This pdf for ICM students only - ebook
and paperback available from amazon.com

Introduction to Computer Music

Week 1

Instructor: Prof. Roger B. Dannenberg

Topics Discussed: Sound, Nyquist, SAL, Lisp and Control Constructs

This pdf for ICM students only - ebook
and paperback available from amazon.com

Chapter 1

Introduction

Topics Discussed: Sound, Nyquist, SAL, Lisp and Control Constructs

Computers in all forms – desktops, laptops, mobile phones – are used to store and play music. Perhaps less obvious is the fact that music is now recorded and edited almost exclusively with computers, and computers are also used to generate sounds either by manipulating short audio clips called samples or by direct computation. In the late 1950s, engineers began to turn the theory of sampling into practice, turning sound into bits and bytes and then back into sound. These analog-to-digital converters, capable of turning one second's worth of sound into thousands of numbers made it possible to transform the acoustic waves that we hear as sound into long sequences of numbers, which were then stored in computer memories. The numbers could then be turned back into the original sound. The ability to simply record a sound had been known for quite some time. The major advance of digital representations of sound was that now sound could be manipulated very easily, just as a chunk of data. Advantages of employing computer music and digital audio are:

1. There are no limits to the range of sounds that a computer can help explore. In principle, *any* sound can be represented digitally, and therefore any sound can be created.

2. Computers bring precision. The process to compute or change a sound can be repeated exactly, and small incremental changes can be specified in detail. Computers facilitate microscopic changes in sounds enabling us to produce desirable sounds.

3. Computation implies automation. Computers can take over repetitive tasks. Decisions about music making can be made at different levels, from the highest level of composition, form and structure, to the minutest detail of an individual sound. Unlike with conventional music, we can use automation to hear the results of these decisions quickly and we can refine computer programs accordingly.

4. Computers can blur the lines between the traditional roles of the composer and the performer and even the audience. We can build interactive systems where, thanks to automation, composition is taking place in real time.

The creative potential for musical composition and sound generation empowered a revolution in the world of music. That revolution in the world of electroacoustic music engendered a wonderful synthesis of music, mathematics and computing.

This pdf for ICM students only - ebook
and paperback available from amazon.com

1.1 Theory and Practice

This book is intended for a course that has two main components: the technology of computer music and making music with computers. The technology of computer music includes *theory*, which covers topics such as digital audio and digital signal processing, software design and languages, data structures and representation. All of these form a conceptual framework that you need to understand the field. The second aspect of the technology of computer music is putting these concepts into *practice* to make them more concrete and practical. We also make music. Making music also has a theoretical side, mainly listening to music and discussing what we find in the music. Of course, there is also a practical side of making music, consisting in this book mostly of writing programs in Nyquist, a powerful composition and sound synthesis language.

A composer is expected to have understanding of acoustics and psychoacoustics. The physics of sound (acoustics) is often confused with the way in which we perceive it (psychoacoustics). We will get to these topics later. In the next section, we discuss sound's physical characteristics and common measurements. Following that, we change topics and give a brief introduction to the Nyquist language.

1.1.1 Warning! Programming!

This is not an introduction to programming! If you do not already know how to program in some language such as Python or Java, you will not understand much of the content in this book. If you already have some programming experience, you should be able to pick up Nyquist quickly with the introduction in this and the next chapter. If you find the pace is too quick, I recommend *Algorithmic Composition: A Guide to Composing Music with Nyquist* [**?**], which was written for ınon-programmers and introduces Nyquist at a slower pace. As the title implies, that book also covers more algorithmic composition techniques than this one.

You should also install Nyquist, the programming language, from SourceForge (sourceforge.net/projects/nyquist/). Once installed, find the *Nyquist Reference Manual*. Whenever you have questions about Nyquist, look in the reference manual for details and more examples.

1.2 Fundamentals of Computer Sound

All musicians work with sound in some way, but many have little understanding of its properties. Computer musicians can benefit in myriad ways from an understanding of the mechanisms of sound, its objective measurements and the more subjective area of its perception. This understanding is crucial to the proper use of common studio equipment and music software, and novel compositional strategies can be derived from exploiting much of the information contained in this section.

1.2.1 What is Sound?

Sound is a complex phenomenon involving physics and perception. Perhaps the simplest way to explain it is to say that sound involves at least three things:

1. something moves,
2. something transmits the results of that movement,
3. something (or someone) hears the results of that movement (though this is philosophically debatable).

This pdf for ICM students only - ebook
and paperback available from amazon.com

All things that make sound move, and in some very metaphysical sense, all things that move (if they don't move too slowly or too quickly) make sound. As things move, they push and pull at the surrounding air (or water or whatever medium they occupy), causing pressure variations (compressions and rarefactions). Those pressure variations, or sound waves, are what we hear as sound. (See Figure 1.1.)

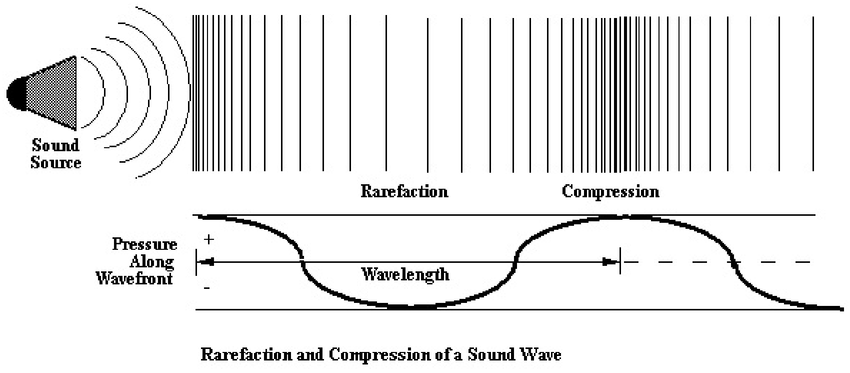

Figure 1.1: Illustration of a wave and the corresponding pressure variations in the air.

