EE323 Digital Signal Processing
Assignment - 2 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. Record speech on Phone A and apply one of three methods: frequency scrambling, adaptive noise masking, or speech watermarking.
2. Transmit the encrypted or watermarked signal from Phone A to Phone B via Bluetooth.
3. Receive the signal on Phone B and perform the corresponding processing:
    - Method 1: De-scramble the frequency components using the correct key.
    - Method 2: Remove or suppress the adaptive masking noise to recover the speech.
    - Method 3: Extract and verify the embedded watermark for authentication before playback.
4. Allow playback of encrypted or decrypted/authenticated audio on Phone B.
5. Compute and display a basic audio quality metric (SNR/PESQ) after decryption.

Implementation

The design flow for the Simulink-based voice encryption and transmission system is centered around frequency and time domain conversions. The process involves converting the speech signal into the frequency domain, than applying encryption. The overall design ensures that both audio and motion sensor data are acquired, encrypted, synchronized, and transmitted efficiently between two mobile devices.

The encryption function works by first converting the user's key into a deterministic numeric seed, ensuring that every encryption with the same key produces identical results. Using this seed, the RNG is controlled so the random operations remain repeatable. Three methods are then applied: (1) Frequency scrambling, which permutes FFT bins based on a key-dependent permutation; (2) Adaptive pink-noise masking, which generates pink noise, shapes it using the speech envelope, scales it, and adds it to the magnitude to hide intelligible content; and (3) Watermarking, which simply adds pre-generated

watermark noise. In all methods, conjugate symmetry is maintained so the IFFT remains real-valued, and the output size matches the input for seamless audio reconstruction.

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.

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.

Method 1:Key based permutation scrambling:

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]

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]

Method 2: Noise Masking with Adaptive Noise Shaping:

It implements Noise Masking with Adaptive Noise Shaping by generating a synthetic noise that adapts itself to fit the spectral shape of the input speech. The code starts by generating Pink Noise using a deterministic seed derived from the encryption key, ensuring the receiver can mathematically recreate the exact same noise pattern to subtract it later. The adaptive behavior is achieved by extracting the spectral envelope of the original audio using a moving average; this envelope represents the curve of the voice's energy across frequencies. The code then multiplies the synthetic pink noise by this envelope, forcing the noise to be loud at frequencies where the voice is loud and quiet where the voice is soft. This results in a masked signal where the speech is effectively buried under a noise that perfectly matches its own frequency profile, making the audio unintelligible while preserving the signal's general energy structure.

Method 3: Speech Watermarking for Authentication

It implements Speech Watermarking for Authentication using a Spread Spectrum technique, where a low-level, inaudible noise signature is embedded into the audio signal to verify its origin. The transmitter generates a unique, wideband Gaussian noise pattern (200 Hz–15 kHz) based on a shared secret seed; this spread spectrum approach ensures the watermark is robust enough to survive quantization (int16 conversion) and transmission while remaining perceptually hidden beneath the audio. On the receiver side, verification is performed using Normalized Cross-Correlation. The code regenerates the expected noise reference from the same seed and compares it against the received audio's spectrum, applying a

frequency mask to ignore non-watermarked bands for higher sensitivity. The correlation result is normalized by the signal's total energy, allowing the detector to function correctly regardless of the volume level. Finally, a persistent latching logic is used: if the mathematical similarity score exceeds a set threshold even for a single frame, the authentication state locks to "True", confirming that the audio was generated by a trusted source possessing the correct key.

Overall Design Pipeline:

Mode Selection (Sender): On Phone A, a Switch Block is placed immediately after the audio acquisition. This block routes the audio stream to one of the three pipelines.

Transmission: The output of the selected pipeline is fed to the TCP/IP for transmission to Phone B.

Receiver Pipeline: Phone B uses theTCP/IP recive to get the signal. The received data is routed to all three corresponding decryption/restoration blocks in parallel.

