ref_Trigrams_s3<- Most_Common_ngrams(Ref_text3, 3,space = TRUE, words.cloud = TRUE)
Ref_Trigrams_s_fr<- Most_Common_ngrams(Ref_text_fr, 3,space = TRUE, words.cloud = TRUE)
Ref_Trigrams_s_it<- Most_Common_ngrams(Ref_text_it, 3,space = TRUE, words.cloud = TRUE)
```

Caesar shift cipher

Method: Frequency Analysis

Letters

```
Caesar_freq<- Frequencies_Symbols(Caesar_ex_whole,bar.plot = TRUE, words.cloud = TRUE)
barplot(Caesar_freq, names=Alphabet_space, main="Frequencies of symbols")

plot(Alphabet_Number, Caesar_freq,pch=20,type="o",cex=1, main="Frequencies of letters", ylab =
     "Frequency", xlab="Letter")
lines(Alphabet_Number, Ref_freq1, pch=20,type="o",cex=1,col="red")
legend(x=0,y=0.2,c("English","Ciphertext"),cex=.8,col=c("red","black"), lty = c(1,1))
```

Bigrams

```
Caesar_bigrams_s<- Most_Common_ngrams(Caesar_ex_whole, 2,space = TRUE, words.cloud = TRUE)
```

Trigrams

```
Caesar_trigrams_s<- Most_Common_ngrams(Caesar_ex_whole, 3,space = TRUE, words.cloud = TRUE)
```

Method: Chi-squared statistic

```
Chi2_Stat<- function(Text, Ref_freq, LUL=FALSE){

  Count<- Char_Count(Text, each.letter=TRUE)

  n<- sum(Count)

  if(LUL==TRUE){
    lul<- Count[10]+Count[11]+Count[17]+Count[24]+Count[26]#store the values of less used
        letters
    Count<- Count[-c(10,11,17,24,26)] #remove those values from the vector
    Count<- c(Count, lul) #add lul value

    lul_ref<- Ref_freq[10]+Ref_freq[11]+Ref_freq[17]+Ref_freq[24]+Ref_freq[26]#store the
        values of less used letters
    Ref_freq<- Ref_freq[-c(10,11,17,24,26)] #remove those values from the vector
    Ref_freq<- c(Ref_freq, lul_ref) #add lul value

    return(list(Count, Ref_freq))
  }
  Term<- (Count-n*Ref_freq)^2/(n*Ref_freq)

  return(sum(Term))
}
```

→ This function computes the Chi-squared statistic for the observed counts of letters of the text and the expected counts of letters in English.

```
Find_Key_Caesar<- function(Ciphertext,Ref_freq, print=FALSE, plot=FALSE){

  Chi2<- NULL
  key<- 0

  for(i in 0:(Alphabet_size-1)){

    text<- Caesar_cipher(Ciphertext, -i)

    Chi2<-  c(Chi2,Chi2_Stat(text, Ref_freq, LUL=FALSE))

    if(print==TRUE){
      cat("Key: ", i, "Plaintext: ", substr(text,1,20), "Chi2", Chi2[i+1], "\n")
    }

  }
  key<- which.min(Chi2)-1

  if(plot==TRUE){
    plot(Alphabet_Number, Chi2,pch=20,type="o",cex=1, main="", ylab = "Chi-squared statistic",
         xlab="Key")
    title(main=paste("Chi-squared statistic obtained when using ","\n",sep=""),cex.main=1)
    title(main=paste("\n","the 27 possible keys",sep=""),cex.main=1)
    }
  if(print==TRUE){

    cat("The key is : ", key, "\n")
  }
  return(key)
}
```

→ This function finds the key used for the Caesar cipher by computing the Chi-squared statistic and taking the key that minimizes it. An option `print=TRUE` must to be added to print, for each key tested, the first 20 symbols of the ciphertext decrypted with the key and the corresponding Chi-squared statistic.

Example

```
Find_Key_Caesar(Caesar_ex_whole, Ref_freq1, print=TRUE, plot=TRUE)
```

Chi-squared test

```
statistic<- NULL
for(i in 0:(Alphabet_size-1)){
  text<- Caesar_cipher(Caesar_ex_whole, -i)
  cat("Key: ", i,"\n")
  Counts<- Chi2_Stat(text,Ref_freq1, LUL=TRUE)
  observed_LUL<- Counts[[1]]
  #observed_LUL
  expected_LUL<- Counts[[2]]
  #expected_LUL
  test<- chisq.test(observed_LUL, p=expected_LUL)
  print(test)
  statistic<- c(statistic, test$statistic)
}
print(statistic)

plot(Alphabet_Number,statistic,pch=20,type="o",cex=1, main="", ylab = "Chi-squared statistic",
     xlab="Key")
title(main=paste("Chi-squared statistic obtained when using ","\n",sep=""),cex.main=1)
title(main=paste("\n","the 27 possible keys",sep=""),cex.main=1)
plot(Alphabet_Number,statistic,pch=20,type="p",cex=1, ylab = "Chi-squared statistic", xlab="
    Key", ylim=c(0,200))
abline(h=qchisq(0.95,22), col="red")
```

Vigenere cipher: known period

Method: Frequency analysis of columns

```
Split_Text_Col<- function(Text,m){

  Split_text<- NULL
  Mat_text<- Text_To_Matrix(Text, m)
  for(i in 1:m)  Split_text[i]<- paste(Mat_text[,i], collapse="")

  return(Split_text)
}
```

→ This function splits a text into m character strings. The text is written out in columns and the i^{th} string is the letters from the i^{th} column.

Example

```
#Suppose it's 8
Split_Text_Col(Vig_ex_whole, 8)
```

```
Col_freq<- function(Text, m, Count=FALSE, plot=FALSE){
  Mat_freq_col<- matrix(0, ncol=m, nrow=Alphabet_size) #matrix whose i-th column will contain
      the frequencies of letters of the i-th column matrix Mat_text

  Mat_text<- Text_To_Matrix(Text, m)
  for(i in 1:m){
    if(Count==TRUE){Mat_freq_col[,i]<- Char_Count(paste(Mat_text[,i], collapse=""), each.
        letter=TRUE )}
    else Mat_freq_col[,i]<- Char_Count(paste(Mat_text[,i], collapse=""), each.letter=TRUE )/
        length(Mat_text[,i])
  }

  #plots frequencies within each column
  if(plot==TRUE){
    for(i in 1:m){
      barplot(Mat_freq_col[,i], names=Alphabet_space, main=substitute(paste("Frequencies of
          letters in column ", a), list(a=i)))
    }
  }
```

```
  return(Mat_freq_col)
}
```

→ This function returns a matrix whose i^{th} column contains the frequencies of the i^{th} column of the text written out as a matrix. An option `plot=TRUE` must be added to plot a barplot of the frequencies of symbols for each column.

```
Find_Key_Vig_FA<- function(Text, m, Mat_freq_col){

  keyword<- NULL

  for(i in 1:m){
    #print(which.max(Mat_freq_col[,i])-1)
    keyword[i]<- (which.max(Mat_freq_col[,i])-1-26)%%27 #letter with highest frequency must be
         decrypted as the space symbol
    #which.max(col_freq[,i])-1 is the Alphabet number of the most frequent symbol
    #26 represents the space symbol
  }

  keyword<- Num_To_Char(keyword)

  return(keyword)
}
```

→ This function finds the keyword used for a Vigenere cipher using the frequency analysis of the m columns of the text written out as a matrix.

Example

```
#Decrypt
t_freq_col<- Col_freq(Vig_ex_whole,8, plot=TRUE)
Find_Key_Vig_FA(Vig_ex_whole, 8, Mat_freq_col)

#Decryption of the text
Vigenere_cipher(Vig_ex_whole, "ORDINARY", Decrypt = TRUE)
```

Statistical Mehtod

```
Find_Key_Vig_Chi2<- function(Text, m, Ref_freq){
  Split_vig<- Split_Text_Col(Text, m)

  #We find the key for each column of Split_vig. For each column it's just a single
  #Caesar cipher.
  #The key will be a serie of numbers corresponding to the letters used
  keyword<- NULL

  for(i in 1:m) {
    keyword[i]<- Find_Key_Caesar(Split_vig[i], Ref_freq)
  }

  keyword<- Num_To_Char(keyword) #converts into a word

  return(keyword)
}
```

→ This function finds the key used for the Vigenere ciphertext by finding the key used for each column using the Chi-squared statistic's method.

Example

```
Find_Key_Vig_Chi2(Vig_ex_whole, 8, Ref_freq1)
```

Vigenere cipher: known period

Method: Frequency analysis

```
for(m in 1:16){

  Mat_vig<- Text_To_Matrix(Vig_ex_whole, m)  #to see the Text divided into columns
  Mat_freq_col<- Col_freq(Vig_ex_whole,m, plot=FALSE, Count=FALSE)

  for(i in 1:m){
     copy<-  Mat_freq_col[,i]
     s<- which.max(copy)
     for(j in 1:27){
       Mat_freq_col[(j-s)%%27 +27*(j-s==0),i]<- copy[j]
     }
  }

   Freq<- NULL
   for(j in 1:27) Freq[j]<- sum(Mat_freq_col[j,])
   Freq<- Freq/m

   max<- max(max(Freq), max(Ref_freq1))
   plot(Alphabet_Number, Freq,pch=20,type="o",cex=1, main=substitute(paste("Frequencies of
       letters with period ", a), list(a=m)), ylim= c(0, max+0.02), ylab = "Frequency", xlab="
       Letter")
   lines(Alphabet_Number, Ref_freq1, pch=20,type="o",cex=1,col="red")
   legend(x=0,y=0.2,c("English","Ciphertext"),cex=.8,col=c("red","black"), lty = c(1,1))

}
```

Kasiski's Method

```
Repetitive_ngrams<- function(Text, n){
  Repet<- NULL

  Text_s<- Most_Common_ngrams(Text, n,space = TRUE)

  for(i in 1: length( Text_s$ngrams)){
    if(Text_s$freq[i]>1) Repet<- c(Repet, Text_s$ngrams[i])
  }
  return(Repet)
}
```

→ This function finds repeated n-grams in the text.

