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

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

Nyquist is an advanced functional language for sound synthesis and composition. One of the goals of Nyquist is to achieve efficiency comparable to more conventional Music N synthesis languages such as Csound (Vercoe 1986). Efficiency can be measured in space and time, and both are important: digital audio takes enormous amounts of memory, and sound synthesis programs are computationally intensive. The efficiency requirement interacts with various language features, leading to a rather elaborate representation for signals. I will show how this representation supports Nyquist semantics in a space and time-efficient manner. Among the features of the representation are incremental computation, dynamic storage allocation and reclamation, dynamic instantiation of new signals, representation of infinite sounds, and support for multi-channel, multi-sample-rate signals.

Introduction

Nyquist is based on an evolving series of languages and implementations that include Arctic (Dannenberg, McAvinney, and Rubine 1986), Canon (Dannenberg 1989), and Fugue (Dannenberg, Fraley, and Velikonja 1991). 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, and trills, as well as a variety of 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 (Mathews 1969) or Csound (Vercoe 1986). 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.

[1] Published as: , Roger B. Dannenberg, "The Implementation of Nyquist, a Sound Synthesis Language," *Computer Music Journal*, 21(3) (Fall 1997), pp. 71-82.

Furthermore, sounds and scores can be written and evaluated even if their durations are infinite.

This paper will focus on the run-time representation of sound in Nyquist, so Nyquist will be described only as needed to motivate the representation issues. To get a more complete picture, see the companion articles (Dannenberg, 1997a, and Dannenberg, 1997b). A final article in this series (Dannenberg and Thompson, 1997) will discuss the problem of organizing and optimizing inner loops for software synthesis. That article will include more measurements of Nyquist and other systems.

The implementation is presented as a set of solutions to various issues and problems, mostly arising from the need to support Nyquist semantics. The following section describes the representation of sounds and lazy evaluation, which work together to achieve space efficiency. Next, the optimized sound addition operation is described. Then, signal transformation and infinite sounds are discussed. The next section discusses how sounds can be ordered sequentially and how logical stop times are used. Following sections describe the implementation of multiple sample rates and multiple channel signals. Next software engineering issues are discussed, including the use of automatic code generation. Finally, performance is evaluated and the overall system is discussed.

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:

```
    (sum (osc c4) (osc c5)),
```

where each `(osc` *`pitch`*`)` expression evaluates to a signal, and `sum` sums the two signals. In Fugue, an earlier implementation, 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 (Dannenberg and Mercer, 1992).

This is similar to the approach taken in Music N languages such as Csound, cmusic, and Cmix (Pope 1993), 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 `sum` suspension is asked to deliver a block of samples. This suspension recognizes that it needs samples 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.

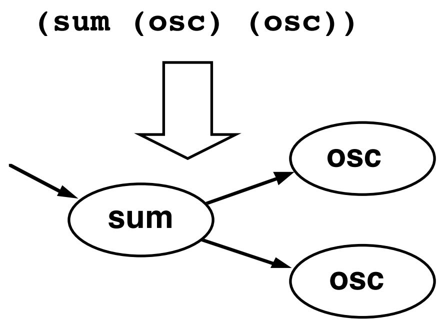

Figure 1. A Nyquist sound expression and resulting representation.

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 1991), where all resources must be pre-allocated.

Samples are computed in blocks so that the overhead of managing data structures and invoking sample computations is amortized over many samples. This same strategy is used in most sound synthesis implementations. Nyquist uses fixed-sized blocks to simplify storage management, but there is a length count to allow partially filled blocks.

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 is copied, each copy will 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 a signal addition suspension 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 (Lansky 1987, 1990), 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 or which reader reads first.

