University of California
Santa Barbara

SCAMP:
Suite for Computer-Assisted Music in Python

A project document submitted in partial satisfaction
of the requirements for the degree

Master of Science
in
Media Arts and Technology

by

Marc Paul Evanstein

Committee in charge:

Professor Curtis Roads, Chair
Professor Clarence Barlow
Lecturer Karl Yerkes

July 2019

SCAMP:

Suite for Computer-Assisted Music in Python

by

Marc Paul Evanstein

To the Making and Breaking of Rules

Acknowledgements

The work presented here owes a great deal to my colleagues and friends at the UC Santa Barbara Departments of Music and of Media Arts and Technology. In particular, I owe a special thanks to the members of my committee: Clarence Barlow, Curtis Roads, and Karl Yerkes. The many unique perspectives on music and music-making that I encountered at UC Santa Barbara were crucial to identifying and understanding the musical problems addressed in this work.

On a personal level, the support of my family and friends, and above all of my wife Emily, lies behind every significant undertaking I accomplish. Our cat Minuet also deserves mention as the inspiration for the cute, yet roguish acronym "scamp".

Abstract

SCAMP:

Suite for Computer-Assisted Music in Python

by

Marc Paul Evanstein

This document consists of two papers describing the SCAMP (Suite for Computer-Assisted Music in Python) framework for music composition. The first — and more substantial — of these papers outlines the framework as a whole, discussing its motivating principles and design goals, stepping through the key features of its API, and situating it within the context of other tools for computer-assisted composition. The second paper goes into further detail about the sub-package of SCAMP for managing the flow of musical time, entitled *clockblocks*.

The Code

Both SCAMP and *clockblocks* are hosted on PyPI (Python Package Index), where instructions for installation and links to the source code can be found:

`https://pypi.org/project/scamp/`

`https://pypi.org/project/clockblocks/`

Contents

Paper 1

SCAMP: A Suite for Computer-Assisted Music in Python

This paper introduces SCAMP, a computer-assisted composition framework in Python designed to bridge the gap between the continuous timing of synthesis-based frameworks and the discrete timing of notation-based frameworks. SCAMP allows the composer to quickly audition and iterate over musical ideas based on the sonic result, and then flexibly quantize and export the music in western notation. SCAMP provides varied and highly extensible utilities for playback, features easy playback and notation of microtonality and glissandi, has a flexible clock system capable of coordinating multiple streams of music following separate tempo curves, and can export notation in the form of either MusicXML or Lilypond (via the *abjad* library). The goal of the framework is to address pervasive technical challenges while imposing as little as possible on the aesthetic choices of the user. For this reason, care has been taken to separate key elements of SCAMP's functionality into self-contained subpackages. This design, along with SCAMP's many output channels, allows a user to pick and choose the functionality they need and to abandon the framework when it no longer serves the aims of a given composition.

1.1 Introduction

1.1.1 Motivation

Consider a composer who wishes compose a piece for string quartet based on climate data. Having downloaded the data in CSV format, they wish to process it and experiment with different mappings by ear. Finally, having crafted their preferred mappings into an overarching musical form, they wish to output some preliminary notation, and then reshape the result by hand in their preferred score-writing software.

Or consider a composer who wants to write a piece for instrument and electronics where a simple mass-spring simulation generates a stream of glissandi, which simultaneously drives a modular synthesizer and results in written notation for the instrumentalist.

Or consider a composer who wishes to write an algorithmically generated piano concerto in which the piano and orchestra follow separate accelerating and decelerating tempo curves. The pianist requires a score notated from the point of view of the piano's tempo curve, while the conductor requires a score notated from the point of view of the orchestra's tempo curve, perhaps with a renotated piano part for reference.

What these scenarios have in common is the translation of musical data between different domains. In particular, all three contend with the transition between the continuous domain of sounding music and the discrete (and idiosyncratic) domain of notated music.

These considerations were the driving forces behind the creation of SCAMP (Suite for Computer-Assisted Music in Python), a GPL3.0-licensed pure-Python framework for music composition, which is the subject of this paper.

1.1.2 Overview

Computer-assisted composition offerings can be broadly divided into two groups: those aimed at the direct creation of sound, which usually treat time, pitch, and other musical parameters as continuous (e.g. *PureData* [1], *Max/MSP* or *SuperCollider* [2]), and those that aimed at the creation of a traditional western score (e.g. *abjad* [3], *OpenMusic* [4], *music21* [5]), which — due to the constraints imposed by notation — generally work with time in a discrete way.

SCAMP is designed to bridge these two worlds. Instead of a score, the composer interacts with an ensemble of virtual instruments, auditioning musical ideas in continuous time, outside of notational constraints. Once this result is deemed satisfactory, the user can then flexibly quantize the result and export it as music notation, in the form of either MusicXML [6] or LilyPond [7] (via the *abjad* library).

Time in SCAMP is managed with a sophisticated system of clocks, capable of remaining tightly coordinated while following arbitrary, intertwining tempo curves. Moreover, nested relationships are possible; processes can spawn accelerating or decelerating subprocesses. Although the concept of tempo suggests beats and fractional units of time, all durations are in fact floating-point, defaulting to seconds if no tempo is indicated. When notation is needed, it can be generated in reference to any of the clocks used.

SCAMP is fundamentally note-event-based; however, notes are very flexible objects, capable of incorporating arbitrary continuous parameter curves. Microtonality, glissandi, and continuous volume envelopes are simple to achieve. In fact, so long as the playback method is designed to respond to them, arbitrary parameters can also be included and shaped in a continuous fashion,

It should be noted that SCAMP is not a synthesis engine, instead offloading playback to one of the following:

- SoundFont-based [8] playback via *pyFluidSynth* [9]. (SCAMP handles the pitch-bends and channel management required for microtonality and glissandi, hiding such complications from the end user.)
- Playback using an external synthesizer via an outgoing MIDI stream. (SCAMP likewise handles pitchbends and microtonality.)
- Playback using an external synthesizer via OSC [10]
- User-defined playback implementations created by implementing atomic playback functions

The goal of the SCAMP framework is to act as a hub that connects the composer to other resources, while offering as open-ended a mental model as possible. As such, it does not aim to offer utilities, such as generative toolkits or scalar / harmonic models, which align with particular aesthetic approaches. Such utilities may be imported from third-party libraries, or may be constructed by the composer. (The *scamp_extensions* package also exists as a repository for such functionality.)

The basic features and limitations of SCAMP are summarized in Table 1.1.

1.1.3 Designed for Flexibility

Fig. 1.1 illustrates both the internal and external dependency structure of SCAMP.

The packages in yellow represent external dependencies. These dependencies connect SCAMP to various forms of musical input and output: LilyPond notation (via *abjad*), SoundFont interpretation and playback (via *sf2utils* [11] and *pyFluidSynth* [9]), MIDI input/output (via *python-rtmidi* [12]), and OSC input/output (via *pythonosc* [13]).

The packages in blue together comprise the SCAMP suite. The *scamp* package itself contains the main functionality of the suite, with time management (*clockblocks*), param-

Features	Limitations
• Compose in continuous time, quantize to notation in discrete time	• Not designed for direct sound synthesis
• Coordinate multiple time streams using nestable clocks with smoothly varying tempi	• Note based (though broadly)
• Continuous pitch-space, glissandi, volume envelopes and other continuous parameter curves	• Generative toolkits, models of scales, pitch class sets, etc. not included in the main library
• Playback using SoundFonts, MIDI stream, or OSC. User-defined implementations also possible.	• Timing is not sample-accurate (does not run on the audio thread)
• Output notation via Lilypond or MusicXML	• Not designed for live-coding (though improving live input is a future goal)
• Easily understood syntax, with significant flexibility hidden behind sensible defaults	• No graphical front-end

Table 1.1: Key features and limitations of SCAMP

eter shaping and control (*expenvelope*), and MusicXML export (*pymusicxml*) separated out into self-contained packages. Note that *clockblocks* relies on *expenvelope*, since the tempo curve of a clock derives from the `Envelope` class.

