Real-time In-browser Time Warping for Live Score Following

Matthew Caren
Massachusetts Institute of Technology
Cambridge, MA
mcaren@mit.edu

Eran Egozy
Massachusetts Institute of Technology
Cambridge, MA
egozy@mit.edu

ABSTRACT

To date, implementations of score-following algorithms have largely been exclusive to standalone computer applications. However, web browser-based systems offer several advantages over standalone applications, including cross-platform compatibility, straightforward setup, and greater accessibility. We present an implementation of a real-time score following algorithm using Online Time Warping (OTW) that runs exclusively in the browser, developed as part of *ConcertCue*, an application that synchronizes content delivery to live musical performances. This OTW implementation was developed in Python, yet runs in the browser without modification, using a framework we developed for shuttling audio and control data between native JavaScript and Pyodide, a browser-based Python interpreter. We discuss the performance of our implementation and the generalizability of this approach to developing other real-time audio analysis applications using Python in the browser. In addition to releasing the source code, we also release the hand-annotated audio alignment dataset used to test the system.

1. INTRODUCTION

Score following (also referred to as "audio synchronization", "audio-to-score alignment", or "live tracking") is the task of identifying temporal location in a musical performance by establishing a mapping to corresponding locations in a reference track. In this paper we focus on the case of real-time score following, where we seek to continuously estimate a correspondence between live audio input (hereafter the "live" datastream) and a prerecorded audio file of the same piece (hereafter the "reference" datastream).

Approaches to score following can generally be separated into two components: feature representation, and alignment technique. Feature representations can be either symbolic or audio-derived. Symbolic representations, most prominently MIDI and MusicXML, are reliable and tend to be more computationally lightweight to process, and are thus prevalent in systems that demand high precision and in earlier score-following implementations restricted by limited computing power [6, 1]. However, they require input data to be provided symbolically (e.g. via MIDI keyboard or external trigger mechanism) which significantly narrows practicable use cases. Audio representations, in contrast, rely on features computed directly from time-based audio data—spectral information, chromagrams, loudness, etc.—and allow for a much broader domain of inputs, though are computationally more expensive to extract and process and can sometimes be less reliable with polyphony and complex timbres [13, 2].

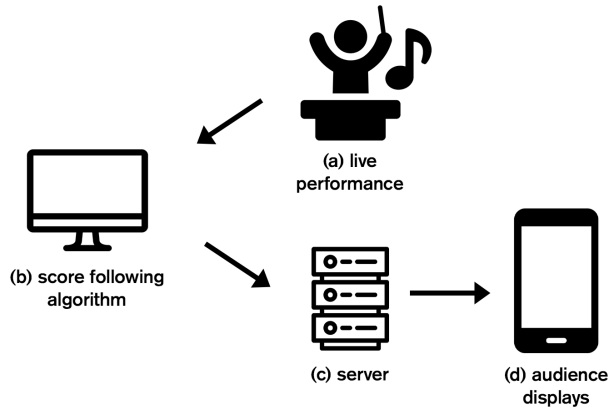

Figure 1: The *ConcertCue* system: audio from a live performance (a) is sent to the score following algorithm (b). Timing data is streamed to a cloud-based server (c) which in turn synchronizes content in real-time on audience display devices (d).

To align audio-based features in real-time, stochastic models like Hidden Markov Models are often used [14, 5], as well as time warping algorithms and variants like Online Time Warping [7], which is used in this paper and will be discussed in greater detail in Section 2.

Regardless of the specific algorithm, score-following applications to date have been largely implemented in standalone computer applications, either on their own or within other applications like PureData or Max/MSP [1]. Though some implementations of web-based score-following exist [4, 8], their performance and robustness are inferior to their standalone counterparts, requiring monophony or relying on substantial user interaction to correct errors.

Adopting an exclusively in-browser approach offers a number of advantages: broad multi-platform compatibility, lack of an installation and setup process, accessibility (with no reliance on proprietary software like Max/MSP), easy distri-

Web Audio Conference WAC-2024, March 15–17, 2024, West Lafayette, IN, USA.

