Real-time Audio Tracking Using Reference Recordings

Nathan J. Gutierrez

6.UAP Spring 2017

Abstract

Over 2016 and 2017, the MIT Music Technology Lab created ConcertCue (formally known as NoteStream), a web platform for streaming rich-content program notes to audience cellphones in synchronization with a live musical performance. ConcertCue aims to make the concert experience more informative, enriching, and appealing for novice and experienced listeners alike. At this time, synchronization of the content to the live performance is done manually by a human operator who advances time in the application by tapping along to the beat of the music. This requires additional training, preparation, and can be prone to error. Methods such as dynamic time warping which have traditionally been used for automatically time-aligning two unique recordings of the same piece do not work in real-time because of their time complexity and requirement of complete data. The goal of this project is to implement and build upon existing algorithms in order to create an algorithm that can provide accurate positional estimates of a live musical piece given an input audio stream and a pre-supplied reference recording of the performance. To accomplish this, a variation of dynamic time warping called “online time warping” was implemented and further developed. The resulting algorithm performed decently well with test content, reaching an average accuracy of 87%. Though these results are promising, further improvement would be necessary to completely automate the synchronization process and remove the need for a human operator.

Motivation

In December 2016, the MIT Music Technology Lab debuted NoteStream at the MIT Wind Ensemble Prism II concert. The NoteStream web app provided rich content notes that were synchronized to the music, providing audience members with a live listening guide and

informational facts about the piece's composition and background as it played. NoteStream was again used in March 2017 for Evan Ziporyn's 'Ambient Orchestra', where it was used with David Bowie's Blackstar arranged for orchestra. In both instances, the app was greeted with largely positive feedback.

The content was synchronized to the live performance through a human operator who kept the system synchronized by tapping along to the beat of the live performance. Custom software made use of pre-annotated audio recordings that had a time annotation for each beat. This software sent time position updates to a centralized time server that then updated the subscribed web clients.

Removing the need of a human operator to synchronize the app to the pieces could allow for more accurate tracking, less setup work, and no necessary training, allowing ConcertCue to be used in more venues. With further improvements, the algorithm could be used directly on phones, allowing individuals to have a live listening guide independent of a specific performance of the given piece, allowing ConcertCue to be used at home or independently of a specific concert.

Prior Work

ConcertCue was inspired by LiveNote[1], a similar platform created by the Drexel MET Lab for the Philadelphia Orchestra. Like ConcertCue, LiveNote provides real time information and images about the piece being played by the orchestra. Unlike ConcertCue, LiveNote is a native iOS application that requires connection to a specialized network. On this network a specialized listening machine records the live performance, calculates the current song position in real time, and distributes the position to the clients. The work that went into LiveNote is

summarized in their paper[2], which discusses the techniques used to achieve live time alignment. This paper and its references were used as a guide for implementing our own time alignment algorithm. It should be noted that LiveNote is now using a human operator for synchronization, as is presently done in ConcertCue.

Background

Audio is a one dimensional signal- digital audio is a discrete time signal where each sample corresponds to the recorded audio level at that sample's moment in time. Two distinct recordings of the same piece will have completely unique audio signal representations that cannot be usefully compared directly. Despite this, a human can recognize and correlate different points in the two recordings. With the proper techniques, computers can as well.

Variations in instrumentation, recording techniques, and performance will result in differences of tempo and timbre between two recordings of the same piece. However the two recordings will share the same overall pitch data. By using the short-term fourier transform (STFT), the one dimensional audio signal can be converted to a two dimensional frequency / magnitude signal. This signal can be binned according to classical western pitch structure (12 distinct pitch classes spread over many octaves) becoming a 12 pitch-class / magnitude signal. Each column of this signal is then normalized such that the sum of the rows is 1, completing the process and making it a chromagram[2].