This modular structure, inspired by the Unix Philosophy, was chosen for two key reasons:

- Easier maintenance: keeping different aspects of functionality separate keeps the scope and upkeep of each piece more manageable.
- Discardability and reusability: not all composers will need all of the functionality that SCAMP has to offer. For instance, some may be interested only in polyphonic

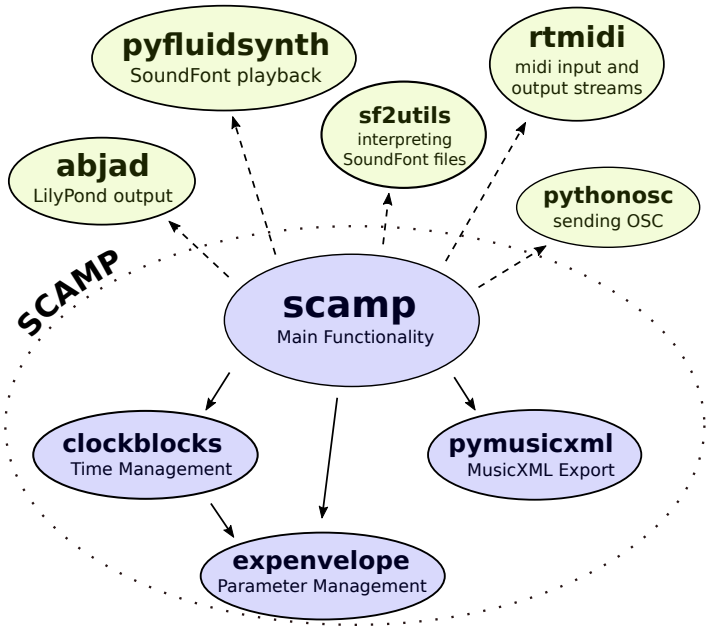

Figure 1.1: Internal and external dependencies of SCAMP

tempo control or direct MusicXML export. By isolating these components from the rest of SCAMP, composers with different goals can potentially incorporate them into other frameworks.

This second point is an important one: every composer develops a unique workflow, reflecting their unique set of aesthetic concerns. For most, this ultimately means patching together a variety of tools. When tools are bundled together, this patching process can become cumbersome. SCAMP's modular structure allows composers to use (and perhaps repurpose) only that which is relevant to them.

One final aspect of SCAMP that provides enormous flexibility for the composer is that it is situated within the broader Python ecosystem. This affords:

- Easy access to a wide variety of libraries that can be adapted for musical purposes, such as data processing (e.g. *numpy*, *sci-py*, *matplotlib*) and machine-learning (e.g. *tensorflow*, *scikit-learn*) toolkits.

- In particular, access to the many forms of data input and output available through both standard Python and third-party libraries.

Taken together, the goal of all of these design choices to create a framework that is as adaptable as possible to the needs of different composers and compositions.

1.2 Introductory Examples

1.2.1 "Hello World"

As an introduction to the SCAMP API, we begin with a program that plays a short arpeggio:

```
# import the scamp namespace
from scamp import *

# construct a session object
s = Session()

# add a new violin part to the session
violin = s.new_part("Violin")

# looping through the MIDI pitches
# of a C major arpeggio...
for pitch in [60, 64, 67, 72]:
   # play each pitch sequentially
   # with volume of 1 (full volume)
   # and duration of half a beat
   violin.play_note(pitch, 1, 0.5)
```

The result of this program will be the sound of a violin playing a C major arpeggio, at max volume, over the course of two seconds. The first step, after importing the SCAMP namespace, is to create a `Session` object. Most SCAMP programs will start in this

way, as the `Session` object is the central hub through which most of the functionality of SCAMP flows. In the same way that a session in a DAW encompasses tracks, transport, and recording functionality, the `Session` object in SCAMP inherits from — and thereby combines the functionality of — an `Ensemble`, a master `Clock`, and a `Transcriber` (see Figure 1.2).

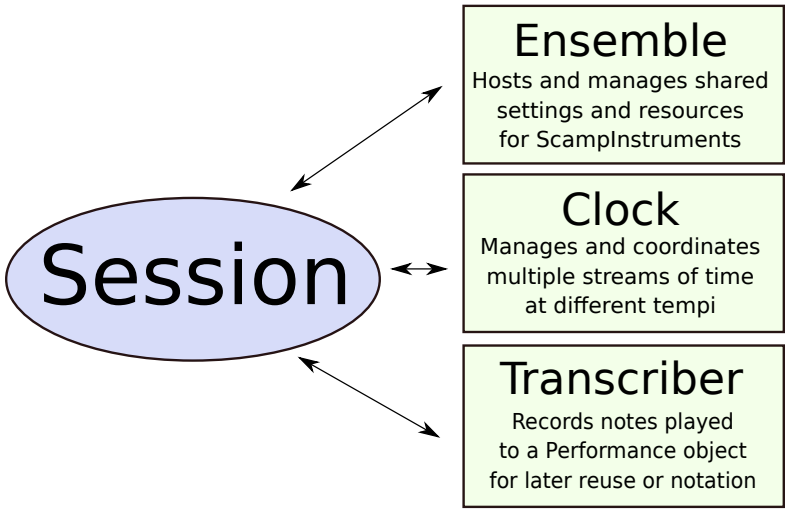

Figure 1.2: The combined functionality of a Session object

The `Ensemble` functionality of the Session object — i.e. its role as a host of the instruments being used — is employed in the third line, where a new violin part is created and stored in the `violin` variable. By default, new parts created in this way will play back via *pyfluidsynth* using a General MIDI Soundfont, with fuzzy string matching being used to find an appropriate preset based on the name of the part. Finally, the call to `play_note` takes three arguments: MIDI pitch (where 60 = Middle C), volume (on a scale from 0 to 1), and length in beats.

1.2.2 Duration

One might reasonably ask at this point how long a beat is. As mentioned above, `Session` inherits from the `Clock` class, which is defined in *clockblocks*, a subpackage of

SCAMP detailed in a separate paper [14]. When a `Session` object (or `Clock` object more generally) is constructed, it becomes the default clock used on the current thread, and is also assigned a default tempo of 60 BPM, or one beat per second. By way of these defaults, when `play_note` is called in above example, it looks to see which clock is operating on the current thread, finds the session (`s`) we created, and plays the note for 0.5 beats on that clock (which corresponds to 0.5 seconds at the default tempo of 60). In this way, the whole four-note arpeggio lasts two seconds.

We can play back the arpeggio faster or slower simply by setting the session's tempo attribute. For instance, if we add the following line directly after the second line in the above example, the arpeggio will last twice as long:

```
s.tempo = 30
```

Calls to `play_note` are blocking by default, not moving on to the next line until the note has finished. This conforms to musical expectations (playing a note takes time), and is a model of musical time similar to that used by the Euterpe language [15]. However, SCAMP also allows the user to start and end notes manually. In fact, internally, a call to `violin.play_note(60, 0.7, 1.5)` is essentially equivalent to:

```
# start playing a middle C with volume 0.7
# returns handle for further manipulation
note = violin.start_note(60, 0.7)
wait(1.5)
note.end()
```

(Note that, like the `play_note` function, the `wait` function captures the current clock from context and takes a number of beats as an argument.)

