# BPMTimeline: Tempo Search and Time Mapping using an Analytical Solution

Bruno Dias
INESC-ID, IST - Universidade de Lisboa
bruno.s.dias@ist.utl.pt

David Martins Matos
INESC-ID, IST - Universidade de Lisboa
david.matos@inesc-id.pt

Helena Sofia Pinto
INESC-ID, IST - Universidade de Lisboa
sofia@ontol.inesc-id.pt

## ABSTRACT

The creation of tempo maps in native Digital Audio Workstations is a common feature, specially in commercial products. These maps allow the producer to automate the tempo of a musical performance or work, in a similar fashion to the automation of audio effect parameters. Unfortunately, the available tools for music composition and remixing in the browser, implemented with Javascript and Web Audio API, do not offer any mechanism for flexible and seamless tempo manipulation.

In this paper, we present BPMTimeline: a mapping between real time (seconds) and score time (beats), using tempo functions. Our implementation does not require the specification of tick period nor does it require tempo markers to be defined at specific time/beat values. To achieve this, we describe the mapping with the closed form of the integral, and its inverse, of each tempo function.

## 1. INTRODUCTION

Mainstream Digital Audio Workstations (DAW), both commercial and/or open-source, like Ableton Live[1], Logic Pro[2] and Reaper[3] allow tempo automation through a set of linear and/or step functions (see figure 1). Tempo manipulation of a musical expression is used to achieve several objectives: (1) to make the musical expression more lively (through a faster tempo) or more solemn (slower tempo); (2) to allow a DJ to evolve the overall tempo of his/her performance in order to mix tracks in a tempo similar to theirs; (3) to create climaxes in an expression. Unfortunately, there are to tempo mapping implementations in JavaScript. That means that live coding frameworks like Flocking [2] or Tone.js [11]. In Tone.js, the developer can deal manipulate tempo with tempo curves but their implementation requires the specification of tick and time signature to perform time mappings (i.e.: it does not provide a seamless mapping like our implementation).

In this paper, we present a tempo map implementation in JavaScript, as well as the theory supporting it. For this implementation, we had 3 important non-functional requirements:

- **No real impact on application performance:** the temporal overhead for tempo mappings and search;
- **Seamless tempo manipulation:** the developer should not be restricted to crisp tempo changes (e.g.: only step functions) nor should tempo markers be allowed on restricted beat/time marks (e.g.: beginning of a beat or measure);
- **Easy to integrate:** the implementation should be self-contained, with no need to include external libraries.

The outline of the paper is the following. In section 2, we state some essential theoretical definitions regarding the relation between score time and real time. In section 3, we detail the our JavaScript implementation. Finally, we describe some use cases for BPMTimeline.

The implementation is currently hosted at GitHub[4].

## 2. THEORETICAL BACKGROUND

We start by describing the basic relation between time, beats and tempo in order to infer the desired mappings. For additional background, please see [8, 3]. According to [8], music tempo $T$[5] is defined as the number of pulses/beats $b$ per $t$ time units,

$$T = \frac{b}{t} \tag{1}$$

the number of beats $b$, after $t$ time units, with tempo $T$ is given by

$$b = T * t \tag{2}$$

the duration $t$ of $b$ beats, with tempo $T$ is defined as

$$t = \frac{b}{T} \tag{3}$$

and the *beat period*, which is the inverse of tempo $T$, is defined as

[1]https://www.ableton.com/
[2]http://www.apple.com/logic-pro/
[3]http://www.reaper.fm/
[4]https://github.com/echo66/bpm-timeline.js
[5]$T$ can be interpreted as the frequency of the beat/pulse "train", in the score timeline [4].


