1 Introduction

Cryptography is now part of the fabric of our society. It is used in cell-phone communication and in internet transactions.

1.1 In fiction and movies

It has been featured in lots of popular fiction, for example,

- Edgar Allan Poe’s “The Gold Bug” (1843) [Po],
  From Wikipedia: Set on Sullivan’s Island, South Carolina, the plot follows William Legrand, who was recently bitten by a gold-colored bug. His servant, Jupiter, fears Legrand is going insane and goes to Legrand’s friend, an unnamed narrator, who agrees to visit his old friend. Legrand pulls the other two into an adventure after deciphering a secret message that will lead to a buried treasure.

- Sherlock Holmes “The Adventure of the Dancing Men” (1903) [D],
  From Wikipedia: Mr. Hilton Cubitt of Ridling Thorpe Manor in Norfolk visits Sherlock Holmes and gives him a piece of paper with this mysterious sequence of stick figures. The little dancing men are at the heart of a mystery which seems to be driving his young wife Elsie to distraction. He married her about a year ago, and until recently, everything was well. Holmes comes to realize that it is a substitution cipher. He cracks the code by frequency analysis. The last of the messages conveyed by the dancing men is a particularly alarming one. Holmes discovers the sender is Elsie’s former fiancé from Chicago and has come to England to woo her back (more precisely to scare Elsie out of Cubitt’s arms into his own.

- Neal Stephenson’s “Cryptonomicon” (1999).
  From Wikipedia: This award-winning novel follows, in part, Lawrence Pritchard Waterhouse, a young United States Navy code breaker and mathematical genius in 1942, as well as his grandson in 1997.
Stephenson also includes a precise description of (and even Perl script for) the Solitaire (or Pontifex) cipher, a cryptographic algorithm developed by Bruce Schneier for use with a deck of playing cards, as part of the plot.

For more cryptography in fiction, see Prof John Dooley’s list at [http://faculty.knox.edu/jdooley/Crypto/CryptoFiction.htm](http://faculty.knox.edu/jdooley/Crypto/CryptoFiction.htm). There are many movies involving some cryptography, such as

- **Breaking the Code (1996), directed by Herbert Wise,**
  
  Breaking the Code is a 1986 play by Hugh Whitemore about British mathematician Alan Turing, who was a key player in the breaking of the German Enigma code at Bletchley Park during World War II and founder of computer science.

- **Enigma (2001), directed by Michael Apted,**
  
  Using a screenplay by Tom Stoppard, adapted from the novel Enigma by Robert Harris, this is a film about a young genius frantically races against time to crack an enemy WWII code and solve the mystery surrounding the woman he loves.

- **Mercury Rising (1998), directed by Harold Becker,**
  
  IMDB logine: Shadowy elements in the NSA target a nine-year old autistic savant for death when he is able to decipher a top secret code.

- **Sneakers (1992), directed by Phil Alden Robinson,**
  
  IMDB logine: Shadowy elements in the NSA target a nine-year old autistic savant for death when he is able to decipher a top secret code.

- **U-571 (2000), directed by Jonathan Mostow,**
  
  IMDB logine: A German submarine is boarded by disguised American submariners trying to capture their Enigma cipher machine.

- **The Thomas Beale Cipher (2010, short film), directed by Andrew Allen ([https://vimeo.com/19115071](https://vimeo.com/19115071)),**
  
  IMDB logine: Professor White, cryptographer extraordinaire, is on the trail of the notoriously uncrackable Thomas Beale cipher, a century-old riddle hiding the location of a fortune in gold. But White is not alone-shadowy forces are tight on his tail.
• National Treasure (2004), directed by Jon Turteltaub,
   IMDB logine: A German submarine is boarded by disguised American submariners trying to capture their Enigma cipher machine.

• Pi (1998), directed by Darren Aronovsky,
   IMDB logine: A paranoid mathematician searches for a key number that will unlock the universal patterns found in nature.

• Windtalkers (2002), directed by John Woo,
   IMDB logine: Two U.S. Marines in WWII are assigned to protect Navajo Marines who use their native language as an unbreakable radio cipher.

• Zodiac (2007), directed by David Fincher.
   IMDB logine: A San Francisco cartoonist becomes an amateur detective obsessed with tracking down the Zodiac killer, who communicates via enciphered messages.

In 2010, there was even an AMC TV series Rubicon (http://en.wikipedia.org/wiki/Rubicon_(TV_series) featuring a running theme of messages hidden in newspaper crossword puzzles.

Cryptography is all around us, any time secrecy is a goal.

1.2 Other types of codes

Error-correcting codes are all around us as well. A sender uses an error-correcting code on a message when they want that message to arrive in a readable form. The communication channel might have noise, but by encoding the message with an error-correcting code, enough redundancy is added so that the receiver can recover the original message from the possible corrupted data they received. Reliable communication, at a low cost, is the goal here. Cell-phones, music CDs, video DVDs, all use error-correcting codes.

There is another type of code which we won’t study here much. Codes were developed by communication companies to send information in a more compressed format (e.g., telegraph companies made telegraph codes, data processing companies made ASCII codes, etc). Here is some Python code for converting back-and-forth between ascii and letter strings:
def num2bin(x):
    ""
    Converts integer in range (1,255) to binary.
    EXAMPLES:
    sage: num2bin(129)
        [1, 0, 0, 0, 0, 0, 0, 1]
    ""
    return [floor(x/2**(7-i))%2 for i in range(8)]

def string2ascii(m):
    ""
    Converts a string of characters to a sequence of
    0’s and 1’s using the Python ord command.
    ""
    L = []
    for a in m:
        L.append(ord(a))
    M = [num2bin(x) for x in L]
    return flatten(M)

def ascii2string(M):
    ""
    M is a ciphertext message of 0’s and 1’s of length 8k.
    This returns a string of characters representing that
    list in ascii.
    ""
    m = len(M)
    k = int(m/8)
    S = []
    for i in range(k):
        s = sum([2**(7-j)*M[8*i+j] for j in range(8)])
        S.append(chr(s))
    sumS = """" 
    for s in S:
        sumS = sumS + s
    return sumS

For example, string2ascii("US") returns [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1], and ascii2string([0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1]) returns 'US'.

1.3 Free online resources

This course will be an introduction to some of its basic aspects, using Sage [http://www.sagemath.org/] or online at https://cloud.sagemath.com/ and SymPy [http://www.sympy.org/], also available at try the installer at http://www.lfd.uci.edu/~gohlke/pythonlibs/) to illustrate some of the computations. For many of the Sage commands, you will need to load the file classical-ciphers-examples.sage (available from the class webpage into Sage. This is done by saving file to your computer, starting Sage, and typing

%attach("/mypath/classical-ciphers-examples.sage")

at the command line or into a cell of the notebook. (Here mypath is the full path to whereever you saved the file to.) For the SymPy commands, the commands are included in version 0.7.4 or later of SymPy, available from the sympy website. For windows installation, you may wish to try

http://nipy.bic.berkeley.edu/sympy-dist/

instead of the sympy website.

Some good ideas for course projects can be found by looking through (a) chapters of the textbook we have not covered in class, or (b) the topics in past issues of Cryptologia:

http://www.tandfonline.com/loi/ucry20

If you have trouble accessing these issues from a USNA computer, please let me know.

2 Security terminology and concepts

Basic setup: A cryptosystem is a collection of

- a finite set \( \mathcal{A} \) of symbols which acts as an alphabet,
- the set \( \mathcal{P} \) of words in \( \mathcal{A} \) which acts as the set of plaintext messages,
- the set \( \mathcal{C} \) of words in \( \mathcal{A} \) which acts as the set of ciphertext messages,
- a finite set \( \mathcal{K} \) of symbols which acts as a key space,
• for each $k \in \mathcal{K}$ an enciphering map

$$E_k : \mathcal{P} \rightarrow \mathcal{C},$$

a corresponding deciphering key $k' \in \mathcal{K}$, and a deciphering map

$$D_{k'} : \mathcal{C} \rightarrow \mathcal{P},$$

such that

$$D_{k'} \circ E_k(p) = p,$$

for any $p \in \mathcal{P}$.

Alice sends message ----------> Bob receives message

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Eve (evil eavesdropper)</td>
</tr>
<tr>
<td>Mallory (malicious attacker, who can modify messages)</td>
</tr>
</tbody>
</table>

You should know all these and be able to provide examples (when appropriate).

1. Basic terms

(a) cryptography or cryptology - (from Greek “study of hidden or secret writing”) the practice and study of techniques for secure communication in the presence of third parties,

(b) cryptanalysis ((from the Greek “hidden, secret” and “to untie”) - the art and science of “attacking” or “breaking” cryptosystems

(c) cryptosystem - the combined systems of the encryption and decryption algorithms

(d) plaintext - the message to be sent (often a string of letters or numbers, possibly encoded with an error-correcting code)
(e) key - a parameter that determines the output of a cryptographic algorithm, as part of the enciphering and deciphering process. In asymmetric cryptosystems, keys are divided into two parts - the private key and the public key.
In general, the (private) key must be kept secret from all but authorized users.

(f) key space - the set of all possible keys

(g) encrypt/encipher - disguising the plaintext in some way, using the key, resulting in ciphertext

(h) ciphertext - the result of encryption (depends on the key)

(i) decrypt/decipher - the process of “undoing” the encryption, (“usually”) resulting in the original plaintext

(j) attack or break - an method of cryptanalysis applied to a specific cryptosystem which circumvents the decryption algorithm and tries to recover the key or the plaintext.

(k) Brute force attack is a search through the entire keyspace to determine the key used. (Other attacks are discussed later.)

(l) A “successful” attack/break occurs when the method results in a decryption method which is significantly more efficient than a brute force attack.

(m) symmetric-key cryptosystems (also called “secret-key cryptosystems”) - a cryptosystem which uses the same key for encryption and for decryption.

(n) public-key cryptosystems (also called “asymmetric-key cryptosystems”) - a cryptosystem which uses one key for encryption and another for decryption; the encryption key is made public, but the decryption key is kept secret.

2. Data security terms.

(a) Secrecy - ensuring that the data is available only to those people who are authorized to have it.

(b) Integrity - ensuring data is not manipulated during the transmission.

\[^1\] Messages sent over a noisy channel can cause data loss, but this is less of a security issue than a reliability issue solved using error-correcting codes.
(c) Authentication - ensuring that the receiver (Bob) can verify that the message was indeed sent by the presumed sender (Alice).

(d) Non-repudiation - ensuring that the sender (Alice) can’t deny sending the message received by Bob.

3. Auguste Kerckhoff articulated six rules for cryptographic security in his 1883 article, “La cryptographie militaire” (available here: http://www.petitcolas.net/fabien/kerckhoffs/).

Kerckhoff’s Principles of Cryptographic Security:

(a) The system should be unbreakable in practice.
(b) The ciphertext must be able to fall into the hands of the enemy without inconvenience. This is “Kerckhoff’s principle.”
(c) Its key must be communicable and retainable without the help of written notes, and changeable or modifiable at the will of the correspondents.
(d) The cryptogram should be transmissible digitally.
(e) Encryption apparatus is portable and operable by a single person.
(f) The system should be easy to use.

Kerckhoff’s principle: The security of a cryptosystem must be measured only after assuming all aspects of the cryptosystem is known except for the key.

Shannon’s maxim: “The enemy knows the system.” (Due to information/communications theorist Claude Shannon, possibly independently.)

David’s Dictum: “Security through obscurity is absurdity.”

4. Attacks:

(a) brute force - a search of the entire key space
(b) frequency analysis - compare the frequency of the usage of symbols in the ciphertext with the frequency of usage in standard English text.
(c) ciphertext-only - the cryptanalyst has access only to a collection of ciphertexts (from which he/she tried to recreate the plaintext)

(d) known plaintext - the attacker has a set of ciphertexts to which he knows the corresponding plaintext.

(e) chosen plaintext - the attacker can obtain the ciphertexts corresponding to an arbitrary set of plaintexts of his own choosing

(f) adaptive chosen-plaintext - like a chosen-plaintext attack, except the attacker can choose subsequent plaintexts based on information learned from previous encryptions

(g) chosen ciphertext - the attacker can obtain the plaintexts corresponding to an “arbitrary” set of ciphertexts of his own choosing

(h) differential analysis - the study of how differences in plaintexts can affect the resultant difference in the ciphertext

(i) linear cryptanalysis - the study of finding affine approximations (e.g., the affine Hill cipher) to a cryptosystem

(j) man-in-the-middle - a form of active eavesdropping in which the attacker makes independent connections with Alice and Bob, and relays messages between them, making them believe that they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker.

5. Cryptographic problems:

(a) key exchange - Alice and Bob want to communicate using a secret-key cryptosystem. For that, they must exchange keys secretly.

(b) digital signatures - a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, and that it was not altered in transit.

(c) timestamps - authenticating the date or time at which a certain event occurred

(d) coin-flipping - find a reliable way to use a “coin flip” to settle a dispute between two parties if they cannot both see the coin

---

2An example is the “lunch-time attack” where the sender’s computer has been infiltrated during their lunchbreak, at which time plaintexts can be encrypted using the sender’s cryptosystem, although the key is unknown.
(e) key distribution - Consider a system with N members, all of whom wish to communicate in secret with each other. In a secret-key cryptosystem, each pair must have a different key (which is a lot of keys).

(f) key generation - In a given cryptographic system, the person or party responsible for generating the key(s) but do so in a secure manner.

6. Types of classical ciphers:

(a) transposition cipher - a method of encryption by which the positions held by units of plaintext (which are commonly characters or pairs of characters) are permuted, so that the ciphertext constitutes a permutation of the plaintext.


Example: A ”route cipher” is one where the characters of the plaintext are written on a pre-arranged route into a matrix agreed upon by the sender and receiver. The ciphertext is then obtained by transcribing the resulting text left-to-right, top-to-bottom.

(b) substitution cipher - a method of encryption by which units of plaintext are substituted with ciphertext.

i. “simple” substitution cipher - substitution operates on single letters of the plaintext
Example: shift cipher

ii. “polygraphic” substitution cipher - substitution operates on larger groups of letters is termed.
Example: Playfair cipher, Hill cipher

iii. “mono-alphabetic” substitution cipher - uses fixed (one-to-one) substitution over the entire plaintext
Example: shift cipher

iv. “polyalphabetic” substitution cipher - uses a different substitutions at different times in the plaintext
Example: Vigenère cipher
Figure 1: The Caesar, or shift, cipher

3 The shift and affine ciphers

The shift cipher is also called the Caesar cipher, named after Julius Caesar, who, according to Suetonius, used it over 2000 years ago with a shift of three to protect messages of military significance.

If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.

-Suetonius, Life of Julius Caesar

http://en.wikipedia.org/wiki/Shift_cipher

Caesar’s nephew Augustus reportedly used a similar cipher, but with a right shift of 1.

See also pp 81-84 in Kahn’s classic book [K] for more on the history of this cipher.

There are also several instances (incredibly enough) of modern uses of this cipher. One by a mobster in 2006 and another by a terrorist in 2011.

ALGORITHM (shift cipher):

INPUT:
  k - an integer from 0 to 25 (the secret "key")
  m - string of upper-case letters (the "plaintext" message)

OUTPUT:
  c - string of upper-case letters (the "ciphertext" message)
Identify the alphabet $A, \ldots, Z$ with the integers $0, \ldots, 25$.

Step 1: Compute from the string $m$ a list $L_1$ of corresponding integers.

Step 2: Compute from the list $L_1$ a new list $L_2$, given by adding $k \pmod{26}$ to each element in $L_1$.

Step 3: Compute from the list $L_2$ a string $c$ of corresponding letters.

There are many alphabets possible in Sage— the usual English letter, the binary alphabet, etc. See the reference manual of Sage’s [http://www.sagemath.org/doc/reference/sage/crypto/classical.html](http://www.sagemath.org/doc/reference/sage/crypto/classical.html) Classical Cryptosystem module, for details.

We shall use the Sage notebook interface in the next few Sage examples, so you will not see the sage: prompt on the input line.

```sage
AS = AlphabeticStrings()
A = AS.alphabet()
```
These commands produce the following output.

Sage


Using the SymPy version, one can use the default alphabet, or you can make up your own:

SymPy

>>> alphabet_of_cipher()

>>> L = [str(i) for i in range(10)]+['a','b','c']; L
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c']

>>> A = ''.join(L); A
'0123456789abc'

>>> alphabet_of_cipher(A)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c']

Whether you use Sage or SymPy, you will need to initialize an alphabet.

Sage

s = "Go Navy! Beat Army!"
m = AS.encoding(s)
print s
print m

These commands produce the following output.

Sage

Go Navy! Beat Army!
GONAVYBEATARMY
This is to set up our plaintext message. As a sequence of numbers, the plaintext is

\[6, 14, 13, 0, 21, 24, 1, 4, 0, 19, 0, 17, 12, 24.\]

Next we select the ShiftCryptosystem Python class in Sage and decide which shift to use.

```
CS = ShiftCryptosystem(AS)
m = CS.encoding(s)  # the plaintext message, again
c = CS.enciphering(1, m)  # the ciphertext
print m
print c
```

These commands produce the following output.
This string “HPOBWZCFBUBSNZ” is the enciphered ciphertext. You can do this in SymPy also:

```
>>> pt = "GONAVYBEATARMY"
>>> shift_cipher_encrypt(pt, 1)
HPOBWZCFBUBSNZ
>>> shift_cipher_encrypt(pt, 0)
GONAVYBEATARMY
>>> shift_cipher_encrypt(pt, -1)
FNMZUXADZSZQLX
```

We turn to the very similar affine cipher.

**ALGORITHM:**

**INPUT:**
- $a, b$ - a pair integers, where $\text{gcd}(a, 26) = 1$ (the secret "key")
- $m$ - string of upper-case letters (the "plaintext" message)

**OUTPUT:**
- $c$ - string of upper-case letters (the "ciphertext" message)

Identify the alphabet $A, \ldots, Z$ with the integers $0, \ldots, 25$. Step 1: Compute from the string $m$ a list $L_1$ of corresponding integers.

Step 2: Compute from the list $L_1$ a new list $L_2$, given by replacing $x$ by $a\cdot x+b \pmod{26}$, for each element $x$ in $L_1$.

Step 3: Compute from the list $L_2$ a string $c$ of corresponding letters.

We use $a = 3$ and $b = 4$ (for encryption) and $a = 9$ and $b = 16$ (for decryption).
3.1 Brute force decryption

Of course, deciphering this ciphertext message “HPOBWZCFBUBSNZ” can be accomplished by either shifting right by 25 (enciphering) or to the left by -1 (deciphering):

```python
print CS.enciphering(25, c)
print CS.deciphering(1, c)
```

These commands produce the following output.

```
GONAVYBEATARMY
GONAVYBEATARMY
```
We could also search over the entire keyspace of 26 letters and see which shift makes the most sense as the plaintext.

**Exercise 3.1.** Crack this ciphertext: “LRZZOACZZQTDZYPESLEXLVDPFDHTDPC.”

(Found in Prof Praeger’s course notes at [P].)

**Solution:** We brute force search over the entire key space. Note the 11th line below.

```
ct = "LRZZOACZZQTDZYPESLEXLVDPFDHTDPC"
c = CS.encoding(ct)
for i in range(26):
    print i, CS.deciphering(i, c)
```

These commands produce the following output.

```
0 LRZZOACZZQTDZYPESLEXLVDPFDHTDPC
1 KQYYNZBYYPSCYXODRKDWUOCECGSCOB
2 JPXXMYAXXORBXWNQCQJCVCVTNBDBFRBNA
3 IOWNLXZWWNQAWVMBPBISMACAEQAMZ
4 HNVVKWYVMPZVULAOHATHRL2BZDPZLY
5 GMUUVXUXULQYUTKZNZGSGQKYACOYKX
6 FLTIIWTTKNTSXJYMFRYRPFJPZ2XBNXJW
7 EKSSHTVSSJMVRSIXQEOIYWHYWAMWIV
8 DJRGRSUVRILVQRHWDNFHNVXVZLVHU
9 CIQFRQTHYUQGVJVCOCMGUWUYKUGT
10 BHPPEQSSPGJTPSOUIBUNBLFTVXJTF
11 AGOODPROOFSONETHATMAKESUSWISER
12 ZFNNCOQNEHRMNDSGZSLJDRTRVHRDQ
13 YEMMBNFMMDGQMCRFIRKYICQSQUGQCP
14 XDILAMOMLCPFLKBEQXQJXHBPRTPF
15 WCKKZLNKBEOKJAPDWIWWGAQSEOAN
16 VBJJYKMJJADNJIQOVHVFZNPNREDNMZ
17 UAIIXJLIIZCMIHYNBGUEYMOMQCMYL
18 TZHHWXIKHBYBLHGMATMFTDXLNLPLB
19 SYGGHVJGGXAKGFWZLSESCWKMOAKWJ
20 RXFUGIFFWZJFEPVYKDRBVJLJN
21 QWEEFTHEEVYIEDUXQJCAUIKIMYIH
22 PVDDSEGDDUXHDCITWPIBPZTHJHLXHTG
```
Using Sympy, the code is:

```python
>>> ct = "LRZZOACZZQTDZYPESLEXLVPDFDHTDPC"
>>> for i in range(1,26): print encipher_shift(ct, i)
MSAAPBDAARUEAZQFTMFYMQWEGEIUEQD
NTBBQCEBBSVFARGUNZXRFFHJVFRE
OUCCRDFCCTWGCHVAOYSGIGKWGSF
PVDDSEGDUXHDCITWIPZTHJHLXHTG
QWEEETFHEEVYIEDUJXQJCQAIKIMYIUH
RXFFUGIFFWZJFVEKRYKBRVJLJNZJVI
SYGGVHJGGXAKGFWLZSLESCWMMKOAJKWJ
TWHWIKHOBHLGHMTDXXLNLPLXK
UAIIIXLIIICMIFHYNBUINGUEYOMQCMYLU
VBIIJYKMMJADNJIZOCHVFZDNPRDNZM
WCKZLNKKBEOKJUDPWCIPWGAQQOSEQOAN
XDLAMOLLCFPLKBQEXQJXHBPRPTFPOBO
YEMMNBPMMDGQMLCFRKYICQSQQGQCP
ZFNNOQCNENNHRNMDGSZLJGDRTRRHDQ
AGOODPROOFISONETHATMAKESSUSWISER
BHPPEQSGJTPFOUIBNBLFTVTXJTFSDCIQQFRTQHKUPQGFVCQCMGUWYKUGT
DJRRGSURRILVQRHWEKDPDHVXLVZLVHU
EKSSHTVSSJWMSIIXLEXEOIYWMWIV
FLTIUWTTKNXSJYMFYRFPPJXZBXNJW
GMUUJXUXUUYUOTWZKUNZSGQKYAYOVXK
HNVSXKGYVZMLPZUVLAOHATRLZBDPZLY
IOWWLYZWNQAWVMPIBUISMACAEQAMZ
JFXXMYAXXRBXWNCQJCVJTNNDDBFRENAKQYYNZBBYJSCPXYODRDKWUOCECGSOCB
```
3.2 Frequency analysis

The most commonly used letters in English language text are e, t, a, o, i, n. How often a letter is used in a selection of text is called the letter frequency. How often a pair of letters occurs together is called the digram frequency. The most commonly occurring digrams in English are th, he, in, er. How often a n-tuple of letters occurs together is called the n-gram frequency.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>8.167%</td>
</tr>
<tr>
<td>b</td>
<td>1.492%</td>
</tr>
<tr>
<td>c</td>
<td>2.782%</td>
</tr>
<tr>
<td>d</td>
<td>4.253%</td>
</tr>
<tr>
<td>e</td>
<td>12.702%</td>
</tr>
<tr>
<td>f</td>
<td>2.228%</td>
</tr>
<tr>
<td>g</td>
<td>2.015%</td>
</tr>
<tr>
<td>h</td>
<td>6.094%</td>
</tr>
<tr>
<td>i</td>
<td>6.966%</td>
</tr>
<tr>
<td>j</td>
<td>0.153%</td>
</tr>
<tr>
<td>k</td>
<td>0.772%</td>
</tr>
<tr>
<td>l</td>
<td>4.025%</td>
</tr>
<tr>
<td>m</td>
<td>2.406%</td>
</tr>
<tr>
<td>n</td>
<td>6.749%</td>
</tr>
<tr>
<td>o</td>
<td>7.507%</td>
</tr>
<tr>
<td>p</td>
<td>1.929%</td>
</tr>
<tr>
<td>q</td>
<td>0.095%</td>
</tr>
<tr>
<td>r</td>
<td>5.987%</td>
</tr>
<tr>
<td>s</td>
<td>6.327%</td>
</tr>
<tr>
<td>t</td>
<td>9.056%</td>
</tr>
<tr>
<td>u</td>
<td>2.758%</td>
</tr>
<tr>
<td>v</td>
<td>0.978%</td>
</tr>
<tr>
<td>w</td>
<td>2.360%</td>
</tr>
<tr>
<td>x</td>
<td>0.150%</td>
</tr>
<tr>
<td>y</td>
<td>1.974%</td>
</tr>
<tr>
<td>z</td>
<td>0.074%</td>
</tr>
</tbody>
</table>

Other languages have different histograms. For example, Spanish and Arabic frequencies are given in Figure 5. (Figures 4 and 5 are courtesy of Wikipedia.) Note that “E” is still in first place!
In the next example, let us try to use frequency analysis to break a monoalphabetic substitution cipher. For this we need a fairly long cipher text, such as the following selection from Edgar Allan Poe’s “The Raven” [Po].

```python
raven = "Once upon a midnight dreary, while I pondered, weak and weary, Over many a quaint and curious volume of forgotten lore, While I nodded, nearly napping, suddenly there came a tapping, As of some one gently rapping, rapping at my chamber door. 'Tis some visitor,' I muttered, 'tapping at my chamber door -- Only this, and nothing more.'"
print raven
```
(The string raven must be entered into Sage all on one line; it appears on 5 lines above merely for typographical reasons.) These two commands produce the following output.

```sage
Once upon a midnight dreary, while I pondered, weak and weary, Over many a quaint and curious volume of forgotten lore, While I nodded, nearly napping, suddenly there came a tapping, As of some one gently rapping, rapping at my chamber door. 'Tis some visitor, ' I muttered, 'tapping at my chamber door -- Only this, and nothing more.'
```

Now we have entered the plaintext as a string, we ask Sage to output a frequency distribution, which we then sort:

```sage
m = CS.encoding(raven)
fds = m.frequency_distribution()
dict_fds = fds.function()list_fds = [(dict_fds[x], x) for x in dict_fds.keys()]list_fds.sort()list_fds.reverse()list_fds
```

These commands produce the following output.

```sage
[[0.108527131782946, E], [0.0930232558139535, N], [0.0891472868217055, A], [0.0852713178294574, O], [0.0736434108527132, I], [0.0620155038759690, T], [0.0581395348837209, D], [0.0465116279069767, P], [0.0465116279069767, M], [0.0348837209302326, Y], [0.0348837209302326, G], [0.0310077519379845, S], [0.0310077519379845, L], [0.0310077519379845, H], [0.0271317829457364, U], [0.0193798449612403, C], [0.0155038759689922, W], [0.0116279069767442, V], [0.0116279069767442, F], [0.0075193798449612, B], [0.0038759899224806, Q], [0.00387596899224806, K]]
```

We ask SymPy to output a frequency distribution, which we then sort:

```python
>>> AoC = alphabet_of_cipher()
>>> n = len(raven) - raven.count(" ")
>>> n
277
```
In either case, we see that “E” is the most frequent and “K” and “Q” are the least frequent (of those letters actually used). The slight differences in the statistics are due to the fact that one of them is counting the non-alphanumeric characters and one is not.