These chromagrams can be used directly as the feature vectors for synchronization using dynamic time warping (DTW)[3]. The first step in DTW is to calculate a cost matrix that represents the local "cost" of moving from an adjacent cell to that target cell. The cost matrix is calculated using the two chromagrams by taking the outer product of the chromagrams and subtracting each value from one. This results in a rectangular cost matrix that has higher values

where the chromagrams are less similar, and lower values where they are more similar. The accumulated cost matrix, which is the same size of the cost matrix, is then calculated; each element is calculated starting at the beginning of the piece (lower left corner of the matrix) and working through to the end of the piece (upper right corner of the matrix). Dynamic time warping then uses this accumulated cost matrix to backtrack and find the optimal path of least cost, which corresponds to the time mapping between the two recordings.[3]

Because the distance is calculated for each location on the NxM accumulated cost matrix, this process takes O(N*M) time, which is too slow for real-time application and also requires knowing the full signals in advance. While there exist techniques for applying global path constraints in the accumulated cost matrix (where the distance cost is not calculated for the entire NxM grid, but rather a small subset around the diagonal) such as Sakoe-Chiba bound[4] these techniques still require knowing the full signal ahead of time in order to know what the slope of this diagonal is.

The Approach

Because data is coming in real time, we needed a way of finding the least cost path in the forward direction. This is achieved using Dixon's "On-Line Time Warping" algorithm[3] as inspiration. Online time warping does not use global path constraints, calculates the minimum cost path in the forward direction, and is computationally linear instead of quadratic.[3] To begin, the data is processed as it would be for dynamic time warping. The short term fourier transform is used to convert the reference audio signal to the frequency domain, and then converted to a chromagram representation. The incoming audio is windowed and has the STFT and chromagram process (with optional exponential smoothing) applied to each incoming window in

real time. The cost matrix is calculated like in DTW, by taking the outer product of the new incoming chromagram column with the existing reference chromagram and subtracting it from one.

The algorithm maintains an x-y cursor into the cost and accumulated cost matrices that corresponds to the most recent location in its estimate of the optimal path. The cursor is also used as the start point for calculating partial rows or columns (where the partial size corresponds to the algorithm's window size, c) in the accumulated cost matrix. The previous positions of this cursor are saved as the optimal path.

As new columns of the cost matrix are calculated, the algorithm calculates partial rows or columns of the accumulated cost matrix using the standard DTW recursion cost formula.

$$D_{x,y} = min(D_{x-1,y-1}, D_{x-1,y}, D_{x,y-1})$$

Where D is the accumulated cost matrix, and (x, y) is the current location.

The algorithm begins by moving the cursor along the diagonal until it has calculated c^2 elements at the beginning. Beyond this initial search area, the algorithm makes use of the minimum path cost (in the accumulated cost matrix). If the minimum path cost occurs at the position of the cursor, the row (y) and column (x) of the cursor are incremented. If the minimum path cost occurs in the current row, but not at the cursor, the row (y) of the cursor is incremented. If the minimum path cost occurs in in the current column but not at the cursor, the column (x) of the cursor is incremented.

Experimental Setup

In order to write, test, and debug the algorithm, a test framework, test data, and the algorithm itself had to be created. This section details the architectural design of the system and

its components. The development and test machine was a 2016 MacBook Pro with 2.7 Ghz Intel i7 processor running macOS 10.12.4 and Python 2.7.13.

Test Data

In order to test the algorithm, a number of orchestrated pieces were collected to test the system. Three versions of each movement were collected; each produced by a different orchestra under differing recording situations and performance speeds. Additionally, two pop songs were included to test the algorithm under the different sonic circumstances that they provide. The selection of pieces is detailed in the following table:

Composer	**Piece**	**Movements**
J.S. Bach	Suite No. 3 in D Major	II. Air
J. Brahms	Symphony No. 3 in F Major	I. Allegro
D. Shostakovich	Symphony No. 5 in D Minor	I. Moderato
A. Vivaldi	Concerto No. 4 in F Minor (“Winter”)	I. Allegro non molto II. Largo (Eb Major) III. Allegro
Pink Floyd	Nobody Home	
Joy Division	Love Will Tear Us Apart	

File preparation for each recording included removal of extraneous sound at the beginning and ending of tracks, as well as conversion to wav format with a sampling rate of 22050 Hz. This sampling rate was chosen in order to preserve sound quality while reducing processing time by two.

Creating Truth

A ground-truth mapping of the individual recordings was created in order to measure the error of the algorithm's estimates. To create this truth map, an open source software tool, Sonic Visualizer, was used to create instant files.

Instant files refer to simple csv (comma separated value) files that contain a line for each beat in the piece. Each line has the time, or instant, in seconds that the beat occurs on and a label (typically the beat number). These files will be of equal length for two recordings of a given piece since they should have the same number of beats. If they were not, due to a repeat or lack thereof, the audio files were edited to include the repeat by simply repeating the unrepeated audio.

