Introduction to Computer Music

Week 6 Programming Techniques
Version 1, 25 Sep 2018

Roger B. Dannenberg

Topics Discussed: **Recursive Sound Sequences, Matching Durations, Control Functions, Stretchable Behaviors, Granular Synthesis**

1 Programming Techniques in Nyquist

One of the unusual things about Nyquist is the support for timing (temporary semantics), which is lacking in most programming languages. Nyquist also has signals as a fundamental data type. Here we introduce a number of different programming techniques related to timing and signals that are useful in Nyquist.

1.1 Recursive Sound Sequences

We first look at the idea of recursive sound sequences. The `seq` function delays evaluation of each behavior until the previous behavior is finished. There are other functions, including `timed-seq` that work in a similar way. This delayed evaluation, a form of *lazy evaluation*, is important in practice because if sound computation can be postponed until needed, Nyquist can avoid storing large signals in memory. Lazy evaluation has another benefit, in that it allows you to express infinite sounds recursively. Consider a *drum roll* as an example. We define a drum roll recursively as follows: Start with one drum stroke and follow it with a drum roll!

The `drum-roll()` function below is a direct implementation of this idea. `drum-roll()` builds up a drum roll with one stroke at a time, recursively, and returns an infinite drum roll sound sequence. Even Google doesn't have that much disk space, so to avoid the obvious problem of storing an infinite sound, we can multiply `drum-roll()` by an envelope. In `limited-drum-roll()`, we make a finite drum roll by multiplying `drum-roll()` by `const(1, 2)`. `const(1, 2)` is actually a unit generator that returns a constant value of 1 until the duration of 2, then it drops to 0. Here, multiplying a limited sound by an infinite sound gives us a finite computation and result. Note that the multiplication operator in Nyquist is quite smart. It knows that when multiplying by 0, the result is always 0; and when a sound reaches its stop time, it remains 0 forever, thus Nyquist can terminate the recursion at the sound stop time.

```
define function drum-stroke()
     return noise() * pwev(1, 0.05, 0.1)
define function drum-roll()
     return seq(drum-stroke(), drum-roll()) ; recursion!
define function limited-drum-roll()
     return const(1, 2) * drum-roll() ; duration=2
play limited-drum-roll()
```

1.2 Matching Durations

In Nyquist, sounds are considered to be functions of time, with no upper bound on time. Sounds *do* have a "stop time" after which the signal is considered to remain at zero forever. This means that you can easily combine sounds of different durations by mistake. For example, you can multiply a 1-second oscillator signal by a 2-second envelope, resulting in a 1-second signal that probably ends abruptly before the envelope goes to zero.

It is a "feature" of Nyquist that you can compose longer sounds from shorter sounds or multiply by short envelopes to isolate sections of longer sounds, but this leads to one of the most common errors in Nyquist: Combining sounds and controls with different durations by mistake.

Here is an example of this common error:

```
play pwl(0.5, 1, 10, 1, 13) *  ; 13-seconds duration
      osc(c4) ; nominally 1-second duration
     ; final result: sound stops at 1 second(!)
```

Remember that Nyquist sounds are immutable. Nyquist will not and cannot go back and recompute behaviors to get the "right" durations–how would it know? There are two basic approaches to make durations match. The first is to make everything have a nominal length of 1 and use the *stretch* operator to change durations:

```
(pwl(0.1, 1, 0.8, 1, 1) * osc(c4)) ~ 13
```

Here we have changed `pwl` to have a duration of 1 rather than 13. This is the default duration of `osc`, so they match. Note also the use of parentheses to ensure that the stretch factor applies to both `pwl` and `osc`.

The second method is to provide duration parameters everywhere:

```
pwl(0.5, 1, 10, 1, 13) * osc(c4, 13)
```

Here, we kept the original 13-second long `pwl` function, but we explicitly set the duration of `osc` to 13. If you provide duration parameters everywhere, you will often end up passing duration as a parameter, but that's not always a bad thing as it makes duration management more explicit.

1.3 Control Functions

Another useful technique in Nyquist is to carefully construct control functions. Synthesizing control is like synthesizing sound, and often control is even more important that waveforms and spectra in producing a musical result.

