`bt` - A Realtime Beat Tracker
Preliminary Project Proposal

Joe Colosimo
colosimo@mit.edu
Group 6

April 2, 2013

I'm looking to create a device that receives a live music stream and probabilistically determines a best fit for the music's tempo. This kind of problem has been solved in many ways before, but implementing a hardware architecture that tries to match a running track against possible tempos affords us the opportunity to apply relatively complex signal processing chains to the audio stream to accurately classify beats while still keeping to precise timing requirements.

1 Motivation

Beat analysis is an important field of audio signal processing. It's used in a variety of applications in the music universe. For example, DJ software finds the tempo and beat locations of the tracks in the user's library beforehand in order to make synchronizing two playing tracks a straightforward process. It's also used in audio-controlled lighting to generate complex displays that react to music.

Most of the time, beat analysis is done before its information is actually needed. As a result, these preprocessing algorithms (there are numerous) can be very accurate.

However, I'm looking to solve a slightly different problem: tempo estimation of a live stream in realtime. This one is a little more difficult to solve. In this case, we're looking not only to accurately identify a best estimate for the tempo, but also to converge on that estimate as quickly as possible.

One algorithm I have seen floating around the audio processing community involves trying to classify portions of a music stream as "beats" and then tries to match those beats against one of many possible tempos, eventually narrowing in on a most likely candidate.

This involves creating a series of phase-locking metronomes, each running at a different tempo, and trying to match detected beats to subdivision of each metronome. Figure 1 illustrates the overall architecture of the system.

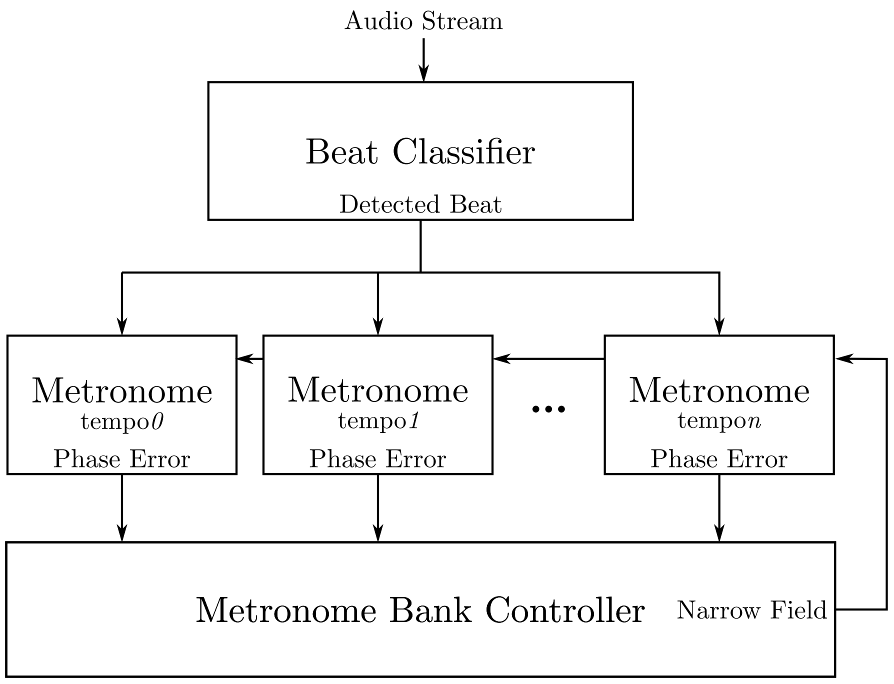


Figure 1: `bt` algorithm architecture

The system starts off by instantiating metronomes with a wide range of tempos. The number of metronomes is limited by the resource constraints of the system. They all wait for the beat classifier to seed them with an initial guess at a beat, at which point they start running. After this initialization phase, the metronomes keep time, each at their own tempo, and when the beat classifier detects a beat in the audio, it delivers a message to all of the metronomes stating the probability that the event it just saw was a beat. The metronomes then all adjust their phase linearly with the probability that the beat classifier thought it was correct and report their current phase difference to the metronome bank controller.

In this manner, metronomes whose tempo is different from the stream's real tempo will consistently report large phase errors, while those closer to the real tempo will report smaller phase errors. We can then use these results to generate a probability distribution for the actual tempo. After a period of convergence, we "zoom in" on the possible tempos, trying smaller and smaller increments around the most likely tempo until we converge on a reasonable result.

