Linear Algebra and the Hill Cipher

Lauren Hardy, Vyankatesh Chandge, Matthew Safar

April 21, 2023

Contents

1 Introduction

Linear algebra is one of the main foundations of cryptology. Everything from AES, to lattices, to hash functions and more all rely on this branch of mathematics. In this paper we focus on the Hill Cipher, which is a classical cipher based on multiplying matrices together for encryption and decryption. We will investigate the optimal size of these matrices that minimize the number of computations needed for encryption and decryption but provide the most amount of security. Furthermore, this project will illustrate how linear algebra is directly applied to other areas of mathematics, and it will ultimately lead to a deeper understanding of both linear algebra and cryptology.

2 Basic Theory and Terminology

2.1 Cryptography Methodology and Terminology

Ever since humans have been able to communicate there has been a need to protect information from unwanted recipients. Throughout history, people have invented various methods to encrypt messages so that only the intended recipient can interpret them. One of the oldest methods includes shaving someone's head, writing a message on the shaven head, and then waiting until the hair has regrown to send the messenger off [6]. As time went on however, technology advanced and so did the techniques of hiding information.

In today's world we send and receive a large amount of information. It is quite easy for someone to get hold of this information, so it becomes imperative that this data is sufficiently encrypted. The process of designing systems in the effort of encrypting of information is called cryptography and its study of these algorithms is referred to as cryptology.

There are two types of cryptographic systems: asymmetric and symmetric. An asymmetric system uses two different keys - one for encryption and one for decrytion. A symmetric key system uses the same key for encryption and decryption. The Hill Cipher (which will be discussed in Section 2.3) is a symmetric cryptosystem.

For a symmetric cryptosystem, there is first a sender; we will refer to the sender as Alice. The message that Alice wants to send to Bob, the receiver, is called the plaintext. In order for Alice to send a message to Bob without anyone else being able to read it, Alice and Bob must agree on a key beforehand. Then, the plaintext is encrypted (through a predetermined method) with this key. The encrypted plaintext message is refered to as the ciphertext message. This ciphertext message appears as gibberish to anyone who does not know the hidden key. Once Bob receives the ciphertext, he will use the same key that Alice used for encrypting to decrypt the ciphertext back into the plaintext (again, through some predetermined method).

It is important that the Alice and Bob's mutually agreed-upon key is always kept private. It is possible that there is an eavesdropper, who we will call Eve. Suppose that Eve obtains some of the data that Alice has been sending to Bob. As long as Alice and Bob's key is kept private, Eve will only see the ciphertext. Eve can read the ciphertext, but without the knowledge of the key, she is unable to understand the message. All she has is some random data.

For the sake of convenience, throughout this paper we refer to the sender as Alice, the receiver as Bob, and the attacker/eavesdropper as Eve.

2.2 Encoding

Alice wants to send a message to Bob in English. However, for most encryption methods, we must work with numbers. To do this, we simply map the letters from a to z onto numbers from 0 to 25 as visualized in the table below.

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12
n	o	p	q	r	s	t	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

2.3 Hill Cipher Basics

The Hill Cipher is named after Lester S. Hill. In 1929, Hill invented a machine which could operate on three symbols at once as a polygraphic cipher.

The Hill Cipher is a type of *block cipher*; it operates on a block of symbols of some given size and returns another block of symbols of the same size. This type of cipher performs a substitution on strings of symbols instead of just a single symbol, as would be seen in a substitution cipher, such as the shift cipher (where the letters are encoded by shifting the alphabet by a set value). In particular, the Hill Cipher is a block cipher that interprets blocks of the message as vectors and uses a linear transformation to act on these vectors of symbols [9].

A useful aspect of the Hill Cipher that is that it creates *diffusion*. Diffusion refers to a cipher's ability to spread out messages, which means that similar values in the plaintext correspond to very different values in the ciphertext. Another desirable property for ciphers is the ability to make the plaintext difficult to obtain from the ciphertext alone, called *confusion*. Intuitively, it corresponds to how convoluted the relationship is between the plaintext, key, and ciphertext. In particular, since the Hill Cipher is based on linear transformations which are relatively easy to invert, it has poor confusion. However, block ciphers (like the Hill Cipher) that offer diffusion are often paired with ciphers that offer better confusion to produce an encryption that is overall very secure [8].

For the Hill Cipher, there is a key matrix, K, a plaintext matrix, P, and an encoding size, n (e.g. 26 for the alphabet). Multiplying P and K together and then taking $\mod n$ of every index produces the encrypted ciphertext matrix, C, which is the same size as the plaintext matrix:

$$C = M_n(PK, n)$$

M_n is an operation on the matrix that applies the binary operation

$$\text{mod}(a_{ij}, n),$$

which returns the remainder of a_{ij} when divided by n, to every element of the matrix.

Decryption is then achieved by using the corresponding inverse key $K^{\%}$ and computing

$$P = M_n(CK^{\%}, n)$$

Therefore, both encryption and decryption are matrix multiplication operations.

In general, messages are in the form of a string and not a matrix. There are preprocessing steps associated with transforming the message into a matrix (see Section 4), and postprocessing steps where in addition to mapping numbers back to whatever data it represents in the encoding, the ciphertext is made back into a string of data rather than a matrix [9].

2.4 Inverses of Matrices Modulo n

When we refer to matrices modulo 26, we mean an operation which takes every element of a matrix and performs the modulo operation on it. Namely, if $M_n(\cdot)$ is the modulo operator $(\text{mod } n)$, then we define

$$(M_n(A))_{ij} = M_n(a_{ij})$$

This operation has several good properties[7], including

- We can define congruence between matrices (and vectors) by:

$$A \equiv B \pmod{n} \iff M_n(A) = M_n(B) \iff \forall i, j,\ a_{ij} \equiv b_{ij} \pmod{n}$$

- If A and B are $m \times k$ matrices and X, Y are $k \times l$ matrices with $A \equiv B \pmod{n}$ and $X \equiv Y \pmod{n}$, then

$$AX \equiv BY \pmod{n}$$

- If $A \equiv B \pmod{n}$ with A, B square, then $\det(A) \equiv \det(B) \pmod{n}$

An useful fact about modular arithmetic is the idea of a multiplicative modular inverse. We define x to be *multiplicative modular inverse* of a scalar a if $ax \equiv 1 \pmod{n}$. An example of this would be that 21 is the multiplicative modular inverse of 5 with modulus 26, since $5 * 21 \equiv 105 \equiv 1 \pmod{26}$ [9]. Here, we denote the modular multiplicative inverse of a as a^{-1}, in the context that a is an integer.

We define a square matrix A to be invertible mod n if there exists its matrix modular inverse, $A^{\%}$, that satisfies that for all $x \in \mathbb{Z}$

$$xA^{\%}A \equiv x \pmod{n}$$

Notice that if A is invertible, then $M_n(A)$ is also invertible with the same matrix modular inverse. Therefore, for the purposes of key generation, we need only to look at invertible matrices with values from 0 to $n-1$.

It is possible to find a relatively simple necessary and sufficient condition for a matrix to be invertible mod n. Namely, a matrix A is invertible mod n if and only if $\det(A)$ is coprime to n (i.e., that $\gcd(\det(A), n) = 1$). A proof for this is in the appendix.

Another result of the proof is that if the matrix $A^{\%}$ exists, it is given by

$$A^{\%} \equiv (\det(A))^{-1} \left(\det(A)A^{-1}\right) \pmod{n}$$

where recall that $(\det(A))^{-1}$ is the multiplicative modular inverse of an integer and A^{-1} is the matrix inverse of A.

2.4.1 Rounding Error

This expression for the Matrix modular inverse,

$$A^{\%} \equiv (\det(A))^{-1} \left(\det(A)A^{-1}\right) \pmod{n}$$

indicates that we generally need to compute both the inverse A^{-1} and the determinant $\det(A)$ of A to compute our matrix modular inverse $A^{\%}$. This introduces a source of floating point error, since while A is an integer matrix in this case, A^{-1} is not strictly an integer matrix and would need to be represented with floating point numbers in the computer. Even worse, calculating the inverse is notoriously unstable. The result $\det(A)A^{-1}$ must be accurate when rounded to the one's place for the mod operator to produce the correct result. For a 10×10 matrix to be used on a message with an ASCII encoding (128 characters), we might feasibly get a value $\det(A) = 127^{10}$. This would greatly magnify errors in the computed A^{-1}, and would result in the inverse key being incorrect, making decryption impossible.