1.3.1 Smooth Transitions

Apply envelopes to almost everything. Even control functions can have control functions! A good example is vibrato. A "standard" vibrato function might look like `lfo(6) * 5`, but this would generate constant vibrato throughout a tone. Instead, consider `lfo(6) * 5 * pwl(0.3, 0, 0.5, 1, 0.9, 1, 1)` where the vibrato depth is controlled by an envelope. Initially, there is no vibrato, the vibrato starts to emerge at 0.3 and reaches the full depth at 0.5, and finally tapers rapidly from 0.9 to 1. Of course this might be stretched, so these numbers are relative to the whole duration. Thus, we not only use envelopes to get smooth transitions in amplitude at the beginnings and endings of notes, we can use envelopes to get smooth transitions in vibrato depth and other controls.

1.3.2 Composing Control Functions

The are a few main "workhorse" functions for control signals.

1. for periodic variation such as vibrato, the `lfo` function generates low-frequency sinusoidal oscillations. The default sample rate of `lfo` is 1/20 of the audio sample rate, so do not use this function for audio frequencies. If the frequency is not constant, the simplest alternative is `hzosc`, which allows its first argument to be a `SOUND` as well as a number.

2. for non-periodic but deterministic controls such as envelopes, `pwl` and the related family (including `pwlv` and `pwlev`) or the envelope function `env` are good choices.

3. for randomness, a good choice is `noise`. By itself, `noise` generates audio rate noise, which is not suitable for adding small random fluctuations to control signals. What we often use in this case is a random signal that ramps smoothly from one amplitude to the next, with a new amplitude every 100 ms or so. You can obtain this effect by making a very low sample rate `noise` signal. When this signal is added to or multiplied by a higher sample rate signal, the `noise` signal is linearly interpolated to match the rate of the other signal, thus achieving a smooth ramp between samples. The expression for this is `sound-srate-abs(10, noise())`, which uses the `sound-srate-abs` transform to change the environment for `noise` to a sample rate of 10 Hz. See Figure 1.

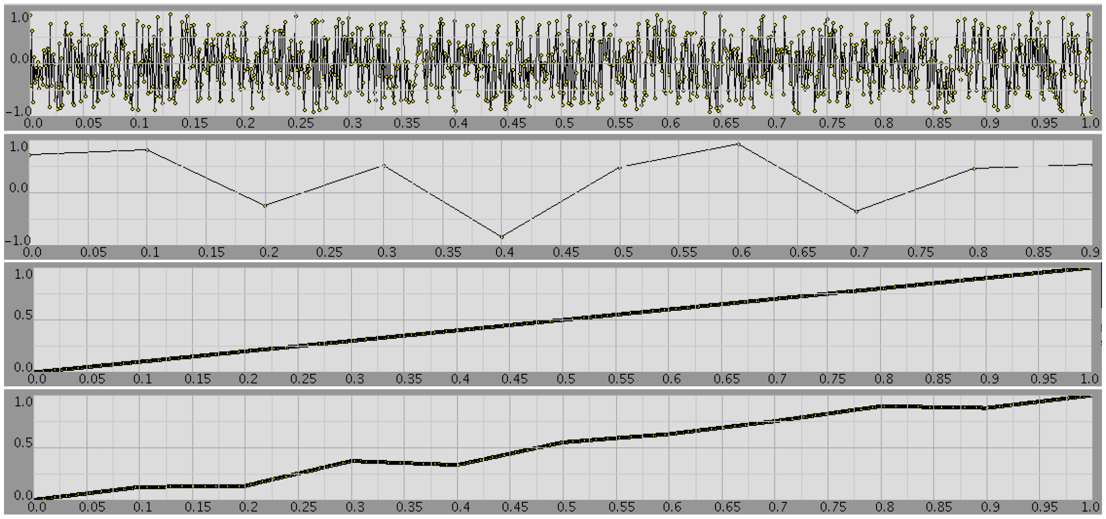


Figure 1: Plots of: `noise`, `sound-srate-abs(10, noise())`, `ramp()` and `sound-srate-abs(10, noise()) * 0.1 + ramp()`. The bottom plot shows how the `noise` function can be used to add jitter or randomness to an otherwise mathematically smooth control function. Such jitter or randomness occurs naturally, often due to minute natural human muscle tremors.

