Technical Appendix for "Enabling Empirical Analysis of Piano Performance Rehearsal with the < Anonymized > MIDI Dataset"

1 Symbolic Fingerprinting

1.1 Fingerprint Hash Generation

- Generates a list of tokens that contains a hash value representing three note sequences that occur in the score/performance, by using their individual pitches and the ratio of the onset time difference between them.

- The parameter 'd' represents the minimum time difference two consecutive notes must have between their onset times (in seconds) in order to consider them as separate notes.

- The n1 and n2 parameters allow for performance errors where certain notes are skipped. n1 determines the number of notes in the note event sequence to consider as options for the second note in the three note sequence, and n2 determines the same for the third note in the sequence. e.g. for a sequence C-D-E-F-G and n1, n2 = 2:

  a = range(0, n1), b = range(0, n2)

  Token 1: (C, D, E); where a = 1 and b = 1

  Token 2: (C, D, F); where a = 1 and b = 2

  Token 3: (C, E, F); where a = 2 and b = 1

  Token 4: (C, E, G); where a = 2 and b = 2

  For generating the fingerprint tokens from the score, both n1 and n2 take the value of 3, in order to allow the tokens of the score to represent note skip errors. For the generation of tokens from each rehearsal, both n1 and n2 take the value of 1, as the rehearsal's tokens need to represent the rehearsal as is played.

Algorithm 1: Token Generation

```
Data: note_array
Result: token_list
d ← 0.05;
n1, n2 ← 3;
token_list ← [ ];
for i ← 0 to length(note_array) do
    note1_pitch ← note_array[i][‘pitch’];
    note1_onset ← note_array[i][‘onset’];
    note2_count ← 0;
    j ← i + 1;
    repeat
        note2_flag ← True;
        repeat
            note2_onset ← note_array[j][‘onset’];
            td_1 ← note2_onset - note1_onset ;
            if td_1 ≥ d then
                note2_pitch ← note_array[j][‘pitch’];
                note2_flag ← False;
            else
                j ← j + 1;
            end
        until note2_flag = False;
        note3_count ← 0;
        k ← j + 1;
        repeat
            note3_flag ← True;
            repeat
                note3_onset ← note_array[k][‘onset’];
                td_2 ← note3_onset - note2_onset ;
                if td_2 ≥ d then
                    note3_pitch ← note_array[k][‘pitch’];
                    td_r ← td_1/td_2;
                    notes_hash ← hash(note1_pitch, note2_pitch,
                     note3_pitch, td_r);
                    token ← [notes_hash, note1_onset, td_1];
                    token_list.Append(token);
                    note3_flag ← False;
                else
                    k ← k + 1;
                end
            until note3_flag = False;
        until note3_count = n2;
    until note2_count = n1;
end
```

1.2 Hash Table Generation

- Given the list of tokens that were created from multiple scores, a hash table is generated where the hash is linked to the list of scores from which it was generated.

Algorithm 2: Update Hash Table with New Score's Token List

```
Data: token_list, score_name
Result: hash_table
hash_table ← { };
for i ← 0 to length(token_list) do
    if token_list[ i ][ 0 ] NOT in hash_table then
        hash ← token_list[ i ][ 0 ];
        hash_table[hash] ← [ ];
    else
        hash_table[ hash ].Append(score_name);
    end
end
```

1.3 Rehearsal Piece Identification

- Given a token list from a rehearsal midi file, the individual hashes from the token list are looked up in the hash table. A counter is maintained for each score in the hash table. If the hash exists in the table, the counters of all the scores that contain the hash are incremented by 1.
- Finally, the score with the highest count is returned as the predicted piece.

Algorithm 3: Predict Rehearsal Piece

```
Data: rehearsal_token_list, hash_table
Result: predicted_piece
score_count ← { };
for i ← 0 to length(rehearsal_token_list) do
    rehearsal_hash ← rehearsal_token_list[ i ][ 0 ];
    if rehearsal_hash NOT in hash_table then
        score_count[ 'None' ] ← score_count[ 'None' ] + 1;
    else
        for j ← 0 to length(hash_table[ hash ] do
            score_name ← hash_table[ hash ][ j ];
            score_count[ score_name ] ← score_count[ score_name ] + 1;
        end
    end
end
predicted_piece ← maximum(score_count);
```

2 Rehearsal Structure Analysis

Our approach for finding related fragments in a rehearsal (referred to in the manuscript as step 1 of an ideal rehearsal structure analysis pipeline) is divided into 2 main phases. First, the Self Similarity Matrix (SSM) is computed. Then, it is used to find relevant diagonals, which pass through a series of grouping and filtering operations. The output is a set of grouped intervals representing related fragments in the input performance.

2.1 Self Similarity Matrix

1. Group notes within a given proximity threshold into one time bin, and create a 'chord group' matrix from this info where a pitch is set to 1 when it appears in a bin.
2. Concatenate the observation probability for each bin with respect to a pitch profile matrix of the whole rehearsal.