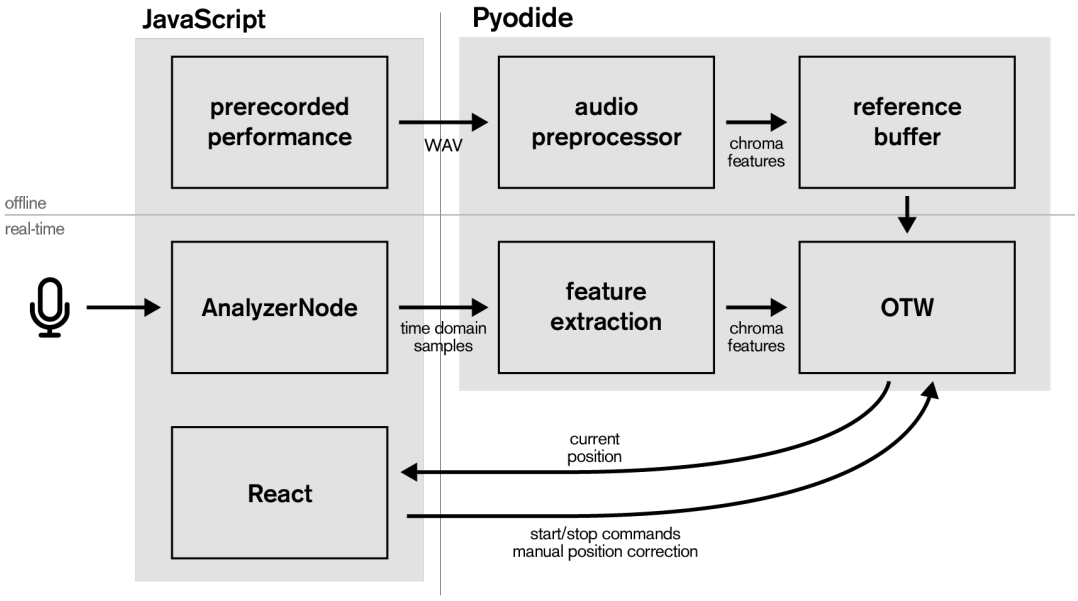

Figure 2: System architecture: the score-following system consists of a native JavaScript/React component that manages audio input, user interaction, and user-facing displays, as well as a Pyodide Python component that extracts audio features and runs the OTW alignment algorithm.

bution and updating, and capability to leverage existing web tools. For these reasons, browser-based versions of audio applications have become increasingly ubiquitous for audio synthesis, audio processing [10], and recording[1]. However, online score-following remains relatively overlooked.

In this paper, we present an implementation of a robust score-following algorithm that runs in real-time from within the browser. It was developed for *ConcertCue*,[2] a web application that streams program notes, supertitles, and other content to audience-facing devices in real-time during a concert. Real-time score following allows this content to be automatically synchronized with a live musical performance, according to the system setup detailed in Figure 1.

2. ONLINE TIME WARPING

Time warping is a widely-used approach for aligning two similar datastreams [9, 3]. The original Dynamic Time Warping (DTW) algorithm is an offline alignment algorithm that takes two temporal sequences $U = u_1, u_2, \ldots, u_m$ and $V = v_1, v_2, \ldots, v_n$ and creates a mapping (or "path") $W = W_1, W_2, \ldots, W_l$, where a given $W_k := (i_k, j_k)$ represents the alignment of u_i with v_j. Each match W_k is evaluated using a local cost function $d_{U,V}(i, j)$, which is 0 for a perfect match and positive otherwise, and the overall path cost is:

$$D(W) = \sum_k d_{U,V}(i_k, j_k)$$

Cost values are stored in an accumulated cost matrix where each element corresponds to the cost of the optimal path up to that position. After computing the entire matrix, the optimal path is found by backtracking from the end of the matrix to the start via the minimum cost measurements.

[1]online DAWs like SoundTrap, BandLab, and Audiotool
[2]https://concertcue.org and https://concertcue.com

In [7], Dixon presents an online variant of DTW called Online Time Warping (OTW), which is capable of aligning sequences in real-time using a piecewise variant of the same technique (which is therefore not guaranteed to be optimal). While the reference datastream is fully known in advance, audio samples from the live datastream only become available in real-time. The algorithm processes each new frame of audio into a feature representation, triggers the calculation of new elements in the cost matrix, and then determines the most likely path alignment up to that point. Unlike DTW, which has $O(n^2)$ time and space complexity, OTW is linearly complex in time and space, and builds the alignment path incrementally as new information is received.

