The Implementation of Nyquist, A Sound Synthesis Language[1]

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

ABSTRACT: Nyquist is a functional language for sound synthesis with an efficient implementation. It is shown how various language features lead to a rather elaborate representation for signals, consisting of a sharable linked list of sample blocks terminated by a suspended computation. The representation supports infinite sounds, allows sound computations to be instantiated dynamically, and dynamically optimizes the sound computation.

1. Introduction

Nyquist is a new language for sound synthesis, based on an evolving series of languages and implementations that include Arctic, Canon, and Fugue. These languages are all based on powerful functional programming mechanisms for describing temporal behavior. From these general mechanisms, composers can create a variety of temporal structures such as notes, chords, phrases, trills, and synthesis elements such as granular synthesis, envelopes and vibrato functions. Unfortunately, previous implementations have had too many limitations for practical use. For example, Canon did not handle sampled audio, and Fugue used vast amounts of memory and was hard to extend.

Nyquist solves these practical problems using new implementation techniques. Declarative programs are automatically transformed into an efficient incremental form taking approximately the same space (within a constant factor) as Music V or Csound. This transformation takes place dynamically, so Nyquist has no need to preprocess an orchestra or ‘‘patch’’. This allows Lisp-based Nyquist programs to construct new synthesis patches on-the-fly and allows users to execute synthesis commands interactively. Furthermore, infinite (in time) sounds and scores can be written and evaluated. Nyquist is intended to operate in both real-time and non-real-time modes.

Due to space limitations, this paper will focus on the run-time representation of sound in Nyquist. I will describe only enough of Nyquist to motivate the representation issues. To get a more complete picture, consult previous articles on the language design [Dannenberg 91, Dannenberg 92a] and performance issues [Dannenberg 92b]. The story is still not complete. An interesting part of the Nyquist implementation is a compiler that translates inner loop expressions into C code for use in Nyquist, making it possible to extend Nyquist without a detailed understanding of the internal data structures and programming conventions. Also, a detailed comparison of Nyquist with other systems is useful in understanding design decisions. These topics await future publication.

2. Incremental (Lazy) Evaluation

Nyquist uses a declarative and functional style, in which expressions are evaluated to create and modify sounds. For example, to form the sum of two sinusoids, write: **`(s-add (osc) (osc))`**, where each **`(osc)`** expression evaluates to a signal, and **`s-add`** sums the two signals. In Fugue, the addition of signals took place as follows: space was allocated for the entire result, then signals were added one-at-a-time. This was workable for small sounds, but practical music synthesis required too much space. The solution in Nyquist is to perform the synthesis and addition incrementally so that at any one time there are only a few blocks of samples in memory.

This is similar to the approach taken in Music *n* languages such as Csound, cmusic, and Cmix [Pope 93], and, in fact, there is a close correspondence between unit generators of Music *n* and functions in Nyquist. The main difference is that in Music *n*, the order of execution is explicit, whereas in Nyquist, evaluation order is deduced from data dependencies. Also, Nyquist sounds are first-class values that may be assigned to variables or passed as parameters.

Figure 1 illustrates an expression and the resulting computation structure consisting of a graph of synthesis objects. This graph is, in effect, a ‘‘suspended computation,’’ that is, a structure that represents a computation waiting to happen. This graph is an efficient way to represent the sound. When actual samples are needed, the **`s-add`**

[1]Published as: Dannenberg, ‘‘The Implementation of Nyquist, A Sound Synthesis Language,’’ in ***Proceedings of the 1993 International Computer Music Conference***, International Computer Music Association, (September 1993), pp. 168-171.

suspension is asked to deliver a block of samples. This suspension recognizes that it needs blocks from each **osc** suspension, so it recursively asks each of them to produce a block of samples. These are added to produce a result block. The suspensions keep track of their state (e.g., current phase and frequency of oscillation) so that computation can be resumed when the next block is requested.

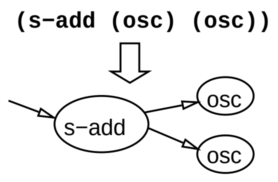

Figure 1: Nyquist sound expression and resulting representation.

