

Indiana University Jacobs School of Music K509 – Seminar in Computer Music

RTcmix Tutorial

RTcmix is a text-based program that generates or processes sound, usually in real time. It is not like the programs we're used to seeing. For example, it does not have a GUI (graphical user interface) — knobs, sliders, buttons, etc. Instead, you write a script in a simple programming language, called “MinC,” and feed that to RTcmix, which then makes sound and/or writes a sound file.

The advantage of this approach is that you can design an *algorithm* — a set of specific instructions that perform a task — and have the algorithm generate sound. This is a very different way of working than the way you work with GUI-based software such as Pro Tools and Absynth, and it opens up a number of sonic possibilities that would not otherwise be available.

RTcmix runs on any computer that uses either the Mac OS X or Linux operating systems. The source code for the program (written in C and C++) is free. The code, as well as documentation, is available at <http://rtcmix.org>.

This tutorial is designed to get you started running and writing RTcmix scripts (or “scores,” as we usually call them).

Preliminaries

The main way to work with RTcmix is to use “the shell,” which is a command-line interface to the operating system of the computer. This takes advantage of the Unix underpinnings of OS X (and Linux, for that matter), and is a way to work very efficiently, once you learn how. The disadvantage is that it will strike most people as unintuitive.

This tutorial assumes a basic familiarity with the OS X Terminal program and shell commands, such as **cd**, **ls**, **rm**, **mv**, **mkdir**, **cat**, etc.) For a review, consult <http://iub.edu/~emusic/unix.html>.

1. Launch the Terminal program from your Dock.
2. Once we start using RTcmix, we'll have to view and edit plain text files, because that's what RTcmix scores are.

`edit blah` (open a new file, “blah,” in the TextWrangler text-editing program)

Type some random stuff into the TextWrangler window, and save the file. Then return to your shell window in the Terminal program. (Type Apple-tab to get there without using the mouse.) Give these commands:

`ls` (you'll see the “blah” file among those listed)
`cat blah` (dump your text file to the screen)
`rm -f blah` (delete “blah” without asking for confirmation)
`clear` (clear the screen)

If you know a Unix text editor, such as **vim**, **pico** or **emacs**, feel free to use that. Anything that edits plain ASCII text is fine.

- Here's one tip that helps when using long file names in the shell. Type a few letters of the file name, and press the tab key: the shell fills in the remaining letters.

```
cd          (cd without a directory name means to change to your home directory)
cd Do <tab>
```

You'll see the name expand to "Documents," as long as you don't have any other file names in the working directory that start with "Do."

- The RTcmix sample scores on the CECM studio computers are located in this directory:

```
/Users/Shared/rtcmix
```

Before starting work, please copy this directory into your home directory. Inside is one directory for scores and one for sample sound files used in the tutorial. If you're working in the music library or at home, you can download an archive of (most of) this directory from

```
<http://iub.edu/~emusic/rtcmix/rtcmix\_examples.zip>
```

Double-click the .zip file to unpack it.

Binary packages of RTcmix, for installation on your own computer, are available for PPC and Intel Macs at

```
<http://iub.edu/~emusic/rtcmix/pkg/>
```

If you're a programmer geek, you can download the source code and compile it. For Linux, see "programmer geek" above. Sorry, we don't do Windows (yet).

Running Scores

Now we'll learn how to view and run RTcmix scores.

- Open a Terminal window, and type this:

```
cd rtcmix/sco
```

If you see an error message like "no such file or directory," try it again, and be careful about all the slashes and letters. The only space is right after "cd." (This assumes that you've already dragged a copy of the rtcmix example directory into your home directory.)

- Issue the **ls** command, and you should see some file names ending in a ".sco" suffix. These are RTcmix MinC scores.
- In our studios, RTcmix sound will play out of the MOTU audio interface (unless you do something special to make it play out of the Digidesign interface).

CAUTION: When you're running RTcmix scores, especially ones that you're working on, you should turn your monitoring volume down a bit. Sometimes you

might get unexpected loud sounds that could damage the speakers and your ears. Turn it up once you're sure of what will come out.

4. Run a score...

```
cmix < falias.sco
```

You should hear some sound and see some stuff printed to the screen.

When the score is finished, you'll see a new command prompt. If you want to stop playback before the score is finished, type control-c.

