Aproximarea unei melodii folosind mai multe metode de calcul numeric

Popa Mircea Alexandru
Sîrghe Matei Ștefan
Ungureanu Robert Afton

June 1, 2025

1 Modelul matematic

- Modelul ales de noi începe de la orice fișier audio de tip .wav, 16 bitrate, mono.

- Fișierul audio poate să fie interpretat ca o amplitudine în funcție de timp. Astfel, putem să ne imaginăm că avem o funcție de tipul $f(t)$, unde t este timpul și $f(t)$ este amplitudinea sunetului la acel moment.

- Ca să interpretăm funcția cu acuratețe mai mare, vom aplica Short-Time Fourier Transform, care ne va permite să vedem frecvențele care compun sunetul.

- Astfel o sa ajungem cu o functie $F(\tau, \omega)$ pe care o împărțim în matricea A care conține modulul părții reale și matricea φ care conține coeficientul de unghi.

- Vom aplica SVD (Singular Value Decomposition) pe matricea $F(\tau, \omega)$. Factorizarea QR va fi folosită ca să descompună A în valori proprii.

- Am aplicat SVD, vom obține matricea $A \approx U\Sigma V^T$, unde U și V sunt matricile ortogonale și Σ este o matrice diagonală cu valorile pozitive.

- Inmulțim cele 3 matrici, obținem o aproximare la matricea originală B care este interpretată ca funcția $G(\tau, \omega)$.

- Cu funcția $G(\tau, \omega)$ vom calcula Short-Time Fourier Transform inversă, care ne va permite să obținem o nouă funcție $g(t)$, aproximarea funcției originale $f(t)$.

- Cu aproximarea $g(t)$ vom putea să obținem un nou fișier audio .wav, care este o aproximare a fișierului original.

$$\begin{cases} F(\tau, \omega) = \Sigma_{t=0}^{N-1} f(t) w(t-\tau) e^{-i\omega t} \\ Z_{i,j} = F(i,j), Z \in \mathbb{M}_{N,M}(\mathbb{C}) \\ A = |Z|, A \in \mathbb{M}_{N,M}(\mathbb{R}) \\ \forall z_{n,m} = a_{n,m} + ib_{n,m} \in Z, n \in [0,N] m \in [0,M], \exists \theta_{n,m} = \tan^{-1}(\dfrac{b_{n,m}}{a_{n,m}}) \\ \varphi = \begin{bmatrix} \theta_{1,1} & \theta_{1,2} & ... & \theta_{1,N} \\ \theta_{2,1} & \theta_{2,2} & ... & \theta_{2,N} \\ ... & ... & ... & ... \\ \theta_{M,1} & \theta_{M,2} & ... & \theta_{M,N} \end{bmatrix} \\ A \approx U\Sigma V^* = \Sigma_{i=1}^{k} \sigma_i u_i v_i^* = B \\ G(i,j) = B_{i,j} + \varphi_{i,j}, B \in \mathbb{M}_{N,M}(\mathbb{C}) \\ g(t) = \frac{1}{w(t-\tau)} \frac{1}{2\pi} \Sigma_{\omega=0}^{N-1} G(\tau, \omega) e^{+i\omega t} \end{cases} \tag{1}$$

1.1 Discretizarea domeniului

Domeniul este reprezentat de catre durata melodiei care este înmultită cu sampling rate-ul de 44100 Hz.

$$N = l \times h \tag{2}$$

- l este durata melodiei în secunde.
- h este sampling rate-ul default de 44100 Hz.
- M este numărul de frecvențe pe care îl conține o melodie.

1.2 Transformata Fourier

Teoretic vorbind, Short-Time Fourier Transform este:

$$F(\tau, \omega) = \int_{-\inf}^{+\inf} f(t) w(t - \tau) e^{-i\omega t} dt \tag{3}$$

dar funcția $f(t)$ este discretizată, deci vom folosi transformarea discretă ca să calculăm $F(\tau, \omega)$.

