EE323 Digital Signal Processing
Assignment - I Report
Secure & Intelligent Speech Communication with Motion-Sensor Integration

Team:

Name	*Roll Number*
Kshitij Agarwal	***23110174***
Patel Krish Badal	***23110235***
Hasan Ali	***23110133***

Objectives

1. Develop a Simulink-based mobile application capable of recording audio input from a smartphone's microphone and acquiring real-time motion sensor data (gyroscope and accelerometer).
2. Implement frequency scrambling encryption using a key-based permutation algorithm to make the recorded audio uninterpretable without the correct decryption key.
3. Enable wireless transmission of the encrypted audio and synchronized motion data from Phone A to Phone B via Bluetooth or TCP/IP protocol.
4. Design a synchronized data logging mechanism to ensure that motion sensor readings are accurately time-aligned with the corresponding audio signal during recording.
5. Develop a receiver-side interface that decrypts the audio (using the key) and displays or logs the received motion data graphically for further analysis.

Implementation

The design flow for the Simulink-based voice encryption and transmission system is centered around **frequency scrambling using key-based permutation**. The process involves converting the speech signal into the frequency domain, permuting its sub-bands based on a secret key, and reconstructing a scrambled version that is uninterpretable without decryption. The overall design ensures that both audio and motion sensor data are acquired, encrypted, synchronized, and transmitted efficiently between two mobile devices.

1. Audio Acquisition and Pre-Processing

The system begins with **audio signal capture** from the smartphone's microphone. The sampled audio is pre-processed to ensure proper normalization and framing for frequency-domain analysis. During this stage, **motion sensor data** (gyroscope and accelerometer readings) are also recorded with precise timestamps to maintain synchronization with the audio stream.

2. Frequency Transformation (FFT)

The time-domain audio signal is converted into the **frequency domain** using the **Fast Fourier Transform (FFT)**.This step splits the voice signal into different frequency parts, which are then rearranged for scrambling.

3. Key-Based Permutation Generation

We chose a key which will be used in the frequency scrambling process which will also be used for the unscrambling of the encrypted audio signal. If the key entered by the receiver is other than the key used then the signal output will be noise rather than the original signal.

4. Frequency Scrambling

Permutation is applied to reorder the frequency spectrum obtained from the FFT. This process **destroys the natural order of spectral components**, removing speech comprehensibility while maintaining the signal's overall structure.the scrambled frequency domain representation is then passed through the **Inverse Fast Fourier Transform (IFFT)** to convert it back into a time-domain waveform, resulting in a distorted and uninterpretable audio signal.

5. Encryption Transmission and Motion Data Integration

The **scrambled audio** is packaged with the **time-synchronized motion sensor data** (accelerometer and gyroscope readings) and transmitted via **TCP-IP protocol** to the receiver device.
This integrated transmission ensures that both data streams can later be correlated for motion–audio analysis.

6. Decryption at Receiver

At the receiver, the same key regenerates the permutation map. The inverse permutation is applied to the FFT of the scrambled audio, and IFFT restores the original clear voice. The received motion data is then displayed or logged for analysis.Simultaneous sampling and timestamping ensure that **motion and audio data remain perfectly synchronized**.

Overall Design Pipeline:

Record audio & motion data → Apply FFT → Generate key-based permutation → Scramble frequency components → Apply IFFT → Transmit audio & motion data → Receiver FFT → Generate same permutation → Apply inverse permutation & IFFT → Recover audio & display motion data

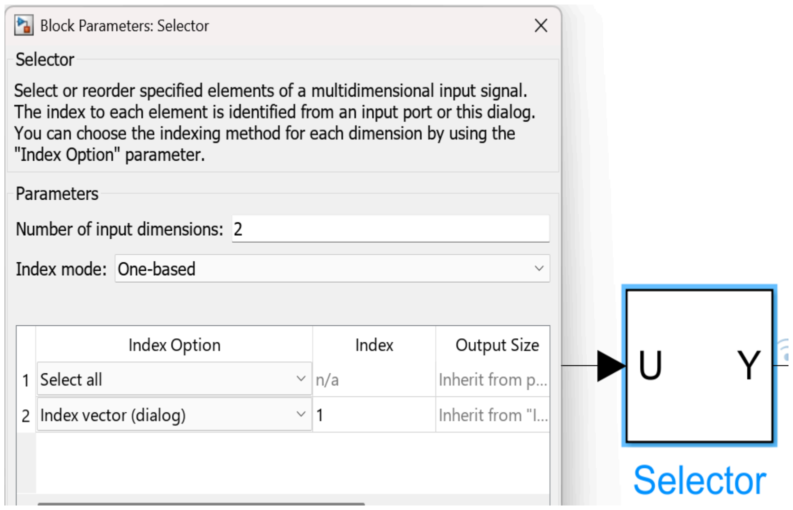
Block by Block Overview:

1. **Android Audio capture:**

Audio Capture

This **Audio Capture block** captures the audio signals from the device microphone. We have set the sampling at standard 44.1kHz frequency with the frame size of 4410. The Frame rate is 0.1 sec. The output given is in format of Nx2 Matrix in int16 data type.

