

Abstract Time Warping of Compound Events and Signals

Author(s): Roger B. Dannenberg

Source: *Computer Music Journal*, Autumn, 1997, Vol. 21, No. 3 (Autumn, 1997), pp. 61-70

Published by: The MIT Press

Stable URL: <https://www.jstor.org/stable/3681014>

JSTOR is a not-for-profit service that helps scholars, researchers, and students discover, use, and build upon a wide range of content in a trusted digital archive. We use information technology and tools to increase productivity and facilitate new forms of scholarship. For more information about JSTOR, please contact support@jstor.org.

Your use of the JSTOR archive indicates your acceptance of the Terms & Conditions of Use, available at <https://about.jstor.org/terms>



The MIT Press is collaborating with JSTOR to digitize, preserve and extend access to *Computer Music Journal*

JSTOR

Abstract Time Warping of Compound Events and Signals

Functions of time are often used to represent continuous parameters and the passage of musical time or tempo. The work described in this article generalizes previous work in three ways. First, common temporal operations of stretching and shifting are shown to be special cases of a new general time-warping operation. Second, we show a language in which these operations are “abstract.” Instead of operating directly on signals or events, time warps operate on abstract behaviors that interpret warping at an appropriate structural level. Third, we show how time warping can be applied to both discrete events and continuous signals. These new generalizations are implemented in Nyquist, and we describe the implementation.

This is the second article of four (starting with Dannenberg 1997) on Nyquist, a high-level language for sound synthesis and composition. Nyquist is used as a basis for discussion, both to ground this work in a concrete realization and to further describe the features of Nyquist. However, the general principles presented here should apply to many composition, control, and synthesis systems.

Background

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; for example, computer music systems often describe sounds in terms of pitch contours, amplitude envelopes, and other controls, all of which are functions of time. It is not surprising that functions have also been proposed as a mapping/warping from beats (“score time”) to real time. This mapping can be used to specify continuous tempo change, rhythmic time

alterations such as “swing,” and phrasing techniques such as *rubato*. Formally, the mapping from beats to time is the integral of $1/\text{tempo}$ as a function of beat position (Rogers, Rockstroh, and Batstone 1980).

Timing manipulations have also been expressed in notations intended for discrete event data, such as notes. Researchers pursuing discrete structural representations for music invented “shift” and “stretch” operations that can be neatly composed (Spiegel 1981; Buxton et al. 1985; Orlarey 1986). 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 languages (Mathews 1969). Every Music N “note” statement specifies a starting time and a duration. These effectively shift and stretch the envelopes and control signals that pertain to the note.

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 (Cointe and Rodet 1984; Dannenberg, McAvinney, and Rubine 1986; Desain and Honing 1992, 1993). In languages such as Nyquist, 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?

Early work (such as Jaffe 1985) offers a clear model for computing, combining, and applying time-warp functions to musical data. However, work previous to Nyquist does not address abstraction issues that arise with time warping. On the other hand, previous work that does address abstraction issues (Dannenberg, McAvinney, and Rubine 1986; Dannenberg 1989; Desain and Honing 1992) does not support continuous time warping of continuous signals. This new work provides

a more general solution in which time warping and other time functions can be applied to abstract behaviors that operate all the way down to the signal level.

In the following sections, we will first describe time warping as a generalization of shift and stretch operations. Next, warping is extended from discrete event timing to continuous functions. This raises abstraction issues that are addressed in the following section. Next, the application of warping to continuous transformations is discussed. An implementation in Nyquist is described, and finally, this work is compared to Generalized Time Functions, or GTF (Desain and Honing 1993).

Shift, Stretch, and Warp

Let us begin by developing a formal model of the seemingly fundamental operations for “shifting” and “stretching.” We define 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:

$$\text{shift}(d)(t) = d + t$$

$$\text{stretch}(s)(t) = s \cdot t$$

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 sec, in Arctic (Dannenberg, McAvinney, and Rubine 1986) one would write:

$$\text{score} \sim 2 @ 10.$$

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 (denoted by “ \circ ”):

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

The shift and stretch operators are just special cases of what Jaffe terms *time maps* (Jaffe 1985), Anderson and Kuivila (1990) call *time deformations*, and I call *time warps* (Dannenberg 1989).

