EEC269A - Error Correcting Codes I
Project Report

Chenye Yang, Pranav Kharche, Parisa Oftadeh

June 12, 2023

Contents

1 Workload

Table 1: Workload

Function	**Workload**	**Contributor**
Src & Dest	Text string (TXT), Image (PNG), Audio (WAV)	Chenye
Encoder & Decoder	$(7,4)$ Systematic Linear Block (Hamming) Code (n,k) Systematic Cyclic Code	Chenye
Channel	Binary Symmetric Channel (BSC), adjustable error prob p	Chenye
Error Corrector	Syndrome Lookup Table for $(7,4)$ Linear Code Syndrome Lookup Table for (n,k) Cyclic Code, $t=1$ Error Trapping for (n,k) Cyclic Code, adjustable t	Chenye Chenye Pranav
Advanced features	**Create generator matrix for** (n,k) **cyclic code** **Adjustable:** (n,k) **and correctable errors** t	Pranav
Data Analysis	**Statistic Results**	Parisa, Chenye
Presentation	Slides	Parisa
Report	Work together	-

2 Source & Destination

2.1 Source

2.1.1 Text string

The very basic function of the information source is to read a hard-coded text file into a bit stream. In our text file, the following string is stored in ASCII format:

Hello World!
EEC269A Error Correcting Code Demo

In ASCII format, each character is represented by 8 bits, shown in Table 2. Then, after transformation, the bit stream is of size 376 bits:

0 1 0 0 1 0 0 0 0 1 1 0 0 1 0 1 ...

2.1.2 Image

PNG (Portable Network Graphics) is a raster graphics file format that supports lossless data compression. PNG supports a large number of colors (up to 16 million), as well as variable transparency, which makes it useful for images with varying degrees of transparency or opacity. Additionally, because PNG is a lossless format, it preserves all the detail in the original image, which is not the case with lossy formats like JPEG. It is used for the storage and display of images on the internet, and is also used in graphic design and editing applications due to its lossless compression.

PNG files consist of a header followed by a series of data chunks, e.g.:

1. Signature: The first eight bytes of a PNG file always contain the following decimal numbers: 137, 80, 78, 71, 13, 10, 26, 10. This signature indicates that the file is a PNG.

Table 2: ASCII Table example

Character	Hexadecimal	Decimal	Binary
...	...	...	...
A	41	65	0 1 0 0 0 0 0 1
B	42	66	0 1 0 0 0 0 1 0
C	43	67	0 1 0 0 0 0 1 1
D	44	68	0 1 0 0 0 1 0 0
E	45	69	0 1 0 0 0 1 0 1
F	46	70	0 1 0 0 0 1 1 0
G	47	71	0 1 0 0 0 1 1 1
H	48	72	0 1 0 0 1 0 0 0
...	...	...	...

2. Header Chunk (IHDR): The first chunk after the signature is the IHDR chunk, which contains basic information about the image, such as width, height, bit depth, color type, compression method, filter method, and interlace method.

3. Palette Chunk (PLTE): This chunk is optional and only present for color type 3 (indexed color). It contains the color palette for the image.

4. Data Chunks (IDAT): These chunks contain the actual image data, which is formed by pixels. This data is compressed to reduce the size of the file.

5. End Chunk (IEND): This is the final chunk in a PNG file. It does not contain any data and its purpose is to indicate the end of the file.

Each chunk contains three standard fields: 4-byte length, 4-byte type code, 4-byte CRC and various internal fields that depend on the chunk type, shown in Figure 1 [1]. For example, the image file we are using has more than one image data (IDAT) chunks, each of which contains a portion of the image, shown in Table 3.

Ideally, the entire image file should be read into a bit stream and transmitted through the channel. However, if there are uncorrectable errors in the chunks other than the image data (IDAT), the received image will not be able to display. Therefore, we shall only work with the image data (IDAT) chunks for the purpose of visualization of a corrupted image. Also, this trick will not affect the statistical analysis of the system.

The information source is able to only extract the color information from a PNG file and convert it into a bit stream to be passed through the channel. This is done by using the Python library *NumPy*.

The library *NumPy* provides a method to only read out the color information of an image. The shape of the result array is typically *(height, width, channels)*, where:

1. *height* is the number of pixels in the vertical direction (i.e. the number of rows of pixels);

2. *width* is the number of pixels in the horizontal direction (i.e. the number of columns of pixels);

3. *channels* is the number of color channels per pixel. This value depends on what type of image it is:

   - In an RGB image, the three channels correspond to Red, Green, and Blue, respectively. Each channel value usually ranges from 0 to 255, where 0 indicates none of that color is present and 255 indicates that color is fully present.
   - In a grayscale image, there is typically only one channel. The value in this single channel indicates the level of gray, where 0 is black and 255 is white.
   - There are many other color spaces that have different meanings for their channels.

[1] Figure 1 is from webpage PNG file chunk inspector - Project Nayuki

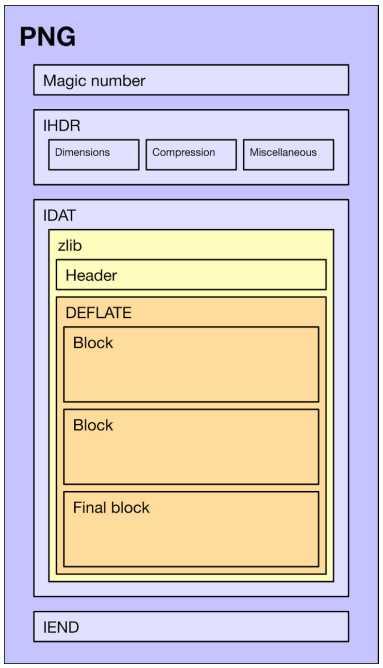


Figure 1: PNG file structure

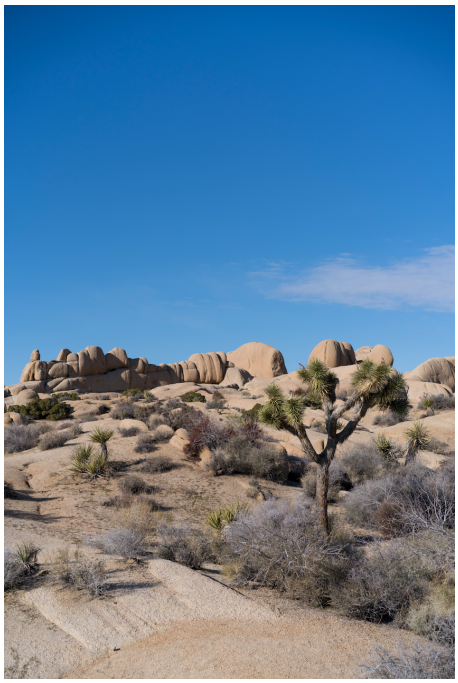


Figure 2: Our testing PNG image

The data type for each channel of each pixel is *uint8*, which is an unsigned integer that takes 8 bits. Then, the array is flattened into a bit stream by converting each channel of each pixel into an 8-bits binary number and appending them together.