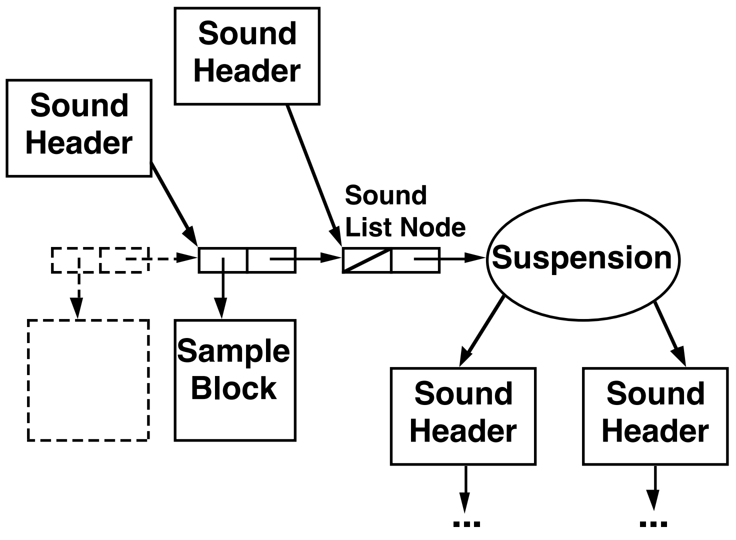

Figure 2. Sound representation in Nyquist.

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 a combination of reference counting and garbage collection (Schorr and Waite, 1967) to move blocks from the head of the list to a free list from which they can be allocated for reuse.

Reference counts (Pratt 1975) 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.

The Lisp interpreter upon which Nyquist is based (XLisp) does not reference count ordinary Lisp objects, which may in turn reference sounds. A problem that arises from this scheme is that a Lisp object can refer to a Nyquist sound, keeping its reference count at 1. When the Lisp object is freed, nothing happens because freed Lisp objects are not detected until the garbage collector runs[2]. This might delay the freeing of a sound reader, causing many sound blocks to be retained in memory unnecessarily.

This is a real problem because whenever a sound is passed as a parameter, the parameter binding is a Lisp object with a reference to the sound. The sound blocks will not be freed until the next garbage collection. The solution is to invoke the garbage collector whenever the list of sample blocks becomes empty. If this fails to free up any sample blocks, 50 new blocks are allocated. Usually, if there are no free sample blocks, it is because there is a Lisp reference to a sound which is growing and allocating blocks from the free list. Running the garbage collector when the free list is empty frees all of those blocks.

In retrospect, it might have been wise to avoid reference counting altogether and simply use the garbage collector. This would have required more modifications to the collector to handle all the sound representation structures. Also, garbage collection would run very frequently if reference counts were not used to free sound sample blocks.

In summary, the linked list representation of sounds allows sounds to be incrementally evaluated, linking sound blocks onto the tail of the list. As reader objects traverse the list, blocks are freed from the head of the list. In normal use, only one block is ever allocated to store samples (as in Music N), but if a sound is assigned to a Lisp variable or data structure, the sound is retained for as long as the reference remains. By manipulating blocks rather than single samples, the space and time overhead of the linked lists is amortized over many samples, making the relative overhead quite small.

Efficient Transformations

Nyquist allows various transformations on sounds, such as shifting or stretching them in time or scaling their amplitudes. These need to be efficient since they are common operations and there is no static structure or code to be compiled and optimized. The sound headers mentioned earlier contain transformation information, whereas sample block lists simply contain samples.

[2] The garbage collector in XLisp is a mark-sweep collector (Schorr and Waite 1967): pointers are followed from the stack and from global variables, and every reachable Lisp object is marked. At this point, any object that is unmarked is unreachable and therefore useless to the computation. The entire Lisp memory is scanned. Any unmarked object is linked onto a free list, and any marked object is unmarked in preparation for the next garbage collection.

To scale a sound, the header is copied and the copy's scale-factor field is modified[3].

A drawback of storing transformations in the header is that all operators must apply the transformations to the raw samples. Time-shifted signals are handled simply by reading them at the appropriate time. If a sound is delayed, the list of sample blocks to be read automatically implements a delay buffer. In the case of scale factors, there are several approaches:

1. Within a suspension, multiply each input sample by the scale factor, costing one multiply *per reader*.
2. Use special-case code if the scale factor is 1.0 so that a penalty is paid only for non-unity scale factors.
3. Implement non-unity scaling using a separate operator, costing one multiply per sample *plus* the overhead of another operator.
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. This costs nothing locally, but "passes the buck" to other operators.
5. The scale factor can be factored into other operations, for example, pre-scaling filter coefficients, to avoid any per-sample cost.

