Abstract Time Warping of Compound Events and Signals[1]

Roger B. Dannenberg
Carnegie Mellon University
Pittsburgh, PA 15213 USA
dannenberg@cs.cmu.edu

ABSTRACT: Functions of time are often used to represent continuous parameters and the passage of musical time (tempo). A new approach generalizes previous work in three ways. First, common temporal operations of stretching and shifting are special cases of a new general time-warping operation. Second, these operations are ‘‘abstract.’’ Instead of operating directly on signals or events, they operate on abstract behaviors that interpret the operations at an appropriate structural level. Third, time warping can be applied to both discrete events and continuous signals.

1. Introduction

Expressive control is a central problem in the field of computer music. The idea of using functions of time for control is an old one. It is not surprising that functions have also been proposed as a mapping/warping from beats (score time) to real time. Formally, this mapping is the integral of 1/tempo as a function of beat position or score time. [Rogers 80] Researchers pursuing discrete structural representations for music invented ‘‘shift’’ and ‘‘stretch’’ operations that can be neatly composed. [Buxton 85, Spiegel 81, Orlarey 86] Originally applied to discrete events (notes), these operations are also a natural way to manipulate continuous time functions such as amplitude or pitch envelopes, and in fact these manipulations are implicit (though not composable) in Music *N* [Mathews 69] languages. Still more recently, abstraction has entered the picture, and researchers are concerned with the details of, for example, whether a ‘‘stretched’’ vibrato slows down or not. [Rodet 83, Dannenberg 86, Desain 92] In languages like Nyquist [Dannenberg 92], where signals are generated, this issue is especially important. Without abstraction, a tempo change would affect the rate of audio oscillators and cause a pitch change! How can we avoid stretched attacks, unsteady vibrato, and erratic drum rolls under the influence of time warps?

Previous work (such as [Jaffe 85]) does not address abstraction issues that arise with time warping. Conversely, previous work that addresses abstraction issues does not support time warping. This new work provides a more general solution in which time warping and other time functions can be applied to abstract behaviors.

2. Shift, Stretch, and Warp

Let us begin by defining two operators, $shift(d)$ and $stretch(s)$, which operate on time points. Note that $shift(d)$ is a function, so $shift(d)(t)$ is a function applied to a time point. We define these operators as follows:

$$shift(d)(t) = d+t$$
$$stretch(s)(t) = st$$

The shift operator corresponds to musical operations of delay, rest, or pause, and the stretch operator corresponds to augmentation, diminution, or tempo. Starting with a score where time is indicated in arbitrary units, *shift* and *stretch* operators can be used to construct a mapping from score time to real time. For example, to perform a score at half speed and shifted by 10 seconds, in Arctic one would write:

score ~ 2 @ 10.

In Nyquist, one would write:

```
(at 10 (stretch 2 (score))).
```

Similar operations are available in many other notations.

The meaning (semantics) of nested operators can be expressed mathematically using function composition:

$$\begin{aligned}(shift(10) \circ stretch(2))(t) \\ &= shift(10)(stretch(2)(t)) \\ &= shift(10)(2t) \\ &= 10 + 2t\end{aligned}$$

The ‘‘shift’’ and ‘‘stretch’’ operators are just special cases of what Jaffe terms *time maps* [Jaffe 85], Anderson and Kuivila term *time deformations* [Anderson 86], and I will call *time*

[1]Published as: Dannenberg, ‘‘Abstract Time Warping of Compound Events and Signals,’’ in ***Proceedings of the 1994 International Computer Music Conference***, International Computer Music Association, (September 1994), pp. 251-254.

warps [Dannenberg 89]. Time warps can be arbitrarily nested, and the effect of nested warp functions is that of function composition.

3. Continuous Functions

If the time warp is to be applied to discrete events such as note-on or note-off, the score time of the event is passed through the warp function to yield the real-time of the event. On the other hand, it is also important to support continuous time functions such as pitch bend, amplitude, articulation, and other continuous parameters. Normally, these functions are expressed in terms of score time.

To warp a function of score time, g, by a warp function, f, we compose g with the inverse of f to obtain a function of real time: $g(f^{-1}(t))$. (Remember that time warp f maps score time to real time, so its inverse f^{-1} maps real time to score time, and g is a function of score time. Therefore, we apply f^{-1} to real time t to get score time, then evaluate g at that point, yielding the value of g at a given real time.)

