Title: Markov Music Box Author: Bruce Land

Introduction

I had two reasons for developing a microcontroller-based music device. First, the synthesis and sequencing algorithms are useful for teaching computational methods on small processors to my microcontroller class, ECE4760 in the School of Electrical and Computer Engineering at Cornell University. But the real reason was that I wanted to build my three year old granddaughter an interactive music box. Traditional music boxes play one or two tunes very well, but are not very interactive. Put differently, they have a high quality of synthesis, but a fixed-pattern note sequencer and fixed tonal quality. I wanted to build a device which would play an interesting music-like note sequence, which constantly changed and evolved, with settable timbre, tempo, and beat.

Synthesis Algorithms

To synthesize nice sounding musical notes you need to control spectral content of the note, the rise time (attack), fall time (decay), and the change in spectral content during attack and decay. Also it is nice to have at least two independent musical voices. And all of this has to be done using the modest arithmetic capability of an 8-bit microcontroller. In the class we use Atmel atmega-series MCUs. The scheme used for the attack and decay envelope was to generate the product of two exponentials, a saturating rise exponential and an exponential decay. The process to convert this complicated sounding envelope into fixed point shifts and adds (and one integer multiply) will be covered below. The spectral content of the notes was enriched using FM modulation, which is widely used for musical and special effects. Since directly computing sine waveforms is also mathematically heavy, direct digital synthesis (DDS) was used to produce waveforms. The DDS scheme explained below is used to generate both the FM modulating sine wave signal and the fundamental sine wave for the musical note.

Let's now simplify the exponential envelope calculation. The amplitude envelope chosen is the product of two exponentials, a relatively fast rise and slow fall of the form:

$a(t) = A*\exp(-t/\tau_{fall)})*(1-\exp(-t/\tau_{rise}))$

To generate this envelope at 8 KHz with no floating point, some rearrangement of the math and a few simplifications were done. First we note that the differential equation with solution equal to the desired exponential exp(-t/k) is dy/dt = -k*x. Then we quantize the differential equation to a difference equation using an Euler approximation with time step equal to the sample time Δt, and where n denotes the current sample and n+1 the updated value.

$[x(n+1) - x(n)]/\Delta t = -k*x(n)$

rearranging gives

$x(n+1) = x(n) - \Delta t*k*x(n)$

Next, we allow k to be only a negative power of two, say 2^{-p} , (corresponding a time constant greater than the sample time), and scale k so that Δt equals one, then we can rewrite the difference equation to

x(n+1) = x(n) - (x(n)>>p)

which is just one shift and add per time step to generate an exponential. Of course, the initial condition sets the overall amplitude of the exponential. I set the initial condition to a large, positive, signed 16 bit number, 24,000. The number could have been as big as 32,767 but making it somewhat smaller helped avoid overflow of the integer arithmetic. Factors of two in decay time at first seems coarse, but our hearing tends to log-compress parameters. Each voice implemented required three exponential generators, two for a(t) and one more to allow the FM modulation to fade away (if desired) during a note decay time. Fading the FM modulation allowed more realistic string instrument sounds. Plucked strings have a loud second and third harmonic, but these fade away faster than the fundamental. The exponential decay differential equations were implemented in 16-bit integer arithmetic to get good dynamic range during decay, but only the top eight bits were used to compute the product of the two exponentials. For the falling exponential x(n) above and the rising exponential y(n) (with initial condition y(0)=24000)

y(n+1) = 24000 - (y(n)-(y(n)>>q))

a(n) = (x(n)>>8) * (y(n)>>8)

The a(n) is thus our exponential amplitude envelope, a(t), computed in a sampled, time-efficient way.

