# Cryptography

Justin Wyss-Gallifent

July 21, 2021

13.1 Introduction . . . . . 2
13.2 Background . . . . . 2
13.3 Preliminary Notes . . . . . 2
13.4 Basic Encryption Technique . . . . . 3
13.4.1 How to Encrypt and Decrypt . . . . . 3
13.4.2 Practical Note . . . . . 4
13.5 Key Creation and Sharing . . . . . 4
13.6 Breaking the Key . . . . . 7
13.6.1 Circumstances . . . . . 7
13.6.2 Brute Force . . . . . 8
13.6.3 Refining Brute Force . . . . . 10
13.7 System of Equations Mod 2 . . . . . 15
13.8 Matlab . . . . . 18
13.9 Exercises . . . . . 22

## 13.1 Introduction

The goal of this chapter is to present a method of encryption which forms the basis of that used in many applications and show how linear algebra can be used to break this encryption. The basic method we present is not used as-is because it is fairly easily broken but it forms a building block for more sophisticated methods.

## 13.2 Background

Imagine we have a stream of bits, `0` and `1`, and we wish to encrypt them dynamically, meaning as each new bit comes along we have to encrypt it and send it along. It would seem reasonable to replace some `0` by `1` and vice versa in a way that the recepient would know how to undo the process.

Places where encryption like this may be useful would be things that are real-time critical like voice conversations, dynamical exchange of data, etc. Variations on the method we'll discuss are used in the Bluetooth protocol, various protocols used with GSM phones and various protocols used by the cable and other communication industries to scramble digital signals.

## 13.3 Preliminary Notes

A few things for this chapter:

(a) Modulo 2 arithmetic.

**Definition 13.3.0.1.** *Modulo 2 arithmetic* is arithmetic defined in such a way that all even numbers are considered equivalent to 0 and all odd numbers are considered equivalent to 1.

We write $a \equiv b \bmod 2$ if $a$ and $b$ are equivalent to one another and typically when we do operations we will write the result as either 0 or 1.

**Example 13.1.** For example we have the following:

$$\begin{aligned} 3 &\equiv 1 \bmod 2 \\ 1+1 &\equiv 0 \bmod 2 \\ (3)(5) &\equiv 1 \bmod 2 \\ -1 &\equiv 1 \bmod 2 \end{aligned}$$

and so on.

This also extends to matrices. For example:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \bmod 2$$

(b) We'll use the notation $[a; b; c]$ often to denote the vector $[a\,b\,c]^T$ because it's a bit neater.

(c) We'll often have vectors whose entries are bits, either 0 or 1, so we might write $\bar{c} \in \{0, 1\}^5$ for example to indicate that $\bar{c}$ is a vector with five entries either 0 or 1.

## 13.4 Basic Encryption Technique

### 13.4.1 How to Encrypt and Decrypt

Suppose Alice wishes to send the following binary stream to Bob without Eve intercepting it and understanding it. This is the *plaintext* and could go on indefinitely:

```
1010001100111110101101011110101001000l...
```

In order to encrypt this so that Eve may not understand it if she intercepts it, what Alice and Bob can do is the following: First they create and share a *key* consisting of a binary string such as `11010`. For now let's not worry about how that key is created or shared.

First a quick observation:

**Fact 13.4.1.1.** If we start with 0 or 1 then doing addition mod 2 twice with the same value (0 or 1) cancels out. In other words we see:

$$\begin{aligned} 0 + 0 \equiv 0 \bmod 2 \text{ then } 0 + 0 \equiv 0 \bmod 2 \\ 0 + 1 \equiv 1 \bmod 2 \text{ then } 1 + 1 \equiv 0 \bmod 2 \\ 1 + 0 \equiv 1 \bmod 2 \text{ then } 1 + 0 \equiv 1 \bmod 2 \\ 1 + 1 \equiv 0 \bmod 2 \text{ then } 0 + 1 \equiv 1 \bmod 2 \end{aligned}$$

Alice takes her binary stream and does addition bit-by-bit with the key mod 2. When she runs out of key she just repeats the key from the beginning. In this way we think of her key as infinitely long and having period 5. We'll use the term "key" to refer to both the repeated fragment and the infinitely long repetition. She then sends the bits one by one.

We can view this as follows:

| | | |
|---|---|---|
| Plaintext | `10100 01100 11111 01011 01011 11010 10010 001...` | |
| + Key | `11010 11010 11010 11010 11010 11010 11010 110...` | bit-by-bit |
| ≡ Ciphertext | `01110 10110 00101 10001 10001 00000 01000 111...` | mod 2 |

She sends this final string, the *ciphertext*, to Bob who decrypts it by doing +m2 again just like Alice did, but this undoes the encryption as noted above.

| | | |
|---|---|---|
| Ciphertext | `01110 10110 00101 10001 10001 00000 01000 111...` | |
| + Key | `11010 11010 11010 11010 11010 11010 11010 110...` | bit-by-bit |
| ≡ Plaintext | `10100 01100 11111 01011 01011 11010 10010 001...` | mod 2 |

If Eve intercepts the ciphertext in the middle of the process she has no way of knowing what either the key or the plaintext are.