The Nyquist implementation uses one of methods 5, 4, and 3, in that order of preference, and methods 2 and 1 can be selectively applied to any operator (unit generator) when Nyquist is compiled.

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. One solution that was rejected is to "coerce" the signal that starts late to supply leading zeros to align the sounds. Because this generates extra work adding blocks of zeros, it was decided instead to handle the misalignment of starting times as a special case.

When one addend is zero, it is sufficient to simply copy the other addend from input to output. Unfortunately, even copying takes time. A better solution is to copy pointers to sample blocks rather than copy the blocks themselves. To enable this optimization, a sample block list is made up of list nodes that point to sample blocks, as shown in Figure 2. Sample blocks can be shared by lists, and both list nodes and sample blocks have reference counts.

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!

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

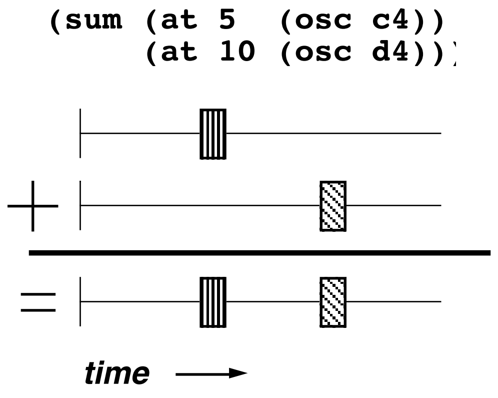

Figure 3. Sounds may have leading zeros, trailing zeros, and internal gaps.

Testing for zero is as expensive as adding, so a unique "block of zeros" is used to indicate large gaps of zero. A simple pointer comparison can determine if the block in question is zero or not. (Of course, if a block of computed samples turns out to be all zeros, they will escape detection and no optimization will be applied.)

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.

Signal Termination

Although lazy evaluation allows Nyquist sounds to be infinite, efficiency concerns dictate that sound computation should terminate as soon as 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. The point at which a signal 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 all 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.

The possibility of infinite signals enables some very interesting sound expressions. Consider the following recursive drum roll:

```
(defun drum-roll ()
   (seq (stroke) (drum-roll)))
```

which says roughly: "a drum roll is defined to be a sequence consisting of a stroke followed by a drum roll." Playing such a behavior will result in an infinite number of samples, but a finite space in primary memory. Now consider the following expression:

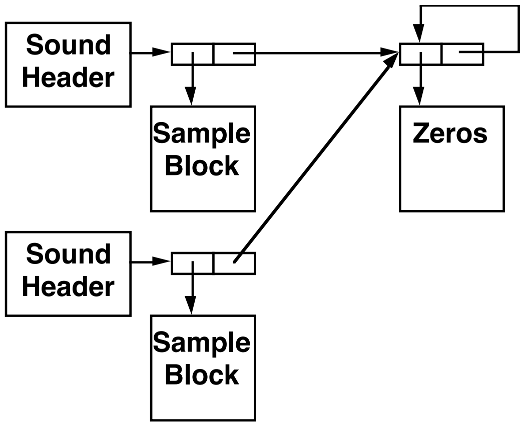

Figure 4. Representation for sound termination. Two sounds are shown, each with one more block to read before termination.

```
(defun short-roll ()
   (mult (drum-roll) (drum-envelope)))
```

This just multiplies a drum roll by a finite-length envelope. When the envelope goes to zero, the multiplication suspension will notice that one operand (the envelope) is zero. Therefore, the product is zero, and the suspension can link its output to the terminal (zero) list node. The suspension frees itself and reference counting and garbage collection dispose of the remaining drum roll.

This kind of recursive infinite structure might also be used in granular synthesis (Roads 1991). A granular synthesis instrument can generate potentially infinite sounds that need only to be multiplied by an envelope to obtain the desired amplitude and duration.

