AUDIO AND SPEECH COMPRESSION USING DCT AND DWT TECHNIQUES

Isaac Dzakpata

Febraury 2026

1 INTRODUCTION

Speech is a basic way for humans to convey information and communicate effectively. The change in the telecommunication infrastructure, in recent years, from circuit switched to packet switched systems has also reflected on the way that speech and audio signals are carried in present systems [4]. Audio compression has since become an important concept in the new multimedia age with a goal of coding audio and speech signals at the lowest possible data rates. The main objective of speech compression is to process human speech signals into an efficient encoded form that can be decoded back to produce a close approximation of the signals [4]. Storage and transmission of uncompressed speech data will be extremely costly and impractical, so we try to reduce the size of the audio signals whiles still maintaining an acceptable quality. Balance is key for audio compression. To compare the efficiency of audio compression methods, this study investigates specialized transform techniques known as Discrete Cosine Transform (DCT) and the Discrete Wavelet Transform (DWT). Discrete Cosine Transform (DCT) is often described as a specialized or "low-level" version of the Discrete Fourier Transform (DFT/FFT). It is frequently used for data compression because it concentrates the energy of a signal (like an image) into a small number of coefficients more effectively.

2 THEORY

There are various techniques for speech compression like waveform coding and parametric coding. This paper focuses on the transform coding techniques which mainly works by converting the signals into the frequency domain and isolating the dominant features only taking out any extra noise which comes off as less dominant peaks or features. In transform method we have used discrete wavelet transform technique and discrete cosine transform technique. When we use wavelet transform technique, the original signal can be represented in terms of wavelet expansion [4]. Similarly in case of DCT transform, speech can be represented in terms of DCT coefficients. Wavelet transform is the latest method of compression because of its ability to describe any type of signals both in time and frequency domain [5]. Transform techniques do not compress the signal, they provide information about the signal and using various encoding techniques like Run-length encoding and Huffman encoding, we then compress the signals with the information deduced. In both methods, the transform coefficients provide an alternative representation of the signal. Many of these coefficients have very small values and contribute little to the overall signal. By removing these small coefficients, significant compression can be achieved while maintaining acceptable signal quality.

3 METHODOLOGY

In this research, speech compression is performed in the following steps using an audio sample:

1. **Transform technique**

   Discrete Cosine Transform (DCT) and Discrete Wavelet Transform (DWT) will both be used on the same audio sample to analyse its effectiveness. The Discrete Cosine Transform (DCT) and Discrete Wavelet Transform (DWT) techniques are applied to convert the speech signal from the time domain into transform coefficients. These coefficients represent the signal in a form where most of the important information is concentrated in a small number of values.

2. **Thresholding of transformed coefficients**

   Hard thresholding was applied to remove coefficients with very small magnitudes, as they contribute minimally to signal quality. This reduces the amount of data required to represent the signal. The threshold value was calculated as 5% of the maximum absolute coefficient value. Mathematically, the threshold is defined as:

   $$T = 0.05 * \max(|C|) \tag{1}$$

   Where:

   - T = threshold value
   - C = transform coefficients
   - $\max(|C|)$ = maximum absolute coefficient value

   All coefficients with absolute values less than the threshold were set to zero:

   $$C_i = \begin{cases} C_i, & |C_i| \geq T \\ 0, & |C_i| < T \end{cases} \tag{2}$$

[1]

Where:

- C_i = transform coefficient at index i
- T = threshold value
- i = index of the coefficient

3. **Quantization**

   It is a process of mapping a set of continuous valued data to a set of discrete valued data. The aim of quantization is to reduce the information found in threshold coefficients. This process makes sure that it produces minimum errors. We perform uniform quantization process using the formular

$$Q_i = \text{round}\left(\frac{C_i}{\Delta}\right) \tag{3}$$

   Where:

   - C_i = original coefficient
   - Δ = quantization step size
   - Q_i = quantized coefficient
   - i = index of the coefficient

4. **Encoding**

   Run Length Encoding method is used to remove data that are repetitively occurring. In encoding we can also reduce the number of coefficients by removing the redundant data. Mathematically, it is represented as

$$X = [x_1, x_2, x_3, ..., x_n] \tag{4}$$

   Where:

   - X = original sequence of coefficients
   - $x_1, x_2, ..., x_n$ = individual coefficient values
   - n = total number of coefficients

   Encoded signal:

$$R = [(v_1, r_1), (v_2, r_2), ..., (v_k, r_k)] \tag{5}$$

   Where:

   - v_i = value
   - r_i = number of repetitions

- k = number of encoded pairs

5. **Reconstruction**

   The speech signal was reconstructed using inverse transform techniques. The compressed coefficients were first dequantized and decoded, and then the inverse Discrete Cosine Transform (IDCT) and inverse Discrete Wavelet Transform (IDWT) were applied to obtain the reconstructed signal in the time domain. The reconstructed signal represents an approximation of the original speech signal, as some information is lost during thresholding and quantization.

6. **Performance Evaluation**

   The time domain signals was used to compare both techniques using various evaluation measures.

   Compression ratio(CR) formula shows how much the file size was reduced

$$CR = \frac{\text{Original Size}}{\text{Compressed Size}} \tag{6}$$

   Where:

   - CR = compression ratio
   - Original Size = size of original signal
   - Compressed Size = size after compression

   Signal to Noise Ratio(SNR) measures how similar reconstructed signal is to the original

$$SNR = 10\log_{10}\left(\frac{\sum signal^2}{\sum(signal - reconstructed)^2}\right) \tag{7}$$

   Where:

   - SNR = signal to noise ratio
   - $signal$ = original speech signal
   - $reconstructed$ = reconstructed speech signal
   - $\sum$ = summation over all samples

   Mean Squared Error(MSE) measures reconstruction error

$$MSE = \frac{1}{N}\sum(signal - reconstructed)^2 \tag{8}$$

   Where:

   - MSE = mean squared error

- $signal$ = original speech signal
- $reconstructed$ = reconstructed speech signal
- N = total number of samples

4 RESULTS

This section presents the results obtained from applying Discrete Cosine Transform (DCT) and Discrete Wavelet Transform (DWT) techniques to the speech sample *LJ025-0076.wav* [2]. The performance of both methods was evaluated using Compression Ratio (CR), Signal-to-Noise Ratio (SNR), and Mean Squared Error (MSE).

4.1 Waveform Comparison

Figure 1 shows the comparison between the original speech signal and the reconstructed signal using DCT. It can be observed that the reconstructed signal closely follows the original waveform, although slight distortions are present due to compression.

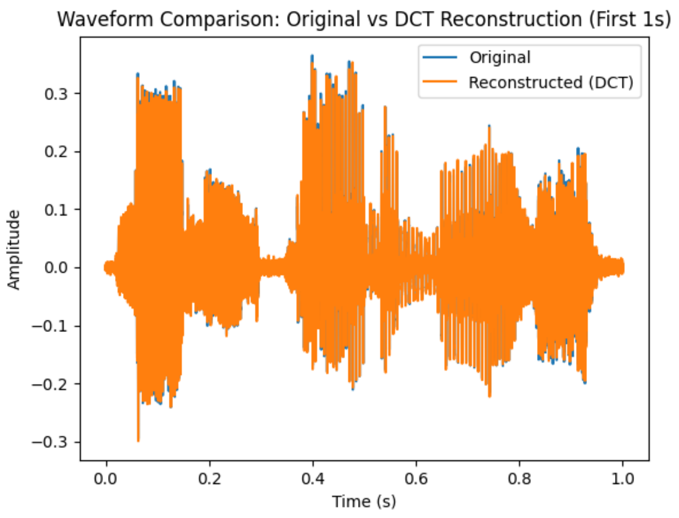

Figure 1: Waveform comparison between original and reconstructed signal using DCT

Figure 2 shows the waveform comparison for the DWT method. The reconstructed signal using DWT shows better similarity to the original waveform.

