Introduction to Computer Music

Week 4 FM Synthesis

Version 1, 17 Sep 2018

Roger B. Dannenberg

Topics Discussed: **Frequency Modulation, FM Examples, Behavioral Abstraction, seq Function, sim Function, Logical Stop Time**

1 Introduction to Frequency Modulation

Frequency modulation occurs naturally. It is rare to hear a completely stable steady pitch. Some percussion instruments are counter-examples, e.g. a piano tone or a tuning fork or a bell, but most instruments and the human voice tend to have at least some natural frequency change.

1.1 Examples

Common forms of frequency variation in music include:

- Voice inflection, natural jitter, and vibrato in singing.
- Vibrato in instruments. Vibrato is a small and quasi-periodic variation of pitch, often around 6 Hz and spanning a fraction of a semitone.
- Instrumental effects, e.g. electric guitars sometimes have “whammy bars” that stretch or relax all the strings to change their pitch. Guitarists can “bend” strings by pushing the string sideways across the fretboard.
- Many tones begin low and come up to pitch.
- Loose vibrating strings go sharp (higher pitch) as they get louder. Loose plucked strings get flatter (lower pitch) especially during the initial decay.
- The slide trombone, Theremin, voice, violin, etc. create melodies by changing pitch, so melodies on these instruments can be thought of as examples of frequency modulation (as opposed to, say, pianos, where melodies are created by selecting from fixed pitches).

1.2 FM Synthesis

Normally, vibrato is a slow variation created by musicians’ muscles. With electronics, we can increase the vibrato rate into the audio range, where interesting things happen to the spectrum. The effect is so dramatic, we no longer refer to it as vibrato but instead call it *FM Synthesis*. See the reading “FM Synthesis” for a detailed description of this important technique.

2 Frequency Modulation with Nyquist

From this section on, we assume you have read and studied “FM Synthesis.” Now that you have a grasp of the theory, let’s use it to make some sound.

2.1 Basic FM

The basic FM oscillator in Nyquist is the function `fmosc`. The signature for `fmosc` is

> `fmosc(`*basic-pitch*, *fm-control* [, *table* [, *phase*]]`)`.

Let's break that down:

- *basic-pitch* is the carrier frequency, expressed in steps. If there is no modulation, this controls the output frequency. Remember this is in steps, so if you put in a number like 440 expecting Hz, you will actually get `step-to-hz(440)`, which is about 0.9 *terahertz*[1]! Use `A4` instead, or write something like `hz-to-step(441.0)` if you want to specify frequency in Hertz.
- *fm-control* is the modulator. The *amplitude* of the modulator controls the *depth* of modulation—how much does the frequency deviate from the nominal carrier frequency based on *basic-pitch*? An amplitude of 1 (the nominal output of `(osc)` for example), gives a frequency deviation of ± 1. If you studied "FM Synthesis," you will realize that gives a *very* low index of modulation—you probably will not hear anything but a sine tone, so typically,
  - you will scale *fm-control* by a large number, and
  - you will also scale *fm-control* by some kind of envelope–by varying the depth of modulation, you will vary the spectrum, which is typical in FM synthesis.

  The *frequency* of *fm-control* is just the modulation frequency. It is the same as the carrier frequency if the C:M ratio is 1:1. This is where you control the C:M ratio.
- *table* is optional and allows you to replace the default sine waveform with any waveform table. The table data structure is exactly the same one you learned about earlier for use with the `osc` function.
- *phase* is also optional and gives the initial phase. Generally, changing the initial phase will not cause any perceptible changes.

2.2 FM Example in Nyquist

The following plays a characteristic FM sound in Nyquist:

```
play fmosc(c4, pwl(1, 4000, 1) * osc(c4)) ~ 10
```

Since the modulation comes from `osc(c4)`, the modulation frequency matches the carrier frequency (given by `fmosc(c4, ...)`, so the C:M ratio is 1:1. The *amplitude* of modulation ramps from 0 to 4000, giving an index of modulation of $I = 4000/steptohz(C4) = 15.289$. Thus, the spectrum will evolve from a sinusoid to a rich spectrum with the carrier and around $I+1$ sidebands, or about 17 harmonics.

3 Behavioral Abstraction

In Nyquist, all functions are subject to transformations. You can think of transformations as additional parameters to every function, and functions are free to use these additional parameters in any way. The set of transformation parameters is captured in what is referred to as the transformation environment. (Note that the term *environment* is heavily overloaded in computer science. This is yet another usage of the term.)

