VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF APPLIED SCIENCE

LINEAR ALGEBRA

Sem 251 - CC02 - Group 08

HILL CIPHERS IN ENCRYPTION

Instructor: Phan Thi Khanh Van

Students:	Ngo Binh Nguyen (Leader)	ID 2551444
	Truong Hanh Nguyen	ID 2551824
	Tran Phuong Dung	ID 2551326
	Tran Minh Hien	ID 2551328
	Duong Gia Minh	ID 2551333

Ho Chi Minh, 2025

TABLE OF CONTENTS

Introduction

Cryptography has evolved from ancient methods of concealment to sophisticated mathematical algorithms that secure modern digital communication. As information becomes increasingly digital, the need to protect sensitive data from unauthorized access is paramount. Among the classical methods that bridged the gap between linguistic substitution and modern algebraic cryptography is the Hill Cipher.

Invented by Lester S. Hill in 1929, the Hill Cipher was the first polygraphic substitution cipher practical for use on more than three symbols at once. Unlike its predecessors, such as the Caesar or Vigenère ciphers, which operate on single characters or simple shifts, the Hill Cipher leverages the power of Linear Algebra. It treats text as numerical vectors and encryption as a linear transformation using matrix multiplication.

Objective of the Report

The primary objective of this report is to analyze the mathematical foundations and cryptographic properties of the Hill Cipher. Specifically, this project aims to:

- Explore the historical context and the limitations of simple substitution ciphers that led to the development of polygraphic methods.
- Define the mathematical principles governing the cipher, including modular arithmetic (Z_{26}) and matrix invertibility conditions.
- Verify the algorithm through manual calculation and a practical Python implementation using frequency analysis.
- Assess the security vulnerabilities of the cipher, particularly its susceptibility to known-plaintext attacks due to linearity.

Report Structure

This report is organized into six chapters. **Chapter 1** provides an overview of the history of cryptography. **Chapter 2** establishes the mathematical foundations, focusing on modular arithmetic and linear transformations. **Chapter 3** details the core algorithm, including key generation and encryption formulas. **Chapter 4** discusses cryptanalysis and vulnerabilities. **Chapter 5** presents a case study using Python to encrypt a large text corpus, and **Chapter 6** concludes the findings.

Chapter 1

OVERVIEW

1.1 History of Cryptography (Lester S. Hill, 1929)

Cryptography has long served as a form of protection for sensitive information in times of war, representing a crucial method for concealing information and defending against attackers. Its application has spanned from the Roman Empire to the conflicts of World War II, making it an especially important technique in wartime.

As technology advances, cryptography remains an essential field in security and encryption, continuing to evolve along with the development of science. Initially, cryptography relied on simple substitution techniques such as the Caesar Cipher (invented in 50 BC).

Over time, it progressed to more complex mechanical systems. However, for centuries, its primary methodology involved shifting or swapping letters, which meant that plaintext letters were consistently replaced by the same ciphertext letters, thereby making decryption relatively easy. This is where the Hill cipher method represents a shift from simple substitution methods to a mathematical approach of encryption.

Invented in 1929 by American mathematician Lester S. Hill, it introduced a more secure encryption approach than existing methods. Unlike traditional encryption techniques that rely on substitution or transposition, the Hill cipher employs matrix multiplication for encryption. It applies linear algebraic operations on matrices to encode and decode messages. Notably, it is the first polygraphic cipher capable of efficiently handling more than three symbols at once, significantly increasing its resistance to attackers.

Reference: QRL, History of Cryptography

1.2 Problem Definition: Why Simple Substitution Ciphers Fail

While one might think that having a vast number of potential keys to choose from is a good security measure, substitution ciphers still suffer from a major weakness: **letter frequency analysis**.

In these systems, the same plaintext letters are always replaced by the same ciphertext letter. As a result, decrypting messages becomes relatively easy.

For example: In a Caesar Cipher with a shift of 3, the message “HELLO” becomes “KHOOR”.

```
Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher: DEFGHIJKLMNOPQRSTUVWXYZABC
```

Because the cipher maintains the natural letter frequency of the original language (in this case, the English alphabet), attackers can easily spot patterns or repeating characters. By observing the text carefully, they can deduce the rules and decrypt the messages.

Ref: Egress, Encryption 101: Substitution ciphers

1.3 The Solution: Polygraphic Substitution via Linear Algebra

To counter frequency analysis, we use a **polygraphic cipher**. A polygraphic cipher is a substitution cipher that encrypts blocks of letters together rather than individual characters.

