Enigma: A Maple Worksheet Implementation

by

John Darren Mummery

Bachelor of Computer Science, UNB, 2009

A Report Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Rodney Harold Cooper, MMath, Computer Science Patricia Evans, Ph.D, Computer Science
Examining Board:	Rongxing Lu, Ph.D, Computer Science, Chair Andrew McAllister, Ph.D, Computer Science

This report is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2021

© John Darren Mummery, 2021

Abstract

The Enigma cipher was utilized by Germany before and during the Second World War. When teaching cryptography, it is best to pick an easily understood example that is representative of actual encryption schemes and display the flaws within it that can be exploited in order to introduce students to the mathematical concepts involved. Enigma is still simple enough that students can trace through the steps by hand, but can also be used to establish a foundation which can ultimately lead to these modern cryptosystems.

For this report, two Maple programs have been created: An encryption program and a decryption program. The decryption program utilizes the card catalog to determine the possible rotor positions and then attempts to solve the plugboard wirings by comparing the unencrypted characteristics with their encrypted forms. If this fails, then all rotor positions are iterated through and attempts are made to solve the plugboard for each one through heavy recursion. This recursive method can handle all possible plugboard settings using any number of leads up to the full 13. Student comprehension of the concepts involved is the primary goal of this work. The advantage of

the use of Maple worksheets compared to other methods is that it is much simpler to follow and step through the program utilizing Maple's internal debugger compared to some black box approaches.

Dedication

This thesis is dedicated to Jill (the cat), who will forever be the picture of comfort as she napped in her bed by the fire.

Acknowledgements

I wish to acknowledge the labor and effort Sonya Leah Clark, and her parents for sticking by me and believing in me through this very difficult time and when I thought no one else would. I also want to give my appreciation to Professor Rodney Harold Cooper for providing the initial idea and being extremely patient with me despite his own difficulties during this time. I also wish to thank Doctor Patricia Evans for her guidance and assistance for helping me across the finish line when I've been frequently felt stuck and unsure how to best proceed.

Table of Contents

Abstract **ii**

Dedication **iv**

Acknowledgments **v**

Table of Contents **vi**

List of Figures **xi**

1 Introduction and Background **1**

1.1 Overview 1

1.2 Background 3

1.3 Comparisons to Related Work 5

2 Defining Enigma **8**

2.1 Historical Overview 8

2.2 Enigma Overview 11

2.2.1 Machine Components 11

2.2.1.1 Keyboard and Lampboard 11

2.2.1.2 Entry Wheel 13
2.2.1.3 Rotors 13
2.2.1.4 Reflector 15
2.2.1.5 Plugboard 16
2.2.2 Electromechanical Process 17
2.2.3 Symmetric Encryption Key 20
2.2.3.1 Daily Setting 21
2.2.3.2 Message Setting 22

3 Unraveling Enigma 24

3.1 Representation of the Machine 24
3.1.1 Permutation Primer 24
3.1.1.1 What is a Permutation? 25
3.1.1.2 What is a Function? 25
3.1.1.3 Permutations 29
3.1.1.4 Groups 29
3.1.1.5 Cycle Notation 31
3.1.2 Representing the Rotors, Plugboard, and Reflector . . 32
3.1.3 Representing the Rotor Rotations 34
3.2 Double Encipherment 38
3.3 Cycle Structure 42
3.4 Constructing a Catalog 43
3.5 Plugboard Wirings 44

3.6 Recursive Algorithm 46

4 **Solving and Implementing Enigma** **47**

4.1 Introduction 47

4.2 Library 48

4.2.1 Convert Letters to Numerical Offsets 49

4.2.2 Generate Rotor Rotation Permutations 50

4.2.3 Construct Enigma Encryption Permutation 51

4.2.4 Step Rotors 52

4.2.5 Encrypt Letter 52

4.2.6 Calculate Index into Catalog 53

4.2.7 Find All Possible Solutions for the Plugboard 54

4.3 Encryption Program 55

4.3.1 Overview 55

4.3.2 Program Setup: 56

4.3.2.1 Import Library 56

4.3.2.2 Import Language and Machine Settings . . . 56

4.3.2.3 Generate Rotor Rotations 61

4.3.3 Message Setup 63

4.3.3.1 Setup Message Settings and Plaintext 63

4.3.3.2 Double Encrypt Operator Key 65

4.3.4 Encryption and Output 67

4.3.4.1 Encrypt Plaintext with Operator key 67

4.3.4.2 Convert into Ciphertext 68
4.3.4.3 Save Characteristic Sets AD, BE, CF and Ciphertext 70
4.4 Decryption Program 72
4.4.1 Overview 73
4.4.2 Program Setup: 73
4.4.2.1 Import Library 73
4.4.2.2 Import Language and Machine Settings . . . 74
4.4.2.3 Import Ciphertext and Cycle Structures . . . 76
4.4.2.4 Generate Rotor Rotations 79
4.4.2.5 Convert Ciphertext into Alphabet Offsets . . 80
4.4.3 Catalog Decryption 81
4.4.3.1 Extract Cycle Structure from AD, BE, and CF 82
4.4.3.2 Create the Catalog 84
4.4.3.3 Search the Catalog 88
4.4.4 Solving the Plugboard 91
4.4.4.1 Derive Plugboard by Simple Comparison . . . 91
4.4.4.2 Derive Plugboard Recursively for All Rotor Settings 100
4.4.4.3 Determine Plugboard Wirings 106

5 Conclusion **125**

References **129**

A Maple Worksheets and Procedures **137**

A.1 EnigmaProcedures.mpl 137

A.2 Encryption Program (Test05Encrypt.mpl) [Output Only] . . . 152

A.3 Decryption Program (Test05Encrypt02.mpl) [Output Only] . . 162

Vita

List of Figures

2.1 View of a four rotor Enigma showing both its keyboard and lampboard [36] 11

2.2 Electrical contacts on Enigma rotors [38] 13

2.3 Enigma Reflector [30] 15

2.4 Enigma Plugboard with two plugs connecting letters 'SJ' & 'AO' [25] 16

2.5 Electrical path through the various components of Enigma ultimately producing an enciphered output [24] 18

2.6 Setting wheels and windows for a four rotor Enigma [37] 21

3.1 Injection Function 26

3.2 Surjection Function 27

3.3 Bijection Function 28

3.4 Diagram showing the cycles formed within the permutation 32

Chapter 1

Introduction and Background

1.1 Overview

In today's networked computer infrastructure, security has become a primary concern. From mass surveillance to malicious access, the growing fear of improperly secured systems and information has made computer security specialists highly valued. The education these specialists receive when they are computer science students lays the groundwork for the knowledge that they will utilize during their professional careers. Ways of appropriately introducing and demonstrating computer security concepts within the classroom to students in a clear and concise manner does pose some difficulties, however. The primary difficulty is how to effectively demonstrate the concepts in a simple enough form to facilitate student comprehension while still

highlighting modern problems and techniques.

In order to provide assistance when introducing the subjects of cryptology and cryptanalysis, a teaching tool can be created that provides an implementation of a relatively simple, flawed, and well-known encryption scheme that students can examine and modify on their own. The tool should also highlight the flaws found within the system by demonstrating cryptanalytical techniques that are applied in order to unmake it. By effectively demonstrating both the creation and destruction of an implemented encryption scheme, it is hoped that this teaching tool can enrich a student's understanding of cryptology and cryptanalysis and how that knowledge can be applied to real problems.

The teaching tool described above forms the basis of this Master's of Computer Science report. The report work involves the creation of an editable implementation of the Enigma electromechanical encryption scheme utilized by Germany before and during the Second World War. The reason for Enigma's selection is that it fits the required criteria mentioned above: it is a simple, representative example of a symmetric key cipher system whose flaws and implementation are known and well documented. Enigma does not suffer from the English letter frequency attack due to the fact that it is a polyalphabetic substitution cipher whose period is the number of different positions of the rotors, which is much larger than the length of the message. A Navy variant of the Enigma machine will be implemented in a modern computing environment in a form that will be easy to examine and modify by students. This

variant, commonly referred to as the M3, is backward compatible with the Enigma I which was used in the German army and air forces [16]. A second program will also be created to highlight and exploit the weaknesses found within, demonstrating to students the execution of actual cryptanalytical techniques on a real-world cipher system.

In addition to being a teaching tool, the re-implementation of an electromechanical cipher machine within the modern computer environment could be used as a launching point for further research into the utility of this older cryptographic approach in today's environment. While the Enigma cipher is famously insecure, the program used to highlight and expose its systemic flaws could also be used to test a strengthened version of Enigma in order to determine its effectiveness.

1.2 Background

Computer security education provides some challenges to integrate concepts and knowledge into a form that can be easily engaged with and absorbed by students in either a classroom or lab setting. The topic of cryptography itself presents challenges due to its heavy background in mathematics and abstract nature which can make engaging the students with the material difficult [2]. Previous efforts to create tools for cryptography education have been met with promising results [2, 3, 5]. Most of the work within this space has been

centered on modern algorithms like DES [35, 9, 5, 26, 1], AES [9, 5, 26, 27], RSA [5, 26, 34], and Elliptic Curve cryptography [26, 33]. The drawback of these approaches, however, is that many modern cryptography algorithms can be difficult to follow and to execute in a step by step fashion. This is why many visualizers have been created in the first place. Simpler, historical cryptographic systems have been utilized to allow students to grasp basic techniques as a stepping stone to larger systems [32, 28, 7, 14]. There does appear to be a shortage of intermediate systems as one progresses between the simpler and more modern systems, however.

It is worth noting that in the case of DES, it can be easily demonstrated. However, DES operates on bitwise operations of length 32; through a Feistel function consisting of four stages (Expansion, Key Mixing, Substitution, and Permutation); and 16 rounds of Feistel functions. Comparatively, Enigma operates on one character at time and has less stages compared to DES. Ultimately, this comes down to the ease of teaching stream ciphers compared to block ciphers. While DES is not considered a secure cipher just like Enigma, it certainly is more modern. The goal here is to provide a set of programs that students can learn, expand, and experiment with and it is the author's opinion that this will be simpler to achieve with Enigma than with DES or another cipher system. It is also worth noting that RSA can also be demonstrated simply enough when the parameters are small, but this does not result in a secure cipher. Therefore, RSA does not scale as well as Enigma does when it comes to examples.

The decision to implement the M3 was done for several reasons and has various advantages and disadvantages in comparison to previous research. Firstly, this approach would allow students to explore a historically significant and engaging topic like Enigma that is commonly recognized. This would also introduce students to rotor-based encryption systems and the mathematics behind them, especially permutations.

1.3 Comparisons to Related Work

Previous work has been done to cover rotor-based encryption in an educational setting [32]. That approach used a half-rotor cipher in order to simplify the problem so that the material and its cryptanalysis could be covered within a single lecture. The drawback of this approach is that this greatly simplifies the system, limiting its true potential for learning all relevant concepts. It also was done only in theory, without a computer implementation that allowed one to examine its full execution. Enigma has been found to be beneficial to implement by students as a learning exercise [20] and computer algebra systems have been used previously to implement cryptosystems and demonstrate concepts to some success [7, 29, 1]. Implementing Enigma in Maple allows students to develop skills not only in the mathematical concepts, but also programming and the use of computer algebra systems. It also enables students to learn how to not only implement cryptosystems in

Maple, but its cryptanalysis and solutions as well. Maple is not as ubiquitous as Microsoft Excel, which has been used successfully to implement several cryptosystems [9, 28], but it is commonly available, will be more flexible with mathematical operations, and will allow easier implementation of Enigma and its cryptanalysis. It will also allow students to examine the inner workings of the implemented procedures through its internal debugger. The use of Maple worksheets will be a more flexible format for its cryptanalysis especially compared to JAVA, which was used in previous work [14]. It will also allow a deeper examination of the inner workings than what is available in standalone programs like CrypTool or web based tools [5][26] and allow students to modify the procedures and sheets to create variants or new methods. The cryptanalysis worksheet implements several techniques to reach a solution for Enigma. Topics include: known plaintext attacks; the Rejewski catalog, along with its mathematical basis of Enigma operations being cycle preserving due to the symmetric nature of the reflector; use of recursive searching for viable plugboard connections; and the combination of all of these attacks to reduce the potential solutions to only a handful. All of these have the potential to introduce important mathematical and cryptography concepts to students. One of the major disadvantages of this approach is that, unlike DES, AES, RSA, or Elliptic Curve cryptography, Enigma is not a currently used cryptological system and its flaws and weaknesses are widely known. While this allows a simpler and more approachable subject matter to introduce concepts within an educational context, its utility to larger current

problems may be limited.

Chapter 2

Defining Enigma

2.1 Historical Overview

Enigma refers to a family of electromechanical cipher machines that Germany utilized in one form or another to encrypt most of their high command messages and orders from as early as 1926 until the end of the Second World War [24]. The versions of the machine that the German military utilized were all derived from a cipher machine invented by Arthur Scherbius, who originally intended for the machine to be used within the business industry [24, 4]. With interest from the German military, the commercial version of Enigma was altered to create the various military versions that were deployed and used within the various branches of Germany's growing armed forces [24].

Shortly after its deployment, the newly machine enciphered radio traffic now

traversing the Germany radio nets had attracted the attention of a number of countries. The Signals Intelligence (SIGINT) divisions of the Polish, French and British Military had each individually attempted to decode this new system without success [24]. Eventually, French Military Intelligence was contacted by a German spy who offered to sell to the French access to various documents that were available to him. One of these documents consisted of operating instructions for Germany's new cipher machine: Enigma [24]. In an act of good faith, and with the hope of initiating a fruitful collaboration, French Signal Intelligence officer Gustave Bertrand had provided copies of their newly acquired documents to his counterparts within the Polish Cipher Bureau. These documents, along with a commercial version of Enigma, were provided to Marian Rejewski, a cryptographer working within the Polish Cipher Bureau's German Division (BS-4). Rejewski was given the sole task of determining what changes were made to the commercial version of Enigma in order to produce the machine that was currently being used to encipher some of Germany's radio traffic [24]. Through a clever combination of mathematics, observation, and luck, Rejewski was slowly able to work out the unknown variables for the new military Enigma variant. This became the foundation for the Polish Cipher Bureau's work on Enigma and would lead to not only the capability of reading actual German communications, but allowed the Polish Cipher Bureau to create working replicas of the military variant of Enigma to ease message decipherment [24]. These replicas and all Enigma information ended up being shared with both the French and

British during the Warsaw-Pyry conference that was held on the 24th-25th of July, 1939 [24], a few months before Poland was invaded by Germany in September.

After the conference, the British continued to work on Enigma at Bletchley Park in order to keep apace with the changes to both machine components and procedures throughout the war. Having more resources available than the Polish Cipher Bureau, the British made numerous advances with respect to Enigma. The end result of this labor was that Enigma communications were intercepted, decoded, and read by the Allies shortly after being transmitted by the German military. All information derived from decoded Enigma communications was classified as "Ultra Secret", and only a select few were allowed access to it directly. This information was considered to be crucial intelligence on the enemy, and special liaison units were created to manage the distribution of the information to anyone who required it [24]. Given such importance placed on intelligence derived from Enigma decoded communications, its impact on the war effort can not not be overstated.

2.2 Enigma Overview

2.2.1 Machine Components

Enigma was roughly the same size as a typewriter and quite similar in appearance. Usually transported inside a wooden box, the machine had a certain pedestrian quality which contrasts its importance to the German war effort. As an electromechanical cipher machine, Enigma was composed of a multitude of individual parts working in conjunction to create the functioning device. There are, however, five components of the machine that are of particular importance: the lampboard, the rotors, the plugboard, the reflector, and the keyboard.

2.2.1.1 Keyboard and Lampboard

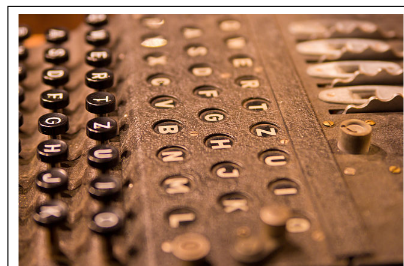


Figure 2.1: View of a four rotor Enigma showing both its keyboard and lampboard [36]

Both the keyboard and lampboard, respectively, form the input and output of the machine from the perspective of its operator. The lampboard[1] consists of an arrangement of light bulbs that are placed into sockets located on the top of the machine above the keyboard.

[1] The German term for the lampboard on the Enigma was the GLÜHLAMPENFELD [21].

Each individual bulb corresponds to a letter of the German alphabet and has an accompanying window with a letter label integrated into a cover which is normally closed during most of the machine's operation. When a key is pushed on the keyboard, the electrical signal is scrambled through different wires and connections until it enters one connected to a bulb. When a bulb is lit with the cover closed, the labeled window above is illuminated and displays the letter that corresponds to the key that was pushed. Both the encryption and decryption of messages are handled one letter at a time in this manner since Enigma employs symmetric encryption. Symmetric encryption is a scheme whose distinguishing feature is that the exact same key, the machine's starting settings in the case of Enigma, can both encrypt and decrypt a message.

The keyboard[2] was of the same construction as those commonly seen in typewriters at the time. The main difference, however, is that the mechanical force of pushing any key down on the keyboard will cause the rightmost rotor to make $\frac{1}{26}th$ of a full rotation before an electrical connection is closed at the base of the downward pressing action of the key.

[2]The German word for the keyboard was the TASTATUR [21].

2.2.1.2 Entry Wheel

The entry wheel[3] was the point where the individual wires that carried the electrical impulses from beneath the keyboard would be collected together and attach to contacts that would allow the electrical signal to pass into the rotors.

2.2.1.3 Rotors

The rotors[4] are drums with flat electrical contacts on one side and spring loaded ones on the other. These contacts are spaced evenly along the entire circumference of the rotor. The rotors contain a ring placed around the drum that has the letters of the alphabet inscribed along its length to identify the different orientations of the rotor. While the number of rotors any machine could take at any one time varied depending on its type, the M3 is used as the basis for the work in this report and the variant operated on three rotors. There were only three rotors to choose from until the 15th of December, 1938 when two more rotors were introduced bring the total to 5 [10]. Another three rotors exclusive to the Kriegsmarine where also introduced in 1939 [16]. The three select rotors can

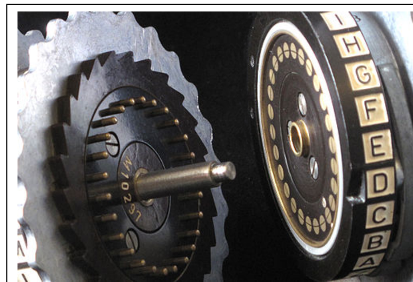


Figure 2.2: Electrical contacts on Enigma rotors [38]

[3]The German name for the entry wheel was the eintrittwalze [21].
[4]The Germans called the rotors: the Chiffrierwalzen [21].

placed in either the leftmost, middle, or rightmost positions of the machine. Inside the rotors are copper wires that connect each of the contact points to another on the other side. It is worth noting that, unlike both the plugboard or reflector, these connections do not pair two letters together. For example, if ‘G’ is connected to the contact that represented ‘K’ in the rotor’s ‘A’ orientation, then that does not necessarily mean that the ‘K’ contact needs to connect to ‘G’ in that same orientation. The rotors sit on an axle and will perform rotations around it in $\frac{1}{26}th$ of a full rotation increment. These rotations occur with every key press for the rightmost rotor, but the middle and leftmost rotors only turn based on where the turnover notch is set on the rotor to its right. This notch will catch on the ratchet pawl of the rotor to its left when aligned and will allow any aligned rotor to perform $\frac{1}{26}th$ of a rotation during its next step. These notches are part of the rotor itself and cannot be repositioned. However, it was possible to rotate the alphabet ring so that the internal wiring with respect the rotor could be offset into 26 different positions. This was referred to as the ring setting[5], and would mainly affect how the internal wiring is positioned relative to the start and turnover positions [41]. The ring setting forms an additional component of the key in addition to the orientation, selection, and positions of the rotors that are loaded into the machine.

[5] The Germans referred to the rings settings as the RINGSTELLUNG [21].

2.2.1.4 Reflector

Figure 2.3: Enigma Reflector [30]

The reflector[6] is a drum that only has spring loaded contacts on one side, with one contact representing each letter of the alphabet. These contacts are wired together in pairs in a similar fashion to the plugboard. Unlike the plugboard, however, every one of the 26 letters of the alphabet is paired with another letter forming a total of 13 pairs. The drum itself does not rotate around an axle; unlike the rotors, it remains completely stationary during its operation. Electrical current enters one of these spring loaded contacts and exits through another. Due to the pairing within the wiring, this exit contact is guaranteed not to be the entry one. Unlike both the rotors and the plugboard, the reflector was rarely replaced and does not contribute significantly to the combinatorial difficulty of the encryption. There were only three different variants of the reflector [10] until the Umkehrwalze D, or UKW-D, was introduced in January 1944. The UKW-D was a field-rewirable reflector, but it was cumbersome to change, so this only happened once every 10 days [17].

[6] The German term for the reflector is the UMKEHRWALZE [24, 21].

2.2.1.5 Plugboard

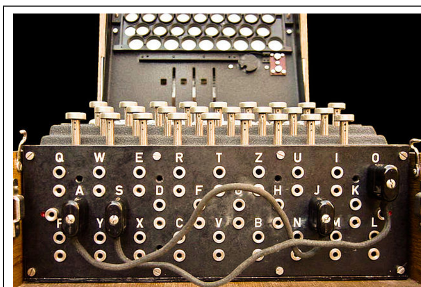


Figure 2.4: Enigma Plugboard with two plugs connecting letters ‘SJ’ & ‘AO’ [25]

The plugboard[7] is the first and last step of the electrical scrambling process because it is placed before all the rotors and the reflector. The plugboard is a board that is positioned at the front of the machine and appears as a series of labeled sockets each corresponding to a letter of the alphabet. These sockets can be paired with a wire possessing a plug on each end that connects into two of the holes. These plug connections[8] effectively pair two letters together and switches their electrical connections. For example, if a wire was connecting both the letters ‘H’ and ‘D’, then the electrical connections would effectively replace every instance of an ‘H’ with a ‘D’ and vice versa. Since the letters are paired, that means that there are a maximum of 13 wires that could be placed in the plugboard. The primary reason for the plugboard is that it is essentially a programmable reflector. While the rotors and most of the reflectors were manufactured with their internal wiring set, the plugboard could be rewired on the fly. This adds an additional combinatorial difficulty to solving the key using brute force and is most of the primary challenges when it come to Enigma’s cryptoanalysis.

[7] The German name for the plugboard is the Steckerbrett [21].

[8] The German word for the plug connections was the STECKERVERBINDUGEN [24, 21].

2.2.2 Electromechanical Process

Now that we have described each of the 5 primary components of the Enigma machine, the step by step process of scrambling an electrical signal and how that translates to performing an encryption operation will be described. First, the downward motion of a key press will trigger what is referred to as a stepping mechanism that is responsible for rotating the rotors around their axle before any electrical signal is fired. This stepping motion will perform a $\frac{1}{26}th$ of a full rotation for the rightmost rotor and any rotors to its left that have their notches and ratchet pawls aligned. All this is completed by the time the key on the keyboard being pressed has reached the base of its downwards motion.

Once this happens, a circuit is closed, and electrical current powered by a battery that is placed inside the machine starts to flow from beneath the keyboard to the plugboard, where any wires that are placed in the plugs switch the current between those two wires. For example, if the key pressed down was 'Q' and there was a wire connecting both 'Q' and 'Z' in the plugboard, then the current in the wire representing the 'Q' character being encrypted will cross over and flow into the 'Z' wire. The electrical current then proceeds up this wire to the entry drum.