If you're wondering about the use of the '<' character in the command above, you can learn all about it here ...

```
<http://www.december.com/unix/tutor/redirect.html>
```

Or you can think of it as telling the shell, "feed the 'falias.sco' file to the **cmix** program."

5. View the score...

```
edit falias.sco
```

Okay, it looks like gibberish. The next section helps you understand scores like this.

Writing Scores

Here is a very simple, and very boring, RTcmix score.

```
rtsetparams(44100, 2)
load("WAVETABLE")
WAVETABLE(0, 10, 10000, 440, 0.5)
```

This plays a beautiful sine wave at A440 for ten seconds.

(All the scores in this tutorial are in your `rtcmix/sco/tut` directory. You should **cd** into that directory and run them while following the tutorial. The first one is called "tut1.sco.")

The score consists of three *function calls*. A *function* is something that performs an action, often taking *arguments* (or *parameters*) that specify details of the action. The arguments are in parentheses following the function name and are separated by commas.

Let's look at this score line by line.

```
rtsetparams(44100, 2)
```

The `rtsetparams` function sets up the sampling rate (44100) and the number of output channels (2) for this score.

```
load("WAVETABLE")
```

The `load` function specifies an *instrument* plug-in that we will use. An *instrument* is a chunk of software that makes sound. Instrument names are always in upper case. The name must appear in double quotes as an argument to the `load` function.

```
WAVETABLE(0, 10, 10000, 440, 0.5)
```

Here's the part that makes the sound. The `WAVETABLE` instrument performs wavetable synthesis. It takes several arguments that specify the start time in seconds of the note (0), the duration in seconds (10), the amplitude (10000), the frequency in Hz (440) and the stereo location (0.5). These arguments must appear in this order, from left to right. Amplitudes for `WAVETABLE` and other synthesis instruments fall between 0 and 32767, which is the maximum amplitude for a 16-bit audio signal (even though the internal sample word length is 32 bits). Pan is from 0 to 1.

Most scores make use of *variables*. These are names to which a value is assigned. Then the names can be used and their values modified in the score. For example, you could have ...

```
homer = 10
marge = 20
bart = homer + marge
print(bart)
```

We assign values to the `homer` and `marge` variables. Then we add together the values of those variables and store the result into the `bart` variable. The `print` function prints the value of `bart` (30) to the screen.

The variable names are arbitrary: they can be nearly anything, as long as they begin with a letter and don't contain weird characters. They probably shouldn't be the same as names of RTcmix functions, just to avoid confusion.

We can use variables to make our first score easier to read and understand.

```
rtsetparams(44100, 2)
load("WAVETABLE")
start = 0
dur = 10
amp = 10000
freq = 440
pan = 0.5
WAVETABLE(start, dur, amp, freq, pan)
```

In the `WAVETABLE` call, RTcmix substitutes the values of the variables for the names that we see in the score. The result is identical to the first score we gave above. Although this one is longer, the clarity is worth the extra typing.

You can also assign values to variables right at the point when they're used.

```
WAVETABLE(start=0, dur=10, amp=10000, freq=440, pan=0.5)
```

This combines the advantages — compactness and clarity — of the previous two approaches.

Playing Multiple Notes, Looping, Randomness

What if you want to play more than one note? Just use WAVETABLE more than once. Play this score (“tut2.sco”).

```
rtsetparams(44100, 2)
load("WAVETABLE")
WAVETABLE(start=0, dur=6, amp=8000, freq=440, pan=0.5)
WAVETABLE(start=0, dur=6, amp=8000, freq=550, pan=1)
WAVETABLE(start=0, dur=6, amp=8000, freq=660, pan=0)
WAVETABLE(start=2, dur=4, amp=12000, freq=220, pan=0.3)
WAVETABLE(start=3, dur=3, amp=10000, freq=330, pan=0.7)
WAVETABLE(start=4, dur=2, amp=10000, freq=800, pan=0.4)
```

You could even write the lines with their start times out of order, and it would still work.

