Introduction to Computer Music

Week 5

Instructor: Prof. Roger B. Dannenberg

Topics Discussed: Short Time Fourier Transform, Spectral Centroid, Patterns, Algorithmic Composition

Chapter 5

Spectral Analysis and Nyquist Patterns

Topics Discussed: Short Time Fourier Transform, Spectral Centroid, Patterns, Algorithmic Composition

In this chapter, we dive into the Fast Fourier Transform for spectral analysis and use it to compute the Spectral Centroid. Then, we learn about some Nyquist functions that make interesting sequences of numbers and how to incorporate those sequences as note parameters in scores. Finally, we consider a variety of algorithmic composition techniques.

5.1 The Short Time Fourier Transform and FFT

In practice, the Fourier Transform cannot be used computationally because the input is infinite and the signals are continuous. Therefore, we use the Discrete Short Time Fourier Transform, where the continuous *integral* becomes a *summation* of discrete time points (samples), and where the summation is performed over a finite time interval, typically around 50 to 100 ms. The so-called Short-Time Discrete Fourier Transform (STDFT, or just DFT) equations are shown in Figure 5.1. Note the similarity to the Fourier Transform integrals we saw earlier. Here, R_k represents the real coefficients and X_k are the imaginary coefficients.

$$R_k = \sum_{i=0}^{N-1} x_i \cos(2\pi ki / N)$$

$$X_k = -\sum_{i=0}^{N-1} x_i \sin(2\pi ki / N)$$

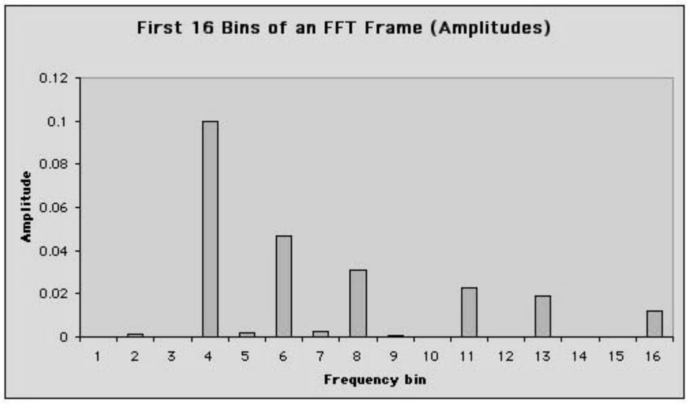
Figure 5.1: Equations for the Short Time Discrete Fourier Transform. Each R_k, X_k pair represents a different frequency in the spectrum.

The *Fast Fourier Transform* (FFT) is simply a fast algorithm for computing the DFT. FFT implies DFT, but rather than directly implementing the equations in Figure 5.1, which has a run time proportional to N^2, the FFT uses a very clever $NlogN$ algorithm to transform N samples.

An FFT of a time domain signal takes the samples and gives us a new set of numbers representing the frequencies, amplitudes, and phases of the sine waves that make up the sound we have analyzed. It is these data points that are displayed in the sonograms we looked at earlier.

Figure 5.2 illustrates some FFT data. It shows the first 16 bins of a typical FFT analysis after the conversion is made from real and imaginary numbers to amplitude/phase pairs. We left out the phases, because, well, it was too much trouble to just make up a bunch of arbitrary phases between 0 and 2. In a lot of cases, you might not need them (and in a lot of cases, you would!). In this case, the sample rate is 44.1 kHz and the FFT size is 1,024, so the bin width (in frequency) is the sample rate (44,100) divided by the FFT size, or about 43 Hz.

We confess that we just sort of made up the numbers; but notice that we made them up to represent a sound that has a simple, more or less harmonic structure with a fundamental somewhere in the 66 Hz to 88 Hz range (you can see its harmonics at around 2, 3, 4, 5, and 6 times its frequency, and note that the harmonics decrease in amplitude more or less like they would in a sawtooth wave).



Bin	Bin Frequency (Hz)	Amplitude	Phase
0	0	(omitted from plot)	
1	43	0.00003	
2	86	0.0015	
3	129	0.0001	
4	172	0.1	
5	215	0.002	
6	258	0.047	
7	301	0.0023	
8	345	0.032	
9	388	0.005	
10	431	0.00026	
11	474	0.023	
12	517	0.0001	
13	560	0.019	
14	603	0.00013	
15	646	0.00005	
16	689	0.0123	

Figure 5.2: Graph and table of spectral components.

One important consequence of the FFT (or equivalently, DFT) is that we only "see" the signal during a short period. We pretend that the spectrum is stable, at least for the duration of the analysis window, but this is rarely true. Thus, the FFT tells us something about the signal over the analysis window, but we should always keep in mind that the FFT changes with window size and the concept of a spectrum at a point in time is not well defined. For example, harmonics of periodic signals usually show up in the FFT as peaks in the spectrum with some spread rather than single-frequency spikes, even though a harmonic intuitively has a single specific frequency.

