A Polytemporal Model for Musical Scheduling

Martin Fouilleul, Jean Bresson, Jean-Louis Giavitto

A Polytemporal Model for Musical Scheduling

Martin Fouilleul, Jean Bresson, and Jean-Louis Giavitto

STMS – Sorbonne Université, IRCAM, CNRS
martin.fouilleul@ircam.fr
jean.bresson@ircam.fr
jean-louis.giavitto@ircam.fr

Abstract. This paper describes the temporal model of a scheduler geared towards show control and live music applications. This model relies on multiple inter-related temporal axes, called timescales. Timescales allow scheduling computations using abstract dates and delays, much like a score uses symbolic positions and durations (e.g. bars, beats, and note values) to describe musical time. Abstract time is ultimately mapped onto wall-clock time through the use of time transformations, specified as tempo curves, for which we provide a formalism in terms of differential equations on symbolic position. In particular, our model allows specifying tempo both as a function of time or as a function of symbolic position, and allows piecewise tempo curves to be built from parametric curves.

Keywords: Symbolic time, Time transformations, Tempo curves, Scheduling

1 Introduction

Timing is of utmost importance in performing arts. Among them, music has developed particularly fine-grained temporal constructs, using both continuous and discrete abstract representations of time. As such, it presents specific and interesting challenges with regard to the composition and interpretation of time at multiple scales, and across multiple independent time-flows.

In this paper we present the temporal model of *Jiffy*, a polytemporal scheduler which is part of an ongoing effort to build a programmable show-controller system for performing arts and interactive multimedia installations[1]. In particular, our temporal model allows specifying tempo either as a function of time or as a function of symbolic position, and allows piecewise tempo curves to be built from parametric curves such as Béziers curves, which are both versatile and intuitive. The scheduler exposes an interface based on fibers, that makes it easy to organize inter-dependant streams of related events.

We first highlight the importance of symbolic time in musical applications (section 2). We then cover the notion of time transformations, and give a differential equation formulation to tempo curves (section 3). We then show how tempo curves equations are solved in Jiffy (section 4). Finally, we present the interface of the scheduler (section 5).

[1] The source code of the scheduler as of the time of writing can be found at https://github.com/martinfouilleul/jiffy_scheduler_standalone/tree/fff78cd5ca6ab895ba3107439e8ac9541811590a.

2 Symbolic Time in Musical Applications

2.1 Common Paradigms in Show-Control Applications

Show controllers are programs used by sound and lighting engineers to create and run temporal scenarios synchronized to the actions of performers on stage. They allow users to launch sound and video samples, control mixing and lighting desks, operate motors for mechatronic stage props, and so on. Several approaches can be identified as to how they present and organize the temporal relations between the cues of the show:

- Timelines, which organize cues on a common, static time axis. Most sequencers, such as ProTools[2] or Cubase[3] fall in that category.
- Cuelists, which organize cues in nested lists with associated timing semantics. Notable examples are QLab[4] or Linux Show Player[5].
- Hybrid models offer both cuelists and timelines, either through separate modes of operation, as in Medialon[6] or Smode[7], or as dual views of the same cues, as in Ableton Live[8].
- Graphical planning environments that allow users to position cues in some abstract space, which maps to time through the use of trajectories, as in Iannix [6], or flow graphs, as in Ossia Score [5].

Despite the diversity of approaches, most show controllers lack an abstract notion of musical time, as they directly map cues to wall-clock dates or to external triggers. Furthermore, musical time is often deployed throughout a work at different scales (e.g. movements, phrases, cells, notes...), and not every scale is tied to the same global tempo, e.g. ornaments such as grace notes and *appogiatura* are not affected in the same way by a change of tempo as a main melody line. Hence it would be more appropriate to consider several abstract musical times, or *timescales*.

2.2 Abstract Timescales

The above discussion emphasizes the need of strong temporal models in composition and performance softwares and highlights the adequacy of *polytemporal abstract time scheduling*, i.e. the ability to organize concurrent computations along multiple logical timescales, that can later get mapped to wall-clock time.