Finally, in addition to the options above, it is also possible to call `play_note` in a non-blocking manner by setting the "blocking" keyword argument to `False`. For instance,

the following would play two notes, each lasting two seconds, that overlap by one second:

```
violin.play_note(60, 1, 2, blocking=False)
wait(1)
violin.play_note(64, 1, 2)
```

It is important to note that durations do not have to be of rational lengths. For instance, after constructing the session and adding a violin part, the following would repeatedly loop through our arpeggio using random (floating point) durations between 0.5 and 1.5 seconds:

```
# import the random module from
# the python standard library
import random

while True: # loop forever
   for pitch in [60, 64, 67, 72]:
      # pick a duration between 0.5 to 1.5
      dur = random.uniform(0.5, 1.5)
      # play a note of that duration
      violin.play_note(pitch, 1, dur)
```

The default tempo of 60 BPM was chosen because it allows the composer to think in terms of durations in seconds if so desired. This exemplifies a guiding philosophy of the SCAMP framework: musical parameters are treated as continuous during the compositional process unless explicitly quantized. It is only later on, when converting the music to a score, that quantization becomes a necessity.

1.2.3 Generating Notation

In order to generate notation from the above examples, we use the third function of the `Session` object: its role as a `Transcriber`. The purpose of a `Transcriber` is to keep

track of any notes played by the instruments that have registered with it, and to save the result as a `Performance`, which is essentially a note-event list. This performance can then be quantized and converted into a `Score`, which is capable of saving either to LilyPond (via the *abjad* library) or to MusicXML (via *pymusicxml*, a part of SCAMP). The advantage of first transcribing the music as a `Performance` and then converting it to a `Score` is that the performance exists in continuous time and parameter space, outside of notational constraints. In fact, `Performances` can be replayed, either in part or in whole, with the same instruments or with different instruments, at the same tempo or at a different or changing tempo, and can even be re-transcribed after such alterations.

To transcribe the first example above and convert it to music notation, we would do the following:

```
from scamp import *

s = Session()
violin = s.new_part("Violin")

# begin transcribing (defaults to using
# all instruments within the session)
s.start_transcribing()

for pitch in [60, 64, 67, 72]:
   violin.play_note(pitch, 1, 0.5)

# stop transcribing and save the
# note events as a performance
performance = s.stop_transcribing()

# quantize and convert the performance to
# a Score object and open it as a PDF
# (by default this is done via abjad)
performance.to_score().show()
```

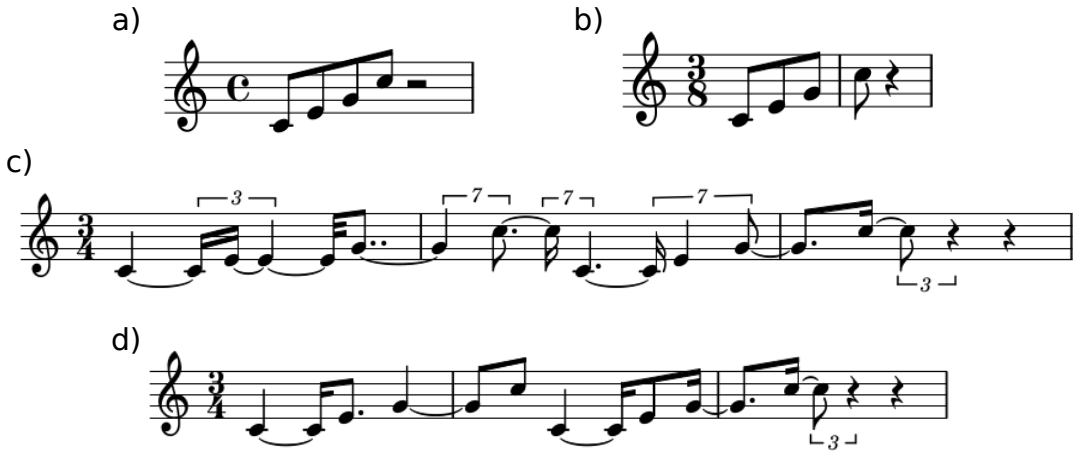

Figure 1.3: Examples of notation generated from introductory examples. Snippets 1) and 2) result from steady note lengths. Snippets 3) and 4) result from random note lengths, with 4) using a simpler QuantizationScheme.

This results in the notation shown in Fig. 1.3a. When `show` is called, the score representation within SCAMP is converted to an *abjad* score, which then outputs and compiles LilyPond code and displays the result as a PDF. It is also possible to instead call `show_xml`, which uses *pymusicxml* to export a MusicXML document and opens the result in a score editor (e.g. MuseScore, Sibelius, Finale).

By default, the `to_score` function quantizes the music to 4/4 time. If a different time signature is desired, one merely has to provide it as an argument:

```
performance.to_score("3/8").show()
```

This results in Fig. 1.3b. A looping or non-looping list of time signatures can also be provided, as can a list of beats on which to place bar lines.

1.2.4 Quantization

Things get more interesting when we use random floating-point durations, since some quantization is required:

```
# ---- setup code omitted for brevity ----
s.start_transcribing()

for _ in range(2): # loop twice
    for pitch in [60, 64, 67, 72]:
        dur = random.uniform(0.5, 1.5)
        violin.play_note(pitch, 1, dur)

performance = s.stop_transcribing()
performance.to_score("3/4").show()
```

This results in Fig. 1.3c. By default, divisors of the beat up to 8 are allowed, which may be more complex than desired. Simpler results can be achieved by using a custom `QuantizationScheme` that limits the max divisor to 4:

```
performance.to_score(
   QuantizationScheme.from_time_signature(
      "3/4", max_divisor=4
   )
).show()
```

This results in Fig. 1.3d. Considerable customizability of `QuantizationScheme` is possible: not only is it possible to set the max divisor, but it is also possible to specify the degree of preference for simpler ratios, or even to provide a list of all allowed divisors. Moreover, while quantization is always done at the beat level, the beat length and divisor preferences can vary from beat to beat, according to the scheme imposed of the user.

As illustrated in Fig. 1.4, the choice of quantization for each beat is made by comparing the (floating-point) note onset and termination times that fall within that beat to the grid formed by each divisor. The winning divisor is that which minimizes the (weighted) square error. To the degree that a preference for simple divisors is specified by the user, this square error is weighted by the *indigestibility* of the divisor (as described by Barlow

in [16]). The user can also specify a relative weighting of onset vs. termination error.

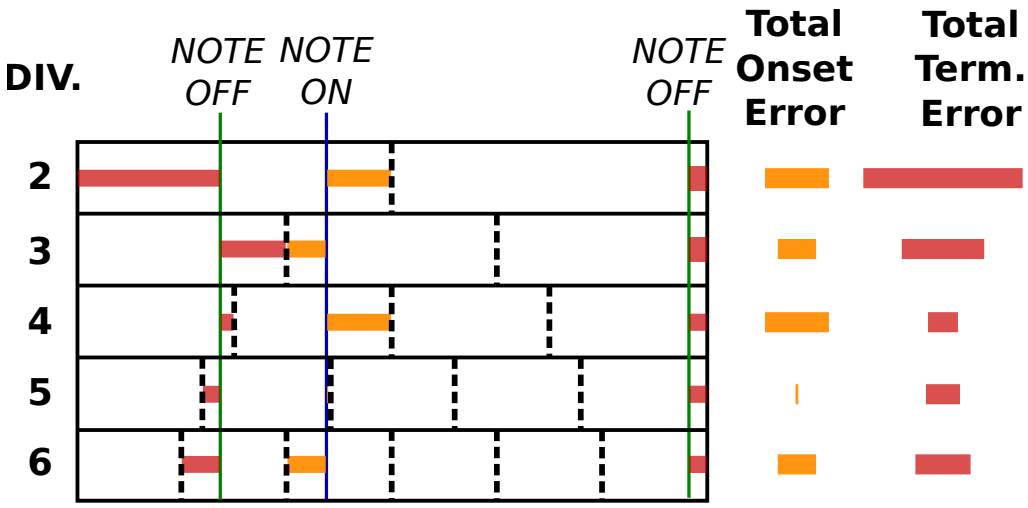

Figure 1.4: Illustration of the quantization process.