We could attempt to avoid possible non-integer operations that come from calculating A^{-1} by instead calculating the adjugate of A (i.e. $\det(A)A^{-1}$) directly; although, this involves computing many determinants of the minors of A. Therefore, it is relatively computationally intensive ($\mathcal{O}(n^5)$), even when using something like the Bareiss Algorithm [2].

A better way to resolve this issue is to instead rely on an LU factorization of A. Given that

$$PA = LU$$

solve the following systems of linear congruences for the inverse $A^{\%}$ (which can be done in exact integer arithmetic):

$$\begin{aligned} LY &\equiv PI \pmod{n} \\ UA^{\%} &\equiv Y \pmod{n} \end{aligned}$$

This is examined more closely in the next section.

3 Key Generation

The only requirement for a key is that it must be invertible. As we have seen, this is when the determinant of the key, $\det(K)$, is coprime to the size of your encoding, in our case the size of our encoding is 26.

There are two methods for generating a key. Naively, we might generate a random matrix, check if it is invertible, and if invertible, then make if our key. To generate the decryption key, we would then compute

$$K^{\%} = (\det(K))^{-1} \det(K) K^{-1}$$

We call this *rejection sampling*, and it is sufficient for small matrices.

Instead, a better way to generate a key is to first generate three random matrices: a permutation matrix, P, a lower triangular matrix, L, with ones on the diagonal, and an upper triangular matrix, U, with elements on the diagonal that are all coprime to 26. Then the result

$$M_{26}(P^T LU)$$

is guaranteed to be invertible mod 26 (proof in the appendix). Also, we can easily find the inverse by solving the two triangular systems

$$\begin{aligned} LY &\equiv P \pmod{26} \\ UX &\equiv Y \pmod{26} \end{aligned}$$

in which $X = K^{\%}$.

Here, we make use of the fact that there are backwards and forwards substitution algorithms for congruences. Namely, since the diagonals of both L and U are guaranteed to have multiplicative inverses, we can use that

$$x + y \equiv z \pmod{n} \implies x \equiv z - y \pmod{n}$$

and

$$ax \equiv z \pmod{n} \implies x \equiv a^{-1} z \pmod{n}$$

to perform the substitution as we normally would when solving a system of linear congruences.

Using this method to generate keys guarantees a modular invertible matrix and allows an easy method for calculating that modular inverse. To begin exchanging encrypted messages, Alice would generate both the matrix K and $K^{\%}$ (taking on all of the computational cost), and then gives $K^{\%}$ to Bob. After that, Alice encrypts her messages to Bob by applying her matrix K and decrypts her messages from Bob by again applying the matrix K. Bob does the same, but with his key $K^{\%}$.

4 Padding Scheme

4.1 Traditional Method

For simplicity, thoughout this section and the rest of the paper, the plaintext matrix will be refered to as *Matrix P*, the key matrix as *Matrix K*, and the ciphertext matrix as *Matrix C*.

Matrix P is of size $m \times n$ and must be completely filled by Alice's message to Bob; there cannot be any empty indices. However, it is unlikely that the length of Alice's message would be exactly of size $m * n$. To prevent any empty indices in *Matrix P*, we must pad the message.

The traditional padding scheme for a Hill Cipher is to fill all of the empty indices after the message in *Matrix P* with an uncommon letter from the English alphabet, such as z, q or j (see below) [9].

$$\begin{bmatrix} a & t & t & a & c \\ k & i & s & a & t \\ d & a & w & n & z \end{bmatrix}$$

Example of traditional padding scheme in a Hill Cipher with plaintext: *attackisatdawn*

4.2 Attack on Traditional Method

As we manipulate the size of m and n in order to investigate the optimal size of *Matrix P* and *Matrix K*, it is very possible that at least one row of *Matrix P* would be a completely filled with padding. If this occurs, the system become susceptible to a weakened known plaintext attack.

A known plaintext attack is when an attacker has both a portion of the plaintext and its corresponding ciphertext. This attack would be somewhat weakened, as Eve, the attacker, would not be sure which letter is used for padding. However, Eve could reasonably assume the letter used is either q, z, or j, as these are the least commonly used letters in the English alphabet. Using this knowledge, Eve can then greatly limit the number of possible keys with the following method.

For simplicity, we will demonstrate the attack when $n = 4$. The following is what the system would look like:

$$\begin{bmatrix} pv & pv & pv & pv \end{bmatrix} \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} \end{bmatrix} \equiv \begin{bmatrix} a & b & c & d \end{bmatrix} \pmod{26}$$

System for weakened plaintext attack when $n = 4$
pv: padding value

In this scenario, Eve needs to determine each $x_{i,j}$ value and she has the values a, b, c, d. She would not be certain of what the pv value is, but she would assume that it is either q, z, j. To solve for each value $x_{i,j}$, Eve would then have the congruences:

$$\begin{aligned} pv * x_{1,1} + pv * x_{2,1} + pv * x_{3,1} + pv * x_{4,1} &\equiv a \pmod{26} \\ pv * x_{1,2} + pv * x_{2,2} + pv * x_{3,2} + pv * x_{4,2} &\equiv b \pmod{26} \\ pv * x_{1,3} + pv * x_{2,3} + pv * x_{3,3} + pv * x_{4,3} &\equiv c \pmod{26} \\ pv * x_{1,4} + pv * x_{2,4} + pv * x_{3,4} + pv * x_{4,4} &\equiv d \pmod{26} \end{aligned}$$

These simplify to:

$$\begin{aligned} pv(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) &\equiv a \pmod{26} \\ pv(x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2}) &\equiv b \pmod{26} \\ pv(x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3}) &\equiv c \pmod{26} \\ pv(x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4}) &\equiv d \pmod{26} \end{aligned}$$

Consider just one of these congruences:

$$pv(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) \equiv a \pmod{26}$$

This can be further simplified to:

$$pv(x) \equiv a \pmod{26}$$

The number of solutions to this congruence depends on the value of pv:

- If $pv = q = 16$, then there are then two solutions to the congruence:

  $$16(x) \equiv a \pmod{26}$$

  Therefore, there are two possibilities for what x could be, call these possibilities i and j. Then solve the congruences:

  $$(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) \equiv i \pmod{26}$$
  $$(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) \equiv j \pmod{26}$$

  There are 26^3 combinations of $x_{i,j}$ values for each simplified congruence (see appendix for proof). So in total, there are $2 * 26^3$ possibilities for each congruence of the form:

  $$16(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) \equiv a \pmod{26}$$

  There are 4 of these congruences, so in all, there are $2 * 26^3 * 4$ possible key matrices, or rather combinations of $x_{i,j}$ values.

- If $pv = j = 9$ or $pv = z = 25$ then the congruences take the form of one of the following:

  $$25(x) \equiv a \pmod{26}$$
  $$9(x) \equiv a \pmod{26}$$

  For either pv values, there is only one solution for x, call this value i. And without loss of generality, suppose that $pv = 25$. Then solve the congruence:

  $$(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) \equiv i \pmod{26}$$

  There are 26^3 possible combinations of $x_{i,j}$ values for this simplified congruence. So in total, there are 26^3 possibilities for each congruence of the form:

  $$25(x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1}) \equiv a \pmod{26}$$

  There are 4 of these congruences, so in all, there are $26^3 * 4$ possible key matrices, or rather combinations of $x_{i,j}$ values.