Keep in mind that in reality both Alice and Bob have the key at their disposal and the message is dealt with bit-by-bit. On a per-bit basis they cycle through the key and add mod 2 as they go. When they get to the end of the key they simply start again at the beginning.

### 13.4.2 Practical Note

This method is nice primarily for two reasons. First, it is very fast. Second, doing mod 2 is the same as doing an XOR (exclusive or) and can easily be built into hardware circuits.

## 13.5 Key Creation and Sharing

The major issue with this is that the key needs to be created and shared between Alice and Bob before any communication can take place.

There are various ways that this can happen. When you rent a cable box the key might be hard-coded into a chip in the cable box, when you use a GSM phone the key might be hard-coded in the same way, and when you connect two Bluetooth devices by typing a code given by one machine into the other machine you are effectively sharing a key.

In both of these cases one problem is that a key with a larger period is more secure (keys with smaller periods can be brute-force guessed, for example there are only 32 possible keys of period 5 and yet a key with a larger period takes more memory to store (in the hardcoded case) and more time to manually transfer (in the Bluetooth case). In light of this we might ask if it is possible to create a key with a longer period using less data than the period? In other words, for example, could six bits of data be used to create a key with a period of more than six?

The answer is that we could create the key recursively in the following sense: We give some initial number of bits and then we define successive bits in terms of previous ones.

**Example 13.2.** We could assign a key $x_1x_2x_3...$ with $x_i \in \{0,1\}$ by assigning:

$$x_1 = 0,\ x_2 = 1, \text{ and } x_3 = 1$$

and for $n \geq 4$ we set:

$$x_n \equiv 1x_{n-3} + 0x_{n-2} + 1x_{n-1} \bmod 2$$

So then:

$$x_4 \equiv 1x_1 + 0x_2 + 1x_3 \equiv 1 \bmod 2$$
$$x_5 \equiv 1x_2 + 0x_3 + 1x_4 \equiv 0 \bmod 2$$

And so on. If we do this repeatedly we get:

`01110100111010...`

which we notice is repeating after 7 bits, meaing we've created the key `0111010` having period 7.

Thus by giving $\bar{s} = [0; 1; 1]$ (initial bits) and $\bar{c} = [1; 0; 1]$ (recursive coefficients) we generate a key with period 7 using only 6 bits.

**Definition 13.5.0.1.** A *linearly recursively defined key of length* $i$ can be defined by two vectors $\bar{s} = [x_1; ...; x_i] \in \{0,1\}^i$ and $\bar{c} = [1; ...; c_i] \in \{0,1\}^i$. The key then starts with the bits $x_1...x_i$ and for $n \geq i+1$ we have

$$x_n \equiv 1x_{n-i} + c_2x_{n-i+1} + ... + c_{i-1}x_{n-2} + c_ix_{n-1} \bmod 2$$

That is, the first vector gives the starting bits of the key and the second vector gives the coefficients of the linear combination which tells us how to build each subsequent bit of the key as a linear combination of the previous $i$ bits.

We insist that $c_1 = 1$ because otherwise any given $x_n$ depends only on the previous $i-1$ bits instead of the previous $i$ and so technically it would not have generating length $i$. It's certainly possible to allow $\bar{s}$ and $\bar{c}$ to have different lengths (necessarily $||\bar{c}|| \leq ||\bar{s}||$) but then in reality the length is just the length of $\bar{c}$ and the extra bits at the start of $\bar{s}$ aren't really part of the problem in the same sense.

We write the generating pair as:

$$K = \{\bar{s}, \bar{c}\}$$

**Theorem 13.5.0.1.** A linear recursively defined key of length $i$ is reversible, meaning if we know any $i$ bits we can recover any previous bits.

*Proof.* Suppose we have the $i$ bits:

$$x_k x_{k+1} ... x_{k+i+1}$$

We can calculate $x_{k-1}$ as follows. Since we know:

$$x_{k+i+1} \equiv x_{k-1} + c_2 x_k + c_3 x_{k+1} + ... + c_i x_{k+i} \bmod 2$$

We can simply solve for $x_{k-1}$. Note that $+c_j \equiv -c_j$ mod 2 and so the solution is:

$$x_{k-1} \equiv c_2 x_k + c_3 x_{k+1} + ... + c_i x_{k+i} + x_{k+i+1} \bmod 2$$

$\square$

**Definition 13.5.0.2.** The *period* of the key is the shortest number of bits after which the key repeats.

**Theorem 13.5.0.2.** A linearly recursively defined key of length $i \geq 1$ has period less than or equal to $2^i - 1$.