It is interesting to think about combining the effects of different time-warp functions. For example, there might be an overall tempo function with several local rubato functions superimposed. There might also be perturbations of beat durations as suggested in the work of Clynes (1987) and Bilmes (1993). Time warps can be combined or nested to arbitrarily deep levels. The effect of nested warp functions is described mathematically by function composition as illustrated above.

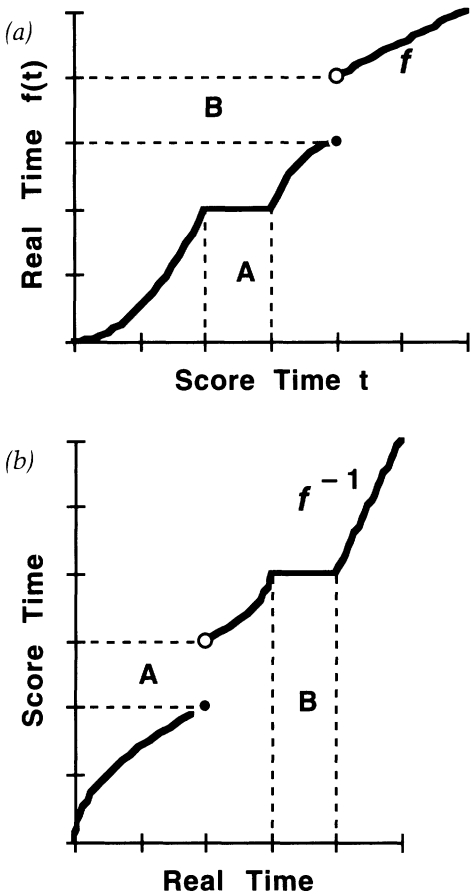
In the general case, a time warp can be any function that maps score time (often measured in beats) to real time (measured in seconds). I will use the term score time, but many terms have been used, including beat time, virtual time, local time, and logical time. When time warps are composed, these terms become relative (the “real time” of one time warp becomes the “score time” of another), so Nyquist uses the terms *local* and *global* time as relative terms.

Normally, a time warp is restricted to be monotonically nondecreasing, corresponding to the idea that time never flows backward. This implies that time warps have well-defined inverses, which is necessary in our formalism. (The inverse of a function *f* is denoted by f^{-1} , and defined such that $f^{-1}(f(t)) = t$. See Figure 1.)

It is sometimes convenient to allow the function to be flat (constant) over some interval (A in Figure 1), meaning that an interval of score time is traversed instantaneously in real time. It is also convenient to allow discontinuities, indicating an interval of real time where logical time stops (B in Figure 1).

The convention of expressing time warps as functions from score time to real time arises because in

Figure 1. A time-warping function (a), and its inverse (b). Interval A in both graphs represents an instantaneous jump ahead in the score time, while interval B represents a pause in the score where the tempo stops.



an implementation, one normally starts with score times attached to sound events, and the problem is to map these events to real time. It is also convenient to construct a time-warp function from tempo specifications, because tempo changes are normally indicated at a particular score time rather than a particular real time. However, it is perfectly possible to develop the theory in terms of maps from real time to score time, as these maps would simply be inverses of the time-map convention we have adopted.

Continuous Functions

If the time warp is to be applied to discrete events such as note-ons or note-offs, 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, so our goal for synthesis or real-time control is to map the score-time functions to real-time functions.

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:

$$g(f^{-1}(t)).$$

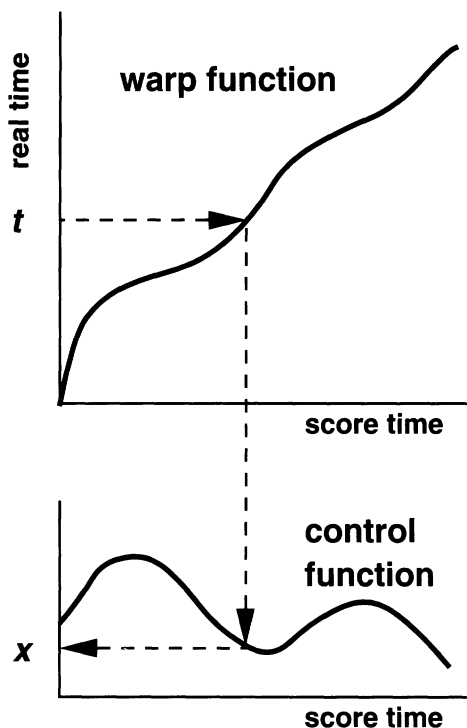
(Remember that time warp f maps score time to real time, so its inverse f^{-1} maps real time to score time, and g is a function of score time. Therefore, f^{-1} is applied to real time t to get score time, then g is evaluated at that point, yielding the value of g at a given real time.) Figure 2 shows this mapping graphically.

To summarize the results so far, it is clear that the shift and stretch operators seen in many music representations and languages are special cases of general time-warping functions. The literature maintains that time-warping functions can be nested using function composition, so this provides a general way to handle nested shifts and stretches in conjunction with other warp operators. Finally, it is shown that continuous functions can be warped using function composition and function inverse.

Abstraction Issues

Once time warps are introduced, however, the “vibrato problem” (Desain and Honing 1992) must be revisited: what is the interpretation of warped vibrato? Should the vibrato rate fluctuate when time is warped? Previous work has generally ignored this problem. One (partial) solution is to “build in” methods for handling warps, so that vibrato would automatically 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.

Figure 2. To evaluate a warped control function at real time t , map t to score time and apply the control function to yield the result x .



An example of this approach is seen in most Music N implementations. As described earlier, the starting time and duration of a note can be viewed as simple time-warping functions. Unit generators that implement the note have different, specific behaviors with respect to shifting and stretching. As the duration increases, oscillators produce more cycles, envelopes often stretch their middle segment (or segments) but not their beginning or end, and sound-file readers produce their samples without stretching or repeating.

The Music N approach works reasonably well, because there are no high-level tempo operations that can be applied at the score level. With only two discrete parameters, start time and duration, the options are fairly limited and it is relatively easy to produce the desired behavior. What would happen if all notes and phrases were synthesized in a richer context with continuous time warping? Furthermore, what if these time warps could be applied at various levels, ranging from entire compositions down to individual grains in granular synthesis?

The problem is that different structures must “respond” to time warping in different ways. Although default behaviors are convenient, it must also be possible for the composer to specify exactly how a behavior should change (or not change) according to time-warp functions. Our solution in Nyquist is to extend the mechanisms introduced in Arctic (Dannenberg, McAvinney, and Rubine 1986) and available in Canon (Dannenberg 1989). 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. Also, 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, a way of communicating transformation parameters to behaviors is needed. 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 clumsy notation system. In Nyquist, 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 automatically. 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 semitones. 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 such as transposition and time warps modify the environment *before* they evaluate the enclosed Lisp functions that implement behaviors. It is critical that transformations operate in this manner. This gives behavioral abstractions a chance to determine exactly which transformations will be implemented, and how. 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. Refer to Dannenberg (1989, 1997) 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 (1992) solve the problem by returning functions of multiple parameters. As in Nyquist, transformations modify parameters rather than actual behaviors, and the “how to transform” knowledge is encapsulated in Lisp functions rather than in transformation operators.

Time Warping and Continuous Transformations

If the environment provides a time-warping function f , any discrete time point t can be mapped to real time simply by computing $f(t)$. For example, the Nyquist `pw1` function computes a piecewise linear function from a list of breakpoints. When `pw1` is warped, the default is to map each breakpoint into real time.

The Nyquist `sine` function generates a sinusoidal signal at a given pitch and duration. (Do not confuse this with the Lisp `sin`, which is the familiar trigonometric function.) 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 an 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 $g(f^{-1}(t)) = (g \circ f^{-1})(t)$), we could write the expression:

```
(snd-compose (warp-abs nil (sine p))
              (snd-inverse (get-warp)
                           (local-to-global 0) *sound-
                           srate*)).
```

Here, `snd-compose` denotes function composition, `snd-inverse` denotes function inverse, and `(get-warp)` returns the time-warp function from 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 warped only once. One could even use this mechanism for FM synthesis, although the built-in Nyquist FM oscillators are more efficient.