Pipeline method 1

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

Pipeline method 2

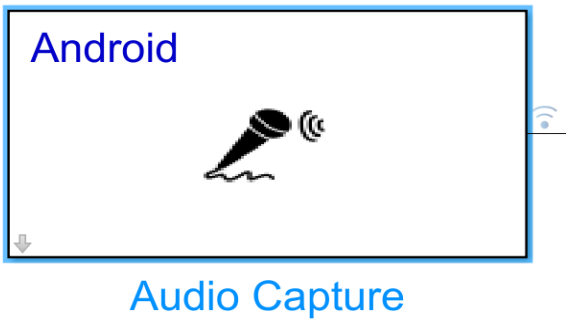
Record audio & motion data → Apply FFT →Add speech based noise masking → Apply IFFT →check the SNR based on original audio → Transmit audio & motion data → Receiver FFT → remove noise through adaptive filtering→ Apply inverse permutation & IFFT → Recover audio & display motion data

Pipeline method 3

Record audio & motion data → Apply FFT →Add seed based watermark→ Apply IFFT → Transmit audio & motion data → Receiver FFT → do watermark check based on noise from same seed

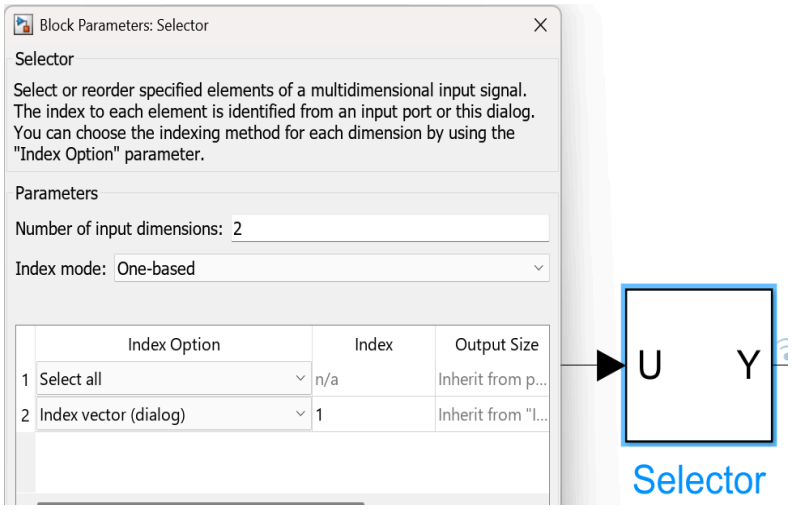
Block by Block Overview:

1. Android Audio capture:



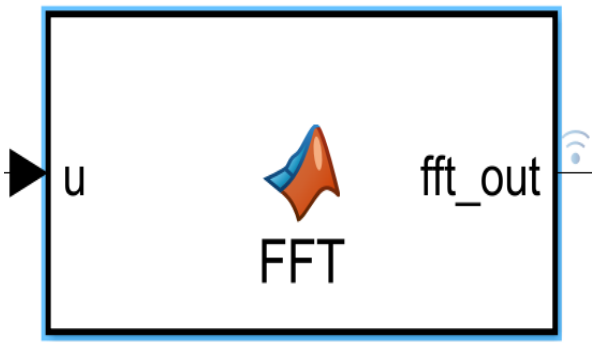
This **Audio Capture block** captures the audio signals from the device microphone. We have set the sampling at standard 44.1kHz frequency with the frame size of 4410. The Frame rate is 0.1 sec. The output given is in format of Nx2 Matrix in int16 data type.

2. **Selector Block:**



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.