*Web Audio Conference WAC-2016, April 4–6, 2016, Atlanta, USA*

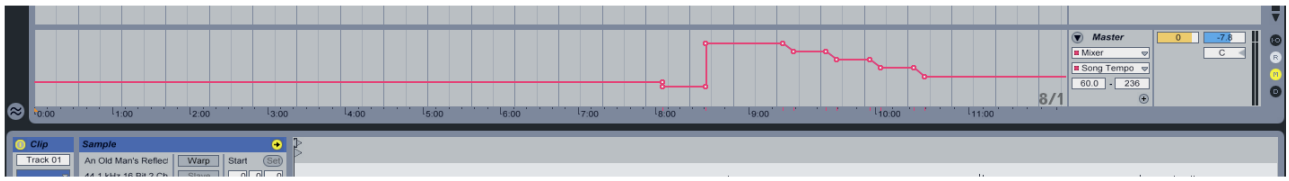


**Figure 1: Tempo automation example in Ableton Live, using step and linear tempo functions.**

$$beatPeriod = \frac{1}{T} = \frac{t}{b} \tag{4}$$

the equations 1, 2, 3 and 4 assumes that $T$, $t$, $b$ and $beatPeriod$ are constants.

To understand how do we map real time to score time using equations 1, 2, 3 and 4, we will use a set of examples. Consider the first one: how many minutes have passed when we reach beat 4, with a constant tempo of 120 bpm (beatPeriod = 0.5 secs)? According to equation 3, $t = \frac{4}{120} = 0.03\,min$, which is the same as the area under the curve in figure 2(a). If we decide to do a sudden change at beat 2, $t = \frac{2}{120} + \frac{2}{110} = 0.0348\,min$. But how do we use those equations to model a linear tempo change, beats 0 and 4, as depicted in 2(d)? One solution is to approximate the linear function through a sum of "steps" (figure 2(c)). If we use infinitesimally small steps, the mapping from score time to real time, $t(b)$, is defined as the integral of the beat period function. To map real time to score time, $b(t)$, we use the inverse of $t(b)$

$$t(b) = \int_0^b beatPeriod(\beta)\, d\beta \tag{5}$$

$$b(t) = t^{-1}(b) \tag{6}$$

$T$ is (usually[6]) expressed in Beats Per Minute (BPM). For example, $120\,bpm = \frac{120\,beats}{1\,min}$. But, usually, BPM is defined as

$$BPM = \frac{60\,(secs)}{beatPeriod} \tag{7}$$

with $beatPeriod$ measured in seconds. How do we deduce equation 5 from equation 1? If music tempo $T$ is measured as BPM, then $t = 1min$ and we can deduce from equation 1 that $BPM = \frac{b}{1} = b$. Due to the fact that $1\,min = 60\,secs$, we can make the following deduction

$$T = \frac{b}{t} \Leftrightarrow \frac{1}{T} = beatPeriod = \frac{t}{b}$$

$$t = 1\,min \Rightarrow T = \frac{b}{1} = b = BPM$$

$$t = 1\,min = 60\,secs \Rightarrow beatPeriod = \frac{60\,(secs)}{BPM} \Leftrightarrow$$

$$\Leftrightarrow BPM = \frac{60\,(secs)}{beatPeriod}$$

Through out the remainder of this paper, unless stated otherwise, $T$ and $t$ is measured as BPM and seconds. Additionally, we use equation 7 to relate BPM and beat period instead of using the generic tempo/time relations stated in equations 1 to 4.

## 2.1 Supported Tempo Functions

In the following three subsections, we will define the closed forms of the tempo functions, integrals and inverse of the integrals for the three included tempo functions: step, linear and exponential functions. All three are similar to the description in Web Audio API specification for AudioParam value changes[7]. All closed forms will share a set of terms:

- $b_0$, $b_1$: the initial and final score times;
- $T_0$, $T_1$: the initial and final tempos;
- $bP_0$, $bP_1$: the initial and final beat periods;
- $t$, $b$: the target time for tempo search and time mappings, $t$ for real time and $b$ for score time;
- $t_s$: the time offset, measure in seconds, for the function.