In our implementation, we further adapt Dixon's real-time variant of OTW for use in a real-time web context. Firstly, we use a different set of features, where each u_i and v_j is a CENS (chroma energy normalized statistics) 12-vector that represents the distribution of spectral energy across the pitch classes [12]. We extract these CENS vectors from windowed segments of both the reference and live input audio samples using specified window and hop sizes. However, for the live input sequence we also no longer maintain a constant hop size, as timing variability is introduced by the signal processing pipeline (more discussion in Section 3.1). The local error for a matching is defined using cosine distance:

$$d_{U,V}(i_k, j_k) = 1 - \frac{u_i \cdot v_j}{||u_i||\ ||v_j||} = 1 - u_i \cdot v_j$$

Note that CENS vector normalization results in $||u_i|| = ||u_j|| = 1\ \forall\ i, j$. To stabilize the estimated output, we clamp OTW predictions to be monotonically non-decreasing.

3. BROWSER IMPLEMENTATION

The time-warping and feature extraction algorithms are implemented in the browser using Pyodide, a CPython distri-

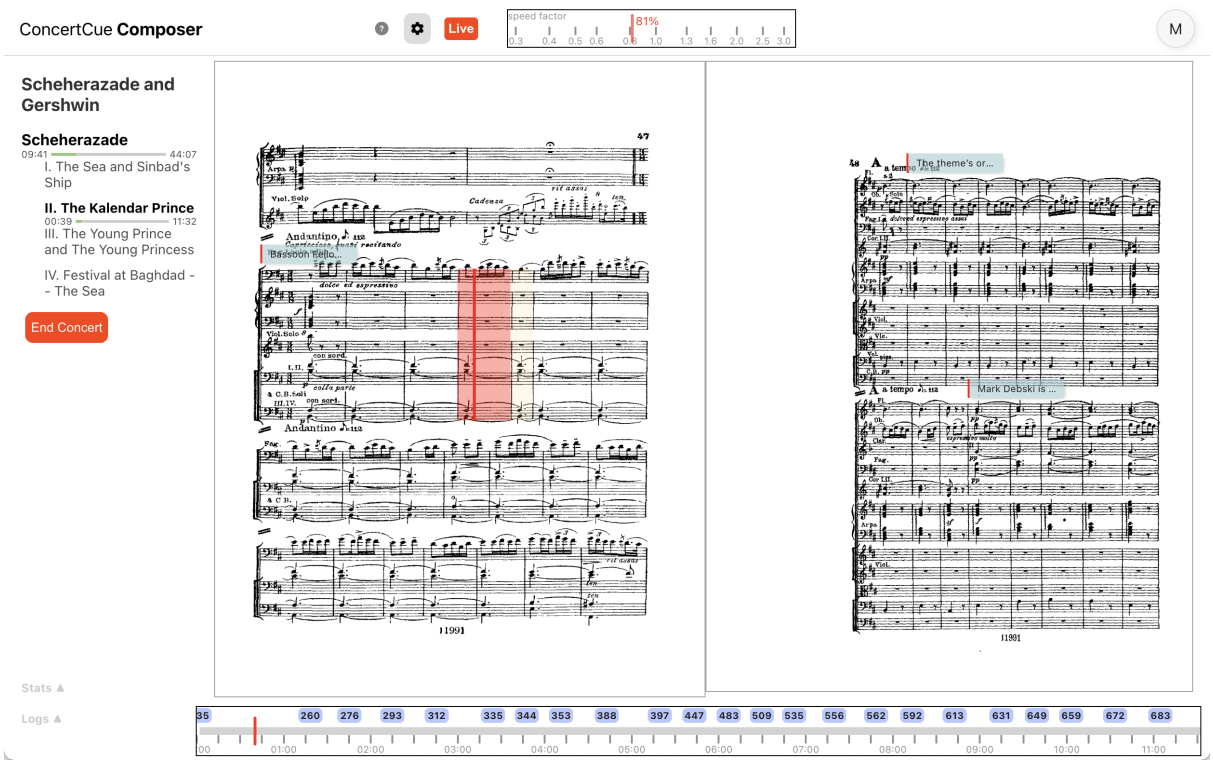

Figure 3: *ConcertCue* directing view: A score is displayed with the current position highlighted in red, as well as the current piece, movement, and speed relative to the reference recording.