```
Kasiski<- function(Text,m_min, m_max){
  period_all<- NULL

  highest_ngram<- Repetitive_ngrams(Text, m_min)

  if(is.null(highest_ngram)==TRUE) return(NULL)


  for(i in (m_min+1):m_max){

    Rep<- Repetitive_ngrams(Text, i) #i-grams

    if(is.null(Rep)==TRUE){
      cat("The highest_ngram is: ",highest_ngram, "\n")
      return(Divisors(GCD(period_all)))
    }
    highest_ngram<- Rep

    Rep<- gsub(" ", "", Rep, fixed = TRUE)

    for(j in 1:length(Rep)){

      x<- str_locate_all(pattern=Rep[j], Text)[[1]][,1]

      period<-x[2]-x[1]
```

```
      if( 3<= length(x)) for(l in 3:length(x)){
       if(x[l]-x[l-1]<  period)  period<- x[l]-x[l-1]
      }
    }
   period_all<- c(period_all, period)
  }

  return(Divisors(GCD(period_all)))

}
```

→ This function implements Kasiski's method. It returns a vector containing possible periods of the Vigenere cipher. The two integers `m_min` and `m_max` indicate to start to search for `m_min`-grams and to end with `m_max`-grams.

Example

```
Kasiski(Vig_ex_whole,7,12)
```

IC Method

```
IC<- function(Text){
  count<- rep(0,Alphabet_size)
  count<- Char_Count(Text, each.letter=TRUE)
  n<- sum(count)

  return(sum(count*(count-1))/(n*(n-1)))
}
```

→ This function computes the IC of a text.

```
Mean_IC<- function(Text,m){
  #For the whole text (period=1)
  Ic<- NULL
  Ic[1]<- IC(Text)
  Period<- 1

  #For period of i
  for(i in 2:m){
      Split<- Split_Text_Col(Text, i)
      Vect<- NULL

      for(j in 1:i){
        Vect[j]<- IC(Split[j])
      }

      Ic[i]<- mean(Vect)

    }
  return(Ic)
}
```

→ This function computes the average IC of a text for different periods (from 1 to m). It returns a vector whose i^{th} component is the average IC assuming a period of i. To compute the i^{th} component the function:

- splits a text into i character strings;

- computes the IC for each of the i strings;

- computes the mean of these IC's.

```
Find_Period_Vig<- function(Text,m, IC_ref, plot=FALSE){
  IC_text<- Mean_IC(Text,m)
  x<- abs(IC_text-IC_ref)

  if(plot==TRUE){

    plot(1:m, IC_text, pch=20, type="o",cex=1, xlab="Period", ylab="IC", ylim = c(0,0.1),
        main="IC for different periods")
    abline(IC_ref,0, col="red")
    abline(0.037,0, col="blue")
    legend(x=1,y=0.1,c("Reference IC","Uniform IC"),cex=.8,col=c("red","blue"), lty = c(1,1))

    }

  return(which.min(x))
}
```

→ This function finds the period using the IC method.

Examples

```
IC_ref<- IC(Ref_text1) #IC of the Ref_text (Jane Austen)
IC_ref2<- IC(Ref_text2) #IC of the Ref_text2 (Consitution)
IC_ref3<- IC(Ref_text3) #IC of the Ref_text3 (Shakespeare)

#Find period (the actual period is 8)
Find_Period_Vig(Vig_ex_whole, 17, IC_ref, plot=TRUE)
```

Chapter 3: Alan Mathison Turing

Encrypted text

```
Vig_ex_ch3<- substr(Vig_ex_whole, 1, 72)
```

Table for scoring a Vigenere in units of half-decibans

```
#Table of factors 26*f_k/(1-fk)
Col1<- round(20*log10(26*Ref_freq1/(1-Ref_freq1)))
#Multiple-columns
Col2<-  round(2*20*log10(26*Ref_freq1/(1-Ref_freq1)))
Col3<- round(3*20*log10(26*Ref_freq1/(1-Ref_freq1)))
Col4<-  round(4*20*log10(26*Ref_freq1/(1-Ref_freq1)))
Col5<-  round(5*20*log10(26*Ref_freq1/(1-Ref_freq1)))

#Tabular display
Mat_mise<- matrix(0,nrow=27, ncol=6)
for(i in 1:27) Mat_mise[i, ]<- c(Col5[i],Col4[i] , Col3[i], Col2[i],Col1[i], Alphabet[i])
```

```
Score<- function(Text, Ref_freq){
  Score<- rep(0, 27)
  counts<- Char_Count(Text, each.letter=TRUE)
  Factors_Col<-NULL

  for(k in 0:26){
    #factors 26*p_alpha/1-p_alpha in half-decibans
    Factors_Col<- 20*log10(26*Ref_freq1/(1-Ref_freq1))

    copy<- Factors_Col
    for(l in 1:27) Factors_Col[(l+k)%%27 +27*((l+k)%%27==0)]<- copy[l]

    Score[k+1]<- sum(round(counts*Factors_Col))

  }
  return(Score)
}
```

→ This function computes the scores of a text for all the possible keys. It returns a vector whose components are the scores for each key. The text should be a string of letters encrypted with the same shift.

Example

```
Score("GJNSNVUFY", Ref_freq1)
```

```
Decrypt_Score<- function(Ciphertext, m,Ref_freq){
  Text_Col<- Split_Text_Col(Ciphertext, m) #Divide the ciphertext into columns

  Mat_mise<- matrix(0,nrow=27, ncol=m)
  key<- rep(0,m)

  for(i in 1:m){ #Score of the ith column for each key
    Mat_mise[,i]<- Score(Text_Col[i],Ref_freq)
  }

  return(Mat_mise)
}
```

→ This function creates a table of scores for the different keys and for the different columns of the ciphertext.

Example

```
#Table of scores for the different keys and for the different columns
Mat_mise<- Decrypt_Score(Vig_ex_ch3, 8,Ref_freq1)
```

```
Find_Key_Turing<- function(Ciphertext, m,Ref_freq){
  Mat_score<- Decrypt_Score(Ciphertext, m,Ref_freq)
  keyword<- NULL
  for(i in 1:m ){
    #print(which.max(Mat_score[,i])-1)
    keyword[i]<- which.max(Mat_score[,i])-1
    #letter with the highest frequency must be the shift

  }

  keyword<- Num_To_Char(keyword)

  return(keyword)
}
```

→ This function finds the keyword of the Vigenère cipher using Turing's method. For each column, the shift suggested corresponds to the key that has the highest score.

Examples: entire process

```
Find_Key_Turing(Vig_ex_ch3, 8, Ref_freq1)
Find_Key_Turing(Vig_ex_whole, 8, Ref_freq1)
```

Chapter 4: Monte Carlo Markov chain

Librairies

```
library(tm)
library(readr)
library("plotrix")
library("stringr")
library("DescTools")
library("purrr")
```

Used alphabet

```
Alphabet <- c("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","0", "1","2","3","4","5","6","7"
                  ,"8","9"," ")

Alphabet_size <- length(Alphabet)
Alphabet_Number <- 1:Alphabet_size
names(Alphabet_Number) <- Alphabet #Names : Functions to get or set the names of an object.
```

Useful functions

```
Format_Text_NP<- function(Text){

  #Remove break lines
  Text <- gsub("\n", " ", Text)
  Text <- gsub("\r", " ", Text)

  #Remove punctuation
  Text<- removePunctuation(Text)

  #Remove double spaces
  Text <- str_squish(Text)

  #Uppercase
  Text<- toupper(Text)
  return(Text)
}
```

→ This function formats the given text by removing break lines, punctuation and double spaces. It returns the text in uppercase format.

Used texts

Reference text

```
# Austen's text
Ref_text1 <- read_file("Austen.txt")
Ref_text1<- Format_Text_NP(Ref_text1)
Ref_size1<- nchar(Ref_text1)
```

Messages

```
# Jane Austen's text (Chapter 1 )
Message1 <- read_file("ch1.txt")
Message1<- Format_Text_NP(Message1)
Message1_size<- nchar(Message1)

# Song "Ordinary love"
Message2 <- read_file("ordinary_love2.txt")
Message2<- Format_Text_NP(Message2)
Message2_size<- nchar(Message2)

# French text about the song "Ordinary Love"
Message3 <- read_file("u2.txt")
Message3<- Format_Text_NP(Message3)
Message3_size<- nchar(Message3)
```

Substitution cipher

Encryption

```
Encryption<- function(Permutation, Text){

  N = nchar(Text)

  Ciphertext <- "" #initialize Ciphertext without characters
  for(i in 1:N){
    if(is.na(Alphabet_Number[substr(Text,i,i)])==1){
      Ciphertext  <- paste0(Ciphertext  , " ")
    }
    else{
      Ciphertext  <- paste0(Ciphertext  , Alphabet[Permutation[Alphabet_Number[substr(Text,i,i
          )]]])
    }
  }

  return(Ciphertext)
}
```

→ This function encrypts a message using a substitution cipher.

Metropolis-Hastings algorithm

Transition matrix

```
Transition_Matrix <- function(Text){

  Mat<- matrix(0, nrow = Alphabet_size , ncol = Alphabet_size)
  N<-  nchar(Text)

  #matrix elements
  for(i in 1:(N-1)){
    Mat[ Alphabet_Number[substr(Text,i,i)] , Alphabet_Number[substr(Text,i+1,i+1)] ] <- Mat[
        Alphabet_Number[substr(Text,i,i)] , Alphabet_Number[substr(Text,i+1,i+1)] ] + 1
  }

  return(Mat)
}
```

→ This function creates a matrix whose element (i, j) records the number of occurrences of the pair (i, j) in the text.