5.1.1 How to Interpret a Discrete Spectrum

Just as we divide signals over time into discrete points (called samples), the DFT divides the spectrum into discrete points we call *frequency bins*. The DFT captures all of the information in the analysis window, so if you analyze N samples, it you should have N degrees of freedom in the DFT. Since each DFT bin has an amplitude and phase, we end up with about $N/2$ bins. The complete story is that there are actually $N/2+1$ bins, but one represents zero frequency where only the real term is non-zero (because $sin(0)$ is 0), and one bin represents the Nyquist frequency where again, only the real term is non-zero (because $sin(2\pi ki/N)$ is zero).[1]

These $N/2+1$ bins are spaced across the frequency range from 0 to the Nyquist frequency (half the sample rate), so the spacing between bins is the sample rate divided by the number of bins:

bin frequency spacing = sample rate / number of samples

We can also relate the frequency spacing of bins to the duration of the analysis window:

number of samples = analysis duration $\times$ sample rate, so
bin frequency spacing =
sample rate / (analysis duration $\times$ sample rate), so
bin frequency spacing = 1 / analysis duration

5.1.2 Frame or Analysis Window Size

So let's say that we decide on a frame size of 1,024 samples. This is a common choice because most FFT algorithms in use for sound processing require a number of samples that is a power of two, and it's important not to get too much or too little of the sound.

A frame size of 1,024 samples gives us 512 frequency bands. If we assume that we're using a sample rate of 44.1 kHz, we know that we have a frequency range (remember the Nyquist theorem) of 0 kHz to 22.05 kHz. To find out how wide each of our frequency bins is, we use one of the formulas above, e.g. sample rate / number of samples, or 44100 / 1024, or about 43 Hz. Remember that frequency perception is logarithmic, so 43 Hz gives us worse resolution at the low frequencies and better resolution (at least perceptually) at higher frequencies.

By selecting a certain frame size and its corresponding bandwidth, we avoid the problem of having to compute an infinite number of frequency components in a sound. Instead, we just compute one component for each frequency band.

5.1.3 Time/Frequency Trade-off and the Uncertainty Principle

A drawback of the FFT is the trade-off that must be made between frequency and time resolution. The more accurately we want to measure the frequency content of

[1]With each bin having 2 dimensions (real and imaginary), $N/2+1$ bins implies the result of the transform has $N+2$ dimensions. That should set off some alarms: How can you transform N input dimensions (real-valued samples) into $N+2$ output dimensions? This apparent problem is resolved when you realize that the first and last imaginary terms are zero, so there are really only N output dimensions that carry any information. Without this property, the DFT would not be invertible.

a signal, the more samples we have to analyze in each frame of the FFT. Yet there is a cost to expanding the frame size—the larger the frame, the less we know about the temporal events that take place within that frame. This trade-off is captured in the expression:

bin frequency spacing = 1 / analysis duration.

Small frequency spacing is good, but so is a short duration, and they are inversely proportional! We simply can't have it both ways. See Figure 5.3 for an illustration.

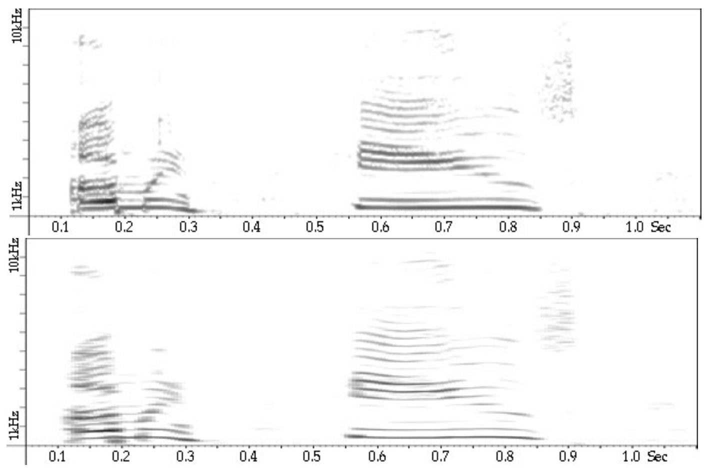


Figure 5.3: Selecting an FFT size involves making trade-offs in terms of time and frequency accuracy. Basically it boils down to this: The more accurate the analysis is in one domain, the less accurate it will be in the other. This figure illustrates what happens when we choose different frame sizes. In the first illustration, we used an FFT size of 512 samples, giving us pretty good time resolution. In the second, we used 2,048 samples, giving us pretty good frequency resolution. As a result, frequencies are smeared vertically in the first analysis, while time is smeared horizontally in the second. What's the solution to the time/frequency uncertainty dilemma? Compromise.

This time/frequency trade-off in audio analysis is mathematically identical to the Heisenberg Uncertainty Principle, which states that you can never simultaneously know the exact position and the exact speed of an object.

5.1.4 Magnitude or Amplitude Spectrum

Figure 5.1 shows that we get real (cosine) and imaginary (sine) numbers from the DFT or FFT. Typically, we prefer to work with amplitude and phase (in fact, we often ignore the phase part and just work with amplitude). You can think of real and imaginary as Cartesian coordinates, and amplitude and phase as radius and angle in polar coordinates. The amplitude components A_i are easily computed from the real R_i and imaginary X_i components using the formula

$$A_i = \sqrt{R_i^2 + X_i^2}$$

For an N-point FFT (input contains N samples), the output will have $N/2+1$ bins and therefore $N/2+1$ amplitudes. The first amplitude corresponds to 0 Hz, the second to (sample rate / N), etc., up to the last amplitude that corresponds to the frequency (sample rate / 2).