$$F(\tau, \omega) = \Sigma_{t=0}^{N-1} f(t) w(t - \tau) e^{-i\omega t} \tag{4}$$

Vom avea matricea care va fi și spectograma originală a funcției $f(t)$:

$$A = \begin{bmatrix} |F(\tau_1, \omega_1)| & |F(\tau_1, \omega_2)| & ... & |F(\tau_1, \omega_M)| \\ |F(\tau_2, \omega_1)| & |F(\tau_2, \omega_2)| & ... & |F(\tau_2, \omega_M)| \\ ... & ... & ... & ... \\ |F(\tau_N, \omega_1)| & |F(\tau_N, \omega_2)| & ... & |F(\tau_N, \omega_M)| \end{bmatrix} \tag{5}$$

Cu partea imaginară a funcției $F(\tau, \omega)$ vom crea matricea φ în acest fel:

$$\varphi = \begin{bmatrix} \tan^{-1}(\frac{b_{1,1}}{a_{1,1}}) & \tan^{-1}(\frac{b_{1,2}}{a_{1,2}}) & ... & \tan^{-1}(\frac{b_{1,M}}{a_{1,M}}) \\ \tan^{-1}(\frac{b_{2,1}}{a_{2,1}}) & \tan^{-1}(\frac{b_{2,2}}{a_{2,2}}) & ... & \tan^{-1}(\frac{b_{2,M}}{a_{2,M}}) \\ ... & ... & ... & ... \\ \tan^{-1}(\frac{b_{N,1}}{a_{N,1}}) & \tan^{-1}(\frac{b_{N,2}}{a_{N,2}}) & ... & \tan^{-1}(\frac{b_{N,M}}{a_{N,M}}) \end{bmatrix} \tag{6}$$

1.3 Sistemului Liniar

Teoretic vorbind, factorizarea SVD este:

$$A = U\Sigma V^T = \Sigma_{i=1}^{k} \sigma_i u_i v_i^* \tag{7}$$

dar o să avem eroare cauzată de floating point, deci vom obține o matrice aproximativă B care este diferită de cea de la care am pornit.

$$B = U\Sigma V^T \approx A \tag{8}$$

Ca să calculam U și V, vom aplica factorizarea QR. Obținem $Vx = U^T b$ iar $A = UV$. Utilizând metoda substituției descendente putem rezolva acest sistem în $O(n^2)$ pași.

$$\begin{cases} A^T Ax = A^T b \\ (UV)^T UVx = (UV)^T b \\ V^T U^T UVx = V^T U^T b \\ (V^T)^{-1} V^T Vx = V^T U^T b \\ Vx = U^T b \end{cases} \tag{9}$$

Pentru matricea Σ valorile proprii ale matricei $A^T A$ se folosesc pentru a determina valorile singulare σ_i. Acestea sunt calculate astfel:

$$\sigma_i = \sqrt{\lambda_i}, \quad \text{unde } \lambda_i \text{ sunt valorile proprii ale } A^T A \tag{10}$$

Matricea Σ este o matrice diagonală de forma unde $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq 0$, iar $k = \min(N, M)$.

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_k \end{bmatrix} \tag{11}$$

Putem să creăm o funcție $G(\tau, \omega)$ care primeste valorile din matricea B și care este definită ca:

$$G(\tau, \omega) = \begin{cases} B_{i,j} & \text{dacă } i \leq N \text{ și } j \leq M \\ 0 & \text{altfel} \end{cases} \tag{12}$$

Utilizând noua funcție, putem crea o nouă spectogramă care o să ne arate eroarea de aproximare pe care o avem:

$$S_e = \begin{bmatrix} |F(\tau_1,\omega_M) - G(\tau_1,\omega_M)| & |F(\tau_2,\omega_M) - G(\tau_2,\omega_M)| & ... & |F(\tau_N,\omega_M) - G(\tau_N,\omega_M)| \\ ... & ... & ... & ... \\ |F(\tau_1,\omega_2) - G(\tau_1,\omega_2)| & |F(\tau_2,\omega_2) - G(\tau_2,\omega_2)| & ... & |F(\tau_N,\omega_2) - G(\tau_N,\omega_2)| \\ |F(\tau_1,\omega_1) - G(\tau_1,\omega_1)| & |F(\tau_2,\omega_M) - G(\tau_2,\omega_M)| & ... & |F(\tau_N,\omega_1) - G(\tau_N,\omega_1)| \end{bmatrix} \tag{13}$$

1.4 Transformata Fourier Inversă

Calculam aproximarea funcției originale $f(t)$ folosind Short-Time Fourier Transform inversă.