```
Probability_Matrix<- function(Mat){

  Sums <- rowSums(Mat)
  Sums <- replace(Sums, Sums==0, 1) #"replace(x, list, values)": replace the values in x with
      indices given in list by those given in values. If necessary, the values in values are
      recycled

  return(Mat/Sums)
}
```

→ This function transforms a matrix into a probability matrix.

For the reference texts

```
# Austen's text

Transition_mat1 <- Transition_Matrix(Ref_text1)
#Probability matrix
Probability_mat1<- Probability_Matrix(Transition_mat1)
#Plot matrix
color2D.matplot(Probability_mat1,cs1=c(1,0),cs2=c(1,0),cs3=c(1,0),
                extremes=NA,cellcolors=NA,show.legend=TRUE,nslices=10,xlab="Column",
                ylab="Row", main="Transition matrix plot: Austen",do.hex=FALSE,axes=TRUE,show.
                    values=FALSE,vcol=NA,vcex=1,
                border="gray",na.color=NA,xrange=NULL,color.spec="rgb",yrev=TRUE,
                xat=NULL,yat=NULL,Hinton=FALSE)

# English Constitution

Transition_mat2<- Transition_Matrix(Ref_text2)
#Probability matrix
Probability_mat2<- Probability_Matrix(Transition_mat2)
#Plot matrix
color2D.matplot(Probability_mat2,cs1=c(1,0),cs2=c(1,0),cs3=c(1,0),
                extremes=NA,cellcolors=NA,show.legend=TRUE,nslices=10,xlab="Column",
                ylab="Row", main="Transition matrix plot: Austen",do.hex=FALSE,axes=TRUE,show.
                    values=FALSE,vcol=NA,vcex=1,
                border="gray",na.color=NA,xrange=NULL,color.spec="rgb",yrev=TRUE,
                xat=NULL,yat=NULL,Hinton=FALSE)

#Shakespeare's text

Transition_mat3 <- Transition_Matrix(Ref_text3)
#Probability matrix
Probability_mat3<- Probability_Matrix(Transition_mat3)
#Plot matrix
color2D.matplot(Probability_mat3,cs1=c(1,0),cs2=c(1,0),cs3=c(1,0),
                extremes=NA,cellcolors=NA,show.legend=TRUE,nslices=10,xlab="Column",
                ylab="Row", main="Transition matrix plot: Austen",do.hex=FALSE,axes=TRUE,show.
                    values=FALSE,vcol=NA,vcex=1,
                border="gray",na.color=NA,xrange=NULL,color.spec="rgb",yrev=TRUE,
                xat=NULL,yat=NULL,Hinton=FALSE)

# French text

Transition_mat_fr <- Transition_Matrix(Ref_text_fr)
#Probability matrix
Probability_mat_fr<- Probability_Matrix(Transition_mat_fr)
#Plot matrix
color2D.matplot(Probability_mat_fr,cs1=c(1,0),cs2=c(1,0),cs3=c(1,0),
                extremes=NA,cellcolors=NA,show.legend=TRUE,nslices=10,xlab="Column",
                ylab="Row", main="Transition matrix plot: Austen",do.hex=FALSE,axes=TRUE,show.
                    values=FALSE,vcol=NA,vcex=1,
                border="gray",na.color=NA,xrange=NULL,color.spec="rgb",yrev=TRUE,
                xat=NULL,yat=NULL,Hinton=FALSE)

# Italian text

Transition_mat_it <- Transition_Matrix(Ref_text_it)
#Probability matrix
Probability_mat_it<- Probability_Matrix(Transition_mat_it)
#Plot matrix
color2D.matplot(Probability_mat_it,cs1=c(1,0),cs2=c(1,0),cs3=c(1,0),
                extremes=NA,cellcolors=NA,show.legend=TRUE,nslices=10,xlab="Column",
```

```
                ylab="Row", main="Transition matrix plot: Austen",do.hex=FALSE,axes=TRUE,show.
                    values=FALSE,vcol=NA,vcex=1,
                border="gray",na.color=NA,xrange=NULL,color.spec="rgb",yrev=TRUE,
                xat=NULL,yat=NULL,Hinton=FALSE)
```

Characters frequencies

```
Char_Freq <- function(Text){

  Vect<- rep(0, Alphabet_size)
  N = nchar(Text) #

  #vector elements
  for(i in 1:(N-1)){
    if(is.na(Alphabet_Number[substr(Text,i,i)])==0){
      Vect[ Alphabet_Number[substr(Text,i,i)]] <- Vect[ Alphabet_Number[substr(Text,i,i)] ] +
          1
    }
  }

  return(Vect/sum(Vect))
}
```

→ This function computes the frequencies of the characters in a text.

```
Ref_freq1 <- Char_Freq(Ref_text1) # Austen's text
Ref_freq2 <- Char_Freq(Ref_text2) # English Constitution
Ref_freq3 <- Char_Freq(Ref_text3) # Shakespeare's text
```

Starting decryption of the message

```
Starting_Decryption <- function(Ref_freq, Text_freq){

  Decryption <- 1:Alphabet_size

  for(i in 1:Alphabet_size){
    n1 <- which.max(Ref_freq)
    n2 <- which.max(Text_freq)
    Decryption[n2] <- n1
    Ref_freq[n1] <- -1
    Text_freq[n2] <- -1
  }

  return(Decryption)
}
```

→ This function computes a first possible key based on frequencies of characters in the reference text and in the encrypted message. The most frequent character of the encrypted text is mapped with the most frequent character in the reference text; the second most common symbol of the ciphertext is mapped with the second most common symbol of the reference text, etc.

```
Swap_Code <- function(Decryption,n1,n2){

  return(replace(Decryption , c(n1,n2) , Decryption[c(n2,n1)]))
}
```

→ This function swaps values in position `n1` and `n2` in the vector `Decryption`.

```
Log_Plausibility <- function(Decrypted,Mat){

  N = nchar(Decrypted)

  Mat<- Mat+1 #Add 1 to avoid problems when taking the log

  #Transition matrix for the decrypted text
```

```
  cmat<- Transition_Matrix(Decrypted)+1 #Add 1 to avoid problems

  Log_Pl<- 0
  for(i in 1:Alphabet_size){
    for(j in 1:Alphabet_size){
      Log_Pl<- Log_Pl+log(Mat[i,j])*cmat[i,j]

    }
  }

  return(Log_Pl)
}
```

→ This function computes the log-plausibility of the decryption key. The log-plausibility is computed using the logarithm of the transition matrix.

```
Efficiency<- function(Decrypted_Message, Text){

  nb_letters<- Alphabet_size

  corr_rev<- 0

  for(i in 1: Alphabet_size){
    #first appearance of the i-th letter in the plaintext
    p1<- StrPos(Text, pattern=Alphabet[i], pos = 1)
    #first appearance of the i-th letter in the decrypted text
    p2<- StrPos(Decrypted_Message, pattern=Alphabet[i], pos = 1)

    if(is.na(p1)==0 & is.na(p2)==0 & p1==p2){
      corr_rev<- corr_rev +1
    }

    if(is.na(p1)==1){  #If the i-th letter  doesn't appear in the plaintext
      nb_letters<- nb_letters-1;
    }
  }

  return(corr_rev/nb_letters)
}
```

→ This function computes the efficiency/accuracy of a decrypted text.

```
Decryption_MH<- function(Ciphertext,Nb_Iter,Decryption,Plaintext=NULL, print=FALSE){

  #instanciate a map to hold previously computed plausibilities
  map <- new.env(hash=T, parent=emptyenv())

  #First decryption
  Decrypted_Message<- Encryption(Decryption,Ciphertext)

  #Log-plausibility
  Log_Pl <- Log_Plausibility(Decrypted_Message,Transition_mat1)
  map[[paste(Decryption, collapse='')]] <- Log_Pl  #add to the map

  #Decryption key with the largest log-plausibility
  Best<- Decrypted_Message
  Best_Decryption<- Decryption
  Best_Log_Pl<- Log_Pl

  if(print==TRUE){

    print("----------------------")
    cat("Iteration : ",1, "\n")
    cat("Log_Plausibility : ", Log_Pl, "\n")
    cat("Decrypted Message : \n", Decrypted_Message, "\n")
  }

  if(is.null(Plaintext)==0){#If the plaintext is available
      Eff<- Efficiency(Decrypted_Message, Plaintext)
  }

  if(print==TRUE){
    cat("Efficiency : ", Eff, "\n")
```

```
  print("----------------------")
}

for(i in 2:Nb_Iter){

  #New proposal by switching randomly two values of the decryption key:
  s <- sample.int(Alphabet_size, 2 , replace = FALSE)
  Decryption2 <- Swap_Code(Decryption,s[1],s[2])
  Decrypted_Message2 <- Encryption(Decryption2,Ciphertext)

