Ioniță Alexandru-Mihail **UNSTPB**
1231B **FILS**

Discrete Cosine Transform (DCT) Compression For Audio Files

Digital Signal

Processing

2024

Ioniță Alexandru-Mihail
1231B

UNSTPB
FILS

Contents:

I) About the Project

For the Digital Signal Processing project, I chose to write a program in Python that selects an input audio file, an output audio file, and compresses the contents by using Discrete Cosine Transform. In order to reconstruct the original audio, we are applying the Inverse Discrete Cosine Transform, which approximates the original data, but with fewer data points, due to the removal of less important coefficients.

After the sound is processed, the program is displaying a graph which contains:

- The spectrogram of the original audio file
- The spectrogram of the compressed audio file
- The waveform of the original audio file
- The waveform of the compressed audio file

These visual representations help us assess the quality of the compressed audio and the efficiency of the compressed audio.

II) Used Libraries

Our code is written in Python, and takes advantage of multiple external libraries. These are:

- **NumPy**: For numerical operations and array handling.
  - **numpy**: Core library for handling arrays and mathematical operations.
- **SciPy**: For advanced mathematical and signal processing operations.
  - **scipy.fftpack**: Contains functions for Discrete Cosine Transform (DCT) and its inverse (IDCT).
  - **scipy.signal**: Provides the spectrogram function for computing spectrograms of audio signals.
- **Tkinter**: For creating the graphical user interface (GUI) to select files and directories.
  - **tkinter**: The main library for GUI creation in Python.
  - **tkinter**.filedialog: Utility for file and directory selection dialogs.
  - **tkinter**.messagebox: For displaying error and success messages.
- **Pydub**: For audio file manipulation and conversion.
  - **pydub**: Core library for handling various audio file formats and conversions.

- **Matplotlib**: For plotting and visualizing data.
    - **matplotlib.pyplot**: Provides functionalities for creating and displaying plots, including spectrograms and waveforms.
- **OS**: For interacting with the operating system, such as creating directories.

III) About DCT Compression

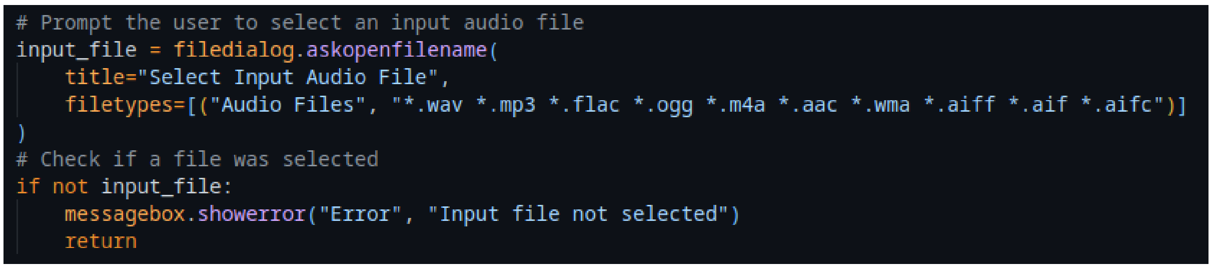
When we want to use Discrete Cosine Transform, the audio data is first divided into smaller segments (or blocks), on which we apply the DCT itself. The DCT is then transforming these blocks from the time domain (which is amplitude vs time) into the frequency domain.

In the frequency domain, many of the high-frequency coefficients will have very small values. Since these coefficients are less perceptible to the human ear, we are setting them to zero in order to achieve compression. The amount of which zeroing that we are doing is given by the compression factor.

After applying the DCT compression, we are reconstructing the audio signal by using the Inverse Discrete Cosine Transform (IDCT). This way, we are converting the data blocks back from the frequency domain to the time domain. The reconstructed signal is one that approximates the original signal but with fewer data points, since we removed the higher frequencies that humans have a harder time to perceive.