For example, as for our testing image[2], shown in Figure 2, the shape of the result array is *(1280, 854, 3)*, which means that the image has 1280 rows of pixels, 854 columns of pixels, and 3 color channels per pixel (RGB). Then, the array is flattened into a bit stream of 26,234,880 bits.

2.1.3 Audio

WAV (Waveform Audio File Format) is a digital audio standard for storing audio bitstream on PCs. WAV is an application of the Resource Interchange File Format (RIFF) method for storing data in chunks, and it is primarily used on Windows systems. WAV files are typically used for raw and uncompressed audio, though they can also contain compressed audio. A WAV file is divided into several sections or chunks, shown in Figure 3 [3]. Each chunk serves a different purpose and holds different types of data. The basic structure of a WAV file includes the following chunks:

1. RIFF Chunk: The RIFF chunk is the first chunk in a WAV file and identifies the file as a WAV file. It includes a header with the "RIFF" identifier and an integer indicating the remaining length of the entire file.

2. Format Chunk: Also known as the "fmt " chunk (with a space after 'fmt'), this contains important information about the audio data. This includes the audio format (e.g., PCM), the number of channels (mono, stereo, etc.), the sample rate, the byte rate, the block alignment, and the bit depth (bits per sample).

3. Data Chunk: This is where the actual audio data is stored. The "data" header is followed by an integer representing the length of the data, and then by the raw audio data itself.

[2]The image file used in this project is photographed by *Chenye Yang*. The material is free from any copyright restrictions and can be used without any potential legal implications.

[3]Figure 3 is from webpage WAV Files: File Structure, Case Analysis and PCM Explained

Table 3: Our PNG file structure

Start offset	Chunk outside
0	Special: File signature; Length: 8 bytes
8	Data length: 13 bytes; Type: IHDR; Name: Image header; CRC-32: CB3954EC
33	Data length: 1 bytes; Type: sRGB; Name: Standard RGB color space; CRC-32: AECE1CE9
46	Data length: 976 bytes; Type: eXIf; Name: Exchangeable Image File (Exif) Profile; CRC-32: 47FCFA4D
1,034	Data length: 9 bytes; Type: pHYs; Name: Physical pixel dimensions; CRC-32: 5024E7F8
1,055	Data length: 4 514 bytes; Type: iTXt; Name: International textual data; CRC-32: C9C76B16
5,581	Data length: 16 384 bytes; Type: IDAT; Name: Image data; CRC-32: 8462CABD
21,977	Data length: 16 384 bytes; Type: IDAT; Name: Image data; CRC-32: 2C9D007C
...	...
1,727,161	Data length: 12 170 bytes; Type: IDAT; Name: Image data; CRC-32: 68067F52
1,739,343	Data length: 0 bytes; Type: IEND; Name: Image trailer; CRC-32: AE426082

Similar to the image file, the information source is able to only extract the audio Data Chunk from a WAV file and convert it into a bit stream to be passed through the channel. This trick ensures both the visualization of the results and the statistical analysis of the system. The extraction is done by using the Python library *soundfile*.

The library *soundfile* provides a method to only read out the audio data from a WAV file. The result contains two parts:

1. audio array: This is a *NumPy* array that contains the audio data from the file. The shape of the array depends on the number of channels in the audio file. If the audio is mono, the array will be one-dimensional. If the audio is stereo, the array will be two-dimensional, with one sub-array for each channel. The values in the array represent the amplitude of the audio signal at each sample point, and are of the data type specified.

2. sample rate: This is an integer that represents the number of samples per second in the audio file, measured in Hertz (Hz). Common sample rates include 44100 Hz (standard for audio CDs), 48000 Hz (standard for video production and DVDs), and 96000 Hz (used in high-definition formats).

Then, the *int16* array is "viewed" as *uint8* array by *numpy.view()* and flattened into a bit stream by converting each sample into an 8-bits binary number and appending them together. Note that the *numpy.view()* operation simply reinterprets the binary data, and does not convert or scale the data, which originally could be negative or positive.

For example, as for our testing audio file[4], shown in Figure 4 and Figure 5, the shape of the result audio array is *(268237, 2)* and the sample rate is 8000 Hz. It means that the 33.5-second audio file has 268237 samples per channel, and there are two channels (stereo). Then, the array is flattened into a bit stream of 8,583,584 bits.

[4]The audio file used in this project is free downloaded from file-examples.com. The material is free from any copyright restrictions and can be used without any potential legal implications.

