

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 bi-rate, 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 să ajungem cu o funcție $F(\tau, \omega)$ 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.
- Înmulțim 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.

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

1.1 Discretizarea domeniului

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

$$N = l \times h \quad (2)$$

- 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:

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

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

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

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

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

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

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

1.3 Sistemului Liniar

Teoretic vorbind, factorizarea SVD este:

$$A = U\Sigma V^T = \sum_{i=1}^k \sigma_i u_i v_i^* \quad (7)$$

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.

$$B = U\Sigma V^T \approx A \quad (8)$$

Ca să calculăm 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.

$$\begin{cases} A^T A x = A^T b \\ (UV)^T UV x = (UV)^T b \\ V^T U^T UV x = V^T U^T b \\ (V^T)^{-1} V^T V x = V^T U^T b \\ V x = U^T b \end{cases} \quad (9)$$

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

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

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

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

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

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

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

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

1.4 Transformata Fourier Inversă

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

Teoretic vorbind, Short-Time Fourier Transform inversă este:

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

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

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

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

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

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

2 Explicații cod

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

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

```

3      #Inputul trebuie sa fie len(x) = k^2
4
5      N = len(x)
6      if N == 1:
7          return x
8      elif N % 2 != 0:
9          raise ValueError("Cum naiba ai intrat cu o
10             valoare cu un length impar")
11
12      x_even = my_fft_recurziv(x[::2])
13      x_odd = my_fft_recurziv(x[1::2])
14
15      factor = np.exp(-2j * np.pi * np.arange(N) / N)
16      return np.concatenate([
17          x_even + factor[:N//2] * x_odd,
18          x_even - factor[:N//2] * x_odd
19      ])

```

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

```

1  def my_stft(signal, fft_size=1024, hop_size=512,
2      window_fn=np.hanning):
3      #Short-Time Fourier Transform (STFT) al unui
4      semnal audio cu windowing de tip Hann
5      window = window_fn(fft_size)
6      num_frames = 1 + (len(signal) - fft_size) //
7          hop_size
8      ///< -> floor division, divizie care da round la
9      rezultat
10     stft_matrix = np.empty((num_frames, fft_size),
11         dtype=np.complex64)
12
13     for i in range(num_frames):
14         start = i * hop_size
15         frame = signal[start:start+fft_size] * window
16         stft_matrix[i, :] = my_fft_recurziv(frame)
17
18     return stft_matrix

```

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

```
1 def my_ifft_recurziv(x):
2     x_conj = np.conj(x)
3     x = my_fft_recurziv(x_conj)
4     return np.conj(x) / len(x)
5
6 def my_istft(stft_matrix, fft_size=1024, hop_size=512,
7             window_fn=np.hanning):
8     window = window_fn(fft_size)
9     num_frames = stft_matrix.shape[0]
10    length_signal = fft_size + hop_size * (num_frames
11        -1)
12    signal = np.zeros(length_signal)
13    window_sum = np.zeros(length_signal)
14
15    for i in range(num_frames):
16        start = i * hop_size
17        frame_time = np.real(my_ifft_recurziv(
18            stft_matrix[i, :]))
19        signal[start:start+fft_size] += frame_time *
20            window
21        window_sum[start:start+fft_size] += window **
22            2
23
24    window_sum[window_sum < 1e-10] = 1e-10
25    signal /= window_sum
26
27    return signal
```

Funcția care dă plot la spectrogramă:

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

```

4 librosa.display.specshow(db_spec, sr=sr, x_axis='
    time', y_axis='log', cmap='viridis')
5 plt.colorbar(format='%+2.0f dB')
6 plt.title(title)
7 plt.tight_layout()
8 if filename:
9     plt.savefig(filename, dpi=150)
10 else:
11     plt.show()
12 plt.close()

```

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

```

1 def plot_waveform_comparison(original, reconstructed,
2   sr, channel_name, filename=None):
3     min_len = min(len(original), len(reconstructed))
4     original = original[:min_len]
5     reconstructed = reconstructed[:min_len]
6     time = np.arange(min_len) / sr
7
8     plt.figure(figsize=(14, 12))
9
10    plt.subplot(4, 1, 1)
11    plt.plot(time, original, 'b-', alpha=0.7, label='
        Original')
12    plt.plot(time, reconstructed, 'r-', alpha=0.5,
13             label='Reconstructed')
14    plt.xlabel('Time (s)')
15    plt.ylabel('Amplitude')
16    plt.title(f'Comparatie Waveform - {channel_name}
17             Channel (Full Duration)')
18    plt.legend()
19    plt.grid(True, linestyle='--', alpha=0.6)
20
21    # Eroarea absoluta
22    plt.subplot(4, 1, 2)
23    amplitude_diff = np.abs(original - reconstructed)