Teoretic vorbind, Short-Time Fourier Transform inversă este:

$$g(t) = \frac{1}{w(t-\tau)}\frac{1}{2\pi}\int_{-\inf}^{+\inf} G(\tau,\omega)e^{+i\omega t}d\omega \tag{14}$$

dar funcția $G(\tau,\omega)$ este discretizată, deci vom folosi trasnformata discretă ca să calculăm $g(t)$.

$$g(t) = \frac{1}{w(t-\tau)}\frac{1}{2\pi}\Sigma_{\omega=0}^{N-1}G(\tau,\omega)e^{+i\omega t} \tag{15}$$

Cu funcția $g(t)$ vom creea o nouă funcție care reprezintă eroarea la un anumit timp f_{ea} și funcția f_{cea} care reprezină eroarea cumulată a aproximării.

$$f_{cea} = \Sigma_{t=0}^{N-1}|f(t) - g(t)| \tag{16}$$

$$f_{ea} = |f(t) - g(t)| \tag{17}$$

2 Explicații cod

Funcția care calculează Fast Fourier Transform. Aceasta o să fie apelată de funcția Short-time Fourier Transform.

```
def my_fft_recursiv(x):
    #Algoritmul Cooley-Tukey recursiv, Radix-2
```

```python
    #Inputul trebuie sa fie len(x) = k^2

    N = len(x)
    if N == 1:
        return x
    elif N % 2 != 0:
        raise ValueError("Cum naiba ai intrat cu o valoare cu un length impar")

    x_even = my_fft_recursiv(x[::2])
    x_odd = my_fft_recursiv(x[1::2])

    factor = np.exp(-2j * np.pi * np.arange(N) / N)
    return np.concatenate([
        x_even + factor[:N//2] * x_odd,
        x_even - factor[:N//2] * x_odd
    ])
```

Funcția Short-time Fourier Transform care utilizează funcția window de tip Hann:

```python
def my_stft(signal, fft_size=1024, hop_size=512, window_fn=np.hanning):
    #SHort-Time Fourier Transform (STFT) al unui semnal audio cu windowing de tip Hann
    window = window_fn(fft_size)
    num_frames = 1 + (len(signal) - fft_size) // hop_size
    #// -> floor division, divizie care da round la rezultat
    stft_matrix = np.empty((num_frames, fft_size), dtype=np.complex64)

    for i in range(num_frames):
        start = i * hop_size
        frame = signal[start:start+fft_size] * window
        stft_matrix[i, :] = my_fft_recursiv(frame)

    return stft_matrix
```

Funcția inversă Fast Fourier Transform care va fi folosită în funcția inversă pentru Short-time Fourier Transform (Împreuna pentru simplitate):

```python
def my_ifft_recursiv(x):
    x_conj = np.conj(x)
    x = my_fft_recursiv(x_conj)
    return np.conj(x) / len(x)

def my_istft(stft_matrix, fft_size=1024, hop_size=512, window_fn=np.hanning):
    window = window_fn(fft_size)
    num_frames = stft_matrix.shape[0]
    length_signal = fft_size + hop_size * (num_frames -1)
    signal = np.zeros(length_signal)
    window_sum = np.zeros(length_signal)

    for i in range(num_frames):
        start = i * hop_size
        frame_time = np.real(my_ifft_recursiv(stft_matrix[i, :]))
        signal[start:start+fft_size] += frame_time * window
        window_sum[start:start+fft_size] += window ** 2

    window_sum[window_sum < 1e-10] = 1e-10
    signal /= window_sum

    return signal
```

Funcția care dă plot la spectogramă:

```python
def plot_spectrogram(magnitude, sr, title, filename=None):
    plt.figure(figsize=(12, 6))
    db_spec = librosa.amplitude_to_db(np.abs(magnitude), ref=np.max)
```

```
    librosa.display.specshow(db_spec, sr=sr, x_axis='
       time', y_axis='log', cmap='viridis')
    plt.colorbar(format='%+2.0f dB')
    plt.title(title)
    plt.tight_layout()
    if filename:
        plt.savefig(filename, dpi=150)
    else:
        plt.show()
    plt.close()
```