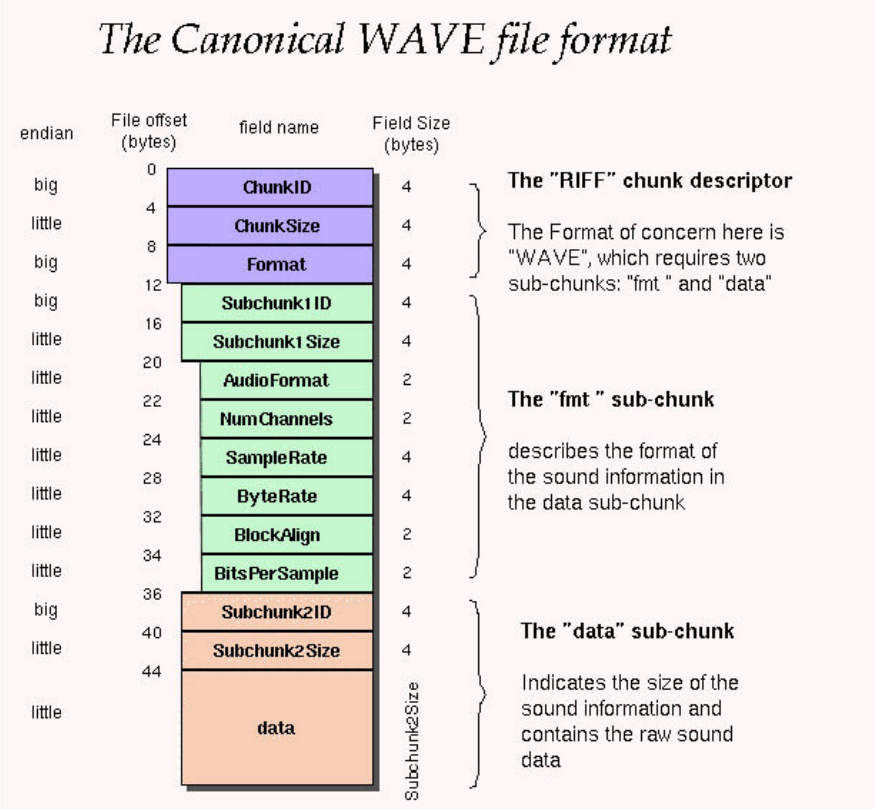


Figure 3: Canonical WAV file structure

2.2 Destination

At the destination, the bit stream is re-construed into the original format, and stored. It is inevitable that some metadata is lost when processed by the system, since they are not transmitted through the channel. For example, the re-constructed image file will not have the camera and lens information. However, this is the trade-off for a better visualization, and will not affect the statistical analysis of the system.

3 Channel

3.1 Binary Symmetric Channel (BSC)

The Binary Symmetric Channel (BSC) is a fundamental concept in information theory and telecommunications, specifically in the field of error detection and correction. It is a model used to represent a communication channel, where the information is transmitted in the form of binary digits, or bits: 0s and 1s.

The "symmetric" aspect of the BSC refers to the fact that it has the same probability of an error occurring whether the transmitted bit is a 0 or a 1. For instance, if the error probability is 0.01, then 1% of 0s are received as 1s, and 1% of 1s are received as 0s.

A BSC is characterized by two parameters: the input bit and the crossover probability / error probability, shown in Figure 6. The input bit is either 0 or 1, which represents the binary information being transmitted. The crossover probability, often denoted as p, represents the likelihood of the transmitted bit being received incorrectly.

Although the model is simple, the BSC is a building block for understanding more complex communication systems. By studying the BSC, researchers can develop algorithms and systems that minimize errors in binary data transmission, improving the reliability of digital communications.

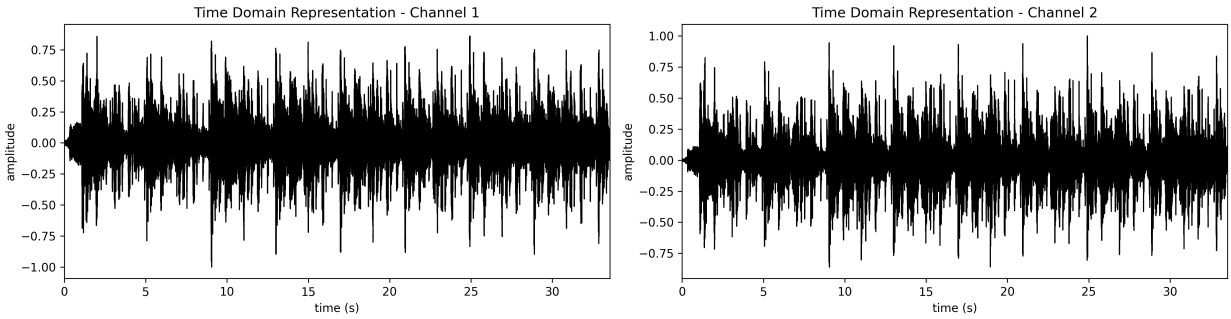


Figure 4: Time domain waveform of our testing audio file

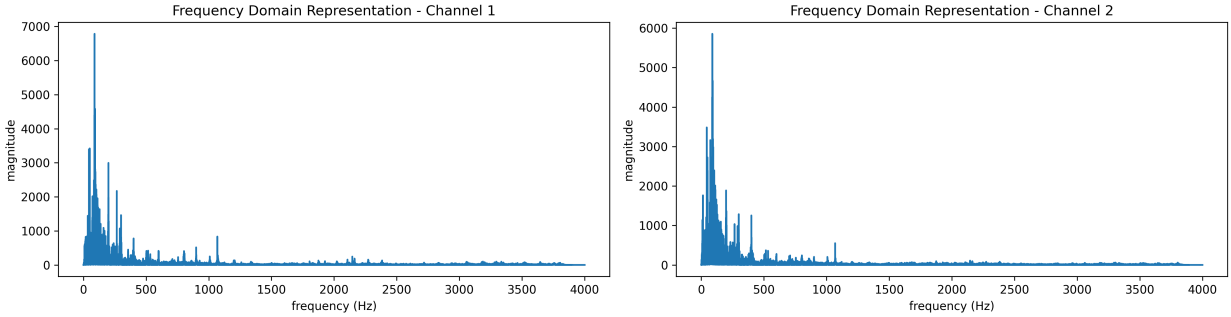


Figure 5: Frequency domain waveform of our testing audio file

All the results are done with error probability $p = 0.01$, even though this p is a configurable value in $[0, 1]$ by users.

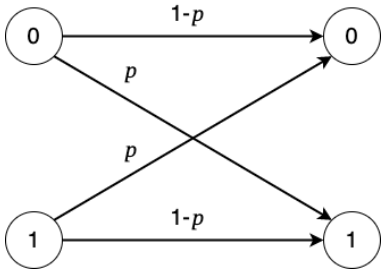


Figure 6: Binary Symmetric Channel (BSC)

4 (7,4) Systematic Linear Block (Hamming) Code

The (7,4) Systematic Linear Block (Hamming) Code, often referred to simply as the Hamming (7,4) Code, is a fundamental concept in the realm of error detection and correction, and plays a crucial role in the field of digital communications and information theory. This coding scheme was introduced by Richard Hamming in the late 1940s to address the problem of error detection and correction in data transmission over noisy channels.