Sound is produced by a rapid variation in the average density or pressure of air molecules above and below the current atmospheric pressure. We perceive sound as these pressure fluctuations cause our eardrums to vibrate. When discussing sound, these usually minute changes in atmospheric pressure are referred to as sound pressure and the fluctuations in pressure as sound waves. Sound waves are produced by a vibrating body, be it an oboe reed, loudspeaker cone or jet engine. The vibrating sound source disturbs surrounding air molecules, causing them to bounce off each other with a force proportional to the disturbance.

The speed at which sound propagates (or travels from its source) is directly influenced by both the medium through which it travels and the factors affecting the medium, such as altitude, humidity and temperature for gases like air. There is no sound in the vacuum of space because there are too few molecules to propagate a wave. The approximate speed of sound at 20° Celsius (68° Fahrenheit) is 1128 feet per second (f/s).

It is important to note that the speed of sound in air is determined by the conditions of the air itself (e.g. humidity, temperature, altitude). It is not dependent upon the sound's amplitude, frequency or wavelength.

Pressure variations travel through air as waves. Sound waves are often characterized by four basic qualities, though many more are related: frequency, amplitude, wave shape and phase.[1] Some sound waves are periodic, in that the change from equilibrium (average atmospheric pressure) to maximum compression to maximum rarefaction back to equilibrium is repetitive. The "round trip" back to the starting point just described is called a cycle or period.

The number of cycles per unit of time is called the frequency. (See Figure 1.2.) For convenience, frequency is measured in cycles per second (cps) or the interchangeable Hertz (Hz) (60 cps = 60 Hz), named after the 19th C. physicist. 1000 Hz is often referred to as 1 kHz (kilohertz) or simply "1k" in studio parlance.

[1]It could be argued that phase is not a characteristic of a single wave, but only as a comparison between two or more waves.

This pdf for ICM students only - ebook
and paperback available from amazon.com

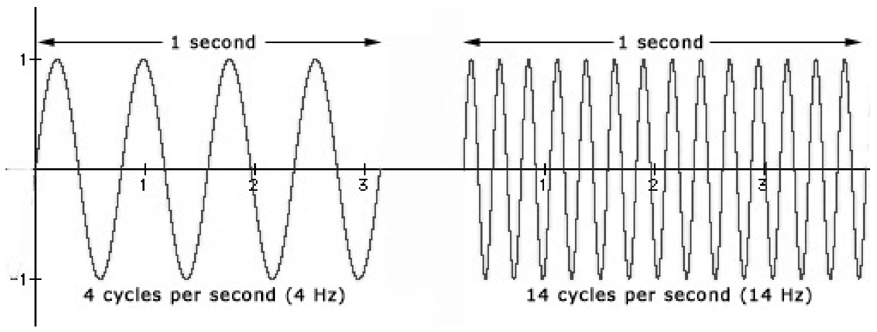

Figure 1.2: Illustration of how waveform changes with the change in frequency.

The range of human hearing in the young is approximately 20 Hz to 20 kHz—the higher number tends to decrease with age (as do many other things). It may be quite normal for a 60-year-old to hear a maximum of 16,000 Hz. Frequencies above and below the range of human hearing are also commonly used in computer music studios.

Amplitude is the objective measurement of the degree of change (positive or negative) in atmospheric pressure (the compression and rarefaction of air molecules) caused by sound waves. Sounds with greater amplitude will produce greater changes in atmospheric pressure from high pressure to low pressure to the ambient pressure present before sound was produced (equilibrium). Humans can hear atmospheric pressure fluctuations of as little as a few billionths of an atmosphere (the ambient pressure), and this amplitude is called the threshold of hearing. On the other end of the human perception spectrum, a super-loud sound near the threshold of pain may be 100,000 times the pressure amplitude of the threshold of hearing, yet only a 0.03% change at your ear drum in the actual atmospheric pressure. We hear amplitude variations over about 5 orders of magnitude from threshold to pain.

1.2.2 Analog Sound

Sound itself is a continuous wave; it is an analog signal. (See Figure 1.3.) When we record audio, we start with continuous vibrations that are analogous to the original sound waves. Capturing this continuous wave in its entirety requires an analog recording system; what the microphone receives is transformed continuously into a groove of a vinyl disk or magnetism of a tape recorder. Analog can be said to be the true representation of the sound at the moment it was recorded. The analog waveform is nice and smooth, while the digital version is kind of chunky. This “chunkiness” is called quantization. Does this really matter? Keep reading...