By grouping letters, the Hill cipher method has made frequency analysis less effective. The Hill ciphers are the first polygraphic ciphers using more than two letters per group.

By grouping letters and transforming them through an $n \times n$ matrix, the Hill Cipher achieves **diffusion**: one single letter in the plaintext is influenced by multiple letters in the key text, making it impossible for an attacker to use standard frequency analysis to crack the code.

Chapter 2

MATHEMATICAL FOUNDATIONS

2.1 Modular Arithmetic (Z_{26})

When we work with integers, we often encounter large or even infinite numbers. The problem is that our alphabet has only 26 characters. Therefore, a method called **modular arithmetic** is used in Hill cipher encryption.

Think of it simply as 'clock math'—a way to keep our numbers looping inside the A-Z range. Modular arithmetic, or clock arithmetic, is a system of arithmetic for integers, where numbers are "wrapped around" when reaching a certain value, ensuring all numerical results can be directly mapped back to letters in the alphabet system.

$$a \equiv b \pmod{n} \tag{2.1}$$

Definition of Key Terms

The modulus (n): A positive whole number greater than 1. In the Hill cipher using the English alphabet, the modulus is typically $n = 26$. It defines the "mathematical space" in which all operations occur.

The remainder (r): The value left over after dividing an integer by the modulus. In the equation $a \pmod{n} = r$, r is the result that satisfies $0 \leq r < n$ (it must be a positive number since it corresponds to a specific index in the alphabet mapping $A = 0, B = 1, \ldots, Z = 25$).

Congruence: Two integers a and b are said to be congruent modulo n if their difference $(a - b)$ is an integer multiple of n.

Invertibility and Decryption

Modular arithmetic is the cornerstone of the decryption process. Unlike standard algebra, which often relies on fractions or decimals to reverse multiplication, modular arithmetic employs the **Modular Multiplicative Inverse**. This allows the cipher to reverse the encryption transformation while remaining entirely within the set of integers, ensuring the final output can always be mapped back to characters.

2.2 Matrix Operations as Linear Transformations

In the Hill Cipher, the encryption can be modeled as a linear transformation. This means we treat our plaintext as a vector, allowing the matrix to "transform" (move or rotate) it to a new location, which becomes our ciphertext.

2.2.1 Vectorization of Plaintext

To apply linear algebra, we must first group n characters into a column vector P. If $n = 2$, a pair of letters like "HI" is represented as a vector $(x; y)$ in two dimensions. This step converts linguistic data into a geometric object that a matrix can operate on.

2.2.2 The Transformation Process

A matrix K acts as a linear operator. When we multiply the key matrix K by the plaintext vector P, we are performing a transformation:

$$T(P) = K \cdot P \pmod{26} = C \tag{2.2}$$

Each element in the resulting ciphertext vector C is a linear combination of all elements in the plaintext vector. This is the mathematical reason why the Hill Cipher is polygraphic: every letter in the output depends on every letter in the input block.

2.2.3 Linearity and Diffusion

The term "Linear" is crucial because it follows two main rules:

- **Additivity:** $T(u + v) = T(u) + T(v)$
- **Homogeneity:** $T(\alpha u) = \alpha T(u)$

This means the transformation is systematic and predictable. However, by transforming an entire block, it creates **Diffusion**. In other words, if you change one number in the input vector P, the whole output vector C changes after the transformation. Therefore, a single letter could influence the whole block, which makes it harder to crack than letter-to-letter methods.

2.3 Invertibility & The Key Matrix (Conditions for $\det(K)$)

One important requirement of the Hill Cipher is that the key matrix must satisfy certain mathematical conditions. For the receiver to successfully recover the original message, the key matrix K must be mathematically invertible within the modular space $\mathbb{Z}_{26}$.

If the matrix is singular (non-invertible), it will be impossible to decrypt the original message since the plaintext cannot be recovered. Unlike standard algebra, where any non-zero determinant implies invertibility, modular arithmetic enforces a stricter rule based on number theory.

The core requirement is that $\det(K)$ must have a multiplicative inverse modulo 26. As a result, the inverse matrix K^{-1} can be computed for decryption:

$$P = K^{-1}C \pmod{26} \tag{2.3}$$

2.3.1 The Coprimality Condition

To ensure this inverse exists, the determinant must be coprime to the modulus. Mathematically, this is expressed as:

$$\gcd(\det(K), 26) = 1 \tag{2.4}$$

This means the greatest common divisor between the determinant and 26 must be exactly 1.