One weakness of this approach is that it does not allow nested tuplet structures. A possible future development, therefore, would be the incorporation of the Q-Grid approach proposed by Nauert [17], which allows for such structures, taking into account their degree of complexity.

1.3 Details and Further Possibilities

We now consider in more depth the range of possibilities for playback and notation that SCAMP offers.

1.3.1 Multi-Part Music

The above examples, for simplicity, were all single-part; however, writing multi-part music is straightforward:

```
from scamp import *
import random

# one way of setting an initial tempo
s = Session(tempo=100)

oboe = s.new_part("oboe")
bassoon = s.new_part("bassoon")

# define a function for the oboe part
def oboe_part():
   # play random notes until we have
   # passed beat 7 in the session
    while s.beats() < 7:
        pitch = int(random.uniform(67, 79))
        volume = random.uniform(0.5, 1)
        length = random.uniform(0.25, 1)
        oboe.play_note(pitch, volume, length)
    # end with a note of exactly the right
    # length to take us to the end of beat 8
    oboe.play_note(80, 1.0, 8-s.beats())


# define a function for the bassoon part
def bassoon_part():
   # simply play quarter notes on random
   # pitches for 8 beats
    while s.beats() < 8:
        bassoon.play_note(
            random.randint(52, 59), 1, 1
        )

s.start_transcribing()
# start the oboe and bassoon parts as
# two parallel child processes
s.fork(oboe_part)
s.fork(bassoon_part)
```

```
# have the session wait for the child
# processes to finish (return)
s.wait_for_children_to_finish()
performance = s.stop_transcribing()
# here we call show_xml, which will open
# the result in, for instance, MuseScore
performance.to_score().show_xml()
```

This results in the notation seen in Fig. 1.5. The `fork` method of the `Session` object, inherited from the *clockblocks* `Clock` class, takes a function as a parameter and runs it as a parallel child process. There is considerable flexibility here: child processes can spawn their own child processes, and so on. Unlike a naive approach to parallelism using Python's built-in `threading` module, all processes forked in this way will remain tightly coordinated, with the clock also compensating for calculation time.

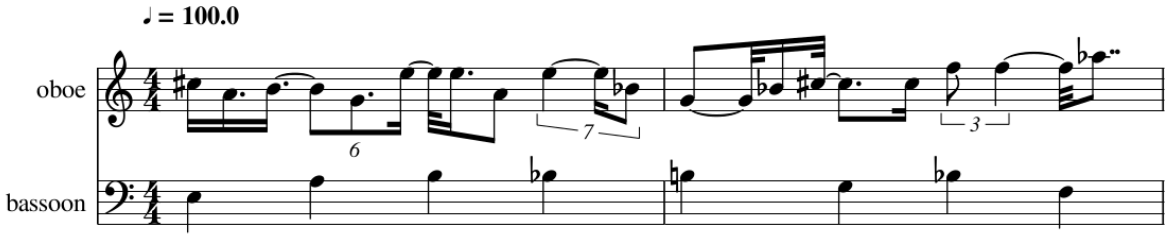

Figure 1.5: Example of notated multi-part music in SCAMP, having been exported as MusicXML and imported into MuseScore.

1.3.2 Multi-Tempo Music

One of the most powerful aspects of SCAMP's approach to time-management is that each forked processes is associated with its own `Clock`, and can have its own smoothly-varying tempo:

```
from scamp import *

s = Session()
trumpet = s.new_part("trumpet")
trombone = s.new_part("trombone")

# Have the session as a whole speed up to
# 100 BPM over the first nine beats
s.set_tempo_target(100, 9)

# here we define a trumpet part to run as
# a child process. The clock argument in
# the function definition gives us access
# to the clock this child process runs on
def trumpet_part(clock: Clock):
   # play eighth notes for three beats
    while s.beats() < 3:
        trumpet.play_note(67, 1, 0.5)
    # tell the clock for this child process
    # to slow down to 1/2 speed over six
    # beats in the parent process
    clock.set_rate_target(0.5, 6, duration_units="time")
    # keep playing eighth notes until 12
    # beats pass in the parent process
    while s.beats() < 12:
        trumpet.play_note(67, 1, 0.5)

# Fork the trumpet part as a child process
# (It will be influenced both by its own
# tempo and that of the session as a whole)
s.fork(trumpet_part)
s.start_transcribing()

# Play quarter notes for 12 beats
while s.beats() < 12:
    trombone.play_note(60, 1, 1)
```

```
# Stop transcribing and show the result
performance = s.stop_transcribing()
performance.to_score("3/4").show_xml()
```

This results in the notation shown in Fig. 1.6a. Within the context of a session-wide acceleration from 60 BPM to 100 BPM, the clock for the trumpet part slows down to half speed. (Note that, in SCAMP, "rate" and "tempo" are alternate names for the same underlying property, just with different units: a tempo of 60 BPM corresponds to a rate of 1, and a tempo of 120 BPM corresponds to a rate of 2, etc. Tempo units are valuable for their musical connotations, while rate units are generally more comprehensible in the context of nested clocks.)

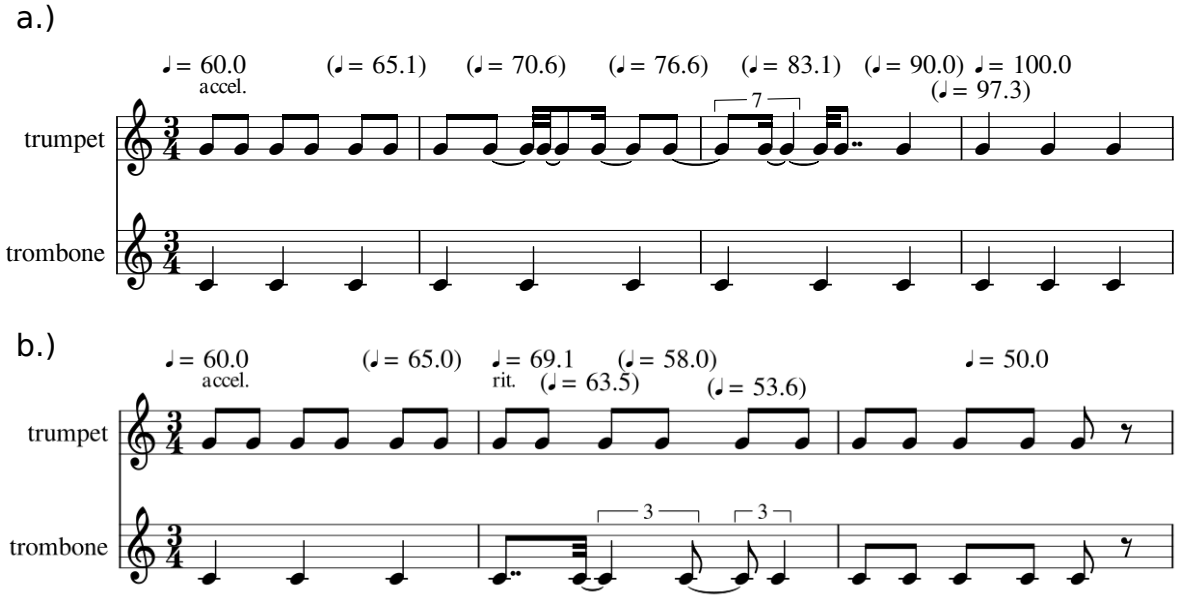

Figure 1.6: The same music quantized to two different clocks, having been exported as MusicXML and imported into MuseScore.

