FACULTY OF ENGINEERING
ALEXANDRIA UNIVERSITY

COMMUNICATION AND ELECTRONICS DEPARTMENT
Academic Year 2023-2024

Advanced Encryption and Steganography Techniques for Multimedia Signals in 5G/6G Systems

Presented By:

Abdelrahman Mohamed Ahmed Mohamed Badr	**19015924**
Ali Ehab Mohamed Bakry Mohamed Sono	**19016012**
Laila-Salma Nasr Ezzelarab Ghaly	**17011396**
Mahmoud Mohamed Muhalal Ahmed Muhalal	**19016589**
Youssef Hamdino Ragab Ibrahim	**19016917**
Nader Magdy Mostafa Ahmed	**18011948**

Supervised By:

Prof.Dr.Said I. El-Khamy

Abstract

The emergence of 5G and the upcoming 6G technologies present unprecedented opportunities and challenges for securing multimedia communications.

This project explores the integration of advanced encryption and steganography techniques to enhance the security of multimedia signals in 5G/6G systems. The project dives into state-of-the-art encryption methods, including chaotic codes. Additionally, innovative steganography techniques are examined, such as adaptive Least Significant Bit (LSB) manipulation approaches for data embedding in images, audio.

These techniques are assessed for their imperceptibility, robustness, and capacity to withstand high-bandwidth and low-latency demands of 5G/6G networks. The synergy between advanced encryption and steganography is highlighted, demonstrating enhanced security layers that protect against interception and unauthorized access.

The project concludes with a discussion on the applications for secure multimedia communication in next-generation networks and proposes future research directions to address emerging security challenges.

This research underscores the critical need for continuous innovation in securing multimedia signals within the dynamic landscape of 5G/6G systems.

Contents

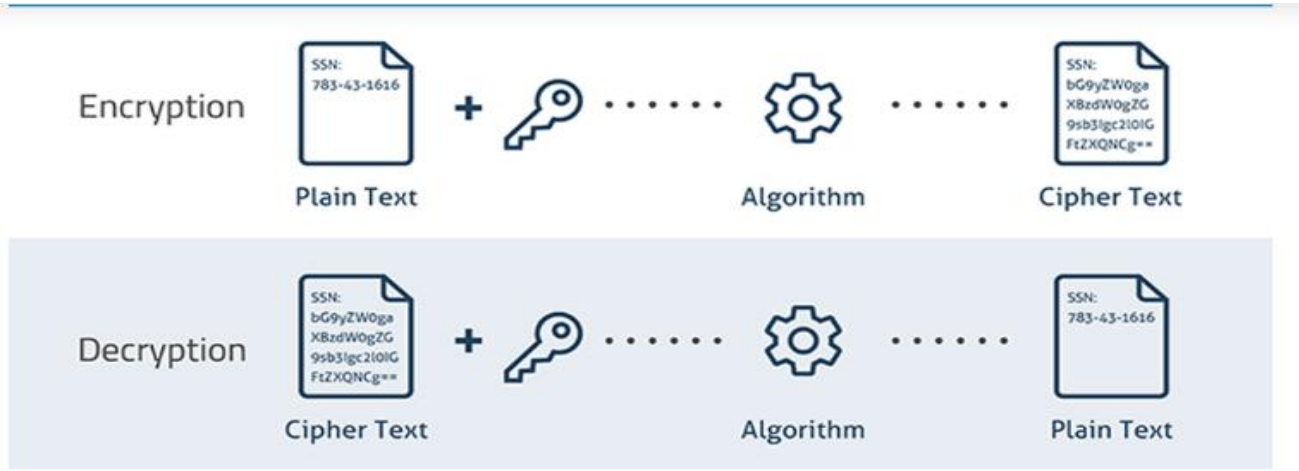
CHAPTER 1

Introduction

1.1 Encryption and Decryption

Encryption and decryption are fundamental components of data security, protecting sensitive information from unauthorized access and ensuring confidentiality, integrity, and authenticity. Encryption transforms readable data (plain text) into an unreadable format (cipher text), while decryption reverses this process, converting cipher text back into plain text. This report explores the definitions, types, advantages, and disadvantages of encryption and decryption.

1.1.1 Definition



Encryption

Encryption is the process of converting plain text into cipher text using an algorithm and an encryption key. The cipher text is not intelligible without the corresponding decryption key. Encryption ensures that data remains confidential and secure from unauthorized access.

Decryption

Decryption is the reverse process of encryption, converting cipher text back into plain text using a decryption key. Only authorized users with the correct decryption key can access the original information.

1.1.2 Types

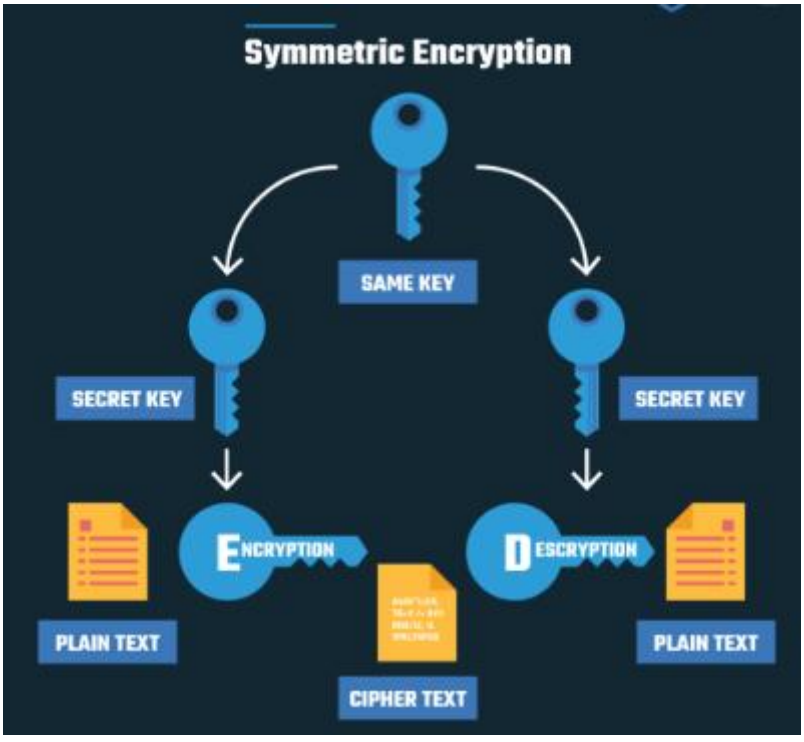
1.1.2.1 Symmetric

Symmetric cryptography, also known as symmetric-key cryptography or private-key cryptography, is a cryptography system where the same key is used for both encryption and decryption of messages. This key must be kept secret between the communicating parties to maintain the confidentiality of the information.

Encryption: The process of converting plaintext (readable data) into ciphertext (unreadable data) using a secret key.

Decryption: The process of converting ciphertext back to plaintext using the same secret key.

Key: A secret value known only to the communicating parties. The security of the symmetric cryptography system relies heavily on the secrecy of this key.



Types of Symmetric

1. Block Ciphers

Encrypts data in fixed-size blocks (e.g., 64-bit or 128-bit blocks). Each block of plain text is transformed into a block of cipher text using the secret key.

Example:

DES

DES (Data Encryption Standard): DES (Data Encryption Standard) is a symmetric encryption algorithm that encrypts and decrypts data using the same key. It operates on data blocks of 64 bits, using a 56-bit key to perform a series of complex transformations on each block.

Steps for DES Encryption

1. **Initial Permutation (IP)**

   The 64-bit plain text block undergoes an initial permutation (IP), which rearranges the bits according to a predefined table.

2. **Key Schedule Generation**

   The 56-bit key is divided into two 28-bit halves.

   These halves are subjected to left rotations and combined to produce 16 sub keys, each 48 bits in length, used in each of the 16 rounds of encryption.

3. **16 Rounds of Feistel Function**

   Round Function: Each of the 16 rounds uses a Feisty structure where the block is split into left (L) and right (R) halves.

   Expansion: The 32-bit right half is expanded to 48 bits using an expansion table.

   Key Mixing: The expanded right half is Xor ed with the 48-bit sub-key for the current round.

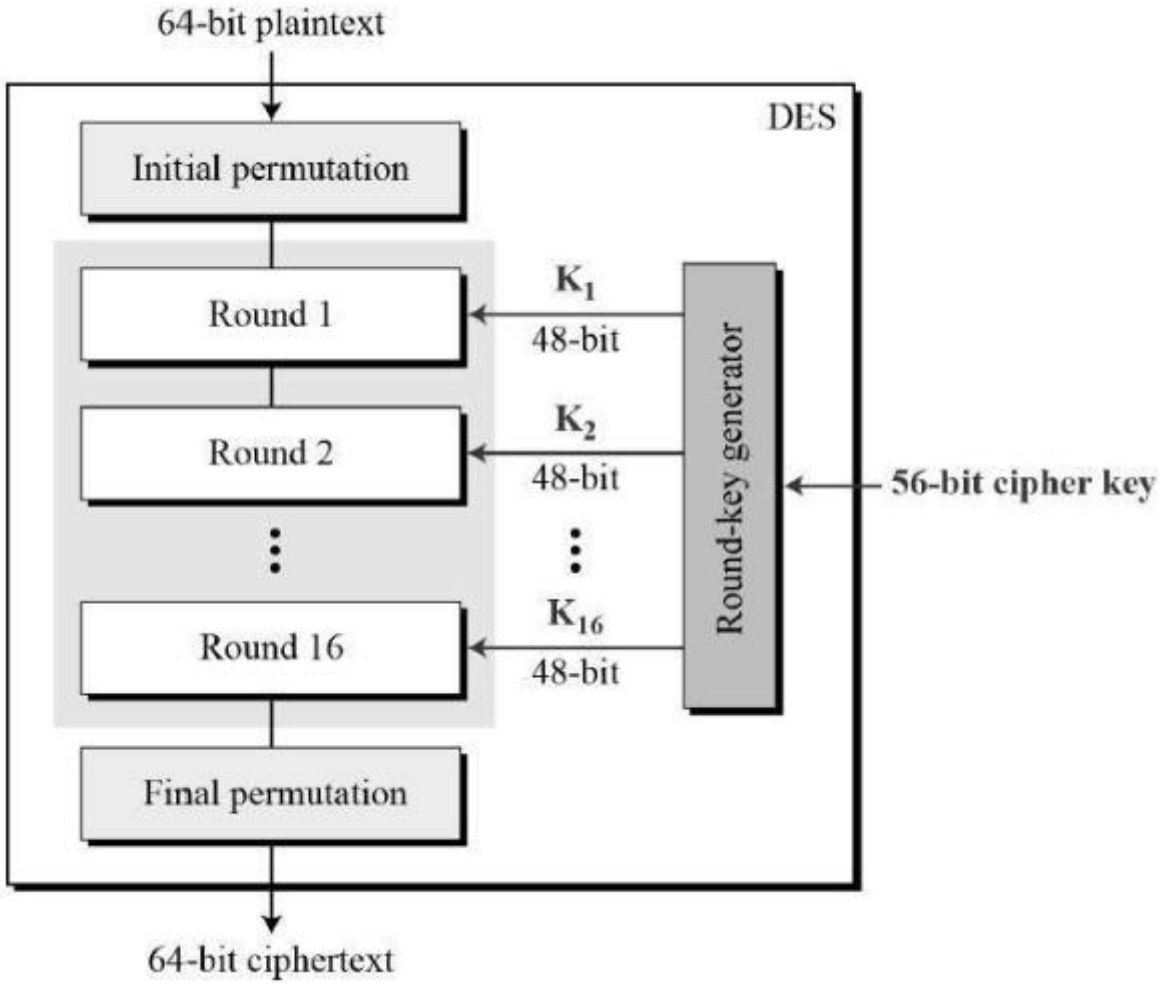
Substitution (S-Boxes): The result is divided into eight 6-bit blocks and passed through eight different S-boxes, each producing a 4-bit output, resulting in a 32-bit block.

Permutation (P-Box): The 32-bit block from the S-boxes is permuted using a P-box.

XOR and Swap: The permuted output is Xor ed with the left half, and the halves are swapped (except in the final round).

4. Final Permutation (FP)

After completing all 16 rounds, the final permutation (FP) is applied to the concatenated left and right halves to produce the 64-bit cipher text.



Steps for DES Decryption

1. **Initial Permutation (IP)**

   The 64-bit cipher text block undergoes the initial permutation (IP), rearranging the bits according to the same table used in encryption.

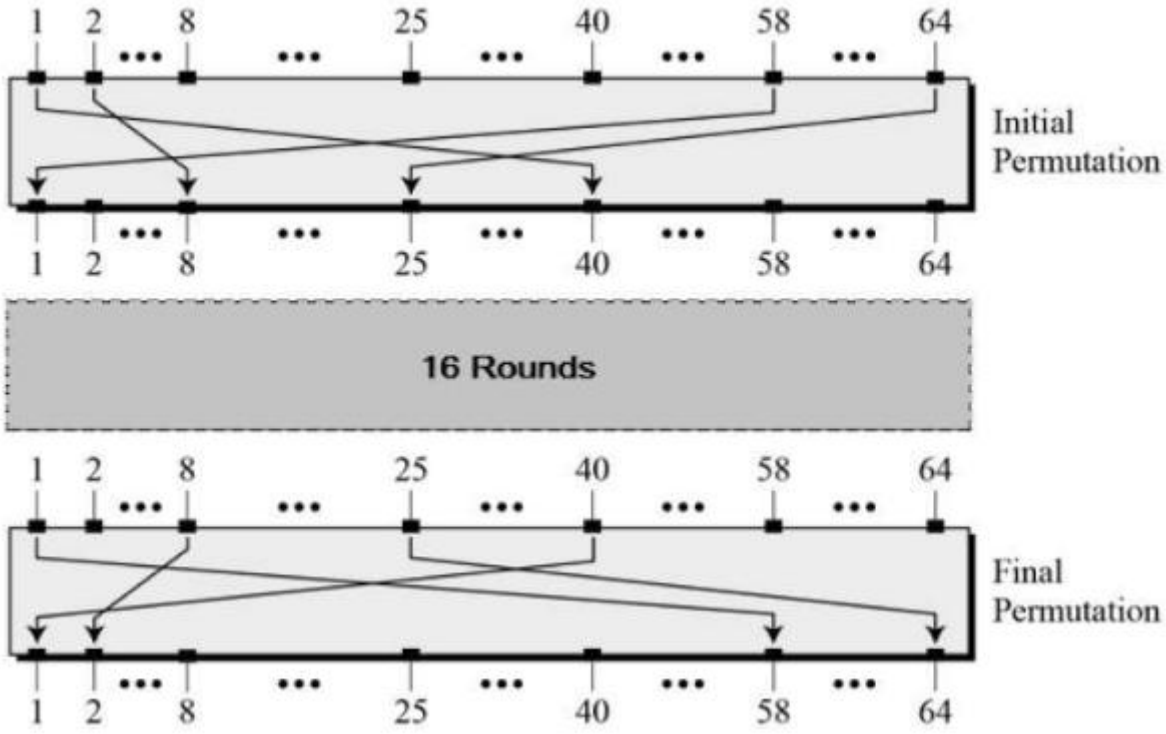
2. **16 Rounds of Inverse Feistel Function**

   Round Function: Each round uses the inverse of the Feisty structure, following the same steps as encryption but with the sub keys applied in reverse order (from round 16 to round 1).

   Expansion, Key Mixing, Substitution, and Permutation: These steps are identical to those in the encryption process, ensuring the correct transformation of the data.

3. **Final Permutation (FP)**

   The final permutation (FP) is applied to the concatenated left and right halves, producing the original 64-bit plain text block.



Example of DES Encryption and Decryption

- Plain text: "0123456789ABCDEF" (in hexadecimal)
- Key: "133457799BBCDFF1" (in hexadecimal)

Encryption Process

1. **Initial Permutation (IP)**
   - Input: "0123456789ABCDEF"
   - Output: Permuted block after applying IP table.
2. **Key Schedule Generation**
   - Key: "133457799BBCDFF1"
   - Sub keys: 16 sub keys generated for each round.
3. **16 Rounds of Feistel Function**

   For each round: Expansion, key mixing, substitution (using S-boxes), permutation, and XOR operations are performed.
4. **Final Permutation (FP)**
   - Output: Cipher text "85E813540F0AB405" (in hexadecimal).

Decryption Process

1. **Initial Permutation (IP)**
   - Input: "85E813540F0AB405"
   - Output: Permuted block after applying IP table.
2. **16 Rounds of Inverse Feistel Function**
   - For each round (subkeys applied in reverse order): Expansion, key mixing, substitution (using S-boxes), permutation, and XOR operations are performed.
3. **Final Permutation (FP)**
   - Output: Original plaintext "0123456789ABCDEF".

Advantages and Disadvantages of DES

Advantages

1. **Historical Significance**

   DES set a foundation for modern cryptographic algorithms and practices.

2. **Simplicity and Speed**

   Simple design and relatively fast execution in hardware.

Disadvantages

1. **Security Vulnerabilities**

   The 56-bit key length is too short by modern standards, making it susceptible to brute-force attacks.

2. **Obsolescence**

   DES has been replaced by more secure algorithms like AES due to its vulnerabilities.

2. Stream Ciphers

Encrypts data one bit or one byte at a time, often using a key stream generated from the secret key.

Examples:

RC4: A widely used stream cipher known for its simplicity and speed. However, vulnerabilities have been discovered, making it less secure for some applications.

ChaCha20: A modern stream cipher designed to be secure and efficient, providing better performance and security than older ciphers like RC4.

MATLAB Example for Stream Cipher

- Read the Image: Load the image into MATLAB.
- Generate Key Stream: Create a pseudo-random key stream.
- Encrypt the Image: XOR the image with the key stream.
- Decrypt the Image: XOR the encrypted image with the same keystream to retrieve the original image.

Examples of Symmetric Cryptography Applications

1. Data Encryption:

File Encryption: Encrypting files on a disk to protect sensitive information. Tools like AES are commonly used for this purpose.

Disk Encryption: Encrypting entire disk drives to protect data from unauthorized access. Examples include BitLocker (Windows) and FileVault (macOS).

2. Network Security:

SSL/TLS: Protocols that use symmetric encryption (along with asymmetric encryption) to secure communications over the internet. AES is often used in TLS for encrypting data during transmission.

VPNs (Virtual Private Networks): Use symmetric encryption to secure data transmitted between remote users and private networks.

3.Secure Messaging:

Applications like WhatsApp and Signal use symmetric encryption to secure messages between users. Typically, symmetric encryption is combined with asymmetric encryption for key exchange.

Advantages and Disadvantages

Advantages of Symmetric Cryptography

I. **Speed:** Symmetric encryption algorithms are generally faster than their asymmetric counterparts, making them suitable for encrypting large amounts of data.
II. **Efficiency:** Requires less computational power, making symmetric cryptography ideal for resource-constrained devices like smartphones and IoT devices.
III. **Simplicity:** Easier to implement and understand compared to asymmetric cryptography.

Disadvantages of Symmetric Cryptography

1. **Key Distribution:** Securely sharing and managing the secret key between communicating parties can be challenging, especially over untrusted networks.
2. **Scalability:** In a system with n users, $n(n-1)/2$ keys are needed for secure pairwise communication, which can become impractical for large networks.
3. **Key Management:** Requires secure storage and management of keys. If the key is compromised, all communications encrypted with that key are at risk.

1.1.2.2 Asymmetric

Asymmetric cryptography, also known as public-key cryptography, is a cryptographic system that uses pairs of keys: a public key, which is disseminated widely, and a private key, which is kept secret. This approach allows for secure communication and data exchange even over untrusted networks.

Key Concepts

Public Key:

A key that can be distributed openly and used by anyone to encrypt a message or verify a digital signature.

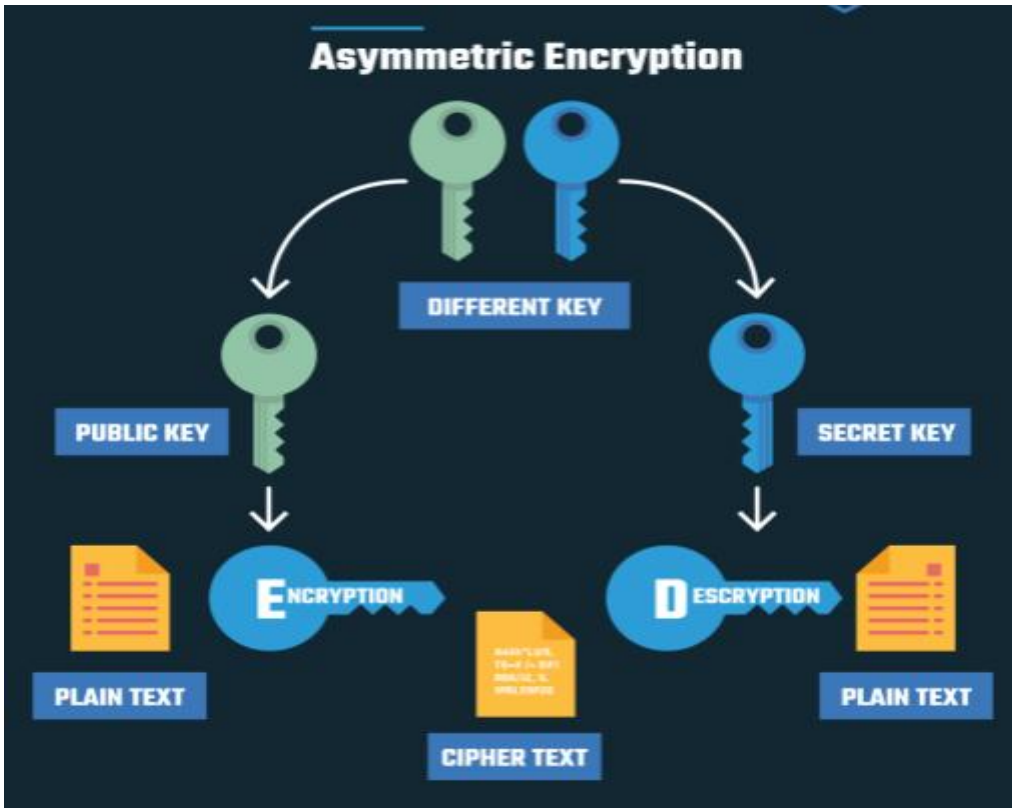
Private Key:

A key that is kept secret by the owner and used to decrypt messages or create digital signatures.

Encryption and Decryption:

Encryption: The process of converting plain text into cipher text using the recipient's public key.

Decryption: The process of converting cipher text back into plain text using the recipient's private key.



Types of Asymmetric Cryptography

1. **RSA (Rivest-Shamir-Adleman):**

   One of the first public-key cryptosystems and widely used for secure data transmission.

   Based on the computational difficulty of factoring large integers.

   Commonly used for securing web traffic (SSL/TLS), email encryption, and digital signatures.

2. **ECC (Elliptic Curve Cryptography):**

   Uses the mathematics of elliptic curves to provide security.

   Offers the same level of security as RSA but with shorter key lengths, leading to faster computations and reduced resource usage.

   Increasingly used in mobile devices, IoT applications, and for securing cryptocurrency transactions.

3. **Diffie-Hellman Key Exchange:**

   A method of securely exchanging cryptographic keys over a public channel.

   Allows two parties to establish a shared secret key, which can then be used for symmetric encryption.

   Not used for encryption or digital signatures directly but as a secure way to negotiate symmetric keys.

RSA

RSA (Rivest-Shamir-Adleman) is a widely used public-key cryptosystem that enables secure communication and digital signatures over insecure channels. The security of RSA is based on the practical difficulty of factoring the product of two large prime numbers, the factoring problem.

Key Generation:

1. Choose two large prime numbers,p and q.
2. Compute $n=p*q$. n is used as the modulus for both the public and private keys.
3. Calculate $\phi(n)=(p-1)(q-1)$, where ϕ is Euler's totient function.
4. Choose an integer e such that $1<e<\phi(n)$, and e is coprime (has no common factors other than 1) with $\phi(n)$. This is the public exponent.
5. Compute d, the modular multiplicative inverse of e modulo $\phi(n)$. In other words, $d=e^{-1}(\mod\phi(n))$. This is the private exponent.
6. The public key is(n,e), and the private key is (n,d).

Encryption:

1. Represent the plain text message as an integer m, where $0 \leq m < n$.
2. Compute the cipher text $C = M^e \pmod n$.

Decryption:

1. Use the private key (n,d).
2. Compute the original message $M = C^d \pmod n$.

Key Generation by Alice	
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$
Calculate d	$d \equiv e^{-1} \pmod{\phi(n)}$
Public key	$PU = \{e, n\}$
Private key	$PR = \{d, n\}$

Encryption by Bob with Alice's Public Key	
Plaintext:	$M < n$
Ciphertext:	$C = M^e \bmod n$

Decryption by Alice with Alice's Private Key	
Ciphertext:	C
Plaintext:	$M = C^d \bmod n$

p, q, two prime numbers (private, chosen)
$n = pq$ (public, calculated)
e, with $\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$ (public, chosen)
$d \equiv e^{-1} \pmod{\phi(n)}$ (private, calculated)

Ex1:

1. Select two prime numbers, $p = 17$ and $q = 11$.
2. Calculate $n = pq = 17 \times 11 = 187$.
3. Calculate $\phi(n) = (p-1)(q-1) = 16 \times 10 = 160$.
4. Select e such that e is relatively prime to $\phi(n) = 160$ and less than $\phi(n)$; we choose $e = 7$.
5. Determine d such that $de \equiv 1 \pmod{160}$ and $d < 160$. The correct value is $d = 23$, because $23 \times 7 = 161 = (1 \times 160) + 1$; d can be calculated using the extended Euclid's algorithm (Chapter 2).

The resulting keys are public key $PU = \{7, 187\}$ and private key $PR = \{23, 187\}$. The example shows the use of these keys for a plaintext input of $M = 88$. For encryption, we need to calculate $C = 88^7 \bmod 187$. Exploiting the properties of modular arithmetic, we can do this as follows.

$$88^7 \bmod 187 = [(88^4 \bmod 187) \times (88^2 \bmod 187) \times (88^1 \bmod 187)] \bmod 187$$

$$88^1 \bmod 187 = 88$$

$$88^2 \bmod 187 = 7744 \bmod 187 = 77$$

$$88^4 \bmod 187 = 59{,}969{,}536 \bmod 187 = 132$$

$$88^7 \bmod 187 = (88 \times 77 \times 132) \bmod 187 = 894{,}432 \bmod 187 = 11$$

For decryption, we calculate $M = 11^{23} \bmod 187$:

$$11^{23} \bmod 187 = [(11^1 \bmod 187) \times (11^2 \bmod 187) \times (11^4 \bmod 187) \times (11^8 \bmod 187) \times (11^8 \bmod 187)] \bmod 187$$

$$11^1 \bmod 187 = 11$$

$$11^2 \bmod 187 = 121$$

$$11^4 \bmod 187 = 14{,}641 \bmod 187 = 55$$

$$11^8 \bmod 187 = 214{,}358{,}881 \bmod 187 = 33$$

$$11^{23} \bmod 187 = (11 \times 121 \times 55 \times 33 \times 33) \bmod 187 = 79{,}720{,}245 \bmod 187 = 88$$

In MATLAB

```
Original Message:
    88

Ciphertext:
    11

Decrypted Message:
    88
```

Advantages:

1. Security Based on Mathematical Complexity:

The security of RSA is based on the difficulty of factoring the product of two large prime numbers. The larger the key size, the more computationally difficult it becomes to factorize, making it a robust encryption algorithm.

2. **Public and Private Key Pairs:**

RSA uses a pair of keys: public key for encryption and private key for decryption. The public key can be freely distributed, allowing anyone to encrypt messages for the owner of the private key. This facilitates secure communication in a scalable way.

3. **Digital Signatures:**

RSA is commonly used for digital signatures, providing a way to verify the authenticity and integrity of messages. The private key is used to create the signature, and the public key is used to verify it.

4. **Key Exchange in Public-Key Infrastructure (PKI):**

RSA is widely used for secure key exchange in Public-Key Infrastructure, enabling secure communication over untrusted networks, such as the internet.

5. **Versatility:**

RSA can be used for both encryption and digital signatures, making it versatile for various security applications.

Disadvantages:

1. **Computational Intensity:**

RSA is computationally intensive, especially for large key sizes. The time required for key generation, encryption, and decryption increases significantly with larger key sizes, making it less efficient than symmetric-key algorithms for encrypting large amounts of data.

2. **Key Size Consideration:**

As computing power increases, the recommended key sizes for RSA also need to increase to maintain security. Larger key sizes result in slower operations and increased demand for computational resources.

3. **Key Management:**

Managing and securely distributing public keys can be challenging in large-scale systems. Public-key infrastructure (PKI) is often used to address key distribution and management issues.

4. **Vulnerability to Quantum Computing:**

RSA is potentially vulnerable to attacks using quantum computers. Shor's algorithm, when implemented on a sufficiently powerful quantum computer, could efficiently factorize large numbers, compromising the security of RSA. Post-quantum cryptography solutions are being explored to address this potential vulnerability.

Examples of Asymmetric Cryptography Applications

I. **Secure Web Browsing (HTTPS):**

SSL/TLS protocols use asymmetric cryptography to establish a secure connection between a client and a server.

The server's public key is used to encrypt data sent from the client, while the server uses its private key to decrypt the data.

II. **Email Security (PGP/GPG):**

PGP (Pretty Good Privacy) and GPG (GNU Privacy Guard) use asymmetric encryption to secure emails.

The sender encrypts the email with the recipient's public key, and the recipient decrypts it with their private key.

III. **Cryptocurrency Transactions:**

Cryptocurrencies like Bitcoin and Ethereum use asymmetric cryptography for securing transactions.

Each user has a pair of keys: a public key (address) and a private key (used to sign transactions).

IV. Digital Signatures:

Used in software distribution, financial transactions, and legal documents to ensure the authenticity and integrity of the data.

The sender signs the data with their private key, and the recipient verifies the signature with the sender's public key.

Advantages and Disadvantages

Advantages of Asymmetric Cryptography

1. **Security:** Eliminates the need to share private keys. Only public keys need to be distributed, reducing the risk of key compromise.
2. **Scalability:** Easier to manage in large systems compared to symmetric cryptography, as each pair of users only needs to know their own private key and the public keys of others.
3. **Digital Signatures:** Provides a way to authenticate and verify the integrity of messages and documents.

Disadvantages of Asymmetric Cryptography

A. **Performance:** Generally slower than symmetric cryptography due to the complexity of mathematical operations involved.

B. **Key Length:** Requires larger key sizes to achieve the same level of security as symmetric algorithms, leading to higher computational overhead.

C. **Complexity:** More complex to implement and manage compared to symmetric cryptography.

1.2 Watermark

Watermarks have been used for centuries to protect the authenticity and ownership of documents, images, and other media. In the digital age, watermarks play a crucial role in safeguarding intellectual property and ensuring the integrity of digital content. This report explores the definitions, types, advantages, and disadvantages of watermarks.

1.2.1 Definition

A watermark is a recognizable pattern or image embedded into a document, photograph, video, or other media, which can be seen or detected to verify authenticity and ownership. In digital contexts, watermarks can be visible or invisible, ensuring the protection of digital assets against unauthorized use or duplication.

1.2.2 Types

Types of Watermarks

1. **Visible Watermarks**

   Definition: Clearly noticeable marks, logos, or text superimposed on an image or document.

   Examples: A company's logo placed on a promotional image, a "Confidential" stamp on a document.

   Applications: Used in photographs, documents, videos to denote ownership or sensitive information.

2. **Invisible Watermarks**

   Definition: Embedded information that is not visible to the naked eye but can be detected using special software.

Examples: Digital signatures, embedded metadata.

Applications: Used for copyright protection, digital rights management (DRM), and tracking unauthorized distribution.

3. **Fragile Watermarks**

Definition: Watermarks that are easily destroyed or altered if the content is modified.

Examples: Authentication marks on sensitive documents.

Applications: Used to detect tampering or unauthorized modifications.

4. **Robust Watermarks**

Definition: Watermarks designed to withstand various transformations, such as compression, re-sizing, and cropping.

Examples: Watermarks on images or videos that remain intact even after editing.

Applications: Used in media files to ensure ongoing protection against unauthorized use.

5. **Spatial Watermarks**

Definition: Watermarks applied directly to the spatial domain of an image.

Examples: Text or logos overlaid on the image.

Applications: Commonly used in visible watermarking of images and videos.

6. **Frequency Domain Watermarks**

Definition: Watermarks embedded into the frequency domain of the content, often making them less perceptible.

Examples: Watermarks embedded using techniques like Discrete Cosine Transform (DCT) or Discrete Wavelet Transform (DWT).

Applications: Used for invisible watermarking in audio, image, and video files.

1.2.3 Advantages and Disadvantages

Advantages of Watermarks

1. **Protection of Intellectual Property** Watermarks help assert ownership and protect against unauthorized use or duplication of digital assets.

2. **Deterrence of Unauthorized Use**
   Visible watermarks can discourage unauthorized copying or distribution by making it clear that the content is protected.
3. **Verification of Authenticity**
   Watermarks can verify the authenticity of documents, images, and other media, ensuring that they have not been altered or tampered with.
4. **Tracking and Monitoring**
   Invisible watermarks can track the distribution and use of digital content, helping to monitor for unauthorized use.
5. **Brand Recognition**
   Watermarks can serve as a branding tool, promoting brand recognition and visibility in images and videos.

Disadvantages of Watermarks

1. **Quality Degradation**
   Visible watermarks can detract from the visual quality and aesthetic of images and videos, potentially reducing their usability.
2. **Complexity of Implementation**
   Implementing invisible or robust watermarks can be technically complex and may require specialized software and expertise.
3. **Vulnerability to Removal**
   Some watermarks, especially visible ones, can be removed or altered by skilled individuals using editing tools.
4. **Impact on User Experience** Watermarks may negatively impact user experience, particularly if they are overly intrusive or detract from the content.

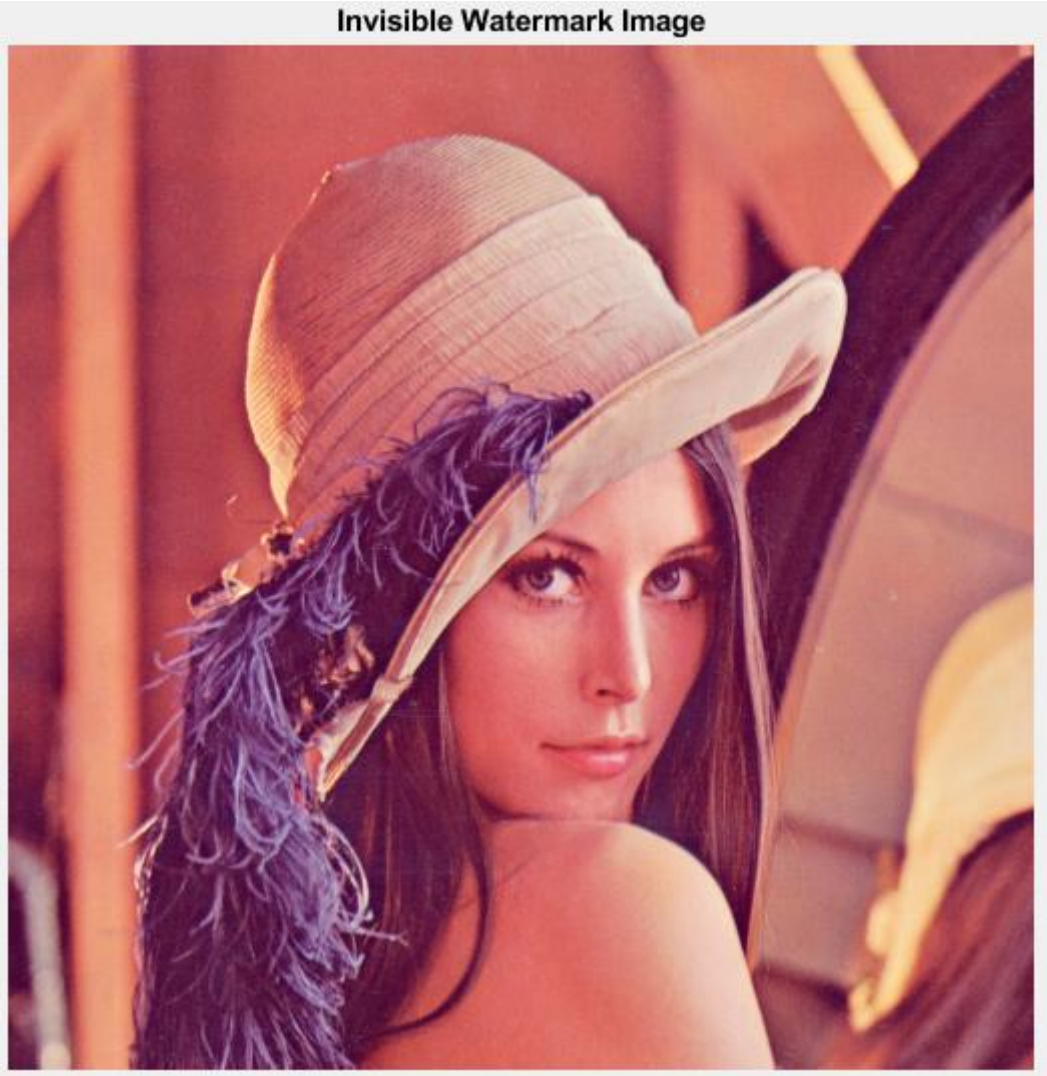
1.2.4 Examples

MATLAB Code for Embedding a Visible Watermark

1. **Load the Host Image:** Load the image into which you want to embed the watermark.
2. **Create the Watermark Text:** Generate an image containing the watermark text.
3. **Embed the Watermark:** Overlay the watermark text image onto the host image.

Embedding an Invisible Watermark using LSB

1. **Load the Host Image:** Load the image into which you want to embed the watermark.
2. **Generate the Watermark:** Create a binary watermark (e.g., a logo or text converted to a binary image).
3. **Embed the Watermark:** Embed the watermark using LSB
4. **Extract the Watermark:** Extract the watermark using inverse LSB



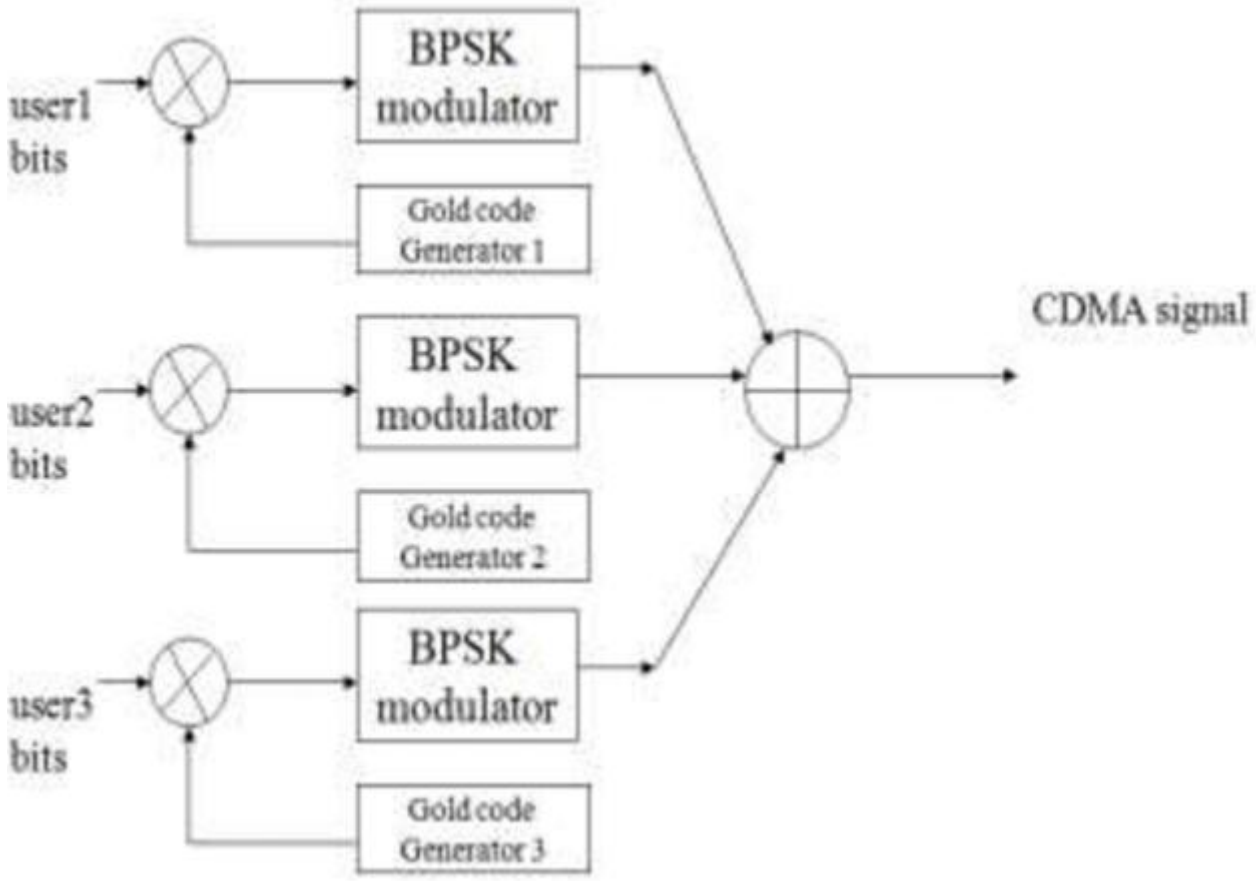
Extract Watermark text

```
Watermark_Text =

    'Hidden text'
```

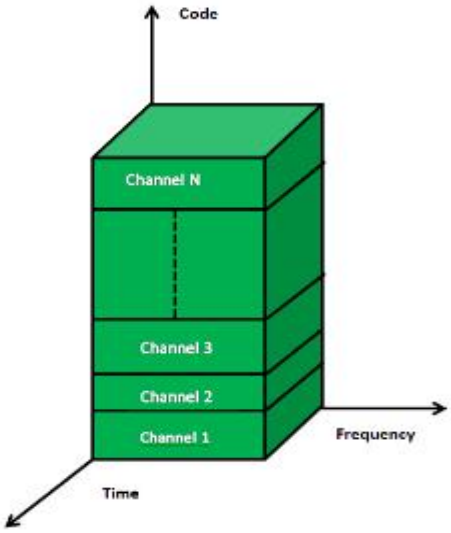
1.3 Security techniques based on spread spectrum techniques

1.3.1 Introduction

We use Code Division Multiple Access (CDMA), which is a channel access method used by various radio communication technologies. It allows multiple signals to occupy a single transmission channel, optimizing the use of available bandwidth. CDMA is a form of multiplexing, which means it allows multiple signals to be combined for transmission over a single communications channel and then separated at the receiver end.