Alternatively, we can use the built-in macro `continuous-sound-warp`, which is a convenient shorthand for the operations shown above:

```
(continuous-sound-warp (sine p))
```

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 values produced by `contour`. The `contour` 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 , 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* (local-to-global 0))
```

or simply `(get-loud)`, where `sref` evaluates a time function at a particular time, and `local-to-global` converts a local time to a global time using the environment variable `*warp*` as a time map. To offset the added complexity of continuous transforms and time warps, users are expected to access the transformation environment through calls to `(get-loud)`, `(get-transpose)`, `(get-sustain)`, etc.

Finally, note that an integration operator is useful for converting a tempo function to a warp function. The time-warp function is the integral of the reciprocal of the tempo function. Integration and reciprocal are signal operators in Nyquist, so these computations can be performed numerically.

The following computes a warp function given a tempo curve:

```
(integrate (reciprocal (tempo-curve)))
```

which can be abbreviated:

```
(tempo-to-warp (tempo-curve))
```

Implementation

In previous systems, time-warp functions were special data types, often restricted to be piecewise linear or polynomial in order to simplify mapping operations, inversion, and integration. It is usually considered important to make mapping from score time to real time efficient.

In contrast, Nyquist offers a rich set of efficient signal-processing operations and a built-in data type for signal representation, so it is only natural to express time maps as Nyquist signals. This allows the full power of a synthesis language to be applied to the task of specifying interesting functions. For example, a piecewise linear tempo function can be smoothed with a Nyquist low-pass filter to achieve more graceful tempo changes.

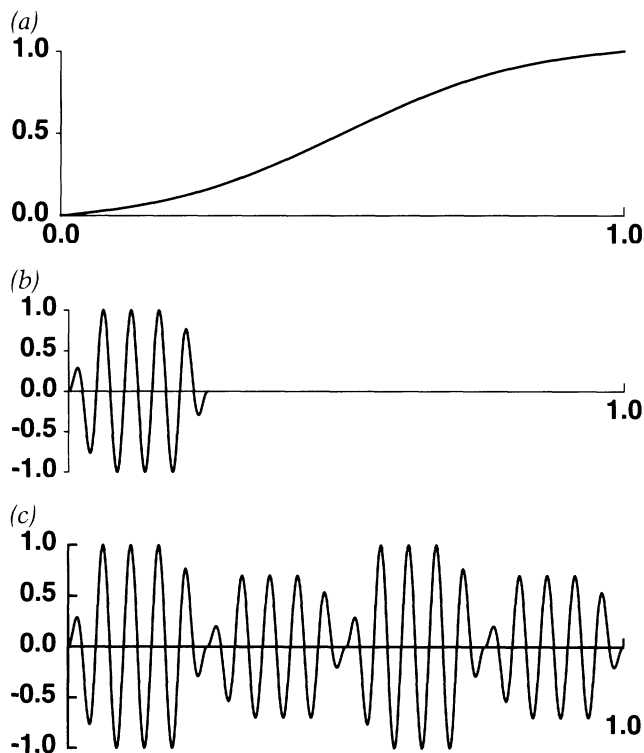
A number of new signal operations have been added to Nyquist to support time warps. These include function composition, function inverse, and evaluation at a point (`sref`). Both composition and inverse can be computed incrementally in the style of other unit generators. Typically, low sample rates of 10 to 100 samples per second are adequate, so the computation and storage overhead is minimal compared to the audio signal computation. Support for multiple sample rates is important here.

One weakness in the implementation is that continuous transformations and time warps are bound to environment variables, which means that they (and their sampled representations) are retained in memory more than is strictly necessary. If a transformation is applied to a long sequence, every element of the sequence sees the same transformation function, and the function is not freed until the entire sequence has been computed. It should be possible to modify the environment incrementally and dispose of the initial transformation samples as the sequence progresses. Since the transformations (and all Nyquist signals) are incrementally computed, this means that only a few samples would ever be in memory at a given time, as is the case with audio signals. This would also speed up function lookups due to `sref`, which currently requires a linear search from the beginning of the signal.

In the current implementation, transformation functions at a sample rate of 100 Hz take about 24 kB of storage per minute, and the allocation of samples in blocks allows `sref` to perform its linear scan very quickly. It is only high-sample-rate control functions that should cause problems. Furthermore, no functions are computed or stored unless a nontrivial transformation is applied. Nyquist handles the trivial but common shift and stretch operations and their composition as special cases that do not require any signal computation.

Figure 3. Time warp variations: a time-warp function (a) used in the following examples; a “wobble” with five oscillations, not

warped (b); and an unwarped compound behavior with four “wiggles” in sequence, with varying amplitudes (c).



A Catalog of Warp Behaviors

Figures 3 through 7 illustrate a variety of ways a behavior can respond to time warping. Figure 3a is a time-warp function used in the other figures. It was computed using the following definition, which simply adds one cycle of a small-amplitude sinusoid to a linear ramp:

```
(defun warpfm ())
  (sum (ramp) (scale -0.1 (lfo 1))))
```

Figure 3b is an unwarped signal consisting of five oscillations under an envelope, a behavior I will call a “wobble,” defined as follows:

```
(defun wobble (amp)
  (stretch .25
    (mult (wobble-env amp)
      (osc (hz-to-step 20))))))
```

Figure 3c shows four of these wiggles in sequence, each with a different amplitude. This behavior is defined by:

Figure 4. The compound behavior warped.

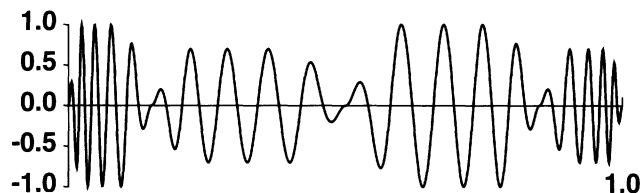


Figure 4

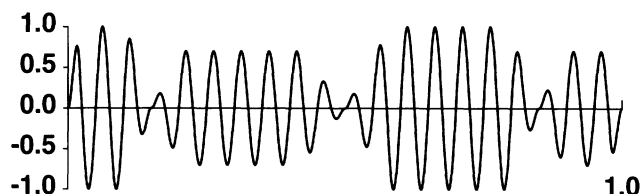


Figure 5

Figure 5. Only start and end points are warped; oscillation frequency is fixed.

```
(defun 4wiggles ())
  (seq (wobble 1) (wobble 0.7)
    (wobble 1) (wobble 0.7)))
```

Now, imagine that these four wiggles are some kind of control function that you want to warp continuously. Figure 4 shows a continuous warp by function composition. The code is as follows:

```
(control-warp (warpfm) (4wiggles))
```

In this code, the `control-warp` function performs function composition on two signals.

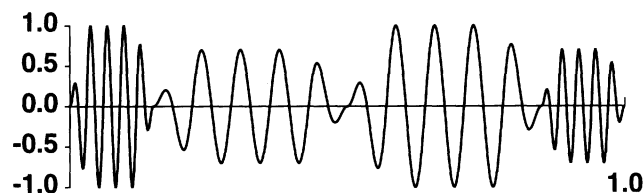
Now suppose these wiggles represent notes. In this case, the warp function should map the starting and ending times, leaving the oscillation frequency untouched. The code is as follows (see also Figure 5):

```
(warp (warpfm) (4wiggles))
```

Note that the only change from the previous example is to use the `warp` transformation rather than a signal-processing operation. Partly by design and partly by luck, `4wiggles` behaves as a sequence of notes when warped.

Another possibility is that the wiggles represent steady vibrato. We want to stretch or shrink the vibrato curve to fit exactly four wiggles from beginning to end, but the frequency should be steady within each wiggle. (This is not a realistic vibrato model, but at least it hints at the idea of fitting an

Figure 6. Start and end points are warped; wiggles are linearly stretched to fit.



integral number of vibrato or trill cycles within a metrical unit.) This is shown in Figure 6. To implement that behavior, we start by redefining wiggle as vib-wiggle, as follows:

```
(defun vib-wiggle (amp)
  (let ((dur (get-duration .25)))
    (stretch-abs dur
      (mult (wiggle-env amp)
        (osc (hz-to-step
              (/ 5 dur)))))))
```

This defines `dur` to be the true duration after warping the start and end times according to the environment. Then, `stretch-abs` is used to eliminate the warp function from the environment that is seen by the `mult` expression. Finally, the frequency of the oscillation is set to cover five periods in the time-span of `dur`.

Using the `vib-wiggle` behavior, we build up a new sequence of four:

```
(defun 4vibs ()
  (seq (vib-wiggle 1) (vib-wiggle .7)
    (vib-wiggle 1) (vib-wiggle .7)))
```

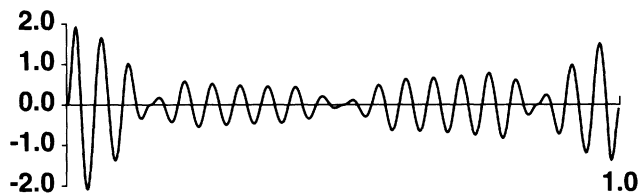
Finally, the sequence is warped as before:

```
(warp (warpfn) (4vibs))
```

Notice the difference between Figures 5 and 6. Also notice that Figures 4 and 6 are similar, but (as is desired) oscillation frequency is constant within each wiggle in Figure 6, while it changes over the course of a wiggle in Figure 4.

Figure 7 shows how behaviors may be linked in complex ways to the transformation environment. Here, amplitude is inversely proportional to tempo. The `get-tempo` function (not shown) computes the slope of the inverse of the environment's warp function.

Figure 7. The amplitude is inversely proportional to the "tempo."