An important aspect of notating polytempo music in SCAMP is that performances can be transcribed in reference to whichever clock is desired. For instance, in the example above, we can generate a score relative to the tempo of the trumpet part by replacing the corresponding lines with:

```
# fork returns the Clock associated
# with the newly forked process
trumpet_clock = s.fork(trumpet_part)
s.start_transcribing(clock=trumpet_clock)
```

This results in the notation shown in Fig. 1.6b.

Further detail on polytempo usage of clocks can be found in [14].

1.3.3 Skipping Forward in Time

It may be that a composer, having devised an algorithmic process lasting 20 minutes, would like quickly test what this process sounds like at the 15-minute mark. In SCAMP, this is possible by simply calling a "fast-forward" method on the master clock (generally the `Session` object):

```
s.fast_forward_to_time(900)
```

Similar to the process of "nesting" described by Smoliar [15], this causes the program to execute as rapidly as possible up to the appointed time, while skipping note playback calls. In this way, the program is in exactly the same state as it would have been had it run normally.

1.3.4 Envelopes and Continuous Parameters

The accelerandi and decelerandi in section 1.3.2 are an example of the continuous shaping of a parameter over time, in this case tempo. As touched on in section 1.1.3, each `Clock` manages it tempo using a `TempoEnvelope`, a subclass of `Envelope`, the piecewise-exponential envelope class defined in the separate package *expenvelope*.

Instances of `Envelope` are used broadly in SCAMP when any musical parameter is

changing over time, such as in glissandi or dynamic playback. They can also be employed by the composer for the larger-scale shaping of algorithmic processes.

A simple, evenly-spaced `Envelope` can be constructed and plotted as follows:

```
e = Envelope.from_levels([60, 72, 66, 70])
e.show_plot("Evenly Spaced")
```

This results in the plot shown in Fig. 1.7 (plotting is done using *matplotlib*).

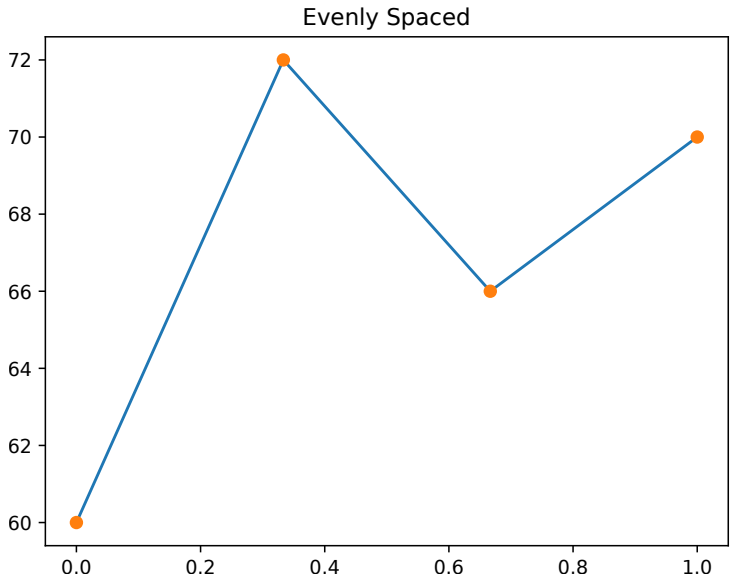

Figure 1.7: A simple `Envelope` with even spacing.

Each segment of the envelope can furthermore be assigned a duration and a shaping attribute:

```
e = Envelope.from_levels_and_durations(
   [60, 72, 66, 70], [3, 1, 1],
   curve_shapes=[2, -2, -2])
e.show_plot("Uneven with Shaping")
```

This results in the plot shown in Fig. 1.8. The values in the `curve_shapes` attribute range from $-\infty$ to ∞, with 0 being linear, negative values corresponding to early change,

positive values corresponding to late change.

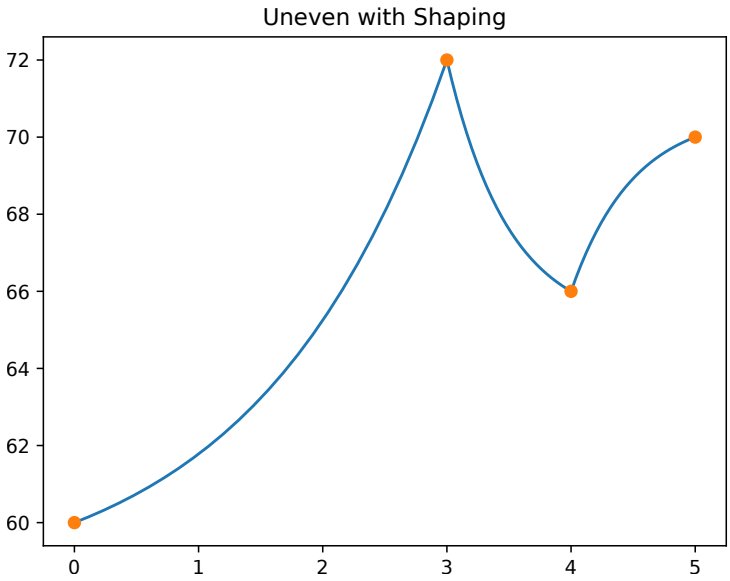

Figure 1.8: An `Envelope` with varied segment duration and shape.

The `Envelope` class is equipped with a range of utilities, including integration (needed by `TempoEnvelope` to determine the length in time corresponding to a certain number of beats at a changing tempo), approximation of arbitrary functions, and mathematical operations, such as addition, multiplication, division, and concatenation.

Glissandi

The `Envelope` in the example above can be interpreted as a glissando by simply passing it to the pitch argument of the `play_note` function:

```
instrument.play_note(e, 1.0, 4)
```

This results in the notation shown in Fig. 1.9. Note that the glissando is scaled to the length of the note, and that, by default, the pitch of the glissando is renotated on every beat and at every local maximum or minimum of the curve. In this case, since the segment durations are in the proportions 3:1:1, this results in quintuplets.

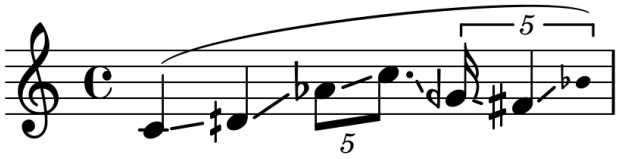

Figure 1.9: Glissando notation resulting from passing an `Envelope` as the pitch argument to `play_note`, as compiled by LilyPond, via *abjad*.

Since directly creating an `Envelope` object each time a glissando is desired is would be cumbersome, any list given as the pitch argument to `play_note` is interpreted as an envelope:

```
# results in evenly spaced glissando
instrument.play_note([60, 70, 55], 1.0, 4)
```

If segment durations and curve shapes are desired, a list consisting of [values, durations, curve shapes] may be used:

```
# results in segment durations of 2 and 1
# and curve shapes of -2 and 0
instrument.play_note([[60, 70, 55], [2, 1], [-2, 0]],
    1.0, 4)
```

Microtonality

The reader may have noticed quarter-tone accidentals in Fig. 1.9. Microtonality in SCAMP is as simple as using floating-point values for pitch. When using SoundFont-based playback via *pyFluidSynth* (which utilizes the MIDI protocol) or a MIDI stream to an external synthesizer, SCAMP internally handles all pitchbend messages. Since these messages are channel-wide, notes that change pitch (or might change pitch) are placed on separate channels, with channels being recycled automatically.

If more specificity of pitch is desired in the notation, the user simply has to turn on

microtonal pitch annotations in the `engraving_settings`:

```
engraving_settings.show_microtonal_annotations = True
piano.play_chord([62.7, 71.3], 1.0, 1)
piano.play_chord([65.2, 70.9], 1.0, 1)
piano.play_chord([71.5, 74.3], 1.0, 1)
```

This results in the notation shown in Fig. 1.10.