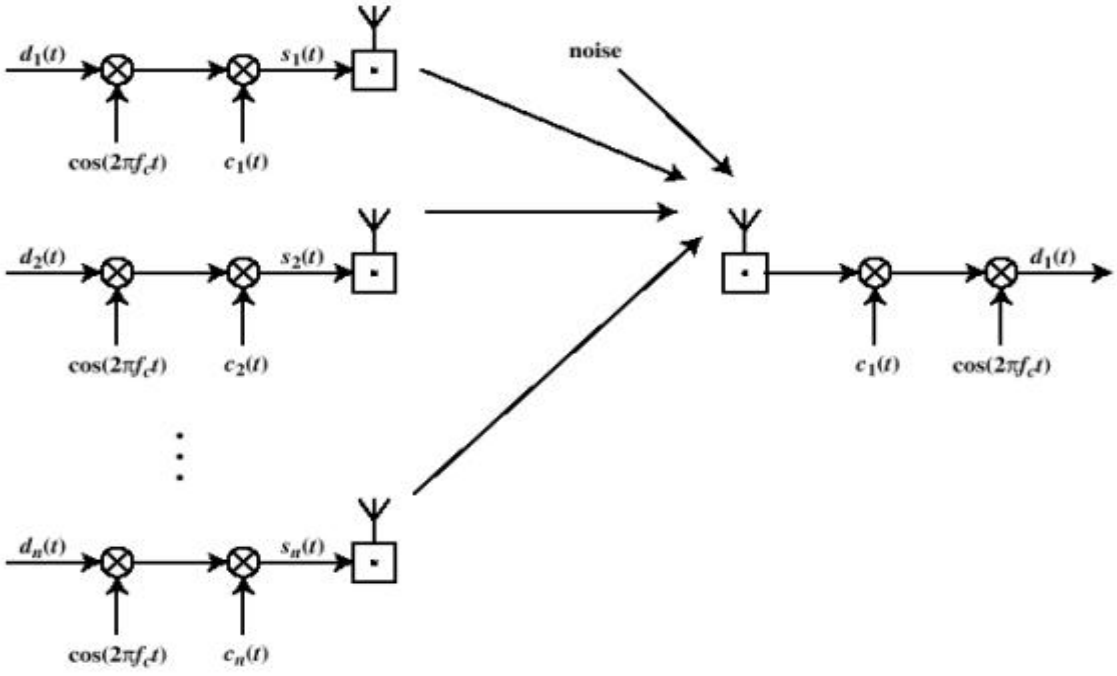


CDMA works by assigning a unique code to each communication signal. These codes are used to modulate the signals, allowing them to be transmitted simultaneously over the same frequency spectrum without interference. At the receiver end, the same unique codes are used to demodulate and separate the signals.



1.3.2 Direct Sequence CDMA (DS-CDMA)

Direct Sequence Code Division Multiple Access (DS-CDMA) is a multiple access technique used in telecommunications that allows multiple users to share the same frequency spectrum simultaneously by spreading each user's signal over a wider bandwidth. This is achieved by multiplying the user's data signal with a high-rate pseudo random sequence (spreading code) unique to each user. The spreading code increases the bandwidth of the transmitted signal, making it more resistant to interference and eavesdropping.



Direct Sequence Spread Spectrum (DSSS)

- Each bit in original signal is represented by multiple bits in the transmitted signal
- Spreading code spreads signal across a wider frequency band Spread is in direct proportion to number of bits used

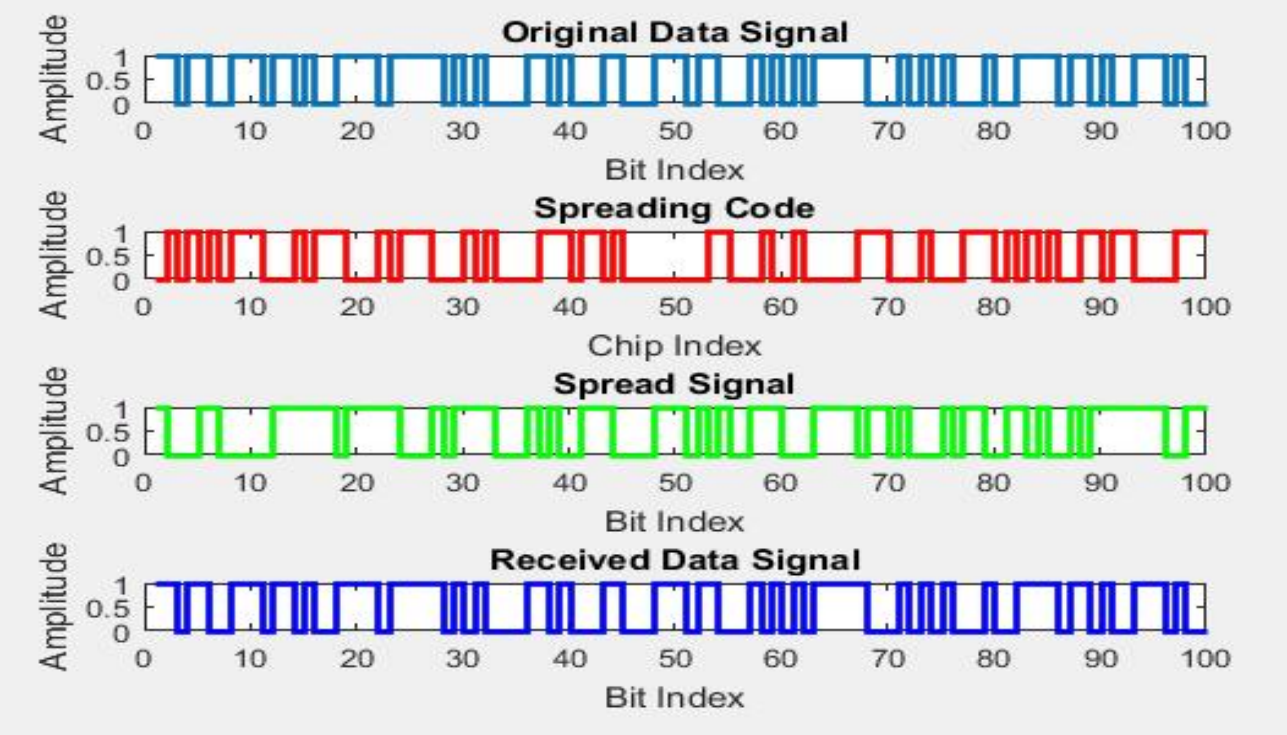
  R_b=bit rate of Data input (bit/sec)

  R_c=chip rate of PN bit stream (chip/sec)

  $R_c >> R_b$ to spread Data input

  Spreading Gain=$R_c/R_b >> 1$
- One technique combines digital information stream with the spreading code bit stream using exclusive-OR
- At receiver we use exclusive-OR between spread signal and spreading code to get original signal at receiver

By MATLAB



Data recovery successful. Received data matches the original.

Advantages of DSSS

1. Best behaviour for multipath rejection

2. Best anti-jam rejection
3. Best interference rejection
4. Simple implementation
5. Most difficult to detect
6. No synchronization among terminals

Disadvantages of DSSS

1. Require coherent bandwidth (Wide band channel)
2. Long acquisition time

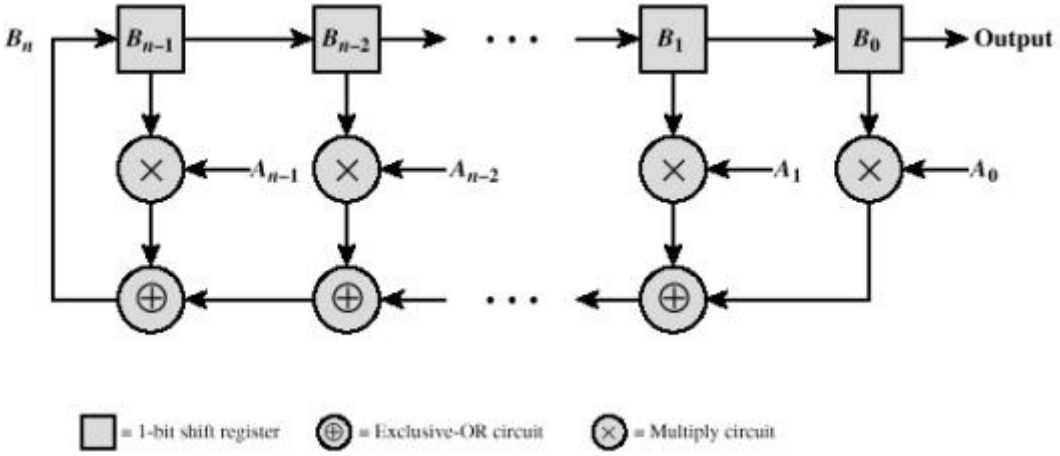
Types of Spread Spectrum Code

1.3.2.1 PN code :

M-sequences, also known as maximum length sequences or pseudorandom sequences, are binary sequences generated using linear feedback shift registers (LFSRs). These sequences have the property that they are maximum length, meaning they have the longest possible period for a shift register of a given length. M-sequences find applications in various areas, including telecommunications, cryptography, and spread spectrum systems.

Linear Feedback Shift Register (LFSR):

An LFSR is a shift register whose input bit is a linear function of its previous state. The shift register has a feedback mechanism that introduces new bits based on XORing (exclusive OR) certain taps of the register.



Connection Polynomial:

The connection polynomial of the LFSR determines the feedback taps. It is usually expressed as a binary polynomial. For example, for a 4-bit LFSR with feedback taps at positions 4 and 1, the connection polynomial would be x^4+x+1, where x represents the variable in the polynomial.

Initialization:

The LFSR is initialized with a seed value (an initial state). The seed value determines the starting point of the sequence.

Sequence Generation:

The LFSR is clocked, and the output bit is taken as the sequence bit. The state of the LFSR is updated at each clock cycle based on the connection polynomial, has 2^{n-1} ones and 2^{n-1}-1 zeros.

Have 2n/2 (number of groups of 1 bit)

Periodicity:

M-sequences generated by LFSRs exhibit maximum length, meaning the sequence repeats only after 2^n-1 clock cycles, where n is the number of bits in the LFSR, except for all zeros sequence

The periodic autocorrelation of a ±1 m-sequence is

$$R(\tau) = \begin{cases} 1 & \tau = 0, N, 2N, \ldots \\ -\frac{1}{N} & \text{otherwise} \end{cases}$$

Important PN Properties

1- Randomness

Uniform distribution

Balance property

Run property

Independence

Correlation property

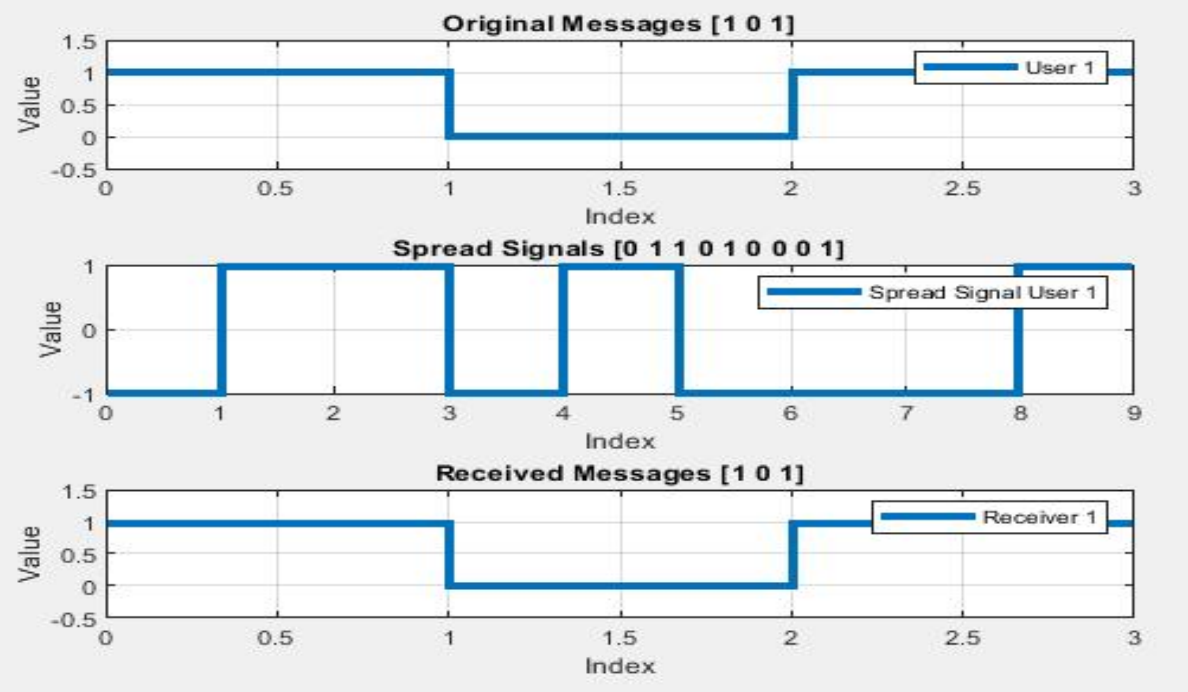
2- Unpredictability

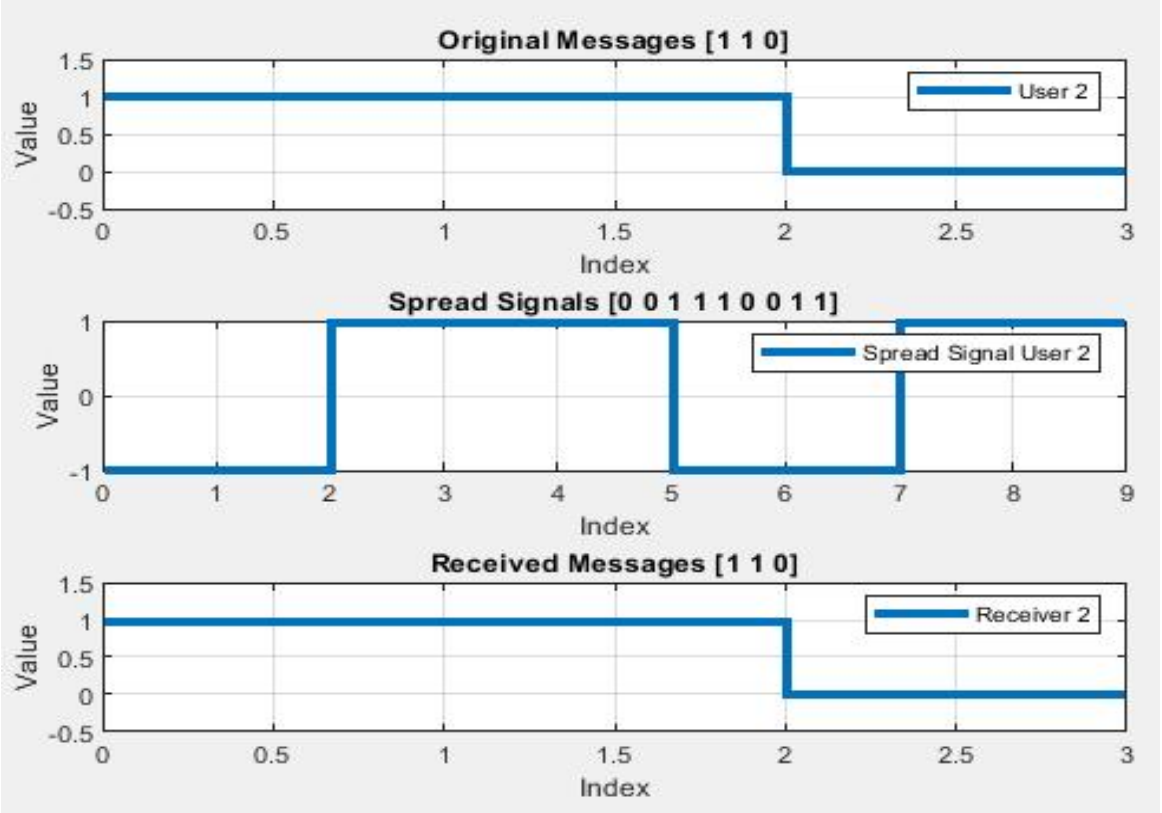
From MATLAB We generate 3 PN Code

- n = 3 (number of bits in the LFSR)
- tap Positions = [3,1] (tap positions for feedback)
- seed = [1, 0, 0] (initial seed for the LFSR)
- PN Code Length = 7

```
PN
  1     1     0     1     0     0     1

  1     1     1     0     1     0     0

  0     1     1     1     0     1     0
```

Ex: have 2 user, u1=101 with c3=0111010, u2=110 with shift right by 1 of c3=0011101, Spreading Gain =3





Cross correlation

The comparison between two sequences from different sources rather than a shifted copy of a sequence with itself

Advantages of Cross Correlation

- The cross correlation between an m-sequence and noise is low
  1. This property is useful to the receiver in filtering out noise
- The cross correlation between two different m-sequences is low
  1. This property is useful for CDMA applications
  2. Enables a receiver to discriminate among spread spectrum signals generated by different m-sequences

Correlation

The concept of determining how much similarity one set of data has with another

Range between –1 and 1

"1" The second sequence matches the first sequence

"0" There is no relation at all between the two sequences

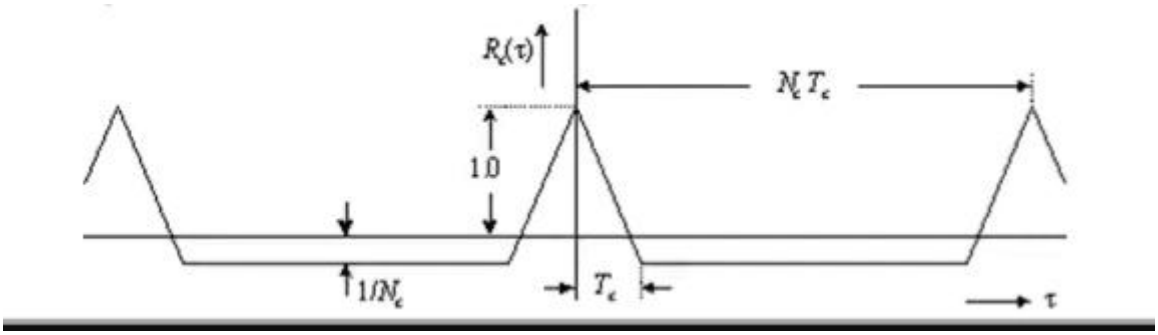
"-1" The two sequences are mirror images

Correlation of a PRN sequence by its shifted version gives low values, except if the shift is 0.

Correlation of a PRN sequence with another PRN sequence gives low values.

Auto correlation

- give high at delay or shift =0 (mean there isn't orthogonality and can't encryption)
- give low value at any delay or shift (mean there is orthogonality and can encryption)



1.3.2.2 Walsh code:

- Walsh code typically refers to a set of binary sequences used in telecommunications and signal processing. These codes are often used in communication systems, such as CDMA (Code Division Multiple Access) in mobile networks.
- A Walsh code is a set of orthogonal binary sequences. These sequences have the property that when multiplied by one another and integrated over a specific interval, the result is zero. In CDMA, each user is assigned a unique Walsh code to distinguish their signals from others in the same frequency band.
- More used in software, Set of Walsh codes of length n consist of the n rows of an n X n Walsh matrix

$W_1 = (0)$ $\qquad W_{2n} = \begin{pmatrix} W_n & W_n \\ W_n & \overline{W_n} \end{pmatrix}$

- Number of rows is number of codes
- Number of columns is length of codes
- Each code is orthogonal to others
- Max Walsh Matrix is 64x64 because after that code isn't orthogonal

By MATLAB

```
Enter first bit of Walsh code= 1
Walsh Matrix:
     1     1     1     1     1     1     1     1
     1     0     1     0     1     0     1     0
     1     1     0     0     1     1     0     0
     1     0     0     1     1     0     0     1
     1     1     1     1     0     0     0     0
     1     0     1     0     0     1     0     1
     1     1     0     0     0     0     1     1
     1     0     0     1     0     1     1     0
```

```
Enter first bit of Walsh code= 0
Walsh Matrix:
     0     0     0     0     0     0     0     0
     0     1     0     1     0     1     0     1
     0     0     1     1     0     0     1     1
     0     1     1     0     0     1     1     0
     0     0     0     0     1     1     1     1
     0     1     0     1     1     0     1     0
     0     0     1     1     1     1     0     0
     0     1     1     0     1     0     0     1
```

- Spread Gain<=size of matrix

I. If we have 64x64 and Spread Gain=32, we have 64 users in half of BW so we can double the users in here BW and size of data (channel capacity) decrease but we increase the number of users (System capacity).

II. If we have 32x32 and Spread Gain=64, we have 32 users in double of BW so each users take 2 channel in here we increase channel capacity and data rate.

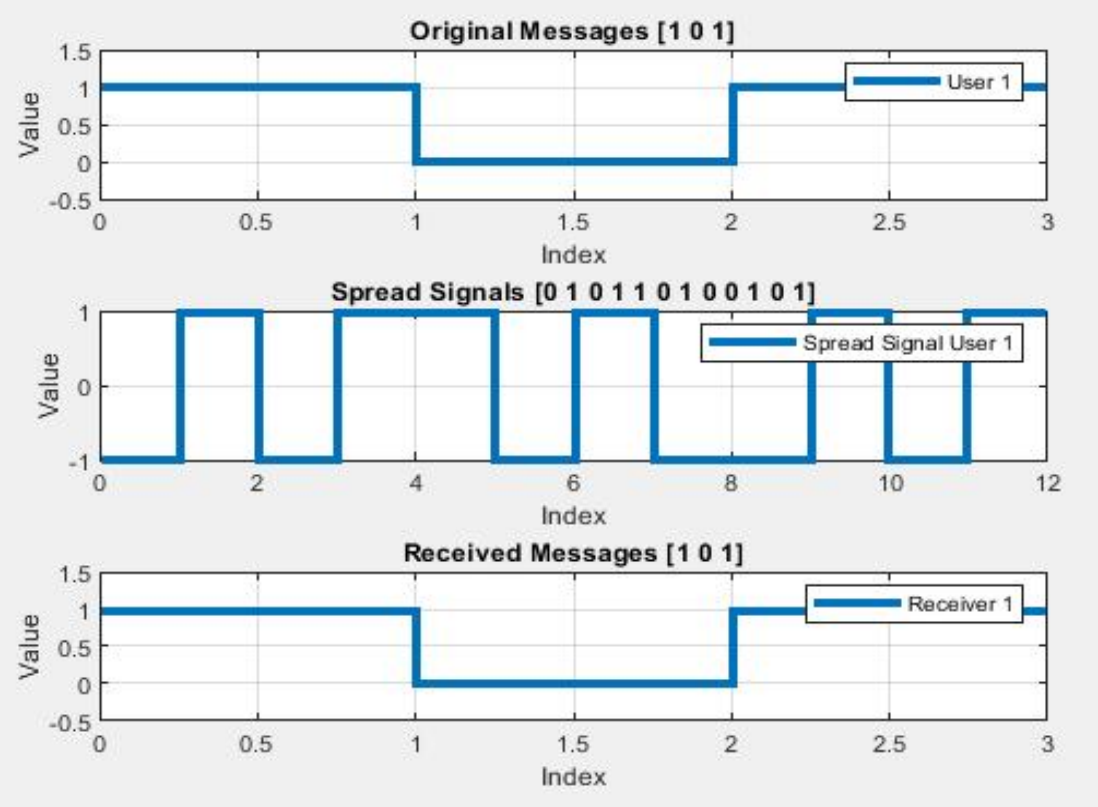
Ex: Generate Walsh code w1= (0) at 8x8, if we have spreading gain=4 and use CDMA And have 2 user, u1=101 with c1, u2=110 with c7

From MATLAB

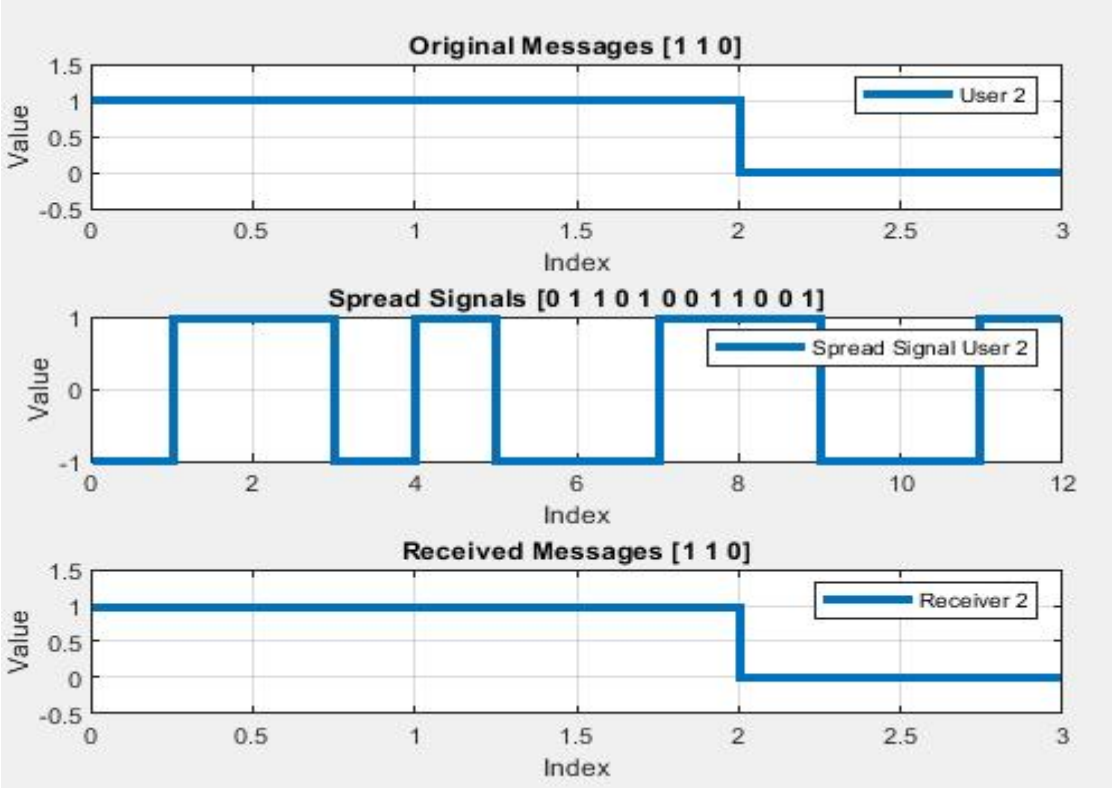
```
Enter first bit of Walsh code= 0
Walsh Matrix:
     0     0     0     0     0     0     0     0
     0     1     0     1     0     1     0     1
     0     0     1     1     0     0     1     1
     0     1     1     0     0     1     1     0
     0     0     0     0     1     1     1     1
     0     1     0     1     1     0     1     0
     0     0     1     1     1     1     0     0
     0     1     1     0     1     0     0     1
```

```
walshCodeUser1 =

     0     1     0     1     0     1     0     1

walshCodeUser2 =

     0     1     1     0     1     0     0     1
```

User 1



User 2



Comparison Between PN code and Walsh code

PN (Pseudo-Noise) codes and Walsh codes are used in telecommunications and digital communications, but they serve different purposes and have distinct characteristics.

	PN code	**Walsh code**
Purpose	**Purpose:** Used for spreading spectrum communication, including CDMA (Code Division Multiple Access) systems. **Functionality:** PN codes spread the signal across a wide frequency band, providing	**Purpose:** Primarily used in the context of multiple access techniques, such as in CDMA systems and orthogonal codes. **Functionality:** Walsh codes are used for channelization and

	multiple users with the ability to share the same frequency spectrum without interference. **Example:** Barker codes, Gold codes, Kasami codes.	orthogonalization of signals in CDMAsystems, allowing multiple users to share the same frequency band without interference. **Example:**Walsh-Hadamard codes.
Sequence Properties	**Generation:** Generated by feedback shift registers or mathematical algorithms. **Periodicity:** Has a long period, and the autocorrelation properties are important for minimizing interference between users. **Correlation:** Cross-correlation properties are crucial for distinguishing between different codes and users.	**Generation:** Formed using Hadamard matrices or Walsh matrices. **Periodicity:** Walsh codes have a fixed length and are usually of length 2^n, where n is an integer. **Orthogonality:** Walsh codes are orthogonal to each other, making them suitable for use in orthogonal multiple access systems.

Application	● Widely used in spread spectrum communication systems, such as CDMA. ● Provides security and robustness against interference. ● Global Positioning System (GPS) ● Cryptography ● Wireless Local Area Networks (WLANs) ● Radio Frequency Identification (RFID)	● Commonly used in CDMA systems for channelization and multiple access. ● Provides orthogonalization of channels, enabling multiple users to share the same frequency band. ● Wireless Communication Systems ● Satellite Communication ● Telecommunications Switching Systems ● Signal Processing and Filtering
Complexity	Generation and synchronization of PN codes can be computationally intensive. The implementation may involve more complex algorithms	Generation is usually simpler, involving the use of matrices. Less computationally intensive compared to some PN code implementations.

1.3.2.3 Gold code

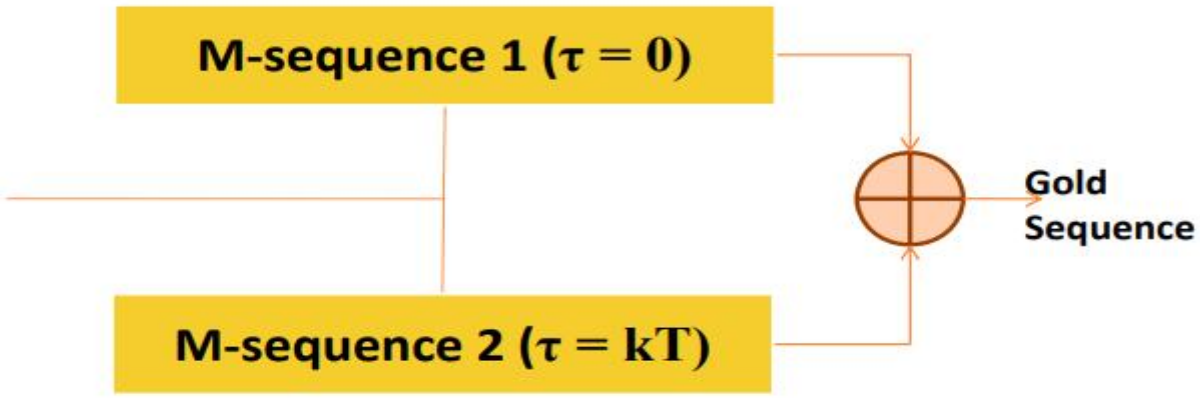
Gold codes are a crucial element in spread spectrum communication systems, particularly in CDMA (Code Division Multiple Access) technologies. They are sequences of binary numbers known for their unique properties that enable efficient and secure transmission of data over a shared channel. Gold codes find applications in

various fields like telecommunications, GPS systems, and military communications due to their ability to minimize interference and enhance data security.

Understanding Gold Codes:

Gold codes belong to the family of pseudorandom noise (PRN) sequences. These sequences are generated using mathematical algorithms and have properties like random sequences, but they are deterministically generated. Gold codes are designed to exhibit certain correlation properties crucial in spread spectrum communication.

They consist of two shorter sequences, often referred to as "m-sequences" or "maximal length sequences," that are combined using a modulo-2 addition (XOR operation). The resulting sequence exhibits desirable properties such as high autocorrelation and low cross-correlation, making it ideal for spreading the signal spectrum over a wide bandwidth and enabling multiple users to share the same frequency channel without interference.



Properties of Gold Codes:

1. **Periodicity and Auto correlation:** Gold codes exhibit a long period and excellent auto correlation properties. resulting in a long sequence before

repetition. This property helps in reducing interference and improving signal detection.

2. **Low Cross-Correlation:** When different gold codes are used simultaneously in a CDMA system, their cross-correlation values are very low, which minimizes interference among users sharing the same channel. This property enables multiple users to communicate simultaneously without significant signal degradation.
3. **Orthogonality:** Gold codes possess a degree of orthogonality, which is beneficial in spreading the signal spectrum efficiently. This property allows multiple users to use the same frequency band without causing interference between different transmissions.

Advantages:

1. **Low Cross-Correlation:** Gold codes exhibit low cross-correlation between different sequences. This property allows multiple users to share the same frequency band without causing significant interference, enabling efficient communication in CDMA systems.
2. **High Autocorrelation:** The autocorrelation of Gold codes is high, making it easier to detect and synchronize the transmitted signal at the receiver. This property enhances signal detection and improves the overall reliability of communication systems.
3. **Security Enhancement:** The pseudo-random nature of Gold codes adds a layer of security to the transmitted data. It becomes more challenging for unauthorized users to intercept or decipher the information due to the complex and unique spreading sequences used for each transmission.
4. **Bandwidth Efficiency:** Gold codes efficiently spread the signal spectrum over a wide bandwidth. This property allows for more simultaneous transmissions within the same frequency band, increasing the overall capacity of the communication system.

Disadvantages:

1. **Complexity in Generation:** The generation of gold codes require the combination of two shorter sequences using XOR operations. Implementing the generation process in hardware or software can be computationally intensive, especially for longer code lengths.
2. **Finite Code Lengths:** Gold codes have a finite length determined by the lengths of the constituent m-sequences used in their generation. As a result, there's a limit to the number of unique codes available, which might be a constraint in systems requiring an extremely high number of unique codes.
3. **Synchronization Challenges:** Achieving synchronization between the transmitter and receiver is crucial for successful decoding of the transmitted signal. In some cases, synchronization errors or difficulties can lead to signal degradation or loss of data integrity.
4. **Interference in Non-Ideal Conditions:** While Gold codes offer low cross-correlation, in practical scenarios with non-ideal conditions such as multipath propagation or high interference environments, there might still be some interference among simultaneous transmissions, impacting the overall performance.

MATLAB Example

n = 5 number of bits in each LFSR

feedback1 = [5, 2] feedback positions for LFSR1 (1-indexed)

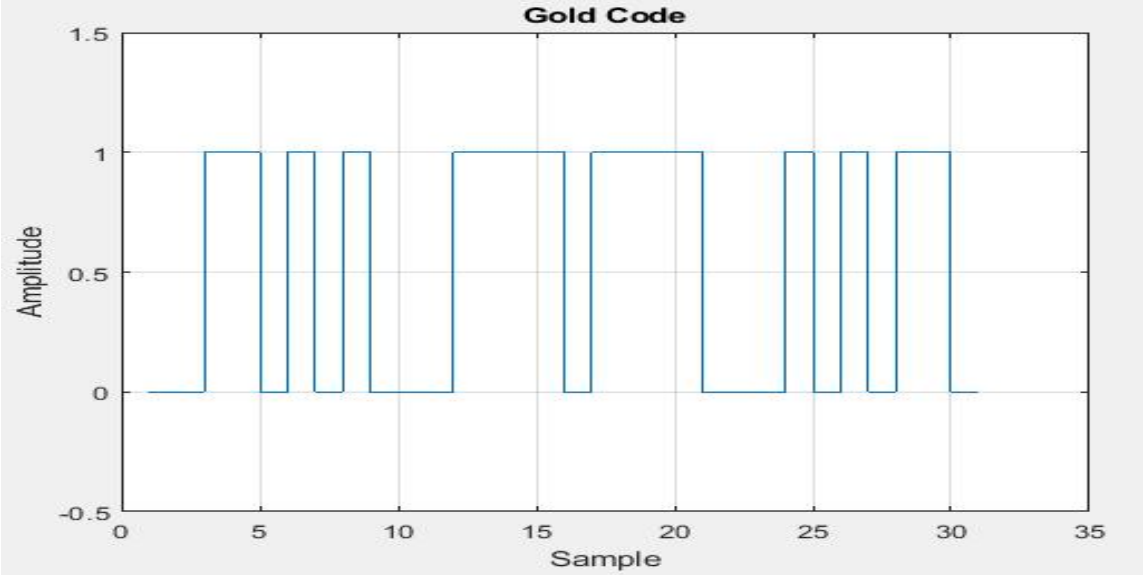
feedback2 = [5, 3] feedback positions for LFSR2 (1-indexed)

seed1 = [1 0 0 0 1] initial seed for LFSR1

seed2 = [1 1 1 0 1] initial seed for LFSR2

length = 31 length of the m-sequences and gold code

```
Gold Code:
  0  0  1  1  0  1  0  1  0  0  0  1  1  1  1  0  1  1  1  1  0  0  0  1  0  1  0
```

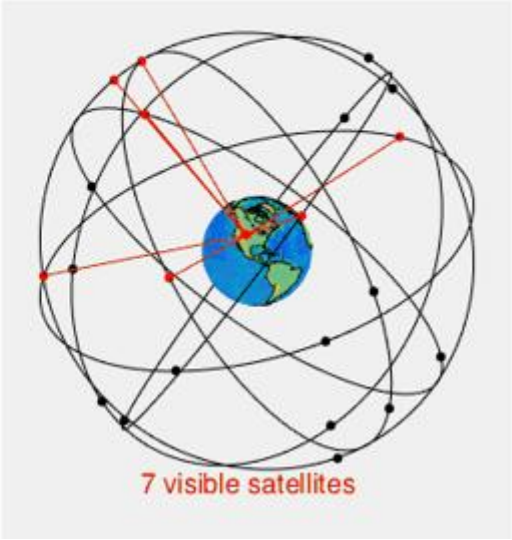


1.3.2.4 Application of DS-CDMA

GPS

- Global Positioning System (GPS), Precise satellite-based navigation and location system originally developed for U.S. military use. GPS is a fleet of more than 24 communications satellites that transmit signals globally around the clock.

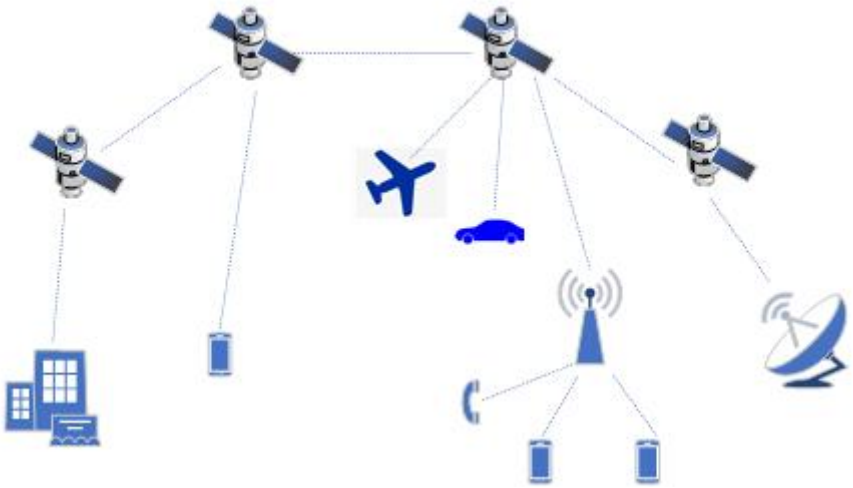
24 satellites in 6 orbital planes → 4 satellites per orbit
Satellites in MEO



- With a GPS receiver, one can quickly and accurately determine the latitude, the longitude, and in most cases the altitude of a point on or above Earth's surface. A

single GPS receiver can find its own position in seconds from GPS satellite signals to an accuracy of one metre; accuracy within one centimetre can be achieved with sophisticated military-specification receivers.

- This capability has reduced the cost of acquiring spatial data for making maps while increasing cartographic accuracy. Other applications include measuring the movement of polar ice sheets or even finding the best automobile route between given points.
- GPS is a critical technology widely used in various applications, including navigation, transportation, agriculture, defense, and emergency services.



How Does GPS Work?

GPS—has satellites orbiting Earth.

- These satellites carry atomic clocks that keep very accurate time. The GPS satellites transmit this time to receivers on Earth. By comparing the time at the

receiver to the time transmitted by the satellite and then multiplying the difference by the speed of light, the distance between the receiver and the GPS satellite can be calculated.

- However, knowing the distance between the receiver—your phone, for example—and one satellite doesn't give you your position on Earth. Instead, it merely tells you that you are somewhere on a sphere with the distance to the satellite as its radius. That's why GPS satellites are spaced in their orbits in such a way that four satellites are always in view. By using signals from four satellites, the receiver can compute its precise position.
- In theory, three satellites would normally provide an unambiguous three-dimensional fix, but in practice at least four are used to offset inaccuracy in the receiver's clock.

User location (unknown):

$$u_x,\ u_y,\ u_z$$

Satellites ephemeris at the time of transmission:

$$x_i,\ y_i,\ z_i \text{ for } i = 1,2,3,4$$

Time of transmission of the satellite signals:

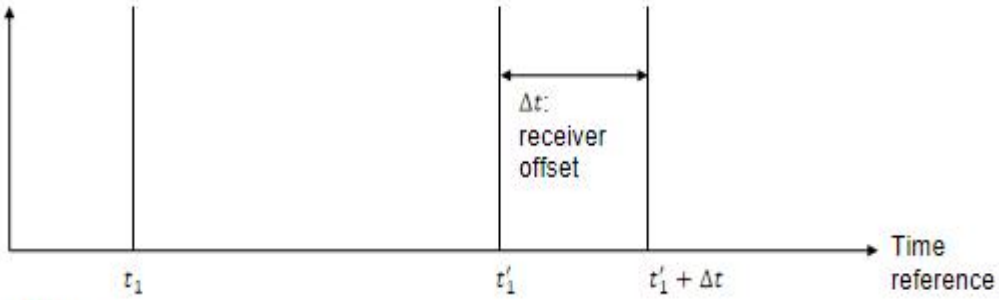
$$t_i \text{ for } i = 1,2,3,4$$

These are broadcast by the satellites

Time of reception of the satellite signals:

$$t_i' \text{ for } i = 1,2,3,4$$

These are determined by the user receiver



Distance value:

$$d_i = C \times (t_i' + \Delta t - t_1)$$

Distance equations:

$$d_1 = \sqrt{(u_x - x_1)^2 + (u_y - y_1)^2 + (u_z - z_1)^2} + C \times \Delta t$$

$$d_2 = \sqrt{(u_x - x_2)^2 + (u_y - y_2)^2 + (u_z - z_2)^2} + C \times \Delta t$$

$$d_3 = \sqrt{(u_x - x_3)^2 + (u_y - y_3)^2 + (u_z - z_3)^2} + C \times \Delta t$$

$$d_4 = \sqrt{(u_x - x_4)^2 + (u_y - y_4)^2 + (u_z - z_4)^2} + C \times \Delta t$$

- Global navigation satellite system (GNSS) is a general term describing any satellite constellation that provides positioning, navigation, and timing (PNT) services on a global or regional basis."