2. **Selector Block:**



Block Parameters: Selector

Selector

Select or reorder specified elements of a multidimensional input signal. The index to each element is identified from an input port or this dialog. You can choose the indexing method for each dimension by using the "Index Option" parameter.

Parameters

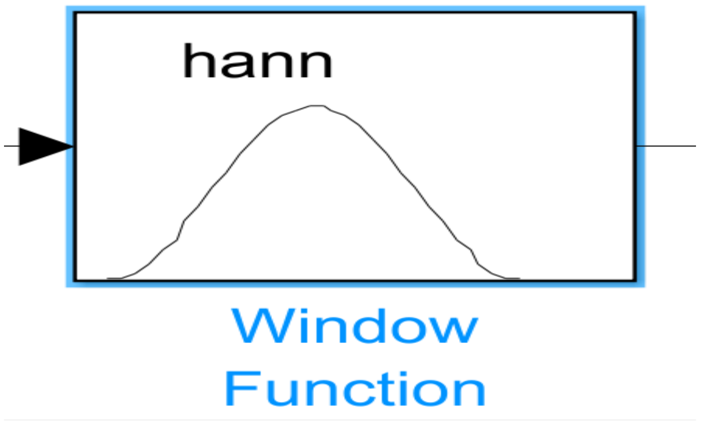
Number of input dimensions: 2

Index mode: One-based

	Index Option	Index	Output Size
1	Select all	n/a	Inherit from p...
2	Index vector (dialog)	1	Inherit from "I...

The **Selector Block** is used to extract a single audio channel (typically the left channel) from the two-channel stereo output of the Audio Capture block. Since most audio processing operations, such as FFT, are performed on a single-channel signal, the Selector block ensures that only one channel is passed forward for proper and consistent processing.

3. **Window Function(Hann):**



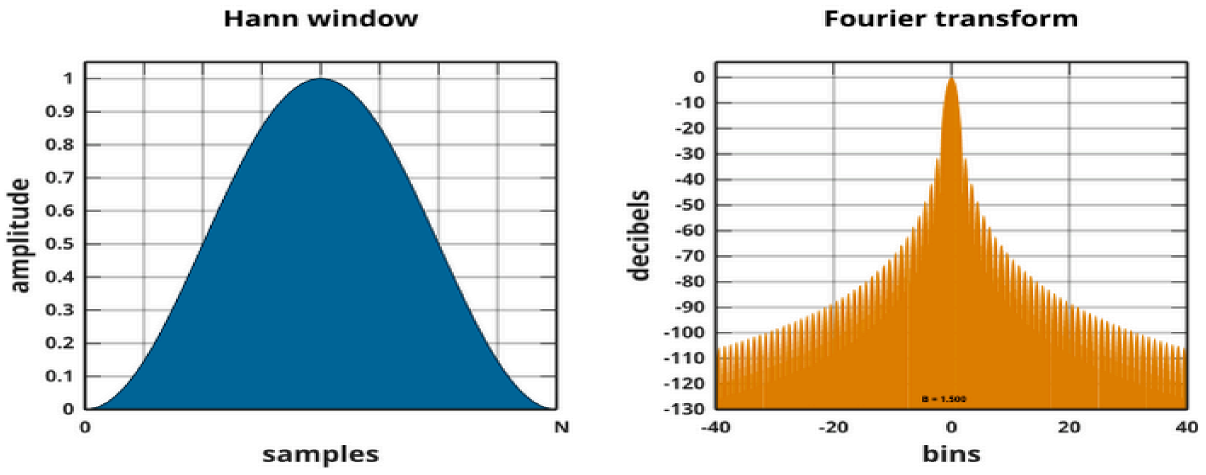
A **Hann window** is applied to the selected audio frame to reduce spectral leakage before performing the FFT. It smooths the edges of the time-domain signal frame so that abrupt discontinuities do not cause problems in the frequency domain.

We have selected the hann function in the Window Function block in simulink.
The following equation generates the coefficients of a Hann window:

$$w(n) = 0.5\left(1 - \cos\left(2\pi \frac{n}{N}\right)\right), \quad 0 \le n \le N$$

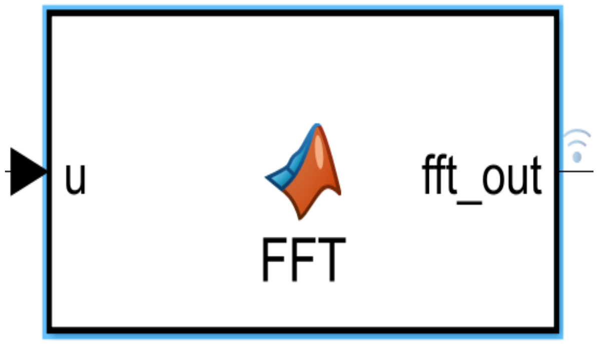
Fig

The window length $L = N + 1$.