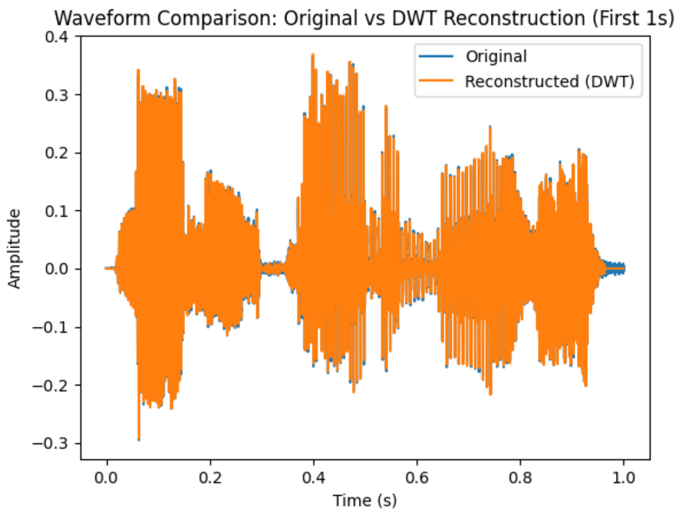

Figure 2: Waveform comparison between original and reconstructed signal using DWT

4.2 Coefficient Analysis

Figure 3 shows the DCT coefficients before and after thresholding. It can be seen that many small coefficients were removed, which contributes to compression.

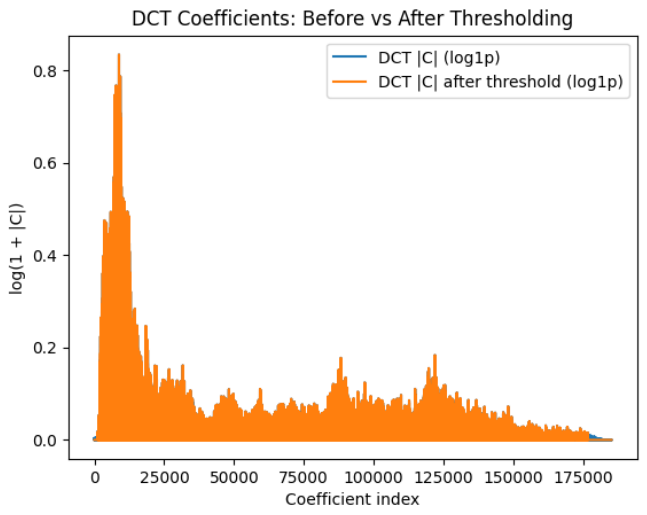

Figure 3: DCT coefficients before and after thresholding

Figure 4 shows the DWT coefficients before and after thresholding. More coefficients were reduced compared to DCT, indicating better compression efficiency.

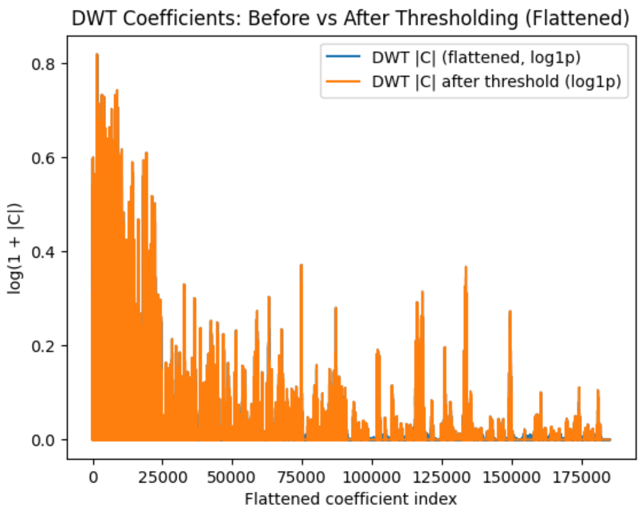

Figure 4: DWT coefficients before and after thresholding

4.3 Threshold Values

The threshold values obtained for both methods are shown below:

- DCT Threshold: $T = 0.013028$
- DWT Threshold: $T = 0.012686$

These threshold values represent 1% of the maximum coefficient magnitude and determine which coefficients were removed during compression.

4.4 Compression Performance

The compression performance of both methods is shown in Table 1.

Table 1: Compression Performance Comparison

Method	Encoded Pairs	Compressed Size (bytes)	Compression Ratio
DCT	130363	1042904	0.355
DWT	53690	429520	0.862

From the results, DWT produced significantly fewer encoded pairs compared to DCT. This indicates that DWT was more efficient in representing the signal using fewer coefficients.