The final output, wave(n), is given by

wave(n) = (a(n)>>8) * sin[2*π*F_0*Δt*n + b(n)*FMdepth*sin(2*π*F_{fm}*Δt*n)]

Where F_0 is the frequency of the musical tone, b(n) is an exponential decay term (computed as above), FMdepth is a scale factor to set the strength of the FM effect, and F_{fm} is the modulating frequency. Of course computing a floating sine function is much too slow for real-time synthesis, so direct digital synthesis was used. DDS is a two-step process: (1) at each time step an integer phase accumulator is incremented by a value proportional to frequency, (2) The top few bits of the phase accumulator are used as an index into a lookup table of sine values. For this work a 16-bit phase accumulator was used, along with a 256 entry lookup table. As indicated in the equation above, there are two DDS units required. The first DDS calculates the FM modulation sine wave, which is scaled and added to the phase of the main sine wave DDS. If the array sinetable contains a 256 entry sine wave, scaled to zero to 255, then at each time step, and for 8000 samples/sec and a 16-bit accumulator, accumFM:

accumFM = accumFM + incFM

sineFM = sinetable[accumFM>>8]

with incFM = $2^{16}/8000* F_{fm}$

and for the main note synthesis with accumulator accumNOTE:

accumNOTE = accumNOTE + incNOTE +sineFM*b(n)*FMdepth

sinNOTE = sinetable[accumNOTE>>8]

with incNOTE = $2^{16}/8000* F_0$

We thus get an FM modulated sine wave output for the computational cost of a few table lookups, a few additions and one multiply.

Sequencing Algorithms

Once you can make a single interesting note you need to put together a tune. Rather than store tunes I decided to use a constrained random process to sequence notes. I also decided to use a pentatonic scale because all possible notes tend to sound good together. This makes it easier to have a two voice synthesizer which avoids dissonance.

A controlled random note selection called a Markov process was used to sequence the notes. A separate set of probabilities for playing a potential next-note is stored for each possible current-note. As a note is being played, a random number generator chooses the next note, depending on the relative probabilities stored for the note following the current note. The note transition probabilities were chosen according to either: (1) A power law (see Pitch Structure of Melodic Lines: An Interface between Physics and Perception) so that the probability of transitioning from one note to another was related to the interval between the two notes raised to a negative power or (2) An exponential so that the probability of transitioning from one note to another was related to a fraction raised to a power equal to interval between the two notes.

The probabilities are arranged into matrix format with each row corresponding to the current note being played, and each entry in the row the relative probability for every possible note to be played after the current note. The two forms of the un-normalized transition matrices are shown below (in Matlab/Octave format). After the matrix is computed each row is normalized to a total probability of unity, because when a note changes, it has to change to some other note. In the actual C program, the rows are normalized to 127, so that short integers can be used for selecting the note rather than fractions or floats. To make it faster to compare matrix entries with the random number generator output, the matrix is converted to cumulative probabilities across each row during program initialization.

Power law:

```
s = -0 .75
A = [1 2 3 4 5 6 7 8;
     2 1 2 3 4 5 6 7;
     3 2 1 2 3 4 5 6;
```

```
    4 3 2 1 2 3 4 5;
    5 4 3 2 1 2 3 4 ;
    6 5 4 3 2 1 2 3 ;
    7 6 5 4 3 2 1 2 ;
    8 7 6 5 4 3 2 1  ;] .^s
```

Exponential:

```
s = 0.75 ; % the ratio of one element to the next in a row
A = [ 1 s s^2 s^3 s^4 s^5 s^6 s^7;
      s 1 s s^2 s^3 s^4 s^5 s^6
       s^2 s 1 s s^2 s^3 s^4 s^5 ;
      s^3 s^2 s 1 s s^2 s^3 s^4 ;
      s^4 s^3 s^2 s 1 s s^2 s^3 ;
      s^5 s^4 s^3 s^2 s 1 s s^2 ;
      s^6 s^5 s^4 s^3 s^2 s 1 s ;
      s^7 s^6 s^5 s^4 s^3 s^2 s 1 ;] ;
```

Implementation

In addition to the usual power supply and crystal clock, the circuit required was just an RC lowpass filter tuned to about 10,000 radians/sec connected to the output of the timer zero PWM channel. On the Atmel atMega644, timer zero was set up to run in fast PWM mode at 62,500 Hz with no interrupts turned on. Running the PWM as fast as possible makes it much easier to filter out the PWM sample frequency. The PWM value was updated 8000 or 16000 times/sec by the timer one compare-match interrupt. The timer one compare-match interrupt service routine (ISR) does all of the per-sample computation including updating the envelope and DDS values, then setting the output compare register of timer zero in order to change the PWM signal.

