4.3.3 Tutorial

```
module Haskore.Interface.CSound.Tutorial where

import Haskore.Interface.CSound.Orchestra    as Orchestra
import Haskore.Interface.CSound.Score        as Score

import Haskore.Interface.CSound.Generator
           (compSine1, compSine2, cubicSpline, lineSeg1)
import Haskore.Interface.CSound.Render (playCS)

import qualified Haskore.Performance         as Performance
import qualified Haskore.Performance.Context as Context
import qualified Haskore.Performance.Player  as Player

import Haskore.Music                         as Music
import Haskore.Melody                        as Melody
import qualified Haskore.Melody.Standard     as StdMelody
import qualified Haskore.Music.Rhythmic      as RhyMusic
```

This brief tutorial is designed to introduce the users to the capabilities of the CSound software synthesizer and sound synthesis in general, and to get users started at using HasSound - the Haskell CSound interface.

Additive Synthesis Additive synthesis is the most basic, yet the most powerful synthesis technique available, giving complete control over the sound waveform. The basic premiss behind additive sound synthesis is quite simple – defining a complex sound by specifying each contributing sine wave. The computer is very good at generating pure tones, but these are not very interesting. However, any sound imaginable can be reproduced as a sum of pure tones. We can define an instrument of pure tones easily in Haskore.

CSound requires both an orchestra file and a score file. The score file contains basically two parts, *function table* and *score*.

First we define a function table containing just a sine wave. We can do this using the `simpleSine` function defined in module module `CSound.Score`

```
pureToneTN :: Score.Table
pureToneTN = 1
pureToneTable :: SigExp
pureToneTable = tableNumber pureToneTN
pureTone :: Score.Statement
pureTone = Score.simpleSine pureToneTN
```

`pureToneTN` is the table number of the simple sine wave. We will adopt the convention in this tutorial that variables ending with `TN` represent table numbers.

To create an Orchestra, we first create a simple instrument

```
oscPure :: SigExp -> SigExp -> SigExp
oscPure = osc AR pureToneTable

oe1 :: Mono
oe1 = let signal = oscPure (dbToAmp noteVel) (pchToHz notePit)
      in Mono signal
```

This instrument will simply oscillate through the function table containing the sine wave at the appropriate frequency given by `notePit`, and the resulting sound will have an amplitude given by `noteVel`, both defined in module module `CSound.Orchestra`.

We'll define our own `Instrument` type as a tuple of a list of parameters (or p-fields, explained in later parts), and an instrument number.

```
type InstrNum = Int

type Instrument = ([Float], InstrNum)

instr1, instr2 :: InstrNum
instr1 = 1
instr2 = 2
```

Note that the `oe1` expression above is a `Mono`, we can turn it into an Orchestra by prefixing it with a standard header of audio rate 44.1kHz and control rate 4.41kHz.

```
o1 :: Orchestra.T Mono
o1 = Cons hdr [ InstrBlock instr1 0 oe1 [] ]

hdr :: Orchestra.Header
hdr = (44100, 4410)
```

where `InstrBlock` is the constructor for Instrument Block, defined on module `CSound.Orchestra`.

We'll then create a simple tune to play with this instrument. But first of all we have to define our own melody type to cope with custom velocity and parameters.

```
type TutMelody = Melody.T TutAttr

data TutAttr = TutAttr {attrVelocity   :: Rational,
                        attrParameters :: [Float]}

tune1 :: TutMelody
tune1 = Music.line (map ($ TutAttr 1.4 [])
               [ c 1 hn, e 1 hn,  g 1 hn,
                 c 2 hn, a 1 hn,  c 2 qn,
                 a 1 qn, g 1 dhn ] ++ [qnr])
```

`tune1` has to be converted to a Rhythmic Music, and then to a Performance and then to a Score. Because we do not have drums here, `()` is used instead.

```
type Drum = ()

musicFromMelody :: InstrNum -> TutMelody -> RhyMusic.T Drum Instrument
musicFromMelody instrId = Music.mapNote (
            \(Melody.Note (TutAttr vel params) p) ->
             RhyMusic.Note vel (RhyMusic.Tone (params, instrId) p))

scoreFromMelody :: InstrNum -> TutMelody -> Score.T
scoreFromMelody i m =
   (Score.fromRhyPerformanceMap
       (error "no drum map defined") id) $
   (Performance.fromMusic Player.fancyMap
       (Context.setDur 1 Context.deflt)) $ musicFromMelody i m
```

`scoreFromMelody` creates a score that plays the given melody using the given instrument.

Then we'll complete the score by prefixing the Function Table `pureTone` to the score from `tune1`

```
score1 :: Score.T
score1 = pureTone : scoreFromMelody instr1 tune1

type Example out = (Orchestra.T out, Score.T)

tut1 :: Example Mono
tut1 = (o1, score1)
```

Or we can have a function to produce tut1 more conveniently by taking a `Score` and a list of instrument output, which is either `Mono` or `Stereo`, and will be numbered in ascending order as instrument 1, 2, ...

```
mkTut :: Output out => Score.T -> [out] -> Example out
mkTut score oes =
   let o = Cons hdr (map (\(i, oe) -> InstrBlock i 0 oe []) (zip [1..] oes))
    in (o, score)
```

so that `tut1` can be written as

```
tut1 = mkTut score1 [oe1]
```

All tutorial examples can be played by function

```
test :: Output out => Example out -> IO ()
test = uncurry playCS
```

If you listen to the tune, you will notice that it sounds very thin and uninteresting. Most musical sounds are not pure. Instead they usually contain a sine wave of dominant frequency, called a *fundamental*, and a number of other sine waves called *partials*. Partials with frequencies that are integer multiples of the fundamental are called *harmonics*. In musical terms, the first harmonic lies an octave above the fundamental, second harmonic a fifth above the first one, the third harmonic lies a major third above the second harmonic etc. This is the familiar *overtone series*. We can add harmonics to our sine wave instrument easily using the `compSine1` function defined in the module `CSound.Score` module. The function takes a list of harmonic strengths as arguments. The following creates a function table containing the fundamental and the first two harmonics at two thirds and one third of the strength of the fundamental:

```
twoHarmsTN :: Score.Table
twoHarmsTN = 2
twoHarms :: Score.Statement
twoHarms = Score.Table twoHarmsTN 0 8192 True (compSine1 [1.0, 0.66, 0.33])
```