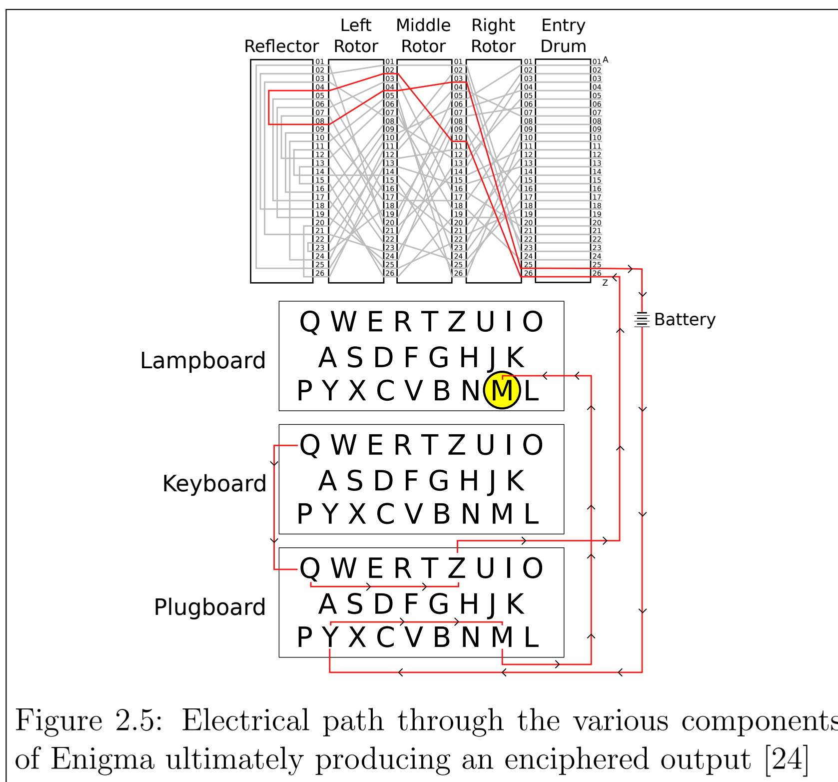


Figure 2.5: Electrical path through the various components of Enigma ultimately producing an enciphered output [24]

Continuing with the previous example as illustrated in Figure 2.5, the 'Z' wire will enter the entry drum and come to an end at an electrical contact on its left side. The current will pass from this contact on the left side of the entry drum into a spring loaded contact on the right side of the rightmost rotor, into its internal wiring, and eventually flowing into a electrical contact on its left side. To illustrate further, the current will exit from the 26th ('Z') electrical contact on the left side of the entry drum and connect to the 26th spring loaded contact on the right side of the rightmost rotor, through the internal wiring, and exit the rightmost rotor through its 10th ('J') contact on the other side. The middle and leftmost rotors will both function similarly, and

so current flows from the 10th ('J') contact on the left side of the rightmost rotor, to the 10th spring loaded contact on the right hand side of the middle rotor, through the internal wiring of the middle rotor, and to the 2nd ('B') contact on the left hand side. Continuing on, it enters into the 2nd ('B') spring loaded contact on the right hand side of the leftmost rotor and exits the 4th ('D') contact on the left hand side of the leftmost rotor.

From here, the current will pass from the contact on the left hand side of the leftmost rotor to the corresponding spring loaded contact located on the reflector. Since the reflector consists of nothing more than the 26 letters of the German alphabet wired into 13 pairs, the current will flow from the 'D' contact on the left hand side of the leftmost rotor to the spring loaded 'D' contact on the reflector, and then out the spring loaded contact representing 'H'.

Now the electrical current will start its journey back through the rotors in the opposite direction from which it came, changing from 'H' to 'D', from 'D' to 'C', from 'C' to 'Y', and then finally from the right hand side of the rightmost rotor 'Y' to the 'Y' wire that is touching the contact on the left side of the entry drum. From there, it will proceed back to the plugboard where it will switch from the 'Y' wire to the 'M' one and pass to the lampboard where it will light up the bulb with the 'M' label. This entire process is how the Enigma machine performs its encryption, and decryption, electromechanically.

2.2.3 Symmetric Encryption Key

Enigma's symmetric encryption key is composed of several elements, most of them involving the rotors. In total, the key is composed of the following elements:

1. Plug Connections (The wiring of the plugboard)
2. Rotor Selection & Order (The rotors chosen and which rotors occupy the leftmost, middle, and rightmost positions)
3. Rotor Starting Positions (The initial rotor orientations marked by the labeled ring)
4. Rotor Ring Settings (The off of the rotor's internal wiring with respect to its turnover and starting positions)
5. Reflector (Which reflector is used)

While the reflector may be part of the key since its internal wiring forms letter pairs and it could be replaced with another with a different wiring, as mentioned previously, this didn't happen often and will be discounted since it does not form a significant portion of the combinatorial difficulty of the key.

Figure 2.6: Setting wheels and windows for a four rotor Enigma [37]

Now that all of the individual components of Enigma's symmetric encryption key have just been described, there is one more important element: the daily and message settings. These settings refer to the starting positions of the rotors and how they were changed during a full message encryption or decryption.

2.2.3.1 Daily Setting

The daily setting is the initial starting positions of the rotors as they are loaded into the machine. This is given as a three letter code that represents the initial positions of the leftmost rotor, middle, and rightmost rotor read from left to right. These positions correspond to letters seen on the ring of each rotor as read from a window that is placed in a cover that closes over top of the rotors once they have been loaded into the machine. For

example, the daily setting 'JKD' means that the leftmost rotor's ring should read 'J' through the window, the middle rotor's ring should read 'K', and the rightmost rotor's ring should read 'D' before any key pressed. The daily setting is given its name because it was assigned to the machine operators each day along with the choice of rotors, what position each rotor should occupy in the machine, and the plugboard settings.

2.2.3.2 Message Setting

The message setting is similar except that it was chosen by the operator for each message. This setting was initially enciphered twice at the start of each message. This was done to minimize transmission errors, but this practice was stopped on May 1st, 1940 [6] and the message setting was only enciphered once due to the realization that this weakened the encryption. In the following examples, it is assumed that this was before May 1st, 1940. Therefore, the message setting was still enciphered twice at the start of each message. The setting utilized for enciphering the message setting was the daily setting and was its only use.

The actual contents of the message would follow after the first six characters were encoded with the daily setting and would be encrypted using the message setting chosen by the operator. For example, an operator was given a plaintext message "ATTACK AT DAWN" and the daily setting 'JDK'. The operator would then chose a message setting to encrypt the plaintext: 'OCP'. The operator would first set their machine rotors to 'JDK', assuming the rest

of the machine has already been set, and then type the first six letters of the message as “OCPOCP”. The operator would then reset the rotors in their machine to ‘OCP’ and then type the actual message contents. Therefore, total sequence of keys pressed on the keyboard to generate the encrypted message would be “OCPOCPATTACKATDAWN”.

Chapter 3

Unraveling Enigma

3.1 Representation of the Machine

Before the cryptanalysis of the Enigma machine is presented, several concepts and ideas from mathematics and group theory will be covered in order to establish the required foundations.

3.1.1 Permutation Primer

The primary concept used in Enigma's cryptanalysis is that of a permutation, which are represented by utilizing cycle notation. These concepts are used to mathematically represent the letter switching functionality of the plugboard, rotors, reflectors, and is used to account for the rotation of the rotors around the axle.

3.1.1.1 What is a Permutation?

With counting problems, it is important to distinguish between ordered and unordered selections. Selections where the order of the element within the selection is important is called a PERMUTATION [8]. Permutations appear quite frequently within the study of mathematics, principally in Group Theory (the study of groups) and Combinatorics (the study of counting). With the cryptanalysis of the Enigma machine, the concept of a permutation is quite useful for mathematically defining its letter switching properties and deriving further information from these.

3.1.1.2 What is a Function?

In group theory, a permutation is primarily defined by utilizing functions. So before we continue with a formal definition of a permutation, we will introduce the concept of a function.

DEFINITION [18]:

A function (or mapping) ϕ from a set A to a set B is a rule that assigns to each element a of A exactly one element b of B.

A FUNCTION, or MAPPING, is an assignment of the elements of one set to another. These mappings can arise in several variations and can be combined, or composed together.

Definition [18]:

Let $\phi : A \to B$ and $\psi : B \to C$. The composition $\psi\phi$ is the mapping from A to C defined by $(\psi\phi)(a) = \psi(\phi(a))$ for all a in A.

Function composition is the act of combining more than one function together to form a singular mapping from one set to another given a common set connecting the two together.

One-to-One (Injective) Functions:

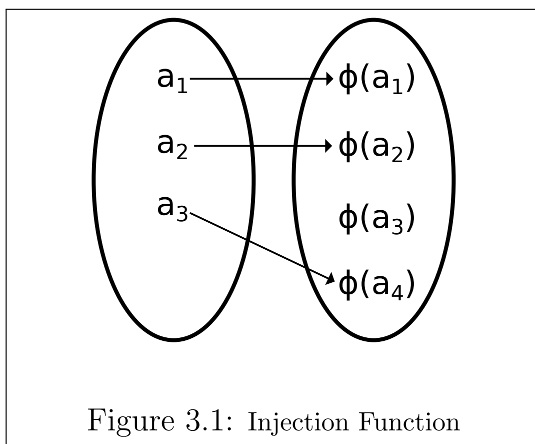


Figure 3.1: Injection Function

Definition [18]:

A function ϕ from a set A is called one-to-one if for every $a_1, a_2 \in A$, $\phi(a_1) = \phi(a_2)$ implies $a_1 = a_2$.

One-to-One, or injective, functions are mappings from one set to another such that every element of the first set is assigned a unique element of

the second set. All elements of the second set, however, do not require an assignment.

Onto (Surjective) Functions:

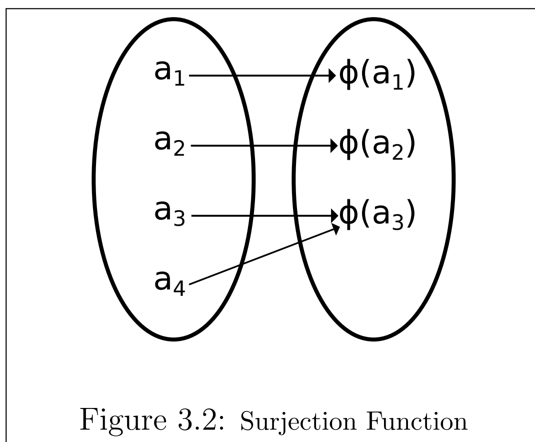


Figure 3.2: Surjection Function

DEFINITION [18]:

A function ϕ from a set A to a set B is said to be onto B if each element of B is the image of at least one element of A. In symbols, $\phi : A \to B$ is onto if for each b in B there is at least one a in A such that $\phi(a) = b$.

ONTO, or SURJECTIVE, functions are functions where all elements of the first set have an assignment to an element in the second set and all elements in the second set have at least one assignment to an element in the first set. An element in the second set can be assigned to more than one element in the first set, however.

Bijection Functions:

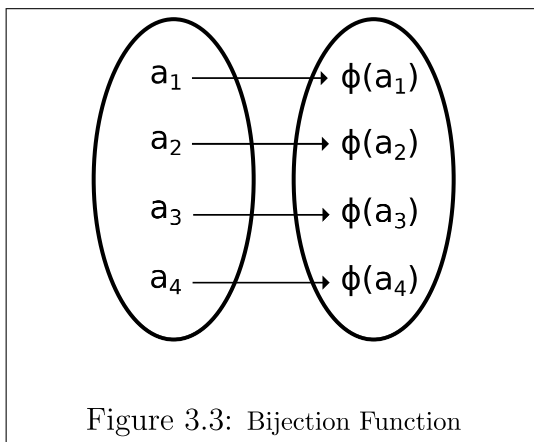


Figure 3.3: Bijection Function

BIJECTIVE FUNCTIONS are mappings that are both one-to-one and onto. So every element in the first set is assigned a unique element of the second set, and every element of the second set is assigned at least one element of the first set. As a consequence of the properties of both functions, all elements in either set have unique assignment to each other. This is because if an element of the second set had no assignment, then it would not be onto; and if an element of the second set was assigned to more than one element of the first set, then it would not be one-to-one.

3.1.1.3 Permutations

Definition [18]:

A *permutation* of a set A is a function from A to A that is both one-to-one and onto *(Bijection)*. A *permutation group* of a set A is a set of permutations of A that forms a group under function composition.

Permutations are mappings from a set to itself that is bijective, that is both one-to-one and onto. This uniquely maps every element of a set to itself. This is useful in cryptology because this allows one to mathematically represent the mapping of every element of a set, say the letters in the alphabet, to itself in a reversible way. This means that encryption and subsequent decryption can be represented utilizing a permutation.

The second part of a definition of a permutation is that a permutation group is a set of permutations that form a group under function composition.

3.1.1.4 Groups

A group, in the mathematical sense, refers to a set that holds certain properties under an operation. More formally, we can define a group as the following.

DEFINITION [18]:

Let G be a nonempty set together with a binary operation. We say G is a group under this operation if the following four properties are satisfied:

1. *Closure.* Assigns to each order pair (a, b) of elements of G, another element in G denoted by ab under this operation.
2. *Associativity.* The operation is associative; that is, $(ab)c = a(bc)$for all a, b, c in G.
3. *Identity.* There is a element e (called the identity) in G such that $ae = ea = a$ for all a in G.
4. *Inverses.* For each element a in G, there is an element b in G (called an inverse of a) such that $ab = ba = e$.

So all groups are sets that are closed, associative, have an identity, and have an inverse for each element under a chosen binary operation. GROUP THEORY is the field of mathematics concerned with the study of groups, their properties, and their applications. This overlaps with combinatorics and the study of counting because the set of all possible permutations of a set forms a group under function composition. Fulfilling the group properties, the set of all possible permutations of a set is closed, each permutation has an inverse due to bijective mapping, an identity exists since one can map each element to itself, and such a set is associative due to function composition [22].

3.1.1.5 Cycle Notation

Products of Disjoint Cycles [18]:

Every Permutation of a finite set can be written as a cycle or as a product of disjoint cycles.

Permutations have a property in group theory that allows them to be written as a product of disjoint cycles. While permutations are normally written in matrix form, since a permutation is a mapping of a set to itself, the second row can be eliminated and the permutation itself can be represented as a product of disjoint cycles.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 4 & 2 & 7 & 6 & 5 & 8 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 6 & 8 & 3 & 7 & 2 & 5 \\ 4 & 6 & 8 & 3 & 7 & 1 & 2 & 5 \end{pmatrix} = (146837)(2)(5) \tag{3.1}$$

In this example, the matrix form of the permutation is found on the leftmost side of Equation (3.1). This permutation is then rewritten with the common elements forming the cycles within the permutation placed next to each other in the rearranged matrix form found in the middle.

As we can see in Figure 3.4, the first cycle is formed starting from 1 and finding that it is mapped to 4. Continuing with 4, we find that it itself is mapped with 6; 6 with 8; 8 with 3; and continuing in this fashion until we reach 7, which maps back to 1. This element sequence forms our first cycle and is found in the leftmost braces of the cycle notation representation of the permutation found on the rightmost side of Equation (3.1) as (146837). The next element that has not been noted in the previous cycle is the number 2, which we find is mapped to itself. This is found in the middle braces of the rightmost side of the equation as (2). The only remaining element left is 5, which is transcribed the same way as the previous cycle and found on the rightmost braces on the rightmost side of Equation (3.1) as (5).

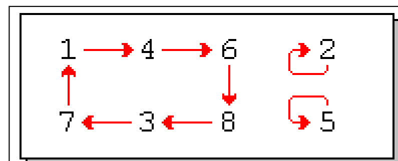


Figure 3.4: Diagram showing the cycles formed within the permutation

3.1.2 Representing the Rotors, Plugboard, and Reflector

Mentioned previously in Subsection 2.2.3, the elements of the key for the Enigma are:

1. Plug Connections
2. Rotor Selection & Order
3. Rotor Starting Positions

4. Rotor Ring Settings
5. Reflector

All of these can be represented using permutations in order to implement their letter switching functionality. Tracing the path through the machine as the electrical signal leaves the keyboard, we first proceed to the plugboard. This is represented as permutation S, for the German term *Steckerbrett* [21]. From the plugboard one continues to the entry wheel, depicted here as permutation E. From there one passes through the right, middle and left rotors portrayed as permutations R, M, and L respectively. One passes from the right, middle, and the left rotors in this fashion because that is the path that the electrical signal takes in the actual machine. Ultimately, one ends up at the reflector, which is signified with the permutation U for the German term *Unkehrwalze* [24]. The letter R is not used since it already is utilized for the rightmost rotor. This currently leaves us with the following permutation composition.

$$SERMLU \ldots$$

Since this is the midpoint of the electrical signal's path, one now reverses and travels backwards through the left, middle, and right rotors, through the entry wheel and the plugboard. Due to the reversal in direction, the

inverse of the permutations are used for the return path. This results in the following function composition used to represent Enigma's symmetrical encryption [24].

$$SERMLUL^{-1}M^{-1}R^{-1}E^{-1}S^{-1} \tag{3.2}$$

Previously, we had mentioned that inverses are a property of groups. Since the Enigma functions on permutation groups, each individual permutation will have an inverse. The inverse of a permutation will be a mapping that will result in the identity when a permutation is composited with its inverse. The identity of a permutation is the mapping of each element to itself. Inverse permutations are represented with a superscript of -1 so in equation 3.2 above, $M^{-1}$is the inverse of permutation M.

3.1.3 Representing the Rotor Rotations

There are elements of the key mentioned in the previous section that are not covered in Equation (3.2). These elements are the ring settings for each rotor and the rotation of the rotors themselves. The ring settings are ignored in this work to place the focus on deriving information from the twice enciphered message setting encrypted with the daily setting at the start of each message [24]. The ring setting, or Ringstellung in German [21], does not alter the wiring of the rotors themselves, it only alters the position of the wiring with

respect to the alphabet ring and the relative position of the turnover position. The turnover positions of each rotor will not affect the leftmost rotor, only any rotors to its right [15]. For this reason, the ring setting only add an additional $26x26 = 676$ possible positions since we can ignore the leftmost rotor since it can't impact any of the other rotors. In the cryptanalysis case, however, one is only examining the first six letters of each message and any rotors to the left of the rightmost rotor will only turnover in a small amount of cases within the first six letters of the starting position. For these reasons, the ring settings have been chosen to be ignored and the following cryptanalysis will do so as well.

The rotation of the rotors is an element that does need consideration. In Equation (3.2), we have permutation R which represents the rightmost rotor, or most specifically, the internal wiring of the rightmost rotor. What is required is a way to alter permutation R as the rotor performs $\frac{1}{26}th$ of a full clockwise rotation around the axle of the machine. A rotor's rotation around the axle alters which contacts form connections from either the entry wheel, another rotor, or reflector, depending on the context. Each individual electrical contact on these components represents a letter in the alphabet, with the electrical switching through wires and contacts implementing the letter scrambling function of Enigma. So, in order to represent the systematic switching of letters as the contacts rotate and form new connections, we need a new permutation P which will map one connection to another in order to represent the altered orientation of the rotor. Since the clockwise rotation

of the rotor and the alphabet wheel go in the same direction, we need a permutation that will take the first contact and map it to the second, the second to the third, and so on, until the last connection maps back to the first. This permutation will be written in cycle notation as follows [24]:

$$P = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ \ldots \ldots\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26) \tag{3.3}$$

Which would result in the following partial permutation composition up to the rightmost rotor:

$$SEPR\ldots$$

In order to depict further rotations around the axle, we can compose this P by itself for two steps of the rightmost rotor:

$$SEPPR\ldots = SEP^2R\ldots$$

This permutation P can be applied up to the 25th stepping point, with P^0equaling to the identity: where P isn't applied at all. This happens when the '*A*' contacts on each rotor line up with each other. The repeated composition of a variable amount of permutation P can be represented by variable

i, which is used to indicate the rightmost rotor's position:

$$SEP^iR\ldots$$

It is worth noting, however, that a permutation only covers the entry into the rightmost rotor so far, it does not cover its exit on the opposing side. This can be depicted by the inverse of P^i.

$$SEP^iRP^{-i}\ldots$$

This can be expanded to the other rotors with variables j and k used to represent the position of the middle and leftmost rotors respectively [24].

$$SEP^iRP^{-i}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^iR^{-1}P^{-i}E^{-1}S^{-1} \tag{3.4}$$

There is one more modification required, however. The entry wheel in the military Enigma mapped each letter to itself, which results in the identity, so it can be safely removed from the equation resulting in:

$$SP^iRP^{-i}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^iR^{-1}P^{-i}S^{-1} \tag{3.5}$$

This now gives us the complete function composition used to represent Enigma's symmetrical encryption.

3.2 Double Encipherment

The message setting of each transmission was double enciphered at the start of every message using the daily setting, as discussed in Subsection 2.2.3.1. This double encipherment was a large mistake since it offered cryptographers a guaranteed relationship between the first and fourth, the second and fifth, and the third and sixth letters, since the same plaintext letters are encrypted within a known distance from each other. In Enigma's cryptanalysis, these first six letters of the message setting are traditionally labeled A through F and are the starting point towards unraveling Enigma's encryption [24].

Utilizing the resulting encryption composition from Equation (3.5), we can mathematically represent A-F as the following:

$$A=(SP^{i}RP^{-i}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{i}R^{-1}P^{-i}S^{-1}) \tag{3.6}$$

$$B=(SP^{(i+1)}RP^{-(i+1)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+1)}R^{-1}P^{-(i+1)}S^{-1}) \tag{3.7}$$

$$C=(SP^{(i+2)}RP^{-(i+2)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L{-}1P^{-k}P^{j}M^{-1}P^{-j}P^{(i+2)}R{-}1P^{-(i+2)}S^{-1}) \tag{3.8}$$

$$D=(SP^{(i+3)}RP^{-(i+3)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L{-}1P^{-k}P^{j}M^{-1}P^{-j}P^{(i+3)}R{-}1P^{-(i+3)}S^{-1}) \tag{3.9}$$

$$E=(SP^{(i+4)}RP^{-(i+4)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L{-}1P^{-k}P^{j}M^{-1}P^{-j}P^{(i+4)}R{-}1P^{-(i+4)}S^{-1}) \tag{3.10}$$

$$F=(SP^{(i+5)}RP^{-(i+5)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L{-}1P^{-k}P^{j}M^{-1}P^{-j}P^{(i+5)}R{-}1P^{-(i+5)}S^{-1}) \tag{3.11}$$

The second step cryptographers performed was the discovery and construction of what is called the CHARACTERISTIC SET. The characteristics are composition functions of the first and fourth letters (AD), the second and fifth (BE), and the third and sixth (CF). These can be constructed from approximately 60 messages sent encrypted with the same daily setting [24]. This is accomplished by examining the messages and building the cycles of the permutation formed from the letters in the first and fourth positions, in the case of characteristic AD, until the permutation is completed for AD, BE, and CF [24]. The significance of the characteristics is that the compositions are formed from two of the same letters a known distance apart encrypted utilizing the same key. It is worth noting, however, that the message setting in the individual communications are not guaranteed to be the same, even if they are encrypted with the same daily setting. This is the reason why the characteristics themselves can be built; otherwise the first and fourth letters would always be the same. The most important element is that not only can these sets be constructed solely from a sufficient number of intercepted messages from the same day and communication net, guaranteeing the usage of the same key, but the permutations have different cycle lengths and breakdowns depending on the key used. Therefore, the cycle type of the characteristics can be used to indicate which key was used if one catalogs the cycle types of all key positions [24]. This will be examined in more detail later.

The characteristic set can be formulated utilizing Equations (3.6)-(3.11) resulting in the following:

$$
\begin{aligned}
AD =&(SP^{i}RP^{-i}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{i}R^{-1}P^{-i}S^{-1})\\
&(SP^{(i+3)}RP^{-(i+3)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+3)}R^{-1}P^{-(i+3)}S^{-1}) \quad (3.12)\\
BE =&(SP^{(i+1)}RP^{-(i+1)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+1)}R^{-1}P^{-(i+1)}S^{-1})\\
&(SP^{(i+4)}RP^{-(i+4)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+4)}R^{-1}P^{-(i+4)}S^{-1}) \quad (3.13)\\
CF =&(SP^{(i+2)}RP^{-(i+2)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+2)}R^{-1}P^{-(i+2)}S^{-1})\\
&(SP^{(i+5)}RP^{-(i+5)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+5)}R^{-1}P^{-(i+5)}S^{-1}) \quad (3.14)
\end{aligned}
$$

These equations can be simplified further, however. First, by eliminating the $S^{-1}S$ composition that results from composing AD, since this results in the identity. Secondly, by simplifying the $P^{-i}P^{i+3}$composition that is left after eliminating $S^{-1}S$ from AD.

$$
\begin{aligned}
AD =&(SP^{i}RP^{-i}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{i}R^{-1}P^{-i}S^{-1}) \\
&(SP^{(i+3)}RP^{-(i+3)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+3)}R^{-1}P^{-(i+3)}S^{-1}) \qquad (3.15) \\
=&(SP^{i}RP^{-i}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{i}R^{-1}P^{-i}\cancel{S^{-1}}) \\
&(\cancel{S}P^{(i+3)}RP^{-(i+3)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+3)}R^{-1}P^{-(i+3)}S^{-1}) \qquad (3.16) \\
=&(SP^{i}RP^{-i}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{i}R^{-1}\cancel{P^{-i}} \\
&P^{(i+\cancel{3})}RP^{-(i+3)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+3)}R^{-1}P^{-(i+3)}S^{-1}) \qquad (3.17) \\
=&(SP^{i}RP^{-i}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{i}R^{-1} \\
&P^{(i+2)}RP^{-(i+3)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+3)}R^{-1}P^{-(i+3)}S^{-1}) \qquad (3.18)
\end{aligned}
$$

BE and CF can also be simplified in this fashion to give

$$
\begin{aligned}
BE =&(SP^{(i+1)}RP^{-(i+1)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+1)}R^{-1}) \\
&(P^{(i+3)}RP^{-(i+4)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+4)}R^{-1}P^{-(i+4)}S^{-1}) \\
CF =&(SP^{(i+2)}RP^{-(i+2)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+2)}R^{-1}) \\
&(P^{(i+3)}RP^{-(i+5)}P^{j}MP^{-j}P^{k}LP^{-k}UP^{k}L^{-1}P^{-k}P^{j}M^{-1}P^{-j}P^{(i+5)}R^{-1}P^{-(i+5)}S^{-1})
\end{aligned}
$$

3.3 Cycle Structure

Starting from Equation (3.18) from the previous section, we can move the parentheses, due to function composition being associative, and substitute the variable Q for the portion between the plugboard permutations. This will result in the following:

$$\begin{aligned}
AD =&(SP^iRP^{-i}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^iR^{-1} \\
&P^{(i+2)}RP^{-(i+3)}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^{(i+3)}R^{-1}P^{-(i+3)}S^{-1}) \qquad (3.19) \\
=&S(P^iRP^{-i}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^iR^{-1} \\
&P^{(i+2)}RP^{-(i+3)}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^{(i+3)}R^{-1}P^{-(i+3)})S^{-1}) \qquad (3.20) \\
=&\boldsymbol{SQS^{-1}} \qquad (3.21)
\end{aligned}$$

Permutation composition SQS^{-1} has a unique property: CONJUGACY [18].