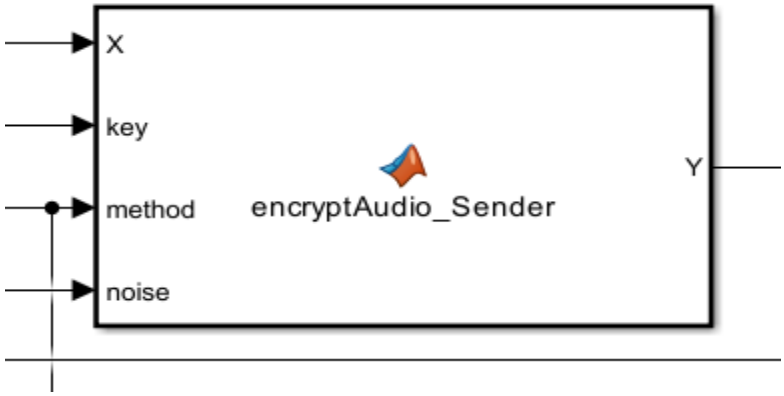
3. **FFT Block:**



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
```

4. **Encryption Block:**



This block takes method, key, audio FFT, noise as input to give the encrypted audio output.

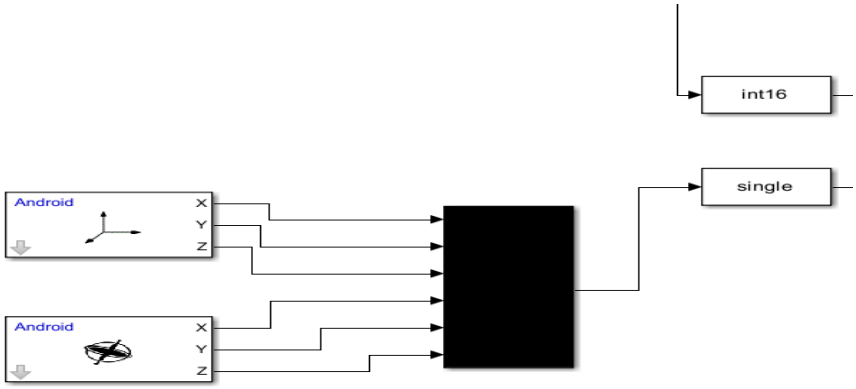
5. **IFFT Block:**

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
```

6. **SNR Block:**
   The function quantifies the quality of the decrypted audio using a time-domain power analysis algorithm that first synchronizes the original and reconstructed signals by truncating them to the shortest common length and converting them to double precision for acurate calculation. It defines noise as the deviation between the original and the processed output by computing the element-wise difference vector. The method then calculates the average power of both the pure signal and this noise vector using the mean squared magnitude, applying a numerical safety floor to prevent division-by-zero errors during periods of silence. Finally, the algorithm derives the Signal-to-Noise Ratio by dividing the signal power by the noise power and converting the result into a logarithmic decibel (dB) scale, where a high value indicates successful, clear decryption and a low value indicates significant distortion or effective encryption.

7. **Gyroscope and Accelerometer readings:**



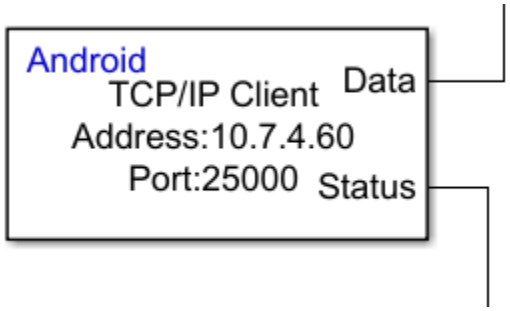
   This Simulink pipeline is designed for processing X, Y, and Z coordinate inputs from sensors such as accelerometers or gyroscopes from Android devices. The system begins with Android sensor blocks that capture real-time motion data along three spatial dimensions. These individual axis signals are then 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.

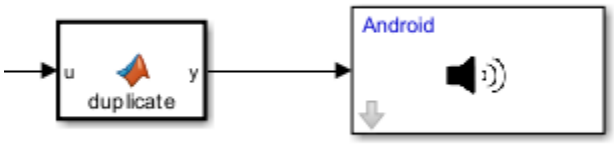
   Following the multiplexing stage, branch converts the data to single-precision floating-point format, which preserves finer measurement details and decimal precision, making it suitable for high-accuracy computations.

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.



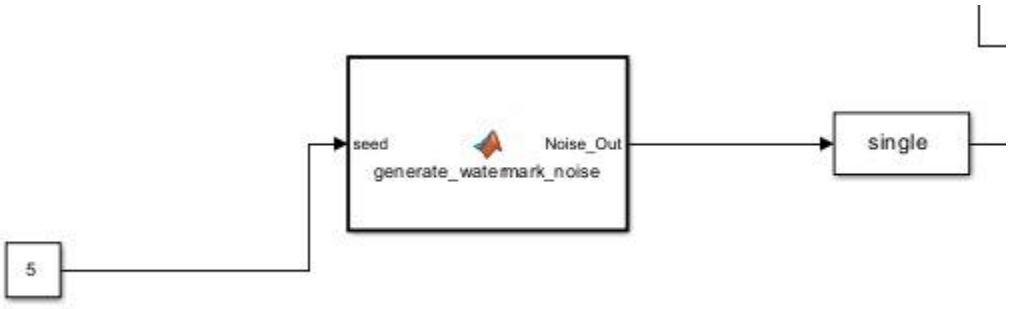
9. Duplicate




```
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.

10. Noise Block



The function creates a seed-dependent spread-spectrum watermark for audio. It uses an FFT size of 4096 and spreads the watermark across 200 Hz to 10 kHz by converting this range into FFT-bin indices:

```
idx_start = floor(f_start_hz / df) + 1;
idx_end   = ceil(f_end_hz / df) + 1;
```

A small amplitude (`alpha =5000`) keeps the watermark inaudible. After seeding MATLAB's RNG:

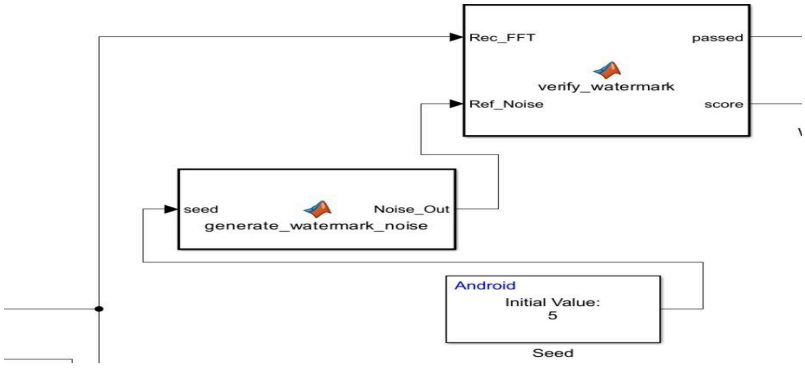
```
rng(uint32(seed), 'twister');
```

The code generates complex Gaussian noise only in the chosen band. These values fill the positive-frequency bins, and then the function applies conjugate mirroring:

```
Noise_Out(mirror_idx) = conj(noise_band(current_bin));
```

This ensures a real signal after IFFT. The result is a 4096-point complex spectrum containing a watermark spread across many frequency bins, making it robust and hard to guess.

11. **Watermark:**

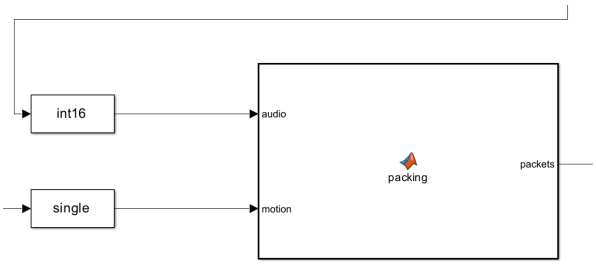


The watermark verification process functions as a spectral fingerprint matcher that determines authenticity by comparing the received audio against a locally regenerated noise reference. The system first applies a frequency mask to isolate the specific spread-spectrum bands (200 Hz–15 kHz) where the watermark resides, discarding irrelevant frequencies to improve sensitivity. It then calculates the energy of both the received signal and the reference noise to serve as a normalization factor. The core detection logic utilizes Normalized Cross-Correlation in the frequency domain. Instead of a standard time-domain convolution, the code performs an element-wise multiplication of the received signal bins with the complex conjugate of the reference noise bins. This effectively aligns the phase of the complex signals to measure their similarity. If the resulting correlation score exceeds the threshold, it indicates that the hidden noise pattern matches the secret key, triggering a permanent True state via a latch.

$$\text{Score} = \frac{\left|\sum_k (R[k] \cdot N^*[k])\right|}{\sqrt{\sum_k |R[k]|^2 \cdot \sum_k |N[k]|^2}}$$

Motion Data Handling

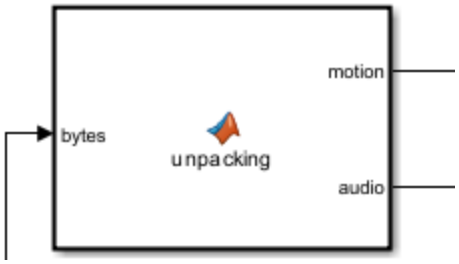
Packing Code




