

Error detection in everyday identification numbers

Foutdetectie in alledaagse identificatienummers

M.M.J. Vons

Student no.
6053084

June 30, 2026



Cover image: own composition based on images [1], [2], [3], and [4].

Error detection in everyday identification numbers

Foutdetectie in alledaagse identificatienummers

by

Marit Vons

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Tuesday June 30, 2026 at 11:00 AM.

Student number: 6053084
Project duration: April 20, 2026 – June 30, 2026
Thesis committee: Dr. ir. J.H. Weber, TU Delft, supervisor
Dr. J.A.M. de Groot, TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Preface

This thesis is written as part of the Bachelor program of Applied Mathematics at Delft University of Technology. It marks the conclusion of my bachelor's degree and represents the result of several months of study and research. In my second year, I took the elective course *Applied Algebra: Codes*, lectured by Dr. J.A.M. de Groot. This course introduced the theory of error-correcting and error-detecting codes. I found this topic particularly interesting because it shows how mathematical ideas are used in practical applications such as identification numbers and data validation. This is therefore also where my interest in error detection started, which eventually led me to choose this topic for my bachelor thesis.

After starting my research into this topic, I discovered how widely check-digit algorithms are used in practice. They appear in many everyday identification systems, often without us even noticing. It is interesting that identification numbers such as credit card numbers, IBANs, and product codes rely on relatively simple mathematical calculations to detect errors. This makes the topic not only mathematically interesting, but also highly relevant in everyday life. As a result, I became more interested in understanding how different check-digit systems work and how well they perform in detecting various types of errors.

I would like to express my sincere gratitude to my supervisor, Jos Weber, for his guidance, support, and valuable feedback throughout this project. His insights and advice were essential in shaping and improving this thesis. I would also like to thank Dr. J.A.M. de Groot for teaching the course that sparked my interest in this subject, and for being part of my graduation committee.

*Marit Vons
Delft, June 2026*

Layman's summary

In everyday life, identification numbers are used in many important systems where even small errors can have significant consequences. Therefore, this thesis studies how identification numbers, such as credit card numbers or product codes, can be checked for mistakes using a check digit. This is an extra digit, usually placed at the end of an identification number. If a number is changed by mistake, the system can often detect that it is invalid.

The main goal of this thesis is to understand which types of errors can or cannot be detected by different check-digit systems. Several common types of errors are investigated, including changing a single digit or swapping two digits. Instead of focusing on one specific system, different existing methods are analysed within a general mathematical setting. This allows a comparison of how reliable different systems are.

The results show that the numbers used in the calculation, such as the weights and the modulus, influence which types of errors are detected. This helps explain why some systems are more effective than others at catching mistakes.

Summary

Since identification numbers play an important role in many everyday systems, even minor typing errors can have significant consequences. This thesis investigates the error detection capabilities of several widely used check digit systems, with a particular focus on modular arithmetic-based algorithms. The study begins by introducing the mathematical background of modular arithmetic and describing different types of common data entry errors.

Next, three real-world systems are analysed in detail. In particular, credit card numbers using the Luhn algorithm, International Bank Account Numbers (IBANs) using the MOD97-10 algorithm, and GTIN-13 product codes using an alternating-weight modulo 10 method. For each system, the structure of the number format and the corresponding check digit algorithm are formally defined and explained. A mathematical analysis is then carried out to determine which types of errors, including single-digit errors, transposition errors, twin errors, jump-transposition errors, and jump-twin errors can be detected. This mathematical analysis includes both formal proofs and a computational experiment.

The final part of the thesis introduces a generalised version of the GTIN-13 check digit algorithm, in which the fixed weight 3 is replaced by an arbitrary integer p . The resulting modulo 10 algorithm with alternating weight p is then analysed in detail. Necessary and sufficient conditions for error detection are derived in terms of greatest common divisors involving p , $p - 1$, $p + 1$, and the modulus 10. This generalisation illustrates how the value of the weighting parameter p affects the error detection properties of the system.

Overall, the results show that check digit algorithms can detect many common types of input errors such as single-digit errors, transposition errors, twin errors, jump-transposition errors, and jump-twin errors. Even though the algorithms have some limitations, they are still relatively effective in their ability to detect errors.

List of abbreviations

Abbreviation	Definition
BBAN	Basic Bank Account Number
BIN	Bank Identification Number
EAN	European Article Number
ECBS	European Committee for Banking Standards
gcd	Greatest common divisor
GTIN	Global Trade Item Number
IBAN	International Bank Account Number
IIN	Issuer Identification Number
ISBN	International Standard Book Number
ISO	International Organization for Standardization
MII	Major Industry Identifier
mod	Modulo
SWIFT	Society for Worldwide Interbank Financial Telecommunication

Contents

List of abbreviations	v
1 Introduction	1
2 Background	3
2.1 Modulo	3
2.2 Error types	4
3 Credit card number	5
3.1 Structure	5
3.1.1 Luhn algorithm	6
3.2 Analysis of error detection	7
3.2.1 Single-digit error	7
3.2.2 Transposition error	9
3.2.3 Twin error	11
3.2.4 Jump-transposition error	13
4 IBAN	14
4.1 Structure	14
4.1.1 MOD 97-10 algorithm	15
4.2 Analysis of error detection	17
4.2.1 Single-character error	18
4.2.2 Transposition error	25
5 GTIN-13	29
5.1 Structure	29
5.1.1 GTIN-13 check digit algorithm	30
5.2 Analysis of error detection	31
5.2.1 Single-digit error	31
5.2.2 Transposition error	32
5.2.3 Twin error	32
5.2.4 Jump-transposition error	33
5.2.5 Jump-twin error	33
6 Modulo 10 algorithm with alternating weight p	35
6.1 Algorithm	35
6.2 Analysis of error detection	36
6.2.1 Single-digit error	36
6.2.2 Transposition error	37
6.2.3 Twin error	38
6.2.4 Jump-transposition error	40
6.2.5 Jump-twin error	40
7 Conclusion and recommendations	42
7.1 Conclusion	42
7.2 Recommendations	43
A Python code for computational analysis of Luhn algorithm	44
B Python code for computational analysis of MOD97-10 algorithm	48
Bibliography	55

1

Introduction

In everyday life, identification numbers are used almost everywhere. Examples of these numbers are passport numbers, ISBNs, credit card numbers, citizen service numbers, barcode identifiers, and international banking codes. So, identification numbers can uniquely identify people, products, documents and bank accounts. Since these numbers are frequently entered manually in systems, there is a high probability of transcription and typing errors. Even a single incorrect digit can lead to failed transactions, incorrect records, data inconsistency, or security risks. Therefore, many numbering systems have a built-in error detection using check digits. A check digit is an additional digit added to a sequence and is calculated by applying a mathematical algorithm to this sequence. If the calculated result does not match the stored check digit, the system can detect that an error has occurred.

In this thesis, we will look at what type of identification numbers exist, how they are structured, and how their check digits are computed. Furthermore, an analysis will be performed on the error detection in these identification numbers. How many errors are detected, and what type of errors? By comparing several commonly used systems, the thesis aims to evaluate their strengths, weaknesses, and suitability for modern identification systems. The central research question of this thesis therefore is; How effective are common check digit algorithms in detecting errors in everyday identification numbers?

There is already a lot of information about the structure of different identification numbers. Most numbering systems, such as credit card numbers, International Bank Account Numbers (IBANs) and International Standard Book Numbers (ISBNs), are internationally standardised in ISO documentations [5][6][7]. ISO is the International Organisation of Standardisation, it develops and publishes standards for products, services, and systems. The numbering system GTIN (Global Trade Item Number) is standardised by the global organisation GS1 [8].

Credit card numbers use the Luhn algorithm to determine their check digit. The Luhn algorithm is effective in detecting all single-digit errors and almost all adjacent transposition errors. However, *Introduction to the Luhn Algorithm* by United Nations states several weaknesses of the algorithm [9]. For example, the transposition of the digit pairs 09 into 90 is not detected. This is also proven in the article *Transposition Error Detection in Luhn's Algorithm* by Waweru and Wachira [10]. The United Nations article also states that some twin errors remain undetected and that there exist more advanced algorithms, such as the Verhoeff algorithm, that can detect more errors but are less commonly used in practice due to their increased complexity. The article *Twin Error Detection in Luhn's Algorithm* by Wachira [11] formally explains that the twin errors, $22 \leftrightarrow 55$, $33 \leftrightarrow 66$, and $44 \leftrightarrow 77$, are not detected by the Luhn algorithm. Furthermore, *Error Detection and Correction on the Credit Card Number using Luhn Algorithm* by Wachira states that jump transpositions of the form $abb \leftrightarrow bba$ are also not detected [12].

To compute the two check digits of IBANs, the MOD 97-10 algorithm is used. ISO 7064 explains which type of errors this algorithm can detect [13]. All single-digit errors are detected. Nearly all transposition errors, either the transposition of two adjacent digits or the transposition of two digits with one character between them ($1432 \leftrightarrow 1234$ or $1324 \leftrightarrow 1234$), and nearly all shift errors, are detected by the MOD 97-10 algorithm. Moreover, a high proportion of double substitution errors are detected. These are two separate single-digit errors in one string.

GTIN-13 numbers use a weighted modulo 10 algorithm to compute their check digit. Since EAN-13 numbers and ISBNs are implemented into the GTIN-13 numbers, they can detect the same type of errors. Most articles on EAN-13 numbers specify error detection in the barcodes. Error detection in ISBN mostly focusses on the 10-digit ISBNs, but the article *Error Detection and Correction in the International Standard Book Number* by Kamaku [14] states that all single-digit error, and almost all transition errors are detected. However, it does not specify which transition errors are not detected. Furthermore, it also states that no jump-transposition error can be detected.

The thesis is structured as follows. Chapter 2 presents the theoretical background required for this study. It begins by introducing the concept of the modulo operation and discusses several properties of modular arithmetic that are frequently used in the proofs throughout this thesis. Furthermore, different types of errors are described, along with their structure and frequency of occurrence.

Chapter 3 begins by discussing the structure of credit card numbers, i.e. how they are composed. Furthermore, the Luhn algorithm, which is used to compute the check digit and validate credit card numbers, is introduced and formally defined. In addition, a theoretical and computational analysis of error detection in credit card numbers is carried out, focusing on different types of errors.

In Chapter 4 the structure of IBANs is introduced, together with the construction and computation of their check digits using the MOD97-10 algorithm. Both the recursive and polynomial methods for computing these check digits are discussed. Furthermore, a detailed mathematical analysis is presented of the error detection capabilities of the MOD97-10 algorithm when applied within the IBAN system. This includes formal proofs for different classes of errors, as well as a computational analysis using Python simulations to validate the theoretical results.

Chapter 5 introduces the structure of a GTIN-13 number. It then presents the algorithm used to compute the check digit and provides a formal mathematical analysis of its error detection capabilities. The analysis includes proofs for single-digit errors, transposition errors, twin errors, jump-transposition errors, and jump-twin errors.

Chapter 6 introduces a generalisation of the GTIN-13 check digit algorithm by replacing the fixed weight 3 with an arbitrary integer p in the alternating weighting scheme. The resulting modulo 10 algorithm with alternating weight p is formally defined and analysed in detail. In particular, the detectability of single-digit errors, transposition errors, twin errors, jump-transposition errors, and jump-twin errors is studied. For each error type, error detection conditions are derived in terms of greatest common divisors involving p , $p - 1$, $p + 1$, and the modulus 10.

Finally, Chapter 7 concludes the thesis by summarising the main findings, addressing the research questions, and outlining directions for future research.

2

Background

In this chapter the modulo operation is explained. Furthermore, congruence and several properties of the operation will be introduced. This chapter also discusses what type of errors exist when manually entering data into a system and what these errors look like.

2.1. Modulo

Dividing an integer by another integer, not equal to zero, does not always result in a new integer. Often, this division leads to a remainder. Determining this remainder is known as the modulo operation [15]. The study of the properties of the system of remainders is called modular arithmetic. It is an essential tool in number theory.

Definition 2.1. Let $a, n \in \mathbb{Z}$ with $n > 0$. The *modulo operation* that determines the remainder r when dividing a by n is denoted by

$$r = a \bmod n$$

The modulo operation is formally derived from Theorem 2.2. In this theorem a is the dividend, n is the modulo, q is the quotient, and r is the remainder [16].

Theorem 2.2. Let $a, n \in \mathbb{Z}$ with $n > 0$. Then, there exist integers q and r such that

$$a = qn + r \quad \text{and} \quad 0 \leq r < n$$

Moreover, q and r are uniquely determined by a and n .

An important concept that is introduced by the modulo operation is congruence between two integers.

Definition 2.3. Let $a, b \in \mathbb{Z}$ and let $n \in \mathbb{N}$. Then, a is *congruent to b modulo n* , written

$$a \equiv b \pmod{n}$$

if $a - b$ is divisible by n .

Congruence modulo n satisfies several important algebraic properties. In particular, congruence is preserved under addition and multiplication [16]. If $a \equiv b \pmod{n}$, then adding the same integer c to both sides preserves the congruence relation. So,

$$a + c \equiv b + c \pmod{n}$$

Similarly, if $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$, then congruence is also preserved under multiplication. Hence,

$$ac \equiv bd \pmod{n}$$

Another property of congruence modulo n is the modular cancellation law [17]. For any integer c ,

$$ac \equiv bc \pmod{n} \Leftrightarrow a \equiv b \pmod{\frac{n}{\gcd(c, n)}}$$

2.2. Error types

When manually entering data into a system, there is always a chance of making mistakes. This specific type of error is called a transmission error. There are different types of transmission errors and many of these follow recognisable patterns [18]. Below is an overview of several common types of transmission errors. Examples of the error types are displayed in Table 2.1

Single-digit error

A single-digit error occurs when one digit in a number is entered incorrectly, while all other digits remain correct. Even though only one digit changes, it can lead to linking an ID entry to the wrong person or failing to match an existing record in a database. 79.1% of all transmission error that occur are single-digit errors, making single-digit errors the most common type of transmission error [18]. This error is denoted as

$$a \leftrightarrow b \quad \text{with } a \neq b$$

Transposition error

A transposition error happens when two adjacent characters are swapped during entry. This type of transmission error is less common than single-digit errors since only 10.2% of transmission errors are transposition errors [18]. Nevertheless, they are still relatively common. This error is of the form

$$ab \leftrightarrow ba \quad \text{with } a \neq b$$

Twin error

When two identical and adjacent digits are replaced by a different pair of identical digits, the error is called a twin error. Of all transmission errors that can occur, only 0.5% are twin errors [18]. Twin errors are denoted by

$$aa \leftrightarrow bb \quad \text{with } a \neq b$$

Jump-transposition error

A jump-transposition error occurs when two different digits, separated by a third digit in between them, are interchanged. 0.8% of all possible transmission errors are jump-transposition errors [18]. Jump-transposition errors are of the form

$$abc \leftrightarrow cba \quad \text{with } a \neq c$$

Jump-twin error

When two identical digits, separated by a third digit in between, are replaced by a different pair of identical digits, the error is called a jump-twin error. Only 0.3% of all transmission errors that occur are jump-twin errors [18], making them the least common type of error compared to the error types described above. Jump-twin errors are denoted by

$$aca \leftrightarrow bcb \quad \text{with } a \neq b$$

Error type	Actual number	Number with error
Single-digit error	<u>1</u> 2345	<u>3</u> 2345
	123 <u>4</u> 5	123 <u>2</u> 5
Transposition error	52 <u>3</u> 46	52 <u>4</u> 36
	<u>1</u> 2367	<u>2</u> 1367
Twin error	24 <u>4</u> 56	21 <u>1</u> 56
	123 <u>4</u> 4	123 <u>7</u> 7
Jump-transposition error	<u>2</u> 3456	<u>2</u> 5436
	13 <u>4</u> 78	13 <u>8</u> 74
Jump-twin error	1 <u>2</u> 425	17475
	<u>1</u> 3145	<u>2</u> 3245

Table 2.1: Examples of the types of data input errors.

3

Credit card number

This chapter discusses the structure of credit card numbers. The Luhn algorithm, which is used to compute the check digit and validate a credit card number is then explained. Furthermore, a theoretical and computational analysis of the error detection in credit card numbers is carried out.

3.1. Structure

Every credit card has a unique identification number that is used for processing and authorising transactions. Furthermore, this number has a built-in error detection to reduce the risk of making mistakes when manually entering it into a system. To check for these mistakes, credit card numbers use the Luhn algorithm which will be discussed further in Subsection 3.1.1.

Depending on the issuer, a credit card number contains between 12 and 19 digits, although most modern cards issued by major payment networks have 15 or 16 digits. For readability, the digits are often grouped in blocks of four. All digits are assigned according to the international standard ISO/IEC 7812 [5].

The first six digits together form the Issuer Identification Number (IIN), also referred to as the bank identification number (BIN). In newer versions of the ISO/IEC 7812 standard, the IIN has been extended to eight digits to allow for a larger number of issuers. This number identifies the issuing financial institution and is used in transaction routing. The first digit of the IIN is known as the Major Industry Identifier (MII) and indicates which industry the identification number is associated with. As shown in Table 3.1, credit card numbers typically begin with digits ranging from 3 to 6, depending on which card processing network they are tied to. For example, American Express cards start with 3, Visa cards with 4, Mastercard cards with 2 or 5, and Discover cards with 6. [19].

Digits	Industry
1, 2	Airlines
3	Travel and entertainment
4, 5, 6	Banking and financial services
7	Petroleum
8	Health care and telecommunications
9	National assignment

Table 3.1: Industry associated with values of the Major Industry Identifier (MII) [20].

The digits following the IIN, except the last, represent the individual account number assigned by the issuing institution. This number ensures that each card issued by a specific institution is unique. The length of this number depends on the total length of the credit card number.

The final digit of the credit card number is the check digit which can be computed using the Luhn algorithm. This algorithm is designed to detect accidental typing errors such as single-digit errors and certain transposition errors, not malicious attacks and fraud.

3.1.1. Luhn algorithm

The Luhn algorithm, also known as the modulus 10 algorithm, is a method used for validating identification numbers such as credit card numbers, national ID numbers of the US, Canada and South Africa, and survey numbers of major restaurant chains [9]. It was developed by Hans Peter Luhn in 1954. Hans Peter Luhn was not a mathematician but a researcher at IBM working in the fields of computer science and information science. He developed a portable device for calculating and verifying check digits, as described in his patent *Computer for verifying numbers* [21]. The verification method introduced in this patent later became known as the Luhn algorithm.

We can formally define a credit card number of length n as a finite sequence of digits,

$$d_{n-1}d_{n-2}\dots d_2d_1d_0$$

where d_0 is the check digit and $d_i \in \{0, \dots, 9\}$ for all $i \in \{0, \dots, n-1\}$.

The Luhn algorithm is applied by processing the digits from right to left, starting from d_1 . Every digit with an odd index is multiplied by 2. Whenever doubling a digit results in a two-digit number, 9 is subtracted to make sure all values remain in $\{0, 1, \dots, 9\}$. So, for all indices $i \in \{1, \dots, n-1\}$, we define a function w in Equation 3.1 to determine the transformed digits.

$$w(d_i) = \begin{cases} 2d_i & \text{if } i \text{ is odd and } d_i \leq 4 \\ 2d_i - 9 & \text{if } i \text{ is odd and } d_i > 4 \\ d_i & \text{if } i \text{ is even} \end{cases} \quad (3.1)$$

We first define the sum s , representing the contribution of the transformed digits, by

$$s = \sum_{i=1}^{n-1} w(d_i)$$

The check digit d_0 is then computed by using the formula in Equation (3.2).

$$d_0 \equiv -s \pmod{10} \quad (3.2)$$

The check digit d_0 is therefore an integer between 0 and 9. The construction of the check digit as in Equation (3.2) ensures that the total sum of all transformed digits, including the check digit, is divisible by 10. Since the check digit is at an even position and is multiplied by 1, the validity condition of the Luhn algorithm is

$$d_0 + s = \sum_{i=0}^{n-1} w(d_i) \equiv 0 \pmod{10}$$

The Luhn algorithm detects all single-digit errors, as will be proven in Section 3.2. However, not all transposition errors and twin errors are detected. Chapter 6 considers a more general alternating weighting scheme in which the adjustment $2d_i - 9$ is replaced by simply doubling, i.e. the subtraction of 9 is omitted. In this simplified setting, all transposition and twin errors are detected. However, this improvement comes at a cost, the resulting scheme no longer detects all single-digit errors.

Example 3.1.

Suppose the first 15 digits of a VISA credit card number are known and we wish to determine the check digit using the Luhn algorithm. The first 15 characters are 4594 8513 0399 128_.

Position i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Digit	4	5	9	4	8	5	1	3	0	3	9	9	1	2	8	

All digits positioned at an odd position i should be doubled. When a digit value is greater than 9, we subtract 9.

Position i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Digit	4	5	9	4	8	5	1	3	0	3	9	9	1	2	8	
		·2		·2		·2		·2		·2		·2		·2		·2
Product	8	5	18	4	16	5	2	3	0	3	18	9	2	2	16	
Result	8	5	9	4	7	5	2	3	0	3	9	9	2	2	7	

The sum of all resulting digits is 75. So the check digit is equal to

$$-75 \bmod 10 = 5$$

Therefore, the only valid credit card number starting with the digits given in this example is 4594 8513 0399 1285.

3.2. Analysis of error detection

The primary purpose of the Luhn algorithm is to detect accidental input errors. This section starts by presenting a theoretical analysis of the different error types. To complement the theoretical results, a computational experiment is conducted to verify which errors are detected in practice.