We can again proceed to create complete score and orchestra just above:

```
score2 = twoHarms : scoreFromMelody instr1 tune1

oe2 :: Mono
oe2 = let signal = osc AR (tableNumber twoHarmsTN)
                       (dbToAmp noteVel) (pchToHz notePit)
      in  Mono signal

tut2 = mkTut score2 [oe2]
```

The orchestra file is the same as before – a single oscillator scanning a function table at a given frequency and volume. This time, however, the tune will not sound as thin as before since the table now contains a function that is an addition of three sine waves. (Note that the same effect could be achieved using a simple sine wave table and three oscillators).

Not all musical sounds contain harmonic partials exclusively, and never do we encounter instruments with static amplitude envelope like the ones we have seen so far. Most sounds, musical or not, evolve and change throughout their duration. Let's define an instrument containing both harmonic and nonharmonic partials, that starts at maximum amplitude with a straight line decay. We will use the function `compSine2` from the module `CSound.Orchestra` module to create the function table. `compSine2` takes a list of triples as an argument. The triples specify the partial number as a multiple of the fundamental, relative partial strength, and initial phase offset:

```
manySinesTN :: Score.Table
manySinesTN = 3
manySinesTable :: SigExp
manySinesTable = tableNumber manySinesTN
manySines :: Score.Statement
manySines = Score.Table manySinesTN 0 8192 True (compSine2 [(0.5, 0.9, 0.0),
                    (1.0, 1.0, 0.0), (1.1, 0.7, 0.0), (2.0, 0.6, 0.0),
                    (2.5, 0.3, 0.0), (3.0, 0.33, 0.0), (5.0, 0.2, 0.0)])
```

Thus this complex will contain the second, third, and fifth harmonic, nonharmonic partials at frequencies of 1.1 and 2.5 times the fundamental, and a component at half the frequency of the fundamental. Their strengths relative to the fundamental are given by the second argument, and they all start in sync with zero offset.

Now we can proceed as before to create score and orchestra files. We will define an *amplitude envelope* to apply to each note as we oscillate through the table. The amplitude envelope will be a straight line signal ramping from 1.0 to 0.0 over the duration of the note. This signal will be generated at *control rate* rather than audio rate, because the control rate is more than sufficient (the audio signal will change volume 4,410 times a second), and the slower rate will improve performance.

```
score3 = manySines : scoreFromMelody instr1 tune1

oe3 :: Mono
oe3 = let ampEnv = Orchestra.line CR 1.0 noteDur 0.0
          signal = osc AR manySinesTable
                         (ampEnv * dbToAmp noteVel) (pchToHz notePit)
      in  Mono signal

tut3 = mkTut score3 [oe3]
```

Not only do musical sounds usually evolve in terms of overall amplitude, they also evolve their *spectra*. In other words, the contributing partials do not usually all have the same amplitude envelope, and so their contribution to the overall sound isn't static. Let us illustrate the point using the same set of partials as in the above example. Instead of creating a table containing a complex waveform, however, we will use multiple oscillators going through the simple sine wave table we created at the beginning of this tutorial at the appropriate frequencies. Thus instead of the partials being fused together, each can have its own amplitude envelope, making the sound evolve over time. The score will be score1, defined above.

```
oe4 :: Mono
oe4 = let pitch       = pchToHz notePit
          amp         = dbToAmp noteVel
          mkLine t    = lineSeg CR 0 (noteDur*t) 1 [(noteDur * (1-t), 0)]
          aenv        = [Orchestra.line CR 1 noteDur 0] ++
                          map mkLine [0.17, 0.33, 0.50, 0.67, 0.83] ++
                          [Orchestra.line CR 0 noteDur 1]
          os (ae, p) = oscPure (ae * amp) (pitch * p)
          a           = map os (zip aenv [0.5, 1.0, 1.1, 2.0, 2.5, 3.0, 5.0])
          out         = 0.5 * sum a
      in  Mono out

tut4 = mkTut score1 [oe4]
```

So far, we have only used function tables to generate audio signals, but they can come very handy in *modifying* signals. Let us create a function table that we can use as an amplitude envelope to make our instrument more interesting. The envelope will contain an immediate sharp attack and decay, and then a second, more gradual one, so we'll have two attack/decay events per note. We'll use the cubic spline curve generating routine to do this:

```
coolEnvTN :: Score.Table
coolEnvTN = 4
coolEnvTable :: SigExp
coolEnvTable = tableNumber coolEnvTN
coolEnv :: Score.Statement
coolEnv = Score.Table coolEnvTN 0 8192 True
            (cubicSpline 1 [(1692, 0.2), (3000, 1), (3500, 0)])

oscCoolEnv :: SigExp -> SigExp -> SigExp
oscCoolEnv = osc CR coolEnvTable
```

Let us also add some *p-fields* to the notes in our score. The two p-fields we add will be used for *panning* – the first one will be the starting percentage of the left channel, the second one the ending percentage (1 means all left, 0 all right, 0.5 middle. Pfields of 1 and 0 will cause the note to pan completely from left to right for example).