To summarize the results so far, we have seen that the ‘‘shift’’ and ‘‘stretch’’ operators seen in many music representations and languages are a special case of general time-warping functions. The literature tells us that time-warping functions can be nested using function composition, so this gives us a general way to handle nested ‘‘shift’’ and ‘‘stretch’’ in conjunction with other ‘‘warp’’ operators. Finally, we see that continuous functions can be warped using function composition and function inverse.

4. Abstraction Issues

Once time warps are introduced, however, we must revisit the ‘‘vibrato problem’’ [Desain 92]: what is the interpretation of warped vibrato? Should the vibrato rate fluctuate when time is warped? Previous work has generally ignored this problem. One solution would be to ‘‘build in’’ methods for handling warps, so that vibrato would get one treatment, envelopes would get another, and so on. This is convenient if and when the default behavior matches the composer’s intentions, but it is usually better to allow the composer to retain complete control over how warping is applied.

Our solution is to extend the mechanisms introduced in Arctic [Dannenberg 86] and available in the author’s current language project Nyquist. These mechanisms are quite general and can be used in many representation and language systems.

In Nyquist, it is not a transformation operation that ‘‘knows’’ how to transform the result of a behavior. Instead, it is the responsibility of the behavior itself to perform the transformation. We call this idea *behavioral abstraction*. This is abstraction in the sense that the behavior ‘‘packages’’ or hides details about how transformations are achieved. Secondly, our behaviors are abstractions in that they represent an infinite class of actual behaviors that vary according to pitch, time, duration, loudness, and so on.

How can a programming language support behavioral abstraction? In Nyquist, sounds are created and manipulated by combining built-in signal-processing primitives (essentially unit generators). The Lisp function definition facility is used to create new behaviors implementing notes, phrases and entire compositions. An abstract behavior is simply a Lisp function that computes and returns a sound. An instance of a behavior is obtained by evaluating (applying) the function.

To support transformation, we need a way to communicate transformation parameters to behaviors. One possibility is to pass every transformation parameter as a function parameter to every behavior. This would result in very long parameter lists and a very clumsy notation system.

Instead, transformation parameters are contained in an *environment* that is implicitly passed to every behavior. Environments are dynamically scoped, meaning that a nested function (the callee) inherits the environment from its (calling) parent. Special forms are used to modify the environment, for example, in

```
(transpose 2 (seq (a) (b) (c))),
```

the environment passed to `seq` will have its transposition attribute incremented by 2. Also, the `seq` operator modifies the time map seen by `b` so that `b` is shifted to the stop time of `a`, and so on.

It is important to note that transformations like transposition and time warps modify the environment *before* evaluating the enclosed Lisp functions which implement behaviors. It is critical that transformations operate in this manner. This gives behavioral abstractions a chance to determine how transformations will be implemented. For example, a fixed-length attack followed by a stretchable decay could be implemented as follows:

```
(defun env ()
  (seq (stretch-abs 1.0 (attack))
       (decay)))
```

The `stretch-abs` transformation replaces the stretch factor in the environment with 1.0 so that `attack` is not stretched. The `decay` sub-behavior inherits the environment from `env` and stretches accordingly. The environment is dynamically scoped. The reader is referred to [Dannenberg 89] for more examples.

Most other systems have avoided the behavioral abstraction issue by restricting the results of functions to note lists and by restricting transformations to operations on note attributes. Desain and Honing [Desain 92] solve the problem by returning functions of multiple parameters. As in Nyquist, transformations modify parameters rather than the actual behaviors, and the ‘‘how to transform’’ knowledge is encapsulated in Lisp functions rather than transformation operators.

5. Time Warping and Continuous Transformations

If the environment provides a time-warping function *f*, any discrete time point *t* affected by *f* can be mapped to real time simply by computing $f(t)$. For example, the Nyquist `pwl` function computes a piece-wise linear function from a list of breakpoints. When `pwl` is warped, the default is to map each breakpoint into real time.

The Nyquist `sine` function generates a sinusoid at a given pitch and duration. Warping a sinusoidal signal would distort the frequency, so in Nyquist the start time and end time of the `sine` behavior are warped, keeping a constant frequency in real time. To compute `sine`, a start time and end time are required, and these are given by $f(0)$ and $f(d)$, where *f* is the time warp function and *d* is the specified (unwarped) duration of the `sine`.

