BPMTimeline: JavaScript Tempo Functions and Time Mappings using an Analytical Solution

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

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

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

ABSTRACT

Time mapping is a common feature in many (commercial and/or open-source) Digital Audio Workstations, allowing the musician to automate tempo changes of a musical performance or work, as well as to visualize the relation between score time (beats) and real/performance time (seconds). Unfortunately, available music production, performance and remixing tools implemented with web technologies like JavaScript and Web Audio API do not offer any mechanism for flexible, and seamless, tempo manipulation and automation.

In this paper, we present BPMTimeline, a time mapping library, providing a seamless mapping between score and performance time. To achieve this, we model tempo changes as tempo functions (a well documented subject in literature) and realize the mappings through integral and inverse of integral of tempo functions.

Keywords

JavaScript, Time Mapping, Tempo Function, Automation

1. INTRODUCTION

Tempo manipulation of a musical expression or works is used to (1) make a musical expression, or work, more lively (through a faster tempo) or more solemn (slower tempo); (2) allow a DJ to synchronize the tempo of several songs playing simultaneously, to align the beats; (3) create climaxes in a musical expression or work. Mainstream Digital Audio Workstations (DAWs), either commercial and/or open-source, like Ableton Live, figure 1, Logic Pro and Reaper allow tempo automation through time maps, offering a seamless relation between performance and score time, using a tempo function. Unfortunately, to the best of our knowledge, there are no time mapping implementations in JavaScript. In this paper, we present the theory supporting time maps as well as our JavaScript implementation, BPMTimeline. The development of BPMTimeline followed three non-functional requirements:

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

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

In section 3, we describe essential theoretical definitions regarding the relation between score and real time. In section 4, we detail our implementation. Finally, in section 5, we describe some use cases for BPMTimeline. Our implementation and demo pages are currently hosted at GitHub.[1]

2. RELATED WORK

Time mapping is a well document subject [2, 5, 4, 9, 3]. The main idea of a time map is to relate score time (beats) with real/performance time (seconds) through mathematical manipulation of tempo functions. A tempo function T maps score, or performance, time to a tempo value, e.g.: $T(63\,secs) = 105.12\,bpm$). Besides tempo functions and time maps, another way of representing expressive tempo manipulation and timing is through time shifts [4], a function that expresses deviations of certain events in relation to a reference score/performance time line. Available JavaScript live coding frameworks and DAWs do not offer time maps. In frameworks like Flocking [1], tempo manipulation is achieved through manual and crisp changes, and tempo automation can be achieved through the use of the *setInterval* or *setTimeout* methods. In Tone.js [7], tempo manipulation and automation is achieved through both (1)

[1] https://github.com/echo66/bpm-timeline.js

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

manual and crisp changes and (2) through a Tone.Signal object, which offers a similar API to AudioParam.[2] Still, Tone.js offers no time mapping facilities.

3. 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 [9, 4]. According to [9], music tempo T is defined as the number of beats b per t time units [6],

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

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

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

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

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

The bP, which is the inverse of tempo T, is defined as

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

Equations 1, 2, 3 and 4 assume that T, t, b and bP are constants. To understand how to map real time to score time, let us analyse the following example: how many minutes have passed when we reach beat 4, with a constant tempo of 120 bpm (bP = 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 the equations to model a linear tempo change between 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 $t(b)$

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

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

In [9], the authors used $T(t)$ instead of $T(b)$, resulting in a different integral and inverse of the integral for the tempo function. Despite the differences, the time map will produce the same results. According to the previous definitions, a time map can be defined by $T(b)$, $bP(b)$, $t(b)$ and $b(t)$. For the current implementation, we provide three default for time mapping and tempo functions: linear, exponential and step tempo functions, all of them inspired in AudioParam automation functions.