Now let’s have some fun. Instead of specifying the notes one by one, let’s get RTcmix to choose them randomly. We’ll also play them using a *loop*, a very powerful construction for algorithmic composition. Play this score (“tut3.sco”).

```
rtsetparams(44100, 2)
load("WAVETABLE")

for (start = 0; start < 10; start = start + 1) {
  freq = irand(100, 2000)
  pan = irand(0.2, 0.8)
  WAVETABLE(start, dur=2, 8000, freq, pan)
}
```

Although we hear ten notes, one after another, there is only one WAVETABLE call in the score. But it’s embedded within a *for loop*, which repeats the WAVETABLE note several times. A *for loop* comprises a description of how the loop should work and a block of statements enclosed in curly braces — ‘{’ and ‘}’. Though it’s not necessary, you should indent the block of statements within the braces to make the loop easier to read. The text in parentheses following “for” tells RTcmix how to perform the loop.

```
for (start = 0; start < 10; start = start + 1) {
  block of statements...
}
```

For people who haven’t programmed in C before, here is a description of what the “for” line is asking RTcmix to do ...

1. set **start** to 0
2. make sure that the value of **start** is less than 10 (**start < 10**)
3. perform the block of statements following the open curly brace
4. set **start** to the current value of **start** plus 1 (**start = start + 1**)

5. make sure that `start` is less than 10
6. perform the block of statements following the open curly brace
7. increase the value of `start` by one again
8. make sure that `start` is less than 10
9. perform the block of statements following the open curly brace
10. keep doing this until the value of `start` is no longer less than 10
11. exit the loop

Computers are really dumb. They will do this sort of thing all day if you let them. That's why we give them the drudge work: we can make RTcmix play thousands of notes, and we only have to type the `WAVETABLE` line once.

Within the block of statements performed by the loop, we set the value of some variables to *random numbers*. This is a way of letting the computer make some of the decisions about how the notes will sound, because — like, you know — we're lazy. We use the `irand` function to generate a random number within a given range, and then assign this number to a variable.

```
freq = irand(100, 2000)
```

Here we ask `irand` to *return* a random value between 100 and 2000. We assign that as the value of `freq`, which we later feed to the `WAVETABLE` instrument call. We do something similar for `pan`.

Envelope Tables

There's one major problem with our score: it clicks. In Pro Tools we use fades to smooth out click-producing discontinuities at the boundaries of a region's waveform. We do this in RTcmix by using an amplitude envelope *table*. Here's how it works ("tut4.sco") ...

```
rtsetparams(44100, 2)
load("WAVETABLE")

env = maketable("line", size=1000, 0,0, 1,1, 9,1, 10,0)

for (start = 0; start < 10; start = start + 1) {
    freq = irand(100, 2000)
    pan = irand(0.2, 0.8)
    WAVETABLE(start, dur=2, 8000 * env, freq, pan)
}
```

You create an envelope table using the `maketable` function. This just writes a bunch of numbers into a table, in such a way as to describe a shape that changes over time. Our shape is a simple attack / release ramp. We store the table into the `env` variable, and then use it within the call to `WAVETABLE`. Instead of using a constant number (8000) as the amplitude for the note, we multiply that number by the changing numbers stored in the table (`8000 * env`). By default, these numbers are between 0 and 1, so they act as a variable volume control, scaling between 0 and 8000 the amplitude that `WAVETABLE` sees.

Here's how the `maketable` function works in detail.

```
env = maketable("line", table_size, time1,value1, time2,value2, ...)
```

The type of table we want is called “line.” (There are many other types.) You specify the size of the table — the number of values in the table. Then you give two or more time-and-value pairs that set the position of breakpoints in the envelope — points connected by straight line segments. The times are in seconds and must be ascending, from left to right.

```
env = maketable("line", size=1000, 0,0, 1,1, 9,1, 10,0)
```

We have 1000 numbers in the table. They start at 0 and increase to 1 over the first second. Then at 9 seconds, the numbers decrease to 0 until the table ends at second 10.

IMPORTANT:

When you give a table to an instrument, the time it takes is scaled to fit the duration of the note played by the instrument. Our envelope lasts ten seconds, but our note lasts only two, so the envelope shrinks to fit two seconds.

We can use the same type of table to create a glissando. Play this score (“tut5.sco”), which is the same as the last, except for an added glissando table.

```
rtsetparams(44100, 2)
load("WAVETABLE")

env = maketable("line", size=1000, 0,0, 1,1, 9,1, 10,0)
gliss = maketable("line", "nonorm", size, 0,1, 1,2)

for (start = 0; start < 10; start = start + 1) {
  freq = irand(100, 2000)
  pan = irand(0.2, 0.8)
  WAVETABLE(start, dur=2, 8000 * env, freq * gliss, pan)
}
```

We multiply the frequency by a table that represents a straight line from 1 to 2, which results in a glissando from the initial randomly-generated frequency, at the beginning of the note, to an octave above that, at the end of the note. The extra “nonorm” argument for the `gliss` table is required to keep RTcmix from normalizing (scaling) the table so that it fits between 0 and 1.

