Music Clip Identification with Randomized Locality Sensitive Hash Tables

Padraig S. Lysandrou *
Samuel W. Wishnek †
The University of Colorado Boulder, Boulder, CO 80301

The identification of detailed, time-varying signals or data is a nontrivial problem. It is even more-so when the signal is only a truncated portion of the larger function and has exogenous components from a noisy channel. One instance of this kind of problem is identifying a recording such as a song or other piece of music based on a noisy recording of a segment of the piece. This paper proposes an algorithm for associating clips of music with the piece they are sampled from. The algorithm implements the minHash algorithm to efficiently store and compare identifying features of recordings.

Introduction and Motivation

Quickly identifying a piece of music given a very small segment of a song with an arbitrary recording start time is certainly a nontrivial problem. With roughly 35 million potential songs of interest, and sample segments as small as a couple seconds, the identification algorithm must be quick and robust to run on embedded devices. Therefore, there is a lot of room for innovation with regards to computational efficiency and data compression. There has been much interest in this subject, especially with the recent acquisition of identification company Shazam by Apple for $400M.

The heart of this problem is a pattern recognition problem where computation must be abstracted between devices. The initial processing, filtering, and compression must occur on a bandwidth limited device without real-time guarantees such as a mobile phone, and the database matching and identification occurs on a server with significant, and likely parallel, computing resources. Songs can be identified by strictly studying interesting points in the frequency domain, but not robustly. The key is to look not only look at key amplitude and frequency features, but the time differences between them. Ultimately these differential components are what can make a set of signatures for a song that are vastly different from similar sounding music.

Problem Statement and Definition

The core problem is to take a brief recording of music and identify the song that the recording is taken from by comparing the observed data to established database of music. There are several factors to take into account. First, the database of music may be large and possibly represent a large fraction of all recorded music. It would be unfeasible to compare an uncompressed clip to all of the uncompressed database, so the implemented method must have some method of compressing the data both for storage and efficient comparison to find a corresponding piece. Next, the clips may be of arbitrary length and at arbitrary positions within the full piece of music. While accuracy for very short clips is not necessarily expected, the implemented method should be able to accept clips of arbitrary duration. Last, the clips are presumably recorded in an uncontrolled environment and there is a reasonable expectation of noise in the clip. The implemented method must be able to contend with a reasonable level of noise in the recording.

The requirements for the method should begin to illuminate some of the difficulties in designing a solution. Efficient storage and search of the database are the essential aspects of the solution for optimization. Accordingly, viable methods are considered those that could feasibly be implemented for large data sets and could take advantage of parallel computing.

*PhD Student, Entry systems Design Laboratory. Student Member of AIAA.
†PhD Student, Marcus Holzinger Research Laboratory.

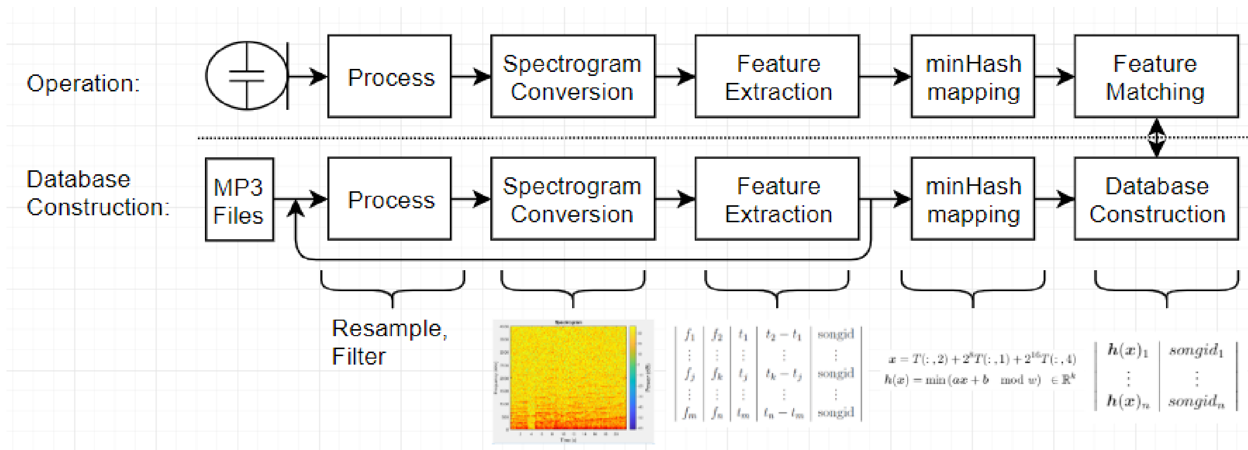


Figure 1: Block diagram describing the numerical processes of our algorithm.

