Introduction to Computer Music

Week 4

Instructor: Prof. Roger B. Dannenberg

Topics Discussed: Frequency Modulation, Behaviors and Transformations in Nyquist

Chapter 4

Frequency Modulation and Behaviors

Topics Discussed: Frequency Modulation, Behaviors and Transformations in Nyquist

Frequency modulation (FM) is a versatile and efficient synthesis technique that is widely implemented in software and hardware. By understanding how FM works, you will be able to design parameter settings and control strategies for your own FM instrument designs.

4.1 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.

4.1.1 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 (accompanied by amplitude and spectral variations), 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).

4.1.2 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*.

Frequency modulation (FM) is a synthesis technique based on the simple idea of periodic modulation of the signal frequency. That is, the frequency of a carrier sinusoid is modulated by a modulator sinusoid. The peak frequency deviation, also known as *depth of modulation*, expresses the strength of the modulator's effect on the carrier oscillator's frequency.

FM synthesis was invented by John Chowning [**?**] and became very popular due to its ease of implementation and computationally low cost, as well as its (somewhat surprisingly) powerful ability to create realistic and interesting sounds.

4.2 Theory of FM

Let's look at the equation for a simple frequency controlled sine oscillator. Often, this is written

$$y(t) = A\sin(2\pi\phi t) \tag{4.1}$$

where ϕ (phi) is the frequency in Hz. Note that if ϕ is in in Hz (cycles per second), the frequency in radians per second is $2\pi\phi$. At time t, the phase will have advanced from 0 to $2\pi\phi t$, which is the integral of $2\pi\phi$ over a time span of t. We can express what we are doing in detail as:

$$y(t) = A\sin(2\pi\int_0^t \phi dx) \tag{4.2}$$

To deal with a time-varying frequency modulation, we will substitute frequency *function* $f(\cdot)$ for ϕ:

$$y(t) = A\sin(2\pi\int_0^t f(x)dx) \tag{4.3}$$

Frequency modulation uses a rapidly changing function

$$f(t) = C + D\sin(2\pi Mt) \tag{4.4}$$

where C is the carrier, a frequency offset that is in many cases is the fundamental or "pitch". D is the depth of modulation that controls the amount of frequency deviation (called modulation), and M is the frequency of modulation in Hz. Plugging this into Equation 4.3 and simplifying gives the equation for FM:

$$y(t) = A\sin(2\pi\int_0^t C + D\sin(2\pi Mx)dx) \tag{4.5}$$

Note that the integral of sin is cos, but the only difference between the two is phase. By convention, we ignore this detail and simplify Equation 4.5 to get this equation for an FM oscillator:

$$f(t) = A\sin(2\pi Ct + I\sin(2\pi Mt)) \tag{4.6}$$

$I = D/M$ is known as the index of modulation. When $D \neq 0$, sidebands appear in the spectra of the signal above and below the carrier frequency C, at multiples of $\pm M$. In other words, we can write the set of frequency components as $C \pm kM$, where k=0,1,2,.... The number of significant components increases with I, the index of modulation.

4.2.1 Negative Frequencies

According to these formulas, some frequencies will be negative. This can be interpreted as merely a phase change: $\sin(-x) = -\sin(x)$ or perhaps not even a phase change: $\cos(-x) = \cos(x)$. Since we tend to ignore phase, we can just ignore the sign of the frequency and consider negative frequencies to be positive. We sometimes say the negative frequencies "wrap around" (zero) to become positive. The main caveat here is that when frequencies wrap around and add to positive frequencies of the same magnitude, the components may not add in phase. The effect is to give FM signals a complex evolution as the index of modulation increases and adds more and more components, both positive and negative.

4.2.2 Harmonic Ratio

The human ear is very sensitive to harmonic vs. inharmonic spectra. Perceptually, harmonic spectra are very distinctive because they give a strong sense of pitch. The harmonic ratio [**?**] is the ratio of the modulating frequency to the carrier frequency, such that $H = M/C$. If H is a rational number, the spectrum is harmonic; if it is irrational, the spectrum is inharmonic.

4.2.3 Rational Harmonic Ratio

If $H = 1$ the spectrum is harmonic and the carrier frequency is also the fundamental, i.e. $F_0 = C$. To show this, remember that the frequencies will be $C \pm kM$, where k=0,1,2,..., but if $H = 1$, then $M = C$, so the frequencies are $C \pm kC$, or simply kC. This is the definition of a harmonic series: multiples of some fundamental frequency C.

When $H = 1/m$, and m is a positive integer, C instead becomes the m^{th} component (harmonic) because the spacing between harmonics is $M = C/m$, which is also the fundamental: $F_0 = M = C/m$.

With $H = 2$, we will get sidebands at $C \pm 2kC$ (where k=0,1,2,...), thus omitting all even harmonics — which is a characteristic of cylindrical woodwinds such as the clarinet.