bution that targets the browser via WebAssembly [15]. A Web Audio API[3] AnalyzerNode is used for real-time audio sample management, and React[4] for event handling and UI. Figure 2 shows the overall system architecture. The complete score-following system is available as a standalone web app, with all code available on GitHub.[5]

Migrating the score following application from a standalone Python environment to the browser required several modifications, many of which trade off between latency, accuracy, and computational cost.

3.1 Processing Pipeline

First, a reference audio file is loaded into the system, which is passed into the MEMFS filesystem within Pyodide and asynchronously preprocessed to extract the CENS features from the entire duration of the file. This sequence of chroma vectors is then stored in a reference buffer to be used later by the real-time portion of the OTW algorithm.

Live audio input is managed in the native JavaScript environment via the Web Audio API, which retrieves the microphone audio input stream and continually passes buffers to Pyodide as JSProxy sample arrays. We trigger computation by maintaining a JavaScript-side `setInterval()` callback timed to the desired live input hop size. Because the construction of the Web Audio API only permits buffer requests to read the last N samples, we read from the AnalyzerNode's running sample buffer maintained for FFT calculation and set its window size to the desired value (we do not directly use the FFT result in the score-following system, but it is preserved for visualization purposes).

Note that the standard OTW algorithm assumes a fixed hop size, which we do not obey in our web implementation due to the imprecision of timed callbacks in JavaScript. We observed the actual hop lengths to be stochastic but predictably distributed: Figure 4 displays actual measured hop lengths when the specified hop size is 84ms, which is well-

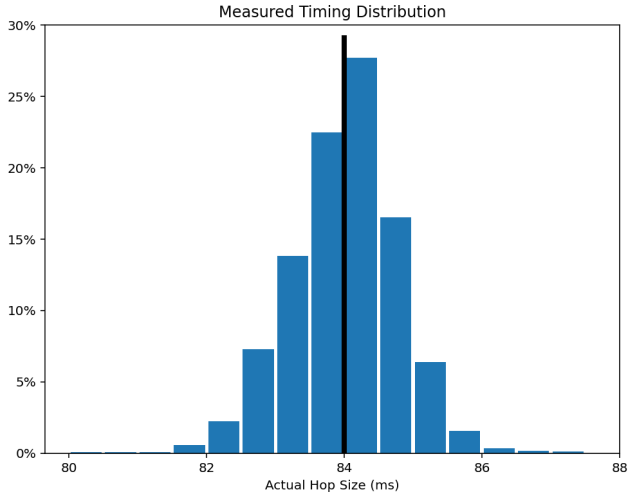

Figure 4: Distribution of measured actual real-time hop sizes with a specified hop size of 84ms (N=6113 measurements)

[3]https://www.w3.org/TR/webaudio
[4]https://react.dev
[5]https://github.com/matthewcaren/web-score-following

