Clockblocks: A Pure-Python Library for Controlling Musical Time

Marc Evanstein
University of California, Santa Barbara
marc@marcevanstein.com

ABSTRACT

This paper describes clockblocks*, a GPL3.0-licensed pure-Python library for controlling the flow of musical time, which is part of a broader framework for music composition in Python called SCAMP. Clockblocks allows for the coordination of parallel and / or nested clocks running at different tempi, facilitates smooth acceleration and deceleration, and sleeps precisely while compensating for calculation time. The approach presented here is compared with other systems for managing musical time, and further development in terms of coordinating metric phase is considered.*

1. INTRODUCTION

1.1 Context

In recent years there has been a proliferation of interest in, and tools designed for, computer-assisted music composition. Among the options available, one might broadly distinguish between domain-specific languages, such as *SuperCollider* or *Max/MSP*, and frameworks that operate within general-purpose programming languages, such as *abjad* [1] or *jMusic* [2]. While both approaches have advantages, one major advantage of situating a composition framework within a general-purpose language is the wide range of powerful libraries that are made readily available.

Another important distinction exists between languages and frameworks aimed at the direct generation of sound, and those aimed at the creation of a score to be played by live performers. These very different aims necessitate significant differences in design; for instance, speed and efficiency are critical concerns when generating real-time audio, while for score-generation they are much less important. On the other hand, traditional music notation places very significant (and idiosyncratic) constraints on timing and rhythm, as well as on other musical parameters.

SCAMP (Suite for Computer-Assisted Music in Python) [3] is a GPL3.0-licensed framework for musical composition that aims to take advantage of the general-purpose nature and compact, readable syntax of Python, while at the same time situating itself as a hybrid between sound-oriented and notation-oriented frameworks. Although the creation of traditionally notated scores is a central aspect of the framework, SCAMP is nevertheless strongly playback-oriented: rather than interacting with a score, the programmer interacts with a virtual ensemble, listening to and tweaking the resulting playback until he or she is ready to translate the music to traditional western music notation.

The key functions of the SCAMP framework are:

- To provide facilities for flexible and extensible note playback, e.g. via FluidSynth or over OSC. (Effortless microtonality and glissandi have been built in.)
- To manage the flow of multiple interconnected streams of musical time.
- To record note-events, quantize recorded performances sensibly and flexibly, and translate the result to legible music notation, either as MusicXML or as LilyPond (via the *abjad* library).

A key value underlying the development of SCAMP is that of modularity and adherence as much as possible to the Unix Philosophy. For instance, the MusicXML export capability is available separately as *pymusicxml*, the flexible musical Envelope class is available separately as *expenvelope*, and the system for managing musical time is available separately as *clockblocks*. It is this last library that is the subject of this paper.

1.2 Goals

Clockblocks arose to address several recurring problems with the scheduling of note playback events and the recording of note event data in Python:

1. The built-in `time.sleep` function has limited accuracy, especially for longer wait times.
2. Playback is slowed down by script execution, noticeably so if extensive calculations are involved.
3. In multi-part music running in parallel threads, differing calculation times result in drift between the parts. This is especially problematic if note events are to be recorded and quantized.
4. A system for controlling and modulating tempo is needed, ideally one allowing for multiple independent streams operating simultaneously.

Clockblocks solves the first of these problems by defining a `sleep_precisely` function which repeatedly sleeps for half the remaining duration until fewer than 500μs are left, at which point it implements a busy wait for the remaining time. The remaining problems are addressed through a system of interconnected clocks that all inherit from a single master clock. In this way, multiple independent streams of musical time, potentially running simultaneously at different tempi, remain perfectly synchronized.

2. A SIMPLE EXAMPLE

2.1 The Code

The following example will serve to introduce the *clockblocks* API:

```
from clockblocks import Clock

clock = Clock(initial_tempo=60)

def log_timing():
    print(
        "Beat:", clock.beats(),
        "Time:", round(clock.time(), 2),
        "Tempo:", round(clock.tempo, 2)
    )

while clock.beats() < 4:
    log_timing()
    clock.wait(1)

# change to 120 bpm (2 beats per second)
clock.tempo = 120
while clock.beats() < 8:
    log_timing()
    clock.wait(1)

# gradually slow to 30 bpm over 8 beats
clock.set_tempo_target(30, 8)
while clock.beats() < 20:
    log_timing()
    clock.wait(1)
```