  #If we have already scored this decoding, we retrieve the plausibility from our map
  if (exists(paste(Decryption2, collapse =''), map)){
    Log_Pl2 <- map[[paste(Decryption2, collapse ='')]]

  }
  else{
    Log_Pl2 <- Log_Plausibility(Decrypted_Message2,Transition_mat1)
    map[[paste(Decryption2, collapse='')]] <- Log_Pl2
  }

  #The permutation changes with probability Pl2/Pl
  if(log(runif(1)) < (Log_Pl2 - Log_Pl)){
    Log_Pl <- Log_Pl2
    Decryption <- Decryption2
    Decrypted_Message<- Decrypted_Message2
    Stop<- 0

  }

  #Remembering the best log_plausibility
  if(Log_Pl>Best_Log_Pl){
    Best_Log_Pl<- Log_Pl
    Best<- Decrypted_Message
    Best_Decryption<- Decryption
  }

  if(is.null(Plaintext)==0){#If the plaintext is available

    Eff2<- Efficiency(Decrypted_Message, Plaintext)

    if(print==TRUE & ((i%%100==0 & Eff2>Eff)|i==100|i==200| i%%500==0 )){
      print("----------------------")
      cat("Iteration : ",i, "\n")
      cat("Log_Plausibility : ", Log_Pl, "\n")
      cat("Decrypted Message : \n", Decrypted_Message, "\n")
      cat("Efficiency : ", Eff, "\n")
      print("----------------------")
    }

    Eff<- Eff2

    if(Eff==1){ #Correct decryption
      Decrypted_Message <- Encryption(Decryption,Ciphertext)

      if(print==TRUE){
        print("Correct Decryption")
        cat("Iteration : ",i, "\n")
        cat("Log_Plausibility : ", Log_Pl, "\n")
        cat("Decrypted Message : \n", Decrypted_Message, "\n")
        cat("Original plaintext : \n ", Plaintext, "\n")
      }

      return(list(Best_Decryption, Eff)) #The algorithm stops if the correct decryption is
          found
    }

  }

}#end for

Best_Eff<- Efficiency(Best, Plaintext)

if(print==TRUE){
  print("Number of iterations reached")
  cat("Best Log_Plausibility", Best_Log_Pl, "\n")
  cat("Best Decrypted Message:  \n", Best, "\n")
```

```
    cat("Efficiency Best : ", Best_Eff, "\n")
    cat("Original plaintext : \n ", Plaintext, "\n")
  }

  return(list(Best_Decryption, Best_Eff))

}
```

→ This function implements the Metropolis-Hastings algorithm. It returns whichever decryption key from whichever iteration which gave the largest log-plausibility. The option `print=TRUE` must be added to print intermediate results.

Examples

```
Permutation <- 1:Alphabet_size #vector containing the integers from 1 to Alphabet_size
Permutation <- sample(Permutation) #random permutation of Permutation
```

Message 1: Austen's text Ch 1

Encryption

```
EncryptedMessage1 <- Encryption(Permutation,Message1)
```

Decryption using MH

```
#Frequencies of letters in the enciphered text
Freq1 <- rep(0,Alphabet_size)
Freq1 <- Char_Freq(EncryptedMessage1)

#Initial guess
Decryption1 <- Starting_Decryption(Ref_freq1,Freq1)
Encryption(Decryption1,EncryptedMessage1)

#Metropolis-Hastings algorithm
Decryption1 <- Decryption_MH(EncryptedMessage1,10000, Decryption1, Message1, print=TRUE)[[1]]
DecryptedMessage1 <- Encryption(Decryption1,EncryptedMessage1)
print(DecryptedMessage1)
```

Message 2: Ordinary love

Encryption

```
EncryptedMessage2 <- Encryption(Permutation,Message2)
```

Decryption using MH

```
#Frequencies of letters in the enciphered text
Freq2 <- rep(0,Alphabet_size)
Freq2 <- Char_Freq(EncryptedMessage2)

#Initial guess
Decryption2 <- Starting_Decryption(Ref_freq1,Freq2)
Encryption(Decryption2,EncryptedMessage2)

#Metropolis-Hastings algorithm
Decryption2 <- Decryption_MH(EncryptedMessage2,10000,Decryption2, Message2, print=TRUE)[[1]]
DecryptedMessage2 <- Encryption(Decryption2,EncryptedMessage2)
print(DecryptedMessage2)
```

Message 3: French text

Encryption

```
EncryptedMessage3 <- Encryption(Permutation,Message3)
```

Decryption using MH

```
#Frequencies of letters in the enciphered text
Freq3 <- rep(0,Alphabet_size)
Freq3 <- Char_Freq(EncryptedMessage3)

#Initial guess
Decryption3 <- Starting_Decryption(Ref_freq1,Freq3)
Encryption(Decryption3,EncryptedMessage3)

#Metropolis-Hastings algorithm
Decryption3 <- Decryption_MH(EncryptedMessage3,10000,Decryption3, Message3, print=TRUE )[[1]]
DecryptedMessage3 <- Encryption(Decryption3,EncryptedMessage3)
print(DecryptedMessage3)
```

Permutation/Transposition cipher

Encryption

```
Permutation_Cipher<- function(Text, m, see.permutation=FALSE){
  N<- nchar(Text)

  Mat_text<- Text_To_Matrix(Text,m)
  N<- nrow(Mat_text)*ncol(Mat_text)
  #random permutation
  Permutation<- sample(1:m)
  if(see.permutation == TRUE) print(Permutation)


  Mat_permutation<- matrix(0, nrow=N/m ,ncol=m)
  for(i in 1:m){
    Mat_permutation[,i]<- Mat_text[,Permutation[i]]
  }
  return(Mat_permutation)
}
```

→ This function encrypts a text using a permutation cipher.

Decryption

```
Decrypt_Permutation_Cipher<- function(Ciphertext,m,Inv_Decryption){

  Mat_text<- Text_To_Matrix(Ciphertext,m)

  N_mat<- nrow(Mat_text)*ncol(Mat_text)

  Mat_permutation<- matrix(0, nrow=N_mat/m ,ncol=m)
  for(i in 1:m){
    Mat_permutation[,i]<- Mat_text[,Inv_Decryption[i]]
  }

  Text<- Matrix_To_Text(Mat_permutation)

  return(Text)
}
```

→ This function decrypts a text encrypted with a transposition cipher.

Metropolis-Hastings algorithm

```
Slide<- function(Decryption, m){

  n<- rdunif(1,m-1,1)
  k1<- rdunif(1,m-n+1,1)
  k2<- rdunif(1,m-n,1) #numbers of shifts

  if(k2==m-n) return(Decryption) #The chain should be allowed to return itself

  if(k1+k2+n-1<=m){

    if(k1==1) return(c(Decryption[(k1+n):(k1+n+k2-1)], Decryption[k1:(k1+n-1)], Decryption[(k1
        +n+k2):m]))
    else if(k1+k2+n-1==m) return(c(Decryption[1:(k1-1)], Decryption[(k1+n):(k1+n+k2-1)],
        Decryption[k1:(k1+n-1)]))
    else return(c(Decryption[1:(k1-1)], Decryption[(k1+n):(k1+n+k2-1)], Decryption[k1:(k1+n-1)
        ], Decryption[(k1+n+k2):m]))

  }

  else{# cyclical modulo m
    copy2<- c(Decryption, Decryption)

    t<- k1+k2+n-1-m

    if(k1+n<=m) return(c(Decryption[1:(t)], Decryption[k1:(k1+n-1)],Decryption[(t+1):(k1-1)],
        Decryption[(k1+n):m] ))

    else return(c(Decryption[1:(t)], Decryption[k1:(k1+n-1)],Decryption[(t+1):(k1-1)]))
  }

}
```

→ This function selects a random contiguous sequence of decryption positions and inserts it at another position. The new proposal is to take a block of n decryption positions from position `k1` and to move it `k2` positions further. If needed, the operation is cyclical modulo m.

```
Check<- function(x){
  for(i in 1:length(x)) if(x[i]==0) return(1)

  return(0)

}
```

→ This function checks if all the elements of a vector are non zeroes. If there is at least one element which is 0, it returns 1. Else, it returns 0.

```
Efficiency_Perm<- function(Message, Text, m){

  n<- nchar(Text)
  Message<- substr(Message, 1, n) #avoid problems of added characters when the Text was
      encrypted

  Eff<- rep(1,ceiling(n/m)) #will contain the efficiency for all substring (each row)
  n_t<- m-2

  for(l in 1:ceiling(n/m)){
    m_t<- 0

    if( n%%m!= 0 & l==ceiling(n/m)){

      length<- n%%m #Message has the same length as Text (n=m)

      if(length==1){
        if(substr(Message, n,n)==substr(Text, n,n)) return(min(Eff))
        else if(length(unique(Eff[-ceiling(n/m)]))==1 & Eff[ceiling(n/m)] != Eff[1] & Eff
            [1]!=1) return(Eff[1])
        else return(min(Eff[-ceiling(n/m)]))
      }
```

```
        if(length==2 ){
          if(substr(Message, n-1,n-1)==substr(Text, n-1,n-1) & substr(Message, n,n)==substr(
              Text, n,n)) return(min(Eff))
          else if(length(unique(Eff[-ceiling(n/m)]))==1 & Eff[ceiling(n/m)] != Eff[1] & Eff
              [1]!=1) return(Eff[1])
          else return(min(Eff[-ceiling(n/m)]))
      }
      n_t<- n-((l-1)*m +1)+1-2 #length Text - position of the first letter of the last block
          +1 -2