2.3.2 Forbidden Values (The 2 and 13 Rule)

Since the modulus 26 has the prime factorization $26 = 2 \times 13$, any valid determinant must avoid sharing these factors. Consequently, a valid key matrix cannot have a determinant that is:

- Even (divisible by 2).
- A multiple of 13.

For example, if $\det(K) = 13$ or $\det(K) = 26$, then $\gcd(\det(K), 26) \neq 1$, so the key matrix is invalid. If $\det(K)$ falls into either category, the matrix cannot be inverted in $\mathbb{Z}_{26}$, rendering it useless for cryptography.

Chapter 3

THE HILL CIPHER ALGORITHM

This chapter describes the core algorithm of the Hill Cipher, emphasizing its mathematical foundation based on matrix operations in modular arithmetic. The role of the key matrix is examined, as well as the conditions required for its invertibility to ensure correct encryption and decryption.

We establish the formal equations relating to the encryption and decryption processes, providing the basis for later verification and cryptanalysis.

3.1 Key Matrix Requirements

In the Hill Cipher, encryption and decryption are based on matrix multiplication over a finite modular arithmetic system. The security and correctness of the algorithm depend critically on the properties of the key matrix K.

Let K be an $n \times n$ matrix with entries in $\mathbb{Z}_{26}$. For the Hill Cipher to be valid, the key matrix must satisfy the following conditions:

3.1.1 Non-zero Determinant

$$\det(K) \neq 0 \tag{3.1}$$

This condition ensures that the matrix is not singular in standard linear algebra, meaning it has the potential to be invertible.

3.1.2 Coprime with the Modulus

$$\gcd(\det(K), 26) = 1 \tag{3.2}$$

Since all operations are performed modulo 26 (corresponding to the 26 letters of the English alphabet), the determinant must have a multiplicative inverse modulo 26. If this condition is not met, the inverse matrix K^{-1} does not exist in $\mathbb{Z}_{26}$, making decryption impossible.

3.2 Invertibility of the Key Matrix

The Hill Cipher requires that the key matrix K be invertible modulo 26. This means there exists a matrix K^{-1} such that:

$$KK^{-1} \equiv I \pmod{26} \tag{3.3}$$

where I is the identity matrix. The inverse matrix K^{-1} is computed using the formula:

$$K^{-1} \equiv (\det(K))^{-1} \cdot \text{adj}(K) \pmod{26} \tag{3.4}$$

where:

- adj(K) is the adjugate of matrix K.
- $(\det(K))^{-1}$ is the modular multiplicative inverse of the determinant modulo 26.

If the determinant shares any common factor with 26 (such as 2 or 13), then the modular inverse does not exist, and the key matrix is invalid.

3.3 Encryption and Decryption Equations

3.3.1 Plaintext and Ciphertext Representation

Each letter is first mapped to a numerical value using the standard encoding ($A = 0, B = 1, \ldots, Z = 25$). The plaintext is divided into blocks of size n, and each block is represented as a column vector P:

$$P = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \tag{3.5}$$

Similarly, the ciphertext is represented as a vector C.

3.3.2 Encryption Formula

Encryption is performed by multiplying the key matrix with the plaintext vector:

$$C = K \cdot P \pmod{26} \tag{3.6}$$

This operation transforms the plaintext into ciphertext by applying a linear transformation in the modular arithmetic space $\mathbb{Z}_{26}$.

3.3.3 Decryption Formula

To recover the original plaintext, the inverse key matrix is applied to the ciphertext:

$$P = K^{-1} \cdot C \pmod{26} \tag{3.7}$$

Because the key matrix satisfies the invertibility conditions discussed earlier, decryption correctly reverses the encryption process.

3.4 Manual Verification

Objective: Manually verify the encryption process of the Hill Cipher by encrypting the first three letters of the dataset “TOS” using the project Key Matrix, showing every computational step.