5.2 Spectral Centroid

Music cognition researchers and computer musicians commonly use a measure of sounds called the *spectral centroid*. The spectral centroid is a measure of the "brightness" of a sound, and it turns out to be extremely important in the way we compare different sounds. If two sounds have a radically different centroid, they are generally perceived to be timbrally distant (sometimes this is called a *spectral metric*).

Basically, the spectral centroid can be considered the average frequency component (taking into consideration the amplitude of all the frequency components), as illustrated in Figures 5.4 and 5.5. The formula for the spectral centroid of one FFT frame of a sound is:

$$c = \frac{\sum f_i a_i}{\sum a_i}, \text{where} f_i \text{ is the frequency of the } i^{\text{th}} \text{ bin and } a_{\text{i}} \text{ is the amplitude.}$$

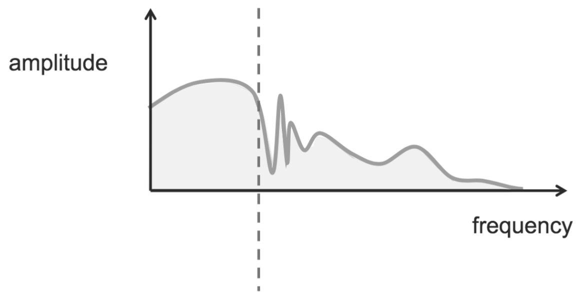


Figure 5.4: A magnitude spectrum and its spectral centroid (dashed line). If you were to cut out the spectrum shape from cardboard, it would balance on the centroid. The spectral centroid is a good measure of the overall placement of energy in a signal from low frequencies to high.

5.3 Patterns

Nyquist offers pattern objects that generate data streams. For example, the `cycle-class` of objects generate cyclical patterns such as "1 2 3 1 2 3 1 2 3 ...", or "1 2 3 4 3 2 1 2 3 4 ...". Patterns can be used to specify pitch sequences, rhythm, loudness, and other parameters.

In this section, we describe a variety of pattern objects. These patterns are pretty abstract, but you might think about how various types of sequences might be applied to music composition. In the following section, we will see how to incorporate patterns into the generation of scores for Nyquist, using a very powerful macro called `score-gen`.

Pattern functions are automatically loaded when you start Nyquist. To use a pattern object, you first create the pattern, e.g.

```
set pitch-source = make-cycle(list(c4, d4, e4, f4))
```

In this example, `pitch-source` is an object of class `cycle-class` which inherits from `pattern-class`.[2]

[2]Because SAL is not an object-oriented language, these classes and their methods are not directly

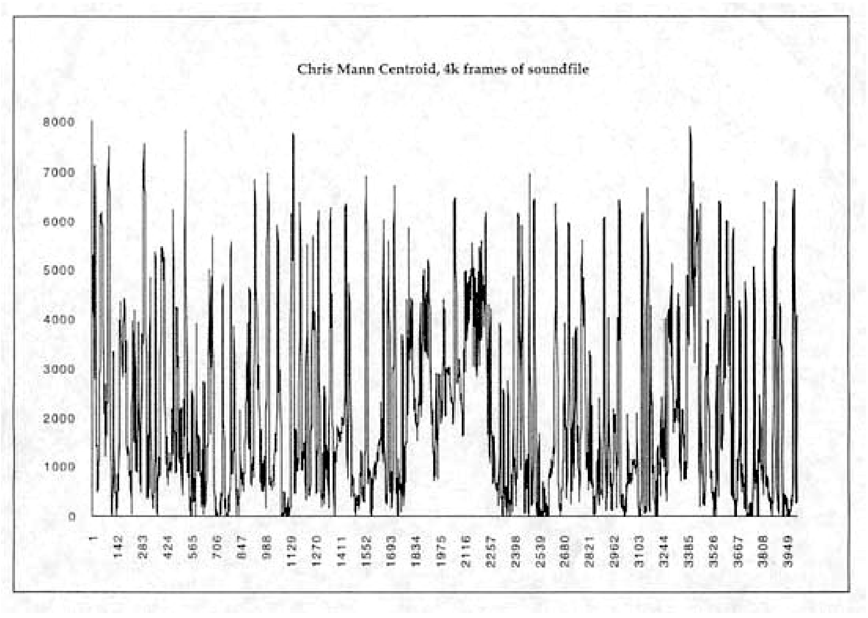


Figure 5.5: The centroid curve of a sound over time. This curve is of a rapidly changing voice (Australian sound poet Chris Mann). Each point in the horizontal dimension represents a new spectral frame. Note that centroids tend to be rather high and never the fundamental (this would only occur if our sound is a pure sine wave).

After creating the pattern, you can access it repeatedly with `next` to generate data, e.g.

```
play seqrep(i, 13, pluck(next(pitch-source), 0.2))
```

This will create a sequence of notes with the following pitches: c, d, e, f, c, d, e, f, c, d, e, f, c. If you evaluate this again, the pitch sequence will continue, starting on d.

It is very important not to confuse the creation of a sequence with its access. Consider this example:

```
play seqrep(i, 13, pluck(next(make-cycle(
                        list(c4, d4, e4, f4))), 0.2))
```

This looks very much like the previous example, but it only repeats notes on middle-C. The reason is that every time `pluck` is evaluated, `make-cycle` is called and creates a new pattern object. After the first item of the pattern is extracted with `next`, the cycle is not used again, and no other items are generated.