      Decrypted_Message<- substr(Message, (l-1)*m +1,n)

      if(nchar(Decrypted_Message)==0){
        return(min(Eff))
      }
    }

    else{
      Decrypted_Message<- substr(Message, (l-1)*m+1, l*m)
      length<- m
    }

    for(i in 2:(length-1)){

        #Position of the i-th letter of Text in Decrypted_Message
        p<- gregexpr(substr(Text, (l-1)*m+i, (l-1)*m+i), Decrypted_Message)[[1]]

        if(length(p)==1){#if the letter appears only once
          if((substr(Decrypted_Message, p-1, p-1)==substr(Text, (l-1)*m+i-1,(l-1)*m+ i-1)) &
             (substr(Decrypted_Message, p+1, p+1)==substr(Text, (l-1)*m+i+1, (l-1)*m+i+1))){
            m_t<- m_t+1
          }
        }

        else{
          count<- 0
          for(j in 1:length(p)){
            if((substr(Decrypted_Message, p[j]-1, p[j]-1)==substr(Text, (l-1)*m+i-1, (l-1)*m+i
                -1)) &
               (substr(Decrypted_Message, p[j]+1, p[j]+1)==substr(Text, (l-1)*m+i+1,(l-1)*m+ i
                  +1)))
              if(count==0) count<- count+1
          }
          m_t<- m_t+count
        }
    }#end for (index i)

    Eff[l]<- m_t/n_t

  } #end for (index l)

  if(length(unique(Eff[-ceiling(n/m)]))==1 & Eff[ceiling(n/m)] != Eff[1] & Eff[1]!=1) return(
      Eff[1])
  #If all the substrings have the same efficiency (different from 1) except the last, it
      returns the efficiency of the first substring
  #If all the substrings have an efficiency of 1, except the last, it returns the efficiency
      of the last substring, if it  is not 1.

  if( Eff[ceiling(n/m)]==0 & Check(Eff[-ceiling(n/m)])==0) return(min(Eff[-ceiling(n/m)]))
  #If only the last substring has efficiency of 0, it returns the minimal efficiency of the
      other substrings.

  return(min(Eff))
}
```

→ This function computes the accuracy/efficiency of a decrypted text.

```
Decryption_Perm_MH<- function(Ciphertext,Nb_Iter,m,Text=NULL, print=FALSE ){

  Decryption<- 1:m #- the suggested inverse permutation is the identity permutation

  # instantiate a map to hold previously computed plausibilities
  map <- new.env(hash=T, parent=emptyenv())

  #First decryption
```

```
Inv_Decryption<- invPerm(Decryption)
Decrypted_Message <- Decrypt_Permutation_Cipher(Ciphertext, m, Inv_Decryption)

#Log-plausibility
Log_Pl <- Log_Plausibility_Perm(Decrypted_Message,m,Transition_mat1)
map[[paste(Inv_Decryption, collapse='')]] <- Log_Pl

#Decryption key with the largest log-plausibility
Best<-  Decrypted_Message
Best_Decryption<- Inv_Decryption
Best_Log_Pl<- Log_Pl

if(print==TRUE){

  print("_____________________")
  cat("Iteration : ",1, "\n")
  cat("Log_Plausibility : ", Log_Pl, "\n")
  cat("Inverse Permutation: ", Inv_Decryption, "\n")
  cat("Decrypted Message : \n", Decrypted_Message, "\n")
}


if(is.null(Text)==0){#If the plaintext is available
  Eff<- Efficiency_Perm(Decrypted_Message, Text, m)
  if(print==TRUE){
    print(Eff)
    print("_____________________")
  }
}

count<- 0 #numbers of changes

for(i in 2:Nb_Iter){

  #New proposal by selecting a random contiguous sequence of decryption positions and
  #inserting it in another position.
  Inv_Decryption2 <- Slide(Inv_Decryption,m)
  Decrypted_Message2 <- Decrypt_Permutation_Cipher(Ciphertext, m,Inv_Decryption2)

  #If we have already scored this decoding, we retrieve the plausibility from our map
  if (exists(paste(Inv_Decryption2, collapse =''), map)){
    Log_Pl2 <- map[[paste(Inv_Decryption2, collapse ='')]]
  }
  else{
    Log_Pl2 <- Log_Plausibility_Perm(Decrypted_Message2,m,Transition_mat1)
    map[[paste(Inv_Decryption2, collapse='')]] <- Log_Pl2
  }

  if(log(runif(1)) < (Log_Pl2 - Log_Pl)){ #The decryption changes
    count<- count+1

    Log_Pl <- Log_Pl2
    Inv_Decryption <- Inv_Decryption2
    Decrypted_Message<- Decrypted_Message2

  }

  #Remembering the best log_plausibility
  if(Log_Pl>Best_Log_Pl){
    Best_Log_Pl<- Log_Pl
    Best<- Decrypted_Message
    Best_Decryption<- Inv_Decryption
  }


  if(is.null(Text)==0){#If the plaintext is available

    Eff2<- Efficiency_Perm(Decrypted_Message, Text, m)

    if(print==TRUE & ((i%%100==0 & Eff2>Eff)|i==100|i==200| i%%500==0 )){
      print("_____________________")
      cat("Iteration : ",i, "\n")
      cat("Log_Plausibility : ", Log_Pl, "\n")
      cat("Decrypted Message : \n", Decrypted_Message, "\n")
      cat("Efficiency : ", Eff, "\n")
      print("_____________________")
```

```
      }

      Eff<- Eff2

      if(Eff==1){

        if(print==TRUE){
          print("Correct Decryption")
          cat("Iteration : ",i, "\n")
          cat("Log_Plausibility : ", Log_Pl, "\n")
          cat("Decrypted Message : \n", Decrypted_Message, "\n")
          cat("Original plaintext : \n ", Text, "\n")
        }

        return(list(Inv_Decryption, Log_Pl, Eff)) #The algorithm stops if the correct
            decryption is found
      }

    }

  }#end for

  if(print==TRUE){
    print("Number of iterations reached")
    cat("Best Log_Plausibility", Best_Log_Pl, "\n")
    cat("Best Decrypted Message:  \n", Best, "\n")
    cat("Original plaintext : \n ", Text, "\n")
    cat("Count: ", count, "\n")

  }
  return(list(Inv_Decryption, Log_Pl, Eff))

}
```

→ This function implements the Metropolis-Hastings algorithm. It returns whichever decryption key from whichever iteration which gave the largest log-plausibility. The option `print=TRUE` must be added to print intermediate results.

Examples

Message 2: Ordinary love

Encryption

```
EncryptedMessage2<- Permutation_Cipher(Message2, 13, see.permutation = TRUE)
EncryptedMessage2<- Matrix_To_Text(EncryptedMessage2)
```

Decryption using MH

```
MH2<- Decryption_Perm_MH(EncryptedMessage2, 20000, 13, Message2, print = TRUE)
Decrypt_Permutation_Cipher(EncryptedMessage2, 13, MH2[[1]])
```

Unknown period

```
Period_transposition<- function(ciphertext,min, max, nb_iter, text, print=FALSE){
  Log_Pl<- rep(0, (max-min+1))

  for(i in (min):max){
    Log_Pl[i-min+1]<- Decryption_Perm_MH(ciphertext, nb_iter, i, text)[[2]]
  }


  if(print==TRUE) for(i in (min):max) cat("Iteration ", i, ": Log-plausibility: ",  Log_Pl[i-
      min+1], "\n")
```

```
  return(which.max(Log_Pl)+min-1)
}
```

→ This function finds the most probable period of the cipher. For each possible period k from `min` to `max` (with `min`≥ 3), the entire Metropolis-Hastings algorithm is run to obtain the best score function for that k, P_k. The key is the value of k which leads to the best log-plausibility function, i.e. $k = argmax_k P_k$. The option `print=TRUE` must be added to print intermediate results.

Example

Message 2: Ordinary love

```
Period_transposition(EncryptedMessage2, 3,20,100,Message2, print=TRUE)
```

Chapter 5

Database

```
database <- read_file("text_database.txt")
database<- Format_Text_NP(database)
print(substr(database, 1, 500))

database_size<- nchar(database) #Number of characters
database_size
```

Vigenère cipher

Generate a ciphertext

```
Random_selection<- function(Length){
  n<- rdunif(1,database_size-Length,1)
  #cat("n : ", n , "\n ")
  text<- substring(database,n,n+Length)
  #cat("Selected text : \n", text, "\n")
  return(text)
}
```

→ This function selects a random portion of consecutive characters from the database of texts.

Generate a ciphertext

```
Random_keyword<- function(max_m){
  #random period
  m<- rdunif(1,max_m,1)
  #cat("Selected period : \n", m, "\n")

  #random keyword
  keyword<- NULL
  for(i in 1:m){
    keyword[i]<- rdunif(1,0,26)
  }
  keyword<- Num_To_Char(keyword)
  return(keyword)
}
```

→ This function generates a random keyword whose length is less than a specific period `max_m`.

Comparison of the techniques

```
Comparison_accuracy<- function(nb_iterations, Length,  max_m, Ref_freq, print=FALSE){
  FA_accuracy<- rep(0, nb_iterations)
  Chi2_accuracy<- rep(0, nb_iterations)
  Turing_accuracy<- rep(0, nb_iterations)

  FA_duration<- rep(0, nb_iterations)
  Chi2_duration<- rep(0, nb_iterations)
  Turing_duration<- rep(0, nb_iterations)

  FA_less<- 0
  Chi2_less<- 0
  Turing_less<- 0

  for(i in 1:nb_iterations){
    #Text selection
    Text<- Random_selection(Length)
    if(print==TRUE) cat("Selected text : \n", Text, "\n")
```