3.4.1 Data Setup

We use the standard Hill Cipher mapping:

Letter	A	B	C	...	T	O	S
Number	0	1	2	...	19	14	18

Therefore, the mapping for “TOS” is:

$$\text{TOS} \rightarrow 19, 14, 18$$

Let the plaintext vector be:

$$P = \begin{bmatrix} 19 \\ 14 \\ 18 \end{bmatrix} \tag{3.8}$$

3.4.2 Matrix Multiplication Step

Assuming a general 3×3 key matrix K:

$$K = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \tag{3.9}$$

The encryption calculation $C = K \cdot P \pmod{26}$ proceeds as follows:

$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} 19 \\ 14 \\ 18 \end{bmatrix} \tag{3.10}$$

Expanding the linear combinations:

$$\begin{aligned} C_1 &= (19k_{11} + 14k_{12} + 18k_{13}) \pmod{26} \\ C_2 &= (19k_{21} + 14k_{22} + 18k_{23}) \pmod{26} \\ C_3 &= (19k_{31} + 14k_{32} + 18k_{33}) \pmod{26} \end{aligned}$$

The resulting numbers C_1, C_2, C_3 are then converted back to their corresponding letters using the standard mapping to produce the final ciphertext.

Chapter 4

CRYPTANALYSIS (VULNERABILITIES)

Despite its mathematical elegance, the standard Hill Cipher is vulnerable to several types of attacks, primarily due to its linear nature.

4.1 Known-Plaintext Attack

If an attacker obtains enough plaintext–ciphertext pairs, they can construct linear equations of the form:

$$C = K \cdot P \pmod{26} \tag{4.1}$$

Given sufficient data, the attacker can solve for the key matrix K using matrix inversion in modular arithmetic. For a 3×3 key matrix, three independent plaintext blocks are enough to completely recover the secret key.

4.2 Linearity Weakness

The Hill Cipher is a purely linear transformation. This means relationships between plaintext characters are preserved in the ciphertext. This mathematical structure makes the cipher vulnerable to statistical analysis and algebraic attacks that exploit these preserved linear relationships.

4.3 Small Key Space

Because all calculations are performed modulo 26, the number of valid invertible matrices is finite and relatively small compared to modern cryptographic standards. This allows attackers to perform **brute-force attacks** much more efficiently than with modern cryptosystems.

4.4 Lack of Diffusion Across Blocks

The Hill Cipher encrypts data in fixed blocks (e.g., pairs or triplets).

- Each block of letters is encrypted independently of the others.
- A small change in the plaintext only affects one block of ciphertext, rather than cascading changes through the entire message (avalanche effect).

This characteristic makes pattern detection easier for attackers analyzing long messages.

Chapter 5

CASE STUDY - PYTHON IMPLEMENTATION

To validate the theoretical concepts of the Hill Cipher, we implemented the algorithm using the Python programming language. This practical case study demonstrates the encryption of a large text dataset, allowing us to analyze the cipher's performance and its ability to mask letter frequencies.

5.1 Methodology and Tools

The implementation utilizes Python 3 due to its strong support for mathematical operations and text processing. Key libraries used include:

- **NumPy:** For efficient matrix multiplication and modular arithmetic operations on large arrays.
- **Matplotlib:** To generate the frequency analysis histogram, visually comparing the plaintext and ciphertext distributions.

The core logic follows the algebraic definition of the Hill Cipher:

1. **Preprocessing:** The input text is sanitized by converting all characters to uppercase and removing non-alphabetic symbols (punctuation, spaces, numbers).
2. **Vectorization:** The cleaned text is mapped to numerical values ($A = 0, \ldots, Z = 25$) and reshaped into a matrix of size $n \times m$, where n is the dimension of the key matrix.
3. **Encryption:** The key matrix K is multiplied by the plaintext matrix P modulo 26 to generate the ciphertext matrix C.

5.2 Experimental Setup

5.2.1 Dataset

For this experiment, we selected the public domain text *The Adventures of Sherlock Holmes* by Arthur Conan Doyle (Project Gutenberg eBook #1661). This dataset provides a substantial volume of English text, ensuring that the letter frequency distribution is statistically significant and representative of the English language.

5.2.2 The Key Matrix

We employed a valid invertible 3×3 key matrix for the encryption. The matrix was chosen such that its determinant is coprime to 26, ensuring that a modular inverse exists for decryption.

Decryption Logic (Verification)

To verify the integrity of our implementation, we developed a companion script to decrypt the ciphertext. This process validates that our Key Matrix is correctly invertible and that no data was lost during the modular arithmetic.