The two p-fields are attributes per note, and will later be passed as parameters to the instrument as p-fields p6 and p7 (Note: this is because p1 and p2 are reserved, and p3, p4 and p5 are duration, pitch and velocity).

```
tune2 :: TutMelody
tune2 = let attr start end = TutAttr 1.4 [start, end]
        in  c 1 hn (attr 1.0 0.75) +:+
            e 1 hn (attr 0.75 0.5) +:+
            g 1 hn (attr 0.5 0.25) +:+
            c 2 hn (attr 0.25 0.0) +:+
            a 1 hn (attr 0.0 1.0)  +:+
            c 2 qn (attr 0.0 0.0)  +:+
            a 1 qn (attr 1.0 1.0)  +:+
           (g 1 dhn (attr 1.0 0.0) =:=
            g 1 dhn (attr 0.0 1.0))+:+ qnr
```

So far we have limited ourselves to using only sine waves for our audio output, even though Csound places no such restrictions on us. Any repeating waveform, of any shape, can be used to produce pitched sounds. In essence, when we are adding sinewaves, we are changing the shape of the wave. For example, adding odd harmonics to a fundamental at strengths equal to the inverse of their partial number (ie. third harmonic would be 1/3 the strength of the fundamental, fifth harmonic 1/5 the fundamental etc) would produce a *square* wave which has a raspy sound to it. Another common waveform is the *sawtooth*, and the more mellow sounding *triangle*. The module `CSound.Orchestra` module already contains functions to create these common waveforms. Let's use them to create tables that we can use in an instrument:

```
triangleTN, squareTN, sawtoothTN :: Score.Table
triangleTN = 5
squareTN   = 6
sawtoothTN = 7
triangleT, squareT, sawtoothT :: Score.Statement
triangleT = triangle triangleTN
squareT   = square   squareTN
sawtoothT = sawtooth sawtoothTN

score4 = squareT : triangleT : sawtoothT : coolEnv :
                   scoreFromMelody instr1 (changeTempo 0.5 tune2)
```

```
oe5 :: SigExp -> SigExp -> Stereo
oe5 panStart panEnd =
      let pitch  = pchToHz notePit
          amp    = dbToAmp noteVel
          pan    = Orchestra.line CR panStart noteDur panEnd
          oscF   = 1 / noteDur
          ampen  = oscCoolEnv amp oscF
          signal = osc AR (tableNumber squareTN) ampen pitch
          left   = signal * pan
          right  = signal * (1-pan)
      in  Stereo left right

tut5 = mkTut score4 [oe5 p6 p7]
```

This will oscillate through a table containing the square wave. Check out the other waveforms too and see what they sound like. This can be done by specifying the table to be used in the orchestra file.

Also note that `oe5` takes two parameters, which corresponds to the two p-fields: p6 and p7.

As our last example of additive synthesis, we will introduce an orchestra with multiple instruments. The bass will be mostly in the left channel, and will be the same as the third example instrument in this section. It will play the tune two octaves below the instrument in the right channel, using an orchestra identical to `oe3` with the addition of the panning feature:

```
score5 = manySines : pureTone : scoreFromMelody instr1 tune1 ++
                                scoreFromMelody instr2 tune1

oe6 :: Stereo
oe6 = let ampEnv = Orchestra.line CR 1.0 noteDur 0.0
          signal = osc AR manySinesTable
                      (2 * ampEnv * dbToAmp noteVel) (pchToHz (notePit - 2))
          left   = 0.8 * signal
          right  = 0.2 * signal
      in  Stereo left right

oe7 :: Stereo
oe7 = let pitch      = pchToHz notePit
          amp        = dbToAmp noteVel
          mkLine t   = lineSeg CR 0 (noteDur*t) 0.5 [(noteDur * (1-t), 0)]
          aenv       = [Orchestra.line CR 0.5 noteDur 0] ++
                       map mkLine [0.17, 0.33, 0.50, 0.67, 0.83] ++
                       [Orchestra.line CR 0 noteDur 0.5]
          os (ae, p) = oscPure (ae * amp) (pitch * p)
          a          = map os (zip aenv [0.5, 1.0, 1.1, 2.0, 2.5, 3.0, 5.0])
          left       = 0.2 * (sum a)
          right      = 0.8 * (sum a)
      in  Stereo left right

tut6 = mkTut score5 [oe6, oe7]
```

Additive synthesis is the most powerful tool in computer music and sound synthesis in general. It can be used to create any sound imaginable, whether completely synthetic or a simulation of a real-world sound, and everyone interested in using the computer to synthesize sound should be well versed in it. The most significant drawback of additive synthesis is that it requires huge amounts of control data, and potentially thousands of oscillators. There are other synthesis techniques, such as *modulation synthesis*, that can be used to create rich and interesting timbres at a fraction of the cost of additive synthesis, though no other synthesis technique provides quite the same degree of control.

Modulation Synthesis While additive synthesis provides full control and great flexibility, it is quiet clear that the enormous amounts of control data make it impractical for even moderately complicated sounds. There is a class of synthesis techniques that use *modulation* to produce rich, time-varying timbres at a fraction of the storage and time cost of additive synthesis. The basic idea behind modulation synthesis is controlling the amplitude and/or frequency of the main periodic signal, called the *carrier*, by another periodic signal, called the *modulator*. The two main kinds of modulation synthesis are *amplitude modulation* and *frequency modulation* synthesis. Let's start our discussion with the simpler one of the two – amplitude synthesis.

We have already shown how to supply a time varying amplitude envelope to an oscillator. What would happen if this amplitude envelope was itself an oscillating signal? Supplying a low frequency ($<$ 20Hz) modulating signal would create a predictable effect – we would hear the volume of the carrier signal go periodically up and down. However, as the modulator moves into the audible frequency range, the carrier changes timbre as new frequencies appear in the spectrum. The new frequencies are equal to the sum and difference of the carrier and modulator. So for example, if the frequency of the main signal (carrier) is C = 500Hz, and the frequency of the modulator is M = 100Hz, the audible frequencies will be the carrier C (500Hz), C + M (600Hz), and C - M (400Hz). The amplitude of the two new sidebands depends on the amplitude of the modulator, but will never exceed half the amplitude of the carrier.