```
#Random keyword
keyword<- Random_keyword(max_m)

m<- nchar(keyword)
if(print==TRUE){
  cat("Selected period : \n", m, "\n")
  cat("Selected keyword : \n", keyword, "\n")
}
#ciphertext
Ciphertext<- Vigenere_cipher(Text, keyword)

if(print==TRUE) cat("ciphertext: \n", Ciphertext, "\n")

#Find the period
#Frequency analysis
Mat<- Col_freq(Ciphertext,m, plot=FALSE)
#print(Mat)

start_time <- Sys.time()
FA_keyword<- Find_Key_Vig_FA(Ciphertext, m, Mat)
end_time <- Sys.time()
FA_duration[i]<- end_time - start_time

if(print==TRUE){
  cat("Duration: ", FA_duration[i], "\n")
  cat("Frequency analysis keyword : \n",  FA_keyword, "\n")
}

#Chi2
start_time <- Sys.time()
Chi2_keyword<- Find_Key_Vig_Chi2(Ciphertext, m, Ref_freq)
end_time <- Sys.time()
Chi2_duration[i]<- end_time - start_time

if(print==TRUE){
  cat("Duration: ", Chi2_duration[i], "\n")
  cat("Chi2 keyword : \n",  Chi2_keyword, "\n")
}

#Turing
start_time <- Sys.time()
Turing_keyword<- Find_Key_Turing(Ciphertext, m, Ref_freq)
end_time <- Sys.time()
Turing_duration[i]<- difftime(end_time, start_time, units="secs")

if(print==TRUE){
  cat("Duration: ", Turing_duration[i], "\n")
  cat("Turing keyword : \n",  Turing_keyword, "\n")
}

FA_counts<- 0
Chi2_counts<- 0
Turing_counts<- 0

for(j in 1:m){

  if(substr(FA_keyword,j,j)==substr(keyword,j,j)) FA_counts<- FA_counts+1
  if(substr(Chi2_keyword,j,j)==substr(keyword,j,j)) Chi2_counts<- Chi2_counts+1
  if(substr(Turing_keyword,j,j)==substr(keyword,j,j)) Turing_counts<- Turing_counts+1
}

FA_accuracy[i]<- sum(FA_counts)/m
Chi2_accuracy[i]<- sum(Chi2_counts)/m
Turing_accuracy[i]<- sum(Turing_counts)/m

  if(FA_accuracy[i]<0.5|Chi2_accuracy[i]<0.5|Turing_accuracy[i]<0.5){
    print("_____________________________________________")
    cat("The correct keyword was :", keyword, "\n")
    }
  if(FA_accuracy[i]<0.5){
    cat("FA_accuracy<0.5 for the iteration ", i, "\n the keyword found is : ", FA_keyword,
        "\n and the selected text was : \n", Text, "\n")
    FA_less<- FA_less +1
    }
    if(Chi2_accuracy[i]<0.5){
```

```
          cat("Chi2_accuracy<0.5 for the iteration ", i, "\n the keyword found is : ", Chi2_
              keyword,  "\n  and the selected text was : \n", Text,  "\n")
          Chi2_less<- Chi2_less +1
          }
        if(Turing_accuracy[i]<0.5){
          cat("Turing_accuracy<0.5 for the iteration ", i, "\n the keyword found is : ",
              Turing_keyword,  "\n and the selected text was : \n", Text, "\n")
          Turing_less<- Turing_less +1
        }

  }
  cat("Percentage of bad decryption FA: ", FA_less/100, "\n")
  cat(" Percentage of bad decryption Chi2: ", Chi2_less/100, "\n")
  cat(" Percentage of bad decryption Turing: ", Turing_less/100 , "\n")

  write.table(list(FA_accuracy, FA_duration, Chi2_accuracy, Chi2_duration, Turing_accuracy,
      Turing_duration),file=paste("Comparison",  Length, ".txt", sep=""), col.names = FALSE)

  return(list(FA_accuracy, FA_duration, Chi2_accuracy, Chi2_duration, Turing_accuracy, Turing_
      duration))
}
```

→ This function compares the three techniques developed in Chapter 1,2 and 3 to break Vigenère cipher. It randomly chooses and encrypts a text `nb_iterations` times. For each encryption, the three techniques will be run. The maximum period allowed for the Vigenère cipher is `max_m`. An option `print=TRUE` must be added in order to print intermediate results.

Results

```
for(i in c(50,100,200,500,1000) ){
  results<-  file(paste("Text",  i, ".txt", sep=""), open = "wt")  #keep results on a .txt
      file
  sink(results)
  Acc<- Comparison_accuracy(100,i,15,Ref_freq1)
  print("____________________")
 cat( "Length of text: ",  i, "\n")
 cat("FA: \n Success : ", sum(Acc[[1]][Acc[[1]]==1]), "\n FA_Accuracy:  ", round(mean(Acc
     [[1]]),4), "Duration: ", sum(Acc[[2]]), "\n")
 cat("Chi2: \n Success : ", sum(Acc[[3]][Acc[[3]]==1]), "\n Chi2_Accuracy:  ", round(mean(Acc
     [[3]]),4),  "Duration: ", sum(Acc[[4]]), "\n")
 cat("Turing: \n Success : ", sum(Acc[[5]][Acc[[5]]==1]), "\n Turing_Accuracy:  ", round(mean(
     Acc[[5]]),4),  "Duration: ", sum(Acc[[6]]), "\n")
 cat("Less duration: Method ", which.min(c( mean(Acc[[2]]),  mean(Acc[[4]]),  mean(Acc[[6]])))
     , "\n")
 print("____________________")
sink()
 }

# FA_accuracy, FA_duration, Chi2_accuracy, Chi2_duration, Turing_accuracy, Turing_duration
Comparison50<- read.table("Comparison50.txt")
Comparison100<- read.table("Comparison100.txt")
Comparison200<- read.table("Comparison200.txt")
Comparison500<- read.table("Comparison500.txt")
Comparison1000<- read.table("Comparison1000.txt")

#Standard-error
#Freq analysis
round(c(sd(Comparison50$V2)/sqrt(length(Comparison50$V2)),
       sd(Comparison100$V2)/sqrt(length(Comparison100$V2)),
       sd(Comparison200$V2)/sqrt(length(Comparison200$V2)),
       sd(Comparison500$V2)/sqrt(length(Comparison500$V2)),
       sd(Comparison1000$V2)/sqrt(length(Comparison1000$V2))), 4)

#Chi2
round(c(sd(Comparison50$V4)/sqrt(length(Comparison50$V4)),
        sd(Comparison100$V4)/sqrt(length(Comparison100$V4)),
        sd(Comparison200$V4)/sqrt(length(Comparison200$V4)),
        sd(Comparison500$V4)/sqrt(length(Comparison500$V4)),
        sd(Comparison1000$V4)/sqrt(length(Comparison1000$V4))), 4)

#Turing
round(c(sd(Comparison50$V6)/sqrt(length(Comparison50$V6)),
        sd(Comparison100$V6)/sqrt(length(Comparison100$V6)),
```

```
        sd(Comparison200$V6)/sqrt(length(Comparison200$V6)),
        sd(Comparison500$V6)/sqrt(length(Comparison500$V6)),
        sd(Comparison1000$V6)/sqrt(length(Comparison1000$V6))), 4)


#Freq analysis
boxplot(Comparison50$V2,Comparison100$V2, Comparison200$V2, Comparison500$V2, Comparison1000$
    V2, main="", names=c(50,100,200,500,1000), ylab= "Accuracy", xlab="Length")
title(main=paste("Box plots of the accuracy of frequency analysis method ","\n",sep=""),cex.
    main=1)
title(main=paste("\n","for different lengths of text",sep=""),cex.main=1)


#Chi2
boxplot(Comparison50$V4,Comparison100$V4, Comparison200$V4, Comparison500$V4, Comparison1000$
    V4, main="", names=c(50,100,200,500,1000), ylab= "Accuracy", xlab="Length")
title(main=paste("Box plots of the accuracy of Chi-squared method","\n",sep=""),cex.main=1)
title(main=paste("\n","for different lengths of text",sep=""),cex.main=1)

#Turing
boxplot(Comparison50$V6,Comparison100$V6, Comparison200$V6, Comparison500$V6, Comparison1000$
    V6, main="", names=c(50,100,200,500,1000), ylab= "Accuracy", xlab="Length")
title(main=paste("Box plots of the accuracy of Turing's method","\n",sep=""),cex.main=1)
title(main=paste("\n","for different lengths of text",sep=""),cex.main=1)


#Comparison of the 3 techniques
#Length: 50
boxplot(Comparison50$V2,Comparison50$V4,Comparison50$V6, main="", names=c("Freq. analysis", "
    Chi-squared", "Turing"), ylab= "Accuracy", xlab="Method")
title(main=paste("Box plots of the accuracy for different methods","\n",sep=""),cex.main=1)
title(main=paste("\n","and a length of text of 50",sep=""),cex.main=1)

#Length 100
boxplot(Comparison100$V2,Comparison100$V4,Comparison100$V6, main="", names=c("Freq. analysis",
     "Chi-squared", "Turing"), ylab= "Accuracy", xlab="Method")
title(main=paste("Box plots of the accuracy for different methods","\n",sep=""),cex.main=1)
title(main=paste("\n","and a length of text of 100",sep=""),cex.main=1)

#Length 200
boxplot(Comparison200$V2,Comparison200$V4,Comparison200$V6, main="", names=c("Freq. analysis",
     "Chi-squared", "Turing"), ylab= "Accuracy", xlab="Method")
title(main=paste("Box plots of the accuracy for different methods","\n",sep=""),cex.main=1)
title(main=paste("\n","and a length of text of 200",sep=""),cex.main=1)