DEFINITION [40]:

A group element h is said to be conjugate to a group element k, $h \sim k$, if there exists a $g \in G$ such that $k = ghg^{-1}$.

Conjugacy is important because it preserves structure [18].

DEFINITION [22]:

Let π and p be permutations in $S(n)$. The cycle decomposition of the element $\pi p \pi^{-1}$is obtained from that of p by replacing each integer i in the cycle decomposition of p with the integer $\pi(i)$.

This results in the composition SQS^{-1} having the same cycle structure as Q. In summary, the plugboard has zero effect on the cycle structure of any of the characteristics.

3.4 Constructing a Catalog

Having established that the cycle structure of the characteristic set is not affected by any of the plugboard wirings, one can construct a catalog of cycle structures for all the possible keys and utilize this to help identify potential settings used for any message. This was a cryptological technique used before and during the Second World War, and was reimplemented with other methods for this report.

This report implemented the construction of the catalog, by simply iterating through all possible keys of a software implementation of Enigma and saving the settings used in a list stored inside a data structure that is indexed by all possible cycle structures. This catalog is accessed by passing a cycle structure as input into a separate function which returns the corresponding index into the data structure containing all the lists. This index is then used to retrieve a list of key settings whose cycle structures match.

The key possibilities can be narrowed down further since we have AD, BE, and CF and can utilize the catalog to collect only the key settings which would result in the same cycle structure as the encrypted message for all three characteristics. There is not a separate catalog for BE and CF since these adjacent settings would appear in the catalog anyway. One just needs to cross reference that any potential settings will generate compatible cycle structures when the rotors are advanced into the positions needed for BE and CF.

3.5 Plugboard Wirings

To reiterate, the elements that make up the key settings of Enigma are the following:

1. Plug Connections
2. Rotor Selection & Order
3. Rotor Starting Positions
4. Rotor Ring Settings
5. Reflector

With the rotor selection and order, as well as the rotor starting positions,

covered by the construction and use of a catalog of characteristic cycle structures, this leaves the rotor ring settings, the reflector and the plugboard as the remaining unknown variables. The rotor ring settings as explained earlier will mostly be ignored. The reflector in Enigma only had three possible options [10] until January 1944, when a rewirable reflector was introduced. Still, even then the wiring of the reflector was only changed every ten days because it was rather tedious [17]. For this reason, we also ignore the reflector.

This ultimately leaves the plugboard as the sole remaining unknown variable. Revisiting the simplified and substituted equation of the characteristic AD:

$$AD = SQS^{-1} \tag{3.22}$$

We can see that permutation S is the primary obstacle remaining, but unfortunately there isn't a way to isolate S any further. The plugboard itself can pair any two letters together resulting in a maximum of 13 pairs in an alphabet with 26 letters, which is the case in English and German. Consequently, one ends up with $537,985,208,200,576$ possible different permutations for the plugboard [31]. Due to the sheer amount of possible combinations, brute forcing the plugboard is not currently feasible within a reasonable amount of time. If a portion of the encrypted message could be guessed, providing a section of ciphertext with its corresponding plaintext of a suitable length, this section could be combined with the information in the catalog in order

to work out the unknown plugboard pairings. This approach was used during the Second World War, especially for Turing's Bombe, for determining the plugboard connections.

3.6 Recursive Algorithm

Most of the approaches used to determine the plugboard wirings during the war, including Turing's Bombe, relied on the fact that very rarely would the maximum amount of 13 pairings be used. Due to the increase in capabilities of modern computer software and hardware, attempts can be made to solve the plugboard even with all 13 pairings used. This report implements a recursive algorithm that uses the matching plaintext/ciphertext segment, or crib, as a starting point and then builds outwards trying different pairings until either a contradiction is found or the crib is exceeded. In the case of a contradiction, the function will then backtrack trying other possible pairings. While many possible pairings exist, a great many of them will end in contradiction and are not possible solutions. Any feasible solutions are returned when the crib is exceeded, so the end user can check them to see if they result in a proper decryption of the message.

Chapter 4

Solving and Implementing Enigma

4.1 Introduction

The programs constructed for this report are two separate encryption and decryption programs. The encryption program is a software implementation of a three rotor Enigma, commonly referred to as an M3, that will encrypt a given message with user defined settings and output the result. The decryption program is an implementation of the methods described in chapter 3 that will attempt to defeat the encryption of a provided message and return all viable solutions to be checked by the end user for one that results in an understandable communication.

The programs for this report do not implement all variables for the three rotor Enigma in order to reduce the amount of time needed to achieve a complete result. These programs do not implement the ring settings of the rotors, any removable reflectors, the rotor selection and order, and double stepping. Frequent testing was required to fine tune the recursive plugboard method, and simplifying the combinatorial possibilities during development allowed for more frequent iterations. The variables ignored were chosen due to the fact that they did not form the primary obstacles to solving Enigma and it should not be too difficult to extend the existing programs to account for these variables at a later date.

The programs were written in Maple due to its existing permutation libraries and numeric computing environment. This allowed the program to be prototyped and written relatively quickly and simply. The algorithms used can be implemented into any other language, provided one finds or creates a library for implementing permutations and operations on permutations.

4.2 Library

Outside the encryption and decryption programs written in Maple, several common methods were constructed and collected together into a file to form a library in order to simplify the development of either program. This library includes methods for converting letters into numerical offsets, generating all possible rotor rotation permutations, constructing the encryption permuta-

tion, advancing the rotors, retrieving the correspondingly mapped letter from another in a permutation, calculating the index used to map cycle structures to array locations, and solving the plugboard recursively. This library is imported into Maple as the first step of both programs. The complete source code for these methods can be found in Appendix A.1

4.2.1 Convert Letters to Numerical Offsets

Input:	ListOfCharacters	(List)
	Alphabet	(List)
Output:	ListOfAlphabetOffsets	(List)

The procedure *CalculateAlphabetOffsets* takes two lists as input: one containing a list of characters (*ListOfCharacters*), and another list containing an alphabet (*Alphabet*). Utilizing a FOR loop, the procedure iterates through each character in *ListOfCharacters* and makes comparisons to each character in *Alphabet* until it finds a match. Once finished, the procedure returns a list containing the index, or offset, into *Alphabet* for each matching character in *ListOfCharacters*. For example, the *Alphabet* list input would be $[A, B, C, D, E, F, ..., X, Y, Z]$ for English. So the list returned for *CalculateAlphabetOffsets* given for *ListOfCharacters*:=$[R, E, S, U, L, T, S]$ and *Alphabet*:=$[A, B, C, D, E, F, ..., X, Y, Z]$ would be $[18, 5, 19, 21, 12, 20, 19]$.

4.2.2 Generate Rotor Rotation Permutations

Input:	RotorStepPermutation	(List)
Output:	ListOfRotorRotationPermutations	(List)

The procedure *GenerateListOfRotorRotationPermutations* takes a permutation (*RotorStepPermutation*) in the form of a list as input. This permutation represents a single step, or rotation, of the rotor around the axle of the machine altering which paths form between the electrical contacts of the rotor and any adjacent rotors, reflectors, or entry wheels. This permutation is normally $[1, 2, 3, 4, 5, ...24, 25, 26]$ for one $\frac{1}{26}th$ clockwise rotation of the rotor around the axle. This method takes the step permutation and multiplies it by itself in order to produce all possible orientations of the rotors around the axle. This list is saved and returned in order to save these permutations in memory, reducing unnecessary repeated calculations.

4.2.3 Construct Enigma Encryption Permutation

Input:	ListOfRotorWirings	(List)
	RotorOffsets	(List)
	Reflector	(List)
	ListOfRotorRotationPermutations	(List)
	Plugboard	(List)
Output:	Output	(List)

The procedure *GenerateEnigmaEncryptionPermutationWithPlugboard* takes as input a list containing permutations simulating the internal wirings of the rotors (*ListOfRotorWirings*), a list containing the rotor settings (*RotorOffsets*), a list containing the permutation simulating the reflector (*Reflector*), a list containing all permutations used to simulate the rotor rotating around the axle of the machine for all possible orientations (*ListOfRotorRotationPermutations*), and a list with the permutation used to simulate the plugboard wirings (*Plugboard*). These permutations are then multiplied together to generate the returned list (*Output*) representing the encrypted Enigma permutation (Equation 3.5):

$$SP^iRP^{-i}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^iR^{-1}P^{-i}S^{-1}$$

where S is the plugboard permutation, $P^i$is the rotation permutation of rotor R at position i, R is the rightmost rotor, $P^j$is the rotation permutation of

rotor M at position j, M is the middle rotor, P^k is the rotation permutation of rotor L at position k, L is the leftmost rotor, and U is the reflector.

4.2.4 Step Rotors

Input:	RotorSettingOffsets	(List)
	Alphabet	(List)
Output:	NewRotorSettingOffsets	(List)

The procedure *AdvanceRotorSetting* takes as input a list containing the rotor settings (*RotorSettingOffsets*) and a list specifying the alphabet used (*Alphabet*) and then returns a new list (*NewRotorSettingOffsets*) containing new rotor settings resulting from advancing the specified rotor settings by one. The program currently does not account for double stepping, but this procedure can be altered in order to account for this.

4.2.5 Encrypt Letter

Input:	TextOffset	(Integer)
	EnigmaEncryptionPermutation	(List)
Output:	CiphertextOffset	(Integer)

The procedure *ExtractCiphertextOffsetFromEnigmaEncryptionPermutation* takes as input an integer (*TextOffset*) and a list (*EnigmaEncryptionPermuta-*

tion) and returns an integer (*CiphertextOffset*) which is the correspondingly mapped integer of *TextOffset* from *EnigmaEncryptionPermutation*. This essentially takes a letter represented as an alphabet offset and returns the encrypted letter as another alphabet offset.

4.2.6 Calculate Index into Catalog

Input:	GivenPartitionSignature	(List)
Output:	(Unnamed Return)	(Integer)

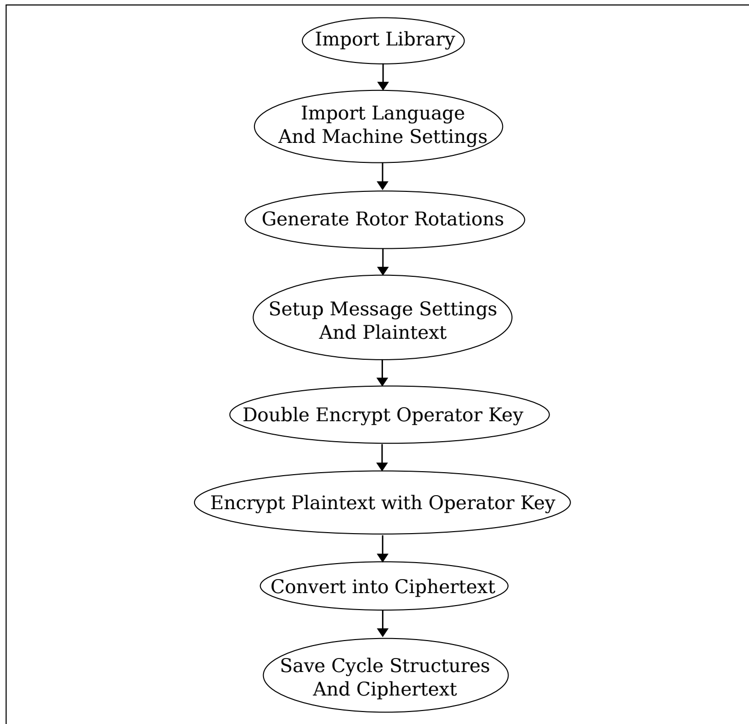
The procedure *GetIndex* takes as input a list specifying the cycle structure of a permutation (*GivenPartitionSignature*) and returns an integer that is calculated by multiplying by a fixed amount for each type of cycle breakdown and summing the results. This is a custom indexing method used to map into a list data structure that holds the settings that can result in the specified cycle structure. Custom indexing was used in order to minimize the amount of storage required due to the total amount of possible cycle structures.

4.2.7 Find All Possible Solutions for the Plugboard

Input:	LetterAssignments	(List)
	AvailableLetters	(List)
	TextOffsetsIndex	(Integer)
	ListOfPossibleSolutions	(List)
	EnigmaEncryptionPermutationWithoutPlugboard	(List)
	PlaintextOffsets	(List)
	CiphertextOffsets	(List)
Output:	ListOfPossibleSolutions	(List)

The procedure *DeterminePlugboardWirings* takes as input a list of letter assignments (*LetterAssignments*), a list of available letters (*AvailableLetters*), an interger which is an offset to mark the current position within the crib (*TextOffsetsIndex*), a list of the possible solutions found so far since this is a recursive procedure (*ListOfPossibleSolutions*), a list containing all the Engima encryption permutations without the influence of the plugboard for each letter of the crib (*EnigmaEncryptionPermuationWithoutPlugboard*), a list containing the alphabet offsets for each letter of the plaintext portion of the crib (*PlaintextOffsets*), a list containing the alphabet offsets for each letter of the ciphertext portion of the crib (*CiphertextOffsets*). The output for the procedure returns the local version of the *ListOfPossibleSolutions* for that recursive call.

4.3 Encryption Program



4.3.1 Overview

The first program written for this report is an encryption program that's intended to replicate Enigma's functionality by allowing the user to set variables that define the encryption keys and message plaintext, apply encryption to the message, and then output the generated ciphertext, as well as save the cycle structure of the characteristics for use in the decryption program. This program can be broken down in distinct phases which will be described step by step.

4.3.2 Program Setup:

4.3.2.1 Import Library

```
> restart:
▼ Importing All Procedure Created to Deal with Enigma
  > read("EnigmaProcedures.mpl"):
```

The first step of the encryption program is to import the constructed library described in Section 4.2. This library contains methods written in order to implement repeated Enigma specific functions for both the encryption and decryption programs implemented in this report. Here, the library is contained within the external *EnigmaProcedures.mpl* file. The file itself is simply Maple input placed in an external text file to simplify the program by not including the complicated procedures that were created for them. This leaves the core steps of the encryption process without the confusion and clutter of the logic of the encryption itself, so this is what students can initially focus on.

4.3.2.2 Import Language and Machine Settings

```
▼ Importing Machine Settings
  > read("Test05Encrypt.mpl"):
```

Next, we import several variables from an external file in order to define the language, machine, and key settings. These variables are lists that define the alphabet to be used, permutations specifying a single rotor rotation around

the axle, the wirings of the available rotors, the order of the rotors, the wiring of the reflector, plugboard wirings, the daily setting, the operator setting, and the plaintext. These settings are defined in an external file, but could just as easily be included in the program itself. The contents of *Test05Encrypt.mpl* in this example looks like the following:

```
> Alphabet := ["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"];
>
> PermutationToSimulateSingleRotationAroundAxle := [
  [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]];
>
> ListOfRotorWirings := [[[1, 5, 12, 20, 16, 8, 17,
  24, 18, 21], [2, 11, 14, 23], [3, 13, 15, 25], [4,
  6, 7], [9, 22], [10, 26]], [[2, 10], [3, 4, 11, 12,
  8, 21, 16], [5, 19, 26], [6, 9, 24, 22, 25, 15, 13,
  23], [7, 18], [14, 20]], [[1, 2, 4, 8, 16, 5, 10,
  20], [3, 6, 12, 22, 13, 26, 15, 25, 17, 9, 18, 23,
  21, 11, 24, 19, 7]], [[1, 5, 16, 12, 9, 25, 23, 3,
  15, 24, 13, 18, 6, 26, 2, 19, 20, 7, 10, 17, 14,
  8], [4, 22], [11, 21]], [[1, 22, 15, 12, 4, 18, 23,
  6, 9, 21, 17], [2, 26, 11, 19, 13, 14, 8, 25, 3],
  [5, 7, 20, 10, 16, 24]]];
>
> ListOfSelectedRotorsLeftToRight :=
  [ListOfRotorWirings[4], ListOfRotorWirings[2],
  ListOfRotorWirings[5]];
```

```
> Reflector := [[1, 25], [2, 18], [3, 21], [4, 8],
  [5, 17], [6, 19], [7, 12], [9, 16], [10, 24], [11,
  14], [13, 15], [20, 26], [22, 23]];
>
> Plugboard := [[5,26], [2,12], [16,24]];
>
> DailySetting := "XCD";
> OperatorSetting := "RHC";
> Plaintext :=
  "ANMARTINOBERZALEKXKEINEBESONDERENEREIGNISSEINDERBUL
  MEHEUTE";
```

Here, we have the alphabet and a permutation used to simulate a single rotation around the axle defined. The axle permutation is P^1 in the previously explained Enigma encryption permutation (Equation 3.5):

$SP^iRP^{-i}P^jMP^{-j}P^kLP^{-k}UP^kL^{-1}P^{-k}P^jM^{-1}P^{-j}P^iR^{-1}P^{-i}S^{-1}$

Next, we have the list of rotor wirings and the selection of three of these that will be loaded into the machine. Here, *ListOfSelectedRotorsLeftToRight* uses rotor 4, 2, 5. The reflector and plugboard are then defined as well. Finally, the daily setting; operator setting; and plaintext of the messages are specified.

All of these variables can be modified freely by the end user. The goal is to allow a student of cryptography freedom to run any sort of test they wish with the program. This external file is simply Maple input saved as text and imported into the program. This means that these statements can be added back into the program rather than an external file by simply replacing the import statement with the file's contents if one wanted to. An import and

export scheme was chosen because it would allow a clear separation of the input from the procedures of the program, especially between the encryption and decryption programs, so that one could be confident that the program was only operating on what it had been given as input.

The open design of the programs were done to allow students full access to the procedures that make up this implementation so that they can experiment with modifications to further their understanding of permutation encryption schemes like Enigma. Since all the variables and procedures are freely available for modification, it would be best to categorize them as variables used as user settings, modifications to the variables and procedures that allow for the creation of Enigma variants, and modifications that allow for the creation of variants that go beyond Enigma. The user settings are the variables that define the different elements of the key used for encryption. These are the *ListOfSelectedRotorsLeftToRight*, *Reflector*, *Plugboard*, *DailySetting*, *OperatorSetting*, and *Plaintext* variables. These variables will change for each test the end user wishes to perform since they form the core of the encryption process.

The first categorization of modifications that are possible with the program are changes to the variables and procedures that allow for the creation of Enigma variants. Various Enigma variants exist as either commercial or military models used before, during, or after the war [13]. Modifications to the program which would allow the implementation of an Enigma variant involve alterations to the variables and methods that control the functions

where Enigma variants frequently differ. Variants differ in the language used, the number of rotors used for encryption, the number and wirings of all available rotors, the stepping function, the turnover positions, the number and wirings of the reflector, and the wiring of the entry wheel. This last one would be essentially a static rotor, so it could be potentially included in the rotor wirings. The elements that would have to be modified within the program to modify all of these would be the variables: *Alphabet*, *PermutationToSimulateSingleRotationAroundAxle*, *ListOfRotorWirings*, *ListOfSelectedRotorsLeftToRight*, *Reflector*, *Plugboard*. The Enigma procedures in *EnigmaProcedures.mpl* that would need to be modified to implement some of these changes are *AdvanceRotorSettings* to implement new stepping and turnover logic and *GenerateEngimaEncryptionPermutationWithPlugboard* to alter the way Enigma is encrypted. It is worth mentioning that the program only implements a naive turnover logic at the moment that turnovers the rotor on its left when it has looped around its alphabet. To implement a better and accurate turnover system would involve creating a new list that would mark which positions a notch is located that would cause it to catch and turn over the rotor to its left. This list would be checked in the *AdvanceRotorSettings* to see if it has reached a turnover position and, if it has, then turn over its rotor to its left when it has advanced. It is also possible that the worksheet itself might require some basic modification depending on the alterations being made so that all steps of the encryption proceed smoothly. This should be done especially if one was going to implement changes to how

the daily and operator keys were used, because this was not standard across the entire war and with all variants [6, 11, 24].

The final types of modification that one could do to the program is to implement related rotor machines that have a shared lineage with Enigma. These machine include the Hagelin B-21, SIGABA, Typex, NEMA, KL-7, Fialka, and Tatjana van Vark's Enigma Inspired Rotor Machine [12, 39]. Such wide ranging modifications have not been attempted with the program at this time, but it is hoped that its open nature could allow implementations of these machines. Most of these modifications should involve the same procedures and variables used for variants, but could potentially involve additional modifications going beyond the work required to implement an Enigma variant.

4.3.2.3 Generate Rotor Rotations

The following section of the program concerns the rotor rotations around the axle of the machine, representing this as a permutation to apply in order to generate the encryption permutation that is used to convert a letter of plaintext to a letter of ciphertext. First, this section utilizes a list variable that was imported from an external file. This list is a permutation that is used to simulate a single step of the rotor around the axle. Secondly, this permutation is passed into the *GenerateListOfRotorRotationPermutations* procedure defined in the library and describe in Subsection 4.2.2 in order to generate all possible rotor orientation permutations and stores these in a list

variable in order to save on repeated calculations.