Solution Description and Work Done

The proposed solution is to take the small segment of music and identify it by locating similar features from the clip in an established collection of known, compressed features collected on a server, then return the identity to the user. These features will be selected from a spectrogram generated from the signal. We can create a topological constellation map of peaks where the y axis encodes the positive frequency domain, the x-axis is the binned time domain, and each point has a magnitude, A. The key in precise identification is using pairs of peaks to gather a time and frequency difference. We then record these details in a table which can be hashed and stored in a database, for database generation. A similar process occurs for clip searching: filtering, feature detection and abstraction, hashing, and matching with the database. Figure 1, shows a block diagram of the proposed algorithm.

First the processing, conversion, feature extraction, and hashing functions are discussed. Then the method for how a database is built is described with the same structure and how a clip can be compared to it.

A. Signal Processing and Spectrogram Conversion

First, for both clip and database operations, we must process the time series data. When creating the database, we only consider one of the channels, making our table mono as opposed to stereo. We then **decimate** or **down-sample** the signal not only to save on memory, but also to make our feature extraction much more efficient. Most songs are sampled at 44.1kHz; after experimentation, we found 8kHz to be a good resample rate. Using the MATLAB command $resample$, the signal is interpolated accordingly, and a new result is returned. Additional filtering can be done after this to get rid of high frequency components that are usually insignificant.

The next step is to take the resampled data and convert it in a log-spectrogram. The MATLAB command $spectrogram$ makes this simple:

```
1  [S,F,T] = spectrogram(x,window,noverlap,f,Fs)
```

where x is the song vector, $window$ divides the signal into segments, $noverlap$ defines the number of overlapped samples between segments, f is the desired length of the fast Fourier transform (effectively frequency resolution), and Fs is our new sampling rate. The positive frequency spectrogram is returned in matrix form as S, with F and T being the frequency and time scales. We found good results with the window length and $nfft$ of 0.006 seconds and an $overlap$ of half of this. Multiply these by the operating sampling frequency for integer values of samples.

As mentioned before, the spectrogram is like a waterfall plot in that it displays frequency domain information over time with an additional amplitude dimension which is usually illustrated by a heat map. At this point, the signal is scaled between -1 and $+1$. In order to look at interesting features throughout the spectrogram matrix we perform:

$$log_S = \log_{10}(abs(S) + 1) \tag{1}$$

The unity addition keeps the matrix simpler to work with numerically by shifting the range of realizable values. Using some classical music as an example, we can now visualize the spectrogram in figure 2.

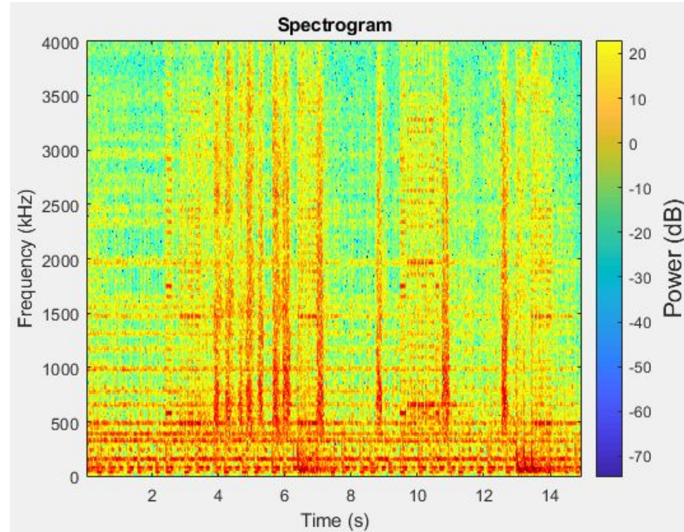


Figure 2: A clip of Darude - Sandstorm after processing and spectrogram conversion.