Algorithm 4: Complete Self-Similarity Matrix Computation from Note Array

```
Data: note_array ;      /* array of (note['onset_sec'], note) */
Result: ssm ;   /* Self Similarity Matrix (n_bins x n_bins) */

ssm ← [ ];
win ← 100 ;                                   /* milliseconds */
profile ← [0.02, 0.02, 1, 0.02, 0.02] ;      /* Default Value */

chord_pitches ← ChordifyProximalPitches(note_array, win);
pitch_profiles ← ComputePitchProfiles(chord_pitches, profile)

for idx ← 0 to length(chord_pitches) - 1 do
    row ← ComputeObsProbability(pitch_profiles, chord_groups[idx]) ;
    row ← Reshape(row, 1, length(row));
    ssm.Append(row);
end
ssm ← Concatenate(self_similarity_matrix, axis=0);

return ssm
```

Algorithm 5: Chordify Proximal Pitches

```
Function ChordifyProximalPitches(note_array, win):
    i ← 0;
    prox_groups ← [];

    while i < length(note_array) do
        (time, note) ← note_array[i];
        if i = 0 then
            group ← EmptyArray();
            group_start ← time;
        end
        else
            if (time - group_start) × 1000 ≥ win then
                prox_groups.Append(group);
                group ← EmptyArray() ;        /* start new group */
                group_start ← time;
            end
        end
        group.Append((time, note));
        i ← i + 1;
    end
    prox_groups.Append(group) ;  /* Add final group to result */

    n ← length(prox_groups);
    chord_pitches ← init_zeros(n, 128);
    grp_starts ← init_zeros(n);

    for i ← 0 to length(prox_groups) - 1 do
        group ← prox_groups[i];
        for j ← 0 to length(group) - 1 do
            (time, note) ← group[j];
            if j = 0 then
                grp_starts.Append(t)
            end
            chord_pitches[i, n.pitch] ← 1;
        end
    end
    return chord_pitches
end
```

Algorithm 6: Compute Pitch Profiles

```
Function ComputePitchProfiles(chord_pitches, profile):
    eps ← 0.01;
    pitch_profiles ← Convolve(chord_pitches, profile);
    pitch_profiles ← pitch_profiles + eps;
    pitch_profiles ← pitch_profiles / Maximum(pitch_profiles)

    return pitch_profiles;
end
```

Algorithm 7: Compute Chord Observation Probabilities

```
Function ComputeObsProbability(pitch_profiles, pitch_obs):
    pitch_prob ← (pitch_profiles^pitch_obs) × ((1 -
      pitch_profiles)^(1−pitch_obs));
    pitch_obs_prob ← Product(pitch_prob, dim=1);

    return pitch_obs_prob;
end
```

2.2 Finding Related Rehearsal Fragments

1. Find all diagonals matching hyperparameter constraints (α: minimum diagonal length, β: similarity threshold, and γ: gap tolerance)

2. Group diagonals by overlaps along the horizontal and vertical SSM dimensions. (λ: overlap ratio)

3. Merge the identified horizontal and vertical groups based on inter-group common diagonal occurrences.

4. Convert diagonals to time intervals $(t_{\text{start}}, t_{\text{end}})$ (in seconds)

   Pseudocode is not provided for the SortByDim and ConvertToIntervals functions. SortByDim sorts diagonals based on their start times along the given dimension. ConvertToIntervals converts the input groups from diagonals into second intervals

Algorithm 8: Finding Related Rehearsal Fragments - Main

```
Data: ssm, α, β, γ, λ
Result: interval_groups;  /* [[(t1_s, t1_e), ..], [ ], ..]    */
unsorted_diagonals ← FindDiagonals(ssm, α, β, γ)

i_sorted_diags ← SortByDim(diagonals, λ, dim=0);  /* by start_i */
h_groups ← GroupByOverlaps(i_sorted_diags, dim="horizontal");
j_sorted_diags ← SortByDim(diagonals, λ, dim=1);  /* by start_j */
v_groups ← GroupByOverlaps(j_sorted_diags, dim="vertical");

merged_groups ← MergeRelatedGroups(v_groups, h_groups)
intervals ← ConvertToIntervals(merged_groups)
return intervals
```