Combining these, Eve would have at most $4 * 26^3 + 4 * 26^3 + 2 * 4 * 26^3 = 16 * 26^3$ keys to test given that $pv = q, j,$ or z. This value is significantly smaller than the total number of keys possible, which is 26^{4^2} (as the key matrix is of size (4×4) and each index has 26 possible values)

For n arbitrary this value becomes:

$$n26^{n-1} + n26^{n-1} + 2n26^{n-1} = 4n * 26^{n-1}$$

To ensure that this attack is not possible for any m and n, we will implement an improved padding scheme.

4.3 Improved Padding Scheme

We must use different characters to pad, as using the same letter for padding gave Eve an advantage. If we use random letters for padding, Eve's attempt at guessing a section of plaintext becomes more futile; however in return, this may also make the padding indistinguishable from the message. To prevent confusion, the first and last letters of the padding will be fixed with q, as q is one of the least used characters in the English alphabet. This way, we may easily generate the padding string, and Bob will know which part of the message is the padding once he decodes the ciphertext. Furthermore, this limits Eve's abilities to assume letters in the plaintext.

For the new padding scheme, we first obtain the plaintext message (in all lowercase letters, for the sake of simplicity and uniformity), the value m, and the value n. Then calculate the length of the plaintext message. By subtracting the length of the message from the value $m * n$, we get the size of the padding string (p). Next, generate a string of random letters of size $p - 2$, leaving 2 spaces where we then append a q at the beginning and end of p. Now we have the padding which is then concatenated to the end of the message. The entire message is now of size $n * m$ and completely fills *Matrix P*.

In theory, if Eve is aware of this padding scheme, then she would know that the last letter in *Matrix C* corresponds to q. However, due to the way we encrypt our messages, which is not a one-to-one correspondence method, Eve cannot know for sure where the padding is. Nor can they predict the random letters in between the 2 q's. This greatly limits the ability for her to perform a known plaintext attack on the padding scheme, and is therefore more secure.

$$\begin{bmatrix} a & t & t & a & c \\ k & n & o & w & q \\ o & h & f & w & q \end{bmatrix}$$

Example of the improved padding scheme in a Hill Cipher with plaintext: *attacknow*

Once we have the complete *Matrix P*, we must convert this matrix into numbers, as detailed by the encoding map in Section 2.2). This the allows for easy matrix multiplication for both encryption and decryption.

5 Computational Complexity

5.1 Key Generation

The computational complexity of key generation depends on the method used to generate keys. However, in general, it is not computationally expensive. Let *Matrix K* be an $n \times n$ matrix and the encoding size be l.

Both methods discussed here will generate around n^2 random numbers. The time it takes to generate these numbers depends greatly on the method used. Pseudo-random numbers can be generated very quickly using any of the various random number generators. Cryptographically safe random numbers, i.e. numbers that are truly random, are more difficult to obtain since they must be based off of a random seed from a source that is inherently random (e.g. quantum, or something similar) These are more difficult to generate (and we don't do that in this project), so we do not worry about the computational cost of generating cryptographically safe random numbers.

If using the rejection sampling method, then one must check if the matrix is invertible. This requires calculating the determinant of a random matrix, which is around $\mathcal{O}(n^3)$ time (if using the method based on Gaussian Elimination, which most are, including the ubiquitous LAPACK implementation and the integer-exact *Bareiss Algorithm*[2], which was useful in testing performance of other methods of calculating the inverse key). However, the number of times the determinant needs to be calculated depends on how many times the generation fails to provide an invertible matrix. From elsewhere, we know that the proportion of invertible matrices, given that l has a prime factorization $l = \prod p_i^{k_i}$, is given by [7]:

$$\prod_i \left(\prod_{j=0}^{n-1} \left(1 - p_i^{j-n}\right) \right)$$

For $l = 26$, we find that the proportion of invertible matrices is around 25% for matrices of size 4 and up. This means that on average, we would need to test four matrices before finding one that is invertible. When l is hightly composite, the proportion of invertible matrices decreases.

On the other hand, with LU factorization, we find guaranteed keys at the simple cost of matrix multiplication between L and U (and a permutation, which can be efficiently done without floating point operations). This means, it is a guaranteed key on the order of $\mathcal{O}(n^3)$, which is less than the rejection sampling. This, in addition to the ease with which an inverse is calculated from an LU factorization, guided us to choose LU factorization as our preferred method of generating keys.

5.2 Encryption and Decryption

There are two types of computational complexities associated with encryption and decryption.

5.2.1 Numerical Computations

For encryption, Alice multiplies *Matrix P* by *Matrix K* to get *Matrix C*. For decryption, Bob multiplies *Matrix C* by the inverse of *Matix K* (which was precomputed) to get *Matrix P* . *Matrix K* matrix is of size $n \times n$ and *Matrix P* is of size $m \times n$, and the matrix multiplication is as follows:

$$(P)_{mn} * (K)_{nn} = (C)_{mn}$$
$$(C)_{mn} * (K^{-1})_{nn} = (P)_{mn}$$

The number of floating point operations (FLOPS) for multiplying two matrices of size $m \times n$ and $n \times p$ is [5]:

$$(2n - 1)mp$$

In our case *Matrix P* and *Matrix P* are of size $m \times n$ and $n \times n$. Therefore, we get the number of operations for encryption and decryption to be:

$$(2n - 1)mn \Rightarrow 2n^2m - mn$$

Clearly, the number of operations is of the order 3.

For better visualization of this equation, below is a table of the number of FLOPS for some n and m values:

n	m	FLOPS
4	4	112
4	25	700
10	20	3800
100	250	4975000

5.2.2 Memory

Each time matrix multiplication is performed, memory must be allocated in the RAM to temporarily store the matrices for computations. The amount of memory needed to store this is refered to as *Memory Operations*, or MEMOPS. For any two given matrices the amount of RAM needed is [5]:

$$2mn + n^2$$

Clearly, this is of the order of two. For better visualization of this equation, below is a table of MEMOPS values for various n and m:

n	m	MEMOPS
4	4	48
4	25	216
10	20	500
100	250	60000

The total number of MEMOPS is insignificant in comparison to the average frequency of million Hz of a typical RAM.

Next we consider the amount of space taken up by a matrix when it is loaded into RAM before calculations. For a double-precision matrix the space used in terms of Gigabytes is [10]:

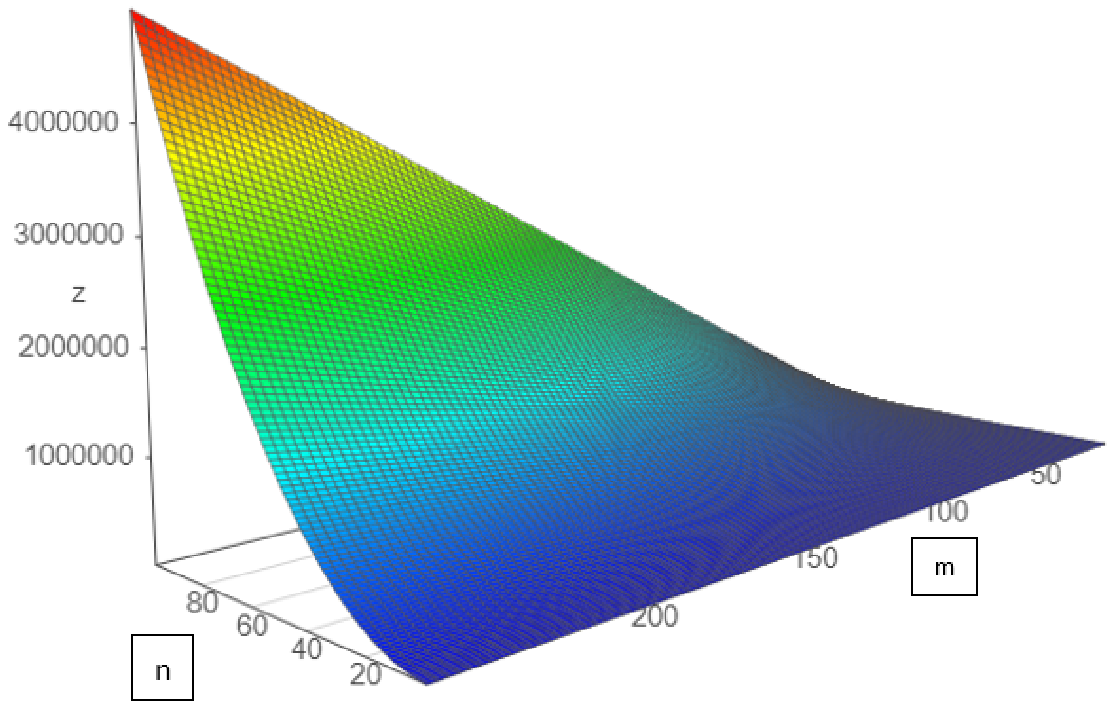
$$\frac{8mn}{10^9}$$

Again, for better visualization of this formula, see the table below with various values of m and n:

n	m	Storage
4	4	128 bytes
4	25	0.8KB
10	20	1.6KB
100	250	0.2MB
1000	1000	8MB

The storage cost of a matrix is not high enough to cause much concern, as the average RAM memory has much more available space at any given time.