Fig

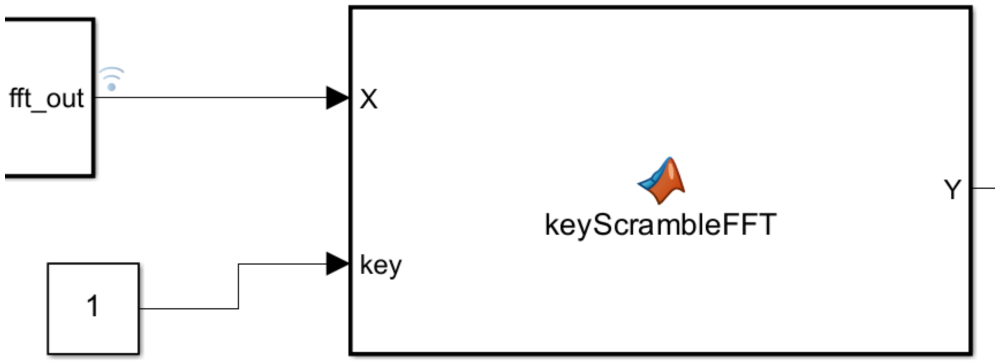
4. FFT Block:



In the **FFT block** we are doing the Fast Fourier transform of our audio signal to convert it from time domain signal into frequency signal using the function fft(u). Here first we input our audio signal in a custom FFT(u) function which is then converted into single channel format. After that we have converted the signal into double precision format and passed it through the inbuilt fft(u) function to get the DFT off our audio signal.

```
function fft_out = FFT(u)
u = u(:,1);
fft_out = fft(double(u)); %double format avoids datatype conflict
end
```

5. **Scrambling of FFT:**



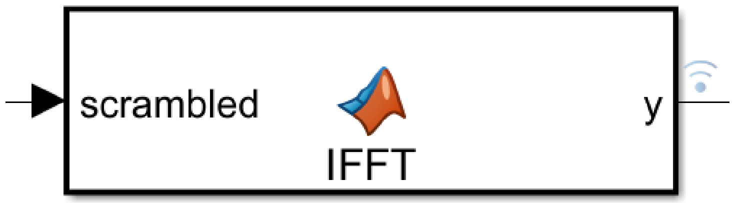
The **keyScrambleFFT** function scrambles the frequency components of an FFT-transformed signal using a key-based permutation, making the signal unintelligible without the correct key. It first determines the positive (posIdx) and negative (negIdx) frequency indices of the FFT output, excluding DC and Nyquist components. Using the input key, it computes two integers a and b (via modular arithmetic) to define a linear congruential permutation which is a way to rearrange a sequence of numbers using a simple modular arithmetic formula

$$perm(i) = mod(a*i + b, K) + 1$$

where K is the number of positive frequencies.

The value of a is chosen such that it is coprime with K, ensuring that the permutation covers all frequency bins uniquely. The positive frequency components of X are then rearranged according to this permutation, and the corresponding negative frequencies are assigned as conjugate pairs to maintain Hermitian symmetry, which is a property of the Fourier Transform of real-valued signals. It means that for a real time-domain signal, the frequency-domain representation has a special mirror symmetry ensuring the resulting signal remains real after IFFT. Because the mapping depends on both a and b derived from the secret key, any change in the key completely alters the scrambling pattern, making it hard to reconstruct the original signal without the exact key, as the original frequency order cannot be recovered without reversing the precise permutation generated by that key.[1]