```



```

22     plt.plot(time, amplitude_diff, 'g-')
23     plt.xlabel('Time (s)')
24     plt.ylabel('Amplitude Difference')
25     plt.title('Absolute Error')
26     plt.grid(True, linestyle='--', alpha=0.6)
27
28     # Eroarea cumulativa
29     plt.subplot(4, 1, 3)
30     cumulative_error = np.cumsum(amplitude_diff)
31     plt.plot(time, cumulative_error, 'm-')
32     plt.xlabel('Time (s)')
33     plt.ylabel('Error')
34     plt.title('Cumulative Error')
35     plt.grid(True, linestyle='--', alpha=0.6)
36
37     # Zoomed-in
38     zoom_duration = 0.2 #secunde
39     center_idx = min_len // 2
40     zoom_samples = int(zoom_duration * sr)
41     start = max(center_idx - zoom_samples // 2, 0)
42     end = min(center_idx + zoom_samples // 2, min_len)
43     zoom_time = time[start:end]
44
45     plt.subplot(4, 1, 4)
46     plt.plot(zoom_time, original[start:end], 'b-',
47             label='Original')
48     plt.plot(zoom_time, reconstructed[start:end], 'r-',
49             , alpha=0.6, label='Reconstructed')
50     plt.xlabel('Time (s)')
51     plt.ylabel('Amplitude')
52     plt.title(f'Zoomed-In Comparison ({zoom_duration
53             :.1f}s window around midpoint)')
54     plt.legend()
55     plt.grid(True, linestyle='--', alpha=0.6)
56
57     plt.tight_layout()
58     if filename:
59         plt.savefig(filename, dpi=150)

```

```

57     else:
58         plt.show()
59     plt.close()

```

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

```

1  def griffin_lim(magnitude, n_iter=50, window='hann',
2      n_fft=2048, hop_length=None):
3      #Griffin_lim :
4      if hop_length is None:
5          hop_length = n_fft // 4
6
7      angles = np.exp(2j * np.pi * np.random.rand(*
8          magnitude.shape))
9      stft = magnitude * angles
10
11     for _ in range(n_iter):
12         if use_librosa_transforms:
13             audio = librosa.istft(stft, hop_length=
14                 hop_length, window=window)
15             stft = librosa.stft(audio, n_fft=n_fft,
16                 hop_length=hop_length, window=window)
17         else:
18             audio = my_istft(stft)
19             stft = my_stft(audio)
20
21         angles = np.exp(1j * np.angle(stft))
22         stft = magnitude * angles
23
24     if use_librosa_transforms:
25         audio = librosa.istft(stft, hop_length=
26             hop_length, window=window)
27     else:
28         audio = my_istft(stft)
29     return audio

```

Funcția Optimizată QR

- Input: Matrice A reala mn

- Output : Matrice ortogonală Q mm
- Matrice R triunghiular superioară
- 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 secvențe de date. (În loc de un for-loop scris în Python)

```

1 def optimised_QR(A):
2     n = A.shape[1]
3     Q = A.copy().astype(np.float32)
4     R = np.zeros((n, n), dtype=np.float32)
5
6     for i in range(n):
7         norm = np.linalg.norm(Q[:, i])
8         if norm < 1e-10: #evitam vectori cu zero
9             R[i, i] = 0.0
10            continue
11
12        R[i, i] = norm
13        Q[:, i] /= R[i, i]
14
15        #Optimizare prin vectorizarea operatiilor,
16        should be a lil better
17        if i < n - 1:
18            R[i, i+1:n] = Q[:, i] @ Q[:, i+1:n] #
19            Produs scalar simultan pt j > i
20            Q[:, i+1:n] -= np.outer(Q[:, i], R[i, i+1:
21            n]) #Produsul exterior tot intr-o
22            singura operatie
23
24    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. $\text{off-diag} = \text{np.abs}(B - \text{np.diag}(\text{np.diag}(B)))$
 3. $\text{np.diag}(\text{np.diag}(B))$ construiește o matrice care are doar diagonală lui B, restul de elemente sunt 0
 4. $B - \text{np.diag}(\text{np.diag}(B))$ = Matrice cu elementele din afara diagonalei. Cu np.abs luăm valorile maxime care nu aparțin diagonalei. Dintre acestea alegem valoarea cea mai mare.
 5. Salvăm cea mai mare valoare în max-off . Dacă aceasta este mai mică decât toleranța predefinită (ex. 10^{-8}), atunci considerăm că matricea a converș.
- Această verificare a convergenței ne permite să ieșim din funcție înaintea terminării numărului de iterații maxime, salvând timp.