How does this compare with “typical” (whatever that means) English language use? See Figure 4.

**Exercise 3.2.** Decipher this: “VUJLBWVUHTPKUPNOAKYLHYFDOSP-SLPAVUKLYLKDHLHR HUKDLHLYFVCYLTHUFHXBHPAHUKJBYVPBVZCVST- BTLVMM VYNVAALUSVYLDOPSLPUVKKLULKYLHYSFUHWWPUNZBKKK USFAOLYLHJHTLDAHWWPUNHZVMZTVLVLVARUS FYHWW PUNY-HWWPUNHATFJOHTILYKVVVAYPZZVTCPZPANYPTBAALYLKAHWW- PUNHATFJOHTILYKVVVYVSFAOPZHUKUVA.” Use frequency analysis and not brute force.

We solve this using Sage.

```python
c = CS.encoding(ct)
f = c.frequency_distribution()
dict_fd = fd.function()
list_fd = [[dict_fd[x], x] for x in dict_fd.keys()]
```
list_fd.sort()
list_fd.reverse()
list_fd

For typographical purposes, ct was printed in 5 lines. Note that ct must be entered as a string into Sage using no carriage returns. These commands produce the following output.

Sage

```
[[0.108527131782946, L], [0.0930232558139535, U], [0.089147285976905, A], [0.0581395348837209, K], [0.0465116279069767, W], [0.0465116279069767, T], [0.0348837209302326, N], [0.0348837209302326, F], [0.0310077519379845, O], [0.0271317829457364, B], [0.01937984469612403, J], [0.0155038759689922, C], [0.0116279069767442, D], [0.00775193798449612, I], [0.00387596899224806, X], [0.00387596899224806, R]]
```

We ask SymPy to output a frequency distribution, which we then sort:

SymPy

```
>>> AoC = alphabet_of_cipher()
>>> ct = 'VUJLBWVUHTFKUPNOAKYLHYFDOPSLPWVKLYLKDRLHRUKDLYVF
CLYTHUFHMHPAHUKJBYFVBZCVSTLVMMVYVAALUSVYLDOPSPLPVKRLKLH
YSFUPKPUNZBBKLUSPAOLYLJHJLHKLHWPUNHIZVMVTYLVLKLUAS FYWWPN
YWWPNUNHATJOINTLYKYYAPZSVTLCFPAZYPTBAALYLMHWWPPUNHATFJOINT
ILYKYYVUSFAPZHUUKUV

>>> n = len(ct)
>>> L = [(1.000*ct.count(x)/n,x) for x in AoC]
>>> L.sort()
>>> L.reverse()
>>> L
[(0.107569721116, L), (0.091633466135, U), (0.091633466135, H), (0.083653386454, V), (0.0717131474104, Y), (0.0717131474104, F), (0.0637450199203, A), (0.0597609561753, K), (0.0478087649402, W), (0.0438247011952, T), (0.0358565737052, F), (0.0318725099602, Z), (0.0318725099602, N), (0.027884462151, O),
```
Comparing frequencies, we might guess E is mapped to L. Using the alphabet circle in Figure 2, we might guess that the shift, i.e., the key, is $11 - 4 = 7$. Let’s try this using Sage.

\[
\text{Sage}\ 
\text{CS.deciphering(11-4, c)}
\]

The output is as follows.

\[
\text{ONCEUPONAMIDNIGHTDREARYWHILEIPONDEREDWEAKANDWEARYOVERMANYAQUINTANDCURIOUSVOLUMEOFFORGOTTENLOREWILEINODDEDNEARLYNAPPINGSUDDENLYTHERECAM\EATAPPINGASOF SOMEONE GENTLY RAPPING RAPPINGAT MY CHAMBER DOOR ONLYTHIS AND NOTHING MORE}
\]

That’s the plain text!

### 3.3 Index of coincidence

Consider a cryptosystem

We can express the index of coincidence IC for a given letter-frequency distribution as a summation:

\[
IC(p) = \sum_{c \in p} \frac{\text{count}(c, p)(\text{count}(c, p) - 1)}{N(N - 1)}
\]

where \(N\) is the length of the text \(p\) and \(\text{count}(c, p)\) is the frequency of the letter \(c\) in the message \(p\) of the alphabet. If \(p\) is a long English message then \(IC(p) \approx 0.067\).

**Lemma 1.** If \(p\) is any plaintext message and if \(c\) is a ciphertext encipherment using a monoalphabetic substitution cipher then

\[
IC(p) = IC(c).
\]
Example 2. Consider the text from Edgar Alan Poe’s “The Raven”.

```
sage: AS = AlphabeticStrings()
sage: CS = ShiftCryptosystem(AS)
sage: m = "ONCEUPONAMIDNIGHTDREARYWHILEIPONDEREDWEAKANDWEARYOVERMANYAQAIANTANDCURIOUSVOLUMEOFFORGOTTENLOREWHILEINODDEDNEARLYNAPPINGSUDDENLYTHERECAMEATAPINGASOFONEGENTLYRAPPINGRAPPINGATMYCHAMBERDOORTISSOMEVISITE\RUMUTTEREDTAPPINGATMYCHAMBERDOORONLYTHISANDNOTHINGMORE"
sage: m = CS.encoding(m)
sage: m.coincidence_index() 0.0614424034024070
sage: c = CS.enciphering(1, m)
sage: c.coincidence_index() 0.0614424034024070
sage: VC = VigenereCryptosystem(AS, 3)
sage: key = AS("KEY")
sage: m = VC.encoding(raven1) # raven1 is in an example
sage: c = VC.enciphering(key, m)
sage: m.coincidence_index() 0.0614424034024070
sage: c.coincidence_index() 0.0468132597351673
```

Note that the index of coincidence did not change under a monoalphabetic substitution cipher but it did change under a polyalphabetic substitution cipher.

4 Substitution cipher

A substitution cipher is a cryptosystem by which string “units” are replaced with other strings. When the “units” are single letters then it is called a simple substitution. A monoalphabetic substitution cipher uses fixed substitution over the entire message, whereas a polyalphabetic substitution cipher uses a number of substitutions at different positions in the message.

The algorithm to encipher a message of a simple monoalphabetic substitution cipher is simply to apply a fixed permutation $p$ of the alphabet to each of the characters in the plaintext, one after the other. The algorithm to decipher a message of a simple monoalphabetic substitution cipher is simply to apply the inverse of this fixed permutation to each of the characters in the ciphertext, one after the other.

In a simple substitution, the key is a permutation of the 26 letters of the alphabet. Therefore, the key space is 26!
Here the key is the substitution sending $A \to Z$, $B \to Y$, $C \to X$, $D \to V$, $E \to U$, $F \to T$, $G \to S$, $H \to R$, $I \to Q$, $J \to P$, $K \to O$, $L \to N$, $M \to L$, $N \to K$, $O \to J$, $P \to I$, $Q \to H$, $R \to G$, $S \to F$, $T \to E$, $U \to D$, $V \to C$, $W \to B$, $X \to A$.

One method of creating a key is to first write out a keyword, remove any repeated letters in it, then writing all the remaining letters in the alphabet in the usual order. Using this system, the keyword “zebras” gives us the following key: “ZEBRASCDFGHIJKLMNPQSTUVWXY.” This method is closely related to the Bifid cipher described in the next section.

Security issues: A disadvantage of this method of the substitution cipher is that the frequency distribution remains unchanged. Although the key space is large (26! ≈ 4 × 10^{26}), this cipher is not very strong, and is easily broken. Provided the message is of reasonable length (say 50 or so), the cryptanalyst can deduce the probable meaning of the most common symbols by analyzing the frequency distribution of the ciphertext.

Example 3. Let us consider the example of the cipher from Poe’s “The Gold Bug” [Po]. The cipher is

```
*53++(1305))6*;4826(4+)+4+;.;806*;4818]60)85;1+8*:++;:**8183(88)5*!; 46(88*96*7;8)**+(485;5*12:*++;4956+2(5*+4)8]8*;4069285;);618)4**; 1(+9;48081;8;8*1;48185;4)485528806*81(+9;48;88;4(*73;48)4*;161;: 188;+?;
```

To analyze this in Sage, we must exchange these 21 symbols (5, 3, 8) for the capital letters of the alphabet (5 → A, 3 → B, 8 → C). This gives us a cipher:

```
*ABCDBEAFGGHIJKSKGFJCFJFCFLIKEGHIJKDKMGEFFKAINCKHCIP1OCCHDDKBPKKFAHDI JGP1KKHRAHGHQIKFRCPIJKAFIAHDSHPHCPIJRAHSPAHTJFKKNNJJEGRSAFIPDGKDFJCCIP NPCRIJKKNIKCKNIJKDAKUKEKMLNPCRIRJIKIPKCIJPCQBJIKFJCINGNIONKKICQI*
```
sage: AS = AlphabeticStrings()
sage: AS
Free alphabetic string monoid on A-Z
sage: goldbugcipher="ABCCDBEAFFGHIJKSGFJCLFJCFIKEGHIJKDKMGEFFK
AINCKHOP10CHKBFPKKFAHDLJQPIKKHRGQIKFHCPIJKAfAHDSOHCP
IJRAGHSPAHJTJKHH1JEGSKAF1FGDKFJCCINFPCR1JKEKNKOKCN
JDKAJFJ
KADASKKEGHNPFCR1JKIPKCIJPQBJIKKFJCINGNIONKCKIQI"
sage: m = AS.encoding(goldbugcipher)
sage: m.frequency_distribution()
Discrete probability space defined by {A: 0.0579710144927536,
C: 0.0821256038647343, B: 0.0193236714975845, E: 0.0289855072463768,
D: 0.0386473429951691, G: 0.0531400966183575, F: 0.0772946859903381,
I: 0.125603864734299, H: 0.0676328902415459, K: 0.164251207729468,
J: 0.0917874396135265, M: 0.00966183574879227, L: 0.00483091787439614,
O: 0.0241545893719807, N: 0.0386473429951691, Q: 0.0144927536231884,
P: 0.0483091787439614, S: 0.0241545893719807, R: 0.0241545893719807,
T: 0.00483091787439614};

sage: m.character_count()
{A: 12, C: 17, B: 4, E: 6, D: 8, G: 11, F: 16, I: 26, H: 14, K: 34, J: 19, M: 2,
L: 1, O: 5, N: 8, Q: 3, P: 10, S: 5, R: 5, T: 1};
sage: fd = m.frequency_distribution()
sage: dict_fd = fd.function()
sage: list_fd = [(dict_fd[x], x) for x in dict_fd.keys()]
sage: list_fd.sort()
sage: list_fd.reverse()
sage: list_fd
[[0.164251207729468, K], [0.125603864734299, I], [0.0917874396135265, J],
[0.0821256038647343, C], [0.0772946859903381, F], [0.0676328902415459, H],
[0.0579710144927536, A], [0.0531400966183575, G], [0.0483091787439614, P],
[0.0386473429951691, N], [0.0386473429951691, D], [0.0289855072463768, E],
[0.0241545893719807, S], [0.0241545893719807, R], [0.0241545893719807, O],
[0.0193236714975845, B], [0.0144927536231884, Q], [0.00966183574879227, M],
[0.00483091787439614, T], [0.00483091787439614, L]]

Now we decipher it using a key (obtained by trial-and-error[^3]):

sage: SC = SubstitutionCryptosystem(AS)
sage: key = "AGODLSINTHBPFVYRUMBCJKQWXZ"; key
AGODLSINTHBPFVYRUMBCJKQWXZ
sage: SC.enciphering(key, AS(goldbugcipher))
AGOODGLASSINTHEBISHOPSHOSTELINTHEDEVILSSEATFORTYONEDEGREESANDTHIRTEENMINUTES
NORTHEASTANDBYNORTHMAINBRANCHEVENTHILMEASTSIDESHOTFROMTHELEFTYEEOFTHE
DEATHSHEADABEE LINEFROM THE TREE THROUGH THE SHOT FIFTY F EET OUT

[^3]: I used a site [http://vorpal.nebrwesleyan.edu/~mcclung/ciphers.php](http://vorpal.nebrwesleyan.edu/~mcclung/ciphers.php) but there are many other similar sites on the web.
(I've written down the inverse of the key, so enciphering is really deciphering.)

```python
>>> goldbug="ABCCDBEAFFGHIJKSGFJCFJCFLIKEGHIJKDRMGEFFKAINCKHOCPIO
  CHKDKBPKFKAHDIJGPIKKnRFGHQIKFHCPIJKAFAIHDSONCHFJLRAHSHPAHTJFKMKHIJE
  GRSKAFIFGDNFJCICINPCRIJKEKNIKOJNIKCIJKDKAIJFJKADASKKEHKNPCRIJJKIPKI
  JPCQBJJJKFCJINGNIONKCIQI"
>>> key = "AGODLSINTHEPVFYRUMBCJKQWXZ"
>>> encipher_substitution(goldbug, key)
AGOODGLASSINTHEBISHOSHOSPTELINTHEDEVILSSEATFOENYORTYONEDEGREESAND
THIRTEENMINUTESPONORHOSPTELINORTHMAINBRANCHSEVENTHLIMBEASTSIDESHOOT
FROMTHELEFTYEOTHEDEATHSHEADABEELINEFROMTHETREETHROUGHTHESHOTFIFTYFEETOUT
```

In either case, this gives us the plaintext in “The Gold Bug”:

“A good glass in the bishop’s hostel in the devil’s seat forty-one
degrees and thirteen minutes northeast and by north main branch
seventh limb east side, shoot from the left eye of the death’s head
a bee-line from the tree through the shot fifty feet out.”

5 Bifid cipher

The Bifid cipher was invented around 1901 by Felix Delastelle. It is a “fractional substitution” cipher, where letters are replaced by pairs of symbols from a smaller alphabet. The cipher uses a $5 \times 5$ square filled with some ordering of the alphabet, except that I’s and J’s are identified (this is a so-called Polybius square). There is a $6 \times 6$ analog if you add back in the J’s and also append onto the usual 26 letter alphabet, the digits 0, 1, \ldots, 9). According to Helen Gaines’ book [C], this type of cipher was used in the field by the German Army during World War I.

**ALGORITHM:** (6x6 case, encryption)

**INPUT:**
- key - a string of letters for the key (no repetitions)
- pt - a string of letters for the plaintext (length n)

**OUTPUT:**
ct - ciphertext message

Step 1:
Create the 6x6 Polybius square S associated to the k as follows:
bottom,
a) starting top left, moving left-to-right, top-to- place the letters
of the key into a 6x6 matrix,
b) when finished, add those letters of the alphabet, followed
by the digits 0, ..., 9, which are not in the key, until the 6x6
square is filled
Step 2:
Create a list P of pairs of numbers which are the coordinates
in the Polybius square of the letters in pt.
Step 3:
Let L1 be the list of all first coordinates of P (length of L1= n),
let L2 be the list of all second coordinates of P (so the length of L2
is also n)
Step 4:
Let L be the concatenation of L1 and L2 (so length L = 2n),
except that consecutive numbers are paired (L[2i], L[2i+1]).
You can regard L as a list of pairs of length n.
Step 5: Let C be the list of all letters which are of the form
S[i,j], fr all (i,j) in L. As a string, this is the ciphertext ct.

Example 4. As an example of a Polybius square for the Bifid cipher, pick
the key to be “encrypt” (as in [McA]). In that case, the Polybius square is

\[
\begin{pmatrix}
E & N & C & R & Y \\
P & T & A & B & D \\
F & G & H & I & K \\
L & M & O & Q & S \\
U & V & W & X & Z \\
\end{pmatrix}
\]

If the key is “encrypt” and the plaintext is “meet me on monday” then the
ciphertext is “LNLLQNPPNPGADK”. We can verify this using SymPy.

```python
>>> pt = "meet me on monday"
>>> key = "encrypt"
>>> print bifid5_square(key)
[E, N, C, R, Y]
```
BTW, the 6 × 6 analog is:

\[
\begin{pmatrix}
E & N & C & R & Y & P \\
T & A & B & D & F & G \\
H & I & J & K & L & M \\
O & Q & S & U & V & W \\
X & Z & 0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 & 8 & 9 \\
\end{pmatrix}
\]

If the key is “encrypt” and the plaintext is “meet me on monday at 8am” then the ciphertext is “HNHOKNTA5MEPEGNQZYG”. We can verify this using SymPy.

```
>>> key = "encrypt"
>>> print bifid_square(key)
[E, N, C, R, Y, P]
[T, A, B, D, F, G]
[H, I, J, K, L, M]
[O, Q, S, U, V, W]
[X, Z, 0, 1, 2, 3]
[4, 5, 6, 7, 8, 9]
>>> pt = "meet me on monday at 8am"
>>> encipher_bifid6(pt, key)
HNHOKNTA5MEPEGNQZYG
>>> encipher_bifid6(pt, key, verbose = True)
[[2, 5], [0, 0], [0, 0], [1, 0], [2, 5], [0, 0], [3, 0], [0, 1], [2, 5], [3, 0], [0, 1], [1, 3], [1, 1], [0, 4], [1, 1], [1, 0], [5, 4], [1, 1], [2, 5]]
HNHOKNTA5MEPEGNQZYG
>>> ct = "HNHOKNTA5MEPEGNQZYG"
>>> decipher_bifid6(ct, key)
MEETMEONMONDAYAT8AM
```

ALGORITHM: (6x6 case, decryption)

INPUT:
key - a string of letters for the key (no repetitions)
ct - a string of letters for the ciphertext (length n)

OUTPUT:
pt - plaintext message

Step 1:
Create the 6x6 Polybius square S associated to the key as follows:
bottom,
a) starting top left, moving left-to-right, top-to- place the letters of the key into a 6x6 matrix,
b) when finished, add those letters of the alphabet, followed by the digits 0, ..., 9, which are not in the key, until the 6x6 square is filled
We call the list of all symbols in the square, taken top-to-bottom, left-to-right, the "long key"
Step 2:
From the "long key", find the coordinates corresponding to each symbol in the cipher text: the coordinate of x is (i,j) if x is the m-th element of the long key and m = 6i+j.
Step 3: Read the i’s then the j’s off consecutively, then use the Polybius square to find the corresponding symbols.
This is the plaintext pt.

Example 5. What about the larger bifid squares? The problem is - how are the squares to be filled in? With integers?
Here is an example:

```python
>>> key = "gold bug"
>>> print bifid7_square(key)
[ G, O, L, D, B, U, A]
[ C, E, F, H, I, J, K]
[ M, N, P, Q, R, S, T]
[ V, W, X, Y, Z, 0, 1]
[ 2, 3, 4, 5, 6, 7, 8]
[ 9, 10, 11, 12, 13, 14, 15]
[16, 17, 18, 19, 20, 21, 22]

Let us use this to encipher “Meet me on Monday at 8am.”
```
You see the ambiguity in deciphering: Is the “11“ an 11 or two 1’s? You can’t use numbers in $7 \times 7$ or higher bifid squares without running into these ambiguities.

In-class team challenge: encrypt a message, then give it to another team to decrypt.

6 The Vigenère cipher

From what I’ve read, the Vigenère cipher is named after Blaise de Vigenère, a sixteenth century diplomat and cryptographer, by a historical accident. Vigenère actually invented a different and more complicated cipher. The so-called “Vigenère cipher “ cipher was actually invented by Giovan Batista Belaso in 1553. In any case, it is this cipher which we shall discuss next.

This cipher has been re-invented by several authors, such as author and mathematician Charles Lutwidge Dodgson (Lewis Carroll) who claimed his 1868 “The Alphabet Cipher“ was unbreakable. Several others claimed the so-called Vigenère cipher was unbreakable (e.g., the Scientific American magazine in 1917). However, Friedrich Kasiski and Charles Babbage broke the cipher in the 1800’s: once it is known that the key is, say, $n$ characters long, frequency analysis can be applied to every $n-th$ letter of the ciphertext to determine the plaintext. This method is called Kasiski examination, although it was first discovered by Babbage.

This cipher was used in the 1700’s, for example, during the American Civil War. The Confederacy used a brass cipher disk to implement the Vigenère cipher (now on display in the NSA Museum in Fort Meade).

The so-called Vigenère cipher is a generalization of the shift cipher. Whereas the shift cipher shifts each letter by the same amount (that amount being the key of the shift cipher) the so-called Vigenère cipher shifts a letter by an amount determined by the key, which is a word or phrase known only to the sender and receiver).
For example, if the key was a single letter, such as “C”, then the so-called Vigenère cipher is actually a shift cipher with a shift of 2 (since “C” is the 2nd letter of the alphabet, if you start counting at 0). If the key was a word with two letters, such as “CA”, then the so-called Vigenère cipher will shift letters in even positions by 2 and letters in odd positions are left alone (or shifted by 0, since “A” is the 0th letter, if you start counting at 0).

**ALGORITHM:**

**INPUT:**
- key - a string of upper-case letters (the secret "key")
- m - string of upper-case letters (the "plaintext" message)

**OUTPUT:**
- c - string of upper-case letters (the "ciphertext" message)

Identify the alphabet A, ..., Z with the integers 0, ..., 25.

1. **Step 1:** Compute from the string key a list L1 of corresponding integers. Let n1 = len(L1).
2. **Step 2:** Compute from the string m a list L2 of corresponding integers. Let n2 = len(L2).
3. **Step 3:** Break L2 up sequentially into sublists of size n1, and one sublist at the end of size <=n1.
4. **Step 4:** For each of these sublists L of L2, compute a new list C given by C[i] = L[i]+L1[i] (mod 26) to the i-th element in the sublist, for each i.
5. **Step 5:** Assemble these lists C by concatenation into a new list of length n2.
6. **Step 6:** Compute from the new list a string c of corresponding letters.