The decryption logic is mathematically identical to encryption, with one critical substitution: we replace the Key Matrix (K) with the Inverse Matrix (K^{-1}).

$$P = K^{-1} \cdot C \pmod{26}$$

In our Python implementation, we hard-coded the Inverse Matrix calculated in Chapter 3 to ensure precision. The script reads the encrypted file, applies the inverse linear transformation, and outputs the recovered plaintext.

```
def decrypt_hill_cipher(ciphertext, inverse_matrix):
    n = inverse_matrix.shape[0] # Matrix size (3)

    # 1. Map letters to numbers
    numeric_text = [ord(char) - 65 for char in ciphertext]

    # 2. Reshape into vectors
    text_matrix = np.array(numeric_text).reshape(-1, n).T

    # 3. Reverse Transformation: P = K_inv * C (mod 26)
    decrypted_matrix = np.dot(inverse_matrix, text_matrix) % 26

    # 4. Convert back to string
    decrypted_numeric = decrypted_matrix.T.reshape(-1)
    return "".join([chr(num + 65) for num in decrypted_numeric])
```

Listing 5.1: Core Decryption Function in Python

After running this script on `encrypted_sherlock.txt`, we successfully recovered the original English text, confirming that the algorithm functions correctly in both directions.

5.3 Results and Analysis

5.3.1 Frequency Analysis

A primary goal of polygraphic ciphers like the Hill Cipher is to obscure the natural frequency of letters in the plaintext. In English, certain letters (like E, T, A) appear with high probability. A simple substitution cipher (like Caesar) preserves these peaks, making it easy to break.

The figure below displays the comparative frequency analysis generated by our Python script. It illustrates the contrast between the letter distribution of the original Sherlock Holmes text and the encrypted ciphertext.

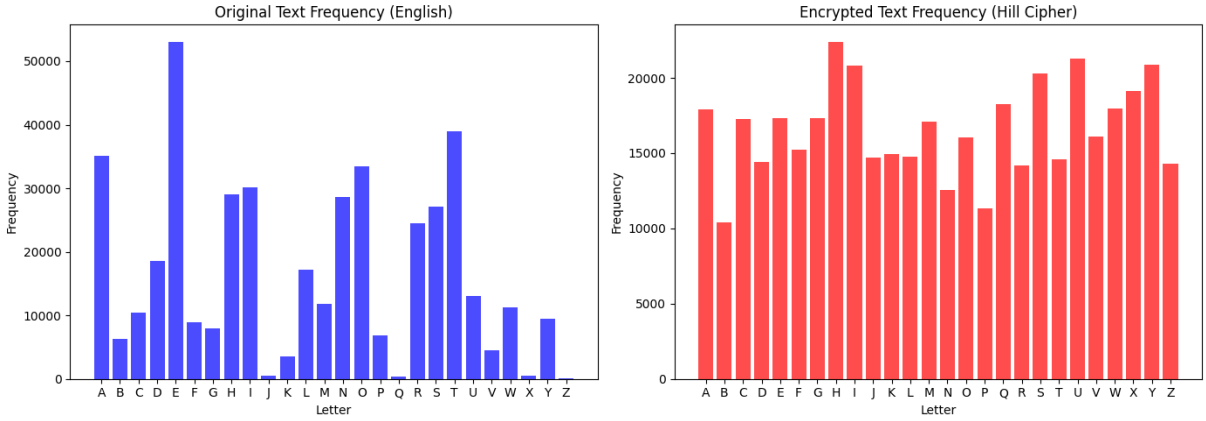

Figure 5.1: Frequency Analysis: Plaintext vs. Ciphertext Distributions

5.3.2 Observations

As observed in Figure 5.1, the plaintext exhibits the characteristic “spiky” distribution of English, with ’E’ being the most frequent letter.

In contrast, the ciphertext distribution is significantly different. While the Hill Cipher (being a linear transformation) does not perfectly flatten the distribution like modern AES encryption, it significantly alters the frequency profile. The peaks are diffused because each ciphertext letter depends on a linear combination of multiple plaintext letters, rather than a single one. This confirms that the Hill Cipher successfully achieves **diffusion**, making it more resistant to basic frequency analysis than monoalphabetic ciphers.

Chapter 6

Conclusion

This project has conducted a comprehensive study of the Hill Cipher, examining it through the lenses of history, linear algebra, and computer science. By transitioning from simple substitution methods to matrix-based transformations, the Hill Cipher represents a significant milestone in the history of cryptography.