1.2.3 Digital Audio Representation

Sounds from the real world can be recorded and digitized using an analog-to-digital converter (ADC). As in Figure 1.4, the circuit takes a sample of the instantaneous amplitude (not frequency) of the analog waveform. Alternatively, digital synthesis software can also create samples by modeling and sampling mathematical functions or other forms of calculation. A sample in either case is defined as a measurement of the instantaneous amplitude of a real or artificial signal. Frequencies will be recreated later by playing back the sequential sample amplitudes at a

This pdf for ICM students only - ebook
and paperback available from amazon.com

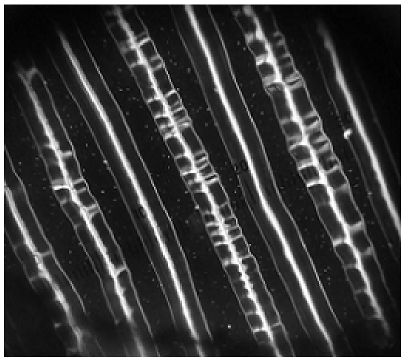

Figure 1.3: Before audio recording became digital, sounds were "carved" into vinyl records or written to tape as magnetic waveforms. Left image shows wiggles in vinyl record grooves and the right image shows a typical tape used to store audio data.

specified rate. It is important to remember that frequency, phase, waveshape, etc. are not recorded in each discrete sample measurement, but will be reconstructed during the playback of the stored sequential amplitudes.

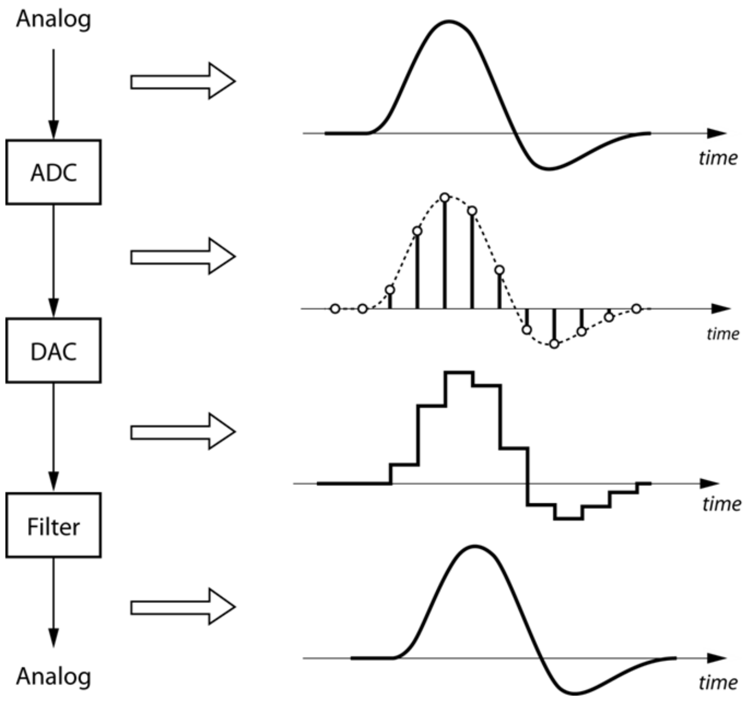

Figure 1.4: An analog waveform is converted by an ADC to digital form. Conversion back to analog is shown in two steps. First, each digital *sample* is converted to a voltage. Second, the signal is *smoothed* by a filter. (Sometimes, the filter step is assumed and the two-step process is referred to as DAC).

This pdf for ICM students only - ebook
and paperback available from amazon.com

Samples are taken at a regular time interval. The rate of sample measurement is called the sampling rate (or sampling frequency). The sampling rate is responsible for the frequency response of the digitized sound.

To convert the digital audio into the analog format, we use *digital-to-analog converters*. A digital-to-analog converter, or *DAC*, is an electronic device that converts a digital code to an analog signal such as a voltage, current, or electric charge. Signals can easily be stored and transmitted in digital form; a DAC is used for the signal to be recognized by human senses or non-digital systems.

1.2.4 Clipping

The input of a digital-to-analog converter is an integer, which implies there are minimum and maximum values for samples. To make calculations simpler and to avoid losing precision, we often use floating point numbers to represent and process samples.[2] For example, we might form the sum of many audio channels, exceeding the maximum sample value, but before converting to analog, we can bring the sum into range with a suitable scale factor. In this example, integer arithmetic would overflow, but floating point numbers would not.

By using floating point representations for samples, we avoid many problems of precision and numerical overflow, but we still face problems when we try to convert samples to sound because there are no floating-point converters, and even if there were, analog circuits and amplifiers have limits. In practice, sample values are limited or "clipped" to a fixed range. This results in a particularly harsh form of distortion called *clipping*. Clipping occurs when we write too large samples to audio files or when we directly write too large samples to a DAC. (Clipping can also occur in audio recording when the input signal is too high for analog circuitry or outside the range of an analog-to-digital converter.)

There is no general remedy for clipping. Once a signal has been clipped, the original cannot be recovered. There are various ways to soften the distortion, but the only "respectable" fix is to adjust amplitudes by scaling, mixing, or even moving the microphone[3]. Figure 1.5 shows a waveform with clipping.

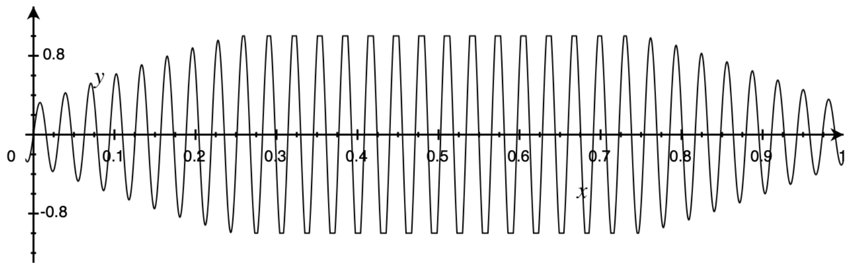

Figure 1.5: A sinusoid with increasing/decreasing volume showing audio clipping at the center of the graph.