We can now combine the cost of computing and storing the matrix by taking the summation of FLOPS and MEMOPS. For a typical range of values of m and n, we can plot the total computational cost as shown below:



5.3 Inverses

In our case, Alice and Bob precomputed both *Matrix K* and its inverse (see Section 3). They do not need to take any additional inverses. However, in order to perform a known plaintext attack, discussed above in Section 6.2, Eve certainly needs to take inverses. Specifically, Eve must invert an $n \times n$ section of *Matrix P*. To do this, she incurs a computational costs.

To find the inverse, Eve must first find the adjugate of the matrix she is inverting. Calculating the adjugate requires finding the cofactor of each element $a_{i,j}, 1 \leq i, j \leq n$ in the matrix. This requires calculating the determinant of a $(n-1) \times (n-1)$ matrix n^2 times. Additionally, we must calculate the sign of each cofactor by computing $(-1)^{i+j}$. This requires two operations. We must also calculate the determinant of the entire $n \times n$ matrix and divide this by one, call this value d. Lastly, we must multiply every value in the adjugate by d, which requires n^2 operations. Calculating the determinant of a $n \times n$ matrix requires $\frac{2n^3}{3}$ computations [4]. Using all of this information, we obtain the following equation:

$$n^2\left(\frac{2(n-1)^3}{3} + 2\right) + \frac{2n^3}{3} + 1 + n^2$$

Calculating the total computational cost for some values of n gives us:

n	FLOPS
4	380
10	49568
100	6469356668
250	643.2E+9
1000	664.6E+12

As the size of the key matrix increases, the computations for calculating its inverse increases much faster than that of encryption and decryption, as this complexity is of the order n^5 plus some order of n^3.

6 Attacks on the Hill Cipher

6.1 Brute Force

There is no elegant way to attack a Hill Cipher if only the cipher text message is known; one must use brute force [9].

There is a finite number of keys. Namely in an $n \times n$ matrix, there are n^2 number of indices and 26 possible values for each index. Therefore, there are 26^{n^2} different keys.

For a brute force method, *Matrix C* must be multiplied by each possible key value. Each of these multiplications consist of multiplying a $m \times n$ matrix by a $n \times n$ matrix. By Section 5.2, we know that each multiplication will require $2 * n^2 * m - n * m$ operations. Therefore, the total number of operations for a brute force attack is [3]:

$$26^{n^2}(2n^2m - nm)$$

.

6.2 Known Plaintext Attack

The Hill Cipher is difficult to attack if Eve only knows the ciphertext. However, if Eve has some plaintext and the corresponding ciphertext, the Hill Cipher key is easily recoverable. This attack is called a known plaintext attack [9]. To best demonstrate how this attack works, we will use an example:

Suppose Eve has the entire plaintext message, *crypto*, and the entire ciphertext message, *hsvqnw*. Further suppose that she knows that $n = 2$. Once Eve translates the letters into number, this is the system:

$$\begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \equiv \begin{bmatrix} 7 & 18 \\ 21 & 16 \\ 13 & 22 \end{bmatrix} \pmod{26}$$

Since the key is of size $(n \times n)$ we only need n rows of the plaintext and ciphertext. Taking the first n rows of the plaintext and ciphertext, Eve would first try to solve the system:

$$\begin{bmatrix} 2 & 17 \\ 24 & 15 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \equiv \begin{bmatrix} 7 & 18 \\ 21 & 16 \end{bmatrix} \pmod{26}$$

However, the determinant of the plaintext matrix is $-378 \equiv 12 \pmod{26}$. Since the $\gcd(12, 26) \neq 1$, this matrix is not invertible $\pmod{26}$.

Eve would then try another combination of the plaintext rows (with corresponding ciphertext rows), such as the system:

$$\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \equiv \begin{bmatrix} 7 & 18 \\ 13 & 22 \end{bmatrix} \pmod{26}$$

This plaintext message is invertible $\pmod{26}$. After premultiplying both sides of the system with the inverse of the plaintext message, Eve would have:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \equiv \begin{bmatrix} 10 & 25 \\ 5 & 20 \end{bmatrix} \begin{bmatrix} 7 & 18 \\ 13 & 22 \end{bmatrix} \equiv \begin{bmatrix} 5 & 2 \\ 9 & 10 \end{bmatrix} \pmod{26}$$

So Eve has found the key used for encryption. If she did not know the entire message, but had the entire ciphertext message, she could now decrypt the entire message.

This attack works flawlessly if any combination of n of the m rows of the plaintext matrix results in an invertible matrix. There are a total of $\binom{m}{n}$ ways to choose the rows to test, and then $n!$ ways to arrange the chosen rows. So in total, Eve would have $n!\binom{m}{n}$ combinations available.

There are $2^n \prod_{i=1}^{n}(1 - \frac{1}{2^i})13^n \prod_{i=1}^{n}(1 - \frac{1}{13^i})$ invertible matrices $\bmod 26$ and a total of 26^{n^2} matrices $\bmod 26$. The percentage of matrices $\bmod 26$ is then

$$\frac{2^n \prod_{i=1}^{n}(1 - \frac{1}{2^i})13^n \prod_{i=1}^{n}(1 - \frac{1}{13^i})}{26^{n^2}} * 100$$

For most n, this number is greater than or approximately 25%. The expected number of combinations that Eve would have to test before finding a combination that is invertible is then 4, as $25\% = \frac{1}{4}$ and the distribution of invertible matrices is assumed to be uniformed [7].

Testing whether or not a combination is invertible means we must find the determinant, which requires $\frac{2n^3}{3}$ computations [4].

Once we find a combination that is invertible, we then have to take the inverse, which requires $n^2(\frac{2(n-1)^3}{3} + 3) + \frac{2n^3}{3} + 1$ floating point operations, given that the determinant of the entire matrix must be calculated (see Section 5.3). Since we just calculated the determinant in order to check whether or not the matrix is invertible, taking the inverse in this case will only require $n^2(\frac{2(n-1)^3}{3} + 3) + 1$ floating point operations. Finally, we must multiply this inverted plaintext matrix by the ciphertext message, which is the multiplication of two $n \times n$ matrices and requires $2n^3 - n^2$ computations (see Section 5.2 and let $m = n$).

