BeatLib: A general-purpose beat detection library

Mark T. Tomczak (mtomczak@andrew.cmu.edu)
Advisor: Professor Roger Dannenberg (rbd@andrew.cmu.edu)

April 29, 2005

1 Problem and Significance

For most modern music, the concept of the beat or rhythm is important to both the understanding and description of the music. An algorithm for determining the beat of a song is useful for automatic music interpretation and analysis, classification, interactive music systems, and music visualization. Several algorithms have been devised for locating the beat of a song in real-time[1]. These algorithms have varying effectiveness on different types of music (heavy percussion, light percussion), different tempos, and different time signatures (3/4, 4/4).

In general, the algorithms devised to date tend to look for nearly periodic peaks of a particular feature. However, human listeners probably form an impression of the beat by combining information of several types, including note onsets, drum beats and patterns, and harmonic changes. We hypothesize that beat detection could be improved by considering and combining information about multiple facets of the song. The first step towards such a goal is the construction of a library of two or more beat detectors with a common interface. Such a system would allow for comparison of the efficacy of the algorithms in the same context and would also serve as a structure that higher-level systems could use to combine the information from the detectors to synthesize an overall "concept" of the music input. This capability would improve the overall accuracy of the system.

We are constructing a common beat detection library for the purpose of implementing multiple beat detectors against a common interface. This library will allow for the creation of high-level monitors that can aggregate the information from multiple detectors to generate a more faithful beat tracking response than the individual detectors operating independently.

2 Definition of Beat

Before we could begin detecting the beat, we must first define the concept of a musical beat. Musical beat is a conceptual framework around which the sounds ("notes") in a piece of music are arranged. It is a periodic division of time into distinct and equally-sized units that organize and constrain the musical components. It is important to observe that the phenomenon of musical beat is complex yet strangely universal. Scheirer observes the curious fact that the human ability to perceive musical beat seems to stem not from formal training, but rather from some innate faculty of our acoustic senses. "Nearly every listener, whether skilled or not according to traditional criteria, can find the beat

[1]Goto, Scheirer

in a piece of music and clap her hands or tap her feet to it" (Scheirer 81). It is for this reason that the beat-detection problem is sometimes referred to as the "foot-tapping" or "clapping" problem. The concept of musical beat appears to be a nearly universal perceptual facet of the human auditory system.

From a theoretical standpoint, we can define the beat of a section of music (from time T_0 to T_1) in terms of two values. The first is the period (expressed in units of time), a constant value denoting the distance between beats. The period can be considered the amount of time between claps if a human listener were following the beat of a piece of music. The second value is the phase, which relates the beat to an arbitrary reference point. We can express the phase as the time of the first beat. Thus, given a specific phase ϕ and period T, we could expect a beat to occur at each time $nT + \phi$, where n is an integer.

It should be noted that if we have a pulse signal generator that generates pulses at times $nT' + \phi$, we could satisfy the beat expectation described using any value of T' satisfying the criterion $T' = \frac{1}{m}T$, where m is an integer. For example, if we increase the frequency of the generator by a factor of 4, the generator still generates a pulse on every beat (in addition to three pulses between each beat). This aliasing effect gives musical structure an inherent hierarchy within beats (which is captured in the Western musical scale in the hierarchy of notes: quarter, half, whole, eighth, etc.). This aliasing effect can prove challenging for many beat detection algorithms, which may flip rapidly between fractionally-related values for the period. Another challenge for beat detection systems is the ability of the musical composer to vary the period over the course of a piece of music. While frequent and uncontrolled tempo variance creates noise, a human listener can generally adapt rapidly to a lengthening or shortening of the period of a piece of music. A beat detector must allow for such flexibility even after a "lock" has been achieved on the beat, or else be unable to mimic a human listener. Many beat detection architectures must be finely tuned between the rigidity needed to avoid period-flipping and the flexibility needed to account for beat variance mid-song.

3 Related work

3.1 Beat Detectors

Multiple beat detection projects have been brought to fruition over the past decade, with varying degrees of success. A variety of techniques and design philosophies have been used.