The resulting output of this program is:

```
Beat: 0.0, Time: 0.0, Tempo: 60.0
Beat: 1.0, Time: 1.0, Tempo: 60.0
Beat: 2.0, Time: 2.0, Tempo: 60.0
Beat: 3.0, Time: 3.0, Tempo: 60.0
Beat: 4.0, Time: 4.0, Tempo: 120.0
Beat: 5.0, Time: 4.5, Tempo: 120.0
Beat: 6.0, Time: 5.0, Tempo: 120.0
Beat: 7.0, Time: 5.5, Tempo: 120.0
Beat: 8.0, Time: 6.0, Tempo: 120.0
Beat: 9.0, Time: 6.59, Tempo: 87.27
Beat: 10.0, Time: 7.37, Tempo: 68.57
Beat: 11.0, Time: 8.34, Tempo: 56.47
Beat: 12.0, Time: 9.5, Tempo: 48.0
Beat: 13.0, Time: 10.84, Tempo: 41.74
Beat: 14.0, Time: 12.37, Tempo: 36.92
Beat: 15.0, Time: 14.09, Tempo: 33.1
Beat: 16.0, Time: 16.0, Tempo: 30.0
Beat: 17.0, Time: 18.0, Tempo: 30.0
Beat: 18.0, Time: 20.0, Tempo: 30.0
Beat: 19.0, Time: 22.0, Tempo: 30.0
```

Note that the faster the tempo, the less time advances for a given beat, and the slower the tempo, the more time advances. The speed of a clock can be set using any of three interrelated properties: its rate, its beat length, and its tempo. These are defined as follows:

$$R = 1/L_b \tag{1}$$

$$T = 60 \cdot R = 60/L_b \tag{2}$$

Where L_b represents beat length and is measured in seconds (at least on a top level clock), R represents rate and is measured in in beats per second, and T represents tempo and is measured in beats per minute. Setting any one of these properties for a clock automatically sets the other two. In some ways, rate and beat length are the most natural descriptors, especially when clocks are nested inside of each other. However, tempo is retained as a property because of its associated musical intuition.

2.2 Implementation

Each clock internally uses a `TempoEnvelope` object to manage changes of tempo. `TempoEnvelope` is a subclass of the `Envelope` class from the SCAMP package *expenvelope*, which defines a piecewise exponential curve similar in function to the `Env` object in *SuperCollider* [4]. In practice, this means that any accelerandi or ritardandi can be given a non-linear shape, increasing the expressive potential of clockblocks.

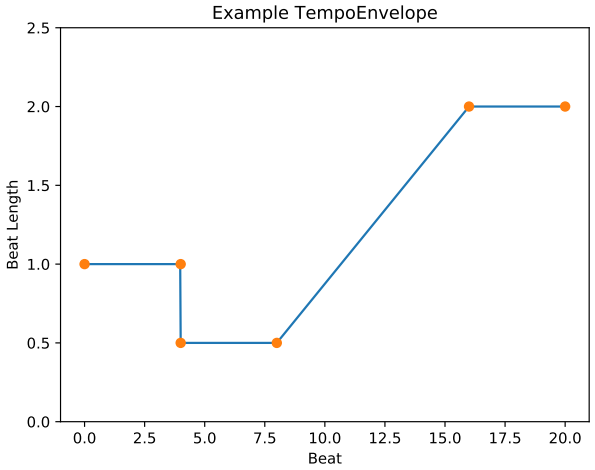

Figure 1. Graph of the clock's TempoEnvelope from the example above

Although the user is likely thinking in terms of rate or tempo, internally the tempo envelope is based on beat length for ease of calculations. Figure 1 shows this beat length curve for the above example; when the tempo jumps to 120 bpm on beat 4, the beat length cuts to 0.5, and then from beat 8 to beat 16 it slowly increases to 2 during the ritardando. The `TempoEnvelope` class keeps track internally of the current beat, and whenever the user calls `clock.wait(beats)`, the area under the curve is integrated from the current beat forward to the destination beat to determine the associated wait time in seconds.