3.1 Supported Functions

In the following three subsections, we present the three forms for time maps included in our implementation: step, linear and exponential forms. All three share a set of terms related to tempo and time:

- b_0, b_1: the initial and final score times;

[2]http://webaudio.github.io/web-audio-api/#AudioParam

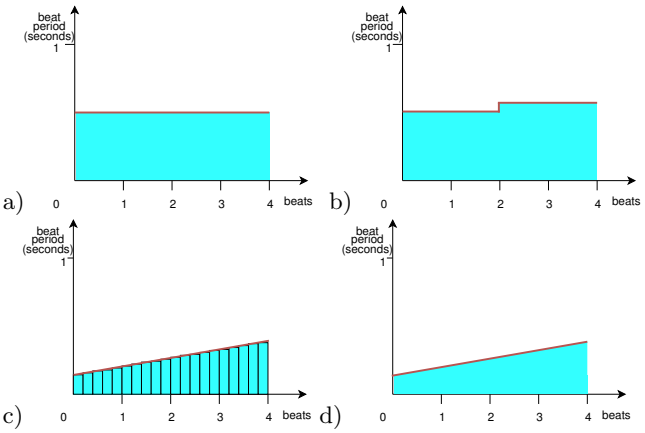

Figure 2: Examples of tempo functions, between beats 0 and 4. In (a), we have a constant tempo of 120 bpm *(bP = 0.5 secs)*. In (b), there is a sudden (step tempo function) change, at beat 2, from 120 to 110 bpm *(bP = 0.(54) secs)*. In (c), there is a sequence of step tempo change that are used to approximate 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 3 explanation, regarding b(t) and t(b), more intuitive.

- 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, measured in seconds, for the function.

For each subsection, we define the generic functions and how are they mapped to a closed form.

3.1.1 Linear

Based on the definition of the formula used in AudioParam *linearRampToValueAtTime* method

$$lin_{X_0,X_1,Y_0,Y_1}(x) = Y_0 + \Delta \cdot (x - X_0) \tag{7}$$

with $\Delta = \frac{Y_1 - Y_0}{X_1 - X_0}$ defining the slope of lin. This function is not defined for $X_0 = X_1$. Using this function, the tempo and beat period functions can be defined as

$$T(b) = lin_{b_0,b_1,T_0,T_1}(b),\ bP(b) = lin_{b_0,b_1,T_0,T_1}(b)$$

where both functions are not defined for $b_0 = b_1$. This "corner case" occurs when there is a sudden jump in tempo values. In that case, one should model the tempo change with a step tempo function. To obtain the mappings, we integrated the linear function and then obtained the inverse of the integral

$$L_{X_0,X_1,Y_0,Y_1,C}(x) = \int_0^x lin_{X_0,X_1,Y_0,Y_1}(x)\, d_x = \\ = \frac{\Delta}{2} \cdot (x - X_0)^2 + Y_0 \cdot (x - X_0) + C \tag{8}$$

$$L^{-1}_{X_0,X_1,Y_0,Y_1,C}(y) = \begin{cases} sol_1(y) + X_0, & if\ sol_1(y) > 0 \\ sol_2(y) + X_0, & otherwise \end{cases} \tag{9}$$

$$sol_1(y) = \frac{-Y_0 + k_1}{2 \cdot \Delta}\ , sol_2(y) = \frac{-Y_0 - k_1}{2 \cdot \Delta}$$

$$k_1 = Y_0^2 - \frac{4}{2} \cdot \Delta \cdot (C - y)$$

According to equations 5 and 6, the time mappings are defined as

$$t(b) = L_{b_0,b_1,bP_0,bP_1,t_s}(b),\ b(t) = L^{-1}_{b_0,b_1,bP_0,bP_1,t_s}(t)$$

3.1.2 Exponential

Consider the exponential function defined in [9] as

$$exp_{X_0,X_1,Y_0,Y_1}(x) = Y_0 \cdot e^{k_2 \cdot (x - X_1)},\ k_2 = \frac{\log \frac{Y_1}{Y_0}}{X_1 - X_0} \quad (10)$$

and the tempo and beat period functions defined as

$$T(b) = exp_{b_0,b_1,T_0,T_1}(b),\ bP(b) = exp_{b_0,b_1,T_0,T_1}(b)$$

where all functions are not defined for $b_0 = b_1$ (the same "corner case" as in the linear case). The time mappings for the exponential function can be obtained in a similar fashion:

$$\begin{aligned} E_{X_0,X_1,Y_0,Y_1,C}(x) &= \int_0^x exp_{X_0,X_1,Y_0,Y_1}(x)\, d_x = \\ &= Y_0 \cdot \frac{e^{k_3 \cdot (x - X_0)} - 1}{k_3} + C \end{aligned} \quad (11)$$

and the inverse of the integral

$$E^{-1}_{X_0,X_1,Y_0,Y_1,C}(y) = \frac{\log \frac{C \cdot (y - C) + 1}{Y_0}}{k_3} + X_0 \quad (12)$$

and the final map between the generic functions and the time mapping functions is

$$t(b) = E_{b_0,b_1,bP_0,bP_1,t_s}(b),\ b(t) = E^{-1}_{b_0,b_1,bP_0,bP_1,t_s}(t)$$

3.1.3 Step

The step function uses a similar tempo function to the AudioParam *setValueAtTime*,

$$step_{X_0,X_1,Y_0,Y_1}(x) = \begin{cases} Y_0, & if\ x < X_1 \\ Y_1, & otherwise \end{cases} \quad (13)$$

with the difference that the value jumps to Y_1 only for $x = X_1$ instead of $x = X_0$, like *setValueAtTime* formula. The integral and inverse of the integral are defined as

$$S_{X_0,X_1,Y_0,Y_1,C}(x) = \begin{cases} Y_0 \cdot x + C, & if\ x < X_1 \\ Y_1 \cdot x + C, & otherwise \end{cases} \quad (14)$$

$$S^{-1}_{X_0,X_1,Y_0,Y_1,C}(y) = \begin{cases} \frac{y - C}{Y_0}, & if\ y < Y_1 \\ \frac{y - C}{Y_1}, & otherwise \end{cases} \quad (15)$$

Again, the inverse of the integral is not defined for $Y_0 = 0 \vee Y_1 = 0$. According to the previous formulas, we define the tempo and beat period functions as

$$T(b) = step_{b_0,b_1,T_0,T_1}(b),\ bP(b) = step_{b_0,b_1,T_0,T_1}(b)$$

and the time mappings as

$$t(b) = S_{b_0,b_1,bP_0,bP_1,t_s}(b),\ b(t) = S^{-1}_{b_0,b_1,bP_0,bP_1,t_s}(t)$$

with $b(t)$ not defined for $T_0 = 0 \vee T_1 = 0$. This "corner case" is only problematic when modelling, with the step closed form, a full stop in the score/performance time.

3.2 Beats Per Minute definition

T is (usually[3]) expressed in BPM. For example, $120\,bpm = \frac{120\,beats}{1\,min}$. But, usually, BPM is defined as

$$BPM = \frac{60\,(secs)}{bP} \quad (16)$$

with beat period 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:

$T = \frac{b}{t} \Leftrightarrow \frac{1}{T} = bP = \frac{t}{b}$
$For\ \ t = 1\,min \Rightarrow T = \frac{b}{1} = b = BPM$
$For\ \ t = 60\,secs \Rightarrow bP = \frac{60\,(secs)}{BPM} \Leftrightarrow BPM = \frac{60\,(secs)}{bP}$
Throughout the remainder of this paper, unless stated otherwise, T is measured as BPM and t as seconds. Additionally, we use equation 16 to relate BPM and beat period instead of using the generic tempo/time relations stated in equations 1 to 4.

4. IMPLEMENTATION

In this section, we describe our JavaScript implementation for time maps, as defined in section 3. 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, e.g.: mapping time to beats;
- find what how much time t has passed at beat b, e.g.: 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 and the last one offers a way to notify JavaScript components of changes in a BPMTimeline. A tempo marker is a JSON object that encodes the information needed to evaluate local and global tempo functions:

- *endBeat*: beat b_1, stated by the developer when inserting the tempo marker in the BPMTimeline instance;
- *endTime*: time t_1, measured in seconds, calculated using $t(b_1)$ when inserting the tempo marker in the BPMTimeline instance;
- *endTempo*: final tempo T_1, stated by the developer when inserting the tempo marker in the BPMTimeline instance;

[3]In many DAWs like Ableton Live, Logic Pro Tools, Reaper, Ardour and LMMS, music tempo is expressed in Beats Per Minute (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.