4.2.4 Irrational Harmonic Ratio

If H is irrational, the negative frequencies that wrap around at 0 Hz tend to land between the positive frequency components, thus making the spectrum denser. If we make H much less than 1 (M much less than C), and if H is irrational (or at least not a simple fraction such as $1/n$), the FM-generated frequencies will cluster around C (the spacing between components will be relatively small); yielding sounds that have no distinct pitch and that can mimic drums and gongs.

4.2.5 FM Spectra and Bessel Functions

The sidebands infused by FM are governed by Bessel functions of the first kind and n^{th} order; denoted $J_n(I)$, where I is the index of modulation. The Bessel functions determine the magnitudes and signs of the frequency components in the FM spectrum. These functions look a lot like damped sine waves, as can be seen in Figure 4.1.

Here are a few insights that help explain why FM synthesis sounds the way it does:

- $J_0(I)$ gives the amplitude of the carrier.

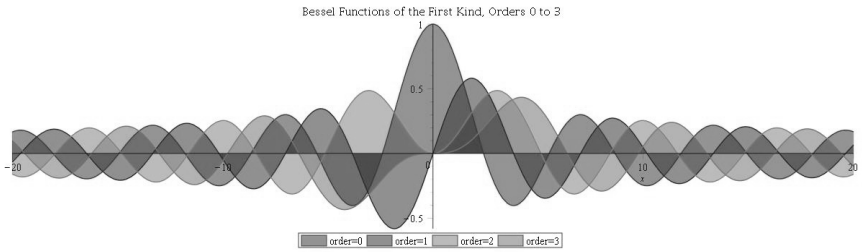

Figure 4.1: Bessel functions of the first kind, orders 0 to 3. The x axis represents the index of modulation in FM.

- $J_1(I)$ gives the amplitude of the first upper and lower sidebands.
- Generally, $J_n(I)$ governs the amplitudes of the n^{th} upper and lower sidebands.
- Higher-order Bessel functions start from zero and increase more and more gradually, so higher-order sidebands only have significant energy when I is large.
- The spectral bandwidth increases with I. The upper and lower sidebands represent the higher and lower frequencies, respectively. The larger the value of I, the more significant sidebands you get.
- As I increases, the energy of the sidebands vary much like a damped sinusoid.

4.2.6 Index of Modulation

The index of modulation, $I = D/M$, allows us to relate the depth of modulation, D, the modulation frequency, M, and the index of the Bessel functions. In practice, this means that if we want a spectrum that has the energy of the Bessel functions at some index I, with frequency components separated by M, then we must choose the depth of modulation according to the relation $I = D/M$ [**?**]. As a rule-of-thumb , the number of sidebands is roughly equivalent to $I + 1$. That is, if $I = 10$ we get $10 + 1 = 11$ sidebands above, and 11 sidebands below the carrier frequency. In theory, there are infinitely many sidebands at $C \pm kM$, where k=0,1,2,... if the modulation is non-zero, but the intensity of sidebands falls rapidly toward zero as k increases, but this rule of thumb considers only significant sidebands. (Note that this formula fails at $I = 0$: it predicts the carrier plus 1 side band above and one below, but there are no side bands in this case.)

4.2.7 Time-Varying Parameters

For music applications, constant values of A, C, D, and M would result in a rock-steady tone of little interest. In practice, we usually vary all of these parameter, replacing them with $A(t)$, $C(t)$, $D(t)$, and $M(t)$. *Usually*, these parameters change slowly compared to carrier and modulator frequencies, so we assume that all the analysis here still applies and allows us to make predictions about the short-time spectrum. FM is particularly interesting because we can make very interesting changes to the spectrum with just a few control parameters, primarily $D(t)$, and $M(t)$.

4.2.8 Examples of FM Signals

Figures 4.2 and 4.3 show examples of FM signals. The X-axes on the plots represent time — here denoted in multiples of π.

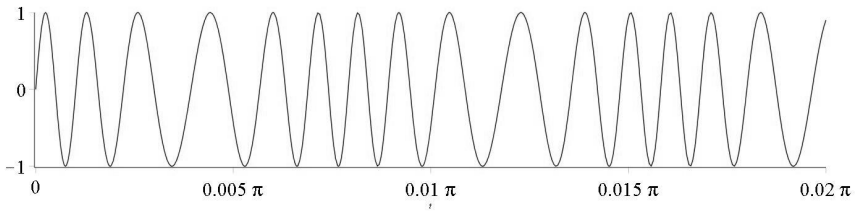

Figure 4.2: A = 1, C = 242 Hz, D = 2, M = 40 Hz.

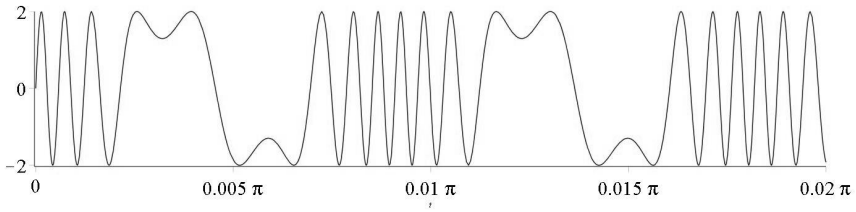

Figure 4.3: A = 2, C = 210 Hz, D = 10, M = 35 Hz.