The computational analysis was carried out using a Python implementation of the Luhn algorithm. This implementation can be found in Appendix A. The simulations begin by generating valid numbers of fixed length 16 digits, corresponding to the standard length of many credit card numbers. To construct such a number, the first 15 digits are chosen independently and randomly from $\{0, \dots, 9\}$. The final digit is then computed using the Luhn checksum formula, ensuring that every generated number is a valid number according to the Luhn algorithm.

For each simulated valid number, different error patterns are introduced, including single-digit errors, transposition errors, twin errors, and jump-transposition errors. In each error simulation, the position at which the error occurs is chosen randomly. The error can therefore also occur at the place of the check digit. Whenever a digit is replaced, the new value is chosen from the remaining possible digits, ensuring that the new digit is different from the original one. For example, in the simulation of the single-digit error, a digit d_k is replaced by

$$(d_k + r) \bmod 10 \quad (3.3)$$

where r is a randomly chosen integer in $\{1, \dots, 9\}$.

The simulations are repeated multiple times in order to estimate the detection rate for each type of error. In each simulation, a valid number was generated and modified by introducing a specific error. The resulting, new number is then tested using the Luhn algorithm. If this new number is no longer valid, the error is considered detected, otherwise it is considered undetected. The detection rate of a specific error is then computed as

$$\text{Detection rate} = \frac{\text{number of detected errors}}{\text{total number of simulated errors}} \quad (3.4)$$

In addition to the overall detection rates, the simulations also keep track of which specific digit combinations resulted in detected or undetected errors. Because of this, it is possible to compare computational results with theoretical characteristics proven in the theorems.

3.2.1. Single-digit error

The most common type of mistake is a single-digit error. This error occurs when one number in a credit card number is entered incorrectly. Theorem 3.1 shows that the Luhn algorithm can always detect such errors.

Theorem 3.1. *Every single-digit error is detected by the Luhn algorithm.*

Proof. Assume $d_{n-1} \dots d_1 d_0$ is a valid number sequence according to the Luhn algorithm. Then, $d_0 + s = \sum_{i=0}^{n-1} w(d_i) \equiv 0 \pmod{10}$.

Suppose a mistake is made at the k^{th} digit for some $k \in \{0, \dots, n-1\}$. The new value at this position is $e_k \neq d_k$. This new credit card number is valid if and only if

$$\sum_{\substack{i=0 \\ i \neq k}}^{n-1} w(d_i) + w(e_k) \equiv 0 \pmod{10}.$$

We consider 3 cases for the positions and values a digit with an error can take.

Case 1: k is even. Then $w(d_k) = d_k$ and $w(e_k) = e_k$, so the number is valid if and only if $\sum_{\substack{i=0 \\ i \neq k}}^{n-1} w(d_i) + e_k \equiv 0$

(mod 10). This is the same as

$$\begin{aligned} \sum_{i=0}^{n-1} w(d_i) + e_k - d_k &\equiv 0 \pmod{10} &\Leftrightarrow e_k - d_k &\equiv 0 \pmod{10} \\ &&\Leftrightarrow e_k &\equiv d_k \pmod{10} \\ &&\Leftrightarrow e_k &= d_k \end{aligned}$$

since e_k and d_k are both integers between 0 and 9. This contradicts the assumption that $e_k \neq d_k$. Hence, the error is detected.

Case 2: k is odd and $e_k \leq 4$. If $d_k > 4$, then the check condition of the algorithm is

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k}}^{n-1} w(d_i) + 2e_k &\equiv 0 \pmod{10} &\Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - (2d_k - 9) + 2e_k &\equiv 0 \pmod{10} \\ &&\Leftrightarrow 2(e_k - d_k) + 9 &\equiv 0 \pmod{10} \end{aligned}$$

However, this cannot hold for any e_k since the left side of the equation is always odd while the right side, which is a multiple of 10, is always even. So d_k must be smaller or equal to 4. Then, the check condition is

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k}}^{n-1} w(d_i) + 2e_k &\equiv 0 \pmod{10} &\Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - 2d_k + 2e_k &\equiv 0 \pmod{10} \\ &&\Leftrightarrow 2e_k &\equiv 2d_k \pmod{10} \\ &&\Leftrightarrow e_k &\equiv d_k \pmod{5} \\ &&\Leftrightarrow e_k &= d_k \end{aligned}$$

since $\gcd(2, 10) = 2$, and e_k and d_k are integers smaller than or equal to 4. This leads to a contradiction, so the error is again detected.

Case 3: k is odd and $e_k > 4$. If $d_k \leq 4$, then the number sequence is valid if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k}}^{n-1} w(d_i) + (2e_k - 9) &\equiv 0 \pmod{10} &\Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - 2d_k + 2e_k - 9 &\equiv 0 \pmod{10} \\ &&\Leftrightarrow 2(e_k - d_k) - 9 &\equiv 0 \pmod{10} \end{aligned}$$

This congruence cannot hold, since the left-hand side is odd while every multiple of 10 is even. Therefore, d_k must be larger than 4. In this case the check condition is

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k}}^{n-1} w(d_i) + (2e_k - 9) &\equiv 0 \pmod{10} &\Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - (2d_k - 9) + 2e_k - 9 &\equiv 0 \pmod{10} \\ &&\Leftrightarrow 2e_k &\equiv 2d_k \pmod{10} \\ &&\Leftrightarrow e_k &\equiv d_k \pmod{5} \\ &&\Leftrightarrow e_k &= d_k \end{aligned}$$

This also contradicts the assumption that $e_k \neq d_k$, so all single-digit errors are detected by the Luhn algorithm. $\square$

The computational results confirm the theoretical result of Theorem 3.1: all single-digit errors are detected. The simulation results are shown in Table 3.2.

Number of simulated errors	Total detected	Detection rate
100	100	100%
10000	10000	100%

Table 3.2: Simulation results for single-digit error detection using the Luhn algorithm.

3.2.2. Transposition error

Another common typing mistake occurs when two adjacent digits are swapped, which is known as a transposition error. The Luhn algorithm can detect almost all of these type of errors, with one exception, $90 \leftrightarrow 09$. This is proved in Theorem 3.2.

Theorem 3.2. *The Luhn algorithm detects all transposition errors $ab \leftrightarrow ba$ with $a \neq b$, except $09 \leftrightarrow 90$.*

Proof. Assume that according to the Luhn algorithm, $d_n \dots d_1 d_0$ is a valid number. Then, it must satisfy $\sum_{i=0}^{n-1} w(d_i) \equiv 0 \pmod{10}$. Now suppose that the transposition error occurs at the k^{th} and $(k-1)^{\text{th}}$ digit of the number for some $k \in \{1, \dots, n-1\}$, so $d_k d_{k-1} \leftrightarrow d_{k-1} d_k$ with $d_k \neq d_{k-1}$.

When computing the checksum s in the algorithm, either d_k or d_{k-1} is doubled. Without loss of generality, assume that k is even. We now consider four cases covering all possible values d_k and d_{k-1} can take.

Case 1: $d_k \leq 4$ and $d_{k-1} \leq 4$. The transposition error is not detected by the algorithm if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k-1}}^{n-1} w(d_i) + 2d_k + d_{k-1} &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - d_k - 2d_{k-1} + 2d_k + d_{k-1} \equiv 0 \pmod{10} \\ &\Leftrightarrow d_k \equiv d_{k-1} \pmod{10} \\ &\Leftrightarrow d_k = d_{k-1} \end{aligned}$$

However, this contradicts the assumption that $d_k \neq d_{k-1}$, so the error is detected.

Case 2: $d_k \leq 4$ and $d_{k-1} > 4$. The resulting number is valid if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k-1}}^{n-1} w(d_i) + 2d_k + d_{k-1} &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - d_k - (2d_{k-1} - 9) + 2d_k + d_{k-1} \equiv 0 \pmod{10} \\ &\Leftrightarrow d_k - d_{k-1} + 9 \equiv 0 \pmod{10} \\ &\Leftrightarrow d_k - d_{k-1} \equiv -9 \pmod{10} \\ &\Leftrightarrow d_k = 0 \text{ and } d_{k-1} = 9 \end{aligned}$$

Hence, in all such cases the error is detected.

Case 3: $d_k > 4$ and $d_{k-1} \leq 4$. Then, the number can only be valid if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k-1}}^{n-1} w(d_i) + 2d_k - 9 + d_{k-1} &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - d_k - 2d_{k-1} + 2d_k - 9 + d_{k-1} \equiv 0 \pmod{10} \\ &\Leftrightarrow d_k - d_{k-1} \equiv 9 \pmod{10} \\ &\Leftrightarrow d_k = 9 \text{ and } d_{k-1} = 0 \end{aligned}$$

This corresponds exactly to the fact that the transposition error $09 \leftrightarrow 90$ cannot be detected by the Luhn algorithm. Therefore, every other transposition error in this specific case is detected.

Case 4: $d_k > 4$ and $d_{k-1} > 4$. The error goes undetected by the Luhn algorithm if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k-1}}^{n-1} w(d_i) + 2d_k - 9 + d_{k-1} &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - d_k - (2d_{k-1} - 9) + 2d_k - 9 + d_{k-1} \equiv 0 \pmod{10} \\ &\Leftrightarrow d_k - d_{k-1} \equiv 0 \pmod{10} \\ &\Leftrightarrow d_k \equiv d_{k-1} \pmod{10} \\ &\Leftrightarrow d_k = d_{k-1} \end{aligned}$$

This contradicts the initial condition that $d_k \neq d_{k-1}$ and therefore the error is detected.

Thus, the Luhn algorithm detects all transposition errors, $ab \leftrightarrow ba$ with $a \neq b$, except $09 \leftrightarrow 90$. $\square$

A computational analysis shows that approximately 97% of all possible transposition errors are detected. The results are shown in Table 3.3.

Number of simulated errors	Total detected	Detection rate
1000	981	98.1%
5000	4880	97.6%
10000	9787	97.87%

Table 3.3: Simulation results for transposition error detection using the Luhn algorithm.

These results are consistent with the theoretical expectation. There are $10 \cdot 9 = 90$ ordered pairs (a, b) with $a \neq b$, corresponding to all possible adjacent transposition errors $ab \leftrightarrow ba$. From Theorem 3.2 we know that exactly two cases, 09 and 90, are not detected by the Luhn algorithm. Therefore, the probability that a randomly chosen transposition error is detected is

$$\frac{90 - 2}{90} = \frac{44}{45} \approx 0.9778$$

Since the transposition errors in the Python implementation of Appendix A are sampled randomly, the observed detection rate should converge to approximately 97.78% as the number of simulations increases. Since all detection rates in Table 3.4 are very close to 97.78% and the error gets smaller when the number of simulations increases, they confirm that almost all transposition errors are detected. To identify individual cases, we further compute the detection rate for each pair $ab \leftrightarrow ba$. The results are displayed in Table 3.4. All transposition errors are detected except $09 \leftrightarrow 90$. Example 3.2 confirms that $09 \leftrightarrow 90$ is not detected by the Luhn algorithm.

$\begin{matrix} b \\ a \end{matrix}$	0	1	2	3	4	5	6	7	8	9
0	-	1	1	1	1	1	1	1	1	0
1	1	-	1	1	1	1	1	1	1	1
2	1	1	-	1	1	1	1	1	1	1
3	1	1	1	-	1	1	1	1	1	1
4	1	1	1	1	-	1	1	1	1	1
5	1	1	1	1	1	-	1	1	1	1
6	1	1	1	1	1	1	-	1	1	1
7	1	1	1	1	1	1	1	-	1	1
8	1	1	1	1	1	1	1	1	-	1
9	0	1	1	1	1	1	1	1	1	-

Table 3.4: Detection rates of transposition errors $ab \leftrightarrow ba$.

Example 3.2.

The transposition error $09 \leftrightarrow 90$ goes undetected by the Luhn algorithm.

Consider the number 11908, where 8 is the check digit. The checksum $s = \sum_{i=1}^{n-1} w(d_i)$ is then equal to 12. The check digit is computed as

$$d_0 = -12 \bmod 10 = 8$$

and therefore the number 11908 is valid. Transposing the part $90 \leftrightarrow 09$, gives the number 11098. The checksum is then $s = 12$. Just like the previous number, the computed check digit is 8 and therefore the number 11098 is also valid. Since both numbers, 11908 and 11098, are valid, the transposition error $09 \leftrightarrow 90$ is not detected.

3.2.3. Twin error

Now that we have seen that the Luhn algorithm detects almost all transposition errors, we can look at what other types of errors it detects. A common type of error that also consists of two adjacent digits is a twin error. It occurs when the two adjacent digits are identical, and are replaced by another pair of identical digits. The Luhn algorithm can detect almost all twin errors. However, there are a few exceptions. Theorem 3.3 characterises which twin error the algorithm can detect.

Theorem 3.3. *All twin errors $aa \leftrightarrow bb$ with $a \neq b$ are detected by the Luhn algorithm, except for*

$$22 \leftrightarrow 55 \quad 33 \leftrightarrow 66 \quad 44 \leftrightarrow 77$$

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid number. Then, the check condition is $s + d_0 = \sum_{i=0}^{n-1} w(d_i) \equiv 0 \pmod{10}$. Suppose an error is made in the k^{th} and $(k-1)^{\text{th}}$ digit of the number for some $k \in \{1, \dots, n-1\}$. Also, suppose that before an error is made, d_k and d_{k-1} both have the value $a \in \{0, \dots, 9\}$. After the error is made they have the value b with $b \neq a$. Without loss of generality, assume that d_k is the digit that is doubled in the checksum computation before an error has occurred. Now we define the auxiliary function $\tilde{w}$ as

$$\tilde{w}(d) = \begin{cases} 2d & \text{if } d \leq 4 \\ 2d - 9 & \text{if } d > 4 \end{cases}$$

A twin error goes undetected by the Luhn algorithm if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k-1}}^{n-1} w(d_i) + b + \tilde{w}(b) &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w(d_i) - a - \tilde{w}(a) + b + \tilde{w}(b) \equiv 0 \pmod{10} \\ &\Leftrightarrow a + \tilde{w}(a) \equiv b + \tilde{w}(b) \pmod{10} \end{aligned}$$

We now split the proof into three cases according to the possible ranges of a and b .

Case 1: $a \leq 4, b \leq 4$. Then $a + \tilde{w}(a) = 3a$ and $b + \tilde{w}(b) = 3b$. So, the error is not detected if and only if

$$\begin{aligned} 3a &\equiv 3b \pmod{10} \Leftrightarrow a \equiv b \pmod{10} \\ &\Leftrightarrow a = b \end{aligned}$$

This contradicts the assumption that $a \neq b$ and therefore the error is detected.

Case 2: $a > 4, b > 4$. Then we have that $a + \tilde{w}(a) = 3a - 9$ and $b + \tilde{w}(b) = 3b - 9$. So, the algorithm cannot detect the error if and only if

$$\begin{aligned} 3a - 9 &\equiv 3b - 9 \pmod{10} \Leftrightarrow 3a \equiv 3b \pmod{10} \\ &\Leftrightarrow a \equiv b \pmod{10} \\ &\Leftrightarrow a = b \end{aligned}$$

For the same reason as in case 1 we know that the error is detected.

Case 3: $a \leq 4, b > 4$. The case where $a > 4$ and $b \leq 4$ can be treated similarly and leads to the same exceptional twin errors. If $a \leq 4$ and $b > 4$, then $a + \tilde{w}(a) = 3a$ and $b + \tilde{w}(b) = 3b - 9$. The error goes undetected if and only if

$$\begin{aligned} 3a &\equiv 3b - 9 \pmod{10} \Leftrightarrow a \equiv b - 3 \pmod{10} \\ &\Leftrightarrow a - b \equiv -3 \pmod{10} \\ &\Leftrightarrow (a = 2 \text{ and } b = 5) \text{ or } (a = 3 \text{ and } b = 6) \text{ or } (a = 4 \text{ and } b = 7) \end{aligned}$$

Hence, all twin errors in this case are detected, except for $22 \leftrightarrow 55$, $33 \leftrightarrow 66$, and $44 \leftrightarrow 77$. □

The results of the computational analysis of twin error in the Luhn algorithm are summarised in Table 3.5. The simulations were repeated for increasing numbers of randomly generated valid numbers to check whether the detection rate stays consistent as the sample size grows.

Number of simulated errors	Total detected	Detection rate
800	744	93%
3980	3724	93.57%
7990	7446	93.19%
95231	88871	93.32%
1032344	963492	93.33%

Table 3.5: Simulation results for twin error detection using the Luhn algorithm.

The results show that the detection rate converges to approximately 93.33% as the number of simulations increases. To understand this behaviour, we look at the probability that a twin error is detected. There are exactly $10 \cdot 9 = 90$ possible twin substitutions $aa \rightarrow bb$. From Theorem 3.3 we know that the six cases, $22 \rightarrow 55$, $55 \rightarrow 22$, $33 \rightarrow 66$, $66 \rightarrow 33$, $44 \rightarrow 77$, and $77 \rightarrow 44$, are not detected by the Luhn algorithm. Therefore, the detection probability is

$$\frac{90 - 6}{90} = \frac{14}{15} \approx 0.9333$$

This theoretical value matches the observed computational results closely, which support the result in Theorem 3.3. Now, we compute the detection rate for each pair $aa \leftrightarrow bb$ to identify individual cases. The results of this are shown in Table 3.6. The table confirms that almost all twin errors are detected, with only a few exceptions. In particular, the pairs $22 \leftrightarrow 55$, $33 \leftrightarrow 66$, and $44 \leftrightarrow 77$ are not detected by the Luhn algorithm. Example 3.3 shows examples of this.

$\begin{matrix} b \\ a \end{matrix}$	0	1	2	3	4	5	6	7	8	9
0	-	1	1	1	1	1	1	1	1	1
1	1	-	1	1	1	1	1	1	1	1
2	1	1	-	1	1	0	1	1	1	1
3	1	1	1	-	1	1	0	1	1	1
4	1	1	1	1	-	1	1	0	1	1
5	1	1	0	1	1	-	1	1	1	1
6	1	1	1	0	1	1	-	1	1	1
7	1	1	1	1	0	1	1	-	1	1
8	1	1	1	1	1	1	1	1	-	1
9	1	1	1	1	1	1	1	1	1	-

Table 3.6: Detection rates of twin errors $aa \leftrightarrow bb$.

Example 3.3.

The twin error $22 \leftrightarrow 55$ cannot be detected by the Luhn algorithm. Take a valid number 12211. Then, the contribution of 22 to the checksum computation is

$$2 + w(2) = 2 + 2 \cdot 2 = 6$$

After the error is made, the number is 15511. Only 55 has changed, all other digits stay the same. The contribution of 55 is

$$5 + w(5) = 5 + (2 \cdot 5 - 9) = 6$$

Since the contributions to the checksum are the same, both before and after the error is made, the number is valid. Therefore, the number is not detected.

The same holds for $33 \leftrightarrow 66$. The contribution of 33 in the valid number 13318 is $3 + w(3) = 3 + 2 \cdot 3 = 9$. And the contribution of the digits 66 in the checksum computation of 16618 is $6 + w(6) = 6 + (2 \cdot 6 - 9) = 9$. Since, the contribution of the digits is the same, the twin error $33 \leftrightarrow 66$ is not detected.

Lastly, the twin error $44 \leftrightarrow 77$ is also not detected. The digits 44 contribute $4 + w(4) = 4 + 2 \cdot 4 = 12$ to the checksum s in the valid number 14415. The contribution of the digits 77 in 17715 is $7 + w(7) = 7 + (2 \cdot 7 - 9) = 12$. So, again the error is not detected.

3.2.4. Jump-transposition error

A jump-transposition error is one of the most dangerous type of errors for the Luhn algorithm. Since the two digits in a jump-transposition error are separated by a third digit, they are multiplied by the same weight. Therefore, the algorithm cannot detect any jump-transposition errors. A more rigorous proof is provided in Theorem 3.4.

Theorem 3.4. *The Luhn algorithm cannot detect any jump-transposition errors $abc \leftrightarrow cba$ with $a \neq c$.*

Proof. Assume $d_{n-1} \dots d_1 d_0$ is a valid number when checked by the Luhn algorithm, so $\sum_{i=0}^{n-1} w(d_i) \equiv 0 \pmod{10}$. Suppose a jump-transposition error is made in the digits d_k and d_{k+2} for some $k \in \{0, 1, \dots, n-3\}$. So they are swapped, $d_{k+2}d_{k+1}d_k \rightarrow d_kd_{k+1}d_{k+2}$. Since k and $k+2$ have the same parity, the digits d_k and d_{k+2} are treated in the same way by the Luhn algorithm. They are either both doubled or both stay the same. We define the function $\tilde{w}$ that represents what happens when a digit is doubled.

$$\tilde{w}(d) = \begin{cases} 2d & \text{if } d \leq 4 \\ 2d - 9 & \text{if } d > 4 \end{cases}$$

Now we consider two cases, one in which the digits are doubled and the other in which the digits stay the same.

Case 1: d_k and d_{k+2} are not doubled. Then the contribution of these digits before an error is made is $d_k + d_{k+2}$. After the error is made the contribution to the computation of the checksum is $d_{k+2} + d_k$. Since the contribution before and after the error is made stays the same, the error cannot be detected.

Case 2: d_k and d_{k+2} are both doubled. This case is very similar to Case 1. The contribution of the digits d_k and d_{k+2} to the computation of the checksum s before an error is made is $\tilde{w}(d_k) + \tilde{w}(d_{k+2})$. After the error is made, the contribution is still the same, $\tilde{w}(d_{k+2}) + \tilde{w}(d_k)$. Again, since the contribution before an error is made is the same as the contribution after the error, the jump-transposition error is not detected.