- *endPeriod*: 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 function (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 non-continuous function, defined as

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

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, sorted 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 are obtained by accessing the *endTempo* (for T_0), *endTime* (for T_0) and *endBeat* of the previous marker in the global tempo function array. As stated in section 1, performance is a very important requirement, therefore, some trade-offs for the different features are needed. We make the following assumptions:

- in DAWs and DJ software, time maps are not changed very frequently during a live performance;

- as such, tempo search and time mapping are more important than tempo markers management.

We should note that the formulation, in section 3, for the tempo function $T(b)$ only mentions b. But we could obtain $T(t)$ by replacing the terms b_0, b_1, bP_0, bP_1 and b for t_0, t_1, T_0, T_1 and t, and the tempo would be the same for both cases, $T(b) = T(t)$.

4.1 Insertion, Edition and Removal of Tempo Markers

There are three methods for tempo markers management:

- *add_tempo_marker(String type, Number b_1, Number T_1)*: performs a binary search in the tempo marker array to find the neighbour tempo markers A and B, where $A.endBeat < b_1 < B.endBeat$. After finding those neighbours, it inserts the new tempo marker between them and updates 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 b_1)*: performs a binary search in the tempo marker array to find the tempo marker A for which $M.endBeat = endBeat$. After that, M from the tempo markers array is removed and the *endTime* field for all tempo markers M that $\forall_M A.endBeat \geq M.endBeat$ are updated. If there is no tempo marker A for which $M.endBeat = endBeat$, then an exception will be thrown.

- *change_tempo_marker(Number b_1, Number b_1', Number T_1', String newType)*: removes the tempo marker from the array and adds the new version. If there is no tempo marker A for which $M.endBeat = b_1$, then an exception will be thrown.

These three methods for tempo marker management have the temporal complexity $O(N)$, 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 new tempo functions, the developer can use the *add_tempo_marker_type(String type, Function tempoFn, Function integralFn, Function inverseIntegralFn)* method:

- *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 3;

- *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 3;

- *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 3.

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

4.2 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, it performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endTime < t < B.endTime$. If the search returns both A and B, it performs 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 $b(t) = A.endBeat$.

- *time(Number b)*: Maps score time b to real time, $t(b)$. First, it performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endBeat < b < B.endBeat$. If the search returns both A and B, it performs 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 $t(b) = A.endTime$.

- *tempo_at_time(Number t)*: Returns tempo at real time t. First, it performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endTime < t < B.endTime$. If the search returns both A and B, it obtains 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, it performs a binary search over the tempo markers array to find tempo markers A and B for which $A.endBeat < b < B.endBeat$. If the search returns both A and B, it obtains 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.

4.3 Event Listeners

Each BPMTimeline instance is observable: every time a marker is added, edited or removed in the instance, an event is 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:

- *add_event_listener(String observerId, String eventType, Function callback)*: registers 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)*: removes 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:

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

When *eventType* is equal to *"add-tempo-marker"*, the *oldMarker* field does not exist. When *eventType* 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 and *endBeat* define where the marker tempo functions start and end in score time and *startTime*, *endTime* state where does the marker tempo functions start and end in real time.

5. 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 oscillator plays; (2) *automatic synchronization* 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.

5.1 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 atm = tl.add_tempo_marker;
atm({ type:"linear", endBeat:10, endTempo
    :200 });
atm({ type:"linear", endBeat:15, endTempo
    :10 });
atm({ type:"linear", endBeat:20, endTempo
    :400});
atm({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.[4] Additionally, this module could be used to control the tempo in libraries like Tuna.js[5] which, as far as we know, does not have any implementation for constant or dynamic tempo.

5.2 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. 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.[6]

5.3 Time Rulers

WAVES UI library [8] 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, 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 which will be "static" time line[7], 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 needs to state the time values for segments, breakpoints, waveforms and traces in score time.

6. COMPARISONS

In this section, we compare BPMTimeline with Tone.js, the most advanced JavaScript implementation for tempo manipulation.

- **Time Maps**: Tone.js does not offer any time mapping feature.

[4]https://github.com/echo66/bpm-timeline.js/blob/master/demos/demo3.html
[5]https://github.com/Theodeus/tuna
[6]https://github.com/echo66/SegmentSequencer.js
[7]If we choose beat time to become "static", the tempo changes affect the real time ruler.