With this evaluation strategy, each block of samples is typically used immediately after it is computed, and the space requirements are similar to those of Music *n*. Furthermore, whenever a block is needed, it is computed on demand, so the order of evaluation is determined automatically. There is no need to order unit generators by hand as in Music *n*. Since the order is determined at the time of evaluation, the computation graph may change dynamically. In particular, when a new ‘‘note’’ is played, the graph is expanded accordingly. This is in contrast to the static graphs used by Max on the ISPW [Puckette 91], where all resources must be pre-allocated.

3. Shared Values

As is often the case, things are not really so simple. In Nyquist, sounds are values that can be assigned to variables and reused any number of times. It would be conceivable (and semantically correct) to simply copy a sound structure whenever it is needed in the same way that most languages copy integer values when they are passed as parameters or read from variables. Unfortunately, sounds can be large structures that are expensive to copy. Furthermore, if a sound were copied, each copy would eventually be called upon to perform identical computations to deliver identical sample streams. Clearly, we need a way to share sounds that eliminates redundant computation.

Nyquist allows great flexibility in dealing with sounds. For example, it is possible to compute the maximum value of a sound or to reverse the sound, both of which require a full representation of the sound. What happens if a maximum value suspension asks a sound to compute and return all of its blocks, and then an addition begins asking for blocks (starting with the first)? If the sound samples are to be shared, it is necessary to save sample blocks for as long as there are potential readers. Note that this problem does not occur in Music *n* because signals are special data types that can only be accessed ‘‘now’’ at a global current time. In Cmix, sounds can be accessed randomly only after writing them to sound files.

The need for sharing leads to a new representation (see Figure 2) in which samples are stored in a linked list of sample blocks. Sound sample blocks are accessed sequentially by following list pointers. Each reader of a sound uses a sound header object to remember the current position in the list and other state information. In the figure, the sound is shared by two readers, each with a sound header. One reader is a block ahead of the other. Incremental evaluation is still used, placing the suspension at the end of the list. When a reader needs to read beyond the last block on the list, the suspension is asked to compute a new block which is inserted between the end of the list and the suspension. The list is organized so that all readers see and share the same samples, regardless of when the samples are produced by the suspension or which reader reads first.

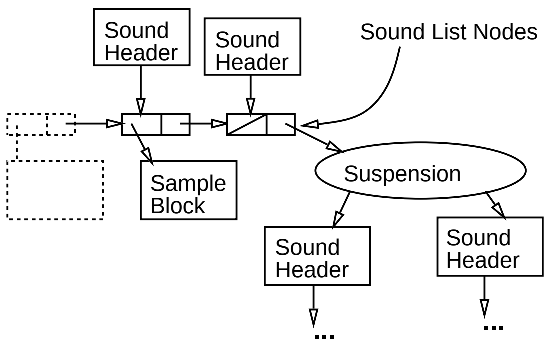

Figure 2: Sound representation in Nyquist.

4. Storage Reclamation

Now a new problem arises. Since blocks are attached to a list as they are generated, what prevents lists from exhausting the available storage? The solution uses reference counting to move blocks from the head of the list to a free list from which they can be allocated for reuse.

Reference counts record the number of outstanding references (pointers) to list nodes and sample blocks. When the count goes to zero, the node or sample block is freed. Reference counting is used so that blocks are freed as early as possible. In Figure 2, the dotted lines illustrate the previous head of the sound list, which was freed when no more sound headers referenced it.

5. Addition

Nyquist can add sounds with different start times, so signal addition must be efficient in the frequent case where one signal is zero. Figure 3 illustrates a case where two sounds at widely spaced times must be added. When sound operands start at different times, the suspension can either ''coerce'' one operand into supplying leading zeros to align the sounds, or the misalignment can be handled as a special case.

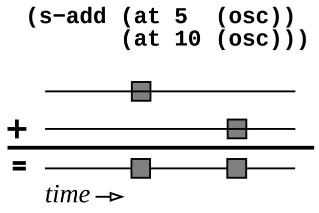

Figure 3: Sounds may have leading zeros, trailing zeros, and internal gaps.

Addition is optimized to handle the case of Figure 3 with maximum efficiency. The addition suspension is implemented as a finite-state machine, where the state indicates which operands are non-zero, and transitions occur at the start and stop times of the operands. When one operand is zero, the sound block from the other operand can simply be linked into the sound list representing the sum. No samples are added or even copied!