The Hamming (7,4) Code is a block code that maps 4-bit messages to 7-bit codewords. It gets its name from the fact that it takes in 4 data bits and outputs 7 bits (including 3 parity bits). The parity bits are extra bits added to the data to enable the detection and correction of errors that might occur during transmission.

A key characteristic of the Hamming (7,4) Code is its systematic form. This means that the bits of the

original message appear unaltered in the encoded output, making it easy to extract the original data from the received message, even if errors are detected.

The Hamming (7,4) Code is designed to detect and correct single-bit errors, meaning it can identify and fix any situation where only one bit in the 7-bit codeword has been flipped due to noise or interference in the communication channel. This makes it a powerful tool in improving the reliability and robustness of data communication systems.

4.1 Encoder

The encoding process in a Hamming (7,4) code involves the multiplication of a 4-bit message vector with a generator matrix. The generator matrix for a (7,4) Hamming Code, in its systematic form, is given as:

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

If we denote the message vector as

$$u = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix},$$

then the encoded codeword v is computed by:

$$v = u \cdot G.$$

This multiplication produces a 7-bit codeword, which is then ready for transmission.

4.2 Syndrome Decoder

First, the parity-check matrix H will be created from the generator matrix G by transposing the 4 by 3 parity-check bits and filling it with an 3 by 3 identity matrix:

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

The decoding process of the (7,4) Hamming code involves utilizing a syndrome lookup table, which is generated from the matrix multiplication:

$$\text{syndrome table} = e \cdot H^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot H^T = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix},$$

where the 7 by 7 identity matrix with an extra row of zeros on the top, e, represents all the possible error patterns which are correctable by a syndrome look-up table for this specific (7,4) hamming code with generator matrix G. Each row of e represents an error pattern identified by the same row of syndrome table.

When a codeword is received, the syndrome s is computed from the received bits and is used to identify the error pattern:

$$s = r \cdot H^T,$$

where the $r = [r_1\ r_2\ \ldots\ r_7]$ represents the received codeword.

Each syndrome corresponds to a potential error pattern in the look-up table. The estimated transmitted codeword can be calculated by modulo 2 addiction of error pattern and received codeword:

$$\tilde{v} = r + e[i:] \mod 2,$$

where the $e[i:]$ represents the i-th row of error patterns e when the calculated syndrome s matches the i-th row of the syndrome table.

Since the code we are working on is systematic code, messages can be directly extracted from the last 4 bits of codewords.

4.2.1 Results: TXT, PNG, WAV

Shown in Table 4 Figure 7

Table 4: Text string encoded with (7, 4) linear Hamming code

Original	**Without correction**	**Corrected**
Hello World! EEC269A Error Correcting Code Demo	Hello Wopld! EEC269A Arror Correcting Kode Demo	Hello World! EEC269A Error Correcting Code Demo

(a) Original (b) Without correction (c) Corrected

Figure 7: Image encoded with (7, 4) linear Hamming code (entire)

5 (n, k) Systematic Cyclic Code

The (n, k) Systematic Cyclic Code is a powerful and widely used error correction code (ECC) in the field of digital communications and information theory. The notation (n, k) denotes that each block of the code consists of 'n' bits, where 'k' bits are the actual information or data, and the remaining '$n - k$' bits are the parity or check bits used for error detection and correction.

The term "systematic" implies that the data bits remain in their original form in the encoded output, while the parity bits are appended to form the full code word. This is in contrast to non-systematic coding schemes, where the data and parity bits are intermixed.

(a) Original (b) Without correction (c) Corrected

Figure 8: Image encoded with (7, 4) linear Hamming code (details)

"Cyclic" codes, meanwhile, have a key property where a cyclic shift of a code word results in another valid code word. This cyclic nature leads to efficient implementation, especially in hardware, as simple shift registers can be used.

When the linear error-correcting codes is able to correct only single-bit errors in codewords, they are also called "Hamming" code. They accomplish this by introducing additional parity bits that allow the identification of single-bit errors.

In the context of (n, k) Systematic Cyclic Codes, the codes are designed such that any t-bit error can be detected and corrected, making them a reliable choice for environments where such errors are common and the integrity of information is paramount.

5.1 Adjustable (n, k) for correcting t errors

Allowing for a customizable n and k required many complex steps but we will work from the base all the way up to the overall algorithm.

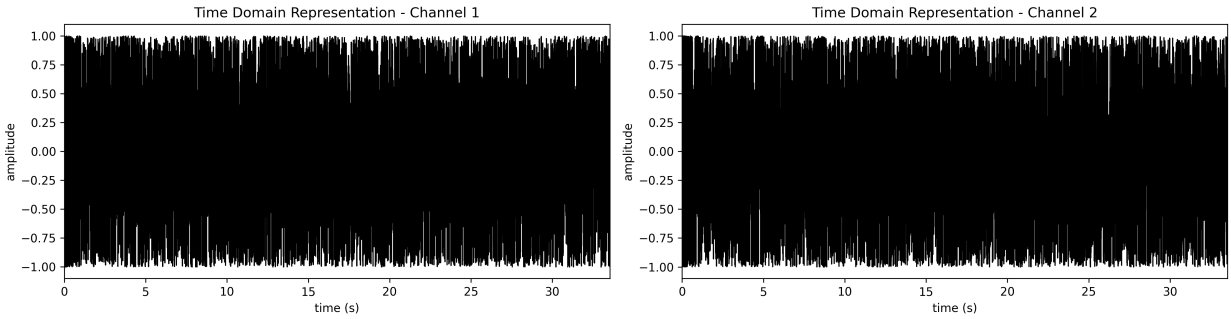
5.1.1 Polynomial Representation

The first step is to represent polynomials in an efficient manner. While the rest of the code uses a numpy array to represent everything, this section exclusively represents polynomials using integers, with each bit representing the coefficients. For example, the number 11 in binary is 0b1011. This means that the code will interpret 11 as the polynomial $x^3 + x + 1$.

