A Javascript Musical Machine Listening Library

Nick Collins
Durham University
nick.collins@durham.ac.uk

Shelly Knotts
Durham University
knotts.shelly@gmail.com

ABSTRACT

More advanced interactive web browser based computer music applications are supported through a new javascript library for musical machine listening, MMLL. The library includes such facilities as beat tracking, pitch tracking, onset detection, major/minor chord detection, IFFT resynthesis and a tracking phase vocoder implementation. The code's efficiency, technical issues, and two example applications built upon the library are discussed.

INTRODUCTION

Web Audio API is a maturing technology for web browser based audio digital signal processing through javascript coding. Due to the efficiency of javascript engines in recent web browsers, it is an attractive option for new computer music work, not least given the promise of inherently cross platform capability, with ease of release to musician end-users (just direct them to a URL!). There remain some question marks over performance in the case of simultaneous use of an intensive audio callback and visual rendering or other heavy GUI.[1] Nonetheless, performance in non-graphics heavy settings is certainly reliable enough for computer music applications, and the tight potential coupling of GUI and low level audio DSP a powerful option.

A few libraries for feature extraction have appeared, including jsXtract, a native javascript port of libXtract [1], and Meyda [2]. These feature extractors provide standard audio descriptors such as MFCCs or the spectral centroid. However, they do not currently include the more mid to high level music analysis associated with such processes as chord detection or beat tracking. A few isolated examples of pitch detection algorithms have appeared online,[2] visualisations based on feature extraction have been explored [3],[3] and Web Audio API has a built in Analyzer node for FFT analysis. Vamp plugins have been ported to javascript via transpilation [4]. These implementations are complementary to the current work, but there is much audio analysis code available that hasn't yet been the subject of porting. We also work here from the ground up aiming for efficient real-time performance for live music use cases.

The present paper proceeds by introducing a new javascript library for musical machine listening with a basic tutorial in its use, considers efficiency and technical concerns, and discusses an example beat tracker driven audio cutting application amongst other application and end-user initiatives.

THE CODE LIBRARY

MMLL is a javascript library intended for use in Web Audio API contexts.[4] It provides a variety of higher level musical listening facilities for computer music, such as onset detection, pitch tracking, (major-minor) chord detection, beat tracking and auditory modeling. All listening objects can run live, or can be called in a simulated block by block way offline (potentially faster than realtime). The library was developed by the first author as part of the AHRC funded Musically Intelligent Machines Interacting Creatively (MIMIC) project, and is released under an MIT license.

MMLL can be deployed just for the machine listening objects within a user's own audio callback (e.g., as part of a ScriptProcessorNode), or via a quick setup frontend that hides Web Audio API details and has a user write setup and audio callback functions analogous to Processing's setup and draw [5].

The latter method is the one explained here, but those more expert in Web Audio API should find it easy enough to just take the analyzers for their own work. Simply linking to the precompiled MMLL.js script file is enough to deploy the library, though from the home directory of the library you can compile it afresh via the shell script provided (The library is just a concatenation of the js source files, where emscripten transpilation of some further C source code has already been conducted).

The typical expectation of a machine listening object is that we are working at 44.1KHz sampling rate and that a mono (single channel) input block of samples will be provided for analysis. The audio callback convenience function supplied by the library assumes stereo audio data, but provides both left and right input buffers, in case

[1] The audio worklets system, which supports separate threads for audio from other processes, is not yet sufficiently proven at the time of writing to constitute mainstream Web Audio API practice, but does promise enhanced safety of audio code execution, at the expense of additional overhead in passing data between different areas of the program.

[2] See for instance: https://webaudiodemos.appspot.com/pitchdetect/index.html

[3] Further, see: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Visualizations_with_Web_Audio_API

[4] https://github.com/sicklincoln/MMLL

audio processing is required, and a mixed mono input buffer (left and right channel average). Output is to separate left and right buffers. The machine listening objects deal themselves with accumulating samples ready for processing (often via an FFT) and the user normally doesn't have to worry about that part. However, objects should cope at other standard sampling rates such as 48KHz, 88.2KHz and 96 KHz, even if performance is sub-optimal; for example, the onset detector was developed based on evaluation over a corpus of 44.1KHz samples, so works best at this home rate.