4.3 Frequency Modulation with Nyquist

Now that you have a grasp of the theory, let's use it to make some sound.

4.3.1 Basic FM

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

`fmosc(`*basic-pitch*`,` *modulation* `[,` *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.
- *modulation* 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 have

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

been paying attention, you will realize that 1 gives a *very* low index of modulation ($1/m$)—you probably will not hear anything but a sine tone. (Note that Nyquist's implementation is close to Equation 4.5—Nyquist performs the integral, so the scale factor really is the *depth D*.) Typically,

  - you will use a large number for modulation, and
  - you will also scale *modulation* 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 *modulation* is just the modulation frequency. It can be the same as the carrier frequency if you want the C:M ratio to be is 1:1. This parameter 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.

4.3.2 Index of Modulation Example

Produce a harmonic sound with about 10 harmonics and a fundamental of 100 Hz. We can choose $C = M = 100$, which gives $C : M = 1$ and is at least one way to get a harmonic spectrum. Since the number of harmonics is 10 we need the carrier plus 9 sidebands, and so $I + 1 = 9$ or $I = 8$. $I = D/M$ so $D = IM$, $D = 8 \times 100 = 800$. Finally, we can write

```
fmosc(hz-to-step(100), 800 * hzosc(100)).
```

4.3.3 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 a final 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. Try it!

For a musical tone, you should multiply the `fmosc` signal by an envelope. Otherwise, the output amplitude will be constant. Also, you should replace the `pwl` function with an envelope that increases and then decreases, at least if you want the sound to get brighter in the middle.

4.4 Behavioral Abstraction

We now continue our introduction to fundamental concepts and structures in Nyquist. Think about the concept of "piano note." Piano notes can have a wide range of pitches, loudnesses and durations, yet it still makes sense to think of "piano note" as a concept that describes the set of all of these instances. It also makes sense to

talk about transformations such as longer, higher or louder notes. In Nyquist, we use *behaviors* to describe classes of sounds, gestures, and even sequences of sub-behaviors (such as melodies), and we use ordinary functions (in the programming language sense) to model behaviors.

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 the sustain portion of the envelope in order to stretch. Stretching recorded audio 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.

4.4.1 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.

4.4.2 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.

4.4.3 Manipulating the Environment Example

A very simple example of a transformation affecting a behavior is 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."

4.4.4 Transformations

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[2] (which we often call *behaviors* to emphasize they behave differently in different contexts) and expressions that call on functions to compute sounds. Transformations do not affect sounds once they are computed.

An Operational View

You can think about this operationally: When Nyquist evaluates *expr* ~ 3, the ~ operator is not a function in the sense that expressions on the left and right are evaluated and passed to a function where "stretching" takes place. Instead, it is better to think of ~ as a control construct that:

- changes the environment by tripling the stretch factor
- evaluates *expr* in this new environment
- restores the environment
- returns the sound computed by *expr*

[2] This is a bit of a simplification. If a function merely returns a pre-computed value of type SOUND, transformations will have no effect.

Thus, if *expr* is a *behavior* that computes a sound, the computation will be affected by the environment. If *expr* is a *sound* (or a variable that stores a sound), the sound is already computed and evaluation merely returns the sound with no transformation.

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.

4.5 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.

4.6 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.

4.7 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 4.4.) It is not too surprising that music notation and theory would have rather sophisticated concepts regarding the organization of sound in time.

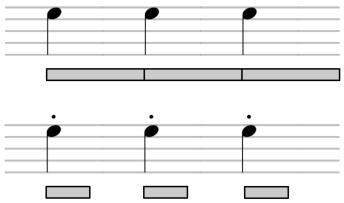

Figure 4.4: Standard quarter notes nominally fill an entire beat. Staccato quarter notes, indicated by the dots, are played shorter, but they still occupy a time interval of one beat. The "on" and "off" time of these short passages is indicated graphically below each staff of notation.

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))
```

4.8 Sequential and Simultaneous Iteration

Just as procedural languages have constructs such as `for` loops that iterate blocks of code, usually with a loop *index* variable that takes on values of 0, 1, 2, ... $n-1$,

Nyquist has special forms to instantiate *n* behaviors simultaneously or sequentially. Try the following two examples to play a chord and an *arpeggio* using `simrep` and `seqrep`:

```
play simrep(i, 6, piano-note(3, c4 + i * 5, 100))
play seqrep(i, 6, piano-note(0.3, c4 + i * 5, 100))
```

In these little programs, the variable `i` takes on values 0, 1, 2, 3, 4 and 5, producing pitches 60, 65, 70, 75, 80 and 85. (Note that if we name the loop variable `i`, it will "hide" the global binding of `i` which represents the `e`Ighth note duration of 0.5.) See the *Nyquist Reference Manual* for more about `simrep` and `seqrep`.

4.9 Scores in Nyquist

Scores in Nyquist (introduced in Section **??**) 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.

4.10 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.