- System is passive (users receives only) can serve unlimited number of users
- The system uses two frequency ranges for transmission

  L1 = 1575.42 MHz

  L2 = 1227.6 MHz

  L5=1176.45 MHz
- Every GPS satellite transmits on multiple links, every link carrier one or more codes, and provide redundancy and ionospheric correction

- We use codes to identify individual satellites and derive pseudo range to each satellite

- Satellites use Code Division Multiple Access (CDMA) to simultaneously communicate on these frequency regions

C/A code

- The Coarse/Acquisition (C/A) code used in GPS is a pseudorandom binary sequence that is specific to each GPS satellite. The C/A code is generated by a feedback shift register (FSR) or a similar structure known as a linear feedback shift register (LFSR).

- Use only on L1, with chipping rate =1.023MHz, Transmission Rate=1023Mbps

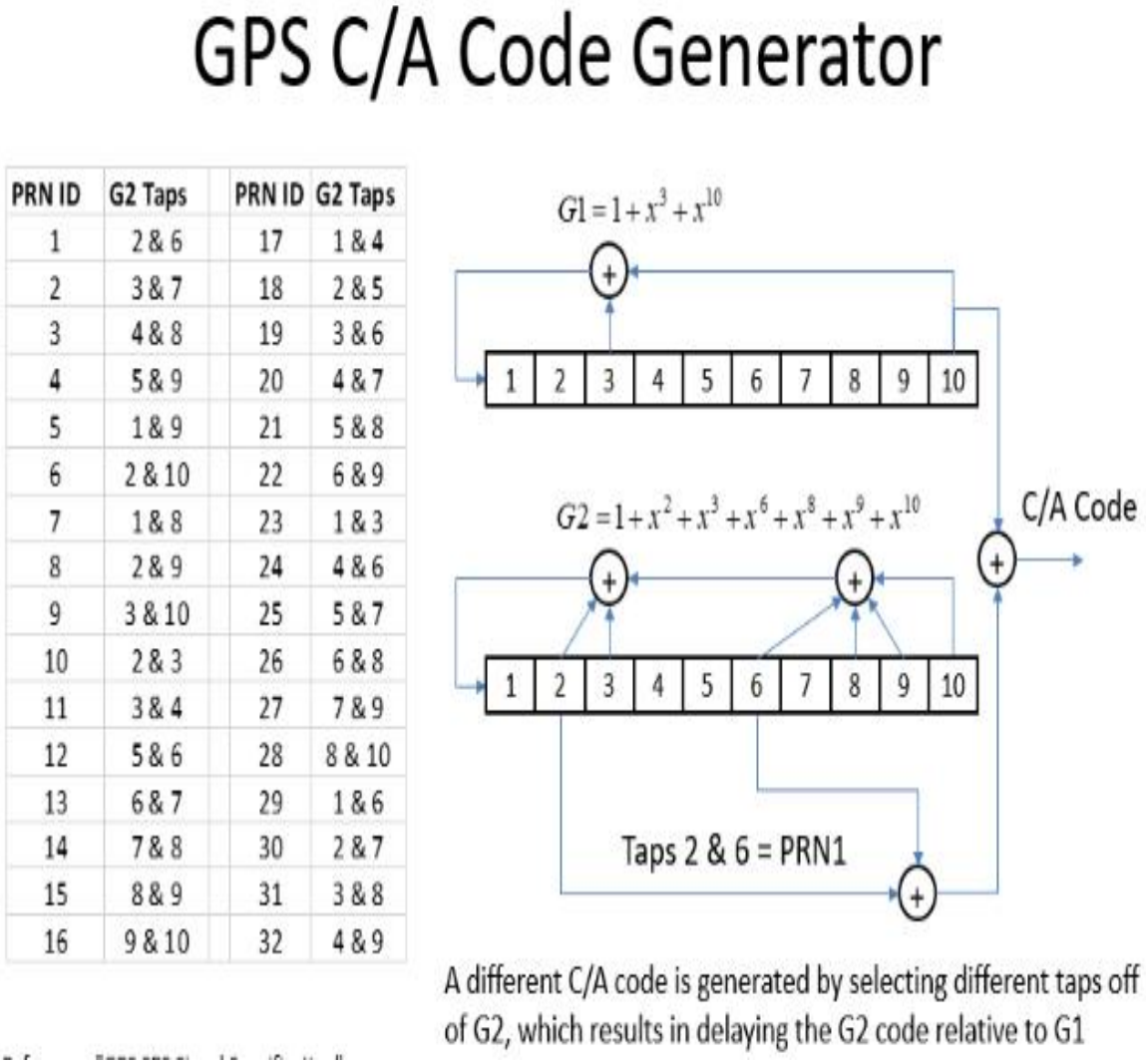
- The C/A code period is 1 ms, meaning the entire sequence repeats every millisecond. This periodicity is essential for GPS receivers to synchronize with satellite signals and chip length corresponding to 293 m (about 961.29 ft).

- The generated C/A code is modulated onto the GPS carrier signal and transmitted by the GPS satellite. GPS receivers on the ground use this transmitted C/A code for signal acquisition and navigation calculations.

- The C/A code plays a critical role in the initial acquisition and synchronization of GPS signals by providing a unique and predictable pseudorandom sequence. This allows GPS receivers to precisely calculate the distances to multiple satellites and determine their position on Earth.

Ex

we generate 2 polynomial (2LFSR), first polynomial=$1+x^3+x^{10}$ (xor between 3,10), Second polynomial=$1+x^2+x^3+x^6+x^8+x^9+x^{10}$ (xor between 2,3,6,8,9,10)



From MATLAB we generate 32 Space Vehicle we can detect each Space vehicle from first 10 bits that help us to know which Space Vehicle We treat

1 vehicle_indices	2 Space_Vehicles									
1	1	1	0	0	1	0	0	0	0	0
2	1	1	1	0	0	1	0	0	0	0
3	1	1	1	1	0	0	1	0	0	0
4	1	1	1	1	1	0	0	1	0	0
5	1	0	0	1	0	1	1	0	1	1
6	1	1	0	0	1	0	1	1	0	1
7	1	0	0	1	0	1	1	0	0	1
8	1	1	0	0	1	0	1	1	0	0
9	1	1	1	0	0	1	0	1	1	0
10	1	1	0	1	0	0	0	1	0	0
11	1	1	1	0	1	0	0	0	1	0
12	1	1	1	1	1	0	1	0	0	0
13	1	1	1	1	1	1	0	1	0	0
14	1	1	1	1	1	1	1	0	1	0
15	1	1	1	1	1	1	1	1	0	1
16	1	1	1	1	1	1	1	1	1	0

1 vehicle_indices	2 Space_Vehicles									
17	1	0	0	1	1	0	1	1	1	0
18	1	1	0	0	1	1	0	1	1	1
19	1	1	1	0	0	1	1	0	1	1
20	1	1	1	1	0	0	1	1	0	1
21	1	1	1	1	1	0	0	1	1	0
22	1	1	1	1	1	1	0	0	1	1
23	1	0	0	0	1	1	0	0	1	1
24	1	1	1	1	0	0	0	1	1	0
25	1	1	1	1	1	0	0	0	1	1
26	1	1	1	1	1	1	0	0	0	1
27	1	1	1	1	1	1	1	0	0	0
28	1	1	1	1	1	1	1	1	0	0
29	1	0	0	1	0	1	0	1	1	1
30	1	1	0	0	1	0	1	0	1	1
31	1	1	1	0	0	1	0	1	0	1
32	1	1	1	1	0	0	1	0	1	0

P-code

- A P-code (Precise code) is a code used in GPS (Global Positioning System) signals for military purposes to provide a more precise level of accuracy than the civilian C/A code (Coarse/Acquisition code).

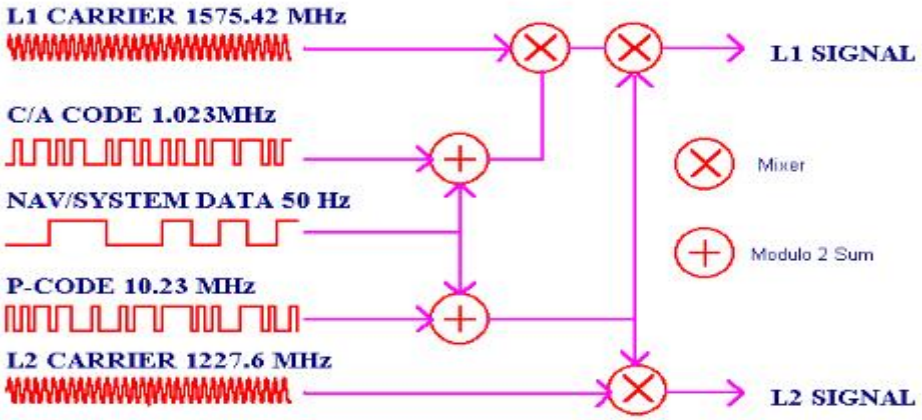
- The P-code is encrypted and reserved for military use to prevent unauthorized users from obtaining the highest level of accuracy provided by the GPS system.

- Transmitted on both L1 and L2 is much longer and has chipping rate=10.23Mcps 10 timer faster than C/A code

- Chip length=29.3 Meters (vs 293 meters (about the height of the Empire State Building) for C/A code)

- When P-code is encrypted, it is known as the P(Y) Code

  The P-code is a long, pseudo-random noise code that repeats every seven days. It is modulated onto the GPS carrier frequency to produce the Y-code, which is then combined with the C/A code to produce the encrypted P(Y)-code. The civilian GPS users primarily rely on the C/A code, which provides less accuracy than the military P-code.



GPS SATELLITE SIGNALS

Firstly, the navigation signal (50 Hz) will either modulate with C/A code or P code (both are the pseudo random code) depending on the type of application.The navigation signal modulated with C/A code will mix with L1 carrier producing a L1 signal. Whereas the navigation signal modulated with P code will mix with both the L1 and L2 carrier produce a L2 signal.

1.3.2.5 Advantages and Disadvantages

Advantages of DS-CDMA

1. **Resistance to Interference:** The spreading of the signal over a wide bandwidth makes it more resistant to narrowband interference and jamming.
2. **Multipath Resistance:** The correlation process helps in distinguishing the original signal from reflected (multipath) signals, improving reliability.
3. **Privacy and Security:** The use of pseudorandom spreading codes makes it difficult for unauthorized users to intercept or jam communication.
4. **Efficient Spectrum Use:** Multiple users can share the same frequency band simultaneously without significant mutual interference.

Disadvantages of DS-CDMA

I. **Complexity:** The system requires complex hardware and software for spreading, despreading, and managing multiple user codes.
II. **Power Control:** Requires precise power control to avoid the near-far problem, where signals from closer users overpower those from farther users.
III. **Capacity Limitations:** The number of users that can be supported is limited by the available spreading codes and the processing gain.

CHAPTER 2

Steganography 1: Hiding text in image

2.1 Definition

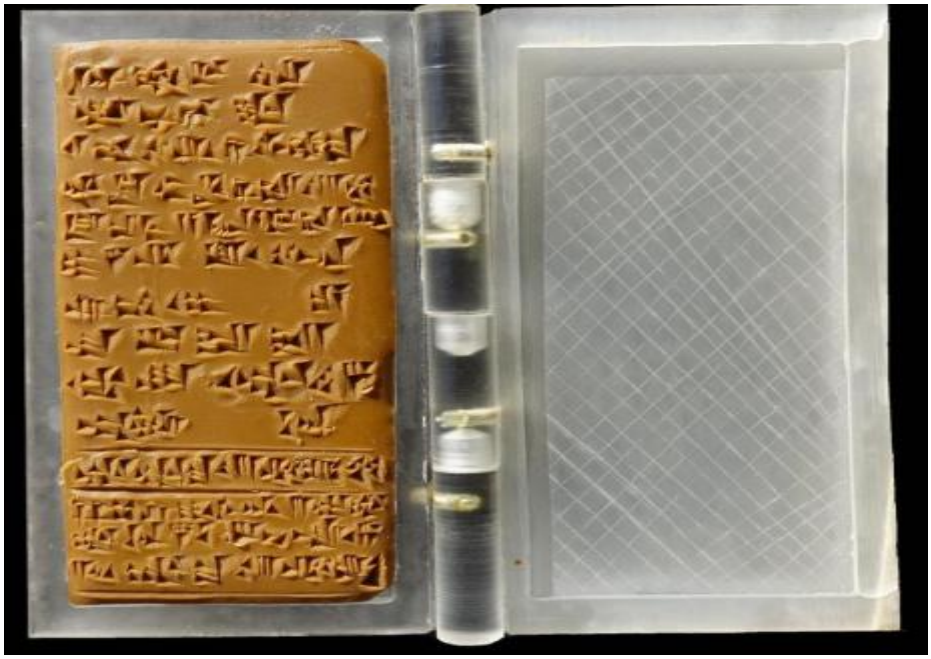
Steganography, derived from the Greek words "steganos" (meaning "covered" or "concealed") and "graphein" (meaning "writing"), is concealing a hidden message within a carrier in a certain way which is not detectable by human senses. Usually, the message is hidden in a sound or an image.

2.2 History

In ancient times:

Histiaeus, a Greek ruler, shaved the head of a slave then tattooed a message on his scalp and waited for the hair to grow back before sending the slave to deliver the message.

Wax tablets messages were written on a wooden tablet and then covered with a layer of wax. The hidden message would be inscribed on the wood, and the wax would be melted and remolded to hide the writing.



In Middle Ages:

Ciphers and letters of the alphabet are replaced with other letters or symbols.

Johann Heidenberg, a German abbot and a polymath, is considered as the father of modern steganography as he wrote “steganographia” a work that appeared to be about magic but contained techniques for encoding messages within text.

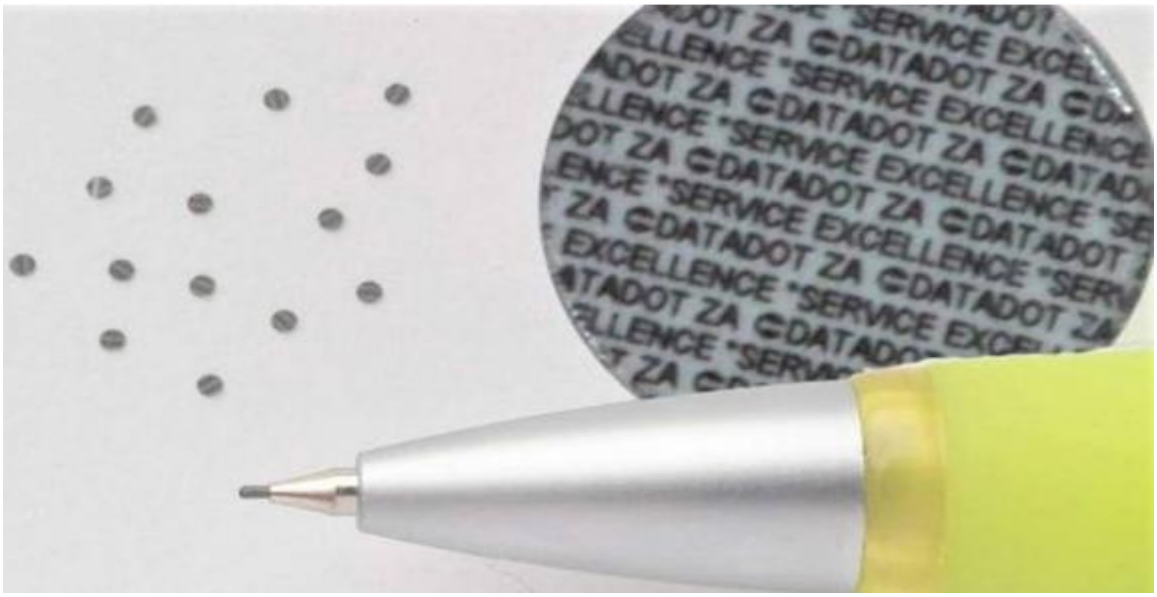
In American civil war:

Cipher disks and cardan grilles were used by the Union and Confederate armies which allowed the message to be hidden in a normal text and revealed when the grille was over it.

During world wars:

Invisible Ink was used extensively in Both World Wars with various chemical compounds being developed for covert communication.

Microdots were used to shrink messages or photographs to the size of a typed period. These were then placed within innocuous documents or even on the edges of letters.



In modern days:

Water marking protects intellectual property by embedding the hidden information with the digital media to assert ownership.

Network steganography by concealing the message in the TCP/IP headers.

Digital steganography where the message is concealed in different digital images, audio files and video files.

2.3 Steganography Techniques

A. Image Steganography

Least Significant Bit (LSB) Method

Frequency Domain Techniques (Discrete Cosine Transform, Discrete Wavelet Transform)

Spatial Domain Techniques

B. Audio Steganography

Low-bit Encoding

Phase Coding

Spread Spectrum Techniques

C. Text Steganography

Embedding in whitespace or formatting characters

Concealing text within seemingly normal passages

2.4 Project of Hiding text in Image

2.4.1 Abstract

- Cryptography and steganography are two issues in security systems. Cryptography scrambles the message to be incomprehensible while steganography shrouds the message to be invisible. Therefore, encryption of

any private data before concealing it in the cover object will provide twofold security.

- In this project, a secure steganographic algorithm based on cover image encryption and Vigenère cipher is proposed, where the secret message is encrypted using Vigenère cipher then the cover image is encrypted using the logical XOR operation. After that, the encrypted form of the ciphered message is embedded within the encrypted cover image using the LSB algorithm in the spatial domain. The simulation consequence illustrates that the scheme provides better protection for confidential messages.

2.4.2 Introduction

I. The transfer of important and confidential data in our time via the internet has become one of the biggest challenges due to the development in technology as well as the accumulation of experience over the years among those interested.

II. Cryptography and steganography provide the most significant techniques for information security.

III. Cryptography is a technique of transforming and transmitting private data in an encoded way so that only authorized and intended users can obtain or work on it. The word cryptography is derived from the Greek words “Κρυπτο” which means hidden or secret.

IV. Steganography is the art and science of invisible communication. It is accomplished by hiding information in other information, thus hiding the existence of the information. Steganography is derived from the Greek words “stegos” meaning “cover” and “grafia” meaning writing. Steganography and cryptography are techniques used to protect information from unwanted parties but neither technology alone is perfect. Once the presence of hidden information is revealed or suspected, the reason of steganography is partly defeated.

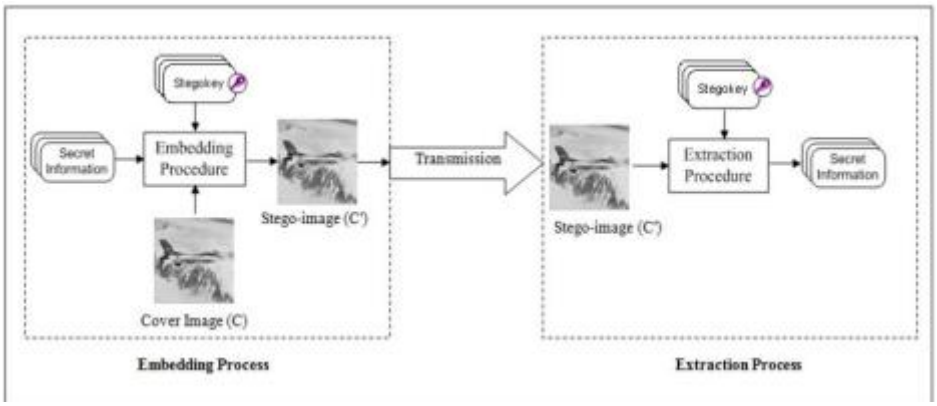
V. The strength of steganography increases by combining it with cryptography.

2.4.3 Digital Image Steganography

I. A digital image is a finite collection of elements called pixels; each pixel has a particular location and value. The most popular medium used for hiding secret data is image files because of their high capacity and easy availability over the internet.

II. At the sender's side, the image used for embedding the secret message is called a cover image, and the secret information that needs to be protected is called a message. As soon as data is embedded using some appropriate embedding algorithm, then it is called stego-image. This stego-image is transferred to the receiver.

III. The receiver extracts out the secret message using an extraction algorithm. The main processes of a steganographic system can be graphically represented as in Figure



C denotes to the cover image and C' denotes to the stego-image (the cover C after embedding the secret information).

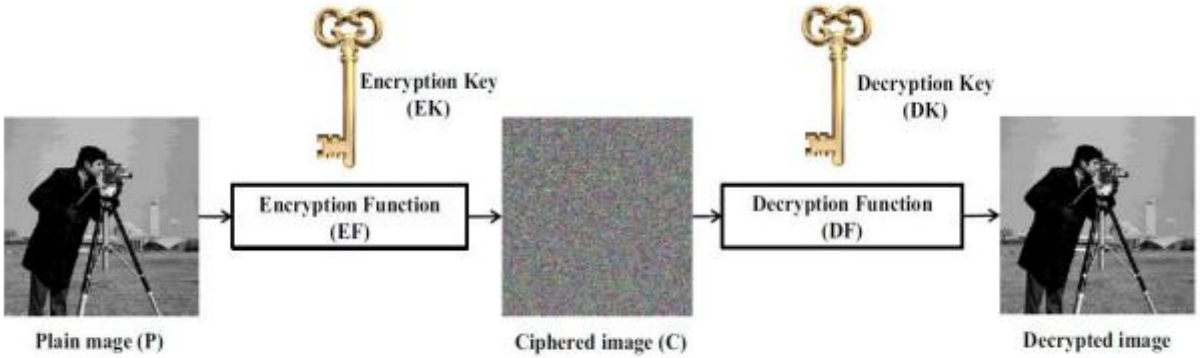
2.4.3.1 Image Encryption

I. Image encryption techniques play a significant role in multimedia applications to secure and authenticate digital images.

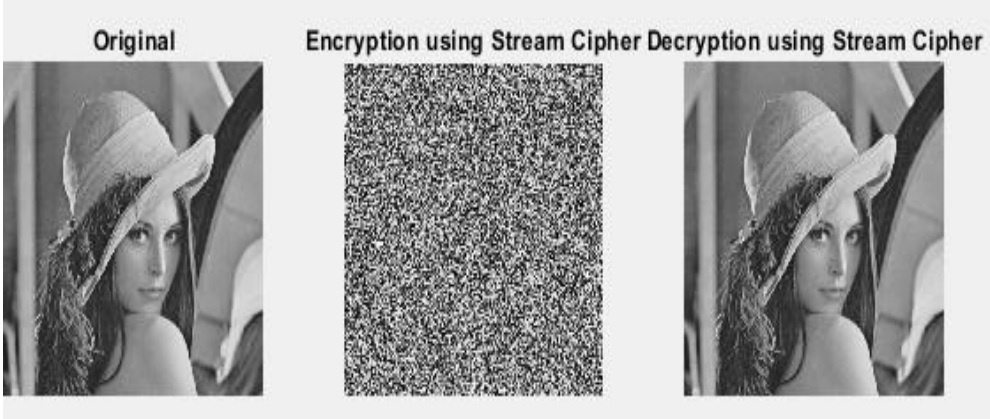
II. From figure illustrates the general framework of image encryption techniques. An input image that needs to be encrypted is called a plain-image and an encrypted image is known as a ciphered image.

III. The plain and ciphered images are represented by P and C, respectively. The image encryption techniques can be divided as symmetric and asymmetric techniques.

IV. In the case of symmetric image encryption, the encryption and decryption keys are the same, i.e., EK = DK. The keys are to be kept secret during communication. When different keys are used for encryption and decryption, the image encryption is known as asymmetric image encryption



We use Stream Cipher from CHAPTER 1 to Encrypt Image Using Symmetric Encryption



2.4.3.2 Vigenère Cipher

The Vigenere cipher is a polyalphabetic substitution cipher in which each letter of the alphabet is replaced with a different letter according to a table of ASCII. It can be expressed as follows

A	0	J	9	S	18
B	1	K	10	T	19
C	2	L	11	U	20
D	3	M	12	V	21
E	4	N	13	W	22
F	5	O	14	X	23
G	6	P	15	Y	24
H	7	Q	16	Z	25
I	8	R	17		

$$\text{Encrypt } (c_i) : E(p_i) = (p_i + k_i) \bmod 26$$
$$\text{Decrypt } (p_i) : D(c_i) = (c_i - k_i) \bmod 26$$

where:

p_i is plaintext; c_i is ciphertext. *And* k_i is key.

For example,to encrypt the message Welcome with the keyword Key. Vigenère proceed by repeating the Keyword as many times as needed above the Message as follow

K	E	Y	K	E	Y	K
10	4	24	10	4	24	10
W	E	L	C	O	M	E
22	4	11	2	14	12	4
10+22-26 =6	4+4=8	11+24-26 =9	10+2=12	4+14=18	24+12-26 =10	10+4=14
G	I	J	M	S	K	O

From MATLAB

```
Input Text: WELCOME
Keyword: key
Ciphertext: GIJMSKO
Output Text: WELCOME
```

2.4.3.3 Least Significant Bit (LSB)

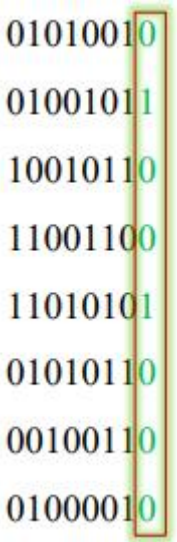
I. Technique The least interesting bit is a basic method for inserting data in the image file. Simple steganography requires embedding the bits of the message in the least important bits of the image.

II. In an image, each pixel is represented in 8 bits. The last bit in a pixel is called as least significant bit as its value will affect the pixel value only by "1". So, this property is used to hide the data in the image.

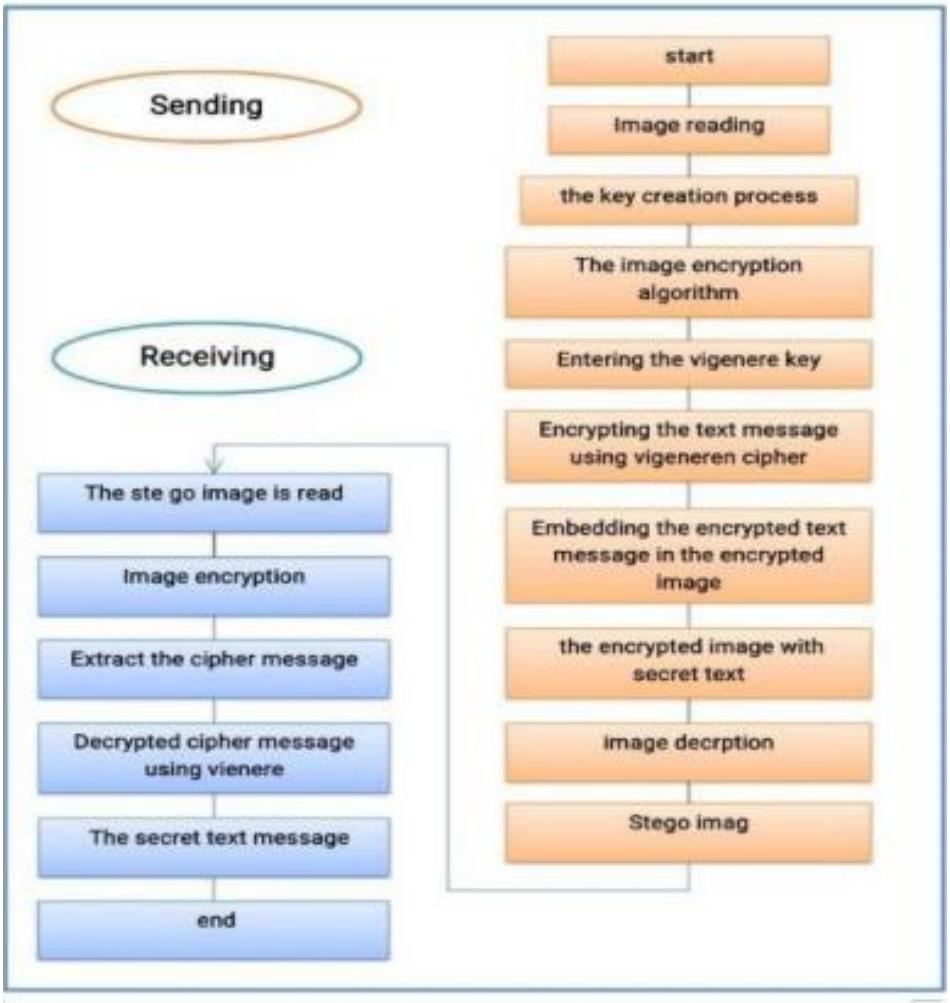
III. we encrypt the raw data before embedding it in the image. Though the encryption process increases the time complexity, but at the same time provides higher security also

01010010
01001010
10010111
11001100
11010101
01010111
00100110
01000011

To hide the letter **H** whose binary value of ASCII code is **01001000** , we would replace the LSBs of these pixels to have the following new values:



2.4.4 The Proposed Method



Steps: In Transmitter

1. Read Image
2. Enter Text message
3. Encrypt Image (First Key)
4. Encrypt Text message (Second Key)
5. Embedding Encrypted Text in Encrypted Image
6. Decrypt Image
7. Send stego image

Steps: In Receiver

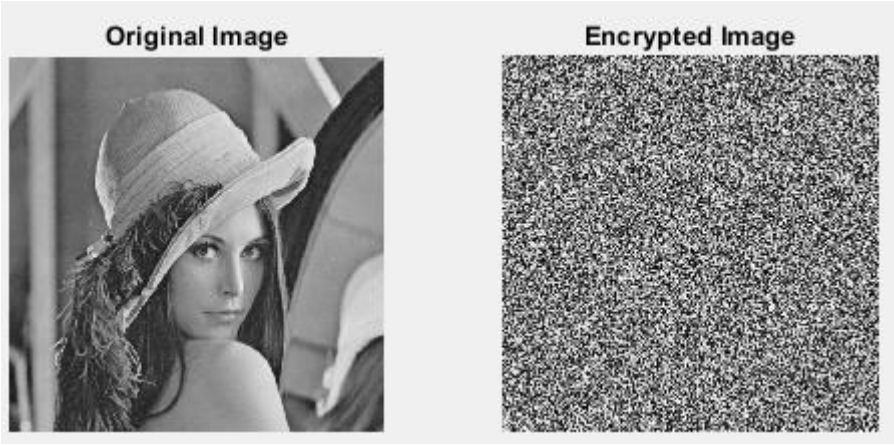
1. Use The First key to Encrypt Stego image
2. Extract Encrypted Text message from Encrypted Image

3. Use The Second key to Decrypt Text message

4. Get Text message

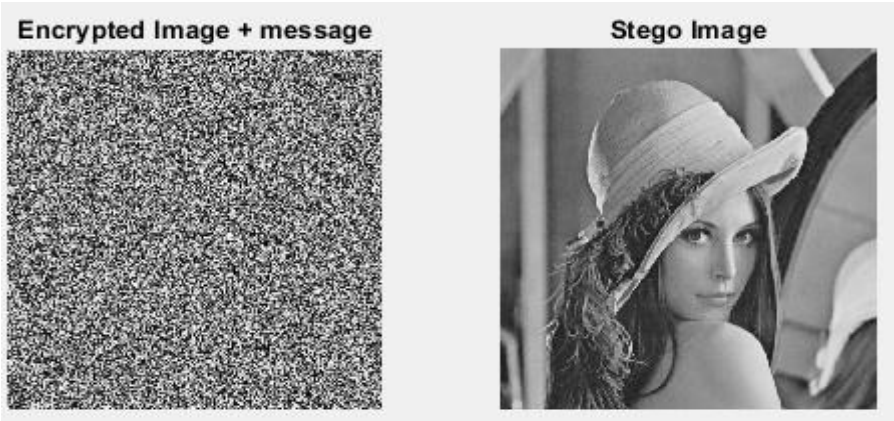
2.4.5 Simulation Results

At Transmitter

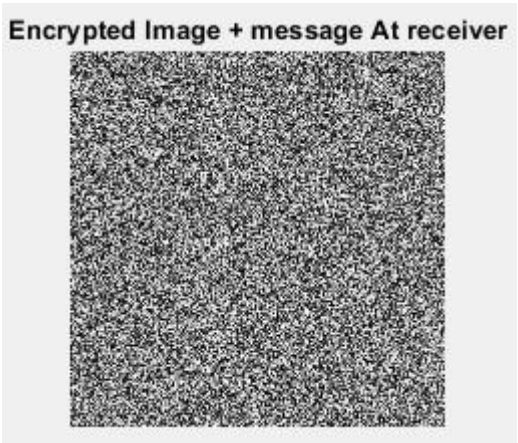


Encrypt Text and Embed it in Image using LSB

```
Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
```



At receiver



Extract Text from Image and Decrypt it

```
Output Text: You can connect with me through abdobadr@gmail.com.
```

	MSE	PSNR	Entropy	Corr
Original and Stego	7.7820e-04	181.6105	7.4451	1

Corr:

The result will be between -1 and 1, where 1 indicates perfect correlation,

0 indicates no correlation, and -1 indicates perfect inverse correlation.

Notes

I. If Receiver try to Extract Text Using LSB from image before Encrypt Image

```
Input Text: You can connect with me through this E-mail Abdelrahmanbadr12112001@gmail.com.
Keyword: key
Ciphertext: ISS GYX AYRLOGR AGDL WI DLPYYER RRMQ I-WEGV YLHCVVYRQYXFYNV12112001@EWEGV.AYQ.
Output Text: -=X□□□Y¬¼§Ð7□□L□□□VÍ□□¾´Î¼`.Ï□□\□¹FTN□V□T  ³¿□&□¹□RAZ□µÆG□|>.FS"Ã□Ò}7□£□□D®|⁵)Á¶Á©¦
```

Receiver can't receive text correctly without encrypting image.

II. Image size (512x512) so we can hide in image message with Max length = 512*(512/8) =32768

If length of message=32768

```
len =

       32768

Error using WithoutCrop (line 36)
ERROR: message is too large for cover image
```

We can hide message with length=32767 or less in image

```
len =

       32767

Output Text: YOU CAN CONNECT WITH ME THROUGH THIS E-MAILABDELRAHMANBADR12112001@GMAIL.COM.
```

2.4.6 Problems and Noises

There are problems, noises that Image may be exposed to before they reach to receiver

2.4.6.1 Salt and Pepper noise

- Add salt and pepper noise, with a noise density of 0.007 (Message doesn't change)

```
52 -    noisy=imnoise(decl,'salt & pepper',0.007); %Salt and pepper noise
```

```
Command Window
New to MATLAB? See resources for Getting Started.

Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
Output Text: You can connect`with me through abdobadr@gmail.com.
```

```
52 -    noisy=imnoise(decl,'salt & pepper',0.0071); %Salt and pepper noise
```

```
Command Window
New to MATLAB? See resources for Getting Started.

Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
Output Text: You can connect witf me through abdobadr@gmail.com.
```

- Add salt and pepper noise, with a noise density of 0.02, to the image

```
52 -    noisy=imnoise(decl,'salt & pepper',0.02); %Salt and pepper noise
```

```
Command Window
New to MATLAB? See resources for Getting Started.

Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
Output Text: You can covoect with me through abdobadr@gmail.com.
```

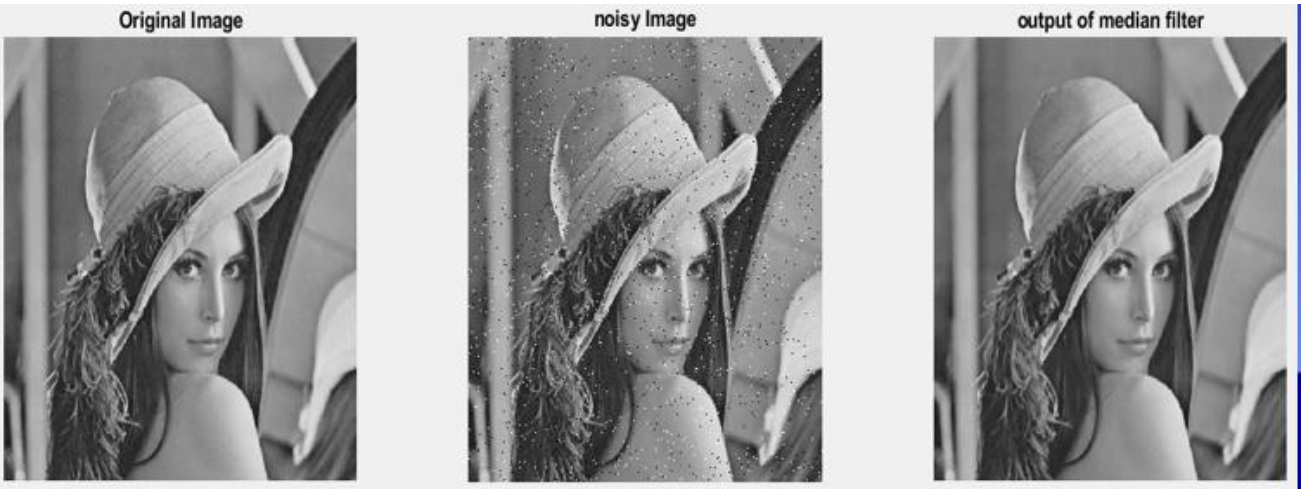
- Different types of filters to remove salt & pepper noise

	Median	Average	Gaussian	Rank order	Wiener
MSE between Original and Filter	22.5757	1.7540e+04	122.3875	22.5757	176.1603

From this table

Median filter and Rank order filter are the best for filtering salt & pepper noise

Average filter is the worst for filtering salt & pepper noise



2.4.6.2 Gaussian noise

- Add Gaussian noise with 0.1

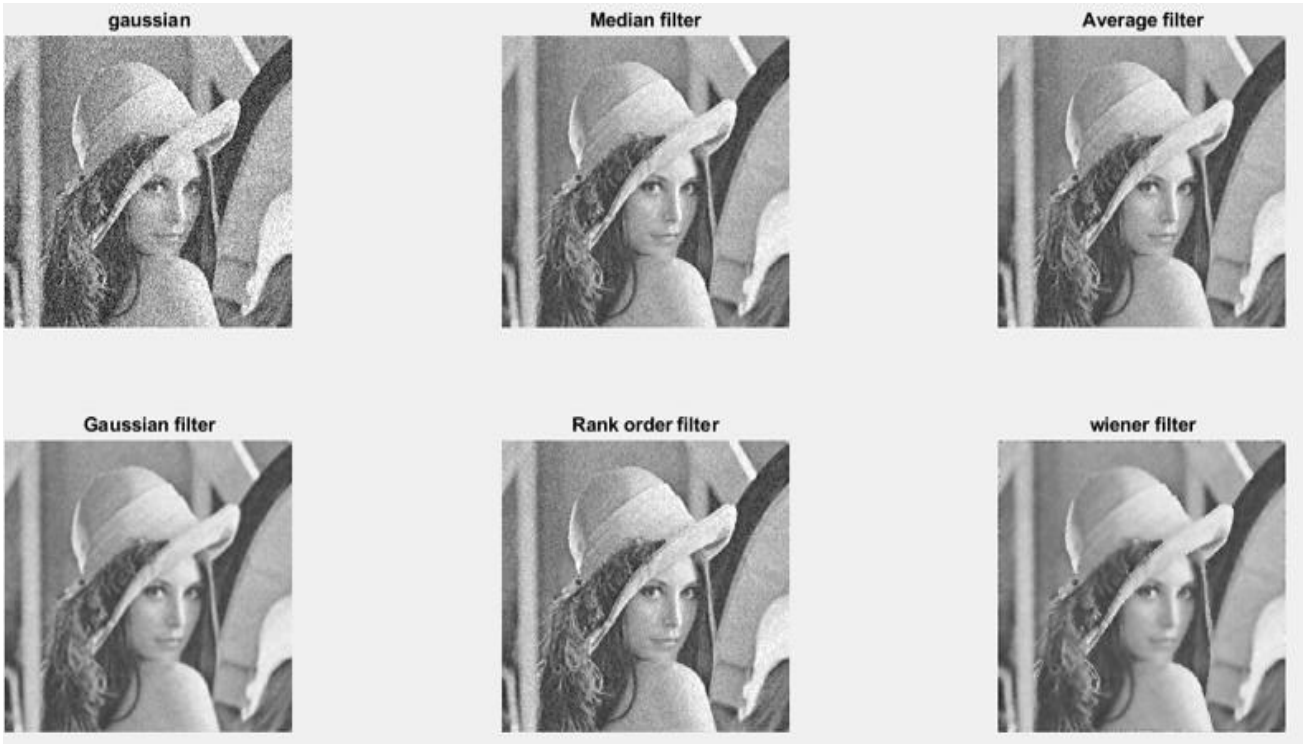

```
53 -    noisy=imnoise(decl,'gaussian',0.1);
```

Command Window

New to MATLAB? See resources for Getting Started.

```
Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
Output Text: h6¶ ri□g0pj□□
or>#□m
```

- Different types of filters to remove Gaussian noise



	Median	Average	Gaussian	Rank order	Wiener
MSE between Original and Filter	781.4785	1.7515e+04	735.7	781.4785	736.6217

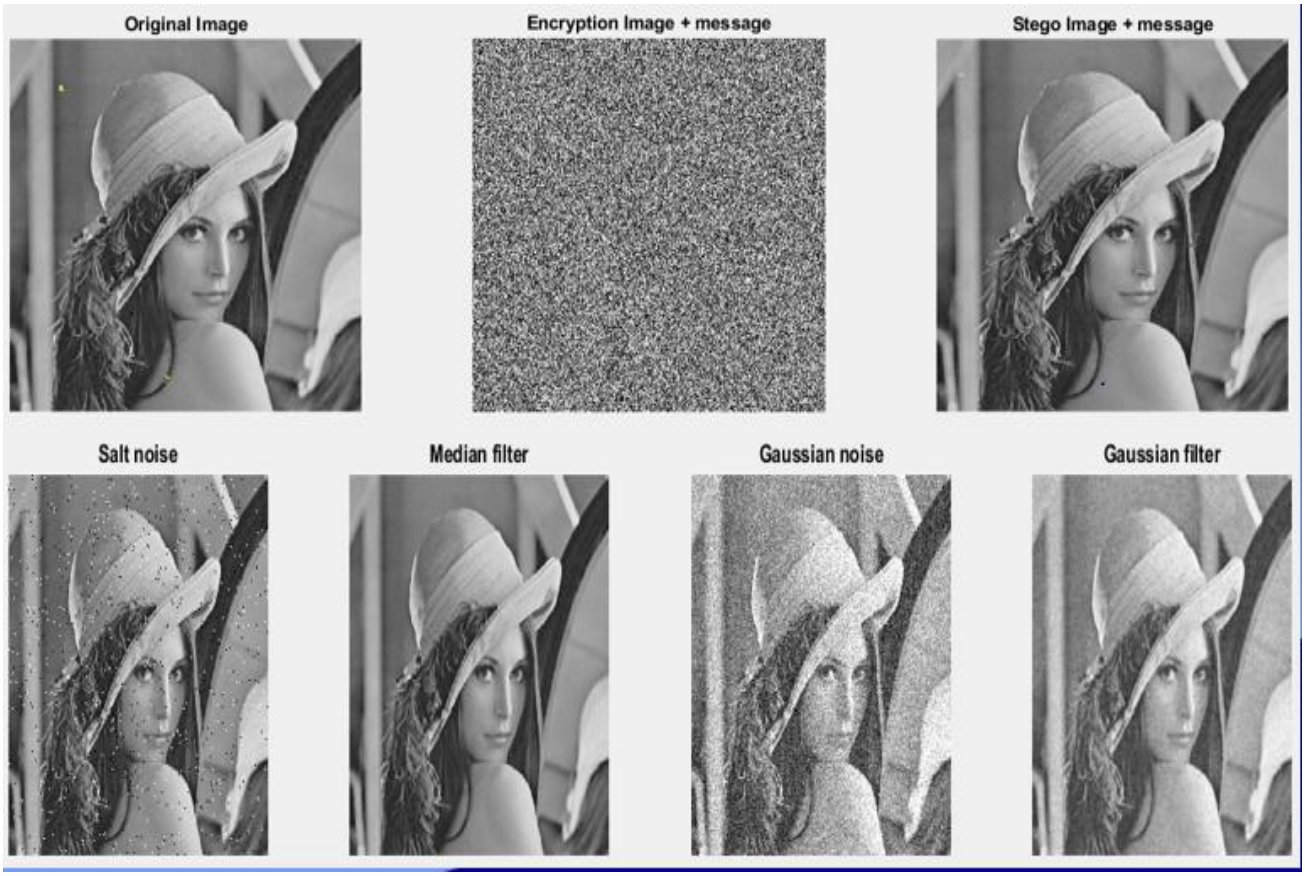
From this table

wiener and gaussian filter are the best for filtering gaussian noise

Average filter is the worst for filtering gaussian noise


Original Image
noisy Image
output of gaussian filter

Summary


Original Image
Encryption Image + message
Stego Image + message
Salt noise
Median filter
Gaussian noise
Gaussian filter

2.4.6.3 Crop Image

- Crop half of image (Left half)

```
54 -    noisy=imcrop(decl,[1,1,256,512]);
```

```
Command Window
New to MATLAB? See resources for Getting Started.

Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
Output Text: You can connect with me through abdobadr@gmail.com.
```

We can hide message Max length = 16448, in This half of image

- Crop half of image (Right half)

```
54 -    noisy=imcrop(decl,[256,1,256,512]);
```

```
Command Window
New to MATLAB? See resources for Getting Started.

Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
Output Text: ᵏz□|□cUzT□ r[Ci g=s &i□□r%?□□O
```

There isn't Message in this half, because message hide as vertical in image from left to right

- Crop quarter of image

```
54 -    noisy=imcrop(decl,[1,1,256,256]);
```

```
Command Window
New to MATLAB? See resources for Getting Started.

Input Text: You can connect with me through abdobadr@gmail.com.
Keyword: key
Ciphertext: Iss gyx ayrlogr agdl wi dlpyyer ylhmlebb@ewegv.ayq.
Output Text: You can connect with me through Y©□uskg±sb□ao□□□□□:□jI§W□×□`@1
```

We can hide message Max length = 32 in this quarter of image

There is problem, there isn't any message in right half of image, so we modified this project to hide the same message in each quarter of image to receive most of message correctly

<u>At transmitter</u>

1- Divide Image into 4 parts.