1.1 A minimal code example

A minimal code example is reproduced below. Note how the machine listener object is prefixed with MMLL, and the SetUp function is passed the sampling rate, needed for initializing the listener. The CallBack is where the main action happens, as each new block of input samples is passed in. The input and output arguments hold MMLLInput and MMLLOutput objects, which make the channels of input and output audio accessible, as well as a special input.monoinput which is a single channel ready for the listener. If a stereo sound file is loaded or two channel live input requested, the monoinput will be the average of the left and right channels. The output object assumes a stereo output for now, exposing the left and right channel data arrays. The final GUISetup takes care of the detail of Web Audio API setup, including calling the Setup function at an appropriate time once the sample rate is confirmed, and establishing the callback. Buttons at the bottom of the webpage provide the option to work with live microphone input, or by loading a sound file; once audio input is underway, the buttons change to a single stop button which finishes a session (the two start options for microphone or audio file are then restored).

```
var audioblocksize = 256;
//lowest latency possible in Web Audio API

var setup = function SetUp(sampleRate) {
sensorydissonance = new
MMLLSensoryDissonance(sampleRate);
};

var callback =
function CallBack(input,output,n) {
var dissonance =
sensorydissonance.next(input.monoinput);
console.log(dissonance);
for (i = 0; i < n; ++i) {
output.outputL[i] = input.inputL[i];
output.outputR[i] = input.inputR[i];
}
};
var gui = new
MMLLBasicGUISetup( callback,setup,audioblo
cksize,true,true );
```

1.2 The main machine listening facilities and their CPU cost

Table 1 lists some of the main machine listening objects available in MMLL at the time of writing, with some indicative CPU costs, benchmarked on a five year old 2013 MacBook Pro (2.3GH i7 running Chrome 67.0.3396.87). Measurement in the final column gives CPU hit on one core; since processing is spread between the coreaudiod daemon and Chrome itself (labelled Google Chrome Helper in ActivityMonitor) two numbers are given. It is clear that the CPU cost is not prohibitive of running multiple machine listening processes with further audio synthesis on an older laptop, thus demonstrating the feasibility of established computer music algorithms for web browsers.

Performance in Firefox is comparable. The library has shortcut functions to work with either audio file input, or live microphone. The latter is a little more expensive in CPU load, due to denormal safety checks.[5]

Most objects have their origin in the machine listening facilities available in SuperCollider [6]. Manual ports from C code to javascript, or transpilation from C to javascript have both been explored. In fact, the performance of MMLL, whilst not as strong as SuperCollider's native C compiled scsynth, is reasonable, working at around double the CPU cost, and in some cases for longer block sizes, near equivalent.

1.3 Emscripten ports

Much legacy machine listening code exists in C, and it is possible to convert C code to Javascript via transpilation, for instance, with emscripten.[6] The BeatTrack UGen is an emscripten port of a SuperCollider UGen written with C (itself converted from research MATLAB code); the algorithm is due to Matthew Davies [7].

The drawback of transpilation is that the transpiler introduces an overhead in terms of code complexity in javascript, and requirements for careful calls to the transpiled functions and associated memory access for passing data.

The FFT library chosen was KissFFT,[7] offering a permissive license compatible with the MIT licensing of MMLL, alongside competitive performance (the code

[5] This is often an issue on Mac for unregulated audio input; without the checks, audio can abruptly cut out for an out of range signal, or increase processing cost for very small floating point values

[6] http://kripken.github.io/emscripten-site/

[7] https://github.com/j-funk/kissfft-js

uses emscripten to port from a C original). Whilst the rival javascript emscripten port of FFTW has been shown to be superior in testing,[8] the GNU GPL license restricts usage, for only a small relative gain in performance.