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 (note1) (note2))`, the start time of the `(note2)` expression depends upon the LST of `(note1)`. 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. Since block lengths are variable, a block can be split at any point to position the LST at any sample. The LST flag serves to communicate the default LST from the creator of the sound to the readers of the sound, but it is also possible to explicitly set the LST using the `set-logical-stop` transformation. This transformation sets an LST field in the sound reader object, and the field takes precedence over a list node's LST flag.

Evaluation of each item in a sequence (`seq`) must be deferred until the LST of the previous item. This is accomplished by capturing the Lisp environment (including local variable bindings and the Nyquist transformation environment) 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. Addition retains any overlapping tail from the first sound.

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.

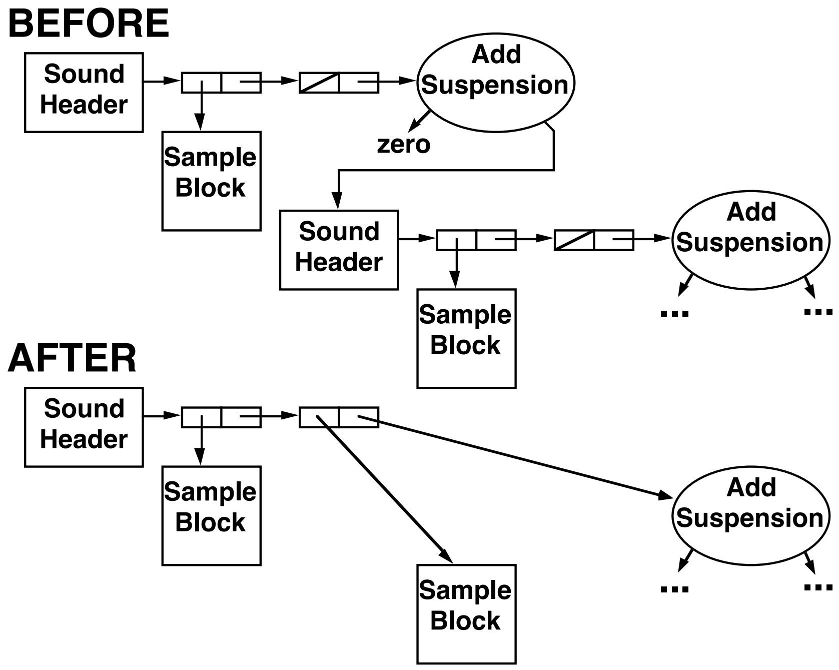

Figure 5. Optimization of add when one operand terminates and one remains.

Multiple Sample Rates

Sample rate is specified in the header of each sound, and Nyquist allows arbitrarily mixed sample rates. It is the responsibility of the operations to apply rate conversion when it is required. Rate conversion is handled in one of three ways.

In the first method, the operator implements linear interpolation in its inner loop. Automatic code generation (described below) is used to create three versions of interpolation: *two-point* interpolation finds the two nearest samples to the desired time and performs a standard linear interpolation. This is expensive, but is the best approach when the interpolated sample rate is near the original rate. *Ramp* interpolation is used when up-sampling by a large factor and there are many interpolated samples between each pair of original samples. Between original samples, an increment is computed. Each interpolated point is computed by adding the increment to the previous point. This is just another algorithm for linear interpolation. Finally, there is the case of *no interpolation* when the sample rates match. To handle interpolation in the inner loop, a compiler generates code for all three versions and the appropriate one is selected at run time.

In the second method, interpolation is performed by a separate operator (unit generator). For example, the function `prod` performs various tests on its arguments, interpolating them if necessary before calling `snd-prod`, the low-level signal processing function.

In the third method, rate conversion is implicit in the operator. For example, the frequency modulation of an oscillator is integrated to produce phase. Since integration smooths the frequency modulation signal, no further interpolation is performed. With variable filters, the computation of coefficients is expensive and is therefore performed at the sample rate of the filter parameter control signals without any interpolation. Of course, explicit interpolation or higher-order interpolation can always be applied explicitly if desired.