#Length 500
boxplot(Comparison500$V2,Comparison500$V4,Comparison500$V6, main="", names=c("Freq. analysis",
     "Chi-squared", "Turing"), ylab= "Accuracy", xlab="Method")
title(main=paste("Box plots of the accuracy for different methods","\n",sep=""),cex.main=1)
title(main=paste("\n","and a length of text of 500",sep=""),cex.main=1)

#Length 1000
boxplot(Comparison1000$V2,Comparison500$V4,Comparison1000$V6, main="", names=c("Freq. analysis
    ", "Chi-squared", "Turing"), ylab= "Accuracy", xlab="Method")
title(main=paste("Box plots of the accuracy for different methods","\n",sep=""),cex.main=1)
title(main=paste("\n","and a length of text of 1000",sep=""),cex.main=1)
```

Substitution

```
Accuracy_Substitution<- function(p, Nb_Iter, length, print=FALSE){

  success<- 0 #successful runs
  Accuracy<- rep(0,p) #Accuracy for each run

  duration<- rep(0,p)

  for(i in 1:p){

    if(print==TRUE){
      print("_________________")
      cat("Accuracy iteration :", i, "\n" )
    }
```

```
    #random portion of text
    selected_text<- Random_selection(length)

    if(print==TRUE)cat("Selected text: \n ",selected_text, "\n")

    #random encryption
    Permutation <- 1:Alphabet_size
    Permutation <- sample(Permutation) #random permutation of Permutation
    Encryptedselected_text<- Encryption(Permutation,selected_text)

    if(print==TRUE){
      cat("Permutation: ", Permutation, "\n")
      cat("Encrypted selected text: \n ", Encryptedselected_text, "\n")
    }

    #Decryption using MH

    start_time <- Sys.time()
    #Frequencies of letters in the enciphered text
    Freq <- rep(0,Alphabet_size)
    Freq <- Char_Freq(Encryptedselected_text)


    #Initial guess
    Decryption <- Starting_Decryption(Ref_freq1,Freq)
    # Encryption(Decryption,Encryptedselected_text)

    #Metropolis-Hastings algorithm
    MH<- Decryption_MH(Encryptedselected_text,Nb_Iter,Decryption,selected_text, print)
    end_time <- Sys.time()
    duration[i] <- difftime(end_time, start_time, units="secs")

    Decryption<- MH[[1]]


    DecryptedMessage <- Encryption(Decryption,Encryptedselected_text)
    if(print==TRUE) print(DecryptedMessage)

    #Accuracy
    Accuracy[i]<-  MH[[2]]
    success<- length(Accuracy[Accuracy==1])


    write.table(list(Accuracy, duration),file=paste("Substitution", length, ".txt", sep=""),
        col.names = FALSE)

    if(print==TRUE){
      cat("Accuracy[i] : ", Accuracy[i], "\n")
      cat("Accuracy vector : ", Accuracy, "\n")
      cat("Number of successful runs : ", success, "\n")
    }

  }

  return( list(mean(Accuracy),success, sum(duration)) )
}
```

→ This function measures the performance of the Metropolis-Hastings algorithm for substitution cipher. It randomly chooses and encrypts a text `p` times. An option `print=TRUE` must be added in order to print intermediate results.

Simulations

```
for(i in c(50,100,200,500,1000) ){
  Acc<- Accuracy_Substitution(100, 10000, i)
  print("--------------------")
  cat( "Length of text: ",  i, "\n")
  cat("Success : ",Acc[[2]], "\n Accuracy:  ", round(Acc[[1]],4), "Duration: ", Acc[[3]], "\n"
      )

  print("--------------------")
}
```

Results

```
Substitution50<- read.table("Substitution50.txt")
Substitution100<- read.table("Substitution100.txt")
Substitution200<- read.table("Substitution200.txt")
Substitution500<- read.table("Substitution500.txt")
Substitution1000<- read.table("Substitution1000.txt")

ES<- c(sd(Substitution50$V2)/sqrt(length(Substitution50$V2)),
       sd(Substitution100$V2)/sqrt(length(Substitution100$V2)),
       sd(Substitution200$V2)/sqrt(length(Substitution200$V2)),
       sd(Substitution500$V2)/sqrt(length(Substitution500$V2)),
       sd(Substitution1000$V2)/sqrt(length(Substitution1000$V2)))
round(ES, 4)

boxplot(Substitution50$V2,Substitution100$V2, Substitution200$V2, Substitution500$V2,
    Substitution1000$V2, main="Box plots of the accuracy for different lengths of text", names
    = c(50,100,200,500,1000), ylab= "Accuracy", xlab="Length")

mean(Substitution50$V2<0.5)
mean(Substitution100$V2<0.5)
mean(Substitution200$V2<0.5)
mean(Substitution500$V2<0.5)
mean(Substitution1000$V2<0.5)
```

Transposition cipher

```
Accuracy_Permutation<- function(p,max_m, Nb_Iter, Length, print=FALSE){

  success<- 0 #successful runs
  Accuracy<- rep(0,p) #Accuracy for each run

  duration<- rep(0,p)

  for(i in 1:p){

    if(print==TRUE){
      print("_________________")
      cat("Accuracy iteration :", i, "\n" )
    }

    #random portion of text
    selected_text<- Random_selection(Length)

    if(print==TRUE) cat("Selected text: \n ",selected_text, "\n")

    #random period
    m<- rdunif(1,max_m,2)

    if(print==TRUE) cat("Period:", m, "\n")

    Encryptedselected_text<- Permutation_Cipher( selected_text, m)
    Encryptedselected_text<- Matrix_To_Text(Encryptedselected_text)

    if(print==TRUE) cat("Encrypted selected text: \n ", Encryptedselected_text, "\n")

    #Decryption using MH
    #Metropolis-Hastings algorithm

    start_time <- Sys.time()

    if(Encryptedselected_text==selected_text){ #if the text was not encrypted a single
        iteration of MH will be sufficient.
      Accuracy[i]<- 1
      next; #go to iteration i+1
    }


    #If the period is 2 and the text was encrypted, then the text will be decrypted using
        permutation (21)
    test<- Decrypt_Permutation_Cipher(Encryptedselected_text, 2, 2:1)
    if(test==selected_text){
```

```
      Accuracy[i]<- 1
      next; #go to iteration i+1
    }


      m_supposed<- Period_transposition(Encryptedselected_text, 3,max_m,100,Message2)
      if(print==TRUE) cat("Supposed period:", m_supposed, "\n")
      MH<- Decryption_Perm_MH( Encryptedselected_text, Nb_Iter,  m_supposed,  selected_text,
          print )
      end_time <- Sys.time()
      duration[i] <- difftime( end_time, start_time, units = "secs")

      DecryptedMessage<- Decrypt_Permutation_Cipher( Encryptedselected_text,  m_supposed, MH
          [[1]])
      if(print==TRUE) print(DecryptedMessage)

      #Accuracy
      Accuracy[i]<-  MH[[3]]
      #success
      success<- length(Accuracy[Accuracy==1])

    write.table(list(Accuracy, duration),file=paste("Transposition", Length, ".txt", sep=""),
        col.names = FALSE)

    if(print==TRUE){
      cat("Accuracy[i] : ", Accuracy[i], "\n")
      cat("Accuracy vector : ", Accuracy, "\n")
      cat("Number of successful runs : ", success, "\n")
    }
  }

  return( list(mean(Accuracy),success, sum(duration)) )
}
```

→ This function measures the performance of the Metropolis-Hastings algorithm for transposition cipher. It randomly chooses and encrypts a text, `p` times. The maximum period allowed for the transposition cipher is `max_m`. An option `print=TRUE` must be added in order to print intermediate results.

Simulations

```
for(i in c(50,100,200,500,1000) ){
   print(i)
  Acc<- Accuracy_Permutation(50,15,20000, i)
  print("____________________")
  cat( "Length of text: ",  i, "\n")
  cat("Success : ",Acc[[2]], "\n Accuracy:  ", round(Acc[[1]],4), "Duration: ", Acc[[3]], "\n"
      )

  print("____________________")
}
```

Results

```
Transposition50<- read.table("Transposition50.txt")
Transposition100<- read.table("Transposition100.txt")
Transposition200<- read.table("Transposition200.txt")
Transposition500<- read.table("Transposition500.txt")
Transposition1000<- read.table("Transposition1000.txt")

ES<- c(sd(Transposition50$V2)/sqrt(length(Transposition50$V2)),
       sd(Transposition100$V2)/sqrt(length(Transposition100$V2)),
       sd(Transposition200$V2)/sqrt(length(Transposition200$V2)),
       sd(Transposition500$V2)/sqrt(length(Transposition500$V2)),
       sd(Transposition1000$V2)/sqrt(length(Transposition1000$V2)))
round(ES, 4)

mean(Transposition50$V2<0.5)
mean(Transposition100$V2<0.5)
mean(Transposition200$V2<0.5)
```