So in total, Eve can expect to perform the following number of computations:

$$\frac{4}{3}(2n^3) + 2n^3 - n^2 + n^2\left(\frac{2(n-1)^3}{3} + 3\right) + 1 = \frac{14}{3}n^3 + n^2\left(\frac{2(n-1)^3}{3} + 2\right) + 1$$

If no combination of n plaintext rows is invertible, the number of possible keys is still grealty reduced. In this case, Eve needs to solve a linear system of congruences. For example, consider the first system from the known plaintext attack:

$$\begin{bmatrix} 2 & 17 \\ 24 & 15 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \equiv \begin{bmatrix} 7 & 18 \\ 21 & 16 \end{bmatrix} \pmod{26}$$

This translates into a linear system of congruences:

$$\begin{aligned} 2a + 17c &\equiv 7 \pmod{26} \\ 24a + 15c &\equiv 21 \pmod{26} \\ 2b + 17d &\equiv 18 \pmod{26} \\ 24b + 15d &\equiv 16 \pmod{26} \end{aligned}$$

To solve for a and c, consider the simplified congruence:

$$x + y \equiv 7 \pmod{26}$$

We know that there are 26 possible solutions to this congruence (see the appendix). So then there are 26 values for a and c such that:

$$\begin{aligned} 2a &\equiv x \pmod{26} \\ 17c &\equiv y \pmod{26} \end{aligned}$$

Once we have found these 26 values, we can then further reduce the number of possible keys by testing the values of a and c in the second congruence:

$$24a + 15c \equiv 21 \pmod{26}$$

The number of possible solutions at this stage will vary, but for this specific instance, there are only 2 solutions for a and c that satisfy both congruences. The same approach can be used to determine possible solutions for b and d. Therefore, even though inverting n rows of the plaintext message may not be invertible, the number of possible keys that Eve must try is greatly decreased.

It is important to note as well that even if Eve does not have any bit of the plaintext, there are techniques and analytical methods she can use to make educated guesses about the plaintext. These may include, but are not limited to, noticing patterns in the ciphertext message or supposing that the message begins with the words *Dear* or *To whom it may concern*. She can then use these conjectures to carry out this attack on the system. Therefore, the known plaintext attack is almost always possible.

7 Analysis

To allow for a better understanding of the computational complexity formulas found throughout this paper, Table 1 (below) displays the number of floating point operations for each function at various n and m values.

As seen in Table 1, brute force requires the most operations out of all the functions. However, to fully understand how this brute force function compares to the others, these values from Table 1 must be converted into computer run time. To convert each value, we must first calculate the average number of FLOPS (floating point operations per second) for a computer.

University of California Berkeley analyzed the CPU performance of 12,318 of different computers. In total, these computers are capable of 502.25 TeraFLOPS. Therefore, on average a computer is capable of around $\frac{502.25e+12}{12318} = 40e + 9$ FLOPS, or 40 GFLOPS [1].

Using this information, the values in Table 2 are then determined by dividing each value from Table 1 by $40e + 9$.

		Number of Computations			
n	**m**	**Encryption/ Decryption**	**Inverse**	**Known Plaintext Attack**	**Brute Force**
	1	1			2.60e+1
	10	10			2.60e+2
1	50	50	5	8	1.30e+3
	100	100			2.60e+3
	500	500			1.30e+4
	1	6			2.74e+6
	10	60			2.74e+7
2	50	300	21	49	1.37e+8
	100	600			2.74e+8
	500	3000			1.37e+9
	1	15			8.14e+13
	10	150			8.14e+14
3	50	750	94	193	4.07e+15
	100	1500			8.14e+15
	500	7500			4.07e+16
	1	28			1.22e+24
	10	280			1.22e+25
4	50	1400	380	620	6.11e+25
	100	2800			1.22e+26
	500	14000			6.11e+26
	1	45			1.07e+37
	10	450			1.07e+38
5	50	2250	1226	1701	5.33e+38
	100	4500			1.07e+39
	500	22500			5.33e+39
	1	190			5.97e+143
	10	1900			5.97e+144
10	50	9500	49568	53468	2.99e+145
	100	19000			5.97e+145
	500	95000			2.99e+146
	1	4.95e+3			#NUM!
	10	4.95e+4			#NUM!
50	50	2.48e+5	1.96e+8	1.97e+8	#NUM!
	100	4.95e+5			#NUM!
	500	2.48e+6			#NUM!
	1	1.99e+4			#NUM!
	10	1.99e+5			#NUM!
100	50	9.95e+5	6.47e+9	6.47e+9	#NUM!
	100	1.99e+6			#NUM!
	500	9.95e+6			#NUM!
	1	5.00e+5			#NUM!
	10	5.00e+6			#NUM!
500	50	2.50e+7	2.07e+13	2.07e+13	#NUM!
	100	5.00e+7			#NUM!
	500	2.50e+8			#NUM!

Table 1: Computational Complexity for Various n and m Values
(Numbers larger than 1.79e+308 cannot be displayed properly and are replaced by NUM!)

Time to Run in Computer (Seconds)							
n	**m**	**Encrypt/Decrypt**	**Inverse**	**K.P.A.**	**K.P.A. (Hours)**	**Brute Force**	**Brute Force (Yrs)**
	1	2.50e-11				6.50e-10	2.06e-17
	10	2.50e-10				6.50e-9	2.06e-16
1	50	1.25e-9	1.25e-10	2.00e-10	5.56e-14	3.25e-8	1.03e-15
	100	2.50e-9				6.50e-8	2.06e-15
	500	1.25e-8				3.25e-7	1.03e-14
	1	1.50e-10				6.85e-5	2.17e-12
	10	1.50e-9				6.85e-4	2.17e-11
2	50	7.50e-9	5.25e-10	1.23e-9	3.40e-13	3.43e-3	1.09e-10
	100	1.50e-8				6.85e-3	2.17e-10
	500	7.50e-8				3.43e-2	1.09e-9
	1	3.75e-10				2.04e+3	6.46e-5
	10	3.75e-9				2.04e+4	6.46e-4
3	50	1.87e-8	2.35e-9	4.83e-9	1.34e-12	1.02e+5	3.23e-3
	100	3.75e-8				2.04e+5	6.46e-3
	500	1.88e-7				1.02e+6	3.23e-2
	1	7.00e-10				3.05e+13	9.68e+5
	10	7.00e-9				3.05e+14	9.68e+6
4	50	3.50e-8	9.50e-9	1.55e-8	4.31e-12	1.53e+15	4.84e+7
	100	7.00e-8				3.05e+15	9.68e+7
	500	3.50e-7				1.53e+16	4.84e+8
	1	1.12e-9				2.66e+26	8.45e+18
	10	1.12e-8				2.66e+27	8.45e+19
5	50	5.62e-8	3.06e-8	4.25e-8	1.18e-11	1.33e+28	4.22e+20
	100	1.12e-7				2.66e+28	8.45e+20
	500	5.63e-7				1.33e+29	4.22e+21
	1	4.75e-9				1.49e+133	4.73e+125
	10	4.75e-8				1.49e+134	4.73e+126
10	50	2.38e-7	1.24e-6	1.34e-6	3.71e-10	7.46e+134	2.37e+127
	100	4.75e-7				1.49e+135	4.73e+127
	500	2.38e-6				7.46e+135	2.37e+128
	1	1.24e-7				#NUM!	#NUM!
	10	1.24e-6				#NUM!	#NUM!
50	50	6.19e-6	4.90e-3	4.92e-3	1.37e-6	#NUM!	#NUM!
	100	1.24e-5				#NUM!	#NUM!
	500	6.19e-5				#NUM!	#NUM!
	1	4.98e-7				#NUM!	#NUM!
	10	4.97e-6				#NUM!	#NUM!
100	50	2.49e-5	1.62e-1	1.62e-1	4.50e-5	#NUM!	#NUM!
	100	4.98e-5				#NUM!	#NUM!
	500	2.49e-4				#NUM!	#NUM!
	1	1.25e-5				#NUM!	#NUM!
	10	1.25e-4				#NUM!	#NUM!
500	50	6.24e-4	5.18e+2	5.18e+2	1.44e-1	#NUM!	#NUM!
	100	1.25e-3				#NUM!	#NUM!
	500	6.24e-3				#NUM!	#NUM!

Table 2: Running Time for Various n and m values with a 2ghz Speed Processor
(Numbers larger than 1.79e+308 cannot be displayed properly and are replaced by NUM!)

Encryption and decryption are very fast in a computer. Even when $n = m = 500$, the functions take less than a second to run. Thus, Alice and Bob are able to encrypt and decrypt messages of any size without any major run-time constraints.