```

1 def eigen_decomp_optimised(A, max_iter=1000, tolerance
  =1e-8):
2     B = A.T @ A
3     n = B.shape[0]
4     V = np.eye(n, dtype=np.float32)
5
6     pbar = tqdm(total=max_iter, desc="Eigen Decomp")
7
8     for i in range(max_iter):
9         Q, R = optimised_QR(B)
10        B = R @ Q
11        V = V @ Q
12
13        off_diag = np.abs(B - np.diag(np.diag(B))) #Pt
        convergenta

```

```

14         max_off = np.max(off_diag)
15
16         pbar.update(1)
17
18         if max_off < tolerance:
19             pbar.set_description(f"Converged la {i+1}
20                                 iteratii")
21             break
22
23     pbar.close()
24     eigenvalues = np.diag(B)
25     eigenvectors = V
26     return eigenvalues, eigenvectors

```

Funcția care rezolvă SVD

- Input: Matricea spectrogramă, opțional o valoare k care determină numărul de iterații SVD
- Output: Cele trei array-uri, U, Sigma (în funcție numită singular-values), și V transpus
- Algoritmul urmărește clasică 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 automată potrivită pentru SVD, altfel putem seta propria valoare. (Cu atât mai mica cu atât aproximarea este mai slabă. $k < 25 \Rightarrow$ apar artefacte auzibile)
- Matricea U este calculată în paralel folosind librăria joblib

```

1 def solve_SVD(spect, k=100):
2     #Cod neoptimizat de SVD. Foarte incet
3     def compute_eigenvectors():
4         #lamda, v = eigen_decomp(spect)
5         lamda, v = eigen_decomp_optimised(spect)
6         return lamda, v
7

```