*Proof.* Consider a string of bits:

$$x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} ...$$

Consider the list of strings of bits (each row is a string of bits):

$$\begin{array}{llll} x_1 & x_2 & ... & x_i \\ x_2 & x_3 & ... & x_{i+1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{2^i} & x_{2^i+1} & ... & x_{2^i+i-1} \end{array}$$

If any of these strings were all zeros then the key would be entirely zero because the fact that it's length $i$ means that each successive bit and previous bit is a linear combination of zeros. If the key were entirely zero it would have period $1 \leq 2^i - 1$ and we are done.

So suppose none of these strings are all zeros. In this list there are $2^i$ strings in this list and each string has $i$ bits. However there are only $2^i - 1$ possible strings of nonzero strings of $i$ bits ($i$ bits and 2 choices per bit, then throw out 00...0). Consequently two of them must be identical.

Thus there exists $1 \leq j < k \leq 2^i$ with:

$$x_j x_{j+1} ... x_{j+i-1} = x_k x_{k+1} ... x_{k+i+1}$$

But then since the key is recursively defined with length $i$ we know that this equality continues beyond the bits shown. In other words:

$$x_j x_{j+1} ... = x_k x_{k+1} ...$$

Thus the key repeats after $k - j$ bits. Since $k \leq 2^i$ and $j \geq 1$ we have:

$$k - j \leq 2^i - 1$$

□

It is not infrequent to get a value of exactly $2^i - 1$ as shown:

**Example 13.2 Revisited.** The pair $K = \{[0; 1; 1], [1; 0; 1]\}$ has length 3 and generates a key with period $2^3 - 1 = 7$.

Okay, our example is not particularly impressive. Here are two more:

**Example 13.3.** The pair $K = \{[0; 1; 0; 0; 0], [1; 0; 1; 0; 0]\}$ has length 5 and generates the key with period $2^5 - 1 = 31$ shown here to 62 bits with a space to see the point at which it repeats.

```
0100001001011001111100011011101 0100001001011001111100011011101...
```

**Example 13.4.** For any $\bar{s} \in \{0, 1\}^{31}$ if $\bar{c} \in \{0, 1\}^{31}$ with $\bar{c} = [1; 0; 0; 1; 0; ...; 0]$ then the pair $K = \{\bar{s}, \bar{c}\}$ generates a key with period $2^{31} - 1 = 2147483647$, not shown.

## 13.6 Breaking the Key

### 13.6.1 Circumstances

The approach we take is to assume that we have obtained some fragment of the key which we'll label as $x_1 x_2 x_3 ....$ We're not assuming this is the start of the key, despite the labeling.

This could happen if we obtain both some ciphertext and some matching plaintext (maybe by snooping) since we can add the two bit-by-bit $\bmod 2$ in order to get the correspoding part of the key.

The goal is to figure out a recursion relation which generates this key fragment, then use it to generate enough key to decrypt the entire ciphertext.

### 13.6.2 Brute Force

One option could be a brute-force approach. Since the recursion relation has each bit being a linear combination of some number of previous bits we could take the portion of the key and take a trial-and-error approach.

What this means is we first check if a linearly recursively defined key of length 2 works. If not, we check if a linearly recursively defined key of length 3 works, then if a linearly recursively defined key of length 4 works, and so on until we either find the recursion relation or run out of key.

This is best illustrated with an example:

**Example 13.5.** Suppose we obtain the following portion of the key:

```
0110101111000100...
```

We could proceed by asking progressively as follows:

Question: Could a linearly recursively defined key of length 2 work?

If so then we would have $x_1 = 0$, $x_2 = 1$, and $x_n = c_1 x_{n-2} + c_2 x_{n-1}$ for $n \geq 3$. Applying this to $x_3$ and $x_4$ we get:

$$\begin{aligned} x_3 &\equiv c_1 x_1 + c_2 x_2 \bmod 2 \\ x_4 &\equiv c_1 x_2 + c_2 x_3 \bmod 2 \end{aligned}$$

which fills in to:

$$\begin{aligned} 1 &\equiv c_1(0) + c_2(1) \bmod 2 \\ 0 &\equiv c_1(1) + c_2(1) \bmod 2 \end{aligned}$$

or as a matrix equation:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix} \bmod 2$$

This matrix equation has the solution $c_1 = 1$, $c_2 = 1$ so we might guess that we have

$$x_n \equiv x_{n-2} + x_{n-1} \bmod 2 \text{ for } n \geq 3$$

If we test this on the key fragment (for $n \geq 5$) we find that $x_5 \equiv x_3 + x_4 \bmod 2$ but $x_6 \not\equiv x_4 + x_5 \bmod 2$ so clearly that didn't work.

Question: Could a linearly recursively defined key of length 3 work?

If we proceed as above we get the matrix equation:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \bmod 2$$

This matrix equation has no solution. This can be seen easily because the columns of the matrix each add to 0 mod 2 but the target column does not and so it is not a linear combination of the columns of the matrix.