3. PARALLELISM

3.1 Parallel Clocks Example

The above example featured a single stream of musical time. However, the true strength of *clockblocks* lies in its ability to coordinate multiple parallel streams of time, as in the following example:

```
from clockblocks import Clock

master = Clock()

def child1(my_clock: Clock):
    while my_clock.beats() < 6:
        print("Child Clock 1 at beat {}".
              format(my_clock.beats())
        my_clock.wait(1)

def child2(my_clock: Clock):
    while my_clock.beats() < 3:
        print("Child Clock 2 at beat {}".
              format(my_clock.beats())
        my_clock.wait(1/3)

master.fork(child1)
master.fork(child2)
master.wait_for_children_to_finish()
```

In this example we create a master clock and define two parallel processes, `child1` and `child2`. The first function prints every beat until beat 6, while the second prints every third of a beat until beat 3. We then fork these functions on the master clock. The results are as follows:

```
Clock 1 at beat 0.0
Clock 2 at beat 0.0
Clock 2 at beat 0.333
Clock 2 at beat 0.667
Clock 1 at beat 1.0
Clock 2 at beat 1.0
Clock 2 at beat 1.333
Clock 2 at beat 1.667
Clock 2 at beat 2.0
Clock 1 at beat 2.0
Clock 2 at beat 2.333
Clock 2 at beat 2.667
Clock 1 at beat 3.0
Clock 1 at beat 4.0
Clock 1 at beat 5.0
```

Note that the child functions take a single argument which gets passed a handle to the clock being forked. It is also possible to call `get_current_clock()` to capture the clock running at any given moment.

3.2 Parallel Clocks Implementation

Whenever a child clock calls `wait`, it registers a wake up time with its parent and then pauses execution until the parent rouses it. The parent clock maintains a cue of wake up calls from its children, and whenever it calls `wait`, it looks to see if there is a child clock with a wake up time in the near future so that it can rouse it at the appropriate time. An example sequence of events with both parent and child starting at t = 0 might be as follows:

```
t = 0:
   - Child calls wait(0.5), registers
     wake up time of 0.5 with parent.
   - Parent calls wait(1), sees that
     a child is set to be woken up at
     t = 0.5, and so waits instead
     for 0.5 beats.

t = 0.5:
   - Parent wakes up and rouses child
   - Child wakes and calls wait(1.0),
     registering a wake up time of 1.5
     with parent.
   - Parent sees no other child wake
     events during the rest of its wait
     of 1, and so sleeps for the
     remaining 0.5.

t = 1.0:
   - Parent wakes from its sleep, calls
     wait(2) this time, and sees that a
     child has registered a wake up
     time of 1.5. As a result, it waits
     0.5 second.

t = 1.5:
   - Parent wakes up and rouses child.
   - Child wakes up and chooses to
     terminate process.
   - Parent sees no other child wake
     events during the rest of its
     wait of 2, and so sleeps for the
     remaining 1.5.
t = 3.0:
   - Parent wakes, sees it has no
     children, suffers from empty
     nest syndrome.
```

When a forked function reaches the end of its execution, the child clock associated with it is terminated. In the example in Sec. 3.1, the master clock, acting as the parent to both child clocks, calls `wait_for_children_to_finish`, which causes it to wait indefinitely, rousing its child clocks at the appropriate times until all children have finished execution.

Note that child clocks can fork their own child clocks, and so on. In this case, a clock may find itself in the role of both parent and child, waking its children at the appointed times, and registering a wake up call with its parent whenever it wishes to wait itself. Only a master clock, a clock with no parent, actually calls `time.sleep` (or rather, the more precise version explained in Sec. 1.2). All other clocks simply register a wake up call with their parent when they wish to sleep.

4. COMPENSATING FOR CALCULATION TIME

One of the initial problems that *clockblocks* was designed to solve was the fact that, unless compensated for in some way, any calculation time on a thread will cause that thread to slow down relative to the sum of all of its calls to `time.sleep`. It should be clear from the above that this problem is already solved for all but the master clock, since wake up times are absolute and will not drift. It only remains to ensure that the master clock itself takes calculation time into account.

When a master clock wakes up, it immediately notes down the current time. Then, after all relevant calculations have taken place and a new `wait` call is made, it refers back to the time that it originally woke up in determining how long to sleep.