The main program loop runs a button debounce state machine to allow the user to change transition matrix, tempo, beat pattern and timbre for each voice (from a list of presets). Four transitions matrices were used. One was hand coded and allowed only the current note, up one and down one note to form the next note. The other three were selected to use power law encoding with exponents of -2, -1, and -0.75. Tempos were set to factors of two from about 4 notes/sec to 0.5 note /sec. Timbre settings were set by trial and error to sound interesting. They include bell-like, string-like, drum-like and some weird nonmusical twangs and croaks. Beat patterns for each voice are represented as 16 bit integers where each 1-bit represents a note played and each 0-bit a rest. As time progresses, each bit is examined from left-to-right for each voice. You can define a very large number of interesting patterns, but I chose eleven:

```
0b0101010101010101, // 1-on 1-off phase
0b1111111011111110, // 7-on 1-off
0b1110111011101110, // 3-on 1-off
0b1100110011001100, // 2-on 2-off phase 1
0b1010101010101010, // 1-on 1-off phase 1
0b1111000011110000, // 4-on 4-off phase 1
```

```
0b1100000011000000, // 2-on 6-off
0b0011001100110011, // 2-on 2-off phase 2
0b1110110011101100, // 3-on 1-off 2-on 2-off 3-on 1-off 2-on 2-off
0b0000111100001111, // 4-on 4-off phase 2
0b1111111111111111  // on
```

The random number generator I used is a 32-bit linear-feedback shift register design which produces good quality pseudorandom numbers with very little computation. At each call to the random number generator, the 32 bit number is shifted left one bit, then bits 27 and 30 are XORed together and placed back in bit 0. For a nonzero seed (initial value of the random number), this process produces a maximum length (2^{32}) pseudorandom bit stream. At the speed of execution, the bit stream repeats after a month or so. When nothing is happening in the main program, the random number generator is repeatedly called to reduce serial correlation effects. To make a seed for the random number generator I read a channel of the ADC, but with no voltage source attached, just a short wire to pick up noise. The floating analog input varies because of 60 Hz and other pickup, making an unpredictable number to seed the linear feedback shift register.

Results

Spectrograms of the sounds produced look reasonable. Figure 1 shows spectra of a "string-like" and "drum-like" sound both tuned to 131 Hz (C3) produced by FM synthesis. The string spectrum is generally the same shape as a plucked string, but lacks some higher harmonics. Drums produce a variety of non-integer harmonics corresponding to modes of the two-dimensional drum head. This particular drum synthesis exhibits 1/2-harmonic structure and sounds drum or gong-like. Figure 1 also shows spectrograms of amplitude as a function of frequency and time which show that the higher harmonics of a string drop in amplitude faster than the fundamental, while the various harmonics of the drum last longer.