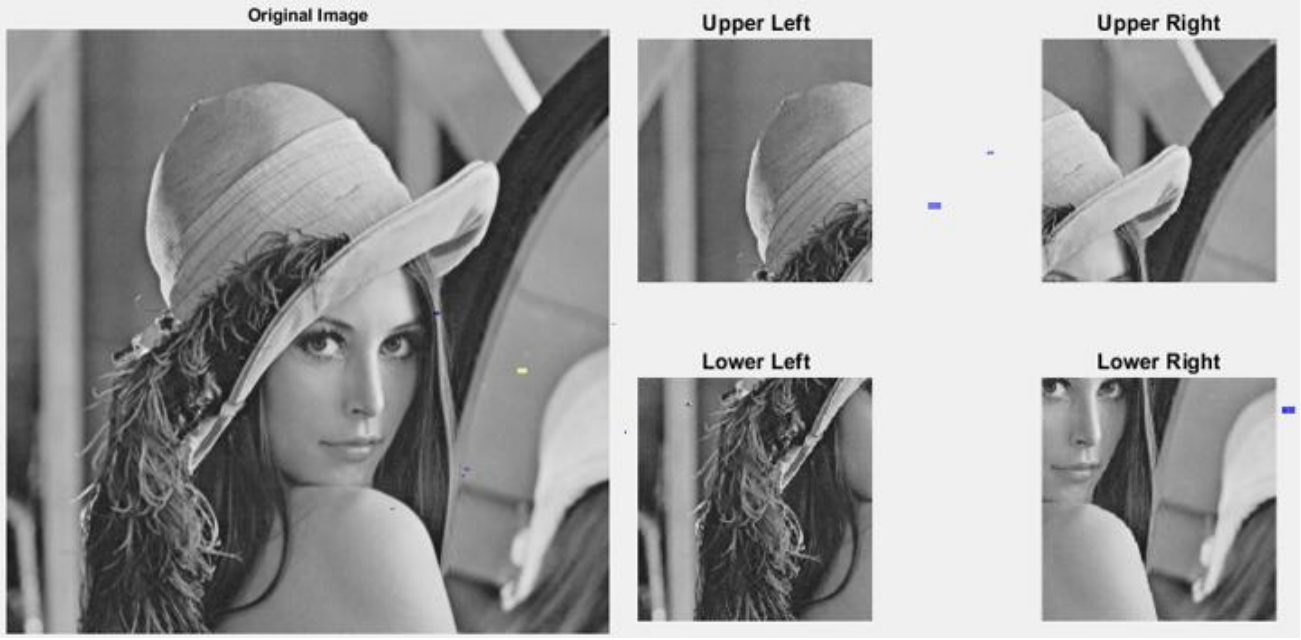
2- Embed message into each image (we decrease the length of message to Extract correctly from quarter of image)

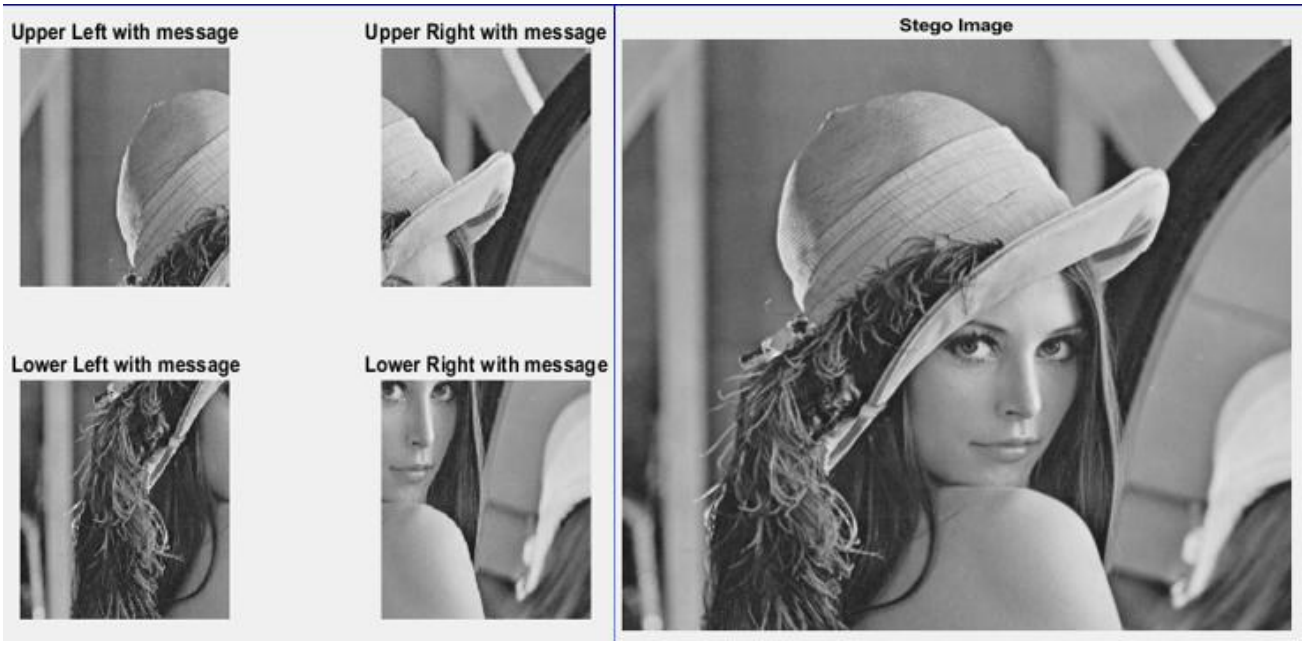
3- Combine each image with message into one image (Stego Image)

At receiver

1- Divide Image into 4 parts.

2- Extract message from each image





	MSE	PSNR	Entropy	Corr
Original and Stego	0.002	171.8390	7.4451	1

```
Input Text: Connect with abdobadr@gmail.com.
Ciphertext: Mslxiad usxf eznszkhp@kkkmj.gmw.
Output Text: Connect with abdobadr@gmail.com.
```

- Crop half of image

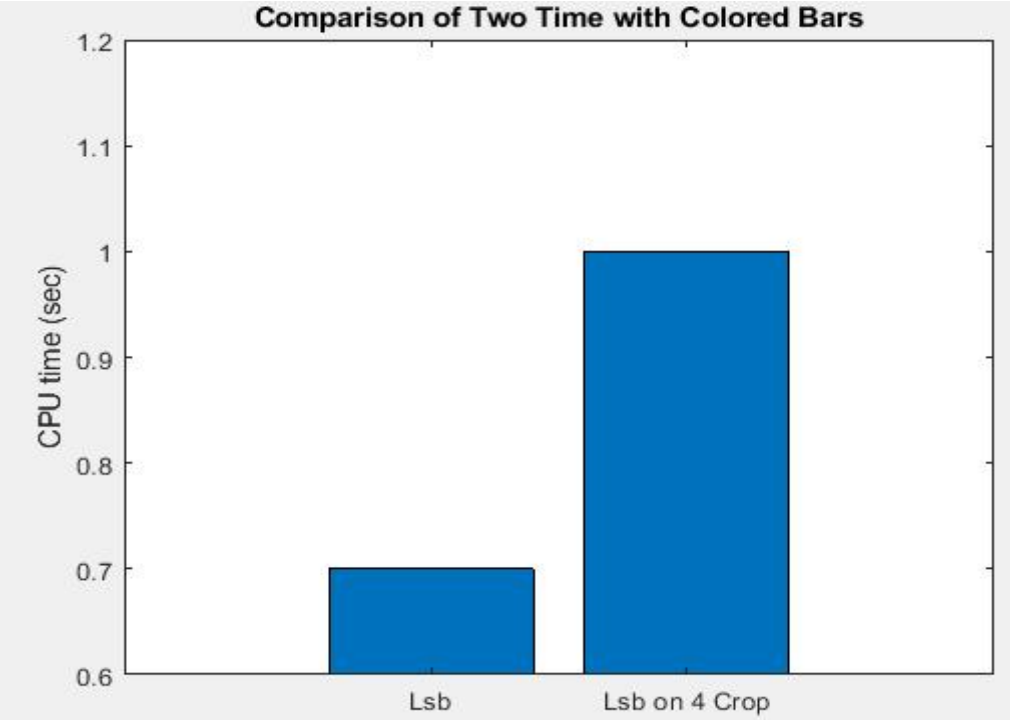
```
Output Text: □ÿ¢¼÷
Clót□`□Ðré}n"G□Y0~ù÷0□@üconnect with abdobadr@gmail.com.}
```

- Crop quarter of image

```
Output Text: connect with abdobadr@gmail.com. b3?Q*èA□³□"ub!âÛW□|ÔÓsí□^□□#Dn×ÀÓû
```

- When increasing the number of crops means increasing the CPU time, MSE and decreasing the PSNR.

	CPU Time
LSB on 4 Crop	1 sec
LSB	0.7 sec



2.4.7 Conclusion

1. This project offers a secure method to hide confidential messages with multiple levels of security: firstly, the secret message is encrypted using Vigenère cipher. Secondly, the cover image is encrypted using the logical XOR operation after creating an array containing the keys of the same size as the cover image to provide another level of security. Thirdly, the encrypted form of the ciphered message is embedded within the encrypted cover image using the LSB algorithm in the spatial domain. The proposed scheme creates better camouflage to evade intruder attention and realize better performance in terms of measurement the steganographic system as demonstrated according to the performance analysis.
2. We modify the project to hide the same message in 4 parts of image to decrease the effect of cropping.

CHAPTER 3

Steganography 2: Hiding text in sound

3.1 Introduction about sound:

Sound is a form of energy that travels through waves, typically propagated through a medium such as air, water, or solids. It is a fundamental aspect of human experience and plays a crucial role in communication, perception, and the functioning of many technologies. Here are some basic concepts about sound:

Properties of Sound

- **Wave Nature:**
  - Sound travels in the form of longitudinal waves, where vibrations propagate through a medium in a direction parallel to the wave's motion.
  - These waves are characterized by compressions (regions of high pressure) and rarefactions (regions of low pressure).
- **Frequency:**
  - Frequency refers to the number of cycles of a sound wave that occur persecond, measured in Hertz (Hz).
  - Higher frequencies correspond to higher pitches, while lower frequencies correspond to lower pitches.
  - The human ear can typically perceive frequencies ranging from about 20 Hz to 20,000 Hz (20 kHz).
- **Amplitude:**
  - Amplitude refers to the magnitude or intensity of sound waves, measured in decibels (dB).

- Higher amplitudes correspond to louder sounds, while lower amplitudes correspond to quieter sounds.
- Amplitude affects the perceived loudness or volume of sound.

- **Wavelength:**
  - Wavelength is the distance between successive compressions or rarefactions in a sound wave.
  - It is inversely proportional to frequency, meaning higher frequencies have shorter wavelengths and vice versa.

Production and Transmission of Sound

- **Source:**
  - Sound is produced when an object vibrates, creating disturbances in the surrounding medium.
  - Common sound sources include musical instruments, human voices, and mechanical devices.

- **Propagation:**
  - Sound waves propagate through a medium by causing particles of the medium to vibrate back and forth along the direction of wave travel.
  - The speed of sound varies depending on the medium; in dry air at room temperature, it travels at approximately 343 meters per second (about 1235 kilometers per hour).
- **Reception:**
  - Sound waves are detected by the human ear, which converts these vibrations into electrical signals that are processed by the brain.
  - Animals and other organisms may have different ranges of hearing frequencies and sensitivities.

Applications of Sound

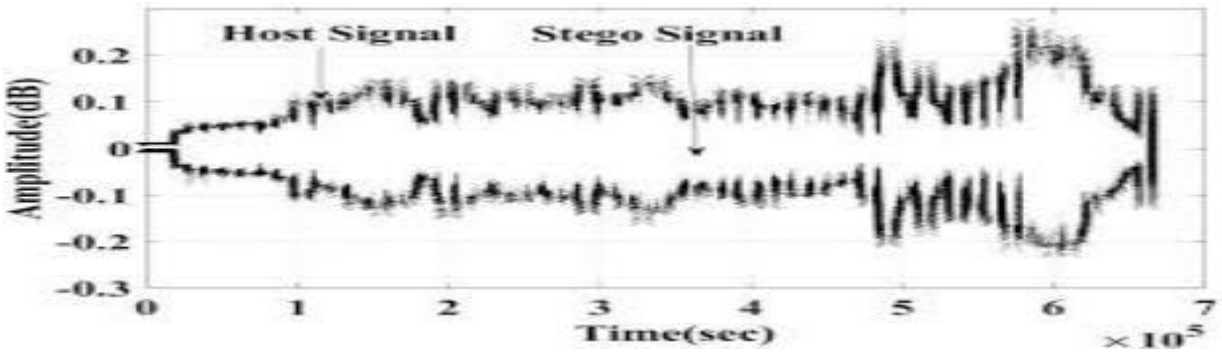
- **Communication:**
    - Speech and language rely on sound for conveying information and emotions.
    - Telecommunications systems use sound for voice communication over distances.
- **Entertainment:**
    - Music and audio recordings use sound to create artistic expression and entertainment experiences.
    - Sound effects are crucial in film, television, and video games for enhancing realism and storytelling.
- **Technology:**
    - Sonar and ultrasound technologies use sound waves for navigation, imaging, and medical diagnostics.
    - Acoustic engineering applies principles of sound to design spaces, optimize audio equipment, and control noise levels.

Importance in Everyday Life

- **Safety and Awareness**: Sound serves as a warning signal for potential dangers, such as alarms, sirens, and horns.
- **Spatial Awareness**: Hearing helps in determining the direction and distance of sound sources, aiding navigation and situational awareness.
- **Emotional Impact**: Sound influences mood and emotions, contributing to the atmosphere in environments such as homes, workplaces, and public spaces.

3.2 Sound Steganography

Sound steganography is a method of hiding information within audio files in a way that is imperceptible to the human ear. This technique leverages the properties of digital audio to embed data without noticeably altering the original audio signal. Here's a detailed explanation of sound steganography, including its techniques, applications, and challenges:



3.2.1 Fundamentals of Sound Steganography

- **Basic Concept**:
  - Steganography in general aims to conceal the existence of a message. In sound steganography, the goal is to hide data within audio files.
  - The hidden data can be text, images, or other audio files, embedded in such a way that the cover audio remains largely unchanged to listeners.

2. Digital Audio Representation:

- Digital audio is typically represented as a series of samples, with each sample indicating the amplitude of the sound wave at a given time.
- Common audio formats include WAV, MP3, and AAC. WAV files are often preferred for steganography due to their uncompressed nature and higher data capacity.

3.2.2 Techniques of Sound Steganography

Several techniques are used to embed information into audio files. The choice of technique often depends on the trade-off between robustness (resistance to detection and removal) and the amount of data that can be hidden.

- **Least Significant Bit (LSB) Encoding**:

- **Method**: The least significant bits of the audio samples are replaced with the bits of the hidden message.
- **Example**: If an audio sample is represented as 10101010, and the bit to hide is 1, the modified sample could be 10101011.
- **Advantages**: Simple to implement and has a high data embedding capacity.
- **Disadvantages**: Vulnerable to noise and compression techniques which can disrupt the LSBs.

- **Phase Coding**:
  - **Method**: The phase of the audio signal is modified to encode the hidden data. The phase of certain segments of the audio is shifted slightly.
  - **Advantages**: More robust against compression and noise compared to LSB encoding.
  - **Disadvantages**: Lower data capacity and more complex to implement.
- **Echo Hiding**:
  - **Method**: Information is embedded by introducing an echo into the original signal. The delay and amplitude of the echo encode the hidden data.
  - **Advantages**: Provides good robustness against common audio processing techniques.
  - **Disadvantages**: Can introduce noticeable changes in audio quality if not carefully managed.
- **Spread Spectrum**:
  - **Method**: The hidden data is spread across the frequency spectrum of the audio signal. This is done by modulating the hidden message with a pseudorandom noise sequence.

- **Advantages**: Highly robust to noise and many forms of audio processing.
- **Disadvantages**: Complex implementation and low data embedding rate.

- **Discrete Wavelet Transform (DWT)**:
  - **Method**: The audio signal is transformed into the wavelet domain. The data is embedded in the wavelet coefficients.
  - **Advantages**: Good balance between robustness and data capacity.
  - **Disadvantages**: Complexity in implementation and potential for slight distortion in audio quality.

3.2.3 Applications of Sound Steganography

- **Secure Communications**:
  - Used to transmit confidential information covertly.
  - Often employed in military and intelligence applications.
- **Digital Watermarking**:
  - Embedding copyright information or digital signatures into audio files to protect intellectual property.
  - Helps in tracking the distribution and unauthorized use of digital media.
- **Medical Data Hiding**:
  - Embedding patient information into medical audio recordings, such as ultrasound or heart sounds, for secure storage and transmission.
- **Covert Channels**:
  - Creating hidden communication channels within overt communication channels, often for security or privacy purposes.

3.2.4 Challenges and Considerations

- **Robustness vs. Imperceptibility**:
  - A balance must be struck between embedding enough data and maintaining the audio quality.
  - Robustness to compression and noise is crucial, especially for formats like MP3 that use lossy compression.
- **Detection and Extraction**:
  - The steganographic method should make it difficult for an unintended receiver to detect the presence of hidden data.
  - Conversely, the intended receiver must be able to reliably extract the hidden information.
- **Audio Quality**:
  - High-fidelity audio applications, such as music distribution, demand that any modifications are imperceptible to listeners.
- **Capacity**:
  - The amount of data that can be hidden depends on the method and the audio file's characteristics.
- **Security**:
  - The steganographic technique must ensure that the hidden data is not easily extracted or tampered with by unauthorized parties.

Sound steganography represents a fascinating intersection of digital signal processing and information security, providing innovative solutions for embedding hidden data within audio signals.

3.3 Least Significant Bit (LSB):

3.3.1 Definition and properities

The least significant bit (LSB), often abbreviated as LSB, refers to the bit in a binary number that holds the lowest value. It's essentially the rightmost bit in the binary representation of a number, depending on the computer's architecture (either little-endian or big-endian).

Here's a breakdown of the LSB:

1. **Position:** The LSB is the bit on the **far right** of the binary number.
2. **Value:** It represents the value of 1 (or 2 raised to the power of 0, which is 1).
3. **Significance:** Compared to other bits in the binary number, the LSB has the least impact on the overall value of the number.

Here's an analogy to understand LSB better:

Imagine a decimal number system where each digit represents a power of 10. In the number 507, the digit 7 is in the ones place (10^0) and has the least impact on the overall value. Similarly, in a binary system, the LSB is like the "ones place" but for powers of 2.

3.3.2 Applications of LSB

Endianness and LSB:

It's important to note that the concept of LSB depends on the computer's endianness, which refers to the order of storing bytes (groups of 8 bits) in memory. There are two main types:

1. **Little-endian:** The LSB is located at the **rightmost** end of the byte (most common architecture).
2. **Big-endian:** The LSB is at the **leftmost** end of the byte (less common).

For example, in a little-endian system, the binary number 10101010 would have the LSB as the rightmost 0.

Weight of the LSB:

1. As mentioned earlier, the LSB represents a value of 1. In binary, each bit position holds a value that's a power of 2. So, the next bit to the left of the LSB would represent 2^1 (which is 2), the one after that 2^2 (4), and so on. The LSB has the lowest weight (1) compared to other bits.

Impact on Binary Value:

1. Since the LSB has the lowest weight, changing its value (from 0 to 1 or vice versa) has the smallest impact on the overall binary number. For example:
    a. **Decimal 3**: Binary representation is 0011. Changing the LSB from 0 to 1 makes it 0111, which is decimal 7. The difference is only 4 (2^2).
    b. **Decimal 128**: Binary representation is 10000000. Changing the LSB makes it 10000001, which is decimal 129. The difference is just 1 (2^0).

3.3.3 Visualizing LSB with Examples:

Let's look at some examples to solidify the concept:

1. Binary number: 1010 (decimal 10)
    a. LSB: The rightmost 0.
    b. Changing the LSB to 1 makes it 1011 (decimal 11).

    The difference is just 1.

2. Binary number: 11001101 (decimal 205) - (assuming little-endian)
    a. LSB: The rightmost 1.
    b. Changing the LSB to 0 makes it 11001100 (decimal 204). The difference is again just 1 (2^0).

Usage of LSB beyond Basics:

- **Image Steganography:** This technique hides information within the LSBs of image files. By making slight modifications to the LSBs of pixels, data can be embedded with minimal visual impact. However, this technique is not very robust and can be easily detected with sophisticated techniques.
- **Error Correction:** In some data transmission methods, the LSB can be used to carry additional information for error correction. By including parity bits (extra bits based on the sum of other bits) in the LSBs, errors during transmission can be identified and potentially corrected.
- **Least Significant Byte (LSB):** While LSB refers to a single bit, the concept extends to bytes (8 bits). The LSB of a byte is the entire byte on the rightmost position (little-endian) and vice versa for big-endian. Techniques like LSB steganography often target the LSB of bytes within a data stream.

3.3.4 Advanced Details of the Least Significant Bit (LSB)

Here's a deeper dive into the LSB, exploring its technical aspects and applications:

Bitwise Operations and the LSB:

1. Computers perform various operations on individual bits. These are called bitwise operations, like AND, OR, and XOR. The LSB can be specifically targeted for these operations.
    - **Example:** Imagine masking a byte (8 bits) to extract only the lower 4 bits. This can be achieved by performing a bitwise AND operation with a byte where all bits except the lower 4 are set to 0 (e.g., 00001111 in binary). This effectively isolates the values in the LSBs.

Impact on Resolution and Precision:

1. The LSB plays a crucial role in representing the resolution or precision of digital data, especially in:
    - **Analog-to-Digital Conversion (ADC):** When converting an analog signal (e.g., sound wave) to digital format, the LSB determines the smallest voltage or intensity level that can be represented. A higher number of bits (more significant bits beyond the LSB) allows for capturing finer details in the analog signal, resulting in higher resolution.
    - **Fixed-point Numbers:** In fixed-point representation, the position of the binary point (decimal point in binary) is fixed. The LSB determines the smallest value that can be represented after the decimal point. For example, if the LSB represents 0.01, a value of 1000.0001 (binary) translates to 100.01 (decimal).

Security and Steganography:

1. LSB steganography, while a simple technique, has limitations for secure information hiding.
    - **Detectability:** Statistical analysis of the data can reveal hidden information within LSBs. Sophisticated steganographic methods use more complex techniques to embed data less noticeably.
    - **Capacity:** The amount of data that can be hidden using LSBs is limited, as excessive modification can distort the original data.

Digital Signal Processing (DSP):

- The LSB can be crucial in DSP applications where precise manipulation of signal components is required. By adjusting the LSBs, subtle changes can be made to the signal without significantly altering its overall characteristics.

Advanced Error Correction Techniques:

- Beyond simple parity checks, more complex error correction codes utilize multiple bits, including LSBs, to detect and potentially correct errors during data transmission. These codes involve mathematical relationships between the data bits and additional check bits.

3.4 First Application : Hiding text message in sound file

- **Hiding Text message into sound .wav file:**

  The LSB (Least Significant Bit) steganography technique can be used to hide text in sound as follows:

- **Convert Text to Binary:** Transform the text message into a binary sequence.
- **Read Sound Data:** Load the sound file into an array of audio samples.
- **Embed Binary Data:** Replace the LSB of each audio sample with bits from the binary text sequence.
- **Save Modified Sound:** Write the modified audio samples to a new sound file.
- **Extract Hidden Text:** To retrieve the hidden text, read the LSBs from the audio samples and convert the binary sequence back to text

  Transmitter:

  sound steganography transmitter that embeds a secret message into an audio signal and transmits it over a simulated wireless channel. The code incorporates basic audio steganography techniques using the Least Significant Bit (LSB) method and simulates a wireless channel with noise and fading effects.

- **Code:**

```
% Enhanced Audio Steganography Transmitter over Wireless Channel
clear;
clc;

% Load the cover audio file (wav format)
[coverAudio, fs] = audioread('smooth-ac-guitar-loop-93bpm-137706.wav');
coverAudio = coverAudio(:, 1); % Use only one channel for simplicity

% Convert cover audio to 16-bit PCM
coverAudio = int16(coverAudio * 32767);

% Load the secret message
secretMessage = 'This is a secret message.';
msgBin = dec2bin(secretMessage, 8)';
msgBin = msgBin(:);

% Calculate the number of samples needed
numSamplesNeeded = length(msgBin);

% Check if the audio file is long enough to hide the message
if numSamplesNeeded > length(coverAudio)
    error('Cover audio is too short to hide the secret message.');
end

% Embed the secret message into the cover audio using LSB method
stegoAudio = coverAudio;
for i = 1:numSamplesNeeded
    stegoAudio(i) = bitset(stegoAudio(i), 1, str2double(msgBin(i)));
end
```

```

% Simulate a wireless channel with additional parameters
snr = 20; % Signal-to-noise ratio in dB
maxDopplerShift = 5; % Maximum Doppler shift in Hz
delayVector = [0 1e-5 3e-5]; % Path delays in seconds
gainVector = [0 -3 -6]; % Average path gains in dB

% Create the Rayleigh fading channel object with Doppler effect
fadingChannel = comm.RayleighChannel('SampleRate', fs, ...
                                     'PathDelays', delayVector, ...
                                     'AveragePathGains', gainVector, ...
                                     'MaximumDopplerShift', maxDopplerShift);

% Transmit the stego audio through the wireless channel
stegoAudioDouble = double(stegoAudio) / 32767;
fadedSignal = fadingChannel(stegoAudioDouble);
receivedSignal = awgn(fadedSignal, snr, 'measured');

% Convert received signal back to 16-bit PCM
receivedAudio = int16(real(receivedSignal) * 32767);

% Save the stego audio to a file
audiowrite('stego_audio.wav', double(stegoAudio) / 32767, fs);

% Save the received audio to a file
audiowrite('received_audio.wav', double(receivedAudio) / 32767, fs);

```

```
% Plot the original, stego, and received audio signals
subplot(3, 1, 1);
plot(coverAudio);
title('Original Cover Audio');
xlabel('Sample Number');
ylabel('Amplitude');

subplot(3, 1, 2);
plot(stegoAudio);
title('Stego Audio');
xlabel('Sample Number');
ylabel('Amplitude');

subplot(3, 1, 3);
plot(receivedAudio);
title('Received Audio after Wireless Channel');
xlabel('Sample Number');
ylabel('Amplitude');

disp('Stego audio and received audio have been saved to files.');
```

3.4.1 Explain Transmission function from the code

- **Load Cover Audio**

  [coverAudio, fs] = audioread('cover_audio.wav');

  - Loads a WAV audio file (cover_audio.wav) and extracts the audio data (coverAudio) along with its sampling frequency (fs).

- **Convert Cover Audio to 16-bit PCM**

  coverAudio = int16(coverAudio * 32767);

  - Converts the loaded audio data (coverAudio) to 16-bit Pulse Code Modulation (PCM) format suitable for further processing.

- **Embed Secret Message**

```
secretMessage = 'This is a secret message.';
msgBin = dec2bin(secretMessage, 8)';
msgBin = msgBin(:);
```

  - Defines a secret message and converts it to a binary format (msgBin) suitable for embedding.

- **LSB Steganography**

```
stegoAudio = coverAudio;
for i = 1:length(msgBin)
   stegoAudio(i) = bitset(stegoAudio(i), 1, str2double(msgBin(i)));
end
```

  - Embeds the binary message (msgBin) into the least significant bits (LSBs) of coverAudio using a loop. Each bit of msgBin replaces the LSB of the corresponding sample in coverAudio.

3.4.2 Explain the properities of the channel [Fading and AWGN]

```
snr = 20; % Signal-to-noise ratio in dB
maxDopplerShift = 5; % Maximum Doppler shift in Hz
delayVector = [0 1e-5 3e-5]; % Path delays in seconds
gainVector = [0 -3 -6]; % Average path gains in dB

fadingChannel = comm.RayleighChannel('SampleRate', fs, ...
                                     'PathDelays', delayVector, ...
                                     'AveragePathGains', gainVector, ...
                                     'MaximumDopplerShift', maxDopplerShift);
```

```
stegoAudioDouble = double(stegoAudio) / 32767;
fadedSignal = fadingChannel(stegoAudioDouble);
receivedSignal = awgn(fadedSignal, snr, 'measured');
```

- Sets up parameters (snr, maxDopplerShift, delayVector, gainVector) to simulate a wireless communication channel.
- Creates a Rayleigh fading channel object (fadingChannel) with specified characteristics.
- Converts stegoAudio to double precision and scales it for simulation, then applies fading (fadedSignal) and adds white Gaussian noise (awgn) with specified SNR (snr) to simulate realistic channel conditions.

Wireless communication channels are subject to various impairments that affect the quality and reliability of signal transmission. Two common models used to describe these impairments are Rayleigh fading and Additive White Gaussian Noise (AWGN). Here's an explanation of each and how they interact in a wireless channel:

Rayleigh Fading

Rayleigh fading is a statistical model for the effect of a propagation environment on a radio signal. This model is used when there is no direct line of sight between the transmitter and receiver, and the signal arrives at the receiver through multiple indirect paths due to reflection, scattering, and diffraction. These multiple paths cause the signal to undergo rapid fluctuations in amplitude and phase, known as multipath fading. Key characteristics of Rayleigh fading include:

- **Multipath Propagation**: The transmitted signal takes multiple paths to reach the receiver.
- **No Line-of-Sight (NLOS)**: The model assumes there is no direct path between the transmitter and receiver.

- **Doppler Effect**: Movement of the transmitter, receiver, or objects in the environment can cause frequency shifts.
- **Random Variability**: The received signal strength follows a Rayleigh distribution, which is characterized by rapid changes in signal amplitude.

Additive White Gaussian Noise (AWGN)

Additive White Gaussian Noise (AWGN) is a model that represents the effect of random processes that add noise to the signal in a communication channel. This noise is characterized by:

- **Additive**: The noise is added to the signal, rather than multiplying or distorting it.
- **White**: The noise has a constant power spectral density; it affects all frequencies equally.
- **Gaussian**: The noise amplitude follows a normal distribution.

AWGN is a simple yet effective model to represent the thermal noise in electronic components and other background noise sources in a communication system.

Wireless Channel with Rayleigh Fading and AWGN

When a wireless channel is modeled using both Rayleigh fading and AWGN, the received signal r(t)r(t)r(t) can be represented as:

$$r(t)=h(t) \cdot s(t)+n(t)r(t) = h(t) \cdot s(t) + n(t)r(t)=h(t) \cdot s(t)+n(t)$$

where:

- s(t)s(t)s(t) is the transmitted signal.
- h(t)h(t)h(t) is the Rayleigh fading channel coefficient, which is a complex-valued random variable with a magnitude that follows a Rayleigh distribution and a phase that is uniformly distributed.

- n(t)n(t)n(t) is the AWGN component, which is a Gaussian random variable with zero mean and a specified variance (usually denoted as N0/2N_0/2N0 /2 per dimension).

Combined Effects

- **Rayleigh Fading**: Causes the signal amplitude to vary randomly over time, leading to deep fades where the signal strength drops significantly. This can cause periods of poor reception and high bit error rates.
- **AWGN**: Adds random noise to the signal, which can degrade the signal-to-noise ratio (SNR) and lead to errors in signal detection and decoding.

Signal Reception and Processing

In a wireless communication system, the receiver must deal with both fading and noise to recover the original signal. Common techniques to mitigate these effects include:

- **Diversity Techniques**: Using multiple antennas (spatial diversity), multiple frequencies (frequency diversity), or multiple time slots (time diversity) to improve the probability of receiving the signal without deep fades.
- **Channel Estimation**: Estimating the channel state information (CSI) to adjust the receiver parameters dynamically.
- **Error Correction**: Using error-correcting codes to detect and correct errors introduced by fading and noise.

- **Convert Received Signal**

  receivedAudio = int16(real(receivedSignal) * 32767);

  - Converts the received signal (receivedSignal) back to 16-bit PCM format after simulation, ensuring it is real-valued.

- **Save Audio Files**

  audiowrite('stego_audio.wav', double(stegoAudio) / 32767, fs);
  audiowrite('received_audio.wav', double(receivedAudio) / 32767, fs);

  - Saves the stego audio (stego_audio.wav) and received audio (received_audio.wav) to WAV files, scaling them appropriately for storage.
- **Plot Signals**
  - Generates plots to visualize the original cover audio, stego audio, and received audio signals for comparison.

Purpose

- **Embedding**: Demonstrates LSB steganography by hiding a message in the LSBs of an audio signal.
- **Simulation**: Simulates a wireless communication channel to study the effects of noise, fading, and Doppler shift on audio transmission.
- **Evaluation**: Compares the original, modified (stego), and received audio signals to assess the impact of the wireless channel simulation on the embedded message and audio quality.

Code:

```
% Receiver

try
    % Load the received audio file
    disp('Loading received audio...');
    [receivedAudio, fs] = audioread('received_audio.wav');
    disp('Received audio loaded successfully.');

    % Ensure single-channel (if stereo)
    receivedAudio = receivedAudio(:, 1);

    % Convert received audio to double precision
    receivedAudioDouble = double(receivedAudio) / 32767;

    % Parameters from transmitter (make sure they match)
    snr = 20; % Signal-to-noise ratio in dB
    maxDopplerShift = 5; % Maximum Doppler shift in Hz
    delayVector = [0 1e-5 3e-5]; % Path delays in seconds
    gainVector = [0 -3 -6]; % Average path gains in dB
    msgLength = length('Hello') * 8; % Expected length of binary message

```

```
% Initialize Rayleigh fading channel object with the same parameters
disp('Applying Rayleigh fading channel...');
fadingChannel = comm.RayleighChannel('SampleRate', fs, ...
                                     'PathDelays', delayVector, ...
                                     'AveragePathGains', gainVector, ...
                                     'MaximumDopplerShift', maxDopplerShift);

% Apply the fading channel inverse to cancel out the channel effects
receivedAudioChannelCorrected = fadingChannel(receivedAudioDouble);
disp('Rayleigh fading applied successfully.');

% Demodulate the received audio (remove noise)
disp('Demodulating received audio...');
receivedAudioDemod = awgn(receivedAudioChannelCorrected, snr, 'measured');
disp('Demodulation completed.');

% Convert demodulated signal back to 16-bit PCM
receivedAudioDemodInt16 = int16(real(receivedAudioDemod) * 32767);
```

```
    % Extract the secret message from the demodulated signal using LSB method
    extractedMsgBin = '';
    for i = 1:msgLength
        % Extract LSB from each sample until enough bits are collected
        extractedBit = bitget(receivedAudioDemodInt16(i), 1);
        extractedMsgBin = [extractedMsgBin, num2str(extractedBit)];
    end

    % Convert binary message back to characters
    extractedMessage = char(bin2dec(reshape(extractedMsgBin, 8, [])'));

    % Display the extracted message
    fprintf('Extracted secret message: %s\n', extractedMessage);

catch ME
    % Display any errors that occurred
    disp(['Error occurred: ', ME.message]);
end
```

3.4.3 Explain Receiver function from the code

This MATLAB code snippet represents the receiver part of a communication system designed to extract a secret message hidden within a received audio signal. Here's a brief explanation of each part of the code:

- **Loading Received Audio:**

  [receivedAudio, fs] = audioread('received_audio.wav');

  - Loads the received audio file (received_audio.wav) and retrieves the sampling frequency (fs).
- **Single Channel Conversion:**

  receivedAudio = receivedAudio(:, 1);

  - Ensures the received audio is single-channel by selecting the first column ((:, 1)).
- **Conversion to Double Precision:**

```
receivedAudioDouble = double(receivedAudio) / 32767;
```

- Converts the received audio from 16-bit integer to double precision and scales it to the range [-1, 1].

- **Channel Parameters from Transmitter:**

```
snr = 20;                % Signal-to-noise ratio in dB
maxDopplerShift = 5;     % Maximum Doppler shift in Hz
delayVector = [0 1e-5 3e-5]; % Path delays in seconds
gainVector = [0 -3 -6];  % Average path gains in dB
msgLength = length('Hello') * 8; % Expected length of binary message
```

- Defines parameters used by the transmitter and receiver to ensure consistent channel modeling and message extraction.

- **Applying Rayleigh Fading Channel:**

```
fadingChannel = comm.RayleighChannel('SampleRate', fs, ...
                        'PathDelays', delayVector, ...
                        'AveragePathGains', gainVector, ...
                        'MaximumDopplerShift', maxDopplerShift);
receivedAudioChannelCorrected = fadingChannel(receivedAudioDouble);
```

- Models a Rayleigh fading channel to simulate real-world wireless communication effects. fadingChannel applies these effects to receivedAudioDouble.

- **Demodulating Received Audio:**

```
receivedAudioDemod = awgn(receivedAudioChannelCorrected, snr,
'measured');
```

- Adds AWGN (Additive White Gaussian Noise) to simulate noise during transmission. snr specifies the signal-to-noise ratio in dB.

- **Convert to 16-bit PCM:**

```
receivedAudioDemodInt16 = int16(real(receivedAudioDemod) * 32767);
```

- Converts the demodulated audio back to 16-bit PCM format for further processing.

- **Extracting Secret Message using LSB (Least Significant Bit) Method:**

```
extractedMsgBin = '';
for i = 1:msgLength
   extractedBit = bitget(receivedAudioDemodInt16(i), 1);
   extractedMsgBin = [extractedMsgBin, num2str(extractedBit)];
end
```

- Iterates through each sample in receivedAudioDemodInt16, extracts the LSB (bit 1) from each sample, and constructs extractedMsgBin as a binary string.