Note: We may also see that keys of length 2 or 3 cannot work since the key has `000` in it, which would produce only `0`s afterwards.

Question: Could a linearly recursively defined key of length 4 work?

If we proceed as above we get the matrix equation:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \equiv \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} \bmod 2$$

This matrix equation has the solution $c_1 = 1$, $c_2 = 1$, $c_3 = 0$, $c_4 = 0$ so we might guess that we have

$$x_n \equiv x_{n-4} + x_{n-3} \bmod 2 \text{ for } n \geq 5$$

If we test this on the remaning bits of the key fragment we find it works, so we believe, with the information we have, that this is it.

Before proceeding there are a few things that we should note:

- The matrix equation we're solving at each step is not as confusing as it may look. When testing whether a linearly recursively defined key of length $m$ could work we simply look at:

$$\begin{bmatrix} x_1 & x_2 & \dots & x_m \\ x_2 & x_3 & \dots & x_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_{m+1} & \dots & x_{2m-1} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} \equiv \begin{bmatrix} x_{m+1} \\ x_{m+2} \\ \vdots \\ x_{2m} \end{bmatrix} \bmod 2$$

Notice the first column is the $m$ bits of the key starting at $x_1$, the second column is the $m$ bits of the key starting at $x_2$, and so on, finishing with the column vector on the right being the $m$ bits of the key starting at $x_{m+1}$.

- This is computationally intensive.

### 13.6.3 Refining Brute Force

Luckily there is a theorem which comes to our rescue. Note that everything is mod 2, meaning a determinant of a matrix is either 0 or 1.

**Theorem 13.6.3.1.** For any given $m$ define

$$M_m = \begin{bmatrix} x_1 & x_2 & \dots & x_m \\ x_2 & x_3 & \dots & x_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_{m+1} & \dots & x_{2m-1} \end{bmatrix}$$

Then:

(i) If $\det(M_m) \equiv 1 \bmod 2$ then no linear recursion of length less than $m$ will satisfy the sequence $x_1, x_2, ..., x_{2m-1}$.

(ii) If $\det(M_m) \equiv 0 \bmod 2$ and if there is a linear recursion of length $m$ which does satisfy the sequence $x_1, x_2, ..., x_{2m-1}$. then there is a linear recursion of length less than $m$ which will also satisfy that sequence.

*Proof.* Proof of (i) by contrapositive:

Suppose for a given $m$ some linear recursion of length $i < m$ will work. Consider the matrix:

$$M_m = \begin{bmatrix} x_1 & x_2 & \dots & x_m \\ x_2 & x_3 & \dots & x_{1+m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i+1} & x_{i+2} & \dots & x_{i+m} \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_{1+m} & \dots & x_{2m-1} \end{bmatrix}$$

(Note that it's possible that $i + 1 = m$ in which case those are the same row.)

The fact that a linear recursion of length $i < m$ works means that there are coefficients $c_1, ..., c_i$ such that the following are all true:

$$\begin{aligned}
x_{i+1} &\equiv c_1x_1 + c_2x_2 + ... + c_ix_i \bmod 2\\
x_{i+2} &\equiv c_1x_2 + c_2x_3 + ... + c_ix_{i+1} \bmod 2\\
\vdots &\equiv \quad \vdots\\
x_{i+m} &\equiv c_1x_m + c_2x_{m+1} + ... + c_ix_{i+m} \bmod 2
\end{aligned}$$

As far as $M_m$ this system is simply saying that:

$$\text{Row } i+1 \equiv c_1(\text{Row } 1) + c_2(\text{Row } 2) + ... + c_i(\text{Row } i) \bmod 2$$

So that the $(i+1)^{\text{th}}$ row of the matrix is a linear combination of the previous rows. Hence the rows are linearly dependent and $\det(M_m) = 0$.

The proof of (ii) is more technical and is omitted.

□

The ramifications of this proof are important.

**Theorem 13.6.3.2.** The length of the shortest linear recursion relation will be the largest $m$ for which $\det(M_m) \equiv 1 \bmod 2$.

*Proof.* There is definitely a shortest linear recursion relation that works because there is definitely a linear recursion relation that works. Suppose its length is $m$, then $\det(M_m) \equiv 1 \bmod 2$ because otherwise $\det(M_m) \equiv 0 \bmod 2$ and a shorter one would work by (ii). Moreover for $k > m$ we must have $\det(M_k) \equiv 0 \bmod 2$ otherwise $m$ would not work by (i). □

Consequently, assuming we can obtain such an $m$ the solution will be given by solving the matrix equation corresponding to, all $\bmod 2$:

$$\begin{aligned}
c_1x_1 + c_2x_2 + ... + c_mx_m &= x_{m+1}\\
c_1x_2 + c_2x_3 + ... + c_mx_{m+1} &= x_{m+2}\\
... &= ...\\
c_1x_m + c_2x_{m+1} + ... + c_mx_{2m-1} &= x_{2m}
\end{aligned}$$

which is

$$\begin{bmatrix} x_1 & x_2 & \dots & x_m \\ x_2 & x_3 & \dots & x_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_{m+1} & \dots & x_{2m-1} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} \equiv \begin{bmatrix} x_{m+1} \\ x_{m+2} \\ \vdots \\ x_{2m} \end{bmatrix} \bmod 2$$

Of course there is still an issue - we can test determinants all day but we'll never know if we have the largest value $m$ for which $\det(M_m) \equiv 1$ since how could we?