Precreate List of Permutations that are Used to Simulate the Rotor Rotating Around the Axle of the Machine
****We Do This Now To Save on Having To Recalculate This With Every Rotor Advance****

```
> ListOfPermutationsUsedToSimulateRotorRotationsAroundA\
    xle :=
    GenerateListOfRotorRotationPermutations(
    PermutationToSimulateSingleRotationAroundAxle);
ListOfPermutationsUsedToSimulateRotorRotationsAround\                (3.1)
    Axle := [[ ], [[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]], [[1, 3,
    5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25], [2, 4, 6, 8, 10, 12,
    14, 16, 18, 20, 22, 24, 26]], [[1, 4, 7, 10, 13, 16, 19,
    22, 25, 2, 5, 8, 11, 14, 17, 20, 23, 26, 3, 6, 9, 12, 15, 18,
    21, 24]], [[1, 5, 9, 13, 17, 21, 25, 3, 7, 11, 15, 19, 23],
    [2, 6, 10, 14, 18, 22, 26, 4, 8, 12, 16, 20, 24]], [[1, 6,
    11, 16, 21, 26, 5, 10, 15, 20, 25, 4, 9, 14, 19, 24, 3, 8,
    13, 18, 23, 2, 7, 12, 17, 22]], [[1, 7, 13, 19, 25, 5, 11,
    17, 23, 3, 9, 15, 21], [2, 8, 14, 20, 26, 6, 12, 18, 24, 4,
    10, 16, 22]], [[1, 8, 15, 22, 3, 10, 17, 24, 5, 12, 19, 26,
    7, 14, 21, 2, 9, 16, 23, 4, 11, 18, 25, 6, 13, 20]], [[1, 9,
    17, 25, 7, 15, 23, 5, 13, 21, 3, 11, 19], [2, 10, 18, 26, 8,
    16, 24, 6, 14, 22, 4, 12, 20]], [[1, 10, 19, 2, 11, 20, 3,
    12, 21, 4, 13, 22, 5, 14, 23, 6, 15, 24, 7, 16, 25, 8, 17,
```

Here in this example we have $P^1 = [1, 2, 3...26]$ and precalculate $P^0...P^{25}$. The output here is truncated for space reasons, but the full output can be found in Appendix

4.3.3 Message Setup

4.3.3.1 Setup Message Settings and Plaintext

Defining and Formatting the Message Settings and the Plaintext

```
> with(StringTools,'LengthSplit', 'UpperCase') :
> DailySetting := [LengthSplit(UpperCase(DailySetting), 1)];
                DailySetting := ["X", "C", "D"]                    (4.1)
> OperatorSetting :=
      [LengthSplit(UpperCase(OperatorSetting), 1)];

              OperatorSetting := ["R", "H", "C"]                   (4.2)
> Plaintext := [LengthSplit(UpperCase(Plaintext), 1)];
Plaintext := ["A", "N", "M", "A", "R", "T", "I", "N", "O", "B",    (4.3)
    "E", "R", "Z", "A", "L", "E", "K", "X", "K", "E", "I", "N", "E",
    "B", "E", "S", "O", "N", "D", "E", "R", "E", "N", "E", "R", "E",
    "I", "G", "N", "I", "S", "S", "E", "I", "N", "D", "E", "R", "B",
    "U", "L", "M", "E", "H", "E", "U", "T", "E"]
```

Next, the message settings and plaintext are defined. These variables are included at the end of the *Test05Encrypt.mpl* file, which can be seen in Subsection . *DailySetting*, *OperatorSetting*, and *Plaintext* are defined here as strings and are then converted into list variables and all uppercase letters by utilizing the *LengthSplit* and *UpperCase* methods from Maple's *StringTools* package. This step is illustrated in the screen capture above.

```
> OperatorSetting :=
      [LengthSplit(UpperCase(OperatorSetting), 1)];

              OperatorSetting := ["R", "H", "C"]                    (4.2)

> Plaintext := [LengthSplit(UpperCase(Plaintext), 1)];
Plaintext := ["A", "N", "M", "A", "R", "T", "I", "N", "O", "B",     (4.3)
    "E", "R", "Z", "A", "L", "E", "K", "X", "K", "E", "I", "N", "E",
    "B", "E", "S", "O", "N", "D", "E", "R", "E", "N", "E", "R", "E",
    "I", "G", "N", "I", "S", "S", "E", "I", "N", "D", "E", "R", "B",
    "U", "L", "M", "E", "H", "E", "U", "T", "E"]

> Plaintext := [op(OperatorSetting), op(OperatorSetting),
      op(Plaintext)];

Plaintext := ["R", "H", "C", "R", "H", "C", "A", "N", "M", "A",     (4.4)
    "R", "T", "I", "N", "O", "B", "E", "R", "Z", "A", "L", "E", "K",
    "X", "K", "E", "I", "N", "E", "B", "E", "S", "O", "N", "D", "E",
    "R", "E", "N", "E", "R", "E", "I", "G", "N", "I", "S", "S", "E", "I",
    "N", "D", "E", "R", "B", "U", "L", "M", "E", "H", "E", "U", "T",
    "E"]
```

After converting these three variables into a list of single uppercase characters, we insert the operator key twice before the plaintext due to the actual practice of enciphering this information twice before each message during transmission. Here, the operator setting "RHC" is inserted twice before the plaintext to form "RHCRHCANMARTINOBERZALEKXKEINEBESOND ERENEREIGNISSEINDERBULMEHUTE."

```
> DailySettingOffsets :=
      CalculateAlphabetOffsets(DailySetting, Alphabet);
              DailySettingOffsets := [24, 3, 4]                    (4.5)
> OperatorSettingOffsets :=
      CalculateAlphabetOffsets(OperatorSetting, Alphabet);
              OperatorSettingOffsets := [18, 8, 3]                 (4.6)
> PlaintextOffsets := CalculateAlphabetOffsets(Plaintext,
      Alphabet);
PlaintextOffsets := [18, 8, 3, 18, 8, 3, 1, 14, 13, 1, 18, 20, 9,  (4.7)
    14, 15, 2, 5, 18, 26, 1, 12, 5, 11, 24, 11, 5, 9, 14, 5, 2, 5,
    19, 15, 14, 4, 5, 18, 5, 14, 5, 18, 5, 9, 7, 14, 9, 19, 19, 5,
    9, 14, 4, 5, 18, 2, 21, 12, 13, 5, 8, 5, 21, 20, 5]
```

Once set, these list variables are converted from single characters to numerical offsets of the alphabet by calling the custom *CalculateAlphabetOffset* defined in *EnigmaProcedures.mpl*. These numbers are specifically an alphabet offset which will allow calculations to be performed during the process of encrypting the plaintext into ciphertext.

4.3.3.2 Double Encrypt Operator Key

This section of the program begins the actual encryption process by encrypting the first six letters of the ciphertext, which are the operator key enciphered twice and inserted before the message plaintext.

Encrypting the Defined Plaintext Message Using the Previously Defined Settings
****Note What We Keep the Generated Permutations for Each Character in a List. This is for Debugging purposes****

First we Encrypt the First Six Letters from the Plaintext that Contain the Message Setting

```
> CurrentRotorOffsets := DailySettingOffsets;
              CurrentRotorOffsets := [24, 3, 4]                (5.1.1)
> EnigmaEncryptionPermutationsWithPlugboard := [ ];
  EnigmaEncryptionPermutationsWithPlugboard := [ ]             (5.1.2)
```

This is done by assigning a list variable controlling the left, middle, and right rotor settings to the starting position specified by the daily key.

```
> for i from 1 to 6 do
    CurrentRotorOffsets :=
      AdvanceRotorSettings(CurrentRotorOffsets,
       Alphabet);

    EnigmaEncryptionPermutationsWithPlugboard :=
      [op(EnigmaEncryptionPermutationsWithPlugboard),
      GenerateEnigmaEncryptionPermutationWithPlugboar\
      d(ListOfSelectedRotorsLeftToRight,
       CurrentRotorOffsets, Reflector,
       ListOfPermutationsUsedToSimulateRotorRotationsA\
      roundAxle, Plugboard)];


      print(EnigmaEncryptionPermutationsWithPlugboard[
      i]);
  end do:
[[1, 26], [2, 14], [3, 22], [4, 17], [5, 18], [6, 10], [7,
    21], [8, 11], [9, 23], [12, 19], [13, 20], [15, 24],
    [16, 25]]
```

The rotor settings are then advanced before encrypting each of the first six converted alphabet offsets of the message, by generating the encryption permutation for each alphabet offset and storing this in a list for future reference (*EnigmaEncryptionPermutationsWithPlugboard*). The generation of the encryption permutation for each alphabet offset is done by calling *GenerateEnigmaEncryptionPermutationWithPlugboard* defined in the library and described in Subsection 4.2.3 with the appropriate parameters.

Here, the first six letters of the message are encrypted with the daily setting. In this case, "RHCRHC" is encrypted with the $[X, C, D]$ rotor settings. It is worth noting that the output is truncated here for space, but the complete output can be found in Appendix

4.3.4 Encryption and Output

4.3.4.1 Encrypt Plaintext with Operator key

This section of the program continues from the previous and encrypts the rest of the message plaintext converted into alphabet offsets. It starts by setting the rotor positions to the operator key and then uses the same process from the previous section to generate the encryption permutations of each alphabet offset and appends this to same list from the previous step that contains each encryption permutation for each element of the message plaintext (*EnigmaEncryptionPermutationsWithPlugboard*).

Next we Encrypt the Rest of the Message

```
> CurrentRotorOffsets := OperatorSettingOffsets;
            CurrentRotorOffsets := [18, 8, 3]                    (5.2.1)

> for i from 7 to nops(Plaintext) do
    CurrentRotorOffsets :=
      AdvanceRotorSettings(CurrentRotorOffsets,
       Alphabet);
    EnigmaEncryptionPermutationsWithPlugboard :=
      [op(EnigmaEncryptionPermutationsWithPlugboard),
      GenerateEnigmaEncryptionPermutationWithPlugboar\
      d(ListOfSelectedRotorsLeftToRight,
       CurrentRotorOffsets, Reflector,
       ListOfPermutationsUsedToSimulateRotorRotationsA\
      roundAxle, Plugboard)];


      print(EnigmaEncryptionPermutationsWithPlugboard[
      i]);
  end do:
[[1, 15], [2, 9], [3, 6], [4, 13], [5, 8], [7, 10], [11,
    12], [14, 21], [16, 23], [17, 26], [18, 24], [19, 25],
    [20, 22]]
[[1, 8], [2, 7], [3, 21], [4, 26], [5, 9], [6, 13], [10,
    12], [11, 14], [15, 20], [16, 22], [17, 18], [19, 25],
    [23, 24]]
```

Here the remaining letters of “ANMARTINOBERZALEKXKEINEBESONDERENEREIGNISSEINDERBULMEHUTE” are encrypted with the $[R, H, C]$ rotor settings. Once again, the output is truncated, but can be found in full in Appendix

4.3.4.2 Convert into Ciphertext

The next section of the encryption program involves extracting the cor-

respondingly mapped characters of the message plaintext as alphabet offsets and converting this back into encrypted letters. This is done by calling the *ExtractCiphertextOffsetFromEnigmaEncryptionPermutation* procedure defined in the library and described in Subsection 4.2.5 for each encryption permutation stored in the list that contains all the encryption permutations for every element of the message plaintext (*EnigmaEncryptionPermutationsWithPlugboard*). The encrypted letter in the form of an alphabet offset is converted back into a character by taking the letter at that index from the list specifying the alphabet to be used. This letter is then concatenated to form a string representing the ciphertext of the encrypted message.

Finally, We Need To Extract the Appropriate Letter From Each Permutation to get the Ciphertext

```
> Ciphertext := "";
                         Ciphertext := ""                        (5.3.1)
> for i from 1 to nops(PlaintextOffsets) do
    Ciphertext := cat(Ciphertext,
      Alphabet[
      ExtractCiphertextOffsetFromEnigmaEncryptionPerm\
      utation(PlaintextOffsets[i],
       EnigmaEncryptionPermutationsWithPlugboard[i])]);

  end do:
> Ciphertext;
"EURQNROKUDFZLGLUNJHIYIMWCXJBMUTKXAWLHGB\                       (5.3.2)
    FKCCQUZKFFAPRDBGBDOXSSXLG"
```

Here, the offsets are indexed into the alphabet permutation (*Alphabet*) and converted back into letters constructing the encrypted ciphertext of "EU-

RQNROKUDFZLGLUNJHIYIMWCXJBMUTKXAWLHGBFKCCQUZKF-FAPRDBGBDOXS SXLG"

4.3.4.3 Save Characteristic Sets AD, BE, CF and Ciphertext

As the final step of the program, the cycle structures of the characteristics are calculated and exported along with the string containing the ciphertext to an external file. The cycle structures of the characteristics, which are defined in Section 3.3, are saved since these can be utilized for the catalog method used in the decryption program. The characteristics AD, BE, CF are calculated by multiplying the first and fourth, the second and fifth, and the third and sixth encryption permutations together. These three lists, each one containing a characteristic, are saved along with the string containing the ciphertext to an external file so that the results of the encryption program can be fed into the decryption program if desired. The decryption program additionally requires a plaintext, ciphertext crib, and crib offset.

Construct Characteristic Sets AD, BE, and CF and Save Them and the Ciphertext to an External File.

```
> ConstructedDoubleEncryptionPermutation1WithPlugboard
    :=
    mulperms(
    EnigmaEncryptionPermutationsWithPlugboard[1],
    EnigmaEncryptionPermutationsWithPlugboard[1
    + nops(ListOfSelectedRotorsLeftToRight)]);
ConstructedDoubleEncryptionPermutation1WithPlugboard        (6.1)
    := [[1, 7, 10, 11, 25, 15, 20, 9, 12, 4, 18, 22, 2], [3, 5,
    17, 19, 23, 13, 24, 16, 8, 6, 21, 26, 14]]
```

Here the AD characteristic is calculated to be $[1, 7, 10, 11, 25, 15, 20, 9, 12, 4, 18, 22, 2]$ $[3, 5, 17, 19, 23, 13, 24, 16, 8, 6, 21, 26, 14]$.

```
> ConstructedDoubleEncryptionPermutation2WithPlugboard
    :=
    mulperms(
    EnigmaEncryptionPermutationsWithPlugboard[2],
    EnigmaEncryptionPermutationsWithPlugboard[2
    + nops(ListOfSelectedRotorsLeftToRight)]);
ConstructedDoubleEncryptionPermutation2WithPlugboard        (6.2)
    := [[1, 16, 5, 4, 23, 7, 6, 13], [2, 22, 21, 14, 3], [8, 9,
    20, 11, 19], [10, 15, 18, 24, 17, 26, 12, 25]]
```

The BE characteristic is calculated to be $[1, 16, 5, 4, 23, 7, 6, 13]$ $[2, 22, 21, 14, 3]$ $[8, 9, 20, 11, 19]$ $[10, 15, 18, 24, 17, 26, 12, 25]$.

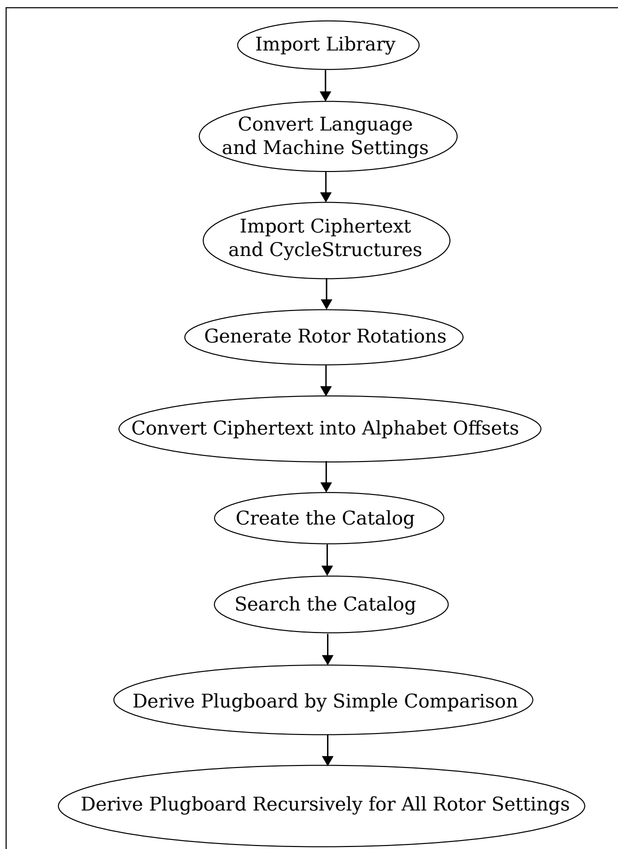
```
> ConstructedDoubleEncryptionPermutation3WithPlugboard
    :=
    mulperms(
    EnigmaEncryptionPermutationsWithPlugboard[3],
    EnigmaEncryptionPermutationsWithPlugboard[3
    + nops(ListOfSelectedRotorsLeftToRight)]);
ConstructedDoubleEncryptionPermutation3WithPlugboard        (6.3)
    := [[1, 22, 8, 24, 16, 11, 7, 4, 5, 23, 19], [2, 10, 17, 26,
    14, 9, 20, 12, 15, 25, 6]]
```

The CF characteristic is calculated to be $[1, 22, 8, 24, 16, 11, 7, 4, 5, 23, 19]$ $[2, 10, 17, 26, 14, 9, 20, 12, 15, 25, 6]$.

```
> save(Ciphertext,
    ConstructedDoubleEncryptionPermutation1WithPlugboa\
   rd,
    ConstructedDoubleEncryptionPermutation2WithPlugboa\
   rd,
    ConstructedDoubleEncryptionPermutation3WithPlugboa\
   rd, "Test05EncryptResults02.mpl") :
```

These results are now saved to a file.

4.4 Decryption Program



4.4.1 Overview

The second program written for this report is the decryption program that takes ciphertext, the characteristic sets, a crib and a crib offset as input and then proceeds to generate a catalog for every rotor setting, find a list of possible rotor settings that match the cycle structure of the characteristic sets, and then attempt to derive the plugboard connections. The program can be broken down into sections and will be described step by step.

4.4.2 Program Setup:

4.4.2.1 Import Library

```
> restart:
Importing All Procedure Created to Deal with Enigma
> read("EnigmaProcedures.mpl"):
```

The first section of the program is to import the procedures defined in the library described in Section 4.2. This library contains Enigma specific procedures that are utilized by both programs. Like the encryption program, *EngimaProcedures.mpl* is the same text file containing Maple input imported here as well.

4.4.2.2 Import Language and Machine Settings

```
Importing Machine Settings
> read("MachineSettingsActual_01.mpl") :
```

Next, we import several variables from an external file in order to define the language and machine settings.

```
> Alphabet := ["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"];
>
> PermutationToSimulateSingleRotationAroundAxle := [
  [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]];
>
> ListOfRotorWirings := [[[1, 5, 12, 20, 16, 8, 17,
  24, 18, 21], [2, 11, 14, 23], [3, 13, 15, 25], [4,
  6, 7], [9, 22], [10, 26]], [[2, 10], [3, 4, 11,
  12, 8, 21, 16], [5, 19, 26], [6, 9, 24, 22, 25,
  15, 13, 23], [7, 18], [14, 20]], [[1, 2, 4, 8, 16,
  5, 10, 20], [3, 6, 12, 22, 13, 26, 15, 25, 17, 9,
  18, 23, 21, 11, 24, 19, 7]], [[1, 5, 16, 12, 9,
  25, 23, 3, 15, 24, 13, 18, 6, 26, 2, 19, 20, 7,
  10, 17, 14, 8], [4, 22], [11, 21]], [[1, 22, 15,
  12, 4, 18, 23, 6, 9, 21, 17], [2, 26, 11, 19, 13,
  14, 8, 25, 3], [5, 7, 20, 10, 16, 24]]];
>
> ListOfSelectedRotorsLeftToRight :=
  [ListOfRotorWirings[4], ListOfRotorWirings[2],
  ListOfRotorWirings[5]];
>
> Reflector := [[1, 25], [2, 18], [3, 21], [4, 8],
  [5, 17], [6, 19], [7, 12], [9, 16], [10, 24], [11,
  14], [13, 15], [20, 26], [22, 23]];
```

These variables are lists that define the alphabet to be used, permutations specifying a single rotor rotation around the axle, the wirings of the available rotors, the order of the rotors, and the wiring of the reflector. Like the encryption program, the alphabet and the permutation to simulate a single rotation around the axle are included. This allows the end user to run decryption scenarios involving Enigma variants that function on a different alphabet and rotation scheme.

To elaborate further, these variables are part of the user settings along with the information in the next section. The difference between the user settings of the encryption program and the user settings of the decryption program is that the *Plugboard*, *DailySetting* and *OperatorSetting* variables are not included. This is because these variables are used for encryption and are part of the key. The variables *ListOfSelectedRotorsLeftToRight* and, depending on the Enigma model, *Reflector* also form part of the key. The program does not currently iterate over the different permutations possible for the rotors and the reflectors. This was done to primarily place the focus on solving the plugboard which is the most combinatorially complex component of Enigma's decryption. These additions can be easily made to the program by removing these variables as part of the user settings and adding FOR loops to iterate over all possible permutations to the decryption program worksheet.

Like the encryption program, possible modifications can be made to support Enigma variants here as well. Changes to the alphabet used, the wirings of all available rotors, the numbers and order of the selected rotors, all

possible reflector wirings, stepping, and turnover positions are all possible. To implement these changes, one would perform modifications to the *Alphabet*, *ListOfRotorWirings*, *ListOfSelectedRotorsLeftToRight*, and *Reflector* variables and the *AdvanceRotorSettings* and *GenerateEnigmaEncryptionPermutationWithPlugboard* methods in the *EnigmaProcedures.mpl* text file. By implementing Enigma variants, one could use the decryption methods on any number of machines and judge their effectiveness.

4.4.2.3 Import Ciphertext and Cycle Structures

Importing Message Information (Ciphertext, Characteristics, etc...)

```
> read("Test05Decrypt02.mpl") :
```

We first start by importing the message information available defined here in file *Test05Decrypt02.mpl*. The contents of this file can be seen below.

```
> Ciphertext :=
> "EURQNROKUDFZLGLUNJHIYIMWCXJBMUTKXAWLHGBFKCCQUZKFFA
  PRDBGBDOXSSXLG";
> CribCiphertext := "CXJBMUTKXAWLHGBFKCCQUZKFF";
> CribPlaintext := "KEINEBESONDERENEREIGNISSE";
> CribOffset := 18;
> ConstructedDoubleEncryptionPermutation1WithPlugboar
  d := [[1, 7, 10, 11, 25, 15,
> 20, 9, 12, 4, 18, 22, 2], [3, 5, 17, 19, 23, 13,
  24, 16, 8, 6, 21, 26, 14]];
> ConstructedDoubleEncryptionPermutation2WithPlugboar
  d := [[1, 16, 5, 4, 23, 7, 6
> , 13], [2, 22, 21, 14, 3], [8, 9, 20, 11, 19],
  [10, 15, 18, 24, 17, 26, 12, 25]
> ];
> ConstructedDoubleEncryptionPermutation3WithPlugboar
  d := [[1, 22, 8, 24, 16, 11,
> 7, 4, 5, 23, 19], [2, 10, 17, 26, 14, 9, 20, 12,
  15, 25, 6]];
>
```

This section imports all three characteristics, a string containing the ciphertext of the encrypted message, a string containing the plaintext of the crib, a string containing the corresponding ciphertext of the plaintext crib, and a number to tell you how many letters are offset from the ciphertext before the crib begins from an external file. Most of this file is generated at the end of the encryption program with the exception of the crib ciphertext, crib plaintext, and crib offset. This information can be added to an encryption program output file or manually created with the necessary variable assignments. As before, this file is simply a text file containing Maple input. The decryption program uses all of this information as part of its decryption by

utilizing the catalog method or the recursive plugboard method.

Possible modifications here include the ability to support other decryption methods. Examples of other possible decryption methods that could be implemented to replace or supplement the existing ones would be the grill method, the Zygalski sheets, the Turing-Welchman Bombe, crib-dragging, or James J. Gillogly's ciphertext only attack [19]. Such changes would require alterations to the primary worksheet and possibly the *EnigmaProcedures.mpl* file as well. Like the use of the catalog or the recursive plugboard methods, it would be useful for students to observe the strengths and weaknesses of various approaches as they compare and contrast decryption methods.

4.4.2.4 Generate Rotor Rotations

Precreate List of Permutations that are Used to Simulate the Rotor Rotating Around the Axle of the Machine **We Do This Now To Save on Having To Recalculate This With Every Rotor Advance**

```
> ListOfPermutationsUsedToSimulateRotorRotationsAroundA\
      xle :=
      GenerateListOfRotorRotationPermutations(
      PermutationToSimulateSingleRotationAroundAxle);
ListOfPermutationsUsedToSimulateRotorRotationsAround\                (4.1)
   Axle := [[ ], [[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]], [[1, 3,
   5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25], [2, 4, 6, 8, 10, 12,
   14, 16, 18, 20, 22, 24, 26]], [[1, 4, 7, 10, 13, 16, 19,
   22, 25, 2, 5, 8, 11, 14, 17, 20, 23, 26, 3, 6, 9, 12, 15, 18,
   21, 24]], [[1, 5, 9, 13, 17, 21, 25, 3, 7, 11, 15, 19, 23],
   [2, 6, 10, 14, 18, 22, 26, 4, 8, 12, 16, 20, 24]], [[1, 6,
```

Now that all our required information has been imported into the program, this section generates all possible rotor orientation permutations and stores these in a list variable in order to save on repeated calculations later. This is the same process used in the encryption program described in Subsection 4.2.2. The output here is truncated, but can be seen in full in Appendix

4.4.2.5 Convert Ciphertext into Alphabet Offsets

Now Proceeding to Try and Break the Ciphertext

Translate the Ciphertext into a List of Alphabet Offsets

```
> with(StringTools,'LengthSplit'):
> Ciphertext := [LengthSplit(Ciphertext, 1)];
Ciphertext := ["E", "U", "R", "Q", "N", "R", "O", "K", "U",        (5.1.1)
    "D", "F", "Z", "L", "G", "L", "U", "N", "J", "H", "I", "Y", "I",
    "M", "W", "C", "X", "J", "B", "M", "U", "T", "K", "X", "A",
    "W", "L", "H", "G", "B", "F", "K", "C", "C", "Q", "U", "Z",
    "K", "F", "F", "A", "P", "R", "D", "B", "G", "B", "D", "O",
    "X", "S", "S", "X", "L", "G"]
> CiphertextOffsets :=
      CalculateAlphabetOffsets(Ciphertext, Alphabet);
CiphertextOffsets := [5, 21, 18, 17, 14, 18, 15, 11, 21, 4,  (5.1.2)
    6, 26, 12, 7, 12, 21, 14, 10, 8, 9, 25, 9, 13, 23, 3, 24,
    10, 2, 13, 21, 20, 11, 24, 1, 23, 12, 8, 7, 2, 6, 11, 3, 3,
    17, 21, 26, 11, 6, 6, 1, 16, 18, 4, 2, 7, 2, 4, 15, 24, 19,
    19, 24, 12, 7]
```

This section of the program takes the imported ciphertext string and converts it into a list of alphabet offsets utilizing the *CalculateAlphabetOffsets* procedure defined in the library and described in Subsection 4.2.1. This is done in preparation for applying mathematical operations on the alphabet offsets which are treated as permutations.

```
> CribCiphertext := [LengthSplit(CribCiphertext, 1)];
CribCiphertext := ["C", "X", "J", "B", "M", "U", "T", "K",        (5.1.3)
    "X", "A", "W", "L", "H", "G", "B", "F", "K", "C", "C", "Q",
    "U", "Z", "K", "F", "F"]
> CribCiphertextOffsets :=
      CalculateAlphabetOffsets(CribCiphertext, Alphabet);
CribCiphertextOffsets := [3, 24, 10, 2, 13, 21, 20, 11,           (5.1.4)
    24, 1, 23, 12, 8, 7, 2, 6, 11, 3, 3, 17, 21, 26, 11, 6, 6]
> CribPlaintext := [LengthSplit(CribPlaintext, 1)];
CribPlaintext := ["K", "E", "I", "N", "E", "B", "E", "S", "O",    (5.1.5)
    "N", "D", "E", "R", "E", "N", "E", "R", "E", "I", "G", "N", "I",
    "S", "S", "E"]
> CribPlaintextOffsets :=
      CalculateAlphabetOffsets(CribPlaintext, Alphabet);
CribPlaintextOffsets := [11, 5, 9, 14, 5, 2, 5, 19, 15, 14,      (5.1.6)
    4, 5, 18, 5, 14, 5, 18, 5, 9, 7, 14, 9, 19, 19, 5]
```

The same is then done for both the ciphertext and plaintext versions of the crib.