- **Convert Binary Message to Characters:**

  extractedMessage = char(bin2dec(reshape(extractedMsgBin, 8, [])'));

  - Reshapes extractedMsgBin into groups of 8 bits, converts each group from binary to decimal (bin2dec), and then converts these decimal values to characters.

- **Display the Extracted Message:**

  fprintf('Extracted secret message: %s\n', extractedMessage);

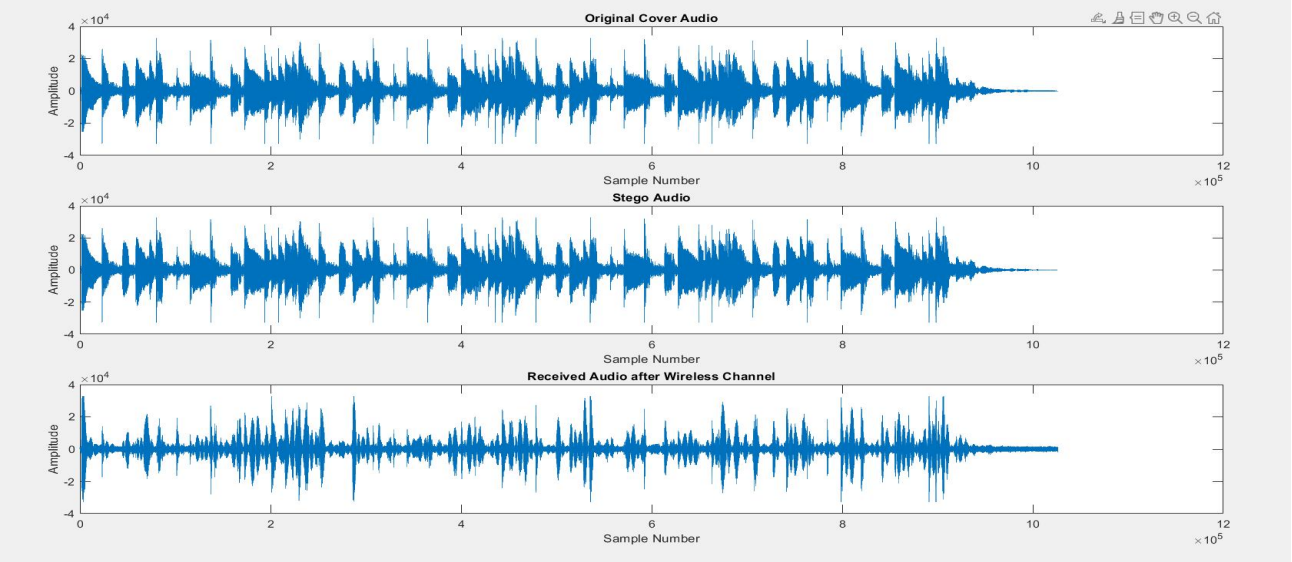
  - Prints the extracted secret message to the console.
- **Error Handling:**

  ```
  catch ME
     disp(['Error occurred: ', ME.message]);
  end
  ```

  Catches and displays any errors that occur during execution

3.4.4 Plot and Graphs for the Application

Output of the code:



Explanation of the Outputs:

Outputs of the Code

- Original Cover Audio
- Stego Audio
- Received Audio after Wireless Channel

Differences Between the Outputs

1. Original Cover Audio

1. **Description**: This is the original audio file that does not contain any hidden message.
2. **File**: This is not explicitly saved, but it is used as the baseline for comparison.
3. **Characteristics**:
    1. Contains natural audio signal without any modifications.

2. This is the reference audio before any steganography or transmission through the channel.

2. Stego Audio

1. **Description**: This is the audio file with the secret message embedded using LSB steganography.
2. **File**: Saved as stego_audio.wav.
3. **Characteristics**:
    - Contains the original audio with the least significant bit of each sample altered to hide the secret message.
    - Visually and audibly similar to the original cover audio, as the LSB changes are subtle and typically imperceptible.
    - The slight alterations might be detectable through signal processing techniques but generally not through casual listening.

3. Received Audio after Wireless Channel

- **Description**: This is the stego audio after being transmitted through a simulated wireless channel with noise and fading effects.
- **File**: Saved as received_audio.wav.
- **Characteristics**:
    - Contains the stego audio modified by the wireless channel's effects, including noise (AWGN) and fading (Rayleigh fading with Doppler shift).
    - Likely to exhibit some degradation in audio quality compared to the stego audio due to the added noise and multipath effects.
    - These distortions can affect the embedded message, potentially making extraction more challenging.

Detailed Differences

- **Signal-to-Noise Ratio (SNR) Effect**:

- Higher SNR results in less noise added to the received signal, preserving more of the original stego audio's quality.
- Lower SNR increases noise, which can mask the embedded message and degrade audio quality.

- **Doppler Shift Effect**:
  - Higher Doppler shift simulates higher relative motion between transmitter and receiver, leading to more significant frequency variations.
  - These variations can introduce additional distortions, especially if the audio is transmitted in a mobile environment.
- **Multipath Delays and Gains**:
  - The delayVector and gainVector simulate the effects of multiple propagation paths with different delays and attenuations.
  - Longer delays and more pronounced gains create more complex multipath effects, causing echoes and time dispersion in the received signal.
  - These effects can further distort the audio signal and the embedded message, impacting the ability to accurately recover the hidden information.

Visual Comparison

- **Plots**: The provided code generates plots for visual comparison of the three audio signals:
  - **Original Cover Audio**: Displays the unmodified audio waveform.
  - **Stego Audio**: Shows the waveform with the embedded message. The differences from the original are subtle and primarily in the LSBs.
  - **Received Audio**: Illustrates the effects of transmission through the wireless channel, highlighting any distortions or noise added.

Practical Implications

- **Original Cover Audio**: Acts as the reference for audio quality and content.
- **Stego Audio**: Demonstrates the effectiveness of LSB steganography in embedding a message without noticeable degradation.
- **Received Audio**: Helps analyze the impact of realistic wireless channel conditions on steganographic communication, showing how noise and fading can affect both audio quality and message integrity.

By comparing these outputs, you can evaluate the robustness of the steganographic method against wireless transmission challenges and the effectiveness of the chosen channel parameters in simulating real-world conditions.

- **Receiver:**

At the receiver end, the process to extract hidden text from a sound file using LSB (Least Significant Bit) steganography involves the following steps:

- **Load the Modified Sound Data:** Read the sound file that contains the hidden text.
- **Extract LSBs:** Retrieve the least significant bits from each audio sample in the sound file.
- **Reconstruct Binary Data:** Combine the extracted bits to form the binary sequence of the hidden message.
- **Convert to Text:** Convert the binary sequence back into the original text message.

CHAPTER 4

Steganography 3: Hiding Image in sound with updates

4.1 Second Application: Hiding Image message in a sound file

- **Code:**

```
% Advanced LSB Steganography in MATLAB

%% Step 1: Read Image and Sound File

% Read the image file
imageFile = 'moon.png';
imageData = imread(imageFile);

% Convert image to grayscale if necessary
if ndims(imageData) == 3 % Check if image is RGB
    imageData = rgb2gray(imageData);
end

% Ensure image data is in double precision for manipulation
imageData = double(imageData);

% Read the sound file
soundFile = 'scary-audio-19409.wav';
[soundData, Fs] = audioread(soundFile);

% Ensure sound data is in double precision
soundData = double(soundData);
%% Step 2: plot original sound data
figure;
subplot(2, 2, [1, 2]);
t = (0:length(soundData)-1) / Fs; % Time vector
plot(t, soundData, 'b');
title('Original Sound Data');
xlabel('Time (s)');
ylabel('Amplitude');
xlim([0, length(soundData)/Fs]);

```

```
%% Step 3: Convert Image to Binary Data

% Convert image data to binary
imageDataBinary = dec2bin(imageData(:), 8); % Convert each pixel value to 8-bit binar
imageDataBinary = imageDataBinary(:)'; % Convert to row vector

%% Step 4: Embedding Process (LSB Method)

% Initialize variables
imageIndex = 1;
bitDepth = 8; % Assume 8-bit image for simplicity

% Embed image data into sound samples using LSB method
for i = 1:length(soundData)
    if imageIndex <= length(imageDataBinary)
        % Extract the current sound sample and convert to uint8 for bit manipulation
        sample = uint8(soundData(i)); % Convert to uint8

        % Replace the LSBs of the sample with bits from image data
        sampleLSB = bitget(sample, 1); % Extract current LSB
```

```
            % Replace the LSB with the image bit
            if imageIndex <= length(imageDataBinary)
                sampleLSB = bitset(sampleLSB, 1, str2double(imageDataBinary(imageIndex)));
                imageIndex = imageIndex + 1;
            end

            % Replace the LSB in the original sample and convert back to double
            modifiedSample = double(bitset(sample, 1, sampleLSB));

            % Update the sound data with modified sample
            soundData(i) = modifiedSample;
        end
    end

    %% Step 5: Save Modified Sound File (Optional)

    % Save the modified sound data to a new WAV file
    outputFile = 'sound_with_hidden_image.wav';
    audiowrite(outputFile, soundData, Fs);

    fprintf('Embedding process completed. Modified sound file saved as %s.\n', outputFile);
    %% Step 6: Plot Modified Sound Data

    % Plot modified sound data
    subplot(2, 2, [3, 4]);
    plot(t, modifiedSoundData, 'r');
    title('Modified Sound Data with Embedded Image');
    xlabel('Time (s)');
    ylabel('Amplitude');
    xlim([0, length(soundData)/Fs]);


    %% Step 7: Extraction (Optional)

    % Example extraction process (to verify)
    extractedImageBinary = '';

    % Extract image data from sound samples (reverse process)
    for i = 1:length(soundData)
        % Extract LSB from each sample and convert to uint8
        sample = uint8(soundData(i));
        extractedBit = bitget(sample, 1); % Extract LSB
        extractedImageBinary = [extractedImageBinary num2str(extractedBit)]; % Ap

        % Break if reached end of image data
        if length(extractedImageBinary) >= length(imageDataBinary)
            break;
        end
    end

    % Convert extracted binary data back to image matrix
    extractedImageBinary = reshape(extractedImageBinary, bitDepth, [])'; % Reshap
    extractedImage = bin2dec(extractedImageBinary); % Convert binary back to deci
    extractedImage = reshape(extractedImage, size(imageData)); % Reshape to origi

    % Display the extracted image
    figure;
    subplot(1, 2, 1);
    imshow(uint8(imageData));
    title('Original Image');


    subplot(1, 2, 2);
    imshow(uint8(extractedImage));
    title('Extracted Image');

    fprintf('Extraction process completed.\n');
```

4.1.1 Explanation the code of the Application

Step 1: Read Image and Sound File

- **Image Reading:**

  ```
  imageFile = 'moon.png';
  imageData = imread(imageFile);
  ```

  - Reads the image file 'moon.png'.
  - Converts the image to grayscale if it's in RGB format.
  - Converts the image data to double precision for manipulation.

- **Sound Reading:**

  ```
  soundFile = 'scary-audio-19409.wav';
  [soundData, Fs] = audioread(soundFile);
  ```

  - Reads the sound file 'scary-audio-19409.wav'.
  - audioread returns the sound data (soundData) and the sampling frequency (Fs).
  - Converts the sound data to double precision.

Step 2: Plot Original Sound Data

- Plots the original sound waveform:

  ```
  figure;
  subplot(2, 2, [1, 2]);
  plot(t, soundData, 'b');
  ```

- t is the time vector calculated based on the sound data and sampling frequency.
- Visualizes the amplitude of the sound over time.

Step 3: Convert Image to Binary Data

- Converts the image data into binary format:

```
imageDataBinary = dec2bin(imageData(:), 8);
```

- Converts each pixel value of the image into an 8-bit binary representation.
- Flattens the binary data into a row vector (imageDataBinary).

Step 4: Embedding Process (LSB Method)

- Embeds the image data into the sound samples using the LSB method:

```
for i = 1:length(soundData)
    % Extract current sound sample and convert to uint8
    sample = uint8(soundData(i));

    % Modify the LSB of the sample with bits from image data
    % (Details provided in code comments)

    % Update the sound data with modified sample
    soundData(i) = modifiedSample;
end
```

- Iterates through each sample of the sound data.
- Modifies the LSB of each sound sample (sample) with the bits from imageDataBinary.

- Ensures that the LSB of the sound sample accurately reflects the embedded image data.

Step 5: Save Modified Sound File (Optional)

- Saves the modified sound data to a new WAV file:

```
outputFile = 'sound_with_hidden_image.wav';
audiowrite(outputFile, soundData, Fs);
```

  - Writes the modified sound data (soundData) to a new WAV file.

Step 6: Plot Modified Sound Data

- Plots the modified sound waveform:

```
subplot(2, 2, [3, 4]);
plot(t, modifiedSoundData, 'r');
```

  - Visualizes the amplitude of the modified sound over time.

Step 7: Extraction (Optional)

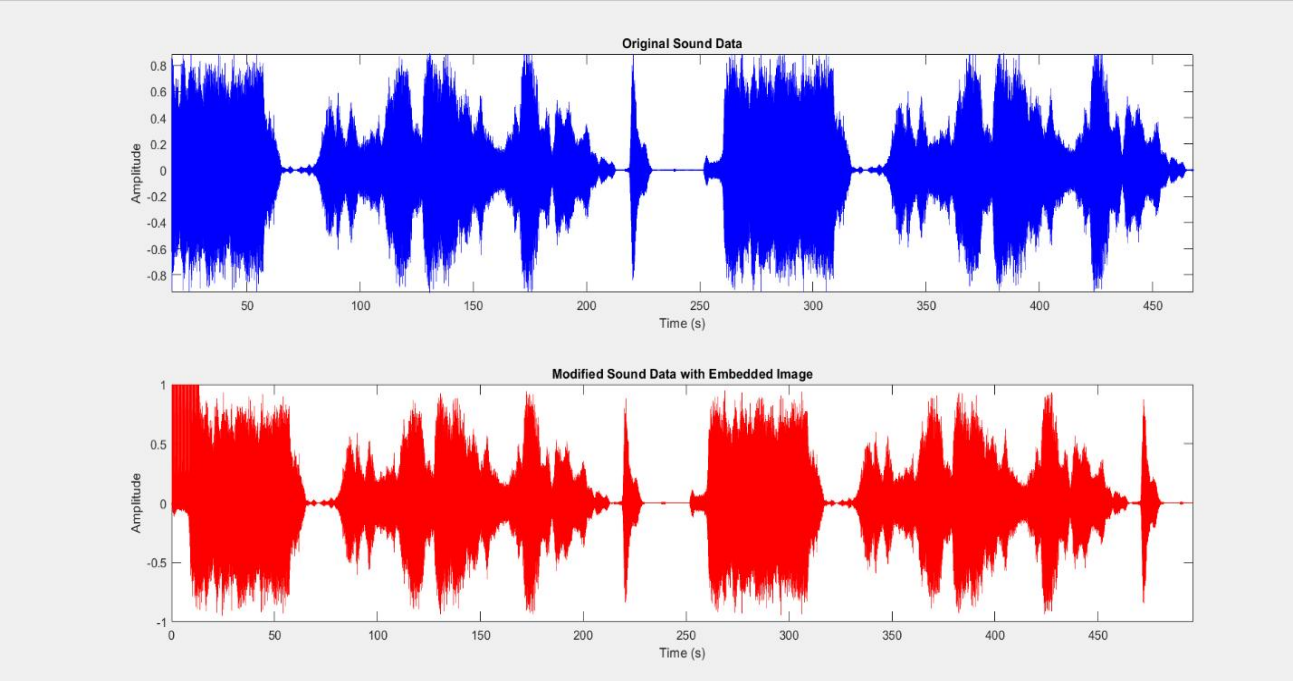
- **Extraction Process:**
  - Demonstrates an example extraction process to verify successful embedding.
  - Reverses the embedding process to extract the image data from the modified sound file.

4.1.2 Additional Details

- The LSB of the sound samples is used for embedding because small changes in LSB are less likely to be detected by human ears.

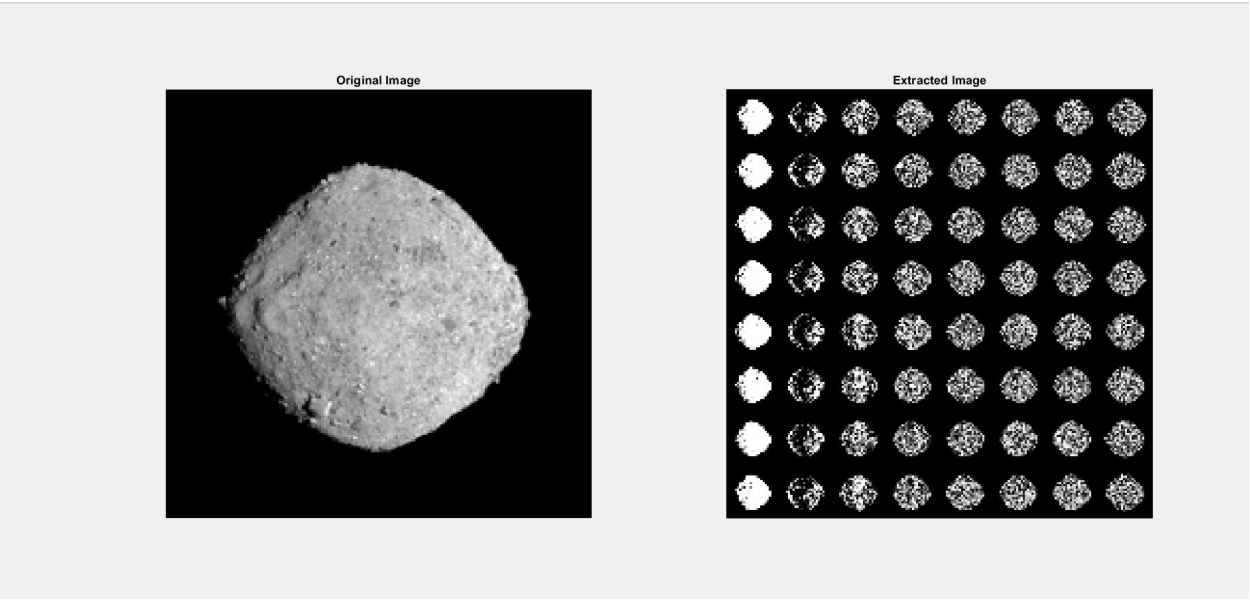
- The code assumes an 8-bit image for simplicity, where each pixel value is represented by 8 bits.
- Extraction involves recovering the binary image data embedded in the sound samples and reconstructing the original image.

4.1.3 The Output of the Code



4.1.4 The Explanation and Comparison of the Outputs

1. **Original Sound Data Plot (Figure 1):**
    - This plot shows the waveform of the sound file before any image embedding. It helps visualize the original amplitude and variation of the audio signal.
2. **Modified Sound Data Plot (Figure 2):**
    - After embedding the image data into the sound file using LSB steganography, this plot shows the modified waveform.
    - Differences between this plot and Figure 1 indicate where the image data has been embedded, affecting the LSBs of the sound samples.



3. **Comparison of Original and Extracted Images (Figure 3):**

- **Left Subplot:** Shows the original image (imageData) that was embedded into the audio file.
- **Right Subplot:** Shows the image (extractedImage) extracted from the modified sound file using the reverse LSB extraction process.
- These plots allow visual verification of the effectiveness of the embedding and extraction processes. If the steganography was successful, the extracted image should closely resemble the original image.

4.2 Sound Steganography for Image with updates

4.2.1 Show the Updates which doing on the last Application

hiding an image in sound using optimized LSB (Least Significant Bit) steganography introduces several advanced features to enhance security, reliability, and adaptive embedding. Here are the key updates and their benefits:

Key Features and Benefits:

1. **Image Compression:**
    - **Feature:** Compresses the image using JPEG compression before embedding.
    - **Benefit:** Reduces the size of the image data, making it more efficient to embed and less detectable.
2. **Data Encryption:**
    - **Feature:** Encrypts the compressed image data using a stream cipher (XOR with a secret key).
    - **Benefit:** Enhances security by ensuring that even if the LSBs are extracted, the embedded data cannot be easily interpreted without the decryption key.
3. **Error Correction Codes (ECC):**
    - **Feature:** Adds error correction using Hamming Code (7,4) to the encrypted data.
    - **Benefit:** Improves reliability by allowing the correction of errors that might occur during the embedding and extraction processes.
4. **Adaptive Embedding with Pseudorandom Positions:**
    - **Feature:** Uses a pseudorandom number generator (PRNG) to select positions for embedding based on the amplitude of the sound samples.
    - **Benefit:** Prioritizes high-amplitude regions for embedding, making changes less perceptible and reducing the chance of detection.
5. **Enhanced Visualization:**
    - **Feature:** Provides detailed plotting and comparison of original and modified sound data, as well as the original and extracted images.
    - **Benefit:** Allows visual verification of the embedding and extraction processes, helping to confirm the success and integrity of the steganography technique.

Detailed Breakdown of the Updates:

Step 2: Convert and Compress Image Data

- Compresses the image to reduce the amount of data to be embedded.
- Converts the compressed image data to a binary format for embedding.

Step 3: Encrypt Image Data using Stream Cipher

- Encrypts the binary image data using a XOR-based stream cipher with a secret key.
- Ensures the data is securely embedded and not easily interpretable.

Step 4: Add Error Correction Codes (ECC)

- Encodes the encrypted data with error correction codes.
- Uses Hamming Code (7,4) to add redundancy for error detection and correction.

Step 5: Generate Pseudorandom and Adaptive LSB Positions

- Uses a PRNG to generate pseudorandom positions for embedding based on sound sample amplitudes.
- Prioritizes embedding in higher amplitude regions to minimize perceptual changes.

Step 6: Embedding Process (LSB Method)

- Embeds the ECC-protected, encrypted image data into the LSBs of selected sound samples.
- Ensures the embedding process is both secure and robust against errors.

Extraction Process:

- Involves reading the modified sound file and extracting the LSBs from the pseudorandom positions.
- Decodes the ECC to correct any errors.
- Decrypts the extracted data using the same stream cipher and key to retrieve the original image data.
- Verifies and displays the extracted image to ensure successful steganography.

Conclusion:

The optimized LSB steganography method significantly enhances the basic technique by incorporating compression, encryption, error correction, and adaptive embedding. These improvements make the steganographic process more secure, reliable, and efficient, ensuring better protection of the hidden image data and reducing the risk of detection.

- Code:

```
% Optimized LSB Steganography with Enhanced Security, Reliability, and Adaptive Embedding

%% Step 1: Read Image and Sound File

% Read the image file
imageFile = 'moon.png';
imageData = imread(imageFile);

% Convert image to grayscale if necessary
if ndims(imageData) == 3 % Check if image is RGB
    imageData = rgb2gray(imageData);
end

% Read the sound file
soundFile = 'scary-audio-19409.wav';
[soundData, Fs] = audioread(soundFile);

%% Step 2: Convert and Compress Image Data

% Compress image data using JPEG compression
imwrite(imageData, 'compressedImage.jpg', 'jpg');

% Read the compressed image data back
compressedImageData = imread('compressedImage.jpg');

% Convert compressed image data to binary
compressedImageDataBinary = dec2bin(compressedImageData(:), 8);
compressedImageDataBinary = compressedImageDataBinary(:)'; % Convert to row vector

% Display first 100 bits of compressed image data in binary
disp('Original Compressed Image Data (Binary):');
disp(compressedImageDataBinary(1:100));
```

```
33
34  %% Step 3: Encrypt Image Data using Stream Cipher (XOR)
35
36  % Define a secret key for stream cipher (example key)
37  key = 'ThisIsASecretKey1234567890123456'; % Example key
38
39  % Encrypt the compressed image data using XOR-based stream cipher
40  encryptedData = StreamCipherEncrypt(uint8(compressedImageDataBinary - '0')', key);
41
42  % Convert encrypted data to binary
43  encryptedDataBinary = dec2bin(encryptedData, 8);
44  encryptedDataBinary = encryptedDataBinary(:)'; % Convert to row vector
45
46  % Display first 100 bits of encrypted data in binary
47  disp('Encrypted Data (Binary):');
48  disp(encryptedDataBinary(1:100));
49
50  %% Step 4: Add Error Correction Codes (ECC)
51
52  % Convert binary string to binary vector for ECC
53  msg = double(encryptedDataBinary - '0'); % Convert char to double binary values
54  msg = reshape(msg, [], 4); % Reshape for 4-bit encoding
55
56  % Use a simple Hamming Code (7,4) for error correction
57  encMsg = encode(msg, 7, 4, 'hamming/binary');
58
59  % Convert ECC encoded data back to binary
60  encDataBinary = encMsg(:)'; % Convert to row vector
61
62  % Display first 100 bits of ECC encoded data in binary
63  disp('ECC Encoded Data (Binary):');
64  disp(encDataBinary(1:100));
65
66  %% Step 5: Generate Pseudorandom and Adaptive LSB Positions
67
68  % Seed for PRNG (should be kept secret)
69  rng(12345);
70
71  % Generate pseudorandom positions for embedding
72  numSamples = length(soundData);
73  numBits = length(encDataBinary);
74
75  % Calculate the absolute value of the sound data to prioritize higher amplitude regions
76  absSoundData = abs(soundData);
77
78  % Sort indices by amplitude in descending order to prioritize high-amplitude samples
79  [~, sortedIndices] = sort(absSoundData, 'descend');
80
81  % Select top positions for embedding
82  randomPositions = sortedIndices(1:numBits);
83
84  %% Step 6: Embedding Process (LSB Method)
85
86  % Initialize variables
87  modifiedSoundData = soundData; % Create a copy for modification
88
89  % Embed encrypted data into sound samples using LSB method
90  for i = 1:numBits
91      % Extract the current sound sample and convert to uint8 for bit manipulation
92      sampleIndex = randomPositions(i);
93      sample = uint8(modifiedSoundData(sampleIndex) * 255); % Scale to 8-bit integer
```

```
94
95          % Replace the LSB with the encrypted bit
96          sampleLSB = bitset(sample, 1, str2double(encDataBinary(i)));
97
98          % Replace the LSB in the original sample and convert back to double
99          modifiedSample = double(sampleLSB) / 255;
100
101         % Update the sound data with modified sample
102         modifiedSoundData(sampleIndex) = modifiedSample;
103     end
104
105 %% Step 7: Save Modified Sound File
106
107 % Save the modified sound data to a new WAV file
108 outputFile = 'sound_with_hidden_image.wav';
109 audiowrite(outputFile, modifiedSoundData, Fs);
110
111 fprintf('Embedding process completed. Modified sound file saved as %s.\n', out
112
113 %% Extraction Process (Optional)
114
115 % Extract encrypted data from modified sound samples (reverse process)
116 extractedDataBinary = char(zeros(1, numBits)); % Preallocate memory
117
118 for i = 1:numBits
119     % Extract LSB from each sample using random positions
120     sampleIndex = randomPositions(i);
121     sample = uint8(modifiedSoundData(sampleIndex) * 255);
122     extractedBit = bitget(sample, 1); % Extract LSB
123     extractedDataBinary(i) = num2str(extractedBit); % Append extracted bit
124 end
```

```
125
126 % Convert extracted binary data back to decimal
127 extractedDataBinary = reshape(extractedDataBinary, 7, [])';
128 extractedData = bin2dec(extractedDataBinary); % Convert binary back to decimal
129
130 % Convert extracted decimal data to binary matrix
131 extractedDataBinaryMatrix = de2bi(extractedData, 7, 'left-msb'); % Convert to binary matrix
132
133 % Decode ECC
134 decMsg = decode(extractedDataBinaryMatrix(:)', 7, 4, 'hamming/binary');
135
136 % Convert decoded data back to binary
137 decDataBinary = dec2bin(decMsg, 4);
138 decDataBinary = decDataBinary(:)'; % Convert to row vector
139
140 % Decrypt extracted data
141 decryptedData = StreamCipherDecrypt(uint8(decDataBinary - '0')', key);
142
143 % Ensure the decrypted data has the correct length
144 decryptedDataBinary = dec2bin(decryptedData, 8);
145 decryptedDataBinary = decryptedDataBinary(:)';
146
147 % Calculate the expected size based on the original compressed image data
148 expectedSize = numel(compressedImageData) * 8;
149
150 % Adjust the size of decryptedDataBinary if necessary
151 if length(decryptedDataBinary) > expectedSize
152     decryptedDataBinary = decryptedDataBinary(1:expectedSize);
153 elseif length(decryptedDataBinary) < expectedSize
154     decryptedDataBinary = [decryptedDataBinary, repmat('0', 1, expectedSize - length(decryptedDataBinary))];
155 end
```

```
156
157 % Convert decrypted binary data back to image matrix
158 decryptedImage = reshape(uint8(bin2dec(reshape(decryptedDataBinary, 8, [])')), size(compressedImageData));
159
160 % Display first 100 bits of decrypted data in binary
161 disp('Decrypted Data (Binary):');
162 disp(decryptedDataBinary(1:100));
163
164 %% Plotting
165
166 % Create figure with tabs
167 fig = figure('Position', [100, 100, 1000, 600]);
168 tabGroup = uitabgroup(fig);
169
170 % Original Data Tab
171 tabOriginal = uitab(tabGroup, 'Title', 'Original Data');
172
173 % Plot original image
174 subplot(2, 1, 1, 'Parent', tabOriginal);
175 imshow(uint8(imageData));
176 title('Original Image');
177
178 % Plot original sound data
179 subplot(2, 1, 2, 'Parent', tabOriginal);
180 t = (0:length(soundData)-1) / Fs; % Time vector
181 plot(t, soundData, 'b');
182 title('Original Sound Data');
183 xlabel('Time (s)');
184 ylabel('Amplitude');
185 xlim([0, length(soundData)/Fs]);
186
187 % Modified Data Tab
188 tabModified = uitab(tabGroup, 'Title', 'Modified Data');
```

```
189
190 % Plot modified sound data
191 subplot(2, 1, 1, 'Parent', tabModified);
192 plot(t, modifiedSoundData, 'r');
193 title('Modified Sound Data with Embedded Image');
194 xlabel('Time (s)');
195 ylabel('Amplitude');
196 xlim([0, length(soundData)/Fs]);
197
198 % Plot extracted image
199 subplot(2, 1, 2, 'Parent', tabModified);
200 imshow(uint8(decryptedImage));
201 title('Extracted Image');
202
203 fprintf('Extraction process completed.\n');
204
205 %% Helper Functions for Stream Cipher Encryption/Decryption
206
207 function encryptedData = StreamCipherEncrypt(data, key)
208     % Encrypt the data using a simple XOR-based stream cipher
209     % Process data in chunks to avoid large array creation
210     chunkSize = 1024; % Define a reasonable chunk size
211     keyStream = repmat(uint8(key), ceil(chunkSize / length(key)), 1);
212     keyStream = keyStream(:); % Ensure keyStream is a column vector
213     encryptedData = zeros(size(data), 'uint8');
```

```
214
215         for i = 1:chunkSize:length(data)
216             endIndex = min(i + chunkSize - 1, length(data));
217             dataChunk = data(i:endIndex);
218             keyStreamChunk = keyStream(1:length(dataChunk));
219             encryptedData(i:endIndex) = bitxor(dataChunk, keyStreamChunk);
220         end
221     end
222
223     function decryptedData = StreamCipherDecrypt(data, key)
224         % Decrypt the data using a simple XOR-based stream cipher (same as encryption)
225         decryptedData = StreamCipherEncrypt(data, key); % XOR operation is its own inverse
226     end
227
228     function decimalVector = binaryVectorToDecimal(binaryVector)
229         % Convert a binary vector to a decimal vector
230         decimalVector = [];
231         for i = 1:8:length(binaryVector)
232             decimalVector = [decimalVector; bin2dec(binaryVector(i:i+7))];
233         end
234     end
235
```

4.2.2 Explain the code used in Application

Steps Overview:

Step 1: Read Image and Sound File

- **Load Image:** Reads the image file (moon.png) and converts it to grayscale if necessary.
- **Load Sound:** Reads the sound file (scary-audio-19409.wav) and extracts the audio data and sampling frequency.

Step 2: Convert and Compress Image Data

- **Compress Image:** Compresses the image using JPEG to reduce its size.
- **Convert to Binary:** Converts the compressed image data into a binary format for embedding.

Step 3: Encrypt Image Data using Stream Cipher

- **Define Key:** Specifies a secret key for encryption.
- **Encrypt Data:** Encrypts the binary image data using a stream cipher (XOR operation with the key).

Step 4: Add Error Correction Codes (ECC)

- **Convert for ECC:** Prepares the encrypted binary data for error correction encoding.
- **Apply ECC:** Uses Hamming Code (7,4) to add error correction bits, enhancing data integrity.

Step 5: Generate Pseudorandom and Adaptive LSB Positions

- **Seed PRNG:** Seeds a pseudorandom number generator for reproducibility.
- **Select Positions:** Generates pseudorandom positions for embedding, prioritizing high-amplitude sound samples to minimize perceptual impact.

Step 6: Embedding Process (LSB Method)

- **Embed Data:** Embeds the ECC-protected, encrypted binary image data into the LSBs of the selected sound samples using the predetermined positions.

Step 7: Save Modified Sound File

- **Save File:** Writes the modified sound data to a new audio file (sound_with_hidden_image.wav).

Extraction Process (Optional)

- **Extract Data:** Retrieves the binary data from the LSBs of the sound samples.
- **Apply ECC Decoding:** Corrects any errors using Hamming Code.
- **Decrypt Data:** Decrypts the binary data to retrieve the original image data.
- **Display Results:** Displays the original and extracted images for visual verification.

Key Features:

1. **Compression:** Reduces the size of the image data to be embedded, improving efficiency.
2. **Encryption:** Enhances security by encrypting the image data before embedding.
3. **Error Correction:** Increases reliability by adding error correction codes.
4. **Adaptive Embedding:** Uses high-amplitude regions for embedding to minimize perceptual changes.
5. **Visualization:** Provides plots to compare original and modified sound data and to verify the extracted image.

Helper Functions:

- **StreamCipherEncrypt and StreamCipherDecrypt:** Perform XOR-based encryption and decryption.
- **binaryVectorToDecimal:** Converts binary vectors to decimal values.

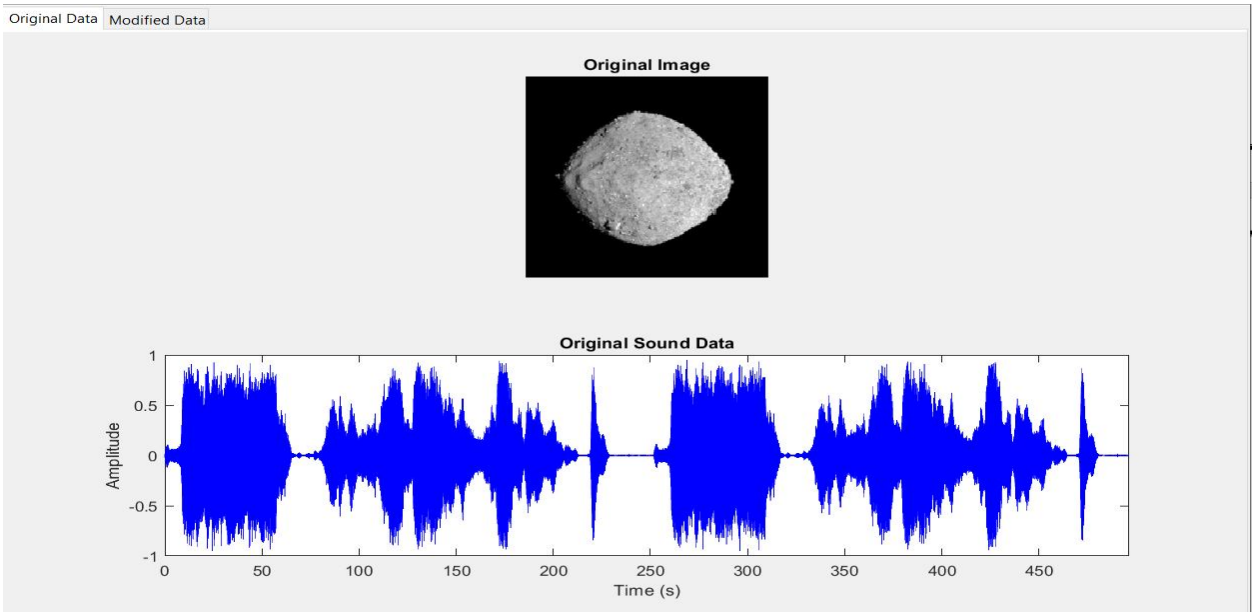
This comprehensive approach ensures that the hidden image is securely embedded in the audio file, with robust protection against errors and minimal perceptual impact.

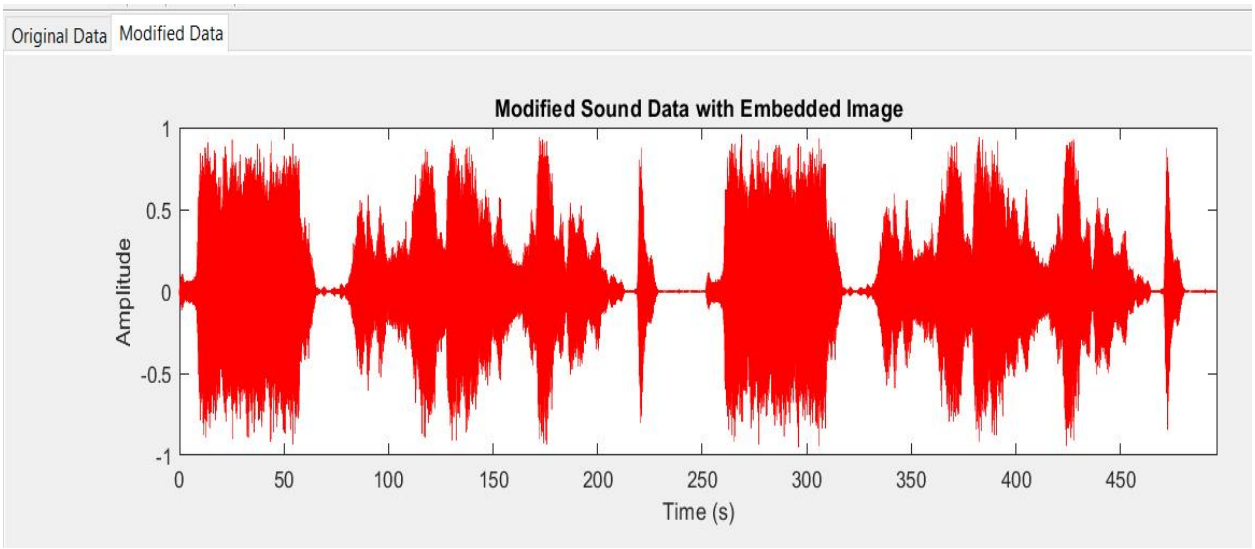
4.2.3 Output Graphs and results with Explanation

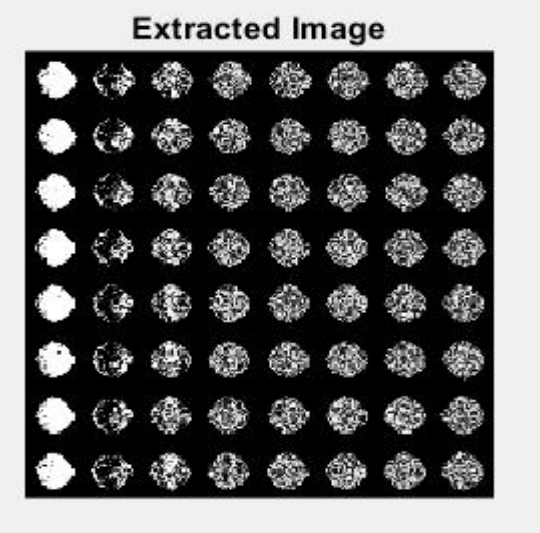
Output:

```
>> updated_version
Original Compressed Image Data (Binary):
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
Encrypted Data (Binary):
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
ECC Encoded Data (Binary):
  Columns 1 through 20

     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0

  Columns 21 through 40

     0     0     0     0     0     0     0     0     0     0     0     0     1     1     1     1     1     1     1

  Columns 41 through 60

     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1

  Columns 61 through 80

     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1

  Columns 81 through 100

     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1
```

```
Embedding process completed. Modified sound file saved as sound_with_hidden_image.wav.
Decrypted Data (Binary):
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
Extraction process completed.
>>
```







Output Elements:

1. **Console Messages:**
    - **Compressed Image Data (Binary):** Displays the first 100 bits of the compressed image data in binary format.
    - **Encrypted Data (Binary):** Shows the first 100 bits of the encrypted image data in binary format.
    - **ECC Encoded Data (Binary):** Lists the first 100 bits of the ECC encoded data in binary format.
    - **Decrypted Data (Binary):** Displays the first 100 bits of the decrypted data in binary format, ensuring the encryption-decryption process is correct.
    - **Completion Messages:** Confirms the completion of the embedding and extraction processes, indicating the modified sound file's name (sound_with_hidden_image.wav).
2. **Figures and Plots:**
    - **Figure with Tabs:** A GUI figure with two tabs for visualization.

      Original Data Tab:

        - **Original Image:** Displays the original image (moon.png), showing what was hidden in the audio file.

- **Original Sound Data:** Plots the original sound waveform (soundData), providing a baseline for comparison.

Modified Data Tab:

- **Modified Sound Data:** Plots the modified sound waveform (modifiedSoundData), showing how the sound data has changed after embedding the image.
- **Extracted Image:** Displays the image extracted from the modified sound file, allowing for visual comparison with the original image to verify the success of the steganography process.

Summary of Expected Outputs:

- **Binary Data Console Outputs:** Visual confirmation of the binary transformations at various stages (compression, encryption, ECC).
- **Completion Messages:** Confirmation of successful embedding and extraction.
- **Figures with Tabs:**
  - **Original Data:** Visualizes the original image and sound data.
  - **Modified Data:** Shows the modified sound data and the extracted image.