A known plaintext attack takes longer to execute than encryption and decryption, as it requires taking an inverse in addition to matrix multiplication. However, even with this additional computation, a known plaintext attack is still fast. For small n values, the attack takes merely seconds to complete. Furthermore, even if n is much larger, say $n = 500$, this attack still takes less than 15 minutes. Therefore, there is no reasonable value for n that would prevent a known plaintext attack. In fact, the speed at which the known plaintext attack occurs, especially for large n values, is one of the main reasons the Hill Cipher is not used for securing data.

At all n values, the brute force attack has a much higher execution time than any of the other functions. When $n \leq 3$, this attack requires less than a year. However, as soon as $n = 4$, the attack takes just under 1 million years to run. Therefore, this attack is widely infeasible. Eve, Alice, and Bob would all die before it fully executes.

If no other attacks except brute force were possible on the Hill Cipher, a key matrix of size of only 4×4 would be extremely secure. Furthermore, a key size of 4×4 allows for encryption and decryption to occur within nanoseconds.

8 Conclusion

The Hill Cipher is a classical cryptographic algorithm that relies on matrix multiplication for encryption and decryption. The number of computations required for encryption and decryption is rather minimal, which allows these functions to run incredibly fast. If the Hill Cipher was only susceptible to a brute force attack, the optimal key matrix size is 4×4. This key matrix size would allow for encryption and decryption to complete within nanoseconds. Furthermore, on a key matrix of this size it would take almost one million years for a brute force attack to fully run. In reality however, the Hill Cipher is easily susceptible to the fast-running known plaintext attack. Therefore, there is no true optimal value of n that makes this cipher secure. The Hill Cipher must not be used as the sole means of securing data.

References

[1] Nov. 2021. URL: https://setiathome.berkeley.edu/cpu_list.php.

[2] Erwin H. Bareiss. "Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination". In: *Mathematics of Computation* 22.103 (1968), pp. 565–578. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2004533.

[3] Alex Barter. *Cryptanalysis of Hill cipher.* Apr. 2020. URL: https://alexbarter.com/cryptanalysis/breaking-hill-cipher/.

[4] Sarah Birdsong. 2010. URL: https://webpages.uncc.edu/~sjbirdso/linalg-spr10/projects/p195%5C%20-%5C%20det%5C%20&%5C%20flops.pdf.

[5] LAFF Linear Algebra - Foundations to Frontiers (www.ulaff.net). *1.4.1.1 LAFF: Counting flops and memops.* Aug. 2017. URL: https://www.youtube.com/watch?v=pba8XFy0qbw.

[6] Christopher McFadden. *11 Cryptographic Methods That Marked History: From the Caesar Cipher to Enigma Code and Beyond.* July 2018. URL: https://interestingengineering.com/11-cryptographic-methods-that-marked-history-from-the-caesar-cipher-to-enigma-code-and-beyond.

[7] Jeffrey Overbey, William Traves, and Jerzy Wojdylo. "On the keyspace of the hill cipher". In: *Cryptologia* 29.1 (2005), pp. 59–72. ISSN: 15581586. DOI: 10.1080/0161-110591893771.

[8] C. E. Shannon. "Communication theory of secrecy systems. 1945." In: *M.D. computing : computers in medical practice* 15.1 (1998), pp. 707–710. ISSN: 07246811.

[9] Wade Trappe and Lawrence C. Washington. *Introduction to cryptography: With coding theory.* Pearson Prentice Hall, 2006.

[10] Rick Wicklin. *How much RAM do I need to store that matrix?* Apr. 2014. URL: https://blogs.sas.com/content/iml/2014/04/28/how-much-ram-do-i-need-to-store-that-matrix.html.

9 Appendix

9.1 Proofs

9.1.1 Necessary and Sufficient Condition for Matrix Mod Invertible

A matrix A is invertible if and only if $\det(A)$ is coprime to n (i.e., that $\gcd(\det(A), n) = 1$).

Proof: ($\Rightarrow$) Assume that A is modular invertible. Then there exists a matrix modular inverse $A^{\%}$ such that for all x,

$$xA^{\%}A \equiv x \pmod{n}.$$

Namely, this means that

$$IA^{\%}A \equiv I \pmod{n}.$$

Taking the determinant,

$$\det(A^{\%})\det(A) = \det(A^{\%}A) \equiv 1 \pmod{n}$$

This is true only if $\det(A)$ has a multiplicative modular inverse, which in turn is only true if $\det(A)$ and n are coprime.

($\Leftarrow$) Assume that $\gcd(\det(A), n) = 1$. Then, necessarily $\det(A) \neq 0$, so A has an inverse A^{-1}. A^{-1} is not necessarily an integer matrix, but we know because of the cofactor method of finding the inverse that $\det(A)A^{-1}$ is. And in fact,

$$x(\det(A)A^{-1})A = x\det(A)$$

Since $\gcd(\det(A), n) = 1$, we know that $\det(A)$ has a multiplicative modular inverse, and we have that

$$x(\det(A))^{-1}(\det(A)A^{-1})A \equiv x \pmod{n}$$

The matrix $(\det(A))^{-1}(\det(A)A^{-1})$, is our matrix modular inverse.

9.1.2 Coprimes to n on Diagonal of U Guarantees that a Key is Modular Invertible

This is a simple proof involving the determinant. Let $K = P^TLU$ be an $m \times m$ matrix and let n be the size of our encoding. Let the diagonal of U contain elements where $gcd(u_{ii}, n) = 1$ for all $i = 1, \dots, m$.

First, notice that

$$\det(K) = \det(P^T)\det(L)\det(U) = \pm\prod_{i=1}^{m} u_{ii}$$

Claim: If $x_1, \dots, x_m$ are numbers coprime to n, then $\prod_{i=1}^{m} x_i$ is also coprime to n.

Proof: Here, we use that the $\gcd(\cdot)$ function is multiplicative, so that

$$\gcd\left(\prod_{i=1}^{m} x_i, n\right) = \prod_{i=1}^{m}(\gcd(x_i, n)) = 1$$

since for each individual x_i, $\gcd(x_i, n) = 1$.

Claim: if x has a multiplicative modular inverse then $-x$ has a multiplicative modular inverse (and it is $n - x^{-1}$)

Proof:

$$\begin{aligned}(n - x^{-1})(-x) &= -nx + x^{-1}x \\ &\equiv x^{-1}x \pmod{n} \\ &\equiv 1 \pmod{n}\end{aligned}$$

So we have that

$$\gcd\left(\prod_{i=1}^{m} u_{ii}, n\right) = 1$$

and therefore

$$\pm\gcd\left(\prod_{i=1}^{m} u_{ii}, n\right) = 1$$

so that $\gcd(\det(K), n) = 1$ and K is guaranteed to be matrix modular invertible.

9.1.3 The number of solutions is 26^{n-1}

Claim: The number of solutions to the congruence $x_1 + x_2 + ... + x_n \equiv a \pmod{26}$, where a is a known and arbitrary value and $0 \le x_i \le 25, i \in [1, n]$, is 26^{n-1}.

Proof by induction on n:

Base Case: $n = 2$:
We must solve the congruence $x_1 + x_2 \equiv a$. Choose a value between 0 and 25 for x_1. There is then exactly one value for x_2 that is between 0 and 25 and will satisfy the congruence. There are 26 options for x_1. Therefore, there are $26 = 26^{2-1}$ possible solutions to the congruence.

Induction Case:
Suppose that the number of solutions to the congruence $x_1 + x_2 + ... + x_n \equiv a \pmod{26}$ is 26^{n-1}. Now consider the congruence $x_1 + x_2 + ... + x_n + x_{n+1} \equiv a \pmod{26}$. Observe that we may group the first n xs together:

$$(x_1 + x_2 + ... + x_n) + x_{n+1} \equiv a \pmod{26}$$

Let $x_1 + x_2 + ... + x_n = X$. Then the congruence simplifies to:

$$X + x_{n+1} \equiv a \pmod{26}$$

Choose a value between 0 and 25 for X, call this value b. Then we also have the congruence:

$$x_1 + x_2 + ... + x_n \equiv b \pmod{26}$$

There are 26^{n-1} possible solutions for this congruence, and there are 26 possible values for X. Thus there are a total of $26^{n-1} * 26 = 26^n$ possible solutions for the congruence $x_1 + x_2 + ... + x_n + x_{n+1} \equiv a \pmod{26}$.