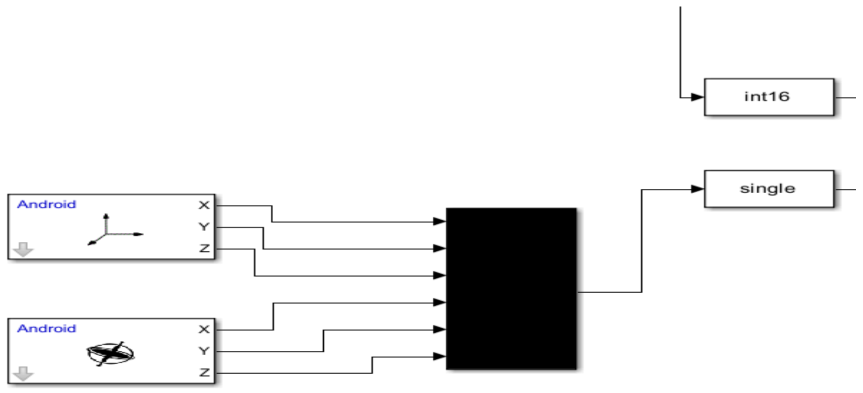
6. **IFFT Block:**



The function **IFFT** performs the inverse Fast Fourier Transform using ifft(scrambled) to convert the frequency-domain signal back to the time domain, and then takes only the real part using real(Channel) to eliminate minor numerical imaginary components caused by computational rounding. Thus, the output y represents the clean, real-valued time-domain signal reconstructed from the scrambled frequency data.

```
function y = IFFT(scrambled)
   Channel = ifft(scrambled);
   y = real(Channel);
end
```

7. **Gyroscope and Accelerometer readings:**

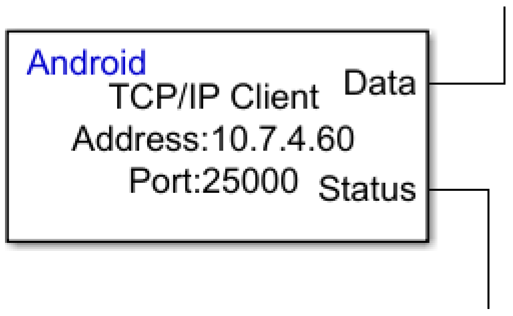


This Simulink pipeline is designed for processing X, Y, and Z coordinate inputs from sensors such as accelerometers or gyroscopes from Android devices. The system begins with Android sensor blocks that capture real-time motion data along three spatial dimensions. These individual axis signals are then fed into a MUX block, which combines the separate X, Y, and Z channels into a single virtual vector output.

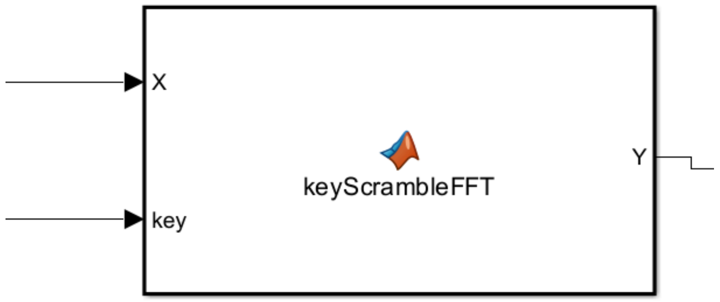
The MUX operates by concatenating the input signals in their connected sequence, effectively packaging the multidimensional sensor data into a unified stream for subsequent processing.

Following the multiplexing stage, branch converts the data to single-precision floating-point format, which preserves finer measurement details and decimal precision, making it suitable for high-accuracy computations.

8. **TCP/IP sender block:**
   This TCP/IP Send block functions as a TCP server that listens for incoming connections on port 25000 and transmits data to connected clients. It accepts uint8 data types, converting them into network packets for transmission. Operating in server mode, it binds to the local port and waits for client connections. Once established, it streams the input data through the TCP connection. This enables real-time data transmission from applications like sensor monitoring or audio streaming to remote clients on the network.

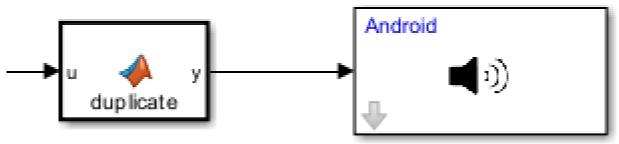


9. UnScrambling



The keyUnscrambleFFT function reverses the scrambling applied by KeyScrambleFFT to recover the original FFT spectrum from the permuted one using the same key. It identifies the positive and negative frequency indices, then reconstructs the same parameters a and b from the key that were used during scrambling. Since the scrambling formula was perm(i) = (a*i + b) mod K, the function computes the modular inverse of a to derive the inverse mapping i = inv_a * (j – b) mod K, ensuring each scrambled index maps back to its original position. This inverse permutation (inv_perm) is applied to reorder the positive-frequency bins of the scrambled spectrum, while DC and Nyquist components are preserved. Finally, the corresponding negative frequencies are restored as conjugates of the positive ones to maintain Hermitian symmetry, ensuring that the recovered spectrum produces the original real-valued time-domain signal after IFFT. Because the process relies on modular arithmetic tied to the secret key, only the exact same key can correctly undo the scrambling.[1]

10. Duplicate

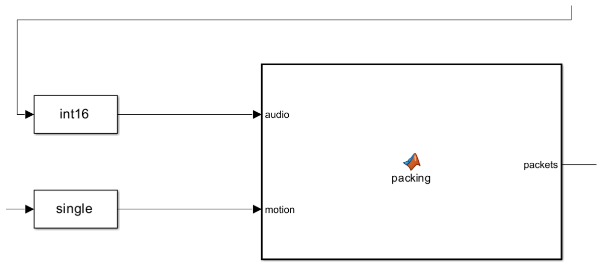


```
function y = duplicate(u)
y = [u, u];
```

The MATLAB function `duplicate(u)` is used to duplicate a single-column signal into two identical columns, matching the input format required by the **Android Audio Capture** block, which expects a **2-channel (stereo)** signal in `'int16'` format.