In the current implementation, this third method is used when it is appropriate. Otherwise, linear interpolation is performed in a separate operator (method 2). Method 1 is not used because it results in a larger program and the performance increase is limited.

Multi-channel Signals

Multi-channel 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 multiplied by an envelope, 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.

Lisp versus C

Nyquist is implemented in both Lisp and C. The XLisp interpreter (Betz 1988) was chosen as a base language for several reasons. First, XLisp itself is written in C, making it possible to port Nyquist to any machine with a C compiler. Second, I had both the prior experience and the tools to extend XLisp with new functions and data types. Most Lisp implementations have some way to handle external or “foreign” functions, but with XLisp, I was also able to add a new SOUND data type and extend the garbage collector appropriately. Finally, XLisp is compact. The entire Nyquist system is smaller than most Common Lisp systems.

A potential drawback of XLisp is that it is an interpreter. This makes it many times slower than C or compiled Lisp. For this reason, all signal processing and generation routines are written in C. Typically, all but a few percent of the computation time takes place in inner loops, so the overhead of the interpreter, garbage collection, and sound data structures is negligible.

Compiling Inner Loops

Most synthesis systems have relatively simple data structures with fixed block sizes, fixed sample rates, and other simplifications. Unit generators typically require only 10 to 100 lines of code apiece. In contrast, Nyquist uses very elaborate structures, and the resulting "unit generators" such as oscillators, filters, and multiplication, are complex. The sources of code complexity include these factors:

1. Suspensions must allocate sound sample blocks and linked list nodes to build sound structures,
2. Reference counts must be maintained,
3. Input and output sample blocks are not necessarily time-aligned,
4. Input and output sample blocks may have differing sample rates,
5. Operands may start at different times,
6. The suspension must watch for termination times and logical stop times, and
7. Sound headers may include an extra scale factor.

Taking all of these factors into consideration while writing a signal processing function would require extraordinary effort, and in fact none of these functions have been written entirely by hand. Instead, a compiler (written in XLisp) generates C code that is then compiled and linked into Nyquist to create a new operator. Thus, C is truly a "portable assembly code."

The compiler is based on the premise that almost all signal processing code in Nyquist is formulaic. The same sorts of tests and control structures occur repeatedly, but there are enough differences that implementation with macros or code templates is not practical. The advantage of the compiler is that whenever a bug is found, the compiler can be modified to systematically eliminate the bug from all operators. The compiler has made Nyquist much more reliable than Fugue (the previous implementation): even though Fugue operations were fewer and simpler, they were implemented by hand.

To give some idea of the function of the compiler, Figure 6 shows the specification of the function `prod` that takes the product of two signals. The specification is an unordered list of attributes and their values. The overall concept is that there is an inner loop that runs once per output sample. The inner loop makes one reference to each input signal. The job of the compiler is to construct all the code surrounding the inner loop so that the inner loop accesses or stores the proper values at the proper times.

```
(PROD-ALG
  (NAME "prod")
  (ARGUMENTS ("sound_type" "s1") ("sound_type" "s2"))
  (START (MAX s1 s2))
  (COMMUTATIVE (s1 s2))
  (INNER-LOOP "output = s1 * s2")
  (LINEAR s1 s2)
  (TERMINATE (MIN s1 s2))
  (LOGICAL-STOP (MIN s1 s2))
)
```

Figure 6. The declarative specification of the operator `prod`.

The `NAME` attribute specifies the name of the function, and `ARGUMENTS` lists the argument types and names. `START` specifies that the start time of the resulting signal is the maximum of the starting times of the two arguments, `s1` and `s2`. The `COMMUTATIVE` attribute indicates that `s1` and `s2` can be swapped, which is sometimes useful to eliminate code duplication. The `INNER-LOOP` attribute is C code, except that `output` is a pseudo-variable that will be replaced with an appropriate sample address calculation. Also, note that sounds `s1` and `s2` are simply referenced by name. The compiler will replace these references by expressions that reference the current values of `s1` and `s2`. The `LINEAR` attribute means that the function is linear with respect to scale factors on `s1` and `s2`. Rather that multiply each sample of `s1` by a scale factor (provided in the header of `s1`), the compiler will propagate the factor to the header of the result. (This is sometimes, but not always an optimization.) `TERMINATE` says that the computation goes to zero when either `s1` or `s2` goes to zero, and `LOGICAL-STOP` says the logical stop time of the result is the minimum of that of `s1` and `s2`.