[2] https://www.avid.com/pro-tools
[3] https://new.steinberg.net/cubase/
[4] https://qlab.app/docs/QLab_4_Reference_Manual.pdf
[5] https://linux-show-player-users.readthedocs.io/en/latest/index.html
[6] https://medialon.com/wp-content/uploads/2019/07/M515-1-Medialon-Control-System-Manual.pdf
[7] https://smode.fr
[8] https://www.ableton.com/en/live/what-is-live/

The notion of abstract timescales has been tackled before by computer music environments or score followers. For instance, FORMULA [1] allows applying independent time deformations on groups of concurrent tasks. David A. Jaffe [9] proposed a recursive scheduler for hierarchical timing control, using explicit time maps. Antescofo [7] allows users to compose independent abstract times through the use of *time scopes* and tempo curves.

In Jiffy, a timescale is a data structure used to maintain a notion of logical time, expressed as a rational number of *symbolic time units*[9] (STU), and to schedule events at specific logical dates. It is analogous in this respect to a score, which organizes musical events in terms of a musical time, that needs to be translated into wall-clock time by a musician according to tempo indications and interpretative choices.

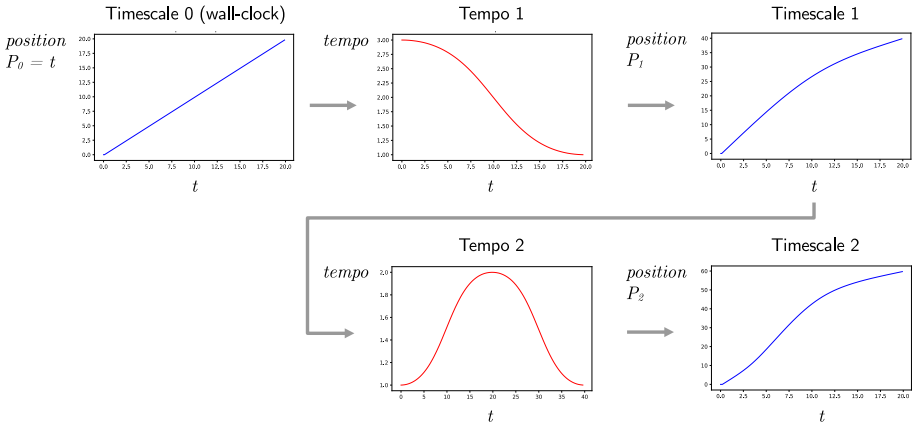

Fig. 1. Composing time deformations using tempo curves.

However, whereas the tempo indication of a score usually prescribes some idealized mapping from musical-time to wall-clock time, a timescale's logical time does not necessarily map directly to wall-clock time. Instead, each timescale has a *time source*, which can be either the wall-clock time or another timescale. A timescale is also associated with a *time transformation*, which maps its internal time to the time of its source. Thus, the scheduler can handle multiple notions of logical time and map dates to wall-clock time through a hierarchy of time transformations.

Figure 1 illustrates a time deformation between a timescale and its source. The time map plots for each timescale show the position of the timescale with respect to wall-clock time. The effect of the first tempo curve (tempo 1) is to warp the time map of

[9] We deliberately avoid the term *beats* here. We think it would bring some confusion by conflating the notion of time unit with the notion of meter, and by suggesting that all beats are of equal conceptual length. This is, in fact, rather a Western exception than a universal norm.

timescale 0 (which represents wall-clock time) into that of timescale 1 (which represent some abstract musical time). Timescale 1 is then transformed by another another tempo curve (tempo 2), to produces the time map of timescale 2.

3 Time Transformations

Jiffy's scheduler must be able to transform timescale-local positions to and from wall-clock time. These transformations are specified by the means of tempo curves, which describe the speed of a timescale's "playhead" with respect to the source time, much like tempo indications in a score prescribe an idealized conversion from durations in beats to durations in wall-clock time[10].

Several methods have been proposed to represent time transformations and to integrate tempo curves to map symbolic position to time. Jaffe [9] proposes to directly use time maps constructed from a collection of predefined time warping functions. Berndt [2] chooses to represent tempo curves by potential functions of symbolic position, matching some specified mean tempo condition. Timewarp [10] is a tool that uses regularized beta functions to define tempo curves satisfying polyrhythmic constraints. Antescofo uses a variety of tweening functions[11] to express tempo as a function of time, and uses closed form expressions to compute time transformation based on tempo curves. When there is no analytical solution to a tempo curve integration, Antescofo samples the curve to produce a piecewise linear approximation, which is then integrated analytically. Antescofo can also use arbitrary expressions to define tempo, although these expressions are not integrated: they are reevaluated each time a variable is updated, and considered constant between updates. As such, they can only represent tempo as step functions.