Motion Data Handling

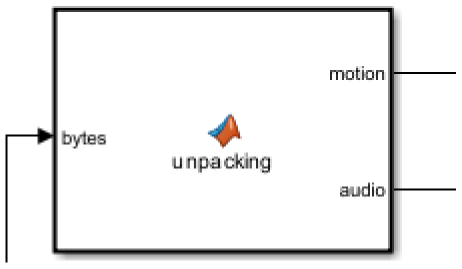
Packing Code



```
function packets = packing(audio, motion)
assert(isa(motion,'single') && numel(motion)==6); % check
assert(isa(audio,'int16') && numel(audio)==4410); % check
packets = zeros(8844,1,'uint8');    % total = 24 + 8820
packets(1:24)     = typecast(motion(:), 'uint8').';
packets(25:8844)  = typecast(audio(:), 'uint8').';
end
```

Packing combines motion sensor data and audio samples into a single data packet for transmission. It first verifies that the motion data is a 6-element vector of type single (each element being 4 bytes, totaling 24 bytes) and that the audio data contains 4410 samples of type int16 (each 2 bytes, totaling 8820 bytes). Together, these form a packet of 8844 bytes. A zero-initialized uint8 array of length 8844 is created since most communication protocols, including TCP/IP, transmit data in bytes (8-bit unsigned integers). The motion data is placed in the first 24 bytes after being typecast to uint8, and the audio data occupies the remaining 8820 bytes. Using uint8 ensures compatibility and consistent byte-level representation for transmission.

Unpacking Code



```
function [motion, audio] = unpacking(bytes)

motion = zeros(1,6,'single');           % 1x6
audio = zeros(4410,1,'int16');          % 4410x1
motion(1,1:6)  = typecast(bytes(1:24), 'single').';   % 6 singles
audio(:,1)  = typecast(bytes(25:8844), 'int16');   % 4410 samples
end
```

The `unpacking(bytes)` function separates a byte array into motion and audio data. It assumes the array has 24 bytes of motion data followed by 8820 bytes of audio, totaling 8844 bytes. The first 24 bytes (`bytes(1:24)`)are converted into six single values, since each requires 4 bytes, giving

$$6\times4=24 \text{ bytes.}$$

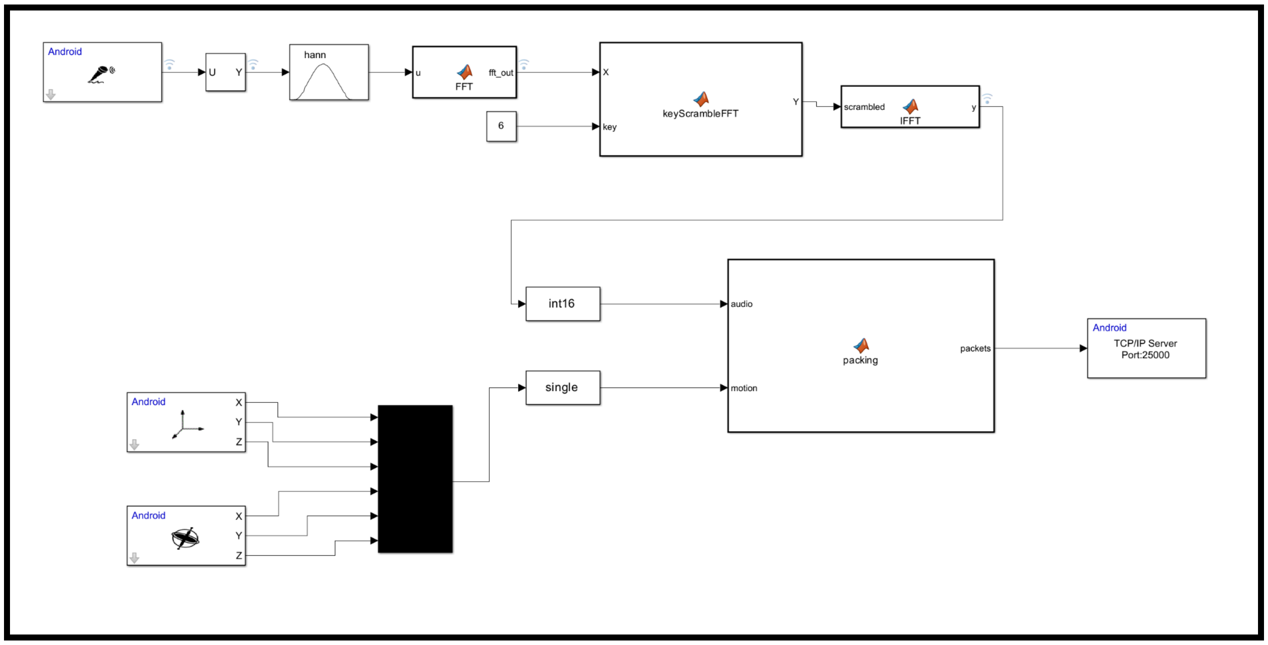
The remaining bytes (`bytes(25:8844)`) are converted into 4410 `'int16'` samples, as each uses 2 bytes,