### 2.1.1 *Linear Tempo Functions*

$$k_1(y_0, y_1) = \frac{y_1 - y_0}{b_1 - b_0}$$

$$T(b) = T_0 + k_1(T_0, T_1) * (t - b_0) \tag{8}$$

$$beatPeriod(b) = bP_0 + k_1(bP_0, bP_1) * (t - b_0) \tag{9}$$

$$t(b) = \frac{k_1(bP_0, bP_1)}{2} * (t - b_0)^2 + bP_0 * (t - b_0) + t_s \tag{10}$$

$$k_2(t) = bP_0^2 - 4 * \frac{1}{2} * k_1 * (t_s - t)$$

$$sol_1(t) = \frac{-bP_0 + k_2(t)}{2 * k_1(bP_0, bP_1)}$$

$$sol_2(t) = \frac{-bP_0 - k_2(t)}{2 * k_1(bP_0, bP_1)}$$

$$b(t) = \begin{cases} sol_1(t) + b_0, & if\ sol_1(t) > 0 \\ sol_2(t) + b_0, & otherwise \end{cases} \tag{11}$$

[6] In many DAWs like Ableton Live, Logic Pro Tools, Reaper, Ardour and LMMS, music tempo is expressed in BPM. Still, it should be noted that music tempo can be expressed using, for example, italian tempo markings like *Largo*, *Adagietto*, *Andante moderato* and so on.

[7] http://webaudio.github.io/web-audio-api/#the-audioparam-interface

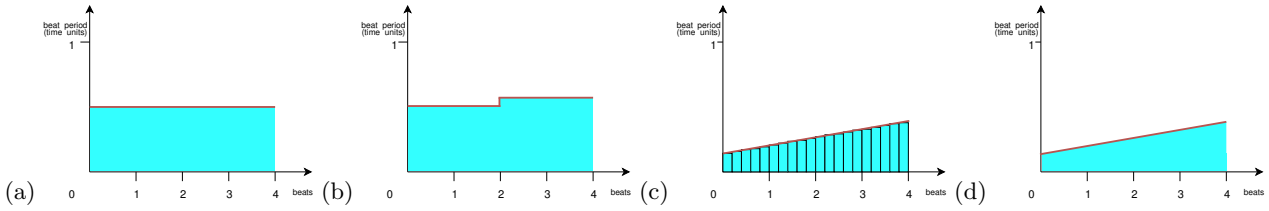


**Figure 2: Examples of tempo functions, between beats 0 and 4. In (a), we have a constant tempo of 120 bpm *(beatPeriod = 0.5 secs)*. In (b), there is a sudden (step tempo function) change, at beat 2, from 120 to 110 bpm *(beatPeriod = 0.(54) secs)*. In (c), there is a sequence of step tempo change that are used to approximate a linear tempo change as seen in (d). In these examples, we chose to use beat period instead of BPM, for the y-axis, to make the section 2 explanation, regarding b(t) and t(b), more intuitive.**

### 2.1.2 *Exponential Tempo Functions*

$$k_3(x_0, x_1) = \frac{\log \frac{x_1}{x_0}}{b_1 - b_0}$$

$$T(b) = T_0 * e^{k_3(T_0,T_1)*(b-b_0)} \tag{12}$$

$$beatPeriod(b) = bP_0 * e^{k_3(bP_0,bP_1)*(b-b_0)} \tag{13}$$

$$t(b) = T_0 * \frac{e^{k_3(bP_0,bP_1)*(b-b_0)} - 1}{k_3(bP_0, bP_1)} + t_s \tag{14}$$

$$b(t) = \frac{\log \frac{t_s*(t-t_s)+1}{bP_0}}{k_3(bP_0, bP_1)} + b_0 \tag{15}$$

### 2.1.3 *Step Tempo Functions*

$$T(b) = \begin{cases} T_0, & if\ b < b_1 \\ T_1, & otherwise \end{cases} \tag{16}$$

$$beatPeriod(b) = \begin{cases} bP_0, & if\ b < b_1 \\ bP_1, & otherwise \end{cases} \tag{17}$$

$$t(b) = \frac{b - t_s}{bP_0} + b_0 \tag{18}$$

$$b(t) = bP_0 * (t - b_0) + t_s \tag{19}$$

## 3. IMPLEMENTATION

In this section, we describe our JavaScript implementation of the tempo and time mappings defined in section 2. The current implementation has seven main features:

- find tempo $T$ at beat $b$ (e.g.: what is the BPM $T$, at beat $b$);
- find tempo $T$ at time $t$ (e.g.: what is the BPM $T$, at $t$ seconds);
- find what is the beat $b$ at time $t$ (i.e.: mapping time to beats);
- find what how much time $t$ has passed at beat $b$. (i.e.: mapping beats to time);
- add, edit and remove tempo markers;
- add new closed forms for tempo functions;
- observe changes in a BPMTimeline instance through event listeners.