The resulting inner loop, including machine-generated comments, is shown in Figure 7. The complete implementation, which includes additional functions to handle initialization, unaligned sounds, garbage collection, debugging, and structure declaration is 237 lines. The inner loop compiles to extremely fast code on the IBM RS/6000 computer running IBM's AIX Operating System. Note the somewhat odd combination of `if` and `while`. This was determined to generate faster code than a `for` loop, and it was relatively easy to modify the compiler to create the `if/while` combination. Other optimizations have been systematically applied in this fashion.

```
if (n) do { /* the inner sample computation loop */
    *out_ptr_reg++ = *s1_ptr_reg++ * *s2_ptr_reg++;
} while (--n); /* inner loop */
```

Figure 7. The inner loop generated from the specification in Figure 6.

Many more attributes exist, and the reader is referred to the implementation and the manual for more detail and further examples. This approach to specification is an excellent way to develop synthesis systems. It leads to reliable code and supports systematic design changes, optimizations, and ports.

Performance Evaluation

To evaluate Nyquist performance, I compared Nyquist performance to that of Csound, a popular software synthesis program. I used a 30MHz IBM RS/6000 Model 530 running AIX; all code was written in C and compiled with optimization. The benchmark is the generation of a sequence of 40 tones, each of which has 12 partials of constant frequency and piece-wise linear amplitude envelopes. The tones are sampled at 44100Hz and the total sound duration is 14.4 seconds. The sound samples are discarded as they are computed to avoid I/O, and I measure total real computation time.

Nyquist is surprisingly efficient. With a block size of 1024, Nyquist spends about 92% of its time in inner loops. Nyquist has nearly the performance of Csound as long as block sizes are large.

However, large block sizes in Csound produce audible distortion when control rate signals are used. Even when audio rate signals are used throughout, notes must always start on block boundaries. Thus, with a block size of 100, Csound at 44.1KHz quantizes times to about 2ms. Nyquist, on the other hand, uses length counts to allow partially filled blocks. Nyquist times are quantized to the audio sample rate, i.e. about 23μs, corresponding to a Csound block size of 1. If quantization is to be avoided, Nyquist is about 6 times faster than Csound. Figure 8 shows the performance of Csound *on this one benchmark* as a function of block size.

Instead of looking at extremes, let us consider typical parameters. A typical use of Csound might be to run with a control rate that is one tenth (0.1) of the audio rate in order to limit distortion. Nyquist can perform the same benchmark computation entirely at the audio rate and still be 20% faster. One might argue that the benchmark penalizes Csound by having only a few simple control-rate envelopes: with enough control rate signals, Csound would probably win out, but if Nyquist is also allowed to compute at control rates (a feature of Nyquist), Nyquist's advantage will actually widen because even Nyquist's control-rate signals are processed in blocks. A forthcoming article studies these sorts of trade-offs in detail (Dannenberg and Thompson, 1997).

Comparison with DSP’s

Comparison with DSP's is difficult due to the difference in program structure and functionality, but after adjusting for clock rate differences, the hand-microcoded Kyma system (Scaletti 1989, Scaletti and Hebel 1991) and NeXT sound kit (Jaffe and Boynton 1989) run our particular benchmark on a single Motorola DSP56001 at most 3 times faster than Nyquist running on an RS/6000. Since these measurements were made, the RS/6000 has been superceded by the PowerPC, available at 5 times the clock rate and one fifth the cost. Meanwhile, the M56001 has seen only moderate increases in clock rate; now, only multiprocessor configurations are faster than Nyquist. Other DSPs offer faster clock rates and floating point, but because DSP code is not very portable, the advantage of these newer DSPs is currently hypothetical.

