# \*\*Documentation for Piano and Electronics Composition Project\*\*

## \*\*Overview\*\*

This project focuses on creating a timbre-based composition for piano and electronics, avoiding symbolic approaches. The workflow involves generating "unwanted" piano music, fragmenting it using content-aware methods, enriching it with sounds retrieved via the Freesound API, and developing spectral techniques to integrate piano materials with electronic textures.

---

## \*\*Workflow\*\*

### \*\*1. Generating the "Unwanted" Piano Piece\*\*

- \*\*Objective\*\*: Use AI to generate a cheesy, pop-style piano piece as raw material.

- \*\*Tool\*\*: Suno (or any AI music generator).

- \*\*Prompt\*\*:

```

[Bright Piano Melody], [Cheerful Pop Style], [Simple Chord Progressions], [120 BPM], [Feel-Good Vibes], [Catchy Repetitive Motifs]

```

- Keep the prompt simple and tag-based to ensure compatibility with Suno.

- Generate multiple versions if necessary to find the most "cheesy" result.

---

### \*\*2. Splitting the Piano Music into Fragments\*\*

- \*\*Objective\*\*: Segment the generated piano music into meaningful fragments using content-aware methods.

- \*\*Criteria for Splitting\*\*:

1. \*\*Onset Detection\*\*: Split at note or chord changes (e.g., using `librosa.onset.onset\_detect`).

2. \*\*Novelty Detection\*\*: Identify transitions between contrasting sections (e.g., via self-similarity matrices).

3. \*\*Spectral Features\*\*: Split based on timbral changes (e.g., spectral centroid or flatness).

4. \*\*Energy-Based Segmentation\*\*: Divide based on loudness levels (e.g., RMS energy).

5. \*\*Harmonic vs. Percussive Decomposition\*\*: Separate harmonic and percussive components (e.g., HPSS in `librosa`).

6. \*\*Silence Detection\*\*: Split at pauses or low-energy regions (e.g., `pydub.silence.detect\_nonsilent`).

7. \*\*Beat and Tempo Analysis\*\*: Align splits with rhythmic patterns or tempo changes.

- \*\*Tools\*\*:

- Python libraries: `Librosa`, `Essentia`, `madmom`, or `pydub`.

- Spectral analysis tools like Sonic Visualiser for manual refinement.

---

### \*\*3. Filtering the Fragments\*\*

- \*\*Objective\*\*: Reduce redundancy and select fragments that align with your aesthetic goals.

- \*\*Filtering Criteria\*\*:

1. \*\*Timbre-Based Filtering\*\*: Use MFCCs or spectral centroid to group fragments by timbre.

2. \*\*Spectral Complexity\*\*: Retain fragments with high or low harmonic richness.

3. \*\*Dynamic Range\*\*: Focus on fragments with strong dynamic contrasts.

4. \*\*Rhythmic Patterns\*\*: Select fragments with regular or irregular rhythmic structures.

5. \*\*Pitch Content\*\*: Choose fragments based on tonal stability, dissonance, or microtonality.

6. \*\*Novelty Scores\*\*: Retain unique fragments with high novelty values.

- \*\*Clustering for Filtering\*\*:

- Use clustering algorithms to organize fragments by feature similarity:

- K-Means for grouping by timbre or rhythm.

- DBSCAN for identifying outliers or unique fragments.

- Hierarchical clustering for nested relationships between fragments.

- Spectral clustering for subtle textural differences.

- \*\*Tools for Clustering\*\*:

- Python libraries: `sklearn.cluster` (K-Means, DBSCAN), `Librosa` (feature extraction), `matplotlib`/`seaborn` (visualization).

- Dimensionality reduction techniques like t-SNE or UMAP for visualizing clusters.

---

### \*\*4. Enriching Fragments Using Freesound API\*\*

- \*\*Objective\*\*: Retrieve similar sounds from Freesound.org to layer with your piano fragments.

- \*\*Steps\*\*:

- Query Freesound API using features extracted from each fragment:

- Spectral descriptors (e.g., pitch, timbre, spectral centroid).

- Tags related to emotional quality or texture (e.g., "bright," "metallic").

- Duration constraints to match fragment lengths.

- \*\*Tools\*\*:

- Freesound API documentation and Python SDK for querying sounds programmatically.

- \*\*Combining Sounds\*\*:

- Layer retrieved sounds with piano fragments using time-stretching, pitch-shifting, or spectral morphing techniques.

---

### \*\*5. Morphing Sounds\*\*

- \*\*Objective\*\*: Blend piano fragments with retrieved sounds to create hybrid textures.

- \*\*Recommended Tools for Morphing\*\*:

- Zynaptiq MORPH (plugin): Real-time audio morphing with multiple algorithms.