So, the Luhn algorithm cannot detect a jump-transposition error when the digits d_k and d_{k+2} are doubled or stay the same. Hence, the algorithm cannot detect any jump-transposition error. $\square$

4

IBAN

In this chapter, the structure of IBANs is discussed as well as the algorithm that is used to compute the check digits of these identification numbers. Furthermore, an analysis of the error detection properties of this algorithm is performed. At first, with a theoretical analysis. These results are then validated using a computational analysis.

4.1. Structure

An International Bank Account Number (IBAN) is a standardised international identifier used to uniquely identify bank accounts across national borders. It was introduced to facilitate international money transfers and to reduce errors in manual data entry. IBAN has been implemented by most European countries, as well as in other regions, particularly the Middle East and the Caribbean. Currently, the IBAN numbering system is used in 89 countries [22].

Before the IBAN was introduced, national bank account numbers differed significantly between countries. This lack of standardisation made it difficult to process cross-border payments efficiently. As a result, costs and the time required to complete transactions increased, and there was a higher likelihood of input errors. The IBAN system was developed in the 1990s and was formally introduced by the European Committee for Banking Standards (ECBS) in 1997 [23]. It was later adapted as the international standard ISO 13616 [6] in 2003 and further revised in subsequent years. The structure of the IBAN is defined by this international standard.

The IBAN consists of up to 34 alphanumeric characters, i.e. letters and digits. Each country uses a fixed number of characters, for example, in The Netherlands IBANs are 18 characters long while in Italy they are 27 characters long [22].

The structure of an IBAN is made up of three parts: a two-letter country code, two check digits, and a Basic Bank Account Number (BBAN).

The country code consists of two capital letters and is determined using ISO 3166-1 alpha-2 [24]. This code is unique for every country and as an example, the Netherlands has country code NL. The next two characters are check digits, which are computed using the ISO 7064, MOD 97-10 algorithm. This algorithm is applied to the entire IBAN and verifies its validity by checking whether the resulting number is congruent to 1 modulo 97. Further explanation of the algorithm is given in Subsection 4.1.1. The remaining characters form the Basic Bank Account Number (BBAN). The BBAN contains country-specific banking information, such as a bank identifier and an account number, and may consist of both letters and digits. Its structure and length depend on the country, with a maximum length of 30 characters [25]. For example, in the Netherlands the BBAN consists of 14 characters: the first four are letters identifying the bank institution, while the remaining ten are digits identifying the account number.

4.1.1. MOD 97-10 algorithm

The MOD 97-10 algorithm is a check-digit system defined by ISO 7064, which was first published in 1983 and later revised in 2003 [13]. It operates on numeric strings and uses two check digits. As indicated by its name, the algorithm is based on modular arithmetic modulo 97 in the decimal (radix-10) system. The algorithm is specifically designed to detect common input errors, including all single-digit errors, transposition errors, and twin errors.

Before explaining the MOD 97-10 algorithm, we first describe the structure of an IBAN and introduce the notation that will be used throughout this section. The structure is given in Definition 4.1.

Definition 4.1. An IBAN of length m is written as

$$x_1 x_2 x_3 x_4 x_5 \dots x_m,$$

where $x_1 x_2$ is the country code, $x_3 x_4$ are the check digits, and $x_5 \dots x_m$ is the BBAN. Hence, $x_1, x_2 \in \{A, \dots, Z\}$, $x_3, x_4 \in \{0, \dots, 9\}$, and each character x_i with $i \geq 5$ is either a letter or a digit.

In order to verify an IBAN, it first needs to be rearranged. The first four characters, the country code and the two check digits, need to be moved to the end of the IBAN. Since the MOD 97-10 algorithm operates on numeric strings, each letter is replaced by its corresponding numerical value, where

$$A = 10, \quad B = 11, \quad \dots, \quad Z = 35.$$

This new rearranged IBAN will be referred to as the converted IBAN and is defined in Definition 4.2.

Definition 4.2. The converted IBAN is represented by

$$a_n a_{n-1} \dots a_3 a_2 a_1$$

where n is the length of the string and $a_i \in \{0, \dots, 9\}$ for all $i \in \{1, \dots, n\}$. Furthermore, a_1 and a_2 are the check digits of the string.

The converted IBAN is at least as long as the original IBAN, since each alphabetic character is replaced by a two-digit number.

Now, we can apply the MOD97-10 algorithm. To check if the IBAN is valid, the converted IBAN is interpreted as a single large integer, which must satisfy

$$\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97} \quad (4.1)$$

There are several methods for computing the check digits. A common approach begins by temporarily replacing the check digits with 00. The resulting IBAN is then processed in the same way as during validation: the first four characters are moved to the end, and all letters are replaced by their corresponding numerical values. According to ISO 7064, the check number must lie in the range 02 to 98. Since Equation (4.1) must hold, the correct check digits are computed by

$$98 - \left(\sum_{i=1}^n a_i \cdot 10^{i-1} \pmod{97} \right)$$

If the resulting value is a single-digit number in $\{2, \dots, 9\}$, a 0 is added in front to ensure the check number consists of two digits.

From a computational perspective, directly applying the above formula is impractical, since an IBAN typically contains around 20 characters. After conversion, the resulting numeric string may contain even more digits, resulting in an integer that is too large to efficiently perform calculations on directly. Therefore, we describe two computational methods that account for this, the recursive method, and the polynomial method. These methods both yield the same result.

Recursive

The recursive method provides an efficient way to compute the remainder obtained when the converted IBAN is divided by 97. Instead of evaluating the entire converted IBAN as a single integer modulo 97, its digits are processed one at a time. Define the sequence $(P_j)_{j \in \{1, \dots, n+1\}}$ recursively by

$$P_{j+1} = (10 \cdot P_j + a_{n-j+1}) \bmod 97, \quad P_1 = 0$$

After processing all digits, the value P_{n+1} equals the remainder of the converted IBAN upon division by 97. Consequently, an IBAN is valid if and only if its converted IBAN satisfies

$$P_{n+1} = 1 \tag{4.2}$$

In order to determine the check digits of an IBAN, the recursion described above is applied to the converted IBAN with check digits equal to 00. First, the resulting remainder P_{n+1} is computed, then the check digits are computed by

$$98 - P_{n+1}$$

Whenever the resulting value lies in $(2, \dots, 9)$, a leading zero is added to ensure that the check number consists of two digits.

Example 4.1.

We want to compute the check digits of a Dutch IBAN using the recursive method. The IBAN is

NL__ RABO 9258 8019 03

Now we set the check digits to 00, move the first four characters to the back of the string, and convert all letter to their numerical values. RABO will become 27 10 11 24 and NL will be converted to 23 21. The new string then is

271011249258801903232100

We compute P_j iteratively for $j = 1, \dots, 25$ to obtain the final remainder.

Steps j	P_j	$\cdot$	10	+	a_{n-j+1}	=	x	$x \bmod 97 = P_{j+1}$
1	0	$\cdot$	10	+	2	=	2	2
2	2	$\cdot$	10	+	7	=	27	27
3	27	$\cdot$	10	+	1	=	271	77
4	77	$\cdot$	10	+	0	=	770	91
$\vdots$	$\vdots$		$\vdots$		$\vdots$		$\vdots$	$\vdots$
22	16	$\cdot$	10	+	1	=	161	64
23	64	$\cdot$	10	+	0	=	640	58
24	58	$\cdot$	10	+	0	=	580	95

We can now compute the check digits of the IBAN by

$$98 - P_{25} = 98 - 95 = 3$$

Therefore, the check digits must be 03. So, the following IBAN is valid.

NL03 RABO 9258 8019 03

Polynomial

The polynomial method provides an alternative way to compute the remainder of the converted IBAN modulo 97. Instead of processing the digits recursively, the converted IBAN is interpreted as a weighted sum of its digits, where each digit is multiplied by a power of 10 modulo 97 determined by its position in the string.

Let the converted IBAN be given by $a_n a_{n-1} \dots a_2 a_1$, where $a_2 = a_1 = 0$ when computing the check digits. For each digit a_i , define the corresponding weight by

$$w_i = 10^{i-1} \mod 97$$

The modulo operation ensures that the weights remain relatively small. The weighted sum is then given by

$$N = \sum_{i=1}^n w_i \cdot a_i$$

An IBAN is valid if and only if the weighted sum satisfies $N \equiv 1 \pmod{97}$. The check digits can therefore be computed using the formula in Equation (4.3). If the resulting value consists of a single digit, a leading zero is added, as described previously with the recursive method. This ensures that the check digits lie in the set $\{02, 03, \dots, 98\}$.

$$98 - (N \mod 97) \tag{4.3}$$

Example 4.2.

This example will use the same IBAN as in Example 4.1.

NL__ RABO 9258 8019 03

First, we set the check digits to 00 and move the first four characters to the back of the string. Then, all non-numerical values will be converted to their respective numerical value. Therefore, we get the string

271011249258801903232100

Now we will determine the weight for all positions of the string using $w_i = 10^{i-1} \mod 97$.

Position i	24	23	22	21	20	...	10	9	8	7	6	5	4	3	2	1
Digit a_i	2	7	1	0	1	...	1	9	0	3	2	3	2	1	0	0
Weight w_i	56	25	51	73	17	...	34	81	76	27	90	9	30	3	10	1
Product	112	175	51	0	17	...	34	729	0	81	180	27	60	3	0	0

The sum of all the products is 2908. We can use this to compute the check digits.

$$98 - (2908 \mod 97) = 98 - 95 = 3$$

Therefore, the check digits are 03, which are the same as in Example 4.1. So, the valid IBAN is indeed

NL03 RABO 9258 8019 03

4.2. Analysis of error detection

This section examines the error detection capabilities of the MOD 97-10 algorithm. Since the algorithm operates on converted IBANs, theoretical results describing which types of errors can and cannot be detected are first presented. These results are then used to analyse the error detection performance of IBANs. To achieve this, the error detection properties of the MOD 97-10 algorithm are first proven mathematically. A computational experiment in Python is then conducted to evaluate the algorithm's performance under more realistic conditions and to verify the analytical results.

The code of the Python implementation can be found in Appendix B. While the theoretical analysis considers IBANs in their general form and is therefore independent of any country-specific format, the computational analysis focuses on Dutch IBANs.

Each simulation begins with generating a valid Dutch IBAN, starting with NL, two check digits, four random letters representing the bank institution, and finally 10 digits for an account number. The check digits are computed using the MOD97-10 algorithm. The generated IBAN is then converted into its corresponding converted IBAN representation by moving the first four characters to the end and replacing each letter by its

numerical value. Several classes of errors are subsequently introduced to investigate the error-detection capabilities of the MOD97-10 checksum system. The considered error types for the analysis of single-character errors in an IBAN are single-digit errors, transposition errors, twin errors, and two-digit substitutions in the converted IBAN, and single-character errors applied to the original IBAN before conversion. Error types involving three or four consecutive digits in the converted IBAN are used in the analysis of transposition errors in IBANs. For each error type, the position at which the error occurs is determined randomly. Whenever a digit or letter is modified, the new value is chosen such that it is different from the original value.

For the single-digit experiment, one digit of the converted IBAN is altered while all remaining digits are left unchanged. In the transposition experiment, two neighbouring digits are interchanged, provided that they are distinct. The twin-error experiment replaces a pair of identical adjacent digits by another pair of identical digits. In the two-digit substitution experiment, both digits of a neighbouring pair are modified independently. Finally, the single-character experiment changes a single character of the original IBAN, where digits are replaced by different digits and letters by different letters. The error types used in the analysis of transposition errors in IBANs are described in Subsection 4.2.2.

In order to determine the detection rate of each type of error, the simulations are repeated multiple times. This detection rate is computed by

$$\text{Detection rate} = \frac{\text{number of detected errors}}{\text{total number of simulated errors}} \quad (4.4)$$

Besides the overall detection rates, the simulations also keep track of the detection rates of specific digit and character combinations. This is especially done for two-digit substitution errors. The computational results are then compared with the theoretical results.

4.2.1. Single-character error

A single-character error in an IBAN occurs when a letter is replaced by another letter, a digit is replaced by a different digit (single-digit error), or when a letter is replaced by a digit or vice versa. To determine which errors are detected in an IBAN, we first look at the error detection in converted IBANs using the MOD97-10 algorithm. These results are then used to determine which errors are detected in an IBAN. More specifically, Theorems 4.3 and 4.6, together with Lemmas 4.4 and 4.5, describe error detection in converted IBANs. Corollary 4.7, together with Theorems 4.8 and 4.9, then translates these results to the IBAN level.

But first we will look at which single-digit errors in a converted IBAN are detected by the MOD97-10 algorithm.

Theorem 4.3. *The MOD97-10 algorithm can detect all single-digit errors in a converted IBAN.*

Proof. Suppose that $a_n \dots a_2 a_1$ is a valid converted IBAN when the MOD97-10 algorithm is applied. Then, the following validity condition holds,

$$\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97}$$

Suppose an error is made at the k^{th} digit. Then $a_k \rightarrow a'_k$ with $a_k \neq a'_k$. The error goes undetected by the MOD97-10 algorithm if and only if

$$\begin{aligned} \sum_{\substack{i=1 \\ i \neq k}}^n a_i \cdot 10^{i-1} + a'_k \cdot 10^{k-1} &\equiv 1 \pmod{97} \Leftrightarrow \sum_{i=1}^n a_i \cdot 10^{i-1} - a_k \cdot 10^{k-1} + a'_k \cdot 10^{k-1} \equiv 1 \pmod{97} \\ &\Leftrightarrow -a_k \cdot 10^{k-1} + a'_k \cdot 10^{k-1} \equiv 0 \pmod{97} \\ &\Leftrightarrow (a'_k - a_k) \cdot 10^{k-1} \equiv 0 \pmod{97} \end{aligned}$$

Since 97 is a prime number and no power of 10 is divisible by 97, we have that $\gcd(97, 10^{k-1}) = 1$ for all $k \in \{1, \dots, n\}$. Therefore,

$$\begin{aligned} (a'_k - a_k) \cdot 10^{k-1} &\equiv 0 \pmod{97} \Leftrightarrow a'_k - a_k \equiv 0 \pmod{97} \\ &\Leftrightarrow a'_k \equiv a_k \pmod{97} \\ &\Leftrightarrow a'_k = a_k \end{aligned}$$

This contradicts the assumption that $a_k \neq a'_k$, and therefore all single-digit errors in a converted IBAN are detected. $\square$

This is also verified by the computational analysis. All single-digit errors in a converted IBAN can be detected by the MOD97-10 algorithm. No matter how many errors are simulated, this is always the case. The results of the computational analysis for single-digit errors in converted IBANs are displayed in Table 4.1.

Number of simulated errors	Total detected	Detection rate
100	100	100%
1000	1000	100%
10000	10000	100%

Table 4.1: Simulation results for detecting single-digit errors in converted IBANs using the MOD97-10 algorithm.

Since letters in an IBAN are converted into two numerical digits, analysing letter substitutions requires understanding how the MOD97-10 algorithm detects errors involving two adjacent digits in the converted IBAN. We therefore first study three classes of adjacent-digit errors in converted IBANs: transposition errors, twin errors, and general two-digit substitution errors. These results will later be used to analyse single-character substitutions in the original IBAN. The first type of error to be considered is the transposition error in a converted IBAN. The error detection capabilities for this error are stated and proven in Lemma 4.4.

Lemma 4.4. *Every transposition error $ab \leftrightarrow ba$ with $a \neq b$ in a converted IBAN is detected by the MOD97-10 algorithm.*

Proof. Let $a_n \dots a_2 a_1$ be a valid converted IBAN. Then, it satisfies $\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97}$. Suppose that the adjacent digits at positions k and $k+1$, where $k \in \{1, \dots, n-1\}$, are interchanged. So $a_{k+1}a_k \rightarrow a_k a_{k+1}$. Then, the transposition error goes undetected by the MOD97-10 algorithm if and only if,

$$\begin{aligned}
& \sum_{\substack{i=1 \\ i \neq k, k+1}}^n a_i \cdot 10^{i-1} + a_{k+1} \cdot 10^{k-1} + a_k \cdot 10^k \equiv 1 \pmod{97} \\
& \Leftrightarrow \sum_{i=1}^n a_i \cdot 10^{i-1} - a_{k+1} \cdot 10^k - a_k \cdot 10^{k-1} + a_{k+1} \cdot 10^{k-1} + a_k \cdot 10^k \equiv 1 \pmod{97} \\
& \Leftrightarrow -a_{k+1} \cdot 10^k - a_k \cdot 10^{k-1} + a_{k+1} \cdot 10^{k-1} + a_k \cdot 10^k \equiv 0 \pmod{97} \\
& \Leftrightarrow (a_k - a_{k+1})(10^k - 10^{k-1}) \equiv 0 \pmod{97} \\
& \Leftrightarrow 9 \cdot (a_k - a_{k+1}) \cdot 10^{k-1} \equiv 0 \pmod{97}
\end{aligned}$$

Since 97 is a prime number and no power of 10 is divisible by 97, we have that $\gcd(97, 10^{k-1}) = 1$ for all $k \in \{1, \dots, n\}$. Furthermore, $\gcd(9, 97) = 1$. Therefore,

$$\begin{aligned}
9 \cdot (a_k - a_{k+1}) \cdot 10^{k-1} \equiv 0 \pmod{97} & \Leftrightarrow a_k - a_{k+1} \equiv 0 \pmod{97} \\
& \Leftrightarrow a_k \equiv a_{k+1} \pmod{97} \\
& \Leftrightarrow a_k = a_{k+1}
\end{aligned}$$

However, this contradicts the initial assumption that $a_k \neq a_{k+1}$. Thus, every transposition error $ab \leftrightarrow ba$ with $a \neq b$ in a converted IBAN is detected by the MOD97-10 algorithm. $\square$

We now consider the second type of error involving two adjacent digits, namely the twin error, in which two identical adjacent digits are replaced by another pair of identical digits. Lemma 4.5 shows that every twin error $aa \leftrightarrow bb$ with $a \neq b$ in a converted IBAN is detected by the MOD97-10 algorithm.

Lemma 4.5. *The MOD97-10 algorithm can detect all twin errors $aa \leftrightarrow bb$ with $a \neq b$ in a converted IBAN.*

Proof. Let $a_n \dots a_2 a_1$ be a valid converted IBAN. Then, the check condition of this converted IBAN is $\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97}$. Suppose that an error occurs at positions k and $k+1$ for some $k \in \{1, \dots, n-1\}$. The value of these digits before the error occurs is equal to a with $a \in \{0, \dots, 9\}$. After the error has occurred the value of a_k and a_{k+1} is equal to $b \neq a$. The twin error in the converted IBAN goes undetected by the MOD97-10 algorithm if and only if

$$\begin{aligned} \sum_{\substack{i=1 \\ i \neq k, k+1}}^n a_i \cdot 10^{i-1} + a \cdot 10^k + a \cdot 10^{k-1} &\equiv 1 \pmod{97} \\ \Leftrightarrow \sum_{i=1}^n a_i \cdot 10^{i-1} - b \cdot 10^k - b \cdot 10^{k-1} + a \cdot 10^k + a \cdot 10^{k-1} &\equiv 1 \pmod{97} \\ \Leftrightarrow -b \cdot 10^k - b \cdot 10^{k-1} + a \cdot 10^k + a \cdot 10^{k-1} &\equiv 0 \pmod{97} \\ \Leftrightarrow (a-b)(10^k + 10^{k-1}) &\equiv 0 \pmod{97} \\ \Leftrightarrow 11 \cdot (a-b) \cdot 10^{k-1} &\equiv 0 \pmod{97} \end{aligned}$$

Since $\gcd(11, 97) = 1$ and $\gcd(10^{k-1}, 97) = 1$ for all $k \in \{1, \dots, n-1\}$, we have that

$$\begin{aligned} 11 \cdot (a-b) \cdot 10^{k-1} &\equiv 0 \pmod{97} \Leftrightarrow a-b \equiv 0 \pmod{97} \\ &\Leftrightarrow a = b \end{aligned}$$

This contradicts the assumption that $a \neq b$ and therefore all twin errors in a converted IBAN are detected. $\square$

Having shown that the MOD97-10 algorithm detects all transposition errors and twin errors in a converted IBAN, we now consider the most general case of errors affecting two adjacent digits. This includes all substitutions of a two-digit block by another two-digit block. The following theorem characterises exactly which of these errors are detected by the MOD97-10 algorithm.

Theorem 4.6 shows that all such errors in a converted IBAN are detected except for $00 \leftrightarrow 97$, $01 \leftrightarrow 98$, and $02 \leftrightarrow 99$.

Theorem 4.6. *Every two-digit substitution error $ab \leftrightarrow cd$ with $10a+b \neq 10c+d$ in a converted IBAN is detected by the MOD97-10 algorithm, except for the errors*

$$00 \leftrightarrow 97, \quad 01 \leftrightarrow 98, \quad 02 \leftrightarrow 99$$

Proof. Let $a_n \dots a_2 a_1$ be a valid converted IBAN according to the MOD97-10 algorithm, then $\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97}$. Suppose that the two adjacent errors occur at positions k and $k+1$ for some $k \in \{1, \dots, n-1\}$. Before the error is made, $a_{k+1} = a$ and $a_k = b$ with $a, b \in \{0, \dots, 9\}$. After the error has occurred the values at these positions are $a_{k+1} = c$ and $a_k = d$ such that $10a+b \neq 10c+d$.