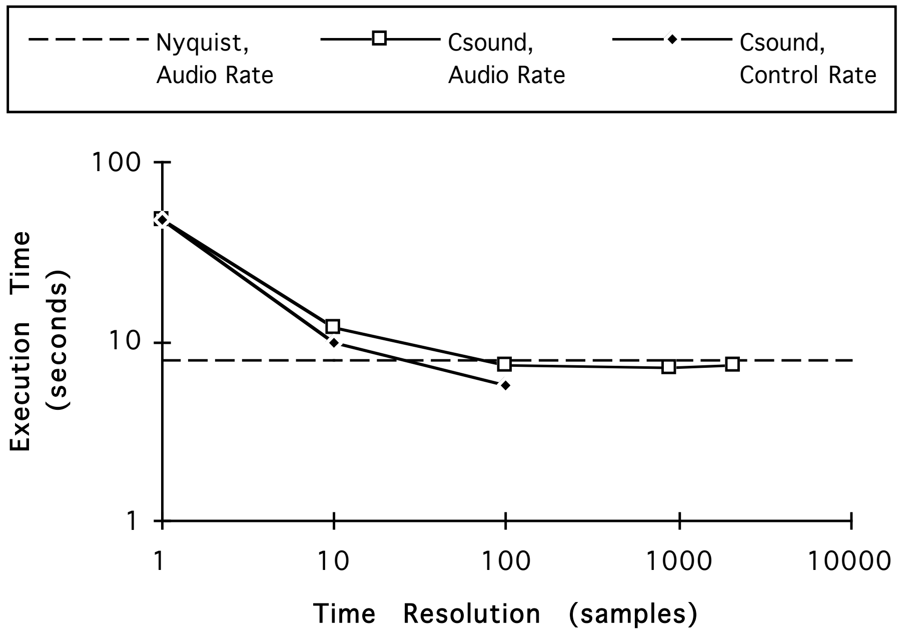

Figure 8. Benchmark execution times of Csound and Nyquist as a function of timing accuracy (in samples). Timing accuracy is equal to the block size in Csound, so performance improvement comes at the cost of timing accuracy. Nyquist always exhibits 1-sample timing accuracy, so performance is shown as a horizontal line.

CMJ Benchmarks

To evaluate Nyquist performance in a broader context, I ran Pope's (1993) benchmarks on a NeXT 68040 Cube. Since Csound was generally the fastest of the three systems originally benchmarked, I chose to normalize Nyquist performance to that of Csound. For each benchmark, Csound performance is represented by 1.0, and Nyquist performance is determined by dividing Csound run time by that of Nyquist. For example, in Benchmark 1, the Nyquist performance of 1.26 indicates that Nyquist was 1.26 times as fast as Csound. Figure 9 shows that overall, Nyquist performance is quite good. In Benchmark 7, the Csound envelope functions are non-interpolated functions sampled at 441 Hz, whereas Nyquist envelope functions are linearly interpolated up to 44.1 kHz. (Benchmark 8 is omitted because Nyquist does not have a non-interpolating FM oscillator.)

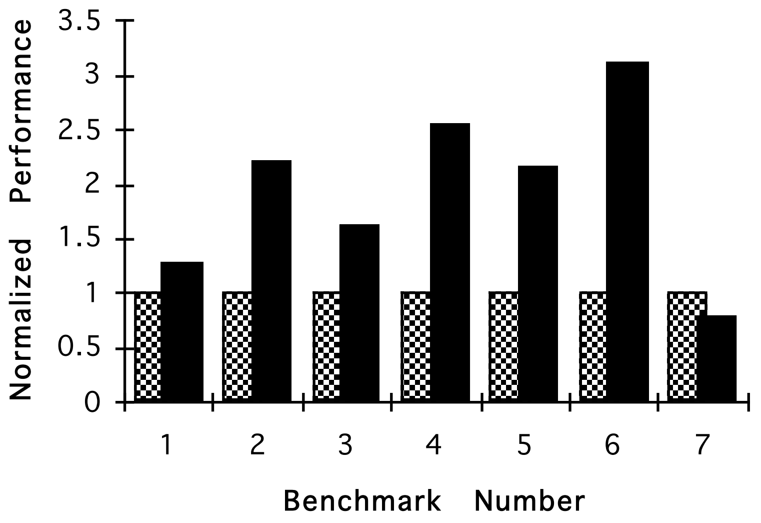

Figure 9. Nyquist performance (black) relative to that of Csound (gray). Higher bars mean faster execution. Y-axis units are normalized to the performance of Csound on each benchmark.