In Jiffy, we allow users to specify a tempo curve either as a function of a timescale's source time, or as a function of symbolic position (which is closer to the way tempo is specified in a score). We use piecewise tempo curves where each piece can be defined by parametric curves. A variable-step numerical method is used to integrate the tempo curves when simple analytical solutions are not readily available.

3.1 Differential Equation Formulation

In the following we will use the variable p to denote the position in a timescale, i.e. the logical time in this timescale's reference frame. The variable t will be used to denote the source time (or simply, *time*), i.e. the time in the timescale's parent reference frame (which could be the wall-clock time).

The function *position function*, $P(t)$, transforms the source time into the internal position of the timescale. The *time function*, $T(p)$, transforms the position into the source time. Obviously, $P = T^{-1}$.

[10] One difference, however, is that we use the word tempo here to refer to the ratio of internal STUs over source STUs, rather than the number of beats per minutes, since the latter could depend on the musical meter of the timescale.

[11] `https://antescofo-doc.ircam.fr/Reference/compound_curve/`

A *tempo curve* $\mathcal{T}$ can be either a function of time or position. It maps its parameter to the value of the derivative of the position function at this instant. In the following we will refer to a tempo curve defined as a function of position as an *autonomous tempo curve*, whereas a tempo curve defined as a function of time will be referred to as a *non-autonomous tempo curve*. This naming stems from the formulation of the tempo curve as the right-hand side of an autonomous or non-autonomous differential equation:

$$\frac{dP}{dt}(t) = \mathcal{T}(P(t)) \quad \text{(autonomous)}\,, \text{ or } \frac{dP}{dt}(t) = \mathcal{T}(t) \quad \text{(non autonomous)}\,, \\ \text{with initial condition } P(0) = 0\,. \tag{1}$$

4 Tempo Curves Integration

Tempo curves in Jiffy are defined as piecewise functions. For the sake of brevity, we may refer to an interval and its associated sub-function as a tempo curve *segment*, or simply as a *curve*, where the meaning should be clear from context. Each segment is defined by a start tempo and an end tempo, a duration, an interpolation mode and optional interpolation parameters. We implemented three interpolation modes, namely *constant*, *linear* and *parametric*.

4.1 Integration of Constant and Linear Tempo Curves

Constant and linear tempo curves can be solved analytically. We show below the differential equation of tempo, and the position and time functions for each case.

Constant Tempo.

$$\mathcal{T}(p) = \mathcal{T}_0\,. \tag{2}$$

$$T(p) = \frac{p}{\mathcal{T}_0}\,,\ P(t) = t \times \mathcal{T}_0\,. \tag{3}$$

Autonomous Linear Tempo.

$$\mathcal{T}(p) = \mathcal{T}_0 + \alpha p\,, \text{ where } \alpha = \frac{\mathcal{T}_1 - \mathcal{T}_0}{L}\,. \tag{4}$$

$$P(t) = \frac{\mathcal{T}_0}{\alpha}(e^{\alpha t} - 1)\,, \text{ and } T(p) = \frac{1}{\alpha}\log(1 + \frac{\alpha p}{\mathcal{T}_0})\,. \tag{5}$$

Non-autonomous Linear Tempo.

$$\mathcal{T}(t) = \mathcal{T}_0 + \alpha t\,, \text{ where } \alpha = \frac{\mathcal{T}_1 - \mathcal{T}_0}{L}\,. \tag{6}$$

$$P(t) = \mathcal{T}_0 t + \frac{\alpha}{2}t^2\,, \text{ and } T(p) = \frac{\sqrt{\mathcal{T}_0^2 + 2\alpha p} - \mathcal{T}_0}{\alpha}\,. \tag{7}$$

Numerical Considerations Some of the above time and position functions are indeterminate forms for $\alpha \to 0$. To avoid that problem, we approximate these expressions by a series expansions in α when $|\alpha|$ is smaller than a given threshold. For instance, our approximation of the position function for the autonomous case when is $|\alpha| < 10^{-9}$ is:

$$P(t) \approx \boldsymbol{\mathcal{T}}_0(t + \frac{\alpha}{2}t^2 + \frac{\alpha^2}{6}t^3 + \frac{\alpha^3}{24}t^4 + \frac{\alpha^4}{120}t^5)\,. \tag{8}$$

4.2 Parametric tempo curves.

In this section we will give a definition of a parametric tempo curve, and show the differential equations that need to be solved in order to compute the time and position functions. These equations are then solved by a numerical solver.