The first four features are related to tempo search and time mapping. The following two are related to tempo markers management. The final one offers a way to notify interested javascript components of changes in a BPMTimeline.

A tempo marker is a JSON object that encodes the necessary information to evaluate local and global tempo functions:

- *endBeat*: Number defining beat $b_1$, stated by the developer when inserting the tempo marker in the BPMTimeline instance;
- *endTime*: Number defining time $t_1$, measured in seconds, calculated using $t(b_1)$ when inserting the tempo marker in the BPMTimeline instance;
- *endTempo*: Number defining the final tempo $T_1$, stated by the developer when inserting the tempo marker in the BPMTimeline instance;
- *endPeriod*: Number defining the duration of a beat at the "end" of the corresponding tempo marker/function, measured in seconds, and calculated when inserting the tempo marker in the BPMTimeline instance;
- *type*: String identifying which closed form (step, linear, exponential or custom) is used to define the tempo and mapping functions, stated by the developer when inserting the tempo marker in the BPMTimeline instance.

The global tempo function, $T_g(b)$, is a (potentially) non-continuous function, defined as

$$T_g(b) = \begin{cases} T_{l_1}(b), & if\ b_0 \le b \le b_1 \\ T_{l_2}(b), & if\ b_1 < b \le b_2 \\ \dots \\ T_{l_N}(b), & if\ b_{N-1} < b \end{cases}$$

where $T_{l_i}(b)$, $i = 1..N$, are (local) tempo functions. A global tempo function is represented as a sorted array of tempo markers, sorting it by *endBeat*. Each BPMTimeline instance has only one global tempo function. Each tempo marker represents a local tempo function. The values $T_0$,

$t_0$, $b_0$ and $t_s$ for each tempo is obtained by accessing the *endTempo* (for $T_0$), *endTime* (for $T_0$) and *endBeat* of the previous marker in the global tempo function array.

We should note that the formulation, in section 2, for the tempo function, $T(b)$, only mentions $b$. But, in practise, we could replace, in equations 8, 11, and 14 the terms $b_0$, $b_1$, $bP_0$, $bP_1$ and $b$ for $t_0$, $t_1$, $T_0$, $T_1$ and $t$ ($T(t)$) and the tempo would will be the same for both cases ($T(b) = T(t)$).

As stated in section 1, (temporal) performance is a very important requirement. To fulfil this requirement, some trade-offs are needed for the different features available. As such, we make the following assumptions:

- in DAWs and DJ software, tempo maps are not changed very frequently during a live performance;
- as such, tempo search and time mapping are more important than tempo markers management.

## 3.1 Insertion, Edition and Removal of Tempo Markers

There are three methods for tempo markers management:

- *add_tempo_marker(String type, Number endBeat, Number endTempo)*: performs a binary search in the tempo marker array to find the neighbouring tempo markers A and B, where $A.endBeat < B.endBeat \wedge A.endBeat < endBeat < B.endBeat$. After finding those neighbours, insert the new tempo marker between them and update the *endTime* field of all markers for which $\forall_M A.endBeat \geq M.endBeat$. If the new marker is the last one, only its *endTime* field will be updated. If there is already a tempo marker with the same value for *endBeat*, an exception will be thrown.
- *remove_tempo_marker(Number endBeat)*: performs a binary search in the tempo marker array to find the tempo marker A for which $M.endBeat = endBeat$. After that, removes M from the tempo markers array and update the *endTime* field for all tempo markers that $\forall_M A.endBeat \geq M.endBeat$. If there is no tempo marker A for which $M.endBeat = endBeat$, then an exception will be thrown.
- *change_tempo_marker(Number oldEndBeat, Number newEndBeat, Number newEndTempo, String newType)*: removes the tempo marker from the array and adds the new version. If there is no tempo marker A for which $M.endBeat = endBeat$, then an exception will be thrown.