To summarize this important point, there are two steps to using a pattern. First, the pattern is created and stored in a variable. Second, the pattern is accessed (multiple times) using `next`.

`next(`*pattern-object* `[,#t ])`

accessible from SAL. Instead, Nyquist defines a functional interface, e.g. `make-cycle` creates an instance of `cycle-class`, and the `next` function, introduced below, retrieves the next value from any instance of `pattern-class`. Using LISP syntax, you can have full access to the methods of all objects.

`next` returns the next element from a pattern generator object. If the optional second argument is true (default value is false), then an entire period is returned as a list.

5.3.1 Pattern Examples

The following descriptions cover the basic ideas of the Nyquist pattern library. In most cases, details and optional parameters are not described here to keep the descriptions short and to emphasize the main purpose of each type of pattern. For complete descriptions, see the *Nyquist Reference Manual*, or better yet, just read the manual instead of these summaries.

Heap

The `heap-class` selects items in random order from a list without replacement, which means that all items are generated once before any item is repeated. For example, two periods of `make-heap({a b c})` might be `(C A B) (B A C)`. Normally, repetitions can occur even if all list elements are distinct. This happens when the last element of a period is chosen first in the next period. To avoid repetitions, the `max:` keyword argument can be set to 1. If the main argument is a pattern instead of a list, a period from that pattern becomes the list from which random selections are made, and a new list is generated every period. (See Section 5.3.2 on nested patterns.)

Palindrome

The `palindrome-class` repeatedly traverses a list forwards and then backwards. For example, two periods of `make-palindrome({a b c})` would be `(A B C C B A) (A B C C B A)`. The `elide`: keyword parameter controls whether the first and/or last elements are repeated:

```
make-palindrome(a b c, elide: nil)
       ;; generates A B C C B A A B C C B A ...

make-palindrome(a b c, elide: t)
       ;; generates A B C B A B C B ...

make-palindrome(a b c, elide: :first)
       ;; generates A B C C B A B C C B ...

make-palindrome(a b c, elide: :last)
       ;; generates A B C B A A B C B A ...
```

Random

The `random-class` generates items at random from a list. The default selection is uniform random with replacement, but items may be further specified with a weight, a minimum repetition count, and a maximum repetition count. Weights give the relative probability of the selection of the item (with a default weight of one). The minimum count specifies how many times an item, once selected at random, will be repeated. The maximum count specifies the maximum number of times an item can be selected in a row. If an item has been generated n times in succession, and the maximum is equal to n, then the item is disqualified in the next random selection. Weights (but not currently minima and maxima) can be patterns. The patterns (thus the weights) are recomputed every period.

Line

The `line-class` is similar to the cycle class, but when it reaches the end of the list of items, it simply repeats the last item in the list. For example, two periods of `make-line({a b c})` would be `(A B C) (C C C)`.

Accumulation

The `accumulation-class` takes a list of values and returns the first, followed by the first two, followed by the first three, etc. In other words, for each list item, return all items form the first through the item. For example, if the list is `(A B C)`, each generated period is `(A A B A B C)`.

Copier

The `copier-class` makes copies of periods from a sub-pattern. For example, three periods of `make-copier( make-cycle({a b c}, for: 1), repeat: 2, merge: t)` would be `(A A) (B B) (C C)`. Note that entire periods (not individual items) are repeated, so in this example the `for:` keyword was used to force periods to be of length one so that each item is repeated by the `repeat:` count.

Length

The `length-class` generates periods of a specified length from another pattern. This is similar to using the `for:` keyword, but for many patterns, the `for:` parameter alters the points at which other patterns are generated. For example, if the palindrome pattern has an `elide:` pattern parameter, the value will be computed every period. If there is also a `for:` parameter with a value of 2, then elide: will be recomputed every 2 items. In contrast, if the palindrome (without a `for:` parameter) is embedded in a *length* pattern with a length of 2, then the periods will all be of length 2, but the items will come from default periods of the palindrome, and therefore the `elide:` values will be recomputed at the beginnings of default palindrome periods.

Window

The `window-class` groups items from another pattern by using a sliding window. If the *skip* value is 1, each output period is formed by dropping the first item of the previous period and appending the next item from the pattern. The *skip* value and the output period length can change every period. For a simple example, if the period length is 3 and the *skip* value is 1, and the input pattern generates the sequence A, B, C, ..., then the output periods will be (A B C), (B C D), (C D E), (D E F), ...

5.3.2 Nested Patterns

Patterns can be nested, that is, you can write patterns of patterns. In general, the `next` function does not return patterns. Instead, if the next item in a pattern is a (nested) pattern, next recursively gets the next item of the nested pattern.

While you might expect that each call to `next` would advance the top-level pattern to the next item, and descend recursively if necessary to the inner-most nesting level, this is not how `next` works. Instead, `next` remembers the last top-level item, and if it was a pattern, `next` continues to generate items from that same inner pattern until the end of the inner pattern's *period* is reached. The next paragraph explains the concept of the *period*.