At run time, one recording is the input while another is the reference. The instant files of the input and reference are mapped together by their beats, such that if beat 1 in the input is at 1.2 seconds, and beat 1 of the reference is at 2.0 seconds, an input time of 1.2 seconds corresponds to a reference time of 2.0 seconds. Values between individual beats are calculated using linear interpolation, allowing calculation the corresponding reference time for any input time and vice versa.

Test Framework

The purpose of the test framework is to load the reference and input data, provide the algorithm with reference data and a simulated input audio stream, measure the performance of the algorithm, and provide visual and text results of the algorithm's performance. The test framework can be run on individual reference-input pairs or can be run as a test suite where all test data is fed sequentially through the algorithm. This framework uses JSON files for test

configuration, which includes the list of available pieces to run in the suite and algorithm settings. This allows variations of the same algorithm to be tested and compared.

Reference Data

Reference data refers to the audio data and its instants file used as the algorithm's reference; it is a pre-existing recording of the piece that the algorithm has full access to. The instants file is a map of all the beats and their positions in seconds and is used by the test framework- see the "Test Data" section below for more details.

Input Data

The input data refers to the audio data that is fed to the algorithm in real-time and its instants file; the audio is fed to the algorithm as though it were a live input signal during the musical performance, making it a causal system. See the "Test Data" section below for more details.

Testing the Algorithm

To test the algorithm, the test framework first loads and converts the input and reference audio and instants files. The audio data is converted to mono and resampled to 22050 samples / second if it is not already. Next the algorithm under test is initialized with the reference audio data and a callback function for time estimate updates.

The test begins when the input data is fed to the algorithm. To do this, the algorithm iterates through the input data in buffer sized increments, where the simulated buffer size corresponds to the buffer size likely found in the input an actual sound card- in this case 512 samples. Buffer size determines the amount of latency that the system will experience and can

be calculated by dividing the buffer size by the sample rate. In this case, 512/22050 = 23 ms, which is negligible. At each iteration, the framework calculates the actual time using the InputMap class which is discussed in the “Test Data” section. This process repeats until the end of the input data is reached.

Scoring

Each time the algorithm calculates an estimated position it provides this estimate through the callback function provided by the test framework. This callback is the score function which uses the error of the estimate to the known actual time as the input of a gaussian function. The score function is as follows:

$$\varepsilon = t_{actual} - t_{estimate}$$
$$a = 1,\ b = 0,\ c = 3$$
$$Score = a * e^{(\varepsilon - b)^2/(2*c^2)} = e^{(\varepsilon^2/18)}$$

The score is then appended to an array for later use. A standard deviation of 3 seconds was chosen in order to have scoring that would be fairly strict while still allowing for reasonable variation without excessive penalty.

Upon the input’s completion, the mean and standard deviation of the scores are computed in order measure the algorithm’s performance on that reference-input pair. Graphs of the reference chromagram, short-term fourier transform, the calculated cost and distance matrices with overlaid path, and time error vs time are also created to debug and judge the algorithm’s performance.

The Algorithm

At initialization, the algorithm creates the STFT and chromagram of the reference audio of length N. The chromagram is then optionally filtered temporally to reduce the effects of noise and sporadic frequency content. In order to avoid costly array expansions, empty arrays and matrices are made for the input STFT, chromagram, and resulting cost and distance matrices. These empty arrays are twice the size (2N) of the reference arrays in order to not grow the arrays during runtime (which was found to be too computationally expensive). Additionally, smaller buffers are created for the incoming audio. The algorithm then waits to receive samples.

Upon receiving samples, it appends the data to an input buffer. Once this buffer is the size of the STFT window or hopsize, whichever is larger, the algorithm creates a single STFT column and its chromagram. This input chromagram column is normalized, optionally filtered, and the outer product of the column with the reference chroma subtracted from 1 is calculated to determine the next cost matrix column. The algorithm is then run on this cost matrix, populating the accumulated cost matrix and calculating the path at the same time. The algorithm updates the container app with its estimate each time it has calculated a predetermined number of updates.