9.2 Code

9.2.1 Padding Scheme

```
import random
import string

inp = input("Enter message: \n")
n = int(input("Enter 'n' "))
m = int(input("Enter 'm' "))
print(len(inp))


def padding(inp, n, m):
    inp=inp.replace(' ','')
    inp=inp.lower()
    if len(inp) > n * m:
        print('Input is too long')
    else:
        a = len(inp)
        b = n * m - a
        print(b)
        a=''.join(random.choice(string.ascii_lowercase) for i in range(b-2))
        a=a.center(b,"q")
        return inp+a


fun=padding(inp, n, m)
print(fun)
```

9.2.2 Basic Hill Cipher

Our basic cipher was implemented as a Python class with a constructor and methods ”encrypt” and ”decrypt”. The user can specify one of three encodings and the size of the key matrix. We also included the helper functions for modular back and forward substitution.

```
""" Cipher Class
Contains all of the functions associated with creating a cipher and then encrypting
and decrypting text.

Create a cipher instance to create a cipher with a specific key.
"""

import numpy as np

class HillCipher4:
    """ A Hill Cipher for encrypting and decrypting text
    This particular version uses LU factorization and stores an inverse key"""

    def __init__(self, size, *, encoding="alphabet"):
        self.encoding = encoding
        if self.encoding == "alphabet":
            self.encoding_size = 26
            self.return_string = True
        elif self.encoding == "image":
            self.encoding_size = 256
            self.return_string = False
        elif self.encoding == "ASCII":
            self.encoding_size = 128
            self.return_string = True
        else:
            raise ValueError("HillCipher doesn't know this particular encoding")

        self.keysize = size
        self.key, self.invkey = self.generate_key(size)

    # generates a key for the cipher. The key needs to be invertible mod n where
    # n is the the number of characters in the encoding (e.g. 26 for the alphabet)
    # Strategy
    #   Create an LU decomposition, which allows us to control the determinant through U
    # Then, finds the inverse key for the matrix by solving a system of congruences
    def generate_key(self, size):
        L = np.identity(size)
        U = np.zeros((size, size))
        P = np.random.permutation(size)
        P = L[P, :]
        diags = allowed_diags(self.encoding_size)
        for i in range(size-1):
            L[(i+1)::, i] = np.asfarray(np.random.randint(0, self.encoding_size-1, size-i-1)
    )
        for i in range(size):
            U[i, i] = diags[np.random.randint(0, len(diags))]
            U[i, i+1::] = np.asfarray(np.random.randint(0, self.encoding_size-1, size-i-1))
        key = np.mod(P.T @ L @ U, self.encoding_size)
        invkey = np.eye(size)
        invkey = mod_forward_sub(L, P @ invkey, self.encoding_size)
        invkey = mod_backward_sub(U, invkey, self.encoding_size)
        return key, invkey

    # forms the plaintext into blocks and pads according to simple padding scheme
    # of adding z's at the end
    def simple_pad(self, ptext):
        list_text = list(ptext)
        ncols = np.int(np.ceil(len(list_text)/self.keysize))
        ptext = np.zeros((self.keysize,ncols))
        for i in range(self.keysize*ncols):
            if i < len(list_text):
                ptext[np.unravel_index(i, ptext.shape, 'F')] = self.t2n_encoding(list_text[i
    ])
            else:
                ptext[np.unravel_index(i, ptext.shape, 'F')] = self.t2n_encoding('q')
        return ptext

    # encrypts the plaintext "ptext" using the matrix "key"
    def encrypt(self, ptext):
        padded_text = self.simple_pad(ptext)
        encrypted = np.mod(self.key @ padded_text, self.encoding_size)

        ctext = []
        for i in range(encrypted.size):
            ctext.append(self.n2t_encoding(encrypted[np.unravel_index(i, encrypted.shape, 'F
    ')]))
        if self.return_string:
            ctextstr= ""
            ctext = ctextstr.join(ctext)
        return ctext

    # applies the inverse key to decrypt a message (does not remove padding)
    def decrypt(self, ctext):
        ctext = self.simple_pad(ctext)
        unencrypted = np.mod(self.invkey @ ctext,self.encoding_size)
        ptext = []
        for i in range(unencrypted.size):
            ptext.append(self.n2t_encoding(unencrypted[np.unravel_index(i, unencrypted.shape
    , 'F')]))
        if self.return_string:
            ptextstr = ""
            ptext = ptextstr.join(ptext)
        return ptext

    def n2t_encoding(self, n):
        if self.encoding == "alphabet":
            t = chr(int(n) + ord('a'))
        elif self.encoding == "image":
            t = n
        elif self.encoding == "ASCII":
            t = chr(int(n))
        else:
            raise ValueError("How did you get here")
        return t

    def t2n_encoding(self, t):
        if self.encoding == "alphabet":
            n = ord(t) - ord('a')
        elif self.encoding == "image":
            n = t
        elif self.encoding == "ASCII":
            n = ord(t)
        else:
            raise ValueError("How did you get here")
        return n


# Returns all of the numbers that are coprime to n from {0:n-1}
def allowed_diags(n):
    diag_list = []
    for i in range(n):
        if np.gcd(i,n) == 1:
            diag_list.append(i)
    return diag_list


# returns the stable product of a vector v
def stable_mod_prod(v, n):
    p = 1
    for i in range(v.size):
        p = p*v[i] % n
    return p


# solve an upper triangular system mod n
def mod_backward_sub(U,B,n):
    for iter in reversed(range(U.shape[0])):
        B[iter, :] = pow(int(U[iter, iter]), -1, n)*B[iter, :] % n
        B[:iter, :] = np.mod(B[:iter, :] - np.outer(U[:iter, iter], B[iter, :]),n)
    return B


# solve a lower triangular system mod n
def mod_forward_sub(L,B,n):
    for iter in range(L.shape[0]):
        B[iter, :] = pow(int(L[iter, iter]), -1, n) * B[iter, :] % n
        B[iter+1:, :] = np.mod(B[iter+1:, :] - np.outer(L[iter+1:, iter], B[iter, :]), n)
    return B
```

9.2.3 Known Plaintext Attack

```
import numpy as np
import itertools as it
import string
import random
import time

# Calculates the adjugate of a matrix
def adjugate(m):
    adj_m = np.zeros(m.shape)
    for i in range(m.shape[1]):
        for j in range(m.shape[1]):
            adj_m[i,j] = ((-1)**(i+j) * np.round(np.linalg.det(matrix_minor(m,i,j))))%26
    return adj_m.T

def modular_inverse(a, n):
    for x in range(1, n):
        if (((a % n) * (x % n)) % n == 1):
            return x
    return -1

# returns the minor of a matrix (whatever is left when you take out row i and column j)
def matrix_minor(m,i,j):
    minor = m
    minor = np.delete(minor,i,0)
    minor = np.delete(minor,j,1)
    return minor

def message_to_matrix(pt, ct, m, n):
    a = list(pt)
    for i in range(len(a)):
        a[i] = ord(a[i]) - 97
    b = np.array(a)

    newb = np.reshape(b, (m, n))
    a2 = list(ct)
    for i in range(len(a2)):
        a2[i] = ord(a2[i]) - 97
    b2 = np.array(a2)

    newb2 = np.reshape(b2, (m, n))


    return [newb, newb2]

def permutations(m,n):
    lst = []
    for i in range(m):
        lst.append(i)

    test = list(it.permutations(lst, n))
    print(len(test), "length of test")

    return test


def known_plaintext_attack(pt, ct, m, n):

    #turn message into matrix
    [newb, newb2] = message_to_matrix(pt, ct, m, n)

    c = np.zeros((n,n))
    d = np.zeros((n,n))
    print(newb)

    print(newb2)

    test = permutations(m,n)

    no_solution = True

    for i in range(len(test)):
        index = test[i]
        for j in range(n):
            c[j] = newb[index[j]]

        det = np.linalg.det(c)
        rdet = round(det)

        if (rdet % 2) != 0 and (rdet % 13) != 0:
            no_solution = False
            break

    if no_solution == False:

        #once we know the permutation that works, set d
        for j in range(n):
            d[j] = newb2[index[j]]

        adj = adjugate(c)

        print(d)

        mod = modular_inverse(rdet, 26)
        inverse = (mod*adj)%26

        print("inverse", inverse)

        key = ((inverse*d)%26)

        key = (np.matmul(inverse, d))%26
        check = (np.matmul(c,key))%26 == d

        return key

    else:
        string = 'There is no solution with this method'
        return string


def table():
    #table for combinatorial values
    countarr = np.zeros((26,26))
    count = 0

    for i in range(26):
        for j in range(26):
            for x in range(26):
                if (i*x)%26 == j:
                    countarr[i][j] = countarr[i][j] + 1
```

9.3 Fun With Images

The Hill Cipher can be used on images for some fun results.