Piece	Reference	Live	Median	95th	Max	Mean
Dvořák, String Quartet No. 12	Cleveland	Emerson	0.062	0.521	2.549	0.137
Dvořák, String Quartet No. 12	Cleveland	New York	0.119	0.781	1.510	0.223
Dvořák, String Quartet No. 12	Emerson	New York	0.101	0.398	2.007	0.153
Mozart, Dies Irae	Abbado	Bernstein	0.076	0.450	0.560	0.128
Mozart, Dies Irae	Abbado	Schreier	0.095	0.334	0.539	0.118
Mozart, Dies Irae	Bernstein	Schreier	0.079	0.383	0.770	0.123
Mozart, Symphony No. 40	Bernstein	Klemperer	0.053	0.210	4.227	0.126
Mozart, Symphony No. 40	Bernstein	Salemkour	0.100	0.732	2.035	0.176
Mozart, Symphony No. 40	Klemperer	Salemkour	0.083	1.264	3.909	0.241
Beach, Pastorale	Borealis	Gianopoulos	0.172	1.378	4.283	0.399
Beach, Pastorale	Borealis	RSC	0.176	1.799	3.632	0.401
Beach, Pastorale	Gianopoulos	RSC	0.125	1.229	2.836	0.297
Prokofiev, Piano Concerto No. 1	Argerich	Berman	0.059	0.538	1.754	0.122
Prokofiev, Piano Concerto No. 1	Berman	Kissin	0.059	0.384	1.933	0.113
Prokofiev, Piano Concerto No. 1	Argerich	Kissin	0.078	0.418	1.522	0.123
Tchaikovsky, Nutcracker Suite	Alphen	Ormandy	0.052	0.221	1.417	0.085
Tchaikovsky, Nutcracker Suite	Alphen	Rostropovich	0.066	0.304	0.594	0.090
Tchaikovsky, Nutcracker Suite	Ormandy	Rostropovich	0.052	0.166	1.125	0.073
Overall (Browser)	-	-	0.089	0.639	2.067	0.174
Overall (Simulated)	-	-	0.077	0.537	1.280	0.130

Table 1: Browser system alignment error metrics (all in seconds) per reference-live performance pair, as well as averaged metrics for both browser and simulated systems.

approximated by a normal distribution with $\mu = 84$ms and $\sigma = 0.88$ms.

Passing audio buffers from JavaScript into the Pyodide environment also incurs latency—in software tests, this was generally found to be linear with respect to the buffer size with a constant overhead of 2 ± 0.3 ms (we use a buffer size of 8192, with which we observe approximately 13ms of latency).

3.2 ConcertCue

Prior to a performance that employs *ConcertCue*, a musical score and reference audio track of a piece are uploaded and aligned such that each measure of the score is matched to a location in the audio. In addition, a timeline of events composed of snippets of text, images, and animations is authored to align with important musical moments of the piece. When the piece is performed live, these events are then displayed in real-time on target devices such as personal mobile devices or screens in the concert hall.

Before the present implementation, a musically literate human operator was responsible for matching the live performance to the score by watching a "directing view" (see Figure 3) and tapping a computer key at the start of every measure of the piece—a tedious and stressful task. With the integration of real-time score-following, the system synchronizes the music timeline to the live performance automatically, thus removing the need for constant human attention.

3.3 A Python-based Web MIR Framework

The architecture developed for this score-following application was designed to be as modular and flexible as possible, so that any real-time Music Information Retrieval (MIR) algorithm written in Python can easily run in a web browser.

Although programming JavaScript applications for the web offers many user-side advantages, Python is a much more popular and fully-featured development language for MIR tasks. A significant amount of current MIR research is conducted in Python, which is enabled by powerful mathematical libraries like NumPy and SciPy, and MIR-specific libraries like librosa [11]. Furthermore, Python is commonly used in reference literature, such as [12], which is accompanied by a set of Python MIR notebooks. The presented system enables the combination of the ubiquitous Python development environment with the many advantages of a web-based platform.

At a high level, the system consists of a JavaScript component that routes audio signals, alongside three Python components respectively triggered: (1) on page open/initialization, (2) continuously, on each new buffer of input audio samples as dictated by a specifiable hop size, and (3) on user input, such as "stop" or "reset" actions. Data resulting from Python processing is then passed back to JavaScript with minimal latency (<10ms).

The modularized framework to accomplish these tasks is open-source and accessible alongside the score-following implementation in the accompanying code repository.

Although the presented algorithm was not significantly affected by the variable hop length, there are applications where greater precision may be required. We found that an effective and computationally cheap approach was to use hardware timestamps on each new audio buffer request to estimate the number of elapsed audio samples since the last buffer request. This input buffer synchronization approach achieves a sample-correct accuracy of 92%–97%, depending on the browser.[6] Applications that demand 100% accu-

[6]some browsers reduce audio timer precision as a security safeguard; this does not exceed 2ms on any modern browser, but does reduce alignment accuracy if enabled

rate alignment (or strict isolation of new samples) can use a pattern-matching algorithm, although this would incur additional computational cost.