Behavioral abstraction is the ability of functions to adapt their behavior to the transformation environment. This environment may contain certain abstract notions, such as loudness, stretching a sound in time, etc. These notions will mean different things to different functions. For example, an oscillator should produce more periods of oscillation in order to stretch its output. An envelope, on the other hand, might only change the duration of

[1] Need we point out this is somewhat higher than the Nyquist rate?

the sustain portion of the envelope in order to stretch. Stretching a sample could mean resampling it to change its duration by the appropriate amount.

Thus, transformations in Nyquist are not simply operations on signals. For example, if I want to stretch a note, it does not make sense to compute the note first and then stretch the signal. Doing so would cause a drop in the pitch. Instead, a transformation modifies the transformation environment in which the note is computed. Think of transformations as making requests to functions. It is up to the function to carry out the request. Since the function is always in complete control, it is possible to perform transformations with "intelligence;" that is, the function can perform an appropriate transformation, such as maintaining the desired pitch and stretching only the "sustain" portion of an envelope to obtain a longer note.

3.1 Behaviors

Nyquist sound expressions denote a whole class of behaviors. The specific sound computed by the expression depends upon the environment. There are a number of transformations, such as `stretch` and `transpose` that alter the environment and hence the behavior.

The most common transformations are `shift` and `stretch`, but remember that these do not necessarily denote simple time shifts or linear stretching: When you play a longer note, you don't simply stretch the signal! The behavior concept is critical for music.

3.2 Evaluation Environment

To implement the behavior concept, all Nyquist expressions evaluate within an *environment*.

The transformation environment is implemented in a simple manner. The environment is simply a set of special global variables. These variables should not be read directly and should never be set directly by the programmer. Instead, there are functions to read them, and they are automatically set and restored by transformation operators, which will be described below. The Nyquist environment includes: starting time, stretch factor, transposition, legato factor, loudness, sample rates, and more.

3.3 Manipulating the Environment Example

A very simple example of a transformation affecting a behavior uses the stretch operator (~):

```
pluck(c4) ~ 5
```

This example can be read as "evaluate `pluck` in an environment that has been stretched by 5 relative to the current environment." Operationally, what happens is that the stretch factor in the environment is multiplied by 5. Then the expression `pluck(c4)` is evaluated. The `pluck` function computes a sound that depends on the duration that in turn depends on the stretch factor in the environment, so in this case, a 5 second sound is computed. Finally, the stretch factor in the environment is restored.

3.4 Transformations

In the previous example, we saw that many transformations are relative and can be nested. Stretch does not just set the stretch factor; instead, it *multiplies* the stretch factor by a factor, so the final stretch factor in the new environment is relative to the current one.

Nyquist also has "absolute" transformations that override any existing value in the environment. For example,

```
function tone2() return osc(c4) ~~ 2
```

returns a 2 second tone, even if you write:

```
play tone2() ~ 100 ; 2 second tone
```

because the "absolute stretch" (~~) overrides the stretch operator (~). Even though tone2 is called with a stretch factor of 100, its absolute stretch transformation overrides the environment and sets it to 2.

Also, note that once a sound is computed, it is immutable. The following use of a global variable to store a sound is not recommended, but serves to illustrate the immutability aspect of sounds:

```
mysnd = osc(c4) ; compute sound, duration = 1
play mysnd ~ 2  ; plays duration of 1
```

The stretch factor of 2 in the second line has no effect because mysnd evaluates to an immutable sound. Transformations only apply to functions (which we sometimes call *behaviors*). Transformations do not affect sounds once they are computed.

Transformations are described in detail in the Nyquist Reference Manual (find "Transformations" in the index). In practice, the most critical transformations are at (@) and stretch (~), which control when sounds are computed and how long they are.

4 Sequential Behavior (seq)

Consider the simple expression:

```
play seq(my-note(c4, q), my-note(d4, i))
```

The idea is to create the first note at time 0, and to start the next note when the first one finishes. This is all accomplished by manipulating the environment. In particular, *warp* is modified so that what is locally time 0 for the second note is transformed, or warped, to the logical stop time of the first note.

One way to understand this in detail is to imagine how it might be executed: first, *warp* is set to an initial value that has no effect on time, and my-note(c4, q) is evaluated. A sound is returned and saved. The sound has an ending time, which in this case will be 1.0 because the duration q is 1.0. This ending time, 1.0, is used to construct a new *warp* that has the effect of shifting time by 1.0. The second note is evaluated, and will start at time 1.0. The sound that is returned is now added to the first sound to form a composite sound, whose duration will be 2.0. Finally, *warp* is restored to its initial value.

Notice that the semantics of seq can be expressed in terms of transformations. To generalize, the operational rule for seq is: evaluate the first behavior according to the current *warp*. Evaluate each successive behavior with *warp* modified to shift the new note's starting time to the ending time of the previous behavior. Restore *warp* to its original value and return a sound which is the sum of the results.