Specifying Waveforms

We’ve been playing sine waves. To get other waveforms, you need to provide the `WAVETABLE` instrument with a particular wavetable, created by `maketable`. You supply this table as the seventh argument to the instrument, following the pan argument. Play this score (“tut6.sco”).

```
rtsetparams(44100, 2)
load("WAVETABLE")
dur = 2
freq = 300
env = maketable("line", 1000, 0,0, 1,1, 3,1, 4,0)
pan = 0.5

wavetable = maketable("wave", size=1000, "square")
WAVETABLE(start=0, dur, 10000 * env, freq, pan, wavetable)

wavetable = maketable("wave", size=1000, "buzz")
WAVETABLE(start=2, dur, 10000 * env, freq, pan, wavetable)

wavetable = maketable("wave", size=1000, 1, 0, .5, 0, .3, 0, .1, 0, 0, .7)
WAVETABLE(start=4, dur, 10000 * env, freq, pan, wavetable)
```

Here we play three consecutive notes, each with a different wavetable. You create a wavetable using the “wave” type for `maketable`. There are two options:

- give the name of a common waveform, such as “square,” “buzz” or “saw”; or
- give a numerical specification of the amplitudes of harmonic partials.

Our third note uses the numeric option. Following the table size, there can be any number of arguments, which specify the amplitudes (from 0 to 1) of successively higher *harmonic* partials: the first is the fundamental, the second is the second harmonic partial, the third is the third harmonic partial, etc. Our waveform has the fundamental at full strength, the third partial at .5, the fifth partial at .3, the seventh partial at .1, the tenth partial at .7, and all the other partials at zero. If you want *non-harmonic* partials, use the “wave3” type of `maketable`, which lets you specify a partial using a decimal point (e.g., partial 2.1, for something a little higher than the second harmonic partial), and a specific amplitude and phase. Or simply play several `WAVETABLE` notes at the same time, with the right frequencies.

We can change the wavetable within a loop, and we can specify the partial amplitudes using random numbers. Each note then has a slightly different wavetable. Play the score on the next page (“tut7.sco”).

```

rtsetparams(44100, 2)
load("WAVETABLE")
total_dur = 10
dur = 0.08
env = maketable("line", size=1000, 0,0, .01,1, dur-.01,1, dur,0)
amp = 16000
freq = 250
increment = dur * 1.3
control_rate(15000) // increase rate of envelope updates (15000 times per sec)

for (start = 0; start < total_dur; start = start + increment) {
  p1 = random()
  p2 = random()
  p3 = random()
  p4 = random()
  p5 = random()
  p6 = random()
  p7 = random()
  p8 = random()
  wavetable = maketable("wave", size=2000, p1, p2, p3, p4, p5, p6, p7, p8)
  pan = irand(0, 1)
  WAVETABLE(start, dur, amp * env, freq, pan, wavetable)
}

```

We use the `random` function, which returns random numbers between 0 and 1, to supply the amplitudes for the wavetable that is recreated during each iteration of the loop. Each call to `random` returns a new random number. Note that the `random` function takes no arguments, but it still needs the empty parentheses.

Notice that we compute the time increment for the loop — the amount of time between successive notes — in a fancier way than we did before. Now it depends on the note duration. Can you figure out how to make the `increment` variable change randomly while the loop executes? This would give us irregularly timed notes, instead of the robotically quantized notes we now have.

This score has a few other new things. When creating the amplitude envelope, we use the note duration to specify some of the times in the time-and-value pairs. There is also a *comment* in the score. A comment is any text following two slashes (`//`) on a line. You use comments to help you (and other readers) understand and remember things about the score. RTcmix ignores them when computing and playing the score.

The `control_rate` function requires more explanation than the comment in the score provides. The *control rate* is the rate at which control information, such as envelopes, is calculated. Usually, this is much slower than the sampling rate, because it doesn't need to be as fast, and it saves computer processing power to keep it slow. The normal control rate in RTcmix is 2000 times per second. For short synthetic sounds like `WAVETABLE` notes, this is too slow. So we bump it up to 15,000 times per second. If you delete the `control_rate` line and run the score, you'll hear the difference: there's a click at the note boundaries.