Although the compression ratio values were less than 1, indicating that the compressed representation was larger than the original file in this implementation, the DWT method still achieved a better compression performance compared to DCT.

4.5 Signal Quality Evaluation

The quality of the reconstructed signals was evaluated using SNR and MSE. The results are presented in Table 2.

Table 2: Signal Quality Comparison

Method	SNR (dB)	MSE
DCT	23.251	0.00002015
DWT	25.289	0.00001260

The DWT method achieved a higher Signal-to-Noise Ratio compared to DCT. A higher SNR indicates that the reconstructed signal is closer to the original signal and has less distortion.

Similarly, the Mean Squared Error for DWT was lower than that of DCT. Lower MSE values indicate better reconstruction accuracy.

4.6 Conclusion

From the results, it is evident that the Discrete Wavelet Transform performed better than the Discrete Cosine Transform in both compression efficiency and signal quality. The performance of DWT can be attributed to its ability to represent signals in both time and frequency domains simultaneously. This allows DWT to capture important speech characteristics more efficiently than DCT, which only represents signals in the frequency domain. Furthermore, the reconstructed speech signals were audible and clear, although slight distortion was present due to the effects of thresholding and quantization. This distortion represents the loss of some information during compression. The results demonstrate that both DCT and DWT can be used for speech compression.

Acknowledgment

ChatGPT (OpenAI) was used to assist with Python code development, debugging, and understanding of Discrete Cosine Transform and Discrete Wavelet Transform implementation for speech compression [3]. The complete Python implementation used in this project is provided in Appendix A for reproducibility.

References

[1] Universal Audio. Audio compression basics, 2024. URL https://www.uaudio.com/blogs/ua/audio-compression-basics. Accessed: 2026.

[2] Keith Ito and Linda Johnson. The lj speech dataset. https://keithito.com/LJ-Speech-Dataset/, 2017.

[3] OpenAI. ChatGPT (GPT-5.2). https://chatgpt.com, 2026. Used for guidance on Python implementation and debugging.

[4] MV Patil, Apoorva Gupta, Ankita Varma, and Shikhar Salil. Audio and speech compression using dct and dwt techniques. *International Journal of Innovative Research in Science, Engineering and Technology*, 2(5):1712–1719, 2013.

[5] K.A. Subramanian and R. Karthigeyan. Wavelet transform and fast fourier transform for signal compression: A comparative study. *International Journal of Engineering Research & Technology (IJERT)*, 2(1), 2014. IFET–2014 Conference.