Masataka Goto's detector [1] utilized a combination of chord-change detection, energy analysis, and drum pattern-matching (low and high frequency) to achieve accurate tracking. Assuming a musical structure in 4/4 time, Goto's detector breaks down musical structure at the quarter-note, half-note, and measure level. The detector was able to achieve a correct beat analysis at the quarter-note level on 35 out of 40 songs without heavy drum components (set A), and 39 out of 45 songs with heavy drum components (set B). On average, the system took 10.99 seconds to achieve an accurate beat lock on set A at the quarter-note level, and 13.87 seconds to achieve a similar lock on set B. While this detector performed well, it relies on a 4/4 measure structure to derive higher-level construct elements and takes a great deal of time to achieve tracking lock.

Eric Scheirer's detector [3] utilizes models derived from signal processing theory to perform band-pass filtering on a frequency analysis of musical input. Envelope analysis results from each band are then fed into a bank of resonant filters, which oscillate most strongly with inputs of a specific frequency. The filter bank is then observed for peak resonance, and the resulting peak frequency is reported as the tempo of the music. Scheirer's results showed 68% of songs from a sample of 60 correctly tracked, taking approximately 2 to 8 seconds to achieve tracking lock. The detector had difficulty with music patterned after the *clave* rhythm style (a style that alternates three "clicks" in one measure and two "clicks" in the next measure). Another issue observed was the detector switching between the upbeat and the downbeat for musical styles such as jazz.

3.2 Applications

The primary benefit realizable with effective, robust beat detection would be musical classification. As a fundamental element of musical structure shared by almost all forms of music, robust knowledge of the beat or tempo of a musical sample opens new opportunities for analysis and classification. An example of beat-related classification is the Beat-ID system described by Kirovski [2], which condenses a sample of music into a 32-byte "fingerprint" describing length of beat period and energy distribution within the sample. By comparing the Beat-ID of a sample against a database of Beat-ID information, the system can determine if an input sample matches a known sample—the basis of an audio-based music search engine architecture.

In addition to music modeling and classification, beat detection is also useful for entertainment purposes. Most music consumers would probably be familiar with the concept of a visualizer—a device that responds to musical input, changing state in an appropriate fashion. An example of this technology is the visualization architecture built into the iTunes [2] music organization / playback program by Apple Computer. Besides the built-in visualizer module, the iTunes program also includes a plug-in architecture that allows for the development of other visualizers that can receive data from the iTunes decoding engine. The current version of BeatLib is tested and verified by an engine built upon this plug-in architecture for convenient access to mp3-format music files. Visualizers such as the iTunes module often include simple beat detectors based upon the local maxima of energy across bandwidths. While this information may be sufficient for reactionary visual generation, it usually lacks higher-level information, such as an actual estimate of tempo (beats-per-minute), which would be useful for analysis purposes.

[2]http://www.apple.com/itunes/

3.3 Libraries

While no general-purpose beat detection libraries were found in a study of the literature, there does exist a general signal-processing library: Marsyas [4]. Based around manipulation of arrays of double-precision floating-point values, the Marsyas library allows for the construction of complicated signal processing paths from simpler building blocks using a source-sink routing mechanism for signal data. This is an extremely effective design for several reasons. One reason is that the loose coupling between sources and sinks allows for dynamic re-configuration at runtime. Additionally, the design uses conceptual metaphors ("source/sink") that are familiar to the signal-processing community, which makes the architecture easier to use for the target audience.

One possible disadvantage to the Marsyas architecture is the lack of typing on the messages passed between components. Since every message has the same form (arrays of floats), the components are maximally interchangeable; however, nothing prevents a designer from connecting two logically incompatible components and getting meaningless results. The BeatLib architecture utilizes a source / sink design similar to Marsyas, but also leverages the C++ template system to enforce typing rules between connections. BeatLib also tags each message with a frame identification stamp, which is important for the final re-construction of the beat information.

4 Library Architecture

4.1 Messages

A **message** is a structure containing one instance of data of a specific type (the type is defined by the source sending the message) and a “frame stamp:” a value denoting the time of the message’s generation relative to an arbitrary time T_0 (usually the start of the music input). A **frame** is a fixed-length, nonzero unit of time in which a small sample of musical information may occur. Messages of a specific type are constructed by a source and then transmitted to its corresponding sink (assuming a sink is connected).

4.2 Components, Sources, and Sinks

BeatLib allows for the definition of **components**, which are simple “black box” constructs represented by one **source** and one **sink**. Each source can have assigned to it a single sink, to which messages are then passed as they become available.