Algorithm 9: Find Diagonals in Self-Similarity Matrix

```
Function FindDiagonals(ssm, α, β, γ):
    diagonals ← [ ];
    similarity_thresh ← β × Maximum(ssm);

    for offset ← 1 to ssm.shape[0] - 1 ;   /* Exclude main diagonal */
    do
        diag ← Diagonal(ssm, k=offset);
        curr_start ← null;
        curr_len ← 0;
        tol_ctr ← 0;
        local_tol_ctr ← 0;
        for i ← 0 to length(diag) - 1 do
            if diag[i] ≥ similarity_thresh then
                if cur_start = null then
                    curr_start ← i ;        /* Start diag.  segment */
                end
                curr_len ← curr_len + 1 ;   /* Grow active segment */
                local_tol_ctr ← 0;
            end
        end
        else if curr_start ≠ null ; /* if < thresh found mid-segment */
        then
            if tol_ctr < γ then
                tol_ctr ← tol_ctr + 1;
                local_tol_ctr ← local_tol_ctr + 1;
                curr_len ← curr_len + 1;
            end
            else
                if curr_len - local_tol_ctr ≥ α then
                    diagonals.Append((curr_start, curr_start + offset,
                     curr_len-local_tol_ctr));
                end
                curr_start ← null;
                curr_len ← 0;
                tol_ctr ← 0;
            end
        end
    end
    if curr_start ≠ null and curr_len - local_tol_ctr ≥ α then
        diagonals.Append((curr_start, curr_start + offset,
         curr_len-local_tol_ctr));
    end
    return diagonals ;  /* list of (start_i, start_j, length) */
end
```

Algorithm 10: Group By Overlaps - Main Algorithm

```
Function GroupByOverlaps(sorted_diagonals, overlap_ratio, dim):
    diagonal_seen ← ZerosArray(length(sorted_diagonals));
    groups ← EmptyArray();

    for i ← 0 to length(sorted_diagonals) - 1 do
        if diagonal_seen[i] = 1 then
            continue ;      /* every diagonal is processed once */
        end
        group_head = sorted_diagonals[i];
        current_group ← EmptyArray();
        current_group.Append(group_head);
        diagonal_seen[i] ← 1;

        /* (start, end) on given dim                              */
        group_interval ← current_cluster[0].GetInterval(dim);

        for j ← 0 to length(sorted_diagonals) - 1 do
            if diagonal_seen[j] = 1 then
                continue;
            end
            interval_j ← sorted_diagonals[j].GetInterval(dim);

            /* CASE 1:  INTERVALS OVERLAP                         */
            if IntervalOverlap(interval_j, group_interval, overlap_ratio)
             then
                current_cluster.Append(sorted_diagonals[j]);
                diagonal_seen[j] ← 1;
                continue;
            end

            /* CASE 2:  CLUSTER HEAD CONTAINS INTERVAL            */
            if IntervalSubset(interval_j, group_head) then
                /* Cut diagonal to match group head span along
                   chosen dim                                     */
                cut_diagonal ← CreateDiagonalSubset(sorted_diagonals,
                 i, j, dim);
                current_group.Append(cut_diagonal)
            end
        end
    end
    groups.Append(current_cluster);
    return groups
end
```

Algorithm 11: Group By Overlaps - Create Diagonal Subset

```
Function CreateDiagonalSubset(sorted_diagonals, i, j, dim):
    new_length ← sorted_diagonals[i].length;
    if dim = 0 then
        /* HORIZONTAL CASE                                   */
        new_start_i ← sorted_diagonals[i].start_i;
        new_start_j ← sorted_diagonals[j].start_j + (new_start_i -
          sorted_diagonals[j].start_i);
    end
    else
        /* VERTICAL CASE                                     */
        new_start_j ← sorted_diagonals[i].start_j;
        num_steps ← new_start_j - sorted_diagonals[j].start_j;
        new_start_i ← sorted_diagonals[j].start_i + num_steps;
    end
    cut_diagonal ← (new_start_i, new_start_j, new_length);
end
return cut_diagonal
```

Algorithm 12: Merging Related Diagonal Groups

```
Function MergeRelatedGroups(dim0_groups, dim1_groups):
    merged_groups ← [ ];
    group_assignment ← { } ;  /* seen diagonals and their groups
     */
    unique_group_id ← 0;

    all_groups ← dim0_groups ∪ dim1_groups;

    /* 2 groups are connected when they share a diagonal   */

    for group in all_groups do
        connected_groups ← [ ];
        for diagonal in group do
            if diagonal in group_assignment then
                connected_groups.append(group_assignment[diagonal]);
            end
        end
        if length(connected_groups) > 0 then
            /* merge connected groups by reassigning ids     */
            new_group_id ← connected_groups[last];
            for diagonal, group_id in group_assignment do
                if group_id ∈ connected_groups then
                    group_assignment[diagonal] ← target_group;
                end
            end
            for each diagonal in group do
                group_assignment[diagonal] ← target_group;
            end
        end
        else
            /* group not connected to another, give a unique id
               */
            for each diagonal in cluster do
                group_assignment[str(diagonal)] ← unique_group_id;
            end
            unique_group_id ← unique_group_id + 1;
        end
    end
    unique_groups ← UniqueValues(group_assignment);
    for each ug in unique_groups do
        new_group ← [ ];
        for each diagonal, group in group_assignment do
            if group_assignment[i] = ug then
                new_group.append(diagonal);
            end
        end
        clusters.Append(new_cluster);
    end
    Result: merged_groups
end
```