```

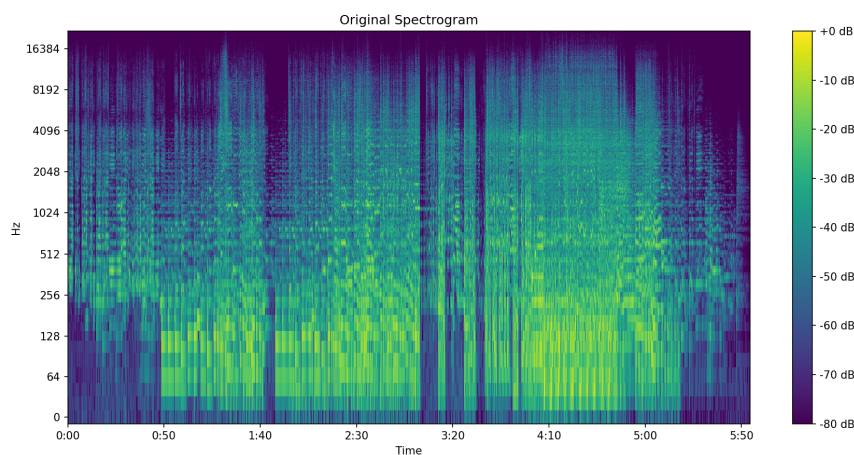
8     lamda, v = compute_eigenvectors()
9     idx = np.argsort(-lamda)
10    lamda = lamda[idx]
11    v = v[:, idx] #sortam descrescator
12
13    singular_values = np.sqrt(np.maximum(lamda, 0))
14
15    # Daca k nu e specificat vom calcula automat pe
baza "energiei", adica suma a elementelor
matricei cu valori singulare (sigma) la puterea
a doua.
16    if k == -1:
17        energy = np.cumsum(singular_values**2)
18        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
19
20    k = min(k, len(singular_values))
21    print(f"{k} Valori proprii")
22
23    def compute_u(i):
24        if singular_values[i] > 1e-10:
25            return (spect @ v[:, i]) / singular_values
                [i]
26        return np.zeros(spect.shape[0])
27