To create a new component, a developer creates a class that inherits from both the Source and Sink template classes. The template given to the classes is the type of message that the component can send and receive, respectively; this type information is used to assure that sources and sinks of incompatible types may not be inadvertently connected together (which would allow for the generation of garbage results). In addition, the class must override two virtual functions in the Sink template to allow for processing of incoming messages:

1. `set_data(T &msg)`: Called when a new message is ready for the component to process. The component should use the information in the message as needed and possibly send a message (via its source) to the next component in the chain.
2. `reset()`: Called when the component should reset. The component should re-initialize any internal state (with the exception of any source / sink connections that have been created), such that for the next message received, the component will respond as if it were the first message.

In addition to the functions described above, the Source and Sink classes also provide several helper and utility functions to facilitate message passing.

In general, each component will have only one source and one sink, allowing the components to be connected in a strictly serial fashion. Two specialized classes–MultiSource and MultiSink–violate this rule. In the case of MultiSource, multiple sinks may be attached to the source and all receive a copy of the same message. For MultiSink, a method is provided to generate single Sink objects from the MultiSink, which can be connected like any sink. When each sink associated with a single MultiSink has received a message, the messages are consumed using three virtual functions defined by MultiSink:

1. `begin()`: Called at the start of processing; initializes the aggregation process.
2. `aggregate(T &msg)`: Called once per received message. The MultiSink may act on each message in turn, probably by accumulating some information about the message.
3. `process()`: Called after all messages have been aggregated, so that the MultiSink may compute final results and act on the messages received.

Note that while a component consists of a single source and a single sink, the Source and Sink classes may be inherited separately. This allows for an interface to be constructed between the BeatLib package and external components through the use of inheritance; an external component may inherit from Source to send messages into a BeatLib component chain, or may inherit from Sink to receive final results from a chain.

4.3 Detectors

A **beat detector** is a component that is a source of beat messages. A **beat message** (BeatMsg) is a message that consists of a period 't' and distance 'd,' both expressed in units of seconds. The distance value denotes the number of seconds until the next beat event occurs; it can be used to derive ϕ, but is generated as output instead of the ϕ value because such information is more useful for visualization applications.

While technically the definition of the beat detector is based purely on its output type, most beat detectors will take as input an **energy message** (EnergyMsg). An energy message is a fixed-length array that represents the output of a Fast Fourier Transform (FFT) executed against a musical signal. One energy message is received per frame; the width of the FFT window in time determines the duration of a frame. Each index in the array represents a specific frequency range, while the value in the element denotes the amount of energy in that frequency. The length of the musical subsample the FFT is executed against determines the length of the frame stamp unit in seconds. By maintaining a relation between frame stamp length and seconds, a developer can convert the output values from the detector source into beats per minute or other familiar metrics. Currently, the FFT algorithm is exectued by components external to BeatLib (the iTunes visualizer framework).

The strongly-typed definition of the source and sink of a beat detector allows for interchangeable design. In any given context, any beat detector can be substituted for any other beat detector; conversely, any object satisfying the requirements above is considered a beat detector. This aspect of the design will be helpful in the construction of high-level modules that can combine information from multiple beat detectors to give more accurate results.

4.4 Scheirer Detector

Using this framework, a beat detector similar to Scheirer's detector has been developed. The detector consists of the following components:

1. *FFT splitter (EnergySplitter)*: Divides the FFT array into multiple sequential subarrays. This is similar to a band-pass filter. Each subband will be processed in parallel by an identical filterbank (bank of comb filters of varying lengths), creating a filterbank array.

2. *FFT envelope finder (EnvelopeFinder)*: Determines the maximum energy (envelope) in a range of FFT values. Passes the single energy value forward as an "energy scalar" message (EnergyScalarMsg).

3. *Energy differential (ScalarDifferential)*: Buffers input values to determine difference between subsequent values (first derivative).

4. *Energy rectifier (ScalarRectify)*: Determines the magnitude of the input value.

5. *Resonator*: A comb filter with a specific length. The resonators used in Scheirer's algorithm take an EnergyScalarMsg as a sink and generate a "resonanace message" (ResonanceMsg) as a source. As described in (Scheirer 87-89), the comb-filter resonator feeds the output value proportionally back to the input. For resonators with a comb length near the period of the beat

(in frames), the feedback will tend to increase the output magnitude of the filter. The Scheirer beat detector uses a bank of combs of various lengths to test for multiple potential beat periods.