IV) Code Description

First, the program asks the user for the audio file that should be compressed and the destination file. For my testing I used a cat sound effect, so our input file is named “cat_sound_effect.mp3” and the output file is named “cat_compressed.mp3”. The file selection UI is done with the TKinter library.



```python
# Prompt the user to select an input audio file
input_file = filedialog.askopenfilename(
    title="Select Input Audio File",
    filetypes=[("Audio Files", "*.wav *.mp3 *.flac *.ogg *.m4a *.aac *.wma *.aiff *.aif *.aifc")]
)
# Check if a file was selected
if not input_file:
    messagebox.showerror("Error", "Input file not selected")
    return
```

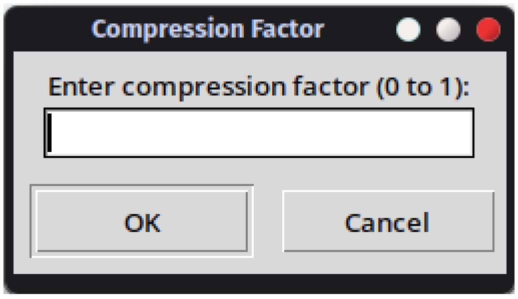
The supported file formats are: "*.wav", "*.mp3", "*.flac", "*.ogg", "*.m4a", "*.aac", "*.wma", "*.aiff", ".aif", ".aifc". For testing I used only ".mp3" files.

```
# Prompt the user to specify the output file location and name
output_file = filedialog.asksaveasfilename(
    title="Save Compressed Audio As",
    defaultextension=".wav",
    filetypes=[("Audio Files", "*.wav *.mp3 *.flac *.ogg *.m4a *.aac *.wma *.aiff *.aif *.aifc")]
)
# Check if a file location was specified
if not output_file:
    messagebox.showerror("Error", "Output file not selected")
    return
```

After selecting the files, a dialog box pops up asking the user for the compression factor. A compression factor of 1 results in no compression taking place, while a compression factor of 0 results in all data being discarded.

```
# Ask the user for the compression factor using a dialog box
compression_factor = simpledialog.askfloat(
    "Compression Factor",
    "Enter compression factor (0 to 1):",
    minvalue=0.0,
    maxvalue=1.0
)

# Check if a valid compression factor was entered
if compression_factor is None:
    messagebox.showerror("Error", "Invalid compression factor")
    return
```



Enter compression factor (0 to 1):

The input file is converted to a numpy array. If the audio is stereo, we only use one channel. After that, we normalize the audio data to a range of [-1, 1], and apply the DCT to the normalized data.

```python
# Load the input audio file
audio = AudioSegment.from_file(input_file)
sample_rate = int(audio.frame_rate)  # Get the sample rate of the audio

# Convert audio data to a numpy array
data = np.array(audio.get_array_of_samples())

# If the audio is stereo, use only one channel
if audio.channels > 1:
    data = data[::audio.channels]

# Normalize audio data to the range [-1, 1]
normalized_data = data / np.max(np.abs(data))

# Apply Discrete Cosine Transform (DCT) to the normalized audio data
dct_transformed = dct(normalized_data, norm='ortho')
```

After applying the DCT, we zero out the coefficients based on the compression factor, and apply IDCT in order to reconstruct the compressed audio. Finally, we denormalize the compressed data back to its original scale.

```python
# Zero out a portion of the DCT coefficients based on the compression factor
n = len(dct_transformed)
compressed_dct = np.copy(dct_transformed)
threshold = int(n * compression_factor)
compressed_dct[threshold:] = 0

# Apply Inverse Discrete Cosine Transform (IDCT) to reconstruct the compressed audio
compressed_audio = idct(compressed_dct, norm='ortho')

# Denormalize the compressed audio data back to its original scale
compressed_audio = np.int16(compressed_audio * np.max(np.abs(data)))
```