```
mean(Transposition500$V2<0.5)
mean(Transposition1000$V2<0.5)

boxplot(Transposition50$V2,Transposition100$V2, Transposition200$V2, Transposition500$V2,
    Transposition1000$V2, main="Box plots of the accuracy for different lengths of text",
    names=c(50,100,200,500,1000), ylab= "Accuracy", xlab="Length")
```

Appendix B

Lorenz addition

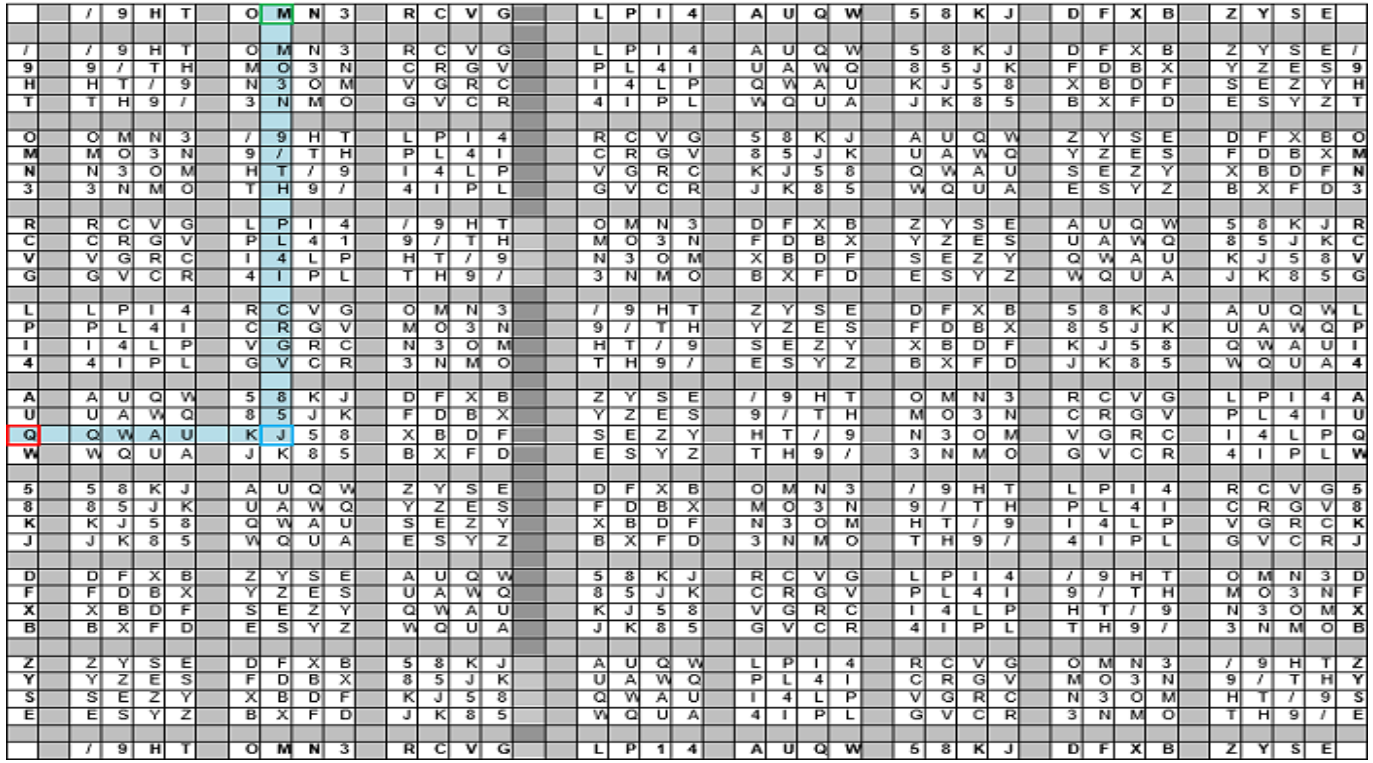


Figure B.1: Table representing the Lorenz addition of two letters. In this figure, the addition of the letter Q and the letter M is shown in blue.

Bibliography

[1] BBC. Code-Breakers Bletchley Park's Lost Heroes. Available via the URL <https://www.dailymotion.com/video/x5g8ie9> (viewed on 11 june 2018).

[2] BHATEJA, Ashok K., Aditi BHATEJA, Santanu CHAUDHURY and P.K. SAXENA. Cryptanalysis of vigenere cipher using cuckoo search. *Applied Soft Computing Journal*. 2015, 26, p. 315–324.

[3] BOHM, W. *Elementary Methods of Cryptography*. University of Vienna, 2016.

[4] CHEN, J. and S. ROSENTHAL. Decrypting classical cipher using Markov Chain Monte Carlo. *Stat. Comput.* 2012, 22(2), p. 397–413.

[5] CONNOR, Stephen B. Simulation and Solving Substitution Codes, 2002. MMathStat Dissertation (3rd year).

[6] COPELAND, J. Alan Turing: The codebreaker who saved "millions of lives". Available via the URL <https://www.bbc.com/news/technology-18419691> (viewed on 11 june 2018).

[7] CRANE, H. Sandy Zabell: Alan Turing and the Applications of Probability to Cryptography (Rutgers). Available via the URL <https://www.youtube.com/watch?v=vQKMdHnCgrs> (viewed on 17 february 2019).

[8] DIACONIS, Persi. The Markov Chain Monte Carlo revolution. *Bull. Am. Math. Soc.* 2009, 46(2), p. 179–205.

[9] DOBROW, R. Probability with applications and R. 2014.

[10] FATHI-VAJARGAH, B. and M. KANAFCHIAN. Decrypting Substitution-Transposition Cipher Using Monte Carlo Method Based on Sobol quasi random generator. *International Journal of Computer Science and Information Security (IJCSIS)*. 9 2016, 14.

[11] FUND, The Bill Tutte Memorial. Teleprinter Code. Available via the URL <https://billtuttememorial.org.uk/codebreaking/teleprinter-code/> (viewed on 11 june 2018).

[12] GOMEZ, J. *Mathématiques, espionnage et piratage informatique - Codage et cryptographie*. 2013. (Le monde est mathématique). ISBN 9782823700992.

[13] GRIME, J. 158,962,555,217,826,360,000 (Enigma Machine) - Numberphile. Available via the URL <https://www.youtube.com/watch?annotation_id=annotation_707724&feature=iv&src_vid=V4V2bpZlqx8&v=G2_Q9FoD-oQ> (viewed on 11 june 2018).

[14] GRIME, J. Flaw in the Enigma Code - Numberphile. Available via the URL <https://www.youtube.com/watch?v=V4V2bpZlqx8> (viewed on 11 june 2018).

[15] GRIME, J. Lorenz: Hitler's "Unbreakable" Cipher Machine. Available via the URL <https://www.youtube.com/watch?v=GBsfWSQVtYA> (viewed on 11 june 2018).

[16] GRIME, J. *An Introduction to Cryptography.* 2016.

[17] HODGES, A. *Alan Turing: The Enigma.* Princeton, N.J.: Princeton University Press, 2012.

[18] JOHANSSON, T. *Lecture 5: Classical cryptography.* Lund University.

[19] KAHN, D. *The codebreakers, The Story of Secret Writing.* The New American Library, Inc., 1973.

[20] KOCMàNEK, T. MCMC Decryption, 2013. Bachelor's Thesis.

[21] LYONS, J. Chi-squared Statistic. Available via the URL <http://practicalcryptography.com/cryptanalysis/text-characterisation/chi-squared-statistic/> (viewed on 10 march 2019).

[22] LYONS, J. Cryptanalysis of the Vigenere Cipher. Available via the URL <http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-vigenere-cipher/> (viewed on 10 march 2019).

[23] LYONS, J. Index of Coincidence. Available via the URL <http://practicalcryptography.com/cryptanalysis/text-characterisation/index-coincidence/> (viewed on 10 march 2019).

[24] MEALING, B. *Bletchley Park Home of the Codebreakers.* Pitkin Publishing, 2014. ISBN 9781841655932.

[25] MIJOULE, G. *Chapitre IV : Tests d'hypothèse.* Université de Liège, 2016-2017.

[26] MOORE, K. et al. Enigma Machine. Brilliant Organisation. Available via the URL <https://brilliant.org/wiki/enigma-machine/> (viewed on 17 february 2019).

[27] RIGO, M. *Mathématiques discètes (3) : Cryptosystèmes historiques.* Université de Liège, 2016-2017.

[28] SALE, T. *The Bletchley Park 1944 Cryptographic Dictionary.* 2001.

[29] SARKAR, Palash. On some connections between statistics and cryptology. *Journal of Statistical Planning and Inference.* 2014, 148, p. 20–37.

[30] SWAN, Y. *Large Sample Analysis : A peak at Markov Chain theory.* University of Liège, 2017-2018.

[31] TURING, A. M. Paper on Statistics of Repetitions. 2012. Unpublished paper, c. 1941, UK National Archives, HW 25/38.

[32] TURING, A. M. The Applications of Probability to Cryptography. 2012. Unpublished paper, c. 1941, UK National Archives, HW 25/37.

[33] WIKIPEDIA. Banburismus. Wikimedia Foundation. Available via the URL <https://en.wikipedia.org/wiki/Banburismus> (viewed on 19 novembre 2018).

[34] WIKIPEDIA. Lorenz cipher. Wikimedia Foundation. Available via the URL <https://en.wikipedia.org/wiki/Lorenz_cipher> (viewed on 11 june 2018).

[35] WIKIPEDIA. One-time pad. Wikimedia Foundation. Available via the URL <https://fr.wikipedia.org/wiki/One-time_pad> (viewed on 11 june 2018).

[36] WIKIPEDIA. Smith's Prize. Wikimedia Foundation. Available via the URL <https://en.wikipedia.org/wiki/Smith%27s_Prize> (viewed on 11 june 2018).

[37] ZABELL, Sandy. Commentary on Alan M. Turing: The Applications of Probability to Cryptography. *Cryptologia*. 2012, 36(3), p. 191–214.