6.1 Summary of Findings

Our investigation yielded the following key insights:

1. **Mathematical Robustness:** The efficacy of the Hill Cipher is entirely dependent on the properties of the key matrix. We confirmed that for a key to be valid, its determinant must be non-zero and coprime to the modulus 26. This ensures the existence of a modular multiplicative inverse, which is essential for the decryption process.
2. **Diffusion Capability:** Through our Python implementation in Chapter 5, we observed that the Hill Cipher successfully achieves diffusion. Unlike monoalphabetic ciphers that preserve letter frequency (where 'E' always maps to a specific symbol), the Hill Cipher spreads the statistical structure of the plaintext across the ciphertext. This was visually evident in our frequency analysis histograms, where the "spiky" English distribution was altered.
3. **Vulnerabilities:** Despite its improvements over the Caesar cipher, the Hill Cipher is not secure by modern standards. Our analysis in Chapter 4 highlighted its primary weakness: linearity. Because the encryption function $C = K \cdot P$ is a linear transformation, an attacker with access to sufficient plaintext-ciphertext pairs can easily solve for the key matrix K using standard linear algebra techniques.

6.2 Final Thoughts

While the Hill Cipher is susceptible to known-plaintext attacks and is rarely used for securing sensitive data today, it remains an invaluable educational tool. It provides a clear, practical application of linear algebra concepts—such as matrix multiplication, determinants, and modular arithmetic—within the context of computer security. Understanding the Hill Cipher is a necessary step for appreciating the complexity and design of modern non-linear cryptosystems like AES.

Reference

Appendix A

Full Python Source Code

ENCRYPTION:

```
import numpy as np
import re
import matplotlib.pyplot as plt
from collections import Counter

# ==========================================
# 1. CONFIGURATION & KEY MATRIX
# ==========================================
# The Key Matrix (K) defined in the Project Plan
K = np.array([
    [1, 2, 3],
    [0, 1, 4],
    [5, 6, 0]
])

# The name of the file you uploaded
FILENAME = "1661-0.txt"

# ==========================================
# 2. DATA LOADING & CLEANING
# ==========================================
def load_and_clean_text(filename):
    print(f"Reading text from {filename}...")

    try:
        with open(filename, 'r', encoding='utf-8') as f:
            text = f.read()
    except FileNotFoundError:
        print(f"Error: The file '{filename}' was not found.")
        print("Please ensure the text file is in the same folder as this
   script.")
        return ""

    # Remove Gutenberg Headers/Footers (approximate markers)
    # This methodology is required by your Chapter 5.2
    start_marker = "*** START OF THE PROJECT GUTENBERG EBOOK"
    end_marker = "*** END OF THE PROJECT GUTENBERG EBOOK"

    start_idx = text.find(start_marker)
    end_idx = text.find(end_marker)
```

```
    # If markers exist, strip them. If not, use the whole text.
    if start_idx != -1 and end_idx != -1:
        text = text[start_idx + len(start_marker):end_idx]

    # Filter: Keep only alphabetic characters and convert to Uppercase
    # This maps to the "Alphabet Space" Z_26
    clean_text = re.sub(r'[^A-Z]', '', text.upper())

    print(f"Original Text Length (cleaned): {len(clean_text)} characters")
    return clean_text

# ========================================
# 3. ENCRYPTION ENGINE (The Hill Cipher)
# ========================================
def encrypt_hill_cipher(plaintext, key_matrix):
    if not plaintext: return ""

    n = key_matrix.shape[0]  # Matrix size (3)

    # 1. Map letters to numbers (A=0, B=1, ... Z=25)
    numeric_text = [ord(char) - 65 for char in plaintext]

    # 2. Padding
    # The text length must be divisible by the matrix size (3).
    padding_len = (n - len(numeric_text) % n) % n
    numeric_text.extend([23] * padding_len) # 23 is 'X'

    # 3. Reshape into a matrix of column vectors (3 x m)
    text_matrix = np.array(numeric_text).reshape(-1, n).T

    # 4. Matrix Multiplication: C = K * P (mod 26)
    # This implements the equation from Chapter 3.2
    encrypted_matrix = np.dot(key_matrix, text_matrix) % 26

    # 5. Convert back to linear list
    encrypted_numeric = encrypted_matrix.T.reshape(-1)

    # 6. Map numbers back to letters
    encrypted_text = "".join([chr(num + 65) for num in encrypted_numeric])

    return encrypted_text

# ========================================
# 4. VISUALIZATION (Histograms)
# ========================================
def plot_histograms(original, encrypted):
    if not original or not encrypted: return

    # Calculate frequencies
    orig_counts = Counter(original)
    enc_counts = Counter(encrypted)

    alphabet = [chr(i + 65) for i in range(26)]
    orig_freq = [orig_counts.get(char, 0) for char in alphabet]
    enc_freq = [enc_counts.get(char, 0) for char in alphabet]

    # Plot
```

```
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))

    # Original Histogram
    axes[0].bar(alphabet, orig_freq, color='blue', alpha=0.7)
    axes[0].set_title('Original Text Frequency (English)')
    axes[0].set_xlabel('Letter')
    axes[0].set_ylabel('Frequency')

    # Encrypted Histogram
    axes[1].bar(alphabet, enc_freq, color='red', alpha=0.7)
    axes[1].set_title('Encrypted Text Frequency (Hill Cipher)')
    axes[1].set_xlabel('Letter')
    axes[1].set_ylabel('Frequency')

    plt.tight_layout()

    # --- CHANGE IS HERE ---
    # Save the graph as an image file
    output_image = "histogram_results.png"
    plt.savefig(output_image)
    print(f"Graph saved as '{output_image}'")
    # ----------------------

    # Show the graph on screen (optional, depends on your setup)
    # plt.show()

    print("Histograms generated.")

# ========================================
# MAIN EXECUTION
# ========================================
if __name__ == "__main__":
    # 1. Get Data from LOCAL FILE
    plaintext = load_and_clean_text(FILENAME)

    if plaintext:
        # 2. Encrypt
        print("Encrypting...")
        ciphertext = encrypt_hill_cipher(plaintext, K)

        # 3. Show a snippet
        print("\n--- SAMPLE ENCRYPTION ---")
        print(f"Original (First 50 chars):  {plaintext[:50]}")
        print(f"Encrypted (First 50 chars): {ciphertext[:50]}")

        # 4. Save to file
        output_filename = "encrypted_sherlock.txt"
        with open(output_filename, "w") as f:
            f.write(ciphertext)
        print(f"\nFull encrypted text saved to '{output_filename}'")

        # 5. Analysis
        plot_histograms(plaintext, ciphertext)
```