28    U = np.column_stack(Parallel(n_jobs=-1)(delayed(
        compute_u)(i) for i in tqdm(range(k), desc="
        Computing U")))
29
30    return U, singular_values[:k], v[:, :k].T

```

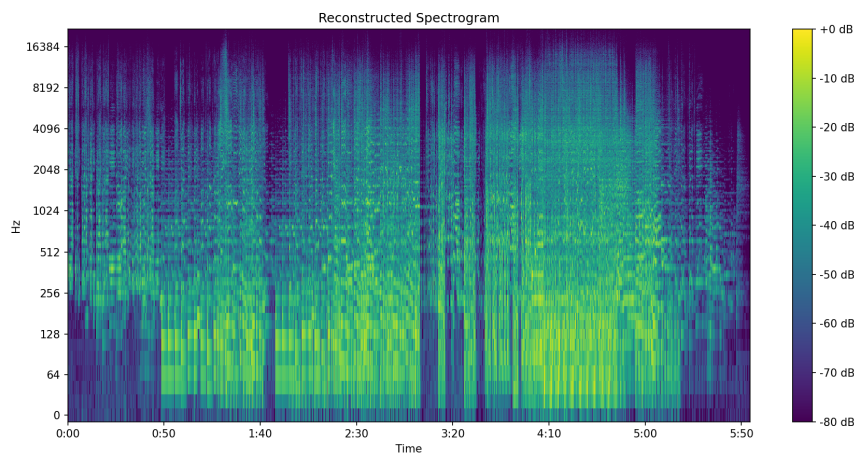
3 Grafice cu aproximarea soluției

3.1 Spectrograma originală

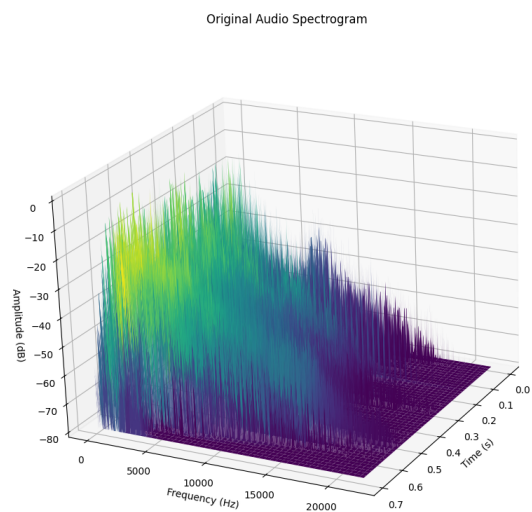
Am ales ca exemplu melodia [Bohemian Rhapsody](#) de Queen și următoarele grafice sunt create de la aceasta:



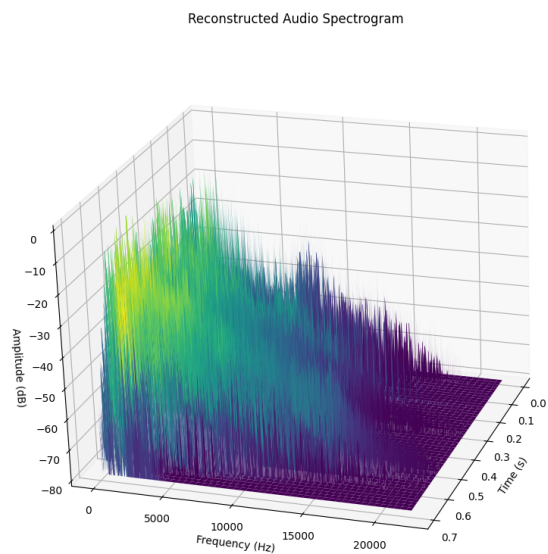
3.2 Spectrograma aproximată



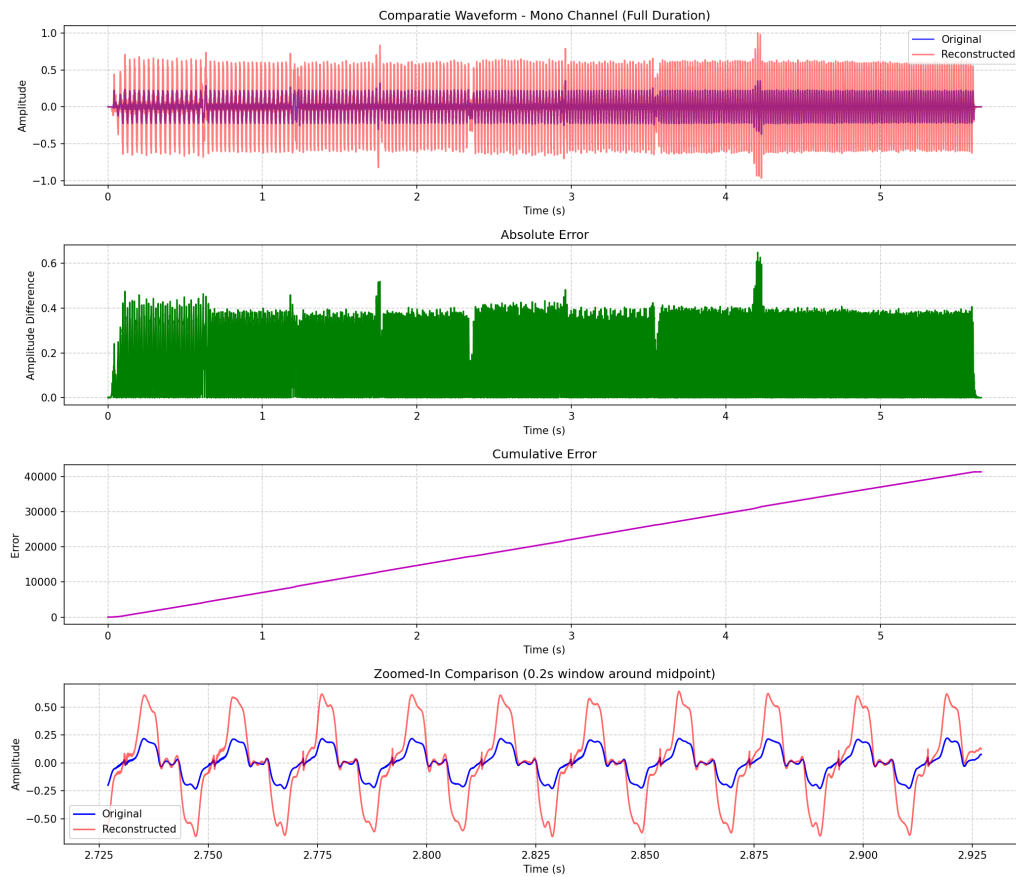
3.3 Spectograma originală 3D



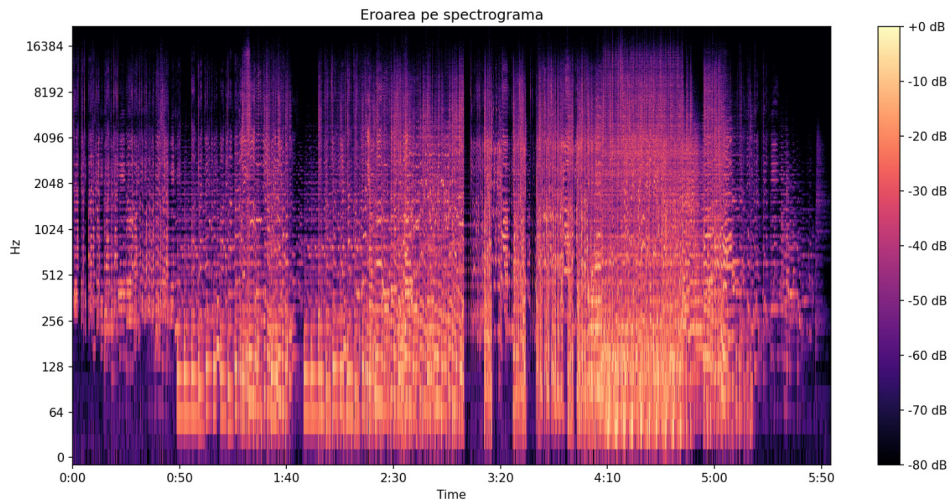
3.4 Spectograma aproximată 3D



3.5 Grafice erori



3.6 Spectograma eroare



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](#)
- [Uncertainty principle](#)
- Ce a fost dezvăluit lui Robert cât timp visa