These four methods for tempo marker management have the temporal complexity is $O(N)$. This is due to the usage of a binary search over a sorted array, $O(log_2 N)$, and the calculation for the *endTime* field, $O(N)$. As such, we have $O(log_2 N) + O(N) = O(N)$.

To add a new closed form for tempo functions, the developer can use the *add_tempo_marker_type(String type, Function tempoFn, Function integralFn, Function inverseIntegralFn)* method. Parameters 2, 3 and 4 are explained in section 2 and their API must be defined as:

- *tempoFn(Number $b_0$, Number $b_1$, Number $T_0$, Number $T_1$, Number b)*: implementation of the tempo function $T(b)$, as defined in section 2;
- *integralFn(Number $b_0$, Number $b_1$, Number $T_0$, Number $T_1$, Number $t_s$, Number t)*: implementation of the mapping $t(b)$, as defined in section 2;
- *inverseIntegralFn(Number $b_0$, Number $b_1$, Number $bP_0$, Number $bP_1$, Number $t_s$, Number b)*: implementation of the mapping $b(t)$, as defined in section 2.

This method has temporal complexity of $O(1)$ (on average) because it only performs an insertion in an associative array.

## 3.2 Tempo Search and Time Mapping Methods

The current implementation has 4 methods for this task:

- *beat(Number t)*: Maps real time $t$ to score time, $b(t)$. First, performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endTime < B.endTime \wedge$ $A.endTime < t < B.endTime$. If the search returns both A and B, perform the mapping $b(t)$ using *inverseIntegralFn( A.endBeat, B.endBeat, A.endPeriod, B.endPeriod, B.endTime, t )*. If the search returns just a marker, then then $b(t) = A.endBeat$.
- *time(Number b)*: Maps score time $b$ to real time, $t(b)$. First, performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endBeat < B.endBeat \wedge A.endBeat < b < B.endBeat$. If the search returns both A and B, perform the mapping $b(t)$ using *integralFn( A.endBeat, B.endBeat, A.endPeriod, B.endPeriod, B.endTime, b )*. If the search returns just a marker, then then $t(b) = A.endTime$.
- *tempo_at_time(Number t)*: Returns tempo at real time $t$. First, performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endTime < B.endTime \wedge$ $A.endTime < t < B.endTime$. If the search returns both A and B, obtain the tempo through *tempoFn( A.endTime, B.endTime, A.endTempo, B.endTempo, t )*. If the search returns just a marker, then the tempo is equal to A.endTempo.
- *tempo_at_beat(Number b)*: Returns tempo at score time $b$. First, performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endBeat < B.endBeat \wedge$ $A.endBeat < b < B.endBeat$. If the search returns both A and B, obtain the tempo through *tempoFn( A.endBeat, B.endBeat, A.endBeat, B.endBeat, b )*. If the search returns just a marker, then the tempo is equal to A.endTempo.

All four methods have temporal complexity of $O(log_2 N)$ due to the usage of binary search of a sorted array. If no marker is found in each of these four methods, an exception is thrown.

## 3.3 Event Listeners

Each BPMTimeline instance is observable: all changes (i.e. everytime a marker is added, edited or removed) in the instance results in an event being created and a set of event listeners will be invoked to deal with that event. In order to register/remove event listeners in a BPMTimeline instance, the class provides two functions:

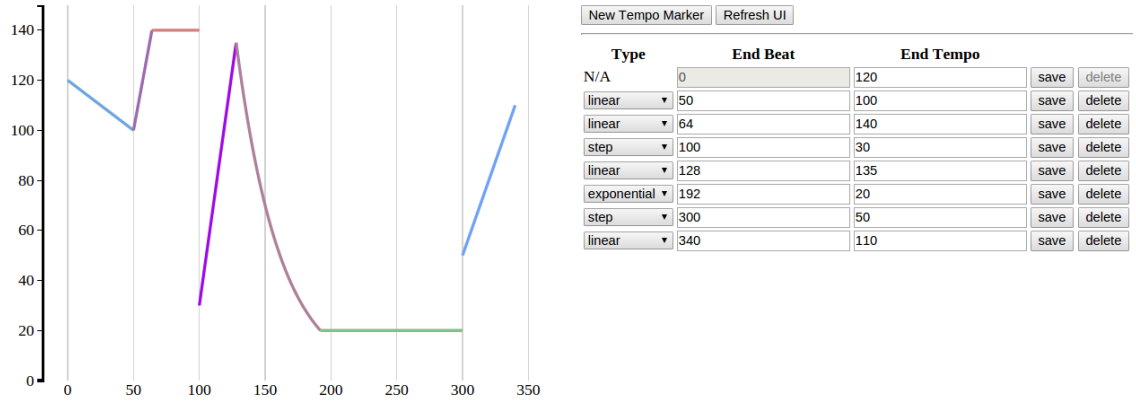


**Figure 3: A simple demo page with a graph plotting the output of BPMTimeline.tempo_at_beat, and a table listing the markers of the BPMTimeline instance.**

- *add_event_listener(String observerId, String eventType, Function callback)*: register an event listener for events of the following types: *"add-tempo-marker"*, *"change-tempo-marker"* and *"remove-tempo-marker"*.
- *remove_event_listener(String observerId, String eventType)*: remove the listeners, that were registered by observerId, for events with eventType type.

The *observerId* argument is used to prevent conflicts between two observers using the same function as event listener. Each event object has the following schema:

```
{
 type: String,
 oldMarker: MarkerDescription,
 newMarker: MarkerDescription,
}
```

The *type* field can have one of the following values: *"add-tempo-marker"*, *"remove-tempo-marker"* or *"edit-tempo-marker"*. When type is equal to *"add-tempo-marker"*, the *oldMarker* field does not exist. When *field* is equal to *"remove-tempo-marker"*, the *newMarker* field does not exit. Each *MarkerDescription* instance has the following schema:

```
{
 startBeat: Number, endBeat: Number,
 startTime: Number, endTime: Number,
 startTempo: Number, endTempo: Number,
 type: String
}
```

*startBeat*, *endBeat*: Numbers stating where does the marker tempo functions start and end in score time.

*startTime*, *endTime*: Numbers stating where does the marker tempo functions start and end in real time.

# 4. USE CASES

To date, we have explored three use case scenarios for BPMTimeline: (1) *event scheduling*, mapping time values in AudioContext.currentTime in order to schedule buffer and oscillator plays; (2) *auto-syncing* of several audio players to a dynamic master tempo, in a similar fashion to Ableton Live; (3) *time rulers*, for real and score times, related through a BPMTimeline instance.

## 4.1 Event Scheduling and Effect Automation

BPMTimeline can be used to control, through the mapping $t(b)$, the scheduling of Web Audio API (WAA) Audio Nodes (like Audio Buffer Source Nodes and Oscillator Nodes) and their Audio Parameters. Assuming that for $beat = 0 \Rightarrow AudioContext.currentTime = 0$, the developer can play a buffer source node and/or an oscillator node using the following code:

```
/* Initial tempo of 60 bpm. */
var tl = new BPMTimeline(60);
var a_t_m = tl.add_tempo_marker;

a_t_m({type:"linear",endBeat:10,endTempo:200});
a_t_m({type:"linear",endBeat:15,endTempo:10});
a_t_m({type:"linear",endBeat:20,endTempo:400});
a_t_m({type:"linear",endBeat:60,endTempo:60});

var ctx = new AudioContext();

var osc = ctx.createOscillator();

// Play 20 beats.
for (var i=0; i<20; i++) {
 scheduleNote(osc, 'G3', i, 0.5, ctx.currentTime);
}

osc.connect(ctx.destination);

osc.start();
```

resulting in a half-beat pulse train, that increases and/or decreases its tempo throughout the time line. This code is available in our Git repository[8]. Additionally, this module could be used to control the tempo in libraries like Tuna.js[9]

[8]https://github.com/echo66/bpm-timeline.js/blob/master/demos/demo3.html
[9]https://github.com/Theodeus/tuna

which, as far as we know, does not have any implementation for constant or dynamic tempo.