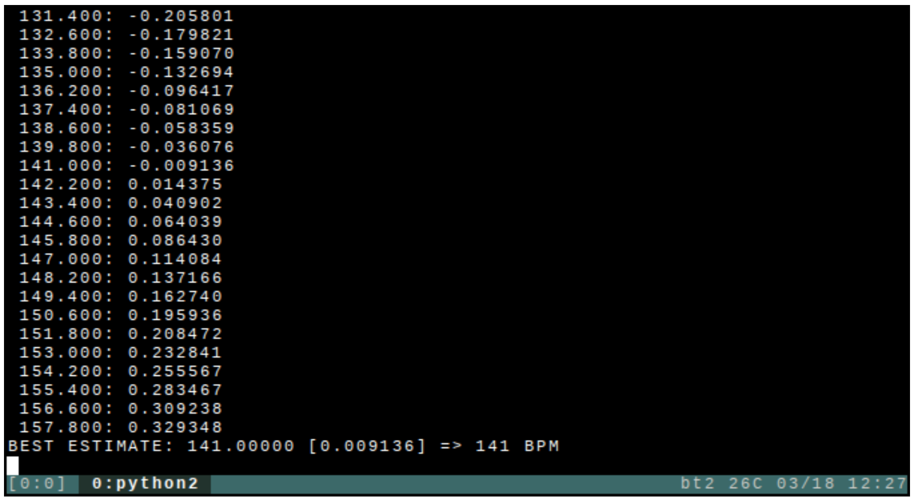
2 Software Implementation

I originally designed the algorithm in software, and even ensured that the software architecture looked somewhat like the hardware architecture that I had in my head. The software simulation illustrates how phase-locking metronomes attempt to correct themselves against injected beats and return a phase error to the bank controller, which then figures out the most likely candidate for the real tempo. Figure 2 shows a screenshot of the metronome bank trying to align to a seed metronome running at 141 BPM, which is just delivering beats. This example converges very quickly because the beat classifier is "100% accurate."

3 Metronome Algorithms

The phase-locking metronomes are extremely simple. Each metronome has a clock running at a multiple of its assigned tempo. I will probably choose a multiple of 4, meaning that the beat will be divided into 16th notes. The beat classifier injects beats into all of the metronomes, providing a probability estimate that what it just saw was a beat.

The metronomes then adjust their current clock phase to be closer to their closest beat (linearly proportional to the probability that the beat classifier thought it detected a beat) and pass the phase error to the metronome bank controller, which generates a probability distribution based on the phase errors of all the respective metronomes.



```
131.400: -0.205801
132.600: -0.179821
133.800: -0.159070
135.000: -0.132694
136.200: -0.096417
137.400: -0.081069
138.600: -0.058359
139.800: -0.036076
141.000: -0.009136
142.200: 0.014375
143.400: 0.040902
144.600: 0.064039
145.800: 0.086430
147.000: 0.114084
148.200: 0.137166
149.400: 0.162740
150.600: 0.195936
151.800: 0.208472
153.000: 0.232841
154.200: 0.255567
155.400: 0.283467
156.600: 0.309238
157.800: 0.329348
BEST ESTIMATE: 141.00000 [0.009136] => 141 BPM

[0:0]  0:python2                                       bt2 26C 03/18 12:27
```

Figure 2: software simulation converging on seed metronome running at 141 BPM

4 Beat Classification Algorithm

Beat classification is the most computationally complex part of this project and possibly the most interesting. There are a few different algorithms that I intend to try. The first will be a very simple peak detector. This algorithm doesn't bother looking at frequency content of the waveform but rather relies on the fact that the energy of a beat is typically much higher than that of the rest of a track, so we can design an algorithm that keeps track of the relative energies of the track in general and events it thinks are beats. As the track continues, this classification should get better.

The second algorithm I'd like to implement is more involved, but much more accurate; it is a variant of one originally designed at the MIT Media Lab.[1] It looks at frequency content of a stream and applies a series of comb filters to various frequency bands. The advantage of this algorithm is that by looking for areas of constructive and destructive interference within individual bands, beats in music with a lot of overall sound energy are detected much more readily.

The link in footnote provides reference Matlab code, meaning that it will be possible to verify my implementation against the original.

5 Hardware

For this project, I plan to use a Spartan 3-AN evaluation board that I already own. I realize that this would result in my operating outside of the SCE-MI

[1]http://www.clear.rice.edu/elec301/Projects01/beat_sync/beatalgo.html

interface, but I'm willing to accept this risk because doing so gets me the extra analog hardware that I need to operate on live audio streams.