The Tabula Recta in Figure 3 is a useful aide to enciphering and deciphering this cipher. To encipher using this table, simply look up each letter of the plaintext along the top (this specifies a column), then the corresponding letter of the key along the side (this specifies the row), and record as ciphertext that letter which is in that row and column of the table. To decrypt, using this table, simply look up each letter of the ciphertext along the top (this specifies a column), then find the corresponding letter of the negative of the key along the side (this specifies the row), and record as plaintext that letter which is in that row and column of the table. For example, the key is USNA (which corresponds to the sequence (20, 18, 13, 0)) then the negative of the key is GINA (which corresponds to the sequence (6, 8, 13, 0) = −(20, 18, 13, 0) (mod 26)).
Sage

```python
sage: AS = AlphabeticStrings()
sage: A = AS.alphabet()
sage: key = AS("A")
sage: VC = VigenereCryptosystem(AS, 1)
sage: m = VC.encoding("Beat Army!"); m
BEATARMY
sage: VC.enciphering(key, m)
BEATARMY
sage: key = AS("B")
sage: VC.enciphering(key, m)
CFBUBSNZ
sage: VC.deciphering(key, c)
BEATARMY
sage: VC.enciphering(AS("Z"), c)
BEATARMY
```

This cipher is also implemented in SymPy:

```python
>>> key = "encrypt"
>>> pt = "meet me on monday"
>>> encipher_vigenere(pt, key)
QRGKKTHRZQEBPR
```

Next, we decryption:

```python
sage: VC = VigenereCryptosystem(AS, 2)
sage: key = AS("CA")
sage: m = VC.encoding("Beat Army!"); m
BEATARMY
sage: VC.enciphering(key, m)
DECTCROY
```
To decipher “QRGKKTHRZQEBPR” (with some spaces added) in SymPy, use the following commands.

```
>>> key = "encrypt"
>>> ct = "QRGK kt HRZQE BPR"
>>> decipher_vigenere(ct, key)
MEETMEONMONDAY
```

The cipher Vigenère actually discovered is an “auto-key” cipher described as follows.

**ALGORITHM:**

**INPUT:**

- key - a string of upper-case letters (the secret "key")
- m - string of upper-case letters (the "plaintext" message)

**OUTPUT:**

- c - string of upper-case letters (the "ciphertext" message)

Identify the alphabet A, ..., Z with the integers 0, ..., 25.

Step 1: Compute from the string m a list L2 of corresponding integers. Let n2 = len(L2).

Step 2: Let n1 be the length of the key. Concatenate the string key with the first n2-n1 characters of the plaintext message. Compute from this string of length n2 a list L1 of corresponding integers. Note n2 = len(L1).

Step 3: Compute a new list C given by C[i] = L1[i]+L2[i] (mod 26), for each i.
Step 5: Compute from the new list a string c of corresponding letters.

7 Number theory background

It is convenient to replace letters and other characters by numbers and have our cipher act by permuting sequences of numbers instead of sequences of characters. For this reason, a good background in number theory (the mathematics of the integers) is very useful in understanding the mathematical aspects of cryptography.

7.1 Binary representation of a number

The material below can be found in any standard textbook, for example [JKT].

Each natural number is most commonly written in decimal form (or base 10),

\[ a = a_k 10^k + \ldots + a_1 10 + a_0, \]

where 0 \leq a_i \leq 9 are the digits. (Without loss of generality we may assume that the leading digit \( a_k \) is non-zero.) This representation is unique. (In spite of the fact that the decimal representation of a real number is not unique - 1.0 = .9999...) Similarly, each natural number can be written (uniquely) in a binary expansion (or base 2),

\[ a = a_k 2^k + \ldots + a_1 2 + a_0, \]

where 0 \leq a_i \leq 1 are the bits. The binary representation of \( a \) is written as \( a \sim a_k \ldots a_1 a_0 \). Clearly, \( a \) is even if and only if \( a_0 = 0 \). To find the binary expansion of a natural number, perform the following steps.

1. Find the “leading bit” \( a_k \) by determining the largest power of 2 less than or equal to \( a \). Call this power \( k \) and let \( a_k = 1 \).

2. Subtract this power from \( a \) and replace \( a \) by this difference.

3. If the result is non-zero, go to step 1; otherwise, stop.

This determines all the non-zero bits in the binary representation of \( a \). The other bits are 0.
Example 6. Find the binary representation of 130. The largest power of 2 less than or equal to 130 is $2^7 = 128$, so $a_7 = 1$. The largest power of 2 less than or equal to $2 = 130 - 128$ is $2^1 = 2$, so $a_1 = 1$. The other bits are zero: $a_6 = a_5 = a_4 = a_3 = a_2 = a_0 = 0$, so

$$130 = 128 + 2 \sim 10000010.$$  

More generally, let us fix an integer $m > 1$. Each natural number can be written in an $m$-ary expansion

$$a = a_k m^k + \ldots + a_1 m + a_0,$$

where $0 \leq a_i \leq m - 1$ are the $m$-ary digits. Again, the $m$-ary representation of $a$ is written as $a \sim a_k \ldots a_1 a_0$. Clearly, $m | a$ if and only if $a_0 = 0$. To find the $m$-ary expansion of a natural number, perform the following steps.

1. Find $a_k$ by determining the largest power of $m$ less than or equal to $a$. Call this power $k$.

2. Find the largest positive integer multiple of this power which is less than or equal to $a$. This multiple will be the $k$-th digit $a_k$.

3. Subtract $a_k m^k$ from $a$ and replace $a$ by this difference.

4. If the result is non-zero, go to step 1; otherwise, stop.

Example 7. Find the 3-ary representation of 211. The largest power of 3 less than or equal to 211 is $3^4 = 81$, and $211 > 2 \cdot 81 = 162$ so $a_4 = 2$. The largest power of 3 less than or equal to 49 = 211 − 162 is $3^2 = 27$, and $49 < 2 \cdot 27$ so $a_3 = 1$. The largest power of 3 less than or equal to 22 = 49−27 is $3 = 3^1$, and $22 > 2 \cdot 9$ so $a_2 = 2$. The largest power of 3 less than or equal to 4 = 22 − 18 is $3 = 3^1$, and $4 < 2 \cdot 3$ so $a_1 = 1$. The last “bit” is 1: $a_4 = 2$, $a_3 = 1$, $a_2 = 2$, $a_1 = 1$, $a_0 = 1$, so

$$211 = 2 \cdot 3^4 + 1 \cdot 3^3 + 2 \cdot 3^2 + 1 \cdot 3 + 1 \sim 12211.$$  

Example 8. Convert 100101 from binary to 5-ary. In decimal (or “10-ary”), 100101 is

$$1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 32 + 4 + 1 = 37.$$  

In 5-ary,

$$37 = 1 \cdot 5^2 + 2 \cdot 5^1 + 2 \cdot 5^0 \sim 122.$$
7.2 Modular inverses and extended Euclidean algorithm

For a positive integer \( m \), two integers \( a \) and \( b \) are said to be congruent modulo \( m \), written:

\[
a \equiv b \pmod{m},
\]

if their difference \( a - b \) is an integer multiple of \( m \). We write \( m|n \) if the integer \( m \) divides the integer \( n \), so \( m|n \) if and only if \( n \equiv 0 \pmod{m} \). The number \( m \) is called the modulus of the congruence. This congruence symbol \( \equiv \) satisfies the same properties as = between integers, except (possibly) for the cancellation law. All the following hold:

- if \( a \equiv b \pmod{m} \) and \( c \equiv d \pmod{m} \) then
  \[
  ac \equiv bd \pmod{m} \quad \text{and} \quad a + c \equiv b + d \pmod{m}.
  \]
- if \( a \equiv b \pmod{m} \) and \( b \equiv c \pmod{m} \) then
  \[
  a \equiv c \pmod{m}.
  \]
- \( a \equiv b \pmod{m} \) if and only if \( b \equiv a \pmod{m} \).
- For all \( c \in \mathbb{Z} \),
  \[
  a \equiv a + cm \pmod{m}.
  \]

As an example, we prove \( ac \equiv bd \pmod{m} \): Since \( a \equiv b \pmod{m} \) and \( c \equiv d \pmod{m} \), we have \( ac \equiv bc \pmod{m} \) and \( bc \equiv bd \pmod{m} \). Therefore \( m \) divides \( ac - bc \) and also divides \( bc - bd \). It therefore must divide their difference, \( ac - bd \). This completes the proof. The others are left to the student, if he or she is interested.

**Example 9.** First, let’s look at the special case \( m = 12 \) and form the multiplication table \( \pmod{12} \) for all the integers in \{1, \ldots, 11\}. 

40
Note that the only elements $a$ in $\{1, \ldots, 11\}$ such that there exists a $b$ in $\{1, \ldots, 11\}$ for which $a \cdot b \equiv 1 \pmod{12}$ are $a = 1, 5, 7, 11$.

Next, let’s look at the special case $m = 11$ and form the multiplication table (mod 11) for all the integers in $\{1, \ldots, 10\}$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>9</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Note that for every element $a$ in $\{1, \ldots, 10\}$ there exists a $b$ in $\{1, \ldots, 10\}$ for which $a \cdot b \equiv 1 \pmod{11}$.

If $bc \equiv 1 \pmod{m}$ then we say $b$ is the modular inverse of $c \pmod{m}$, written

$$b = c^{-1} \pmod{m}.$$  

**Lemma 10.** The inverse of $c \pmod{m}$ exists if and only if $c$ and $m$ are relatively prime, i.e., they have no prime factors in common.
The greatest common divisor of \( c, m \) is denoted \( \gcd(c, m) \).

We say \( c, m \) are relatively prime if and only if \( \gcd(c, m) = 1 \). For example, 7 and 12 are relatively prime but 12 and 15 are not. The number of positive integers \( c \) less than \( m \) which are relatively prime to \( m \) is called \textit{Euler’s phi function} (also called \textit{Euler’s totient function}) and denoted

\[
\phi(m) = |\{1 \leq c \leq m - 1 \mid \gcd(c, m) = 1\} = m \prod_{p|m}(1 - 1/p). \tag{1}
\]

For example, \( \phi(90) = 90(1 - 1/2)(1 - 1/3)(1 - 1/5) = 24 \), so there are exactly 24 integers less than 90 which have no 2 or 3 or 5 in their prime factorization.

\textit{Question} How do you compute the modular inverse \( c^{-1} \mod m \) (if it exists)?

There are several methods, some better than others.

\textbf{Method 1}: Search over all \( b \in \{1, 2, \ldots, m - 1\} \) and test if \( bc \equiv 1 \mod m \) holds or not.

This is very inefficient.

\textbf{Method 2}: Use

\textbf{Lemma 11}. (Fermat’s Little Theorem/Euler’s Theorem) If \( \gcd(c, m) = 1 \) then

\[ c^{\phi(m)} \equiv 1 \mod m. \]

This gives us a formula for the inverse:

\[ c^{-1} = c^{\phi(m) - 1} \mod m. \tag{2} \]

It turns out \( c^{\phi(m) - 1} \mod m \) is easy to compute using repeated squaring:

To compute \( c^n \mod m \) (where \( n > 1 \))

1. Write \( n \) as a sum of powers of 2 (the “binary expansion” of \( n \)):

\[ n = 2^k + \ldots \text{ (smaller powers of 2)}, \]

where \( k \) is about \( \log_2(n) \).

2. To compute \( c^{2^k} \), compute

\[ c_1 = c^2 \mod m, \]

then
3. \[ c_2 = c_1^2 \pmod{m} = (c^2)^2 = c^4 \pmod{m}, \]

   etc, until you get to \( c^k \). This requires \( k \) multiplications \( \pmod{m} \).

4. To compute \( c^n \pmod{m} \), multiply those \( c_1, c_2, \ldots, c_k \) occuring the in the binary expansion above.

   As there are at most \( k = O(\log n) \) multiplications in the last step, and each multiplication takes \( O((\log n)^2) \) operations, in general, we can compute \( c^n \pmod{m} \) in \( O((\log n)^3) \) steps\(^4\). This is an example of an algorithm which is “linear time” (the number of computations is linear in the number of digits needed to write \( n \) down).

**Example 12.** Here is a table of inverses \( \pmod{11} \):

<table>
<thead>
<tr>
<th>( a )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{1}{a} )</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

For example, \( 7^{-1} \equiv 8 \pmod{11} \) because \( 7 \cdot 8 = 56 = 1 + 55 \equiv 1 \pmod{11} \).

**Method 3:** (Extended Euclidean algorithm) This is based on

**Lemma 13.** (Bezout’s Lemma) If \( a > 1 \) and \( b > 1 \) are integers with greatest common divisor \( d \), then there exist integers \( x \) and \( y \) such that

\[ ax + by = d. \]

**proof:** Consider the set

\[ \langle a, b \rangle = \{ ra + sb \mid r \in \mathbb{Z}, \ s \in \mathbb{Z} \}. \]

Since \( d \) divides \( a \) and \( b \), this set \( \langle a, b \rangle \) must be contained in the set

\[ \langle d \rangle = \{ td \mid t \in \mathbb{Z} \}, \]

i.e., \( \langle a, b \rangle \subset \langle d \rangle \).

Let \( c > 0 \) be the smallest integer such that

\[ f(n) = O(\log(n)) \]

The “big O” notation is defined as follows: We say \( f(n) = O(\log(n)) \) if and only if there is a constant \( C > 0 \) such that \( f(n) \leq C \log(n) \) for all \( n > 1 \). The constant \( C \) will, in this case, implicitly depends on \( m \).
\[ \langle c \rangle \subset \langle a, b \rangle. \]

(Note that \( \langle d \rangle \subset \langle a, b \rangle \) so we have \( c \leq d \).) Suppose now \( \langle c \rangle \neq \langle a, b \rangle \), so \( \langle c \rangle \) is a proper subset of \( \langle a, b \rangle \). Suppose \( n = ax + by \) is the smallest positive integer in \( \langle a, b \rangle \) which is not in \( \langle c \rangle \). By the integer “long division” algorithm, there is a remainder \( r < c \) and a quotient \( q \) such that \( n = qc + r \). But \( n \in \langle a, b \rangle \) and \( qc \in \langle c \rangle \subset \langle a, b \rangle \), so therefore \( r = n = qc \in \langle a, b \rangle \). Therefore, \( \langle r \rangle \subset \langle a, b \rangle \).

This is a contradiction to the assumption that \( c \) was as small as possible. Therefore,

\[ \langle c \rangle = \langle a, b \rangle. \]

In fact, \( \langle c \rangle = \langle a, b \rangle \) implies \( c | a \) and \( c | b \), so \( c = d = \gcd(a, b) \). Bézout’s lemma follows immediately from \( \langle d \rangle = \langle a, b \rangle \). □

**Example 14.** For example, \( \gcd(12, 15) = 3 \). Obviously, \( 15 - 12 = 3 \), so with \( a = 12 \) and \( b = 15 \), we have \( x = -1 \) and \( y = 1 \).

*Here is the SymPy syntax:*

```
>>> gcdex(15,12)
(1, -1, 3)
```

*Here is the Sage syntax:*

```
sage: xgcd(15,12)
(3, 1, -1)
```

Additionally, \( d \) is the smallest positive integer for which there are integer solutions \( x \) and \( y \) for the preceding equation.

- Initial table: \( (a < b) \)

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>q</th>
<th>r</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td>b</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

44
• As you increment $i$, apply the recursive equations: $q_i = \lfloor r_{i-2}/r_{i-1} \rfloor$, $r_i = r_{i-2} - r_{i-1}q_i$, $u_i = u_{i-2} - u_{i-1}q_i$, $v_i = v_{i-2} - v_{i-1}q_i$.

• Stop when $r_k = 0$ and then let $x = v_{k-1}$, $y = u_{k-1}$.

To compute the inverse of $c \pmod{m}$, assume $\gcd(c, m) = 1$ and compute $x, y$ such that $cx + my = 1$. We have $x \pmod{m} = c^{-1} \pmod{m}$.

How many steps does this take in the worst-case situation? This method requires $O((\log n)^2)$ operations and is very efficient.

Why is this?

Suppose that $b > a$ and that $b$ is an $n$-bit integer (i.e., $b \leq 2^n$). The recursive equations are repeated over and over, as long as $r_i$ (which gets re-assigned each step of the loop) stays strictly positive.

Some notation will help us understand the steps better. Call $(a_0, b_0)$ the original values of $(a, b)$. After the first step of the while loop, the values of $a$ and $b$ get re-assigned. Call these updated values $(a_1, b_1)$. After the second step of the while loop, the values of $a$ and $b$ get re-assigned again. Call these updated values $(a_2, b_2)$. Similarly, after the $j$-th step, denote the updated values of $(a, b)$, by $(a_j, b_j)$. After the first step, $(a_0, b_0) = (a, b)$ is replaced by $(a_1, b_1) = (b \pmod{a}, a)$. Note that $a > b/2$ implies $b \pmod{a} < b/2$, therefore we must have either $0 \leq a_1 \leq b_0/2$ or $0 \leq b_1 \leq b_0/2$ (or both). If we repeat this while loop step again, then we see that $0 \leq a_2 \leq b_0/2$ and $0 \leq b_2 \leq b_0/2$. Every 2 steps of the while loop, we decrease the value of $a$ by a factor of 2. Therefore, this algorithm requires $O(\log_2(b))$ steps. The $i$th step takes $O((\log n)(\log q_i))$ operations (due to the multiplication), so the total number of operations is

$$\prod_{i=1}^{k-1} O((\log n)(\log q_i)) = O((\log n)^2).$$

Such an algorithm is called a quadratic time algorithm, since it complexity is bounded by a polynomial in $n$ of degree 2.
Example 15.

<table>
<thead>
<tr>
<th>i</th>
<th>q</th>
<th>r</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>2697</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2553</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>144</td>
<td>1</td>
<td>−1</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>105</td>
<td>−17</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>39</td>
<td>18</td>
<td>−19</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>27</td>
<td>−53</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>12</td>
<td>71</td>
<td>−75</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>−195</td>
<td>206</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thus $k = 7$, so $x = v_6 = −195$, $y = u_6 = 206$. Indeed, $3 = \gcd(2697, 2553) = (−195) \cdot 2697 + (206) \cdot 2553$.

### 7.3 Finite fields

A finite field is, as you might expect, a finite set which has a multiplication operation “.” and an addition operation “+” and, together, they satisfy all the expected properties of a field such as the real or complexes or rationals.

**Definition 16.** A field is a set $F$ which has two binary operations, denoted $+$ and $\cdot$, satisfying the following properties. For all $a, b, c \in F$, we have

1. $a + b = b + a,$ \quad (“addition is commutative”)
2. $a \cdot b = b \cdot a,$ \quad (“multiplication is commutative”)
3. $(a + b) + c = a + (b + c),$ \quad (“addition is associative”)
4. $(a \cdot b)c = a(b \cdot c),$ \quad (“multiplication is associative”)
5. $(a + b) \cdot c = a \cdot c + b \cdot c,$ \quad (“distributive”)
6. there is an element $1 \in F$ such that $a \cdot 1 = a,$ \quad (“$1$ is a multiplicative identity”)
7. there is an element $0 \in F$ such that $a + 0 = a$ \quad (“$0$ is a additive identity”),
8. if $a \neq 0$ then there is an element, denoted $a^{-1}$, such that $a \cdot a^{-1} = 1$ \quad (“the inverse of any non-zero element exists”).
Here is the simplest example, sometimes called the Boolean field, written $GF(2)$.  

**Example 17.** Let $F = \{0, 1\}$, with two binary operations, $+$ (addition (mod 2)), and $\cdot$ (integer multiplication). This field is used in the mathematics of electrical circuits, with 1 being “on” and 0 being “off”.  
The addition table is

\[
\begin{array}{c|cc}
+ & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

The multiplication table is

\[
\begin{array}{c|cc}
\cdot & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

**Example 18.** Let $F = \{0, 1, 2\}$, with two binary operations, $+$ (addition), and $\cdot$ (multiplication), where each is computed (mod 3). This field is sometimes denoted $GF(3)$.  
The addition table is

\[
\begin{array}{c|ccc}
+ & 0 & 1 & 2 \\
\hline
0 & 0 & 1 & 2 \\
1 & 1 & 2 & 0 \\
2 & 2 & 0 & 1 \\
\end{array}
\]

The multiplication table is

\[
\begin{array}{c|ccc}
\cdot & 0 & 1 & 2 \\
\hline
0 & 0 & 0 & 0 \\
1 & 0 & 1 & 2 \\
2 & 0 & 2 & 1 \\
\end{array}
\]

If $p$ is any prime, then the integers (mod $p$), written $\mathbb{Z}/p\mathbb{Z}$, is a finite field with $p$ elements. It will also be denoted $GF(p)$ and is called a prime (Galois) field.

However, if $m$ is any composite then $\mathbb{Z}/m\mathbb{Z}$ (the integers (mod $m$)) is not a field. For example, $\mathbb{Z}/4\mathbb{Z}$ and $\mathbb{Z}/6\mathbb{Z}$ is not a field. In neither case does the element 2 have an inverse. (If you don’t know this already, then this is a good exercise for yourself to verify this.)
Example 19. Here is an example of using Sage to compute in $GF(5)$:

```sage
sage: GF5 = GF(5)
sage: F = GF(5)
sage: a = F(2); b = F(3)  # 2 and 3 in GF(5)
sage: a+b
  0
sage: -a
  3
sage: -b
  2
sage: 1/a
  3
sage: 1/b
  2
```

Sage computed the additive inverse of 2 in $GF(5)$ and the multiplicative inverse of 2 in $GF(5)$. Both (by an unusual coincidence) are equal to 3.

Here is a similar example using SymPy:

```python
>>> F = FF(2)
>>> F = FF(5)
>>> a= F(1)
>>> a+a
2 mod 5
>>> a+a+a
3 mod 5
>>> a+a+a+a
4 mod 5
>>> a+a+a+a+a
0 mod 5
>>> b = a+a+a
>>> a*b
3 mod 5
>>> a/b
2 mod 5
```
Here are some SymPy commands to produce (more or less) a multiplication table for $\mathbb{Z}/5\mathbb{Z}$:

```
>>> L = ["*"]+range(5)
>>> L = L+[i]+[(i*x)%5 for x in range(5)] for i in range(5)]
>>> L
[[*, 0, 1, 2, 3, 4],
 [0, 0, 0, 0, 0, 0],
 [1, 0, 1, 2, 3, 4],
 [2, 0, 2, 4, 1, 3],
 [3, 0, 3, 1, 4, 2],
 [4, 0, 4, 3, 2, 1]]
```

Similarly, here are SymPy commands to produce a multiplication table for $\mathbb{Z}/6\mathbb{Z}$:

```
>>> L = ["*"]+range(6)
>>> L = L+[i]+[(i*x)%6 for x in range(6)] for i in range(6)]
>>> L
[[*, 0, 1, 2, 3, 4, 5],
 [0, 0, 0, 0, 0, 0, 0],
 [1, 0, 1, 2, 3, 4, 5],
 [2, 0, 2, 4, 0, 2, 4],
 [3, 0, 3, 0, 3, 0, 3],
 [4, 0, 4, 2, 0, 4, 2],
 [5, 0, 5, 4, 3, 2, 1]]
```

You see the difference? The “lower right-hand corner” of the $5 \times 5$ table is a Latin square. The “lower right-hand corner” of the $6 \times 6$ table is not.

If $E$ and $F$ are fields and
(a) as sets, $F \subseteq E$,
(b) the field operations for $F$ are the restrictions of the field operations for $E$,
then $F$ is called a subfield of $E$ and $E$ is called an extension field of $F$, written $E/F$.

It is a theorem (proven in most any textbook in abstract algebra) that if $F$ is a finite field then $F$ must have $p^k$ elements, where $p$ is a prime and $k$ is an integer. If $k = 1$ then $F$ must be isomorphic to $GF(p)$. (Two fields are “isomorphic” if there is a bijective map from one to the other which preserves the addition, multiplication operations.) The fields with $k > 1$ are so-called extension fields and will not arise in these lectures.

8 The Hill cipher

The Hill cipher, invented by Lester S. Hill in 1920’s, it was the first polygraphic cipher in which it was practical to operate on more than three symbols at once. However, there are no public records of it being used in practice, to my knowledge.

However, there is a fascinating story of how Hill and his colleague Wisner and Hunter College filed a patent for a telegraphic device encryption and error-correction device which was roughly based on ideas arising form the Hill cipher. See Christensen, Joyner and Torres [CJT] for more details.