Table 1 List of relative performance of some machine listening algorithms within MMLL

Algorithm	Explanation	CPU cost (one instance) % coreaudio/ chrome
Control case: random noise + sample	https://webaudioapi.com/samples/script-processor/ (Random noise added to sample)	6/2
Control case: Pitch detector	https://webaudiodemos.appspot.com/pitchdetect/index.html	5/15
Control case: tuner	https://developer.microsoft.com/en-us/microsoft-edge/testdrive/demos/webaudiotuner/	6/6
FFT	Basic short time Fourier transform	5/4
IFFT resynthesis	Overlap add resynthesis via IFFT after FFT and frequency band filter	6/4
Onset detector	Algorithm by the first author, MIREX 2005 [8]	5/6
Beat tracker	Longer time window decision, stable but slower reacting to change [7]	5/7
Fast reacting beat tracker	Less stable, fast reacting, based on a variation of Scheirer's algorithm [9] where the comb filters are leaky integrators	5/6
Chord detection/key detection	Discriminates major and minor chord forms, by proximity to template chroma profiles [10]. Will also attempt to discriminate key if given longer decay times.	5/7
Sensory dissonance	After Sethares [11]	5/8
Gammatone auditory filterbank	88 filters spaced according to the frequencies of the 88 piano keys (in standard 12TET)	5/24
Gammatone filter	Single filter at 1000Hz, 200Hz bandwidth	5/2
Haircell model	Basic compressive nonlinear haircell model based on accumulation of transmitter (integrate and fire).	5/3
Tracking phase vocoder	After [12], sinusoidal oscillator bank resynthesis allowing f0 change without affecting duration	10/10
Constant Q pitch detector	After Brown and Puckette [13]	12/13
YIN autocorrelation pitch detector	After [14]. Block by block caching of difference function calculations is used to improve efficiency (otherwise runs at around 10/75 CPU cost)	10/20

[8] https://github.com/j-funk/js-dsp-test/

EXAMPLE APPLICATIONS

BBCut is an example application created using MMLL which rests upon beat tracking, allowing the triggering of rhythmic stutters locked to the beat, as well as a comb filter delay effect.[9] It uses code converted from a C language original (originally available as an iPhone app), manually ported to javascript. The screenshot reveals the main interface; the 'Open Microphone' and 'Open Audio File' buttons are automatically added programmatically by the javascript library's helper shortcut functions. The sliders and buttons control the available cuts, via automatic or manual triggering, and an additional comb filter delay.

Figure 1 Excerpt screenshot of BBCut showing the main controls

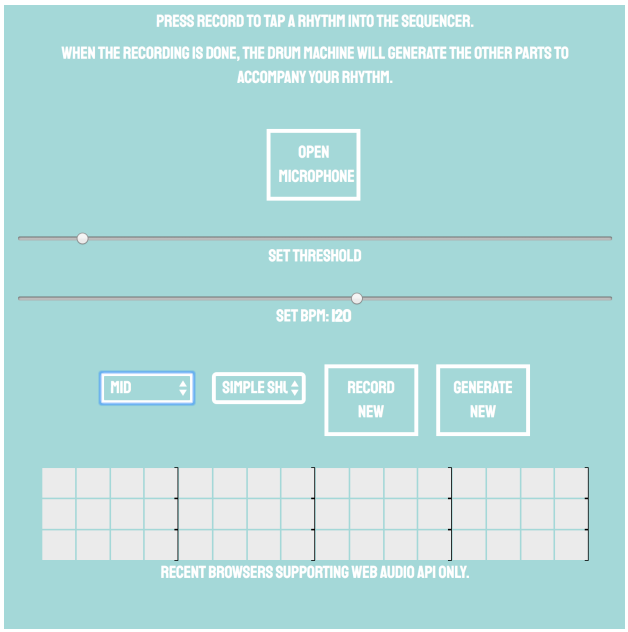

Figure 2 Screenshot of Rhythm Remixer

Rhythm Remixer is another example application which uses MMLL's onset detector. The interface shows a step sequencer, and provides controls allowing the user to set a tempo and threshold for detecting onsets. The user can choose a part to record, and a remixing algorithm. When recording is activated the user can tap a rhythm into the step sequencer using the computer microphone. Accompanying parts are generated using the selected remixing algorithm.

[9] https://dev.codecircle.gold.ac.uk/code/5ed346fe-f7d5-b7ce-87a4-df6e352dbb4a

USER REPORTS

Though the library has just been released on github, and this conference paper will form part of a strategy to more widely disseminate the software, early live performance experiments have taken place within a research project team.