Plotează funcțiile care arată eroarea absolută, eroarea cumulativă. În același timp, compară funcția originală cu cea aproximată în același grafic:

```
def plot_waveform_comparison(original, reconstructed,
   sr, channel_name, filename=None):
    min_len = min(len(original), len(reconstructed))
    original = original[:min_len]
    reconstructed = reconstructed[:min_len]
    time = np.arange(min_len) / sr

    plt.figure(figsize=(14, 12))


    plt.subplot(4, 1, 1)
    plt.plot(time, original, 'b-', alpha=0.7, label='
       Original')
    plt.plot(time, reconstructed, 'r-', alpha=0.5,
       label='Reconstructed')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.title(f'Comparatie Waveform - {channel_name}
       Channel (Full Duration)')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.6)

    # Eroarea absoluta
    plt.subplot(4, 1, 2)
    amplitude_diff = np.abs(original - reconstructed)
```

```python
    plt.plot(time, amplitude_diff, 'g-')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude Difference')
    plt.title('Absolute Error')
    plt.grid(True, linestyle='--', alpha=0.6)

    # Eroarea cumulativa
    plt.subplot(4, 1, 3)
    cumulative_error = np.cumsum(amplitude_diff)
    plt.plot(time, cumulative_error, 'm-')
    plt.xlabel('Time (s)')
    plt.ylabel('Error')
    plt.title('Cumulative Error')
    plt.grid(True, linestyle='--', alpha=0.6)

    # Zoomed-in
    zoom_duration = 0.2 #secunde
    center_idx = min_len // 2
    zoom_samples = int(zoom_duration * sr)
    start = max(center_idx - zoom_samples // 2, 0)
    end = min(center_idx + zoom_samples // 2, min_len)
    zoom_time = time[start:end]

    plt.subplot(4, 1, 4)
    plt.plot(zoom_time, original[start:end], 'b-',
       label='Original')
    plt.plot(zoom_time, reconstructed[start:end], 'r-'
       , alpha=0.6, label='Reconstructed')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.title(f'Zoomed-In Comparison ({zoom_duration
       :.1f}s window around midpoint)')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.6)

    plt.tight_layout()
    if filename:
        plt.savefig(filename, dpi=150)
```

```
    else:
        plt.show()
    plt.close()
```

Griffin Lim folosește prima matrice și ghicește faza făcând Transformări Fourier repetitive. În mod normal, în codul nostru nu este activată:

```
def griffin_lim(magnitude, n_iter=50, window='hann', n_fft=2048, hop_length=None):
    #Griffin_lim :
    if hop_length is None:
        hop_length = n_fft // 4

    angles = np.exp(2j * np.pi * np.random.rand(*magnitude.shape))
    stft = magnitude * angles

    for _ in range(n_iter):
        if use_librosa_transforms:
            audio = librosa.istft(stft, hop_length=hop_length, window=window)
            stft = librosa.stft(audio, n_fft=n_fft, hop_length=hop_length, window=window)
        else:
            audio = my_istft(stft)
            stft = my_stft(audio)

        angles = np.exp(1j * np.angle(stft))
        stft = magnitude * angles

    if use_librosa_transforms:
        audio = librosa.istft(stft, hop_length=hop_length, window=window)
    else:
        audio = my_istft(stft)
    return audio
```

Funcția Optimizată QR

- Input: Matrice A reala mn

- Output : Matrice ortogonala Q mm
- Matrice R triunghiular superioara
- Deoarece funcția optimizată QR va fi apelată de funcția descompunerii în valori și vectori proprii, am aplicat vectorizarea în calculul produsului scalar și a celui exterior. Operațiile vectorizate sunt mai rapide deoarece limbajul Python se folosește de cod pre-compilat optimizat care calculează operații matematice pe secvente de date. (In loc de un for-loop scris in Python)

```
def optimised_QR(A):
    n = A.shape[1]
    Q = A.copy().astype(np.float32)
    R = np.zeros((n, n), dtype=np.float32)

    for i in range(n):
        norm = np.linalg.norm(Q[:, i])
        if norm < 1e-10: #evitam vectori cu zero
            R[i, i] = 0.0
            continue

        R[i, i] = norm
        Q[:, i] /= R[i, i]

        #Optimizare prin vectorizarea operatiilor, should be a lil better
        if i < n - 1:
            R[i, i+1:n] = Q[:, i] @ Q[:, i+1:n] # Produs scalar simultan pt j > i
            Q[:, i+1:n] -= np.outer(Q[:, i], R[i, i+1:n]) #Produsul exterior tot intr-o singura operatie

    return Q, R
```