1.3.3 Global vs. Local Control Functions

Nyquist allows you to use control functions including envelopes at different levels of hierarchy. For example, you could synthesize a sequence of notes with individual amplitude envelopes, then multiply the whole sequence by an overarching envelope to give a sense of phrase. Figure 2 illustrates how global and "local" envelopes might be combined.

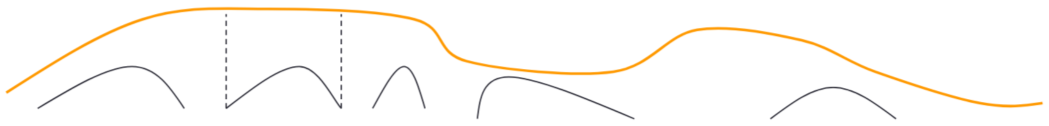


Figure 2: Hierarchical organization of envelopes.

In the simple case of envelopes, you can just apply the global envelope through multiplication after notes are synthesized. In some other cases, you might need to actually pass the global function as a parameter to be used for synthesis. Suppose that in Figure 2, the uppermost (global) envelope is supposed to control the index of modulation in FM synthesis. This effect cannot be applied after synthesis, so you must pass the global envelope as a parameter to each note. Within each note, you might expect the whole global envelope to somehow be shifted and stretched according to the note's start time and duration, but that does not happen. Instead, since the control function has already been computed as a SOUND, it is immutable and fixed in time. Thus, the note only "sees" the portion of the control function over the duration of the note, as indicated by the dotted lines in Figure 2.

In some cases, e.g. FM synthesis, the control function determines the note start time and duration, so fmosc might try to create a tone over the entire duration of the global envelope, depending on how you use it. One way to "slice" out a piece of a global envelope or control function according to the current environment is to use const(1) which forms a rectangular pulse that is 1 from the start time to the nominal duration according to the environment. If you multiply a global envelope parameter by const(1), you are sure to get something that is confined within the nominal starting and ending times in the environment.

1.4 Stretchable Behaviors

An important feature of Nyquist is the fact that functions represent *classes* of behaviors that can produce signals at different start times, with different durations, and be affected by many other parameters, both explicit and implicit.

Nyquist has default stretch behaviors for all of its primitives, and we have seen this many times. Often, the default behavior does the "right thing," (see the discussion above about matching durations). But sometimes you need to customize what it means to "stretch" a sound. For example, if you stretch a melody, you make notes longer, but if you stretch a drum roll, you do not make drum strokes slower. Instead, you add more drum strokes to fill the time. With Nyquist, you can create your own abstract behaviors to model things like drum rolls, constant-rate trills, and envelopes with constant attack times, none of which follow simple default rules of stretching.

Here is an example where you want the number of things to increase with duration:

```
define function n-things()
  begin
    with dur = get-duration(1),
         n = round(dur / *thing-duration*)
   return seqrep(i, n, thing() ~~ 1)
end
```

The basic idea here is to first "capture" the nominal duration using get-duration(1). Then, we compute n in terms of duration. Now, if we simply made n things in the current stretch environment (stretch by dur), we would create a sound with duration roughly n $\times$ *thing-duration* $\times$ dur, but we want to compute thing() without stretching. The ~~ operator, also called *absolute stretch*, resets the environment stretch factor to 1 when we call thing(), so now the total duration will be approximately n $\times$ *thing-duration*, which is about equal to get-duration(1) as desired.

Here is an example where you want an envelope to have a fixed rise time.

```
define function my-envelope()
  begin
    with dur = get-duration(1)
    return pwl(*rise-time*, 1, dur - *fall-time*, 1, dur) ~~ 1
  end
```

As in the previous example, we "capture" duration to the variable dur and then compute within an *absolute stretch* (~~) of 1 so that all duration control is explicit and in terms of dur. We set the rise time to a constant, *rise-time* and compute the beginning and ending of the "release" relative to dur which will be the absolute duration of the envelope.