The following is a simple example that demonstrates amplitude modulation. The carrier will be a 10 second pure tone at 500Hz. The frequency of the modulator will increase linearly over the 10 second duration of the tone from 0 to 200 Hz. Initially, you will be able to hear the volume of the signal fluctuate, but after a couple of seconds the volume will seem constant as new frequencies appear.

Let us first create the score file. It will contain a sine wave table, and a single note event:

```
score6 = pureTone : [ Score.Note 1 0.0 10.0 (Cps 500.0) 10000.0 [] ]
```

The orchestra will contain a single AM instrument. The carrier will simply oscillate through the sine wave table at frequency given by the note pitch (500Hz, see the score above), and amplitude given by the modulator. The modulator will oscillate through the same sine wave table at frequency ramping from 0 to 200Hz. The modulator should be a periodic signal that varies from 0 to the maximum volume of the carrier. Since the sine wave goes from -1 to 1, we will need to add 1 to it and half it, before multiplying it by the volume supplied by the note event. This will be the modulating signal, and the carrier's amplitude input. (note that we omit the conversion functions dbToAmp and notePit, since we supply the amplitude and frequency in their raw units in the score file)

```
oe8 :: Mono
oe8 = let modFreq = Orchestra.line CR 0.0 noteDur 200.0
          modAmp  = oscPure 1.0 modFreq
          modSig  = (modAmp + 1.0) * 0.5 * noteVel
          carrier = oscPure modSig notePit
      in  Mono carrier

tut7 = mkTut score6 [oe8]
```

Next synthesis technique on the palette is *frequency modulation*. As the name suggests, we modulate the frequency of the carrier. Frequency modulation is much more powerful and interesting than amplitude modulation, because instead of getting two sidebands, FM gives a *number* of spectral sidebands. Let us begin with an example of a simple FM. We will again use a single 10 second note and a 500Hz carrier. Remember that when we talked about amplitude modulation, the amplitude of the sidebands was dependent upon the amplitude of the modulator. In FM, the modulator amplitude plays a much bigger role, as we will see soon. To negate the effect of the modulator amplitude, we will keep the ratio of the modulator amplitude and frequency constant at 1.0 (we will explain shortly why). The frequency and amplitude of the modulator will ramp from 0 to 200 over the duration of the note. This time, though, unlike with AM, we will hear a whole series of sidebands. The orchestra is just as before, except we modulate the frequency instead of amplitude.

```
oe9 :: Mono
oe9 = let modFreq = Orchestra.line CR 0.0 noteDur 200.0
          modAmp  = modFreq
          modSig  = oscPure modAmp modFreq
          carrier = oscPure noteVel (notePit + modSig)
      in  Mono carrier

tut8 = mkTut score6 [oe9]
```

The sound produced by FM is a little richer but still very bland. Let us talk now about the role of the *depth* of the frequency modulation (the amplitude of the modulator). Unlike in AM, where we only had one spectral band on each side of the carrier frequency (ie we heard C, C+M, C-M), FM gives a much richer spectrum with many sidebands. The frequencies we hear are C, C+M, C-M, C+2M, C-2M, C+3M, C-3M etc. The amplitudes of the sidebands are determined by the *modulation index* I, which is the ratio between the amplitude (also referred to as depth) and frequency of the modulator (I = D / M). As a rule of thumb, the number of significant sideband pairs (at least 1number of sidebands) increases, energy is "stolen" from the carrier and distributed among the sidebands. Thus if I=1, we have 2 significant sideband pairs, and the audible frequencies will be C, C+M, C-M, C+2M, C-2M, with C, the carrier, being the dominant frequency. When I=5, we will have a much richer sound with about 6 significant sideband pairs, some of which will actually be louder than the carrier. Let us explore the effect of the modulation index in the following example. We will keep the frequency of the carrier and the modulator constant at 500Hz and 80 Hz respectively. The modulation index will be a stepwise function from 1 to 10, holding each value for one second. So in effect, during the first second (I = D/M = 1), the amplitude of the modulator will be the same as its frequency (80). During the second second (I = 2), the amplitude will be double the frequency (160), then it will go to 240, 320, etc:

```
oe10 :: Mono
oe10 = let modInd  = lineSeg CR 1 1 1 [(0,2), (1,2), (0,3), (1,3), (0,4),
                                       (1,4), (0,5), (1,5), (0,6), (1,6),
                                       (0,7), (1,7), (0,8), (0,9), (1,9),
                                       (0,10), (1,10)]
           modAmp  = 80.0 * modInd
           modSig  = oscPure modAmp 80.0
           carrier = oscPure noteVel (notePit + modSig)
       in  Mono carrier

tut9 = mkTut score6 [oe10]
```

Notice that when the modulation index gets high enough, some of the sidebands have negative frequencies. For example, when the modulation index is 7, there is a sideband present in the sound with a frequency C - 7M = 500 - 560 = -60Hz. The negative sidebands get reflected back into the audible spectrum but are *phase shifted* 180 degrees, so it is an inverse sine wave. This makes no difference when the wave is on its own, but when we add it to its inverse, the two will cancel out. Say we set the frequency of the carrier at 100Hz instead of 80Hz. Then at I=6, we would have present two sidebands of the same frequency - C-4M = 100Hz, and C-6M = -100Hz. When these two are added, they would cancel each other out (if they were the same amplitude; if not, the louder one would be attenuated by the amplitude of the softer one). The following flexible instrument will sum up simple FM. The frequency of the modulator will be determined by the C/M ratio supplied as p6 in the score file. The modulation index will be a linear slope going from 0 to p7 over the duration of each note. Let us also add panning control as in additive synthesis - p8 will be the initial left channel percentage, and p9 the final left channel percentage:

```
oe11 :: SigExp -> SigExp -> SigExp -> SigExp -> Stereo
oe11 modFreqRatio modIndEnd panStart panEnd =
       let carFreq = pchToHz notePit
           carAmp  = dbToAmp noteVel
           modFreq = carFreq * modFreqRatio
           modInd  = Orchestra.line CR 0 noteDur modIndEnd
           modAmp  = modFreq * modInd
           modSig  = oscPure modAmp modFreq
           carrier = oscPure carAmp (carFreq + modSig)
           mainAmp = oscCoolEnv 1.0 (1/noteDur)
           pan     = Orchestra.line CR panStart noteDur panEnd
           left    = mainAmp * pan * carrier
           right   = mainAmp * (1 - pan) * carrier
       in  Stereo left right
```