The following discussion assumes an elementary knowledge of matrices. First, each letter is first encoded as a number. We assume here that $A \leftrightarrow 0, B \leftrightarrow 1, \ldots, Z \leftrightarrow 25$, as in Figure 2. We denote the integers $\{0, 1, \ldots, 25\}$ by $\mathbb{Z}/26\mathbb{Z}$. This is closed under addition and multiplication and satisfies the associative and distributive properties, as the integers $\mathbb{Z}$ do.

Suppose your message $m$ consists of $n$ capital letters, with no spaces. This may be regarded an $n$-tuple $M$ of elements of $\mathbb{Z}/26\mathbb{Z}$. A key in the Hill cipher is a $k \times k$ matrix $K$, all of whose entries are in $\mathbb{Z}/26\mathbb{Z}$, such that the matrix $K$ is invertible (ie, that the linear transformation $K : (\mathbb{Z}/26\mathbb{Z})^k \rightarrow (\mathbb{Z}/26\mathbb{Z})^k$ is one-to-one).

The message $m$, a plaintext string over an alphabet of order 26, is rewritten as a sequence of column vectors $p \in (\mathbb{Z}/26\mathbb{Z})^k$ using the correspondence in Figure 2 above. A matrix $K \in GL(k, \mathbb{Z}/26\mathbb{Z})$ is chosen to be the key matrix, where both $k$ and $K$ are kept a secret. The encryption is performed by computing

\[
5\text{Here } GL(m, R) \text{ denotes the set of invertible transformations from } R^m \rightarrow R^m, \text{ regarded as } m \times m \text{ matrices.}
\]
\[ c = Kp, \quad (3) \]

and rewriting (using Figure 2) the resulting matrix as a string over the same alphabet. Decryption is performed similarly by computing

\[ p = K^{-1}c. \quad (4) \]

8.1 The 2 × 2 case

Suppose that the plaintext is broken into blocks of size 2 as follows:

\[ p = ((p_0, p_1), (p_2, p_3), \ldots, (p_{2\ell}, p_{2\ell+1})), \]

where each \( p_i \in \mathbb{Z}/26\mathbb{Z} \). Suppose \( K \) is a 2 × 2 matrix of the form

\[ K = \begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix}. \]

Write the ciphertext as follows:

\[ c = ((c_0, c_1), (c_2, c_3), \ldots, (c_{2\ell}, c_{2\ell+1})), \]

where each \( c_i \in \mathbb{Z}/26\mathbb{Z} \) is determined by

\[
\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \end{pmatrix}, \quad \begin{pmatrix} c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix} \begin{pmatrix} p_2 \\ p_3 \end{pmatrix}, \ldots.
\]

Here is a decryption algorithm, assuming we know \( c_0, c_1, c_2, c_3 \) and \( p_0, p_1, p_2, p_3 \). The first two matrix equations above can be rewritten as

\begin{align*}
p_0 \cdot k_{11} + p_1 \cdot k_{12} + 0 \cdot k_{21} + 0 \cdot k_{22} &= c_0, \\
0 \cdot k_{11} + 0 \cdot k_{12} + p_0 \cdot k_{21} + p_1 \cdot k_{22} &= c_1, \\
p_2 \cdot k_{11} + p_3 \cdot k_{12} + 0 \cdot k_{21} + 0 \cdot k_{22} &= c_2, \\
0 \cdot k_{11} + 0 \cdot k_{12} + p_2 \cdot k_{21} + p_3 \cdot k_{22} &= c_3,
\end{align*}

in other words, as
\[
\begin{pmatrix}
p_0 & p_1 & 0 & 0 \\
0 & 0 & p_0 & p_1 \\
p_2 & p_3 & 0 & 0 \\
0 & 0 & p_2 & p_3
\end{pmatrix}
\begin{pmatrix}
k_{11} \\
k_{12} \\
k_{21} \\
k_{22}
\end{pmatrix}
= 
\begin{pmatrix}
c_0 \\
c_1 \\
c_2 \\
c_3
\end{pmatrix}.
\]

This is invertible if and only if
\[
\begin{pmatrix}
p_0 & p_1 \\
p_2 & p_3
\end{pmatrix}
\]
is invertible if and only if \(p_0p_3 \not\equiv p_1p_2 \pmod{26}\).

**Example 20.** Suppose \(p_0 = 1, p_1 = 2, p_2 = 11, p_3 = 13\) and we know \(c_0 = 7, c_1 = 3, c_2 = 24, c_3 = 3\) and that the key length is 2. We can find \(K\) using matrix theory.

Here’s an example computation using **Sage**:

```
sage: ZZ26 = IntegerModRing(26)
sage: K = matrix(ZZ26, \[
[[1, 3], 
[5, 12]]\])
sage: det(K)
23
sage: Kˆ(-1)
\[
\begin{pmatrix}
22 & 1 \\
19 & 17
\end{pmatrix}
\]
sage: m1 = vector(ZZ26, \[1, 2\]); m1
\(1, 2\)
sage: m2 = vector(ZZ26, \[11, 13\]); m2
\(11, 13\)
sage: P0 = matrix(ZZ26, \[
[[1, 2], [11, 13]]\])
sage: det(P0)
17
sage: c1 = K*m1; c1
\(7, 3\)
sage: c2 = K*m2; c2
\(24, 3\)
sage: P1 = matrix(ZZ26, \[
[[1, 2, 0, 0], [0, 0, 1, 2], [11, 13, 0, 0], [0, 0, 1, 2]]\])
sage: P1
\[
\begin{pmatrix}
1 & 2 & 0 & 0 \\
0 & 0 & 1 & 2
\end{pmatrix}
\]
```

52
We see that the entries in the key $K$ are determined by knowing only $c_0, c_1, c_2, c_3$ and $p_0, p_1, p_2, p_3$.

### 8.2 The 3 × 3 case

Suppose that the plaintext is broken into blocks of size 3 as follows:

$$p = ((p_0, p_1, p_3), (p_4, p_5, p_6), \ldots, (p_3\ell, p_3\ell+1, p_3\ell+2)),$$

where each $p_i \in \mathbb{Z}/26\mathbb{Z}$. Suppose $K$ is a 3 × 3 matrix of the form

$$K = \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix}.$$

Write the ciphertext as follows:

$$c = ((c_0, c_1, c_2), (c_3, c_4, c_5), \ldots, (c_3\ell, c_3\ell+1, c_3\ell+2)),$$

where each $c_i \in \mathbb{Z}/26\mathbb{Z}$ is determined by

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix}, \ldots.$$

As in the 2 × 2 case, we can convert this into a system of linear equations of the form

\[
\begin{bmatrix}
11 & 13 & 0 & 0 \\
0 & 0 & 11 & 13
\end{bmatrix}
\]

```sage
sage: det(P1)
sage: C = vector(ZZ26, [7, 3, 24, 3])
sage: P1^(-1)*C
```

(1, 3, 5, 12)
We abbreviate this as $P \vec{k} = \vec{c}$. In the $2 \times 2$ example above, it turned out that $P$ was a $4 \times 4$ invertible matrix. In the next example, we see this is not necessary.

**Example 21.** Suppose we know

\[
p_0 = 11, p_1 = 12, p_2 = 13, p_3 = 21, p_4 = 22, p_5 = 23, p_6 = 7, p_7 = 8, p_8 = 9,
\]

and we know

\[
c_0 = 22, c_1 = 0, c_2 = 17, c_3 = 4, c_4 = 20, c_5 = 5, c_6 = 24, c_7 = 18, c_8 = 1,
\]

and that the key length is 3. We have $P \vec{k} = \vec{c}$, where

\[
P = \begin{pmatrix}
11 & 12 & 13 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 11 & 12 & 13 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 11 & 12 & 13 & 0 \\
21 & 22 & 23 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 21 & 22 & 23 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 21 & 22 & 23 & 0 \\
7 & 8 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 7 & 8 & 9 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 7 & 8 & 9 & 0
\end{pmatrix}
\]

\[
\begin{pmatrix}
k_{11} \\
k_{12} \\
k_{13} \\
k_{21} \\
k_{22} \\
k_{23} \\
k_{31} \\
k_{32} \\
k_{33}
\end{pmatrix}
= \begin{pmatrix}
c_0 \\
c_1 \\
c_2 \\
c_3 \\
c_4 \\
c_5 \\
c_6 \\
c_7 \\
c_8
\end{pmatrix}.
\]

We can find $K$ using matrix theory.
Here's an example computation using Sage:
This yields the congruences:

\[2k_{32} + 4k_{33} \equiv 24 \pmod{26}, k_{31} - k_{33} \equiv 5 \pmod{26},\]
\[
2k_{22} + 4k_{23} \equiv 8 \pmod{26}, k_{21} - k_{23} \equiv 24 \pmod{26}, \\
2k_{12} + 4k_{13} \equiv 16 \pmod{26}, k_{11} - k_{13} \equiv 24 \pmod{26}.
\]

One solution is
\[
K = \begin{pmatrix}
25 & 19 & 1 \\
25 & 2 & 1 \\
6 & 10 & 1
\end{pmatrix},
\]
which does indeed satisfy \(c_1 = Km_1, c_2 = Km_2, c_3 = Km_3\). Another solution is
\[
K = \begin{pmatrix}
25 & 6 & 1 \\
25 & 2 & 1 \\
6 & 10 & 1
\end{pmatrix},
\]
which also satisfies \(c_1 = Km_1, c_2 = Km_2, c_3 = Km_3\). Finally, the \(K\) given in the Sage code above (used to compute \(C_1, c_2, c_3\)) satifies the system of congruences above and, of course, also \(c_1 = Km_1, c_2 = Km_2, c_3 = Km_3\).

In conclusion, we see that the entries in the key \(K\) are determined (but not uniquely) by knowing only \(c_0, c_1, \ldots, c_8\) and \(p_0, p_1, \ldots, p_3\).

### 8.3 Matrix inverses \(\pmod{m}\)

Clearly, we must learn how to invert matrices \(\pmod{m}\). Here are a few methods to find the matrix inverse of \(K\) \(\pmod{m}\).

- **Method 1**: Use row-reduction \(\pmod{m}\) applied to the block matrix \((K, I_n)\)
- **Method 2**: Find the order of \(K\), call it \(d\). Compute \(K^{d-1} \pmod{m}\), using repeated squaring.
- **Method 3**:
  - Compute \(\text{adj}(K) = \text{cof}(K)^t\), the adjoint matrix of \(K\).
  - Compute \(r = 1/det(K) \pmod{m}\).
  - Compute \(K^{-1} = r \cdot \text{adj}(K) \pmod{m}\).

**Example 22.** Here’s an example using SymPy:
Example 23. The example of \( \begin{pmatrix} 5 & 2 \\ 7 & 3 \end{pmatrix} \) (mod 35) is not so easy to compute using row-reduction by hand (try it!). However, it is easy to compute using the adjoint formula and, of course, Sage can do it.

So. \( K^{-1} = \begin{pmatrix} 3 & 33 \\ 28 & 5 \end{pmatrix} \).

8.4 Enciphering and deciphering

The enciphering algorithm uses a key \( K \), which is a \( k \times k \) matrix of integers which is invertible (mod 26).

ALGORITHM:

\[ \text{INPUT:} \]
- key - a \( k \times k \) invertible matrix \( K \), all of whose entries are in \( \mathbb{Z}_{26} \)
- \( m \) - string of \( n \) upper-case letters (the "plaintext" message)
  (Note: Sage assumes that \( n \) is a multiple of \( k \).)

\[ \text{OUTPUT:} \]
- \( c \) - string of upper-case letters (the "ciphertext" message)
Identify the alphabet A, ..., Z with the integers 0, ..., 25.

Step 1: Compute from the string m a list L of corresponding integers. Let n = len(L).

Step 2: Break the list L up into t = ceiling(n/k) sublists L_1, ..., L_t of size k (where the last list might be "padded" by 0’s to ensure it is size k).

Step 3: Compute new list C_1, ..., C_t given by C[i] = K*L_i (arithmetic is done mod 26), for each i.

Step 4: Concatenate these into a list C = C_1 + ... + C_t.

Step 5: Compute from C a string c of corresponding letters. This has length k*t.

The deciphering algorithm is basically the same, except that it uses the inverse matrix $K^{-1}$ (mod 26).

**Example 24.** Here is an example. We will encrypt “Go Navy! Beat Army!” using Sage’s Hill cipher implementation.

```
sage: AS = AlphabeticStrings()
sage: HC = HillCryptosystem(AS, 3)
sage: Z26 = IntegerModRing(26)
sage: K = matrix(Z26, [[1,0,1], [0,1,1], [2,2,3]])
sage: det(K)
25
sage: m = "GONAVYBEATARMYX"
sage: m = HC.encoding(m)
sage: ct = HC.enciphering(K, m)
sage: ct
'GOHWRPBEFBISGSB'
sage: HC.deciphering(K, c)
'GONAVYBEATARMYX'
# You can always have \sage pick the key for you:
sage: K = HC.random_key(); K
[ 0  3 25]
[ 0 13  2]
[19  8  8]
sage: det(K)
```

58
Let’s break this down by steps, as in the algorithm presented above.
First, we see what the corresponding numbers are in the plaintext “GO-NAVYBEATARMYX” (we added an “X” at the end to make the length of the message divisible by 3).

```
sage: AS = AlphabeticStrings()
sage: HC = HillCryptosystem(AS, 3)
sage: Z26 = IntegerModRing(26)
sage: K = matrix(Z26, [[1, 0, 1], [0, 1, 1], [2, 2, 3]])
sage: m = "GONAVYBEATARMYX"
sage: pt = HC.encoding(m)
sage: ct = HC.enciphering(K, pt); ct
GOHWRPBEFBISGSB
sage: [A.index(x) for x in m]
[6, 14, 13, 0, 21, 24, 1, 4, 0, 19, 0, 17, 12, 24, 23]
```

Next, we create vectors corresponding to each block of size 3 and encrypt them.

```
sage: v1 = vector(Z26, [6, 14, 13])
sage: c1 = K*v1; c1
(19, 1, 1)
sage: v2 = vector(Z26, [0, 21, 24])
sage: c2 = K*v2; c2
(24, 19, 10)
sage: v3 = vector(Z26, [1, 4, 0])
sage: c3 = K*v3; c3
(1, 4, 10)
sage: v4 = vector(Z26, [19, 0, 17])
sage: c4 = K*v4; c4
(10, 17, 11)
sage: v5 = vector(Z26, [12, 24, 23])
```

59
Next, we put these numbers into a list and find the corresponding letters:

```
sage: ct0 = [19, 1, 24, 19, 10, 1, 4, 10, 10, 17, 11, 9, 21, 11]
sage: [A[i] for i in ct0]
```

Something is wrong - this doesn’t look like “GOHWRPBEFBISGSB” at all! The problem is that the Sage implementation uses matrix multiplication on the left, not on the right as we are used to in the United Stated (and many other parts of the world). Therefore, we must replace $K$ by its transpose, $K^t$.

```
sage: Kt = K.transpose()
sage: Kt
[1 0 2]
[0 1 2]
[1 1 3]
sage: c1t = Kt*v1; c1t
(6, 14, 7)
sage: c2t = Kt*v2; c2t
(22, 17, 15)
sage: c3t = Kt*v3; c3t
(1, 4, 5)
sage: c4t = Kt*v4; c4t
(1, 8, 18)
sage: c5t = Kt*v5; c5t
(6, 18, 1)
```

This does indeed agree with the above list, [6, 14, 13, 0, 21, 24, 1, 4, 0, 19, 0, 17, 12, 24, 23], of numbers corresponding to the letters in the cipher text.
Example 25. Next, let’s try using SymPy to work through an example.

```python
>>> pt = "meet me on monday"
>>> key = Matrix( ((1,2), (3,5)) )
>>> encipher_hill(pt, key)
UEQDUEODOCTCWQ
>>> pt = "meet me on tuesday"
>>> encipher_hill(pt, key)
UEQDUEODHBOYDJYU
```

Using the plaintext “Meet me on Monday” (which is translated into “MEET-MEONMONDAY”) and a block size of $k = 2$, SymPy gives the ciphertext “UEQDUEODOCTCWQ.” On the other hand, using the similar plaintext “Meet me on Tuesday” (which is translated into “MEETMEONTUESDAY”) and a block size of $k = 2$, SymPy gives the similar ciphertext “UEQDUEOD-HBOYDJYU.”

We can also decrypt using SymPy:

```python
>>> ct = "UEQDUEODOCTCWQ"
>>> key = Matrix( ((1,2), (3,5)) )
>>> decipher_hill(ct, key)
MEETMEONMONDAY
>>> ct = "UEQDUEODHBOYDJYU"
>>> decipher_hill(ct, key)
MEETMEONTUESDAYA
```

In each example, we have recovered the plaintext, except that “Meet me on Tuesday” got padded by an extra character (“A”).

Note: When you use SymPy, you don’t need to worry about using the transpose matrix, as we did with Sage.

Example 26. Let us try another example, this time using Sage.
We see the block size is 2. Again, let's encrypt “Go Navy! Beat Army!” First, we use Sage's implementation of Hill's cipher.

We have the corresponding numbers, so let's perform the block encryption “by hand”:
We got the ciphertext returned by Sage, as expected.

A few words regarding Sage’s syntax. We see that to decipher using Sage’s implementation, we can’t use \( (Kt)^{-1} \) of \( Kt \) with the deciphering method, but we can use \( K \) with the deciphering method, or \( K^{-1} \) with the enciphering method:

```
sage: HC.deciphering(Kt^(-1), ct)
YANANKHIQVCBUM
sage: HC.deciphering(Kt, ct)
GANABOLGKTYROW
sage: HC.deciphering(K, ct)
GONAVYBEATARMY
sage: HC.enciphering(K^(-1), ct)
GONAVYBEATARMY
```

Security concerns: This cipher is linear, so can be broken by a known plaintext attack. To determine the key from known plaintext, you set up the corresponding equations \( c = Kp \) and solve for \( K \). It is also susceptible to a known-ciphertext attack. You set up the equations \( p = K^{-1}c \) and adaptively try various keys \( K \) for which \( p \) matches some dictionary entries.
9 PK Cryptosystems

This section discusses some commonly used public key (PK) cryptosystems.

9.1 RSA

RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described the algorithm in 1977. In 1997, the work of Clifford Cocks, a mathematician at GCHQ, was declassified. He had developed an equivalent cryptosystem in 1973.

9.1.1 Set up and examples

Alice wants to talk to Bob.

Bob secretly chooses two large distinct primes \( p, q \). He computes \( n = pq \) and sends that to Alice. The length of \( n \) in bits is the “key length.”

Next, Bob computes \( \phi(n) = (p - 1)(q - 1) \), where \( \phi \) is Euler’s totient function, and chooses an integer \( e \) such that \( 1 < e < \phi(n) \) and greatest common divisor \( \gcd(e, \phi(n)) = 1 \). Bob sends \( e \) to Alice - this is the “public key exponent.”

The public key is the pair \( (n, e) \).

Finally, Bob determine \( d \), the multiplicative inverse of \( e \pmod{\phi(n)} \). We call \( d \) the private key exponent or private key.

Encryption: Alice wishes to send a message \( M \) to Bob. She first turns \( M \) into an integer (or a sequence of such integers) \( m \), such that \( 0 \leq m < n \) by using an agreed-upon protocol. She then computes the ciphertext \( c \) corresponding to \( m \) by

\[
c \equiv m^e \pmod{n}.
\] (5)

Decryption: Bob can recover \( m \) from \( c \) by using the private key exponent \( d \) via computing

\[
m \equiv c^d \pmod{n}.
\] (6)

Note that the private key exponent \( d \) can be computed using the extended Euclidean algorithm, by either

\[6\]Sometimes - with some slight inaccuracy - we call \( (n, d) \) the private key.
(a) computing $x$ and $y$ such that $ex + \phi(n)y = 1$ and taking $d = x \pmod{\phi(n)}$, or

(b) using the fact that $e^{\phi(\phi(n))} \equiv 1 \pmod{\phi(n)}$ (by Lemma 11) which implies $e^{-1} \equiv e^{\phi(\phi(n)) - 1} \pmod{\phi(n)}$ (see equation (2)).

However, the first method is usually preferable since $\phi(\phi(n))$ requires knowing the prime factorizations of $p - 1$ and $q - 1$.

Here is a Sage example.

**Example 27.** First, we pick our primes.

```sage
sage: p = next_prime(1000) ## secret
sage: q = next_prime(1010) ## secret
sage: n = p*q ## public
sage: n; p; q
1022117
1009
1013
```

Next, we compute the public key.

```sage
sage: k = euler_phi(n) ## = (p-1)(q-1),
sage: e = 123451 ## public key exponent
sage: k; xgcd(k, e)
1020096
(1, -36308, 300019)
```

Since $p, q$ are unknown, we assume (with no evidence otherwise) that $\phi(n)$ is also unknown, or at least impractical to compute. Here $\phi$ is the Euler $\phi$-function in (7). Now, we compute our private key.
Finally, we encrypt, then decrypt a message.

```
sage: m = randint(100, k); m
19861
sage: c = m^e%n; c  ## slow method
482896
sage: power_mod(m,e,n)  ## faster method
482896
sage: power_mod(c,d,n)
19861
```

Here is another example.

**Example 28.** *This time, we use some Sage code to do the computations.*

```
sage: p, q, e = 3, 5, 7
sage: n, e = rsa_public_key(p, q, e)
sage: n; e
15
7
sage: rsa_private_key(p, q, e)
(15, 7)
sage: pt = 12
sage: rsa_encrypt(p, q, e, pt)
66
```
This works, but it is a bit strange. We see $m = 12$ is not relatively prime to $n = 15$ and the private key exponent is $d = 7$, which is the same as the public key exponent $e = 7$.

Here is another Sage example.

**Example 29.**

```
sage: p,q,e = 7,11,17
sage: n, e = rsa_public_key(p,q,e); n; e
77
17
sage: rsa_private_key(p,q,e)
(77, 53)
sage: pt = 51
sage: rsa_encrypt(p,q,e,pt)
39
sage: ct = 39
sage: rsa_decrypt(p,q,e,ct)
51
```

Here is a SymPy example.

**Example 30.** First, we pick our primes and compute the public key.

```
>>> p,q,e = 3,5,7
>>> n, e = rsa_public_key(p,q,e)
>>> n
15
>>> e
7
```
Now, we compute our private key.

\[
\begin{verbatim}
>>> p, q, e = 3, 5, 7
>>> rsa_private_key(p, q, e)
(15, 7)
\end{verbatim}
\]

Finally, we encrypt, then decrypt a message.

\[
\begin{verbatim}
>>> p, q, e = 3, 5, 7
>>> pt = 12
>>> encipher_rsa(p, q, e, pt)
3
>>> ct = 3
>>> decipher_rsa(p, q, e, ct)
12
\end{verbatim}
\]

Security issues: (1) Of course, if integers were easy to factor quickly in general, then RSA would be insecure. However, it appears to be a strange fact of life that large “random” integers are typically very hard to factor into a product of primes.

(2) Since \( \phi(n) = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1 \), if we could somehow get only \( p + q \) (as opposed to \( p \) and \( q \) individually), that would be enough to break RSA. In other words, given that an integer \( n \) is a product of two primes, can you find the sum of the factors? This too seems to be very hard.

(3) When \( e = 3 \), it turns out that RSA is insecure in some cases. See Boneh’s survey [Bo].

(4) Under the “generalized Riemann hypothesis” (which is too technical to define here, but widely believed by mathematicians to be true), it has been shown by the computer scientist Gary Miller that the following computations are of equal computational complexity, up to a polynomial: (1) factoring \( n \) into \( p \) and \( q \), (2) computing \( d \), given \( n \) and \( e \). For details, see Miller [Mi].
9.2 RSA signature protocol

In this system, Alice wants to send a secret message \( m \) to Bob and to sign it.

1. Bob and Alice share a hash function, \( H \). We assume that this function takes values in \( \{1, 2, \ldots, N\} \), where \( N \) is smaller than \( N_{Alice} \) and \( N_{Bob} \) below.

2. Bob generates RSA keys: \((N_{Bob}, e_{Bob})\) is his public key, \((p_{Bob}, q_{Bob}, d_{Bob})\) is his private key, so \( d_{Bob}e_{Bob} \equiv 1 \pmod{\phi(N_{Bob})} \).

3. Alice generates RSA keys: \((N_{Alice}, e_{Alice})\) is her public key, \((p_{Alice}, q_{Alice}, d_{Alice})\) is her private key, so \( d_{Alice}e_{Alice} \equiv 1 \pmod{\phi(N_{Alice})} \).