The approach we take is therefore as follows:

Refined Brute Force Method

(a) Calculate $\det(M_m) \bmod 2$ for $m = 1, 2, 3, ...$ until we encounter a 1 followed by some reasonable number of 0s. The term "reasonable" is ambiguous and might depend upon the technology being used.

(b) Solve the matrix equation:

$$\begin{bmatrix} x_1 & x_2 & \dots & x_m \\ x_2 & x_3 & \dots & x_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_{m+1} & \dots & x_{2m-1} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} \equiv \begin{bmatrix} x_{m+1} \\ x_{m+2} \\ \vdots \\ x_{2m} \end{bmatrix} \bmod 2$$

(c) Check whether the recursion relation given by the solution works for the entire key fragment. If so, stop and conclude we have found the recursion relation. If not, proceed with higher values of $m$.

(d) If we run out of key fragment before encountering a solution then we simply do not have enough key fragment to find the recursion relation.

**Example 13.6.** Suppose we obtain the following key fragment consisting of fifty bits:

```
10011011001001010001110100010001111100111110111101
```

We calculate determinants of $M_m$ for $m = 1, 2, 3, ...$ until we see a few 0s in a row. All are $\bmod 2$:

$$
\begin{aligned}
\det(M_1) &\equiv 1 \\
\det(M_2) &\equiv 0 \\
\det(M_3) &\equiv 1 \\
\det(M_4) &\equiv 1 \\
\det(M_5) &\equiv 0 \\
\det(M_6) &\equiv 1 \\
\det(M_7) &\equiv 0 \\
\det(M_8) &\equiv 0 \\
\det(M_9) &\equiv 1 \\
\det(M_{10}) &\equiv 0 \\
\det(M_{11}) &\equiv 0 \\
\det(M_{12}) &\equiv 0 \\
\det(M_{13}) &\equiv 0
\end{aligned}
$$

We notice that we've run into a string of 0s after $m = 9$ so we suggest that $m = 9$ might give us the solution since it gave the seemingly final determinant of 1.

We therefore examine the matrix equation for $m = 9$:

$$
\begin{bmatrix}
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \end{bmatrix}
\equiv
\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \mod 2
$$

This has the solution $c_1 = 1$, $c_2 = 0$, $c_3 = 1$, $c_4 = 1$, $c_5 = 0$, $c_6 = 1$, $c_7 = 0$, $c_8 = 0$, $c_9 = 0$ which suggests that $x_n \equiv x_{n-9} + x_{n-7} + x_{n-6} + x_{n-4} \mod 2$ for $n \geq 9$.

If we test this we find that it works on all of our key fragment so we accept it as a solution.

If this had failed to work we would have needed to calculate further determinants knowing another 1 followed by 0s on the horizon.

**Example 13.7.** Suppose we obtain the following key fragment consisting of fifty bits:

```
01110111110111110101111110001000011110110001111011
```

We calculate determinants of $M_m$ for $m = 1, 2, 3, ...$ until we see a few 0s in a row. All are $\bmod 2$:

$$\begin{aligned}
\det(M_1) &\equiv 0\\
\det(M_2) &\equiv 1\\
\det(M_3) &\equiv 1\\
\det(M_4) &\equiv 1\\
\det(M_5) &\equiv 1\\
\det(M_6) &\equiv 0\\
\det(M_7) &\equiv 1\\
\det(M_8) &\equiv 0\\
\det(M_9) &\equiv 0\\
\det(M_{10}) &\equiv 0
\end{aligned}$$

We notice that we've run into a string of 0s after $m = 7$ so we suggest that $m = 7$ might give us the solution since it gave the seemingly final determinant of 1.

We therefore examine the matrix equation for $m = 7$:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1\\ 1 & 1 & 1 & 0 & 1 & 1 & 1\\ 1 & 1 & 0 & 1 & 1 & 1 & 1\\ 1 & 0 & 1 & 1 & 1 & 1 & 1\\ 0 & 1 & 1 & 1 & 1 & 1 & 0\\ 1 & 1 & 1 & 1 & 1 & 0 & 1\\ 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_1\\ c_2\\ c_3\\ c_4\\ c_5\\ c_6\\ c_7 \end{bmatrix} \equiv \begin{bmatrix} 1\\ 1\\ 1\\ 0\\ 1\\ 1\\ 1 \end{bmatrix} \bmod 2$$

This has the solution $c_1 = 0$, $c_2 = 1$, $c_3 = 0$, $c_4 = 0$, $c_5 = 0$, $c_6 = 0$, $c_7 = 0$ which suggests that

$$x_n \equiv x_{n-6} \bmod 2 \text{ for } n \geq 8$$

However this has a problem in that $c_1 \neq 1$. Moreover even if we overlook that and test it for our key fragment we find it fails because $x_{19} \not\equiv x_{13} \bmod 2$.