A Python Implementation Code

The following Python code was used to implement the speech compression using Discrete Cosine Transform (DCT) and Discrete Wavelet Transform (DWT), including thresholding, quantization, encoding, reconstruction, and performance evaluation.

```
"""
AUDIO & SPEECH COMPRESSION USING DCT AND DWT

Steps implemented:
1) Transform technique (DCT and DWT)
2) Hard Thresholding: T = alpha * max(|C|)
3) Uniform Quantization: Q_i = round(C_i /   )
4) Run-Length Encoding (RLE) + Decoding
5) Reconstruction (IDCT / IDWT)
6) Performance Evaluation (CR, SNR, MSE) + Plots

USAGE:
  python3 speechcompression.py --input "LJ025-0076.wav" --delta 0.001
  python3 speechcompression.py --input "LJ025-0076.wav" --alpha 0.01 --delta 0.00025

Optional parameters:
  --alpha 0.05        (controls how many transform coefficients are removed)
  --delta 0.001       (controls how much rounding happens)
  --wavelet db4
  --level 4
  --channel left

Outputs:
  - reconstructed_dct.wav
  - reconstructed_dwt.wav
  - plots: waveform comparisons + coefficient plots
"""

import os
import argparse
import numpy as np
import matplotlib.pyplot as plt

import soundfile as sf
import pywt
from scipy.fftpack import dct, idct


# ----------------------------
# Utility: Metrics
# ----------------------------
def mse(original: np.ndarray, reconstructed: np.ndarray) ->
    float:
    """Mean Squared Error (MSE) = (1/N) * sum((x - x_hat)^2)
        """
    original = original.astype(np.float64)
    reconstructed = reconstructed.astype(np.float64)
    return float(np.mean((original - reconstructed) ** 2))


def snr_db(original: np.ndarray, reconstructed: np.ndarray,
    eps: float = 1e-12) -> float:
    """
    SNR = 10 log10( sum(x^2) / sum((x-x_hat)^2) )
    """
    original = original.astype(np.float64)
    reconstructed = reconstructed.astype(np.float64)
    noise = original - reconstructed
    num = np.sum(original ** 2)
    den = np.sum(noise ** 2) + eps
    return float(10.0 * np.log10(num / den))


def compression_ratio(original_size_bytes: int,
    compressed_size_bytes: int) -> float:
    """CR = Original Size / Compressed Size"""
    if compressed_size_bytes <= 0:
```

```python
        return float("inf")
    return original_size_bytes / compressed_size_bytes


# ----------------------------
# Utility: RLE
# ----------------------------
def run_length_encode_int(arr: np.ndarray):
    """
    Run-Length Encode a 1D integer array into list of (value
        , count).
    """
    if arr.size == 0:
        return []

    encoded = []
    prev = int(arr[0])
    count = 1

    for x in arr[1:]:
        x = int(x)
        if x == prev:
            count += 1
        else:
            encoded.append((prev, count))
            prev = x
            count = 1

    encoded.append((prev, count))
    return encoded


def run_length_decode_int(encoded):
    """Decode list of (value, count) back to 1D integer
       numpy array."""
    if not encoded:
        return np.array([], dtype=np.int64)
    out = []
    for value, count in encoded:
        out.extend([int(value)] * int(count))
    return np.array(out, dtype=np.int64)


def estimate_rle_storage_bytes(encoded) -> int:
    """
    Rough storage estimate for RLE payload.
    Assumption: store each pair (value, count) as two 32-bit
         signed ints => 8 bytes/pair.
    """
    return len(encoded) * 8
```

```python


# ----------------------------
# Audio helpers
# ----------------------------
def normalize_to_unit(x: np.ndarray, eps: float = 1e-12) ->
    np.ndarray:
    """
    Normalize to [-1, 1] using peak normalization.
    Keeps relative shape but avoids huge coefficient scales.
    """
    x = x.astype(np.float64, copy=False)
    m = np.max(np.abs(x)) + eps
    if m > 1.0:
        return x / m
    return x


def normalize_for_wav(x: np.ndarray, eps: float = 1e-12) ->
    np.ndarray:
    """
    Normalize ONLY for saving as WAV so playback is audible.
    (Does not change metrics if you compute metrics before
        calling this.)
    """
    x = x.astype(np.float64, copy=False)
    m = np.max(np.abs(x)) + eps
    return (x / m * 0.95).astype(np.float32)