4. Alice computes \( h = H(m) \) and \( h' = h^{d_{Alice}} \pmod{N_{Alice}} \).

5. Alice sends to Bob the encrypted message \( c = m^{e_{Bob}} \pmod{N_{Bob}} \) and the signature \( s = (h')^{e_{Bob}} \pmod{N_{Bob}} \).

6. Bob decrypts
\[
   c^{d_{Bob}} = m^{e_{Bob}d_{Bob}} \equiv m \pmod{N_{Bob}}
\]
and also computes
\[
   s^{d_{Bob}} = (h')^{e_{Bob}d_{Bob}} = h' \pmod{N_{Bob}},
\]
and then he computes
\[
   (h')^{e_{Alice}} = h^{e_{Alice}d_{Alice}} \equiv h \pmod{N_{Alice}}
\]

7. Bob checks that \( h \equiv H(m) \pmod{N_{Alice}} \) and, if true, approves the signature.

**Example 31.** Let \( p_{Alice} = 11003, q_{Alice} = 12007, \) so \( N_{Alice} = 132113021, \phi(N_{Alice}) = 132090012 \). Pick \( e_{Alice} = 123451 \), so that \( d_{Alice} = 75649675 \). Let \( p_{Bob} = 21001, q_{Bob} = 22003, \) so \( N_{Bob} = 462085003, \phi(N_{Bob}) = 462042000 \). Pick \( e_{Bob} = 234511 \), so that \( d_{Bob} = 47356591 \).

Let \( H(x) = x^5 \pmod{10000} \). If \( m = 93219 \) then \( H(m) = 2099, c = 92728104 \) and \( s = 402546440 \).
9.3 Kid RSA

Kid RSA is a version of RSA useful to teach grade school children since it does not involve exponentiation.

Alice wants to talk to Bob. Bob generates keys as follows. Key generation:

- Select positive integers $a, b, A, B$ at random.
- Compute $M = ab - 1$, $e = AM + a$, $d = BM + b$, $n = (ed - 1)/M$.
- The public key is $(n, e)$. Bob sends these to Alice. The private key is $d$, which Bob keeps secret.

**Example 32.** If $a = 3$, $b = 4$, $A = 5$, $B = 6$ then the public key is $(n, e) = (369, 58)$. The private key is $d = 70$.

Encryption: If $m$ is the plaintext message then the ciphertext is $c = me \pmod{n}$.

**Example 33.** If $n = 369$, $e = 58$ and the plaintext is $m = 200$ then the ciphertext is $c = 200 \cdot 58 \pmod{369} = 161$.

Decryption: If $c$ is the ciphertext message then the plaintext is $m = cd \pmod{n}$.

**Example 34.** If $n = 369$, $e = 58$ and the ciphertext is $c = 161$ then the plaintext is $m = 161 \cdot 70 \pmod{369} = 200$.

Here is **Sage** code to compute the public and private keys, with an example.

```
sage: a, b, A, B = 3, 4, 5, 6
sage: n, e = kid_rsa_public_key(a,b,A,B)
sage: n; e
369
58
sage: d = kid_rsa_private_key(a,b,A,B); d
70
```
Here is Sage code for encryption and decryption, with an example.

```sage
sage: pt = 200
sage: pk = kid_rsa_public_key(a,b,A,B)
sage: kid_rsa_encrypt(pt, pk)
161
sage: d = kid_rsa_private_key(a,b,A,B)
sage: pt = 200
sage: pk = kid_rsa_public_key(a,b,A,B)
sage: ct = kid_rsa_encrypt(pt, pk)
sage: kid_rsa_decrypt(ct, pk, d)
200
```

Here is the (very similar) SymPy code for Kid RSA. First, the keys.

```sympy
>>> a, b, A, B = 3, 4, 5, 6
>>> n, e = kid_rsa_public_key(a,b,A,B)
>>> n, e
(369, 58)
>>> d = kid_rsa_private_key(a,b,A,B); d
70
```

Next, the encryption and decryption.

```sympy
>>> pt = 200
>>> pk = kid_rsa_public_key(a,b,A,B)
>>> kid_rsa_encrypt(pt, pk)
161
>>> d = kid_rsa_private_key(a,b,A,B)
>>> pt = 200
>>> pk = kid_rsa_public_key(a,b,A,B)
>>> ct = encipher_kid_rsa(pt, pk)
>>> decipher_kid_rsa(ct, pk, d)
200
```
The security issues are discussed in the next subsection.

### 9.4 Breaking Kid RSA

How do we break Kid RSA? If the public key is \((n, e)\), we need to find a way to compute the private key from the public key more efficiently than “brute force” searching each value of \(d\) which works. It turns out, one can efficiently compute the private key without having any ciphertext.

Recall \(a, b, A, B\) were selected at random and then we computed \(M = ab - 1, e = AM + a, d = BM + b, n = (ed - 1)/M\). Note

\[
d e = (AM + a)(BM + b) = (ABM + aB + bA)M + ab = (ABM + aB + bA + 1)M + 1,
\]

therefore \(n = (ed - 1)/M = ABM + aB + bA + 1\), so

\[
de = nM + 1.
\]

This last equation allows us to compute \(d \pmod n\) by computing \(e^{-1} \pmod n\) using Bezout’s Lemma \(^{13}\).

**Example 35.** Here is a Sage example.

```sage
sage: n = 369; e = 58
sage: xgcd(n,e)
(1, -11, 70)
sage: d = xgcd(n,e)[2]; d
70
```

We see that if \((n, e) = (369, 58)\) then the Extended Euclidean algorithm computes the private key \(d = 70\).

Similar commands will work using SymPy’s \texttt{gcdex} command instead. (See Example \(^{14}\) for comparison.)
9.5 The discrete log problem

In some cases, keys must be shared in a secure manner, and to solve this problem, key exchange methods have been developed. Before discussing one of the most popular, the Diffie-Hellman key exchange, we discuss the mathematical problem that underlies it. Some of this discussion follows David Kohel’s notes [Ko].

Suppose \( m > 1 \) and \( a, b \) are integers relatively prime to \( m \). When

\[
b \equiv a^x \pmod{m},
\]

then we say that \( x \) is the discrete log of \( b \pmod{m} \): \( x = \log_a(b) \). Finding \( x \), given \( a, b \) (and of course \( m \)) is called the discrete log problem (DLP). There are no known “fast, efficient” ways to solve this problem in general.

The “brute force” algorithm for solving the discrete logarithm problem for \( \log_a(b) \) is to compute \( 1, a, a^2, \ldots \) until a match is found with \( b \). There are faster methods than this, but none that are polynomial time in the length of \( m \) (as a binary string).

9.5.1 Baby step, giant step

The simplest algorithm to use is called baby step, giant step. It has large memory requirements but is very easy to explain and computes discrete log of \( b \pmod{p} \) in time \( O(\sqrt{p}) \) (not including table look-ups).

Input: A cyclic group \( G \) of order \( m \), having a generator \( a \) and an element \( y \).

Output: A value \( x \) satisfying \( y = a^x \), so \( x = \log_a(y) \) in \( G \).

- If \( r = \lceil \sqrt{m} \rceil \), for all \( j \) where \( 0 \leq j < r \): Compute \( a^j \) and store the pair \( (j, a^j) \) in a table.

- for all \( k \) where \( 0 \leq k < r \): Compute \( z = y/a^{rk} \) and check to see if \( z \) is the second component of any pair in the table. If so, return \( kr + j \).

Example 36. Here is an example in Sage. We compute \( \log_2(17) \) in \( (\mathbb{Z}/29\mathbb{Z})^\times \).

```
sage: m = 29
sage: a = primitive_root(m); a
2
```
We see $x = 4 \cdot 5 + 1 = 21$.

Here is an example in Sage. We compute $\log_2(17)$ in $(\mathbb{Z}/29\mathbb{Z})^\times$. 

```python
sage: m = next_prime(1000); m
1009
sage: a = primitive_root(m); a
11
sage: y = power_mod(a, 567, m); y
830
sage: r = int(sqrt(m)); r
31
sage: S = [(j, power_mod(a, j, m)) for j in range(r)]; S
[(0, 1), (1, 11), (2, 121), (3, 322), (4, 515),
 (5, 620), (6, 766), (7, 354), (8, 867), (9, 456),
 (10, 980), (11, 690), (12, 527), (13, 752),
 (14, 200), (15, 182), (16, 993), (17, 833),
 (18, 82), (19, 902), (20, 841), (21, 170),
 (22, 861), (23, 390), (24, 254), (25, 776),
 (26, 464), (27, 59), (28, 649), (29, 76), (30, 836)]
sage: P = [(k, (y/power_mod(a, r*k, m))%m) for k in range(r+1)]
sage: P
[(0, 830), (1, 753), (2, 998), (3, 35), (4, 439),
 (5, 346), (6, 275), (7, 134), (8, 124), (9, 431),
 (10, 188), (11, 686), (12, 936), (13, 324),
 (14, 345), (15, 3), (16, 816), (17, 981),
 (18, 456), (19, 934), (20, 789), (21, 700),
 (22, 708), (23, 866), (24, 455), (25, 662),
 (26, 462), (27, 548), (28, 733), (29, 603),
 (30, 558), (31, 426)]
sage: for p in P:
```
if p[1] in S1:
    print p
....:
(18, 456)

We see \( x = k \cdot r + j = 18 \cdot 31 + 9 = 567. \)

### 9.6 Diffie-Hellman-Merkle key exchange

Whitfield Diffie and Martin Hellman proposed the following scheme for establishing a common key. The Diffie-Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

The scheme was first published by Diffie and Hellman in 1976, although it later emerged that it had been separately invented a few years earlier within GCHQ, the British signals intelligence agency, by Malcolm J. Williamson but was kept classified. In 2002, Hellman suggested the algorithm be called Diffie-Hellman-Merkle key exchange in recognition of Ralph Merkle’s contribution to the invention of public-key cryptography.

1. Alice and Bob decide on a large prime number \( p \) and a primitive element \( a \) of \( \mathbb{Z}/p\mathbb{Z} \), both of which can be made public.

2. Alice chooses a secret random \( x \) with \( \gcd(x, p - 1) = 1 \) and Bob chooses a secret random \( y \) with \( \gcd(y, p - 1) = 1 \).

3. Alice sends Bob \( a^x \pmod{p} \) and Bob sends Alice \( a^y \pmod{p} \).

4. Each is able to compute a session key \( K = a^{xy} = (a^x)^y = (a^y)^x \pmod{p} \).

**Example 37.** Here is a Sage example.

```plaintext
sage: p = 101; is_prime(p); a = primitive_root(p) # Alice and Bob
```
We see that the shared secret key is 66.

9.7 ElGamal encryption

Bob sends message $m$ to Alice as follows:

- Bob obtains Alice’s public key $(p, a, a^k \pmod{p})$.
- Bob represents the message $m$ as an integer $0 \leq m \leq p - 1$ (by choosing $p$ sufficiently large, this is always possible; alternatively, the message may be broken up into smaller pieces, each piece belonging to the range $0, 1, ..., p - 1$).
- Bob selects a random integer $\ell$, $1 \leq \ell \leq p - 2$.
- Bob computes $d \equiv a^\ell \pmod{p}$ and $e \equiv m \cdot (a^k)^\ell \pmod{p}$, $1 \leq d, e \leq p - 1$.
- Bob sends the “ciphertext” $c = (d, e)$ to Alice.

Alice decrypts $m$ as follows:
• Alice uses her private key to compute $d^{p-1-k} \pmod{p}$ and $d^{p-1-k} \equiv (a^{-k\ell}) \pmod{p}$.

• Alice computes $m \equiv (d^{p-1-k}) \cdot e \pmod{p}$.

**Example 38.** Using the numerical position in the alphabet, “A” is the $1^{st}$ letter, which corresponds numerically to 01, “B” to 02, ..., “Z” to 26. A word corresponds to the associated string of numbers. Note, for example, “ab” is 0102 = 102 but “ob” is 1502.

Pick $p = 150001$, $a = 7$. Pick $k = 113$, so that $a^k = 7^{113} \equiv 66436 \pmod{p}$. Alice’s public key is $(150001, 7, 66436)$. Alice’s private key is 113. Suppose Bob is Alice’s son and wants to send the message “Hi Mom”. He converts this to $m = 080913 = 809$ and $m = 131513$. Bob picks $\ell = 1000$ and computes $7^\ell \equiv 90429 \pmod{150001}$, so $d = 90429$. The encryption is $809 \cdot (7^{113})^{1000} \equiv 15061 = e \pmod{150001}$ and $131513 \cdot (7^{113})^{1000} \equiv 57422 = e \pmod{150001}$, so $c = (90429, 15061)$ and $c = (90429, 57422)$ are sent to Alice.

Alice computes $d^{p-1-k} = 90427^{149887} \equiv 80802 \pmod{150001}$. Then $m \equiv 80802 \cdot e \pmod{150001}$ returns the original messages $m = 809$ and $m = 131513$. Since 809 has an odd number of digits, the first digit must have been a 0. Alice can see that 809 must mean that the first letter comes from the 8 = 08, the second letter comes from a 09. These spell out “hi”. For $m = 131513$, the third letter must come from the 13, the fourth letter must come from the 15, and the last must be a 13. These spell out “hi” and “mom”. Adding a space and proper capitalization, this is “Hi Mom”!

**Exercise 9.1.** (a) Let $p = 541$, $a = 2$. Pick $k = 113$, $\ell = 101$. Encrypt the messages $m = 200$ and $m = 201$ using ElGamal.

(a) Let $p = 541$, $a = 2$, $k = 101$. Decrypt the ElGamal ciphers $c = (54, 300)$ and $c = (54, 301)$.

## 10 Stream ciphers

Think of a (plaintext, binary) data stream as an infinite (or at least potentially infinite) sequence of 0s and 1s:

$$d = d_0d_1d_2 \ldots$$

A “cryptographic stream” is a “random” binary sequence of as the key,

$$k = k_0k_1k_2 \ldots$$
and the enciphering map is obtained by adding these together bit-wise to create the ciphertext. In this section, we look at ways to construct such stream ciphers.

10.1 Binary stream ciphers

A cryptographic bit-stream is a cryptosystem derived using a sequence of binary digits for the key, where the plaintext is also a binary sequence and these streams are added (componentwise, \((\text{mod } 2)\)) to create the ciphertext.

Ideally, a cryptographic bit stream would have infinite length and could achieve complete randomness. The reality of practical application and construction techniques necessitates the use of only finite sequences. Since finite sequences can never be truly random, there are certain properties singled out that are associated with randomness. Golomb states these properties (roughly speaking) as:

1. The number of 1’s is approximately equal to the number of 0’s.
2. The runs of consecutive 1’s or 0’s frequently occur with short runs more frequent than long runs.
3. The sequence possesses an auto-correlation function, which is peaked in the middle and tapering of rapidly at the ends.

More precise version will be given later in Definition 46.

Example 39. Following Brock [Br], we provide a simplified example of how a message might be encrypted for secrecy and then decrypted by the authorized receiver so that the information can be read. In order for Alice and Bob to send messages to each other using computers, they must convert the English syntax of the message into a binary form. For our purposes we will allow the following table to define our correspondence between characters and binary 4-tuples:

- SPACE = 0000
- L = 1111
- A = 0001
- ! = 1000

78
Suppose Alice wanted to send the message \( M = \text{"BEAT ARMY!"} \) to Bob, but she wanted to keep it secret. By converting \( \text{"BEAT ARMY!"} \) from English to binary, we have a plaintext string, \( pt \), of binary bits:

\[
pt = 0010001100010110000000010101010001111000.
\]

In order to generate a stream cipher that only Alice and Bob know, to be used as the key, some method of secret key exchange is required. For example, each person can generate a pseudo-random sequence of sufficient length. Alice will send her sequence to Bob and Bob will send his sequence to Alice. They will then add their sequences together bit-wise to produce a common stream cipher. (It will be known to only the two of them unless someone has intercepted both of their communications.) For our purposes, suppose the resultant sequence from the two individual sequences is

\[
k = 1101011001000111101011001000111101011001
\]

Alice now has a message in binary format and a cipher to encode the message. Adding the cipher to the message, Alice gets the encrypted message or ciphertext, \( ct \):

\[
\begin{align*}
0010001100010110000000010101010001111000 + 1101011001000111101011001000111101011001 & = 1111010101010100011011011101101100100001
\end{align*}
\]
If a third party, Charlie, were to intercept the ciphertext and tried to read it, knowing that the computers used the above table to talk to one another, he would decrypt the ciphertext as “LRRA?.;BA”. Notice that both ‘A’s in the original message where changed to two different letters (‘R’ the first time and ‘.’ the second time) and that ‘E’ and ‘A’ both are changed to ‘R’ and ‘T’ and ‘!’ are both changed to ‘A’. This helps prevent a cryptanalyst from breaking the code by mapping each character to something different everytime. The mapping appears random since the stream cipher used was a pseudo-random sequence. When Bob receives the ciphertext, however, he is able to decrypt it since he has the stream cipher that he and Alice created earlier. By adding the stream cipher to the ciphertext, he will uncover the original message:

\[ ct + k = pt. \] 

(7)

Now Bob can use the table to decode the message from binary into English and receives “BEAT ARMY!” from Alice.

10.2 Background on solving recurrence equations

Solving linear homogeneous recurrence equations with constant coefficients

A linear homogeneous recurrence relation with constant coefficients is an equation of the form:

\[ a_n = k_1a_{n-1} + k_2a_{n-2} + \cdots + k_La_{n-L} \] 

(8)

where the \( L \) coefficients \( k_i \) (for all \( i \)) are constants. The integer \( L \) is called the order or length and the numbers \( a_0, \ldots, a_{d-1} \) form the initial values. More precisely, this is an infinite list of simultaneous linear equations, one for each \( n > d - 1 \). A sequence which satisfies a relation of this form is called a linear recurrence sequence (LRS). There are \( L \) degrees of freedom for LRS, i.e., the initial values \( a_0, \ldots, a_{L-1} \) can be taken to be any values but then the linear recurrence determines the sequence uniquely.

The same coefficients yield the characteristic polynomial (also auxiliary polynomial or a connection polynomial)

\[ p(t) = t^L - k_1t^{L-1} - k_2t^{L-2} - \cdots - k_L \] 

(9)

whose \( L \) roots play a crucial role in finding and understanding the sequences satisfying the recurrence. If the roots \( r_1, r_2, \ldots, r_L \) are all distinct, then the solution to the recurrence takes the form
\[ a_n = c_1 r_1^n + c_2 r_2^n + \cdots + c_L r_L^n, \quad (10) \]

where the coefficients \( c_i \) are determined in order to fit the initial conditions of the recurrence. When the same roots occur multiple times, the terms in this formula corresponding to the second and later occurrences of the same root are multiplied by increasing powers of \( n \). For instance, if the characteristic polynomial can be factored as \((x - r)^L\), with the same root \( r \) occurring \( L \) times, then the solution would take the form

\[ a_n = c_1 r^n + c_2 nr^n + \cdots + c_L n^{L-1} r^n. \]

Linear recursive sequences are precisely the sequences whose generating function is a rational function: the denominator is the polynomial obtained from the auxiliary polynomial by reversing the order of the coefficients, and the numerator is determined by the initial values of the sequence.

### 10.3 Matrix reformulation

Recall, the \textit{companion matrix} of the characteristic polynomial

\[ q(t) = t^d + q_1 t^{d-1} + q_2 t^{d-2} + \cdots + q_d \]

is the \( d \times d \) matrix

\[
M_q = \begin{bmatrix}
0 & 0 & \cdots & 0 & -q_d \\
1 & 0 & \cdots & 0 & -q_{d-1} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & -q_2 \\
0 & 0 & \cdots & 1 & -q_1 
\end{bmatrix},
\]

so \( \det(M_q - tI_d) = q(t) \). For example, if \( q(t) = x^4 + x^3 + 2x^2 + 3x + 4 \) then

\[
M_q = \begin{pmatrix}
0 & 0 & 0 & -4 \\
1 & 0 & 0 & -3 \\
0 & 1 & 0 & -2 \\
0 & 0 & 1 & -1 
\end{pmatrix}.
\]

Given a linearly recursive sequence, let \( C \) be
This is the transpose of the companion matrix of the characteristic/connection polynomial \( p(t) \) in (9). Since the determinant of a matrix and its transpose are the same, we have,

\[
\det(C - tI_L) = p(t),
\]

where \( p(t) \) is as in (9). In particular, the roots of \( p(t) \) are the eigenvalues of \( C \). Furthermore, the eigenvectors can be determined explicitly: if \( p(t) \) has distinct roots \( \lambda_1, \ldots, \lambda_L \) then the the eigenvector of \( C \) corresponding to \( \lambda_i \) is the vector

\[
\begin{pmatrix}
1 \\
\lambda_i \\
\vdots \\
\lambda_i^{L-1}
\end{pmatrix}.
\]

Observe that

\[
a_{n+L} = k_1a_{n+L-1} + k_2a_{n+L-2} + \cdots + k_La_n
\]

implies

\[
\begin{bmatrix}
a_{n+1} \\
\vdots \\
a_{n+L}
\end{bmatrix} = C \cdot \begin{bmatrix}
a_n \\
\vdots \\
a_{n+L-1}
\end{bmatrix},
\]

which, in turn, implies

\[
\begin{bmatrix}
a_n \\
\vdots \\
a_{n+L-1}
\end{bmatrix} = C^n \begin{bmatrix}
a_0 \\
\vdots \\
a_{L-1}
\end{bmatrix}
\]

\(82\)
Example 40. For the Fibonacci sequence over $\mathbb{Z}$,

$$f_n = f_{n-1} + f_{n-2}, f_0 = 0, f_1 = 1.$$  

We have $p(t) = t^2 - t - 1$, and

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

and the above identity becomes

$$\begin{bmatrix} f_{n-1} \\ f_n \end{bmatrix} = C \cdot \begin{bmatrix} f_{n-2} \\ f_{n-1} \end{bmatrix}$$

For example, taking $n = 2$ gives the initial condition

$$\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$$

Here is a Sage example.

```
Sage
sage: C = matrix(QQ, [[0,1],[1,1]])
sage: f0 = vector(QQ, [0,1])
sage: C*f0
(1, 1)
sage: C^2*f0
(1, 2)
sage: C^3*f0
(2, 3)
sage: C^4*f0
(3, 5)
sage: C^5*f0
(5, 8)
```

The SymPy commands are similar:

```
Sage
>>> C = Matrix(2, 2, [0, 1, 1, 1])
>>> f0 = Matrix(2, 1, [0, 1])
```
The Fibonacci sequence over $GF(2)$ is handled similarly:

```sage
sage: C = matrix(GF(2), [[0,1],[1,1]])
sage: f0 = vector(GF(2), [0,1])
sage: C*f0
(1, 1)
sage: C^2*f0
(1, 0)
sage: C^3*f0
(0, 1)
sage: C^4*f0
(1, 1)
```

Here is how to compute $C^n$ in general: Determine an eigenbasis $v_1, \ldots, v_L$ corresponding to eigenvalues $\lambda_1, \ldots, \lambda_L$. Then express the seed (the initial conditions of the recurrence sequence) as a linear combination of the eigenbasis vectors:

$$
\begin{bmatrix}
a_0 \\
\vdots \\
a_{L-1}
\end{bmatrix} = b_1 v_1 + \cdots + b_L v_L
$$
Then it conveniently works out that:

\[
\begin{bmatrix}
  a_{n+L-1} \\
  \vdots \\
  a_{n}
\end{bmatrix} = C^n \begin{bmatrix}
  a_{L-1} \\
  \vdots \\
  a_{0}
\end{bmatrix} = C^n (b_1 v_1 + \cdots + b_L v_L) = \lambda_1^n b_1 v_1 + \cdots + \lambda_L^n b_L v_L .
\]

(13)

**Example 41.** The explicit formula in the example of the Fibonacci sequence in \( \mathbb{Z} \) has become known as Binet’s formula:

\[
f_n = \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\sqrt{5}}
\]

(14)

where

\[
\varphi = \frac{1 + \sqrt{5}}{2} \approx 1.6180339887 \ldots
\]

is the golden ratio (sequence A001622 in the online OEIS, [http://oeis.org/](http://oeis.org/)), and

\[
\psi = \frac{1 - \sqrt{5}}{2} = 1 - \varphi = -\frac{1}{\varphi}.
\]

To see this, note that \( \varphi \) and \( \psi \) are both solutions of the equation \( x^2 - x - 1 = 0 \).