The Spartan 3-AN FPGA on the eval board has 700,000 gates, 20 18×18 multipliers, and 360kb of BRAM. Although it's an older FPGA without many of the shinier features added in later FPGAs, such as dedicated DSP blocks, given the targeted small size of my design, I fully expect to stay in the resource constraints of the platform. I have worked with this board before and already have a variety of tools that I've built for interfacing, so during the project time, I will be working on the actual implementation of the beat-tracking algorithm.

The evaluation board has a variety of hardware, but the most important is the onboard ADC and pre-amplifier, which will allow me to sample audio directly from an input jack. My plan is to have two input interfaces to the beat tracker: the first, which will be useful in simulation, will accept digital samples via serial. The second will interface with the ADC, which is stupidly simple to control (send a pulse to initiate a conversion, get a response a short time later). Both of these will provide an identical output interface to the beat tracker algorithm. Figure 3 illustrates the overall hardware implementation plan. The ADC interface and serial interface are part of the hardware test harness. In hardware, one of the two is sent to the sample injector, which sends data to `bt`. In simulation, the sample injector will read from an input data file, but will present data exactly in the same way it does to the `bt` module. The tempo and beat output of `bt` go to the tempo processor interface, which controls the serial controller and blinky LED controller (which makes the onboard LEDs blink to the stream). In simulation, the tempo controller will report to `stdout` the tempo estimate.

I should be able to reuse a major portion of the audio pipeline's test harnesses, especially for developing the simulation environment.

6 Test Vectors

Fortunately, testing this project is relatively simple and can be replicated between simulation and hardware testing. The tempo of most music is well-known; there are also many computer programs that provide accurate estimates. So, primarily, I'll be feeding music tracks into the system and seeing how fast the track converges to its actual tempo as calculated beforehand. Bluespec's cycle-accurate simulation capabilities should make this task fairly easy (I can start with the audio pipeline labs' test driver).

The simplest test vector will be a track containing silence except for beats spaced to some tempo. This is exceptionally easy for the beat classifier, so it should provide a "best-case" scenario. After I can prove the overall functionality of the system, I'm going to feed in music of various genres. Some are easy to classify, such as a many genres of electronic music, and some are much harder, such as rock.

The performance of the system is measured by its accuracy as well as its convergence time.

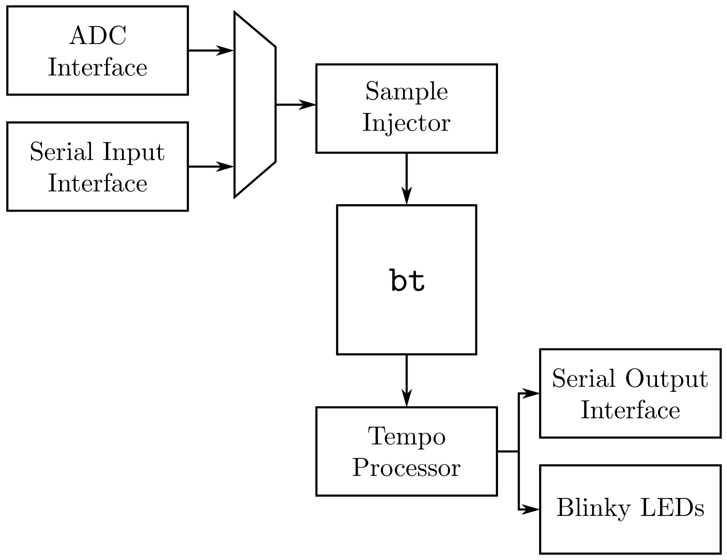


Figure 3: planned hardware architecture

7 Why Implement on an FPGA?

Since I've already written software that can sort of do this, why should I go ahead and implement it on an FPGA? In short, there are some subtle issues with software that I think hardware can do a lot better.

The first is the matter of beat classification. This overall algorithm is fairly robust against mediocre beat classification algorithms that sacrifice accuracy and precision in favor of lower computational complexity. However, I'm quite convinced that with better classification algorithms, the system will converge to an accurate measurement significantly faster.

FPGAs afford the hardware capability to perform complex tasks like FFTs and convolutions with very low latency, which is highly desired in a streaming audio platform. I'm looking to make the convergence to an accurate tempo estimate as fast as possible.

8 Implementation Plan

week	**tasks to be completed**
3/25	surrounding hardware and simulation harnesses
4/1	phase-locking metronomes and metronome bank controller (no field narrowing)
4/8	first pass at a beat classifier (working demo should be available at this point)
4/15	field narrowing for better convergence
4/22	refined beat classifier
4/29	parameter tuning
5/6	debugging and hardware tuning
5/13	final report