Pitch Specification and Computation, Conditionals

We’ve been specifying the pitch of `WAVETABLE` notes as frequencies in Herz. There are other ways. The most common is to use *octave-point-pitch-class* notation. It works like this: you give a decimal number, where the number to the left of the decimal point indicates the octave, and the number to the right of the decimal point gives semitones. Middle C is 8.00. The D above that is 8.02. The B above that is 8.11. The C above that is either 8.12 or 9.00. You can be more precise: 8.015 is a quarter tone above the C \sharp above middle C.

`WAVETABLE` accepts octave-point-pitch-class (aka “oct.pc” or “pch”) numbers directly, as an alternative to frequencies in Herz.

```
WAVETABLE(start, dur, amp, pitch = 9.07, pan)
```

This `WAVETABLE` note plays the G that is a twelfth above middle C.

Sometimes it’s useful to convert octave-point-pitch-class to frequency in Herz, or vice versa. RTcmix has many pitch conversion functions. Here are some.

```
freq = cpspch(9.07)           // convert pch to Hz (cps, or cycles per second)
pitch = pchcps(freq)         // convert back from cps to pch
pitch = pchmidi(60)          // convert MIDI note number to pch
freq = cpslet("C#5")         // convert letter/octave name to Hz
freq = cpslet("B2 +23")      // letter names can have inflection in cents
```

With all of these, notice that the conversion function name is composed of abbreviations of the two pitch formats, with the left one indicating the format to which you’re converting. (In other words, read “cpspch” as “convert to cps from pch.”)

Sometimes we want to specify pitch directly, as we’ve been doing. But you can also construct an algorithm that computes pitches according to a formula. It turns out that when doing this, we need to be careful when subtracting one pitch from another. For example, what will happen if we try to subtract 0.02 (two semitones) from a pitch whose oct.pc value is 8.01? We would get 7.99, which, as an oct.pc value, is a very high pitch. (Keep in mind that 8.12 in oct.pc is an octave above 8.00, and 8.24 is two octaves above 8.00.) The right answer is 7.11, two semitones lower than 8.01. There is an alternative pitch representation, called “linear octaves,” that solves this problem. Unfortunately, linear octaves are cumbersome and unintuitive to use. Instead, we use the handy `pchadd` function to perform addition or subtraction of two pitches in oct.pc notation. Here’s how it works.

```
result = pchadd(8.01, -.02)
print(result)
```

This prints “7.11” to the screen. Notice that in order to subtract a positive number from another one, you need to put the minus sign in front of the second number (with no intervening space).

Here’s a score that uses pitch computation (“tut8.sco”); the melodic line goes up a minor third or down a major second, depending on a random number.

```

rtsetparams(44100, 2)
load("WAVETABLE")

env = maketable("line", size=1000, 0,0, 1,1, 30,0)
control_rate(20000) // need faster envelope updates to avoid clicks

srand(1) // seed the random number generator

increment = 0.14
dur = increment * 0.8

pitch = 7.02 // starting pitch

for (start = 0; start < 15; start = start + increment) {
  if (random() < 0.5) { // if random number is less than 0.5
    transp = 0.03 // set transposition to minor 3rd up
  }
  else { // otherwise...
    transp = -0.02 // set transp to major 2nd down
  }
  pitch = pchadd(pitch, transp)

  decibel = irand(70, 92) // get random amp between 70 and 92 dB
  amp = ampdb(decibel) // convert dB to linear amplitude
  pan = irand(0.1, 0.9) // get random pan

  WAVETABLE(start, dur, amp * env, pitch, pan)
}

```

We start with a particular pitch (D below middle C). Each time through the loop, we get a random number between 0 and 1. If that number is less than 0.5, then we transpose the current pitch up a minor third; if the random number is 0.5 or above, then we transpose the current pitch down a major second. Over the long haul, the resulting melodic line trends upward, but it does so in an irregular way.

The score above includes a few unfamiliar features. The first is the `srand` function, which *seeds* the random number generator. Computer-generated random numbers are not really random, in the sense that if we start with the same seed each time, we'll get the same series of random numbers. Changing the seed to a different integer gives us a different series. Try replacing the seed (1) in our score with another integer. You'll hear a different melodic line.