1.5 Summary

We have seen a number of useful Nyquist programming techniques. To make sure durations match, either normalize durations to 1 and stretch everything as a group using the stretch operator (~), or avoid stretching altogether and use explicit duration parameters everywhere. In general, use envelopes everywhere to achieve smooth transitions, and never let an oscillator output start or stop without a smooth envelope. Control is all important, so in spite of many simplified examples you will encounter, every parameter should be a candidate for control over time: amplitude, modulation, vibrato, filter cutoff, and even control functions themselves can be modulated or varied at multiple time scales. And speaking of multiple time scales, do not forget that musical gestures consisting of multiple notes or sound events can be sculpted using over-arching envelopes or smooth control changes that span whole phrases.

2 Granular Synthesis

We know from our discussion of the Fourier transform that that complex sounds can be created by adding together a number of sine waves. Granular synthesis uses a similar idea, except that instead of a set of sine waves whose frequencies and amplitudes change over time, we use many thousands of very short (usually less than 100 milliseconds) overlapping sound bursts or grains. The waveforms of these grains are often sinusoidal, although any waveform can be used. (One alternative to sinusoidal waveforms is to use grains of sampled sounds, either pre-recorded or captured live.) By manipulating the temporal placement of large numbers of grains and their frequencies, amplitude envelopes, and waveshapes, very complex and time-variant sounds can be created.

2.1 Grains

To make a grain, we simply take any sound (e.g. a sinusoid or sound from a sound file) and apply a short smoothing envelope to avoid clicks. (See Figure 3.) The duration is typically from around 20ms to 200ms: long enough to convey a little content and some spectral information, but short enough to avoid containing an entire note or word (from speech sounds) or anything too recognizable.

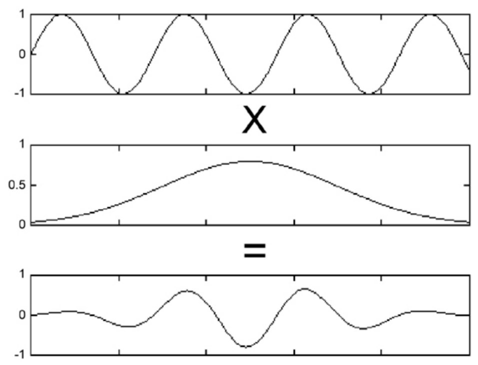


Figure 3: Applying a short envelope to a sound to make a grain for granular synthesis. How would a different amplitude envelope, say a square one, affect the shape of the grain? What would it do to the sound of the grain? What would happen if the sound was a recording of a natural sound instead of a sinusoid? What would be the effect of a longer or shorter envelope?

2.2 Clouds of Sound

Granular synthesis is often used to create what can be thought of as "sound clouds"—shifting regions of sound energy that seem to move around a sonic space. A number of composers, like Iannis Xenakis and Barry Truax, thought of granular synthesis as a way of shaping large masses of sound by using granulation techniques. These two composers are both considered pioneers of this technique (Truax wrote some of the first special-purpose software for granular synthesis). Sometimes, cloud terminology is even used to talk about ways of arranging grains into different sorts of configurations.

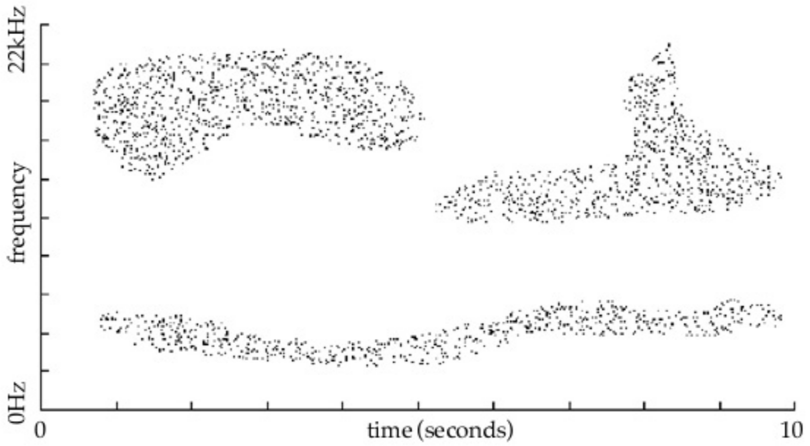


Figure 4: Visualization of a granular synthesis "score." Each dot represents a grain at a particular frequency and moment in time. An image such as this one can give us a good idea of how this score might sound, even though there is some important information left out (such as the grain amplitudes, waveforms, amplitude envelopes, and so on). What sorts of sounds does this image imply? If you had three vocal performers, one for each "cloud," how would you go about performing this piece? Try it!