For the Fibonacci sequence in \( GF(2) \),

\[
\{f_n\} = \{0, 1, 1, 0, 1, 1, \ldots\},
\]

we also have

\[
f_n = c_1 r_1^n + c_2 r_2^n,
\]

where \( r_1 \) and \( r_2 \) are the two roots of \( x^2 - x - 1 = 0 \) in \( GF(4) \), and \( c_1, c_2 \) are constants. We may take

\[
GF(4) = \{0, 1, \alpha, \alpha + 1\},
\]

where \( \alpha \) is a root of \( x^2 - x - 1 = 0 \). Taking \( c_1 = c_2 = 1, r_1 = \alpha, r_2 = \alpha + 1 \).

**Example 42.** Compute, using Sage, the example:

\[
x_n = 4x_{n-1} - x_{n-2} - 6x_{n-3}, \quad x_0 = 0, \quad x_1 = 1, \quad x_2 = 2.
\]
\[ \{x_n\} = \{0, 1, 2, 7, 20, \ldots \}. \]

This can be written as a matrix equation as
\[
\begin{bmatrix}
  x_{n-2} \\
  x_{n-1} \\
  x_n
\end{bmatrix} = 
\begin{pmatrix}
  0 & 1 & 0 \\
  0 & 0 & 1 \\
  -6 & -1 & 4
\end{pmatrix}
\begin{bmatrix}
  x_{n-3} \\
  x_{n-2} \\
  x_{n-1}
\end{bmatrix}
\]

or as
\[
\begin{bmatrix}
  x_n \\
  x_{n-1} \\
  x_{n-2}
\end{bmatrix} = 
\begin{pmatrix}
  4 & -1 & -6 \\
  1 & 0 & 0 \\
  0 & 1 & 0
\end{pmatrix}
\begin{bmatrix}
  x_{n-1} \\
  x_{n-2} \\
  x_{n-3}
\end{bmatrix}.
\]

Let us use the former equation. To compute \[ \left( \begin{array}{ccc}
  0 & 1 & 0 \\
  0 & 0 & 1 \\
  -6 & -1 & 4
\end{array} \right)^n \], we compute the diagonalization of \[ \left( \begin{array}{ccc}
  0 & 1 & 0 \\
  0 & 0 & 1 \\
  -6 & -1 & 4
\end{array} \right) \]. The Sage code below shows that the eigenvalues are
\[ \lambda_1 = 3, \quad \lambda_1 = 2, \quad \lambda_1 = -1, \]
and corresponding eigenvectors are
\[ \vec{v}_1 = \begin{bmatrix} 1 \\ 3 \\ 9 \end{bmatrix}, \quad \vec{v}_2 = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}, \quad \vec{v}_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}. \]

If
\[ P = \begin{pmatrix}
  1 & 1 & 1 \\
  3 & 2 & -1 \\
  9 & 4 & 1
\end{pmatrix} \]
is the matrix of eigenvectors and
\[ D = \begin{pmatrix}
  3 & 0 & 0 \\
  0 & 2 & 0 \\
  0 & 0 & -1
\end{pmatrix} \]
the diagonal matrix of eigenvalues then
\[ A = PDP^{-1} \]

is the diagonalization. We compute arbitrary powers using \( A^n = PD^nP^{-1} \). This can be computed efficiently \( \pmod p \) using repeated squares.

Incidently, the eigenvalues of
\[
\begin{pmatrix}
4 & -1 & -6 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix}
\]
are
\[ \lambda_1 = 3, \quad \lambda_1 = 2, \quad \lambda_1 = -1, \]
and corresponding eigenvectors are
\[
\vec{v}_1 = \begin{bmatrix} 9 \\ 3 \\ 1 \end{bmatrix}, \quad \vec{v}_2 = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix}, \quad \vec{v}_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}.
\]

Therefore, a similar calculation works for the latter matrix reformulation above.

```
sage: A = matrix(QQ, 
               [[0,1,0],
                [0,0,1],
                [-6,-1,4]])
sage: A.eigenspaces_right()
[(3, Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix: 
  [1 3 9]),
 (2, Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix: 
  [1 2 4]),
 (-1, Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix: 
  [ 1 -1 1])]
sage: P = matrix(QQ, 
               [[1,1,1],
                [3,2,-1],
                [9,4,1]])
sage: D = P^(-1)*A*P; D
[ 3  0  0]
[ 0  2  0]
[ 0  0 -1]
```

This is the diagonalization of \( A \).

Here is the corresponding computation in SymPy:

```
>>> A = Matrix(3, 3, [0,1,0,0,0,1,-6,-1,4])
>>> print A
```

87
(Note that SymPy returns “right” eigenvectors by default.)

10.4 Linear feedback shift registers

The initial fill is the initial values of the state cells,

\[ s_0, s_1, s_2, \ldots, s_{L-1} \]

the initial contents of the \( L \) stages of the register. In general, the binary linear feedback shift register (LFSR) sequence is defined by the following recursion relation

\[ s_j = \sum_{i=1}^{L} k_i s_{j-i} \quad (\text{mod } 2), \]

for \( j \geq L \). The coefficients \( k_1, \ldots, k_L \) are fixed and form the key to the cryptographic bitstream.
The key may be represented as a vector
\[ k = [k_1, k_2, \ldots, k_L], \]
but is more often defined by a polynomial, known as the connection polynomial. Unfortunately, there is no universally used convention for this - it is either defined by (9) or by its reverse polynomial
\[ p(t) = 1 + k_1 t + k_2 t^2 + \cdots + k_L t^L. \] (15)
We shall use (9) unless stated otherwise.

We say a binary sequence \( a_0, a_1, \ldots \) is periodic with period \( P \) if
\[ a_i = a_{i+p}, \]
for all \( i \). By this definition, if \( \{a_n\}_{n \geq 0} \) is periodic with period \( P \) then it also has period \( 2P, 3P, \ldots \). Therefore, we often assume that \( P > 0 \) is chosen as small as possible.

**Example 43.** If you divide a \( k \) digit positive integer by the number with \( k \) 9s, the digits in the decimal representation of the resulting rational number is a repeating decimal. For example, \( 274/999 = 0.274274274274\ldots \).

We say a binary sequence \( a_0, a_1, \ldots \) is eventually periodic with period \( P \) if there is an \( i_0 > 0 \) such that \( a_i = a_{i+p}, \) for all \( i > i_0 \).

**Example 44.** The decimal expansion of the rational number obtained by dividing any positive integer by another is eventually repeating decimal. For example, \( 1/7 = 0.142857142857143 \ldots \).

**Question:** Why is this eventually periodic?

**Example 45.** If we are given the key as a vector \( c = [1, 0, 0, 1] \) and the initial fill as a vector \( s = [1, 1, 0, 1] \) in \( GF(2) \), we can create the sequence
\[ 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, \ldots. \]

The auto-correlation function of a periodic sequence \( X = \{x_0, x_1, \ldots\} \) (of real numbers) with period \( P \) is defined by
\[ AC(X, k) = (1/P) \sum_{i=1}^{P} x_i x_{i+k} \] (16)
By abuse of terminology, if $S = \{s_0, s_1, \ldots\}$ is a periodic binary sequence with period $P$ (in $GF(2)$) then the auto-correlation function is defined by

$$AC(S, k) = \frac{1}{P} \sum_{i=1}^{P} (-1)^{s_i}(-1)^{s_{i+k}} = \frac{1}{P} \sum_{i=1}^{P} (-1)^{s_i+s_{i+k}}$$

**Definition 46.** (Pseudo-random binary sequence) Let $A = \{a_0, a_1, \ldots\}$, be a periodic binary sequence with period $P$. We say this sequence is pseudo-random provided the following conditions hold (Golomb’s Principles):

1. **Balance:** $|\sum_{i=1}^{P} (-1)^{a_i}| \leq 1$

2. **Low Autocorrelation:**

$$AC(A, k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0, \end{cases}$$

where $\epsilon$ is “small.”

3. **Proportional Runs Property:** In each period, half the runs are length 1, one-fourth are length 2, one-eighth are length 3, etc. Moreover, there are as many runs of 1’s as there are of 0’s.

**Example 47.** Consider the sequence of period 7 defined by the values of the linear function $f : GF(2)^3 \rightarrow GF(2)$ given by $f_1(x_0, x_1, x_2) = x_1 + x_2$,

$$A = \{0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, \ldots\},$$

where $A_i = f(v_i)$ and $GF(2)^3 \setminus \{(0, 0, 0)\} = \{v_1, v_2, \ldots, v_7\}$. The following Sage code computes the values of the autocorrelation function:

```sage
def f(x): return (x[1]+x[2])%2
sage: F = GF(2)
sage: V = F^3
sage: flist = [f(x) for x in V if x<>V(0)]
sage: flist2 = 2*flist
sage: autocorr = [[j,sum([(-1)^x[i]+flist2[i+j] for i in range(7)])] for j in range(7)]
sage: autocorr
```

$\begin{array}{cccc}
| (0, 7), (1, 3), (2, -1), (3, -5), (4, -5), (5, -1), (6, 3) \\
\end{array}$
In other words,

\[
AC(A, k) = \begin{cases} 
1, & k = 0, \\
3/7, & k = 1, 6, \\
-1/7, & k = 2, 5, \\
-5/7, & k = 3, 4.
\end{cases}
\]

**Example 48.** Consider the sequence of period 7 defined by the values of the linear function \( f : GF(2)^3 \rightarrow GF(2) \) given by \( f_1(x_0, x_1, x_2) = x_0 + x_1 + x_2, \)

\[
A = \{1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, \ldots \}
\]

where \( A_i = f(v_i) \) and \( GF(2)^3 - \{(0, 0, 0)\} = \{v_1, v_2, \ldots, v_7\} \). The following Sage code computes the values of the autocorrelation function:

```
sage: f = lambda x: (x[0]+x[1]+x[2])%2
sage: flist = [f(x) for x in V if x<>V(0)]
sage: flist2 = 2*flist
sage: flist2
[1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1]
sage: autocorr = [[j,sum([-1]^(flist2[i]+flist2[i+j]) for i in range(7))] for j in range(7)]
sage: autocorr
[[0, 7], [1, -1], [2, -1], [3, -1], [4, -1], [5, -1], [6, -1]]
```

In other words,

\[
AC(A, k) = \begin{cases} 
1, & k = 0, \\
-1/7, & k \neq 0.
\end{cases}
\]

**Example 49.** Consider the sequence of period 7 defined by the values of the non-linear function \( f : GF(2)^3 \rightarrow GF(2) \) given by \( f_1(x_0, x_1, x_2) = x_1 + x_0x_2, \)

\[
A = \{0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, \ldots \}
\]

where \( A_i = f(v_i) \) and \( GF(2)^3 - \{(0, 0, 0)\} = \{v_1, v_2, \ldots, v_7\} \). The following Sage code computes the values of the autocorrelation function:

```
sage: f = lambda x: (x[0]*x[2]+x[1])%2
sage: flist = [f(x) for x in V if x<>V(0)]
sage: flist2 = 2*flist
sage: flist2
[1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1]
sage: autocorr = [[j,sum([-1]^(flist2[i]+flist2[i+j]) for i in range(7))] for j in range(7)]
sage: autocorr
[[0, 7], [1, -1], [2, -1], [3, -1], [4, -1], [5, -1], [6, -1]]
```
In other words,

\[ AC(A, k) = \begin{cases} 
1, & k = 0, \\
3/7, & k = 3, 4, \\
-1/7, & k = 1, 6, \\
-5/7, & k = 2, 5.
\end{cases} \]

### 10.5 Computations with LFSRs

We look at several examples:

```sage
sage: F = GF(2)
sage: o = F(0); l = F(1)
sage: key = [l,o,o,l]; fill = [l,l,o,l]; n = 20
sage: s = lfsr_sequence(key,fill,n); s
[1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
sage: massey(s)
x^4 + x + 1
```

Suppose \( \{a_n\} \) is a sequence in a finite field or even \( \mathbb{Q} \). If

\[ a_n = k_1a_{n-1} + ... + k_La_{n-L} \]

then the sequence \( a_0, ..., a_{L-1} \) is called the *initial fill*. The connection polynomial is

\[ p(x) = t^L - k_1t^{L-1} - ... - k_{L-1}t - k_L. \]

For the sequence above, it is \( p(t) = t^4 + t^3 + 1 \). As stated above, there is no universally accepted terminology. However, sometimes the reciprocal, or reverse, polynomial is the *feedback polynomial*. A polynomial \( b_nx^n + b_{n-1}x^{n-1} + ... + b_1x + b_0 \) is called *monic* if the leading coefficient is one, \( b_n = 1 \). The connection polynomial is monic.
Example 50. Consider the example,

\[ a_n = a_{n-1} + a_{n-4}, \]

and \( a_0 = 1, \ a_1 = 1, \ a_2 = 0, \ a_3 = 1. \) This gives

\[ a_0, a_1, \cdots = 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, \ldots. \]

Let us check this using Sage:

```
sage: F = GF(2)
sage: o = F(0); l = F(1)
sage: key = [l,o,o,l]; fill = [l,l,o,l]; n = 20
sage: s = lfsr_sequence(key,fill,n); s
[1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
sage: # Let's do this "by hand":
sage: a1 = l
sage: a2 = o
sage: a3 = o
sage: a4 = l
sage: s0 = fill
sage: lfsr(s0)
0
sage: s1 = [1,0,1,0]
sage: lfsr(s1)
1
sage: s2 = [0,1,0,1]
sage: lfsr(s2)
1
sage: s3 = [1,0,1,1]
sage: lfsr(s3)
0
sage: A = matrix(GF(2), [[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: s0 = vector(GF(2), [1, 1, 0, 1])
sage: s1 = A*s0; s1
(1, 0, 1, 0)
sage: s2 = A*s1; s2
(0, 1, 0, 1)
```

Now, suppose we know the \( \{a_n\} \)'s but not the key. Suppose we know

\[ a_n = c_1a_{n-1} + c_2a_{n-2} + c_3a_{n-3} + c_4a_{n-4}, \]

but not what the key \((c_1, c_2, c_3, c_4)\) is. Let's try to find the key by hand, solving these "state equations."
$c_1, c_2, c_3, c_4 = \text{var}("c_1, c_2, c_3, c_4")$

$sage: \text{solve}([c_1+2+3+0+c_4*1 == 0,}$
$c_1+c_2+0+c_3+1+c_4*0 == 1,$
$c_1+0+c_2+1+c_3+0+c_4*1 == 1,$
$c_1+c_2+0+c_3+1+c_4*1 == 0], [c_1, c_2, c_3, c_4])$

$[[c_1 == -1, c_2 == 2, c_3 == 2, c_4 == -1]]$

Note that mod 2, this is [1, 0, 1], the key. Another way:

$B = \text{matrix}(\text{GF}(2), [[1, 1, 0, 1, 0], [1, 0, 1, 0, 1], [0, 1, 0, 1, 1], [1, 0, 1, 0, 1]])$

$sage: B.\text{echelon\_form}()$

$[1 0 0 0 1]$
$[0 1 0 0 0]$
$[0 0 1 0 0]$
$[0 0 0 1 1]$

Again, the solution is [1, 0, 1], the key.

This cannot be done in SymPy since, as far as I can tell, at the present time, SymPy does not have row reduction over finite fields.

**Example 51.** Consider the example,

\[ x_n = 4x_{n-1} - x_{n-2} - 6x_{n-3}, \]

and \(x_0 = 0, \ x_1 = 1, \ x_2 = 2.\) This gives

\[ x_0, x_1, \cdots = 0, 1, 2, 7, 20, 61, 182, 547, \ldots . \]

Let us check this using Sage (the SymPy syntax is exactly the same):

$sage: a1 = -6$
$sage: a2 = -1$
$sage: a3 = 4$
$sage: rr = \text{lambda x: a1*x[0]+a2*x[1]+a3*x[2]}$
$sage: s0 = [0, 1, 2]$
$sage: rr(s0)
7$
$sage: s1 = [1, 2, 7]$
We can generate the sequence using the recurrence relation. We can also use the corresponding matrix to generate the terms.

For

\[ x_n = 4x_{n-1} - x_{n-2} - 6x_{n-3}, \]

and \( x_0 = 0, \ x_1 = 1, \ x_2 = 2, \) we have:

\[
\begin{align*}
\text{sage: } & A = \text{matrix}(\text{QQ}, \begin{bmatrix} 0, 1, 0 \end{bmatrix}, \begin{bmatrix} 0, 0, 1 \end{bmatrix}, \begin{bmatrix} -6, -1, 4 \end{bmatrix}) \\
& s0 = \text{vector}(\text{QQ}, \begin{bmatrix} 0, 1, 2 \end{bmatrix}) \\
& s1 = A \ast s0 \\
& s2 = A \ast s1 \\
& s3 = A \ast s2 \\
& s4 = A \ast s3 \\
& s5 = A \ast s4 \\
& s6 = A \ast s5 \\
& s1; s2; s3; s4; s5; s6
\end{align*}
\]

\( (1, 2, 7) \)
\( (2, 7, 20) \)
\( (7, 20, 61) \)
\( (20, 61, 182) \)
\( (61, 182, 547) \)
\( (182, 547, 1640) \)

For

\[ x_n = 3x_{n-1} + x_{n-2} - 3x_{n-3}, \]

and \( x_0 = 0, \ x_1 = 1, \ x_2 = 2, \) we have:

\[
\begin{align*}
\text{sage: } & A = \text{matrix}(\text{QQ}, \begin{bmatrix} 0, 1, 0 \end{bmatrix}, \begin{bmatrix} 0, 0, 1 \end{bmatrix}, \begin{bmatrix} -3, 1, 3 \end{bmatrix}) \\
& s0 = \text{vector}(\text{QQ}, \begin{bmatrix} 0, 1, 2 \end{bmatrix}) \\
& s1 = A \ast s0
\end{align*}
\]
For

\[ x_n = 7x_{n-2} + 6x_{n-3}, \]

and \( x_0 = 0, x_1 = 1, x_2 = 2, \) we have:

We see that several different matrices can be used to generate the terms.

To “solve” for \( x_n, \) we want to compute the diagonalization of one of these matrices, say

\[
A = \begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
-6 & -1 & 4
\end{pmatrix}.
\]

To find the diagonalization, we need the eigenvalues and eigenvectors. Sage can do this easily.
sage: A = matrix(QQ, [[0,1,0],[0,0,1],[-6,-1,4]])
sage: A.eigenspaces_right()
[(3, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[1 3 9]),
(2, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[1 2 4]),
(-1, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -1 1])
]
sage: P = matrix(QQ, [[1, 3, 9], [1, 2, 4], [1, -1, 1]]).transpose()
sage: D = matrix(QQ, [[3,0,0], [0,2,0], [0,0,-1]])
sage: P*D*P^(-1)
[ 0 1 0]
[ 0 0 1]
[-6 -1 4]

Now, let’s work over the finite field $GF(5)$. In this case, the recursion becomes

$$x_n = 4x_{n-1} + 4x_{n-2} + 4x_{n-3},$$

and the sequence becomes

$$x_0, x_1, \cdots = 0, 1, 2, 2, 0, 1, 2, 2, \ldots .$$

We check this using Sage:

```python
sage: A = matrix(GF(5), [[0,1,0],[0,0,1],[-3,1,3]])
sage: s0 = vector(GF(5), [0,1,2])
sage: s1 = A*s0
sage: s2 = A*s1
sage: s3 = A*s2
sage: s4 = A*s3
sage: print s0.list()+[s1[2]]+[s2[2]]+[s3[2]]+[s4[2]]
[0, 1, 2, 2, 0, 1, 2, 2]
sage: A = matrix(GF(5), [[0,1,0],[0,0,1],[-3,1,3]])
sage: s0 = vector(GF(5), [0,1,2])
sage: s1 = A*s0
sage: s2 = A*s1
sage: s3 = A*s2
sage: s4 = A*s3
sage: print s0.list()+[s1[2]]+[s2[2]]+[s3[2]]+[s4[2]]
```
Suppose we know the \( \{x_n\} \)’s but not the key. Suppose we know
\[
x_n = c_1 x_{n-1} + c_2 x_{n-2} + c_3 x_{n-3},
\]
but not what the key \((c_1, c_2, c_3)\) is. Let’s try to find the key by hand, solving these “state equations.” We find that
\[
\begin{align*}
2 &= 2 \cdot c_1 + 1 \cdot c_2 + 0 \cdot c_3, \\
0 &= 2 \cdot c_1 + 2 \cdot c_2 + 1 \cdot c_3, \\
1 &= 0 \cdot c_1 + 2 \cdot c_2 + 2 \cdot c_3, \\
2 &= 1 \cdot c_1 + 0 \cdot c_2 + 2 \cdot c_3, \\
\vdots
\end{align*}
\]
and so on. Taking, say, the last three equations above, we obtain a system of linear equations with augmented matrix
\[
A = \begin{pmatrix}
2 & 2 & 1 & 0 \\
0 & 2 & 2 & 1 \\
1 & 0 & 2 & 2
\end{pmatrix}
\]
with row-reduced echelon form
\[
\begin{pmatrix}
1 & 0 & 2 & 2 \\
0 & 1 & 1 & 3 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]
Therefore,
\[
c_1 = 2 + 3c_3, \quad c_2 = 3 + 4c_3.
\]
Let’s look at this case-by-case.

- \(c_3 = 0\). Then the key is \((c_1, c_2, c_3) = (2, 3, 0)\). The equation \(x_n = 2x_{n-1} + 3x_{n-2}\) gives
\[
x_0, x_1, \ldots = 0, 1, 2, 2, 0, 1, \ldots,
\]
agreeing with the original sequence, but with a different key than the one we started with.
• \( c_3 = 1 \). Then the key is \((c_1, c_2, c_3) = (0, 2, 1)\). The equation \(x_n = 2x_{n-2} + x_{n-3}\) gives

\[
x_0, x_1, \ldots = 0, 1, 2, 2, 0, 1, \ldots,
\]
agreeing with the original sequence, but with a different key than the one we started with.

• \( c_3 = 2 \). Then the key is \((c_1, c_2, c_3) = (3, 1, 2)\). The equation \(x_n = 3x_{n-1} + x_{n-2} + 2x_{n-3}\) gives

\[
x_0, x_1, \ldots = 0, 1, 2, 2, 0, 1, \ldots,
\]
agreeing with the original sequence, but with a different key than the one we started with.

• \( c_3 = 3 \). Then the key is \((c_1, c_2, c_3) = (1, 0, 3)\). The equation \(x_n = x_{n-1} + 3x_{n-3}\) gives

\[
x_0, x_1, \ldots = 0, 1, 2, 2, 0, 1, \ldots,
\]
agreeing with the original sequence, but with a different key than the one we started with.

• \( c_3 = 4 \). Then the key is \((c_1, c_2, c_3) = (4, 4, 4)\). The equation \(x_n = x_{n-1} + 3x_{n-3}\) gives

\[
x_0, x_1, \ldots = 0, 1, 2, 2, 0, 1, \ldots,
\]
agreeing with the original sequence, and with same key than we started with.

In any case, we can “break” this cipher very easily using linear algebra.

Here are some more Sage computations regarding this example. First, we show that, in this example, we can diagonalize the matrix over \( GF(5) \).

```sage
sage: A = matrix(GF(5), [[0,1,0],[0,0,1],[-6,-1,4]])
sage: A.eigenspaces_right()
[(4, User basis matrix:
```

99
This recursive equation can be written having a “shorter” key length (as we already saw in the above itemized list, case $c_3 = 0$):

Fact 1: If $r_1, \ldots, r_k$ are distinct roots of the connection polynomial of a LFSR $\{a_n\}$ then
\[ a_n = \alpha_1 r_1^n + \cdots + \alpha_k r_k^n \]

for some \( \alpha_1, \ldots, \alpha_k \) determined by the initial fill.

The generating function \( A(x) \) of a sequence \( \{a_n\} \) is

\[
A(x) = a_0 + a_1 x + \ldots = \sum_{n=0}^{\infty} a_n x^n.
\] (17)

**Fact 2:** A sequence \( \{a_n\} \) is eventually periodic if and only if \( A(x) \) is a rational function.

**Fact 3:** If \( \{a_n\} \) is a LFSR then \( A(x) = B(x)/C(x) \). If we put this quotient in least common terms and assume \( C(x) \) is monic then \( C(x) \) is the connection polynomial of \( \{a_n\} \).

**Example 52.**

\[
(1 + x + x^3)/(x^4 + x^3 + 1) = 1 + x + x^3 + x^4 + x^5 + x^7 + x^8 + x^9 + x^{11} + x^{12} + x^{13} + x^{15} + x^{16} + x^{17} + x^{19} + x^{20} + x^{21} + x^{23} + x^{24} + x^{25} + x^{27} + x^{28} + x^{29} + O(x^{30}).
\]

The coefficients are 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, \ldots.