We can already see features of interest that may populate our feature table.

B. Feature Extraction and Table Generation

Now we must perform one of the most important tasks: selecting significant song features, creating a table of these relatively unique values, and passing them to our hashing function. This is also one of the trickier tasks. As you see in Darude-Sandstorm in figure 2, there can be large areas of dense features. How do you select from these? Our heuristic is to search the log-spectrogram matrix in chunks, looking for local amplitude. The grid search is done in chunk sizes of $p \times p$. This can be done simply by performing a circular shift on the columns as you iterate.

```
1  c_shift = circshift(log_S, [0, -1]);
2  local_peaks = (log_S - c_shift) > 0;
```

In addition to the circular shift search, we make a Boolean matrix $local_peaks$ having 1 entry for the selected features. Making this grid search very fine, we can increase the diversity with further filtering. We iterate through the columns (fixed time, variable frequency) of the matrix, and limit the features to 30 per second. Once this is done, we have a large sparse-Boolean matrix, whose nonzero values act as addresses. The fascinating part of this process is that we have retained no actual information about the signal. Only the location of the values in this new matrix matter.

The row-space address describes the frequency, while the column-space address describes the time at which that event occurs. Now, pattern recognition is inherently based upon differential elements: looking at the difference between two items, images, etc. Our next step is to populate a feature table with pairs of frequencies, their time differences, and an initial time. We must select frequencies that are local to each other in both the frequency and time dimensions. We limit the number of pairs allowed to form an entry in our table of features using a search depth **fan-out**.

A peak located at (t_1, f_1) is only paired with peaks (t_2, f_2) admitted by the following inequalities:

$$t_1 - \delta_t^l \leq t_2 \leq t_1 + \delta_t^u \tag{2}$$
$$f_1 - \delta_f \leq f_2 \leq f_1 + \delta_f \tag{3}$$

The constants δ_t^l, δ_t^u, and δ_f are search terms which allow us to further specify and minimize the entries in our

table. Once we generate a list of 4-tuples admitted by these criteria we can generate our table (4).

$$\begin{vmatrix} f_1 & f_2 & t_1 & t_2 - t_1 \\ \vdots & \vdots & \vdots & \vdots \\ f_j & f_k & t_j & t_k - t_j \\ \vdots & \vdots & \vdots & \vdots \\ f_m & f_n & t_m & t_n - t_m \end{vmatrix} \tag{4}$$

Recall that we have thrown away the actual signal information, and that these entries are simply addresses, or indices, of the frequencies and times attributed to those peaks.

C. Hashing with the minHash Algorithm

With the key features identified, the features need to be encoded in a space conserving format that allows for efficient identification of similar features between clips and the appropriate song. The chosen method creates a deterministic hash for each feature. The pair of frequencies are scaled to eight bit numbers as well as the time delay between the peaks. Then the hash is constructed as the three byte concatenation of the three byte-sized numbers. This calculation is shown in equation 5. A functional song-identification algorithm could be constructed by simply directly storing these hashes for each database song and testing if a clip contains some of the same hashes as a given song. However, there is room for improvement on this by implementing a randomized locality-sensitive hashing algorithm on top of the deterministic algorithm. The deterministic algorithm stores each song's features as a list of three-byte hashes with as many hashes as there are identified features in a given song. This creates a trade-off between the density of identified hashes in a song and the number of hashes to store and compare against. Additionally, longer songs will take up more space.

$$\boldsymbol{x} = T(:,2) + 2^8 T(:,1) + 2^{16} T(:,4) \tag{5}$$

To further compress the data, the deterministic hashes are fed into an implementation of the minHash algorithm. The minHash algorithm was developed in the late 1990's as a method for determining if two documents are similar or one is contained in the other.[1] The hashing function operates as a method for approximating the Jaccard similarity of two sets. In its original implementation, these sets were collections of adjacent letters in a document and referred to as shingles. The Jaccard similarity of two sets is the ratio of the number of elements in the sets' intersection with the sets' union as expressed in equation 6.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{6}$$