4.4.3 Catalog Decryption

Due to the fact that the brute-force recursive method takes some time to return its results, a catalog search was implemented into the program in order to more effectively handle simpler and trivial test cases that may not require a full search as well as presenting another way of defeating Enigma's encryption.

4.4.3.1 Extract Cycle Structure from AD, BE, and CF

Generate Cycle Structure from AD, BE, CF

```
> ConstructedDoubleEncryptionPermutation1CycleStructur\
     e := [ ];
ConstructedDoubleEncryptionPermutation1CycleStruct\     (5.2.1)
     ure := [ ]
> for i from 1
     to
     nops(
     ConstructedDoubleEncryptionPermutation1WithPlugb\
     oard) do

     ConstructedDoubleEncryptionPermutation1CycleStruc\
     ture :=
     [op(
     ConstructedDoubleEncryptionPermutation1CycleStruc\
     ture),
      nops(
     ConstructedDoubleEncryptionPermutation1WithPlugb\
     oard[i]) ];
  end do:
> ConstructedDoubleEncryptionPermutation1CycleStructur\
     e :=
     sort(
     ConstructedDoubleEncryptionPermutation1CycleStruc\
     ture, `>`);
ConstructedDoubleEncryptionPermutation1CycleStruct\     (5.2.2)
     ure := [13, 13]
```

This first thing done for the catalog method is to take all three characteristics and calculate their cycle structures, to compare against a constructed catalog to identify a list of possible rotor settings that would generate those characteristics. The cycle structure is calculated by taking the size of the

disjointed permutations and sorting these in decreasing order. Here, we see that AD has a cycle structure of $[13, 13]$.

```
> ConstructedDoubleEncryptionPermutation2CycleStructur\
    e := [ ];
ConstructedDoubleEncryptionPermutation2CycleStruct\ (5.2.3)
    ure := [ ]
> for i from 1
    to
    nops(
    ConstructedDoubleEncryptionPermutation2WithPlugb\
    oard) do

    ConstructedDoubleEncryptionPermutation2CycleStruc\
    ture :=
    [op(
    ConstructedDoubleEncryptionPermutation2CycleStruc\
    ture),
     nops(
    ConstructedDoubleEncryptionPermutation2WithPlugb\
    oard[i]) ];
  end do:
> ConstructedDoubleEncryptionPermutation2CycleStructur\
    e :=
    sort(
    ConstructedDoubleEncryptionPermutation2CycleStruc\
    ture, `>`);
ConstructedDoubleEncryptionPermutation2CycleStruct\ (5.2.4)
    ure := [8, 8, 5, 5]
```

BE has a cycle structure of $[8, 8, 5, 5]$.

```
> ConstructedDoubleEncryptionPermutation3CycleStructur\
    e := [ ];
ConstructedDoubleEncryptionPermutation3CycleStruct\    (5.2.5)
    ure := [ ]
> for i from 1
    to
    nops(
    ConstructedDoubleEncryptionPermutation3WithPlugb\
    oard) do

    ConstructedDoubleEncryptionPermutation3CycleStruc\
    ture :=
    [op(
    ConstructedDoubleEncryptionPermutation3CycleStruc\
    ture),
     nops(
    ConstructedDoubleEncryptionPermutation3WithPlugb\
    oard[i])];
  end do:
> ConstructedDoubleEncryptionPermutation3CycleStructur\
    e :=
    sort(
    ConstructedDoubleEncryptionPermutation3CycleStruc\
    ture, `>`);
ConstructedDoubleEncryptionPermutation3CycleStruct\    (5.2.6)
    ure := [11, 11]
```

CF has a cycle structure of $[11, 11]$.

These cycle structures for AD, BE and CF are stored to use in the next steps.

4.4.3.2 Create the Catalog

This section of the program concerns the creation of a catalog and filling

this catalog with the rotor settings that result from any of the possible cycle structures of the characteristics. This is done by creating and initializing an array to store the rotor settings. This array has 1400 elements in order to assign one for each of the possible cycle structures of the characteristics. Assignment of a cycle structure of an array index is done through the use of the procedure *GetIndex,* declared in the library and described in Subsection 4.2.6, in order to provide an optimized mapping of cycle structures to array indexes. The array elements themselves store lists containing the rotor settings that generate that cycle structure. This is done by iterating through all the possible rotor setting ($26^3 = 17,576$, for English and German) and constructing the characteristic AD by multiplying the Enigma encryption permutation of the current setting with the setting three steps later. The cycle lengths are then calculated by iterating through the resulting product's list variable and returning the amount of elements contained within. The final step of this section of the program is to sort the calculated cycle lengths and to pass this to the *GetIndex* procedure and then store the current rotor settings in the returned index mapped to that cycle structure.

Create the Catalog of Rotor Settings Used to Generate Each Characteristic Set

```
> RotorPermutationsArray := array(1..1400) :
> for i from 1 to 1400 do
    RotorPermutationsArray[i] := { };
  end do:
```

Here, we initialize an array consisting of 1400 elements which will contain

our catalog.

```
> CurrentRotorOffsets := [1, 1, 1];
            CurrentRotorOffsets := [1, 1, 1]                (5.3.1)
> for i from 1 to nops(Alphabet) · nops(Alphabet)
      · nops(Alphabet) do

    InitialRotorSettings := CurrentRotorOffsets;
    CurrentRotorOffsets :=
      AdvanceRotorSettings(CurrentRotorOffsets,
       Alphabet);
    Permutation1 :=
      GenerateEnigmaEncryptionPermutationWithPlugbo\
      ard(ListOfSelectedRotorsLeftToRight,
       CurrentRotorOffsets, Reflector,
      ListOfPermutationsUsedToSimulateRotorRotations\
      AroundAxle, [ ]);
```

We then set the current rotor positions to an initial setting of $[A, A, A]$ to begin our iteration. We then proceed to iterate through all the possible rotor settings and calculate the first permutation taking into account that the electrical encryption is performed on the downward action of the key which closes the circuit. This permutation is the A component of the AD characteristic.

```
CurrentRotorOffsets2 := CurrentRotorOffsets;

for j from 1 to 3 do
 CurrentRotorOffsets2 :=
   AdvanceRotorSettings(CurrentRotorOffsets2,
    Alphabet);
end do:

Permutation2 :=
   GenerateEnigmaEncryptionPermutationWithPlugbo\
   ard(ListOfSelectedRotorsLeftToRight,
    CurrentRotorOffsets2, Reflector,
    ListOfPermutationsUsedToSimulateRotorRotation\
   sAroundAxle, [ ]);
ResultingPermutation := mulperms(Permutation1,
   Permutation2);
```

A second permutation is then calculated which is the permutation that will be the permutation that occurs once the rotor advances three more positions. This permutations is the D component of the AD characteristic. Finally, we multiply both permutation together to form characteristic AD.

```
FinalResult := [ ];

for j from 1 to nops(ResultingPermutation) do
  FinalResult := [op(FinalResult),
   nops(ResultingPermutation[j])];
end do:

Index := GetIndex(sort(FinalResult, `>`));

RotorPermutationsArray[Index] :=
   {op(RotorPermutationsArray[Index]),
   InitialRotorSettings};
end do:
```

Now we iterate through the cycles in characteristic AD and store the cycles lengths. The sort them in descending numerical order and get an index by calling the *GetIndex* method imported from the *EnigmaProcedures.mpl* file. We then append the initial setting which would have generated this AD characteristic at the calculated index within the catalog array.

4.4.3.3 Search the Catalog

Now that a catalog has been generated, associating every possible rotor setting with a characteristic cycle structure, we can retrieve a list of rotor settings that match the cycle structure of the characteristic AD given as input to the program. We can, however, narrow this down further by retrieving the list of rotor settings that could generate BE and CF as well. Now, we can create a reduced list of rotor settings by only counting those that could have generated characteristic AD, have a matching rotor setting one rotor step away in the list of rotor settings that could have generated BE, and have another matching rotor setting two rotor steps away in the list of rotor settings that could have generated CF.

Search Catalog for all Possible Rotor Settings that could Generate these Characteristics

```
> ListOfPossibleRotorSettingsForConstructedDoubleEncrypt\
      ionPermutation1 :=
      RotorPermutationsArray[GetIndex(
      ConstructedDoubleEncryptionPermutation1CycleStruc\
      ture)]:
> ListOfPossibleRotorSettingsForConstructedDoubleEncrypt\
      ionPermutation2 :=
      RotorPermutationsArray[GetIndex(
      ConstructedDoubleEncryptionPermutation2CycleStruc\
      ture)]:
> ListOfPossibleRotorSettingsForConstructedDoubleEncrypti\
      onPermutation3 :=
      RotorPermutationsArray[GetIndex(
      ConstructedDoubleEncryptionPermutation3CycleStruc\
      ture)]:

> ListOfPossibleRotorSettingsForAllConstructedDoubleEncry\
      ptionPermutations := [ ];
ListOfPossibleRotorSettingsForAllConstructedDoubleEnc\          (5.4.1)
    ryptionPermutations := [ ]
```

We start by retrieving all the possible rotor settings for characteristics AD, BE, and CF. We also initialize a list that will contain our final candidates that could have resulting in all three characteristics.

```
> for i from 1
    to
    nops(
    ListOfPossibleRotorSettingsForConstructedDoubleE\
    ncryptionPermutation1) do:
 CurrentCandidate :=
    ListOfPossibleRotorSettingsForConstructedDoubleE\
    ncryptionPermutation1[i];

 for j from 1
    to
    nops(
    ListOfPossibleRotorSettingsForConstructedDoubleE\
    ncryptionPermutation2) do
  if (AdvanceRotorSettings(CurrentCandidate,
    Alphabet)
    =
    ListOfPossibleRotorSettingsForConstructedDoubleE\
    ncryptionPermutation2[j]) then
   for k from 1
    to
    nops(
    ListOfPossibleRotorSettingsForConstructedDoubleE\
    ncryptionPermutation3) do

    if
    (AdvanceRotorSettings(AdvanceRotorSettings(
    CurrentCandidate, Alphabet), Alphabet)
    =
    ListOfPossibleRotorSettingsForConstructedDoubleE\
    ncryptionPermutation3[k]) then

    ListOfPossibleRotorSettingsForAllConstructedDoubl\
    eEncryptionPermutations :=
    [op(
    ListOfPossibleRotorSettingsForAllConstructedDoubl\
    eEncryptionPermutations), CurrentCandidate];
   end if;
  end do;
  end if;
 end do;
end do:
```

The program filters these results with FOR loops and generates a list of rotor settings that could have formed not only characteristic AD, but BE and CF as well. For example: if we had a rotor setting of $[12, 19, 7]$ in AD, then there would be a rotor setting of $[12, 19, 8]$ in BE and a rotor setting of $[12, 19, 9]$ in CF. If this was the case, then we would add $[12, 19, 7]$ to the *ListOfPossibleRotorSettingsForAllConstructedDoubleEncryptionPermutations*.

```
> print(
      ListOfPossibleRotorSettingsForAllConstructedDoubleE\
      ncryptionPermutations);
    [[12, 19, 7], [19, 2, 22], [24, 3, 4], [24, 21, 1]]        (5.4.2)
> nops(
      ListOfPossibleRotorSettingsForAllConstructedDoubleE\
      ncryptionPermutations);
                          4                                   (5.4.3)
```

In the end, we find four possible rotor settings which could have resulted in all three characteristics. These settings are $[12, 19, 7]$, $[19, 2, 22]$, $[24, 3, 4]$, and $[24, 21, 1]$ which are the $[L, S, G]$, $[S, B, V]$, $[X, C, D]$, and $[X, U, A]$ settings.

4.4.4 Solving the Plugboard

4.4.4.1 Derive Plugboard by Simple Comparison

Now that we have a list of possible rotor settings that could have generated all three characteristics of this message, we can attempt to solve the plugboard by generating the characteristic permutations without a plugboard and making comparisons between the two. For each possible rotor setting, we can

create a characteristic that is encrypted without any plugboard connections. By making comparisons with the cycles of the same length in both the encrypted and decrypted characteristics, we can derive a possible plugboard by assigning the letters together where they differ.

Derive Plugboard by Simple Comparison

```
> with(ListTools, MakeUnique) :
> ListOfPossibleDailyKeyRotorSettingsWithFoundOperato\
      rAndPlugboardSettings := [ ];
ListOfPossibleDailyKeyRotorSettingsWithFoundOpera\                    (5.5.1)
      torAndPlugboardSettings := [ ]
> ListOfDecryptedOperatorSettings := [ ];
      ListOfDecryptedOperatorSettings := [ ]                            (5.5.2)
> ListOfDerivedPlugboardSettings := [ ];
      ListOfDerivedPlugboardSettings := [ ]                             (5.5.3)
```

Here we start off by initalizing *ListOfPossibleDailyKeyRotorSettingsWith-FoundOperatorAndPlugboardSettings*, *ListOfDecryptedOperatorSettings*, and *ListOfDerivedPlugboardSettings*. These lists will store our final results when we find viable solutions.

```
> for i from 1
    to
    nops(
    ListOfPossibleRotorSettingsForAllConstructedDoubl\
    eEncryptionPermutations) do
  CurrentRotorOffsets :=
    ListOfPossibleRotorSettingsForAllConstructedDoubl\
    eEncryptionPermutations[ i];
  DecryptedOperatorSetting := [ ];

  ConstructedDoubleEncryptionPermutation1WithoutPlu\
    gboard :=
    mulperms(
    GenerateEnigmaEncryptionPermutationWithPlugbo\
    ard( ListOfSelectedRotorsLeftToRight,
     AdvanceRotorSettings( CurrentRotorOffsets,
     Alphabet), Reflector,
     ListOfPermutationsUsedToSimulateRotorRotation\
    sAroundAxle, [ ]),
    GenerateEnigmaEncryptionPermutationWithPlugbo\
    ard( ListOfSelectedRotorsLeftToRight,
     AdvanceRotorSettings( AdvanceRotorSettings(
    AdvanceRotorSettings( AdvanceRotorSettings(
    CurrentRotorOffsets, Alphabet), Alphabet),
     Alphabet), Alphabet), Reflector,
     ListOfPermutationsUsedToSimulateRotorRotation\
    sAroundAxle, [ ]) );

    DerivedPlugboardForConstructedDoubleEncryption\
    Permutation1 := { };
```

For the outermost FOR loop, we iterate through all the possible rotor settings found via the catalog. We then construct a version of *AD* without the influence of the plugboard using the current rotor setting from the catalog. *DecryptedOperatorSetting* and *DerivedPlygboardForConstructedDoubleEncryp-*

tionPermuation1 are also initialized in this step.

```
for j from 1
    to
    nops(
    ConstructedDoubleEncryptionPermutation1WithPlu\
    gboard) do
 for k from 1
    to
    nops(
    ConstructedDoubleEncryptionPermutation1WithPlu\
    gboard[ j ] ) do

    MappedOffsetInConstructedDoubleEncryptionPerm\
    utation1WithPlugboard :=
    ConstructedDoubleEncryptionPermutation1WithPlu\
    gboard[ j ] [ `mod` ( k,
     nops(
    ConstructedDoubleEncryptionPermutation1WithPlu\
    gboard[ j ] ) ) + 1];

   for l from 1
    to
    nops(
    ConstructedDoubleEncryptionPermutation1Without\
    Plugboard) do
   for m from 1
    to
    nops(
    ConstructedDoubleEncryptionPermutation1Without\
    Plugboard[ l ] ) do
```

For our second FOR loop, we iterate through the cycles of the characterist *AD* with the plugboard given as input to the program through the *Test05Decrypt02.mpl* file. For the third FOR loop, we iterate through all the elements within the cycles of characterist *AD*. From here, we see what

letter is mapped to the current one by checking the next one in sequence since the permutations are written in cycle notation. The fourth and fifth FOR loops will iterate through the cycles and elements of the version of AD without the influence of the plugboard.

```
if
ConstructedDoubleEncryptionPermutation1WithPlu\
gboard[j][k]
=
ConstructedDoubleEncryptionPermutation1Without\
Plugboard[l][m] then

MappedOffsetInConstructedDoubleEncryptionPerm\
utation1WithoutPlugboard :=
ConstructedDoubleEncryptionPermutation1Without\
Plugboard[l][ `mod`(m,
 nops(
ConstructedDoubleEncryptionPermutation1Without\
Plugboard[l])) + 1];
```

If we find the current element from characteristic AD inside the the version of AD without the influence of the plugboard, then we see what letter is mapped to the current element of AD without plugboard.

```
if
(
MappedOffsetInConstructedDoubleEncryptionPerm\
utation1WithPlugboard
≠ MappedOffsetInConstructedDoubleEncryptionPer\
mutation1WithoutPlugboard)
and
( nops(
ConstructedDoubleEncryptionPermutation1WithPlu\
gboard[ j ] )
=
nops(
ConstructedDoubleEncryptionPermutation1Without\
Plugboard[ l ] ) ) then

DerivedPlugboardForConstructedDoubleEncryption\
Permutation1 :=
{ op(
DerivedPlugboardForConstructedDoubleEncryption\
Permutation1 ),
sort( [
MappedOffsetInConstructedDoubleEncryptionPerm\
utation1WithPlugboard,
MappedOffsetInConstructedDoubleEncryptionPerm\
utation1WithoutPlugboard ] ) };
 end if;
end if;
end do;
end do;
end do;
end do;
```

Finally, if the mapped letter from the current element in characteristic element AD and mapped letter from the current element in the version of AD without the plugboard are not the same letter and the length of the cycles from both are the same, then we map these letters to each other in

the *DerivedPlugboardForConstructedDoubleEncryptionPermutation1* list.

For example: the AD characteristic with the plugboard is the permutation $[[1, 7, 10, 11, 25, 15, 20, 9, 12, 4, 18, 22, 2][3, 5, 17, 19, 23, 13, 24, 16, 8, 6, 21, 26, 14]]$ written in disjoint cycle notation. The AD characteristic without the plugboard is $[[1, 24, 26, 20, 14, 6, 7, 13, 12, 10, 4, 17, 22][2, 11, 21, 5, 9, 15, 25, 3, 23, 16, 18, 8, 19]]$. Starting off with 1 in each cycle we see that the next numbers are 7 and 24. These are paired together and added to *DerivedPlugboardForConstructedDoubleEncryptionPermutation1.* Continuing on we find 7 in each cycle and see that they next numbers in their cycles are 10 and 13, so we pair these together and add them as well. Eventually, once with cover both cycles of 13, we end up with following list for DerivedPlugboardForConstructedDoubleEncryptionPermutation1 when sorted in ascending order:

$$\{[1,2],[1,11],[2,23],[3,6],[3,15],[4,10],[4,11],[5,23],[5,26],[6,19], [7,21],[7,24],[8,18],[8,22],[9,14],[9,17],[10,13],[12,15],[12,24], [13,16],[14,20],[16,26],[17,18],[19,22],[20,25],[21,25]] \tag{4.1}$$

This approach can, however, end up with false positive plugboard connections. To compensate, we use a similar approach as we did with the catalog and repeat the process for the other two characteristics, to calculate *DerivedPlugboardForConstructedDoubleEncryptionPermutation2* and *Derived-*

PlugboardForConstructedDoubleEncryptionPermutation3. The resulting plugboard is made up of only the plugboard connections which are common between all of them. The code for interating through *ConstructedDoubleEncryptionPermutation2* and *ConstructedDoubleEncryptionPermutation3* are omitted because they are the exact same process as *ConstructedDoubleEncryptionPermutation1*.

```
DerivedPlugboard := 'intersect
    '(
    DerivedPlugboardForConstructedDoubleEncryption\
    Permutation1,
     DerivedPlugboardForConstructedDoubleEncryptio\
    nPermutation2,
     DerivedPlugboardForConstructedDoubleEncryptio\
    nPermutation3);

DerivedPlugboard := sort(convert(DerivedPlugboard,
    list));

if type(DerivedPlugboard, 'disjcyc') then
 for j from 1 to 6 do
  CurrentRotorOffsets :=
    AdvanceRotorSettings(CurrentRotorOffsets,
     Alphabet);
  DecryptedOperatorSetting :=
    [op(DecryptedOperatorSetting),
    ExtractCiphertextOffsetFromEnigmaEncryptionPer\
    mutation(CiphertextOffsets[j],
    GenerateEnigmaEncryptionPermutationWithPlugbo\
    ard(ListOfSelectedRotorsLeftToRight,
    CurrentRotorOffsets, Reflector,
     ListOfPermutationsUsedToSimulateRotorRotation\
    sAroundAxle, DerivedPlugboard))];
 end do:
end if;
```

Once we have all three derived plugboards for each characteristic, we take the intersection of all three as described above and then attempt to decrypt the first 6 letters of the ciphertext which are the operator setting repeated twice using the derived plugboard and the current rotor positions from the catalog.

```
    if DecryptedOperatorSetting ≠ [ ]
       and DecryptedOperatorSetting[1]
       = DecryptedOperatorSetting[4]
       and DecryptedOperatorSetting[2]
       = DecryptedOperatorSetting[5]
       and DecryptedOperatorSetting[3]
       = DecryptedOperatorSetting[6] then

       ListOfPossibleDailyKeyRotorSettingsWithFoundOpe\
       ratorAndPlugboardSettings :=
       [op(
       ListOfPossibleDailyKeyRotorSettingsWithFoundOpe\
       ratorAndPlugboardSettings),
       ListOfPossibleRotorSettingsForAllConstructedDoubl\
       eEncryptionPermutations[i]];
     ListOfDecryptedOperatorSettings :=
       [op(ListOfDecryptedOperatorSettings),
        DecryptedOperatorSetting[1..3]];
     ListOfDerivedPlugboardSettings :=
       [op(ListOfDerivedPlugboardSettings),
        DerivedPlugboard];
    end if;
   end do:
```

If the resulting decrypted text is not empty, the first letter matches the fourth, the second letter matches the fifth, and the third letter matches the sixth; then we have a candidate and its rotor setting, decrypted operator setting, and plugboard settings are saved in lists and printed out once all

possible rotor settings from the previous section have been exhausted.

```
> ListOfPossibleDailyKeyRotorSettingsWithFoundOperato\
    rAndPlugboardSettings;
                    [[24, 3, 4]]                         (5.5.4)
> ListOfDecryptedOperatorSettings;
                    [[18, 8, 3]]                         (5.5.5)
> ListOfDerivedPlugboardSettings;
             [[[2, 12], [5, 26], [16, 24]]]              (5.5.6)
```

In this example: After comparing the permutations of AD, BE, and CF with and without the plugboard and noting where they differ, we can derive a valid plugboard and operator settings for one of the possible rotor settings found with the catalog. Rotor setting $[24, 3, 4]$ has valid plugboard settings$[[2, 12], [5, 26], [16, 24]]$ and a decrypted operator setting of $[18, 8, 3]$. This is the only candidate found.

4.4.4.2 Derive Plugboard Recursively for All Rotor Settings

If there was a viable candidate found via the catalog method described in the previous Subsection 4.4.3, then the ciphertext of the message is decrypted and the decrypted message plaintext, the daily key, the operator key, and the plugboard settings are printed out to the user.

