Symbolic Music Generation from a Single MIDI File
via Positive Definite Kernels, Interval Graphs, Kernel PCA, and Bezier Trajectories

Orges Leka

Germany, Limburg
orges.leka@gmail.com

November 4, 2024

Abstract

We develop a pipeline that, given a single input MIDI (or MusicXML) file, constructs a novel symbolic piano composition by (i) defining a positive definite similarity kernel on musical notes, (ii) building an interval graph to capture temporal overlaps, (iii) extracting connected components as structural units, (iv) applying Kernel Principal Component Analysis (KernelPCA) via Nyström approximation for dimensionality reduction, (v) planning smooth trajectories in the reduced space with Bezier curves, and (vi) reconstructing music via nearest-neighbor mapping. We present each step with mathematical formulations and pseudocode, and discuss implementation details.

1 Introduction

Automatic music generation often requires large corpora. Here, we demonstrate *single-instance learning*: generating new music solely from one input file. This report introduces a novel method that utilizes a combination of mathematical and machine learning techniques to generate music sequences that are both structurally coherent and musically pleasing, derived solely from a single input MIDI file. We combine four mathematical tools:

1. Positive definite kernels to quantify note-to-note similarity based on musical perception.
2. Interval graphs to model polyphonic temporal structure and relationships.
3. Kernel PCA (with Nyström approximation) to embed graph components in low-dimensional Euclidean space.
4. Bezier curves to define smooth, controllable trajectories through that space.

A final k-Nearest Neighbors (k-NN) step maps trajectory points back to original musical units (connected components of the graph), allowing reconstruction of the generated piece. This method yields coherent, musically valid output without large training sets by focusing on the intrinsic structure of the input.

2 Mathematical Preliminaries

2.1 Positive Definite Kernels

A function $K\colon X \times X \to \mathbb{R}$ is a *positive definite kernel* if for any finite set $\{x_i\}_{i=1}^n \subset X$ the Gram matrix $[K(x_i, x_j)]_{i,j=1}^n$ is symmetric positive semidefinite. Kernels correspond to inner products in some (possibly high-dimensional) Hilbert space $\mathcal{H}$, $K(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{H}}$, where $\phi : X \to \mathcal{H}$ is a feature map.

2.2 Bezier Curves

Given control points $P_0, \ldots, P_N \in \mathbb{R}^D$, the degree-$N$ Bezier curve is defined for $s \in [0, 1]$ as:

$$B(s) \;=\; \sum_{k=0}^{N} \binom{N}{k} (1-s)^{N-k} s^k \, P_k.$$

These curves provide smooth interpolation between the control points.

3 Note Similarity Kernel

We represent each note or rest as a tuple:

$$n = (p, d, v, r),$$

where:

- $p \in \mathbb{Z}$ is the MIDI pitch (a default value, e.g., 60, is used for rests).
- $d \in \mathbb{R}^+$ is duration (e.g., in quarter note lengths, normalized if needed).
- $v \in \{0, \ldots, 127\}$ is MIDI velocity (volume); rests typically have $v = 0$ or a low default.
- $r \in \{0, 1\}$ indicates rest ($r = 1$) or note ($r = 0$).

We define attribute kernels to capture similarity across dimensions:

- **Pitch Similarity (K_p):** Captures perceptual consonance based on the simplicity of the frequency ratio $f_2/f_1 = 2^{(p_2-p_1)/12}$. We find the best rational approximation a/b for this ratio using continued fractions (`getlowestfraction`). The similarity is then based on the greatest common divisor (GCD):

$$K_p(p_1, p_2) = \frac{\gcd(a, b)^2}{a \cdot b}. \tag{1}$$

  This assigns higher similarity to consonant intervals like octaves, fifths, and fourths.