2.3 Grain Selection, Control and Organization

In granular synthesis, we make and combine thousands of grains (sometimes thousands of grains per second), which is too much to do by hand, so we use computation to do the work, and we use high-level parameters to control things. Beyond this, granular synthesis is not a specific procedure and there is no right or wrong way to do it. It is good to be aware of the range of mechanisms by which grains are selected, processed, and organized.

One important control parameter is density. Typically granular synthesis is quite dense, with 10 or more grains overlapping at any given time. But grains can also be sparse, creating regular rhythms or isolated random sound events.

Stochastic or statistical control is common in granular synthesis. For example, we could pick grains from random locations in a file and play them at random times. An interesting technique is to scan through a source file, but rather than taking grains sequentially, we add a random offset to the source location, taking grains *in the neighborhood* of a location that progresses through the file. This produces changes over time that mirror what is in the file, but at any given time, the cloud of sound can be quite chaotic, disguising any specific audio content or sound events in the file.

It is also possible to resample the grain to produce pitch shifts. If pitch shifting is random, a single tone in the source can become a multi-pitch cluster or cloud in the output. If you resample grains, you can shift the pitch by octaves or harmonics, which might tend to harmonize the output sound when there is a single pitch on the input, or you can resample by random ratios, creating an atonal or microtonal effect. When you synthesize grains, you can use regular spacing, e.g. play a grain every 10 ms, which will tend to create a continuous sounding texture, or you can play grains with randomized inter-onset intervals, creating a more chaotic or bubbly sound.

Some interesting things to do with granular synthesis include vocal mumblings using grains to chop up speech and make speech-sounding nonsense, especially using grains with approximately the duration of phonemes so that whole words are obliterated. Granular synthesis can also be used for time stretching: By moving through a file very slowly, fetching overlapping grains and outputting them with less overlap, the file is apparently stretched, a shown in Figure 5. There will be artifacts because grains will not add up perfectly smoothly to form a continuous sound, but this can be a feature as well as a limitation, depending on your musical goals.

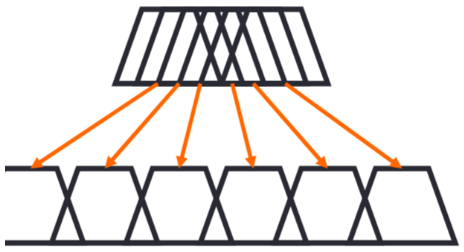


Figure 5: Stretching with granular synthesis.

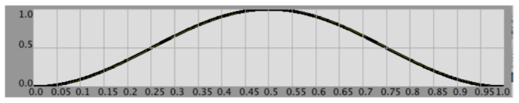
2.4 Granular Synthesis in Nyquist

Nyquist does not have a "granular synthesis" function because there are so many ways to implement and control granular synthesis. However, Nyquist is almost unique in its ability to express granular synthesis using a combination of signal processing and control constructs.

2.4.1 Generating a Grain

Figure 6 illustrates Nyquist code to create an smooth envelope and read a grain's worth of audio from a file to construct a smooth grain. Note the use of duration `d` both to stretch the envelope and control how many samples are read from the file. Below, we consider two approaches to granular synthesis implementation. The first uses scores and the second uses `seqrep`.

```
function cos-pulse()
       return 0.5 * (1 + hzosc(1 / get-duration(1),
                                *sine-table*, 270.0))
```



```
s-read("filename.wav", time-offset: seconds, dur: d) *
  (cos-pulse() ~ d)
```

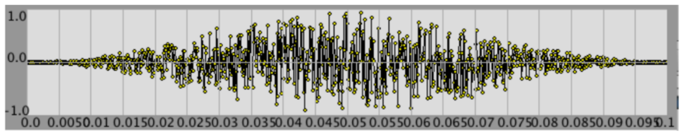


Figure 6: Contructing a grain in Nyquist.