$$(8844-24)/2=4410 \text{ bytes}$$

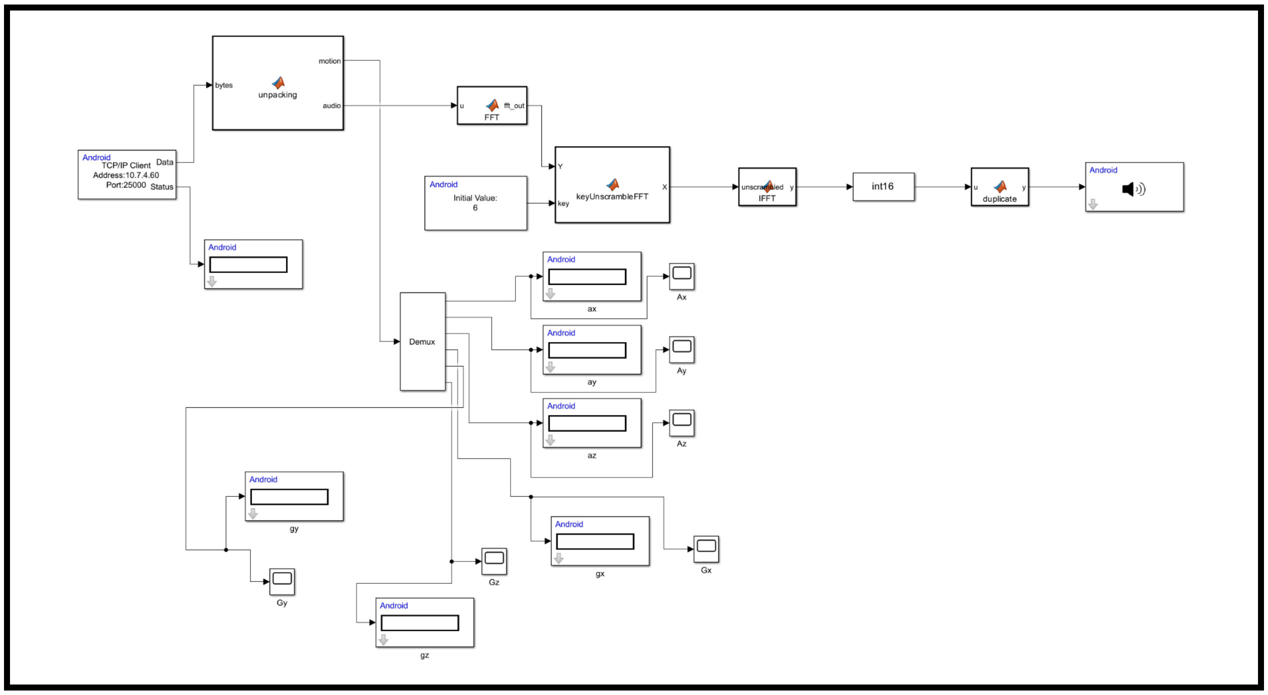
Thus, the function reconstructs structured data from a continuous stream, mapping