Filtering of the chromagram was made optional in order to compare the results with and without filtering. Filtering was performed using exponential smoothing. A simple moving average filter was attempted with window sizes of 2 and 4, but its performance was not found to be as good. In our configuration, the filter function was constructed with a smoothing function of ½, such that:

$$y[n] = \tfrac{1}{2}(x[n] + y[n-1])$$

Audio is processed at a sample rate of 22050, which is half “CD quality”. For the STFTs, a hop size of 2048 and windows size of 4096 was chosen as it provided a good balance of performance and processing speed. Chromagrams were both unfiltered and average filtered, corresponding to two variations of the algorithm. The algorithm features a max run parameter to ensure that the algorithm does not step in a given direction for too long. This is currently set to 3 steps, which corresponds to a slope of 3 or ⅓ on the cost matrix. Given that it is unlikely for an orchestra to play a piece 3 times faster or slower than any particular reference recording, it was chosen to allow the algorithm catch-up and slow-down time. The algorithm’s search width, *c*, was set to 100, corresponding to about 10 seconds of audio.

Performance

To gauge the performance of the algorithms, the test suite was run on each piece and each combination of recordings. With five pieces and three recordings each, this resulted in 45 trials when recordings were tested against themselves, and 30 when self tests were excluded. The overall average score is shown in the below table, including with self test (where the reference and input were equal) and without.

Algorithm	**With Self Tests**	**Without Self Tests**
Online Time Warping (No Filter)	88.6% w/ std dev. 15.7%	83.6% std dev. 20.6%
Online Time Warping (Chromagram Filter)	91.2% w/ std dev. 13.2%	87.5% std dev. 17.0%

Full Data

The left columns correspond to Online Time Warping with no filtering, while the right columns correspond to Online Time Warping with chromagram filtering.

Vivaldi - Four Seasons (Winter) - Movement 1

		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
0	0	00:06.7	99.5%	3.1%	3.04		00:07.2	99.4%	3.1%	3.04
0	1	00:07.1	88.8%	22.2%	-7.82		00:07.3	95.7%	10.5%	-3.84
0	2	00:06.9	88.5%	22.0%	-6.57		00:07.3	92.1%	18.9%	-5.41
1	0	00:07.5	87.9%	21.6%	7.88		00:06.8	95.2%	8.8%	3.58
1	1	00:07.4	99.8%	0.6%	1.29		00:06.9	99.7%	0.5%	1.20
1	2	00:07.2	91.7%	14.3%	4.16		00:07.0	97.8%	4.2%	-2.15
2	0	00:07.3	88.7%	20.5%	6.13		00:07.0	92.5%	16.8%	5.17
2	1	00:07.8	92.5%	14.6%	-4.37		00:07.1	98.9%	2.2%	1.59
2	2	00:07.0	99.5%	3.1%	3.05		00:07.2	99.4%	3.1%	3.04

Vivaldi - Four Seasons (Winter) - Movement 2

		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
0	0	00:04.2	95.5%	17.9%	8.83		00:04.3	95.5%	17.9%	8.83
0	1	00:04.6	90.7%	17.1%	6.04		00:04.9	94.1%	8.0%	4.28
0	2	00:04.4	96.0%	7.6%	-3.25		00:04.4	97.8%	5.2%	3.16
1	0	00:04.3	86.4%	15.5%	-4.56		00:04.7	88.0%	10.3%	3.06
1	1	00:04.8	99.5%	2.9%	2.69		00:05.0	99.4%	2.9%	2.69
1	2	00:04.7	81.6%	25.0%	-7.62		00:04.6	92.0%	13.4%	-3.74
2	0	00:04.5	92.6%	12.5%	3.86		00:04.3	93.5%	10.4%	3.38
2	1	00:04.8	84.9%	23.9%	9.60		00:05.1	94.5%	9.8%	3.14
2	2	00:04.4	99.4%	3.1%	2.73		00:04.6	99.4%	2.9%	2.64

Vivaldi - Four Seasons (Winter) - Movement 3

		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
0	0	00:06.2	99.7%	1.5%	-2.04		00:07.2	99.6%	1.5%	-2.04
0	1	00:06.7	57.8%	43.6%	33.85		00:07.5	67.6%	39.6%	23.30
0	2	00:06.5	87.6%	23.5%	9.34		00:07.1	90.6%	20.9%	8.23
1	0	00:06.2	57.4%	43.7%	-38.04		00:06.8	66.4%	39.4%	-29.33
1	1	00:06.7	99.8%	0.1%	-0.37		00:07.0	99.7%	0.1%	-0.28
1	2	00:06.5	81.2%	26.2%	6.71		00:06.6	82.1%	23.9%	-6.00
2	0	00:06.3	87.1%	23.4%	-11.45		00:06.2	88.7%	21.3%	-10.21
2	1	00:06.7	83.5%	23.0%	-7.30		00:07.6	84.9%	21.5%	-6.00
2	2	00:06.5	99.2%	5.2%	4.03		00:06.4	99.1%	5.1%	4.03