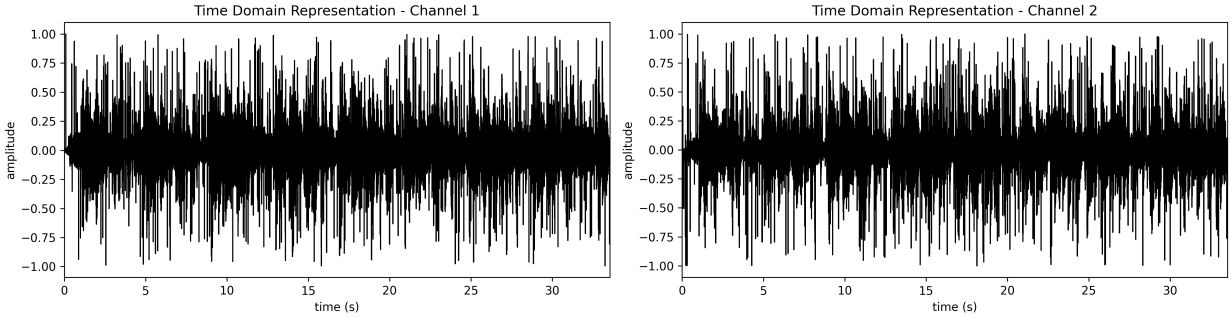
5.1.2 Polynomial Division

Next we need to be able to divide polynomials. This was done with an algorithm similar to how you would divide by hand:

1. Initialize the remainder to the dividend
2. Left shift divisor so that top bits align with remainder
3. Subtract the result of step 2 from the remainder.
4. Set the bit in the quotient according to the number of bits shifted
5. Repeat 2-4 until the remainder is less than the divisor.



(a) Without correction



(b) Corrected

Figure 9: Audio encoded with (7, 4) linear Hamming code

5.1.3 Finding a Generator Polynomial

The search for a generator polynomial relies on two mathematical facts. Firstly, for any (n, k) cyclic code, the generator polynomial $g(X)$ fits the equation $X^n + 1 = g(X) * h(X)$. This means that the generator polynomial must evenly divide $X^n + 1$. Additionally, the generator polynomial must have order $n - k$. Using these two facts as constraints, we can brute force search through possible generators until a viable candidate is found.

5.1.4 Building the Generator Matrix

Normally, a generator matrix for cyclic codes can be easily built using cyclic shifts. However, building it in a systematic form requires extra care. Any block code can be converted to systematic form using row operations. Below is an example of the generation of a $(7, 4)$ code.

Initial:

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

Right shift:

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

Correct systematic:

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

Right shift:

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

Correct systematic:

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

Right shift:

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

Done

5.1.5 Checking the Correctable Errors

The number of correctable errors for a given code is based on the minimum distance between the codewords. The minimum distance of a code is equal to the minimum weight of all the codewords. Finding the minimum wait codeword can be difficult since there are 2^k codewords. However, since this is a systematic code, every codeword will at least have a weight equal to the information encoded in it. This means we can guide our search by iterating through minimum weight information words. It will also stop the search preemptively if the weight of the information words becomes larger than an already determined minimum. With a minimum distance of d, the number correctable errors is $(d-1)//2$

5.1.6 Overall Algorithm

1. Find a suitable generator polynomial given n and k
2. Build the generator matrix
3. Check how many errors the code can correct
4. If this does not meet the target correctable errors, repeat.

This algorithm can also be modified to search for all possible generators and find the best one.

5.2 Encoder

The encoding process in a (n, k) Systematic Cyclic Code involves the multiplication of a k-bit message vector with a generator matrix. The generator matrix for a (n, k) Cyclic Code, in its systematic form, is given as:

$$G = \begin{bmatrix} P & I_k \end{bmatrix}$$

where I_k represents a k by k identity matrix, and P represents a k by $(n-k)$ matrix for parity-check bits.

Following is an example of the generator matrix for a $(15, 11)$ Cyclic Hamming Code ($t = 1$), which is generated with procedures in Section 5.1:

$$G = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & 1 & 1 & 0 & 0 & \cdots & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}.$$

If we denote the message vector as

$$u = \begin{bmatrix} u_1 & u_2 & \dots & u_k \end{bmatrix},$$

then the encoded codeword v is computed by:

$$v = u \cdot G.$$

This multiplication produces a n-bit codeword, which is then ready for transmission.

5.3 Syndrome Decoder

When there are only 1-bit errors in the received codewords, the error bit can be corrected by using the syndrome lookup table with the following general strategy.

First, the parity-check matrix H will be created from the generator matrix G by transposing the k by $(n-k)$ parity-check bits and filling it with an $(n-k)$ by $(n-k)$ identity matrix:

$$H = \begin{bmatrix} I_{n-k} & P^T \end{bmatrix}.$$

The decoding process of the (n, k) cyclic code for $t = 1$ error involves utilizing a syndrome lookup table, which is generated from the matrix multiplication:

$$\text{syndrome table} = e \cdot H^T = \begin{bmatrix} 0_{1\times n} \\ I_n \end{bmatrix} \cdot H^T.$$

where the n by n identity matrix with an extra row of zeros on the top, e, represents all the possible error patterns which are correctable by a syndrome look-up table for this specific (n, k) cyclic code with generator matrix G. Each row of e represents an error pattern identified by the same row of syndrome table.

Following is an example of the parity-check matrix H and syndrome look-up table for a (15, 11) Cyclic Hamming Code ($t = 1$):

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & \dots & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & \dots & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \dots & 1 & 1 \end{bmatrix},$$

$$\text{syndrome table} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}. \tag{1}$$

When a codeword is received, the syndrome s is computed from the received bits and is used to identify the error pattern:

$$s = r \cdot H^T,$$

where the $r = [r_1\ r_2\ \dots\ r_n]$ represents the received codeword.

Each syndrome corresponds to a potential error pattern in the look-up table. The estimated transmitted codeword can be calculated by modulo 2 addiction of error pattern and received codeword:

$$\tilde{v} = r + e[i:] \mod 2,$$

where the $e[i:]$ represents the i-th row of error patterns e when the calculated syndrome s matches the i-th row of the syndrome table.

Since the code we are working on is systematic code, messages can be directly extracted from the last k bits of codewords.