If we really want the effect of warping the `sine` (as in $g(f^{-1}(t)) = (g \circ f)(t)$), we could write the expression:

```
(s-compose
   (warp-abs nil (sine  p))
   (s-inverse *warp*)).
```

Here, `s-compose` denotes function composition, `s-inverse` denotes function inverse, and `*warp*` is a special variable reflecting the time-warp function in the environment. (Note: ‘‘function’’ here means function of time in the form of a Nyquist Sound data type. Do not confuse this with Lisp functions.) The `(warp-abs nil ...)` construct removes the time-warp function from the environment seen by `sine` so that the sinusoid is only warped once. One could even use this mechanism for FM synthesis, although the built-in Nyquist FM oscillators are more efficient.

Time-warp transformations can be composed with other continuous transformations. Consider this example:

```
(warp (f) (loud (contour)
                (behavior))).
```

Roughly, this expression says: compute `behavior` with a time-varying loudness given by `contour`, with everything warped according to `f`. In keeping with the behavioral abstraction concept, `(contour)` is computed within the environment warped by `f`. The environment, modified by this warped loudness contour, is then used for the evaluation of `(behavior)`.

Within `behavior`, it may be necessary to access current values of transformation functions such as produced by `(contour)`. This function will be bound to the special variable `*loud*`, but `*loud*` is a function of ‘‘post-warp’’ real time, whereas `behavior` will generally be written in terms of ‘‘pre-warp’’ score time. Let *f* be the time warp function and *g* be some transformation function. To get the current value of *g*, we map the logical ‘‘now’’ (0) into real time and then evaluate *g* at that point: $g(f(0))$. In Nyquist, this is written:

```
(sref *loud* (sref *warp* 0))
```
or simply `(get-loud)`,

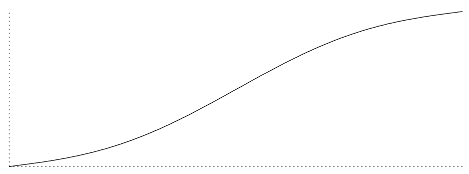
where `sref` evaluates a time function at a particular time. Finally, we note that an integration operator is useful for converting tempo function to a warp function. Figure 1 illustrates a variety of ways a behavior can respond to time warping.

6. Summary and Conclusions

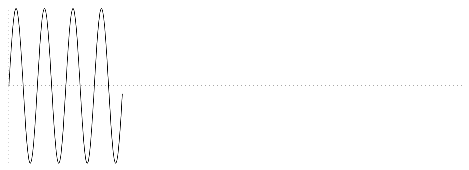
We have considered several aspects of Nyquist and their interaction. We discussed the idea of behavioral abstraction, whereby a single abstract behavior can be instantiated in different contexts to instantiate any number of concrete, or actual behaviors. An important principle is that ‘‘transformations’’ on behaviors, such as stretching and shifting, are interpreted in an abstract way. The details of ‘‘how to stretch’’ are encapsulated within the abstraction. Time mapping or warping functions were presented along with a semantics that are consistent with abstraction principles. We showed how *shift* and *stretch* operators are just special cases of general time warping. Finally, we showed that continuous control parameters can be integrated with behavioral abstraction and time mapping. Transformation functions are ordinary signals in Nyquist, so the full range of signal processing and behavioral abstraction facilities can be applied to transformation functions.

An efficient implementation is possible in Nyquist because `s-compose`, `s-inverse` and `s-integrate` all have fast, incremental implementations. Furthermore, time-maps and continuous transformations can be computed at low sample rates. Implementation details are the subject of a forthcoming paper, and the full Nyquist implementation itself is available from the author.

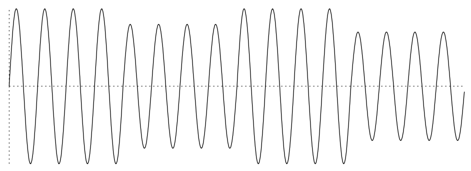
New compositional techniques are facilitated by the unification of audio signals, control signals, time warping, and transformations. For example, tempo changes can be smoothed with low-pass filters, the rate of a drum roll can track a pitch contour analyzed from speech input, and doppler shift and phasing effects can be achieved using arbitrary, even audio-rate synthesized control functions. The encapsulation of behavioral details offers the composer a powerful new tool for expressive control. We look forward to the exploration of these possibilities.