4.3 Sound Spread Spectrum Technique

Spread spectrum steganography is a sophisticated method used to hide information within an audio signal by distributing the hidden data across a wide range of frequencies, making it difficult to detect and robust against various types of audio processing. Here's a detailed explanation of the spread spectrum steganography method:

4.3.1 Fundamentals of Spread Spectrum Technique

1. **Concept**:
    - Spread spectrum techniques spread the signal over a wide frequency band, much wider than the minimum bandwidth required to transmit the information.
    - This spreading is achieved using a code that is independent of the original signal.
2. **Advantages**:
    - **Robustness**: The hidden message is spread across many frequencies, making it resistant to noise, compression, and other distortions.
    - **Security**: Without knowledge of the spreading code, it is challenging for unauthorized parties to detect the presence of hidden data.
    - **Low Detectability**: The power of the hidden signal is spread so thinly across the spectrum that it is below the noise floor and thus difficult to detect.

4.3.2 Steps in Spread Spectrum Steganography

1. **Message Preparation**:
    - The message to be hidden is first converted into a binary format if it's not already in this form.

2. **Spreading Code Generation**:
    - A pseudorandom noise (PN) sequence or spreading code is generated. This code is typically known only to the sender and intended receiver.
    - The spreading code has properties similar to white noise and is used to modulate the hidden message.

3. **Modulation**:
    - The binary message is modulated using the spreading code. Common modulation techniques include Direct Sequence Spread Spectrum (DSSS) and Frequency Hopping Spread Spectrum (FHSS).
    - **Direct Sequence Spread Spectrum (DSSS)**:
        - Each bit of the message is multiplied by the spreading code sequence.
        - This process spreads the message signal across a wider frequency band.
    - **Frequency Hopping Spread Spectrum (FHSS)**:
        - The message is divided into small chunks and transmitted over different frequencies in a pseudo-random order.
        - The hopping pattern is determined by the spreading code.
4. **Embedding Process**:
    - The modulated message is embedded into the audio signal. This can be done in the time domain or frequency domain.
    - **Time Domain Embedding**:
        - The modulated signal is added directly to the audio samples.
        - Care is taken to ensure that the added signal is of low amplitude to avoid perceptible distortion.
    - **Frequency Domain Embedding**:
        - The audio signal is transformed using a method like the Fast Fourier Transform (FFT).
        - The modulated message is added to the frequency components of the audio signal.
        - The inverse FFT is then applied to transform the signal back to the time domain.
5. **Transmission or Storage**:
    - The modified audio signal, now containing the hidden message, is transmitted or stored as usual.

4.3.3 Extraction Process

1. **Reception**:
    - The stego-audio signal is received or retrieved.
2. **Spreading Code Synchronization**:
    - The receiver synchronizes with the spreading code used during the embedding process.
3. **Demodulation**:
    - The spread signal is demodulated using the same spreading code.
    - This process extracts the hidden message by correlating the received signal with the spreading code.
4. **Message Reconstruction**:
    - The demodulated signal is processed to reconstruct the original hidden binary message.
    - This may involve error correction if redundant encoding was used.

Example of Direct Sequence Spread Spectrum (DSSS)

1. **Message Preparation**:
    - Original Message: 1011
    - Binary format: 1011
2. **Spreading Code**:
    - Assume a simple spreading code: 1101

3. **Modulation**:
    - Each bit of the message is XORed with the spreading code:
        - Message bit 1: 1 XOR 1101 = 0010
        - Message bit 0: 0 XOR 1101 = 1101
        - Message bit 1: 1 XOR 1101 = 0010
        - Message bit 1: 1 XOR 1101 = 0010

4. **Embedding**:
    - The modulated bits (0010 1101 0010 0010) are added to the audio signal at a low amplitude.

4.3.4 Applications and Considerations with Example

- **Applications**:
    - **Secure Communications**: Transmitting confidential information covertly.
    - **Digital Watermarking**: Protecting intellectual property by embedding ownership information.
    - **Covert Channels**: Creating hidden communication channels within overt audio transmissions.
- **Considerations**:
    - **Synchronization**: Precise synchronization between the sender and receiver is crucial for effective extraction of the hidden message.
    - **Audio Quality**: Ensuring the embedded data does not significantly degrade the audio quality.
    - **Robustness**: The technique should withstand common audio processing operations like compression, filtering, and re-sampling.

Spread spectrum steganography is a powerful and versatile technique, providing a high degree of security and robustness for embedding hidden data within audio signals.

CHAPTER 5
Chaotic Code: Generation and Properties

5.1 Introduction

-Chaos theory is a branch of mathematics and physics that deals with systems that appear to be disordered or chaotic, but are actually governed by underlying patterns and deterministic laws. These systems are highly sensitive to initial conditions, a phenomenon often referred to as the "butterfly effect," where small changes in the initial state of a system can lead to vastly different outcomes.

5.1.1 Definition of chaos theory

-Chaos theory is the study of how small changes in the initial conditions of a system can lead to vastly different outcomes over time. This concept has some relevant applications in the field of cryptography.

5.1.2 Relevance to Cryptography

-In cryptography, chaos theory can be used to design secure encryption algorithms. Chaotic systems are highly sensitive to initial conditions, meaning that even a tiny change in the input can result in a completely different output. This property can be leveraged to create encryption algorithms that are resistant to brute-force attacks and other attempts to crack the cipher.

-One way chaos theory is applied in cryptography is through the use of chaotic maps or functions in the encryption/decryption process. These chaotic functions exhibit the sensitivity to initial conditions that makes the resulting ciphertext unpredictable, even if an attacker has knowledge of the algorithm. This can enhance the overall security of the cryptographic system.

-Additionally, the complex, nonlinear behavior of chaotic systems can be used to generate strong pseudorandom number sequences, which are essential for tasks like key generation, initialization vectors, and masking techniques in cryptography.

-By incorporating the principles of chaos theory, cryptographic algorithms can become more robust against attacks and provide a higher level of confidentiality and data protection. The unpredictable nature of chaotic systems can make it extremely difficult for attackers to reverse-engineer the encryption process and recover the original plaintext.

-In the context of cryptography, chaos theory principles can be applied to the design of secure ciphers and encryption algorithms. Chaotic systems exhibit properties that are desirable for cryptographic applications, such as:

1. **Unpredictability:** The output of a chaotic system is extremely sensitive to initial conditions, making it very difficult to predict the future state of the system based on its current state. This unpredictability is crucial for secure encryption.

2. **Randomness:** Chaotic systems can generate output that appears random, even though the underlying system is deterministic. This apparent randomness is important for generating secure cryptographic keys and initialization vectors.

3. **Diffusion and Confusion:** Chaotic processes can be used to achieve the cryptographic properties of diffusion (where a small change in the plaintext or key results in a large change in the ciphertext) and confusion (where the relationship between the plaintext, key, and ciphertext is obscured).

5.2 The fundamental chaos theory :-

5.2.1 Sensitivity to Initial Conditions:

- Chaotic systems are extremely sensitive to their initial conditions.

- A tiny change in the starting point of a system can lead to vastly different outcomes over time.

- This is often described as the "butterfly effect", where a small flap of a butterfly's wings can ultimately contribute to a major storm elsewhere.

5.2.2 Nonlinearity

- Chaotic systems exhibit nonlinear behavior, meaning the relationship between inputs and outputs is not proportional.

- Small changes in the system can lead to disproportionately large effects.

- Nonlinearity is a critical aspect of chaos, as linear systems cannot exhibit chaotic behavior.

Unpredictability:

- The long-term behavior of chaotic systems is inherently unpredictable, even if the system's governing equations are known.

- This is because the sensitivity to initial conditions makes it impossible to precisely predict the future state of the system.

5.2.3 Attractors and Fractals

- Chaotic systems often have "attractors" - specific patterns or states that the system tends to settle into over time.

- These attractors can be strange or fractal in nature, exhibiting complex, irregular shapes.

- The unpredictable movement of a system between these attractors is a hallmark of chaos.

Determinism:

- Chaotic systems are deterministic, meaning their behavior is governed by a set of well-defined rules or equations.

- However, the complex, nonlinear nature of these systems makes their behavior appear random or stochastic.

5.3 Mathematical Background

5.3.1 key equation

The key equation in chaos theory is the logistic map, which is a discrete-time dynamical system described by the following equation:

$$x_{n+1} = r * x_n * (1 - x_n)$$

Where:

- x_n is the state of the system at the nth iteration

- r is the bifurcation parameter, which is a real number between 0 and 4

The logistic map is a simple mathematical model that exhibits a wide range of complex and chaotic behavior. It was introduced by the Polish mathematician Michał Władysław Rychlik in the 1940s, but it was later popularized by the biologist Robert May in the 1970s.

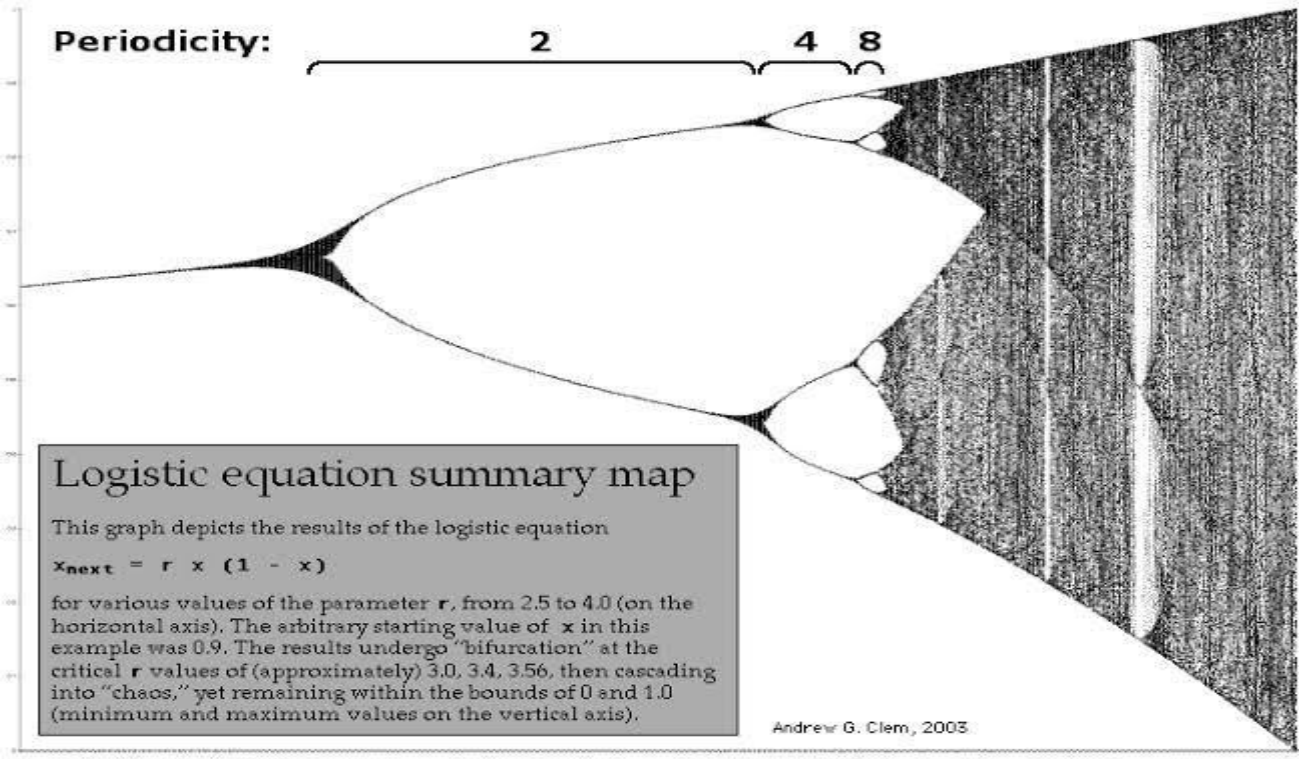
The behavior of the logistic map depends on the value of the bifurcation parameter, r:

- When $0 \leq r < 1$, the system converges to a stable fixed point at $x = 0$.

- When $1 \leq r < 3$, the system converges to a stable fixed point at $x = (r-1)/r$.

- When $3 \leq r < 3.56$, the system exhibits period-doubling bifurcations, leading to a period-2 cycle, then period-4 cycle, and so on, until the system becomes chaotic.

- When $3.56 \leq r \leq 4$, the system exhibits fully developed chaos, with a sensitive dependence on initial conditions.



5.3.2 phase space and state space:

Phase Space:

-Phase space is a mathematical representation of the possible states of a dynamical system. For a system with N variables, the phase space is an N-dimensional space, where each dimension corresponds to one of the system's variables. The state of the system at any given time is represented by a point in this N-dimensional phase space.

-In the case of a simple pendulum, the phase space would be a 2-dimensional space, where one dimension represents the angle of the pendulum, and the other dimension represents the angular velocity. As the pendulum moves, its state traces out a trajectory in this 2-dimensional phase space.

State Space:

-The state space is a more general concept that encompasses the phase space and includes any other relevant parameters or variables that may influence the behavior of the system. For example, in addition to the angle and angular velocity of a pendulum, the state space might also include factors such as the length of the pendulum, the gravitational acceleration, and the damping coefficient.

-The state space provides a complete description of the system, including all the information necessary to determine the future evolution of the system given its current state.

Chaos and Phase/State Spaces:

-In the context of chaos theory, the phase space or state space of a chaotic system is characterized by the presence of strange attractors. A strange attractor is a set of points in the phase/state space that the system's trajectory is drawn towards, even though the trajectory never exactly repeats itself. The fractal-like structure of strange attractors is a hallmark of chaotic systems.

-The sensitivity to initial conditions in chaotic systems manifests as nearby trajectories in the phase/state space diverging exponentially over time. This divergence makes it extremely difficult to predict the long-term behavior of chaotic systems, as small changes in the initial conditions can lead to vastly different outcomes.

-Understanding the phase space and state space of a chaotic system is crucial for analyzing its behavior, identifying the presence of chaos, and developing effective strategies for controlling or predicting the system's dynamics.

5.3.3 Lyapunov exponents:

-The Lyapunov exponent is a crucial tool for understanding the dynamics of chaos maps, such as the logistic map, the tent map, and the Hénon map, among others. It provides a quantitative measure of the chaotic nature of the system and helps in the classification and analysis of different types of chaotic behavior.

-Lyapunov exponents are an important concept in the study of chaotic dynamical systems. In the context of a chaos map, the Lyapunov exponents quantify the rate of divergence (or convergence) of nearby trajectories in the phase space of the map.

For a one-dimensional chaos map, the Lyapunov exponent, denoted as λ, is defined as:

$$\lambda = \lim_{n\to\infty} \frac{1}{n} * \log|f'(x_n)|$$

Where:

- f(x) is the map function

- x_n is the n-th iterate of the initial condition x_0

- f'(x_n) is the derivative of the map function at the point x_n

-The sign and magnitude of the Lyapunov exponent provide important information about the dynamics of the chaos map:

1. Positive Lyapunov exponent:

- Indicates that the map exhibits sensitive dependence on initial conditions, which is a hallmark of chaotic behavior.

- Nearby trajectories in the phase space diverge exponentially over time.

2. Negative Lyapunov exponent:

- Indicates that the map has a stable fixed point or periodic orbit.

- Nearby trajectories converge exponentially to the stable attractor.

3. Zero Lyapunov exponent:

- Indicates the presence of a neutral fixed point or a periodic orbit with neutral stability.

- Nearby trajectories neither diverge nor converge exponentially.

5.4 Chaotic Maps

Definition :

Chaotic maps are mathematical functions that exhibit chaotic behavior, meaning their outcomes are highly sensitive to initial conditions, appear random, and are deterministic. They are used in various fields, including cryptography, due to their unpredictability and complex behavior.

Examples :

5.4.1 Logistic Map

The logistic map is a simple mathematical model of population growth and is given by the equation:

$$x_{n+1} = r x_n (1 - x_n)$$

where:

- Xn : is a number between 0 and 1 that represents the population at time n
- r is a positive constant representing the growth rate

- The logistic map is a classic example of how complex, chaotic behavior can arise from very simple non-linear dynamical equations.

Behavior of the Logistic Map:

The behavior of the logistic map changes dramatically depending on the value of the parameter r

1. $0 < r \leq 1$: The population will eventually die off regardless of the initial population x_0. The fixed point $x = 0$ is stable.
2. $1 < r \leq 2$: The population will stabilize at a single non-zero value. The fixed point $x = \frac{r-1}{r}$ is stable.
3. $2 < r \leq 3$: The population will stabilize at a value, but it will oscillate around this value before stabilizing.
4. $3 < r \leq 3.45$: The population enters a period-doubling route to chaos. It will oscillate between two values, then four values, and so on as r increases.
5. $3.45 < r \leq 3.57$: The logistic map shows chaotic behavior. Small changes in initial conditions result in vastly different outcomes.
6. $3.57 < r \leq 4$: The system is fully chaotic, with occasional windows of periodicity. For certain ranges within this interval, the system can display periodic behavior again before returning to chaos.

Example of Chaotic Behavior:

For r=3.9 , the logistic map exhibits chaotic behavior. Even a tiny difference in the initial value Xo will result in significantly different sequences of Xn .

For instance, if x0 = 0.5:

$x1 = 3.9 \times 0.5 \times (1 - 0.5) = 0.975$

$x2 = 3.9 \times 0.975 \times (1 - 0.975) \approx 0.095$

$x3 = 3.9 \times 0.095 \times (1 - 0.095) \approx 0.335$

5.4.2 Henon Map:

The Henon map is a discrete-time dynamical system that serves as a well-known example of a system exhibiting chaotic behavior. It is defined by the following equations:

$$x_{n+1} = 1 - ax_n^2 + y_n$$
$$y_{n+1} = bx_n$$

where:

- Xn and Yn are the state variables at iteration n,
- a and b are parameters that control the behavior of the map.

Behavior of the Henon Map

The behavior of the Henon map is highly sensitive to the values of parameters aaa and b. For certain values of aaa and b, the map exhibits chaotic behavior, while for others, it can exhibit periodic or quasi-periodic behavior.

Example of Chaotic Behavior

For the classic Henon map, the typical parameters are:

- a=1.4
- b=0.3

Starting with an initial condition (e.g., x0=0, y0=0), the iteration of the Henon map generates a sequence of points that can be plotted in the x-y plane. This sequence forms a fractal structure known as the Henon attractor.

Properties of the Henon Map

Sensitivity to Initial Conditions: Small differences in initial conditions lead to vastly different trajectories, a hallmark of chaotic systems.

Deterministic Nature: Despite its complex behavior, the system is deterministic, meaning that its future behavior is fully determined by its initial conditions and parameters.

Strange Attractors: The Henon map generates strange attractors, which are fractal structures that appear complex and detailed at any level of magnification.

Topological Mixing: The points in the space are mixed in such a way that any given region of the space will eventually overlap with any other region.

Security Analysis of the Henon Map

Strengths:

High Complexity: The fractal nature and complexity of the Henon attractor make it difficult to predict or reverse-engineer without precise knowledge of the initial conditions and parameters.

Sensitive Dependence on Initial Conditions: This property ensures that even tiny differences in initial conditions or parameters result in drastically different outcomes, providing robustness against brute-force attacks.

Deterministic Chaos: The deterministic nature ensures reproducibility for encryption and decryption processes while maintaining unpredictability.

Weaknesses:

Implementation Complexity: Ensuring numerical precision and stability can be challenging, especially in digital implementations.

Parameter Sensitivity: Improper selection of parameters a and b can lead to non-chaotic behavior, reducing the effectiveness of the map in cryptographic applications.

Finite Precision Arithmetic: Digital systems with finite precision can introduce periodicity and reduce the map's chaotic properties.

Comparison with Traditional Methods:

- **Traditional Encryption**: Methods like AES and RSA rely on well-established mathematical problems (e.g., factoring large numbers, discrete logarithms) and are widely studied and standardized.
- **Chaotic Encryption (Henon Map)**: Uses chaotic maps to achieve encryption, leveraging properties like high sensitivity to initial conditions and complex behavior.

Advantages:

- Potentially faster computations for certain applications.
- Adds an additional layer of complexity that can deter attackers due to the chaotic nature.

Disadvantages:

- Less mature and standardized.
- Potential vulnerabilities if not carefully implemented, particularly regarding numerical precision and parameter selection.

5.4.3 Tent Map

The Tent Map is another simple yet powerful chaotic map used in the study of dynamical systems and chaos theory. It is defined by the following piecewise linear function :

$$\begin{cases} \mu x_n & \text{if } x_n < 0.5 \\ \mu(1 - x_n) & \text{if } x_n \geq 0.5 \end{cases}$$

where:

- x_n is the state variable at iteration n,
- μ is a parameter that controls the behavior of the map.

Behavior of the Tent Map

The behavior of the Tent Map is highly dependent on the value of the parameter μ :

1. $0 < \mu < 1$: The system converges to the fixed point $x = 0$.
2. $1 \leq \mu \leq 2$: The map exhibits increasingly complex behavior, transitioning from periodic to chaotic as μ approaches 2.

For $\mu = 2$, the Tent Map exhibits fully developed chaotic behavior. It can be used to generate sequences that appear random, despite being generated by a deterministic process.

Properties of the Tent Map:

Sensitivity to Initial Conditions: Small differences in initial conditions lead to vastly different trajectories, a key feature of chaotic systems.

Deterministic Nature: The map's future behavior is fully determined by its initial conditions and the parameter μ.

Piecewise Linearity: The Tent Map is defined by linear segments, making it simpler to analyze and implement compared to some other chaotic maps.

Topological Mixing: Points are mixed thoroughly in the space, leading to a uniform distribution over time.

Security Analysis of the Tent Map

Strengths

High Sensitivity to Initial Conditions: This provides robustness against brute-force and differential attacks.

Simple Implementation: The piecewise linear nature makes it easier to implement and analyze, reducing the risk of errors in practical applications.

Uniform Distribution: For μ=2, the map has a uniform invariant measure, making it useful for generating pseudo-random sequences.

Weaknesses

Parameter Sensitivity: If μ is not chosen correctly, the system may not exhibit chaotic behavior.

Finite Precision Issues: In digital implementations, finite precision arithmetic can introduce periodicity and degrade chaotic properties.

Known Structures: Being simpler than some other chaotic maps, the Tent Map's structure might be more predictable under certain conditions.

Comparison with Traditional Methods

- **Traditional Encryption**: Established methods like AES and RSA rely on complex mathematical problems for security.
- **Chaotic Encryption (Tent Map)**: Utilizes the chaotic properties of the Tent Map to achieve encryption, offering benefits of simplicity and high sensitivity.

Advantages:

- Simple and fast to compute.
- High sensitivity to initial conditions adds an extra layer of security.

Disadvantages:

- Less thoroughly studied and standardized compared to traditional methods.

- Potential vulnerabilities if not carefully implemented, particularly with respect to parameter choice and numerical precision.

And There is -> **Arnold's Cat Map**: Defined on a torus, this map stretches and folds the space in a manner that produces chaotic behavior.

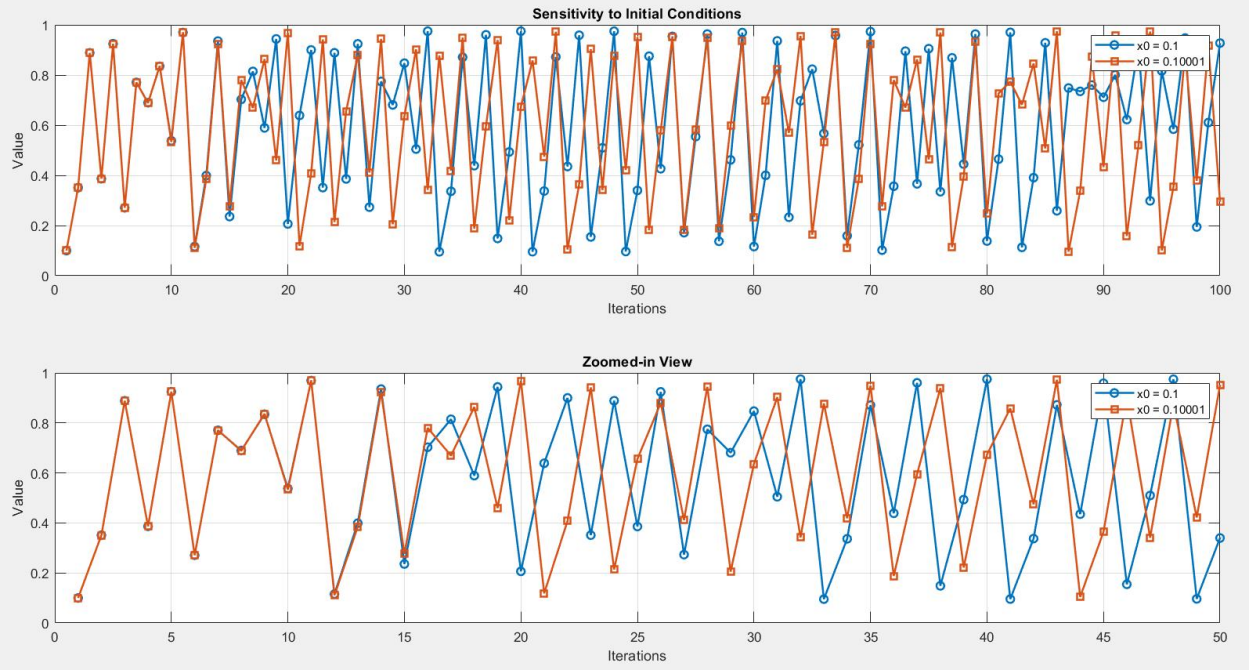
CHAPTER 6
Application Chaotic Code in Image Encryption

6.1 Chaos in Cryptography

Chaotic systems have become an intriguing subject of study in cryptography due to their inherent properties which can be leveraged to enhance security. Let's delve into the specific benefits of using chaotic systems in cryptography, focusing on their high sensitivity to initial conditions, pseudo-randomness, and other relevant characteristics.

High Sensitivity to Initial Conditions

Definition: Chaotic systems are extremely sensitive to initial conditions, meaning that even a tiny difference in the starting state can lead to vastly different outcomes. This phenomenon is often referred to as the "butterfly effect".

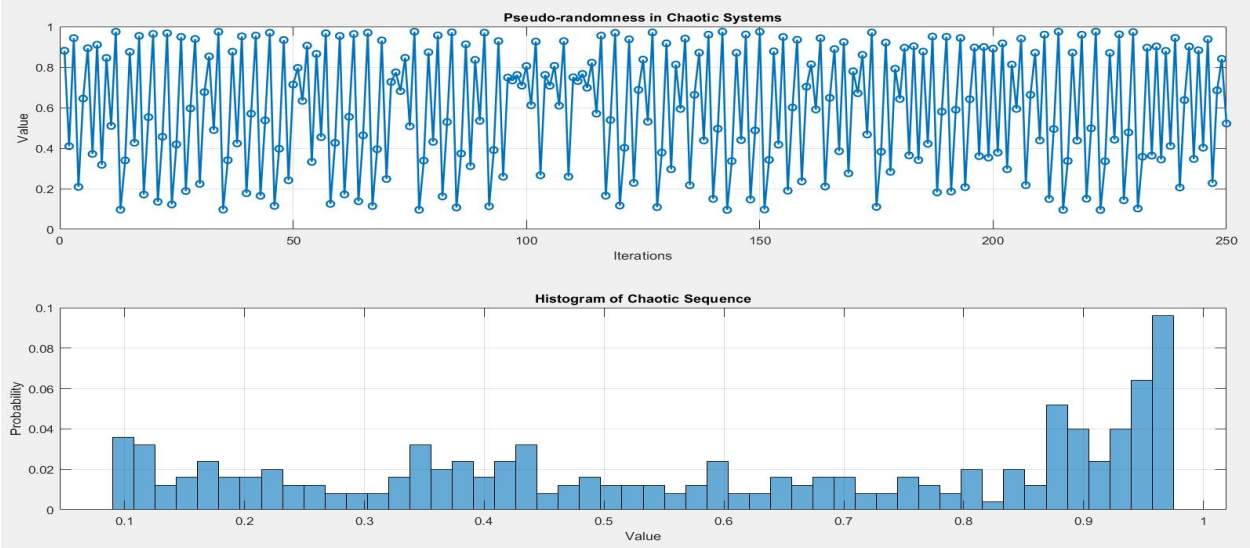


Benefits in Cryptography:

- **Key Sensitivity**: This property ensures that a minute change in the encryption key or the initial state will result in a completely different encrypted message. This makes it nearly impossible for an attacker to predict or reproduce the original message without the exact key .
- **Enhanced Security**: The high sensitivity makes brute-force attacks much more difficult. Since small changes in keys result in dramatically different ciphertexts, attackers cannot easily use patterns or statistical methods to crack the encryption.
- **Robustness**: It provides robustness against differential cryptanalysis, a method where attackers analyse the differences between pairs of plaintexts and their corresponding ciphertexts to find the secret key .

Pseudo-Randomness

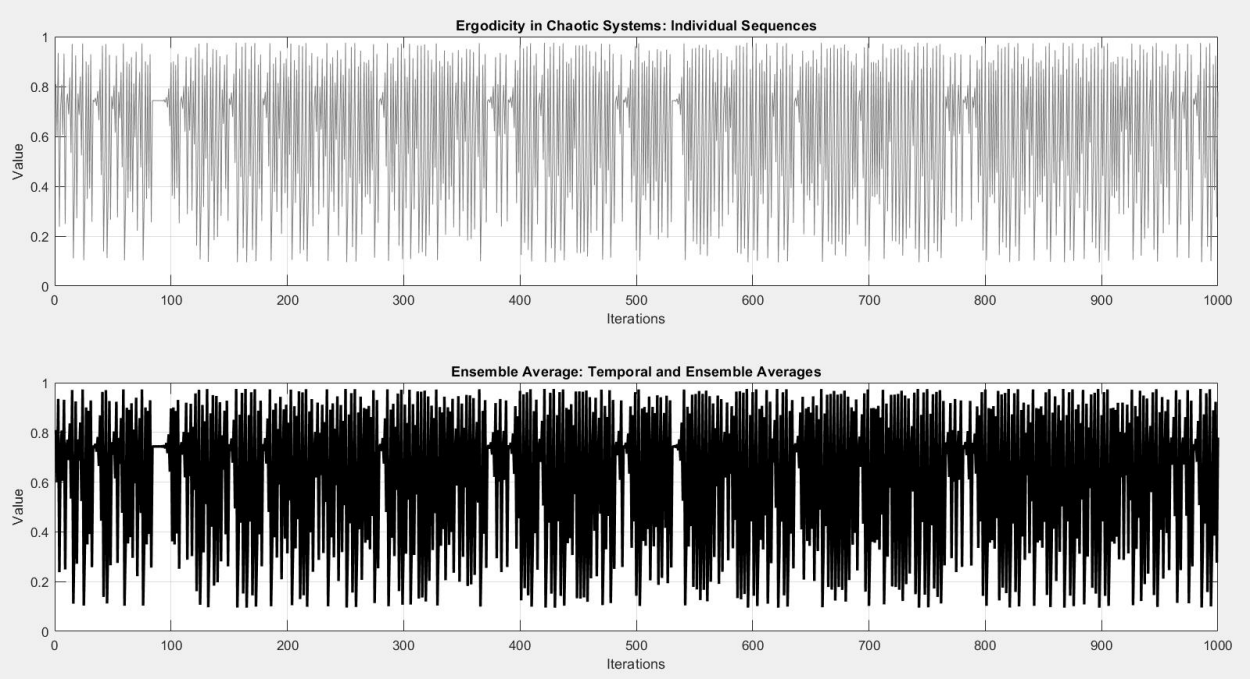
Definition: Pseudo-randomness in chaotic systems refers to the generation of sequences that appear random, although they are generated by deterministic rules.



Benefits in Cryptography:

- **Random Key Generation:** The unpredictability of chaotic sequences can be used to generate random cryptographic keys, making it harder for attackers to guess or reproduce them.
- **Secure Stream Ciphers:** Chaotic maps can be used to create stream ciphers, where the pseudo-random sequence generated by the chaotic system is combined with the plaintext to produce ciphertext. This method ensures high security due to the unpredictability of the chaotic sequence.
- **Initialization Vectors (IVs):** Chaotic sequences can be used to generate initialization vectors that ensure different encryption results even when the same plaintext and key are used multiple times
- **Ergodicity**

  Definition: Ergodicity in chaotic systems means that the system will eventually explore all possible states in its phase space given enough time.

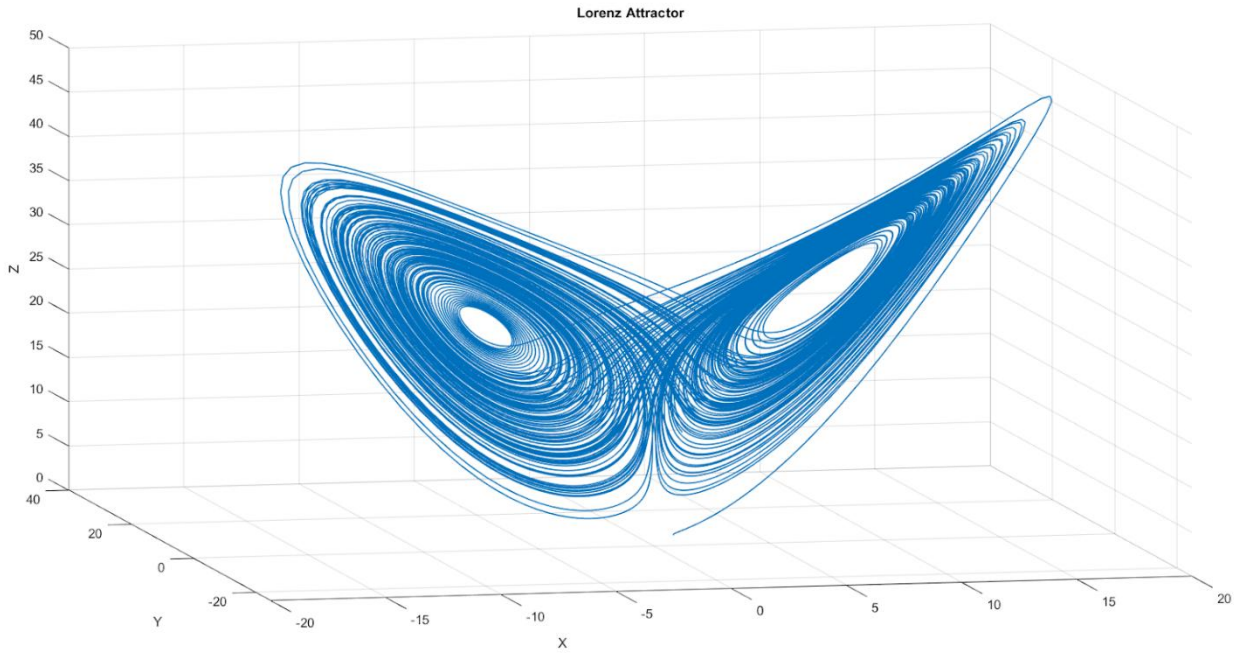


Benefits in Cryptography:

- **State Space Coverage**: This property ensures that all possible states are eventually visited, providing a thorough exploration of the key space. This makes it harder for attackers to find patterns or weaknesses in the encryption algorithm.
- **Uniform Distribution**: Ergodic behaviour contributes to the uniform distribution of cryptographic outputs, reducing the chances of biases that could be exploited in attacks.

Deterministic yet Unpredictable

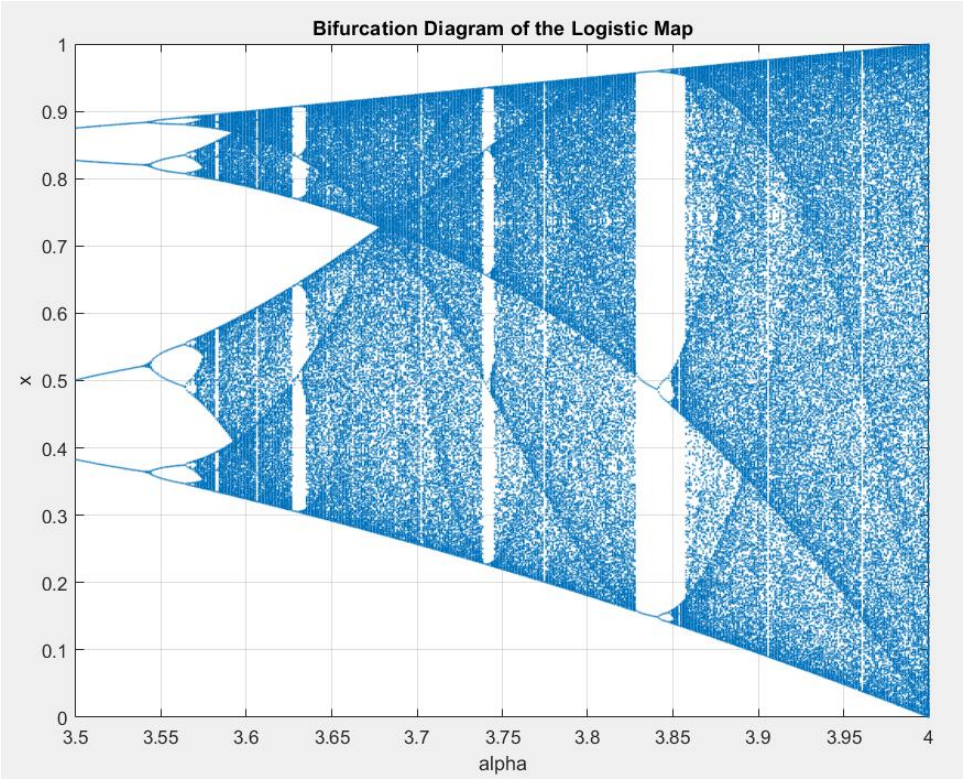
Definition: Chaotic systems are deterministic, meaning their future behavior is fully determined by their initial conditions and rules, yet they appear unpredictable due to their complex dynamics.



Benefits in Cryptography:

- **Repeatability:** The deterministic nature allows for the encryption and decryption process to be repeatable, as long as the initial conditions (keys) are known. This is crucial for reliable encryption.
- **Unpredictability:** Despite being deterministic, the output appears random and unpredictable, providing strong security against attempts to predict or reproduce the key or the encrypted message.
- **Complexity and Nonlinearity**

  Definition: Chaotic systems exhibit complex and nonlinear behaviour, meaning their evolution cannot be described by simple linear equations.



Benefits in Cryptography:

- **Enhanced Complexity:** Nonlinearity introduces a high level of complexity that increases the difficulty for attackers to model or predict the system's behaviour.
- **Resistance to Linear Attacks:** Many cryptographic attacks, such as linear cryptanalysis, rely on the linear approximation of the encryption process. The inherent nonlinearity of chaotic systems makes such attacks ineffective .

6.2 Chaotic Encryption Algorithms

Chaotic encryption algorithms leverage the properties of chaotic systems to achieve secure encryption, particularly suitable for images. These algorithms rely on the fundamental characteristics of chaos, such as sensitivity to initial conditions, pseudo-randomness, and ergodicity, to create robust cryptographic methods for image data.

Basic Principles

Definition: Chaotic encryption algorithms are based on chaos theory, which deals with deterministic systems that exhibit random-like behavior due to their sensitivity to initial conditions and nonlinear dynamics.

Key Characteristics:

- **Deterministic yet Unpredictable**: Chaotic systems are deterministic, meaning their future behaviour is fully determined by their initial conditions and rules, yet they appear unpredictable and random due to their complex dynamics. This characteristic is crucial for creating encryption algorithms that are both reliable and secure, as discussed in the *"Chaos in Cryptography"* section on the benefits of using chaotic systems in cryptography .

- **Sensitivity to Initial Conditions**: Small differences in initial conditions lead to significantly different outcomes, ensuring that slight changes in encryption keys or states produce vastly different ciphertexts. This high sensitivity enhances key sensitivity and security, making it difficult for attackers to predict or reproduce the original image without the exact key .
- **Ergodicity**: Chaotic systems explore all possible states over time, ensuring thorough coverage of the key space and uniform distribution of cryptographic outputs. This property contributes to the robustness of the encryption algorithm, making finding patterns or weaknesses significantly harder for attackers.

Shuffling and Substitution

Chaotic encryption algorithms for images commonly use two main techniques: shuffling and substitution.

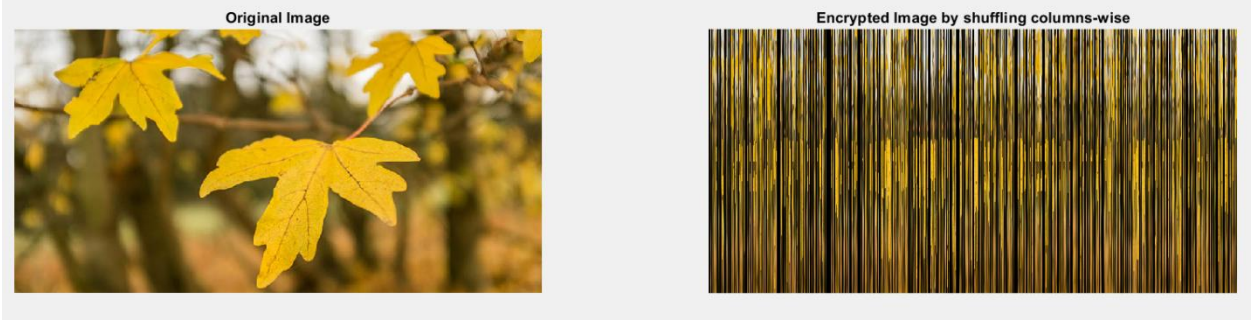
6.2.1 Shuffling:

- **Definition:** Shuffling involves rearranging the positions of elements within the data based on a chaotic sequence.

- **Process:**
  - **Chaotic Maps**: Chaotic maps like the Logistic map or Tent map are used to generate a pseudo-random sequence of indices.
  - **Permutation**: The generated sequence is then used to permute the positions of the elements in the plaintext, or the pixels in the image, effectively scrambling the data.
  - **Example**: In image encryption, shuffling might involve rearranging the pixel positions in an image based on the chaotic sequence, making the image unrecognizable without the correct chaotic key.

- **Shuffling Techniques:**
    - **Row-wise**: It involves rearranging the rows of pixel values in the image, while keeping the columns intact. This method alters the spatial arrangement of pixel rows to obscure the original image content, enhancing security through spatial permutation.