We set the interval of transposition using a *conditional* — a test, and a series of actions to perform depending on the result of the test. Study the syntax of the test below.

```

if (homer < 0.5) {
  bart = 1.3
}
else {
  bart = -99
}

```

If our test succeeds (the value of `homer` is less than 0.5), then RTcmix performs whatever is within the next set of curly braces (“`bart = 1.3`”). Otherwise, it performs whatever is within the set of braces following “else.” Indenting the statements that are within curly braces makes for easier reading. The tests you can use are ...

```
>    greater than
<    less than
>=   greater than or equal to
<=   less than or equal to
==   equals (note: 2 equal signs! Or else you might assign by mistake.)
!=   not equal to
```

The last new feature in the score above has to do with setting the amplitude of each note. We set each amplitude to a random value in decibels. This gives us a smoother result than using linear amplitudes. But instruments generally do not understand decibels, so we have to convert them to linear amplitudes. We do this with the `ampdb` function, which operates similarly to the pitch conversion functions.

Arrays

Sometimes random pitches are just too ... random. What if we want randomly selected pitches, but we want them to come from a specific scale, rather than from thin air? This is where *arrays* come in handy. An array is just a list of objects. Here’s a list of pitches ...

```
notes = { 8.00, 8.02, 8.04, 8.05, 8.07, 8.09, 8.11, 9.00 }
```

To access one of these notes, you need to use its *index*. Array indices start from zero and go up to one less than the number of objects in the array.

```
anote = notes[2]
anothernote = notes[7]
```

Now `anote` is 8.04 and `anothernote` is 9.00. You can assign to an array in a similar way.

```
notes[2] = 8.03
notes[5] = 8.08
```

This turns the major scale into a harmonic minor scale. Notice that you use curly braces when defining the array and square brackets when accessing the array with an index.

You can use arrays inside of a loop to get a randomly-selected pitch from a collection of pitches you define in advance. Here we make an array out of the pitches in the tone row of Berg’s *Violin Concerto*. Just out of familiarity, we use letter name notation for the pitches, which requires that we use the `pchlet` function to convert the names to oct.pc in the loop. This score (“`tut9.sco`,” on next page) uses a triangle wave, and it sets the loop increment and note duration so that the notes overlap. We play each note using two calls to `WAVETABLE`; the second call plays with slightly detuned pitch, to make for a richer sound.

```

rtsetparams(44100, 2)
load("WAVETABLE")

wavetable = maketable("wave", 1000, "tri")
env = maketable("line", 1000, 0,0, 1,1, 2,0)
control_rate(20000) // need faster envelope updates to avoid clicks

srand(3) // seed the random number generator

increment = 0.3
dur = increment * 5
amp = 8000

row = { "G4", "Bb4", "D4", "F#4", "A4", "C5",
        "E5", "G#4", "B4", "C#5", "Eb5", "F5"}
numnotes = len(row) // returns number of elements in array

for (start = 0; start < 20; start = start + increment) {
  index = trand(numnotes) // get random integer for index
  lettername = row[index] // access pitch array at that index
  pitch = pchlet(lettername) // convert letter name to oct.pc
  pan = irand(0, 1)
  WAVETABLE(start, dur, amp * env, pitch, pan, wavetable)
  WAVETABLE(start, dur, amp * env, pitch + 0.001, pan, wavetable)
}

```

We call the `trand` function to get a random number for use as an index. This function returns a random *integer* between zero and the supplied argument, which can be the size of the array. (We got this earlier using the `len` function.) We access the `row` array at that index, and then convert the result into `oct.pc` notation for use in `WAVETABLE`.

With what you know now, you would be able to add logic to the loop for varying the octave of the notes, either randomly or based on some kind of condition. (For example, if the start time is greater than 10, add 1 to the pitch.) Try it!

Sound File Input and Output

RTcmix is not limited to synthesis. You can also read sound files into your score. The two most basic instruments for playing sound files are `STEREO`, which simply plays the file into two output channels, and `TRANS`, which lets you transpose the sound.