[2]A *floating point* number is a digital representation that uses the idea of scientific notation, e.g. 0.1234×10^7, by storing a sign, a fraction, and an exponent, giving a much larger range of values than an integer of the same size.

[3]"Louis Armstrong was famously placed 20 feet away for his solos." – Clive Thompson in www.smithsonianmag.com/arts-culture/phonograph-changed-music-for forever-180957677/

This pdf for ICM students only - ebook
and paperback available from amazon.com

1.2.5 The Table-Lookup Oscillator

Although we still have a lot to learn about digital audio, it is useful to look at a very concrete example of how we can use computers to *synthesize* audio in addition to simply recording and playing it back. One of the simplest synthesis algorithms is the table-lookup oscillator. The goal is to create a periodic, repeating signal. Ideally, we should have control over the amplitude and frequency, because scaling the amplitude makes the sound quieter or louder, and making the frequency higher makes the pitch higher.

To get started, consider Algorithm 1.1, which simply outputs samples sequentially and repeatedly from a table that stores one period of the repeating signal.

```
// create a table with one period of the signal:
table = [0, 0.3, 0.6, 0.9, 0.6, 0.3,
         0, -0.3, -0.4, -0.9, -0.6, -0.3]
// output audio samples:
repeat 10000 times:
    for each element of table:
        write(element)
```

Algorithm 1.1: Simple table-lookup oscillator

This will produce 120,000 samples, or about 3 seconds of audio if the sample rate is 44,100 samples per second. The frequency (rate of repetition in the signal) will be $44,100/12 = 3675$, which is near the top note of the piano. We want very fine control over frequency, so this simple algorithm with integer-length repeating periods is not adequate. Instead, we need to use some kind of *interpolation* to allow for fractional periods. This approach is shown in Algorithm 1.2.

```
// create a table with one period of the signal:
table = [0, 0.3, 0.6, 0.9, 0.6, 0.3,
         0, -0.3, -0.4, -0.9, -0.6, -0.3]
// make a variable to keep track of phase:
phase = 0.0
// increment phase by this to get 440 Hz:
phase_incr = 440 * len(table) / 44100.0
// output audio samples:
repeat 44100 * 10 times: // 10 seconds of audio
    i1 = floor(phase) // integer part of phase, first sample index
    frac = phase - i1 // fractional part of phase
    i2 = (i1 + 1) mod len(table) // index of next sample in table
    // linearly interpolate between two samples in the table:
    y = (1 - frac) * table[i1] + frac * table[i2]
    write(y * amplitude)
    // increment phase and wrap around when we reach the end of the table
    phase = (phase + phase_incr) mod len(table)
```

Algorithm 1.2: Interpolating table-lookup oscillator

This code example is considerably longer than the first one. It uses the variable `phase` to keep track of how much of the waveform period we have output so far. After each sample is output, `phase` is incremented by `phase_incr`, which is initialized so that `phase` will reach the table length and wrap around to zero (using the `mod` operator) 440 times per second. Since `phase` is now fractional,

This pdf for ICM students only - ebook
and paperback available from amazon.com

we cannot simply write `table[phase]` to look up the value of the waveform at location `phase`. Instead, we read *two* adjacent values from the table and form a weighted sum based on the fractional part (`frac`) of `phase`. Even though the samples may not exactly repeat due to interpolation, we can control the overall frequency (or repetition rate) at which we sweep through the table very precisely.

In addition, this version of the code multiplies the computed sample (`y`) by `amplitude`. This scale factor gives us control over the overall amplitude, which is related to the loudness of the resulting sound.

In practice, we would normally use a much larger table, e.g. 2048 elements, and we would use a smoother waveform. (We will talk about why digital audio waveforms have to be smooth later.) It is common to use this technique to generate sinusoids. Of course, you *could* just call $sin(phase)$ for every sample, but in most cases, pre-computing values of the sin function and saving them in a table, then reading the samples from memory, is much faster than computing the sin function once per sample.

Instead of synthesizing sinusoids, we can also synthesize complex waveforms such as triangle, sawtooth, and square waves of analog synthesizers, or waveforms obtained from acoustic instruments or human voices.

We will learn about many other synthesis algorithms and techniques, but the table-lookup oscillator is a computationally efficient method to produce sinusoids and more complex periodic signals. Besides being efficient, this method offers direct control of amplitude and frequency, which are very important control parameters for making music. The main drawback of table-lookup oscillators is that the waveform or wave shape is fixed, whereas most musical tones vary over time and with amplitude and frequency. Later, we will see alternative approaches to sound synthesis and also learn about filters, which can be used to alter wave shapes.

1.3 Nyquist, SAL, Lisp

Nyquist[4] is a language for sound synthesis and music composition. Unlike score languages that tend to deal only with events, or signal processing languages that tend to deal only with signals and synthesis, Nyquist handles both in a single integrated system. Nyquist is also flexible and easy to use[5] because it is based on an interactive Lisp interpreter (XLisp).