I tested it on this 880×708px image:

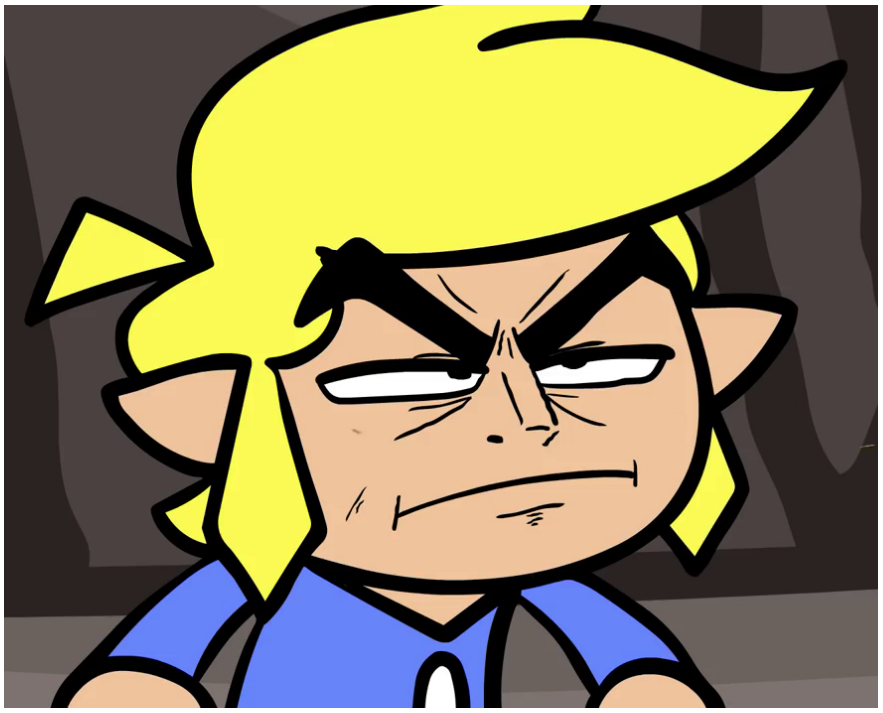


Figure 1: A cartoon picture of Link from the Legend of Zelda Series

First, with a key size of 20:

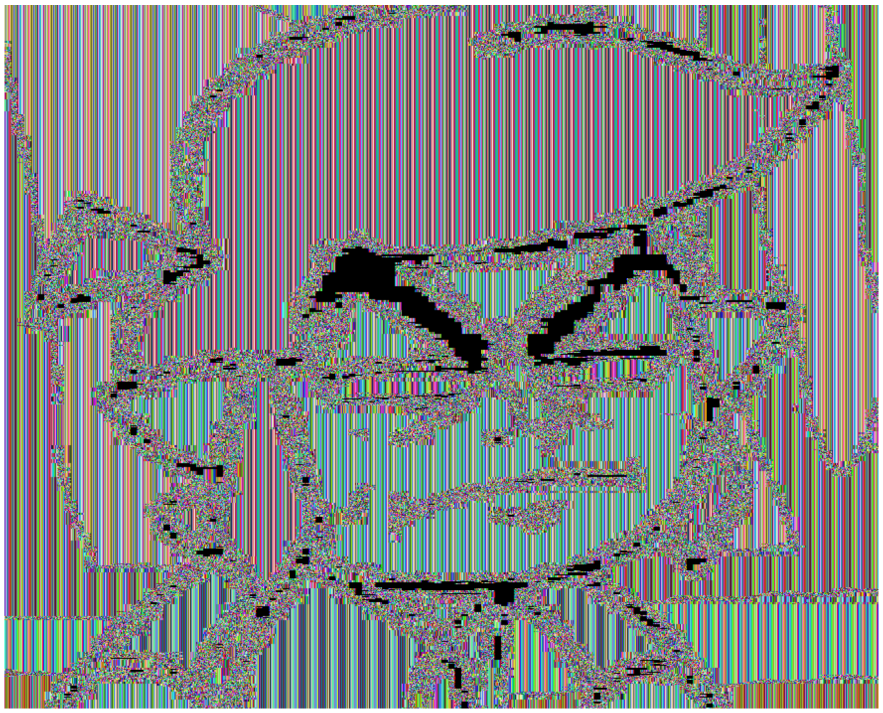


Figure 2: Here, you can see the artefacts caused by having a keysize that is too small. Since there are many places where the same color is repeated for 20 pixels in the row, the same output is produced for those 20 pixels, so scrambling only happens where there are edges, and the outline of the image is still visible. Also notice that places with the color black (corresponding to a value of (0,0,0) RGB), always maps to black, if there are twenty pixels in a row. This is because a linear transformation always maps the vector 0 to itself.

Then, with a key size of 880 (to cover the whole row):

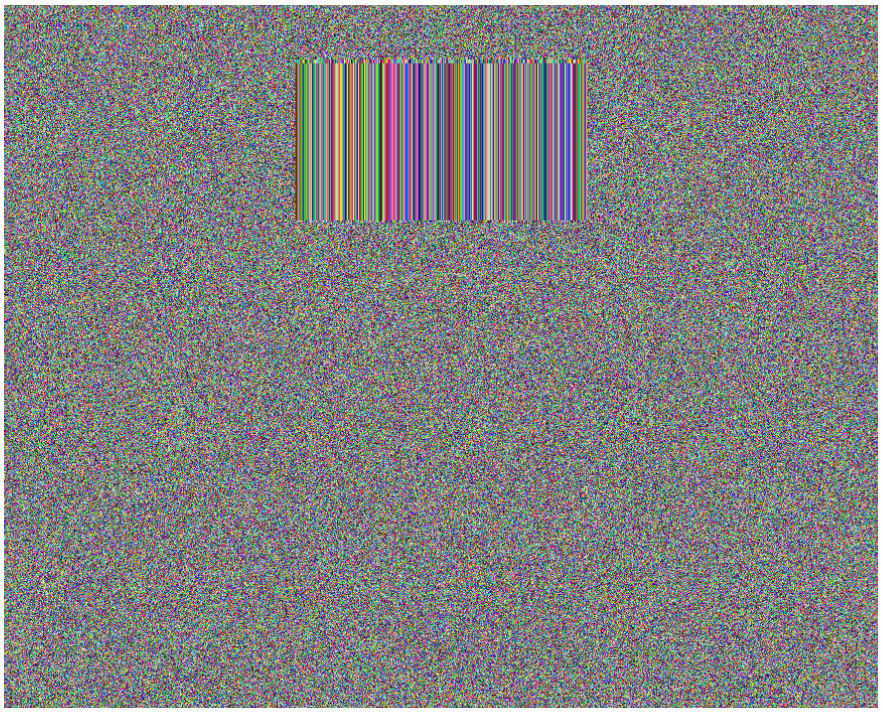


Figure 3: The artefact in the center is still a bit of a mystery to me, but at least the image is totally scrambled.

10 Link to All Code

Here is a link to the folder where we post our code: Google Drive Link

Here is a link to the Google Colab notebook used for animations: Google Colab Notebook