In some cases, when calculations are intensive and the wait time is short, it may already be past the the desired wake up time, and the clock finds itself running behind. At this point there are two main options:

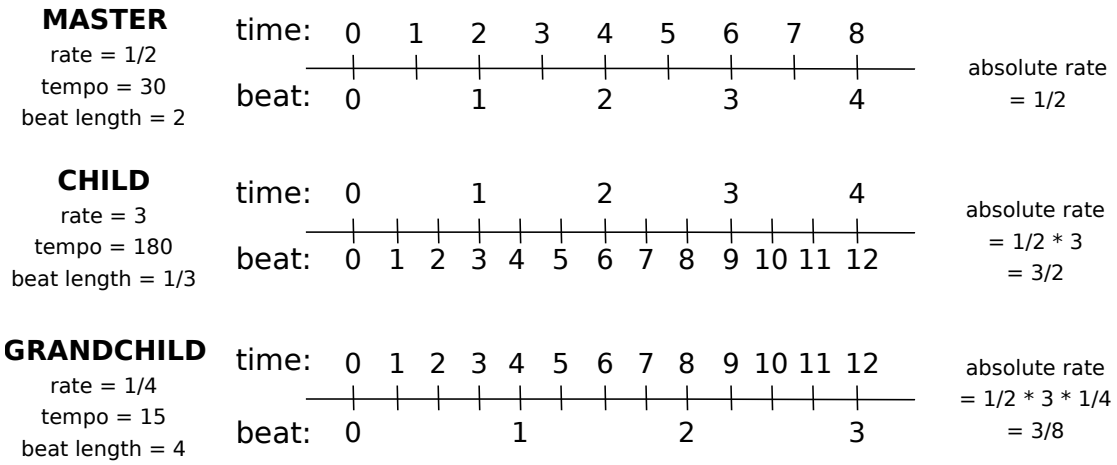

Figure 2. Relationship between the rates of three "generations" of clocks running at different tempi

1. Allow the clock to stay behind and handle subsequent wait times as faithfully as possible.

2. Try to catch up in future calls to `wait` by not waiting at all until the clock has caught up.

Both options are available in *clockblocks*. The first is termed a "relative" timing policy (since it emphasizes keeping individual wait times as accurate as possible), while the latter is termed an "absolute" timing policy (since it emphasizes not drifting from the absolute time at which events should have occurred). If the playback from clockblocks is being coordinated in any way with that of an external application, an absolute timing policy would likely be preferred; if not, a relative policy may be preferred.

The default used by *clockblocks* is actually a third, hybrid approach. This policy allows for time to be shaved off of subsequent calls to `wait`, but only to a certain degree. Thus, rather than catching up all at once, the clock catches up in small increments. In many cases this is sufficient to remain faithful to the absolute times at which events should occur without distorting subsequent wait times noticeably.

5. NESTED CLOCKS

5.1 Tempo Inheritance

The true power of *clockblocks* as a library lies in the fact that each clock, regardless of its place in the family hierarchy, is allowed to have and manipulate its own tempo. However, the actual speed at which a clock runs depends not only on its own tempo, but also on the tempo of its parent, and its parent's parent, etc. all the way on up to the master clock.

To understand this better, consider that a clock has two different views on the passage of time: what beat it is on and how much actual time has passed. As we have seen above, these two properties are related by the clock's beat length; time passed is the integral of beat length with respect to beats passed.

In *clockblocks*, each clock inherits its sense of time from its parent; a beat in the parent clock constitutes a "second" of time in the child clock. The word "second" here is in quotes, because unless the clock in question is the master clock, it is not a true second, but rather a second as filtered through temporal distortions of its parents.

For instance, in Figure 2, we consider three generations of nested clocks: a master clock running at rate $1/2$, its child (e.g. the result of a call to `fork`) running at rate 3, and its child's child, running at rate $1/4$. Note that the child's sense of time is inherited from the master clock's beat rate, and the grandchild's sense of time is inherited from the child's beat rate. Thus is should be clear from this picture that the tempi of clocks in a parent / child relationship multiply. The *absolute rate* of the grandchild clock – its rate with respect to wall time – is the product of its own rate, its parent's rate, and its parent's parent's rate.

What is not depicted in the above example is that each clock can in fact be smoothly changing rate according to manipulations of its tempo envelope. The actual amount of wall time corresponding to a wait of, say, two beats in the grandchild clock is calculated by first integrating for two beats under the grandchild clock's tempo envelope, then taking the result and integrating for that many beats under the child clock's tempo envelope, and then taking *that* result and integrating for that many beats under the master clock's tempo envelope.