The NyquistIDE program (Figure 1.6) combines many helpful functions and interfaces to help you get the most out of Nyquist. NyquistIDE is implemented in Java, and you will need the Java runtime system or development system installed on your computer to use NyquistIDE. The best way to learn about NyquistIDE is to just use it. NyquistIDE helps you by providing a Lisp and SAL editor, hints for command completion and function parameters, some graphical interfaces for editing envelopes and graphical equalizers, and a panel of buttons for common operations.

1.3.1 SAL

Nyquist is based on the Lisp language. Many users found Lisp's syntax unfamiliar, and eventually Nyquist was extended with support for SAL, which is similar in semantics to Lisp, but similar in syntax to languages such as Python

[4]The *Nyquist Reference Manual* is included as PDF and HTML in the Nyquist download; also available online: www.cs.cmu.edu/~rbd/doc/nyquist

[5]All language designers tell you this. Don't believe any of them.

This pdf for ICM students only - ebook
and paperback available from amazon.com

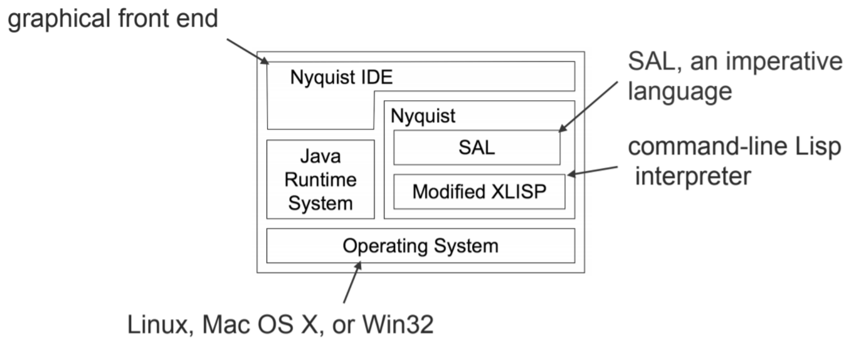

Figure 1.6: NyquistIDE System Architecture

and Javascript. The NyquistIDE supports two modes, Lisp and SAL. SAL mode means that Nyquist reads and evaluates SAL commands rather than Lisp. The SAL mode prompt is "`SAL> `" while the Lisp mode prompt is "`> `". When Nyquist starts, it normally enters SAL mode automatically, but certain errors may exit SAL mode. You can reenter SAL mode by typing the Lisp expression `(sal)` or finding the button labeled SAL in the IDE.

In SAL mode, you type commands in the SAL programming language. Nyquist reads the commands, compiles them into Lisp, and evaluates the commands. Some examples of SAL commands are the following:

- `print` *expression* – evaluate and expression and print the result.
- `exec` *expression* – evaluate expression but do not print the result.
- `play` *expression* – evaluate an expression and play the result, which must be a sound.
- `set` *var* = *expression* – set a variable.

1.4 Using SAL In the IDE

It is important to learn to use the NyquistIDE program, which provides an interface for editing and running Nyquist programs. The NyquistIDE is discussed in the *Nyquist Reference Manual*. You should take time to learn:

- How to switch to SAL mode. In particular, you can "pop" out to the top level of Nyquist by clicking the "Top" button; then, you can enter SAL mode by clicking the "SAL" button.
- How to run a SAL command, e.g. type `print "hello world"` in the input window at the upper left.
- How to create a new file. In particular, you should normally *save* a new empty file to a file named *something*`.sal` in order to tell the editor this is a SAL file and thereby invoke the SAL syntax coloring, indentation support, etc.
- How to execute a file by using the *Load* menu item or keyboard shortcut.

This pdf for ICM students only - ebook
and paperback available from amazon.com

1.5 Examples

This would be a good time to install and run the NyquistIDE. You can find Nyquist downloads on sourceforge.net/projects/nyquist, and "readme" files contain installation guidelines.

The program named NyquistIDE is an "integrated development environment" for Nyquist. When you run NyquistIDE, it starts the Nyquist program and displays all Nyquist output in a window. NyquistIDE helps you by providing a Lisp and SAL editor, hints for command completion and function parameters, some graphical interfaces for editing envelopes and graphical equalizers, and a panel of buttons for common operations. A more complete description of NyquistIDE is in Chapter "The NyquistIDE Program" in the *Nyquist Reference Manual*.

For now, all you really need to know is that you can enter Nyquist commands by typing into the upper left window. When you type return, the expression you typed is sent to Nyquist, and the results appear in the window below. You can edit files by clicking on the New File or Open File buttons. After editing some text, you can load the text into Nyquist by clicking the Load button. NyquistIDE always saves the file first; then it tells Nyquist to load the file. You will be prompted for a file name the first time you load a new file.

Try some of these examples. These are SAL commands, so be sure to enter SAL mode. Then, just type these one-by-one into the upper left window.

```
play pluck(c4)

play pluck(c4) ~ 3

play piano-note(5, fs1, 100)

play osc(c4)

play osc(c4) * osc(d4)

play pluck(c4) ~ 3

play noise() * env(0.05, 0.1, 0.5, 1, 0.5, 0.4)
```

1.6 Constants, Variables and Functions

As in XLISP, simple constant value expressions include:

- integers (FIXNUM's), such as `1215`,
- floats (FLONUM's) such as `12.15`,
- strings (STRING's) such as `"Magna Carta"`,
- symbols (SYMBOL's) can be denoted by `quote(`*name*`)`, e.g. symbol `FOO` is denoted by `quote(foo)`. Think of symbols as unique strings. Every time you write `quote(foo)`, you get *exactly* the same identical value. Symbols in this form are not too common, but "raw" symbols, e.g. `foo`, are used to denote values of variables and to denote functions.