    - **Column-wise**: It entails rearranging the columns of pixel values while maintaining the integrity of the rows. This approach modifies the spatial layout of pixel columns, thereby obscuring the original image content and bolstering security through spatial permutation .



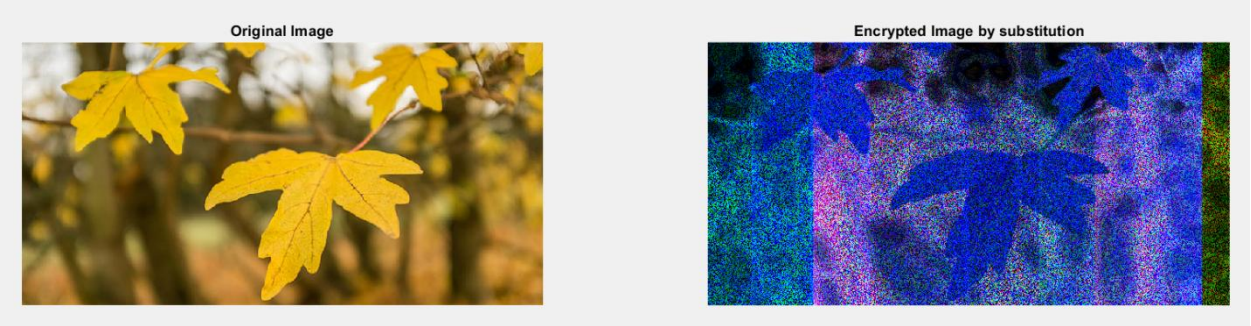
    - **Pixel-wise**: Involves rearranging individual pixels randomly throughout the image. This method offers enhanced distortion compared to row-wise and column-wise shuffling because it disrupts the spatial coherence of the image at a finer granularity. By scrambling pixels across the entire image,

pixel-wise shuffling significantly alters the visual structure of the image, thereby increasing its resistance to decryption and enhancing security.

6.2.2 Substitution:

- **Definition**: Substitution involves replacing the elements of the data with other elements according to a predefined scheme, typically based on chaotic sequences.
- **Process**:
    - **Chaotic Maps**: A chaotic map generates a pseudo-random sequence used as a basis for substitution.
    - **Replacement**: Each element in the plaintext is substituted with another element as dictated by the chaotic sequence. This could involve simple substitutions like replacing each byte with another byte or more complex transformations.
    - **Example**: In image encryption, each pixel value might be replaced with a new value derived from a chaotic sequence, further obscuring the original image.

Substitution example: As shown in the image below, when encrypting an image using substitution, the colours (or pixel values) are altered, but the overall shape and structure of the image remain recognizable to the naked eye.



Key Generation for Image Encryption

Key generation in chaotic encryption algorithms leverages the pseudo-randomness and sensitivity of chaotic maps.

Process:

- **Chaotic Maps**: Key generation begins with the selection of an appropriate chaotic map, such as the Logistic map, Henon map, or Lorenz system.
- **Initial Conditions and Parameters**: The initial conditions and parameters of the chaotic map serve as the key. Due to the high sensitivity to these initial conditions, even minute changes result in entirely different chaotic sequences .
- **Generation of Pseudo-Random Sequences**: The chaotic map iterates from the initial conditions to produce a long sequence of pseudo-random numbers .
- **Key Derivation**: These pseudo-random numbers can be directly used as cryptographic keys or further processed (e.g., hashed) to produce keys of desired length and format .

6.3 Chaotic Encryption Image

This MATLAB code demonstrates how to encrypt and decrypt an image using chaotic maps. The process involves three main steps: chaotic key generation, the encryption algorithm, and the decryption algorithm. Let's break down each part of the code to understand how it works.

Chaotic Key Generation

Purpose: Generate a pseudo-random key based on a chaotic map, which will

```
% Function to generate chaotic sequence
function chaotic_sequence = generate_chaotic_sequence(alpha, x_initial, image_size)
    chaotic_sequence = zeros(1, image_size);
    x = x_initial;
    for i = 1:image_size
        x = alpha * x * (1 - x);
        chaotic_sequence(i) = x;
    end
    chaotic_sequence = mod(floor(chaotic_sequence * image_size), image_size) + 1;
end

% Function to generate key using chaotic map
function key = generate_key(alpha, x_initial)
    x = x_initial;
    for i = 1:1000 % Iterate to get a more "random" value
        x = alpha * x * (1 - x);
    end
    key = x; % Use the chaotic value as the key
end
```

be used for encrypting and decrypting the image.

Explanation:

- **Chaotic Map**: The code uses the Logistic map, defined by the equation $x_{n+1}=\alpha \cdot x_n \cdot (1-x_n)$. The parameter `alpha` controls the behavior of the map, and `x_initial` is the initial condition.

- **Generate Chaotic Sequence**: The `generate_chaotic_sequence` function iterates the Logistic map to produce a sequence of values. This sequence is later on used to shuffle and substitute pixel values in the image.
- **Generate Key**: The `generate_key` function iterates the Logistic map 1000 times to ensure that the generated value `x` is sufficiently "random" and can be used as a key.

Encryption Algorithm

Purpose: Encrypt the image by shuffling and substituting its pixel values based on the chaotic sequence generated using the chaotic key.

```
% Function to encrypt image using chaotic map
function encrypted_image = chaos_encrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols;
    encrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);
    for channel = 1:channels
        channel_data = image(:,:,channel);
        flattened_channel = channel_data(:);
        for i = 1:image_size
            temp = flattened_channel(i);
            flattened_channel(i) = flattened_channel(chaotic_sequence(i));
            flattened_channel(chaotic_sequence(i)) = temp;
        end
        flattened_channel = bitxor(flattened_channel, uint8(key * 255));
        encrypted_image(:,:,channel) = reshape(flattened_channel, [rows, cols]);
    end
end
```

Explanation:

- **Initialization**: The `chaos_encrypt` function initializes the encrypted image and generates a chaotic sequence using the `generate_chaotic_sequence` function.

- **Shuffling**: The pixel values in each channel of the image are shuffled based on the chaotic sequence. This scrambling makes the image unrecognizable without the key.
- **Substitution**: The shuffled pixel values are then substituted using a bitwise XOR operation with the key (scaled to the range 0:255). This further obfuscates the image data.

Decryption Algorithm

Purpose: Decrypt the encrypted image by reversing the shuffling and substitution processes using the same chaotic key.

```
% Function to decrypt image using chaotic map
function decrypted_image = chaos_decrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols;
    decrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);
    for channel = 1:channels
        channel_data = image(:,:,channel);
        flattened_channel = channel_data(:);
        flattened_channel = bitxor(flattened_channel, uint8(key * 255));
        for i = image_size:-1:1
            temp = flattened_channel(i);
            flattened_channel(i) = flattened_channel(chaotic_sequence(i));
            flattened_channel(chaotic_sequence(i)) = temp;
        end
        decrypted_image(:,:,channel) = reshape(flattened_channel, [rows, cols]);
    end
end
```

Explanation:

- **Initialization**: The `chaos_decrypt` function initializes the decrypted image and generates the same chaotic sequence using the key.
- **Substitution**: The encrypted pixel values are first XORed with the key to reverse the substitution process.

- **Shuffling**: The pixels are then shuffled in reverse order using the chaotic sequence, restoring the original positions of the pixels to reconstruct the original image.

Main Script

```
% Image Encryption and Decryption using Chaotic Maps

% Load an image
original_image = imread('image1.jpg'); % Provide the path to your image

% Parameters
alpha = 3.93; % Chaotic map parameter
x_initial = rand; % Initial condition for chaotic map

% Get image dimensions
[rows, cols, channels] = size(original_image);

% Generate key using chaotic map
key = generate_key(alpha, x_initial);

% Encryption
encrypted_image = chaos_encrypt(original_image, rows, cols, channels, alpha, key);

% Decryption
decrypted_image = chaos_decrypt(encrypted_image, rows, cols, channels, alpha, key);
```

Explanation:

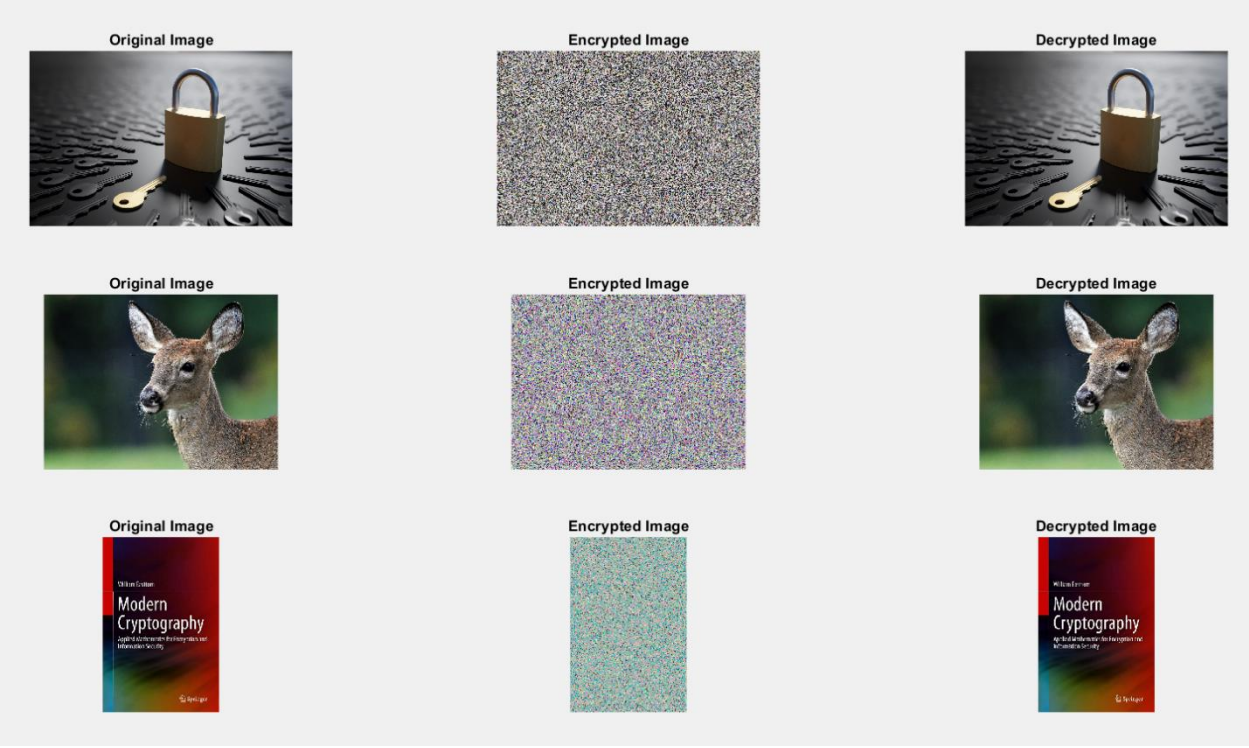
1. **Load Image**: The original image is loaded using `imread`.
2. **Parameters**: Set the chaotic map parameter alpha and the initial condition `x_initial`.
3. **Image Dimensions**: Get the dimensions of the image for processing.
4. **Key Generation**: Generate the chaotic key using the `generate_key` function.
5. **Encryption**: Encrypt the image using the `chaos_encrypt` function.

6. **Decryption**: Decrypt the image using the `chaos_decrypt` function with the correct and wrong keys.

Results

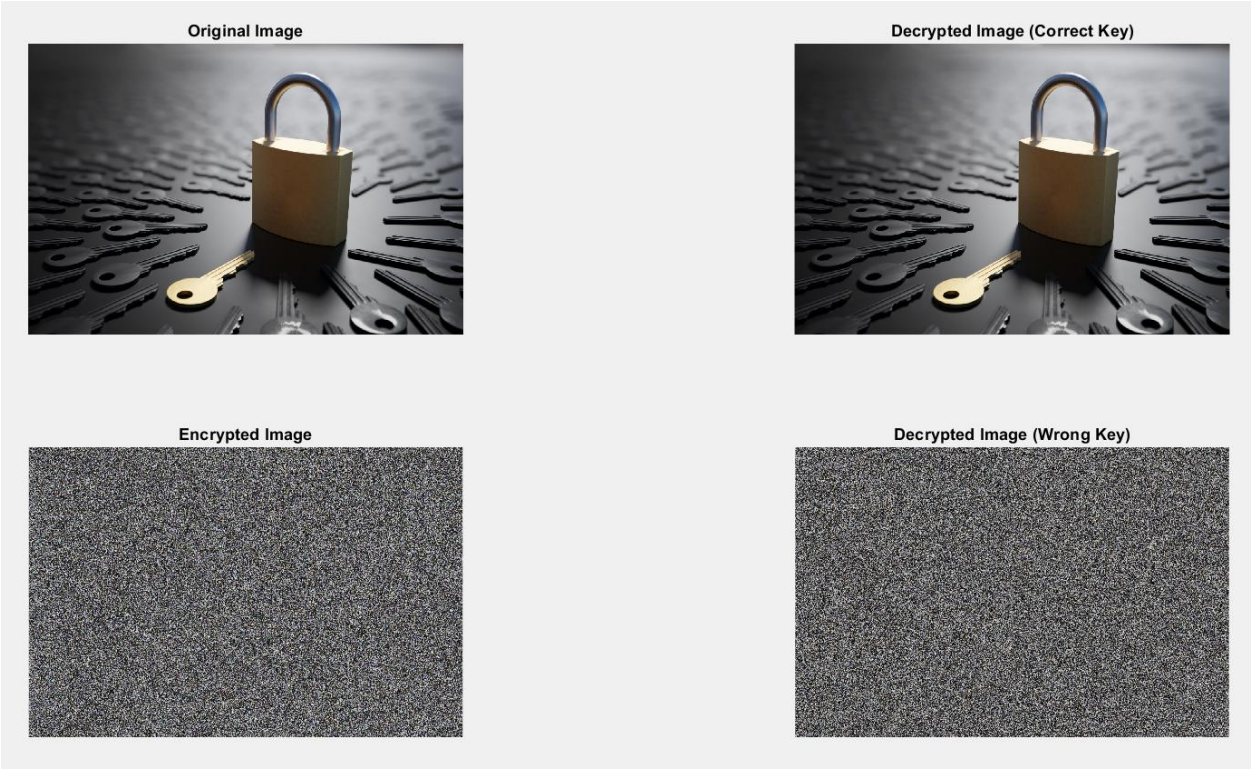
Using Correct Key:

The figure below shows the results of encrypting and decrypting three different images using the correct key and an optimal chaotic map parameter 'alpha' value of 3.93. The first column presents the original images, the middle column shows the images after encryption, and the last column displays the encrypted images after decryption.



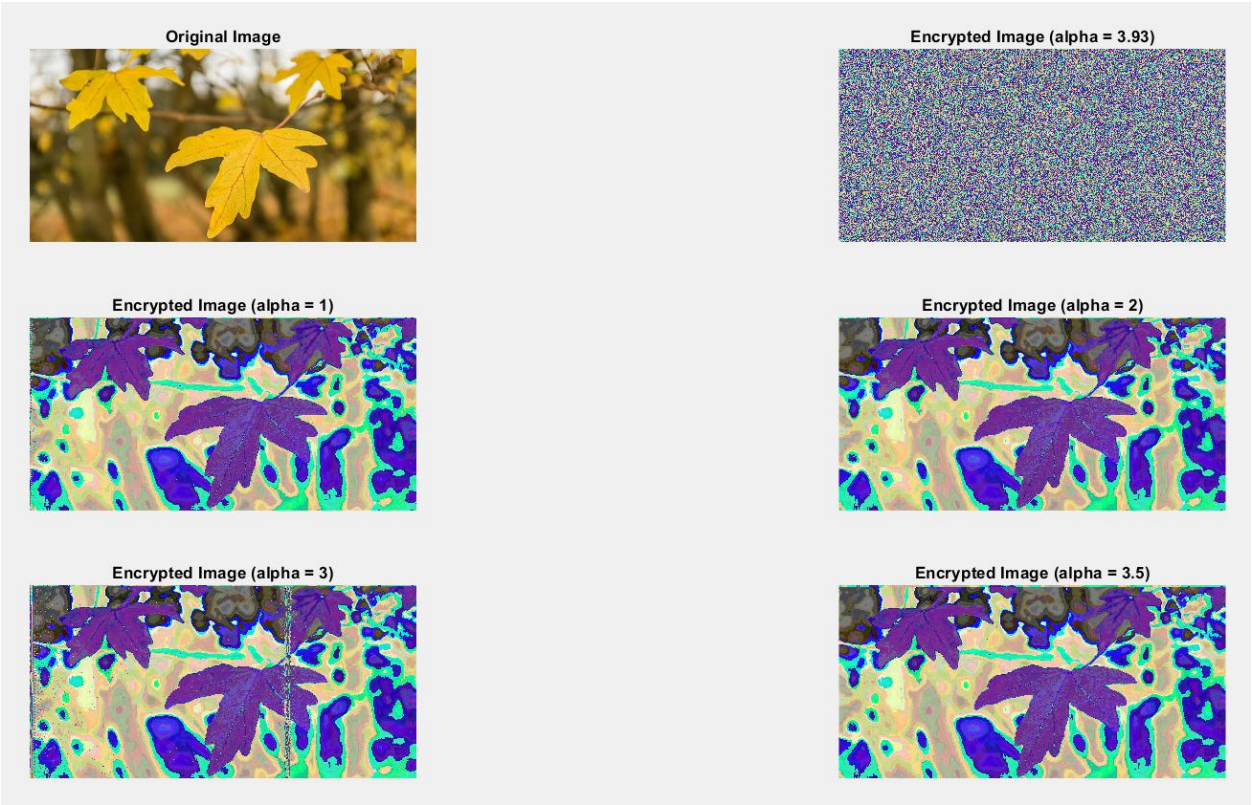
Using Incorrect Key:

The figure below depicts the outcome of decryption when an incorrect key is used. In the top right corner, the image decrypted with the correct key matches the original image. However, in the bottom corner, the image decrypted with the wrong key remains distorted, resembling the encrypted image itself.



Using non-optimal Chaotic Map Parameter (`alpha`):

The figure below displays the encryption outcomes of the same image using non-optimal chaotic map parameters (alpha) set at values: 1, 2, 3, and 3.5. The results indicate that as the parameter deviates further from its optimal value, the chaotic sequence generated becomes less random. Consequently, the encryption quality diminishes, resulting in poorer encryption.



6.4 Security Analysis of Chaotic Maps in Cryptographic Systems

Strengths

High Sensitivity to Initial Conditions: Chaotic maps exhibit extreme sensitivity to initial conditions, meaning small differences in initial input can lead to vastly different outputs. This property ensures that even slight variations in the key or plaintext result in completely different ciphertexts, providing robust security against differential cryptanalysis.

Deterministic yet Unpredictable: Chaotic maps are deterministic, which means they follow precise mathematical rules. However, their output appears random and unpredictable, making it difficult for attackers to reverse-engineer the encryption without the exact initial conditions and parameters.

High Complexity and Entropy: The complex behavior and high entropy generated by chaotic systems make them suitable for cryptographic applications, providing strong resistance against statistical attacks.

Simple Implementation: Many chaotic maps, such as the Tent Map and Logistic Map, have simple mathematical forms that are easy to implement in both hardware and software, ensuring fast encryption and decryption processes.

Efficiency: Chaotic systems can be computationally efficient, especially for applications requiring lightweight cryptography, such as in Internet of Things (IoT) devices, where resources are limited.

Weaknesses

Parameter Sensitivity: The security of chaotic encryption systems heavily depends on the choice of parameters. Incorrect parameter selection can lead to non-chaotic behavior, reducing the effectiveness and security of the system.

Finite Precision Arithmetic: In digital implementations, the finite precision of arithmetic operations can introduce periodicity and predictability, undermining the chaotic properties and potentially making the system vulnerable to attacks.

Implementation Complexity: While the mathematical forms of chaotic maps are simple, their practical implementation can be complex, especially ensuring numerical stability and precision in digital environments.

Lack of Standardization: Chaotic cryptographic systems are less standardized compared to traditional encryption methods like AES and RSA. This lack of standardization can lead to inconsistent security levels and difficulties in evaluating and comparing different implementations.

Known Attacks: Certain chaotic maps are susceptible to specific cryptanalytic attacks if not properly implemented. For instance, the choice of initial conditions and parameters needs to be carefully managed to avoid vulnerabilities.

Comparison with Traditional Methods

- **Traditional Encryption Methods**: Methods like AES, DES, and RSA are based on well-established mathematical principles such as prime factorization and discrete logarithms. These methods have been extensively studied, standardized, and proven to be secure against various types of attacks.
    - **Strengths**:
        - Proven security models and extensive analysis.
        - Standardized algorithms and implementations.
        - Widely adopted and trusted in the industry.
    - **Weaknesses**:
        - Can be computationally intensive, especially for resource-constrained devices.
        - Larger key sizes required to maintain security with increasing computational power of attackers.
- **Chaotic Encryption Methods**: Utilizes the inherent unpredictability and complexity of chaotic maps to encrypt data.
    - **Strengths**:

- Potential for high efficiency and speed, particularly in lightweight cryptographic applications.
- High sensitivity to initial conditions and parameters ensures strong diffusion and confusion properties.
- Simple mathematical structures make them suitable for hardware implementations.

- **Weaknesses**:
  - Less thoroughly studied and standardized compared to traditional methods.
  - Susceptible to implementation-specific vulnerabilities, particularly related to numerical precision and parameter selection.
  - Security highly dependent on maintaining the chaotic nature of the map, which can be challenging in digital systems.

6.5 Applications of Chaotic Encryption

Chaotic encryption leverages the inherent unpredictability and complex behavior of chaotic maps for securing data. Here are some practical uses and case studies demonstrating its application in various fields.

Practical Uses Secure Communication:

- **Message Encryption**: Chaotic encryption can be used to secure text messages transmitted over insecure channels. The sensitivity to initial conditions ensures that the same plaintext encrypted with different keys produces entirely different ciphertexts.
- **Signal Masking**: Chaotic signals can mask communication signals, making them difficult to intercept and decode without the correct key and initial conditions.

Image and Video Encryption:

- **Image Scrambling**: Pixels of an image can be rearranged based on chaotic sequences, making the image unrecognizable without the correct decryption key. This is particularly useful for secure image transmission and storage.
- **Video Encryption**: Similar techniques can be applied to video frames, ensuring that each frame is securely encrypted, preventing unauthorized access to video content.

IoT Security:

- **Lightweight Encryption**: Chaotic encryption algorithms can be optimized for resource-constrained IoT devices, providing secure communication and data storage with minimal computational overhead.
- **Device Authentication**: Using chaotic maps for generating unique authentication keys for IoT devices ensures secure identification and access control.

Cryptographic Key Generation:

- **High-Entropy Keys**: The chaotic nature of these maps can be used to generate cryptographic keys with high entropy, making them resistant to brute-force attacks.
- **Key Exchange**: Secure key exchange protocols can leverage the unpredictability of chaotic maps to ensure that the exchanged keys remain confidential.

Random Number Generation:

- **Pseudo-Random Numbers**: Chaotic maps can be used to generate high-quality pseudo-random numbers for use in cryptographic applications, simulations, and gaming.

Case Studies

Image Encryption Using Logistic Map:

- **Objective**: To secure image data using a logistic map-based encryption algorithm.
- **Method**: Each pixel value of the image is scrambled based on a sequence generated by the logistic map. The initial conditions and parameter rrr serve as the encryption key.
- **Result**: The encrypted image appears as noise, making it impossible to recognize the original image without the correct key. Statistical analysis shows high entropy and low correlation between the original and encrypted images, demonstrating strong security against attacks.

Secure Communication Systems Using Henon Map:

- **Objective**: To implement a secure communication protocol using the Henon map for encryption and decryption.
- **Method**: The plaintext message is converted into a numerical sequence and then encrypted using the Henon map equations. The receiver, with the correct initial conditions and parameters, decrypts the message using the inverse map.
- **Result**: The encrypted message is resistant to eavesdropping and replay attacks. The sensitivity to initial conditions ensures that even minor deviations in the key result in failed decryption, providing robust security.

Lightweight Encryption for IoT Devices Using Tent Map:

- **Objective**: To develop a lightweight encryption algorithm suitable for IoT devices using the Tent map.
- **Method**: The Tent map generates a chaotic sequence used to encrypt data transmitted between IoT devices. The simplicity of the Tent map ensures minimal computational requirements, making it suitable for devices with limited processing power.

- **Result**: The encryption algorithm provides a good balance between security and efficiency. It successfully encrypts data with low latency and minimal resource consumption, making it practical for real-world IoT applications.

Video Encryption Using Chaotic Maps:

- **Objective**: To secure video content using a chaotic map-based encryption scheme.
- **Method**: Each frame of the video is treated as an image and encrypted using a chaotic map, such as the Logistic map or Henon map. The sequence of frames is further scrambled to enhance security.
- **Result**: The encrypted video is highly resistant to unauthorized access. Frame-by-frame analysis shows that the encryption effectively prevents recognition of the original content. The method provides real-time encryption and decryption capabilities, suitable for streaming applications.

Conclusions

It’s impressive that we’ve been working on enhancing security in 5G and 6G. Let’s break down the key points:

- **Advanced Encryption Techniques:**

  We mentioned using advanced encryption methods. In 5G and 6G, robust encryption is crucial to protect data during transmission. Techniques like Spread Spectrum can enhance security against quantum attacks.

- **Steganography:**

  Steganography involves hiding information within other media (like images or sound). By embedding text within images or sound files, you can achieve covert communication channels.

  For example, we can use text-in-image or text-in-sound techniques to hide sensitive data. These methods make it challenging for attackers to detect the hidden information.

- **Chaotic Code:**

  Instead of linear codes, we opted for Chaotic Code. Chaotic systems exhibit sensitive dependence on initial conditions, making them suitable for encryption. Chaotic systems can be used for secure key generation, data scrambling, and even watermarking.

- **Deep Learning and Security:**

  You rightly pointed out that these techniques can also benefit deep learning. For instance, using steganography to hide adversarial examples or enhancing model robustness with Chaotic Code.

Appendix

MATLAB

CHAPTER 1

I. RSA

Code

```
% Step 1: Key Generation
p = 73; % Prime number 1
q = 151; % Prime number 2

n = p * q;
phi_n = (p - 1) * (q - 1);

% Choose a public exponent e
e = 11;

% Calculate the private exponent d
d = modinv(e, phi_n);

% Step 2: Encryption
originalMessage = 'HI';
ciphertext = modexp(originalMessage, e, n);

% Step 3: Decryption
decrypted_ascii = modexp(ciphertext, d, n);
decryptedMessage = char(decrypted_ascii);
```

Modinv

```
% Modular Multiplicative Inverse
function inv = modinv(a, m)
    [g, x, ~] = gcd(a, m); %compute the greatest common divisor
    if g == 1 %The GCD being equal to 1 is a
        %necessary condition for the existence
        %of the modular inverse. If the GCD is 1, the inverse exists.
        inv = mod(x, m);
    else
        error('Modular inverse does not exist.');
    end
end
```

Modexp

```
% Modular Exponentiation
function result = modexp(base, exponent, modulus)
    result = 1;
    base = mod(base, modulus);

    while exponent > 0
        if mod(exponent, 2) == 1
            result = mod(result.* base, modulus);
        end
        exponent = floor(exponent / 2);
        base = mod(base.^2, modulus);
    end
end
```

II. DSSS

```
Rb = 1000;           % Bit rate of data input (bits per second)
Rc = 10000;          % Chip rate of PN bit stream (chips per second)
Tb = 1/Rb;           % Bit duration (seconds)
Tc = 1/Rc;           % Chip duration (seconds)
SpreadingGain = Rc/Rb;  % Spreading gain

% Generate random data signal (0s and 1s)
dataInput = randi([0, 1], 1, 100); % Adjust the length as needed

% Generate spreading code (PN sequence)
spreadingCode = randi([0, 1], 1, length(dataInput) * SpreadingGain);

% Repeat each bit in the spreading code to match the chip rate
spreadingCode = repmat(spreadingCode, 1, round(Tb/Tc));

% Spread the data signal using XOR with the spreading code
spreadSignal = xor(dataInput, spreadingCode(1:length(dataInput)));

% Receiver: XOR the received spread signal with the spreading code
receivedData = xor(spreadSignal, spreadingCode(1:length(dataInput)));
```

III. PN Code Example

```
% Define the parameters of the LFSR
n = 3; % number of bits in the LFSR
tapPositions = [3, 1]; % tap positions for feedback

% Initialize the LFSR with the given seed
seed = [1, 0, 0]; % initial seed for the LFSR

% Generate the PN code using the LFSR with XOR between first and third registers
pnCodeLength = 7; % length of the PN code to generate
pnCode = zeros(1, pnCodeLength); % initialize the PN code
lfsr = seed; % initialize the LFSR with the seed

for i = 1:pnCodeLength
    pnCode(i) = xor(lfsr(1), lfsr(3)); % output the XOR of the first and third registers as the PN code bit
    feedbackBit = mod(sum(lfsr(tapPositions)), 2); % calculate the feedback bit
    lfsr = [feedbackBit, lfsr(1:end-1)]; % shift the LFSR register and insert the feedback bit at the first position
end
disp(pnCode)
```

```
for i = 1:pnCodeLength
    pnCode1(i) = lfsr(1); % output the last bit of the LFSR as the PN code bit
    pnCode2(i) = lfsr(2); % output the last bit of the LFSR as the PN code bit
    pnCode3(i) = lfsr(3); % output the last bit of the LFSR as the PN code bit
    feedbackBit = xor(lfsr(end),lfsr(end-2)); % calculate the feedback bit
    lfsr = [feedbackBit, lfsr(1:end-1)]; % shift the LFSR register and insert the feedback bit at the first position
end
disp(pnCode1)
disp(pnCode2)
disp(pnCode3)


% CDMA Spread Spectrum with PN Codes
% Define parameters
numUsers = 2;         % Number of users
spreadingGain = 3;    % Spreading gain
messageUser1 = [1 0 1]   % Original message for User 1
desiredSpreadedMessage1 = repelem(messageUser1, spreadingGain);
desiredSpreadedMessage1(desiredSpreadedMessage1 == 0) = -1;
```

```
% If spreading gain is greater than 1, repeat each symbol
spreadUser1Repeated = repmat(pnCode2, 1, spreadingGain);
spreadUser2Repeated = repmat(pnCode3, 1, spreadingGain);

% Trim the spreaded message to the desired length
trimmedSpreadedMessage1 = spreadUser1Repeated(1:length(desiredSpreadedMessage1));
trimmedSpreadedMessage1(trimmedSpreadedMessage1 == 0) = -1;

trimmedSpreadedMessage2 = spreadUser2Repeated(1:length(desiredSpreadedMessage2));
trimmedSpreadedMessage2(trimmedSpreadedMessage2 == 0) = -1;


spreadsignal1=desiredSpreadedMessage1.*trimmedSpreadedMessage1
spreadsignal2=desiredSpreadedMessage2.*trimmedSpreadedMessage2

Totalspreadsignal=spreadsignal1+spreadsignal2;
receive1=Totalspreadsignal.*trimmedSpreadedMessage1;
receive2=Totalspreadsignal.*trimmedSpreadedMessage2;


numGroups = numel(receive1) / spreadingGain;
reshapedMatrix = reshape(receive1, spreadingGain, numGroups).';
```

```
% Sum each group of 4 columns
columnSums = sum(reshapedMatrix, 2);

% Convert based on the sum
 receiver1= columnSums >= 4;

numGroups = numel(receive2) / spreadingGain;
reshapedMatrix = reshape(receive2, spreadingGain, numGroups).';

% Sum each group of 4 columns
columnSums = sum(reshapedMatrix, 2);

% Convert based on the sum
receiver2 = columnSums >= 4;
```

IV. Generate Walsh Code

```
function walshMatrix = generateWalshMatrix(N)
w=input('Enter first bit of Walsh code= ');
if w==1
    if log2(N) ~= round(log2(N))
        error('N must be a power of 2.');
    end
    % Generate Hadamard matrix
    hadamardMatrix = hadamard(N);
    % Adjust signs to convert Hadamard to Walsh
    walshMatrix = hadamardMatrix;
    walshMatrix(hadamardMatrix == -1) = 0;
else
    if log2(N) ~= round(log2(N))
        error('N must be a power of 2.');
    end
    % Generate Hadamard matrix
    hadamardMatrix = hadamard(N);
    % Adjust signs to convert Hadamard to Walsh
    walshMatrix = -1*hadamardMatrix;
    walshMatrix(hadamardMatrix == 1) = 0;
end
end
```

V. Walsh Code Example

```
% CDMA Spread Spectrum with Walsh Codes
% Define parameters
numUsers = 2;          % Number of users
spreadingGain = 4;     % Spreading gain
messageUser1 = [1 0 1]    % Original message for User 1
desiredSpreadedMessage1 = repelem(messageUser1, spreadingGain);
desiredSpreadedMessage1(desiredSpreadedMessage1 == 0) = -1;

messageUser2 = [1 1 0]    % Original message for User 2
desiredSpreadedMessage2 = repelem(messageUser2, spreadingGain);
desiredSpreadedMessage2(desiredSpreadedMessage2 == 0) = -1;

% Generate Walsh Code matrix
walshMatrix =generateWalshMatrix(8);  % Assuming an 8x8 Walsh matrix

% Select specific row for the user
selectedRowsUser1 = 2;  % Selected row for User 1
selectedRowsUser2 = 8;  % Selected row for User 2

% Extract selected row for the user
walshCodeUser1 = walshMatrix(selectedRowsUser1, :)
walshCodeUser2 = walshMatrix(selectedRowsUser2, :)

% If spreading gain is greater than 1, repeat each symbol
spreadUser1Repeated = repmat(walshCodeUser1, 1, spreadingGain);
spreadUser2Repeated = repmat(walshCodeUser2, 1, spreadingGain);

% Trim the spreaded message to the desired length
trimmedSpreadedMessage1 = spreadUser1Repeated(1:length(desiredSpreadedMessage1));
trimmedSpreadedMessage1(trimmedSpreadedMessage1 == 0) = -1;

trimmedSpreadedMessage2 = spreadUser2Repeated(1:length(desiredSpreadedMessage2));
trimmedSpreadedMessage2(trimmedSpreadedMessage2 == 0) = -1;


spreadsignal1=desiredSpreadedMessage1.*trimmedSpreadedMessage1
spreadsignal2=desiredSpreadedMessage2.*trimmedSpreadedMessage2

Totalspreadsignal=spreadsignal1+spreadsignal2;
receive1=Totalspreadsignal.*trimmedSpreadedMessage1;
receive2=Totalspreadsignal.*trimmedSpreadedMessage2;


numGroups = numel(receive1) / spreadingGain;
reshapedMatrix = reshape(receive1, spreadingGain, numGroups).';

% Sum each group of 4 columns
columnSums = sum(reshapedMatrix, 2);

% Convert based on the sum
 receiver1= columnSums >= 4;

numGroups = numel(receive2) / spreadingGain;
reshapedMatrix = reshape(receive2, spreadingGain, numGroups).';

% Sum each group of 4 columns
columnSums = sum(reshapedMatrix, 2);

% Convert based on the sum
receiver2 = columnSums >= 4;
```

VI. Gold Code and C/A Code

```
% Example usage
seed1 = [1,1,1,1,1,1,1,1,1,1];
tap1 = [3, 10];  % XOR between bits 3 and 10
seed2 = [1,1,1,1,1,1,1,1,1,1];
tap2 = [2, 3, 6, 8, 9, 10];  % XOR between bits 2, 3, 6, 8, 9, and 10
code_length = 10;

% Generate the Gold code with the custom polynomial tap sequences
first = generate_gold_code_custom(seed1, tap1, seed2, tap2, code_length);
```

```
%from first 10 bit we detect the space vehicle
code_length = 10;
    lfsr1 = seed1;
    lfsr2 = seed2;

    code = zeros(1, code_length);

    for i = 1:code_length
        % Output the XOR of the specified taps for the first LFSR
        feedback_bit1 = xor(lfsr1(tap1(1)), lfsr1(tap1(2)));
        A = lfsr1(tap1(2));%the MSBs of the polynomial 1
        % Output the XOR of the specified taps for the second LFSR
        feedback_bit2 = xor(lfsr2(tap2(1)), xor(lfsr2(tap2(2)), xor(lfsr2(tap2(3)), xor(lfsr2(tap2(4)), xor(lfsr2(tap2(5)), lfsr2(tap2(6)))))));
```

```
B=xor(lfsr2(2),lfsr2(6));%xor between 2,6 from polynomial 2
C=xor(lfsr2(3),lfsr2(7));%xor between 3,7 from polynomial 2
D=xor(lfsr2(4),lfsr2(8));%xor between 4,8 from polynomial 2
E=xor(lfsr2(5),lfsr2(9));%xor between 5,9 from polynomial 2
F=xor(lfsr2(1),lfsr2(9));%xor between 1,9 from polynomial 2
G=xor(lfsr2(2),lfsr2(10));
H=xor(lfsr2(1),lfsr2(8));
I=xor(lfsr2(2),lfsr2(9));
J=xor(lfsr2(3),lfsr2(10));
K=xor(lfsr2(2),lfsr2(3));
L=xor(lfsr2(3),lfsr2(4));
M=xor(lfsr2(5),lfsr2(6));
N=xor(lfsr2(6),lfsr2(7));
O=xor(lfsr2(7),lfsr2(8));
P=xor(lfsr2(8),lfsr2(9));
Q=xor(lfsr2(9),lfsr2(10));
```

```
r=xor(lfsr2(1),lfsr2(4));
s=xor(lfsr2(2),lfsr2(5));
t=xor(lfsr2(3),lfsr2(6));
u=xor(lfsr2(4),lfsr2(7));
v=xor(lfsr2(5),lfsr2(8));
w=xor(lfsr2(6),lfsr2(9));
x=xor(lfsr2(1),lfsr2(3));
y=xor(lfsr2(4),lfsr2(6));
z=xor(lfsr2(5),lfsr2(7));
aa=xor(lfsr2(6),lfsr2(8));
bb=xor(lfsr2(7),lfsr2(9));
cc=xor(lfsr2(8),lfsr2(10));
dd=xor(lfsr2(1),lfsr2(6));
ee=xor(lfsr2(2),lfsr2(7));
ff=xor(lfsr2(3),lfsr2(8));
gg=xor(lfsr2(4),lfsr2(9));
```

```
% Output the XOR of the MSBs of the first LFSRs
Space_Vehicle_1(i) = xor(A,B);%first space vehicle
Space_Vehicle_2(i) = xor(A,C);%second space vehicle
Space_Vehicle_3(i) = xor(A,D);%third space vehicle
Space_Vehicle_4(i) = xor(A,E);%fourth space vehicle
Space_Vehicle_5(i) = xor(A,F);%fifth space vehicle
Space_Vehicle_6(i) = xor(A,G);
Space_Vehicle_7(i) = xor(A,H);
Space_Vehicle_8(i) = xor(A,I);
Space_Vehicle_9(i) = xor(A,J);
Space_Vehicle_10(i) = xor(A,K);
Space_Vehicle_11(i) = xor(A,L);
Space_Vehicle_12(i) = xor(A,M);
Space_Vehicle_13(i) = xor(A,N);
Space_Vehicle_14(i) = xor(A,O);
Space_Vehicle_15(i) = xor(A,P);
Space_Vehicle_16(i) = xor(A,Q);
```

```
    Space_Vehicle_17(i) = xor(A,r);
    Space_Vehicle_18(i) = xor(A,s);
    Space_Vehicle_19(i) = xor(A,t);
    Space_Vehicle_20(i) = xor(A,u);
    Space_Vehicle_21(i) = xor(A,v);
    Space_Vehicle_22(i) = xor(A,w);
    Space_Vehicle_23(i) = xor(A,x);
    Space_Vehicle_24(i) = xor(A,y);
    Space_Vehicle_25(i) = xor(A,z);
    Space_Vehicle_26(i) = xor(A,aa);
    Space_Vehicle_27(i) = xor(A,bb);
    Space_Vehicle_28(i) = xor(A,cc);
    Space_Vehicle_29(i) = xor(A,dd);
    Space_Vehicle_30(i) = xor(A,ee);
    Space_Vehicle_31(i) = xor(A,ff);
    Space_Vehicle_32(i) = xor(A,gg);

    % Shift the LFSR contents
    lfsr1 = [feedback_bit1, lfsr1(1:end-1)];
    lfsr2 = [feedback_bit2, lfsr2(1:end-1)];
end
```

CHAPTER 2

I. Vigenère

Encryption

```
function ciphered_text = vigenere_encrypt(plain_text, key)
% Convert the key to lowercase
    key = lower(key);
    % Initialize the ciphertext
    ciphered_text = '';
    % Encryption
    for i = 1:length(plain_text)
        if isletter(plain_text(i)) % Process only alphabetic letters
            % Convert to lowercase for consistency
            plain_char = lower(plain_text(i));
            key_char = key(mod(i - 1, length(key)) + 1); % Repeating key
            key_num = double(key_char) - 97;
            plain_num = double(plain_char) - 97;
            ciphered_num = mod(plain_num + key_num, 26);
```

```
            % Preserve original case
            if ismember(plain_text(i), 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
                ciphered_text = [ciphered_text, char(ciphered_num + 65)];
            else
                ciphered_text = [ciphered_text, char(ciphered_num + 97)];
            end
        else
            % Non-letter character remains unchanged
            ciphered_text = [ciphered_text, plain_text(i)];
        end
    end
end
```

Decryption

```
function plain_text = vigenere_decrypt(ciphered_text, key)
E:\Seconterm_4\Graduation Project\FULL Code\vigenere_encrypt.m
    key = lower(key);

    % Initialize the plaintext
    plain_text = '';

    % Decryption
    for i = 1:length(ciphered_text)
        if isletter(ciphered_text(i)) % Process only alphabetic letters
            % Convert to lowercase for consistency
            ciphered_char = lower(ciphered_text(i));
            key_char = key(mod(i - 1, length(key)) + 1); % Repeating key
            key_num = double(key_char) - 97;
            ciphered_num = double(ciphered_char) - 97;
            deciphered_num = mod(ciphered_num - key_num, 26);
            % Preserve original case
            if ismember(ciphered_text(i), 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
                plain_text = [plain_text, char(deciphered_num + 65)];
            else
                plain_text = [plain_text, char(deciphered_num + 97)];
            end
        else
            % Non-letter character remains unchanged
            plain_text = [plain_text, ciphered_text(i)];
        end
    end
end
```

