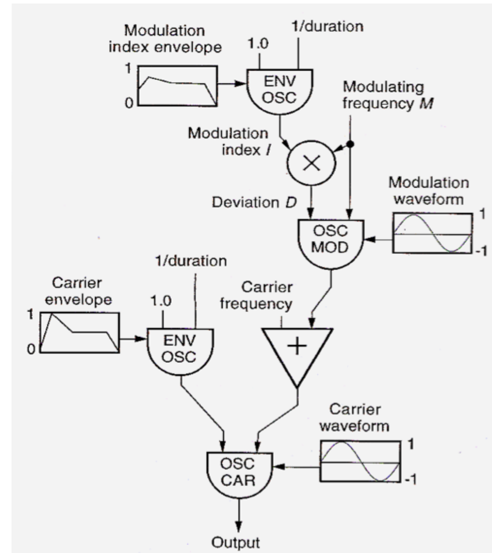
Music 220A – Homework 3 – Lab, Part 2: FM Function



In this lab, we'll write a function to carry out the FM synthesis patch on the right. The challenge in writing this function is thinking about how to structure the patch, and the best way to structure the main function and any sub-functions. A recommended way is walked through below, though if you had another preferred strategy, you are welcome to do that instead. At the end of this lab, your script will output a .wav file that is ONE 10-second note at 440 Hz, in which the modulator frequency is 75 Hz, the modulator gain ramps from [0.0 800.0 0.0] at [0 5 10] seconds, and the carrier amplitude ramps from [0.0 1.0 0.5 0.0] at [0 3 5 10] seconds.

As always, if anything isn't clear, refer to the UGen page on the ChucK Programming Guide to look up objects and their functions. http://chuck.cs.princeton.edu/doc/program/ugen.html

Step 1: Determine the patch hookup order

Analyze the above diagram. It outlines the basic building blocks we need to build the patch. There are two oscillators (a modulator and a carrier) and two envelopes (one for the index, or gain, of the modulator, and another for the carrier).

Since all of these components are being connected to calculate the resulting sound, they should be "hooked up" in our patch line before being sent to the `dac`. This should looks as follows – note that the modulator's values are essentially applied to those of the carrier!

```
SinOsc m => Envelope envm => SinOsc c => ADSR envc => dac; //modulator to carrier
```

You can use different envelopes if you please – either `Envelope` (which simply ramps to one value and ramps back down in ChucK) or `ADSR` (which has an attack, decay, sustain, and release component – of which sustain is just a pure gain value, and the other three are durations) could be used.

Step 2: Add the line to calculate FM values

ChucK takes care of some of the math of FM synthesis for you through the line below. Include this line. If you were doing FM synthesis 'by hand' (example at the end of this lab), you'd have to handle directly the math surrounding changing the carrier frequency.

```
//set to do fm synthesis
2 => c.sync;
```

Step 2: Finishing the Envelope Functions

We present below a main function, the `playFM()` function, that will call two other functions and run them in parallel – one that controls the envelope of the modulator, `playEnvm()`, and the other that controls the envelope of the carrier, `playEnvc()`. It sets the frequency of the oscillators based on the inputs, the gain of the modulator, the length of the note, and some inputs to control your envelopes.

```
//play function for our FM instrument
//this function calls our two envelopes and runs them in parallel
fun void playFM( float mfreq, float cfreq, float mgain, dur length, float
      mPeakPoint, float cADSR[]  )
{
    //set frequency values
    cfreq => c.freq;
    mfreq => m.freq;

    //open and close envelopes
    spork ~ playEnvm( mgain, length, mPeakPoint );
    spork ~ playEnvc( length, cADSR );

    length => now;
}
```

Note that the last line in this function advances time – you need this line to allow enough time for the sporked functions to run in their entirety!

Critically, Note that we are sending only one value to the `Envelope` function, which controls the envelope of the modulator, to tell it at what time (expressed as a fraction of the total length of the note) it should reach its peak value. By sending in a value of a proportion of the total length, instead of a duration at which it should reach its peak value, we minimize the likelihood that we'll actually send a duration that is *longer* than our total note length! Similarly, we are sending in all four values of the carrier's `ADSR` envelope as floats – the attach, decay, and sustain, are proportions of the total note length, and the sustain is just a gain value. Therefore, all of them are floats by this conceptualization.

Below are our two envelope functions. Since you're getting more familiar with ChucK at this point, we are not going to go through the functions line-by-line, sufficed to say that each function serves to tend to each envelope!

```
fun void playEnvm( float mgain, dur length, float mPeakPoint )
{
    //get value of a half
    length * mPeakPoint => dur mPeakPointDur;
    //set target value for envelope for env1
    envm.target( mgain );
    //set time to reach target
    envm.duration( mPeakPointDur );
    //turn on the modulator!
    envm.keyOn();
    mPeakPointDur => now;
    envm.keyOff();
    length - mPeakPointDur => now;
}

fun void playEnvc( dur length, float cADSR[] )
{
```

```
    //get values for carrier ADSR envelope
    length * cADSR[0] => dur A;
    length * cADSR[1] => dur D;
    cADSR[2] => float S;
    length * cADSR[3] => dur R;

    //set ADSR envelope for envc
    envc.set( A, D, S, R );
    // open envelope (start attack)
    envc.keyOn();
    // wait through A+D+S, before R
    length-envc.releaseTime() => now;
    // close envelope (start release)
    envc.keyOff();
    // wait for release
    envc.releaseTime() => now;
}

playFM( 75, 440, 800, 10::second, .5, [.3,.2,.5,.1] );
```

As in the last lab, you should record your sound as a deliverable. To record the output of a ChucK file to a .wav, include the following in your code before the playFM function:

```
// write to a file
dac => WvOut out => blackhole;
me.sourceDir() + "/LASTNAME_FM.wav" => string _capture;
_capture => out.wavFilename;
```

Label your file as LASTNAME_FM.wav.

For those interested, if you had to actually calculate the carrier frequency values, you'd had to do the following calculations. Essentially, FM synthesis "by hand":

```
SinOsc carrier => dac; //carrier
SinOsc modulator => blackhole; //modulator

220 => float cf; //carrier freq (pitch)
6 => float mf; //modulator freq (vibrato rate)
1 => float index; //index of modulation (vibrato depth)

mf => modulator.freq; //set the modulator frequency

//loop
while( true ) {
   cf + (index*modulator.last()) => carrier.freq; //setting carrier freq
   1::samp => now;
}
```