- MeldaProduction MMorph (plugin): Spectral morphing based on harmonic features.

- CDP (Composers' Desktop Project): Command-line tools for experimental sound transformations.

- iZotope Iris: Spectral editing and layering of audio components.

---

### \*\*6. Developing Piano Materials Using Spectral Techniques\*\*

- Analyze combined sounds to extract spectral features such as harmonic content, inharmonicity, or formant structures.

- Translate these features into playable piano material:

- Microtonal clusters derived from partials.

- Rhythmic patterns inspired by transient behavior in electronic textures.

- Tools:

- SPEAR (Sinusoidal Partial Editing Analysis and Resynthesis) for detailed spectral analysis.

- Python libraries (`Librosa`, `Essentia`) for extracting spectral data programmatically.

---

### \*\*7. Structuring the Composition\*\*

- Develop a structure where electronic textures interact dynamically with live piano material:

- Start with fragmented electronic textures that gradually merge with live piano playing.

- Alternate between sections dominated by electronics and those focusing on acoustic piano.

- Consider spatialization techniques to enhance interaction between acoustic and electronic elements:

- Assign electronic sounds to different speakers in a multi-channel setup.

---

## Tools Summary

| Task | Recommended Tools |

|-----------------------------|----------------------------------------------------|

| Generating Piano Music | Suno |

| Splitting Audio | Librosa, Essentia, madmom |

| Filtering Fragments | sklearn.cluster, t-SNE/UMAP |

| Querying Sounds | Freesound API |

| Morphing Sounds | Zynaptiq MORPH, Melda MMorph, CDP |

| Spectral Analysis | SPEAR, iZotope RX, Python Libraries |

| Spatialization | Ambisonics tools (Reaper plugins like IEM Suite) |

---

## Final Notes

This documentation provides a detailed roadmap to guide your creative process while leaving room for experimentation at each stage. By following these steps, you'll be able to craft a cohesive yet exploratory composition that merges acoustic and electronic elements seamlessly.

Here’s a comprehensive document to guide you through implementing a \*\*Hidden Markov Model (HMM)\*\* for your audio synthesis project. This includes learning the hidden probabilities, preparing timbre-focused training data, and synthesizing layered audio outputs based on the HMM.

---

# \*\*Documentation: Hidden Markov Model for Audio Synthesis\*\*

## \*\*Overview\*\*

This project uses a Hidden Markov Model (HMM) to organize and synthesize audio chunks derived from three piano songs. The HMM introduces structure and coherence by modeling transitions between \*\*hidden states\*\* (abstract timbral or musical qualities) and their corresponding \*\*observations\*\* (audio chunks). The goal is to create layered, organic textures that evolve probabilistically.

---

## \*\*Workflow\*\*

### \*\*1. Preparing Training Data\*\*

#### 1.1 Extract Audio Chunks

- Split your three piano songs into meaningful chunks using content-aware segmentation techniques (e.g., onset detection, novelty detection).

- Save these chunks as individual audio files.

#### 1.2 Extract Features

Since you are concerned with \*\*timbre\*\*, focus on features that capture timbral characteristics:

- \*\*MFCCs\*\*: Represent the spectral envelope of the sound, often used for timbre analysis.

- \*\*Spectral Centroid\*\*: Indicates the "brightness" of the sound.

- \*\*Spectral Flatness\*\*: Measures how noise-like or tonal a sound is.

- \*\*Zero-Crossing Rate\*\*: Captures percussive or noisy elements.

Feature extraction example:

```python

import librosa

import numpy as np

# Load an audio chunk

y, sr = librosa.load("chunk.wav", sr=None)

# Extract timbre-related features

mfcc = np.mean(librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=13), axis=1) # Mean MFCCs

spectral\_centroid = np.mean(librosa.feature.spectral\_centroid(y=y, sr=sr)) # Brightness

spectral\_flatness = np.mean(librosa.feature.spectral\_flatness(y=y)) # Tonality

# Combine features into a single vector

features = np.hstack([mfcc, spectral\_centroid, spectral\_flatness])

```

#### 1.3 Normalize Features

Normalize all feature dimensions to ensure they contribute equally during training:

```python

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

features\_normalized = scaler.fit\_transform(features)

```

#### 1.4 Organize Training Data

Prepare a dataset where:

- Each row corresponds to a feature vector for a chunk.

- Each chunk is labeled with its cluster ID (if pre-clustered) or left unlabeled for unsupervised learning.

---

### \*\*2. Training the Hidden Markov Model\*\*

#### 2.1 Define the HMM

Use `hmmlearn` to define an HMM with:

- `n\_components`: Number of hidden states (e.g., timbral archetypes).