Funcția Eigen Decomposition

- Input: Matrice A

- Output : eigenvalues, eigenvectors
- Această funcție se folosește de factorizarea QR pentru a descompune o matrice în valorile și vectorii săi proprii. O primă optimizare implementată este verificarea convergenței matricii B la o matrice diagonală. Acest lucru se face prin următorul procedeu:
- 1. Calculăm matricea cu elementele care nu sunt pe diagonală
- 2. off-diag= $np.abs(B - np.diag(np.diag(B)))$
- $np.diag(np.diag(B))$ construiește o matrice care are doar diagonala lui B, restul de elemente sunt 0
- B - $np.diag(np.diag(B))$ = Matrice cu elementele din afara diagonalei. Cu np.abs luăm valorile maxime care nu aparțin diagonalei. Dintre acestea alegem valoarea cea mai mare.
- 3. Salvăm cea mai mare valoare în max-off. Dacă aceasta este mai mică decât toleranța predefinită (ex. $10^{-}8$), atunci considerăm că matricea a convers.
- Această verificare a convergenței ne permite să ieșim din funcție înaintea terminării numărului de iterații maxime, salvând timp.

```python
def eigen_decomp_optimised(A, max_iter=1000, tolerance=1e-8):
    B = A.T @ A
    n = B.shape[0]
    V = np.eye(n, dtype=np.float32)

    pbar = tqdm(total=max_iter, desc="Eigen Decomp")

    for i in range(max_iter):
        Q, R = optimised_QR(B)
        B = R @ Q
        V = V @ Q

        off_diag = np.abs(B - np.diag(np.diag(B))) #Pt convergenta
```

```
        max_off = np.max(off_diag)

        pbar.update(1)

        if max_off < tolerance:
            pbar.set_description(f"Converged la {i+1}
               iteratii")
            break

    pbar.close()
    eigenvalues = np.diag(B)
    eigenvectors = V
    return eigenvalues, eigenvectors
```

Funcția care rezolvă SVD

- Input: Matricea spectogramă, opțional o valoare k care determină numărul de iterații SVD
- Output: Cele trei array-ul, U, Sigma(în funcție numită singular-values), și V transpus
- Algoritmul urmărește clasica variantă teoretică de SVD. Apelează funcțiile de descompunere a valorilor proprii, sortează vectorul de valori în ordine descrescătoare
- Dacă variabila $k = -1$ atunci codul va calcula o valoare automata potrivita pentru SVD, altfel putem seta propria valoare. (Cu atât mai mica cu atât aproximarea este mai slabă. k<25 => apar artefacte auzibile)
- Matricea U este calculată în paralel folosind librăria joblib

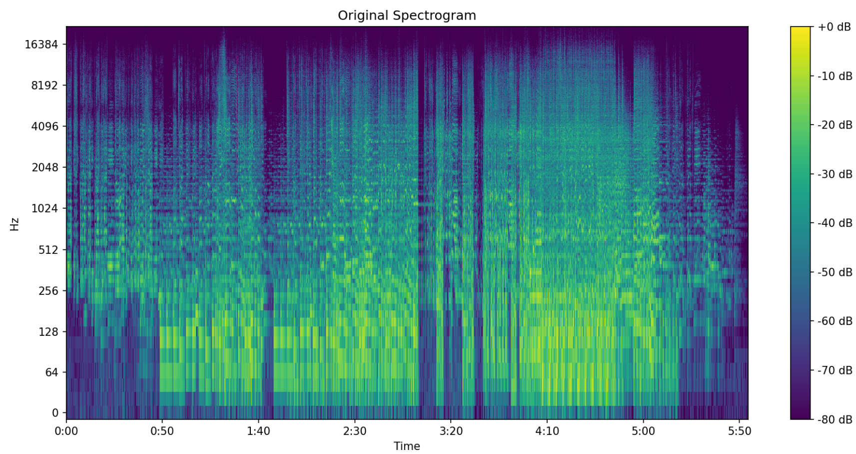
```
def solve_SVD(spect, k=100):
    #Cod neoptimizat de SVD. Foarte incet
    def compute_eigenvectors():
        #lamda, v = eigen_decomp(spect)
        lamda, v = eigen_decomp_optimised(spect)
        return lamda, v