- **Duration Similarity (K_d):** Uses a Jaccard-like index for positive durations:

$$K_d(d_1, d_2) = \frac{\min(d_1, d_2)}{\max(d_1, d_2)}. \tag{2}$$

- **Volume Similarity (K_v):** Similarly measures the ratio of MIDI velocities:

$$K_v(v_1, v_2) = \frac{\min(v_1, v_2)}{\max(v_1, v_2)}, \quad \text{for } v_1, v_2 > 0. \tag{3}$$

  (Special handling might be needed if velocities can be zero).

- **Rest Similarity (K_r):** Binary indicator:

$$K_r(r_1, r_2) = \begin{cases} 1, & \text{if } r_1 = r_2, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

These are aggregated into a composite kernel K_{note}, implemented as `kernNote`, using weights $(\alpha_p, \alpha_d, \alpha_v, \alpha_r)$:

$$K_{\text{note}}(n_1, n_2) = \frac{\alpha_p\, K_p(p_1, p_2) + \alpha_d\, K_d(d_1, d_2) + \alpha_v\, K_v(v_1, v_2) + \alpha_r\, K_r(r_1, r_2)}{\alpha_p + \alpha_d + \alpha_v + \alpha_r}. \tag{5}$$

The code uses typical weights $(2, 4, 1, 8)$, emphasizing rests and duration, then pitch, then volume. This kernel is designed to be positive definite if the individual attribute kernels are (which holds for these definitions).

Algorithm 1 Compute $K_{\text{note}}(n_1, n_2)$ with weights $(2, 4, 1, 8)$

Require: Notes $n_i = (p_i, d_i, v_i, r_i)$ for $i = 1, 2$.

1: Compute $k \leftarrow p_2 - p_1$.
2: Use continued fraction (`getlowestfraction`, tolerance $\varepsilon = 0.01$) to approximate $2^{k/12} \approx a/b$.
3: Compute $K_p \leftarrow \gcd(a, b)^2/(a\,b)$ (`kernPitch`).
4: Compute $K_d \leftarrow \min(d_1, d_2)/\max(d_1, d_2)$ if $d_1, d_2 > 0$ else $1_{d_1=d_2=0}$ (`kernDuration`).
5: Compute $K_v \leftarrow \min(v_1, v_2)/\max(v_1, v_2)$ if $v_1, v_2 > 0$ else $1_{v_1=v_2=0}$ (`kernVolume`).
6: Compute $K_r \leftarrow 1$ if $r_1 = r_2$ else 0 (`kernPause`).
7: **return** $(2K_p + 4K_d + 1K_v + 8K_r)/15$.

4 Interval Graph Construction

To capture temporal overlap and harmonic similarity in polyphonic music, we build an *interval graph* $G = (V, E)$.

- **Vertices (V):** Each vertex $i \in V$ represents a musical interval derived from a note or chord in the input file, parsed using `music21`. An interval is represented as:

$$i = (v, t^i_{\min}, t^i_{\max}, \text{notes}_i),$$

  where v is the voice index, $[t^i_{\min}, t^i_{\max}]$ is the time interval, and notes_i is a list containing one or more note tuples (p, d, v, r).

- **Edges (E):** An edge $(i, j) \in E$ connects two intervals if they overlap temporally and are sufficiently similar musically:

$$(i, j) \in E \iff \text{overlap}(i, j) \wedge K_{\text{intChord}}(i, j) > \varepsilon_q.$$

Temporal overlap is defined as:

$$\text{overlap}(i,j) \iff [t^i_{\min}, t^i_{\max}] \cap [t^j_{\min}, t^j_{\max}] \neq \emptyset.$$

Musical similarity between intervals i and j is measured by K_{intChord}:

$$K_{\text{intChord}}(i,j) = K_{\text{chord}}(\text{notes}_i, \text{notes}_j) \times J(i,j),$$

where $J(i,j)$ is the Jaccard index of their time intervals $\frac{|[t^i_{\min}, t^i_{\max}] \cap [t^j_{\min}, t^j_{\max}]|}{|[t^i_{\min}, t^i_{\max}] \cup [t^j_{\min}, t^j_{\max}]|}$, and K_{chord} compares the note content:

$$K_{\text{chord}}(C_1, C_2) = \frac{1}{\min(|C_1|, |C_2|)} \sum_{m=1}^{\min(|C_1|,|C_2|)} K_{\text{note}}(C_1[m], C_2[m]).$$