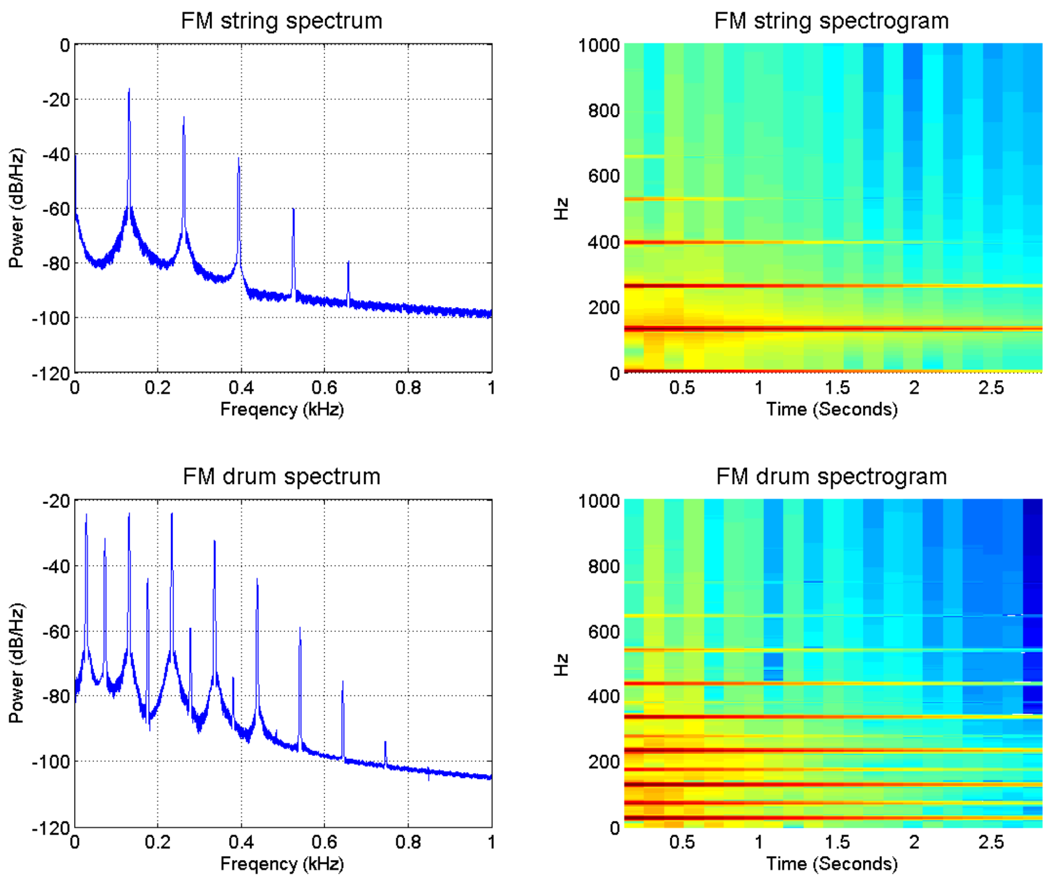

Figure 1. Spectra and spectrogram for two FM notes produced by FM modulation. Both are tuned to 131 Hz (C below middle C). In the spectrograms, red indicates higher energy at a given time and frequency.

But, of course, what matters is what the music box sounds like. I would characterize the musical result as a kind of inoffensive elevator music. It goes nowhere globally, but in a locally pleasing fashion. I leave it playing quietly on my desk to make other noises. I liked the power law probability matrix slightly better than the exponential, but for reasonable values of s, the results were not too much different. The envelope generator and FM synthesis is quite flexible and good quality. Examples of the synthesis may be heard at the link given below.

Conclusion

My students were amused by the Markov Music Box and have improved on it. One group incorporated learning, so that a human trainer could affect the construction of the Markov transition matrix by playing example music on a keyboard (see Auto-Composing Piano, Chaorong Chen and Siyu Zhan).   Another group programmed in familiar tunes. (for example, Fur Elise, see link below).

My granddaughter, however, was more impressed with my Android phone.

References

Markov Music Box Examples and code: http://people.ece.cornell.edu/land/courses/ece4760/labs/s2012/synth_test/Markov_music.html

Auto-Composing Piano, Chaorong Chen and Siyu Zhan http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/cc899_sz259/cc899_sz259/index.html and http://youtu.be/Fttsc25U5tQ

Tunes from students: *Fur Elise*: http://people.ece.cornell.edu/land/courses/ece4760/StudentWork/synth/Fur_Elise.mp3

Pitch Structure of Melodic Lines: An Interface between Physics and Perception, Jorge E. Useche and Rafael G. Hurtado, http://palm.mindmodeling.org/cogsci2011/papers/0843/paper0843.pdf