```
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

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

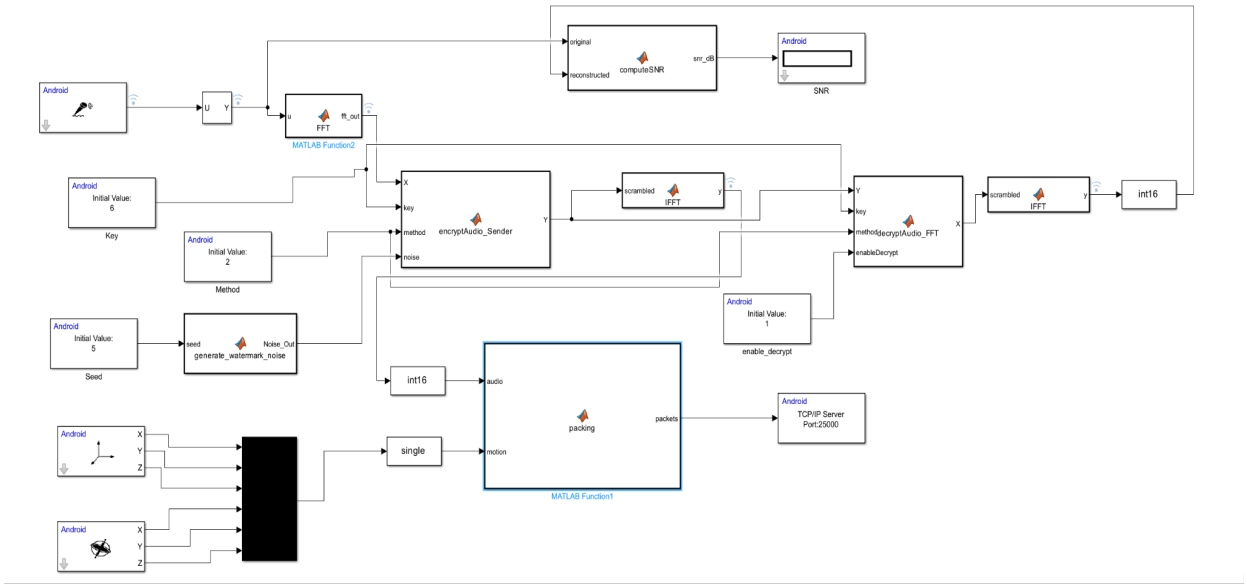
The remaining bytes (`bytes(25:8844)`) are converted into 4410 `'int16'` samples, as each uses 2 bytes,

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

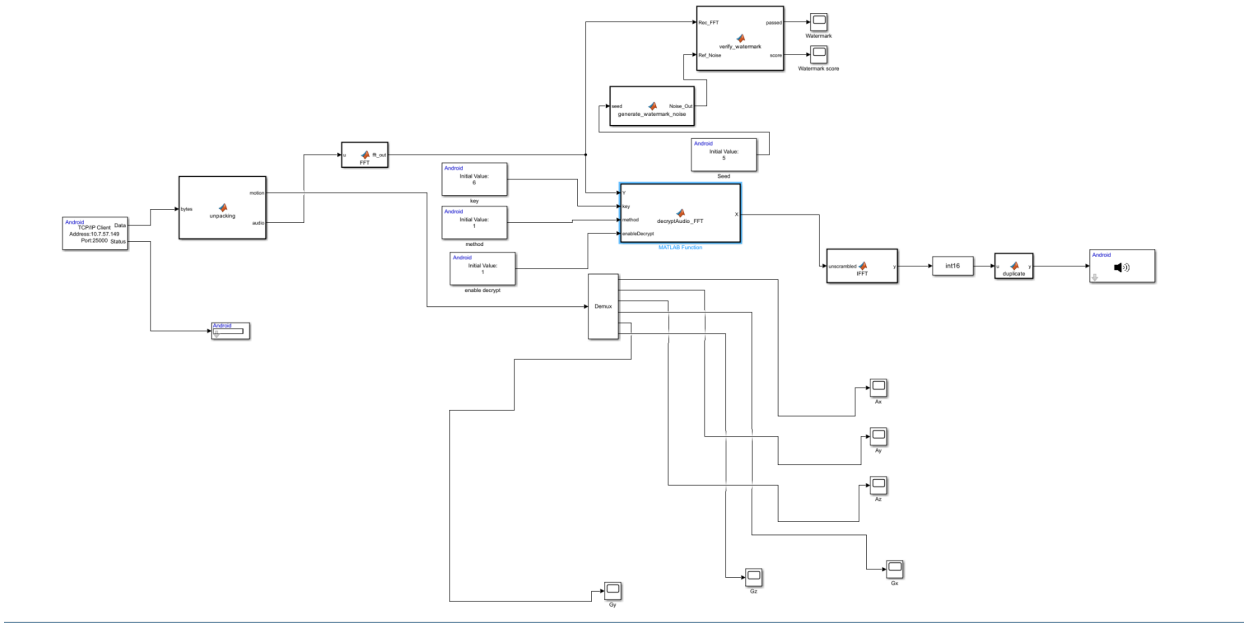
Thus, the function reconstructs structured data from a continuous stream, mapping