Here, $C_1 = \text{notes}_i$ and $C_2 = \text{notes}_j$ are lists of notes, compared element-wise. The threshold ε_q is chosen as a quantile (e.g., median or lower quartile) of the K_{intChord} values among all temporally overlapping pairs found using an IntervalTree, ensuring that edges represent meaningful musical connections beyond mere temporal coincidence.

Algorithm 2 Build pruned interval graph G

Require: List of voices $\{\text{score}_v\}$, each a sequence of (`M21_note`, notes list n). Parameter τ for quantile threshold.

Parse scores into a list of intervals $I \leftarrow \{(v, t_{\min}, t_{\max}, n)\}$.
Build `IntervalTree` 'tree' over time intervals $[t_{\min}, t_{\max}]$ from I.
Initialize empty graph G_f with nodes I.
Create a list 'overlap_similarities'.
for all time-slices t subdividing $[0, T_{\max}]$ (or use efficient tree query) **do**
    $S \leftarrow$ intervals in 'tree' covering t.
    for all pairs $\{i, j\} \subseteq S$, $i \neq j$ **do**
        if edge (i, j) not already processed in G_f **then**
            Add edge (i, j) to G_f.
            Compute $k_{ij} = K_{\text{intChord}}(i, j)$.
            Append k_{ij} to 'overlap_similarities'.
if 'overlap_similarities' is not empty **then**
    Let $\varepsilon_q \leftarrow \text{Quantile}_\tau(\text{overlap\_similarities})$.
else
    Let $\varepsilon_q \leftarrow 0$.
Initialize graph G with nodes I.
for all edges $(i, j) \in E(G_f)$ **do**
    Compute $k_{ij} = K_{\text{intChord}}(i, j)$ (or retrieve if stored)
    if $k_{ij} > \varepsilon_q$ **then**
        Add edge (i, j) to G.
return G.

5 Connected Components

Compute the set of connected components $\mathcal{C} = \{C_1, \ldots, C_M\}$ of the pruned interval graph G using standard algorithms like Breadth-First Search (BFS) or Depth-First Search

(DFS). Each component C_k, which is a list of interval nodes $\{i_1, \ldots, i_{m_k}\}$, represents a structurally and temporally related group of musical events. These components are treated as the fundamental *musical units* for analysis and generation. The components are typically sorted based on their earliest start time or index for consistent processing.

6 Kernel PCA via Nyström Approximation

Direct KernelPCA on the $M \times M$ Gram matrix $K_{ij} = K_{\text{cc}}(C_i, C_j)$ can be computationally expensive ($O(M^3)$ for eigendecomposition). We use the Nyström method combined with standard PCA to approximate the Kernel PCA mapping $\Phi : \mathcal{C} \to \mathbb{R}^D$ efficiently. The kernel between two components C_1, C_2 is defined as:

$$K_{\text{cc}}(C_1, C_2) = \frac{1}{\min(|C_1|, |C_2|)} \sum_{m=0}^{\min(|C_1|,|C_2|)-1} K_{\text{intChord}}(C_1[m], C_2[m]), \tag{6}$$

assuming an internal ordering (e.g., temporal) within the component lists C_1, C_2.

Algorithm 3 Approximate Kernel PCA embedding $\Phi(C_k) \in \mathbb{R}^D$

Require: Components $\{C_k\}_{k=1}^M$, component kernel K_{cc}, target dimension D, landmarks $L \ll M$.