5.2 A Nested Tempo Example

The following example will server to illustrate how nested clocks can follow different, but interacting, tempo envelopes. It also introduces several new features for shaping a clock's tempo over time:

```
from clockblocks import Clock
import math


def child_process_1(clock: Clock):
    clock.apply_tempo_function(
        lambda t: 60 + 30 * math.sin(t),
        duration_units="time"
    )
    # do something musical here
    clock.wait(40, units="time")
```

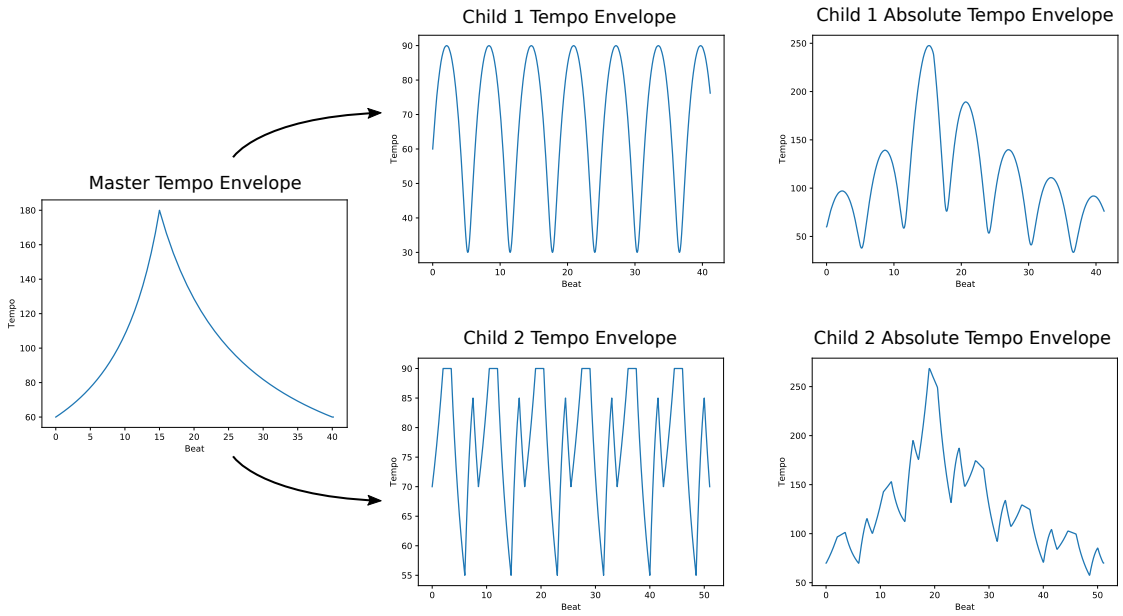

Figure 3. Effect of a master tempo envelope of the absolute tempo envelopes of its children. Note that, unlike in Figure 1, these graphs are of tempo, rather than the beat length.

```
def child_process_2(clock: Clock):
    clock.apply_tempo_envelope(
        [70, 90, 90, 55, 85, 70],
        [2, 1.5, 2.5, 1.5, 1.0],
        curve_shapes=[0, 3, 0, -2, 0],
        loop=True
    )
    # do something musical here
    clock.wait(40, units="time")

master = Clock()
child_1 = master.fork(child_process_1)
child_2 = master.fork(child_process_2)

master.set_rate_target(3, 15)
master.set_rate_target(1, 40,
                       truncate=False)
master.wait_for_children_to_finish()
```

Here, we create a master clock and spawn two child processes. One of these processes follows a sinusoidal tempo envelope, which we create by calling `clock.apply_tempo_function`. (Internally, this tempo function is being approximated by exponential curve segments, since all tempo envelopes are piece-wise exponential.) The other clock instead calls `apply_tempo_envelope` to define this piece-wise exponential curve directly. Since the loop flag has been set to `True`, the tempo envelope repeats for as long as the clock is alive.

The master clock itself changes tempo over the course of the example, going to a rate of 3 over the course of 15 beats and back to a rate of 1 after 40 beats have passed. Note that the truncate flag has been set to `False` in the second call to `set_rate_target`; by default when a rate / tempo / beat length target is set, any existing targets are cut off, but by setting the truncate flag to `False`, the first target remains in place.