Additional constant expressions in SAL are:

- *lists* such as `{c 60 e 64}`. Note that there are no commas to separate list elements, and symbols in lists are not evaluated as variables but stand for themselves. Lists may contain numbers, booleans (which represent XLisp's `T` or `nil`, SAL identifiers (representing XLisp symbols), strings, SAL operators (representing XLisp symbols), and nested lists.

This pdf for ICM students only - ebook
and paperback available from amazon.com

- *Booleans*: SAL interprets `#t` as true and `#f` as false. (But there is also the variable `t` to indicate "true," and `nil` to indicate "false." Usually we use these cleaner and prettier forms instead of `#t` and `#f`.

A curious property of Lisp and Sal is that false and the empty list are the same value. Since SAL is based on Lisp, `#f` and `{}` (the empty list) are equal.

Variables are denoted by symbols such as `count` or `duration`. Variable names may include digits and the characters `- + * $ ~ ! @ # % ^ & \ : < > . / ? _`; however, it is strongly recommended to avoid special characters when naming variables and functions. One exception is that the dash (`-`) is used to create compound names.

Recommended form: `magna-carta`, `phrase-len`; *to be avoided*: `magnaCarta`, `magna_carta`, `magnacarta`, `phraseLen`, `phrase_len`, `phraselen`.

SAL and Lisp convert all variable letters to upper case, so `foo` and `FOO` and `Foo` all denote the same variable. The preferred way to write variables and functions is in *all lower case*. (There are ways to create symbols and variables with lower case letters, but this should be avoided.)

A symbol with a leading colon (:) evaluates to itself. E.g. `:foo` has the value `:FOO`. Otherwise, a symbol denotes either a local variable, a formal parameter, or a global variable. As in Lisp, variables do not have data types or type declarations. The type of a variable is determined at runtime by its value.

Functions in SAL include both operators, e.g. `1 + 2` and standard function notation, e.g. `sqrt(2)`. The most important thing to know about operators is that *you must separate operators from operands with white space*. For example, `a + b` is an expression that denotes "a plus b", but `a+b` (no spaces) denotes the value of a variable with the unusual name of "`A+B`".

Functions are invoked using what should be familiar notation, e.g. `sin(pi)` or `max(x, 100)`. Some functions (including `max`) take a variable number of arguments. Some functions take keyword arguments, for example

```
string-downcase("ABCD", start: 2)
```

returns `ABcd` because the keyword parameter `start: 2` says to convert to lower case starting at position 2.

1.7 Defining Functions

Before a function can be called from an expression (as described above), it must be defined. A function definition gives the function name, a list of parameters, and a statement. When a function is called, the actual parameter expressions are evaluated from left to right and the formal parameters of the function definition are set to these values. Then, the function body, a statement, is evaluated. The syntax to define functions in SAL is:

```
[ define ] function name ( [parameter {, parameter }*] )
           statement
```

This syntax *meta*-notation uses brackets [...] to denote optional elements and braces with a star {...}* to denote zero or more repetitions, but you do not literally write brackets or braces. *Italics* denote place-holders, e.g. *name* means you write the name of the function you are defining, e.g. `my-function` (remember names in SAL can have hyphens). Beginning a function definition with the keyword `define` is optional, so a minimal function definition is:

This pdf for ICM students only - ebook
and paperback available from amazon.com

```
function three() return 3
```

Note that space and newlines are ignored, so that could be equivalently written:

```
function three()
  return 3
```

A function with two parameters is:

```
function simple-adder(a, b) return a + b
```

The formal parameters may be positional parameters that are matched with actual parameters by position from left to right. Syntactically, these are symbols and these symbols are essentially local variables that exist only until *statement* completes or a *return* statement causes the function evaluation to end. As in Lisp, parameters are passed by value, so assigning a new value to a formal parameter has no effect on the caller. However, lists and arrays are not copied, so internal changes to a list or array produce observable side effects.

Alternatively, formal parameters may be keyword parameters. Here the parameter is actually a pair: a keyword parameter, which is a symbol followed by a colon, and a default value, given by any expression. Within the body of the function, the keyword parameter is named by a symbol whose name matches the keyword parameter except there is no final colon.

```
define function foo(x: 1, y: bar(2, 3))
    display "foo", x, y

exec foo(x: 6, y: 7)
```

In this example, `x` is bound to the value 6 and `y` is bound to the value 7, so the example prints "`foo : X = 6, Y = 7`". Note that while the keyword parameters are `x:` and `y:`, the corresponding variable names in the function body are `x` and `y`, respectively.

The parameters are meaningful only within the lexical (static) scope of *statement*. They are not accessible from within other functions even if they are called by this function.

Use a `begin-end` compound statement if the body of the function should contain more than one statement or you need to define local variables. Use a `return` statement to return a value from the function. If statement completes without a `return`, the value false (`nil`) is returned.

See the *Nyquist Reference Manual* for complete information and details of `begin-end`, `return`, and other statements.