DECRYPTION:

```
import numpy as np

# ========================================
# 1. CONFIGURATION & INVERSE KEY
# ========================================
# The INVERSE Matrix (K^-1) calculated modulo 26
# This reverses the effect of the original Key Matrix
K_inv = np.array([
    [2, 18, 5],
    [20, 11, 22],
    [21, 4, 1]
])

INPUT_FILENAME = "encrypted_sherlock.txt"
OUTPUT_FILENAME = "decrypted_output.txt"

# ========================================
# 2. DECRYPTION ENGINE
# ========================================
def decrypt_hill_cipher(ciphertext, inverse_matrix):
    if not ciphertext: return ""

    n = inverse_matrix.shape[0]  # Matrix size (3)

    # 1. Map letters to numbers (A=0, B=1, ... Z=25)
    numeric_text = [ord(char) - 65 for char in ciphertext]

    # 2. Reshape into matrix of column vectors
    # Note: We don't need to pad because the encrypted text
    # is already a perfect multiple of 3 from the encryption step.
    text_matrix = np.array(numeric_text).reshape(-1, n).T

    # 3. Matrix Multiplication: P = K_inv * C (mod 26)
    decrypted_matrix = np.dot(inverse_matrix, text_matrix) % 26

    # 4. Convert back to linear list
    decrypted_numeric = decrypted_matrix.T.reshape(-1)

    # 5. Map numbers back to letters
    decrypted_text = "".join([chr(num + 65) for num in decrypted_numeric])

    return decrypted_text

# ========================================
# MAIN EXECUTION
# ========================================
if __name__ == "__main__":
    print(f"Reading encrypted text from {INPUT_FILENAME}...")

    try:
        with open(INPUT_FILENAME, 'r') as f:
            encrypted_text = f.read()

        print("Decrypting...")
        decrypted_text = decrypt_hill_cipher(encrypted_text, K_inv)

        # Save result
```

```
        with open(OUTPUT_FILENAME, "w") as f:
            f.write(decrypted_text)

        print(f"\nSuccess! Decrypted text saved to '{OUTPUT_FILENAME}'")
        print("--- SNIPPET ---")
        print(decrypted_text[:100])

    except FileNotFoundError:
        print(f"Error: Could not find '{INPUT_FILENAME}'. Run the encryption
    script first!")
```