If, after the code above, we call the following, we generate the plots shown in Figure 3:

```
master.tempo_envelope.show_plot()
child_1.tempo_envelope.show_plot()
child_2.tempo_envelope.show_plot()
child_1.extract_absolute_tempo_envelope().
        show_plot()
child_2.extract_absolute_tempo_envelope().
        show_plot()
```

As Figure 3 illustrates, the tempo envelope of the master clock affects those of the two child clocks. The plots in the middle column show the tempo envelopes of the child clocks with respect to their parent (the master clock), and show the sinusoidal variation and explicitly defined tempo envelope described above. On the other hand, the plots in the right column show their *absolute tempo envelopes*, i.e. their tempos with respect to wall time, having been altered by the acceleration and deceleration of the master clock.

Before we leave this example, it should be pointed out that the child clocks call `wait` with the additional keyword argument `units="time"`. What this does is instruct the clock to wait however many beats will correspond to 40 units of time, (which is the same as 40 beats in the parent clock). This affords the ability to coordinate with the parent clock; for instance, in this case the master clock has been instructed to accelerate and decelerate over the course of 40 beats, which will be the exact same length as 40 units of time in the two child processes.

Notice also that time units have been specified for the sinusoidal tempo function defined on the first child clock. This is why, in the graph of its tempo envelope, the peaks are wider than the troughs; this is a graph of tempo with respect to *beats*, and it will take more beats to cover a given amount of time at a faster tempo than at a slower tempo. On the other hand, the tempo envelope applied to the second child clock is in the units of beats (which is the default), so the graph appears undistorted.

6. COMPARISON WITH OTHER APPROACHES

In contrast to audio programming environments like *Gibber* [5] and *ChucK* [6], *clockblocks* does not concern itself with the audio thread directly, or with sample accuracy. As the time management engine of SCAMP, *clockblocks* is designed for scheduling events at the rate of notes and sound objects. The actual production of sound samples happens externally, for instance via FluidSynth or OSC messages to some other external instrument. Temporal precision is of course desired, but sample accuracy is not necessary.

Clockblocks does bear some similarity to ChucK in its syntax, however: the user performs operations, sets up processes and then then advances time. As in ChucK, subprocesses can be forked, and these subprocesses can themselves fork subprocesses. However, *clockblocks* makes use of nested tempo relationships in a way that ChucK does not, or at least not natively.

SuperCollider [7] provides the ability to control multiple independent streams of tempo via the TempoClock object. In some ways, *clockblocks* also resembles SuperCollider in its server / client dichotomy; the client language that is in charge of scheduling is separate from the process of generating audio samples. However, it is not possible in Supercollider to nest TempoClocks, and any accelerandi and ritardandi must be accomplished through rapid incremental changes of tempo.

Thus, the main contribution of *clockblocks* is that it combines the nested structure of an environment like ChucK with the ability to smoothly manipulate tempo at any layer of this structure. It also provides this functionality in Python, a general purpose programming language that offers access to a vast array of packages from a wide variety of disciplines, and one that has at present a dirth of options for managing musical time.

7. DIRECTIONS FOR FURTHER DEVELOPMENT

Although in its current implementation *clockblocks* does not offer sample accuracy (nor is this necessary for its role within SCAMP), the nested tempo approach presented here has the potential for broader application, including some contexts (like the scheduling of microsonic events) where sample-accuracy would be desired. Therefore, one natural direction for further work would be to translate this system to a language like C++, using it generating a cue of sample-clock timestamped events. Even within its current Python implementation, one planned development is to allow time-sensitive playback events, such as OSC messages or calls to an external synthesizer, to be scheduled on a queue for later dispatching by an audio callback (for instance, via a PyAudio's PortAudio bindings).

Another area of *clockblocks* currently being developed is the ability to specify and synchronize rhythmic phase. Here we take inspiration from "Tempocurver" [8], developed by Matthew Wright in collaboration with composer Edmund Campion, as well as from CNMAT's subsequently developed Max / MSP external, "Timewarp" [9].