Periods

The data returned by a pattern object is structured into logical groups called *periods*. You can get an entire period (as a list) by calling `next(pattern, t)`. For example:

```
set pitch-source = make-cycle(list(c4, d4, e4, f4))
print next(pitch-source, t)
```

This prints the list `(60 62 64 65)`, which is one period of the cycle.

You can also get explicit markers that delineate periods by calling `send(pattern, :next)`. In this case, the value returned is either the next item of the pattern, or the symbol `+eop+` if the end of a period has been reached. What determines a period? This is up to the specific pattern class, so see the documentation for specifics. You can override the "natural" period by using the keyword `for:`, e.g.

```
set pitch-source =
        make-cycle(list(c4, d4, e4, f4), for: 3)
print next(pitch-source, t)
print next(pitch-source, t)
```

This prints the lists `(60 62 64) (65 60 62)`. Notice that these periods just restructure the stream of items into groups of 3.

Nested patterns are probably easier to understand by example than by specification. Here is a simple nested pattern of cycles:

```
set cycle-1 = make-cycle({a b c})
set cycle-2 = make-cycle({x y z})
set cycle-3 = make-cycle(list(cycle-1, cycle-2))
loop
  repeat 9
  exec format(t, "~A ", next(cycle-3))
end
```

This will print "A B C X Y Z A B C." Notice that the inner-most cycles `cycle-1` and `cycle-2` generate a period of items before the top-level `cycle-3` advances to the next pattern.

5.3.3 Summary of Patterns

Pattern generators are a bit like unit generators in that they represent sequences or streams of values, they can be combined to create complex computations, they encapsulate state on which the next output depends, and they can be evaluated incrementally. Patterns produce streams of numbers intended to become parameters for algorithmic compositions, while unit generators produce samples.

Patterns can serve as parameters to other pattern objects, enabling complex behaviors to run on multiple time scales. Since patterns are often constructed from lists (e.g. cycle, random, heap, copier, line, palindrome patterns), the output of each pattern is structured into groupings called *periods*, and pattern generators have parameters (especially *for:*) to control or override period lengths. Even period lengths can be controlled by patterns!

5.4 Score Generation and Manipulation

A common application of pattern generators is to specify parameters for notes. (It should be understood that "notes" in this context means any Nyquist behavior, whether it represents a conventional note, an abstract sound object, or even some micro-sound event that is just a low-level component of a hierarchical sound organization. Similarly, "score" should be taken to mean a specification for a sequence of these "notes.") The `score-gen` macro establishes a convention for representing scores and for generating them using patterns.

The `timed-seq` macro already provides a way to represent a score as a list of expressions. We can go a bit further by specifying that *all notes are specified by an alternation of keywords and values, where some keywords have specific meanings and interpretations.* By insisting on this keyword/value representation, we can treat scores as self-describing data, as we will see below.

To facilitate using *patterns* to create *scores*, we introduce the `score-gen` construct, which looks like a function but is actually a *macro*. The main difference is that a *macro* does not evaluate every parameter immediately, but instead can operate on parameters as expressions. The basic idea of `score-gen` is you provide a template for notes in a score as a set of keywords and values. For example,

```
set pitch-pattern = make-cycle(list(c4, d4, e4, f4))
score-gen(dur: 0.4, name: quote(my-sound),
          pitch: next(pitch-pattern), score-len: 9)
```

Generates a score of 9 notes as follows:

```
((0 0 (SCORE-BEGIN-END 0 3.6))
 (0 0.4 (MY-SOUND :PITCH 60))
 (0.4 0.4 (MY-SOUND :PITCH 62))
 (0.8 0.4 (MY-SOUND :PITCH 64))
 (1.2 0.4 (MY-SOUND :PITCH 65))
 (1.6 0.4 (MY-SOUND :PITCH 60))
 (2 0.4 (MY-SOUND :PITCH 62))
 (2.4 0.4 (MY-SOUND :PITCH 64))
 (2.8 0.4 (MY-SOUND :PITCH 65))
 (3.2 0.4 (MY-SOUND :PITCH 60)))
```

The use of keywords like `:PITCH` helps to make scores readable and easy to process without specific knowledge about the functions called in the score. For example, one could write a transpose operation to transform all the `:pitch` parameters in a score without having to know that pitch is the first parameter of `pluck` and the second parameter of `piano-note`. Keyword parameters are also used to give flexibility to note specification with `score-gen`. Since this approach requires the use of keywords, the next section is a brief explanation of how to define functions that use keyword parameters.

5.4.1 Keyword Parameters

Keyword parameters are parameters whose presence is indicated by a special symbol, called a keyword, followed by the actual parameter. Keyword parameters in SAL have default values that are used if no actual parameter is provided by the caller of the function.

To specify that a parameter is a keyword parameter, use a keyword symbol (one that ends in a colon) followed by a default value. For example, here is a function that accepts keyword parameters and invokes the `pluck` function:

```
define function k-pluck(pitch: 60, dur: 1)
  return pluck(pitch, dur)
```

Notice that within the body of the function, the actual parameter value for keywords `pitch:` and `dur:` are referenced by writing the keywords without the colons (`pitch` and `dur`) as can be seen in the call to `pluck`. Also, keyword parameters have default values. Here, they are 60 and 1, respectively.