# ----------------------------
# Step 2: Thresholding
# ----------------------------
def hard_threshold(coeffs: np.ndarray, alpha: float):
    """
    Hard threshold:
      T = alpha * max(|C|)
      C_i = 0 if |C_i| < T
    """
    coeffs = coeffs.astype(np.float64, copy=True)
    T = alpha * np.max(np.abs(coeffs)) if coeffs.size else
        0.0
    coeffs[np.abs(coeffs) < T] = 0.0
    return coeffs, T


# ----------------------------
# Step 3: Quantization
# ----------------------------
def uniform_quantize(coeffs: np.ndarray, delta: float):
    """
    Uniform quantization:
      Q_i = round(C_i /    )
    Returns integer Q array.
    """
    if delta <= 0:
        raise ValueError("delta must be > 0")
    Q = np.round(coeffs / delta).astype(np.int64)
    return Q


def uniform_dequantize(Q: np.ndarray, delta: float):
    """Dequantization: C'_i = Q_i *    """
    return (Q.astype(np.float64) * delta).astype(np.float64)


# ----------------------------
# DCT Pipeline
# ----------------------------
def dct_compress_decompress(signal: np.ndarray, alpha: float, delta: float):
    C = dct(signal, norm="ortho")
    C_thr, T = hard_threshold(C, alpha)
    Q = uniform_quantize(C_thr, delta)
    encoded = run_length_encode_int(Q)
    compressed_bytes_est = estimate_rle_storage_bytes(encoded)

    Q_dec = run_length_decode_int(encoded)
    C_deq = uniform_dequantize(Q_dec, delta)
    recon = idct(C_deq, norm="ortho")

    return {
        "C": C,
        "threshold_T": T,
        "C_thresholded": C_thr,
        "Q": Q,
        "encoded": encoded,
        "compressed_bytes_est": compressed_bytes_est,
        "reconstructed": recon,
    }


# ----------------------------
# DWT Pipeline
# ----------------------------
def flatten_dwt_coeffs(coeff_list):
    shapes = [c.shape for c in coeff_list]
    flat = np.concatenate([c.ravel() for c in coeff_list]).astype(np.float64)
    return flat, shapes


def unflatten_dwt_coeffs(flat: np.ndarray, shapes):
    coeffs = []
    idx = 0
    for shp in shapes:
        size = int(np.prod(shp))
        part = flat[idx: idx + size].reshape(shp)
        coeffs.append(part)
        idx += size
    return coeffs


def dwt_compress_decompress(signal: np.ndarray, alpha: float
    , delta: float, wavelet: str, level: int):
    coeff_list = pywt.wavedec(signal, wavelet=wavelet, level
        =level)
    flat, shapes = flatten_dwt_coeffs(coeff_list)

    flat_thr, T = hard_threshold(flat, alpha)
    Q = uniform_quantize(flat_thr, delta)
    encoded = run_length_encode_int(Q)
    compressed_bytes_est = estimate_rle_storage_bytes(
        encoded)

    Q_dec = run_length_decode_int(encoded)
    flat_deq = uniform_dequantize(Q_dec, delta)
    coeffs_rebuilt = unflatten_dwt_coeffs(flat_deq, shapes)
    recon = pywt.waverec(coeffs_rebuilt, wavelet=wavelet)

    return {
        "coeff_list": coeff_list,
        "flat": flat,
        "threshold_T": T,
        "flat_thresholded": flat_thr,
        "Q": Q,
        "encoded": encoded,
        "compressed_bytes_est": compressed_bytes_est,
        "reconstructed": recon,
    }


# ----------------------------
# Main
# ----------------------------
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--input", required=True, help="Path
         to WAV file (e.g., LJ025-0076.wav)")
    parser.add_argument("--alpha", type=float, default=0.05, help="Hard threshold factor (default 0.05)")
    parser.add_argument("--delta", type=float, default=0.001, help="Quantization step size (default 0.001)")
    parser.add_argument("--wavelet", type=str, default="db4", help="Wavelet name (default db4)")
    parser.add_argument("--level", type=int, default=4, help="DWT level (default 4)")
    parser.add_argument("--channel", type=str, default="left",
                        choices=["left", "right", "avg"],
                        help="If stereo: choose left/right/avg (default left)")
    args = parser.parse_args()

    # Load audio
    signal, sr = sf.read(args.input, always_2d=True)  # shape (N, channels)
    n_samples, n_channels = signal.shape

    # Select channel handling
    if n_channels == 1:
        mono = signal[:, 0].astype(np.float64)
        channel_used = "mono"
    else:
        if args.channel == "left":