So then we calculate further determinants of $M_m$. All are $\bmod 2$:

$$\begin{aligned}\det(M_{11}) &\equiv 0\\ \det(M_{12}) &\equiv 1\\ \det(M_{13}) &\equiv 0\\ \det(M_{14}) &\equiv 0\\ \det(M_{15}) &\equiv 0\end{aligned}$$

Aha, a new 1 showed up at $m = 12$, followed by some 0s.

We therefore examine the matrix equation for $m = 12$:

$$\begin{bmatrix} 0&1&1&1&0&1&1&1&1&1&0&1\\ 1&1&1&0&1&1&1&1&1&0&1&1\\ 1&1&0&1&1&1&1&1&0&1&1&1\\ 1&0&1&1&1&1&1&0&1&1&1&1\\ 0&1&1&1&1&1&0&1&1&1&1&1\\ 1&1&1&1&1&0&1&1&1&1&1&0\\ 1&1&1&1&0&1&1&1&1&1&0&1\\ 1&1&1&0&1&1&1&1&1&0&1&0\\ 1&1&0&1&1&1&1&1&0&1&0&1\\ 1&0&1&1&1&1&1&0&1&0&1&1\\ 0&1&1&1&1&1&0&1&0&1&1&1\\ 1&1&1&1&1&0&1&0&1&1&1&1 \end{bmatrix} \begin{bmatrix} c_1\\ c_2\\ c_3\\ c_4\\ c_5\\ c_6\\ c_7\\ c_8\\ c_9\\ c_{10}\\ c_{11}\\ c_{12} \end{bmatrix} \equiv \begin{bmatrix} 1\\1\\1\\1\\0\\1\\0\\1\\1\\1\\1\\1 \end{bmatrix} \bmod 2$$

This has the solution $c_1 = 1$, $c_2 = 1$, $c_3 = 0$, $c_4 = 1$, $c_5 = 0$, $c_6 = 1$, $c_7 = 1$, $c_8 = 0$, $c_9 = 1$, $c_{10} = 0$, $c_{11} = 0$, $c_{12} = 0$ which suggests that

$$x_n \equiv x_{n-12} + x_{n-11} + x_{n-9} + x_7 + x_6 + x_4 \bmod 2 \text{ for } n \geq 13$$

If we test this we find that it works on all of our key fragment so we accept it as a solution.

## 13.7 System of Equations Mod 2

In this chapter we needed to solve certain matrix equations mod 2. To do this we should be aware of a few points.

**Theorem 13.7.0.1.** A square matrix $A$ with entries in $\{0,1\}$ is invertible mod 2 if and only if $\det(A) \equiv 1 \bmod 2$.

Note that "is invertible $\bmod 2$" means there is another matrix $B$, also having entries in $\{0,1\}$, such that $AB \equiv BA \equiv I \bmod 2$.

*Proof.* $\Leftarrow$:

If $A$ is invertible $\bmod 2$ then there is a matrix $B$ with entries in $\{0,1\}$ and with $AB \equiv I \bmod 2$. Taking the determinant of both sides yields:

$$\det(A)\det(B) \equiv \det(I) \bmod 2$$

which is the same as:

$$\det(A)\det(B) \equiv 1 \bmod 2$$

So that both $\det(A) \equiv \det(B) \equiv 1 \bmod 2$.

$\Rightarrow$:

Suppose $\det(A) \equiv 1$ mod 2. Then $\det(A) \neq 0$ and so $A$ has an inverse which can be calculated via the adjugate method:

$$\left[\frac{1}{\det(A)}\operatorname{adj}(A)\right] A = I$$

If we multiply both sides by $\det(A)$:

$$\operatorname{adj}(A)A = \det(A)$$

Then take both sides mod 2:

$$\operatorname{adj}(A)A \equiv I \bmod 2$$

This shows us that:

$$A^{-1} \equiv \operatorname{adj}(A) \bmod 2$$

So that in the mod 2 case if the matrix has determinant 1 mod 2 then the inverse equals the adjugate.

$\square$

Moreover from a calculational perspective since the adjugate is just the inverse multiplied by the determinant, in order to find the inverse of $A$ mod 2 we can simply take the inverse of $A$ in the regular sense and multiply the determinant of $A$, then by reducing mod 2.

**Example 13.8.** Consider the matrix:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

We have $\det(A) = -1$ and

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Therefore:

$$\det(A)A^{-1} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}$$

And taken mod 2 we get:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

So this is the inverse of $A$ mod 2.