Live premieres for many of the machine listeners are at the time of writing planned for an algorave [15] at the Sheffield AlgoMech festival in May 2019. The second author will play supported by the library where hypnotic noisy loops are transformed via musical machine listening data. The first author will deploy variations on the BBCut application in section 3 alongside gammatone filterbank vocoding, and further web audio API apps, across many browser tabs.

CONCLUSIONS

A machine listening library has been released for javascript that makes available some musical audio analysis processes ready for web browser computer music. There still remain many machine listening facilities in SuperCollider which can be ported from UGen C source code, as well as plenty of algorithms across the computer music and music information retrieval literature to implement directly in javascript or transpile via emscripten. Future planned additions to the library include the following SuperCollider UGens:

- PolyPitch: multiple f0 tracking UGen
- SMS: spectral modeling synthesis implementations
- Median Separation: percussive/tonal source separation algorithm

Further work would explore automatic drum detection, matching pursuit and concatenative synthesis, alongside integration with machine learning code.

Having completed the javascript porting of many established computer music algorithms, we are confident that web audio API provides a reasonably efficient, powerfully cross-platform and easily deployable project base for future computer music.

Acknowledgments

This work was funded under Arts and Humanities Research Council grant AH/R002657/1

REFERENCES

[1] Jillings, Nicholas, Jamie Bullock, and Ryan Stables (2016) "JS-Xtract: A realtime audio feature extraction library for the web." In the *International Society for Music Information Retrieval* Conference

[2] Rawlinson, H., N. Segal, and J. Fiala (2015) "Meyda: An audio feature extraction library for the Web Audio API." In *Proceedings of the 1st Web Audio Conference*.

[3] Roma, Gerard, Anna Xambó, Owen Green, and Pierre Alexandre Tremblay (2018) "A Javascript Library for Flexible Visualization of Audio Descriptors." *Proceedings of the 4th Web Audio Conference*, Berlin.

[4] Thompson, Lucas, Chris Cannam, and Mark Sandler (2017) "Piper: Audio Feature Extraction in Browser and Mobile Applications." *Proceedings of the 3rd Web Audio Conference*, London.

[5] Reas, Casey, and Ben Fry (2007) *Processing: a programming handbook for visual designers and artists*. Cambridge, MA: MIT Press, 2007.

[6] Wilson, Scott, David Cottle, and Nick Collins. (2011) *The SuperCollider Book*. Cambridge, MA: The MIT Press

[7] Davies, M. E. P. and Plumbley, M. D. (2005) "Beat Tracking With A Two State Model." *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing* (ICASSP 2005), Philadelphia, USA, March 19-23

[8] Collins, N. (2005) "A change discrimination onset detector with peak scoring peak picker and time domain correction." *Proc. 1st Annu. Music Inf. Retrieval Evaluation eXchange* (MIREX).

[9] Scheirer, Eric D. (1998) "Tempo and beat analysis of acoustic musical signals." *Journal of the Acoustical Society of America* 103(1): 588-601.

[10] Gomez, Emilia (2006) *Tonal Description of Music Audio Signals*. Doctoral dissertation, Department of Information and Communication Technologies, Universitat Pompeu Fabra

[11] Sethares, William A. (1998) "Consonance-Based Spectral Mappings." *Computer Music Journal* 22(1): 56-72

[12] McAulay, R. and Quatieri, T. (1986) Speech analysis/synthesis based on a sinusoidal representation. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 34(4): 744-754

[13] Brown, J. C. and Puckette, M. S. (1993) A High-Resolution Fundamental Frequency Determination Based on Phase Changes of the Fourier Transform. *Journal of the Acoustical Society of America* 94(2): 662-7

[14] De Cheveigné, A. and Kawahara, H. (2002) "YIN, a fundamental frequency estimator for speech and music." *The Journal of the Acoustical Society of America* 111(4): 1917-1930

[15] Collins, Nick, and Alex McLean. (2014) "Algorave: Live performance of algorithmic electronic dance music." In *Proceedings of New Interfaces for Musical Expression*, London