            mono = signal[:, 0].astype(np.float64)
            channel_used = "left"
        elif args.channel == "right":
            mono = signal[:, 1].astype(np.float64)
            channel_used = "right"
        else:
            mono = signal.mean(axis=1).astype(np.float64)
            channel_used = "avg"

    # Normalize input to [-1, 1] for stable transforms
    mono = normalize_to_unit(mono)

    original_file_bytes = os.path.getsize(args.input)

    print("\n--- INPUT INFO ---")
    print(f"File: {args.input}")
    print(f"Sample rate: {sr} Hz")
    print(f"Samples (N): {mono.size}")
    print(f"Channels in file: {n_channels} (used: {channel_used})")
    print(f"Original file size: {original_file_bytes} bytes")
    print(f"alpha (threshold factor): {args.alpha}")
```

```python
    print(f"delta (quant step): {args.delta}")
    print(f"wavelet: {args.wavelet}, level: {args.level}")

    # ---- DCT pipeline ----
    dct_res = dct_compress_decompress(mono, alpha=args.alpha
        , delta=args.delta)
    recon_dct = dct_res["reconstructed"][: mono.size]

    # ---- DWT pipeline ----
    dwt_res = dwt_compress_decompress(mono, alpha=args.alpha
        , delta=args.delta, wavelet=args.wavelet, level=args.
        level)
    recon_dwt = dwt_res["reconstructed"][: mono.size]

    # ---- Performance evaluation ----
    cr_dct = compression_ratio(original_file_bytes, dct_res[
        "compressed_bytes_est"])
    cr_dwt = compression_ratio(original_file_bytes, dwt_res[
        "compressed_bytes_est"])

    snr_dct = snr_db(mono, recon_dct)
    snr_dwt = snr_db(mono, recon_dwt)

    mse_dct = mse(mono, recon_dct)
    mse_dwt = mse(mono, recon_dwt)

    print("\n--- THRESHOLDS ---")
    print(f"DCT threshold T = {dct_res['threshold_T']:.6f}")
    print(f"DWT threshold T = {dwt_res['threshold_T']:.6f}")

    print("\n--- COMPRESSION (Estimated) ---")
    print(f"DCT encoded pairs: {len(dct_res['encoded'])},
        est. bytes: {dct_res['compressed_bytes_est']}")
    print(f"DWT encoded pairs: {len(dwt_res['encoded'])},
        est. bytes: {dwt_res['compressed_bytes_est']}")
    print(f"CR (DCT) = {cr_dct:.3f}")
    print(f"CR (DWT) = {cr_dwt:.3f}")

    print("\n--- QUALITY ---")
    print(f"SNR (DCT) = {snr_dct:.3f} dB")
    print(f"SNR (DWT) = {snr_dwt:.3f} dB")
    print(f"MSE (DCT) = {mse_dct:.8f}")
    print(f"MSE (DWT) = {mse_dwt:.8f}")

    # Save reconstructed audio for listening (normalize ONLY
         for saving)
    sf.write("reconstructed_dct.wav", normalize_for_wav(
        recon_dct), sr)
    sf.write("reconstructed_dwt.wav", normalize_for_wav(
        recon_dwt), sr)
    print("\nSaved: reconstructed_dct.wav, reconstructed_dwt.wav")

    # ---- Plots ----
    # Show first 1 second (or less if file shorter)
    L = int(min(mono.size, sr * 1))
    t = np.arange(L) / sr

    plt.figure()
    plt.plot(t, mono[:L], label="Original")
    plt.plot(t, recon_dct[:L], label="Reconstructed (DCT)")
    plt.title("Waveform Comparison: Original vs DCT Reconstruction (First 1s)")
    plt.xlabel("Time (s)")
    plt.ylabel("Amplitude")
    plt.legend()
    plt.show()

    plt.figure()
    plt.plot(t, mono[:L], label="Original")
    plt.plot(t, recon_dwt[:L], label="Reconstructed (DWT)")
    plt.title("Waveform Comparison: Original vs DWT Reconstruction (First 1s)")
    plt.xlabel("Time (s)")
    plt.ylabel("Amplitude")
    plt.legend()
    plt.show()

    plt.figure()
    plt.plot(np.log1p(np.abs(dct_res["C"])), label="DCT |C| (log1p)")
    plt.plot(np.log1p(np.abs(dct_res["C_thresholded"])), label="DCT |C| after threshold (log1p)")
    plt.title("DCT Coefficients: Before vs After Thresholding")
    plt.xlabel("Coefficient index")
    plt.ylabel("log(1 + |C|)")
    plt.legend()
    plt.show()

    plt.figure()
    plt.plot(np.log1p(np.abs(dwt_res["flat"])), label="DWT |C| (flattened, log1p)")
    plt.plot(np.log1p(np.abs(dwt_res["flat_thresholded"])), label="DWT |C| after threshold (log1p)")
    plt.title("DWT Coefficients: Before vs After Thresholding (Flattened)")
    plt.xlabel("Flattened coefficient index")
    plt.ylabel("log(1 + |C|)")
    plt.legend()
    plt.show()


if __name__ == "__main__":
    main()
```