In the Nyquist implementation, audio samples are only computed when they are needed, and the second part of the seq is not evaluated until the ending time (called the logical stop time) of the first part. It is still the case that when the second part is evaluated, it will see *warp* bound to the ending time of the first part.

A language detail: Even though Nyquist defers evaluation of the second part of the seq, the expression can reference variables according to ordinary Lisp/SAL scope rules. This is because the seq captures the expression in a *closure*, which retains all of the variable bindings.

5 Simultaneous Behavior (sim)

Another operator is sim, which invokes multiple behaviors at the same time. For example,

```
play 0.5 * sim(my-note(c4, q), my-note(d4, i))
```

will play both notes starting at the same time.

The operational rule for sim is: evaluate each behavior at the current *warp* and return the sum of the results. (In SAL, the sim function applied to sounds is equivalent to adding them with the infix + operator.

6 Logical Stop Time

We have seen that the default behavior of `seq` is to cause sounds to begin at the end of the previous sound in the sequence. What if we want to play a sequence of short sounds separated by silence? One possibility is to insert silence (see `s-rest`), but what if we want to play equally spaced short sounds? We have to know how long each short sound lasts in order to know how much silence to insert. Or what if sounds have long decays and we want to space them equally, requiring some overlap?

All of these cases point to an important music concept: We pay special attention to the beginnings of sounds, and the time interval between these onset times (often called *IOI* for "Inter-Onset Interval") is usually more important than the specific duration of the sound (or note). Thus, spacing notes according to their duration is not really a very musical idea.

Instead, we can try to separate the concepts of duration and IOI. In music notation, we can write a quarter note, meaning "play this sound for one beat," but put a dot over it, meaning "play this sound short, but it still takes a total of one beat." (See Figure 1.) It is not too surprising that music notation and theory would have rather sophisticated concepts regarding the organization of sound in time.

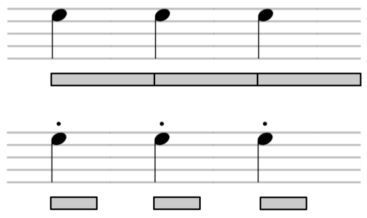

Figure 1: Standard quarter notes nominally fill an entire beat. Staccato quarter notes, indicated by the dots, are played shorter, but they still occupy a time interval of one beat. The "on" and "off" time of these short passages is indicated graphically below each staff of notation.

Nyquist incorporates this musical thinking into its representation of sounds. A sound in Nyquist has one start time, but there are effectively two "stop" times. The *signal* stop is when the sound samples end. After that point, the sound is considered to continue forever with value of zero. The *logical* stop marks the "musical" or "rhythmic" end of the sound. If there is a sound after it (e.g. in a sequence computed by `seq`), the next sound begins at this *logical stop* time of the sound.

The logical stop is usually the signal stop by default, but you can change it with the `set-logical-stop` function. For example, the following will play a sequence of 3 plucked-string sounds with durations of 2 seconds each, but the IOI, that is, the time between notes, will be only 1 second.

```
play seq(set-logical-stop(pluck(c4) ~ 2, 1)
         set-logical-stop(pluck(c4) ~ 2, 1),
         set-logical-stop(pluck(c4) ~ 2, 1))
```

7 Scores in Nyquist

Scores in Nyquist indicate "sound events," which consist of functions to call and parameters to pass to them. For each sound event there is also a start time and a "duration," which is really a Nyquist stretch factor. When you render a score into a sound (usually by calling `timed-seq` or executing the function `sound-play`), Each sound event is evaluated by adjusting the environment according to the time and duration as if you wrote

sound-event(*parameters*) ~ *duration* @ *time*

The resulting sounds are then summed to form a single sound.

8 Summary

In this unit, we covered frequency modulation and FM synthesis. Frequency modulation refers to anything that changes pitch within a sound. Frequency modulation at audio rates can give rise to many partials, making FM Synthesis a practical, efficient, and versatile method of generating complex evolving sounds with a few simple control parameters.

All Nyquist sounds are computed by *behaviors* in an environment that can be modified with *transformations*. Functions describe behaviors. Each behavior can have explicit parameters as well as implicit environment values, so behaviors represent classes of sounds (e.g. piano tones), and each instance of a behavior can be different. Being "different" includes having different start times and durations, and some special constructs in Nyquist, such as `sim`, `seq`, `at` (`@`) and `shift` (`~`) use the environment to organize sounds in time. These concepts are also used to implement *scores* in Nyquist.

9 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/~eemusic/etext/toc.shtml`) by Jeffrey Hass. I would like to thank these authors for generously sharing their work and knowledge.