Shostakovich - Symphony No. 5 - Movement 4

		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
0	0	00:23.2	99.8%	0.1%	0.59		00:25.1	99.8%	0.1%	0.59
0	1	00:25.5	89.3%	22.9%	10.34		00:26.3	87.4%	25.9%	10.15
0	2	00:28.5	74.1%	34.1%	13.62		00:25.5	58.3%	44.7%	43.45
1	0	00:25.7	89.9%	22.5%	-16.33		00:23.6	88.2%	25.2%	-16.20
1	1	00:26.9	97.0%	15.8%	21.31		00:26.2	97.0%	15.8%	21.22
1	2	00:27.3	85.7%	24.2%	7.70		00:29.0	83.8%	27.7%	13.23
2	0	00:25.6	75.9%	35.7%	-25.69		00:25.7	69.1%	41.9%	-74.73
2	1	00:27.6	83.7%	27.1%	-12.69		00:26.1	82.6%	29.3%	-15.37
2	2	00:27.3	99.8%	0.1%	0.75		00:27.4	99.7%	0.1%	0.75

Bach - Orchestral Suite No. 3 - Movement 2

		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
0	0	00:09.8	95.4%	19.5%	16.40		00:10.3	95.4%	19.5%	16.30
0	1	00:10.4	88.1%	20.4%	15.65		00:11.2	90.0%	19.0%	15.56
0	2	00:11.1	96.6%	4.6%	2.21		00:12.2	95.1%	7.7%	3.11
1	0	00:11.0	84.2%	22.8%	19.57		00:10.8	84.9%	21.8%	19.57
1	1	00:11.1	94.7%	21.1%	19.47		00:11.4	94.7%	21.1%	19.38
1	2	00:11.5	34.3%	39.3%	54.46		00:12.3	76.1%	28.6%	7.60
2	0	00:10.4	92.5%	9.0%	-3.46		00:11.3	89.2%	13.3%	-4.31
2	1	00:10.9	31.4%	38.3%	-47.97		00:11.2	74.0%	32.2%	-9.56
2	2	00:11.6	99.8%	0.0%	-0.28		00:12.2	99.8%	0.1%	-0.28

Brahms - Symphony No. 3 in F Major - Movement 1

		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
0	0	00:33.3	99.7%	1.1%	-2.41		00:35.7	99.7%	1.1%	-2.41
0	1	00:37.6	84.2%	22.3%	7.17		00:39.9	83.8%	21.7%	7.72
0	2	00:33.2	96.6%	6.9%	3.56		00:34.4	97.2%	4.7%	-2.56
1	0	00:35.0	76.6%	27.9%	-9.03		00:35.4	75.6%	27.6%	-9.21
1	1	00:38.8	99.8%	0.0%	0.44		00:39.3	99.8%	0.1%	0.44
1	2	00:34.3	83.9%	19.6%	-7.17		00:34.7	78.7%	23.7%	-8.40
2	0	00:35.4	96.2%	8.0%	-4.92		00:33.5	97.0%	5.8%	-3.34
2	1	00:38.2	90.6%	13.2%	5.06		00:37.8	87.1%	17.3%	6.24
2	2	00:34.1	99.7%	1.7%	-3.24		00:32.6	99.7%	1.7%	-3.24