- `n\_iter`: Number of iterations for training.

Example:

```python

from hmmlearn import hmm

# Define an HMM with 3 hidden states

model = hmm.GaussianHMM(n\_components=3, covariance\_type="diag", n\_iter=100)

```

#### 2.2 Train the HMM

Fit the model to your normalized feature data:

```python

# Train the HMM on feature vectors

model.fit(features\_normalized)

```

#### 2.3 Inspect Learned Probabilities

After training, inspect the learned transition and emission probabilities:

```python

print("Transition Matrix:", model.transmat\_) # Transition probabilities between hidden states

print("Means of Emissions:", model.means\_) # Mean feature vectors for each state

print("Covariances of Emissions:", model.covars\_) # Variance of feature vectors for each state

```

---

### \*\*3. Generating Sequences\*\*

#### 3.1 Generate Hidden State Sequence

Use the trained HMM to generate a sequence of hidden states and their corresponding observations:

```python

# Generate a sequence of hidden states and observations

num\_chunks = 10 # Number of chunks to generate

hidden\_states, observations = model.sample(num\_chunks)

print("Hidden States:", hidden\_states)

print("Observations:", observations)

```

#### 3.2 Map Observations Back to Audio Chunks

Map the generated observation indices back to your original audio chunks and combine them into a single audio file:

```python

from pydub import AudioSegment

# Map observation indices to chunk file paths

chunk\_paths = ["chunk1.wav", "chunk2.wav", "chunk3.wav"] # Replace with actual paths

selected\_chunks = [chunk\_paths[i] for i in observations]

# Combine selected chunks into one audio file with overlaps or crossfades

combined\_audio = AudioSegment.empty()

for chunk\_path in selected\_chunks:

chunk = AudioSegment.from\_file(chunk\_path)

combined\_audio += chunk.crossfade(chunk) # Add crossfade if desired

combined\_audio.export("output.wav", format="wav")

```

---

### \*\*4. Layering and Overlapping Chunks\*\*

To create richer textures, layer multiple chunks together based on their hidden states:

1. For each hidden state, select multiple chunks with high emission probabilities.

2. Offset their start times slightly to create overlapping layers.

3. Use effects like time-stretching or pitch-shifting for additional variation.

Example with `pydub`:

```python

layered\_audio = AudioSegment.empty()

for chunk\_path in selected\_chunks[:3]: # Layer first three chunks as an example

chunk = AudioSegment.from\_file(chunk\_path)

layered\_audio = layered\_audio.overlay(chunk)

layered\_audio.export("layered\_output.wav", format="wav")

```

---

### \*\*5. Refining Hidden States\*\*

If you want more control over the hidden states:

- Manually assign labels to clusters (e.g., "soft," "bright") and use these as initial guesses for training.

- Adjust transition probabilities manually if certain transitions are more musically meaningful (e.g., avoid jumping directly from "soft" to "chaotic").

---

## \*\*Tips for Timbre-Focused Training\*\*

1. \*\*Choose Relevant Features\*\*:

- Focus on timbre-related features like MFCCs, spectral centroid, flatness, and bandwidth.

- Avoid features unrelated to timbre (e.g., tempo) unless rhythm is also important.

2. \*\*Pre-Clustering\*\*:

- Use clustering methods (e.g., K-Means or DBSCAN) to group similar chunks before training the HMM. This can help initialize meaningful hidden states.

3. \*\*Balance Your Dataset\*\*:

- Ensure that all timbral categories are well-represented in your training data to avoid biasing the model toward certain states.

4. \*\*Experiment with Hidden States\*\*:

- Start with a small number of states (e.g., $$ n\\_components=3 $$) and increase gradually to capture finer distinctions in timbre.

---

## \*\*Creative Applications\*\*

1. \*\*Dynamic Transitions\*\*:

- Use the transition matrix to control how textures evolve over time (e.g., smooth vs. abrupt changes).

2. \*\*Layering by State\*\*:

- Assign overlapping layers based on hidden states (e.g., multiple "chaotic" chunks played together).

3. \*\*Interactive Performance\*\*:

- Use real-time input (e.g., MIDI or sensor data) to influence state transitions during live performance.

4. \*\*Hybrid Textures\*\*:

- Combine generated sequences with additional sounds retrieved from Freesound.org using similar timbral features.

---

## Final Thoughts

This workflow leverages the power of HMMs to create structured yet unpredictable audio textures based on your piano fragments' timbral qualities. By learning hidden probabilities from your dataset, you can generate evolving sequences that feel organic and musically coherent while leaving room for creative exploration.

Let me know if you'd like further help refining any part of this process!