Let's write a cool tune to show off this instrument. Let's keep it simple and play the chord progression Em - C - G - D a few times, each time changing some of the parameters:

```
emChord, cChord, gChord, dChord ::
   Float -> Float -> Float -> Float -> TutMelody

quickChord ::
   [Music.Dur -> TutAttr -> TutMelody] ->
   Float -> Float -> Float -> Float -> TutMelody
quickChord ns x y z w = chord $
   map (\p -> p wn (TutAttr 1.4 [x, y, z, w])) ns

emChord = quickChord [e 0, g  0, b 0]
cChord  = quickChord [c 0, e  0, g 0]
gChord  = quickChord [g 0, b  0, d 1]
dChord  = quickChord [d 0, fs 0, a 0]

tune3 :: TutMelody
tune3 = transpose (-12) $
        emChord 3.0 2.0 0.0 1.0  +:+  cChord  3.0  5.0 1.0 0.0  +:+
        gChord  3.0 8.0 0.0 1.0  +:+  dChord  3.0 12.0 1.0 0.0  +:+
        emChord 3.0 4.0 0.0 0.5  +:+  cChord  5.0  4.0 0.5 1.0  +:+
        gChord  8.0 4.0 1.0 0.5  +:+  dChord 10.0  4.0 0.5 0.0  +:+
        (emChord 4.0 6.0 1.0 0.0  =:=  emChord 7.0  5.0 0.0 1.0)  +:+
        (cChord  5.0 9.0 1.0 0.0  =:=  cChord  9.0  5.0 0.0 1.0)  +:+
        (gChord  5.0 5.0 1.0 0.0  =:=  gChord  7.0  7.0 0.0 1.0)  +:+
        (dChord  2.0 3.0 1.0 0.0  =:=  dChord  7.0 15.0 0.0 1.0)
```

Now we can create a score. It will contain two wave tables – one containing the sine wave, and the other containing an amplitude envelope, which will be the table coolEnv which we have already seen before

```
score7 = pureTone : coolEnv :
                scoreFromMelody instr1 (changeTempo 0.5 tune3)

tut10 = mkTut score7 [oe11 p6 p7 p8 p9]
```

Note that all of the above examples of frequency modulation use a single carrier and a single modulator, and both are oscillating through the simplest of waveforms – a sine wave. Already we have achieved some very rich and interesting timbres using this simple technique, but the possibilities are unlimited when we start using different carrier and modulator waveshapes and multiple carriers and/or modulators. Let us include a couple more examples that will play the same chord progression as above with multiple carriers, and then with multiple modulators.

The reason for using multiple carriers is to obtain /em formant regions in the spectrum of the sound. Recall that when we modulate a carrier frequency we get a spectrum with a central peak and a number of sidebands on either side of it. Multiple carriers introduce additional peaks and sidebands into the composite spectrum of the resulting sound. These extra peaks are called formant regions, and are characteristic of human voice and most musical instruments

```
oe12 :: SigExp -> SigExp -> SigExp -> SigExp -> Stereo
oe12 modFreqRatio modIndEnd panStart panEnd =
       let car1Freq = pchToHz notePit
           car2Freq = pchToHz (notePit + 1)
           car1Amp  = dbToAmp noteVel
           car2Amp  = dbToAmp noteVel * 0.7
           modFreq  = car1Freq * modFreqRatio
           modInd   = Orchestra.line CR 0 noteDur modIndEnd
           modAmp   = modFreq * modInd
           modSig   = oscPure modAmp modFreq
           carrier1 = oscPure car1Amp (car1Freq + modSig)
           carrier2 = oscPure car2Amp (car2Freq + modSig)
           mainAmp  = oscCoolEnv 1.0 (1/noteDur)
           pan      = Orchestra.line CR panStart noteDur panEnd
           left     = mainAmp * pan * (carrier1 + carrier2)
           right    = mainAmp * (1 - pan) * (carrier1 + carrier2)
       in  Stereo left right

tut11 = mkTut score7 [oe12 p6 p7 p8 p9]
```

In the above example, there are two formant regions – one is centered around the note pitch frequency provided by the score file, the other an octave above. Both are modulated in the same way by the same modulator. The sound is even richer than that obtained by simple FM.

Let us now turn to multiple modulator FM. In this case, we use a signal to modify another signal, and the modified signal will itself become a modulator acting on the carrier. Thus the wave that wil be modulating the carrier is not a sine wave as above, but is itself a complex waveform resulting from simple FM. The spectrum of the sound will contain a central peak frequency, surrounded by a number of sidebands, but this time each sideband will itself also by surrounded by a number of sidebands of its own. So in effect we are talking about "double" modulation, where each sideband is a central peak in its own little spectrum. Multiple modulator FM thus provides extremely rich spectra

```
oe13 :: SigExp -> SigExp -> SigExp -> SigExp -> Stereo
oe13 modFreqRatio modIndEnd panStart panEnd =
       let carFreq  = pchToHz notePit
           carAmp   = dbToAmp noteVel
           mod1Freq = carFreq * modFreqRatio
           mod2Freq = mod1Freq * 2.0
           modInd   = Orchestra.line CR 0 noteDur modIndEnd
           mod1Amp  = mod1Freq * modInd
           mod2Amp  = mod1Amp * 3.0
           mod1Sig  = oscPure mod1Amp mod1Freq
           mod2Sig  = oscPure mod2Amp (mod2Freq + mod1Sig)
           carrier  = oscPure carAmp  (carFreq  + mod2Sig)
           mainAmp  = oscCoolEnv 1.0 (1/noteDur)
           pan      = Orchestra.line CR panStart noteDur panEnd
           left     = mainAmp * pan * carrier
           right    = mainAmp * (1 - pan) * carrier
       in  Stereo left right

tut12 = mkTut score7 [oe13 p6 p7 p8 p9]
```