Derive Plugboard Recursively for All Rotor Settings

```
> if
    ListOfPossibleDailyKeyRotorSettingsWithFoundOpe\
    ratorAndPlugboardSettings ≠ [ ]
    and ListOfDecryptedOperatorSettings ≠ [ ]
    and ListOfDerivedPlugboardSettings ≠ [ ] then
  for i from 1
    to
    nops(
    ListOfPossibleDailyKeyRotorSettingsWithFoundOpe\
    ratorAndPlugboardSettings) do
  CurrentRotorOffsets :=
    ListOfDecryptedOperatorSettings[ i];
  DerivedCiphertext := "";

  for j from 7 to nops( CiphertextOffsets) do
   CurrentRotorOffsets :=
    AdvanceRotorSettings( CurrentRotorOffsets,
     Alphabet);
   DerivedCiphertext := cat( DerivedCiphertext,
     Alphabet[
    ExtractCiphertextOffsetFromEnigmaEncryptionPer\
    mutation( CiphertextOffsets[j],
    GenerateEnigmaEncryptionPermutationWithPlugbo\
    ard(ListOfSelectedRotorsLeftToRight,
    CurrentRotorOffsets, Reflector,
     ListOfPermutationsUsedToSimulateRotorRotation\
    sAroundAxle, ListOfDerivedPlugboardSettings[i]))
     ]);
  end do;
```

This section of code checks for a solution from the catalog and simple plugboard derivation steps. If there are some, then it will iterate throught the possible solutions and decrypts the remaining ciphertext.

```
print(DerivedCiphertext);
print("Daily Key: ",
    ListOfPossibleDailyKeyRotorSettingsWithFoundOpe\
   ratorAndPlugboardSettings[i]);
 print("Operator Key: ",
    ListOfDecryptedOperatorSettings[i]);
print("Plugboard Settings: ",
    ListOfDerivedPlugboardSettings[i]);
print( );

end do:
```

Here, we print the decypted ciphertext along with Daily Key, Operator Key, and Plugboard Settings used so it can be checked by the end user to see if the output makes sense.

```
else
 InitialRotorSettings := [1, 1, 1];

 for i from 1 to nops(Alphabet) · nops(Alphabet)
     · nops(Alphabet) do

     ListOfPossibleSolutionsForPlugboardAtCurrentRoto\
     rOffsets := [ ] :
  CurrentRotorOffsets := InitialRotorSettings :
  EnigmaEncryptionPermutationsWithoutPlugboard :=
     [ ] :

 for j from 1 to CribOffset do
  CurrentRotorOffsets :=
     AdvanceRotorSettings( CurrentRotorOffsets,
      Alphabet) :
  end do:

  for j from 1 to nops( CribPlaintext) do
  CurrentRotorOffsets :=
     AdvanceRotorSettings( CurrentRotorOffsets,
      Alphabet) :
  EnigmaEncryptionPermutationsWithoutPlugboard :=
     [op(
     EnigmaEncryptionPermutationsWithoutPlugboard)
     ,
     GenerateEnigmaEncryptionPermutationWithPlugbo\
     ard(ListOfSelectedRotorsLeftToRight,
      CurrentRotorOffsets, Reflector,
      ListOfPermutationsUsedToSimulateRotorRotation\
     sAroundAxle, [ ]) ] :
  end do:
```

If no viable candidate was found, however, then the program proceeds to attempt to recursively derive the plugboard for all rotor settings. The initial rotor settings are set to $[1, 1, 1]$. The program then iterates through all

possible rotor settings and calculates a list of the encryption permutations without the influence of the plugboard for each letter of the crib plaintext, which is provided as part of the user settings in an external text file containing Maple input, starting at the current rotor settings.

```
LetterAssignments := [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] :
AvailableLetters := [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] :
TextOffsetsIndex := 1 :

   ListOfPossibleSolutionsForPlugboardAtCurrentRoto\
   rOffsets :=
   DeterminePlugboardWirings( LetterAssignments,
    AvailableLetters, TextOffsetsIndex,
    ListOfPossibleSolutionsForPlugboardAtCurrentRoto\
   rOffsets,
    EnigmaEncryptionPermutationsWithoutPlugboard,
    CribPlaintextOffsets, CribCiphertextOffsets) :

   if
   ListOfPossibleSolutionsForPlugboardAtCurrentRoto\
   rOffsets ≠ [ ] then
 print( InitialRotorSettings);

   print(
   ListOfPossibleSolutionsForPlugboardAtCurrentRoto\
   rOffsets);
 print( );
end if;
InitialRotorSettings :=
   AdvanceRotorSettings( InitialRotorSettings,
    Alphabet);
end do;
end if:
```

This information is then fed into the recursive *DeterminePlugboardWirings* method contained in the *EnigmaProcedures.mpl* library. If this procedure returns with a solution, then the current rotor settings and the plugboard wirings are printed out to the user. The program then continues to iterate through all possible rotor settings until all options are exhausted.

```
"ANMARTINOBERZALEKXKEINEBESONDERENEREIGN\
    ISSEINDERBULMEHEUTE"
                "Daily Key: ", [24, 3, 4]
              "Operator Key: ", [18, 8, 3]
  "Plugboard Settings: ", [[2, 12], [5, 26], [16, 24]]
                                                        (5.6.1)
```

In the current example, however, a viable candidate was found via the catalog methods, so this approach is skipped.

4.4.4.3 Determine Plugboard Wirings

```
> DeterminePlugboardWirings := proc
  (LetterAssignments::list,
  AvailableLetters::list,
  TextOffsetsIndex::integer,
  ListOfPossibleSolutions::list,
  EnigmaEncryptionPermutationsWithoutPlugboard::lis
  t, PlaintextOffsets::list, CiphertextOffsets)
  ::list;
>         local i, CompanionLetterOffset,
  LocalListOfPossibleSolutions;
>
>         LocalListOfPossibleSolutions :=
  ListOfPossibleSolutions;
>
>         if TextOffsetsIndex = nops
  (PlaintextOffsets)+1 then
>                 LocalListOfPossibleSolutions :=
  [op(LocalListOfPossibleSolutions), [convert
  (LetterAssignments, 'disjcyc'),
  AvailableLetters]];
>
>                 return
  LocalListOfPossibleSolutions;
>         end if;
>
```

DeterminePlugboardWirings starts off its iteration by checking if the end of the crib has been reached. If it has, then save the plugboard connections as one of the possible solutions and return.

To review, in the physical machine, each letter of the alphabet used has a corresponding socket in the plugboard that can be assigned to itself or to another letter. If it is assigned to another letter, then a wire with a male plug on either end is inserted into each corresponding socket to switch the

electrical connections of these two letters. If it is assigned to itself, then no cable is inserted into its socket. Here, we differentiate between letters which are considered to not contain any plugs versus letters we have not tried any assignments with yet. Therefore, just because a plugboard mapping does not contain any plugs, does not mean that this letter is unassigned. This is why we use a list to keep track of the available letters as well as one to store any letter assignments made.

```
>         if AvailableLetters[PlaintextOffsets
  [TextOffsetsIndex]] = 1 then
>                 if AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 then
>                         if
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]) = CiphertextOffsets
  [TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
  CiphertextOffsets[TextOffsetsIndex] =
  CiphertextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, CiphertextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                         end if;
```

The next section of the method checks several conditions when the current plaintext letter of the crib is unassigned a plugboard connection. Then, it checks within these cases for ones where the current matching ciphertext letter of the crib is unassigned as well. The first case it checks within these conditions is if the current plaintext/ciphertext letter pairs of the crib are unassigned and the encrypted plaintext letter is the same as the ciphertext letter. If it is, then the plaintext letter of the alphabet is assigned to itself and the corresponding ciphertext letter is assigned to itself as well.

```
>                       CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>
>                       if AvailableLetters
  [CompanionLetterOffset] = 0 and
  CompanionLetterOffset <> CiphertextOffsets
  [TextOffsetsIndex] and CompanionLetterOffset =
  LetterAssignments[CiphertextOffsets
  [TextOffsetsIndex]] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                       end if;
>               end if;
```

The next case checks to see if the corresponding letter of the encrypted plaintext letter, which is not the same as the ciphertext letter, are both unassigned. If they are, then try assigning the encrypted plaintext letter and the ciphertext letter together while assigning the plaintext letter to itself.

```
>                 else
>                         if
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]) = CiphertextOffsets
  [TextOffsetsIndex] and LetterAssignments
  [CiphertextOffsets[TextOffsetsIndex]] =
  CiphertextOffsets[TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                         end if;
```

The next series of checks are ones where the current plaintext letter of the crib is unassigned and the corresponding ciphertext letter of the crib has been assigned. The first case is one where the corresponding encrypted plaintext letter of the crib is the same letter as the ciphertext letter and the ciphertext letter is assigned to itself. In this case, it will assign the plaintext letter to itself and call itself to recurse.

```
>                       CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>
>                       if AvailableLetters
  [CompanionLetterOffset] = 0 and
  CompanionLetterOffset <> CiphertextOffsets
  [TextOffsetsIndex] and CompanionLetterOffset =
  LetterAssignments[CiphertextOffsets
  [TextOffsetsIndex]] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                       end if;
>               end if;
```

The next case is if the encrypted plaintext letter is assigned to the ciphertext letter, then the plaintext letter should be assigned to itself and the method recurses.

```
>         else CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(LetterAssignments[PlaintextOffsets
  [TextOffsetsIndex]],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>                 if AvailableLetters
  [CompanionLetterOffset] = 1 and AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 and
  CompanionLetterOffset <> CiphertextOffsets
  [TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (CompanionLetterOffset = CiphertextOffsets
  [TextOffsetsIndex], CiphertextOffsets
  [TextOffsetsIndex] = CompanionLetterOffset,
  LetterAssignments), subsop(CompanionLetterOffset
  = 0, CiphertextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                 end if;
```

The next series of cases are where the current plaintext letter has already received an assignment. The first case is if the corresponding plaintext letter and ciphertext letter are unassigned and the encrypted plaintext letter is not the same letter as the ciphertext letter. In this case, the encrypted plaintext letter and the ciphertext letter should be assigned to each other.

```
>                if
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(LetterAssignments[PlaintextOffsets
  [TextOffsetsIndex]],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]) = LetterAssignments
  [CiphertextOffsets[TextOffsetsIndex]] then
>                       if AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 0 then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(LetterAssignments,
  AvailableLetters, TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
```

The next case is to encrypt the letter that the plaintext letter is assigned to and if this letter is the same as the letter that is assigned to the ciphertext letter, then provided that the ciphertext letter is assigned, then recurse on the next plaintext letter of the crib.

```
>                       else
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (CiphertextOffsets[TextOffsetsIndex] =
  LetterAssignments[CiphertextOffsets
  [TextOffsetsIndex]], LetterAssignments), subsop
  (CiphertextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                       end if;
```

In the event that the ciphertext letter is unassigned in the above case, then officially assign the ciphertext letter its currently mapped letter.

```
>                 else
>                        return
  LocalListOfPossibleSolutions;
>                 end if;
>         end if;
```

Finally, in the case that the letter assigned to the current plaintext letter is encrypted and the resulting letter is not the same letter as letter that is mapped to the ciphertext letter, then the methods should return since we now have a contradiction in our plugboard assignments.

```
>         for i to nops(LetterAssignments) do
>                if AvailableLetters
  [PlaintextOffsets[TextOffsetsIndex]] = 1 and
  AvailableLetters[i] = 1 and i <>
  PlaintextOffsets[TextOffsetsIndex] then
```

In the last section of the recursive method, we iterate over all the letter assignments made. If the current plaintext crib letter and the current letter in the letter assignments list are unassigned and these letters are not the same, then we will check for several cases.

```
>                         CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(i,
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>                         if AvailableLetters
  [CompanionLetterOffset] = 1 and AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 and
  CompanionLetterOffset <> PlaintextOffsets
  [TextOffsetsIndex] and CompanionLetterOffset <>
  CiphertextOffsets[TextOffsetsIndex] and i <>
  CiphertextOffsets[TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] = i, i =
  PlaintextOffsets[TextOffsetsIndex],
  CompanionLetterOffset = CiphertextOffsets
  [TextOffsetsIndex], CiphertextOffsets
  [TextOffsetsIndex] = CompanionLetterOffset,
  LetterAssignments), subsop(i = 0,
  PlaintextOffsets[TextOffsetsIndex] = 0,
  CompanionLetterOffset = 0, CiphertextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
```

1. For the first case, we will assign the current letter in the letter assignments list and the current crib plaintext letter to each other. We will also assign the currently mapped encrypted letter and the current crib ciphertext letter to each other as well. When we refer to the current mapped encrypted letter, we are referring to the letter that will result

when the current letter in the letter assignment list is encrypted without the plugboard at the current rotor settings. This case will execute when all of the follow are true:

- The current mapped encrypted letter is unassigned.
- The current crib ciphertext letter is unassigned.
- The current encrypted mapped letter and the current crib plaintext letter are not the same.
- The current encrypted mapped letter and the current crib ciphertext letter are not the same.
- The current letter in the letter assignments list is not the same as the current crib ciphertext letter.

```
>                             elif AvailableLetters
  [CompanionLetterOffset] = 1 and AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 and
  CompanionLetterOffset <> PlaintextOffsets
  [TextOffsetsIndex] and CompanionLetterOffset =
  CiphertextOffsets[TextOffsetsIndex] and i <>
  CiphertextOffsets[TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] = i, i =
  PlaintextOffsets[TextOffsetsIndex],
  CiphertextOffsets[TextOffsetsIndex] =
  CiphertextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(i = 0,
  PlaintextOffsets[TextOffsetsIndex] = 0,
  CiphertextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                             elif AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 0 and
  CompanionLetterOffset = LetterAssignments
  [CiphertextOffsets[TextOffsetsIndex]] and i <>
  CiphertextOffsets[TextOffsetsIndex] then
```

2. For the second case, we will assign the current letter in the letter assignments list and the current crib plaintext letter to each other. We will also assign the current crib ciphertext letter to itself. This case will execute when all of the following are true:

- The encrypted current mapped letter is unassigned.
- The current crib ciphertext letter is unassigned.
- The current encrypted mapped letter and the current crib plaintext letter are not the same.
- The current encrypted mapped letter and the current crib ciphertext letter are the same.
- The current letter in the letter assignments list is not the same as the current crib ciphertext letter.

```
>                         elif AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 0 and
  CompanionLetterOffset = LetterAssignments
  [CiphertextOffsets[TextOffsetsIndex]] and i <>
  CiphertextOffsets[TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] = i, i =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(i = 0,
  PlaintextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
```

3. For the third case, we will assign the current letter in the letter assignment list and the current crib plaintext letter to themselves. This case

will execute when all of the following are true:

- The ciphertext letter is assigned.
- The current encrypted mapped letter is equal to the letter assigned to the current crib ciphertext letter.
- The current letter in the letter assignments list is not the same as the current crib ciphertext letter

```
                        elif PlaintextOffsets
[TextOffsetsIndex] = CompanionLetterOffset and
CiphertextOffsets[TextOffsetsIndex] = i then

LocalListOfPossibleSolutions :=
DeterminePlugboardWirings(subsop
(PlaintextOffsets[TextOffsetsIndex] = i, i =
PlaintextOffsets[TextOffsetsIndex],
LetterAssignments), subsop(i = 0,
PlaintextOffsets[TextOffsetsIndex] = 0,
AvailableLetters), TextOffsetsIndex+1,
LocalListOfPossibleSolutions,
EnigmaEncryptionPermutationsWithoutPlugboard,
PlaintextOffsets, CiphertextOffsets);
                        end if;
                end if;
        end do;
```

4. For the final case, we assign the current letter in the letter assignment

list and the current plaintext crib letter to themselves. This case will execute when all of the following are true:

- The current encrypted mapped letter is the same as the current crib plaintext letter.
- The current letter in the letter assignment list is the same as the current crib ciphertext letter.

```
>         return LocalListOfPossibleSolutions;
> end proc;
```

At the end of the procedure, we return from recursion.

To illustrate the use of this method, another example will be provided since the previous example made use of the catalog method:

```
> Alphabet := ["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"];
>
> PermutationToSimulateSingleRotationAroundAxle :=
  [[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]]
  ;
>
> ListOfRotorWirings := [[[1, 5, 12, 20, 16, 8,
  17, 24, 18, 21], [2, 11, 14, 23], [3, 13, 15,
  25], [4, 6, 7], [9, 22], [10, 26]], [[2, 10],
  [3, 4, 11, 12, 8, 21, 16], [5, 19, 26], [6, 9,
  24, 22, 25, 15, 13, 23], [7, 18], [14, 20]], [
  [1, 2, 4, 8, 16, 5, 10, 20], [3, 6, 12, 22, 13,
  26, 15, 25, 17, 9, 18, 23, 21, 11, 24, 19, 7]],
  [[1, 5, 16, 12, 9, 25, 23, 3, 15, 24, 13, 18, 6,
  26, 2, 19, 20, 7, 10, 17, 14, 8], [4, 22], [11,
  21]], [[1, 22, 15, 12, 4, 18, 23, 6, 9, 21, 17],
  [2, 26, 11, 19, 13, 14, 8, 25, 3], [5, 7, 20,
  10, 16, 24]]];
>
> ListOfSelectedRotorsLeftToRight :=
  [ListOfRotorWirings[3], ListOfRotorWirings[1],
  ListOfRotorWirings[4]];
>
> Reflector := [[1, 25], [2, 18], [3, 21], [4, 8],
  [5, 17], [6, 19], [7, 12], [9, 16], [10, 24],
  [11, 14], [13, 15], [20, 26], [22, 23]];
```

Here we have the encryption settings for our new example. These include the same *Alphabet*, *PermutationToSimulateSingleRotationAroundAxle*, *ListOfRotorWirings*, and *Reflector* as the previous example. This example does differ with the *ListOfSelectedRotorsLeftToRight*, *Plugboard*, *DailySetting*, *OperatorSetting*, and *Plaintext*.

```
> Ciphertext :=
> "SZBPOQGEMFQRETHFDYFZZGXMAROHTOYLSUSEVTLUCM";
> CribCiphertext := "HFDYFZZGXMAROH";
> CribPlaintext := "BROWNFOXJUMPED";
> CribOffset := 8;
> ConstructedDoubleEncryptionPermutation1WithPlugbo
  ard := [[1, 3, 8, 9, 6, 24, 5,
> 18], [2, 12, 22, 7], [4, 15, 21, 20, 17, 14, 13,
  10], [11, 19, 16, 26]];
> ConstructedDoubleEncryptionPermutation2WithPlugbo
  ard := [[1, 25, 26, 15, 22, 24
> , 6, 21, 4, 7, 14, 2, 19], [3, 12, 18, 17, 8, 5,
  16, 23, 11, 9, 10, 20, 13]];
> ConstructedDoubleEncryptionPermutation3WithPlugbo
  ard := [[1, 23, 11, 6, 20, 4,
> 26, 8, 13, 5, 10], [2, 17], [3, 14, 25, 7, 18,
  12, 15, 16, 24, 9, 21], [19, 22]
> ];
```

Here are the primary decryption settings outside of the machine settings for the new example:

- The encrypted ciphertext from the output of the encryption program. This includes the double enciphered operator key.
- The crib ciphertext
- The corresponding crib plaintext
- The crib offset, which indication where the crib ciphertext starts in relation to the full ciphertext. It is worth noting that this offset ignores the first 6 characters which are the double enciphered operator key.

- The three characteristic permutations of AD, BE, and CF.

We skip most of the steps for the new example because they will almost be exactly the same as the decryption example used above. The primary difference arises when no solution is found via the catalog and simple plugboard derivation method. This will then branch off to the recursive DeterminePlugboardWirings method. The output of this method in this case is shown below.

```
print(
ListOfPossibleSolutionsForPlugboardAtCurrentRoto\
rOffsets);
print( );
end if;
InitialRotorSettings :=
AdvanceRotorSettings( InitialRotorSettings,
Alphabet);
end do;
end if:
                    [7, 12, 19]
[[[[2, 22], [3, 4], [6, 19], [7, 11]], [0, 0, 0, 0, 0, 0,
   0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0]],
   [[[1, 17], [2, 22], [3, 4], [6, 19], [7, 11], [13,
   21]], [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,
   0, 1, 0, 0, 0, 0, 0, 0]], [[[2, 22], [3, 4], [6, 19],
   [7, 23], [10, 20], [11, 25], [17, 24]], [0, 0, 0, 0,
   0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0]], [[[2, 22], [3, 4], [6, 19], [7, 25], [10, 24],
   [11, 23]], [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0,
   1, 0, 0, 1, 0, 0, 0, 0, 0, 0]], [[[1, 17], [2, 22], [3,
   4], [6, 19], [7, 25], [10, 24], [11, 23], [13, 21]],
   [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
   0, 0, 0, 0, 0, 0]], [[[2, 22], [3, 4], [6, 19], [7,
   11], [23, 25]], [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0,
   0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0]], [[[1, 17], [2, 22],
   [3, 4], [6, 19], [7, 11], [13, 21], [23, 25]], [0, 0,
   0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0]]]
                                                   (5.6.1)
```

For the output of the method, we only find possible plugboard solutions at the rotor setting $[7, 12, 19]$. Several possibilities are found, due to the crib used

since it may not have enough contradictions within it to narrow the search down to a single possibility. We do see, however, that the correct solution is actually the first possibility presented. The output seen here are two lists that are returned for each plugboard possibility. The first list is of the pairs of plugboard wirings. The second list is the AvailableLetters list of the same length as the alphabet list. There is a position in this list for each letter of the alphabet. Possible values are either zero or one at each position: zero means that the letter at this position is either assigned to another letter or to itself, and one means that this letter is available for assignment. The use of this list is to keep track of what letters are available as the recursive method proceeds. The exit values of both the plugboard and AvailableLetters list are what are appended in the *ListOfPossibleSolutionsForPlugboardAtCurrentRotorOffsets* list, which is the output seen.

Chapter 5

Conclusion

For this report, two Maple programs have been created to implement and demonstrate the Enigma cipher: An encryption program and a decryption program. The decryption program utilizes the card catalog to determine the possible rotor positions. The program then attempts to solve the plugboard wirings by comparing the unencrypted characteristics with their encrypted forms. Where these two forms differ, plugboard substitutions are made to resolve them. This is done in an attempt to find simpler substitution solutions before attempting a brute force attempt to solve the plugboard utilizing a crib and heavy recursion.

The recursive method is of interest because it can handle all possible plugboard settings using any number of leads up to the full 13. This is of note because it had been procedure to utilize exactly 7 leads until January 1st,

1939 when this was changed to exactly 10 leads [24]. Due to this policy, most Enigma solving methods make this assumption. Due to the fact that extensibility of the program was a design goal, the use of exactly 10 leads was not appropriate for the cryptanalysis and decryption methods used.

The advantage of the use of Maple worksheets compared to other methods, like CrypTool 2's implementation of Gillogly's ciphertext-only attack [23], is that it is much simpler to follow and step through the program utilizing Maple's internal debugger. Most implementations have a tendency to be more of a black box approach, especially CrypTool 2's visual programming workflow where one is not always given a meaningful understanding of what is happening inside the blocks besides inputs and outputs. The ability to freely modify Maple worksheets can also allow further understanding and comprehension of the concepts involved through experimentation, which is also something one cannot accomplish through more black box approaches.

Student comprehension of the concepts involved is the primary goal of this work. The intention is to demonstrate an actual encryption scheme and the methods in which its flaws can be exploited in order to decrypt it without the key. In the case of Enigma, and the work done for this report, this flaw is the double encipherment of the message key which was done earlier in the war. These weaknesses are highlighted in order to introduce students to the mathematical concepts involved through a representative example which is still simple enough that student can trace through the steps by hand. This is something that is difficult to do for most modern cryptosystems, which

was why Enigma was selected in order to establish a foundation that can ultimately lead to these modern cryptosystems.

This program can be both incorporated into a class and worked with on an individual level with a student exploring and experimenting with the software. The worksheets can be covered in a group setting and the instructor can cover how the worksheets work, the output of different examples for both the encryption and decryption program, as well as possible introductions on how these worksheets can be modified through altering the methods in the EngimaProcedures.mpl file. Test cases and modifications can potentially be covered through an directed exploration of the software through an assignment or a tutorial. Since the worksheets are fairly well documented and clear to read, a student could also explore, expand, and experiment with the software on their own as well. It was the goal of these worksheets that they can accommodate as many different learning styles as possible.

The potential for future work and expansion involves implementing several variables that were ignored in order to focus on the core combinatorial problem. These variables include the ring settings, the rotor selection and order, as well as rotor double stepping. Now that the core of the program has been solidified, the implementation of these variables would allow the program to be a full implementation of Enigma. From there, there is the potential to use this as a foundation in order to implement various Enigma variants that were used within different divisions of the war as well as other rotor cipher systems that followed that utilized the same principles. Potential examples

of the latter include Hagelin B-21, SIGABA, Typex, NEMA, KL-7, Fialka, and Tatjana van Vark's Enigma Inspired Rotor Machine [12, 39]. An exploration of the cryptanalysis methods used for these derivatives would allow a expansion of the mathematical concepts covered in order to lay further knowledge and foundations as one progresses to modern systems.

References

[1] N. Abdullah, R. Mutalip, and K. Abdullah, "The enhancement of existing DES Maplet interface," in *21st National Symposium on Mathematical Sciences (SKSM21): Germination of Mathematical Sciences Education and Research towards Global Sustainability, 6-8 Nov. 2013*, ser. AIP Conf. Proc. (USA), vol. 1605. USA: American Institute of Physics, 2014, pp. 769–74.