6. *Resonator output merger (ResonatorMerge)*: Merges the output of multiple resonators of the same width together through a simple summing operation. This component combines the output from each subband in the filterbank, allowing the detector to aggregate resonance information by comb length.

7. *Resonator picker and query tool (ResonatorPicker)*: Analyzes the resonators and chooses the most responsive one as representative of the actual beat. Once the most responsive resonator is chosen, the resonator picker queries the chosen resonator (referenced by a back-pointer stored in the ResonanceMsg) to determine the duration until the next beat. The output of the resonator picker is a "tempo message" (TempoMsg), consisting of the period of the beat and the duration until then next beat.

8. *Tempo to beat converter (TempoToBeat)*: Simple converter class to turn the period information in the TempoMsg to the beat frequency information required of the BeatMsg.

9. *Scheirer Detector (ScheirerDetector)*: Wrapper class to create the entire detector. The ScheirerDetector class can be used to generate all the needed components and organize them in the fashion described in (Scheirer 81-94).

5 Evaluation

Once the Scheirer detector has been constructed, its performance will be evaluated against music files of varying genres with varying tempos. Before evaluation, an experimenter will manually record the "foot-tapping" location of beat events in the music. The detectors will then be executed against the music inputs, and the location of the beats identified by the detectors will be determined by observing the d value's approach to 0. The beat locations will be recorded, and their distance from the human-assigned beats will be evaluated.

6 Future Work

The BeatLib architecture could serve as a valuable framework for building a standardized toolkit of beat detector components. to improve its utility, more detectors could be implemented under the framework—preferably using classes that have already been constructed for the implementation of Scheirer's algorithm. Goto's algorithm for beat detection is somewhat similar, and could serve well as the next test case for the framework. A script parsing system allowing for file-based description of a beat detector configuration could also be created. By augmenting the already-existing factory functions with the ability to consume serialized descriptions, the functions could create detector components and entire detectors from a textual description.

One major hinderance in the design of accurate beat detectors is computational efficiency. A detector must operate at a rate that allows it to keep pace with the musical input; a processing step that is too intensive could result in dropped messages or poor accuracy. If the beat detector is running on a modern multitasking operating system, the amount of processing time allowed to the detector could change dynamically, making performance tuning more difficult. One solution to this dynamic tuning problem could be the addition of a regulator to the detector architecture. The regulator would determine the amount of time consumed in generating the detector output and dynamically throttle non-essential elements of the detector (trading accuracy for speed).

References

[1] M. Goto. An audio-based real-time beat tracking system for music with or without drum-sounds. *Journal of New Music Research*, 30(2):159–171, Jun 2001.

[2] D. Kirovski and H. Attias. Beat-id: Identifying music via beat analysis.

[3] E. D. Scheirer. *Music-Listening Systems*. PhD thesis, Massachusetts Institute of Technology, Apr 2000.

[4] G. Tzanetakis. Marsyas sound feature processing library, 2003.

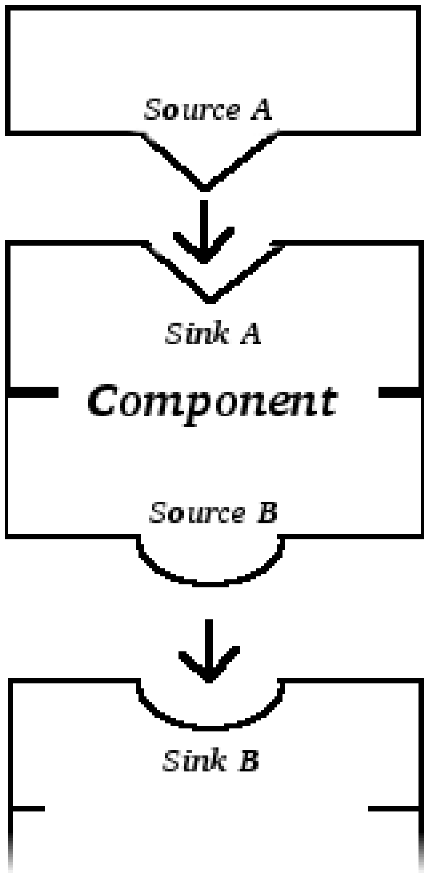

Figure 1: Graphical representation of source-sink relation and component. The shape of the source-sink interfaces denote the template-based types assigned to each source and sink.