5.3.1 Results: TXT, PNG, WAV

Table 5: Text string encoded with (15, 11) cyclic code with 1 correctable error (syndrome decoder)

Original	**Without correction**	**Corrected**
Hello World! EEC269A Error Correcting Code Demo	Hu,lo World! EEC269A Error Correctifg Code Demo	Hello World! EEC269A Error Correcting Code Demo

(a) Original (b) Without correction (c) Corrected

Figure 10: Image encoded with (15, 11) cyclic code with 1 correctable error (syndrome decoder) (entire)

5.4 Error Trapping Decoder

When there are 1-bit errors or more than one bit errors in the received codewords, the error bits can be corrected by error trapping decoder with the following general strategy.

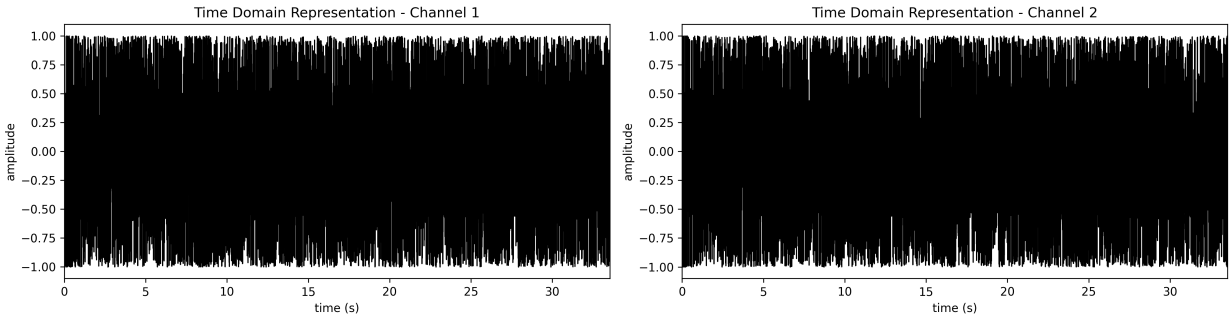
This decoder starts off the same as the syndrome LUT decoder. It uses the parity matrix to calculate the syndrome of the codeword. If the syndrome is zero, then the word has no errors and is passed to the output.

That is where the similarities end. To correct any errors, the decoder uses the syndrome itself as the error pattern. If the syndrome has weight less than or equal to the amount of errors the code can correct, then the syndrome can be added to the top portion of the codeword to remove any errors.

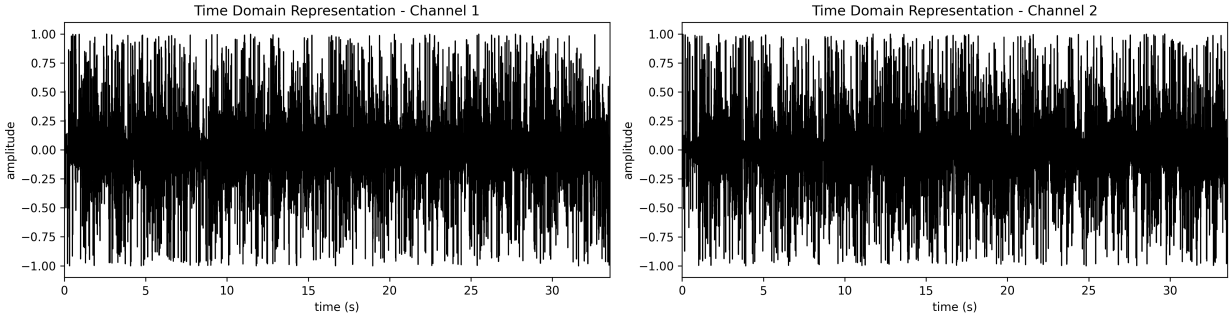
If the weight of the syndrome is too high, then the decoder performs a cyclic shift on the codeword and tries again. Eventually, the decoder will reach a correctable state. After fixing the error, the cyclic shifts are undone and it is passed to the output.

(a) Original (b) Without correction (c) Corrected

Figure 11: Image encoded with (15, 11) cyclic code with 1 correctable error (syndrome decoder) (details)



(a) Without correction



(b) Corrected

Figure 12: Audio encoded with (15, 11) cyclic code with 1 correctable error (syndrome decoder)

If the decoder shifts through the whole codeword without finding a correctable syndrome, the codeword is deemed uncorrectable. For our case, we decided to simply ignore the error and pass the erroneous codeword to the output.

Here is an example:

Let's say for a given $(15, 7)$ code with $t = 2$, the received codeword is:

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

This has a resulting syndrome of:

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

which has a weight greater than 2, meaning that the error cannot be corrected.

However, after a cyclic shift, the codeword and syndrome become:

$$[1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1]$$
$$\implies [0\ 0\ 0\ 0\ 0\ 0\ 0\ 1]$$

This now a correctable pattern and the decoder repair the codeword as follows:

Original:

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

Shift 1:

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

Correct:

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

Shift -1:

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

Table 6: Text string encoded with (15, 5) cyclic code with 3 correctable error (error trapping decoder)

Original	**Without correction**	**Corrected**
Hello World! EEC269A Error Correcting Code Demo	Hello World! EEC26^{A} Error Co‘recting C”de Demk	Hello World! EEC269A Error Correcting Code Demo

(a) Original (b) Without correction (c) Corrected

Figure 13: Image encoded with (15, 5) cyclic code with 3 correctable error (error trapping decoder) (details)

5.4.1 Results: TXT, PNG, WAV

6 Statistics Analysis

In this section, we discuss the statistical results of all the 3 coding and decoding approaches that we had in this project. The results for (7, 4) Hamming code are brought in Table 7. Accordingly, the bit error rate for all file types (including text, image, and audio file) is almost the same which is equal to the error probability of the binary symmetric channel (BSC with $p = 0.01$). The linear hamming code is capable of correcting up to 1 error per codeword. Consequently, the bit error rate is decreased to a certain amount after correction. However, some errors remain uncorrected.