An autonomous (resp. non-autonomous) parametric tempo curve segment is defined as a function $\mathcal{C}$ of the position p (resp. of the time t), which describes the same curve in the plane $(p, \boldsymbol{\mathcal{T}})$ (resp. $(t, \boldsymbol{\mathcal{T}})$) as a parametric curve $\boldsymbol{B}(s)$ with components $B_x(s)$ and $B_y(s)$.

Autonomous Parametric Tempo. The differential equation corresponding to an autonomous tempo curve can be written as

$$\frac{dP}{dt}(t) = C(P(t))\,. \tag{9}$$

Position function $P(t)$. The derivative of the position with respect to time is directly expressed by the autonomous tempo curve,

$$\frac{dP}{dt}(t) = B_y(s)\,,\ \text{where } s = B_x^{-1}(P(t))\,. \tag{10}$$

Time function $T(p)$. We operate the change of variable $s = B_x^{-1}(p)$ on Equation 9. Finding the time function is then a matter of solving the differential equation

$$\frac{d\tilde{T}}{ds}(s) = \frac{B'_x(s)}{B_y(s)}\,,\ \text{with } \tilde{T}(s) = T(p)\,. \tag{11}$$

Non-autonomous Parametric Tempo. The definition of the non-autonomous parametric tempo curves can be written as

$$\frac{dP}{dt}(t) = C(t)\,. \tag{12}$$

Position function $P(t)$. Using the change of variable $s = B_x^{-1}(t)$ and the chain rule, we can write the differential equation for the position function as

$$\frac{d\tilde{P}}{ds}(s) = B_y(s)B'_x(s)\,,\ \text{with } \tilde{P}(s) = P(t)\,. \tag{13}$$

Time function $T(p)$. Using the formula for the derivative of inverse functions on Equation 12, we get

$$\frac{dT}{dp}(p) = \frac{1}{\mathcal{C}(T(p))} = \frac{1}{B_y(s)}\,, \text{ where } s = B_x^{-1}(T(p))\,. \tag{14}$$

Numerical Resolution. Although some of the above equations can be solved analytically, using a numerical solver has the advantage of allowing us to control the tradeoff between accuracy and speed, and opens up the possibility of supporting other arbitrary functions to define tempo curves. We use a Cash-Karp [4] solver to numerically solve the tempo curve equations. We follow the general architecture proposed in [11], optimized further by leveraging the fact that these equations are either autonomous or directly integrable.

Bézier Tempo Curves. The above formulation allows the use of any parametric curve, provided that it describes a derivable, non null function. Our specific implementation uses cubic Bézier curves, which are especially versatile, as they allow putting constraints on both endpoints and their first derivative, while ensuring that the curve remains contained inside its control points' convex hull. They are also intuitive to manipulate and map well to the curve-editing interfaces commonly used in animation, audio, and video applications.

An autonomous (resp. non-autonomous) Bézier tempo curve segment is defined by the parametric curve

$$\boldsymbol{B}(s) = \boldsymbol{C}_3 s^3 + \boldsymbol{C}_2 s^2 + \boldsymbol{C}_1 s + \boldsymbol{C}_0\,, \tag{15}$$

where the $\boldsymbol{C}_i$ are the power basis coefficients computed from the Bézier curve's control points. To ensure that the curve describes a function, the cubic function $\boldsymbol{B}_x(s)$ must be monotonous, i.e. if the x_i are the abscissae of the $\boldsymbol{C}_i$, the condition $c_2^2 - 3c_3c_1 \leq 0$ must hold.