The two-digit substitution error in the converted IBAN is not detected by the MOD97-10 algorithm if and only if

$$\begin{aligned} \sum_{\substack{i=1 \\ i \neq k, k+1}}^n a_i \cdot 10^{i-1} + c \cdot 10^k + d \cdot 10^{k-1} &\equiv 1 \pmod{97} \\ \Leftrightarrow \sum_{i=1}^n a_i \cdot 10^{i-1} - a \cdot 10^k - b \cdot 10^{k-1} + c \cdot 10^k + d \cdot 10^{k-1} &\equiv 1 \pmod{97} \\ \Leftrightarrow -a \cdot 10^k - b \cdot 10^{k-1} + c \cdot 10^k + d \cdot 10^{k-1} &\equiv 0 \pmod{97} \\ \Leftrightarrow (c-a) \cdot 10^k + (d-b) \cdot 10^{k-1} &\equiv 0 \pmod{97} \\ \Leftrightarrow 10^{k-1} \cdot (10 \cdot (c-a) + (d-b)) &\equiv 0 \pmod{97} \end{aligned}$$

Because 97 is a prime number and no power of 10 is divisible by 97, $\gcd(10^{k-1}, 97) = 1$ for all $k \in \{1, \dots, n-1\}$. Therefore,

$$\begin{aligned} 10^{k-1} \cdot (10 \cdot (c-a) + (d-b)) &\equiv 0 \pmod{97} \Leftrightarrow 10 \cdot (c-a) + (d-b) \equiv 0 \pmod{97} \\ &\Leftrightarrow 10c+d \equiv 10a+b \pmod{97} \end{aligned}$$

Since $(10a + b) \in \{0, \dots, 99\}$ and $(10c + d) \in \{0, \dots, 99\}$, the two-digit substitution error in the converted IBAN cannot be detected if and only if the difference between $10a + b$ and $10c + d$ is equal to ± 97 or 0. However, the latter option contradicts the initial assumption that $10a + b \neq 10c + d$. Hence, the difference must be equal to ± 97 . This is possible if and only if

$$(a = 0, b = 0, c = 9, d = 7) \quad \text{or} \quad (a = 0, b = 1, c = 9, d = 8) \quad \text{or} \quad (a = 0, b = 2, c = 9, d = 9)$$

$$(a = 9, b = 7, c = 0, d = 0) \quad \text{or} \quad (a = 9, b = 8, c = 0, d = 1) \quad \text{or} \quad (a = 9, b = 9, c = 0, d = 2)$$

Thus, all two-digit substitution errors in a converted IBAN are detected, except for the errors $00 \leftrightarrow 97$, $01 \leftrightarrow 98$, and $02 \leftrightarrow 99$. $\square$

The findings in Theorem 4.6 are also verified by the computational analysis. The results of the computational analysis are displayed in Table 4.2. These simulations are repeated for an increasing amount of randomly generated valid converted IBANs.

Number of simulated errors	Total detected	Detection rate
1000	999	99.9%
5000	4997	99.94%
10000	9995	99.95%
150000	149938	99.96%
1000000	999567	99.96%

Table 4.2: Simulation results for two-digit substitution error detection using the MOD97-10 algorithm.

So, the detection rate approximates 99.96% as the number of simulations increases. To understand why this happens, we can look at the probability that a two-digit substitution error is detected. First, we look at how many possible combinations of (a, b, c, d) there are. The digit combinations ab and cd both represent an integer between 0 and 99. Since $10a + b \neq 10c + d$, these integers representations of ab and cd must be different. The first pair ab can represent any value in $0, \dots, 99$, giving 100 possibilities. The second pair must differ from the first, giving 99 possibilities. Thus, there are $100 \cdot 99 = 9900$ possible two-digit substitution errors $ab \leftrightarrow cd$ with $10a + b \neq 10c + d$. In only 6 cases, the error is not detected.

$$(a = 0, b = 0, c = 9, d = 7) \quad \text{or} \quad (a = 0, b = 1, c = 9, d = 8) \quad \text{or} \quad (a = 0, b = 2, c = 9, d = 9)$$

$$(a = 9, b = 7, c = 0, d = 0) \quad \text{or} \quad (a = 9, b = 8, c = 0, d = 1) \quad \text{or} \quad (a = 9, b = 9, c = 0, d = 2)$$

Therefore, the probability that an error of the form $ab \leftrightarrow cd$ with $10a + b \neq 10c + d$ in a converted IBAN is detected is

$$\frac{9900 - 6}{9900} = \frac{1649}{1650} \approx 0.9994$$

This is very close to the result found in Table 4.2. The theoretical model assumes all possible two-digit substitution errors occur with equal probability. However, in the simulation, errors are generated through a specific random digit replacement process, which does not sample all error patterns uniformly. As a result, some of the rare undetected cases appear less often than expected in the theoretical model, which leads to a slightly higher observed detection rate than the theoretical value. Now, we can compute the detection rate of all possible cases $ab \leftrightarrow cd$ separately. The results of this are displayed in a heatmap in Figure 4.1.

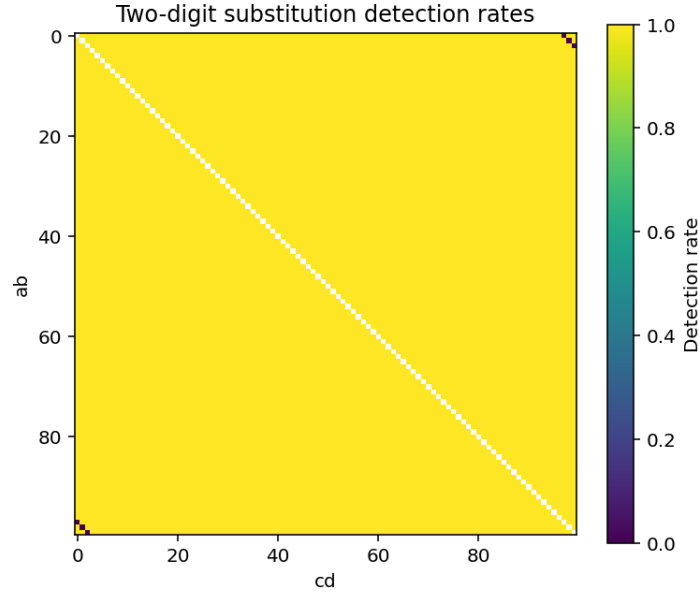


Figure 4.1: Heatmap displaying the error detection rates of two-digit substitution errors under the MOD97-10 algorithm.

To better understand the patterns in the upper-right and lower-left corners of the heatmap, Table 4.3 focuses on these regions separately. It confirms that all two-digit substitution errors in a converted IBAN are detected, except for the cases $00 \leftrightarrow 97$, $01 \leftrightarrow 98$, and $02 \leftrightarrow 99$.

ab	cd	00	01	02	03	04
95		1	1	1	1	1
96		1	1	1	1	1
97		0	1	1	1	1
98		1	0	1	1	1
99		1	1	0	1	1

ab	cd	95	96	97	98	99
00		1	1	0	1	1
01		1	1	1	0	1
02		1	1	1	1	0
03		1	1	1	1	1
04		1	1	1	1	1

Table 4.3: Results of heatmap in Figure 4.1, lower-left corner (left) and upper-right corner (right).

Using the results derived for converted IBANs under the MOD97-10 algorithm, we can now translate these results to error detection in IBANs. In particular, we obtain conditions for which single-character substitutions are detected in an IBAN. An IBAN can detect all digit-to-digit and all letter-to-letter substitutions. This is formalised in Corollary 4.7, which is a direct consequence of the results for converted IBANs.

Corollary 4.7. *An IBAN can detect all digit-to-digit and letter-to-letter substitutions.*

Proof. This proof is divided into two cases, when the error occurs with a digit and when it occurs with a letter.

Case 1: digit substitution

When a digit is replaced by another digit in an IBAN, the converted IBAN after an error has occurred differs in exactly one digit position from the original converted IBAN. Therefore, a digit error in an IBAN corresponds to a single-digit error in its converted IBAN. From Theorem 4.3 we know that all single-digit errors in a converted IBAN are detected by the MOD97-10 algorithm. So, every digit-to-digit error in an IBAN is detected.

Case 2: letter substitution

When an IBAN is rearranged into its converted IBAN, all letters are converted using

$$A = 10, \quad B = 11, \quad \dots, \quad Z = 35.$$

Therefore, a letter substitution in an IBAN is equivalent to a two-digit substitution error in its converted IBAN. Theorem 4.6 states that all two-digit substitution errors in a converted IBAN can be detected by the MOD97-10 algorithm, except for the cases $00 \leftrightarrow 97$, $01 \leftrightarrow 98$, and $02 \leftrightarrow 99$. However, these undetected errors cannot occur since letter values are always between 10 and 35. Hence, all letter-to-letter substitutions in an IBAN are detected.

Thus, all digit-to-digit and letter-to-letter substitutions in an IBAN are detected. $\square$

The previous corollary completely characterises substitutions between characters of the same type, namely digit-to-digit and letter-to-letter substitutions. It remains to consider substitutions between different character types. When a digit in an IBAN is replaced by a letter, or a letter is replaced by a digit, the corresponding converted IBAN changes in length because letters are represented by two decimal digits. Consequently, these errors are not covered by Theorems and Lemmas 4.3-4.6 proved above.

In practice, many banking systems first verify that an IBAN has the correct format, including its prescribed length, format, and character structure before applying the MOD97-10 algorithm [26]. Consequently, digit-to-letter and letter-to-digit substitutions are generally rejected immediately. However, this precheck is not part of the MOD97-10 algorithm itself. Therefore, for completeness, we also analyse these substitutions mathematically by considering their effect on the converted IBAN. The Theorems 4.8 and 4.9 characterise exactly when digit-to-letter and letter-to-digit substitutions are not detected in IBANs.

Theorem 4.8. *Let $a_n \dots a_2 a_1$ be the converted IBAN corresponding to a valid IBAN. Suppose that a digit is replaced by a letter in the IBAN. This digit corresponds to the digit a_k in the converted IBAN and is replaced by a letter with two-digit decimal representation $10a_{k,1} + a_{k,0}$. So, $a_k \rightarrow 10a_{k,1} + a_{k,0}$ in the converted IBAN. Then the digit-to-letter substitution in the IBAN is undetected if and only if its converted IBAN representation satisfies*

$$9 \sum_{i=k+1}^n a_i \cdot 10^{i-k} \equiv a_k - 10a_{k,1} - a_{k,0} \pmod{97} \quad (4.5)$$

Proof. Since $a_n \dots a_2 a_1$ is a valid converted IBAN, it must satisfy

$$\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97}.$$

We can use this to show that a digit-to-letter substitution cannot be detected in an IBAN if and only if

$$\begin{aligned} & 10 \sum_{i=k+1}^n a_i \cdot 10^{i-1} + a_{k,1} \cdot 10^k + a_{k,0} \cdot 10^{k-1} + \sum_{i=1}^{k-1} a_i \cdot 10^{i-1} \equiv 1 \pmod{97} \\ \Leftrightarrow & 10 \sum_{i=k+1}^n a_i \cdot 10^{i-1} + a_{k,1} \cdot 10^k + a_{k,0} \cdot 10^{k-1} + \sum_{i=1}^n a_i \cdot 10^{i-1} - a_k \cdot 10^{k-1} - \sum_{i=k+1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97} \\ \Leftrightarrow & 9 \sum_{i=k+1}^n a_i \cdot 10^{i-1} + a_{k,1} \cdot 10^k + a_{k,0} \cdot 10^{k-1} - a_k \cdot 10^{k-1} \equiv 0 \pmod{97} \\ \Leftrightarrow & 10^{k-1} \left(9 \sum_{i=k+1}^n a_i \cdot 10^{i-k} + 10a_{k,1} + a_{k,0} - a_k \right) \equiv 0 \pmod{97} \end{aligned}$$

Since $\gcd(10^{k-1}, 97) = 1$ for all $k \in \{1, \dots, n\}$ we have that

$$\begin{aligned} & 10^{k-1} \left(9 \sum_{i=k+1}^n a_i \cdot 10^{i-k} + 10a_{k,1} + a_{k,0} - a_k \right) \equiv 0 \pmod{97} \\ \Leftrightarrow & 9 \sum_{i=k+1}^n a_i \cdot 10^{i-k} + 10a_{k,1} + a_{k,0} - a_k \equiv 0 \pmod{97} \\ \Leftrightarrow & 9 \sum_{i=k+1}^n a_i \cdot 10^{i-k} \equiv a_k - 10a_{k,1} - a_{k,0} \pmod{97} \end{aligned}$$

So, the digit-to-letter substitution error in an IBAN is undetected if and only if its converted IBAN satisfies the congruence modulo 97 condition stated in the theorem (Equation (4.5)). $\square$

Theorem 4.9. *Let $a_n \dots a_2 a_1$ be the converted IBAN corresponding to a valid IBAN. Suppose that a letter is replaced by a digit in the IBAN. This letter is represented by $10a_{k+1} + a_k$ in the converted IBAN and is replaced by a digit $a'_k \in \{0, \dots, 9\}$. So, $10a_{k+1} + a_k \rightarrow a'_k$ in the converted IBAN. Then the letter-to-digit substitution in the IBAN is undetected if and only if its converted IBAN representation satisfies*

$$9 \sum_{i=k+2}^n a_i \cdot 10^{i-k-1} \equiv a'_k - 10a_{k+1} - a_k \pmod{97}. \quad (4.6)$$

Proof. $a_n \dots a_2 a_1$ is a valid converted IBAN and must therefore satisfy $\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97}$. The substitution error from a letter to a digit in an IBAN cannot be detected if and only if

$$\begin{aligned} 10^{-1} \sum_{i=k+2}^n a_i \cdot 10^{i-1} + a'_k \cdot 10^{k-1} + \sum_{i=1}^{k-1} a_i \cdot 10^{i-1} &\equiv 1 \pmod{97} \\ \Leftrightarrow \sum_{i=k+2}^n a_i \cdot 10^{i-2} + a'_k \cdot 10^{k-1} + \sum_{i=1}^n a_i \cdot 10^{i-1} - a_k \cdot 10^{k-1} - a_{k+1} \cdot 10^k - \sum_{i=k+2}^n a_i \cdot 10^{i-1} &\equiv 1 \pmod{97} \\ \Leftrightarrow \sum_{i=k+2}^n a_i \cdot 10^{i-2} + a'_k \cdot 10^{k-1} - a_k \cdot 10^{k-1} - a_{k+1} \cdot 10^k - 10 \sum_{i=k+2}^n a_i \cdot 10^{i-2} &\equiv 0 \pmod{97} \\ \Leftrightarrow -9 \sum_{i=k+2}^n a_i \cdot 10^{i-1} + a'_k \cdot 10^{k-1} - a_k \cdot 10^{k-1} - a_{k+1} \cdot 10^k &\equiv 0 \pmod{97} \\ \Leftrightarrow 10^{k-1} \left(-9 \sum_{i=k+2}^n a_i \cdot 10^{i-k-1} - 10a_{k+1} - a_k + a'_k \right) &\equiv 0 \pmod{97} \end{aligned}$$

$\gcd(10^{k-1}, 97) = 1$ for all $k \in \{0, \dots, n-1\}$ and therefore we have that

$$\begin{aligned} 10^{k-1} \left(-9 \sum_{i=k+2}^n a_i \cdot 10^{i-k-1} - 10a_{k+1} - a_k + a'_k \right) &\equiv 0 \pmod{97} \\ \Leftrightarrow -9 \sum_{i=k+2}^n a_i \cdot 10^{i-k-1} - 10a_{k+1} - a_k + a'_k &\equiv 0 \pmod{97} \\ \Leftrightarrow 9 \sum_{i=k+2}^n a_i \cdot 10^{i-k-1} &\equiv a'_k - 10a_{k+1} - a_k \pmod{97} \end{aligned}$$

Hence, a letter-to-digit substitution error in an IBAN cannot be detected if and only if its converted IBAN representation satisfies the condition stated in the theorem (Equation (4.6)). $\square$

The results derived above show IBANs provide strong error detection for single-character errors. In particular, all digit-to-digit and letter-to-letter substitutions are always detected. Substitutions between digits and letters are also detected in almost all cases. Undetected errors occur only when the corresponding converted IBAN satisfies a specific congruence condition modulo 97.

This is also confirmed by the computational analysis on Dutch IBANs. Figure 4.2 shows the detection rates of all possible single-character substitutions in an IBAN. The figure displays a clear block structure. The upper-left and lower-right blocks, corresponding to digit-to-digit and letter-to-letter substitutions, have detection rates of 100%. This confirms the results in Corollary 4.7, which states that all digit-to-digit and letter-to-letter substitutions in an IBAN are detected. The upper-right and lower-left blocks correspond to digit-to-letter and letter-to-digit substitutions respectively. Their detection rates are slightly below 100%, although still very high. Note that the colour scale ranges from 0.965 to 1, which visually exaggerates the observed differences in detection rates. These results are consistent with Theorems 4.8 and 4.9, which show that such substitutions are undetected only when a specific congruence condition modulo 97 is satisfied. Since this condition is only rarely met, the corresponding detection rates remain close to 100%.

Furthermore, no strong patterns are visible within the upper-right and lower-left blocks. This suggests that the probability of detecting a digit-to-letter or letter-to-digit substitution is not determined primarily by the substituted symbols themselves, but rather by properties such as the position at which the substitution occurs and the surrounding IBAN structure.

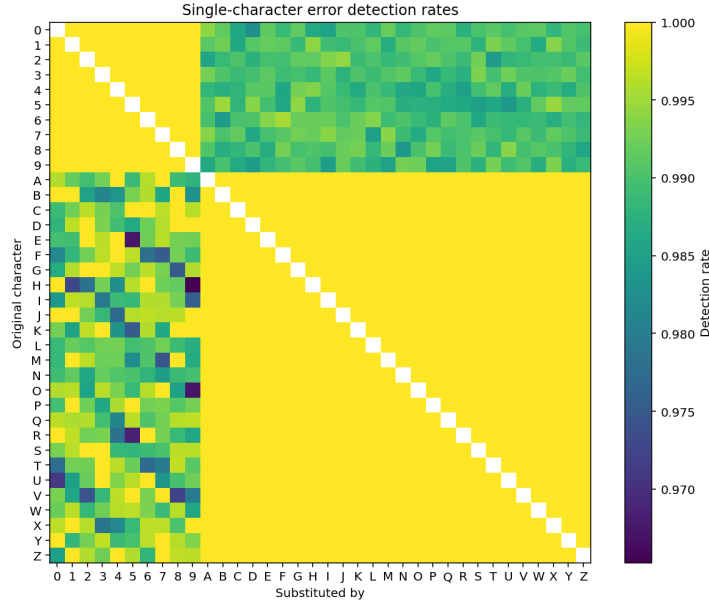


Figure 4.2: Heatmap displaying the error detection rates of single-character errors in Dutch IBANs.

The upper-right and lower-left blocks are not exact mirror images of one another. This is consistent with the theoretical results, since digit-to-letter and letter-to-digit substitutions are characterised by different congruence conditions (Theorems 4.8 and 4.9). Consequently, digit-to-letter and letter-to-digit substitutions do not necessarily have the same detection-rate patterns.

4.2.2. Transposition error

In the previous subsection, single-character errors in an IBAN were analysed. We now extend the analysis to transposition errors, where two adjacent characters in an IBAN are interchanged. Since letters in an IBAN are converted into two digits in the converted IBAN, a transposition error in an IBAN corresponds to an error pattern affecting two, three, or four consecutive digits in the converted IBAN under the MOD97-10 algorithm. Therefore, before determining which transposition errors can be detected in an IBAN, we first analyse which of these multi-digit error patterns are detected in the converted IBAN. Theorem 4.10 considers a specific class of three-digit transpositions, while Theorem 4.11 considers a specific class of four-digit transpositions. Both results concern error patterns in converted IBANs. Corollary 4.12 then translates these findings back to the IBAN level.

When a letter and a digit occur next to each other in an IBAN, the letter is represented by two digits in the converted IBAN, while the digit remains a single digit. As a result, a local transposition involving a letter and a digit leads to an error pattern affecting three consecutive digits in the converted IBAN. This motivates the analysis of errors of the form $abc \leftrightarrow cab$ with $a \neq c$ or $b \neq c$ in the converted IBAN.

Theorem 4.10 proves that all such three-digit transposition errors in a converted IBAN are detected by the MOD97-10 algorithm, except for the cases $970 \leftrightarrow 097$ and $029 \leftrightarrow 902$.