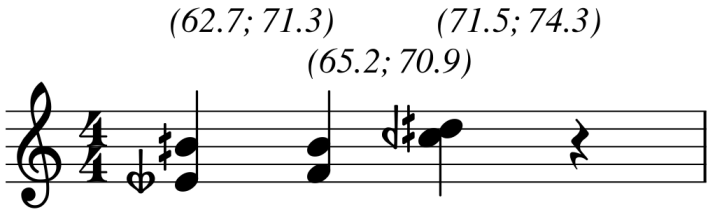

Figure 1.10: Microtonal pitch annotations.

Dynamics

An `Envelope` object (or its corresponding list shorthand) may also be used to apply a continuous volume curve to a note. For instance, a forte-piano-crescendo can be achieved in the following way:

```
fp_cresc = Envelope.from_levels_and_durations(
   [1.0, 0.3, 1.0], [0.1, 0.9]
)
piano.play_note(60, fp_cresc, 4)
```

Since dynamic notation is more subjective than pitch notation, arbitrary parameter curves like this are not currently notated. This is an area for future development.

Anatomy of a `play_note` call

As mentioned in Section 1.2.2, `play_note` internally breaks down into separate calls to `start_note`, `wait`, and `end`. With continuous changes of parameter, further calls are involved. For instance, consider the following:

```
# glissando and crescendo over 4 beats
instrument.play_note([60, 70, 55], [0.3, 1.0], 4)
```

This is equivalent to the following separate calls:

```
note = instrument.start_note(60, 0.3)
# start a 2 beat pitch-change ramp
note.change_pitch(70, 2)
# start a 4 beat volume-change ramp
note.change_volume(1.0, 4)
wait(2)
# start the second pitch ramp
note.change_pitch(55, 2)
wait(2)
note.end()
```

While a call to `play_note` is clearly more succinct, there are situations where the length and course of the note are not known in advance, and which are therefore better suited to the second approach. This is especially true with the incorporation of live input.

1.3.5 Playback Implementations

In the preceding examples, all playback is done via *pyfluidsynth* using a default General MIDI soundfont. However, one of the major flexibilities built into SCAMP is its variety of available playback implementations, as well as the ability to define custom implementations.

Consider the following setup:

```
s = Session()

# Calling "new_part" results in a default
# SoundfontPlaybackImplementation, and
# add_streaming_midi_playback gives this a
# MIDIStreamPlaybackImplementation as well
# (here, port 2 is used for output)
piano =
    s.new_part("piano").add_streaming_midi_playback(2)

# "new_osc_part" initializes the instrument
#  with an OSCPlaybackImplementation
synth = s.new_osc_part("synth", ip_address="127.0.0.1",
                       port=57120)

# Calling "new_silent_part" results in an
# instrument with no PlaybackImplementation
silent = s.new_silent_part("silent")
```

As illustrated by Fig. 1.11, session `s` will be set up with contain an ensemble of three `ScampInstruments` named "piano", "synth", and "silent". The "piano" instrument is created with the default `SoundfontPlaybackImplementation` to which we add a `MIDIStreamPlaybackImplementation`. By calling `new_osc_part`, the "synth" instrument is created with an `OSCPlaybackImplementation` instead of the default. Finally, the "silent" instrument is created with no `PlaybackImplementation` at all.

When a note is played on an instrument, each of its `PlaybackImplementations` is told to start a note, do any pitch / volume / parameter changes, and then end the note at the appropriate time. Thus, when a note is played using "piano" above, MIDI messages will be sent to *fluidsynth* to generate sound, as well as sent out of port 2. When a note is played using "synth", appropriate OSC messages (with address patterns like "synth/start_note"

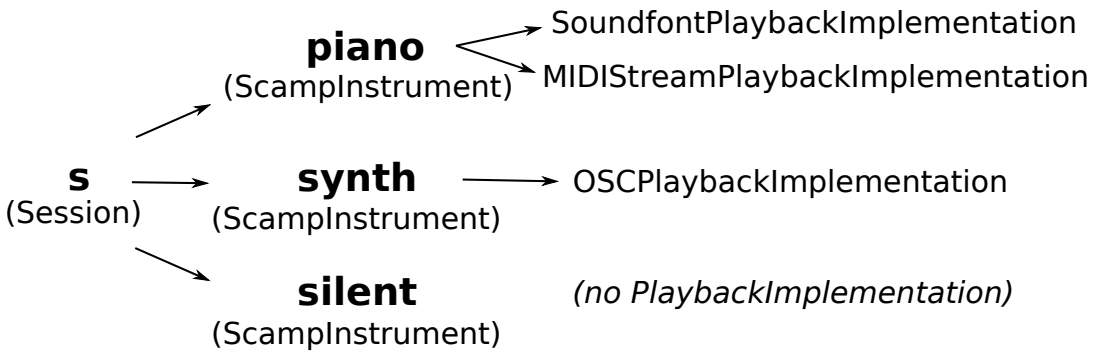

Figure 1.11: Illustration of playback implementations.

and "synth/change_pitch") will be sent to port 57120. Finally, when a note is played using "silent", nothing will happen, since it has no `PlaybackImplementations`. (Silent instruments can nevertheless be useful: for instance, as a reference, one might want to notate the pitch collection being used by an algorithmic process without generating sound. Or one could use a silent instrument to aggregate the notations of several sounding instruments using different SoundFont presets, which might come in handy for, say, a violin part using multiple bow techniques as well as pizzicato.)

1.3.6 Additional Note Properties

The `play_note` function in SCAMP can take a fourth, optional properties argument. This argument acts as a wildcard, accepting a properly formatted string or a dictionary specifying a variety of playback and notation details:

```
# passing comma-separated key value pairs
piano.play_note(60, 1, 1,
   "notehead: x, articulation: staccato")
# just a value; articulation type inferred
piano.play_note(60, 1, 1, "staccato")
# "play_chord" can take multiple noteheads
piano.play_chord([60, 65], 1, 1, "noteheads: x/circle-x")
# passing a dictionary also possible
```

```
piano.play_note(60, 1, 1, {"articulations": ["tenuto",
    "accent"]})
```

This produces the notation shown in Fig. 1.12. In addition to modifying the notation, articulations like staccato and tenuto will affect playback duration, and accents will affect volume. Exactly how these notations affect playback is an adjustable (and disableable) setting with sensible defaults.

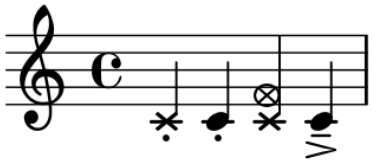

Figure 1.12: Illustration of notation resulting from various properties

The properties argument may also be used to determine note spelling, either directly or by suggesting a key:

```
# spell the note with a sharp
piano.play_note(63, 1.0, 1.0, "#")

# play some notes in F minor
pitches = [65, 63, 61, 60]
durations = [1/3] * 3 + [1]
for pitch, dur in note_info:
   piano.play_note(pitch, 1.0, dur, "key: F minor")
```

This results in the notation shown in Fig. 1.13.