Pink Floyd - Nobody Home

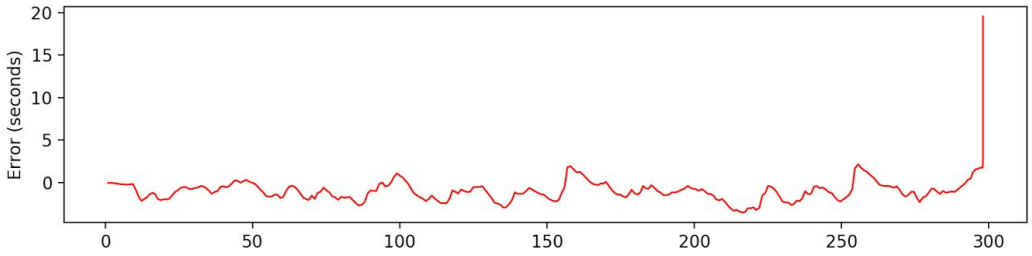
		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
Original	Original	00:06.1	99.7%	1.2%	1.83		00:06.5	99.6%	1.1%	1.74
Original	Metric Cover	00:06.5	85.4%	19.7%	5.88		00:06.7	89.5%	12.2%	-4.19
Original	BritFloyd Cover	00:06.3	91.8%	12.4%	4.08		00:06.3	96.7%	6.8%	2.71
Metric Cover	Original	00:06.1	84.8%	20.2%	-5.97		00:06.1	89.5%	13.7%	3.86
Metric Cover	Metric Cover	00:06.3	95.0%	19.9%	12.53		00:06.4	95.0%	19.8%	12.44
Metric Cover	BritFloyd Cover	00:06.1	93.6%	10.7%	-3.66		00:06.1	94.5%	9.2%	-3.29
BritFloyd Cover	Original	00:06.1	89.6%	13.7%	-4.47		00:06.0	96.0%	8.2%	-3.27
BritFloyd Cover	Metric Cover	00:06.2	91.1%	11.1%	4.02		00:06.2	94.3%	7.9%	2.77
BritFloyd Cover	BritFloyd Cover	00:06.0	99.8%	0.5%	1.22		00:06.6	99.7%	0.6%	1.22

Joy Division - Love Will Tear Us Apart

		No Filter					Filter			
Reference	**Input**	**Runtime**	**Score**	**Std. Dev**	**Worst Error**		**Runtime**	**Score**	**Std. Dev.**	**Worst Error**
Original	Original	00:06.7	99.8%	0.2%	-0.82		00:07.3	99.8%	0.3%	-0.82
Original	The Cure Cover	00:07.7	88.8%	17.5%	-6.30		00:08.1	94.6%	4.5%	2.05
Original	David Gahan Cover	00:07.2	86.6%	12.3%	-4.10		00:07.2	92.5%	5.6%	-2.29
The Cure Cover	Original	00:07.3	85.7%	19.7%	6.48		00:07.0	89.1%	7.9%	-2.66
The Cure Cover	The Cure Cover	00:08.6	98.3%	10.2%	6.60		00:07.5	98.2%	10.0%	6.60
The Cure Cover	David Gahan Cover	00:07.3	67.7%	23.2%	7.16		00:06.5	73.7%	16.2%	6.42
David Gahan Cover	Original	00:06.9	90.7%	9.5%	3.52		00:06.8	96.5%	3.3%	1.78
David Gahan Cover	The Cure Cover	00:07.3	76.0%	20.9%	7.60		00:08.5	82.2%	15.9%	7.97
David Gahan Cover	David Gahan Cover	00:06.7	97.5%	13.2%	7.97		00:07.3	97.5%	13.1%	7.97