4. RESULTS

The system was evaluated against a corpus of six Western classical musical pieces, each represented by three different performances, for a total of eighteen recordings. For each recording, precise locations of downbeats (the first beat of each measure) were hand-annotated. In a trial, the alignment algorithm is run with one recording used as the reference data and another as the live datastream. For each piece, three permutations of pairs of recordings was tested in both the real-time browser environment and a native Python OTW simulation, resulting in over 4,000 measures automatically aligned by each system. The system was tested in most common web browsers (Chrome, Firefox, Safari, Edge), and performance was found to be essentially identical; the tests below were performed using Google Chrome on macOS.

In the browser test, live audio is played from a digital audio workstation and routed through a virtual audio input connection to the web browser. "Time zero" of the live input is set to be the time of the first non-zero input sample received, which enables a sample-precise and reproducible testing environment.

Each execution of a score-following system (both in-browser and simulated) generates a W path stored as matchings of live input stream timestamps to estimated reference recording timestamps. These are represented as a list of pairs $W_k = (i_k, j_k) = (t_{\text{live}}, t_{\text{ref}})$, with approximately 12 pairs per second of live audio given our average hop size of 84ms. This path is then evaluated with respect to the ground truth downbeat annotations to arrive at a set of statistics per trial run, as follows:

For each downbeat timestamp of the live datastream, we find the nearest point t_{live} in the generated path and its matching estimate t_{ref}. The error associated with each downbeat is the absolute value of the difference between t_{ref} and corresponding downbeat timestamp in the reference. For each trial, we compute the mean, median, 95th percentile, and maximum error of all downbeat estimates.

Table 1 shows the resulting metrics for the in-browser tests across all trials, as well as the average metrics for the in-browser tests and the Python simulation tests. Figure 5 visually compares the performances of the two systems. While the in-browser system did not perform quite as well as the simulated system—which we attribute to the variance introduced by the in-browser environment—it was consistently within a fraction of a second from the correct alignment, which is sufficient for most score-following applications, including the *ConcertCue* system.

Across all trials of the browser system, we observed a mean alignment error of 0.147 seconds, but a median error of only 0.089 seconds. An example error distribution, typical of almost all tests, is given in Figure 6. The path corresponding to these error measurements is shown in Figure 7.

These figures show that tracking is extremely accurate almost all the time, but has an occasional moment of a fairly large error. We observed that the algorithm occasionally "gets stuck" on a particular segment and fails to progress to the next corresponding position in the reference stream. Within a few seconds, the path then abruptly catches up to the correct alignment, leaving behind a large error spike.

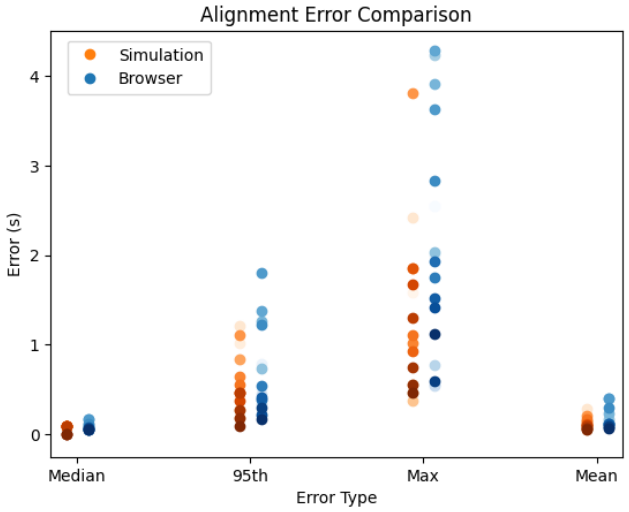

Figure 5: Comparison of the alignment error metrics for in-browser trials versus Python-only simulation trials.

Parameter	Value
Audio sample rate	44100 Hz
Reference hop size	4096 samples
Live hop size	3686 samples
CENS FFT window size	8192 samples
OTW "c"	300
OTW "Max Run Count"	3
OTW "Diagonal Weight"	0.4

Table 2: Tuned algorithm parameters

This behavior can be seen slightly past halfway in the alignment path at 315 seconds, with the corresponding spike in the error plot at measure 240. While the error attributed to this behavior pattern is slightly worse in-browser, it appears nearly identically in the Python simulations as well. We therefore conclude that this behavior is a property of the algorithm itself as opposed to a porting artifact.