$$\text{Total bytes}=(6\times4)+(4410\times2)=8844 \text{ bytes .}$$

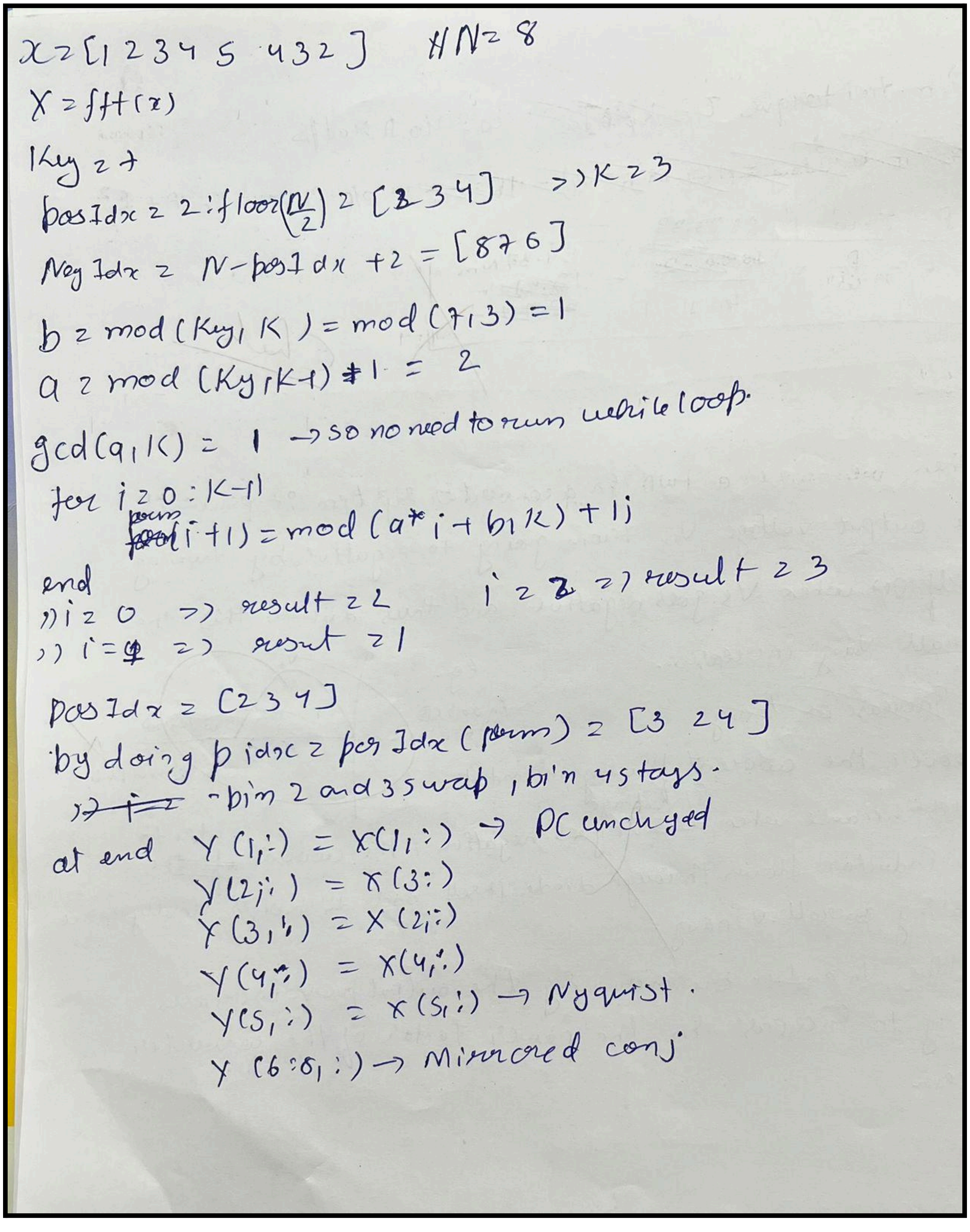
Sender Pipeline



Receiver Pipeline



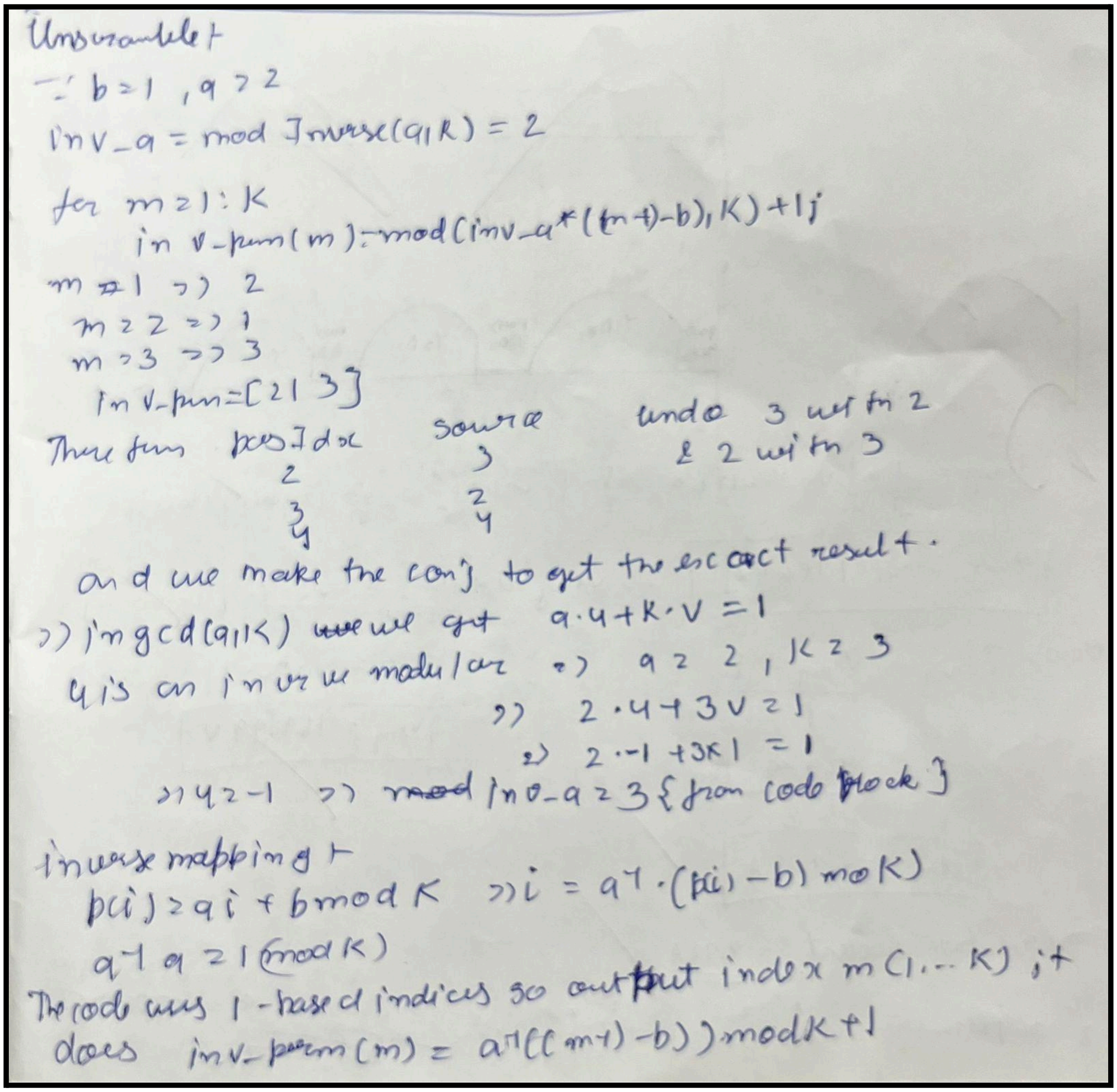
Scrambling Code Algorithm explained with example



```
x = [1 2 3 4 5 4 3 2]     #N = 8
X = fft(x)
Key = 7
posIdx = 2:floor(N/2) = [2 3 4]     => K = 3
NegIdx = N - posIdx + 2 = [8 7 6]
b = mod(Key, K) = mod(7,3) = 1
a = mod(Key, K-1) + 1 = 2
gcd(a, K) = 1  -> so no need to run while loop.
for i = 0:K-1
    perm(i+1) = mod(a*i + b, K) + 1;
end
=> i = 0  =>  result = 2          i = 2 => result = 3
=> i = 1  =>  result = 1

posIdx = [2 3 4]
by doing pidx = posIdx(perm) = [3  2 4]
  ~~=> i =~~ - bin 2 and 3 swap, bin 4 stays.
at end  Y(1,:) = X(1,:) -> DC unchanged
        Y(2,:) = X(3,:)
        Y(3,:) = X(2,:)
        Y(4,:) = X(4,:)
        Y(5,:) = X(5,:) -> Nyquist.
        Y(6:8,:) -> Mirrored conj
```

UnScrambling Code Algorithm explained with example