Bézier curves evaluation. We should stress out that, although each coordinate of the parametric Bézier curve is cubic with respect to its parameter s, the second coordinate is *not* a cubic function of the first, i.e. the tempo is not a cubic function of position (resp. time). Analytically finding the tempo for a given position (resp. time) indeed requires solving a third order equation.

A faster method is to numerically find the parameter s for a given position (resp. time), up to some desired precision, and then compute the tempo from s. Our implementation first uses the Newton-Raphson root-finding method up to a fixed number of iterations, and falls back to a bisection algorithm if either the value of the derivative falls behind some threshold, or the desired precision is not reached within the maximum iteration count.

An example of a time map produced by tempo curve composed of two Bézier segments is shown in Figure 2. The blue curve shows position as a function of time. The orange stems mark the timeline STUs. The red curve shows the tempo curve, as a function of

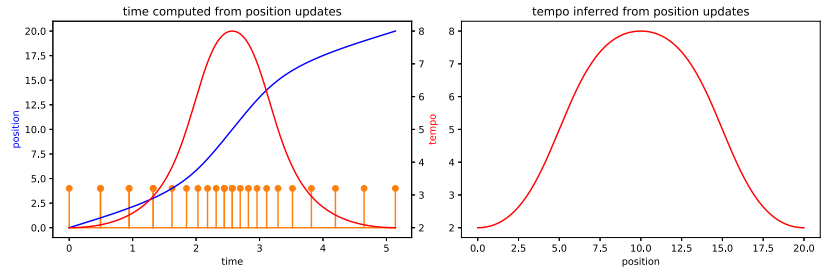

Fig. 2. Time map and beats trace for a tempo curve defined by two Bézier curves.

time (on the left), or as a function of position (on the right). The figure is produced by computing the positions corresponding to a regularly spaced time grid.

5 Scheduler Interface

The Jiffy scheduler is designed to run user code in fibers[12]. Compared to callback-based scheduling APIs (such as [3], [13] or [12]), this doesn't compel the user to break down the control flow of their code into lots of small functions, keeps logically related computations in the same local context, and allows users to easily express dependencies between several workloads. Fibers can also be migrated between threads, allowing a very streamlined way to handle blocking calls without hanging the scheduler.

The scheduler uses the notion of *tasks* to represent a group of fibers executing within the same timescale. Tasks, like timescales, are organized in a parent-children relationship. The API exposes functions to launch new tasks and fibers, to yield and reschedule the current fiber to a future date, or to wait on the completion of other tasks or fibers. It also features functions to move fibers into background jobs to perform blocking operations without blocking the scheduler, and bringing them back to the foreground once done.

Listing 1.1 shows a simple example that launches a task to print a message at regular symbolic intervals with a varying tempo. This task lives for 40 time units unless it is canceled from another fiber, which waits for user input in the background.

6 Conclusion and Future Work

In this paper, we highlighted the need for symbolic time scheduling in show-control software and musical applications. We then described a temporal model based on time

[12] The notions of *fiber*, *coroutine*, or *green thread* are so closely related that the distinction between them, if any, is amenable to debate. One could argue that *green thread* is more appropriate in the context of a virtual machine or runtime environment, while *coroutine* originates from programming language design. The term *fiber* may capture a more general view of the concept.

```
i64 my_task_proc(void* userPointer)
{
        // main task: print a message at each symbolic time unit
        for(int i=0; i<40; i++)
        {
                printf("Hello, world: %i\n", *count);
                sched_wait(1);
        }
        return(0);
}

i64 user_cancel_fiber(void* userPointer)
{
        // go to background so we don't block the scheduler, and wait user input,
        // then bring the fiber to the foreground and cancel the main task
        sched_background();
        while(getchar() != 'q') /* wait 'quit' command */ ;
        sched_foreground();
        sched_task_cancel(*(sched_task*)userPointer);
        return(0);
}

int main()
{
        // launch our main task and apply a tempo curve to it, then launch the
        // user canceling fiber, and wait for the main task to complete.

        sched_curve_descriptor_elt elements[2] = {
                {.type = SCHED_CURVE_BEZIER,
                 .startValue = 2, .endValue = 8, .length = 20,
                 .p1x = 0.5, .p1y = 0, .p2x = 0.5, .p2y = 1},
                {.type = SCHED_CURVE_BEZIER,
                 .startValue = 8, .endValue = 2, .length = 20,
                 .p1x = 0.5, .p1y = 0, .p2x = 0.5, .p2y = 1}};

        sched_curve_descriptor desc = {.axes = SCHED_CURVE_POS_TEMPO,
                                       .eltCount = 2, .elements = elements};
        sched_init();
        sched_task task = sched_task_create(my_task_proc, 0);
        sched_task_timescale_set_tempo_curve(task, &desc);

        sched_create_fiber(user_cancel_fiber, &task, 0);

        sched_wait_completion(task);
        sched_end();
        return(0);
}
```