Now, we can call k-pluck with keyword parameters. A function call would look like:

```
k-pluck(pitch: c3, dur: 3)
```

Usually, it is best to give keyword parameters useful default values. That way, if a parameter such as `dur:` is missing, a reasonable default value (1) can be used automatically. It is never an error to omit a keyword parameter, but the called function can check to see if a keyword parameter was supplied or not. Because of default values, we can call `k-pluck(pitch:  c3)` with no duration, `k-pluck(dur:  3)` with only a duration, or even `k-pluck()` with no parameters.

5.4.2 Using score-gen

The `score-gen` macro computes a score based on keyword parameters. Some keywords have a special meaning, while others are not interpreted but merely placed in the score. The resulting score can be synthesized using `timed-seq`. The form of a call to `score-gen` is simply:

```
score-gen(k1: e1, k2: e2, ...)
```

where the *k*'s are keywords and the *e*'s are expressions. A score is generated by evaluating the expressions once for each note and constructing a list of keyword-value pairs. A number of keywords have special interpretations. The rules for interpreting these parameters will be explained through a set of "How do I ..." questions below.

How many notes will be generated? The keyword parameter `score-len:` specifies an upper bound on the number of notes. The keyword `score-dur:` specifies an upper bound on the starting time of the last note in the score. (To be more precise, the `score-dur:` bound is reached when the default starting time of the next note is greater than or equal to the `score-dur:` value. This definition is necessary because note times are not strictly increasing.) When either bound is reached, score generation ends. At least one of these two parameters must be specified or an error is raised. These keyword parameters are evaluated just once and are not copied into the parameter lists of generated notes.

What is the duration of generated notes? The keyword `dur:` defaults to 1 and specifies the nominal duration in seconds. Since the generated note list is compatible with `timed-seq`, the starting time and duration (to be precise, the stretch factor) are not passed as parameters to the notes. Instead, they control the Nyquist environment in which the note will be evaluated.

What is the start time of a note? The default start time of the first note is zero. Given a note, the default start time of the next note is the start time plus the inter-onset time, which is given by the `ioi:` parameter. If no `ioi:` parameter is specified, the inter-onset time defaults to the duration, given by `dur:`. In all cases, the default start time of a note can be overridden by the keyword parameter `time:`. So in other words, to get the time of each note, compute the expression given as `(time:)`. If there is no `time:` parameter, compute the time of the previous note plus the value of `ioi:`, and if there is no `ioi:`, then use `dur:`, and if there is no `dur:`, use 1.

When does the score begin and end? The behavior `SCORE-BEGIN-END` contains the beginning and ending of the score (these are used for score manipulations, e.g. when scores are merged, their begin times can be aligned.) When `timed-seq` is used to synthesize a score, the `SCORE-BEGIN-END` marker is not evaluated. The `score-gen` macro inserts an event of the form

```
(0 0 (SCORE-BEGIN-END begin-time end-time))
```

with begin-time and end-time determined by the `begin:` and `end:` keyword parameters, respectively. (Recall that these `score-begin-end` events do not make sound, but they are used by score manipulation functions for splicing, stretching, and other operations on scores.) If the `begin:` keyword is not provided, the score begins at zero. If the `end:` keyword is not provided, the score ends at the default start time of what would be the next note after the last note in the score (as described in the previous paragraph). Note: if `time:` is used to compute note starting times, and these times are not increasing, it is strongly advised to use end: to specify an end time for the score, because the default end time may not make sense.

What function is called to synthesize the note? The name: parameter names the function. Like other parameters, the value can be any expression, including something like next(`fn-name-pattern`), allowing function names to be recomputed for each note. The default value is `note`.

Can I make parameters depend upon the starting time or the duration of the note? `score-gen` sets some handy variables that can be used in expressions that compute parameter values for notes:

- the variable `sg:start` accesses the start time of the note,
- `sg:ioi` accesses the inter-onset time,
- `sg:dur` accesses the duration (stretch factor) of the note,
- `sg:count` counts how many notes have been computed so far, starting at 0.

The order of computation is: `sg:count`, then `sg:start`, then `sg:ioi` and finally `sg:dur`, so for example, an expression for `dur:` can depend on `sg:ioi`.

Can parameters depend on each other? The keyword `pre:` introduces an expression that is evaluated before each note, and `post:` provides an expression to be evaluated after each note. The `pre:` expression can assign one or more global variables which are then used in one or more expressions for parameters.

How do I debug score-gen expressions? You can set the `trace:` parameter to true (t) to enable a print statement for each generated note.

How can I save scores generated by `score-gen` that I like? If the keyword parameter `save:` is set to a symbol, the global variable named by the symbol is set to the value of the generated sequence. Of course, the value returned by `score-gen` is just an ordinary list that can be saved like any other value.

In summary, the following keywords have special interpretations in `score-gen`: `begin:`, `end:`, `time:`, `dur:`, `name:`, `ioi:`, `trace:`, `save:`, `score-len:`, `score-dur:`, `pre:`, `post:`. All other keyword parameters are expressions that are evaluated once for each note and become the parameters of the notes.

5.5 Introduction to Algorithmic Composition

There are many types and approaches to *machine-aided composition*, which is also called *algorithmic composition*, *computer-assisted composition*, *automatic composition*, and *machine generated music* (all of which have varying shades of meaning depending on the author that uses them and the context).