In fact, the spectra produced by multiple modulator FM are so rich and complicated that even the moderate values used as arguments in our tune produce spectra that are saturated and otherworldly. And we did this while keeping the ratios of the two modulators frequencies and amplitudes constant; introducing dynamics in those ratios would produce even crazier results. It is quite amazing that from three simple sine waves, the purest of all tones, we can derive an unlimited number of timbres. Modulation synthesis is a very powerful tool and understanding how to use it can prove invaluable. The best way to learn how to use FM effectively is to dabble and experiment with different ratios, formant regions, dynamic relationships betweeen ratios, waveshapes, etc. The possibilities are limitless.

Other Capabilities Of CSound In our examples of additive and modulation synthesis we only used a limited number of functions and routines provided us by CSound, such as Osc (oscillator), Line and LineSig (line and line segment signal generators) etc. This tutorial intends to briefly explain the functionality of some of the other features of CSound. Remember that the CSound manual should be the ultimate reference when it comes to using these functions.

Let us start with the two functions `buzz` and `genBuzz`. These functions will produce a set of harmonically related cosines. Thus they really implement simple additive synthesis, except that the number of partials can be varied dynamically through the duration of the note, rather than staying fixed as in simple additive synthesis. As an example, let us perform the tune defined at the very beginning of the tutorial using an instrument that will play each note by starting off with the fundamental and 70 harmonics, and ending with simply the sine wave fundamental (note that cosine and sine waves sound the same). We will use a straight line signal going from 70 to 0 over the duration of each note for the number of harmonics. The score used will be score1, and the orchestra will be:

```
oe14 :: Mono
oe14 = let numharms = Orchestra.line CR 70 noteDur 0
           signal   = buzz pureToneTable numharms
                          (dbToAmp noteVel) (pchToHz notePit)
       in  Mono signal

tut13 = mkTut score1 [oe14]
```

Let's invert the line of the harmonics, and instead of going from 70 to 0, make it go from 0 to 70. This will produce an interesting effect quite different from the one just heard:

```
oe15 :: Mono
oe15 = let numharms = Orchestra.line CR 0 noteDur 70
           signal   = buzz pureToneTable numharms
                          (dbToAmp noteVel) (pchToHz notePit)
       in  Mono signal

tut14 = mkTut score1 [oe15]
```

The `buzz` expression takes the overall amplitude, fundamental frequency, number of partials, and a sine wave table and generates a wave complex.

In recent years there has been a lot of research conducted in the area of *physical modelling*. This technique attempts to approximate the sound of real world musical instruments through mathematical models.

One of the most widespread, versatile and interesting of these models is the *Karplus-Strong algorithm* that simulates the sound of a plucked string. The algorithm starts off with a buffer containing a user-determined waveform. On every pass, the waveform is ”smoothed out” and flattened by the algorithm to simulate the decay. There is a certain degree of randomness involved to make the string sound more natural.

There are six different ”smoothing methods” available in CSound, as mentioned in the CSound module. The `pluck` constructor accepts the note volume, pitch, the table number that is used to initialize the buffer, the smoothing method used, and two parameters that depend on the smoothing method. If zero is given as the initializing table number, the buffer starts off containing a random waveform (white noise). This is the best table when simulating a string instrument because of the randomness and percussive attack it produces when used with this algorithm, but you should experiment with other waveforms as well.

Here is an example of what Pluck sounds like with a white noise buffer and the simple smoothing method. This method ignores the parameters, which we set to zero.

```
oe16 :: Mono
oe16 = let signal = pluck 0 (pchToHz notePit)
                          PluckSimpleSmooth
                          (dbToAmp noteVel) (pchToHz notePit)
       in  Mono signal

tut15 = mkTut score1 [oe16]
```

The second smoothing method is the *stretched smooth*, which works like the simple smooth above, except that the smoothing process is stretched by a factor determined by the first parameter. The second parameter is ignored. The third smoothing method is the *snare drum* method. The first parameter is the ”roughness” parameter, with 0 resulting in a sound identical to simple smooth, 0.5 being the perfect snare drum, and 1.0 being the same as simple smooth again with reversed polarity (like a graph flipped around the x-axis). The fourth smoothing method is the *stretched drum* method which combines the roughness and stretch factors – the first parameter is the roughness, the second is the stretch. The fifth method is *weighted average* – it combines the current sample (ie. the current pass through the buffer) with the previous one, with their weights being determined by the parameters. This is a way to add slight reverb to the plucked sound. Finally, the last method filters the sound so it doesn’t sound as bright. The parameters are ignored. You can modify the instrument `oe16` easily to listen to all these effects by simply replacing the variable `simpleSmooth` by `stretchSmooth, simpleDrum, stretchDrum, weightedSmooth` or `filterSmooth`.

Here is another simple instrument example. This combines a snare drum sound with a stretched plucked string sound. The snare drum as a constant amplitude, while we apply an amplitude envelope to the string sound. The envelope is a spline curve with a hump in the middle, so both the attack and decay are gradual. The drum roughness factor is 0.3, so a pitch is still discernible (with a factor of 0.5 we would get a snare drum sound with no pitch, just a puff of white noise). The drum sound is shifted towards the left channel, while the string sound is shifted towards the right.

```
midHumpTN :: Score.Table
midHumpTN = 8
midHump :: Score.Statement
midHump = Score.Table midHumpTN 0 8192 True
             (cubicSpline 0.0 [(4096, 1.0), (4096, 0.0)])
```

```
score8 = pureTone : midHump : scoreFromMelody instr1 tune1

oe17 :: Stereo
oe17 = let string = pluck 0 (pchToHz notePit)
                          (PluckStretchSmooth 1.5)
                          (dbToAmp noteVel) (pchToHz notePit)
           drum   = pluck 0 (pchToHz notePit)
                          (PluckSimpleDrum 0.3)
                          6000 (pchToHz notePit)
           ampEnv = osc CR (tableNumber midHumpTN) 1.0 (1 / noteDur)
           left   = (0.65 * drum) + (0.35 * ampEnv * string)
           right  = (0.35 * drum) + (0.65 * ampEnv * string)
       in  Stereo left right

tut16 = mkTut score8 [oe17]
```

Let us now turn our attention to the effects we can achieve using a *delay line*.