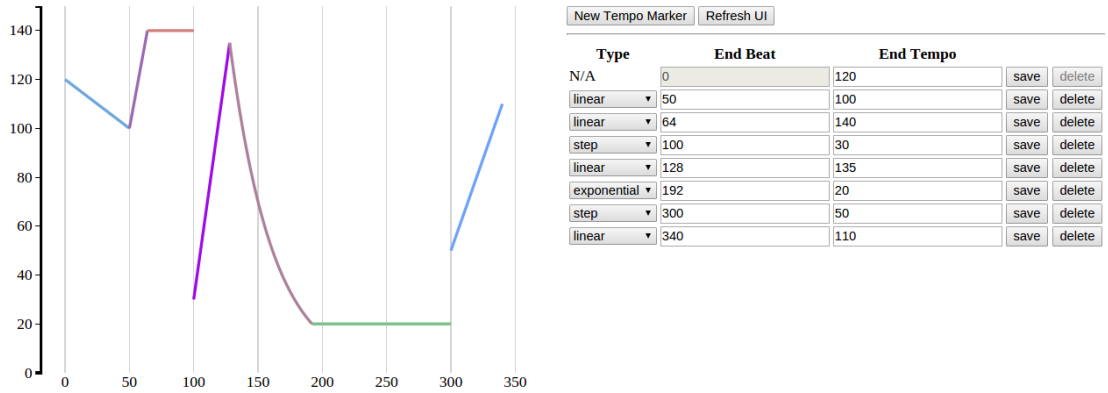

Figure 3: A simple demo page with a graph plotting the output of BPMTimeline.tempo_at_beat, and a table listing the markers of the BPMTimeline instance.

- **Tempo Functions**: Tone.js and BPMTimeline offer a similar set of basic tempo functions: step, linear and exponential. Still, with Tone.js, it is possible to schedule an arbitrarily complex tempo function using *setValueCurveAtTime* Tone.Signal method. Currently, BPMTimeline does not offer a similar feature to *setValueCurveAtTime*.
- **WAA**: Tone.js uses a Tone.Signal object for tempo automation, a class that uses a Gain Node to schedule changes for tempo, effect parameters, etc. BPMTimeline does not have any dependence on specific JavaScript environments besides a compliant to ECMAScript 5 implementation.
- **NPM**: Tone.js has a NPM module available. Still, the current implementation does not allow to use Tone.js within node.js or io.js due to its dependence on WAA.
- **Time markers**: BPMTimeline requires the user/developer to schedule tempo changes using score time values. In Tone.js, due to its dependence on WAA, tempo changes are scheduled with performance time.

7. CONCLUSIONS AND FUTURE WORK

BPMTimeline provides an API for developers to relate time and beats, according to a custom tempo function. Due to the used search method, 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 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 [1], Tone.js [7] and UI libraries/frameworks like WAVES UI [8].

8. ACKNOWLEDGMENTS

This work was partly supported by national funds through FCT – Fundação para a Ciência e Tecnologia, under projects EXCL/EEI-ESS/0257/2012 and UID/CEC/50021/2013 and by Luso-American Development Foundation.

9. REFERENCES

[1] C. Clark and A. Tindale. Flocking: A Framework for Declarative Music-Making on the Web. In *International Computer Music Conference*, Athens, 2014.

[2] R. B. Dannenberg. Abstract Time Warping of Compound Events and Signals. *Computer Music Journal*, 21(3):pp. 61–70, 1997.

[3] P. Desain and H. Honing. Tempo curves considered harmful. *Contemporary Music Review*, 7(2):123–138, 1993.

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

[5] A. Kirke and E. R. Miranda. A survey of computer systems for expressive music performance. *ACM Computer Surveys*, 42(1):3:1–3:41, Dec. 2009.

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

[7] Y. Mann. Interactive Music with Tone.js. In *1st Web Audio Conference*, Paris, 2015.

[8] V. Saiz, B. Matuszewski, and S. Goldszmidt. Audio Oriented UI Components for the Web Platform. In *1st Web Audio Conference*, Paris, 2015.

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