Theorem 4.10. *The MOD97-10 algorithm can detect all errors $abc \leftrightarrow cab$ with $a \neq c$ or $b \neq c$ in a converted IBAN, except $970 \leftrightarrow 097$ and $029 \leftrightarrow 902$.*

Proof. Let $a_n \dots a_2 a_1$ be a valid converted IBAN. The validity condition then is

$$\sum_{i=1}^n a_i \cdot 10^{i-1} \equiv 1 \pmod{97}$$

Suppose that the error occurs at the positions $k, k+1$, and $k+2$ for some $k \in \{0, \dots, n-2\}$. Let $a_{k+2} = a$, $a_{k+1} = b$, and $a_k = c$ with $a, b, c \in \{0, \dots, 9\}$ and $a \neq c$ or $b \neq c$. Then, the error $a_{k+2}a_{k+1}a_k \leftrightarrow a_k a_{k+2}a_{k+1} \Leftrightarrow abc \leftrightarrow cab$ goes undetected by the MOD97-10 algorithm if and only if,

$$\begin{aligned}
 \sum_{\substack{i=1 \\ i \neq k, k+1, k+2}}^n a_i \cdot 10^{i-1} + c \cdot 10^{k+1} + a \cdot 10^k + b \cdot 10^{k-1} &\equiv 1 \pmod{97} \\
 \Leftrightarrow \sum_{i=1}^n a_i \cdot 10^{i-1} - a \cdot 10^{k+1} - b \cdot 10^k - c \cdot 10^{k-1} + c \cdot 10^{k+1} + a \cdot 10^k + b \cdot 10^{k-1} &\equiv 1 \pmod{97} \\
 \Leftrightarrow -a \cdot 10^{k+1} - b \cdot 10^k - c \cdot 10^{k-1} + c \cdot 10^{k+1} + a \cdot 10^k + b \cdot 10^{k-1} &\equiv 0 \pmod{97} \\
 \Leftrightarrow -100a - 10b - c + 100c + 10a + b &\equiv 0 \pmod{97} \\
 \Leftrightarrow 99c - 90a - 9b &\equiv 0 \pmod{97} \\
 \Leftrightarrow 11c - 10a - b &\equiv 0 \pmod{97}
 \end{aligned}$$

Since $11c - 10a - b$ ranges from -99 to 99, it can only take the values 0, -97, and 97. The only possibility to have $11c - 10a - b = 0$ is if the value a, b , and c are all equal to 0. However, this contradicts the initial assumption that $a \neq c$ or $b \neq c$. Thus, $11c - 10a - b = 0$ is not possible for any choices of a, b , and c . To have $11c - 10a - b = \pm 97$ we must have that either $a = 9, b = 7, c = 0$ or $a = 0, b = 2, c = 9$. So,

$$11c - 10a - b \equiv 0 \pmod{97} \Leftrightarrow a = 9, b = 7, c = 0 \text{ or } a = 0, b = 2, c = 9$$

Thus, all errors $abc \leftrightarrow cab$ with $a \neq c$ or $b \neq c$ in a converted IBAN are detected by the MOD97-10 algorithm, except for the errors $970 \leftrightarrow 097$ and $029 \leftrightarrow 902$. $\square$

When two adjacent letters occur in an IBAN, each letter is represented by two digits in the converted IBAN. As a result, a transposition of two letters in an IBAN corresponds to an error affecting four consecutive digits in its converted IBAN. This motivates the analysis of errors of the form $abcd \leftrightarrow cdab$ in the converted IBAN. Theorem 4.11 shows that all such four-digit transposition errors with $10a + b \neq 10c + d$ are detected by the MOD97-10 algorithm.

Theorem 4.11. *Every four-digit error $abcd \leftrightarrow cdab$ with $10a + b \neq 10c + d$ in a converted IBAN can be detected by the MOD97-10 algorithm.*

Proof. Let $a_n \dots a_2 a_1$ be a valid converted IBAN according to the MOD97-10 algorithm. Suppose an error occurs at the digits a_k, a_{k+1}, a_{k+2} , and a_{k+3} , so $10a_{k+3} + a_{k+2} \neq 10a_{k+1} + a_k$. Then, the four-digit error $a_{k+3}a_{k+2}a_{k+1}a_k \leftrightarrow a_{k+1}a_k a_{k+3}a_{k+2}$ cannot be detected by the MOD97-10 algorithm if and only if

$$\begin{aligned}
 \sum_{\substack{i=1 \\ i \neq k, k+1, k+2, k+3}}^n a_i \cdot 10^{i-1} + a_{k+1} \cdot 10^{k+2} + a_k \cdot 10^{k+1} + a_{k+3} \cdot 10^k + a_{k+2} \cdot 10^{k-1} &\equiv 1 \pmod{97} \\
 \Leftrightarrow \sum_{i=1}^n a_i \cdot 10^{i-1} - a_{k+3} \cdot 10^{k+2} - a_{k+2} \cdot 10^{k+1} - a_{k+1} \cdot 10^k - a_k \cdot 10^{k-1} + a_{k+1} \cdot 10^{k+2} + a_k \cdot 10^{k+1} & \\
 + a_{k+3} \cdot 10^k + a_{k+2} \cdot 10^{k-1} &\equiv 1 \pmod{97} \\
 \Leftrightarrow -a_{k+3} \cdot 10^{k+2} - a_{k+2} \cdot 10^{k+1} - a_{k+1} \cdot 10^k - a_k \cdot 10^{k-1} + a_{k+1} \cdot 10^{k+2} + a_k \cdot 10^{k+1} + a_{k+3} \cdot 10^k & \\
 + a_{k+2} \cdot 10^{k-1} &\equiv 0 \pmod{97} \\
 \Leftrightarrow 10^{k-1} (99 (-10a_{k+3} - a_{k+2} + 10a_{k+1} + a_k)) &\equiv 0 \pmod{97} \\
 \Leftrightarrow -990a_{k+3} - 99a_{k+2} + 990a_{k+1} + 99a_k &\equiv 0 \pmod{97} \\
 \Leftrightarrow 10a_{k+1} + a_k &\equiv 10a_{k+3} + a_{k+2} \pmod{97} \\
 \Leftrightarrow 10a_{k+3} + a_{k+2} &= 10a_{k+1} + a_k
 \end{aligned}$$

However, this contradicts the assumption that $10a_{k+3} + a_{k+2} \neq 10a_{k+1} + a_k$ and therefore all four-digit errors $abcd \leftrightarrow cdab$ with $10a + b \neq 10c + d$ in a converted IBAN are detected by the MOD97-10 algorithm. $\square$

We now use the results obtained for two-, three-, and four-digit error patterns in converted IBANs to determine which adjacent transposition errors are detected in an IBAN. This is characterised in Corollary 4.12. The only transpositions excluded from the analysis are those involving the right check digit x_4 and the first BBAN character x_5 . Recall from Definition 4.1 that an IBAN consists of a two-letter country code, two check digits, and a BBAN consisting of both letters and digits. Consequently, x_4 is always a digit, whereas x_5 may be either a digit or a letter depending on the country-specific BBAN format. Transpositions involving x_4 and x_5 are not covered by the previous results of error detection in converted IBANs by the MOD97-10 algorithm because, after the IBAN rearrangement step, the affected characters are no longer represented by adjacent digits in the converted IBAN. The error then involves digit at both the beginning and end of the converted IBAN.

Corollary 4.12. *Every adjacent transposition error in an IBAN is detected, except for the cases*

$$B1 \leftrightarrow 1B, \quad M2 \leftrightarrow 2M, \quad X3 \leftrightarrow 3X,$$

provided that the transposition does not occur between the right check digit x_4 and the first BBAN character x_5 .

Proof. Let $x_1 x_2 x_3 x_4 x_5 \dots x_m$ be a valid IBAN. Suppose an error occurs at the position k and $k + 1$ for some $k \in \{1, \dots, m - 1\} \setminus \{4\}$. The first four characters of an IBAN are moved to the back in order to obtain its converted IBAN representation. Since $k \neq 4$, the transposition of two adjacent characters in an IBAN is equivalent to an error involving two, three or four adjacent digits in the converted IBAN. Therefore, there are three different error classes in an IBAN to distinguish between, an error between two digits, an error between two letters, and an error between a digit and a letter.

Case 1: two digits.

A digit in an IBAN corresponds to a single digit in its converted IBAN. Therefore, a transposition between two adjacent digits in an IBAN is equivalent to a transposition error between two adjacent digits in the converted IBAN. By Lemma 4.4, the MOD97-10 algorithm detects all such transposition errors. Hence, all transpositions between two adjacent digits in an IBAN are detected.

Case 2: two letters.

In order to apply the MOD97-10 algorithm, all letters are replaced by their numerical value in $\{10, \dots, 35\}$. So, a transposition error between two letters in a valid IBAN corresponds to an error $abcd \leftrightarrow cdab$ with $10a + b \neq 10c + d$ in its converted IBAN. By Theorem 4.11 we know that the MOD97-10 algorithm can detect all errors of this form in a converted IBAN. Therefore, an IBAN detects all transpositions between two letters.

Case 3: a digit and a letter.

Since each letter in an IBAN is replaced by its numerical value in $\{10, \dots, 35\}$ in the converted IBAN, a transposition between a digit and a letter in an IBAN affects a block of three consecutive digits in the converted IBAN. More specifically, such a transposition corresponds to an error of the form $abc \leftrightarrow cab$ in the converted IBAN, where ab is the numerical value of the letter. By Theorem 4.10, all errors of this form in a converted IBAN are detected by the MOD97-10 algorithm, except for the cases $970 \leftrightarrow 097$ and $029 \leftrightarrow 902$. However, these exceptional patterns cannot result from a valid IBAN, since the numerical values of letters lie in the range 10 to 35. It remains to consider the special cases $1B \leftrightarrow B1$, $2M \leftrightarrow M2$, and $3X \leftrightarrow X3$. In each of these cases, both symbols are converted into the same three-digit block in the converted IBAN; $1B$ and $B1$ are converted into 111, $2M$ and $M2$ into 222, and $3X$ and $x3$ into 333. As a result, a transposition between the two characters does not change the converted IBAN at all. Therefore, the transposition errors $1B \leftrightarrow B1$, $2M \leftrightarrow M2$, and $3X \leftrightarrow X3$ cannot be detected in an IBAN.

Thus, all transposition errors, $x_4 \leftrightarrow x_5$ not included, are detected in an IBAN, except for the errors $B1 \leftrightarrow 1B$, $M2 \leftrightarrow 2M$, and $X3 \leftrightarrow 3X$. $\square$

Until now, we have only considered transposition errors that do not involve the second check digit x_4 and the first character of the BBAN x_5 . We now consider the remaining case, namely transpositions at this boundary position.

When rearranging an IBAN into its converted IBAN representation, the first four characters (country code and check digits) are moved to the back. As a result, the character x_5 in the original IBAN corresponds to left most digit positions in the converted IBAN. The positional weights of these digits depend on the length of

the converted IBAN, and therefore on the length of the original IBAN. Consequently, the length of an IBAN influences which transposition errors at this boundary position are detected by the MOD97-10 algorithm.

To illustrate this behaviour, we consider a computational example based on Dutch IBANs. A Dutch IBAN consists of the country code NL, two check digits, and a BBAN. The BBAN starts with a four-letter bank code followed by a numerical account number. Since the BBAN in Dutch IBANs always begins with a letter, we restrict attention to digit–letter transpositions at the boundary.

Figure 4.3 shows the error detection rates for Dutch IBANs with varying lengths. Varying the IBAN length is achieved by adding or removing digits from the account number in the BBAN.

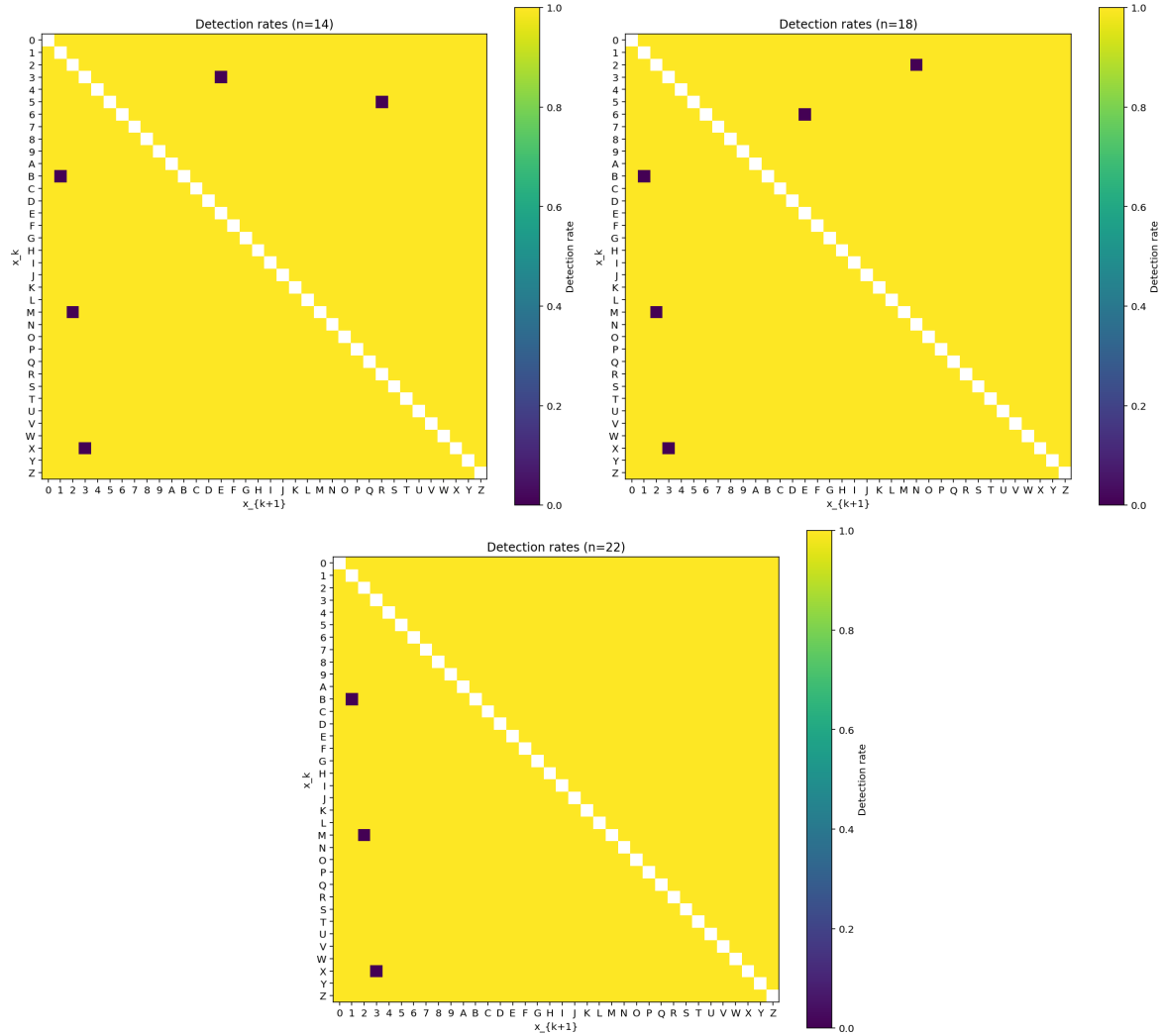


Figure 4.3: Heatmaps of transposition error detection rates for IBANs of length $n = 14$ (upper-left), $n = 18$ (upper-right), and $n = 22$ (lower-middle).

In the Dutch IBAN format considered here, a transposition of the form *digit–letter* can only occur at the boundary between the second check digit and the first BBAN character. Consequently, the upper triangular part of the matrix corresponds to boundary transpositions in this setting.

For the IBAN of length 18, the errors $2N \rightarrow N2$ and $3X \rightarrow X3$ are not detected. For the extended IBAN of length 22, no undetected boundary transpositions are observed. Finally, for the shortened IBAN of length 14, the errors $3E \rightarrow E3$ and $5R \rightarrow R5$ are not detected.

Thus, the length of an IBAN influences the detectability of boundary transposition errors between the second check digit and the first BBAN character.

5

GTIN-13

This chapter discusses the structure of a GTIN-13 number. It then explains the algorithm used to compute the check digit of this number and concludes with an analysis on the type of errors the algorithm is able to detect.

5.1. Structure

The Global Trade Item Number (GTIN) is a globally standardised identifier developed by GS1 to uniquely identify trade items. The GTIN is a numerical identifier that is placed underneath barcodes of products. They are commonly used in supermarkets, warehouse, e-commerce platforms, healthcare systems, and international trade.

There are several different formats of GTINs, namely, GTIN-8, GTIN-12, GTIN-13, and GTIN-14. The number in each format name indicates the total number of digits in the GTIN [8]. In this section, the focus will be on GTIN-13, the 13-digit GTIN format.

GTIN-13 is also known as EAN-13, which is the European Article Number consisting of 13 digits. EAN-13 was the original name of the 13-digit barcode system. After this system became internationally standardised by GS1 in 2005, its name was changed to GTIN-13 [27]. As a result, GTIN-13 and EAN-13 refer to the same 13-digit identification system and are often used interchangeably.

The GTIN-13 consists of 13 digits and its structure can be divided into three parts, the GS1 Company Prefix, the item reference, and one check digit. The first four to twelve digits of the GTIN-13 represent the GS1 Company Prefix [28]. This is assigned by the national organisation GS1 and identifies the company responsible for the numbering of a product. The shorter the Company Prefix, the more digits remain available for the item reference, allowing the company to identify a larger number of products. The Company Prefix itself can also be divided into two parts, the GS1 Prefix and the company identifier. The GS1 Prefix consists of three digits and identifies the GS1 organisation that registered a product. Consequently, it does not necessarily indicate where the product was manufactured. Examples of GS1 Prefixes can be found in Table 5.1. The company identifier indicates the specific business that owns the product.

Following the GS1 Company Prefix is the item reference. This is assigned by the company itself and uniquely identifies a specific product or product variation. Different package sizes, flavours, colours, and configurations of a product therefore require different GTIN-13 numbers [29]. The final digit of the GTIN-13 is the check digit and can be computed using the algorithm described in Subsection 5.1.1. This check digit is used to detect input errors.

A special type of GTIN-13 is the 13-digit International Standard Book Number (ISBN), referred to as ISBN-13. ISBN-13 numbers uniquely identify books and other published materials. Unlike ordinary GTIN-13 numbers, which begin with GS1 prefixes assigned to national GS1 organisations such as GS1 Netherlands or GS1 France, ISBN-13 numbers always begin with the reserved prefixes 978 or 979, as in Table 5.1. These prefixes are collectively referred to as “Bookland” prefixes and are used exclusively for publications within the inter-

national book industry. Since ISBN-13 is integrated into the GTIN-13 system, the check digit of an ISBN-13 is computed using the same algorithm as for GTIN-13 numbers.

GS1 Prefix	GS1 Organisation
300 - 379	GS1 France
450 - 459 & 490 - 499	GS1 Japan
500 - 509	GS1 UK
622	GS1 Egypt
754 - 755	GS1 Canada
870 - 879	GS1 Netherlands
930 - 939	GS1 Australia
978 - 979	Bookland (ISBN)

Table 5.1: Examples of GS1 Prefixes used in GTIN-13 and their corresponding GS1 organisations [30].

5.1.1. GTIN-13 check digit algorithm

Before presenting the algorithm, we first introduce a general representation of a GTIN-13. Since a GTIN-13 consists of 13 digits, it can be represented as

$$d_{12}d_{11}\dots d_2d_1d_0$$

where $d_i \in \{0, \dots, 9\}$ for all $i \in \{0, \dots, 12\}$, and d_0 is the check digit.

The main idea of the algorithm is that all digits are multiplied by an alternating weight, one and three. The resulting products are then summed. The check digit is chosen such that this sum is divisible by 10 [31]. So, like the Luhn algorithm discussed in Section 3.1.1, this algorithm uses weighted sums to compute a check digit.

In order to calculate the check digit, the algorithm is applied to the digits $d_1, d_2, \dots$, and d_{12} from right to left. The weights w_i that the digits d_i are multiplied by can be determined using

$$w_i = \begin{cases} 3 & \text{if } i \text{ is odd} \\ 1 & \text{if } i \text{ is even} \end{cases}$$

So, the weighted sum that is used to determine the check digit d_0 is

$$s = \sum_{i=1}^{12} w_i \cdot d_i$$

A GTIN-13 is valid if the sum of the weighted sum s and the check digit d_0 is divisible by 10. So, when we define $w_0 = 1$, the validity condition is

$$s + d_0 = \sum_{i=0}^{12} w_i \cdot d_i \equiv 0 \pmod{10} \quad (5.1)$$

We use this condition to determine the check digit, which itself has to be a number in $\{0, \dots, 9\}$. Therefore, the check digit is given by

$$d_0 = -s \bmod 10$$

This ensures that the check digit will always be an integer between 0 and 9.

Example 5.1.

The first 12 digits of an ISBN are known, and we want to determine the check digit. The ISBN is

978-7-0024-2074-

All digits in an odd position when counted from the right, excluding the check digit, must be multiplied with 3. All other digits are multiplied with 1. Then all the resulting products are summed.

Position i	12	11	10	9	8	7	6	5	4	3	2	1	0
Digit	9	7	8	7	0	0	2	4	2	0	7	4	
		·3		·3		·3		·3		·3		·3	
Product	9	21	8	21	0	0	2	12	2	0	7	12	

The sum of all these products is 94. Therefore, the check digit is

$$d_0 = -94 \bmod 10 = 6$$

Therefore, the complete ISBN is

$$978-7-0024-2074-6$$

5.2. Analysis of error detection

Now that we know how check digits are computed in a GTIN-13 number and how the validity of such a number is determined, we can investigate its error-detection capabilities. Although the GTIN-13 check digit system is designed to be robust against common input mistakes, its effectiveness depends on the structure of the error considered. We therefore analyse several types of errors that commonly arise in practice and determine whether they can be detected by the algorithm of GTIN-13 or not. The error types that are evaluated in this section are single-digit errors, transposition errors, twin errors, jump-transposition errors, and jump-twin errors.

5.2.1. Single-digit error

The most common type of input error is the single-digit error, where a single digit in a code is incorrectly entered or transmitted. Because such errors occur frequently in practical applications, it is essential for a coding system such as GTIN-13 to be able to detect them. Theorem 5.1 formalises this observation and shows that every single-digit error in a GTIN-13 number is indeed detected.