1: Select L landmark indices $\{i_1, \ldots, i_L\}$ uniformly at random from $\{1, \ldots, M\}$.
2: Compute $W \in \mathbb{R}^{L \times L}$ with $W_{ab} = K_{\text{cc}}(C_{i_a}, C_{i_b})$.
3: Compute $A \in \mathbb{R}^{M \times L}$ with $A_{ka} = K_{\text{cc}}(C_k, C_{i_a})$.
4: Compute $W^{\dagger 1/2}$, the pseudo-inverse square root of W (e.g., via SVD or regularized Cholesky).
5: $\widetilde{\Phi} \leftarrow AW^{\dagger 1/2} \in \mathbb{R}^{M \times L}$. This is the Nyström approximation.
6: Perform standard linear PCA on the columns of $\widetilde{\Phi}$ to reduce dimension from L to D. Let the projection matrix be $P \in \mathbb{R}^{L \times D}$.
7: Compute the final embeddings $\mathbf{\Phi} \leftarrow \widetilde{\Phi} P \in \mathbb{R}^{M \times D}$.
8: **return** Rows $\mathbf{\Phi}_k$ as embeddings of C_k.

The resulting matrix $\mathbf{\Phi}$ (denoted `XNotes` in code) contains the D-dimensional Euclidean embeddings of the musical units.

7 Trajectory Planning with Bezier Curves

We generate smooth trajectories through the D-dimensional embedding space using Bezier curves. The embeddings $\{\mathbf{\Phi}_k\}_{k=1}^M$ of the connected components, potentially ordered (e.g., by time or index), serve as the control points $P_k = \mathbf{\Phi}_k$. Given a user-supplied sequence of normalized parameters $ss = \{s_t\}_{t=1}^T$ where $s_t \in [0, 1]$ (derived from normalizing the `numberInput` sequence), we evaluate points along the Bezier curve $B(s)$:

$$\mathbf{Q}_t \;=\; B(s_t) \;=\; \sum_{k=1}^{M} \binom{M-1}{k-1} (1 - s_t)^{M-k} s_t^{k-1} \, \mathbf{\Phi}_k.$$

The sequence $\{\mathbf{Q}_t\}_{t=1}^{T}$ represents the generated musical trajectory in the feature space. Different input sequences ss, potentially derived from functions like cosine or Weierstrass (`funcType`), yield varied musical outputs.

8 Nearest-Neighbor Reconstruction

To translate the trajectory $\{\mathbf{Q}_t\}$ back into music, we map each point $\mathbf{Q}_t$ to the most similar original musical unit (connected component C_k). This is done using k-Nearest Neighbors (k-NN) in the embedding space $\mathbb{R}^D$.

- **k-NN Model Fitting:** Fit a k-NN search structure (e.g., Ball Tree or KD Tree via `sklearn.neighbors.NearestNeighbors`) on the embedding vectors $\{\mathbf{\Phi}_k\}_{k=1}^{M}$.
- **Mapping:** For each trajectory point $\mathbf{Q}_t$, find the index k_t of the nearest neighbor among the component embeddings:

$$k_t = \underset{k \in \{1,\ldots,M\}}{\arg\min} \|\mathbf{Q}_t - \mathbf{\Phi}_k\|_2.$$

  The implementation (`findBestMatches`) typically uses $k = 1$ neighbor.
- **Sequence Reconstruction:** The final musical output is generated by concatenating the sequences of notes contained within the selected components $\{C_{k_t}\}_{t=1}^{T}$. The function `writeM21Lists` handles the assembly into `music21` streams and saving to MIDI or MusicXML format, preserving the voice structure based on the original voice index stored in the interval tuples.

Algorithm 4 End-to-end music generation pipeline

Require: Input MIDI/MusicXML file `infile`, parameters $\tau, D, L, \{s_t\}_{t=1}^{T}$.