**Example 53.** We return to the finite field \( GF(5) \) and the recursion

\[ x_n = 4x_{n-1} + 4x_{n-2} + 4x_{n-3}, \]

associated to the sequence

\[ x_0, x_1, \ldots = 0, 1, 2, 2, 0, 1, 2, 2, \ldots. \]

A key is \((c_1, c_2, c_3) = (2, 3, 0)\). We check Fact 2 above using **Sage**:

```
sage: LSR.<q> = LaurentSeriesRing(GF(5))
sage: f = q; g = 1+3*q+2*q^2; f/g
q + 2*q^2 + 2*q^3 + q^5 + 2*q^6 + 2*q^7 + q^9 + 2*q^10 + 2*q^11 + q^13 + 2*q^14 + 2*q^15 + q^17 + 2*q^18 + 2*q^19 + O(q^21)
```

Sage
10.6 Blum-Blum-Shub (BBS) stream cipher

The number theory behind the BBS cipher uses quadratic residues.

**Definition 54.** (Quadratic residue) Let $a > 0$ be an integer. We say $a$ is a quadratic residue (mod $m$) if the congruence $x^2 ≡ a$ (mod $m$) is solvable. The set of all quadratic residues $a$, $0 < a < m$, is denoted by $Q_m$.

Let $m$ be a prime number, $m > 2$. $Q_m$ is a subgroup of $(\mathbb{Z}/m\mathbb{Z})^\times = \{1, \ldots, m-1\}$ of index 2. In other words, $|Q_m| = (m-1)/2$.

**Definition 55.** (Special Blum Prime) Let $p$ be a prime number. We say $p$ is a special Blum prime if and only if $p ≡ 3$ (mod 4), $p = 2p_1 + 1$ and $p_1 = 2p_2 + 1$ with $p_1, p_2$ prime numbers. A number $n = pq$ is special if and only if $p, q$ are distinct special Blum primes.

**Definition 56.** (Blum-Blum-Shub streamcipher) Let $p, q$ be two distinct prime numbers such that $p ≡ 3$ (mod 4) and $q ≡ 3$ (mod 4). Let $n = pq$ and let $0 < r < n$ be a random number. We define $x_0$, the first number of the Blum-Blum-Shub (BBS) pseudo-random number generator, as $x_0 ≡ r^2$ (mod $n$). Each proceeding seed can be defined as

$$x_{i+1} = x_i^2 \pmod{2}.$$  

The streamcipher, $b = b_1b_2\ldots b_n\ldots$, is created by setting

$$b_i = x_i \pmod{2},$$  

thus yielding a pseudo-random string of 0s and 1s.

Note: The BBS stream cipher is most secure when $n$ is special. See Hogan’s thesis [H] for more details.

11 Bent functions

Recall a LFSR in $GF(q)$ of length $k$ is defined by an order $k$ linear homogeneous recurrence relation with constant coefficients of the form:
\[ a_n = c_1a_{n-1} + c_2a_{n-2} + \cdots + c_ka_{n-k} \]

where the \( k \) coefficients \( c_i \) (for \( 1 \leq i \leq k \)) are given (the key is \( (c_1, \ldots, c_k) \)) and the initial values \( a_0, \ldots, a_{k-1} \) are given values in \( GF(q) \). We have looked at these when \( q = 2 \); the more general case is similar. These can have good pseudo-random properties with period \( P = q^k - 1 \), but are insecure.

### 11.1 Functions with a given least support

This section follows Celerier’s USNA thesis closely \cite{C}.

It turns out it is rather easy to determine the algebraic normal form of a function if you know its support. This section discusses this idea in the Boolean case.

For each \( v \in GF(2)^n \), define a monotone function \( f = f_v \) to be atomic based on \( v \) if its support consists of all vectors greater than or equal to \( v \), i.e., if

\[ \Omega_f = \{ x \in GF(2)^n \mid v \leq x \}, \]

where \( \leq \) is the partial order defined above. We call \( f \) atomic if there is some \( v \neq 0 \) such that \( f \) is atomic based on \( v \).

**Definition 57.** Let \( f : GF(2)^n \to GF(2) \) be any monotone function. We say that \( \Gamma \subset GF(2)^n \) is the least support of \( f \) if \( \Gamma \) consists of all vectors in \( \Omega_f \) which are smallest in the partial ordering \( \leq \) on \( GF(2)^n \).

**Theorem 58.** Let \( f \) be a monotone Boolean function whose least support vectors are given by \( \Gamma \subset GF(2)^n \). Then

\[ f(x) = 1 + \prod_{v \in \Gamma} (x^v + 1). \quad (18) \]

**Proof.** Define a Boolean function \( g : GF(2)^n \to GF(2) \) such that

\[ g(x) = 1 + \prod_{v \in \Gamma} (x^v + 1) \]

where \( \Gamma \) is the set of least support vectors for a monotone Boolean function \( f \).
For $x \in GF(2)^n$, define the subset $S_x$ of least support vectors $v \in \Gamma$ such that $v \leq x$ as

$$S_x = \{ v \in \Gamma \mid v \leq x \}.$$  

We will show $f = g$ by proving $f(x) = 0 \iff g(x) = 0$.

$(\Rightarrow)$ Let $y \in GF(2)^n$ satisfy $f(y) = 0$. Then, $y \notin \Omega_f$ and $S_y = \emptyset$. Thus, for every $v \in \Gamma$, there exists an $i$ such that $v_i = 1$ and $y_i = 0$. Consequently, from the definition of $g$, we have

$$g(y) = 1 + \prod_{v \in \Gamma} (y^v + 1) = 1 + 1 = 0.$$

$(\Leftarrow)$ The converse is exactly the reverse of the above argument. We provide details for the convenience of the reader. Let $y \in GF(2)^n$ satisfy $g(y) = 0$. Since $g(y) = 1 + \prod_{v \in \Gamma} (y^v + 1)$, this means that for each $v \in \Gamma$, we have $y^v = 0$. Thus, for every $v \in \Gamma$, there exists an $i$ such that $v_i = 1$ and $y_i = 0$. This means that $y \geq v$ is false for each $v \in \Gamma$. Since $f$ is monotone, this implies $y \notin \Omega_f$, which means that $f(y) = 0$. \(\square\)

### 11.2 Cipherstreams via a filter function

To make them more secure, there are a few ways to construct a stream cipher from a “filter function.”

**Method 1** Let

$$f : GF(q)^m \rightarrow GF(q)$$

be a given function. Label the $N = q^k$ elements of $GF(q)^n$ as follows:

$$v_0 = (0, \ldots, 0, 0), v_1 = (0, \ldots, 0, 1), v_{N-1} = (1, \ldots, 1).$$

In other words, construct the “filtered” binary sequence

$$a_1 = f(v_1), \ldots, a_{N-1} = f(v_{N-1}), a_P = f(v_1), a_{N+1} = a_2, \ldots.$$

This has period $P = N - 1 = q^k - 1$.

**Method 2** Let

$$f : GF(q)^m \rightarrow GF(q)$$

be a given function. Let $a_0, a_1, \ldots$ be a LFSR of key length $k$ over $GF(q)$ having period $P = q^k - 1$.  

104
let
\[ s_0 = (a_0, ..., a_{k-1}), s_1 = (a_1, ..., a_k), ..., s_{P-1} = (a_{P-1}, a_0, ..., a_{P-k}) , \]
and produce the new sequence
\[ b_0 = f(s_0), b_1 = f(s_1), ... \]
If \( f \) is linear then this is another LFSR. However, if \( f \) is non-linear then this could be more secure.

Let
\[ \text{GF}(2)^m = \{v_0, v_1, \ldots, v_{M-1}\} , \]
where \( M = 2^m \). For any Boolean function \( f : \text{GF}(2)^m \to \text{GF}(2) \), define
\[ \hat{f} = ((-1)^{f(v_0)}, (-1)^{f(v_1)}, \ldots, (-1)^{f(v_{M-1})}) \in \mathbb{R}^M . \]
Finally, define \( \mathcal{L} \) to be the vector space of all linear functions \( \text{GF}(2)^m \to \text{GF}(2) \). Since every linear function \( f(x) \) has the form \( f(x) = x \cdot a \), for some unique \( a \in \text{GF}(2)^m \), there are \( 2^m \) elements of \( \mathcal{L} \). An affine function \( f : \text{GF}(2)^m \to \text{GF}(2) \) is one of the form \( f(x) = L(x) + \epsilon \), where \( L \in \mathcal{L} \) and \( \epsilon \in \text{GF}(2) \) is a constant.

**Example 59.** Let \( V = \text{GF}(2)^2 = \{(0,0), (1,0), (0,1), (1,1)\} \). The linear functions on \( V \) are
\[ f_0(x_0, x_1) = 0, \quad f_1(x_0, x_1) = x_0, \quad f_2(x_0, x_1) = x_1, \quad f_3(x_0, x_1) = x_0 + x_1 . \]
These correspond to
\[ \hat{f}_0 = (1,1,1,1), \quad \hat{f}_1 = (1,-1,1,-1), \quad \hat{f}_2 = (1,1,-1,-1), \quad \hat{f}_3 = (1,-1,-1,1) . \]
It is easy to check that \( \hat{f}_i \perp \hat{f}_j \), for \( i \neq j \). For a non-linear example, consider \( g(x_0, x_1) = x_0 x_1 \), which corresponds to \( \hat{g} = (1,1,1,-1) \). Note this is not perpendicular to any of the \( \hat{f}_i \).

**Example 60.** Consider the function on \( \text{GF}(2)^4 \) given by \( f(x_0, x_1, x_2, x_3) = x_0 + x_1 + x_3 \). Method 1 yields the stream cipher of period 15 given by
Figure 6: The plot of the autocorrelation function of $f$.

$$X = \{ x_n \} : 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, \ldots$$

The table of values of the autocorrelation function is

<table>
<thead>
<tr>
<th>$k$</th>
<th>$AC(k,X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1/15</td>
</tr>
<tr>
<td>2</td>
<td>-3/5</td>
</tr>
<tr>
<td>3</td>
<td>-1/15</td>
</tr>
<tr>
<td>4</td>
<td>1/5</td>
</tr>
<tr>
<td>5</td>
<td>1/5</td>
</tr>
<tr>
<td>6</td>
<td>1/5</td>
</tr>
<tr>
<td>7</td>
<td>-1/3</td>
</tr>
<tr>
<td>8</td>
<td>-1/3</td>
</tr>
<tr>
<td>9</td>
<td>1/5</td>
</tr>
<tr>
<td>10</td>
<td>1/5</td>
</tr>
<tr>
<td>11</td>
<td>-1/15</td>
</tr>
<tr>
<td>12</td>
<td>-3/5</td>
</tr>
<tr>
<td>13</td>
<td>-1/15</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

This is plotted as in Figure 6.

In other words, linear functions yield stream ciphers which do not have “small” autocorrelation.

**Example 61.** Consider the function on $GF(2)^4$ given by $f(x_0, x_1, x_2, x_3) = x_0x_3 + x_1x_2$. Method 1 yields the stream cipher of period 15 given by

$$X = \{ x_n \} : 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$$

106
Figure 7: The plot of the autocorrelation function of $f$.

The table of values of the autocorrelation function is

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC($k$,X)</td>
<td>1</td>
<td>-1/15</td>
<td>1/5</td>
<td>-1/15</td>
<td>-1/15</td>
<td>-1/15</td>
<td>-1/3</td>
<td>1/5</td>
<td>1/5</td>
<td>-1/3</td>
<td>-1/15</td>
<td>-1/15</td>
<td>-1/15</td>
<td>1/5</td>
<td>-1/15</td>
</tr>
</tbody>
</table>

This is plotted as in Figure 7.

In other words, this particular nonlinear function yields a stream cipher whose has a “small” autocorrelation.

### 11.3 The Walsh transform

This section will introduce a transform which will measure “how non-linear” a function is.

**Lemma 62.** Let
\( \mathcal{L}^\wedge = \{ \hat{f} \mid f \in \mathcal{L} \} \).

A Boolean function \( f : GF(2)^m \rightarrow GF(2) \) is affine if and only if \( \hat{f} \) is perpendicular to all but one of the vectors in \( \mathcal{L}^\wedge \).

**Proof.** Consider the dot product

\[
\hat{f} \cdot \hat{g} = \sum_{x \in V} (-1)^{f(x)}(-1)^{g(x)} = \sum_{x \in V} (-1)^{f(x)+g(x)}.
\]

Suppose \( g \in \mathcal{L} \). If \( g(x) = x \cdot b \), for some \( b \in V \), then this is

\[
\hat{f} \cdot \hat{g} = \sum_{x \in V} (-1)^{f(x)+x \cdot b}.
\]

(\( \Leftarrow \)): First, let us assume \( f \) is linear and show that it is perpendicular to all but one of the vectors in \( \mathcal{L}^\wedge \). Suppose \( f(x) = x \cdot a \), for some \( a \in V \). Then

\[
\hat{f} \cdot \hat{g} = \sum_{x \in V} (-1)^{x \cdot a + x \cdot b} = \sum_{x \in V} (-1)^{x \cdot (a+b)}.
\]

The following fact is easy to verify: given any non-zero \( c \in V \), the number of times \( x \cdot c = 0 \) is \( 2^m / 2 \) and the number of times \( x \cdot c = 1 \) is \( 2^m / 2 \). Based on this, we see that

\[
\hat{f} \cdot \hat{g} = \begin{cases} 2^m, & a + b = 0, \\ 0, & a + b \neq 0. \end{cases} \tag{19}
\]

(\( \Rightarrow \)): Now, suppose \( f \) is any function which has the property that

\[
\hat{f} \cdot \hat{g} = \sum_{x \in V} (-1)^{f(x)+x \cdot b}
\]

is equal to 0 for all \( b \in V \) except for one (to be discussed below). In other words, if we multiply the vector \( \hat{f} \) by the \( 2^m \times 2^m \) matrix

\[
H = ((-1)^{v_i \cdot u_j})_{0 \leq i,j \leq M-1},
\]

then we get a vector in \( \mathbb{R}^M \) in which all but one entry is 0. The matrix \( H \) is invertible. In fact, it is not hard to check using (19) that

\[
H^2 = M \cdot I_M,
\]

108
where $I_M$ is the $m \times M$ identity matrix. Therefore, $H^{-1} = M^{-1}H$. This implies that
\[ \hat{f} = H^{-1}\bar{v} = M^{-1}H\bar{v}, \]
where $\bar{v} = (w_1, \ldots, w_M) \in \mathbb{R}^M$ is zero for all but one coordinate, say $w_\ell \neq 0$. Therefore, $\hat{f}$ is a constant multiple of the $\ell$-th column of $H$. This forces $\hat{f}$ to be of the form $c\hat{g}$, for some constant $c$ and some linear function $g$. If $c = 1$ then $\hat{f}$ is linear and if $c = -1$ then $\hat{f}$ is affine.

□

This motivates the following question: What is the most non-linear function one can find?

A Boolean function
\[ f : GF(2)^m \to GF(2) \]
is bent (also called perfectly non-linear) if, for all $a \in GF(2)^m$,
\[ |W_f(a)| = 2^{m/2}, \]
where $W$ is the Walsh transform. If $f$ is any Boolean function as above then its Walsh transform is
\[ W_f(a) = \sum_{x \in GF(2)^m} (-1)^{f(x)+x \cdot a}. \]

Two questions arise:

1. How do you compute the Walsh transform?

2. Why does this condition on the Walsh transform of $f$ guarantee that $f$ is “very” non-linear?

Computing $W_f(a)$:

Order $GF(2)^m$ lexicographically
\[ v_0 = 000\ldots0 < v_1 = 100\ldots0 < v_2 = 010\ldots0 < \cdots < v_M = 111\ldots1, \]
where $M = 2^m - 1$. Using this order, define the character of $f$ by
It turns out that $f$ is bent if and only if its character $f^*$ is “as far as possible” from the characters $\ell^*$, where $\ell$ is taken from the collection of affine functions $\ell(x) = b \cdot x + \epsilon$. Define the Hadamard matrix $H$ by

$$H = ((-1)^{v_i \cdot v_j} \mid 0 \leq i, j \leq M)$$

Then the Walsh transform can be expressed in terms of this matrix transformation:

$$W_f = H f^*.$$ 

**Definition 63.** We say that a Boolean function

$$f : GF(2)^m \to GF(2)$$

is linear if there is a $b \in GF(2)^m$ such that, for all $x \in GF(2)^m$,

$$f(x) = x \cdot b.$$ 

**Lemma 64.** Suppose $f(x) = x \cdot b$. Then

$$W_f(a) = \begin{cases} 
0, & \text{if } a \neq b, \\
2^m, & \text{if } a = b,
\end{cases}$$

Conversely, if

$$f : GF(2)^m \to GF(2)$$

is any Boolean function satisfying

$$W_f(a) = \begin{cases} 
0, & \text{if } a \neq b, \\
2^m, & \text{if } a = b,
\end{cases}$$

for some $b \in GF(2)^m$, then $f(x) = x \cdot b$.

**Example 65.** The function $f(x_1, x_2, x_3, x_4) = x_1 x_3 + x_2 x_4$ defines a bent function on $GF(2)^4$. Exercise: Verify this by computing $W_f(a)$ for each $a \in GF(2)^4$. 

110
12 Error-correcting codes

Roughly speaking a code is a system for converting a message into another form for the purpose of communicating the message more efficiently or reliably. A few examples are listed below.

- Semaphore, where a message is converted into a sequence of flag movements for communication across a distance.

- Morse code, where a message is converted into a sequence of dots and dashes for communication using telegraph. For example, LEG is 
  
  \[ \cdot - \cdot \cdot \cdot \cdot \cdot \cdot \cdot \]

  and RUN is

  \[ \cdot - \cdot - \cdot \cdot \cdot \cdot \]

  (They share the same bit pattern in Morse code, as do EARN and URN [FC].)

- Marconi Telegraph Code, where a commonly used phrase is converted into a more compact 5-letter sequence. For example, “what do you suggest?” is encoded as “VYHIC” [SM].

A code could be used as a cipher, but most codes are not created with security in mind. For example, during the Prohibition Era, rumrunners used slightly modified telegraph codes to transmit shipment information and meeting places for ship-loads of alcohol. Such ciphers were routinely broken by Coast Guard cryptographers.

Some codes are designed for compression - to store digital data more compactly. Some codes are designed for reliability - to communicate information over a noisy channel, yet to correct the errors which arise.

12.1 The communication model

Consider a source sending messages through a noisy channel. The message sent will be regarded as a vector of length \( n \) whose entries are taken from a given finite field \( F \) (typically, \( F = GF(2) \)).
For simplicity, assume that the message being sent is a sequence of 0’s and 1’s. Assume that, due to noise, when a 0 is sent, the probability that a 1 is (incorrectly) received is \(p\) and the probability that a 0 is (correctly) received is \(1 - p\). The error rate \(p\) is a small positive number (such as \(1/10000\)) which represents the “noisiness” of the channel. Assume also that the error rate (and channel noise) is not dependent on the symbol sent: when a 1 is sent, the probability that a 1 is (correctly) received is \(1 - p\) and the probability that a 0 is (incorrectly) received is \(p\).

### 12.2 Basic definitions

The theory of error-correcting codes was originated by Richard Hamming in the late 1940’s, a mathematician who worked for Bell Telephone. Some specific examples of his codes actually arose earlier in various isolated connections - for example, statistical design theory and in soccer betting(!). Hamming’s motivation was to program a computer to correct “bugs” which arose in punch-card programs. The overall goal behind the theory of error-correcting codes is to reliably enable digital communication.

Let \(F = GF(q)\) be any finite field.

A (linear error-correcting) code \(C\) of length \(n\) over \(F\) is a vector subspace of \(F^n\) (provided with the standard basis\(7\)) and its elements are called code-words. When \(F = GF(2)\) it is called a binary code. These are the most important codes from the practical point of view. Think of the following scenario: You are sending an \(n\)-vector of 0’s and 1’s (the codeword) across a noisy channel to your friend. Your friend gets a corrupted version (the received word differs from the codeword in a certain number of error positions). Depending on how the code \(C\) was constructed and the number of errors made, it is possible that the original codeword can be recovered. This raises the natural question: given \(C\), how many errors can be corrected? Stay tuned...

A code of length \(n\) and dimension \(k\) (as a vector space over \(F\)) is called an \([n, k]\)-code. In abstract terms, an \([n, k]\)-code is given by a short exact sequence\(8\).

---

\(7\)It is important that the code be provided with a fixed basis which never changes. This is because the minimum distance function is not invariant under a change of basis. However, the minimum distance is one quantity used to measure how “good” a code is, from the practical point of view.

\(8\)“Short exact” is a compact way of specifying the following three conditions at once:
\[ 0 \rightarrow \mathbb{F}^k \overset{G}{\rightarrow} \mathbb{F}^n \overset{H}{\rightarrow} \mathbb{F}^{n-k} \rightarrow 0. \] (20)

We identify \( C \) with the image of \( G \).

**Example 66.** The matrix \( G = (1, 1, 1) \) defines a map \( G : GF(2) \rightarrow GF(2)^3 \).

The image is

\[ C = Im(G) = \{(0, 0, 0), (1, 1, 1)\}. \]

The matrix

\[ H = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \]

defines a map \( H : GF(2)^3 \rightarrow GF(2)^2 \). It is not hard to check \( G \cdot H = 0 \).

The function

\[ G : \mathbb{F}^k \rightarrow C, \quad \vec{m} \mapsto \vec{m}G, \]

is called the *encoder*. Since the sequence (20) is exact, a vector \( \vec{v} \in \mathbb{F}^n \) is a codeword if and only if \( H(\vec{v}) = 0 \). If \( \mathbb{F}^n \) is given the usual standard vector space basis then the matrix of \( G \) is a *generating matrix* of \( C \) and the matrix of \( H \) is a *check matrix* of \( C \). In other words,

\[ C = \{ \vec{c} | \vec{c} = \vec{m}G, \text{ some } \vec{m} \in \mathbb{F}^k \} \]

\[ = \{ \vec{c} \in \mathbb{F}^n | H\vec{c} = 0 \}. \]

When \( G \) has the block matrix form

\[ G = (I_k | A), \]

where \( I_k \) denotes the \( k \times k \) identity matrix and \( A \) is some \( k \times (n - k) \) matrix, then we say \( G \) is in *standard form*. By abuse of terminology, if this is the case then we say \( C \) is in *standard form*.

1. the first map \( G \) is injective, i.e., \( G \) is a full-rank \( k \times n \) matrix,
2. the second map \( H \) is surjective, and
3. image(\( G \)) = kernel(\( H \)).
Example 67. The matrix
\[
G = \begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1
\end{pmatrix},
\]
is a generating matrix for a code in standard form.

The matrix $G$ has rank $k$, so the row-reduced echelon form of $G$, call it $G'$, has no rows equal to the zero vector. In fact, the standard basis vectors $\vec{e}_1, \ldots, \vec{e}_k$ of the column space $F^k$ occur amongst $k$ columns of those of $G'$. The corresponding coordinates of $C$ are called the information coordinates (or information bits, if $C$ is binary) of $C$.

Aside: For a “random” $k \times k$ matrix with real entries, the “probability” that its rank is $k$ is of course 1. This is because “generically” a square matrix with real entries is invertible. In the case of finite fields, this is not the case. For example, the probability that a “large random” $k \times k$ matrix with entries in $GF(2)$ is invertible is
\[
\lim_{k \to \infty} \frac{(2^k - 1)(2^k - 2)\ldots(2^k - 2^{k-1})}{2^{k^2}} = \prod_{i=1}^{\infty} (1 - 2^{-i}) = 0.288\ldots.
\]

The Hamming metric is the function
\[
d : F^n \times F^n \to \mathbb{R},
\]
\[
d(\vec{v}, \vec{w}) = |\{i \mid v_i \neq w_i\}| = d(\vec{v} - \vec{w}, \vec{0}).
\]
The Hamming weight of a vector is simply its distance from the origin:
\[
wt(\vec{v}) = d(\vec{v}, \vec{0}).
\]

Question: How many vectors belong to the “shell” of radius $r$ about the origin $\vec{0} \in GF(q)^r$?

Answer: $\left(\begin{array}{c}n \\ r\end{array}\right)(q-1)^r$. Think about it! (Hint: “distance $r$” means that there are exactly $r$ non-zero coordinates. The binomial coefficient describes the number of ways to choose these $r$ coordinates.)

The minimum distance of $C$ is defined to be the number
\[
d(C) = \min_{\vec{c} \neq \vec{0}} d(\vec{c}, \vec{0}).
\]
(It is not hard to see that this is equal to the closest distance between any two distinct codewords in \( C \).) An \([n,k,d]\)-code with minimum distance \( d \) is called an \([n,k,d]\)-code.