This is useful when we are solving matrix equations mod 2.

**Example 13.9.** Consider the matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \bar{c} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Since the determinant of the matrix equals 1 mod 2 we can find the solution mod 2 by multipying both sides by the inverse mod 2:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \bar{c} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \bar{c} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\bar{c} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

## 13.8 Matlab

The following Matlab m-file will generate $n$ digits of the key with initial vector `s` and coefficient vector `c`:

```
function x = genkey(s,c,n)
  % Generates n bits of the recursively defined key
  % using initial string s and vector c.
  % The result is returned as a vector
  % so it's usually good to wrap it in
  % transpose or ' it to look at it.
  % Usage:
  % >> genkey([1;0;1;1],[1;1;0;0],10)'
  % ans =
  %   1     0     1     1     1     1     0     0     0     1
  x = s;
  l = length(s);
  for j = [l+1:n]
    x = [x;mod(transpose(x(j-l:j-1))*c,2)];
  end
end
```

Usage as per the help:

```
>> genkey([1;0;1;1],[1;1;0;0],10)'
ans =
  1     0     1     1     1     1     0     0     0     1
```

The following Matlab m-file will generate the matrix $M_m$ for a specific $m$ for a specific key fragment.

```
function M = genmatrixfromfragment(v,m)
  % Generates the matrix M_m from the key fragment vector v.
  % Note that the length of v must be >= 2m-1.
  % Usage:
  % >> genmatrixfromfragment([1;0;1;1;0;1;1;1;1;0;1;1;0],3)
  % ans =
  %      1     0     1
  %      0     1     1
  %      1     1     0
  M = [];for i=1:m;M=[M v(i:i+m-1)];end;
end
```

Usage as per the help:

```
>> genmatrixfromfragment([1;0;1;1;0;1;1;1;1;0;1;1;0],3)
ans =
     1     0     1
     0     1     1
     1     1     0
```

If you don't give it enough key fragment it will error:

```
>> genmatrixfromfragment([1;0;1;1;0;1;1;1;1;0;1;1;0],10)
Index exceeds matrix dimensions.
Error in genmatrixfromfragment (line 10)
  M = [];for i=1:m;M=[M v(i:i+m-1)];end;
```

Notice that we can generate the matrix from the key vectors by combining the two commands:

```
>> v=genkey([1;0;1;1;1],[1;1;0;0;1],20);
>> genmatrixfromfragment(v,4)
ans =
     1     0     1     1
     0     1     1     1
     1     1     1     0
     1     1     0     1
```

Or in one fell swoop:

```
>> genmatrixfromfragment(genkey([1;0;1;1;1],[1;1;0;0;1],20),4)
ans =
     1     0     1     1
     0     1     1     1
     1     1     1     0
     1     1     0     1
```

Here is Example 13.6 worked out via Matlab. First we define the vector containing the key fragment. Here it's broken over several lines to fit on the page but of course it doesn't have to be entered this way.

```
 x=[
1;0;0;1;1;0;1;1;0;0;1;0;0;1;0;1;0;0;0;
1;1;1;0;1;0;0;0;1;0;0;0;1;1;1;1;1;0;0;
1;1;1;1;1;0;1;1;1;1;0;1];
```

Then we check determinants of $M_m$ until we hit a 1 followed by a bunch of 0s. We can do this with a `for` loop. The choice of going to $m = 15$ is just experimenting. Here we've also used `round` to round the determinant before taking the `mod`. The reason for this is that the precision of the determinant can be slightly off and so rounding it makes sure that we get the integer that we know we should get:

```
>> for m=1:15
mod(round(det(genmatrixfromfragment(x,m))),2)
end
ans =
     1
ans =
     0
ans =
     1
ans =
     1
ans =
     0
ans =
     1
ans =
     0
ans =
     0
ans =
     1
ans =
     0
ans =
     0
ans =
     0
ans =
     0
ans =
     0
ans =
     0
```

So now we see that $m = 9$ might be our goal. Thus we solve $M_m\bar{c} = [x_{m+1}; ...; x_{2m}]$ using the inverse of $M_m$ mod 2, again using `round` in there:

```
>> M = genmatrixfromfragment(x,9);
>> c = mod(round(det(M)*inv(M)*x(10:18)),2)
c =
     1
     0
     1
     1
     0
     1
     0
     0
     0
```

Finally we need to check that the key that this generates matches the key fragment at the start.

If we take the first nine bits from the key fragment and these nine bits we found and we use them to generate a key with the same length as the original fragment we can compare this generated key with the key fragment to see if this actually works. To be really fancy we can just take the difference between the vectors:

```
>> norm(genkey(x(1:9),c,length(x)) - x)
ans =
     0
```

## 13.9 Exercises

**Exercise 13.1.** Encrypt the stream `10110100010100101` using the key `10111`.

**Exercise 13.2.** Encrypt the stream `1101101001000001011001010` using the key `110101`.

**Exercise 13.3.** Write down the first 30 digits of the key defined by

$$\{[1;1;0;0],[1;0;0;1]\}$$

Can you see what the key length is?