Some examples of approaches include:

- Use of music notation software, which is arguably computer-assisted, but rarely what we mean by “machine-aided composition.”
- Cutting and pasting of music materials (notation or audio) in editors can be effective, but leaves all the decision making to the composer, so again, this is not usually considered to be machine-aided composition.
- Editing macros and other high-level operations enable composers to (sometimes) delegate some musical decision-making to machines and perform high-level operations on music compositions.
- *Algorithmic composition* usually refers to simple numerical or symbolic algorithms to generate musical material, e.g. mapping the digits of π to pitches.
- Procedures + random numbers allow composers to create structures with randomized details.
- *Artificial intelligence* seeks to model music composition as search, problem solving, knowledge-directed decisions or optimization.
- Music models attempt to formalize music theory or perhaps particular musical styles to enable examples to be generated computationally.
- *Machine learning* typically seeks to learn music models automatically from examples, often using sequence-learning techniques from other domains such as natural language processing.
- *Constraint solving and search* techniques allow composers to describe music in terms of properties or restrictions and then use computers to search for solutions that satisfy the composers’ specifications.

The following sections describe a number of approaches that I have found useful. The list is by no means exhaustive. One thing you will discover is that there are few if any music generation systems that one can simply pick up and use. Almost every approach and software package offers a framework within which composers can adjust parameters or provide additional details to produce material which is often just the starting point in producing a complete piece of music. There is no free lunch!

For much more on algorithmic composition, see *Algorithmic Composition: A Guide to Composing Music with Nyquist* [**?**]. This book contains many examples created with Nyquist.

5.5.1 Negative Exponential Distribution

Random numbers can be interesting, but what does it mean to be “random” in time? You might think that it would be useful to make the inter-onset interval (IOI) be a uniform random distribution, but this turns out to be not so interesting because the resulting sequence or rhythm is a little too predictable.

In the real world, we have random events in time such as atomic decay or something as mundane as the time points when a yellow car drives by. The inter-arrival time of these random events has a negative exponential distribution, as shown in Figure 5.6. The figure shows that the probability of longer and longer intervals is less and less likely.

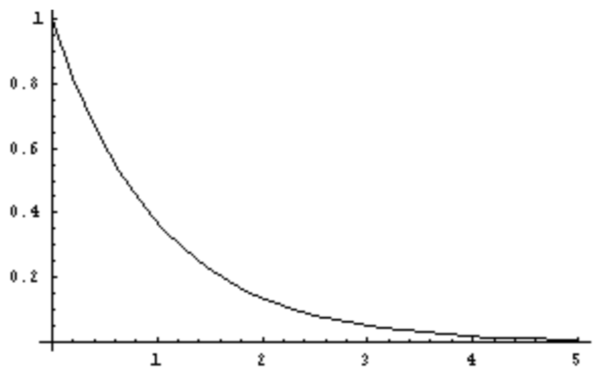


Figure 5.6: The negative exponential distribution.

See "Distributions" in the *Nyquist Reference Manual* for implementations. (The `exponential-dist` function can be used directly to compute `ioi:` values in `score-gen`. You might want to code upper and lower bounds to avoid the extremes of this infinite distribution.) Another way to generate something like a (negative) exponential distribution is, at every time point (e.g. every sixteenth note or every 10 ms), play a note with some low probability p. In this approach, IOI is not explicitly computed, but the distribution of IOI's will approximate the negative exponential shown in Figure 5.6.

Yet another way to achieve a negative exponential distribution is to choose time points uniformly within a time span (e.g. in Nyquist, the expression `rrandom() * dur` will return a random number in [0, `dur`).) Then, *sort* the time points into increasing order, and you will obtain points where the IOI has a negative exponential distribution.

Nyquist supports many other interesting probability distributions. See "Distributions" in the *Nyquist Reference Manual* for details.

5.5.2 Random Walk

What kinds of pitch sequences create interesting melodies? Melodies are mostly made up of small intervals, but a uniform random distribution creates many large intervals. Melodies often have fractal properties, with a mix of mainly small intervals, but occasional larger ones. An interesting idea is to randomly choose a direction (up or down) and interval size in a way that emphasizes smaller intervals over larger ones. Sometimes, this is called a "random walk," as illustrated in Figure 5.7.

5.5.3 Markov Chains

A Markov Chain is a formal model of random sequential processes that is widely used in many fields. A Markov Chain consists of a set of states (in music, these could be chords, for example). The output of a Markov Chain is a sequence of

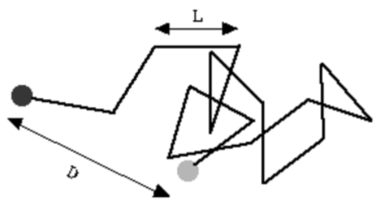


Figure 5.7: A random walk in two dimensions (from http://www2.ess.ucla.edu/ jewitt/images/random.gif).

states. The probability of making a transition to a state depends *only* on the previous state. Figure 5.8 illustrates a Markov Chain.