Lastly, our OTW implementation has several free parameters that must be properly tuned and are described in Table 2. We chose these parameters by running the Python OTW simulation against a training set of pieces via a grid-search. In the parameter-tuning process, we observed several trends: first, that the best performance is *not* achieved with equal hop sizes for the reference and live sequences; rather, there is a distinct "sweet spot" of best performance where the live hop size is 80%-90% of that of the reference. Second, that a lower diagonal weight is highly correlated with lower overall error, as it discourages the aforementioned "stuck" behavior. The audio sample rate (22 kHz vs. 44.1 kHz), CENS FFT size, reference hop size, and other OTW parameters were far less influential on the measured performance of the system.

5. NEXT STEPS

Our next steps are to complete more extensive testing and performance analysis, as well as implement algorithmic improvements. As part of the *ConcertCue* project, we have accumulated over 20 hours of time-aligned audio and score

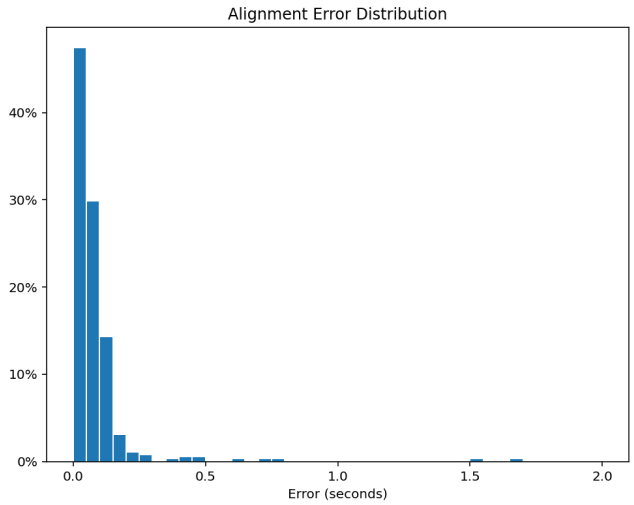

Figure 6: Error histogram for an example trial run of Mozart Symphony No. 40, mvt. 1; Reference: Bernstein, Live: Klemperer.

data (which we hope to publish as a complete dataset in the near future), which could be used to perform a more robust analysis of the system's performance.

Several improvements to the algorithm could further increase accuracy and reliability. There is often a slight difference in global pitch tuning between the reference recording and live performance. Chroma features are particularly sensitive to these differences, and we observed that several of our less-accurate trials involved such tuning discrepancies. To address this, we plan to incorporate a tuning measurement that accompanies the reference chroma features, and a real-time tuning estimator for the live features. Another improvement is to implement the concept of "momentum" by considering a weighted average of estimated position time-derivatives. This could help reduce momentary errors in alignment and make the tracking smoother, as the tempo ratio between live and reference audio is typically a smooth and slowly varying function (i.e. the tempo of a given measure in a performance is often very similar to the tempo of the preceding measures).

Though this system was built to implement score-following with OTW, the overall framework—which is open-sourced and supports real-time audio input, stateful computation with Python/NumPy, and live user input handling—can be generalized to run a wide variety of real-time MIR algorithms in the browser. It streamlines the development-to-production pipeline, allowing for direct transfer from common Python development environments directly to the browser without porting to a different language and with minimal refactoring. With this implementation, we hope to not only advance the feasibility of robust score-following in the browser, but also of the practicality of real-time MIR in the browser as a whole.

6. ACKNOWLEDGEMENTS

We would like to thank Kai Xu and Iris Yang for their help in creating the downbeat annotation dataset and Nathan Gutierrez for his extensive work on *ConcertCue* code and design.