Unscramble:

∴ $b=1$, $a=2$

inv_a = mod Inverse(a,K) = 2

```
for m=1:K
    inv_perm(m) = mod(inv_a*((m-1)-b), K)+1;
```

m=1 => 2

m=2 => 1

m=3 => 3

inv_perm=[2 1 3]

Therefore

posIdx	source
2	3
3	2
4	4

undo 3 with 2 & 2 with 3

and we make the conj to get the exact result.

=> in gcd(a,K) ~~we~~ we get $a\cdot 4+K\cdot v=1$

4 is an inverse modular => $a=2$, $K=3$

=> $2\cdot 4+3v=1$

=> $2\cdot -1+3\times 1=1$

=> $4=-1$ => ~~mod~~ inv_a = 3 {from code block}

inverse mapping:

$p(i)=ai+b \bmod K$ => $i = a^{-1}\cdot(p(i)-b) \bmod K)$

$a^{-1}a=1 \pmod K$

The code uses 1-based indices so output index m (1…K); it does inv_perm(m) = $a^{-1}((m-1)-b)) \bmod K+1$

Link to Demonstration video - Click here

References

[1] A. Gaffar et al., "A technique for securing multiple digital images based on 2D linear congruential generator, silver ratio, and Galois field," *IEEE Access*, vol. 9, pp. 96126-96150, 2021, doi: 10.1109/ACCESS.2021.3094129.