Listing 1.1. An example of Jiffy's scheduling API.

transformations expressed through tempo curves, and gave a formalism of such curves. We then described how these curves are implemented in the Jiffy scheduler, and presented the API of the scheduler.

In its current form, the scheduler is a local system, only maintaining proper time flow for its host process. Synchronizing timescales across multiple scheduler instances (potentially running on different machines) is the subject of ongoing work.

The kind of synchronization we considered in this paper was only concerned about relative speeds. However, when dealing with ensemble music, the notion of synchronization is really about the relative phase of each musician. We could refer to this type of synchronization as *metric synchronization*. Ableton Link [8] is one of the tools that tackle this problem, and offers an elegant model to build musical structure on top of beat synchronization. However it has some limitations when it comes to multiple tempos and complex polyrhythms. Addressing these scenarios will be the subject of further research.

References

[1] David P. Anderson and Ron Kuivila. “A System for Computer Music Performance”. In: *ACM Transactions on Computer Systems* 8.1 (1990), pp. 56–82.

[2] A. Berndt. “Musical Tempo Curves”. In: *ICMC*. 2011.

[3] Dimitri Bouche and Jean Bresson. “Planning and Scheduling Actions in a Computer-Aided Music Composition System”. In: *Scheduling and Planning Applications Workshop (SPARK)*. Ed. by Israel Science Foundation. Proceedings of the 9th International Scheduling and Planning Applications Workshop. Jerusalem, Israel: Steve Chien and Mark Giuliano and Riccardo Rasconi, 2015, pp. 1–6.

[4] J. R. Cash and Alan H. Karp. “A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right-Hand Sides”. In: *ACM Trans. Math. Softw.* 16.3 (1990), pp. 201–222.

[5] Jean-Michaël Celerier et al. “OSSIA: Towards a Unified Interface for Scoring Time and Interaction”. In: *TENOR 2015 - First International Conference on Technologies for Music Notation and Representation*. Paris, France, 2015.

[6] Thierry Coduys and G. Ferry. “Iannix. Aesthetical/Symbolic Visualisations for Hypermedia Composition”. In: *Sound and Music Computing Conference (SMC)*. 2004.

[7] Arshia Cont. “ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music.” In: *International Computer Music Conference (ICMC)*. Belfast, Ireland, 2008, pp. 33–40.

[8] Florian Goltz. “Ableton Link – A Technology to Synchronize Music Software”. In: *Proceedings of the Linux Audio Conference (LAC 2018)*. 2018, p. 4.

[9] David Jaffe. “Ensemble Timing in Computer Music”. In: *Computer Music Journal* 9.4 (1985), pp. 38–48.

[10] John MacCallum and Andrew Schmeder. “Timewarp: A Graphical Tool for the Control of Polyphonic Smoothly Varying Tempos”. In: *International Computer Music Conference, ICMC 2010* (2010), p. 4.

[11] William H. Press et al. “Integration of Ordinary Differential Equations”. In: *Numerical Recipes in c (2nd Ed.): The Art of Scientific Computing*. USA: Cambridge University Press, 1992, pp. 710–722.

[12] Charles Roberts, Graham Wakefield, and Matthew Wright. “2013: The Web Browser as Synthesizer and Interface”. In: *A NIME Reader*. Ed. by Alexander Refsum Jensenius and Michael J. Lyons. Vol. 3. Cham: Springer International Publishing, 2017, pp. 433–450.

[13] Norbert Schnell et al. “Of Time Engines and Masters an API for Scheduling and Synchronizing the Generation and Playback of Event Sequences and Media Streams for the Web Audio API”. In: *WAC*. Paris, France, 2015.