[2] S. Adamovic, M. Sarac, D. Stamenkovic, and D. Radovanovic, "The importance of the using software tools for learning modern cryptography," *International Journal of Engineering Education*, vol. 34, no. 1, pp. 256–262, 2018.

[3] S. Adamovic, M. Sarac, M. Veinovic, M. Milosavljevic, and A. Jevremovic, "An Interactive and Collaborative Approach to Teaching Cryptology," *Educational Technology & Society*, vol. 17, no. 1, pp. 197–205, Jan. 2014.

[4] F. L. Bauer, "An Error in the History of Rotor Encryption Devices," *Cryptologia*, vol. 23, no. 3, pp. 206–210, Jul. 1999.

[5] M. S. U. Bhuiyan and M. L. Ali, "Crypto-master: A Computer Aided Learning Tool for Cryptography," in *2018 International Conference on Innovation in Engineering and Technology (ICIET)*. Dhaka, Bangladesh: IEEE, Dec. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8660899/

[6] G. Bloch and R. Erskine, "Enigma: The Dropping of the Double Encipherment," *Cryptologia*, vol. 10, no. 3, pp. 134–141, Jul. 1986. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/0161-118691860958

[7] A. N. Borodzhieva and P. K. Manoilov, "MATLAB-based module for encryption and decryption using bifid ciphers applied in cryptosystems," in *2014 IEEE 20th International Symposium for Design and Technology in Electronic Packaging (SIITME)*. Bucharest, Romania: IEEE, Oct. 2014, pp. 287–291. [Online]. Available: http://ieeexplore.ieee.org/document/6967046/

[8] R. A. Brualdi, *Introductory Combinatorics*. Hoboken, New Jersey: Pearson Prentice Hall, 2004.

[9] O.-S. Chok and S. Herath, "Computer Security Learning Laboratory: Implementation of DES and AES Algorithms using Spreadsheets," in

Midwest Instruction and Computing Symposium, University of Minnesota Morris, Morris, Minnesota, Apr. 2004.

[10] Crypto Museum Foundation, "Enigma I," Dec. 2020. [Online]. Available: https://cryptomuseum.com/crypto/enigma/i/index.htm

[11] ——, "Bombe," Mar. 2021. [Online]. Available: https://cryptomuseum.com/crypto/bombe/index.htm

[12] ——, "Enigma Cipher Machine," May 2021. [Online]. Available: https://cryptomuseum.com/crypto/enigma/index.htm

[13] ——, "Enigma family tree," May 2021. [Online]. Available: https://cryptomuseum.com/crypto/enigma/tree.htm

[14] A. A. H. El Farra and E. Zahedi, "Interactive educational tool for teaching a simple cipher," in *2014 International Symposium on Biometrics and Security Technologies (ISBAST)*. Kuala Lumpur, Malaysia: IEEE, Aug. 2014, pp. 102–105. [Online]. Available: http://ieeexplore.ieee.org/document/7013102/

[15] G. Ellsbury, "Complexity of the Enigma," 1998. [Online]. Available: http://www.ellsbury.com/enigma4.htm

[16] C. M. Foundation, "Enigma M3," Sep. 2020. [Online]. Available: https://cryptomuseum.com/crypto/enigma/m3/index.htm

[17] ——, "UKW-D," May 2021. [Online]. Available: https://cryptomuseum.com/crypto/enigma/ukwd/index.htm

[18] J. A. Gallian, *Contemporary Abstract Algebra*. Boston, MA: Houghton Mifflin Company, 2006.

[19] J. Gillogly, "Ciphertext-only cryptanalysis of Enigma," *Cryptologia*, vol. 19, no. 4, pp. 405–13, Oct. 1995, place: USA.

[20] T. Goulding, "A first semester freshman project: the enigma encryption system in C," *ACM Inroads*, vol. 4, no. 1, p. 43, Mar. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2432596.2432613

[21] F. H. Hinsley and A. Stripp, *Codebreakers: The inside story of Bletchley Park*. New York: Oxford University Press, 1993.

[22] J. F. Humphreys, *A Course in Group Theory*. New York: Oxford Science Publications, 1996.

[23] N. Kopal, "CrypTool 2 Release 2021.1 published," May 2021. [Online]. Available: https://www.cryptool.org/en/posts/2021-05-12/cryptool-2-release-2021.1-published

[24] W. Kozaczuk, *Enigma: How the German Machine Cipher Was Broken, and How It Was Read by the Allies in World War Two*, ser. Foreign Intelligence Books Series, C. Kasparek, Ed. University Publications of America, Inc., 1984.

[25] B. Lord, "Deutsch: Das Steckerbrett einer Enigma (im Bild ist A mit J und S mit O gesteckert)." [Online]. Available: https://commons.wikimedia.org/wiki/File:Enigma-plugboard.jpg

[26] N. Luburic, M. Stojkov, G. Savic, G. Sladic, and B. Milosavljevic, "Crypto-tutor: An educational tool for learning modern cryptography," in *2016 IEEE 14th International Symposium on Intelligent Systems and Informatics (SISY)*. Subotica, Serbia: IEEE, Aug. 2016, pp. 205–210. [Online]. Available: http://ieeexplore.ieee.org/document/7601498/

[27] J. Ma, J. Tao, J. Mayo, C.-K. Shene, M. Keranen, and C. Wang, "AESvisual: A Visualization Tool for the AES Cipher," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '16. New York, NY, USA: Association for Computing Machinery, Jul. 2016, pp. 230–235. [Online]. Available: https://doi.org/10.1145/2899415.2899425

[28] R. Marwati and K. Yulianti, "Cryptanalysis on classical cipher based on Indonesian language," *Journal of Physics: Conference Series*, vol. 1013, May 2018. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/1013/1/012147/pdf

[29] M. May S. J., "Using Maple Worksheets to Enable Student Explorations of Cryptography," *Cryptologia*, vol. 33, no. 2, pp. 151–157, Apr. 2009. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01611190802323798

[30] T. Perera, "EnigmaReflector.jpg," Feb. 2007. [Online]. Available: https://commons.wikimedia.org/wiki/File:EnigmaReflector.jpg

[31] T. Sale and A. Hodges, "Counting the Possible Plugboard Settings." [Online]. Available: https://www.codesandciphers.org.uk/enigma/steckercount.htm

[32] N. P. Schembari, "A Half-Rotor Cipher for the Classroom," *PRIMUS*, Jun. 2019. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/10511970.2019.1619003

[33] J. Tao, J. Ma, M. Keranen, J. Mayo, and C.-K. Shene, "ECvisual: a visualization tool for elliptic curve based ciphers," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education - SIGCSE '12*. Raleigh, North Carolina, USA: ACM Press, 2012, p. 571. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2157136.2157298

[34] J. Tao, J. Ma, M. Keranen, J. Mayo, C.-K. Shene, and C. Wang, "RSAvisual: a visualization tool for the RSA cipher," in *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*. Atlanta, Georgia, USA: ACM Press, 2014, pp. 635–640. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2538862.2538891

[35] J. Tao, J. Ma, J. Mayo, C.-K. Shene, and M. Keranen, “DESvisual: a visualization tool for the DES cipher,” *Journal of Computing Sciences in Colleges*, vol. 27, no. 1, pp. 81–89, Oct. 2011.

[36] A. Taveneaux, “English: Enigma machine (Science Museum, London),” Jul. 2012. [Online]. Available: https://commons.wikimedia.org/wiki/File:Enigma_machine2.jpg

[37] ——, “English: Enigma machine (Science Museum, London),” Jul. 2012. [Online]. Available: https://commons.wikimedia.org/wiki/File:Enigma_machine5.jpg

[38] Ted Coles, “English: Two Enigma rotors and the spindle on which they are mounted.” Jul. 2011. [Online]. Available: https://commons.wikimedia.org/wiki/File:Enigma_rotors_and_spindle_showing_contacts_rachet_and_notch.jpg

[39] The Joe Martin Foundation for Exceptional Craftsmanship, “Scientific Instruments – Tatjana van Vark,” 2012. [Online]. Available: https://www.craftsmanshipmuseum.com/vanVark.htm

[40] J. Willard Miller, *Symmetry Groups and Their Applications*, ser. Pure and Applied Mathematics, P. A. Smith and S. Eilenberg, Eds. New York, New York: Academic Press, Inc., 1972, vol. 50.

[41] Woodbank Communications Ltd, “The Enigma Cipher Machine and Breaking the Enigma Code,” 2005. [Online]. Available: https://mpoweruk.com/enigma.htm

Appendix A

Maple Worksheets and Procedures

A.1 EnigmaProcedures.mpl

```
> with(group, invperm, mulperms);
> with(ListTools, Occurrences);
>
> CalculateAlphabetOffsets := proc
  (ListOfCharacters::list, Alphabet::list)::list;
>         local ListOfAlphabetOffsets,
  ListOfCharactersIndex, AlphabetIndex;
>
>         ListOfAlphabetOffsets := [];
>
>         for ListOfCharactersIndex to nops
  (ListOfCharacters) do
>                 for AlphabetIndex to nops
  (Alphabet) do
```

```
>                        if ListOfCharacters
  [ListOfCharactersIndex] = Alphabet
  [AlphabetIndex] then
>                          ListOfAlphabetOffsets :=
  [op(ListOfAlphabetOffsets), AlphabetIndex];
>                        end if;
>                end do;
>        end do;
>
>        return ListOfAlphabetOffsets;
>
> end proc;
>
> GenerateListOfRotorRotationPermutations := proc
  (PermutationToSimulateSingleRotationAroundAxle::l
  ist)::list;
>        local
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle, AlphabetIndex;
>
>
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle := [[]];
>
>        for AlphabetIndex from 2 to nops
  (Alphabet) do
>
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle := [op
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle), mulperms
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[AlphabetIndex-1],
  PermutationToSimulateSingleRotationAroundAxle)];
>        end do;
>
>        return
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle;
```

```
>                           if ListOfCharacters
  [ListOfCharactersIndex] = Alphabet
  [AlphabetIndex] then
>                             ListOfAlphabetOffsets :=
  [op(ListOfAlphabetOffsets), AlphabetIndex];
>                           end if;
>                  end do;
>         end do;
>
>         return ListOfAlphabetOffsets;
>
> end proc;
>
> GenerateListOfRotorRotationPermutations := proc
  (PermutationToSimulateSingleRotationAroundAxle::l
  ist)::list;
>         local
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle, AlphabetIndex;
>
>
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle := [[]];
>
>         for AlphabetIndex from 2 to nops
  (Alphabet) do
>
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle := [op
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle), mulperms
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[AlphabetIndex-1],
  PermutationToSimulateSingleRotationAroundAxle)];
>         end do;
>
>         return
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle;
```

```
> end proc;
>
> GenerateEnigmaEncryptionPermutationWithPlugboard
  := proc (ListOfRotorWirings::list,
  RotorOffsets::list, Reflector::list,
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle::list, Plugboard::list)::list;
>         local Output;
>
>         Output := mulperms([], Plugboard);
>
>         Output := mulperms(Output,
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle[RotorOffsets[3]]);
>         Output := mulperms(Output,
  ListOfRotorWirings[3]);
>         Output := mulperms(Output, invperm
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[RotorOffsets[3]]));
>
>         Output := mulperms(Output,
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle[RotorOffsets[2]]);
>         Output := mulperms(Output,
  ListOfRotorWirings[2]);
>         Output := mulperms(Output, invperm
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[RotorOffsets[2]]));
>
>         Output := mulperms(Output,
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle[RotorOffsets[1]]);
>         Output := mulperms(Output,
  ListOfRotorWirings[1]);
>         Output := mulperms(Output, invperm
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[RotorOffsets[1]]));
>
>         Output := mulperms(Output, Reflector);
```

```
>         Output := mulperms(Output, 
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle[RotorOffsets[1]]);
>         Output := mulperms(Output, invperm
  (ListOfRotorWirings[1]));
>         Output := mulperms(Output, invperm
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[RotorOffsets[1]]));
> 
>         Output := mulperms(Output, 
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle[RotorOffsets[2]]);
>         Output := mulperms(Output, invperm
  (ListOfRotorWirings[2]));
>         Output := mulperms(Output, invperm
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[RotorOffsets[2]]));
> 
>         Output := mulperms(Output, 
  ListOfPermutationsUsedToSimulateRotorRotationsAro
  undAxle[RotorOffsets[3]]);
>         Output := mulperms(Output, invperm
  (ListOfRotorWirings[3]));
>         Output := mulperms(Output, invperm
  (ListOfPermutationsUsedToSimulateRotorRotationsAr
  oundAxle[RotorOffsets[3]]));
> 
>         Output := mulperms(Output, invperm
  (Plugboard));
> 
>         return Output ;
> end proc;
> 
> 
> AdvanceRotorSettings := proc
  (RotorSettingOffsets::list, Alphabet::list)
  ::list;
>         local NewRotorSettingOffsets;
```

```
>         NewRotorSettingOffsets :=
  RotorSettingOffsets;
>
>         if `mod`(NewRotorSettingOffsets[3], nops
  (Alphabet))+1 = 1 and `mod`
  (NewRotorSettingOffsets[2], nops(Alphabet))+1 =
  1 then
>                 NewRotorSettingOffsets[1] :=
  `mod`(NewRotorSettingOffsets[1], nops(Alphabet))
  +1;
>                 NewRotorSettingOffsets[2] :=
  `mod`(NewRotorSettingOffsets[2], nops(Alphabet))
  +1;
>                 NewRotorSettingOffsets[3] :=
  `mod`(NewRotorSettingOffsets[3], nops(Alphabet))
  +1;
>         elif `mod`(NewRotorSettingOffsets[3],
  nops(Alphabet))+1 = 1 and `mod`
  (NewRotorSettingOffsets[2], nops(Alphabet))+1 <>
  1 then
>                 NewRotorSettingOffsets[2] :=
  `mod`(NewRotorSettingOffsets[2], nops(Alphabet))
  +1;
>                 NewRotorSettingOffsets[3] :=
  `mod`(NewRotorSettingOffsets[3], nops(Alphabet))
  +1;
>         else
>                 NewRotorSettingOffsets[3] :=
  `mod`(NewRotorSettingOffsets[3], nops(Alphabet))
  +1;
>         end if;
>
>         return NewRotorSettingOffsets;
> end proc;
>
> ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation := proc (TextOffset::integer,
  EnigmaEncryptionPermutation::list)::integer;
>         local CiphertextOffset, ExternalIndex,
  InternalIndex;
```

```
>         CiphertextOffset := TextOffset;
>
>         for ExternalIndex to nops
  (EnigmaEncryptionPermutation) do
>                 for InternalIndex to nops
  (EnigmaEncryptionPermutation[ExternalIndex]) do
>                         if
  EnigmaEncryptionPermutation[ExternalIndex]
  [InternalIndex] = TextOffset then
>                                 CiphertextOffset
  := EnigmaEncryptionPermutation[ExternalIndex]
  [`mod`(InternalIndex, nops
  (EnigmaEncryptionPermutation[ExternalIndex]))+1]
  ;
>                         end if;
>                 end do;
>         end do;
>
>         return CiphertextOffset;
> end proc;
>
> GetIndex := proc (GivenPartitionSignature::list)
  ::integer;
>         return Occurrences(13,
  GivenPartitionSignature) * 684 + Occurrences(12,
  GivenPartitionSignature) * 683 + Occurrences(11,
  GivenPartitionSignature) * 681 + Occurrences(10,
  GivenPartitionSignature) * 673 + Occurrences(9,
  GivenPartitionSignature) * 643 + Occurrences(8,
  GivenPartitionSignature) * 554 + Occurrences(7,
  GivenPartitionSignature) * 369 + Occurrences(6,
  GivenPartitionSignature) * 184 + Occurrences(5,
  GivenPartitionSignature) * 88 + Occurrences(4,
  GivenPartitionSignature) * 29  + Occurrences(3,
  GivenPartitionSignature) * 7 + Occurrences(2,
  GivenPartitionSignature);
> end proc;
>
```

```
> DeterminePlugboardWirings := proc
  (LetterAssignments::list,
  AvailableLetters::list,
  TextOffsetsIndex::integer,
  ListOfPossibleSolutions::list,
  EnigmaEncryptionPermutationsWithoutPlugboard::lis
  t, PlaintextOffsets::list, CiphertextOffsets)
  ::list;
>         local i, CompanionLetterOffset,
  LocalListOfPossibleSolutions;
>
>         LocalListOfPossibleSolutions :=
  ListOfPossibleSolutions;
>
>         if TextOffsetsIndex = nops
  (PlaintextOffsets)+1 then
>                 LocalListOfPossibleSolutions :=
  [op(LocalListOfPossibleSolutions), [convert
  (LetterAssignments, 'disjcyc'),
  AvailableLetters]];
>
>                 return
  LocalListOfPossibleSolutions;
>         end if;
>
>         if AvailableLetters[PlaintextOffsets
  [TextOffsetsIndex]] = 1 then
>                 if AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 then
>                         if
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]) = CiphertextOffsets
  [TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
```

```
  CiphertextOffsets[TextOffsetsIndex] =
  CiphertextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, CiphertextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                             end if;
>
>                             CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>
>                             if AvailableLetters
  [CompanionLetterOffset] = 1 and AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 and
  CompanionLetterOffset <> CiphertextOffsets
  [TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
  CompanionLetterOffset = CiphertextOffsets
  [TextOffsetsIndex], CiphertextOffsets
  [TextOffsetsIndex] = CompanionLetterOffset,
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, CompanionLetterOffset =
  0, CiphertextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                             end if;
>                   else
```

```
>                               if
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]) = CiphertextOffsets
  [TextOffsetsIndex] and LetterAssignments
  [CiphertextOffsets[TextOffsetsIndex]] =
  CiphertextOffsets[TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                               end if;
>
>                               CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(PlaintextOffsets[TextOffsetsIndex],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>
>                               if AvailableLetters
  [CompanionLetterOffset] = 0 and
  CompanionLetterOffset <> CiphertextOffsets
  [TextOffsetsIndex] and CompanionLetterOffset =
  LetterAssignments[CiphertextOffsets
  [TextOffsetsIndex]] then
```

```
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(PlaintextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                       end if;
>                end if;
>         else CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(LetterAssignments[PlaintextOffsets
  [TextOffsetsIndex]],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>                if AvailableLetters
  [CompanionLetterOffset] = 1 and AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 and
  CompanionLetterOffset <> CiphertextOffsets
  [TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (CompanionLetterOffset = CiphertextOffsets
  [TextOffsetsIndex], CiphertextOffsets
  [TextOffsetsIndex] = CompanionLetterOffset,
  LetterAssignments), subsop(CompanionLetterOffset
  = 0, CiphertextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                end if;
```

```
>                 if
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(LetterAssignments[PlaintextOffsets
  [TextOffsetsIndex]],
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]) = LetterAssignments
  [CiphertextOffsets[TextOffsetsIndex]] then
>                         if AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 0 then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(LetterAssignments,
  AvailableLetters, TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                         else
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (CiphertextOffsets[TextOffsetsIndex] =
  LetterAssignments[CiphertextOffsets
  [TextOffsetsIndex]], LetterAssignments), subsop
  (CiphertextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                         end if;
>                 else
>                         return
  LocalListOfPossibleSolutions;
>                 end if;
>         end if;
>
>         for i to nops(LetterAssignments) do
```

```
>                             CompanionLetterOffset :=
  ExtractCiphertextOffsetFromEnigmaEncryptionPermut
  ation(i,
  EnigmaEncryptionPermutationsWithoutPlugboard
  [TextOffsetsIndex]);
>                             if AvailableLetters
  [CompanionLetterOffset] = 1 and AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 and
  CompanionLetterOffset <> PlaintextOffsets
  [TextOffsetsIndex] and CompanionLetterOffset <>
  CiphertextOffsets[TextOffsetsIndex] and i <>
  CiphertextOffsets[TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] = i, i =
  PlaintextOffsets[TextOffsetsIndex],
  CompanionLetterOffset = CiphertextOffsets
  [TextOffsetsIndex], CiphertextOffsets
  [TextOffsetsIndex] = CompanionLetterOffset,
  LetterAssignments), subsop(i = 0,
  PlaintextOffsets[TextOffsetsIndex] = 0,
  CompanionLetterOffset = 0, CiphertextOffsets
  [TextOffsetsIndex] = 0, AvailableLetters),
  TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                             elif AvailableLetters
  [CompanionLetterOffset] = 1 and AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 1 and
  CompanionLetterOffset <> PlaintextOffsets
  [TextOffsetsIndex] and CompanionLetterOffset =
  CiphertextOffsets[TextOffsetsIndex] and i <>
  CiphertextOffsets[TextOffsetsIndex] then
>
```

```
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] = i, i =
  PlaintextOffsets[TextOffsetsIndex],
  CiphertextOffsets[TextOffsetsIndex] =
  CiphertextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(i = 0,
  PlaintextOffsets[TextOffsetsIndex] = 0,
  CiphertextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                       elif AvailableLetters
  [CiphertextOffsets[TextOffsetsIndex]] = 0 and
  CompanionLetterOffset = LetterAssignments
  [CiphertextOffsets[TextOffsetsIndex]] and i <>
  CiphertextOffsets[TextOffsetsIndex] then
>
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] = i, i =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(i = 0,
  PlaintextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                       elif PlaintextOffsets
  [TextOffsetsIndex] = CompanionLetterOffset and
  CiphertextOffsets[TextOffsetsIndex] = i then
```

```
  LocalListOfPossibleSolutions :=
  DeterminePlugboardWirings(subsop
  (PlaintextOffsets[TextOffsetsIndex] = i, i =
  PlaintextOffsets[TextOffsetsIndex],
  LetterAssignments), subsop(i = 0,
  PlaintextOffsets[TextOffsetsIndex] = 0,
  AvailableLetters), TextOffsetsIndex+1,
  LocalListOfPossibleSolutions,
  EnigmaEncryptionPermutationsWithoutPlugboard,
  PlaintextOffsets, CiphertextOffsets);
>                               end if;
>                     end if;
>           end do;
>
>           return LocalListOfPossibleSolutions;
> end proc;
```

A.2 Encryption Program (Test05Encrypt.mpl) [Output Only]

ListOfPermutationsUsedToSimulateRotorRotationsAroundAxle :=

[[], [[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]], [[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25], [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26]], [[1, 4, 7, 10, 13, 16, 19, 22, 25, 2, 5, 8, 11, 14, 17, 20, 23, 26, 3, 6, 9, 12, 15, 18, 21, 24]], [[1, 5, 9, 13, 17, 21, 25, 3, 7, 11, 15, 19, 23], [2, 6, 10, 14, 18, 22, 26, 4, 8, 12, 16, 20, 24]], [[1, 6, 11, 16, 21, 26, 5, 10, 15, 20, 25, 4, 9, 14, 19, 24, 3, 8, 13, 18, 23, 2, 7, 12, 17, 22]], [[1, 7, 13, 19, 25, 5, 11, 17, 23, 3, 9, 15, 21], [2, 8, 14, 20, 26, 6, 12, 18, 24, 4, 10, 16, 22]], [[1, 8, 15, 22, 3, 10, 17, 24, 5, 12, 19, 26, 7, 14, 21, 2, 9, 16, 23, 4, 11, 18, 25, 6, 13, 20]], [[1, 9, 17, 25, 7, 15, 23, 5, 13, 21, 3, 11, 19], [2, 10, 18, 26, 8, 16, 24, 6, 14, 22, 4, 12, 20]], [[1, 10, 19, 2, 11, 20, 3, 12, 21, 4, 13, 22, 5, 14, 23, 6, 15, 24, 7, 16, 25, 8, 17, 26, 9, 18]], [[1, 11, 21, 5, 15, 25, 9, 19, 3, 13, 23, 7, 17], [2, 12, 22, 6, 16, 26, 10, 20, 4, 14, 24, 8, 18]], [[1, 12, 23, 8, 19, 4, 15, 26, 11, 22, 7, 18, 3, 14, 25, 10, 21, 6, 17, 2, 13, 24, 9, 20, 5, 16]], [[1, 13, 25, 11, 23, 9, 21, 7, 19, 5, 17, 3, 15], [2, 14, 26, 12, 24, 10, 22, 8, 20, 6, 18, 4, 16]], [[1, 14], [2, 15], [3, 16], [4, 17], [5, 18], [6, 19], [7, 20], [8, 21],