But first you need to tell RTcmix which sound file to open. Here's a score ("tut10.sco") that plays a sound file several times, using various offsets into the file.

```

rtsetparams(44100, 2)
load("STEREO")
rtinput(".././snd/carolking.aif")
STEREO(start=0, inskip=0.2, dur=0.4, amp=1.0, pan=0.4)
STEREO(start=0.7, inskip=0.6, dur=0.8, amp=1.0, pan=0.9)

```

```
STEREO(start=1.0, inskip=3.7, dur=0.8, amp=0.8, pan=0.1)
STEREO(start=1.5, inskip=1.7, dur=1.0, amp=1.0, pan=0.6)
```

You open a sound file with the `rtinput` function. The sampling rate of this file must match that given in the `rtsetparams` call. Above, we give the *relative path name* of the file. The score is in the “`rtcmix/sco/tut`” directory, and that’s our current *working directory*. The sound file is in the “`rtcmix/snd`” directory, so the “`.././snd`” path component takes us up two levels (`../.`), into the `rtcmix` directory, and then down into the `snd` directory.

The second argument to `STEREO` is the amount of time in seconds to skip into the input sound file before reading from it — thus the variable name “`inskip`.”

Amplitude for instruments that take sound input works differently than for synthetic instruments like `WAVETABLE`. For the former, you use an *amplitude multiplier* rather than an absolute amplitude value. So in the third call to `STEREO` above, we’re asking RTcmix to multiply each sample point in the sound file by a factor of 0.8, reducing the volume of the sound slightly.

We’ve been using a `pan` argument all along. You may have noticed that it works differently than you might expect. In RTcmix, a `pan` value of 1 means to pan completely to the *left* channel. Think of `pan` as meaning “the percentage of sound (from 0 to 1) to place in the left channel.” So a `pan` value of 0 means to place the sound hard right. Go figure.

A common thing to do in RTcmix is to read random bits of sound from a file within a loop, as in this score (“`tut11.sco`”).

```
rtsetparams(44100, 2)
load("STEREO")
rtinput(".././snd/carolking.aif")
filedur = DUR() // get duration of most recently opened sound file
notedur = 0.2
env = maketable("line", 1000, 0,0, 1,1, 9,1, 10,0)
for (start = 0; start < 10; start = start + 0.15) {
    inskip = irand(0, filedur - notedur)
    STEREO(start, inskip, notedur, env, pan=random())
}
```

We get a random `inskip` from 0 to a point in time that is one note’s duration shy of the end of the file. This is to avoid “running off the end of the file” when reading the input sound.

Transposing an input file works similarly, but uses the `TRANS` instrument. The interval of transposition is specified in `oct.pc` format. The sort of transposition used here does not maintain duration, so it works like a variable speed tape deck. Here’s a score (“`tut12.sco`”) that modifies the previous one to randomly transpose snippets.

```
rtsetparams(44100, 2)
load("TRANS")
rtinput(".././snd/carolking.aif")
filedur = DUR() // get duration of most recently opened sound file
notedur = 0.2
```

```

env = maketable("line", 1000, 0,0, 1,1, 9,1, 10,0)
for (start = 0; start < 10; start = start + 0.15) {
  inskip = irand(0, filedur - notedur)
  transp = irand(-.12, .12) // random transposition up or down an octave
  TRANS(start, inskip, notedur, env, transp, pan=random())
}

```

At some point we'd like to capture the audio we're hearing into a sound file, for use in Pro Tools or elsewhere. You do this with the `rtoutput` function, which lets you specify an output sound file name and a data format, such as 16-bit, 24-bit or 32-bit floating point. The type of sound file header used is automatically taken from the file name suffix (".wav" or ".aif," usually).

```

rtsetparams(44100, 2)
load("WAVETABLE")
set_option("play = off") // don't play while writing file
rtoutput("/Users/yournamehere/totallycoolsound.aif") // change this
env = maketable("line", 1000, 0,0, 1,1, 9,1, 10,0)
for (start = 0; start < 10; start = start + 0.06) {
  freq = irand(60, 6000)
  WAVETABLE(start, dur=0.1, 8000 * env, freq, pan=random())
}

```

You'll have to change the file name so that it points to a place where you have permission to write. Be careful not to give the name of an existing file, like your score!