2.5 Grains In Scores

A score for granular synthesis treats each grain as a sound event:

```
{{0 0.05 {grain offset: 2.1}}
 {0.02 0.06 {grain offset: 3.0}}
 ...}
```

The score calls upon `grain`, which we can define as follows. Notice that grain durations are specified in the score and implemented through the environment, so the `cos-pulse` signal will be stretched by the duration, but `s-read` is unaffected by stretching. Therefore, we must obtain the stretch factor using `get-duration(1)` and pass that value to `s-read` as the optional `dur:` keyword parameter:

```
function grain(offset: 0)
  begin with dur = get-duration(1)
    return s-read("filename.wav",
                  time-offset: offset, dur: dur) *
           cos-pulse()
```

Now, we can make make a score with `score-gen`. In the following expression, we construct 2000 grains with randomized inter-onset intervals and using pattern objects to compute the grain durations and file offsets:

```
score-gen(score-len: 2000,
          ioi: 0.05 + rrandom() * 0.01,
          dur: next(dur-pat),
          offset: next(offset-pat))
```

You could also use more randomness to compute parameters, e.g. the duration could come from a Gaussian distribution (see the Nyquist function `gaussian-dist`), and `offset:` could be computed by slowly moving through the file and adding a random jitter, e.g. `max(0, sg-count * 0.01 + rrandom() * 0.2)`.

2.6 Grains With Seqrep

Rather than computing large scores, we can use Nyquist control constructs to creates grains "on the fly." In the following example, `seqrep` is used to create and sum 2000 sounds. Each sound is produced by a call to `grain`, which is stretched by values from `dur-pat`. To obtain grain overlap, we use `set-logical-stop` with an IOI (logical stop time) parameter of `0.05 + rrandom() * 0.01`, so the grain IOI will be 50 ms $\pm$ 10 ms:

```
seqrep(i, 2000,
       set-logical-stop(
           grain(offset: next(offset-pat)) ~
               next(dur-pat),
           0.05 + rrandom() * 0.01))
```

2.7 The gran Extension

For another example, you can install the `gran` extension using Nyquist's Extension Manager. The package includes a function `sf-granulate` that implements granular synthesis using a sound file as input.

2.8 Other Ideas

You might want to implement something like the tendency masks we described earlier or use a `pwl` function to describe how some granular synthesis parameters evolve over time. Since `pwl` produces a signal and you often need *numbers* to control each grain, you can use `sref` to evaluate the signal at a specific time, e.g. `s-ref(sound, time)` will evaluate `sound` at `time`.

Another interesting idea is to base granular synthesis parameters on the source sound itself. A particularly effective technique is to select grains by slowly advancing through a file, causing a time-expansion of the file content. The most interesting portions of the file are usually note onsets and places where things are changing rapidly, which you might be able to detect by measuring either amplitude or spectral centroid. Thus, if the rate at which you advanced through the file can be inversely proportional to amplitude or spectral centroid, then the "interesting" material will be time-expanded, and you will pass over the "boring" steady-state portions of the signal quickly.

2.9 Summary

Granular synthesis creates sound by summing thousands of sound particles or grains with short durations to form clouds of sounds. Granular synthesis can construct a wide range of textures, and rich timbres can be created by taking grains from recordings of natural sounds. Granular synthesis offers many choices of details including grain duration, density, random or deterministic timing, pitch shifts and source sounds.

3 Acknowledgments

Thanks to Shuqi Dai and Sai Samarth for editing assistance.

Portions of this work are taken almost verbatim from *Music and Computers, A Theoretical and Historical Approach* (http://sites.music.columbia.edu/cmc/MusicAndComputers/) by Phil Burk, Larry Polansky, Douglas Repetto, Mary Robert, and Dan Rockmore. Other portions are taken almost verbatim from *Introduction to Computer Music: Volume One* (http://www.indiana.edu/~emusic/etext/toc.shtml) by Jeffrey Hass. I would like to thank these authors for generously sharing their work and knowledge.