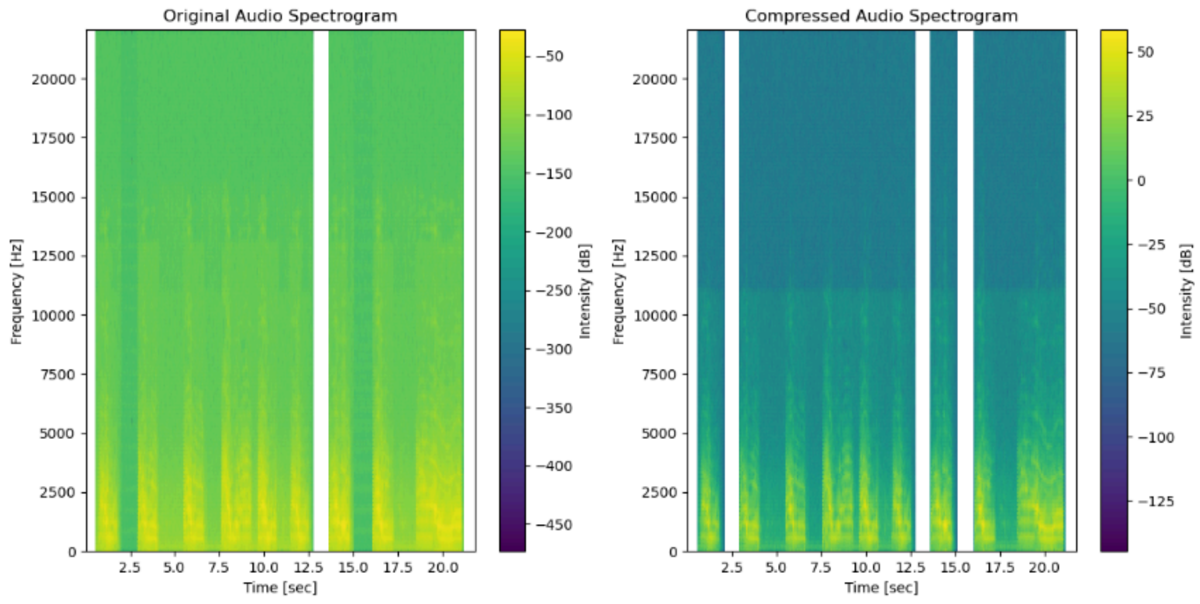
Finally, the output audio is saved in the previously selected file.

```python
# Save the compressed audio file
compressed_audio_segment.export(output_file, format=output_file.split('.')[-1])
messagebox.showinfo("Success", f"Compressed audio saved to {output_file}")
```