Multiplication can use a similar optimization: if one operand block is all zero (tested by a pointer comparison), the zero block can be linked into the result with no multiplication or zero-fill required.

Some of these optimizations require block alignment. List nodes have a length field, allowing suspensions to generate partially filled blocks. Since blocks can vary in size and sample rate, suspensions are written to compute samples up to the next operand block boundary, fetch a new block, and resume until an output block is filled.

6. Efficient Transformations

Nyquist allows various transformations on sounds, such as shifting in time, scaling, and stretching. These need to be efficient since they are common operations. The sound headers mentioned earlier contain transformation information: to scale a sound, the header is copied and the copy's scale-factor field is modified[2].

A drawback of storing transformations in the header is that all operators must apply the transformations to the raw samples. We have already seen how time-shifted signals are handled. In the case of scale factors, there are several approaches:

1. The operator object can always multiply each sample by the scale factor, costing one multiply *per reader*.
2. The operator object can use special-case code if the scale factor is 1.0 so that a penalty is paid only for non-unity scale factors.
3. A scaling function can be applied to operands with non-unity scale factors, creating a new header, sound list, and suspension.
4. The scale factor can be commuted to the result, e.g., the multiply operator returns a sound whose scale factor is the product of the scale factors of the operand sounds.
5. The scale factor can be factored into other operations, for example, pre-scaling filter coefficients, to avoid any per-sample cost.

The Nyquist compiler chooses one of methods 5, 4, and 3, in that order of preference.

[2]The copy is necessary because the sound might be shared. Remember that sounds are immutable values, so all operators and transformations generate new sounds rather than modify old ones.

7. Signal Termination

Although lazy evaluation allows Nyquist sounds to be infinite, efficiency concerns dictate that sound computation should come to an end if possible. Most signal generators in Nyquist produce a signal only over some time interval, and Nyquist semantics say that the sound is zero outside of this interval. A signal that goes to zero is represented by a list node that points to itself (see Figure 4), creating a virtually infinite list of zero sound blocks. When a suspension detects that its future output will be zero, it links the tail of its sound list to the special terminal list node. The suspension then deletes itself. Other suspensions can check for the terminal list node to discover when their operands have gone to zero.

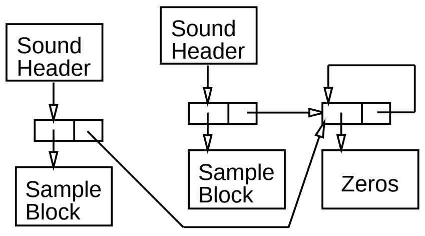

Figure 4: Representation for sound termination. Two sounds are shown, each with one more block to read before termination.

8. Logical Stop Time and Sequences

Another feature of Nyquist is that sounds have intrinsic ending times called the *logical stop time* (LST). A **`seq`** operator allows sounds to be added together, aligning the start time of one sound with the LST of the previous sound. The LST may be earlier or later than the termination time. For example, the LST may correspond to a note release time, after which the note may decay until the termination time.

In the example, **`(seq (osc) (osc))`**, the start time of the second **`(osc)`** expression depends upon the LST of the first **`(osc)`**. We reserve a flag in each list node to mark the logical stop location. When the flag is set, it indicates the LST is the time of the first sample of the block pointed to by the list node[3]. Since block lengths are variable, the LST is accurate to the nearest sample.

Evaluation of each item in a sequence must be deferred until the LST of the previous item. This is accomplished by capturing the Lisp environment (including local variable bindings) in a closure and saving the closure in a special seq suspension. The closure is evaluated when the LST is reached. At this point, the seq suspension is converted to an addition suspension, and the signals are added.

Since seq suspensions are converted to additions, there is the danger that a long sequence will degenerate to a deeply nested structure of additions. The addition suspension is optimized to link its output list to its operand list when only one operand remains. (See Figure 5.) In effect, this simplifies computations of the form ‘‘0 + *x*’’ to ‘‘*x*’’ by eliminating one addition. This is only possible, however, when the operand’s scale factor is one, and the sample rate matches that of the sum.