```
(warp (warpfn)
  (mult (get-tempo) (4wiggles)))
```

As in Figure 5, each wiggle's duration is affected, but not its oscillation frequency. In Figure 7, however, the tempo affects not only the wiggle's duration, but also its amplitude.

Comparison with GTF

The publication of "Time Functions . . ." (Desain and Honing 1992) led to some debate over the differences between Time Functions and Arctic (Dannenberg 1992). Time Functions were generalized (Desain and Honing 1993) to become GTF. After much discussion among the authors, the issues have become more clear (Honing 1995). An important point of the Time Functions articles is the idea that control functions need more than one parameter. In Arctic and Nyquist, the primary parameters are starting time, stretch factor, and time. The starting time and stretch factor are used to create a function of time, which is independent of any object. In GTF, a function of three parameters (start, duration, and progress) is generated and attached to a note or compound object.

One nice property of GTF is that time functions are explicitly attached to objects that are subject to transformations such as stretching. When an object is stretched, the attached time functions receive a correspondingly larger duration parameter, and adjust accordingly. In Arctic and Nyquist, there is no such attachment. This makes some operations simpler, such as having one behavior depend upon information computed by another, or having several behaviors depend upon a single function for synchronized vibrato. On the other hand, a common er-

ror in Nyquist programs is to compute an envelope that is too short or is not time-aligned with the signal it is intended to control. This error is harder to make with the GTF abstractions.

Ultimately, the two approaches seem to have taken the fundamental concept of behavioral abstraction in different directions. Arctic and Nyquist took an approach intended for real-time control where stretch makes sense, and duration requires knowledge of the future. Control functions have no attachment, and are strictly functions of the environment in Arctic or Nyquist. The creators of GTF took a more non-real-time approach, and introduced a clever use of closures to achieve late binding of time and duration parameters. The GTF control functions depend upon the objects to which they are attached, which in turn are subject to duration and starting-time transformations. Given that Nyquist turns out to be best suited to non-real-time applications, it would be nice to offer more flexibility for specifying time and duration.

In comparing these two approaches, perhaps the most important question is "Where does duration come from?" In Arctic and Nyquist, the behavioral abstraction principle is carried to the extreme. Ultimately, time and duration are determined by the behavior. If you tell a behavior to start at time T , it is perfectly acceptable for the behavior to delay for some interval (perhaps it is a phrase with an initial rest). Nyquist semantics are a consequence of eliminating the concepts of scores, note lists, and objects. With GTF, duration is passed from object to time function, and objects are not free to make timing decisions.

This design seems to work against abstraction in GTF. For example, one can define a Nyquist "note" behavior that plays longer when the `loud` transformation indicates the note should be louder (a technique that many musicians should recognize). Since control functions in GTF are orthogonal to the events they act upon, this sort of dependency would be difficult to create. In my view, the designers of GTF and Nyquist have applied abstraction differently to achieve differing goals.

Summary and Conclusions

This article has discussed several aspects of Nyquist and their interactions. The idea of behavioral abstraction is that a single abstract behavior can be instantiated in different contexts for 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. The language allows the programmer to specify "how to stretch" or how to interpret any number of other transformations.

Time warping or mapping functions have been presented along with semantics that are consistent with abstraction principles. Shift and stretch operators are just special cases of general time warping. Finally, continuous control parameters can be integrated with behavioral abstraction and time warping. 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`, the primitives on which warp computations rest, all have fast, incremental implementations. Furthermore, time warps and continuous transformations can be computed at low sample rates.

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. Time transformations in Nyquist are powerful enough to express new concepts such as expressive timing offsets (Bilmes 1993). The encapsulation of behavioral details offers the composer a powerful new tool for expressive control. We look forward to the exploration of these possibilities.

Acknowledgments

The author wishes to thank Henkjan Honing and Peter Desain for many stimulating conversations about this and related work.

References

- Anderson, D. P., and R. Kuivila. 1990. "A System for Computer Music Performance." *ACM Transactions on Computer Systems* 8(1):56–82.
- Bilmes, J. 1993. "Timing Is of the Essence: Perceptual and Computational Techniques for Representing, Learning, and Reproducing Expressive Timing in Percussive Rhythm." Master's thesis, Massachusetts Institute of Technology.
- Buxton, W., W. Reeves, R. Baecker, and L. Mezei. 1985. "The Use of Hierarchy and Instance in a Data Structure for Computer Music." In C. Roads and J. Strawn, eds. *Foundations of Computer Music*. Cambridge, Massachusetts: MIT Press, pp. 443–466.
- Clynes, M. 1987. "What Can a Musician Learn about Music Performance from Newly Discovered Microstructure Principles (PM and PAS)?" In A. Gabrielsson, ed. *Action and Perception in Rhythm and Music*. Publications issued by the Royal Swedish Academy of Music, no. 55, pp. 201–233.
- Cointe, P., and X. Rodet. 1984. "Formes: An Object and Time Oriented System for Music Composition and Synthesis." *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. New York: ACM, pp. 85–95.
- Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal* 13(1):47–56.
- Dannenberg, R. B. 1992. "Time Functions" (letter). *Computer Music Journal* 16(3):7–8.
- Dannenberg, R. B. 1997. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis." *Computer Music Journal* 21(3): 50–60.
- Dannenberg, R. B., P. McAvinney, and D. Rubine. 1986. "Arctic: A Functional Language for Real-Time Systems." *Computer Music Journal* 10(4):67–78.
- Desain, P., and H. Honing. 1992. "Time Functions Function Best as Functions of Multiple Times." *Computer Music Journal* 16(2):17–34. Reprinted in P. Desain and H. Honing. 1992. *Music, Mind and Machine, Studies in Computer Music, Music Cognition and Artificial Intelligence*. Amsterdam: Thesis Publishers.
- Desain, P., and H. Honing. 1993. "On Continuous Musical Control of Discrete Musical Objects." *Proceedings of the 1993 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 218–221.
- Honing, H. 1995. "The Vibrato Problem, Comparing Two Solutions." *Computer Music Journal* 19(3):32–49.
- Jaffe, D. 1985. "Ensemble Timing in Computer Music." *Computer Music Journal* 9(4):38–48.
- Mathews, M. V. 1969. *The Technology of Computer Music*. Cambridge, Massachusetts: MIT Press.
- Orlarey, Y. 1986. "MLOGO: A MIDI Composing Environment." *Proceedings of the 1986 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 211–213.
- Rogers, J., J. Rockstroh, and P. Batstone. 1980. "Music-Time and Clock-Time Similarities under Tempo Changes." *Proceedings of the 1980 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 404–442.
- Spiegel, L. 1981. "Manipulations of Musical Patterns." *Proceedings of the Symposium on Small Computers in the Arts*. Los Alamitos, California: IEEE Computer Society, pp. 19–22.