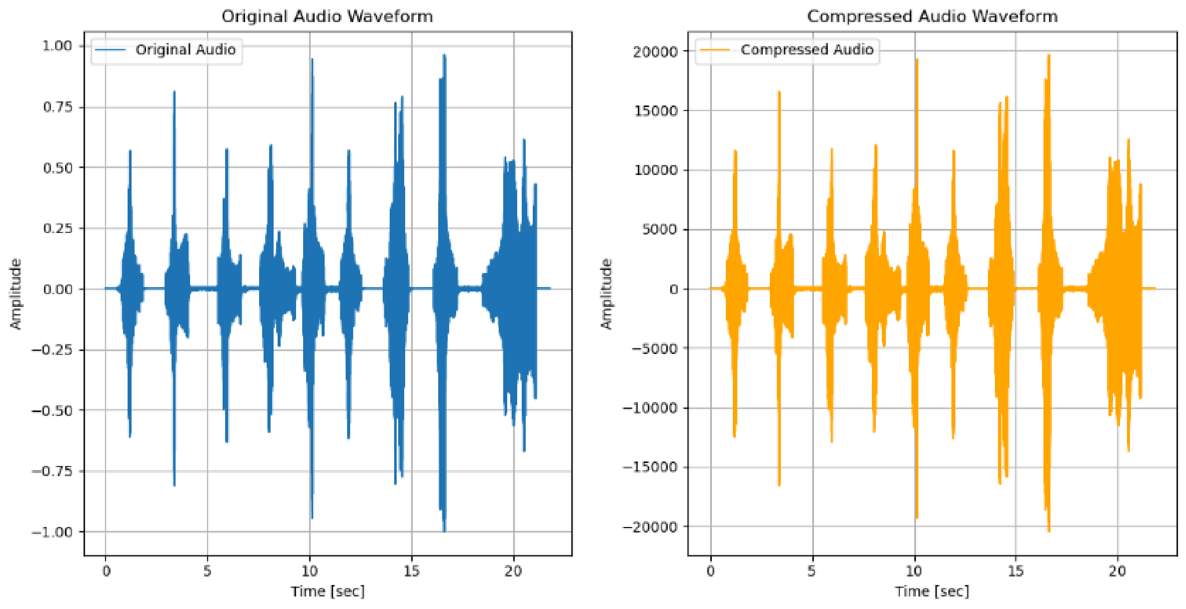
V) Results Analysis

The following graphs are the result of compressing with a compression factor of 0.5.

The spectrogram on the left is the one of the original audio. The second spectrogram shows how on the higher frequencies, the intensity of the sound is significantly reduced, and the areas of the file where the sound was already of low intensity were completely zeroed out.



The waveforms of the sound files show how the audio signal is accurately reconstructed, with minimal loss.



The original audio file had a dimension of 854.2 KiB. After compression, the audio file has a size of 170.8 KiB, which is 19.99% of the dimension of the original size.