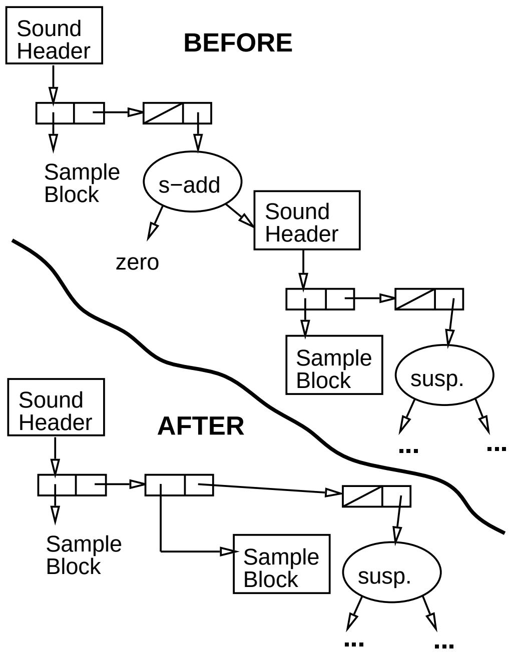

Figure 5: Optimization of add when one operand terminates and one remains.

9. Sample Rate and Multichannel Signals

Sample rate is specified in the header of each sound, and Nyquist allows arbitrarily mixed sample rates. It is the responsibility of the suspension to interpolate operand samples (linearly) when rate conversion is required. Originally, it was planned to fold interpolation into the inner loop, but it was discovered that this does not result in substantial savings, so sample interpolation is performed by a separate suspension when needed.

Multichannel signals are represented by Lisp arrays where each element of the array is a single channel sound. Nyquist operators are generalized in the expected way. For example, when a stereo signal is enveloped, the left and right channels are each multiplied by the envelope signal, yielding a stereo signal. If the envelope is also stereo, then the corresponding channels are multiplied.

10. Discussion

What started out as a fairly simple idea (linked sound blocks with sharing and lazy evaluation) has become quite complex. The complexity is a direct result of supporting a set of powerful language features. For example, the linked list of blocks occurs because Nyquist sound values must be easy to copy and share.

The order of invoking suspensions is dynamically determined because sound graphs in Nyquist are dynamic. However, it should be possible for a compiler to find static schedules for subgraphs; e.g., the patch for a single note. Static graphs allow other optimizations that might not be possible with Nyquist.

An interesting feature of Nyquist is the **`seq`** operator, which instantiates a new signal computation when another reaches its logical stop time. This can take place on any sample boundary, and the location can be computed at the signal processing level. This is in contrast to most systems where the stop time (logical or otherwise) is considered control information to be passed ‘‘down’’ to the signal processing objects rather than passed ‘‘up’’ from signals to the control level.

Nyquist, with its support for multiple sample rates and dynamic computation ordering, has a very distributed style of control. Compare this to Music *n*, where there is a global sample rate and global block size, and all unit generators are kept in lock step. For large blocks, overhead is small, but there could be a problem in real-time systems with smaller block sizes. We need experience with a multi-sample-rate language with sample-accurate controls (like Nyquist) to judge which of these features justify the overhead and complexity. To this end, Nyquist is available from the author.

References

[Dannenberg 91] Dannenberg, R. B., C. L. Fraley, and P. Velikonja. Fugue: A Functional Language for Sound Synthesis. *Computer* 24(7):36-42, July, 1991.

[3]The LST can be changed by a transformation, indicated by an LST field in each reader. If specified, this overrides the flag in the list node.

[Dannenberg 92a] Dannenberg, R. B., C. L. Fraley, and P. Velikonja. A Functional Language for Sound Synthesis with Behavioral Abstraction and Lazy Evaluation. *Readings in Computer-Generated Music.* In Denis Baggi, IEEE Computer Society Press, Los Alamitos, CA, 1992.

[Dannenberg 92b] 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.

[Pope 93] Pope, S. T. Machine Tongues XV: Three Packages for Software Sound Synthesis. *Computer Music Journal* 17(2):23-54, Summer, 1993.

[Puckette 91] Puckette, M. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal* 15(3):68-77, Fall, 1991.