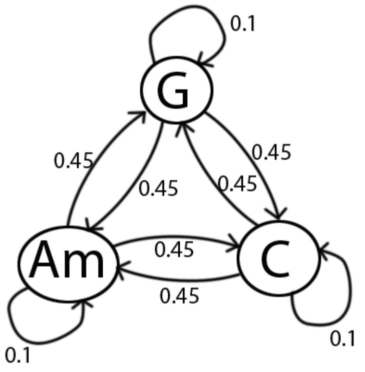


Figure 5.8: A Markov Chain for simple chord progressions (from https://luckytoilet.files.wordpress.com/2017/04/3.png).

5.5.4 Rhythmic Pattern Generation

There are many techniques for rhythmic patterns. An interesting perceptual fact about rhythms is that almost any rhythm will make a certain amount of musical sense if it has at least two or three sound events, not so many sound events that we cannot remember the sequence, and *the rhythm is repeated*. Therefore, a simple but effective "random rhythm generator" is the following: Decide on a length. Generate this many random Boolean values in sequence. Make each element of the sequence represent an onset or a rest and play at least several repetitions of the resulting rhythm. For example, if each 0 or 1 represents a sixteenth note, and we play a sound on each "1", a clear structured rhythm will be heard:

01101110100 01101110100 01101110100 01101110100 ...

5.5.5 Serialism

At the risk of over-simplifying, serialism arose as an attempt to move beyond the tonal concepts that dominated Western musical thought through the beginning of the 20th Century. Arnold Schoenberg created his twelve-tone technique that organized music around "tone rows" which are permutations of the 12 pitch classes of the chromatic scale (namely, C, C#, D, D#, E, F, F#, G, G#, A, A#, B). Starting with a single permutation, the composer could generate new rows through transposition, inversion, and playing the row backwards (called retrograde). Music created from these sequences tends to be *atonal* with no favored pitch center or scale. Because of the formal constraints on pitch selection and operations on tone rows, serialism has been an inspiration for many algorithmic music compositions.

5.5.6 Fractals and Nature

Melodic contours are often fractal-like, and composers often use fractal curves to generate music data. Examples include Larry Austin's *Canadian Coastline*, based on the fractal nature of a natural coastline, and John Cage s *Atlas Eclipticalis*, based on positions of stars.

5.5.7 Grammars

Formal grammars, most commonly used in formal descriptions of programming languages, have been applied to music. Consider the formal grammar:

```
melody ::= intro middle ending
middle ::= phrase | middle phrase
phrase ::= A B C B | A C D A
```

which can be read as: "A *melody* is an *intro* followed by a *middle* followed by and *ending*. A *middle* is recursively defined as either a *phrase* or a *middle* followed by a *phrase* (thus, a *middle* is one or more *phrases*), and a *phrase* is the sequence A B C B or the sequence A C D A. We haven't specified yet what is an *intro* or *ending*.

Grammars are interesting because they give a compact description of many alternatives, e.g. the the formal grammar for Java allows for all Java programs, and the music grammar above describes many possibilities as well. However, grammars give clear constraints—a Nyquist program is rejected by the grammar for Java, and music in the key of C# cannot be generated by the music grammar example above.

5.5.8 Pitch and Rhythm Grids

Traditional Western music is based on discrete pitch and time values. We build scales out of half steps that we often represent with integers, and we divide time into beats and sub-beats. One interesting and useful technique in algorithmic music is to compute parameters from continuous distributions, but then *quantize* values to fit pitches into conventional scales or round times and durations to the nearest sixteenth beat or some other rhythmic grid.

5.6 Tendency Masks

A problem with algorithmic composition is that the output can become static if the same parameters or probability distributions are used over and over. At some point, we recognize that even random music has an underlying probability distribution, so what is random becomes just a confirmation of our expectations.

One way to combat this perception of stasis is to create multi-level structures, e.g. patterns of patterns of patterns. A more direct way is to simply give the high-level control back to the composer. As early as 1970, Gottfried Michael Koenig used *tendency masks* to control the evolution of parameters in time in order to give global structure and control to algorithmic music composition. Figure 5.9 illustrates tendency masks for two parameters. The masks give a range of possible values for parameters. Note that the parameter values can be chosen between upper and lower bounds, and that the bounds change over time.

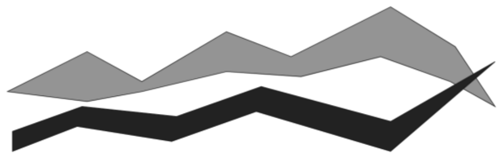


Figure 5.9: Tendency masks offer a way for composers to retain global control over the evolution of a piece even when the moment-by-moment details of the piece are generated algorithmically. Here, the vertical axis represents parameter values and the horizontal axis represents time. The two colored areas represent possible values (to be randomly selected) for each of two parameters used for music generation.

5.7 Summary

We have learned about a variety of approaches to algorithmic music generation. Nyquist supports algorithmic music generation especially through the `score-gen` macro, which iteratively evaluates expressions and creates note lists in the form of Nyquist scores. Nyquist also offers a rich pattern library for generating parameter values. These patterns can be used to implement many "standard" algorithmic music techniques such as serialism, random walks, Markov Chains, probability distributions, pitch and rhythmic grids, etc.

We began by learning details of the FFT and Spectral Centroid. The main reason to introduce these topics is to prepare you to extract spectral centroid data from one sound source and use it to control parameters of music synthesis. Hopefully, you can combine this sonic control with some higher-level algorithmic generation of sound events to create an interesting piece. Good luck!