VI) Sources & References

The Audio Sample which I used for testing

Wikipedia - About Discrete cosine transform

VII) Source Code

```
import numpy as np
from scipy.fftpack import dct, idct
import tkinter as tk
from tkinter import filedialog, messagebox, simpledialog
from pydub import AudioSegment
import matplotlib.pyplot as plt
from scipy.signal import spectrogram
import os

def select_files():
   # Create and hide the main Tkinter window
   root = tk.Tk()
   root.withdraw()

   # Prompt the user to select an input audio file
   input_file = filedialog.askopenfilename(
       title="Select Input Audio File",
       filetypes=[("Audio Files", "*.wav *.mp3 *.flac *.ogg *.m4a *.aac
*.wma *.aiff *.aif *.aifc")]
   )
   # Check if a file was selected
   if not input_file:
       messagebox.showerror("Error", "Input file not selected")
       return

   # Prompt the user to specify the output file location and name
   output_file = filedialog.asksaveasfilename(
       title="Save Compressed Audio As",
       defaultextension=".wav",
       filetypes=[("Audio Files", "*.wav *.mp3 *.flac *.ogg *.m4a *.aac
*.wma *.aiff *.aif *.aifc")]
   )
   # Check if a file location was specified
   if not output_file:
       messagebox.showerror("Error", "Output file not selected")
```

```
        return

    # Ask the user for the compression factor using a dialog box
    compression_factor = simpledialog.askfloat(
        "Compression Factor",
        "Enter compression factor (0 to 1):",
        minvalue=0.0,
        maxvalue=1.0
    )

    # Check if a valid compression factor was entered
    if compression_factor is None:
        messagebox.showerror("Error", "Invalid compression factor")
        return

    # Call the function to compress audio and handle the output
    compress_audio(input_file, output_file, compression_factor)

def compress_audio(input_file, output_file, compression_factor):
    try:
        # Load the input audio file
        audio = AudioSegment.from_file(input_file)
        sample_rate = int(audio.frame_rate)  # Get the sample rate of
the audio

        # Convert audio data to a numpy array
        data = np.array(audio.get_array_of_samples())

        # If the audio is stereo, use only one channel
        if audio.channels > 1:
            data = data[::audio.channels]

        # Normalize audio data to the range [-1, 1]
        normalized_data = data / np.max(np.abs(data))

        # Apply Discrete Cosine Transform (DCT) to the normalized audio
data
        dct_transformed = dct(normalized_data, norm='ortho')

        # Zero out a portion of the DCT coefficients based on the
compression factor
        n = len(dct_transformed)
        compressed_dct = np.copy(dct_transformed)
        threshold = int(n * compression_factor)
        compressed_dct[threshold:] = 0

        # Apply Inverse Discrete Cosine Transform (IDCT) to reconstruct
the compressed audio
        compressed_audio = idct(compressed_dct, norm='ortho')

        # Denormalize the compressed audio data back to its original
scale
        compressed_audio = np.int16(compressed_audio *
np.max(np.abs(data)))

        # Ensure the output directory exists
        output_dir = os.path.dirname(output_file)
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)

        # Convert the compressed audio data back to an AudioSegment
        compressed_audio_segment = AudioSegment(
            compressed_audio.tobytes(),
            frame_rate=sample_rate,
            sample_width=audio.sample_width,
            channels=1
        )

        # Save the compressed audio file
        compressed_audio_segment.export(output_file,
format=output_file.split('.')[-1])
        messagebox.showinfo("Success", f"Compressed audio saved to
{output_file}")

        # Plot and save the spectrograms and waveforms
        plot_spectrograms_and_waveforms(normalized_data,
compressed_audio, sample_rate, output_dir, compression_factor)
    except Exception as e:
        # Display an error message if something goes wrong
        messagebox.showerror("Error", str(e))

def plot_spectrograms_and_waveforms(original_audio, compressed_audio,
sample_rate, output_dir, compression_factor):
    # Compute the spectrogram for the original and compressed audio
    f1, t1, Sxx1 = spectrogram(original_audio, sample_rate)
    f2, t2, Sxx2 = spectrogram(compressed_audio, sample_rate)
```

```
   # Create a figure with 4 subplots (2 rows x 2 columns)
   plt.figure(figsize=(12, 12))

   # Plot the spectrogram of the original audio
   plt.subplot(2, 2, 1)
   plt.pcolormesh(t1, f1, 10 * np.log10(Sxx1), shading='gouraud')
   plt.title('Original Audio Spectrogram')
   plt.ylabel('Frequency [Hz]')
   plt.xlabel('Time [sec]')
   plt.colorbar(label='Intensity [dB]')

   # Plot the spectrogram of the compressed audio
   plt.subplot(2, 2, 2)
   plt.pcolormesh(t2, f2, 10 * np.log10(Sxx2), shading='gouraud')
   plt.title('Compressed Audio Spectrogram')
   plt.ylabel('Frequency [Hz]')
   plt.xlabel('Time [sec]')
   plt.colorbar(label='Intensity [dB]')

   # Plot the waveform of the original audio
   plt.subplot(2, 2, 3)
   plt.plot(np.arange(len(original_audio)) / sample_rate,
original_audio, label='Original Audio')
   plt.title('Original Audio Waveform')
   plt.xlabel('Time [sec]')
   plt.ylabel('Amplitude')
   plt.grid()
   plt.legend()

   # Plot the waveform of the compressed audio
   plt.subplot(2, 2, 4)
   plt.plot(np.arange(len(compressed_audio)) / sample_rate,
compressed_audio, label='Compressed Audio', color='orange')
   plt.title('Compressed Audio Waveform')
   plt.xlabel('Time [sec]')
   plt.ylabel('Amplitude')
   plt.grid()
   plt.legend()

   # Save the plot as a PNG file
   output_path = os.path.join(output_dir,
f'spectrogram_waveform_{compression_factor}.png')
```

```
    plt.tight_layout()
    plt.savefig(output_path)
    print(f"Saved plot to {output_path}")

    # Display the plot
    plt.show()

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