As we have seen, *clockblocks* makes it possible to coordinate clocks so that they reach specific tempi at specifically appointed times. However, it may also be musically important to coordinate rhythmic phase; for instance, we may want two clocks that are following different tempo curves to land on the beat at the exact same moment in time. This requires fine tuning of the relationship between beats passed and time passed,

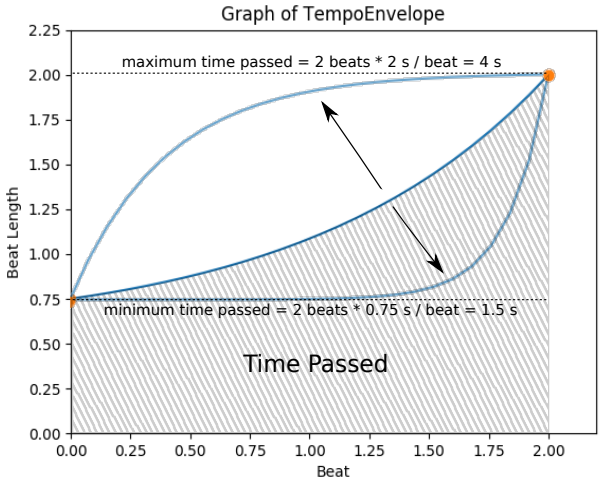

Figure 4. Illustration of the relationship between curvature and the length (in time) of an accelerando.

As Figure 4 illustrates, the relationship between the number of beats passed and the amount of time passed during a segment of a TempoEnvelope is mediated not only by the start and end tempo, but also by the curve shape. In the illustrated accelerando, by varying the curvature, we can adjust the amount of time passed to anywhere between 1.5 and 4 seconds. Thus, if we wished for this 2 beat accelerando to last precisely 3 seconds, we would need only to find the appropriate curve shape.

By using adjustments of this nature, it should be possible to specify the desired metric phase (in terms of time, or beats in the parent clock) at the end of an accelerando or ritardando.

8. CONCLUSIONS

Reviewing the initial goals of *clockblocks*, it is hopefully now clear to the reader that the system described:

1. Is capable of waiting significantly more precisely than the standard call to `time.sleep`.
2. Takes efforts to compensate for calculation time, and has an intelligent system for adjusting when calculation time lasts longer than an intended wait time.
3. Keeps track of multiple interconnected threads of musical time and ensures that these threads remain in lockstep with one another.
4. Allows for complex control and modulation of tempo, and for nested tempo relationships.

These properties make *clockblocks* an excellent foundation for a note-event-based playback and recording framework in Python. Several different streams of time can easily coexist, and the resulting music can be recorded and quantized in relation to any one of these streams. This has particularly exciting implications for the playback and score generation of polytemporal music.

9. REFERENCES

[1] T. Bača, J. Oberholtzer, J. Treviño, and V. Adán, "Abjad: An Open-Source Software System for Formalized Score Control," in *Proceedings of the First International Conference on Technologies for Music Notation and Representation – TENOR'15*, 2015. [Online]. Available: http://tenor-conference.org/proceedings.html#2015

[2] A. Brown, *Making Music with Java*. Andrew R. Brown, May 2009.

[3] M. Evanstein, "SCAMP: a Suite for Computer-Assisted Music in Python," https://github.com/MarcTheSpark/scamp, 2018.

[4] SuperCollider 3 Documentation Contributors, "Env — SuperCollider 3.10.0 Help," http://doc.sccode.org/Classes/Env.html, 2018.

[5] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer, "Gibber: Abstractions for Creative Multimedia Programming," in *Proceedings of the ACM International Conference on Multimedia - MM '14*. Orlando, Florida, USA: ACM Press, 2014, pp. 67–76. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2647868.2654949

[6] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, vol. 39, no. 4, pp. 10–29, Dec. 2015. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/COMJ_a_00324

[7] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, Dec. 2002. [Online]. Available: https://doi.org/10.1162/014892602320991383

[8] M. Wright and E. Campion, "Tempocurver," https://github.com/CNMAT/CNMAT-Externs/tree/master/java/tempocurver, 2001.

[9] J. MacCallum and A. Schmeder, "Timewarp: A Graphical Tool For The Control Of Polyphonic Smoothly Varying Tempos," in *Proceedings of the 2010 International Computer Music Conference, ICMC 2010, New York, USA, 2010*, 2010. [Online]. Available: http://hdl.handle.net/2027/spo.bbp2372.2010.075