1.8 Simple Commands

1.8.1 exec

`exec` *expression*
Unlike most other programming languages, you cannot simply type an expression as a statement. If you want to evaluate an expression, e.g. call a function, you must use an `exec` statement. The statement simply evaluates the *expression*. For example,

```
exec set-sound-srate(22050.0)   ; change default sample rate
```

This pdf for ICM students only - ebook
and paperback available from amazon.com

1.8.2 load

`load` *expression*
The `load` command loads a file named by *expression*, which must evaluate to a string path name for the file. To load a file, SAL interprets each statement in the file, stopping when the end of the file or an error is encountered. If the file ends in `.lsp`, the file is assumed to contain Lisp expressions, which are evaluated by the XLISP interpreter. In general, SAL files should end with the extension .sal.

1.8.3 play

`play` *expr*
The `play` statement plays the sound computed by *expr*, an expression.

1.8.4 plot

`plot` *expr* `,` *dur* `,` *n*
The `plot` statement plots the sound denoted by *expr*, an expression. If you plot a long sound, the plot statement will by default truncate the sound to 2.0 seconds and resample the signal to 1000 points. The optional *dur* is an expression that specifies the (maximum) duration to be plotted, and the optional *n* specifies the number of points to be plotted. Executing a `plot` statement is equivalent to calling the `s-plot` function.

1.8.5 print

`print` *expr* `,` *expr ...*
The `print` statement prints the values separated by spaces and followed by a newline. There may be 0, 1, or more expressions separated by commas (,).

1.8.6 display

`display` *string*, *expression* `,` *expression ...*
The `display` statement is handy for debugging. When executed, `display` prints the *string* followed by a colon (:) and then, for each *expression*, the expression and its value are printed; after the last expression, a newline is printed. For example,

```
display "In function foo", bar, baz
```

prints

```
In function foo : bar = 23, baz = 5.3
```

SAL may print the expressions using Lisp syntax, e.g. if the expression is "`bar + baz`," do not be surprised if the output is:

```
(sum bar baz) = 28.3
```

1.8.7 set

`set` *var* `op` *expression*
The `set` statement changes the value of a variable *var* according to the operator *op* and the value of *expression*. The operators are:

`=` The value of expression is assigned to var.

`+=` The value of *expression* is added to *var*.

This pdf for ICM students only - ebook
and paperback available from amazon.com

`*=` The value of *var* is multiplied by the value of the *expression*.

`&=` The value of *expression* is inserted as the last element of the list referenced by *var*. If *var* is the empty list (denoted by `nil` or `\#f`), then *var* is assigned a newly constructed list of one element, the value of *expression*.

`^=` The value of *expression*, a list, is appended to the list referenced by *var*. If *var* is the empty list (denoted by `nil` or `\#f`), then *var* is assigned the (list) value of *expression*.

`@=` Pushes the value of *expression* onto the front of the list referenced by *var*. If *var* is empty (denoted by `nil` or `\#f`), then *var* is assigned a newly constructed list of one element, the value of *expression*.

`<=` Sets the new value of *var* to the minimum of the old value of *var* and the value of *expression*.

`>=` Sets the new value of *var* to the maximum of the old value of *var* and the value of *expression*.

The `set` command can also perform multiple assignments separated by commas (,):

```
; example from Rick Taube's SAL description
loop
  with a, b = 0, c = 1, d = {}, e = {}, f = -1, g = 0
  for i below 5
  set a = i, b += 1, c *= 2, d &= i, e @= i, f <= i, g >= i
  finally display "results", a, b, c, d, e, f, g
end
```

1.9 Control Constructs

1.9.1 begin end

A `begin-end` statement consists of a sequence of statements surrounded by the `begin` and `end` keywords. This form is often used for function definitions and after `then` or `else` where the syntax demands a single statement but you want to perform more than one action. Variables may be declared using an optional `with` statement immediately after `begin`. For example:

```
begin
  with db = 12.0,
       linear = db-to-linear(db)
  print db, "dB represents a factor of", linear
  set scale-factor = linear
end
```

1.9.2 if then else

`if` *expression* `then` *statement* [`else` *statement*]
An `if` statement evaluates a test expression. If it is true, it evaluates the statement following `then`. If false, the statement after `else` is evaluated. Use a `begin-end` statement to evaluate more than one statement in `then` or `else` parts.

Here are some examples...

```
if x < 0 then x = -x  ; x gets its absoute value
```

This pdf for ICM students only - ebook
and paperback available from amazon.com

```
if x > upper-bound then
  begin
    print "x too big, setting to", upper-bound
    x = upper-bound
  end
else
  if x < lower-bound then
    begin
      print "x too small, setting to", lower-bound
      x = lower-bound
    end
```

Notice in this example that the `else` part is another `if` statement. An `if` may also be the `then` part of another `if`, so there could be two possible `if`'s with which to associate an `else`. An `else` clause always associates with the closest previous `if` that does not already have an `else` clause.

1.9.3 loop

The `loop` statement is by far the most complex statement in SAL, but it offers great flexibility for just about any kind of iteration. However, when computing sounds, *loops are generally the wrong approach*, and there are special functions such as `seqrep` and `simrep` to use iteration to create sequential and simultaneous combinations of sounds as well as special functions to iterate over *scores*, apply synthesis functions, and combine the results.

Therefore, loops are mainly for imperative programming where you want to iterate over lists, arrays, or other discrete structures. You will probably need loops at some point, so at least scan this section to see what is available, but there is no need to dwell on this section for now.

The basic function of a loop is to repeatedly evaluate a sequence of actions which are statements. The syntax for a `loop` statement is:

`loop` [*with-stmt*] { *stepping* }* { *stopping* }* { *action* }+
    [*final*] `end`

Before the loop begins, local variables may be declared in a `with` statement.