Theorem 5.1. *Every single-digit error in a GTIN-13 number is detected.*

Proof. Let $d_{12}d_{11}...d_2d_1d_0$ be a valid GTIN-13 number. The validity condition of this sequence then is

$$s + d_0 = \sum_{i=0}^{12} w_i \cdot d_i \equiv 0 \pmod{10}.$$

Suppose that the single-digit error occurs at the k^{th} position for some $k \in \{0, ..., 12\}$. So, d_k is replaced by d'_k under the condition that $d_k \neq d'_k$. The single-digit error cannot be detected in a GTIN-13 number if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k}}^{12} w_i \cdot d_i + w_k \cdot d'_k &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{12} w_i \cdot d_i - w_k \cdot d_k + w_k \cdot d'_k \equiv 0 \pmod{10} \\ &\Leftrightarrow w_k \cdot d'_k \equiv w_k \cdot d_k \pmod{10} \end{aligned}$$

Since $\gcd(1, 10) = 1$ and $\gcd(3, 10) = 1$ we know that $\gcd(w_k, 10) = 1$ for all $k \in \{1, ..., 12\}$. Therefore,

$$\begin{aligned} w_k \cdot d'_k &\equiv w_k \cdot d_k \pmod{10} \Leftrightarrow d'_k \equiv d_k \pmod{10} \\ &\Leftrightarrow d'_k = d_k \end{aligned}$$

However, this contradicts the assumption that $d_k \neq d'_k$ and therefore all single-digit errors in a GTIN-13 number can be detected. $\square$

5.2.2. Transposition error

Another common type of input error is a transposition error, in which two adjacent digits are interchanged. A GTIN-13 number is able to detect almost all such errors. There are a few exceptional cases in which a transposition error is not detected. The errors $05 \leftrightarrow 50$, $16 \leftrightarrow 61$, $27 \leftrightarrow 72$, $38 \leftrightarrow 83$, and $49 \leftrightarrow 94$ are not detected. This is proven in Theorem 5.2.

Theorem 5.2. *In a GTIN-13 number, every transposition error $ab \leftrightarrow ba$ with $a \neq b$ can be detected, except for the cases*

$$05 \leftrightarrow 50, \quad 16 \leftrightarrow 61, \quad 27 \leftrightarrow 72, \quad 38 \leftrightarrow 83, \quad 49 \leftrightarrow 94.$$

Proof. Assume that $d_{12}d_{11}...d_2d_1d_0$ is a valid GTIN-13 number, then $\sum_{i=0}^{12} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose that a transposition occurs in the digits d_k and d_{k+1} for some $k \in \{0, ..., 11\}$, so $d_k \neq d_{k+1}$.

Since the weights alternate between 1 and 3, exactly one of the two adjacent digits is multiplied by 3. Without loss of generality, assume that d_k is multiplied by 3 before an error has occurred. The error is undetected if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k+1}}^{12} w_i \cdot d_i + d_k + 3d_{k+1} &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{12} w_i \cdot d_i - 3d_k - d_{k+1} + d_k + 3d_{k+1} \equiv 0 \pmod{10} \\ &\Leftrightarrow 2d_{k+1} - 2d_k \equiv 0 \pmod{10} \\ &\Leftrightarrow 2d_{k+1} \equiv 2d_k \pmod{10} \\ &\Leftrightarrow d_{k+1} \equiv d_k \pmod{5} \\ &\Leftrightarrow (d_{k+1} = 0, d_k = 5) \text{ or } (d_{k+1} = 1, d_k = 6) \text{ or } \\ &\quad (d_{k+1} = 2, d_k = 7) \text{ or } (d_{k+1} = 3, d_k = 8) \text{ or } \\ &\quad (d_{k+1} = 4, d_k = 9) \end{aligned}$$

Therefore, all transposition errors $ab \leftrightarrow ba$ with $a \neq b$ are detected, except for the errors $05 \leftrightarrow 50$, $16 \leftrightarrow 61$, $27 \leftrightarrow 72$, $38 \leftrightarrow 83$, and $49 \leftrightarrow 94$. $\square$

5.2.3. Twin error

Twin errors are also errors involving two adjacent digits like transposition errors. However, with twin errors the digits must be identical. Almost all twin errors can be detected in a GTIN-13 number, with the exception of $00 \leftrightarrow 55$, $11 \leftrightarrow 66$, $22 \leftrightarrow 77$, $33 \leftrightarrow 88$, and $44 \leftrightarrow 99$. So, when the difference between the original value and the changed value is equal to 5, the error cannot be detected. Theorem 5.3 proves this.

Theorem 5.3. *Every twin error $aa \leftrightarrow bb$ with $a \neq b$ is detected in a GTIN-13 number, except for the errors*

$$00 \leftrightarrow 55, \quad 11 \leftrightarrow 66, \quad 22 \leftrightarrow 77, \quad 33 \leftrightarrow 88, \quad 44 \leftrightarrow 99.$$

Proof. Let $d_{12}d_{11}...d_2d_1d_0$ be a valid GTIN-13 number. Then, it satisfies $s + d_0 = \sum_{i=0}^{12} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose that the twin error occurs in the two digits d_k and d_{k+1} for some $k \in \{0, ..., 11\}$. Before an error has occurred, suppose that the values of these two digits are both equal to a digit a . After the error has occurred, the values of the digits d_k and d_{k+1} are both equal to b such that $b \neq a$.

Without loss of generality, assume that k is even. Then, the twin error cannot be detected in the GTIN-13 number if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k+1}}^{12} w_i \cdot d_i + b + 3b &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{12} w_i \cdot d_i - 3a - a + b + 3b \equiv 0 \pmod{10} \\ &\Leftrightarrow 4b - 4a \equiv 0 \pmod{10} \\ &\Leftrightarrow 4b \equiv 4a \pmod{10} \end{aligned}$$

Since $\gcd(4, 10) = 2$ we have that

$$\begin{aligned} 4b \equiv 4a \pmod{10} &\Leftrightarrow b \equiv a \pmod{5} \\ &\Leftrightarrow (a=0, b=5) \text{ or } (a=1, b=6) \text{ or } (a=2, b=7) \text{ or } \\ &\quad (a=3, b=8) \text{ or } (a=4, b=9) \end{aligned}$$

Thus, all twin errors $aa \leftrightarrow bb$ with $a \neq b$ are detected in a GTIN-13 number, except for the cases $00 \leftrightarrow 55$, $11 \leftrightarrow 66$, $22 \leftrightarrow 77$, $33 \leftrightarrow 88$, and $44 \leftrightarrow 99$. $\square$

5.2.4. Jump-transposition error

Now we look at a variation of a transposition error involving three adjacent digits. A jump-transposition error occurs when the two outer digits of the three adjacent digits are interchanged. Since a GTIN-13 number uses alternating weights to determine the check digit, every jump-transposition error goes undetected. This is proven in Theorem 5.4 by splitting the proof into two cases.

Theorem 5.4. *No jump-transposition error $abc \leftrightarrow cba$ with $a \neq c$ and $c \in \{0, \dots, 9\}$ is detected in a GTIN-13 number.*

Proof. Let $d_{12}d_{11}\dots d_2d_1d_0$ be a valid GTIN-13 number, then $\sum_{i=0}^{12} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose the jump-transposition error occurs at the k^{th} and $(k+2)^{\text{th}}$ digit with $d_k \neq d_{k+2}$ for some $k \in \{0, \dots, 10\}$. Now we have to distinguish between two cases, when k is even and when k is odd.

Case 1: k is even. Then, both d_k and d_{k+2} are multiplied by 1 in the checksum computation. Interchanging these two digits does not alter the weighted sum, since both carry the same weight. Therefore, when the jump-transposition error has occurred, the new GTIN-13 number is still valid. Thus, the error is not detected.

Case 2: k is odd. Then, the digit d_k and d_{k+2} are both multiplied by 3 when determining the value of the check digit. By the same reasoning as in case 1, the contribution to the checksum computation does not change when switching the positions of d_k and d_{k+2} . So, the GTIN-13 number with an error is also valid and therefore the error cannot be detected.

Since a GTIN-13 number cannot detect the error, whether k is even or odd, no jump-transposition errors $abc \leftrightarrow cba$ with $a \neq c$ is detected in a GTIN-13. $\square$

5.2.5. Jump-twin error

Jump-twin errors involve three digits and can be viewed as an extension of twin errors, in which two identical digits are separated by a single intermediate digit. Almost all jump-twin errors can be detected in a GTIN-13 number. However, the errors $0c0 \leftrightarrow 5c5$, $1c1 \leftrightarrow 6c6$, $2c2 \leftrightarrow 7c7$, $3c3 \leftrightarrow 8c8$, and $4c4 \leftrightarrow 9c9$ are not detected for any integer c between 0 and 9.

Theorem 5.5. *All jump-twin errors $aca \leftrightarrow bcb$ with $a \neq b$ and $c \in \{0, \dots, 9\}$ are detected in a GTIN-13 number, except for the cases*

$$0c0 \leftrightarrow 5c5, \quad 1c1 \leftrightarrow 6c6, \quad 2c2 \leftrightarrow 7c7, \quad 3c3 \leftrightarrow 8c8, \quad 4c4 \leftrightarrow 9c9.$$

Proof. Assume that the GTIN-13 number $d_{12}d_{11}\dots d_2d_1d_0$ is valid. Then, it satisfies $s + d_0 = \sum_{i=0}^{12} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose that an error is made and the digits d_k and d_{k+2} with $d_k \neq d_{k+2}$ are involved. Also suppose that before the error has occurred, the value of these two digits is both equal to $a \in \{0, \dots, 9\}$ while after the error has occurred, the value is equal to $b \neq a$. The proof is split into two parts, when k is even and when it is odd.

Case 1: k is even. Then, the error cannot be detected if and only if

$$\begin{aligned} \sum_{\substack{i=1 \\ i \neq k, k+2}}^{12} w_i \cdot d_i + b + b &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{12} w_i \cdot d_i - a - a + b + b \equiv 0 \pmod{10} \\ &\Leftrightarrow 2b \equiv 2a \pmod{10} \\ &\Leftrightarrow b \equiv a \pmod{5} \\ &\Leftrightarrow (a=0, b=5) \text{ or } (a=1, b=6) \text{ or } (a=2, b=7) \text{ or } \\ &\quad (a=3, b=8) \text{ or } (a=4, b=9) \end{aligned}$$

Case 2: k is odd. The jump-twin error is not detected in the GTIN-13 number if and only if

$$\begin{aligned}
 \sum_{\substack{i=1 \\ i \neq k, k+2}}^{12} w_i \cdot d_i + 3b + 3b &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{12} w_i \cdot d_i - 3a - 3a + 3b + 3b \equiv 0 \pmod{10} \\
 &\Leftrightarrow 6b \equiv 6a \pmod{10} \\
 &\Leftrightarrow b \equiv a \pmod{5} \\
 &\Leftrightarrow (a=0, b=5) \text{ or } (a=1, b=6) \text{ or } (a=2, b=7) \text{ or} \\
 &\quad (a=3, b=8) \text{ or } (a=4, b=9)
 \end{aligned}$$

Thus, all jump-twin errors $aca \leftrightarrow bcb$ with $a \neq b$ are detected in a GTIN-13 number, except for the errors

$$0c0 \leftrightarrow 5c5, \quad 1c1 \leftrightarrow 6c6, \quad 2c2 \leftrightarrow 7c7, \quad 3c3 \leftrightarrow 8c8, \quad 4c4 \leftrightarrow 9c9.$$

□

6

Modulo 10 algorithm with alternating weight p

In Chapter 5 we have seen how the algorithm used to compute the check digit of a GTIN-13 number works and which errors it can detect. The algorithm alternates multiplying the digits with the weights 1 and 3. So, a follow-up question could be; What happens if you do not multiply with the value 3, but instead with an arbitrary integer p ? In this chapter, this specific question is discussed. First, the new algorithm with an alternating weight p is defined. Section 6.2 then explains which errors can and cannot be detected under specific conditions of p .

6.1. Algorithm

The algorithm applies a modulo 10 operation on a sequence of n numeric digits with $n \geq 1$. In this chapter the algorithm used will be referred to as the 'modulo 10 algorithm with weight p '. The sequence to which the algorithm is applied can be denoted as in Equation (6.1). In this equation d_0 denotes the check digit to be determined and $d_i \in \{0, \dots, 9\}$ for all $i \in \{0, \dots, n-1\}$.

$$d_{n-1}d_{n-2}\dots d_3d_2d_1d_0 \quad (6.1)$$

The algorithm works the same as the algorithm used to compute the check digit of a GTIN-13 number described in Section 5.1.1. However, instead of multiplying by 3 when the position i is odd, this algorithm multiplies by an integer p . Therefore, the weights w_i that the digits d_i are multiplied with are defined as

$$w_i = \begin{cases} p & \text{if } i \text{ is odd} \\ 1 & \text{if } i \text{ is even} \end{cases}$$

In order for a number to be valid, it must satisfy that the sum of the digits multiplied by their weights has remainder 0 when divided by 10. Therefore, it must satisfy the validity condition

$$s = \sum_{i=0}^{n-1} w_i \cdot d_i \equiv 0 \pmod{10}$$

This validity condition is used to determine the value of the check digit d_0 . The value of this digit must be an integer between 0 and 9. Therefore, the check digit is computed by

$$d_0 = \left(- \sum_{i=1}^{n-1} w_i \cdot d_i \right) \pmod{10}$$

The modulo operation ensures that the check digit lies in $\{0, \dots, 9\}$.

6.2. Analysis of error detection

In this section we will determine under which condition certain errors can and cannot be detected. The error types discussed in Section 2.2 will be covered. So, we will look at single-digit errors, transposition errors, twin errors, jump-transposition errors, and jump-twin errors.

6.2.1. Single-digit error

The error-detecting capability of the modulo 10 algorithm with weight p depends on the relationship between this p and the modulus 10, more specifically the greatest common divisor of p and 10. Since the possible values of $\gcd(p, 10)$ are 1, 2, 5, and 10, we analyse each of these cases separately. The corresponding results are stated and proved in Theorems 6.1, 6.2, 6.3, and 6.4.

Theorem 6.1. *If p and 10 are coprime, $\gcd(p, 10) = 1$, then all single-digit errors are detected by the modulo 10 algorithm with weight p .*

Proof. Let $d_{n-1} \dots d_2 d_1 d_0$ be a valid number, then it satisfies $s = \sum_{i=0}^{n-1} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose that the error occurs in the k^{th} digit for some $k \in \{0, \dots, n-1\}$. The value of this digit after the error is made is d'_k such that $d_k \neq d'_k$. Now, we consider two cases, either k is even or k is odd.

Case 1: k is even. Then, the single-digit error cannot be detected if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k}}^{n-1} w_i \cdot d_i + d'_k &\equiv 0 \pmod{10} &\Leftrightarrow &\sum_{i=0}^{n-1} w_i \cdot d_i - d_k + d'_k \equiv 0 \pmod{10} \\ &&\Leftrightarrow &d'_k - d_k \equiv 0 \pmod{10} \\ &&\Leftrightarrow &d'_k \equiv d_k \pmod{10} \\ &&\Leftrightarrow &d'_k = d_k \end{aligned}$$

However, this contradicts the assumption that $d_k \neq d'_k$. Thus, all single-digit errors where k is even are detected.

Case 2: k is odd. The error is not detected if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k}}^{n-1} w_i \cdot d_i + p d'_k &\equiv 0 \pmod{10} &\Leftrightarrow &\sum_{i=0}^{n-1} w_i \cdot d_i - p \cdot d_k + p \cdot d'_k \equiv 0 \pmod{10} \\ &&\Leftrightarrow &p \cdot d'_k - p \cdot d_k \equiv 0 \pmod{10} \\ &&\Leftrightarrow &p \cdot d'_k \equiv p \cdot d_k \pmod{10} \end{aligned}$$

Since $\gcd(p, 10) = 1$, we have that

$$\begin{aligned} p \cdot d'_k \equiv p \cdot d_k \pmod{10} &\Leftrightarrow d'_k \equiv d_k \pmod{10} \\ &\Leftrightarrow d'_k = d_k \end{aligned}$$

This contradicts the assumption $d_k \neq d'_k$ and therefore the single-digit error at the k^{th} position is always detected.

Hence, all single-digit errors are detected by the modulo 10 algorithm with weight p if $\gcd(p, 10) = 1$. $\square$

Theorem 6.2. *If $\gcd(p, 10) = 2$, then the modulo 10 algorithm with weight p can detect all single-digit errors, except when the error occurs at a position whose assigned weight in the checksum is p . In that case, the error is not detected for the digit pairs $0 \leftrightarrow 5$, $1 \leftrightarrow 6$, $2 \leftrightarrow 7$, $3 \leftrightarrow 8$, and $4 \leftrightarrow 9$.*

Proof. Assume $d_{n-1} \dots d_2 d_1 d_0$ is a valid number. Suppose that the error occurs at the k^{th} position and the value of this digit after the error is $d'_k \neq d_k$. The argument is identical to that in Theorem 6.1, so we only consider the case where k is odd. Then the single digit error cannot be detected if and only if

$$\begin{aligned} p \cdot d'_k \equiv p \cdot d_k \pmod{10} &\Leftrightarrow d'_k \equiv d_k \pmod{5} \\ &\Leftrightarrow |d_k - d'_k| \in \{0, 5, 10, 15, \dots\} \end{aligned}$$

Since d_k and d'_k are both integers between 0 and 9, and $d_k \neq d'_k$, we have that $|d_k - d'_k| = 5$. Thus, all single-digit errors are detected except for $0 \leftrightarrow 5$, $1 \leftrightarrow 6$, $2 \leftrightarrow 7$, $3 \leftrightarrow 8$, and $4 \leftrightarrow 9$ if $\gcd(p, 10) = 2$. $\square$

Theorem 6.3. *If $\gcd(p, 10) = 5$, every single-digit error, $a \leftrightarrow b$ with $a \neq b$, occurring at a position with weight 1 is detected. At digit positions with weight p , the only undetected errors are those for which the difference between a and b is even.*

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid code for the modulo 10 algorithm with weight p . Suppose that the single-digit error occurs at the digit d_k for some $k \in \{0, \dots, n-1\}$. The value of this digit after the error has occurred is $d'_k \neq d_k$. The proof follows the same argument as in Theorem 6.1, we therefore only look at when k is odd. Then, the single-digit error is not detected if and only if

$$\begin{aligned} p \cdot d'_k &\equiv p \cdot d_k \pmod{10} &\Leftrightarrow d'_k &\equiv d_k \pmod{2} \\ &&\Leftrightarrow |d_k - d'_k| &\in \{0, 2, 4, 6, \dots\} \end{aligned}$$

Since $d_k, d'_k \in \{0, \dots, 9\}$ and $d_k \neq d'_k$, the error is detected if and only if $|d_k - d'_k| \notin \{2, 4, 6, 8\}$. Thus all single-digit errors are detected, except for when a digit is at a position with weight p . The only undetected single-digit errors $a \leftrightarrow b$ with $a \neq b$ are those for which the difference between a and b is even. $\square$

Theorem 6.4. *If $\gcd(p, 10) = 10$, then every single-digit error occurring at a position with weight 1 is detected, whereas no single-digit error occurring at a position with weight p is detected.*

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid code. Suppose the error occurs at the k^{th} digit and the value of this digit after the error has been made is $d'_k \neq d_k$. The reduction as in Theorem 6.1 shows that single-digit errors cannot be detected if and only if

$$p \cdot d'_k \equiv p \cdot d_k \pmod{10} \Leftrightarrow d'_k \equiv d_k \pmod{1}$$

This holds for any digit combination of d_k and d'_k . Thus, no error can be detected when a digit is at a position with weight p . $\square$

These four cases completely characterize the detection of single-digit errors by the modulo 10 algorithm with alternating weights 1 and p . As the value of the greatest common divisor $\gcd(p, 10)$ increases, the algorithm becomes less effective at detecting errors occurring at positions with weight p .

6.2.2. Transposition error

Unlike single-digit errors, the detection of transposition errors depends on the difference between the two alternating weights 1 and p . Consequently, the error detection depends on $\gcd(p-1, 10)$. As before, there are four possible values of this greatest common divisor, namely 1, 2, 5, and 10. For each case, the corresponding error-detection capability of the modulo 10 algorithm with weight p is stated and proven in Theorems 6.5, 6.6, 6.7, and 6.8.