**Exercise 13.4.** Write down the first 30 digits of the key defined by

$$\{[1;0;1;1],[1;0;0;1]\}$$

Can you see what the key length is?

**Exercise 13.5.** Use brute force to find the recursion relation for the key fragment `1011100101` and use it to find the period.

**Exercise 13.6.** Use brute force to find the recursion relation for the key fragment `0011101001` and use it to find the period.

**Exercise 13.7.** Use brute force to find the recursion relation for the key fragment `010001101000110` and use it to find the period.

**Exercise 13.8.** Use brute force to find the recursion relation for the key fragment `010110010001111` and use it to find the period.

**Exercise 13.9.** Use refined brute force (look for a determinant of 1 followed by at least three determinants of 0) to find the recursion relation for the key fragment `01010010001011111011` and use it to find the next three bits of the key.

**Exercise 13.10.** Use refined brute force (look for a determinant of 1 followed by at least three determinants of 0) to find the recursion relation for the key fragment `01111000010111010101` and use it to find the next three bits of the key.

**Exercise 13.11.** Use refined brute force (look for a determinant of 1 followed by at least three determinants of 0) to find the recursion relation for the key fragment `101011110000010011101101101010` and use it to find the next three bits of the key.

**Exercise 13.12.** Use refined brute force (look for a determinant of 1 followed by at least three determinants of 0) to find the recursion relation for the key fragment `010010111000100111001111000001` and use it to find the next three bits of the key.

**Exercise 13.13.** Suppose you intercept the following ciphertext along with a fragment of the plaintext:

| | |
|---|---|
| Ciphertext: | `11101001011001101101011001011011111111000000110001` |
| Plaintext: | `0011101111011110010101` |

(a) Obtain as large a key fragment as you can.

(b) Use refined brute force to find the recursion relation for the key.

(c) Continue building the key until you see it repeat. At this point you know the full key.

(d) Use the key to decrypt the full ciphertext.

(e) If groups of five bits, converted to decimal, indicate letters with $1 = A$, $2 = B$, etc., what does the message say?

**Exercise 13.14.** Suppose you intercept the following ciphertext along with a fragment of the plaintext:

| | |
|---|---|
| Ciphertext: | `0010001111010010111100011110111100011100110011011011011` |
| Plaintext: | `1001101101000011001010100` |

(a) Obtain as large a key fragment as you can.

(b) Use refined brute force to find the recursive relation for the key.

(c) Continue building the key until you see it repeat. At this point you know the full key.

(d) Use the key to decrypt the full ciphertext.

(e) If groups of five bits, converted to decimal, indicate letters with $1 = A$, $2 = B$, etc., what does the message say?

**Exercise 13.15.** This method of producing a key can also be used to produce pseudorandom numbers. These are numbers that appear random but are not (generating random numbers and even defining what it means to be random is a difficult thing).

Create the first ten digits of a pseudorandom string of numbers between 0 and 15 as follows:

(a) Consider the key defined by the vectors $[1; 0; 1; 1; 1; 0; 0]$ and $[1; 0; 1; 1; 0; 0; 0]$. Write down 40 bits of this key.

(b) Break the key into 4-bit chunks and convert each chunk from binary to decimal.

**Exercise 13.16.** Suppose you encounter the pseudorandom number sequence:

`9,9,10,0,12,7,15,12,5`

Assuming these came from 4-bit chunks of a key generated using recursion, find the next three digits of the sequence.

**Exercise 13.17.** Suppose you encounter the pseudorandom number sequence:

`12,11,4,8,12,11,13,1,15,13`

Assuming these came from 4-bit chunks of a key generated using recursion, find the next three digits of the sequence.

**Exercise 13.18.** Suppose when attempting to break a recursively defined key (from a key fragment you have) you apply the Theorem and find for $m = 1$, 2, ... that:

$$\det(M_m) = 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0$$

at which point you can no longer calculate determinants because you run out of key fragment.

(a) What is your guess for the length of the recursive relation?

(b) Suppose you solve for the corresponding $c_i$ but they don't actually work when applied to the key fragment. What does this mean?
Hint: If this is confusing, think about what would happen if you'd only had enough key fragment to generate determinants $0, 1, 0, 0, 1, 1, 0$.

**Exercise 13.19.** Consider the following (repeating key):

`110010111001011110010`

(a) What is the key length?

(b) Find the recursion relation.
Note: The recursion relation is short and the systems of equations you need to solve are easy by hand.

**Exercise 13.20.** Consider the key recursively defined using the vectors $\bar{s} = [1; 0; 1; 1]$ and $\bar{c} = [1; 1; 0; 1]$. This recursive relation has length 4 but show that the key it generates can in fact be genereated using a recursive relation of length 2.