Discussion

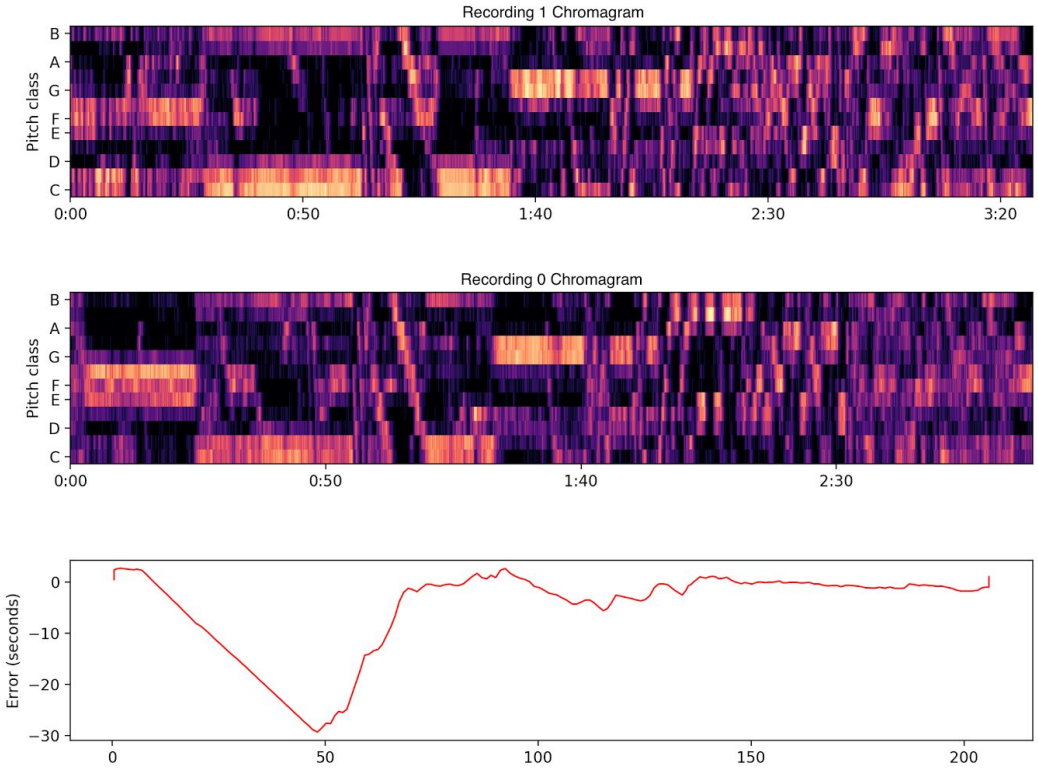
The “Worst Error” metric doesn’t paint a particularly useful picture, as a lot of the times this value grew very large at the end of pieces. The following error plot demonstrates this issue in the case of Bach 1 vs 0. Here the algorithm performs very well up until the very end, where the error grows very fast..



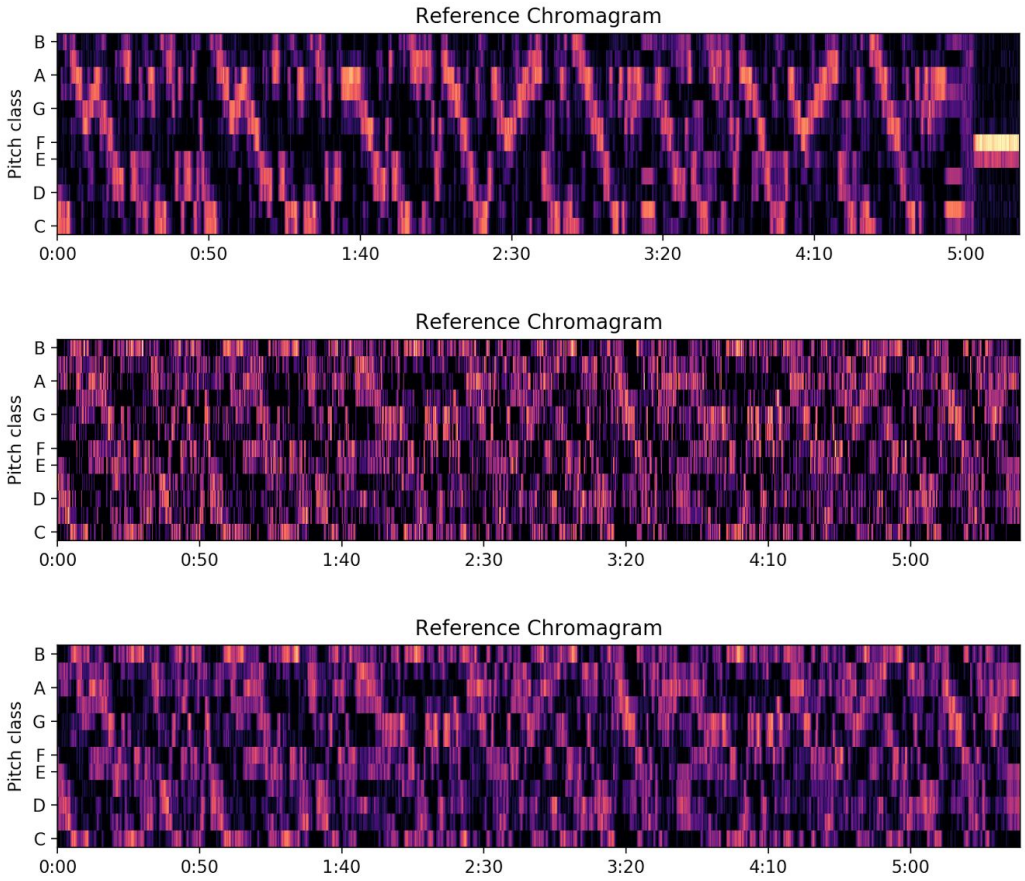
Upon closer inspection these types of errors seem to occur is silence at the end of the piece. Solving this problem with the current algorithm would require adding a lot more intelligence to it (such as not continuing unless the signal is greater than a given noise floor). Gating techniques may be tricky given the huge dynamic range of a typical orchestra. Likely the easiest solution is to start and stop the algorithm as close as possible to the beginning and end of the piece.