Theorem 6.5. *Every transposition error $ab \leftrightarrow ba$ with $a \neq b$ is detected by the modulo 10 algorithm with weight p if $\gcd(p-1, 10) = 1$.*

Proof. Assume $d_{n-1} \dots d_1 d_0$ is a valid code. Suppose that the error occurs at the digits d_k and d_{k+1} with $d_k \neq d_{k+1}$ for some $k \in \{0, \dots, n-2\}$. Without loss of generality, assume the k is even, so that d_k has weight 1 and d_{k+1} has weight p before an error has occurred. The transposition error cannot be detected if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k+1}}^{n-1} w_i \cdot d_i + p \cdot d_k + d_{k+1} &\equiv 0 \pmod{10} &\Leftrightarrow \sum_{i=0}^{n-1} w_i \cdot d_i - d_k - p \cdot d_{k+1} + p \cdot d_k + d_{k+1} &\equiv 0 \pmod{10} \\ &&\Leftrightarrow (p-1)d_k &\equiv (p-1)d_{k+1} \pmod{10} \end{aligned}$$

Since $\gcd(p-1, 10) = 1$ we have that

$$\begin{aligned} (p-1)d_k &\equiv (p-1)d_{k+1} \pmod{10} &\Leftrightarrow d_k &\equiv d_{k+1} \pmod{10} \\ &&\Leftrightarrow d_k &= d_{k+1} \end{aligned}$$

However, this contradicts the assumption that $d_k \neq d_{k+1}$, so all transposition errors are detected if $\gcd(p-1, 10) = 1$. $\square$

Theorem 6.6. *If $\gcd(p-1, 10) = 2$, then all transposition errors are detected by the modulo 10 algorithm with weight p except for $05 \leftrightarrow 50$, $16 \leftrightarrow 61$, $27 \leftrightarrow 72$, $38 \leftrightarrow 83$, and $49 \leftrightarrow 94$.*

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid number and suppose the digits d_k and d_{k+1} are interchanged for some $k \in \{0, \dots, n-2\}$. Without loss of generality, assume that k is even. The argument is identical to the one used in Theorem 6.5. Since $\gcd(p-1, 10) = 2$ we have that the error cannot be detected if and only if

$$\begin{aligned} (p-1)d_k \equiv (p-1)d_{k+1} \pmod{10} &\Leftrightarrow d_k \equiv d_{k+1} \pmod{5} \\ &\Leftrightarrow (d_{k+1} = 0, d_k = 5) \text{ or } (d_{k+1} = 1, d_k = 6) \text{ or } (d_{k+1} = 2, d_k = 7) \text{ or} \\ &\quad (d_{k+1} = 3, d_k = 8) \text{ or } (d_{k+1} = 4, d_k = 9) \end{aligned}$$

Thus all transposition errors $ab \leftrightarrow ba$ with $a \neq b$ are detected, except for when the difference between d_k and d_{k+1} is equal to 5. $\square$

Theorem 6.7. *If $\gcd(p-1, 10) = 5$ then all transposition errors $ab \leftrightarrow ba$ with $a \neq b$ are detected, except for when $|a-b| \in \{2, 4, 6, 8\}$.*

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid code, then $\sum_{i=0}^{n-1} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose an error has occurred at the digits d_k and d_{k+1} with $d_k \neq d_{k+1}$. Without loss of generality, assume k is even. This proof follows the same argument as in Theorem 6.5, so since $\gcd(p-1, 10) = 5$, the transposition error is undetected if and only if

$$\begin{aligned} (p-1)d_k \equiv (p-1)d_{k+1} \pmod{10} &\Leftrightarrow d_k \equiv d_{k+1} \pmod{2} \\ &\Leftrightarrow |d_k - d_{k+1}| \in \{2, 4, 6, 8\} \end{aligned}$$

Thus all transposition errors $ab \leftrightarrow ba$ with $a \neq b$ are detected, except for when the difference between a and b is equal to 2, 4, 6, or 8. Or equivalently, $|a-b| \in \{2, 4, 6, 8\}$. $\square$

Theorem 6.8. *No transposition error can be detected by the modulo 10 algorithm with weight p if $\gcd(p-1, 10) = 10$.*

Proof. Assume $d_{n-1} \dots d_1 d_0$ to be a valid number, then $\sum_{i=0}^{n-1} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose the error involves the digits d_k and d_{k+1} for some $k \in \{0, \dots, n-2\}$. Without loss of generality, assume that k is even. We proceed as in Theorem 6.5. Since $\gcd(p-1, 10) = 10$, the error is undetected if and only if

$$(p-1)d_k \equiv (p-1)d_{k+1} \pmod{10} \Leftrightarrow d_k \equiv d_{k+1} \pmod{10}$$

This holds for any combination of the digits d_k and d_{k+1} , so no transposition error can be detected if $\gcd(p-1, 10) = 10$. $\square$

The four cases above show that the detectability of transposition errors is completely determined by the value of $\gcd(p-1, 10)$. Depending on this value, the algorithm either detects all swaps, fails only for specific structured pairs of digits, or in the extreme case fails to detect any transposition error at all.

6.2.3. Twin error

Twin errors differ from the other error types because both affected digits are equal before the error occurs. This means the effect of the error depends on the combined contribution of two identical digits under alternating weights 1 and p . As a result, detectability depends on the factor $(p+1)$ rather than p or $p-1$. As before, we therefore consider the cases $\gcd(p+1, 10) \in \{1, 2, 5, 10\}$ separately. The results are stated in Theorems 6.9, 6.10, 6.11, and 6.12.

Theorem 6.9. *If $\gcd(p+1, 10) = 1$, then all twin errors $aa \leftrightarrow bb$ with $a \neq b$ are detected by the modulo 10 algorithm with weight p .*

Proof. Let d_{n-1} be a valid number, then $\sum_{i=0}^{n-1} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose the twin error occurs in the digits d_k and d_{k+1} for some $k \in \{0, \dots, n-2\}$. Without loss of generality, assume k is even. Suppose that before the error has occurred, the value of d_k and d_{k+1} is equal to some integer $a \in \{0, \dots, 9\}$. After the error has been made, assume that this value is $b \neq a$. The twin error cannot be detected if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k+1}}^{n-1} w_i \cdot d_i + b + p \cdot b &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w_i \cdot d_i - a - p \cdot a + b + p \cdot b \equiv 0 \pmod{10} \\ &\Leftrightarrow (p+1)b \equiv (p+1)a \pmod{10} \end{aligned}$$

Since we know that $\gcd(p+1, 10) = 1$, we have

$$\begin{aligned} \Leftrightarrow (p+1)b &\equiv (p+1)a \pmod{10} \Leftrightarrow b \equiv a \pmod{10} \\ &\Leftrightarrow a = b \end{aligned}$$

However, this contradicts the assumption that $a \neq b$. Thus, all twin errors can be detected by the modulo 10 algorithm with weight p if $\gcd(p+1, 10) = 1$. $\square$

Theorem 6.10. *If $\gcd(p+1, 10) = 2$, then the modulo 10 algorithm with weight p can detect all twin errors, except $00 \leftrightarrow 55$, $11 \leftrightarrow 66$, $22 \leftrightarrow 77$, $33 \leftrightarrow 88$, and $44 \leftrightarrow 99$.*

Proof. Assume $d_{n-1} \dots d_1 d_0$ to be a valid number and suppose that the twin error involves the digits d_k and d_{k+1} for some $k \in \{0, \dots, n-2\}$. The argument used in this proof is identical to that of Theorem 6.9, so the twin error is undetected if and only if

$$\begin{aligned} (p+1)b &\equiv (p+1)a \pmod{10} \Leftrightarrow b \equiv a \pmod{5} \\ &\Leftrightarrow (a=0, b=5) \text{ or } (a=1, b=6) \text{ or } (a=2, b=7) \\ &\quad \text{or } (a=3, b=8) \text{ or } (a=4, b=9) \end{aligned}$$

Thus all twin errors are detected if $\gcd(p+1, 10) = 2$, except for $00 \leftrightarrow 55$, $11 \leftrightarrow 66$, $22 \leftrightarrow 77$, $33 \leftrightarrow 88$, and $44 \leftrightarrow 99$. $\square$

Theorem 6.11. *Let $\gcd(p+1, 10) = 5$, then all twin errors $aa \leftrightarrow bb$ with $a \neq b$ are detected, except for when $|a-b| \in \{2, 4, 6, 8\}$.*

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid number. Suppose that the error has occurred at the digit d_k and d_{k+1} for some $k \in \{0, \dots, n-2\}$. Without loss of generality, assume that k is even. The same reduction as in Theorem 6.9 shows that the twin error cannot be detected if and only if

$$\begin{aligned} (p+1)b &\equiv (p+1)a \pmod{10} \Leftrightarrow b \equiv a \pmod{2} \\ &\Leftrightarrow |a-b| \in \{0, 2, 4, 6, \dots\} \end{aligned}$$

Since $a, b \in \{0, \dots, 9\}$ and $a \neq b$, the error is undetected if and only if $|a-b| \in \{2, 4, 6, 8\}$. Thus, if $\gcd(p+1, 10) = 5$, all twin errors are detected except for when the difference between a and b is equal to 2, 4, 6, or 8. $\square$

Theorem 6.12. *Every twin error is undetected by the modulo 10 algorithm with weight p if $\gcd(p+1, 10) = 10$.*

Proof. Assume that $d_{n-1} \dots d_1 d_0$ is a valid number, so $\sum_{i=0}^{n-1} w_i \cdot d_i \equiv 0 \pmod{10}$. Suppose the twin error involves the digits d_k and d_{k+1} for some $k \in \{0, \dots, n-2\}$. We proceed as in Theorem 6.9, so if $\gcd(p+1, 10) = 10$ a twin error cannot be detected if and only if

$$(p+1)b \equiv (p+1)a \pmod{10} \Leftrightarrow b \equiv a \pmod{1}$$

Since this holds for any digit combination of a and b , no twin error can be detected by the modulo 10 algorithm with weight p when $\gcd(p+1, 10) = 10$. $\square$

6.2.4. Jump-transposition error

As in the Luhn algorithm (Section 3.2) and the GTIN-13 check digit system (Section 5.2), jump-transposition errors are never detected. This is because both algorithms work with alternating weights, so no matter the value of p , a jump-transposition error can never be detected. This is proven in Theorem 6.13.

Theorem 6.13. *No jump-transposition error $abc \leftrightarrow cba$ with $a \neq c$ is detected by the modulo 10 algorithm with weight p independent of the value of p .*

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid number and suppose that a jump-transposition error occurs at positions k and $k+2$, where $k \in \{0, \dots, n-3\}$. Since k and $k+2$ are either both even or both odd, they are assigned the same weight in the checksum computation. Therefore, interchanging d_k and d_{k+2} does not change the contribution to the checksum. So, the jump-transposition error cannot be detected. $\square$

6.2.5. Jump-twin error

A jump-twin error affects digits two positions apart, which always have the same weight in an alternating scheme. As a result, the effect depends on whether these positions have weight 1 or p . Unlike the previous error types, the detection capability for jump-twin errors depends on the value of $\gcd(p, 5)$. Since $\gcd(p, 5) \in \{1, 5\}$, these two cases are considered separately. The corresponding results are stated and proven in Theorems 6.14 and 6.15.

Theorem 6.14. *If $\gcd(p, 5) = 1$, all jump-twin errors $aca \leftrightarrow bcb$ with $a \neq b$ are detected, except for $0c0 \leftrightarrow 5c5$, $1c1 \leftrightarrow 6c6$, $2c2 \leftrightarrow 7c7$, $3c3 \leftrightarrow 8c8$, and $4c4 \leftrightarrow 9c9$.*

Proof. Let $d_{n-1} \dots d_1 d_0$ be a valid code and suppose that the error occurs at the digits d_k and d_{k+2} for some $k \in \{0, \dots, n-3\}$. Before the error has occurred, the value of d_k and d_{k+2} is equal to $a \in \{0, \dots, 9\}$. After the error has occurred, the value of these two digits is $b \neq a$. Now we consider two cases.

Case 1: k is even. Then the jump-twin error cannot be detected if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k+2}}^{n-1} w_i \cdot d_i + b + b &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w_i \cdot d_i - a - a + b + b \equiv 0 \pmod{10} \\ &\Leftrightarrow 2b \equiv 2a \pmod{10} \\ &\Leftrightarrow b \equiv a \pmod{5} \\ &\Leftrightarrow (a=0, b=5) \text{ or } (a=1, b=6) \text{ or } (a=2, b=7) \\ &\quad \text{or } (a=3, b=8) \text{ or } (a=4, b=9) \end{aligned}$$

Case 2: k is odd. The jump-twin error goes undetected if and only if

$$\begin{aligned} \sum_{\substack{i=0 \\ i \neq k, k+2}}^{n-1} w_i \cdot d_i + p \cdot b + p \cdot b &\equiv 0 \pmod{10} \Leftrightarrow \sum_{i=0}^{n-1} w_i \cdot d_i - d_k - p \cdot a - p \cdot a + p \cdot b + p \cdot b \equiv 0 \pmod{10} \\ &\Leftrightarrow 2p \cdot b \equiv 2p \cdot a \pmod{10} \\ &\Leftrightarrow p \cdot b \equiv p \cdot a \pmod{5} \end{aligned}$$

Since $\gcd(p, 5) = 1$ we have that

$$\begin{aligned} p \cdot b \equiv p \cdot a \pmod{5} &\Leftrightarrow b \equiv a \pmod{5} \\ &\Leftrightarrow (a=0, b=5) \text{ or } (a=1, b=6) \text{ or } (a=2, b=7) \\ &\quad \text{or } (a=3, b=8) \text{ or } (a=4, b=9) \end{aligned}$$

Thus, all jump-twin errors are detected except for the cases $0c0 \leftrightarrow 5c5$, $1c1 \leftrightarrow 6c6$, $2c2 \leftrightarrow 7c7$, $3c3 \leftrightarrow 8c8$, and $4c4 \leftrightarrow 9c9$. $\square$

Theorem 6.15. *Let $\gcd(p, 5) = 5$. If the outer two digits of a jump-twin error occur in positions with weight 1, then the only undetected errors are*

$$0c0 \leftrightarrow 5c5, \quad 1c1 \leftrightarrow 6c6, \quad 2c2 \leftrightarrow 7c7, \quad 3c3 \leftrightarrow 8c8, \quad \text{and} \quad 4c4 \leftrightarrow 9c9.$$

with $c \in \{0, \dots, 9\}$. If the outer two digits occur in positions with weight p , no jump-twin errors are detected.

Proof. Let $d_{n-1}, \dots, d_1 d_0$ be a valid number. Suppose that the jump-twin error involves the digits d_k and d_{k+2} for some $k \in \{0, \dots, n-3\}$. Before the error has occurred, the value of these two digits is $a \in \{0, \dots, 9\}$. After the error has occurred, the value is $b \neq a$. This proof follows the same argument as in Theorem 6.14, so we only have to look at the case when k is odd. Since $\gcd(p, 5) = 5$, the error cannot be detected if and only if

$$p \cdot b \equiv p \cdot a \pmod{5} \quad \Leftrightarrow \quad b \equiv a \pmod{1}$$

This holds for any digit combination of a and b and therefore no jump-twin error is detected when the outer two digits are assigned weight p in the checksum. When the assigned weight is 1, the errors $0c0 \leftrightarrow 5c5$, $1c1 \leftrightarrow 6c6$, $2c2 \leftrightarrow 7c7$, $3c3 \leftrightarrow 8c8$, and $4c4 \leftrightarrow 9c9$ are undetected. $\square$

The results obtained in this section are summarized in Table 6.1, which shows the conditions under which each error type is detected.

Error type	Detection depends on
Single-digit	$\gcd(p, 10)$
Transposition	$\gcd(p - 1, 10)$
Twin	$\gcd(p + 1, 10)$
Jump-transposition	Independent of p
Jump-twin	$\gcd(p, 5)$

Table 6.1: Summary of the error detection conditions for the modulo 10 algorithm with weight p .

For single-digit errors, detection improves when $\gcd(p, 10) = 1$, i.e. when p is coprime with 10. However, there exists no value of p for which $\gcd(p, 10) = 1$, $\gcd(p - 1, 10) = 1$ and $\gcd(p + 1, 10) = 1$ simultaneously hold. Therefore, improving detection in one error category typically weakens it in another. This confirms that the system does not have an 'optimal' weight p , but instead shows a trade-off between the detection capabilities for different types of input errors.

The Luhn algorithm described in Chapter 3 can be seen as a special case of the modulo 10 algorithm with alternating weight $p = 2$. The key difference is the subtraction of 9 whenever a doubled digit exceeds 9. This '-9 correction' guarantees detection of all single-digit errors. However, this improvement comes at a cost. It introduces error patterns that cannot be detected, including the $09 \leftrightarrow 90$ transposition and certain twin errors like $22 \leftrightarrow 55$, $33 \leftrightarrow 66$, and $44 \leftrightarrow 77$. Conversely, removing the adjustment of subtracting 9 improves the detection of transposition and twin errors, all of them are detected. However, this simplification reintroduces undetected single-digit errors. In particular, detection fails when an error occurs at a digit position that is multiplied by the weight p , and the difference between the value of the original digit and the value of this digit after an error has occurred equals 5.

Overall, both the modulo 10 algorithm with alternating weight p and the Luhn algorithm illustrate the same fundamental principle: improving robustness against one class of errors inevitably weakens performance against others. The choice of p therefore reflects a trade-off between guaranteeing detection of all single-digit errors and maintaining strong error detection against multi-digit errors such as transposition and twin errors.

Conclusion and recommendations

7.1. Conclusion

The aim of this thesis was to evaluate the effectiveness of common check digit algorithms in detecting errors in everyday identification numbers. More specifically, the research question was:

"How effective are common check digit algorithms in detecting errors in everyday identification numbers?"

To answer this question, the error detection capabilities of the Luhn algorithm, the MOD 97-10 algorithm used in IBANs, the GTIN-13 check digit algorithm, and a general modulo 10 algorithm with alternating weight p were analysed mathematically and the first two were also verified through computational experiments.

The Luhn algorithm detects all single-digit errors and almost all transposition errors. However, it also has several well-defined weaknesses. In particular, the transposition error $09 \leftrightarrow 90$, the twin errors $22 \leftrightarrow 55$, $33 \leftrightarrow 66$, and $44 \leftrightarrow 77$, and all jump-transposition errors remain undetected. These theoretical results were also confirmed by the computational simulations.

The MOD 97-10 algorithm applied to a converted IBAN performs significantly better. The mathematical analysis showed that it detects all single-digit errors, all transposition errors, all twin errors, and almost all two-digit substitution errors, with only a very small number of exceptional cases. Furthermore, digit addition and digit deletion in a converted IBAN is detected unless it satisfies a specific congruence condition modulo 97. Consequently, an IBAN can detect all letter to letter, and digit to digit substitutions and almost all substitutions of digit to letter or vice versa. The MOD97-10 algorithm can also detect almost all errors $abc \leftrightarrow cab$ with $a \neq c$ or $b \neq c$, and all errors of the form $abcd \leftrightarrow cdab$ with $10a + b \neq 10c + d$. In particular, the errors $970 \leftrightarrow 097$ and $029 \leftrightarrow 902$ remain undetected. Therefore, an IBAN can detect all transposition errors whenever the second check digit is not involved. Otherwise the specific undetected errors depend on the length of an IBAN.

The algorithm of GTIN-13 numbers detects all single-digit errors and, like the Luhn algorithm, cannot detect jump-transposition errors because digits that are separated by one position receive the same weight in the checksum computation. It also detects almost all transposition and twin errors. The remaining undetected transposition, twin, and jump-twin errors all share the same property: the digit values involved differ by five. Despite these theoretical exceptions, the algorithm performs well in detecting the vast majority of common input errors.

Finally, the general modulo 10 algorithm with alternating weight p illustrates how the choice of the weight influences the error detection properties of a checksum algorithm. In particular, the ability to detect transposition, twin, and jump-transposition errors depends on the greatest common divisors of p , $p \pm 1$, and 10, as summarised in Table 6.1. Since no single choice of p eliminates all types of errors simultaneously, every checksum algorithm based on this weighting scheme represents a trade-off between different error detection capabilities.

In conclusion, modern check digit algorithms are effective at detecting many common types of accidental human input errors, including single-digit errors, transposition errors, twin errors, jump-transposition errors, and jump-twin errors. Although no algorithm is perfect and every algorithm has theoretical limitations, the undetected error patterns form only a limited subset of all possible transmission errors. This confirms the importance of check digits in ensuring the reliability of widely used identification systems where manual data entry errors are unavoidable.

7.2. Recommendations

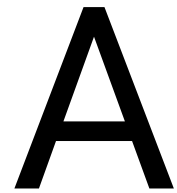
This research has several limitations. First, the computational experiments focused on randomly generated errors and assumed that all error patterns occur with equal probability within each error class. In practice, some typing mistakes occur more frequently than others. For example, certain digits or keys may be more likely to be confused than others. Consequently, the observed detection rates may differ from those encountered in real-world applications.

A second limitation is that this thesis considered only four checksum algorithms. Other algorithms, such as the Verhoeff and Damm algorithms, provide stronger error detection capabilities than the Luhn algorithm and may detect error patterns that remain undetected by simpler weighting schemes. Including these algorithms in the analysis would provide a broader comparison of existing checksum systems.

Several directions for future research arise from these limitations. One possible extension is to analyse checksum algorithms under realistic models of human typing behaviour instead of assuming uniformly distributed errors. This would provide a more accurate assessment of their practical error detection performance.

Another interesting direction is the study of error correction. This thesis focused exclusively on determining which errors can and cannot be detected by checksum algorithms. However, some algorithms are capable not only of detecting errors but also of identifying and correcting them. Investigating the mathematical conditions under which identification numbers can be corrected would be a natural continuation of this thesis.

Finally, the modulo 10 algorithm with weight p could be generalised in several directions. Rather than alternating between the fixed weights 1 and p , one could investigate the error detection properties when using two arbitrary weights q and p , where $q \in \mathbb{Z}$. In addition, it would be interesting to study how the choice of modulus affects the detection capability of the system. Changing the modulus may change which error patterns are detectable.