Let's define a simple percussive instrument. It's strong attack let us easily perceive the reverberation.

```
ping :: SigExp
ping =
   let ampEnv = expon CR 1.0 1.0 (1/100)
   in  osc AR manySinesTable
           (ampEnv * dbToAmp noteVel) (pchToHz notePit)
```

There is still the problem, that subsequent notes truncate preceding ones. This would suppress the reverb. In order to avoid this we add a *legato* effect to the music. That is we prolong the notes such that they overlap.

```
score9 = manySines : scoreFromMelody instr1 (legato 1 tune1)
```

Here we take the ping sound and add a little echo to it using delay:

```
oe18 :: Stereo
oe18 = let dping1 = Orchestra.delay 0.05 ping
           dping2 = Orchestra.delay 0.1  ping
           left   = (0.65 * ping) + (0.35 * dping2) + (0.5 * dping1)
           right  = (0.35 * ping) + (0.65 * dping2) + (0.5 * dping1)
       in  Stereo left right

tut17 = mkTut score9 [oe18]
```

The constructor `delay` establishes a *delay line*. A delay line is essentially a buffer that contains the signal to be delayed. The first argument to the `delay` constructor is the length of the delay (which determines the size of the buffer), and the second argument is the signal to be delayed. So for example, if the delay time is 1.0 seconds, and the sampling rate is 44,100 Hz (CD quality), then the delay line will be a buffer containing 44,100 samples of the delayed signal. The buffer is rewritten at the audio rate. Once `Delay t` sig writes t seconds of the signal sig into the buffer, the buffer can be *tapped* using the delTap or the delTapI constructors. delTap t dline will extract the signal from dline at time t seconds. In the exmaple above, we set up a delay line containing 0.1 seconds of the audio signal, then we tapped it twice – once at 0.05 seconds and once at 0.1 seconds. The output signal is a combination of the original signal (left channel), the signal delayed by 0.05 seconds (middle), and the signal delayed by 0.1 seconds (right channel).

CSound provides other ways to reverberate a signal besides the delay line just demonstrated. One such way is achieved via the Reverb constructor introduced in the module CSound.Orchestra module. This constructor tries to emulate natural room reverb, and takes as arguments the signal to be reverberated, and the reverb time in seconds. This is the time it takes the signal to decay to 1/1000 its original amplitude. In this example we output both the original and the reverberated sound.

```
oe19 :: Stereo
oe19 = let rev    = reverb 0.3 ping
           left   = (0.65 * ping) + (0.35 * rev)
           right  = (0.35 * ping) + (0.65 * rev)
       in  Stereo left right

tut18 = mkTut score9 [oe19]
```

The other two reverb functions are comb and alpass. Each of these requires as arguments the signal to be reverberated, the reverb time as above, and echo loop density in seconds. Here is an example of an instrument using comb.

```
oe20 :: Mono
oe20 = Mono (comb 0.22 4.0 ping)

tut19 = mkTut score9 [oe20]
```

Delay lines can be used for effects other than simple echo and reverberation. Once the delay line has been established, it can be tapped at times that vary at control or audio rates. This can be taken advantage of to produce effects like chorus, flanger, or the Doppler effect. Here is an example of the flanger effect. This instrument adds a slight flange to oe11.

```
oe21 :: SigExp -> SigExp -> SigExp -> SigExp -> Stereo
oe21 modFreqRatio modIndEnd panStart panEnd =
       let carFreq = pchToHz notePit
           ampEnv  = oscCoolEnv 1.0 (1/noteDur)
           carAmp  = dbToAmp noteVel * ampEnv
           modFreq = carFreq * modFreqRatio
           modInd  = Orchestra.line CR 0 noteDur modIndEnd
           modAmp  = modFreq * modInd
           modSig  = oscPure modAmp modFreq
           carrier = oscPure carAmp (carFreq + modSig)
           ftime   = oscPure (1/10) 2
           flanger = ampEnv * vdelay 1 (0.5 + ftime) carrier
           signal  = carrier + flanger
           pan      = Orchestra.line CR panStart noteDur panEnd
           left     = pan * signal
           right    = (1 - pan) * signal
       in  Stereo left right

tut20 = mkTut score7 [oe21 p6 p7 p8 p9]
```

The last two examples use generic delay lines. That is we do not rely on special echo effects but build our own ones by delaying a signal, filtering it by low pass or high pass filters and feeding the result back to the delay function.

```
lowPass, highPass :: EvalRate -> SigExp -> SigExp -> SigExp
lowPass  rate cutOff sig = sigGen "tone"  rate 1 [sig, cutOff]
highPass rate cutOff sig = sigGen "atone" rate 1 [sig, cutOff]

oe22 :: Stereo
oe22 = let left  = rec (\x -> ping + lowPass  AR
500 (Orchestra.delay 0.311 x))
           right = rec (\x -> ping + highPass AR 1000 (Orchestra.delay 0.271 x))
       in  Stereo left right

tut21 = mkTut score9 [oe22]

oe23 :: Mono
oe23 = let rev = rec (\x -> ping +
                        0.7 * (lowPass  AR  500 (Orchestra.delay 0.311 x)
                            + highPass AR 1000 (Orchestra.delay 0.271 x)))
       in  Mono rev

o22 = Cons hdr [ InstrBlock instr1 0 oe23 [] ]

tut22 = mkTut score9 [oe23]
```