## 4.2 Automatic Audio Player Synchronization

Another use case is the synchronization of audio audio players to a master tempo time line, which is very common in live music performance applications like Ableton Live and Traktor Pro[10]. In one of our prototypes, we share a BPMTimeline instance (the master tempo) with several audio players. The stretching factor for each player is determined through the relation between the master tempo and the tempo of the audio segment being played/stretched[11].

## 4.3 Time Rulers

WAVES UI library [10] provides a set of music related UI components (automation lines/breakpoints, waveforms, time annotations, time rulers, etc...) for HTML5. One of those components, time axis[12], allows developers to present two time rulers: one for real time (seconds) and another for score time (beats), both related through a constant tempo (BPM). In order to use BPMTimline with WAVES UI library, the developer must decide what will be "static" time line[13], either real time or score time, in the rulers. Traditionally, DAWs make score time as the "static" time line. In order to do the same with WAVES UI, the developer will need to state the time values for segments, breakpoints, waveforms and traces in score time.

# 5. CONCLUSIONS AND FUTURE WORK

BPMTimeline provides an intuitive API for developers to relate time and beats, according to a custom tempo function. Due to the search method employed, we expect to minimize the (temporal) performance impact of time mapping and tempo search functions.

The next step will be the support for arbitrarily complex functions. Instead of specifying the closed form for all tempo functions ( there are infinite tempo functions), one could sample an tempo function and interpolate the resulting sequence with a set of linear tempo functions.

After that, we plan to integrate BPMTimeline in browser live coding environments like Flocking [2], Timbre.js [5], Lissajous [9], Neume.js [7], Ciseaux [6], Tone.js [11] and UI libraries/frameworks like Nexus UI [1] and WAVES UI [10].

Finally, we expect to port our code to C++ to include BPMTimeline in projects like QTractor [14], Ardour and LMMS.

# 6. ACKNOWLEDGMENTS

# 7. REFERENCES


[1] Ben Taylor and Jesse Allison. BRAID: A Web Audio Instrument Builder with Embedded Code Blocks. In *Proceedings of the 1st Web Audio Conference*, Paris, 2015.

[2] Colin Clark and Adam Tindale. Flocking: A Framework for Declarative Music-Making on the Web. In *Proceedings of the 40st International Computer Music Conference*, Athens, 2014.

[3] Henkjan Honing. From Time to Time: The Representation of Timing and Tempo. *Computer Music Journal*, 25(3):50–61, June 2001.

[4] J. MacCallum and A. Schmeder. Timewarp: A Graphical Tool for the Control of Polyphonic Smoothly Varying Tempos. In *Proceedings of International Computer Music Conference*, New York, 2010.

[5] Mohayonao. Timbre.js: Javascript library for objective sound programming (http://mohayonao.github.io/timbre.js/), 2014.

[6] Mohayonao. Ciseaux: Javascript utility to chop an audio buffer (http://mohayonao.github.io/ciseaux/), 2015.

[7] Mohayonao. Neume.js: Web audio api library for developing browser music, 2015.

[8] J. C. Schacher and M. Neukom. Where's the beat? Tools for Dynamic Tempo Calculations. In *Proceedings of International Computer Music Conference*. Zurich University of Arts, 2007.

[9] K. Stetz. Lissajous: Performing music with javascript, presentation at the 1st web audio conference, 2015.

[10] Victor Saiz and Benjamin Matuszewski and Samuel Goldszmidt. Audio Oriented UI Components for the Web Platform. In *Proceedings of the 1st Web Audio Conference*, Paris, 2015.

[11] Yotam Mann. Interactive Music with Tone.js. In *Proceedings of the 1st Web Audio Conference*, Paris, 2015.


[10] Traktor Pro has a master tempo clock to control effects and the time stretching factor of audio players. Still, that clock does not support tempo curves.

[11] https://github.com/echo66/OLA-TS.js/blob/master/SegmentProcessorV3.js#L121

[12] https://github.com/wavesjs/ui/tree/develop/es6/axis

[13] If we choose beat time to become "static", the tempo changes affect the real time ruler.

[14] As far as we know, QTractor provides a less sofisticate method for tempo map (http://bit.ly/1KVsQps).