```

```python
    lamda, v = compute_eigenvectors()
    idx = np.argsort(-lamda)
    lamda = lamda[idx]
    v = v[:, idx] #sortam descrescator

    singular_values = np.sqrt(np.maximum(lamda, 0))

    # Daca k nu e specificat vom calcula automat pe baza "energiei", adica suma a elementelor matricei cu valori singulare (sigma) la puterea a doua.
    if k == -1:
        energy = np.cumsum(singular_values**2)
        k = np.searchsorted(energy, 0.95 * energy[-1]) + 1 #95% din valorile din matricea sigma. Teoretic acest approach poate scoate noise si alte elemente irelevante din matrice

    k = min(k, len(singular_values))
    print(f"{k} Valori proprii")

    def compute_u(i):
        if singular_values[i] > 1e-10:
            return (spect @ v[:, i]) / singular_values[i]
        return np.zeros(spect.shape[0])

    U = np.column_stack(Parallel(n_jobs=-1)(delayed(compute_u)(i) for i in tqdm(range(k), desc="Computing U")))

    return U, singular_values[:k], v[:, :k].T
```

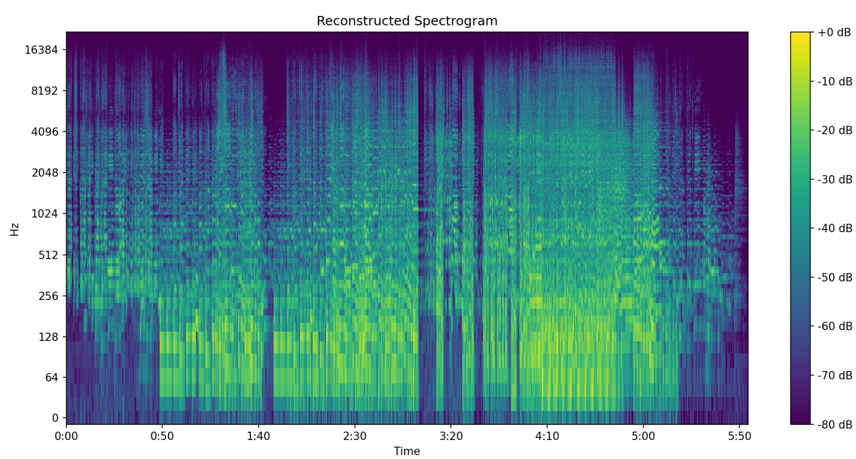
3 Grafice cu aproximarea soluției

3.1 Spectograma originală

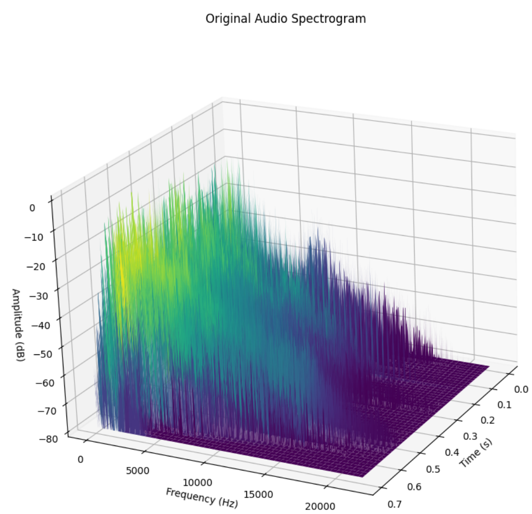
Am ales ca exemplu melodia Bohemian Rhapsody de Queen și următoarele grafice sunt create de la aceasta:



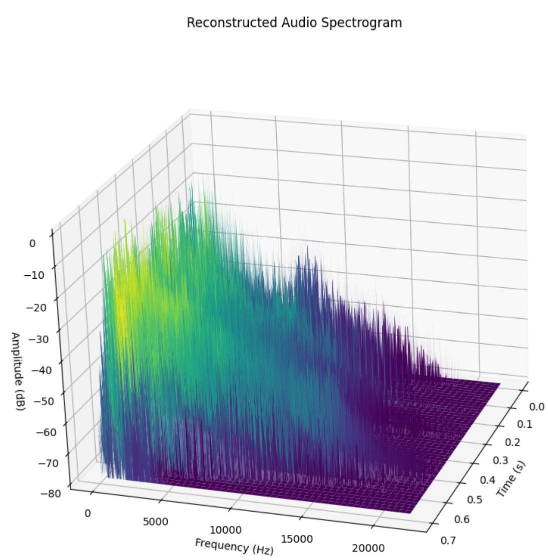
3.2 Spectograma aproximată



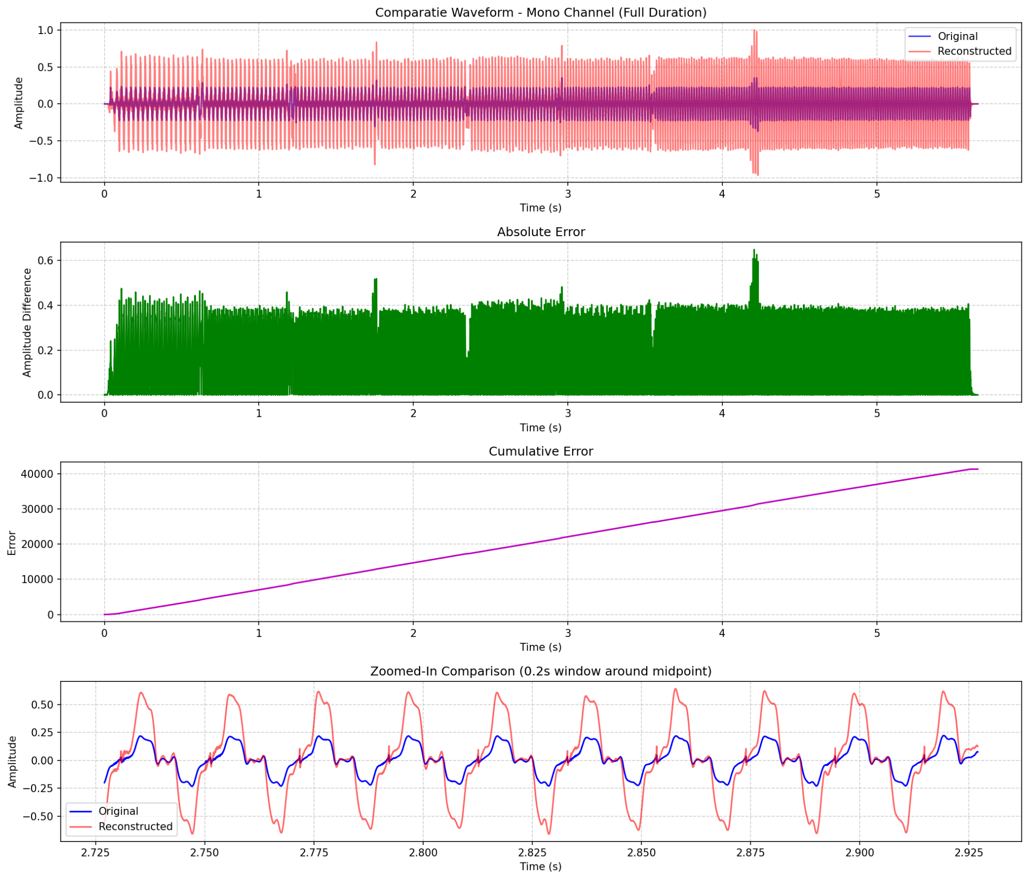
3.3 Spectograma originală 3D



3.4 Spectograma aproximată 3D



3.5 Grafic erori



4 Bibliografia

- Vectorized Numpy Operations
- Cooley–Tukey FFT Algorithm
- Curs CN - UNIBUC
- SVD - Wikipedia
- SVD Audio Compression - GitHub

- Spectrogram - Wikipedia
- STFT - Wikipedia
- Hann and Hamming Windows
- Ce a fost dezvăluit lui Robert cât timp visa