Other Instruments

There are many other instruments to use in RTcmix. Here are two more: `FMINST` and `STRUM2`. `FMINST` is a simple two-operator FM synthesis instrument. It works similarly to `WAVETABLE`, except that you have to give additional arguments for the modulator frequency and the modulation index. The latter is kind of complicated: you set a minimum and maximum modulation index, and then make a table, the *index guide*, that describes the shape used to traverse between these extreme values. Here's a sample score ("tut14.sco").

```

rtsetparams(44100, 2)
load("FMINST")

amp = 10000
env = maketable("line", 1000, 0,0, 1,1, 5,1, 7,0)
carfreq = 150 // carrier frequency
modfreq = carfreq * 2 // modulator frequency
min_index = 0 // modulation index range
max_index = 25

pan = maketable("line", 1000, 0,0, 1,1)
wavetable = maketable("wave", 1000, "sine")

index_guide = maketable("line", 1000, 0,0, 1,1, 2,0)

```

```

    FMINST(start=0, dur=10, amp * env, carfreq, modfreq, min_index, max_index,
           pan, wavetable, index_guide)

```

Play around with the computation of `modfreq`, try different minimum and maximum modulation index values, and specify different shapes for the various tables used in the score. Incidentally, this score shows how to pan during the course of a single note. Previously, we've had only static pan locations.

`STRUM2` is a synthetic plucked string instrument. It sounds like a cross between a harpsichord and a hammer dulcimer, but more artificial. The nice thing about it is that you can vary the thickness of the plectrum, via the “squish” parameter. Here's a sample score (“tut15.sco”).

```

rtsetparams(44100, 2)
load("STRUM2")
dur = 1.2
decay_time = dur * 0.4
base_increment = 0.16
minfreq = 400
maxfreq = 435

increment = base_increment
for (start = 0; start < 16; start = start + increment) {
    amp = irand(8000, 32000)
    freq = irand(minfreq, maxfreq)
    if (start > 10) {
        maxfreq = maxfreq + 150
    }
    squish = irand(0, 8) // how squishy is the guitar pick?
    pan = random()
    STRUM2(start, dur, amp, freq, squish, decay_time, pan)
    increment = base_increment + irand(-0.01, 0.01)
}

```

The score begins with unison pitch (around G4), but widely detuned. Then after ten seconds, the tessitura rapidly rises, due to `maxfreq` increasing.

This score shows how to randomize the time between successive notes (`increment`). Most of the other parameters are randomized as well.

Here are a few other instruments to explore (see docs at rtcmix.org).

COMBIT	MBLOWBOTL	REVMIX
DELAY	MBANDEDWG	STRUMFB
FILTERBANK	MSAXOFONY	SYNC
GRANULATE	MULTEQ	WAVY

Audio Buses

Often we want to process the sound of one instrument using another. RTcmix implements a bus connection scheme that lets you route audio within a score. You set this up using two or more calls to the `bus_config` function. The following score (“`tut16.sco`”) plays `WAVETABLE` notes and runs them through a stereo delay processor. The `DELAY` instrument plays one “note,” which spans the total duration of the `WAVETABLE` phrase. An instrument accepting input from a bus, such as `DELAY` below, must have an `inskip` of zero.

```

rtsetparams(44100, 2)
load("WAVETABLE")
load("DELAY")

bus_config("WAVETABLE", "aux 0-1 out")          // send WAVETABLE to stereo bus
bus_config("DELAY", "aux 0-1 in", "out 0-1")    // read bus into DELAY, then out

total_dur = 10
dur = 0.1
increment = 0.6
amp = 15000
env = maketable("line", 2000, 0,0, 1,1, 10,1, 30,0)

for (start = 0; start < total_dur; start = start + increment) {
    freq = irand(120, 2500)
    WAVETABLE(start, dur, amp * env, freq, pan = random())
}

//-----
delttime = 0.2
feedback = 0.6
ringdur = 2.8    // seconds to ring out delay line after note is finished
DELAY(start=0, inskip=0, total_dur, amp=1, deltime, feedback, ringdur,
    inchan=0, pan=1)
delttime += 0.02 // shorthand for "delttime = deltime + 0.02"
DELAY(start=0, inskip=0, total_dur, amp=1, deltime, feedback, ringdur,
    inchan=1, pan=0)

```

Many instruments that take input accept only mono input. These instruments have an `inchan` argument that lets you tell it which channel to read. Since we want to retain the random stereo panning done in the `WAVETABLE` loop, we need to read each `WAVETABLE` output channel into its own mono-input `DELAY` instrument.

Where to go from here

There’s a lot more to learn about RTcmix. The best place to turn is the documentation at rtcmix.org and example scores. Additional example scores are included with the RTcmix source code distribution (in `/usr/local/src/rtcmix/docs/sample_scores` on the CECM computers, or in `/Network/Applications/rtcmix/docs/sample_scores` in the library).