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, Nyquist 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.

In the future, I plan to modify Nyquist to provide better support for spectral analysis and synthesis and MIDI. The problem with spectral analysis is that it typically results in many channels at low sample rates. This would logically be implemented in Nyquist as a multi-channel sound, but each channel would have at least one block of samples. Since Nyquist operators try to use large blocks, these multi-channel sounds would take up a large amount of storage. Wasinee Rungsarityotin and I have started developing an alternate approach using sequences of Lisp arrays to represent analysis frames.

A MIDI sequence data type has been added to XLisp, and a new control construct based on `seq` has been added so that sequences can be synthesized using information from standard MIDI files. We are implementing operations that extract MIDI continuous controls from MIDI files and convert them to Nyquist signals for use as envelopes and gestural control.

Acknowledgements

Cliff Mercer worked on the design and implementation of the addition operator for Nyquist and Joe Newcomer assisted with the design and implementation of the Nyquist inner loop code generator. The author would like to thank the IBM Corporation for supporting some of the development of Nyquist through the Information Technology Center at Carnegie Mellon.

References

Betz, D. 1988. *XLISP: An Object-oriented Lisp, Version 2.0*. (program documentation).

Dannenberg, R. B. 1989. “The Canon Score Language.” *Computer Music Journal* 13(1): 47-56.

Dannenberg, R. B. 1997a. “Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis,” *Computer Music Journal*, 21(3):50-60.

Dannenberg, R. B. 1997b. “Abstract Time Warping of Compound Events and Signals,” *Computer Music Journal*, 21(3):61-70.

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

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

Dannenberg, R. B., and C. W. Mercer. 1992. “Real-Time Software Synthesis on Superscalar Architectures.” In *Proceedings of the 1992 International Computer Music Conference*. International Computer Music Association. pp. 174-177.

Dannenberg, R. B., and N. Thompson. 1997. “Real-Time Software Synthesis on Superscalar Architectures,” *Computer Music Journal*, 21(3):83-94

Jaffe, D., and L. Boynton. 1989. “An Overview of the Sound and Music Kit for the NeXT Computer.” *Computer Music Journal* 13(2): 48-55.

Lansky, Paul. 1987. *CMIX*. Princeton, New Jersey: Princeton University.

Lansky, Paul. 1990. “The Architecture and Musical Logic of Cmix.” In S. Arnold and G. Hair (editors): *ICMC Glasgow 1990 Proceedings*. San Francisco: International Computer Music Association. pp. 91-94.

Mathews, M. V. 1969. *The Technology of Computer Music.*. Cambridge, Massachusetts: MIT Press.

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

Pratt, T. 1975. *Programming Languages: design and implementation.* Englewood Cliffs: Prentice Hall.

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

Roads, C. 1991. "Asynchronous Granular Synthesis," in De Poli, Piccialli, and Roads, eds.,*Representations of Musical Signals.* Massachusetts: MIT Press, pp. 143-186.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2):23-38.

Scaletti, C. and Hebel, K. 1991. An Object-based Representation for Digital Audio Signals. In G. DePoli, A. Picialli, and C. Roads (Eds.): *Representations of Musical Signals*, Cambridge MA: M.I.T. Press, pp. 371-389.

Schorr, H. and W. Waite. 1967. "An Efficient and Machine Independent Procedure for Garbage Collection in Various List Structures." *Communications of the ACM* 10(8):501-506.

Vercoe, B. 1986. Csound: A Manual for the Audio Processing System and Supporting Programs. MIT Media Lab. Cambridge, Massachusetts: MIT.