II. Advanced Hide text in Image

```
% Get the cover image
x = imread('lena.png');
x=rgb2gray(x);
```

```
%% Encrypt Image using Stream cipher

[n1 n2]=size(x);
key1=uint8(255*rand(2000,2000));%key1
pk=key1(1:n1,1:n2);
enc=bitxor(x,pk);
```

```
% Get message to embed in image
textMessage='You can connect with me through abdobadr@gmail.com.';
len=length(textMessage);
disp(['Input Text: ' textMessage]);
key2 = 'key';
disp(['Keyword: ' key2]);

%% Encrypt message using the Vigenère cipher
textMessage = vigenere_encrypt(textMessage, key2);
disp(['Ciphertext: ' textMessage]);

%% Embeed Encrypted message in Encrypted Image using LSB
% Ensure that the input message does not exceed the size of the cover image
assert(numel(enc) > numel(textMessage)*8, 'ERROR: message is too large for cover image');

% Insert some uncommon character to append to the message text as a
% terminator.
eof = 'þ';
messageText = [uint8(textMessage), eof];

% Each character needs to be converted into 8-bit binary form
binary = transpose(dec2bin(messageText, 8));
% Find the least-significant bits to be set to one or zero.
zeroBit = find(binary == '0');
oneBit = find(binary == '1');
% Set the values of the least-significant bits
enc(zeroBit) = bitset(enc(zeroBit), 1, 0);
enc(oneBit) = bitset(enc(oneBit), 1, 1);

%% Decrypt Image
dec1=bitxor(enc,pk);
dec=noisy;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Enctypted image
[n3, n4]=size(dec);
pk2=key1(1:n3,1:n4);
cover=bitxor(dec,pk2);

%% Extract Encrypted message from Encrypted Image
OutputMessage = [];
for i = 1:8:numel(cover)
    chars = bitget(cover(i:i+7), 1);
    chars = bin2dec(num2str(chars));
    if(chars == eof)
        break;
    else
        OutputMessage(end+1) = chars;
    end

end
Extracted_Message = char(OutputMessage);
```

III. **Modified**

```
%% Divide into 4 parts

[rows, columns, numberOfColorChannels] = size(enc);

% Calculate the middle row and column
middleRow = round(rows / 2);
middleColumn = round(columns / 2);

% Divide the image into four equal parts
A = enc(1:middleRow, 1:middleColumn, :);
C = enc(1:middleRow, middleColumn+1:end, :);
B = enc(middleRow+1:end, 1:middleColumn, :);
D = enc(middleRow+1:end, middleColumn+1:end, :);
```

CHAPTER 3

Project of Hide text in Sound

1. Transmitter

```
% Enhanced Audio Steganography Transmitter over Wireless Channel
clear;
clc;

% Load the cover audio file (wav format)
[coverAudio, fs] = audioread('smooth-ac-guitar-loop-93bpm-137706.wav');
coverAudio = coverAudio(:, 1); % Use only one channel for simplicity

% Convert cover audio to 16-bit PCM
coverAudio = int16(coverAudio * 32767);

% Load the secret message
secretMessage = 'This is a secret message.';
msgBin = dec2bin(secretMessage, 8)';
msgBin = msgBin(:);

% Calculate the number of samples needed
numSamplesNeeded = length(msgBin);

% Check if the audio file is long enough to hide the message
if numSamplesNeeded > length(coverAudio)
    error('Cover audio is too short to hide the secret message.');
end

% Embed the secret message into the cover audio using LSB method
stegoAudio = coverAudio;
for i = 1:numSamplesNeeded
    stegoAudio(i) = bitset(stegoAudio(i), 1, str2double(msgBin(i)));
end
```

```
30
31  % Simulate a wireless channel with additional parameters
32  snr = 20; % Signal-to-noise ratio in dB
33  maxDopplerShift = 5; % Maximum Doppler shift in Hz
34  delayVector = [0 1e-5 3e-5]; % Path delays in seconds
35  gainVector = [0 -3 -6]; % Average path gains in dB
36
37  % Create the Rayleigh fading channel object with Doppler effect
38  fadingChannel = comm.RayleighChannel('SampleRate', fs, ...
39                                       'PathDelays', delayVector, ...
40                                       'AveragePathGains', gainVector, ...
41                                       'MaximumDopplerShift', maxDopplerShift);
42
43  % Transmit the stego audio through the wireless channel
44  stegoAudioDouble = double(stegoAudio) / 32767;
45  fadedSignal = fadingChannel(stegoAudioDouble);
46  receivedSignal = awgn(fadedSignal, snr, 'measured');
47
48  % Convert received signal back to 16-bit PCM
49  receivedAudio = int16(real(receivedSignal) * 32767);
50
51  % Save the stego audio to a file
52  audiowrite('stego_audio.wav', double(stegoAudio) / 32767, fs);
53
54  % Save the received audio to a file
55  audiowrite('received_audio.wav', double(receivedAudio) / 32767, fs);
56
57  % Plot the original, stego, and received audio signals
58  subplot(3, 1, 1);
59  plot(coverAudio);
60  title('Original Cover Audio');
61  xlabel('Sample Number');
62  ylabel('Amplitude');
63
64  subplot(3, 1, 2);
65  plot(stegoAudio);
66  title('Stego Audio');
67  xlabel('Sample Number');
68  ylabel('Amplitude');
69
70  subplot(3, 1, 3);
71  plot(receivedAudio);
72  title('Received Audio after Wireless Channel');
73  xlabel('Sample Number');
74  ylabel('Amplitude');
75
76  disp('Stego audio and received audio have been saved to files.');
77
```

2. Receiver

```
1   % Receiver
2
3   try
4       % Load the received audio file
5       disp('Loading received audio...');
6       [receivedAudio, fs] = audioread('received_audio.wav');
7       disp('Received audio loaded successfully.');
8
9       % Ensure single-channel (if stereo)
10      receivedAudio = receivedAudio(:, 1);
11
12      % Convert received audio to double precision
13      receivedAudioDouble = double(receivedAudio) / 32767;
14
15      % Parameters from transmitter (make sure they match)
16      snr = 20; % Signal-to-noise ratio in dB
17      maxDopplerShift = 5; % Maximum Doppler shift in Hz
18      delayVector = [0 1e-5 3e-5]; % Path delays in seconds
19      gainVector = [0 -3 -6]; % Average path gains in dB
20      msgLength = length('Hello') * 8; % Expected length of binary message
21
```

```
        % Initialize Rayleigh fading channel object with the same parameters
        disp('Applying Rayleigh fading channel...');
        fadingChannel = comm.RayleighChannel('SampleRate', fs, ...
                                             'PathDelays', delayVector, ...
                                             'AveragePathGains', gainVector, ...
                                             'MaximumDopplerShift', maxDopplerShift);

        % Apply the fading channel inverse to cancel out the channel effects
        receivedAudioChannelCorrected = fadingChannel(receivedAudioDouble);
        disp('Rayleigh fading applied successfully.');

        % Demodulate the received audio (remove noise)
        disp('Demodulating received audio...');
        receivedAudioDemod = awgn(receivedAudioChannelCorrected, snr, 'measured');
        disp('Demodulation completed.');

        % Convert demodulated signal back to 16-bit PCM
        receivedAudioDemodInt16 = int16(real(receivedAudioDemod) * 32767);

         % Extract the secret message from the demodulated signal using LSB method
         extractedMsgBin = '';
         for i = 1:msgLength
             % Extract LSB from each sample until enough bits are collected
             extractedBit = bitget(receivedAudioDemodInt16(i), 1);
             extractedMsgBin = [extractedMsgBin, num2str(extractedBit)];
         end

         % Convert binary message back to characters
         extractedMessage = char(bin2dec(reshape(extractedMsgBin, 8, [])'));

         % Display the extracted message
         fprintf('Extracted secret message: %s\n', extractedMessage);

    catch ME
        % Display any errors that occurred
        disp(['Error occurred: ', ME.message]);
    end
```

CHAPTER 4

1. Hiding Image message in a sound file

```
% Advanced LSB Steganography in MATLAB

%% Step 1: Read Image and Sound File

% Read the image file
imageFile = 'moon.png';
imageData = imread(imageFile);

% Convert image to grayscale if necessary
if ndims(imageData) == 3 % Check if image is RGB
    imageData = rgb2gray(imageData);
end

% Ensure image data is in double precision for manipulation
imageData = double(imageData);

% Read the sound file
soundFile = 'scary-audio-19409.wav';
[soundData, Fs] = audioread(soundFile);

% Ensure sound data is in double precision
soundData = double(soundData);
%% Step 2: plot original sound data
figure;
subplot(2, 2, [1, 2]);
t = (0:length(soundData)-1) / Fs; % Time vector
plot(t, soundData, 'b');
title('Original Sound Data');
xlabel('Time (s)');
ylabel('Amplitude');
xlim([0, length(soundData)/Fs]);

```

```
%% Step 3: Convert Image to Binary Data

% Convert image data to binary
imageDataBinary = dec2bin(imageData(:), 8); % Convert each pixel value to 8-bit binar
imageDataBinary = imageDataBinary(:)'; % Convert to row vector

%% Step 4: Embedding Process (LSB Method)

% Initialize variables
imageIndex = 1;
bitDepth = 8; % Assume 8-bit image for simplicity

% Embed image data into sound samples using LSB method
for i = 1:length(soundData)
    if imageIndex <= length(imageDataBinary)
        % Extract the current sound sample and convert to uint8 for bit manipulation
        sample = uint8(soundData(i)); % Convert to uint8

        % Replace the LSBs of the sample with bits from image data
        sampleLSB = bitget(sample, 1); % Extract current LSB

         % Replace the LSB with the image bit
         if imageIndex <= length(imageDataBinary)
             sampleLSB = bitset(sampleLSB, 1, str2double(imageDataBinary(imageIndex)));
             imageIndex = imageIndex + 1;
         end

         % Replace the LSB in the original sample and convert back to double
         modifiedSample = double(bitset(sample, 1, sampleLSB));

         % Update the sound data with modified sample
         soundData(i) = modifiedSample;
     end
end

%% Step 5: Save Modified Sound File (Optional)

% Save the modified sound data to a new WAV file
outputFile = 'sound_with_hidden_image.wav';
audiowrite(outputFile, soundData, Fs);

fprintf('Embedding process completed. Modified sound file saved as %s.\n', outputFile);
%% Step 6: Plot Modified Sound Data

% Plot modified sound data
subplot(2, 2, [3, 4]);
plot(t, modifiedSoundData, 'r');
title('Modified Sound Data with Embedded Image');
xlabel('Time (s)');
ylabel('Amplitude');
xlim([0, length(soundData)/Fs]);
```

```
%% Step 7: Extraction (Optional)

% Example extraction process (to verify)
extractedImageBinary = '';

% Extract image data from sound samples (reverse process)
for i = 1:length(soundData)
    % Extract LSB from each sample and convert to uint8
    sample = uint8(soundData(i));
    extractedBit = bitget(sample, 1); % Extract LSB
    extractedImageBinary = [extractedImageBinary num2str(extractedBit)]; % Ap

    % Break if reached end of image data
    if length(extractedImageBinary) >= length(imageDataBinary)
        break;
    end
end

% Convert extracted binary data back to image matrix
extractedImageBinary = reshape(extractedImageBinary, bitDepth, [])'; % Reshap
extractedImage = bin2dec(extractedImageBinary); % Convert binary back to deci
extractedImage = reshape(extractedImage, size(imageData)); % Reshape to origi

% Display the extracted image
figure;
subplot(1, 2, 1);
imshow(uint8(imageData));
title('Original Image');

subplot(1, 2, 2);
imshow(uint8(extractedImage));
title('Extracted Image');

fprintf('Extraction process completed.\n');
```

2. Hiding Image message in a sound file update

```
% Optimized LSB Steganography with Enhanced Security, Reliability, and Adaptive Embedding

%% Step 1: Read Image and Sound File

% Read the image file
imageFile = 'moon.png';
imageData = imread(imageFile);

% Convert image to grayscale if necessary
if ndims(imageData) == 3 % Check if image is RGB
    imageData = rgb2gray(imageData);
end

% Read the sound file
soundFile = 'scary-audio-19409.wav';
[soundData, Fs] = audioread(soundFile);

%% Step 2: Convert and Compress Image Data

% Compress image data using JPEG compression
imwrite(imageData, 'compressedImage.jpg', 'jpg');

% Read the compressed image data back
compressedImageData = imread('compressedImage.jpg');

% Convert compressed image data to binary
compressedImageDataBinary = dec2bin(compressedImageData(:), 8);
compressedImageDataBinary = compressedImageDataBinary(:)'; % Convert to row vector

% Display first 100 bits of compressed image data in binary
disp('Original Compressed Image Data (Binary):');
disp(compressedImageDataBinary(1:100));
```

```
33
34  %% Step 3: Encrypt Image Data using Stream Cipher (XOR)
35
36  % Define a secret key for stream cipher (example key)
37  key = 'ThisIsASecretKey1234567890123456'; % Example key
38
39  % Encrypt the compressed image data using XOR-based stream cipher
40  encryptedData = StreamCipherEncrypt(uint8(compressedImageDataBinary - '0')', key);
41
42  % Convert encrypted data to binary
43  encryptedDataBinary = dec2bin(encryptedData, 8);
44  encryptedDataBinary = encryptedDataBinary(:)'; % Convert to row vector
45
46  % Display first 100 bits of encrypted data in binary
47  disp('Encrypted Data (Binary):');
48  disp(encryptedDataBinary(1:100));
49
50  %% Step 4: Add Error Correction Codes (ECC)
51
52  % Convert binary string to binary vector for ECC
53  msg = double(encryptedDataBinary - '0'); % Convert char to double binary values
54  msg = reshape(msg, [], 4); % Reshape for 4-bit encoding
55
56  % Use a simple Hamming Code (7,4) for error correction
57  encMsg = encode(msg, 7, 4, 'hamming/binary');
58
59  % Convert ECC encoded data back to binary
60  encDataBinary = encMsg(:)'; % Convert to row vector
61
62  % Display first 100 bits of ECC encoded data in binary
63  disp('ECC Encoded Data (Binary):');
64  disp(encDataBinary(1:100));

65
66  %% Step 5: Generate Pseudorandom and Adaptive LSB Positions
67
68  % Seed for PRNG (should be kept secret)
69  rng(12345);
70
71  % Generate pseudorandom positions for embedding
72  numSamples = length(soundData);
73  numBits = length(encDataBinary);
74
75  % Calculate the absolute value of the sound data to prioritize higher amplitude regions
76  absSoundData = abs(soundData);
77
78  % Sort indices by amplitude in descending order to prioritize high-amplitude samples
79  [~, sortedIndices] = sort(absSoundData, 'descend');
80
81  % Select top positions for embedding
82  randomPositions = sortedIndices(1:numBits);
83
84  %% Step 6: Embedding Process (LSB Method)
85
86  % Initialize variables
87  modifiedSoundData = soundData; % Create a copy for modification
88
89  % Embed encrypted data into sound samples using LSB method
90  for i = 1:numBits
91      % Extract the current sound sample and convert to uint8 for bit manipulation
92      sampleIndex = randomPositions(i);
93      sample = uint8(modifiedSoundData(sampleIndex) * 255); % Scale to 8-bit integer
```

```
94
95          % Replace the LSB with the encrypted bit
96          sampleLSB = bitset(sample, 1, str2double(encDataBinary(i)));
97
98          % Replace the LSB in the original sample and convert back to double
99          modifiedSample = double(sampleLSB) / 255;
100
101         % Update the sound data with modified sample
102         modifiedSoundData(sampleIndex) = modifiedSample;
103     end
104
105     %% Step 7: Save Modified Sound File
106
107     % Save the modified sound data to a new WAV file
108     outputFile = 'sound_with_hidden_image.wav';
109     audiowrite(outputFile, modifiedSoundData, Fs);
110
111     fprintf('Embedding process completed. Modified sound file saved as %s.\n', out
112
113     %% Extraction Process (Optional)
114
115     % Extract encrypted data from modified sound samples (reverse process)
116     extractedDataBinary = char(zeros(1, numBits)); % Preallocate memory
117
118     for i = 1:numBits
119         % Extract LSB from each sample using random positions
120         sampleIndex = randomPositions(i);
121         sample = uint8(modifiedSoundData(sampleIndex) * 255);
122         extractedBit = bitget(sample, 1); % Extract LSB
123         extractedDataBinary(i) = num2str(extractedBit); % Append extracted bit
124     end
```

```
125
126 % Convert extracted binary data back to decimal
127 extractedDataBinary = reshape(extractedDataBinary, 7, [])';
128 extractedData = bin2dec(extractedDataBinary); % Convert binary back to decimal
129
130 % Convert extracted decimal data to binary matrix
131 extractedDataBinaryMatrix = de2bi(extractedData, 7, 'left-msb'); % Convert to binary matrix
132
133 % Decode ECC
134 decMsg = decode(extractedDataBinaryMatrix(:)', 7, 4, 'hamming/binary');
135
136 % Convert decoded data back to binary
137 decDataBinary = dec2bin(decMsg, 4);
138 decDataBinary = decDataBinary(:)'; % Convert to row vector
139
140 % Decrypt extracted data
141 decryptedData = StreamCipherDecrypt(uint8(decDataBinary - '0')', key);
142
143 % Ensure the decrypted data has the correct length
144 decryptedDataBinary = dec2bin(decryptedData, 8);
145 decryptedDataBinary = decryptedDataBinary(:)';
146
147 % Calculate the expected size based on the original compressed image data
148 expectedSize = numel(compressedImageData) * 8;
149
150 % Adjust the size of decryptedDataBinary if necessary
151 if length(decryptedDataBinary) > expectedSize
152     decryptedDataBinary = decryptedDataBinary(1:expectedSize);
153 elseif length(decryptedDataBinary) < expectedSize
154     decryptedDataBinary = [decryptedDataBinary, repmat('0', 1, expectedSize - length(decryptedDataBinary))];
155 end
```

```
% Convert decrypted binary data back to image matrix
decryptedImage = reshape(uint8(bin2dec(reshape(decryptedDataBinary, 8, [])')), size(compressedImageData));

% Display first 100 bits of decrypted data in binary
disp('Decrypted Data (Binary):');
disp(decryptedDataBinary(1:100));

%% Plotting

% Create figure with tabs
fig = figure('Position', [100, 100, 1000, 600]);
tabGroup = uitabgroup(fig);

% Original Data Tab
tabOriginal = uitab(tabGroup, 'Title', 'Original Data');

% Plot original image
subplot(2, 1, 1, 'Parent', tabOriginal);
imshow(uint8(imageData));
title('Original Image');

% Plot original sound data
subplot(2, 1, 2, 'Parent', tabOriginal);
t = (0:length(soundData)-1) / Fs; % Time vector
plot(t, soundData, 'b');
title('Original Sound Data');
xlabel('Time (s)');
ylabel('Amplitude');
xlim([0, length(soundData)/Fs]);

% Modified Data Tab
tabModified = uitab(tabGroup, 'Title', 'Modified Data');

% Plot modified sound data
subplot(2, 1, 1, 'Parent', tabModified);
plot(t, modifiedSoundData, 'r');
title('Modified Sound Data with Embedded Image');
xlabel('Time (s)');
ylabel('Amplitude');
xlim([0, length(soundData)/Fs]);

% Plot extracted image
subplot(2, 1, 2, 'Parent', tabModified);
imshow(uint8(decryptedImage));
title('Extracted Image');

fprintf('Extraction process completed.\n');

%% Helper Functions for Stream Cipher Encryption/Decryption

function encryptedData = StreamCipherEncrypt(data, key)
    % Encrypt the data using a simple XOR-based stream cipher
    % Process data in chunks to avoid large array creation
    chunkSize = 1024; % Define a reasonable chunk size
    keyStream = repmat(uint8(key), ceil(chunkSize / length(key)), 1);
    keyStream = keyStream(:); % Ensure keyStream is a column vector
    encryptedData = zeros(size(data), 'uint8');

    for i = 1:chunkSize:length(data)
        endIndex = min(i + chunkSize - 1, length(data));
        dataChunk = data(i:endIndex);
        keyStreamChunk = keyStream(1:length(dataChunk));
        encryptedData(i:endIndex) = bitxor(dataChunk, keyStreamChunk);
    end
end

function decryptedData = StreamCipherDecrypt(data, key)
    % Decrypt the data using a simple XOR-based stream cipher (same as encryption)
    decryptedData = StreamCipherEncrypt(data, key); % XOR operation is its own inverse
end

function decimalVector = binaryVectorToDecimal(binaryVector)
    % Convert a binary vector to a decimal vector
    decimalVector = [];
    for i = 1:8:length(binaryVector)
        decimalVector = [decimalVector; bin2dec(binaryVector(i:i+7))];
    end
end
```

CHAPTER 6

Choatic Code

I. Sensitivity

```
% MATLAB code to illustrate chaotic systems sensitivity to initial conditions
% Parameters
alpha = 3.9; % Logistic map parameter
iterations = 100; % Number of iterations
x0_1 = 0.1; % Initial condition 1
x0_2 = 0.10001; % Initial condition 2 (slightly different)

% Arrays to store iteration values
x1 = zeros(1, iterations);
x2 = zeros(1, iterations);

% Initialize
x1(1) = x0_1;
x2(1) = x0_2;

% Logistic map iteration
for i = 2:iterations
    x1(i) = alpha * x1(i-1) * (1 - x1(i-1));
    x2(i) = alpha * x2(i-1) * (1 - x2(i-1));
end

% Plotting
figure;
subplot(2, 1, 1);
plot(1:iterations, x1, '-o', 'LineWidth', 1.5);
title('Sensitivity to Initial Conditions');
xlabel('Iterations');
ylabel('Value');
grid on;
hold on;
% Plot 2: Initial condition 2
plot(1:iterations, x2, '-s', 'LineWidth', 1.5);
legend('x0 = 0.1', 'x0 = 0.10001', 'Location', 'NorthEast');
% Plot 3: Zoomed-in view
subplot(2, 1, 2);
plot(1:iterations, x1, '-o', 'LineWidth', 1.5);
title('Zoomed-in View');
xlabel('Iterations');
ylabel('Value');
grid on;
hold on;
plot(1:iterations, x2, '-s', 'LineWidth', 1.5);
xlim([0, iterations-50]); % Zoom in to the last 20 iterations
legend('x0 = 0.1', 'x0 = 0.10001', 'Location', 'NorthEast');
% Adjust figure size
set(gcf, 'Position', [100, 100, 800, 600]);
```

II. Pseudo-Randomness

```
% MATLAB code to illustrate pseudo-randomness in chaotic systems (Logistic Map)
% Parameters
alpha = 3.93; % Chaotic map parameter
iterations = 250; % Number of iterations
x_initial = rand; % Initial condition for chaotic map

% Initialize array to store chaotic sequence
chaotic_sequence = zeros(1, iterations);

% Compute chaotic sequence
x = x_initial;
for i = 1:iterations
    x = alpha * x * (1 - x);
    chaotic_sequence(i) = x;
end

% Plotting
figure;
% Plot chaotic sequence
subplot(2, 1, 1);
plot(1:iterations, chaotic_sequence, '-o', 'LineWidth', 1.5);
title('Pseudo-randomness in Chaotic Systems');
xlabel('Iterations');
ylabel('Value');
grid on;

% Histogram of chaotic sequence
subplot(2, 1, 2);
histogram(chaotic_sequence, 50, 'Normalization', 'probability');
title('Histogram of Chaotic Sequence');
xlabel('Value');
ylabel('Probability');
grid on;

% Adjust figure size
set(gcf, 'Position', [100, 100, 800, 600]);
```

III. Ergodicity

```
% Parameters
alpha = 3.9;              % Chaotic map parameter
iterations = 1000;        % Number of iterations
ensemble_size = 100;      % Ensemble size for averaging
x_initial = rand;         % Initial condition for chaotic map

% Initialize arrays for storing chaotic sequences
chaotic_sequence = zeros(iterations, ensemble_size);

% Generate chaotic sequences
for ens = 1:ensemble_size
    x = x_initial;   % Reset initial condition for each ensemble
    for i = 1:iterations
        x = alpha * x * (1 - x);
        chaotic_sequence(i, ens) = x;
    end
end

% Plotting
figure;
% Plot individual chaotic sequences
subplot(2, 1, 1);
plot(1:iterations, chaotic_sequence, 'Color', [0.5 0.5 0.5]);  % Gray lines
title('Ergodicity in Chaotic Systems: Individual Sequences');
xlabel('Iterations');
ylabel('Value');
grid on;

% Plot ensemble average
subplot(2, 1, 2);
ensemble_average = mean(chaotic_sequence, 2);  % Calculate mean along rows
plot(1:iterations, ensemble_average, 'k', 'LineWidth', 2);  % Black line
title('Ensemble Average: Temporal and Ensemble Averages');
xlabel('Iterations');
ylabel('Value');
grid on;

% Adjust figure size
set(gcf, 'Position', [100, 100, 800, 600]);
```

IV. Lorenz

```
% Parameters
sigma = 10;
beta = 8/3;
rho = 28;
dt = 0.01;
tspan = 0:dt:150;
% Initial conditions
x0 = 1;
y0 = 1;
z0 = 1;
% Lorenz equations
lorenz = @(t, xyz) [sigma*(xyz(2) - xyz(1));
                    xyz(1)*(rho - xyz(3)) - xyz(2);
                    xyz(1)*xyz(2) - beta*xyz(3)];
% Solve using ode45
[t, xyz] = ode45(lorenz, tspan, [x0, y0, z0]);
% Plotting
figure;
plot3(xyz(:,1), xyz(:,2), xyz(:,3), 'LineWidth', 1);
title('Lorenz Attractor');
xlabel('X');
ylabel('Y');
zlabel('Z');
grid on;

% Adjust figure properties
set(gcf, 'Color', 'w');
view(-37.5, 30);

% End of code
```

V. Logistic Map

```
% Parameters for the logistic map
r_values = linspace(3.5, 4, 500);
num_iterations = 500;
num_skip_iterations = 100;

% Preallocate arrays for efficiency
bifurcation_r = [];
bifurcation_x = [];

% Generate the bifurcation diagram
for r = r_values
    x = rand;  % Start with a random initial condition
    % Skip the initial iterations to let the system settle
    for i = 1:num_skip_iterations
        x = r * x * (1 - x);
    end
    % Store the x values for the bifurcation diagram
    for i = 1:num_iterations
        x = r * x * (1 - x);
        bifurcation_r = [bifurcation_r, r];
        bifurcation_x = [bifurcation_x, x];
    end
end
```

```
% Plot the bifurcation diagram
figure;
plot(bifurcation_r, bifurcation_x, '.', 'MarkerSize', 1);
title('Bifurcation Diagram of the Logistic Map');
xlabel('alpha');
ylabel('x');
xlim([3.5, 4]); % Adjust the x-axis limits
grid on;

% Display the figure
set(gcf, 'Position', [100, 100, 800, 600]);
```

VI. Shuffling Row-Wise

```
% Image Encryption and Decryption using Chaotic Maps

% Load an image
original_image = imread('lena2.jpg'); % Provide the path to your image

% Parameters
alpha = 3.93; % Chaotic map parameter
x_initial = rand; % Initial condition for chaotic map

% Get image dimensions
[rows, cols, channels] = size(original_image);

% Generate key using chaotic map
key = generate_key(alpha, x_initial);

% Encryption
encrypted_image = chaos_encrypt(original_image, rows, cols, channels, alpha, key);

% Decryption
decrypted_image = chaos_decrypt(encrypted_image, rows, cols, channels, alpha, key);

 % Display Images
 figure;
 subplot(1, 2, 1);
 imshow(original_image);
 title('Original Image');

 subplot(1, 2, 2);
 imshow(encrypted_image);
 title('Encrypted Image by shuffling rows-wise');


 % Function to generate chaotic sequence
function chaotic_sequence = generate_chaotic_sequence(alpha, x_initial, image_size)
    chaotic_sequence = zeros(1, image_size);
    x = x_initial;
    for i = 1:image_size
        x = alpha * x * (1 - x);
        chaotic_sequence(i) = x;
    end
    chaotic_sequence = mod(floor(chaotic_sequence * image_size), image_size) + 1;
end
```

```
% Function to generate key using chaotic map
function key = generate_key(alpha, x_initial)
    x = x_initial;
    for i = 1:1000 % Iterate to get a more "random" value
        x = alpha * x * (1 - x);
    end
    key = x; % Use the chaotic value as the key
end

% Function to encrypt image using chaotic map (shuffle rows)
function encrypted_image = chaos_encrypt(image, rows, cols, channels, alpha, key)
    encrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, rows);
    for channel = 1:channels
        for row = 1:rows
            shuffled_row = mod(row + chaotic_sequence(row), rows);
            if shuffled_row == 0
                shuffled_row = rows;
            end
            encrypted_image(shuffled_row,:,channel) = image(row,:,channel);
        end
    end
end

% Function to decrypt image using chaotic map (unshuffle rows)
function decrypted_image = chaos_decrypt(image, rows, cols, channels, alpha, key)
    decrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, rows);
    for channel = 1:channels
        for row = 1:rows
            shuffled_row = mod(row + chaotic_sequence(row), rows);
            if shuffled_row == 0
                shuffled_row = rows;
            end
            decrypted_image(row,:,channel) = image(shuffled_row,:,channel);
        end
    end
end
```

Column-Wise

```
% Load an image
original_image = imread('lena2.jpg'); % Provide the path to your image

% Parameters
alpha = 3.93; % Chaotic map parameter
x_initial = rand; % Initial condition for chaotic map

% Get image dimensions
[rows, cols, channels] = size(original_image);

% Generate key using chaotic map
key = generate_key(alpha, x_initial);

% Encryption
encrypted_image = chaos_encrypt(original_image, rows, cols, channels, alpha, key);

% Decryption
decrypted_image = chaos_decrypt(encrypted_image, rows, cols, channels, alpha, key);
```

```
% Display Images
figure;
subplot(1, 2, 1);
imshow(original_image);
title('Original Image');

subplot(1, 2, 2);
imshow(encrypted_image);
title('Encrypted Image by shuffling columns-wise');

% Function to generate chaotic sequence
function chaotic_sequence = generate_chaotic_sequence(alpha, x_initial, image_size)
    chaotic_sequence = zeros(1, image_size);
    x = x_initial;
    for i = 1:image_size
        x = alpha * x * (1 - x);
        chaotic_sequence(i) = x;
    end
    chaotic_sequence = mod(floor(chaotic_sequence * image_size), image_size) + 1;
end
 % Function to generate key using chaotic map
function key = generate_key(alpha, x_initial)
    x = x_initial;
    for i = 1:1000 % Iterate to get a more "random" value
        x = alpha * x * (1 - x);
    end
    key = x; % Use the chaotic value as the key
end

 % Function to encrypt image using chaotic map (shuffle columns)
function encrypted_image = chaos_encrypt(image, rows, cols, channels, alpha, key)
    encrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, cols);
    for channel = 1:channels
        for col = 1:cols
            shuffled_col = mod(col + chaotic_sequence(col), cols);
            if shuffled_col == 0
                shuffled_col = cols;
            end
            encrypted_image(:,shuffled_col,channel) = image(:,col,channel);
        end
    end
end

 % Function to decrypt image using chaotic map (unshuffle columns)
function decrypted_image = chaos_decrypt(image, rows, cols, channels, alpha, key)
    decrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, cols);
    for channel = 1:channels
        for col = 1:cols
            shuffled_col = mod(col + chaotic_sequence(col), cols);
            if shuffled_col == 0
                shuffled_col = cols;
            end
            decrypted_image(:,col,channel) = image(:,shuffled_col,channel);
        end
    end
end
```

Pixel Wise

```
% Load an image
original_image = imread('lena2.jpg'); % Provide the path to your image

% Parameters
alpha = 3.93; % Chaotic map parameter
x_initial = rand; % Initial condition for chaotic map

% Get image dimensions
[rows, cols, channels] = size(original_image);

% Generate key using chaotic map
key = generate_key(alpha, x_initial);

% Encryption
encrypted_image = chaos_encrypt(original_image, rows, cols, channels, alpha, key);

% Decryption
decrypted_image = chaos_decrypt(encrypted_image, rows, cols, channels, alpha, key);

  % Display Images
  figure;
  subplot(1, 2, 1);
  imshow(original_image);
  title('Original Image');

  subplot(1, 2, 2);
  imshow(encrypted_image);
  title('Encrypted Image by shuffling pixels-wise');

  % Function to generate chaotic sequence
function chaotic_sequence = generate_chaotic_sequence(alpha, x_initial, image_size)
    chaotic_sequence = zeros(1, image_size);
    x = x_initial;
    for i = 1:image_size
        x = alpha * x * (1 - x);
        chaotic_sequence(i) = x;
    end
    chaotic_sequence = mod(floor(chaotic_sequence * image_size), image_size) + 1;
end
  % Function to generate key using chaotic map
function key = generate_key(alpha, x_initial)
    x = x_initial;
    for i = 1:1000 % Iterate to get a more "random" value
        x = alpha * x * (1 - x);
    end
    key = x; % Use the chaotic value as the key
end
```

```
% Function to encrypt image using chaotic map (shuffle pixels)
function encrypted_image = chaos_encrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols;
    encrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);
    for channel = 1:channels
        channel_data = image(:,:,channel);
        flattened_channel = channel_data(:);
        for i = 1:image_size
            temp = flattened_channel(i);
            % Swap pixels based on chaotic sequence (pixel shuffling)
            swap_index = chaotic_sequence(i);
            flattened_channel(i) = flattened_channel(swap_index);
            flattened_channel(swap_index) = temp;
        end
        encrypted_image(:,:,channel) = reshape(flattened_channel, [rows, cols]);
    end
end
```

```
% Function to decrypt image using chaotic map (unshuffle pixels)
function decrypted_image = chaos_decrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols;
    decrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);
    for channel = 1:channels
        channel_data = image(:,:,channel);
        flattened_channel = channel_data(:);
        for i = image_size:-1:1
            temp = flattened_channel(i);
            % Reverse the pixel shuffling based on chaotic sequence
            swap_index = chaotic_sequence(i);
            flattened_channel(i) = flattened_channel(swap_index);
            flattened_channel(swap_index) = temp;
        end
        decrypted_image(:,:,channel) = reshape(flattened_channel, [rows, cols]);
    end
end
```

VII. Substitution

```
% Load an image
original_image = imread('lena2.jpg'); % Provide the path to your image

% Parameters
alpha = 3.93; % Chaotic map parameter
x_initial = rand; % Initial condition for chaotic map

% Get image dimensions
[rows, cols, channels] = size(original_image);

% Generate key using chaotic map
key = generate_key(alpha, x_initial);

% Encryption
encrypted_image = chaos_encrypt(original_image, rows, cols, channels, alpha, key);

% Decryption
decrypted_image = chaos_decrypt(encrypted_image, rows, cols, channels, alpha, key);
```

```
figure;
subplot(1, 2, 1);
imshow(original_image);
title('Original Image');

subplot(1, 2, 2);
imshow(encrypted_image);
title('Encrypted Image by substitution');

% Function to generate chaotic sequence
function chaotic_sequence = generate_chaotic_sequence(alpha, x_initial, image_size)
    chaotic_sequence = zeros(1, image_size);
    x = x_initial;
    for i = 1:image_size
        x = alpha * x * (1 - x);
        chaotic_sequence(i) = x;
    end
    chaotic_sequence = mod(floor(chaotic_sequence * image_size), image_size) + 1;
end

% Function to generate key using chaotic map
function key = generate_key(alpha, x_initial)
    x = x_initial;
    for i = 1:1000 % Iterate to get a more "random" value
        x = alpha * x * (1 - x);
    end
    key = x; % Use the chaotic value as the key
end

% Function to encrypt image using chaotic map (substitution only)
function encrypted_image = chaos_encrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols * channels;
    encrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);

    % Flatten the image into a single vector
    image_vector = image(:);

    % Apply substitution using chaotic sequence and key
    for i = 1:image_size
        index = chaotic_sequence(i);
        encrypted_image(index) = bitxor(image_vector(index), uint8(key * 255));
    end

    % Reshape the encrypted image to original dimensions
    encrypted_image = reshape(encrypted_image, [rows, cols, channels]);
end

% Function to decrypt image using chaotic map (substitution only)
function decrypted_image = chaos_decrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols * channels;
    decrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);

    % Flatten the image into a single vector
    image_vector = image(:);

    % Reverse substitution using chaotic sequence and key
    for i = 1:image_size
        index = chaotic_sequence(i);
        decrypted_image(index) = bitxor(image_vector(index), uint8(key * 255));
    end

    % Reshape the decrypted image to original dimensions
    decrypted_image = reshape(decrypted_image, [rows, cols, channels]);
end
```

VIII. Chaotic Encryption Image

```
% Load an image
original_image = imread('image2.jpg'); % Provide the path to your image

% Parameters
alpha = 3.93; % Chaotic map parameter
x_initial = rand; % Initial condition for chaotic map

% Get image dimensions
[rows, cols, channels] = size(original_image);

% Generate key using chaotic map
key = generate_key(alpha, x_initial);

% Encryption
encrypted_image = chaos_encrypt(original_image, rows, cols, channels, alpha, key);

% Decryption
decrypted_image = chaos_decrypt(encrypted_image, rows, cols, channels, alpha, key);

% Display Images
figure;
subplot(1, 3, 1);
imshow(original_image);
title('Original Image');

subplot(1, 3, 2);
imshow(encrypted_image);
title('Encrypted Image');

subplot(1, 3, 3);
imshow(decrypted_image);
title('Decrypted Image');

 % Function to generate chaotic sequence
function chaotic_sequence = generate_chaotic_sequence(alpha, x_initial, image_size)
    chaotic_sequence = zeros(1, image_size);
    x = x_initial;
    for i = 1:image_size
        x = alpha * x * (1 - x);
        chaotic_sequence(i) = x;
    end
    chaotic_sequence = mod(floor(chaotic_sequence * image_size), image_size) + 1;
end

 % Function to generate key using chaotic map
function key = generate_key(alpha, x_initial)
    x = x_initial;
    for i = 1:1000 % Iterate to get a more "random" value
        x = alpha * x * (1 - x);
    end
    key = x; % Use the chaotic value as the key
end
```

```
% Function to encrypt image using chaotic map
function encrypted_image = chaos_encrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols;
    encrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);
    for channel = 1:channels
        channel_data = image(:,:,channel);
        flattened_channel = channel_data(:);
        for i = 1:image_size
            temp = flattened_channel(i);
            flattened_channel(i) = flattened_channel(chaotic_sequence(i));
            flattened_channel(chaotic_sequence(i)) = temp;
        end
        flattened_channel = bitxor(flattened_channel, uint8(key * 255));
        encrypted_image(:,:,channel) = reshape(flattened_channel, [rows, cols]);
    end
end
```

```
% Function to decrypt image using chaotic map
function decrypted_image = chaos_decrypt(image, rows, cols, channels, alpha, key)
    image_size = rows * cols;
    decrypted_image = zeros(rows, cols, channels, 'uint8');
    chaotic_sequence = generate_chaotic_sequence(alpha, key, image_size);
    for channel = 1:channels
        channel_data = image(:,:,channel);
        flattened_channel = channel_data(:);
        flattened_channel = bitxor(flattened_channel, uint8(key * 255));
        for i = image_size:-1:1
            temp = flattened_channel(i);
            flattened_channel(i) = flattened_channel(chaotic_sequence(i));
            flattened_channel(chaotic_sequence(i)) = temp;
        end
        decrypted_image(:,:,channel) = reshape(flattened_channel, [rows, cols]);
    end
end
```

References

1. Spread Spectrum Systems" Dixon ,RC

2. "Communication Systems Engineering" by John G. Proakis and Masoud Salehi

3. "Cryptography.and.Network.Security".Global.Edition.8th.Edition.William. Stallings.

4. "A Survey of Digital Watermarking Techniques and Its Applications" published in Applied Computing and Informatics.

5. Agrawal, E., and J. Jain. "A Review on Various Methods of Cryptography for Cyber Security." Int. J. Recent Innov. Trends Comput. Commun 6.7 (2018): 5

6. Halboos, Estabraq Hussein Jasim, and Abbas M. Albakry. "Hiding text using the least significant bit technique to improve cover image in the steganography system." Bulletin of Electrical Engineering and Informatics 11.6 (2022): 3258-3271.

7. Voleti, Lasya, et al. "A secure image steganography using improved Lsb technique and vigenere cipher algorithm." 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS). IEEE, 2021.

8. Rojali, and Afan Galih Salman. "Website-based PNG image steganography using the modified Vigenere Cipher, least significant bit, and dictionary-based compression methods." AIP Conference Proceedings. Vol. 1867. No. 1. AIP Publishing LLC, 2017.

9. Adhiya, K. P., & Patil, S. A. (2012). Hiding text in audio using LSB based steganography. Information and Knowledge Manage-ment, 2(3), 8-14.

10. Chadha, A., & Satam, N. (2013). An efficient method for image and audio steganography using Least Significant Bit (LSB) substi-tution. International Journal of Computer Applications, 77(13), 37-45

11. Chowdhury, R., Bhattacharyya, D., Bandyopadhyay, S. K., & Kim, T. H. (2016). A view on LSB based audio steganogra-phy. International Journal of Security and Its Applications, 10(2), 51-62.

12. Alvarez, G., & Li, S. (2006). Some Basic Cryptographic Requirements for Chaos-Based Cryptosystems. *International Journal of Bifurcation and Chaos*, 16(8), 2129-2151.

13. Baptista, M. S. (1998). Cryptography with chaos. *Physics Letters A*, 240(1-2), 50-54.

14. Kocarev, L. (2001). Chaos-based cryptography: a brief overview. *IEEE Circuits and Systems Magazine*, 1(3), 6-21.

15. Pareek, N. K., Patidar, V., & Sud, K. K. (2006). Image encryption using chaotic logistic map. *Image and Vision Computing*, 24(9), 926-934.

16. Lian, S., Sun, J., & Wang, Z. (2005). Security analysis of a chaos-based image encryption algorithm. *Physics Letters A*, 351(2-3), 152-157.