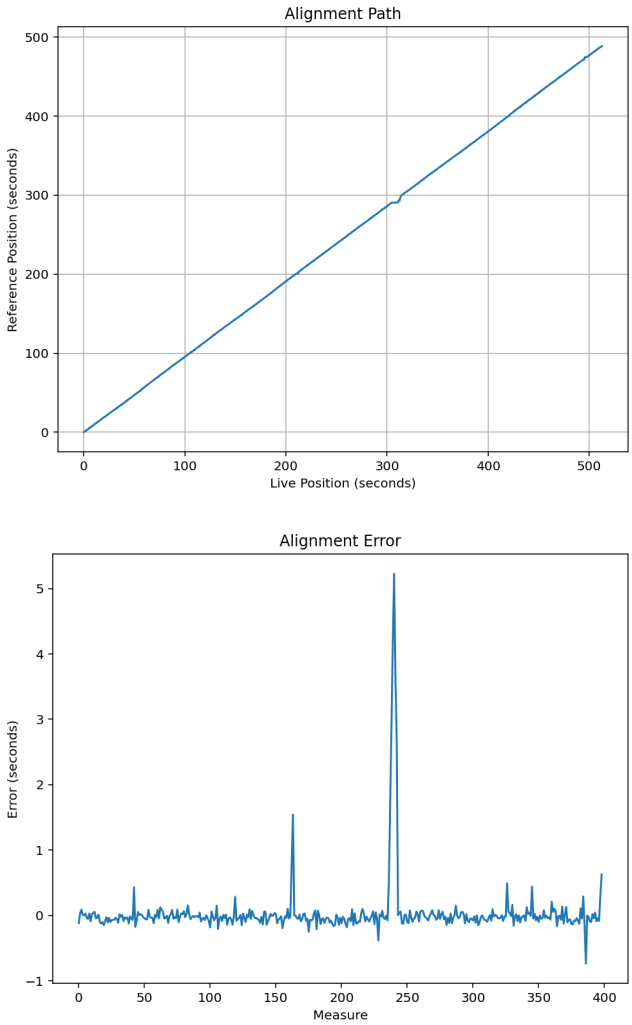

Figure 7: Alignment path and error for Mozart Symphony No. 40, mvt. 1; Reference: Bernstein, Live: Klemperer.

7. REFERENCES

[1] Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference (ICMC)*, pages 33–40, 2008.

[2] A. Arzt. *Score following with dynamic time warping: An automatic page-turner*. PhD thesis, Johannes Kepler University Linz, 2008.

[3] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *KDD Workshop*, volume 10, pages 359–370. Seattle, WA, USA:, 1994.

[4] C. H. Chan. *Simple Score Follower: A Contextual Switching Approach to Polyphonic Score Following on the Web using Deep-Learning Pitch Detection*. PhD thesis, Harvard University, 2022.

[5] A. Cont. A coupled duration-focused architecture for real-time music-to-score alignment. *IEEE transactions on pattern analysis and machine intelligence*, 32(6):974–987, 2009.

[6] R. B. Dannenberg. An on-line algorithm for real-time

accompaniment. In *ICMC*, volume 84, pages 193–198, 1984.

[7] S. Dixon. Live tracking of musical performances using on-line time warping. In *Proceedings of the 8th International Conference on Digital Audio Effects*, volume 92, page 97. Citeseer, 2005.

[8] Y. Han, S. Kwon, K. Lee, and K. Lee. A musical performance evaluation system for beginner musician based on real-time score following. In *NIME*, pages 120–121, 2013.

[9] F. Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 23(1):67–72, 1975.

[10] S. Letz, S. Denoux, D. Fober, et al. Faust audio DSP language in the web. In *Linux Audio Conference*, pages 29–36, 2015.

[11] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th Python in Science Conference*, volume 8, 2015.

[12] M. Müller. *Fundamentals of Music Processing: Audio, Analysis, Algorithms, Applications*. Springer Publishing Company, Inc., second edition, 2021.

[13] N. Orio, S. Lemouton, and D. Schwarz. Score following: State of the art and new developments. *New Interfaces for Musical Expression (NIME)*, 2003.

[14] C. Raphael. Aligning music audio with symbolic scores using a hybrid graphical model. *Machine learning*, 65(2-3):389–409, 2006.

[15] The Pyodide development team. pyodide/pyodide: 0.22.1, Jan. 2023.