This completes our discussion of sound synthesis and Csound. For more information, please consult the CSound manual or check out

http://mitpress.mit.edu/e-books/csound/frontpage.html

Here are some bonus instruments for your pleasure and enjoyment. The first ten instruments are lifted from

http://wings.buffalo.edu/academic/department/AandL/music/pub/accci/01/01_01_1b.txt.html

The tutorial explains how to add echo/reverb and other effects to the instruments if you need to. This instrument sounds like an electric piano and is really simple – `pianoEnv` sets the amplitude envelope, and the sound waveform is just a series of 10 harmonics. To make the sound brighter, increase the weight of the upper harmonics.

```
piano, reedy, flute
   :: Example Mono

pianoScore, reedyScore, fluteScore :: Score.T
pianoEnv, reedyEnv, fluteEnv,
  pianoWave, reedyWave, fluteWave :: Score.Statement
pianoEnvTN, reedyEnvTN, fluteEnvTN,
  pianoWaveTN, reedyWaveTN, fluteWaveTN :: Score.Table
pianoEnvTable, reedyEnvTable, fluteEnvTable,
  pianoWaveTable, reedyWaveTable, fluteWaveTable :: SigExp

pianoEnvTN  = 10; pianoEnvTable  = tableNumber pianoEnvTN
pianoWaveTN = 11; pianoWaveTable = tableNumber pianoWaveTN

pianoEnv    = Score.Table pianoEnvTN 0 1024 True (lineSeg1 0 [(20, 0.99),
                                      (380, 0.4), (400, 0.2), (224, 0)])
pianoWave   = Score.Table pianoWaveTN  0 1024 True (compSine1 [0.158, 0.316,
                      1.0, 1.0, 0.282, 0.112, 0.063, 0.079, 0.126, 0.071])

pianoScore = pianoEnv : pianoWave : scoreFromMelody instr1 tune1

pianoOE :: Mono
pianoOE = let ampEnv = osc CR pianoEnvTable (dbToAmp noteVel) (1/noteDur)
              signal = osc AR pianoWaveTable ampEnv (pchToHz notePit)
          in  Mono signal

piano = mkTut pianoScore [pianoOE]
```

Here is another instrument with a reedy sound to it

```
reedyEnvTN  = 12; reedyEnvTable  = tableNumber reedyEnvTN
reedyWaveTN = 13; reedyWaveTable = tableNumber reedyWaveTN

reedyEnv    = Score.Table reedyEnvTN   0 1024 True (lineSeg1 0 [(172, 1.0),
     (170, 0.8), (170, 0.6), (170, 0.7), (170, 0.6), (172,0)])
reedyWave   = Score.Table reedyWaveTN  0 1024 True (compSine1 [0.4, 0.3,
                      0.35, 0.5, 0.1, 0.2, 0.15, 0.0, 0.02, 0.05, 0.03])

reedyScore = reedyEnv : reedyWave : scoreFromMelody instr1 tune1

reedyOE :: Mono
reedyOE = let ampEnv = osc CR reedyEnvTable (dbToAmp noteVel) (1/noteDur)
              signal = osc AR reedyWaveTable ampEnv (pchToHz notePit)
          in  Mono signal

reedy = mkTut reedyScore [reedyOE]
```

We can use a little trick to make it sound like several reeds playing by adding three signals that are slightly out of tune:

```
reedy2OE :: Stereo
reedy2OE = let ampEnv = osc CR reedyEnvTable (dbToAmp noteVel) (1/noteDur)
               freq   = pchToHz notePit
               reedyOsc = osc AR reedyWaveTable
               a1     = reedyOsc ampEnv freq
               a2     = reedyOsc (ampEnv * 0.44) (freq + (0.023 * freq))
               a3     = reedyOsc (ampEnv * 0.26) (freq + (0.019 * freq))
               left   = (a1 * 0.5) + (a2 * 0.35) + (a3 * 0.65)
               right  = (a1 * 0.5) + (a2 * 0.65) + (a3 * 0.35)
           in  Stereo left right

reedy2 = mkTut reedyScore [reedy2OE]
```

This instrument tries to emulate a flute sound by introducing random variations to the amplitude envelope. The score file passes in two parameters – the first one is the depth of the random tremolo in percent of total amplitude. The tremolo is implemented using the `randomI` function, which generates a signal that interpolates between 2 random numbers over a certain number of samples that is specified by the second parameter.

```
fluteTune :: TutMelody

fluteTune = Music.line
               (map ($ TutAttr 1.6 [30, 40])
                  [c 1 hn, e 1 hn, g 1 hn, c 2 hn,
                   a 1 hn, c 2 qn, a 1 qn, g 1 dhn]
                ++ [qnr])

fluteEnvTN  = 14; fluteEnvTable  = tableNumber fluteEnvTN
fluteWaveTN = 15; fluteWaveTable = tableNumber fluteWaveTN

fluteEnv    = Score.Table fluteEnvTN   0 1024 True (lineSeg1 0 [(100, 0.8),
                          (200, 0.9), (100, 0.7), (300, 0.2), (324, 0.0)])
fluteWave   = Score.Table fluteWaveTN  0 1024 True (compSine1 [1.0, 0.4,
                                                  0.2, 0.1, 0.1, 0.05])

fluteScore = fluteEnv : fluteWave : scoreFromMelody instr1 fluteTune

fluteOE :: SigExp -> SigExp -> Mono
fluteOE depth numSam =
          let vol    = dbToAmp noteVel
              rand   = randomI AR numSam (vol/100 * depth)
              ampEnv = oscI AR fluteEnvTable
                           (rand + vol) (1 / noteDur)
              signal = oscI AR fluteWaveTable
```

```
                              ampEnv (pchToHz notePit)
          in  Mono signal

flute = mkTut fluteScore [fluteOE p6 p7]
```