We show the results for (15, 5) cyclic code with error trapping decoder and (15, 11) cyclic code with syndrome decoder in Table 9 and Table 8, respectively. These 2 approaches compete with each other in a trade-off between accuracy and transmission rate concepts. Hence, we decide to use the coding technique according to the problem requirements and there is no perfect coding that can satisfy the mentioned metrics at the maximum possible value simultaneously. The (15,11) Cyclic code can correct up to 1 error per codeword, while the (15,5) Cyclic code corrects up to 3 errors. However, the total codewords generated by the (15, 11) CRC is 35, and this value is equal to 75 codewords (more than twice) for the (15, 5) CRC. Consequently, we need more time or bandwidth to transfer the same file with the second technique. It is worth mentioning that the final bit error rates (after correction) of these 2 models are equal to 0.0000 for text files, which indicates that the (15, 11) outperforms the (15, 5) due to its lower bandwidth/time consumption. But for larger data types, like audio or image files, the after-correction error rate of (15,11) is less than (15,5), which promises a more reliable communication.

(a) Original (b) Without correction (c) Corrected

Figure 14: Image encoded with (15, 5) cyclic code with 3 correctable error (error trapping decoder) (details)

7 Other Challenges

7.1 Why PNG and WAV

The primary difference between file formats such as PNG and JPEG, or WAV and MP3, is the way the data is compressed. More specifically, the difference is in whether the compression is lossless or lossy.

PNG vs JPEG:

PNG uses a lossless compression algorithm, meaning that when the image data is compressed and then decompressed, there is no loss of information. The image remains exactly the same as the original.

On the other hand, JPEG uses a lossy compression algorithm. This means that some data is lost when the image is compressed and then decompressed. JPEG's algorithm reduces file size by discarding some of the less important data or information that's less perceivable to the human eye. This results in smaller file sizes which is beneficial for web usage or storing many photos where high fidelity isn't the priority. However, the image quality decreases with the level of compression, potentially leading to noticeable artifacts or blurring.

WAV vs MP3:

Similarly, WAV (Waveform Audio File Format) and MP3 (MPEG-1 Audio Layer III) are both audio file formats that differ in their compression methods.

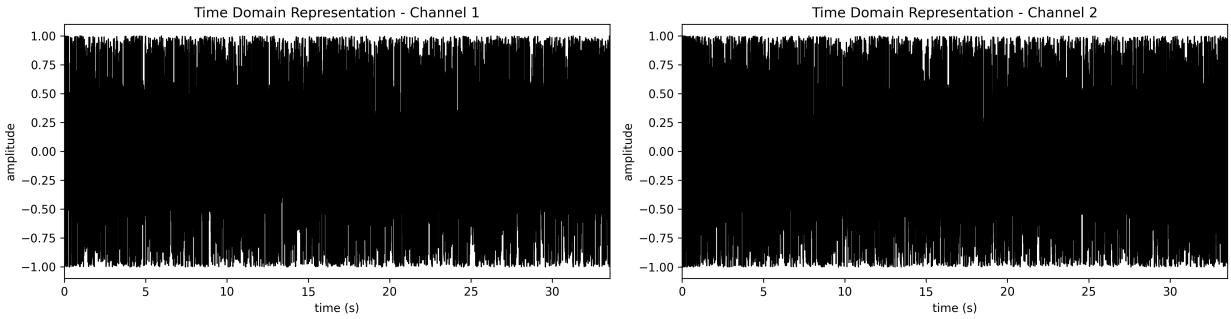
WAV files are a raw audio format created by Microsoft and IBM. They are lossless, uncompressed, and retain full quality of the audio data. This results in high-quality audio but also significantly larger file sizes. It's commonly used in professional settings where high audio quality is a must, like music production or broadcasting.

MP3, however, uses a form of lossy data compression to encode data using inexact approximations, significantly reducing the file size. It was designed to greatly reduce the amount of data required to represent audio, yet still sound like a faithful reproduction of the original uncompressed audio to most listeners. This has made it a common format for consumer audio streaming and storage, but the lossy nature means that some audio fidelity is sacrificed for smaller file size.

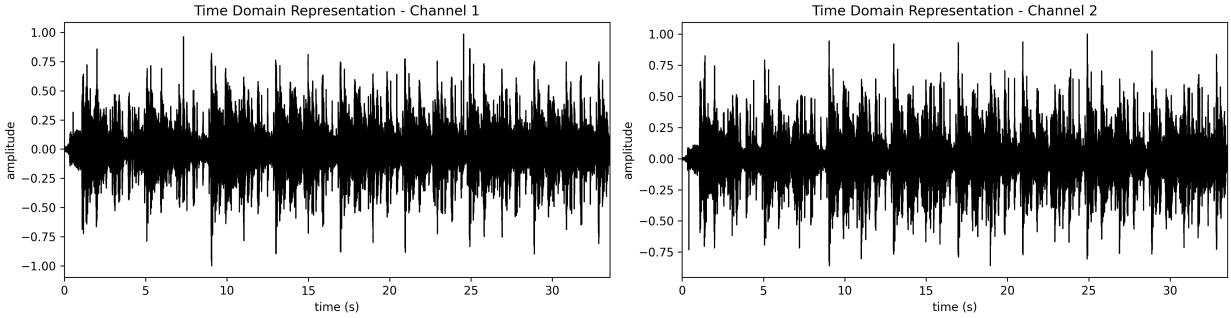
In summary, the choice between these formats depends on your needs. If retaining the original quality of the image or sound is of utmost importance, then lossless formats like PNG and WAV are the way to go. However, if you need to save storage space or bandwidth and can afford some loss in quality, then lossy formats like JPEG and MP3 would be more suitable.

7.2 Why only extract DATA chunk

When we talk about chunks in the context of PNG and WAV files, we're referring to sections of data within these files that have different purposes. In both file types, each chunk begins with a header that identifies its type and carries other relevant information such as the chunk's size. The way these chunks are handled



(a) Without correction



(b) Corrected

Figure 15: Audio encoded with (15, 5) cyclic code with 3 correctable error (error trapping decoder)

contributes significantly to how these file types work. But what happens if these chunks get messed up? Let's take a closer look.

PNG:

In a PNG file, the chunks store different types of data such as the image's metadata, palette data, and actual image data (in the IDAT chunk), among other things.