1: **Parse** `infile` into voices $\{\text{score}_v\}$ and compute list of intervals I.
2: **Build** pruned interval graph G using I and quantile threshold τ (Algorithm 2).
3: **Extract** connected components $\{C_k\}_{k=1}^{M}$ from G.
4: **Embed** components $\{C_k\}$ into $\mathbb{R}^D$ via Nyström-KPCA using L landmarks, yielding $\{\mathbf{\Phi}_k\}_{k=1}^{M}$ (Algorithm 3).
5: **Bezier Curve:** Define $B(s)$ using $\{\mathbf{\Phi}_k\}$ as control points.
6: **k-NN Model:** Fit `NearestNeighbors` model on $\{\mathbf{\Phi}_k\}$.
7: Initialize output voices `output_voices`.
8: **for** $t = 1$ to T **do**
9: Compute trajectory point $\mathbf{Q}_t = B(s_t)$.
10: Find index k_t of nearest neighbor to $\mathbf{Q}_t$ in $\{\mathbf{\Phi}_k\}$.
11: Retrieve intervals in component C_{k_t}.
12: Append notes from intervals in C_{k_t} to the appropriate `output_voices` based on voice index.
13: **Write** `output_voices` to a new MIDI/MusicXML file using `music21`.

9 Implementation Details

- **Language:** Python 3.

- **Core Libraries:** `music21`, `portion`, `networkx`, `numpy`, `scikit-learn` (for KernelPCA, Nystroem, PCA, NearestNeighbors, StandardScaler), `bezier`.
- **Web Interface:** `Flask` (`main.py`) handles file upload, parameter input (tempo, ϵ, sequence 'numbers', PCA dimensions 'title'/'author' used as L/D), and download of results.
- **Key Modules:**
  1. `connected_components.py`: Contains kernel definitions (`kernNote`, `kernIntChord`, etc.), parsing (`parse_file`), interval graph construction (`interval_graph_0`), component extraction (implicitly via `nx.connected_components`), KPCA embedding (`getCoordinatesOf`), k-NN fitting/querying (`get_knn_model`, `findBestMatches`), and reconstruction (`processFile`, `writeM21Lists`).
  2. `main.py`: Flask web application.
- **Performance:** The Nyström method significantly reduces the complexity of Kernel PCA from $O(M^3)$ to approximately $O(ML^2 + MD^2)$, where M is the number of components, L is the number of landmarks, and D is the target dimension, making the approach feasible for reasonably complex input files.

10 Conclusion

We have detailed a complete pipeline to generate new symbolic piano music from a single input file, harnessing kernel methods, graph theory, dimensionality reduction, and geometric curve modeling. This approach, based on a comprehensive similarity measure informed by music theory and structural analysis via interval graphs, effectively models units of the input music and generates new, coherent, and aesthetically pleasing sequences by navigating trajectories in a reduced feature space. The integration of these mathematical tools facilitates the creation of musically interesting variations derived from the input file's characteristics. The user interface further enables creative exploration by allowing user control over the generative trajectory via numerical sequences. This work highlights the potential of combining kernel methods, graph theory, and geometric modeling for tasks in computational creativity and music generation, particularly in single-instance learning scenarios. Future work may explore alternative kernels, dynamic graph weighting schemes, adaptive Bezier curve degrees, or incorporating user feedback into the trajectory planning.

References

[1] P. E. Bézier, *Numerical Control—Mathematics and Applications*, Wiley, 1972.

[2] Interactive platform for music generation: `https://musescore1983.pythonanywhere.com/` (Accessed Nov 2024).

[3] B. Schölkopf, A. J. Smola, K.-R. Müller, *Nonlinear Component Analysis as a Kernel Eigenvalue Problem*, Neural Computation, 10(5):1299–1319, 1998.

[4] Examples for Piano solo generated with the method above, SoundCloud. Retrieved from `https://soundcloud.com/user-919775337/sets/waves-experiment`.

[5] "Interval Graph," Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Interval_graph (Accessed Nov 2024).

[6] "Twelfth root of two," Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Twelfth_root_of_two (Accessed Nov 2024).