Python code for computational analysis of Luhn algorithm

```
import random
import matplotlib.pyplot as plt
import numpy as np

def luhn_validity(number: str) -> bool:
    """ check wether a number is valid or not according to the Luhn algorithm """
    digits = [int(d) for d in number]
    reverse = digits[:-1][::-1]          # first n-1 digits reversed

    s = 0
    for i, digit in enumerate(reverse):
        if i%2 == 0 and digit <= 4:
            s += 2*digit
        elif i%2 == 0 and digit > 4:
            s += 2*digit - 9
        else:
            s += digit

    if (s + digits[-1]) % 10 == 0:
        return True
    else:
        return False

def create_valid_number(n: int) -> str:
    """ creates a valid number randomly """
    begin = [random.randint(0, 9) for _ in range(n-1)]
    begin_reversed = begin[::-1]

    s=0
    for i, digit in enumerate(begin_reversed):
        if i%2 == 0 and digit <= 4:
            s += 2*digit
        elif i%2 == 0 and digit > 4:
            s += 2*digit - 9
        else:
            s += digit

    valid_number = begin + [-s % 10]
```

```

    string_valid = ''
    for d in valid_number:
        string_valid += str(d)

    return string_valid

def single_digit_error(number: str) -> str:
    """ alters one digit in the number to create a single-digit error """
    digits = list(number)

    position = random.randint(0, len(digits)-1)
    value = digits[position]

    new = (int(value) + random.randint(1,9)) % 10
    digits[position] = str(new)

    return ''.join(digits)

def transposition_error(number: str) -> (str, int, int):
    """ swaps two non-identical, adjacent digits randomly """
    digits = list(number)

    while True:
        position = random.randint(0, len(digits)-2)
        if digits[position] != digits[position+1]:
            a, b = int(digits[position]), int(digits[position+1])
            digits[position], digits[position+1] = digits[position+1], digits[position]
            break

    return ''.join(digits), a, b

def twin_error(number: str) -> (str, int, int):
    """ replaces two identical, adjacent digits with another set of identical digits """
    digits = list(number)

    candidates = []

    for i in range(len(digits)-1):
        if digits[i] == digits[i+1]:
            candidates.append(i)

    if not candidates:
        return None, None, None

    position = random.choice(candidates)

    a = int(digits[position])
    new_value = (int(digits[position]) + random.randint(1,9)) % 10
    digits[position], digits[position+1] = str(new_value), str(new_value)

    return ''.join(digits), a, new_value

def jump_transposition_error(number: str) -> (str, int, int):
    """ swaps two non-identical digits, separated by a third digit """
    digits = list(number)

```

```

while True:
    position = random.randint(0, len(digits)-3)
    if digits[position] != digits[position+2]:
        a, b = digits[position], digits[position+2]
        digits[position], digits[position+2] = digits[position+2], digits[position]
        break

    return ''.join(digits), a, b

def jump_twin_error(number: str) -> (str, int, int):
    """ replaces two identical digits, seperated by a third digit, with another set of
    identical digits """
    digits = list(number)

    candidates = []

    for i in range(len(digits)-2):
        if digits[i] == digits[i+2]:
            candidates.append(i)

    if not candidates:
        return None, None, None

    position = random.choice(candidates)

    a = int(digits[position])
    new_value = (int(digits[position]) + random.randint(1,9)) % 10
    digits[position], digits[position+2] = str(new_value), str(new_value)

    return ''.join(digits), a, new_value

# initialising
sim = 1300000
detected_sde = 0
detected_tre = 0
detected_twe = 0
count_twe = 0
detected_jtre = 0
detected_jtwe = 0
count_jtwe = 0

detec_trans = np.zeros((10,10))
total_trans = np.zeros((10,10))
detec_twin = np.zeros((10,10))
total_twin = np.zeros((10,10))
detec_jump_twin = np.zeros((10,10))
total_jump_twin = np.zeros((10,10))

for i in range(sim):
    num = create_valid_number(16)
    sde = single_digit_error(num)
    tre, a_tre, b_tre = transposition_error(num)
    twe, a_twe, b_twe = twin_error(num)
    jtre, a_jtre, b_jtre = jump_transposition_error(num)
    jtwe, a_jtwe, b_jtwe = jump_twin_error(num)

```

```

if not luhn_validity(sde):
    detected_sde += 1

if not luhn_validity(tre):
    detected_tre += 1
    detec_trans[a_tre, b_tre] += 1
    total_trans[a_tre, b_tre] += 1
if luhn_validity(tre):
    total_trans[a_tre, b_tre] += 1

if twe is not None:
    count_twe += 1
    total_twin[a_twe, b_twe] += 1
    if not luhn_validity(twe):
        detected_twe += 1
        detec_twin[a_twe, b_twe] += 1

if not luhn_validity(jtre):
    detected_jtre += 1

if jtwe is not None:
    count_jtwe += 1
    total_jump_twin[a_jtwe, b_jtwe] += 1
    if not luhn_validity(jtwe):
        detected_jtwe += 1
        detec_jump_twin[a_jtwe, b_jtwe] += 1

# printing detection rates of different type of errors
print('Single digit errors: ', (detected_sde / sim)*100, '%')
print('Transposition errors: ', (detected_tre / sim)*100, '%', ' ', detected_tre)
print('Twin errors: ', (detected_twe / count_twe)*100, '%', detected_twe, count_twe)
print('Jump-transposition errors: ', (detected_jtre / sim)*100, '%')
print('Jump-twin errors: ', (detected_jtwe / count_jtwe)*100, '%')

# generating a barplot of the detection rates
labels = ['Single-digit\nerror', 'Transposition\nerror', 'Twin error',
          'Jump-transposition\nerror', 'Jump-twin\nerror']
rates = [detected_sde/sim, detected_tre/sim, detected_twe/count_twe,
         detected_jtre/sim, detected_jtwe/count_jtwe]

plt.bar(labels, rates, width=0.05)
plt.ylabel('Detection Rate')
plt.ylim(0, 1)
plt.show()

# printing matrices showing the detection rate of every error type
print('Transposition error: \n', detec_trans / total_trans)
print('Twin error: \n', detec_twin / total_twin)
print(detec_jump_twin)
print(total_jump_twin)
print('Jump-twin error: \n', detec_jump_twin / total_jump_twin)

```

B

Python code for computational analysis of MOD97-10 algorithm

```
import random
import numpy as np
import string
import matplotlib.pyplot as plt

def iban_to_int(iban: str) -> str:
    """ converts an iban with alphanumerical characters to an integer """
    rearranged = iban[4:] + iban[:4]

    convert = ''
    for a in rearranged:
        if a.isdigit():
            convert += a
        else:
            convert += str(ord(a)-55)

    return convert

def mod97_validity(number: str) -> bool:
    """ check validity of converted iban """

    return int(number) % 97 == 1

def create_valid_iban() -> str:
    """ creates a valid iban number (Netherlands) """
    country = 'NL'

    bank = ''
    for i in range(3):
        bank += chr(random.randint(65,90))

    extra = str(random.randint(0,9))

    account = ''.join(str(random.randint(0, 9)) for _ in range(10))

    iban = country + '00' + extra + bank + account
```

```

    num = int(iban_to_int(iban))
    check_digits = 98 - (num % 97)

    return country + f"{check_digits:02d}" + extra + bank + account

def single_digit_error(number: str) -> str:
    """ alters one digit in the number to create a single-digit error """
    digits = list(number)

    position = random.randint(0, len(digits)-1)
    value = digits[position]

    new = (int(value) + random.randint(1,9)) % 10
    digits[position] = str(new)

    return ''.join(digits)

def two_adjacent_digit_error(number: str) -> (str, int, int):
    """ changes two adjacent digits to two new digits randomly """
    digits = list(number)

    position = random.randint(0, len(digits)-2)
    value_1, value_2 = digits[position], digits[position+1]

    new_1 = random.choice([x for x in range(10)])
    if new_1 == int(value_1):
        new_2 = random.choice([x for x in range(10) if x != int(value_2)])
    else:
        new_2 = random.choice([x for x in range(10)])

    digits[position], digits[position+1] = str(new_1), str(new_2)

    return ''.join(digits), int(value_1+value_2), int(str(new_1)+str(new_2))

def transposition_error(number: str) -> (str, int, int):
    """ swaps two non-identical, adjacent digits randomly """
    digits = list(number)

    while True:
        position = random.randint(0, len(digits)-2)
        if digits[position] != digits[position+1]:
            a, b = int(digits[position]), int(digits[position+1])
            digits[position], digits[position+1] = digits[position+1], digits[position]
            break

    return ''.join(digits), a, b

def twin_error(number: str) -> (str, int, int):
    """ replaces two identical, adjacent digits with another set of identical digits """
    digits = list(number)

    candidates = []

    for i in range(len(digits)-1):
        if digits[i] == digits[i+1]:
            candidates.append(i)

```

```

    if not candidates:
        return None, None, None

    position = random.choice(candidates)

    a = int(digits[position])
    new_value = (int(digits[position]) + random.randint(1,9)) % 10
    digits[position], digits[position+1] = str(new_value), str(new_value)

    return ''.join(digits), a, new_value

def single_char_error(iban: str) -> (str, int, int):
    """ creates error in IBAN, digit to digit, letter to letter """
    chars = list(iban)

    position = random.choice([i for i in range(len(chars)) if i not in (2, 3)])
    current = chars[position]

    if current.isdigit():
        new_char = random.choice(string.digits.replace(current, ''))
    else:
        new_char = random.choice(string.ascii_uppercase.replace(current, ''))

    chars[position] = new_char

    return ''.join(chars), char_to_value(current), char_to_value(new_char)

def char_to_value(ch: str) -> int:
    """ gives the numerical value of uppercase letters """
    if ch.isdigit():
        return int(ch)
    else:
        return ord(ch.upper()) - ord('A') + 10

def iban_transposition(number: str) -> (str, int, int):
    """ switches two characters in the iban """
    chars = list(iban)

    while True:
        position = random.randint(0, len(chars)-2)
        if chars[position] != chars[position+1]:
            a, b = char_to_value(chars[position]), char_to_value(chars[position+1])
            chars[position], chars[position+1] = chars[position+1], chars[position]
            break

    return ''.join(chars), a, b

def abc_cab(number: str) -> (str, int, int):
    """ simulates and error abc <-> cab of three digits """
    digits = list(number)

    position = random.randint(0, len(digits)-3)
    value1, value2, value3 = digits[position], digits[position+1], digits[position+2]

    digits[position] = value3

```

```

    digits[position+1] = value1
    digits[position+2] = value2

    return ''.join(digits), int(value3), int(value1+value2)

def iban_random_char_substitution(iban: str) -> (str, int, int):
    """ creates a single character errors, letter-to-letter, digit-to-digit,
    digit-to-letter, and letter-to-digit """
    chars = list(iban)

    position = random.choice([i for i in range(len(chars))])
    current = chars[position]

    alphabet = string.digits + string.ascii_uppercase
    new_char = random.choice([c for c in alphabet if c != current])

    chars[position] = new_char

    return ''.join(chars), char_to_value(current), char_to_value(new_char)

# initialising
sim = 1000000
detected_sde = 0
detected_tre = 0
detected_twe = 0
detected_double = 0
count_twe = 0
detected_char_error = 0
detected_trans_iban = 0
detected_abc = 0

detec_trans = np.zeros((10,10))
total_trans = np.zeros((10,10))
detec_twin = np.zeros((10,10))
total_twin = np.zeros((10,10))
detec_single_char = np.zeros((36,36))
total_single_char = np.zeros((36,36))
detec_double = np.zeros((100,100))
total_double = np.zeros((100,100))
detec_trans_iban = np.zeros((36,36))
total_trans_iban = np.zeros((36,36))
detec_abc = np.zeros((100,10))
total_abc = np.zeros((100,10))
detec_sub = np.zeros((36,36))
total_sub = np.zeros((36,36))

for i in range(sim):
    iban = create_valid_iban()
    num = iban_to_int(iban)
    sde = single_digit_error(num)
    tre, a_tre, b_tre = transposition_error(num)
    twe, a_twe, b_twe = twin_error(num)
    double, a_double, b_double = two_adjacent_digit_error(num)
    abc, a_abc, bc_abc = abc_cab(num)

    char_error, a_char, b_char = single_char_error(iban)

```

```

char_num = iban_to_int(char_error)
trans_error, a_trans, b_trans = iban_transposition(iban)
trans_num = iban_to_int(trans_error)
sub, a_sub, b_sub = iban_random_char_substitution(iban)
sub_num = iban_to_int(sub)

if not mod97_validity(sde):
    detected_sde += 1

if not mod97_validity(tre):
    detected_tre += 1
    detec_trans[a_tre, b_tre] += 1
    total_trans[a_tre, b_tre] += 1
if mod97_validity(tre):
    total_trans[a_tre, b_tre] += 1

if twe is not None:
    count_twe += 1
    total_twin[a_twe, b_twe] += 1
    if not mod97_validity(twe):
        detected_twe += 1
        detec_twin[a_twe, b_twe] += 1

if not mod97_validity(char_num):
    detected_char_error += 1
    detec_single_char[a_char, b_char] += 1
    total_single_char[a_char, b_char] += 1
if mod97_validity(char_num):
    total_single_char[a_char, b_char] += 1

if not mod97_validity(double):
    detected_double += 1
    detec_double[a_double, b_double] += 1
    total_double[a_double, b_double] += 1
if mod97_validity(double):
    total_double[a_double, b_double] += 1

if not mod97_validity(trans_num):
    detected_trans_iban += 1
    detec_trans_iban[a_trans, b_trans] += 1
    total_trans_iban[a_trans, b_trans] += 1
if mod97_validity(trans_num):
    total_trans_iban[a_trans, b_trans] += 1

if not mod97_validity(abc):
    detected_abc += 1
    detec_abc[bc_abc, a_abc] += 1
    total_abc[bc_abc, a_abc] += 1
if mod97_validity(abc):
    total_abc[bc_abc, a_abc] += 1

if not mod97_validity(sub_num):
    detec_sub[a_sub, b_sub] += 1
    total_sub[a_sub, b_sub] += 1
if mod97_validity(sub_num):
    total_sub[a_sub, b_sub] += 1

```

```

# printing the detection rate of different error types
print('Single digit errors: ', (detected_sde / sim)*100, '%')
print('Transposition errors: ', (detected_tre / sim)*100, '%')
print('Twin errors: ', (detected_twe / count_twe)*100, '%')
print('Single character error iban: ', (detected_char_error / sim)*100, '%')
print('IBAN transposition error: ', (detected_trans_iban / sim)*100, '%')
print('abc error: ', (detected_abc / sim)*100, '%')
print('Two adjacent digits error', (detected_double / sim)*100, '%')

# printing which specific errors are detected
ratios = detec_double / total_double
mask = ((ratios != 1) & (~np.isnan(ratios)))
indices = np.where(mask)
print('Two-digit substitution errors')
for i, j in zip(indices[0], indices[1]):
    print(f"({i},{j}) -> {ratios[i,j]}")
print()

ratios_trans = detec_trans_iban / total_trans_iban
mask_trans = ((ratios_trans != 1) & (~np.isnan(ratios_trans)))
indices_trans = np.where(mask_trans)
print('IBAN transposition errors')
for i, j in zip(indices_trans[0], indices_trans[1]):
    print(f"({i},{j}) -> {ratios_trans[i,j]}")
print()

ratios_abc = detec_abc / total_abc
mask_abc = ((ratios_abc != 1) & (~np.isnan(ratios_abc)))
indices_abc = np.where(mask_abc)
print('Three-digit errors (abc <-> cab)')
for i, j in zip(indices_abc[0], indices_abc[1]):
    print(f"({i},{j}) -> {ratios_abc[i,j]}")

# generating heatmap
labels = [str(i) for i in range(10)] + list(string.ascii_uppercase)

data = np.ma.masked_invalid(ratios)
plt.figure(figsize=(6,5))
plt.imshow(data)
plt.colorbar(label="Detection rate")
plt.title("Two-digit substitution detection rates")
plt.xlabel("cd")
plt.ylabel("ab")
plt.show()

data2 = np.ma.masked_invalid(ratios_trans)
plt.figure(figsize=(10,9))
plt.imshow(data2)
plt.colorbar(label="Detection rate")
plt.xticks(range(36), labels)
plt.yticks(range(36), labels)
plt.title("Detection rates (n=18)")
plt.xlabel("x_{k+1}")
plt.ylabel("x_k")
plt.show()

```

```
rates_sub = detec_sub / total_sub
plt.figure(figsize=(10,8))
plt.imshow(rates_sub)
plt.colorbar(label='Detection rate')
plt.xticks(range(36), labels)
plt.yticks(range(36), labels)
plt.xlabel("Substituted by")
plt.ylabel("Original character")
plt.title("Single-character error detection rates")
plt.show()
```

Bibliography

- [1] starline. (n.d.). *Floating realistic credit cards concept background*. Magnific. https://www.magnific.com/free-vector/floating-realistic-credit-cards-concept-background_22451352.htm#from_element=cross_selling__vector
- [2] sergnewa. (n.d.). *Black icon operator holds a barcode scanner hand*. Magnific. https://www.magnific.com/premium-vector/black-icon-operator-holds-barcode-scanner-hand_35612765.htm#from_element=cross_selling__vector
- [3] *IBAN nummer*. (n.d.). e-Boekhouden.nl. <https://cdn.e-boekhouden.nl/pub/iban-nummer.webp>
- [4] *BNP Paribas Fortis/VISA Card*. (n.d.). BNP Paribas Fortis. <https://newsroom.bnpparibasfortis.be/bnp-paribas-fortis-and-visa-launch-visa-debit-in-belgium>
- [5] International Organization for Standardization. (2017). *Identification cards — Identification of issuers — Part 1: Numbering system* (ISO Standard No. 7812-1:2017). <https://www.iso.org/standard/70484.html>
- [6] International Organization for Standardization. (2020). *Financial services — International bank account number (IBAN) — Part 1: Structure of the IBAN* (ISO Standard No. 13616-1:2020). <https://www.iso.org/standard/81090.html>
- [7] International Organization for Standardization. (2017). *Information and documentation — International Standard Book Number (ISBN)* (ISO Standard No. 2108:2017). <https://www.iso.org/standard/65483.html>
- [8] GS1 US. (n.d.). *What is a GTIN?* <https://www.gs1us.org/upcs-barcodes-prefixes/what-is-a-gtin>
- [9] United Nations. (2020). Introduction to the Luhn Algorithm. In *United Nations Economic Commission for Europe* (ECE/TRANS/SC.2/PIRRS/2020/4).
- [10] Wachira, W., Waweru, K., & Nyaga, L. (2015). *Transposition Error Detection in Luhn's Algorithm*. https://www.academia.edu/19957955/Transposition_Error_Detection_in_Luhn_s_Algorithm
- [11] Kamaku, W., Wachira, W., Eric, W., & Gallian, J. (2017). Twin Error Detection in Luhn's Algorithm. *International Journal of Science and Research (IJSR)*, 6(7), 308–310. <https://doi.org/10.21275/art20174899>
- [12] Wachira, L. W. (2016). *Error Detection and Correction on the Credit Card Number using Luhn Algorithm*. <http://ir.jkuat.ac.ke/handle/123456789/2418>
- [13] International Organization for Standardization. (2003). *Information technology — Security techniques — Check character systems* (ISO Standard No. 7064:2003). <https://www.iso.org/standard/31531.html>
- [14] Kamaku, P. W. (2013). *Error Detection and Correction in the International Standard Book Number*. <http://ir.jkuat.ac.ke/handle/123456789/1617>
- [15] Gijswijt, D. C. (2023). *Algebra 1*. Delft University of Technology.
- [16] Ireland, K., & Rosen, M. (1990). A classical introduction to modern number theory. In *Graduate texts in mathematics*. <https://doi.org/10.1007/978-1-4757-2103-4>
- [17] Daigle, J. (n.d.). *Math 322: Number Theory*. Occidental College.

- [18] Kirtland, J. (2000). Identification numbers and check digit schemes. In *American Mathematical Society eBooks*. <https://doi.org/10.1090/clrm/018>
- [19] Hussein, K. W., Fazlida, M. S. N., Mahmod, R., & Taufik, A. S. M. (2013). Enhance luhn algorithm for validation of credit cards numbers. *Universiti Putra Malaysia Institutional Repository (Universiti Putra Malaysia)*. <https://www.ijcsmc.com/docs/papers/July2013/V2I7201373.pdf>
- [20] Abdul, J. M. (2018, April 24). Mathematics behind Credit/Debit card numbering. *Medium*. <https://medium.com/@ma.juber/mathematics-behind-credit-debit-card-numbering-340bf68d27d2>
https://unece.org/fileadmin/DAM/trans/doc/2020/sc2/ECE-TRANS-SC.2-EG_PIRRS-2020-4e_01.pdf
- [21] Luhn, H. P. (1960). *Computer for verifying numbers* (Patent No. US2950048DA). USPTO.
- [22] IBAN. (2026, April 16). *IBAN Examples, Structure and Length*. <https://www.iban.com/structure>
- [23] Wikipedia contributors. (2026, April 7). *International Bank Account number*. Wikipedia. https://en.wikipedia.org/wiki/International_Bank_Account_Number
- [24] ISO. (n.d.). *ISO 3166 - Country Codes*. <https://www.iso.org/iso-3166-country-codes.html>
- [25] SWIFT. (2025). *IBAN registry*. <https://www.swift.com/sites/default/files/files/iban-registry-v101.pdf>
- [26] *IBAN Checker: Validate & check IBAN number for errors*. (n.d.). IBAN. <https://www.iban.com/iban-checker>
- [27] GS1 Netherlands. (n.d.). *The standard GS1 article code*. <https://www.gs1.nl/en/products-services/digital-identity/ean-13gtin/>
- [28] GS1. (2026). *GS1 General Specifications Standard*. <https://ref.gs1.org/standards/genspecs/>
- [29] GS1 Belgium & Luxembourg. (n.d.). *Global Trade Item Number (GTIN)*. <https://www.gs1belu.org/en/global-trade-item-number-gtin>
- [30] GS1. (n.d.). *GS1 Company prefix*. <https://www.gs1.org/standards/id-keys/company-prefix>
- [31] GS1. (n.d.). *How to calculate a check digit manually - Services*. <https://www.gs1.org/services/how-calculate-check-digit-manually>