The algorithm tended to suffer at times when the chromagram remained static- (that is the same pitch content repeated for some amount of time). At these times the algorithm's estimation was significantly faster or slower than the reference time. Despite this, that algorithm demonstrated fairly good recovery. This is particularly observable in the third movement of Vivaldi’s Winter, with recording 1 vs 0. The below graphs, starting from top show the filtered chromagram of recording 1, the filtered chromagram of recording 0, the error of 1 vs 0, and the cost matrix cropped to the first minute of the piece. The error builds up when there is little

change in the chromagram because the cost matrix for these areas tend to have very poorly defined features- one would expect to see a diagonal of low cost, but instead there are horizontal and vertical bands. As soon as the chromagram begins to change, the cost matrix begins to have a more distinct diagonal and the the algorithm begins to catch up.



Filtering the chromagram provided anywhere from a slight improvement (1%) to significant improvement (30% with the Bach 1 vs 2) to the overall scores while very minimally impacting performance (generally adding only a second to runtime). Chromagram filtering was most beneficial when the source audio was noisy. Noise contributes energy to the entire frequency spectrum, making the chromagram less distinct. This can be seen in the below three chromagrams of Bach's Movement 2 of Orchestral Suite No. 3. The topmost filtered chromagram is a clean recording of the piece, the middle unfiltered chromagram and bottom filtered chromagrams are from a noisy recording of the same piece.



The filtered version of the noisy chromagram performed much better, providing a 30% increase to the overall score. Better filtering of noise would likely return even better results and should be pursued in later implementations of this algorithm.

The average runtime of both algorithms was significantly faster than real-time, processing with an average rate of 26.5 input seconds / real second for both algorithms. This provides ample room for more complex processing of the signal.

Future Work

With a large portion of the work having already gone into the test framework and test data, next steps can focus more on implementing and testing a variety of algorithms. Given that the field of music technology is currently active and always finding improvements, there may exist better algorithms already and in the near future.

The algorithm should be tested under a wider variety of more real world conditions. This may include simulating noise, audience, delays in start, nonlinear effects, and other similar factors. It should also be tested with an actual microphone and playback system in order to determine the effects of signal distortion.

Conclusion

Though online time warping worked decently, it is still too early to have the system completely rely on an automated process. With better signal processing and more intelligent algorithms, the goal of automating ConcertCue may become more realistic. With some modification, the algorithm could potentially be used in conjunction with a human operator who could override it if performance becomes too poor. This defeats the goal, but would provide useful real world data for algorithm weaknesses.

References

1. *LiveNote*. Computer Software. *Apple App Store.* Vers. 1.2.0. The Philadelphia Orchestra Association, 15 Apr. 2015. Web. 1 May 2016. https://itunes.apple.com/us/app/livenote/id825459337?mt=8
2. Prockup, M., et al. "Orchestral Performance Companion: Using Real-Time Audio to Score Alignment." *IEEE Multimedia*, vol. 20, no. 2, 01 Apr. 2013, pp. 52-60. EBSCO*host*, doi:DOI: 10.1109/MMUL.2013.26.
3. Müller, M., *Fundamentals of Music Processing*. Springer, 2015.
4. H. Sakoe and S. Chiba, “Dynamic programming algorithm Optimization for Spoken Word Recognition,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, pp. 43-49, 1978.
5. Dixon, S., “Live Tracking of Musical Performances Using On-Line Time Warping.” *Proc. of the 8th Int. Conference on Digital Audio Effects*, Sept. 2005, pp. 92-97.