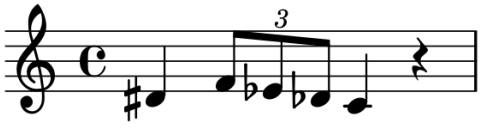

Figure 1.13: Notes spelled both directly and by specifying a key context

The properties argument can also be used to define aspects of playback other than

pitch and volume, if implemented by the `PlaybackImplementation`. For instance, with a properly configured OSC receiver, the following will produce a 3 beat glissando with increasing vibrato frequency and depth:

```
# on the other end, an OSC receiver is
# setup to play notes that take vibrato
# OSC messages as well as the usual
vib = s.new_osc_part("vibrato_inst", 57120)

# any property entries starting or ending
# with "param" will be treated as extra
# playback parameters
vib.play_note(
    [72, 64], 0.5, 3,
   {"vib_depth_param": [0.5, 2],
    "vib_freq_param": [3, 8]}
)
```

Any keys in the properties argument starting with the prefix `"param_"` or ending with the suffix `"_param"` are interpreted as additional playback parameters. In the case of an `OSCPlaybackImplementation` this results in outgoing OSC messages with address patterns "vibrato_inst/change_parameter/vib_depth" and "vibrato_inst/change_parameter/vib_freq". As with dynamics playback, translating this to some form of notational output is an area for future development.

1.4 Conclusions

1.4.1 Directions for Further Development

As mentioned above, some form of notation for dynamics and other user-defined playback parameters is needed. For dynamics, a distinction will need to be made between

note-level dynamics (such as ***sfz*** or ***fp***) and phrase-level dynamic spans (such as *cresc.* or *dim.*). The easiest approach will likely be to allow specific dynamic notations such as these to be defined and applied to notes or phrases, influencing both notation and playback, as opposed to trying to infer notation from playback (which is the general approach taken in SCAMP). For other parameters, some form of graphical notation incorporating the shape of the envelope curve may be appropriate. In general, since the `Performance` class retains continuous-time musical data, alternate mappings to different kinds of score are possible.

This points to the value of expanding the options for translating musical data. Currently, a `Performance` can be converted to a `Score`, but not vice-versa. Likewise, a `Score` can be translated to an *abjad*/LilyPond or MusicXML representation, but not vice-versa. One development goal is to enable two-way translation between all of these representations. Also, although SoundFont playback and streaming MIDI output are possible, it is not currently possible to convert a `Performance` to a MIDI file, or to directly save playback to a sound file. These are planned improvements.

In terms of playback, a medium-term goal is to offer SFZ-file-based playback using LinuxSampler [18]. As a modern, open standard for sampled sound playback, SFZ files offer exactly the kind of flexibility that SCAMP is designed to afford [19]. Another playback goal is to enable certain note properties, such as "pizzicato", to trigger a preset switch, much as they do in professional score-writing software, such as Sibelius. Finally, *MAX/MSP* and *SuperCollider* abstractions are currently being developed for receiving OSC output from SCAMP.

One last area of development is in the domain of live input. Although SCAMP is not designed for live coding, the ability to receive real-time MIDI and OSC input, and to thereby shape the result of an algorithmic process, would be of great value to many composers' workflow.

1.4.2 Evaluation

Although all of the planned developments above are worthy goals, the most important next step is to create music with SCAMP and to cultivate a broad and varied user-base. This will allow any future development to be guided by the true obstacles that composers face while using this framework.

To this end, I plan to lead workshops and classes devoted to exploring computer-assisted composition using SCAMP, as well as to develop detailed online documentation and tutorials. SCAMP was developed and re-designed over a period of many years, based on my own compositional practice. Its true potential, however, can only be discovered through interaction with other composers, approaching it with potentially radically different compositional aims.

Paper 2

Clockblocks: A Pure-Python Library for Controlling Musical Time

This paper describes *c*lockblocks, 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.

2.1 Introduction

2.1.1 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* [3] or *jMusic* [20]. 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) [21] 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.

2.1.2 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 `time.sleep` function from the Python Standard Library 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 are coordinated through with a single master clock, which does the actual sleep calls. In this way, multiple

independent streams of musical time, potentially running simultaneously at different tempi, remain perfectly synchronized.

2.2 A Simple Example

2.2.1 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:

$$R = 1/L_b \tag{2.1}$$

$$T = 60 \cdot R = 60/L_b \tag{2.2}$$

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.

2.2.2 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* [22]. In practice, this means that any accelerandi or ritardandi can be given a non-linear shape, increasing the expressive potential of clockblocks.

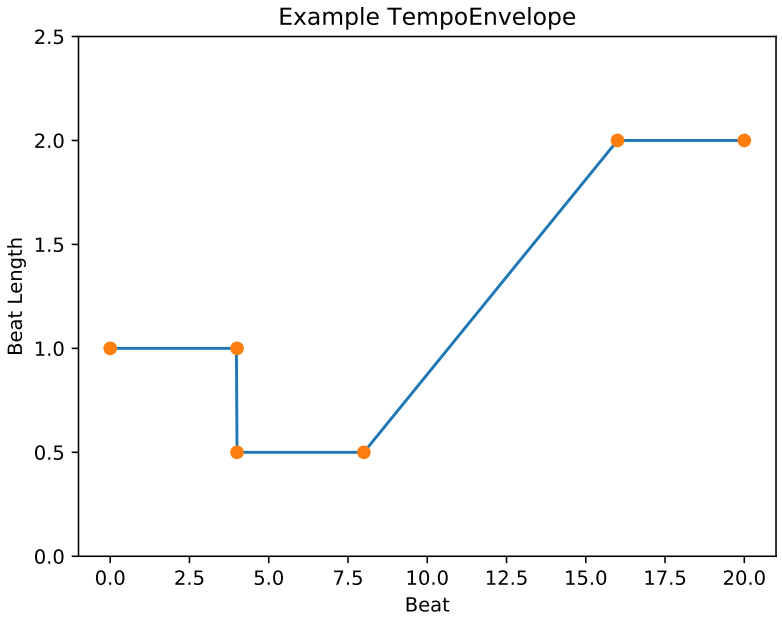

Figure 2.1: Graph of the clock's TempoEnvelope from the example above

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 2.1 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.

2.3 Parallelism

2.3.1 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.

2.3.2 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. 2.3.1, 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. 2.1.2). All other clocks simply register a wake up call with their parent when they wish to sleep.

2.4 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:

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.

2.5 Nested Clocks

2.5.1 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 2.2, 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.

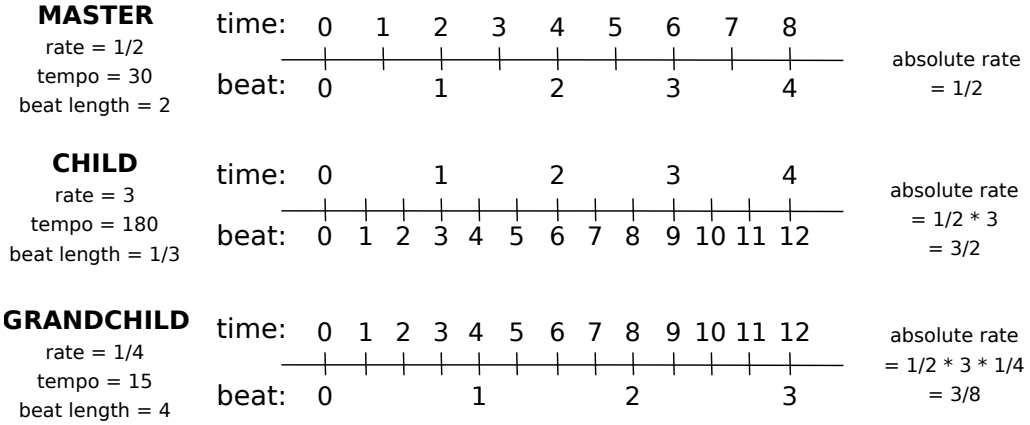

Figure 2.2: Relationship between the rates of three "generations" of clocks running at different tempi

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.

2.5.2 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")

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.