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

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) = 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 = perm Idx (perm) = [3 2 4]

- bin 2 and 3 swap, bin 4 stays.

at end Y(1,:) = X(1,:) → DC unchanged

Y(2,:) = X(3,:)

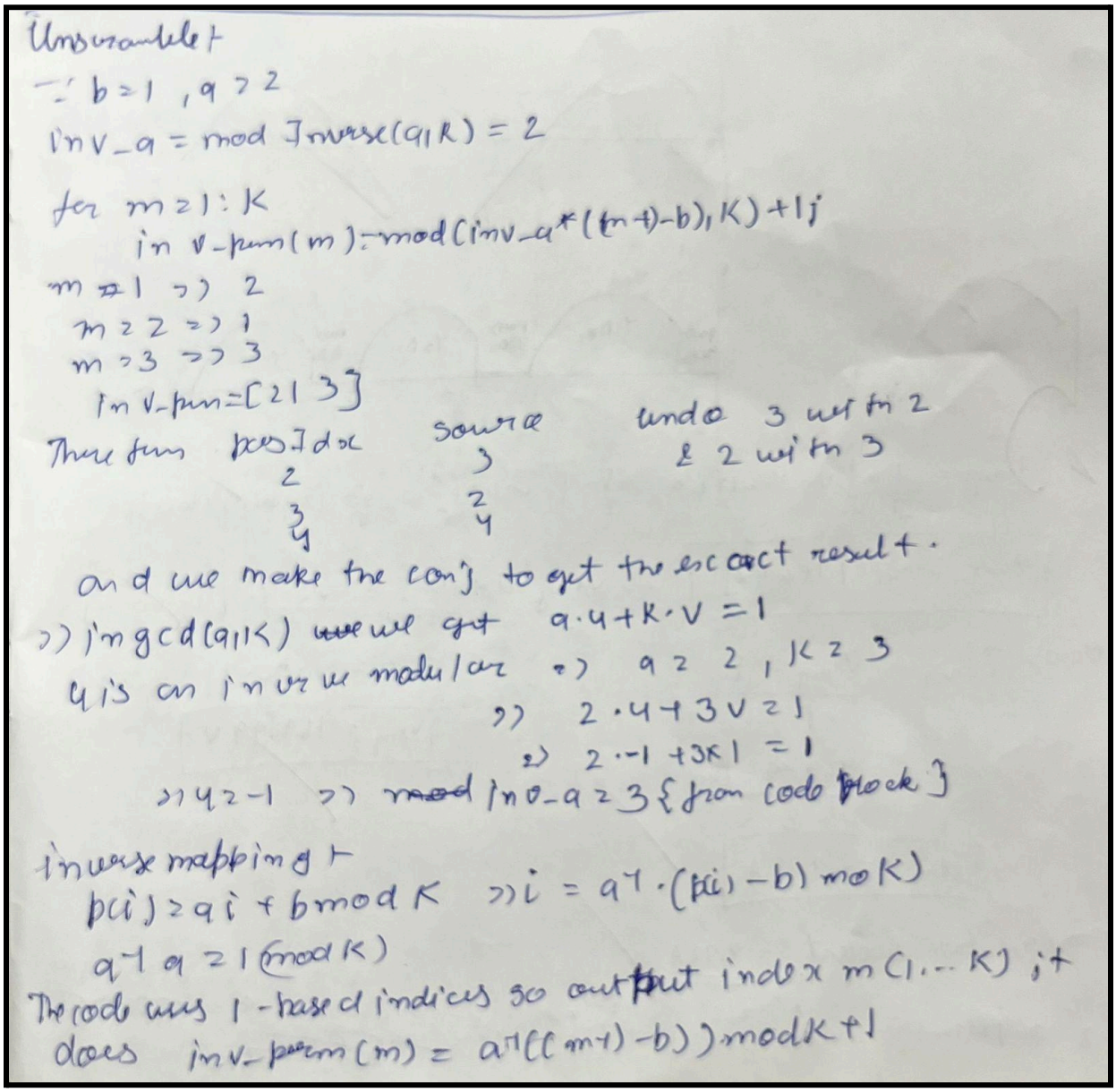
Y(3,:) = X(2,:)

Y(4,:) = X(4,:)

Y(5,:) = X(5,:) → Nyquist.

Y(6:8,:) → Mirrored conj

UnScrambling Code Algorithm explained with example



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.