If a chunk other than the IDAT chunk is messed up or corrupted, the PNG viewer or editor will still attempt to display the image data. However, depending on the severity and location of the corruption, different issues may arise.

For example, if the IHDR chunk (header chunk containing crucial details like width, height, bit depth, etc.) is corrupted, the image may not display correctly or at all, because these fundamental details are required to interpret the image data correctly. A corrupted PLTE chunk (palette data) might result in wrong colors being displayed.

If ancillary chunks like tEXt (textual information), pHYs (physical pixel dimensions), or tIME (last modification time) are corrupted, it's less severe since these chunks do not directly affect the display of the image, but it may cause incorrect metadata to be displayed or used.

WAV:

In a WAV file, data is also organized into chunks. The two primary chunks are the 'fmt ' chunk, containing format information like sample rate, bit depth, and number of channels, and the 'data' chunk, which contains the actual audio data.

Corruption or misinterpretation of the 'fmt ' chunk can result in the audio file being unplayable because the player wouldn't know how to correctly interpret the raw audio data. The audio could also be played back at the wrong speed, or with incorrect bit depth or number of channels, causing significant distortions or alterations to the sound.

Other optional chunks in a WAV file, such as 'LIST' (metadata), 'fact' (additional format information), or 'cue ' (cue points), might not prevent the audio from being played if they get messed up, but they could lead to loss of additional information or features associated with the audio file.

In both cases, it's essential to note that how severely the corruption of a chunk impacts the file can also

depend on the software used to view or play the file, as different software may have different ways of handling such errors.

7.3 Speed up program

Loops in programming languages like Python can be slow, especially when iterating through large matrices. There are several ways we can speed up a for loop when performing matrix operations:

1. **Use NumPy library:** NumPy is a library in Python specifically designed to handle numerical computations much faster than Python's built-in methods, mainly when dealing with matrices or arrays. NumPy performs operations in compiled C code under the hood, making it much more efficient. Suppose you have a matrix A and you want to add a vector v to each row. In a for loop, you might write:

   ```
   for i in range(A.shape[0]):
       A[i, :] += v
   ```

   But with NumPy, you can write:

   ```
   A += v
   ```

   This will perform the addition in a vectorized manner, which is much faster than using a for loop.

2. **Use Broadcasting:** Broadcasting is a powerful mechanism that allows NumPy to work with arrays of different shapes when performing arithmetic operations. You can often use broadcasting to eliminate the need for loops entirely.

3. **Use Vectorized Operations:** Instead of using loops, you can often rephrase your problem in a way that allows you to use vectorized operations. These are operations that are performed on whole arrays at once, which can be much faster than looping through individual elements.

4. **Parallelize your code:** If your loop iterations are independent (i.e., the result of one iteration doesn't depend on the result of any other), you might be able to speed up your code significantly by running multiple iterations in parallel. Libraries like *joblib* and *multiprocessing* can help with this in Python.

Table 7: Statistic results of (7,4) linear Hamming code

File Type	**Metric**	**Before Correction**	**After Correction**
TXT	Total codewords	94	-
	Correct codewords	88	-
	Codeword error rate	0.063830	-
	1 error codewords	6	-
	Uncorrectable codewords	0	-
	Number of correct bits	372	376
	Number of incorrect bits	4	0
	Bit error rate	0.010638	0.000000
PNG	Total codewords	6558720	-
	Correct codewords	6113686	-
	Codeword error rate	0.067854	-
	1 error codewords	431981	-
	Uncorrectable codewords	13053	-
	Number of correct bits	25973102	26212419
	Number of incorrect bits	261778	22461
	Bit error rate	0.009978	0.000856
WAV	Total codewords	2145896	-
	Correct codewords	2000385	-
	Codeword error rate	0.067809	-
	1 error codewords	141137	-
	Uncorrectable codewords	4374	-
	Number of correct bits	8497569	8576040
	Number of incorrect bits	86015	7544
	Bit error rate	0.010021	0.000879

Table 8: Statistic results of (15, 11) cyclic code with 1 correctable error (syndrome decoder)

File Type	**Metric**	**Before Correction**	**After Correction**
TXT	Total codewords	35	-
	Correct codewords	29	-
	Codeword error rate	0.171429	-
	1 error codewords	6	-
	Uncorrectable codewords	0	-
	Number of correct bits	372	376
	Number of incorrect bits	4	0
	Bit error rate	0.010638	0.000000
PNG	Total codewords	2384990	-
	Correct codewords	2051296	-
	Codeword error rate	0.139914	-
	1 error codewords	310904	-
	Uncorrectable codewords	22790	-
	Number of correct bits	25972541	26184088
	Number of incorrect bits	262339	50792
	Bit error rate	0.010000	0.001936
WAV	Total codewords	780326	-
	Correct codewords	671482	-
	Codeword error rate	0.139485	-
	1 error codewords	101430	-
	Uncorrectable codewords	7414	-
	Number of correct bits	8497796	8567137
	Number of incorrect bits	85788	16447
	Bit error rate	0.009994	0.001916

Table 9: Statistic results of (15, 5) cyclic code with 3 correctable error (error trapping decoder)

File Type	Metric	Before Correction	After Correction
TXT	Total codewords	76	-
	Correct codewords	64	-
	Codeword error rate	0.157895	-
	1 error codewords	11	-
	2 error codewords	1	-
	3 error codewords	0	-
	Uncorrectable codewords	0	-
	Number of correct bits	372	376
	Number of incorrect bits	4	0
	Bit error rate	0.010638	0.000000
PNG	Total codewords	5246976	-
	Correct codewords	4514112	-
	Codeword error rate	0.139674	-
	1 error codewords	682449	-
	2 error codewords	48298	-
	3 error codewords	2051	-
	Uncorrectable codewords	66	-
	Number of correct bits	25972642	26234721
	Number of incorrect bits	262238	159
	Bit error rate	0.009996	0.000006
WAV	Total codewords	1716717	-
	Correct codewords	1476183	-
	Codeword error rate	0.140113	-
	1 error codewords	223879	-
	2 error codewords	15896	-
	3 error codewords	731	-
	Uncorrectable codewords	28	-
	Number of correct bits	8497463	8583525
	Number of incorrect bits	86121	59
	Bit error rate	0.010033	0.000007