[9, 22], [10, 23], [11, 24], [12, 25], [13, 26]], [[1, 15, 3, 17, 5, 19, 7, 21, 9, 23, 11, 25, 13], [2, 16, 4, 18, 6, 20, 8, 22, 10, 24, 12, 26, 14]], [[1, 16, 5, 20, 9, 24, 13, 2, 17, 6, 21, 10, 25, 14, 3, 18, 7, 22, 11, 26, 15, 4, 19, 8, 23, 12]], [[1, 17, 7, 23, 13, 3, 19, 9, 25, 15, 5, 21, 11], [2, 18, 8, 24, 14, 4, 20, 10, 26, 16, 6, 22, 12]], [[1, 18, 9, 26, 17, 8, 25, 16, 7, 24, 15, 6, 23, 14, 5, 22, 13, 4, 21, 12, 3, 20, 11, 2, 19, 10]], [[1, 19, 11, 3, 21, 13, 5, 23, 15, 7, 25, 17, 9], [2, 20, 12, 4, 22, 14, 6, 24, 16, 8, 26, 18, 10]], [[1, 20, 13, 6, 25, 18, 11, 4, 23, 16, 9, 2, 21, 14, 7, 26, 19, 12, 5, 24, 17, 10, 3, 22, 15, 8]], [[1, 21, 15, 9, 3, 23, 17, 11, 5, 25, 19, 13, 7], [2, 22, 16, 10, 4, 24, 18, 12, 6, 26, 20, 14, 8]], [[1, 22, 17, 12, 7, 2, 23, 18, 13, 8, 3, 24, 19, 14, 9, 4, 25, 20, 15, 10, 5, 26, 21, 16, 11, 6]], [[1, 23, 19, 15, 11, 7, 3, 25, 21, 17, 13, 9, 5], [2, 24, 20, 16, 12, 8, 4, 26, 22, 18, 14, 10, 6]], [[1, 24, 21, 18, 15, 12, 9, 6, 3, 26, 23, 20, 17, 14, 11, 8, 5, 2, 25, 22, 19, 16, 13, 10, 7, 4]], [[1, 25, 23, 21, 19, 17, 15, 13, 11, 9, 7, 5, 3], [2, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4]], [[1, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2]]]

DailySetting := ["X", "C", "D"]

OperatorSetting := ["R", "H", "C"]

Plaintext := ["A", "N", "M", "A", "R", "T", "I", "N", "O", "B", "E", "R", "Z", "A", "L", "E", "K", "X", "K", "E", "I", "N", "E", "B", "E", "S", "O", "N", "D", "E", "R", "E", "N", "E", "R", "E", "I", "G", "N", "I", "S", "S", "E", "I", "N", "D", "E", "R", "B", "U", "L",

"M", "E", "H", "E", "U", "T", "E"]

Plaintext := ["R", "H", "C", "R", "H", "C", "A", "N", "M", "A", "R", "T", "I", "N", "O", "B", "E", "R", "Z", "A", "L", "E", "K", "X", "K", "E", "I", "N", "E", "B", "E", "S", "O", "N", "D", "E", "R", "E", "N", "E", "R", "E", "I", "G", "N", "I", "S", "S", "E", "I", "N", "D", "E", "R", "B", "U", "L", "M", "E", "H", "E", "U", "T", "E"]

DailySettingOffsets := [24, 3, 4]

OperatorSettingOffsets := [18, 8, 3]

PlaintextOffsets := [18, 8, 3, 18, 8, 3, 1, 14, 13, 1, 18, 20, 9, 14, 15, 2, 5, 18, 26, 1, 12, 5, 11, 24, 11, 5, 9, 14, 5, 2, 5, 19, 15, 14, 4, 5, 18, 5, 14, 5, 18, 5, 9, 7, 14, 9, 19, 19, 5, 9, 14, 4, 5, 18, 2, 21, 12, 13, 5, 8, 5, 21, 20, 5]

CurrentRotorOffsets := [24, 3, 4]

EnigmaEncryptionPermutationsWithPlugboard := []

[[1, 26], [2, 14], [3, 22], [4, 17], [5, 18], [6, 10], [7, 21], [8, 11], [9, 23], [12, 19], [13, 20], [15, 24], [16, 25]]

[[1, 24], [2, 20], [3, 11], [4, 10], [5, 15], [6, 26], [7, 12], [8, 21], [9, 22], [13, 17], [14, 19], [16, 18], [23, 25]]

[[1, 15], [2, 5], [3, 18], [4, 10], [6, 23], [7, 17], [8, 20], [9, 24], [11, 26], [12, 22], [13, 21], [14, 16], [19, 25]]

[[1, 14], [2, 3], [4, 19], [5, 22], [6, 11], [7, 26], [8, 25], [9, 13], [10, 21], [12, 23], [15, 16], [17, 18], [20, 24]]

[[1, 17], [2, 11], [3, 19], [4, 15], [5, 18], [6, 12], [7, 25], [8, 14], [9, 21], [10, 23], [13, 26], [16, 24], [20, 22]]

[[1, 25], [2, 23], [3, 18], [4, 17], [5, 10], [6, 19], [7, 26], [8, 12], [9, 16], [11, 14], [13, 21], [15, 22], [20, 24]]

CurrentRotorOffsets := [18, 8, 3]

[[1, 15], [2, 9], [3, 6], [4, 13], [5, 8], [7, 10], [11, 12], [14, 21], [16, 23], [17, 26], [18, 24], [19, 25], [20, 22]]

[[1, 8], [2, 7], [3, 21], [4, 26], [5, 9], [6, 13], [10, 12], [11, 14], [15, 20], [16, 22], [17, 18], [19, 25], [23, 24]]

[[1, 15], [2, 10], [3, 12], [4, 7], [5, 20], [6, 17], [8, 18], [9, 19], [11, 14], [13, 21], [16, 24], [22, 23], [25, 26]]

[[1, 4], [2, 11], [3, 22], [5, 17], [6, 8], [7, 12], [9, 10], [13, 23], [14, 20], [15, 26], [16, 19], [18, 21], [24, 25]]

[[1, 2], [3, 7], [4, 26], [5, 23], [6, 18], [8, 19], [9, 22], [10, 20], [11, 24], [12, 14], [13, 16], [15, 21], [17, 25]]

[[1, 3], [2, 15], [4, 14], [5, 17], [6, 18], [7, 11], [8, 23], [9, 12], [10, 21], [13, 16], [19, 22], [20, 26], [24, 25]]

[[1, 15], [2, 20], [3, 19], [4, 10], [5, 14], [6, 23], [7, 24], [8, 13], [9, 12], [11, 21], [16, 18], [17, 22], [25, 26]]

[[1, 24], [2, 3], [4, 26], [5, 19], [6, 23], [7, 14], [8, 25], [9, 10], [11, 13], [12, 16], [15, 20], [17, 21], [18, 22]]

[[1, 5], [2, 18], [3, 7], [4, 22], [6, 9], [8, 21], [10, 14], [11, 25], [12, 15], [13, 17], [16, 23], [19, 20], [24, 26]]

[[1, 8], [2, 21], [3, 10], [4, 20], [5, 18], [6, 22], [7, 23], [9, 16], [11, 26], [12, 25], [13, 19], [14, 17], [15, 24]]

[[1, 25], [2, 12], [3, 15], [4, 26], [5, 14], [6, 23], [7, 21], [8, 17], [9, 13], [10, 22], [11, 16], [18, 24], [19, 20]]

[[1, 22], [2, 14], [3, 20], [4, 8], [5, 11], [6, 21], [7, 25], [9, 19], [10, 18], [12, 15], [13, 17], [16, 26], [23, 24]]

[[1, 21], [2, 11], [3, 25], [4, 6], [5, 12], [7, 15], [8, 26], [9, 16], [10, 19], [13, 14], [17, 23], [18, 22], [20, 24]]

[[1, 9], [2, 22], [3, 13], [4, 8], [5, 24], [6, 7], [10, 15], [11, 16], [12, 25], [14, 23], [17, 19], [18, 26], [20, 21]]

[[1, 15], [2, 21], [3, 17], [4, 24], [5, 7], [6, 9], [8, 22], [10, 18], [11, 13],

[12, 25], [14, 19], [16, 23], [20, 26]]

[[1, 14], [2, 17], [3, 22], [4, 20], [5, 9], [6, 7], [8, 10], [11, 26], [12, 16], [13, 23], [15, 18], [19, 25], [21, 24]]

[[1, 20], [2, 6], [3, 9], [4, 19], [5, 21], [7, 15], [8, 18], [10, 14], [11, 13], [12, 24], [16, 22], [17, 26], [23, 25]]

[[1, 3], [2, 19], [4, 11], [5, 17], [6, 12], [7, 10], [8, 21], [9, 25], [13, 16], [14, 22], [15, 26], [18, 20], [23, 24]]

[[1, 25], [2, 26], [3, 11], [4, 18], [5, 8], [6, 17], [7, 13], [9, 10], [12, 19], [14, 21], [15, 20], [16, 22], [23, 24]]

[[1, 16], [2, 26], [3, 8], [4, 14], [5, 24], [6, 21], [7, 17], [9, 10], [11, 20], [12, 22], [13, 19], [15, 18], [23, 25]]

[[1, 13], [2, 11], [3, 8], [4, 22], [5, 19], [6, 20], [7, 24], [9, 10], [12, 25], [14, 15], [16, 18], [17, 23], [21, 26]]

[[1, 26], [2, 14], [3, 4], [5, 9], [6, 21], [7, 10], [8, 12], [11, 22], [13, 16], [15, 20], [17, 24], [18, 19], [23, 25]]

[[1, 12], [2, 14], [3, 15], [4, 9], [5, 13], [6, 26], [7, 20], [8, 11], [10, 16], [17, 23], [18, 19], [21, 24], [22, 25]]

[[1, 16], [2, 21], [3, 5], [4, 12], [6, 8], [7, 20], [9, 10], [11, 25], [13, 23], [14, 18], [15, 22], [17, 19], [24, 26]]

[[1, 19], [2, 12], [3, 23], [4, 21], [5, 20], [6, 11], [7, 13], [8, 25], [9, 18], [10, 24], [14, 16], [15, 26], [17, 22]]

[[1, 26], [2, 8], [3, 14], [4, 5], [6, 18], [7, 22], [9, 21], [10, 17], [11, 19], [12, 24], [13, 25], [15, 16], [20, 23]]

[[1, 19], [2, 5], [3, 18], [4, 12], [6, 22], [7, 9], [8, 13], [10, 21], [11, 26], [14, 17], [15, 24], [16, 20], [23, 25]]

[[1, 14], [2, 9], [3, 4], [5, 24], [6, 7], [8, 23], [10, 26], [11, 13], [12, 16], [15, 21], [17, 20], [18, 19], [22, 25]]

[[1, 22], [2, 15], [3, 17], [4, 23], [5, 26], [6, 21], [7, 8], [9, 11], [10, 20], [12, 24], [13, 19], [14, 16], [18, 25]]

[[1, 9], [2, 14], [3, 16], [4, 22], [5, 12], [6, 13], [7, 19], [8, 15], [10, 17], [11, 21], [18, 24], [20, 23], [25, 26]]

[[1, 16], [2, 12], [3, 24], [4, 20], [5, 7], [6, 17], [8, 18], [9, 19], [10, 14], [11, 22], [13, 21], [15, 26], [23, 25]]

[[1, 16], [2, 3], [4, 11], [5, 7], [6, 20], [8, 17], [9, 18], [10, 25], [12, 24], [13, 23], [14, 15], [19, 26], [21, 22]]

[[1, 22], [2, 14], [3, 25], [4, 15], [5, 7], [6, 20], [8, 23], [9, 13], [10, 19], [11, 18], [12, 24], [16, 17], [21, 26]]

[[1, 22], [2, 13], [3, 7], [4, 11], [5, 6], [8, 24], [9, 15], [10, 19], [12, 23], [14, 18], [16, 17], [20, 21], [25, 26]]

[[1, 12], [2, 6], [3, 5], [4, 26], [7, 17], [8, 24], [9, 22], [10, 23], [11, 18], [13, 16], [14, 15], [19, 21], [20, 25]]

[[1, 6], [2, 10], [3, 5], [4, 26], [7, 22], [8, 25], [9, 12], [11, 15], [13, 16],

[14, 23], [17, 19], [18, 24], [20, 21]]

[[1, 2], [3, 9], [4, 16], [5, 15], [6, 11], [7, 22], [8, 25], [10, 23], [12, 14], [13, 20], [17, 19], [18, 26], [21, 24]]

[[1, 21], [2, 13], [3, 8], [4, 26], [5, 12], [6, 23], [7, 17], [9, 25], [10, 24], [11, 14], [15, 19], [16, 18], [20, 22]]

[[1, 16], [2, 4], [3, 5], [6, 13], [7, 24], [8, 22], [9, 26], [10, 20], [11, 18], [12, 23], [14, 21], [15, 25], [17, 19]]

[[1, 14], [2, 4], [3, 5], [6, 18], [7, 10], [8, 13], [9, 26], [11, 22], [12, 23], [15, 17], [16, 20], [19, 25], [21, 24]]

[[1, 7], [2, 20], [3, 24], [4, 15], [5, 9], [6, 8], [10, 12], [11, 19], [13, 21], [14, 16], [17, 22], [18, 23], [25, 26]]

[[1, 22], [2, 12], [3, 25], [4, 10], [5, 21], [6, 19], [7, 15], [8, 17], [9, 14], [11, 23], [13, 20], [16, 26], [18, 24]]

[[1, 25], [2, 15], [3, 23], [4, 14], [5, 6], [7, 18], [8, 22], [9, 12], [10, 11], [13, 20], [16, 17], [19, 24], [21, 26]]

[[1, 9], [2, 3], [4, 7], [5, 12], [6, 22], [8, 20], [10, 23], [11, 15], [13, 21], [14, 24], [16, 25], [17, 26], [18, 19]]

[[1, 3], [2, 17], [4, 9], [5, 7], [6, 19], [8, 24], [10, 25], [11, 22], [12, 23], [13, 20], [14, 16], [15, 26], [18, 21]]

[[1, 14], [2, 5], [3, 11], [4, 18], [6, 23], [7, 16], [8, 21], [9, 20], [10, 22], [12, 13], [15, 24], [17, 26], [19, 25]]

[[1, 10], [2, 15], [3, 12], [4, 5], [6, 14], [7, 22], [8, 16], [9, 18], [11, 17], [13, 26], [19, 23], [20, 24], [21, 25]]

[[1, 9], [2, 18], [3, 5], [4, 8], [6, 24], [7, 20], [10, 12], [11, 25], [13, 15], [14, 17], [16, 26], [19, 23], [21, 22]]

[[1, 14], [2, 7], [3, 24], [4, 26], [5, 11], [6, 12], [8, 23], [9, 22], [10, 13], [15, 17], [16, 19], [18, 21], [20, 25]]

[[1, 24], [2, 21], [3, 12], [4, 26], [5, 11], [6, 10], [7, 16], [8, 20], [9, 25], [13, 22], [14, 19], [15, 18], [17, 23]]

[[1, 25], [2, 26], [3, 23], [4, 12], [5, 11], [6, 14], [7, 17], [8, 22], [9, 21], [10, 13], [15, 18], [16, 24], [19, 20]]

[[1, 10], [2, 23], [3, 25], [4, 26], [5, 14], [6, 11], [7, 17], [8, 19], [9, 21], [12, 16], [13, 15], [18, 20], [22, 24]]

[[1, 18], [2, 8], [3, 9], [4, 12], [5, 24], [6, 17], [7, 23], [10, 16], [11, 15], [13, 20], [14, 26], [19, 25], [21, 22]]

[[1, 4], [2, 12], [3, 15], [5, 7], [6, 13], [8, 19], [9, 21], [10, 26], [11, 17], [14, 18], [16, 25], [20, 24], [22, 23]]

[[1, 22], [2, 20], [3, 15], [4, 8], [5, 19], [6, 12], [7, 17], [9, 23], [10, 13], [11, 26], [14, 25], [16, 24], [18, 21]]

[[1, 3], [2, 14], [4, 18], [5, 11], [6, 23], [7, 26], [8, 17], [9, 25], [10, 13], [12, 15], [16, 19], [20, 22], [21, 24]]

[[1, 6], [2, 7], [3, 11], [4, 26], [5, 15], [8, 10], [9, 19], [12, 20], [13, 22],

[14, 24], [16, 25], [17, 18], [21, 23]]

[[1, 12], [2, 6], [3, 9], [4, 15], [5, 7], [8, 13], [10, 19], [11, 21], [14, 20], [16, 26], [17, 18], [22, 23], [24, 25]]

Ciphertext := ""

"EURQNROKUDFZLGLUNJHIYIMWCXJBMUTKXAWLHGB FKCCQUZKFFAPRDBGBDOXSSXLG"

ConstructedDoubleEncryptionPermutation1WithPlugboard := [[1, 7, 10, 11, 25, 15, 20, 9, 12, 4, 18, 22, 2], [3, 5, 17, 19, 23, 13, 24, 16, 8, 6, 21, 26, 14]]

ConstructedDoubleEncryptionPermutation2WithPlugboard := [[1, 16, 5, 4, 23, 7, 6, 13], [2, 22, 21, 14, 3], [8, 9, 20, 11, 19], [10, 15, 18, 24, 17, 26, 12, 25]]

ConstructedDoubleEncryptionPermutation3WithPlugboard := [[1, 22, 8, 24, 16, 11, 7, 4, 5, 23, 19], [2, 10, 17, 26, 14, 9, 20, 12, 15, 25, 6]]

A.3 Decryption Program (Test05Encrypt02.mpl) [Output Only]

ListOfPermutationsUsedToSimulateRotorRotationsAroundAxle :=

[[], [[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]], [[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25], [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26]], [[1, 4, 7, 10, 13, 16, 19, 22, 25, 2, 5, 8, 11, 14, 17, 20, 23, 26, 3, 6, 9, 12, 15, 18, 21, 24]], [[1, 5, 9, 13, 17, 21, 25, 3, 7, 11, 15, 19, 23], [2, 6, 10, 14, 18, 22, 26, 4, 8, 12, 16, 20, 24]], [[1, 6, 11, 16, 21, 26, 5, 10, 15, 20, 25, 4, 9, 14, 19, 24, 3, 8, 13, 18, 23, 2, 7, 12, 17, 22]], [[1, 7, 13, 19, 25, 5, 11, 17, 23, 3, 9, 15, 21], [2, 8, 14, 20, 26, 6, 12, 18, 24, 4, 10, 16, 22]], [[1, 8, 15, 22, 3, 10, 17, 24, 5, 12, 19, 26, 7, 14, 21, 2, 9, 16, 23, 4, 11, 18, 25, 6, 13, 20]], [[1, 9, 17, 25, 7, 15, 23, 5, 13, 21, 3, 11, 19], [2, 10, 18, 26, 8, 16, 24, 6, 14, 22, 4, 12, 20]], [[1, 10, 19, 2, 11, 20, 3, 12, 21, 4, 13, 22, 5, 14, 23, 6, 15, 24, 7, 16, 25, 8, 17, 26, 9, 18]], [[1, 11, 21, 5, 15, 25, 9, 19, 3, 13, 23, 7, 17], [2, 12, 22, 6, 16, 26, 10, 20, 4, 14, 24, 8, 18]], [[1, 12, 23, 8, 19, 4, 15, 26, 11, 22, 7, 18, 3, 14, 25, 10, 21, 6, 17, 2, 13, 24, 9, 20, 5, 16]], [[1, 13, 25, 11, 23, 9, 21, 7, 19, 5, 17, 3, 15], [2, 14, 26, 12, 24, 10, 22, 8, 20, 6, 18, 4, 16]], [[1, 14], [2, 15], [3, 16], [4, 17], [5, 18], [6, 19], [7, 20], [8, 21],

[9, 22], [10, 23], [11, 24], [12, 25], [13, 26]], [[1, 15, 3, 17, 5, 19, 7, 21, 9, 23, 11, 25, 13], [2, 16, 4, 18, 6, 20, 8, 22, 10, 24, 12, 26, 14]], [[1, 16, 5, 20, 9, 24, 13, 2, 17, 6, 21, 10, 25, 14, 3, 18, 7, 22, 11, 26, 15, 4, 19, 8, 23, 12]], [[1, 17, 7, 23, 13, 3, 19, 9, 25, 15, 5, 21, 11], [2, 18, 8, 24, 14, 4, 20, 10, 26, 16, 6, 22, 12]], [[1, 18, 9, 26, 17, 8, 25, 16, 7, 24, 15, 6, 23, 14, 5, 22, 13, 4, 21, 12, 3, 20, 11, 2, 19, 10]], [[1, 19, 11, 3, 21, 13, 5, 23, 15, 7, 25, 17, 9], [2, 20, 12, 4, 22, 14, 6, 24, 16, 8, 26, 18, 10]], [[1, 20, 13, 6, 25, 18, 11, 4, 23, 16, 9, 2, 21, 14, 7, 26, 19, 12, 5, 24, 17, 10, 3, 22, 15, 8]], [[1, 21, 15, 9, 3, 23, 17, 11, 5, 25, 19, 13, 7], [2, 22, 16, 10, 4, 24, 18, 12, 6, 26, 20, 14, 8]], [[1, 22, 17, 12, 7, 2, 23, 18, 13, 8, 3, 24, 19, 14, 9, 4, 25, 20, 15, 10, 5, 26, 21, 16, 11, 6]], [[1, 23, 19, 15, 11, 7, 3, 25, 21, 17, 13, 9, 5], [2, 24, 20, 16, 12, 8, 4, 26, 22, 18, 14, 10, 6]], [[1, 24, 21, 18, 15, 12, 9, 6, 3, 26, 23, 20, 17, 14, 11, 8, 5, 2, 25, 22, 19, 16, 13, 10, 7, 4]], [[1, 25, 23, 21, 19, 17, 15, 13, 11, 9, 7, 5, 3], [2, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4]], [[1, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2]]]

Ciphertext := ["E", "U", "R", "Q", "N", "R", "O", "K", "U", "D", "F", "Z", "L", "G", "L", "U", "N", "J", "H", "I", "Y", "I", "M", "W", "C", "X", "J", "B", "M", "U", "T", "K", "X", "A", "W", "L", "H", "G", "B", "F", "K", "C", "C", "Q", "U", "Z", "K", "F", "F", "A", "P", "R", "D", "B", "G", "B", "D", "O", "X", "S", "S", "X", "L", "G"]

CiphertextOffsets := [5, 21, 18, 17, 14, 18, 15, 11, 21, 4, 6, 26, 12, 7, 12, 21, 14, 10, 8, 9, 25, 9, 13, 23, 3, 24, 10, 2, 13, 21, 20, 11, 24, 1, 23, 12, 8, 7, 2, 6, 11, 3, 3, 17, 21, 26, 11, 6, 6, 1, 16, 18, 4, 2, 7, 2, 4, 15, 24, 19, 19, 24, 12, 7]

CribCiphertext := ["C", "X", "J", "B", "M", "U", "T", "K", "X", "A", "W", "L", "H", "G", "B", "F", "K", "C", "C", "Q", "U", "Z", "K", "F", "F"]

CribCiphertextOffsets := [3, 24, 10, 2, 13, 21, 20, 11, 24, 1, 23, 12, 8, 7, 2, 6, 11, 3, 3, 17, 21, 26, 11, 6, 6]

CribPlaintext := ["K", "E", "I", "N", "E", "B", "E", "S", "O", "N", "D", "E", "R", "E", "N", "E", "R", "E", "I", "G", "N", "I", "S", "S", "E"]

CribPlaintextOffsets := [11, 5, 9, 14, 5, 2, 5, 19, 15, 14, 4, 5, 18, 5, 14, 5, 18, 5, 9, 7, 14, 9, 19, 19, 5]

ConstructedDoubleEncryptionPermutation1CycleStructure := []

ConstructedDoubleEncryptionPermutation1CycleStructure := [13, 13]

ConstructedDoubleEncryptionPermutation2CycleStructure := []

ConstructedDoubleEncryptionPermutation2CycleStructure := [8, 8, 5, 5]

ConstructedDoubleEncryptionPermutation3CycleStructure := []

ConstructedDoubleEncryptionPermutation3CycleStructure := [11, 11]

CurrentRotorOffsets := [1, 1, 1]

ListOfPossibleRotorSettingsForAllConstructedDoubleEncryptionPermutations := []

[[**12, 19, 7**], [**19, 2, 22**], [**24, 3, 4**], [**24, 21, 1**]]

4

ListOfPossibleDailyKeyRotorSettingsWithFoundOperatorAndPlugboardSettings := []

ListOfDecryptedOperatorSettings := []

ListOfDerivedPlugboardSettings := []

[[**24, 3, 4**]]

[[**18, 8, 3**]]

[[[**2, 12**], [**5, 26**], [**16, 24**]]]

"ANMARTINOBERZALEKXKEINEBESONDERENEREIGNISSEINDERBULMEHEUTE"

"Daily Key: ", [24, 3, 4]

"Operator Key: ", [18, 8, 3]

"Plugboard Settings: ", [[2, 12], [5, 26], [16, 24]]

Vita

Candidate's full name: John Darren Mummery

University attended (with dates and degrees obtained): University of New Brunswick, Bachelor of Computer Science, 2009

Publications: None

Conference Presentations: None