After running the code above, the following lines will generate the plots shown in Figure 2.3:

```
# show_plot uses matplotlib under the hood
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()
```

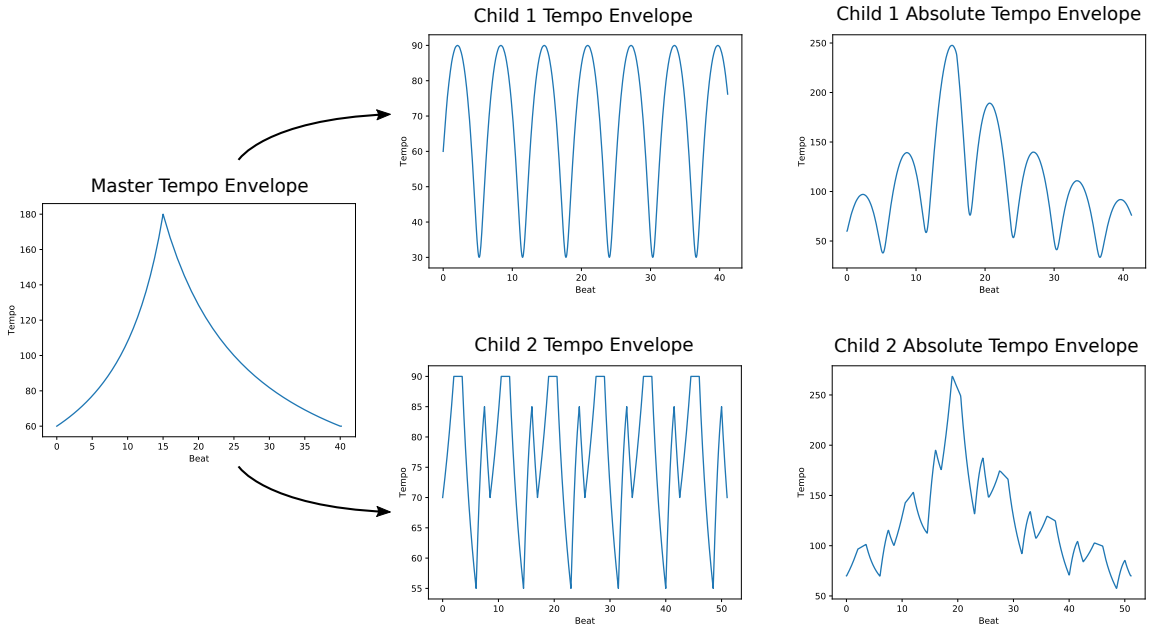

Figure 2.3: Effect of a master tempo envelope of the absolute tempo envelopes of its children. Note that, unlike in Figure 1, these graphs are of tempo, rather than the beat length.

As Figure 2.3 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.

2.6 Comparison with other approaches

In contrast to audio programming environments like *Gibber* [23] and *ChucK* [24], *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 [2] 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.

2.7 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" [25], developed by Matthew Wright in collaboration with composer Edmund Campion, as well as from CNMAT's subsequently developed Max / MSP external, "Timewarp" [26].

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,

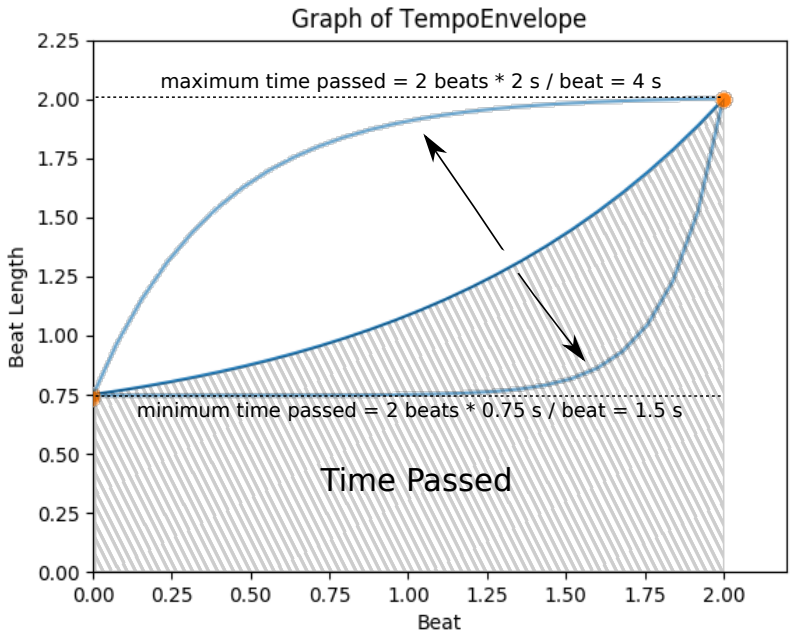

Figure 2.4: Illustration of the relationship between curvature and the length (in time) of an accelerando.

As Figure 2.4 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.

2.8 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 creates the potential 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.

Bibliography

[1] M. Puckette, *Pure data: another integrated computer music environment*, in *in Proceedings, International Computer Music Conference*, pp. 37–41, 1996.

[2] J. McCartney, *Rethinking the Computer Music Language: SuperCollider*, *Computer Music Journal* **26** (Dec., 2002) 61–68.

[3] 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.

[4] J. Bresson, C. Agon, and G. Assayag, *OpenMusic: visual programming environment for music composition, analysis and research*, in *Proceedings of the 19th ACM international conference on Multimedia - MM '11*, (Scottsdale, Arizona, USA), p. 743, ACM Press, 2011.

[5] M. S. Cuthbert and C. Ariza, *music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data*, in *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, International Society for Music Information Retrieval, Aug., 2010.

[6] M. Good, *MusicXML: An Internet-Friendly Format for Sheet Music*, in *Proceedings of XML*, (Boston, Massachusetts, USA), 2001.

[7] H.-W. Nienhuys and J. Nieuwenhuizen, *LilyPond, a System for Automated Music Engraving*, *Proceedings of the XIV Colloquium on Musical Informatics* (2003).

[8] D. Rossum, "The SoundFont® 2.0 File Format."

[9] N. Whitehead, "pyfluidsynth." `https://github.com/nwhitehead/pyfluidsynth`, 2018.

[10] A. Freed and A. Schmeder, *Features and Future of Open Sound Control version 1.1*, in *Proceedings of the 2009 New Interfaces for Musical Expression (NIME) Conference*, 2009.

[11] O. Jolly, "sf2utils." `https://gitlab.com/zeograd/sf2utils`, 2018.

[12] C. Arndt, "python-rtmidi." `https://github.com/SpotlightKid/python-rtmidi`, 2019.

[13] attwad (username), "python-osc." `https://github.com/attwad/python-osc`, 2018.

[14] M. Evanstein, *Clockblocks: A Pure-Python Library for Controlling Musical Time*, in *Proceedings of the 2019 International Computer Music Conference*, (New York, New York, USA), 2019.

[15] S. W. Smoliar, *A Parallel Processing Model of Musical Structures*, .

[16] C. Barlow, *On Musiquantics: Von der Musiquantenlehre translated*. Royal Conservatory The Hague, 2012.

[17] P. Nauert, *A Theory of Complexity to Constrain the Approximation of Arbitrary Sequences of Timepoints*, *Perspectives of New Music* **32** (1994), no. 2 226–263.

[18] "The Linux Sampler Project."

[19] "The SFZ Format."

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

[21] M. Evanstein, "Scamp: a suite for computer-assisted music in python." `https://github.com/MarcTheSpark/scamp`, 2018.

[22] SuperCollider 3 Documentation Contributors, "Env — supercollider 3.10.0 help." `http://doc.sccode.org/Classes/Env.html`, 2018.

[23] 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), pp. 67–76, ACM Press, 2014.

[24] G. Wang, P. R. Cook, and S. Salazar, *ChucK: A Strongly Timed Computer Music Language*, *Computer Music Journal* **39** (Dec., 2015) 10–29.

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

[26] 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.