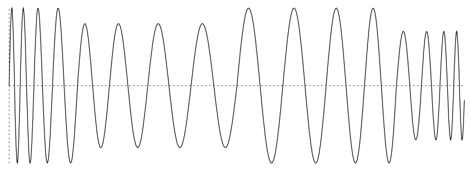
A time warp function.



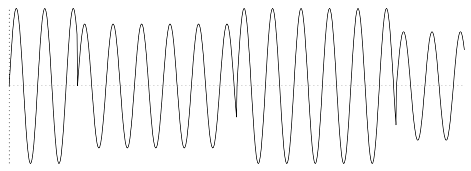
A ‘‘wiggle’’: 4 oscillations, not warped.



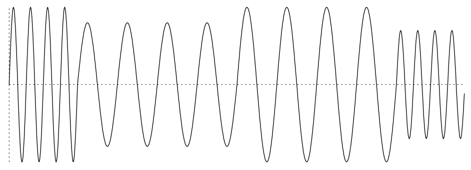
An unwarped compound behavior: 4 ‘‘wiggles’’ in sequence, with varying amplitudes.



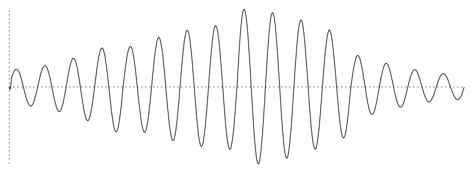
The compound behavior warped.



Only start and end points are warped; oscillation frequency is fixed.



Start and end points are warped, wiggles are linearly stretched to fit.



Amplitude inversely proportional to ‘‘tempo.’’

Figure 1: Time warp variations.

References

[Anderson 86] Anderson, D. P. and R. Kuivila. Accurately Timed Generation of Discrete Musical Events. *Computer Music Journal* 10(3):48-56, Fall, 1986.

[Buxton 85] Buxton, W., W. Reeves, R. Baecker, and L. Mezei. The Use of Hierarchy and Instance in a Data Structure for Computer Music. *Foundations of Computer Music.* In C. Roads and J. Strawn, MIT Press, 1985, pages 443-466.

[Dannenberg 86] Dannenberg, R. B., P. McAvinney, and D. Rubine. Arctic: A Functional Language for Real-Time Systems. *Computer Music Journal* 10(4):67-78, Winter, 1986.

[Dannenberg 89] Dannenberg, R. B. The Canon Score Language. *Computer Music Journal* 13(1):47-56, Spring, 1989.

[Dannenberg 92] Dannenberg, R. B. Real-Time Software Synthesis on Superscalar Architectures. In *Proceedings of the 1992 ICMC,* pages 174-177. International Computer Music Association, San Francisco, 1992.

[Desain 92] Desain, P. and H. Honing. Time Functions Function Best as Functions of Multiple Times. *Computer Music Journal* 16(2):17-34, Summer, 1992.

[Jaffe 85] Jaffe, David. Ensemble Timing in Computer Music. *Computer Music Journal* 9(4):38-48, 1985.

[Mathews 69] Mathews, M. V. *The Technology of Computer Music.* MIT Press, Boston, 1969.

[Orlarey 86] Orlarey, Y. MLOGO: A MIDI Composing Environment. In P. Berg (editor), *Proceedings of the International Computer Music Conference 1986,* pages 211-213. International Computer Music Association, 1986.

[Rodet 83] Rodet, Xavier, Pierre Cointe, Jean-Baptiste Barriere, Yves Potard, B. Serpette, and J. J. Briot. Applications and Developments of the FORMES programming environment. In *Proceedings of the 1983 International Computer Music Conference*. International Computer Music Association, 1983.

[Rogers 80] Rogers, J., J. Rockstroh, and P. Batstone. Music-Time and Clock-Time Similarities Under Tempo Changes. In *Proceedings of the 1980 International Computer Music Conference*, pages 404-442. International Computer Music Association, 1980.

[Spiegel 81] Spiegel, L. Manipulations of Musical Patterns. In *Proceedings of the Symposium on Small Computers in the Arts*, pages 19-22. IEEE Computer Society, 1981.