**Lemma 68.** (Singleton bound) Every linear \([n,k,d]\) code \( C \) satisfies
\[
k + d \leq n + 1.
\]

Note: this bound does not depend on the size of \( \mathbb{F} \). A code \( C \) whose parameters satisfy \( k + d = n + 1 \) is called maximum distance separable or MDS. Such codes, when they exist, are in some sense best possible.

**proof:** Fix a basis of \( \mathbb{F}_q^n \) and write all the codewords in this basis. Delete the first \( d - 1 \) coordinates in each code word. Call this new code \( C' \). Since \( C \) has minimum distance \( d \), these codewords of \( C' \) are still distinct. There are therefore \( q^k \) of them. But there cannot be more than \( q^{n-d+1} = |\mathbb{F}_q^{n-d+1}| \) of them. This gives the inequality. \( \square \)

The rate of the code is \( R = k/n \) - this measures how much information the code can transmit. The relative minimum distance of the code is \( \delta = d/n \) - this is directly related to how many errors can be corrected.

**Lemma 69.** If \( \vec{v} \in \mathbb{F}^n \) is arbitrary and \( 0 < r \leq \left\lfloor \frac{d-1}{2} \right\rfloor \) then the “ball” about \( \vec{v} \) with radius \( r \),
\[
B_r(\vec{v}) = \{ \vec{w} \in \mathbb{F}^n \mid d(\vec{v},\vec{w}) \leq r \}
\]
contains at most one codeword in \( C \).

This follows easily from the fact that the Hamming metric is, in fact, a metric. Here is a picture of the idea.
Lemma 70. (sphere-packing bound) For any code \( C \subset \mathbb{F}^n \), we have

\[
|C| \sum_{i=0}^{t} \binom{n}{i} (q-1)^i \leq q^n,
\]

where \( t = [(d-1)/2] \).

**proof:** For each codeword of \( C \), construct a ball of radius \( t \) about it. These are non-intersecting, by definition of \( d \) and the previous lemma. Each such ball has

\[
\sum_{i=0}^{t} \binom{n}{i} (q-1)^i
\]

elements. The result follows from the fact that \( \bigcup_{\vec{c} \in C} B_t(\vec{c}) \subset \mathbb{F}^n \) and \( |\mathbb{F}^n| = q^n \). \( \square \)

Suppose (a) you sent \( \vec{c} \in C \), (b) your friend received \( \vec{v} \in \mathbb{F}^n \), (c) you know (or are very confident) that the number \( t \) of errors made is less than or equal to \( [(d-1)/2] \). By the lemma above, the “ball” about \( \vec{v} \) of radius \( t \) contains a unique codeword. It must be \( \vec{c} \), so your friend can recover what you sent (by searching though all the vectors in the ball and checking which one is in \( C \)) even though she/he only knows \( C \) and \( \vec{v} \). This is called the nearest neighbor decoding algorithm:

1. **Input:** A received vector \( \vec{v} \in \mathbb{F}^n \).
   **Output:** A codeword \( \vec{c} \in C \) closest to \( \vec{v} \).

2. Enumerate the elements of the ball \( B_t(\vec{v}) \) about the received word. Set \( \vec{c} = \text{“fail”} \).
3. For each \( \vec{w} \in B_t(\vec{v}) \), check if \( \vec{w} \in C \). If so, put \( \vec{c} = \vec{w} \) and break to the next step; otherwise, discard \( \vec{w} \) and move to the next element.
4. Return \( \vec{c} \).

Note “fail” is not returned unless \( t > [(d-1)/2] \), by the above lemma.

**Definition 71.** We say that a linear \( C \) is \( t \)-error correcting if \( |B_t(\vec{w}) \cap C| \leq 1 \).

Note that \( t \leq [(d-1)/2] \) if and only if \( d \geq 2t + 1 \).

The general goal in the theory is to optimize the following properties:
• the rate, \( R = k/n \),
• the relative minimum distance, \( \delta = d/n \),
• the speed at which a “good” encoder for the code can be implemented,
• the speed at which a “good” decoder for the code can be implemented.

There are (sometimes very technical) constraints on which these can be achieved, as we have seen with the Singleton bound and the sphere-packing bounds.

### 12.3 Binary hamming codes

This material can be found in many standard textbooks, such as [JKT].

A Hamming code is a member of a family of binary error-correcting codes defined by Richard Hamming, a Bell telephone mathematician, in the 1940s.

**Definition 72.** Let \( r > 1 \). The Hamming \([n,k,3]\)-code \( C \) is the linear code with

\[
n = 2^r - 1, \quad k = 2^r - r - 1,
\]

and parity check matrix \( H \) defined to be the matrix whose columns are all the (distinct) non-zero vectors in \( GF(2)^r \). By Lemma 73, this code has minimum distance \( d = 3 \).

**Lemma 73.** Every binary Hamming code \( C \) has minimum distance \( 3 \).

**Proof.** Indeed, if \( C \) has a code word of weight 1 then the parity check matrix \( H \) of \( C \) would have to have a column which consists of the zero vector, contradicting the definition of \( H \). Likewise, if \( C \) has a code word of weight 2 then the parity check matrix \( H \) of \( C \) would have to have two identical columns, contradicting the definition of \( H \). Thus \( d \geq 3 \).

Since

\[
\begin{pmatrix}
1 \\
0 \\
0 \\
\vdots \\
0
\end{pmatrix}, \quad \begin{pmatrix}
0 \\
1 \\
0 \\
\vdots \\
0
\end{pmatrix}, \quad \text{and} \quad \begin{pmatrix}
1 \\
1 \\
0 \\
\vdots \\
0
\end{pmatrix},
\]

form three columns of the parity check matrix \( H \) of \( C \) - say the 1st, 2nd, and 3rd columns - the vector \((1,1,1,0,...,0)\) must be a code word. Thus \( d \leq 3 \).

\(\square\)
Example 74. $r = 2$: The Hamming $[3, 1]$-code has parity check matrix

$$H = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

The matrix $G = (1, 1, 1)$ is a generating matrix.

$r = 3$: The Hamming $[7, 4]$-code has parity check matrix

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

The matrix

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

is a generating matrix.

Example 75. Consider the Hamming $[7, 4]$ example above. The meaning of the statement that $G$ is a generator matrix is that a vector

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}$$

is a codeword if and only if $\vec{x}$ is a linear combination of the rows of $G$. The meaning of the statement that $H$ is a check matrix is $H\vec{x} = \vec{0}$, ie

$$x_1 + x_4 + x_6 + x_7 = 0, x_2 + x_4 + x_5 + x_7 = 0, x_3 + x_5 + x_6 + x_7 = 0.$$

This may be visualized via a Venn diagram (see Figure 75).

Decoding algorithm for the Hamming $[7, 4]$-code

Denote the received word by

$$\vec{w} = (w_1, w_2, w_3, w_4, w_5, w_6, w_7).$$
1. Put $w_i$ in region $i$ of the Venn diagram above, $i = 1, 2, ..., 7$.

2. Do parity checks on each of the circles $A$, $B$, and $C$.

<table>
<thead>
<tr>
<th>parity failure region(s)</th>
<th>error position</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>A, B, and C</td>
<td>7</td>
</tr>
<tr>
<td>B and C</td>
<td>5</td>
</tr>
<tr>
<td>A and C</td>
<td>6</td>
</tr>
<tr>
<td>A and B</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
</tbody>
</table>

Here is some Sage code to illustrate this:

```
sage: C = codes.HammingCode(3,GF(2)); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.minimum_distance()
3
sage: H = matrix(GF(2), 3, 7, [[1, 0, 0, 1, 0, 1, 1], [0, 1, 0, 1, 1, 0, 1], [0, 0, 1, 0, 1, 1, 1]])
sage: H
[1 0 0 1 0 1 1]
[0 1 0 1 1 0 1]
[0 0 1 0 1 1 1]
sage: C = codes.LinearCodeFromCheckMatrix(H)
sage: C.check_mat()
[1 0 0 1 0 1 1]
[0 1 0 1 1 0 1]
[0 0 1 0 1 1 1]
sage: C.minimum_distance()
3
sage: C.list()
[(0, 0, 0, 0, 0, 0, 0), (1, 0, 0, 0, 1, 0, 1), (0, 1, 0, 0, 0, 1, 1), (1, 1, 0, 0, 1, 1, 0), (0, 0, 1, 0, 1, 1, 1), (1, 0, 1, 0, 0, 1, 0), (0, 1, 1, 0, 1, 0, 0), (1, 1, 1, 0, 0, 0, 1), (0, 0, 0, 1, 1, 1, 0), (1, 0, 0, 1, 0, 1, 1), (0, 1, 0, 1, 0, 1, 1), (1, 1, 0, 1, 0, 0, 1), (0, 0, 1, 1, 0, 0, 0), (1, 1, 1, 1, 0, 0, 1), (0, 0, 1, 1, 0, 0, 1)]
```
12.4 Coset leaders and the covering radius

Let $C \subset GF(q)^n$ be a linear block code with generator matrix $G$ and check matrix $H$.

*Question:* What is the largest radius $r$ such that the balls of radius $r$ centered about all the codewords,

$$B(c, r) = \{v \in GF(q)^n \mid d(c, v) \leq r\}$$

are disjoint?

*Answer:* $\left\lceil (d-1)/2 \right\rceil$. By the above proof, we see that the triangle inequality will not allow two balls centered at neighboring codewords are disjoint if and only if they have radius $\leq \left\lceil (d-1)/2 \right\rceil$.

The union of all these disjoint balls of radius $\left\lceil (d-1)/2 \right\rceil$ centered at the codewords in $C$ usually does not equal the entire space $V = GF(q)^n$. (When it does, $C$ is called *perfect*).

How much larger do we have to make the radius so that the union of these balls does cover all of $V$? In other words, we want to answer the following question:

*Question:* What is the smallest radius $\rho$ such that

$$\bigcup_{c \in C} B(c, \rho) = V?$$

*Answer:* At the present time, there are no simple general formulas for $\rho$ and, in general, it is hard to even find good upper bounds on $\rho$. However, there is a sharp lower bound:

$$\rho \geq \left\lceil (d-1)/2 \right\rceil.$$ 

This radius $\rho$ is called the *covering radius*.

If you think about it for a moment, you’ll realize that the covering radius is the maximum value of

$$\text{dist}(v, C) = \min_{c \in C} \text{wt}(v - c) = \min_{c \in C} \text{wt}(v + c),$$
over all $v \in GF(q)^n$.

A coset is a subset of $GF(q)^n$ of the form $C + v$ for some $v \in GF(q)^n$. Equivalently, a coset is a pre-image of some $y$ in $GF(q)^{n-k}$ under the check matrix $H : GF(q)^n \to GF(q)^{n-k}$. Let $S$ be a coset of $C$. A coset leader of $S$ is an element of $S$ having smallest weight. The covering radius is, evidently, the highest weight of all the coset leaders of $C$.

**Theorem 76.** The coset leaders of a Hamming code are those vectors of wt $\leq 1$.

**Proof.** Let the Hamming code be defined as a $[n, k, d]$ code as above where for some integer $r$, $n = 2^r - 1$, $k = 2^r - 1 - r$, and $d = r$. In the binary case, the size of the ambient space is $q^n = 2^n = |GF(q)^n|$ and the size of the code is $q^k = 2^k = |C|$. Thus, the size of any coset $S$ of $C$ is

$$|S| = |GF(q)^n|/|C| = 2^{n-k} = 2^r = n + 1.$$ 

**Claim:** Each coset contains a coset leader of wt $\leq 1$ and no coset contains more than one vector of wt $\leq 1$. **Proof of claim:** Assume that $v + C$ is one such coset with two distinct vectors $w_1, w_2$ of wt $\leq 1$. Then,

$$w_1 = v + c_1, w_2 = v + c_2.$$ 

So,

$$w_1 - w_2 = c_1 - c_2 \in C.$$ 

And, since $wt(w_1 - w_2) = 2$ and $d(C) = 3$ for a Hamming code, we have a contradiction. Also, by the Pigeonhole Principle, each coset contains exactly one vector of wt $= 1$. Thus, the claim holds, and this also proves the theorem. □

**Theorem 77.** Hamming codes are perfect.

**Proof.** Since $d(C) = 3$ for Hamming codes, we desire to show that equality holds in

$$\rho = \lfloor (d - 1)/2 \rfloor = 1.$$ 

To attain a contradiction, assume

$$\rho = \max_{x \in GF(q)^n} d(x, C) > 1;$$

then for some $x \in GF(q)^n$, $d(x, C) > 1$. But by the previous theorem, the coset $x + C$ must contain a coset leader of wt $\leq 1$, a contradiction to the assumption that $d(x, C) > 1$. Thus, $\rho = 1$. □
A final remark on coset leaders. These were introduced by Slepian in one of the first papers published after Hamming’s paper introducing the subject in the late 1940s. Slepian also developed the following general decoding algorithm:

1. Input: A received vector \( \vec{v} \in \mathbb{F}^n \).
   Output: A codeword \( \vec{c} \in C \) closest to \( \vec{v} \).

2. Compute the coset \( S = \vec{v} + C \) of \( \vec{v} \), the received word. Compute the coset leader of \( S \), call it \( \vec{c} \).
   Slepian’s way to do this:
   - Precompute all the coset leaders \( \vec{u} \) of \( C \) and tabulate all the values \( (\vec{u}, H\vec{u}) \).
   - Compute the syndrome of \( \vec{v} \): \( \vec{s} = H\vec{v} \). Search the 2nd coordinate of the tabulated pairs \( (\vec{u}, H\vec{u}) \) for this syndrome. Select the 1st coordinate from that pair, \( \vec{u} \).
   - Let \( \vec{c} = \vec{u} \).

3. Put \( \vec{c} = \vec{v} - \vec{c} \).

4. Return \( \vec{c} \).

**Example 78.** Let \( C \subset GF(3)^5 \) be the code with generator matrix
\[
G = \begin{pmatrix}
1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0
\end{pmatrix}
\]
and check matrix
\[
H = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 \\
0 & 1 & 2 & 0 & 0 \\
0 & 0 & 0 & 1 & 2
\end{pmatrix}
\]
The minimum distance of this code is 2. The 27 cosets of \( C \) are

\[
\begin{align*}
\{(0,0,0,0,0), (0,1,1,0,0), (0,2,2,0,0), (1,0,0,1,1), (1,1,1,1,1), (1,2,2,1,1), (2,0,0,2,2), (2,1,1,2,2), (2,2,2,2,2)\}, \\
\{(0,0,0,2,2), (0,1,1,2,2), (0,2,2,2,2), (1,0,0,0,0), (1,1,1,0,0), (1,2,2,0,0), (2,0,0,1,1), (2,1,1,1,1), (2,2,2,1,1)\}, \\
\{(0,0,0,1,1), (0,1,1,1,1), (0,2,2,1,1), (1,0,0,2,2), (1,1,1,2,2), (1,2,2,2,2), (2,0,0,0,0), (2,1,1,0,0), (2,2,2,0,0)\}, \\
\{(0,0,2,0,0), (0,1,1,0,0), (0,2,2,1,1), (1,0,0,1,1), (1,1,1,1,1), (1,2,2,1,1), (2,0,2,2,2), (2,1,0,2,2), (2,2,1,2,2)\}, \\
\{(0,0,2,2,2), (0,1,0,2,2), (0,2,1,2,2), (1,0,2,0,0), (1,1,0,0,0), (1,2,1,0,0), (2,0,2,1,1), (2,1,0,1,1), (2,2,1,1,1)\}
\end{align*}
\]
The corresponding coset leaders are

\{ (0, 0, 0, 0, 0), (0, 0, 0, 2, 0), (0, 0, 0, 1, 1), (0, 0, 0, 1, 2), (0, 0, 2, 2, 2), (0, 0, 2, 2, 0), (0, 0, 2, 1, 1), (0, 0, 1, 0, 0), (0, 0, 1, 2, 0) \}

The covering radius is therefore 3.
13 Steganography

Basic idea:
Steganography, meaning “covered writing,” is the science of secret communication. The medium used to carry the information is called the ”cover” or ”stego-cover.” The term ”digital steganography” refers to secret communication where the cover is a digital media file.

Cryptography is usually devoted to messages between parties where the ciphertext is known. In steganography, one tries to hide the existence of the communication itself. Early examples include writing on the backing of a wax tablet before covering it, shaving and tattooing the head of a slave, only to let the hair re-grow, or using invisible ink.

One of the most common systems of digital steganography is the Least Significant Bit (LSB) system. In this system, the encoder embeds one bit of information in the least significant bit of a binary number representing the darkness of a pixel of a given image. In this situation, a greyscale image is regarded as an array of pixels, where each pixel is represented by a binary vector of a certain fixed length. Care must be taken with this system to ensure that the changes made do not betray the stego-cover, while still maximizing the information hidden.

From a short note of Crandell [Cr] in 1998, it was realized that error-correcting codes can give rise to “stego-schemes,” ie, methods by which a message can be hidden in a digital file efficiently.

Usages:
Espionage: In the summer of 2010, 11 suspected Russian spies were arrested in the United States. According to the FBI’s charging documents, these individuals used a steganographic system that embedded information into images.

Terrorism: Steganography is discussed in “Technical Mujahid, Issue #2” from February 2007. Despite numerous newspaper reports (eg, USA Today in the early 2000’s), there are no (publicly) known cases of terrorist groups using steganography.

A really simple illustration can be made using Sage’s matrix_plot command.

Here is Sage code used to create the “tank” on the left side of Figure 8. What matrix produces the “tank” on the right?

\[
T = \begin{bmatrix} 1, 1, 1, 1, 1, 1, 1, 1, 1 \end{bmatrix},
\]
The plot of two “tanks” using `matrix_plot`.

The images look very similar, since we have only changed one “pixel.” Where that changed pixel is located can communicate (hidden) information.

The rest of our discussion will follow Tucker-Davis’ thesis [TD].

### 13.1 Basic terminology

Following [Mu], a **steganographic system** $S$ can be formally defined as

$$S = \{C, M, K, \text{emb}, \text{rec}\},$$

where

1. $C$ is a set of all possible covers
2. $M$ is a set of all possible messages
3. $K$ is a set of all possible keys
(iv) \( \text{emb} : \mathcal{C} \times \mathcal{M} \times \mathcal{K} \to \mathcal{C} \) is an embedding function

(v) \( \text{rec} : \mathcal{C} \times \mathcal{K} \to \mathcal{M} \) is a recovery function

and

\[
\text{rec}(\text{emb}(c, m, k), k) = m, \quad (21)
\]

for all \( m \in \mathcal{M} \), \( c \in \mathcal{C} \), \( K \in \mathcal{K} \). We will assume that a fixed key \( K \in \mathcal{K} \) is used, and therefore, the dependence on an element in the keyspace \( \mathcal{K} \) can be ignored. The original cover \( c \) is called the \textit{plain cover}, \( m \) is called the \textit{message}, and \( \text{emb}(c, m, k) = \text{emb}(c, m) \) is called the \textit{stegocover}. Let \( \mathcal{C} \) be \( GF(q)^n \), representing both plain and stego covers. Also, let \( \mathcal{M} \) be \( GF(q)^k \), where \( k \) is a fixed integer such that \( 0 \leq k \leq n \).

An \((n,k,t)\) \textit{stegocoding function} over finite field \( GF(q) \) is a vector-valued linear transformation \( L(x) = (\ell_1(x), \ell_2(x), \ldots, \ell_k(x)) : GF(q)^n \to GF(q)^k \) satisfying the following condition: For any given \( x \in GF(q)^n \) and \( y \in GF(q)^k \), there exists a \( z \in GF(q)^n \) such that \( \text{wt}(z) \leq t \) and \( L(x + z) = y \). We call the matrix \( L \) an \((n,k,t)\) \textit{stego-code matrix}.

In this case,

- \( x \) is the plain cover,
- \( y \) is the message,
- \( x + z \) is the stegocover (so \( z \) is the change made to the plain cover).

Of course, \( z \) is derived from the specific stegosystem used, but generally it is desired that the weight of \( z \) is small (so the plain cover is not too easily distinguishable from the stegocover) while still retaining the ability to embed as much information as possible (so that the stegosystem efficiently conveys information).

### 13.2 Examples

First, we consider and example from a linear error-correcting block code, \( C \).

\textbf{Example 79.} Let \( C \) be the \([n,n-k,d]\) binary linear code of dimension \( n-k \) and minimum distance \( d \) having check matrix \( L \). In other words, \( C \) is given by \( C = \ker(L) \), where \( L \) is a \( k \times n \) matrix of full rank over \( GF(2) \). Choose \( t \) such that it is the largest weight of all the coset leaders of \( C \). For each
$y \in GF(2)^k$, let $v = v(y) \in GF(2)^n$ be such that $L^{-1}(y) = C + v$. For each $x \in GF(2)^n$, select $z$ to be an element of $C + v + x$ of smallest weight, i.e. a coset leader.

**Theorem 80.** Assume $L$ is a $k \times n$ matrix as above. $L$ is a $(n, k, t)$ linear stego-code matrix iff $t$ is the covering radius of the code $C$ whose check matrix is $L$.

The proof of the theorem above can be found in [Mu].

Let us consider an example using a $[7, 4, 3]$ Hamming code. Define the check matrix as follows:

$$
H = \begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 1
\end{pmatrix}.
$$

Denote the coset space by $Q = GF(2)^7/C$, where $C = \ker(H)$ is a $[7, 4, 3]$ Hamming code.

Let $x = (1, 1, 1, 1, 1, 1, 1)$ and $y = (1, 0, 0)$ We verify that $z = (0, 0, 0, 0, 1, 0, 0)$, since

$$
\begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 1
\end{pmatrix}
\begin{pmatrix}
1 \\
1 \\
1 \\
1 \\
0 \\
1 \\
1
\end{pmatrix} = \begin{pmatrix}
0 \\
1
\end{pmatrix} = y.
$$

Next, we consider the extremely clever “F5” stegosystem of Ron Crandall [Cr].

**Example 81.** First, new notation must be defined. For any positive integer $m$, $[m]_2$ denotes the binary representation of $m$. Similarly, for any binary representation $x$, $[x]_{10}$ denotes the associated positive integer. Therefore, for any binary $x$, $[[x]_{10}]_2 = x$, and for any positive integer $m$, $[[m]_2]_{10} = m$.

In this system, the embedding map is defined as

$$
\text{emb} : GF(2)^n \times GF(2)^k \to GF(2)^n,
$$

$$
\text{emb}(c, m) = c + e\left([m + \sum_{i=1}^{n} c_i [i]_2]_{10}\right). \quad (22)
$$

127
At this point, note that the message is in $GF(2)^q$, and both the plaintext and stegocover are in $GF(2)^n$.

The recovery map is

$$\text{rec} : GF(2)^n \rightarrow GF(2)^k$$

$$\text{rec}(c') = \sum_{i=1}^{n} c'_{[i]}_2 .$$

(24)

(25)

**Theorem 82.** With $\text{emb}$ and $\text{rec}$ as in (23) and (25), respectively, we have $\text{rec}(\text{emb}(c,m)) = m$.

This is proven in the USNA thesis of Tucker-Davis [TD].

**References**

http://www.math.mtu.edu/jbierbra/


http://crypto.stanford.edu/~dabo/abstracts/RSAattack-survey.html

http://www.usna.edu/Users/math/wdj/_files/documents/brock/

http://www.usna.edu/Users/math/wdj/_files/documents/celerier/


http://os.inf.tu-dresden.de/westfeld/crandall.pdf

[Cry] Cryptologia, a scholarly journal dealing with the history and technology of communications intelligence.

http://www.tandfonline.com/loi/ucry20


[FC] Isomorses article on *Futility Closet* website by Greg Ross,

http://www.futilitycloset.com/2015/01/23/isomorses/


http://books.google.com/books/about/Cryptanalysis.html?id=fKNB-7y_Hs4C


[H] Michael Hogan’s Honors Project 2009-2010, *The Blum-Goldwasser cryptosystem*,

http://www.usna.edu/Users/math/wdj/_files/documents/hogan/


http://arxiv.org/abs/cs/0505072
Index

auxiliary polynomial, 80
baby step, giant step, 73
binary expansion, 38
Binet’s formula, 85
characteristic polynomial, 80
code
  perfect, 120
  companion matrix, 81
  connection polynomial, 80
coset leader, 121
covering radius, 120
digits, 38
ElGamal encryption, 76
feedback polynomial, 92
field, 46
Hamming codes, 117
initial fill, 92
length
  LRS, 80
letter frequency, 21
linear recurrence sequence, 80
order
  LRS, 80
Slepian, 122