The *stepping* clauses do several things. They introduce and initialize additional local variables similar to the `with` statement. However, these local variables are updated to new values after the actions. In addition, some *stepping* clauses have associated stopping conditions, which are tested on each iteration before evaluating the actions.

There are also *stopping* clauses that provide additional tests to stop the iteration. These are also evaluated and tested on each iteration before evaluating the actions.

When some *stepping* or *stopping* condition causes the iteration to stop, the *final* clause is evaluated (if present). Local variables and their values can still be accessed in the *final* clause. After the *final* clause, the `loop` statement completes.

The *stepping* clauses are the following:

`repeat` *expression*

sets the number of iterations to the value of *expression*, which should be an integer (FIXNUM).

`for` *var* = *expression* `[ then` *expr2*]

This pdf for ICM students only - ebook
and paperback available from amazon.com

introduces a new local variable named *var* and initializes it to *expression*. Before each subsequent iteration, *var* is set to the value of *expr2*. If the `then` part is omitted, *expression* is re-evaluated and assigned to *var* on each subsequent iteration. Note that this differs from a `with` statement where expressions are evaluated and variables are only assigned their values once.

`for` *var* `in` *expression*

evaluates *expression* to obtain a list and creates a new local variable initialized to the first element of the list. After each iteration, *var* is assigned the next element of the list. Iteration stops when *var* has assumed all values from the list. If the list is initially empty, the `loop` actions are not evaluated (there are zero iterations).

`for` *var* `[ from` *from-expr* `] [ [ to | below | downto | above ]`
to-expr `] [ by` *step-expr* `]`

is yet another *stepping* clause. Note that here we have introduced a new meta-syntax notation: [*term1* | *term2* | *term3*] means a valid expression contains one of *term1*, *term2*, or *term3*.

This `for` clause introduces a new local variable named *var* and initialized to the value of the expression *from-expr* (with a default value of 0). After each iteration of the `loop`, *var* is incremented by the value of *step-expr* (with a default value of 1). The iteration ends when *var* is greater than the value of *to-expr* if there is a `to` clause, greater than or equal to the value of *to-expr* if there is a `below` clause, less than the value of *to-expr* if there is a `downto` clause, or less than or equal to the value of *to-expr* if there is an `above` clause. (In the cases of `downto` and `above`, the default increment value is -1. If there is no `to`, `below`, `downto`, or `above` clause, no iteration stop test is created for this *stepping* clause.)

The *stopping* clauses are the following:

`while` *expression*

The iterations are stopped when *expression* evaluates to false. Anything not false is considered to be true.

`until` *expression*

The iterations are stopped when *expression* evaluates to anything that is not false (nil).

The loop *action* consists of one or more SAL statements (indicated by the "+" in the meta-syntax).
The *final* clause is defined as follows:

`finally` *statement*

The statement is evaluated when one of the *stepping* or *stopping* clauses ends the `loop`. As always, *statement* may be a `begin-end` statement. If an *action* in the loop body evaluates a `return` statement, the `finally` statement is not executed. Loops often fall into common patterns, such as iterating a fixed number of times, performing an operation on some range of integers, collecting results in a list, and linearly searching for a solution. These forms are illustrated in the examples below.

```
; iterate 10 times
loop
  repeat 10
  print random(100)
end
```

This pdf for ICM students only - ebook
and paperback available from amazon.com

```
 ; print even numbers from 10 to 20
 ; note that 20 is printed. On the next iteration,
 ;   i = 22, so i >= 22, so the loop exits.
loop
  for i from 10 to 22 by 2
  print i
end
```

```
 ; collect even numbers in a list
loop
  with lis
  for i from 0 to 10 by 2
  set lis @= i  ; push integers on front of list,
                ; which is much faster than append,
                ; but list is built in reverse
  finally set result = reverse(lis)
end
```

```
 ; now, the variable result has a list of evens
 ; find the first even number in a list
result = #f  ; #f means "false"
loop
  for elem in lis
  until evenp(elem)
  finally result = elem
end
 ; result has first even value in lis (or it is #f)
```

1.9.4 simrep Example

We can define function `pluck-chord` as follows:

```
function pluck-chord(pitch, interval, n)
  begin
    with s = pluck(pitch)
    loop
      for i from 1 below n
      set s += pluck(pitch + interval * i)
    end
  return s
end

play pluck-chord(c3, 5, 2)
play pluck-chord(d3, 7, 4) ~ 3
play pluck-chord(c2, 10, 7) ~ 8
```

But we mentioned earlier that loops should not normally be used to compute sounds. Just to preview what is coming up ahead, here is how `pluck-chord` should be written:

```
function pluck-chord(pitch, interval, n)
  return simrep(i, n, pluck(pitch + i * interval))

play pluck-chord(c3, 5, 2)
play pluck-chord(d3, 7, 4) ~ 3
play pluck-chord(c2, 10, 7) ~ 8
```

This pdf for ICM students only - ebook
and paperback available from amazon.com

Note that this version of the function is substantially smaller (`loop` is powerful, but sometimes a bit verbose). In addition, one could argue this `simrep` version is more correct – in the case where `n` is 0, this version returns silence, whereas the `loop` version always initializes `s` to a `pluck` sound, even if `n` is zero, so it never returns silence. The `simrep` construct and its cousin, `seqrep`, are described in the *Nyquist Reference Manual*, and we will return to them in Section **??**.