The algorithm is able to approximate the Jaccard similarity by generating a series of linear functions with randomly selected values of the slope and y-intercept. Each randomly generated function is applied to every shingle in the document, and the minimum value for each function modulo a prime larger than any possible shingle value is kept and stored as part of a document 'signature.' The equation is shown in equation 7. There $\boldsymbol{x}$ is the vector of shingles in the document, $\boldsymbol{a}$ and $\boldsymbol{b}$ are randomly generated integers less than the maximum possible value of x, and w is a constant prime number larger than the maximum possible value of x. The generated signature will be as long as the number of hash functions generated and independent of the length of the document. As the number of hash functions approaches infinity, the number of identical values in the signatures of two distinct documents should approach the Jaccard similarity.[1]

$$h(\boldsymbol{x}) = \min\left(\boldsymbol{a} \circ \boldsymbol{x} + \boldsymbol{b} \mod w\right) \tag{7}$$

For the music identification problem, the song features deterministically hashed as shown in equation 5 are treated as the 'shingles' of the music, and each piece, whether full or a clip, is treated as a 'document.' The chosen values for selecting frequency peaks yield about one thousand points of interest per minute of a song. Choosing seven hundred randomized hash functions provides a reasonable compression ratio of about 4:1 for typical two to three minute songs. For a database of thirty five million songs, this would total 735 gigabytes and fit in a single terabyte hard drive.

D. Database Construction

Once a popular title is added to our queue of serviceable music, all the steps above are performed for the entire song, and the static k hash map is appended to the bottom of our existing hash table. In addition to the standard array of hashes, we append a classification label seen in table 8.

$$\begin{vmatrix} \boldsymbol{h}(\boldsymbol{x})_1 & song_id_1 \\ \vdots & \vdots \\ \boldsymbol{h}(\boldsymbol{x})_n & song_id_n \end{vmatrix} \tag{8}$$

This is stored on a remote server that the mobile device can access when searching for music.

E. Clip Processing and Identification

Steps **A** through **C** are repeated when a user is trying to identify a song. Of course, in this scenario, the algorithm will operate in a cycle and must work reliably with small segments of songs. Therefore, the processing and hashing steps must be very quick and should not have to send large amounts of data to the server between iterations. Thankfully, the output size of the randomized hashing methodology employed in this paper is deterministic and amenable to online implementation on processors without real-time guarantees.

The only difference between generating the database and identifying the song is the matching step. Once a set of k hashes is generated for the clip, we perform a matching algorithm proposed in 1.

Algorithm 1 Simple Matching Function

1: $Cliphashtable = minhash(make_table(clip));$
2: $k =$ const, $\dim(h(x))$
3: $clip_score = 0$
4: **for** i = 1:N **do**
5: $\quad hashTable_sentence = minhashTable((i-1)*k+1 : i*k, 1)$
6: $\quad local_sens_bool = abs(Cliphashtable - hashTable_sentence) \leq 100$
7: $\quad local_score = \sum(local_sens_bool)$
8: $\quad$ **if** $song_score < local_score$ **then**
9: $\quad\quad song_score = local_score$
10: $\quad\quad songName = i$
11: $\quad$ **end if**
12: **end for**

Given that our hashing scheme is locality sensitive, we can take advantage of this in our search by taking the difference between our hash set and the current iterate set. We generate a Boolean vector and sum this as a value function. During our search we aim to maximize this value function, and once we have determined that it has been maximized, the song identity is returned from the final column of the database matrix. This naive implementation could be improved in a number of ways which is discussed in the alternative approach section.

Alternative Approaches Considered

Initial Signal Compression

An obvious opportunity to implement a randomized sketch would be in sampling the music for database generation. However, in reality it ends up being simpler and more efficient to resample and truncate in the frequency domain. Once the feature selection step comes around, we have already shrunk the song to a sparse matrix of at most a couple thousand bytes. If apriori, we were given a full song and asked to identify it, it may have computational speedups on the matching side. But given that this must operate on-line with small segments of music, applying sketch methodologies did not make the most sense.

Hashing

Alternative locality-sensitive hashing algorithms to minHash were considered and tested against minHash. However no tested algorithm worked as well as minHash. The first alternative tested was a Euclidean distance hashing function. This functions by treating the points of interest in each recording as elements in a vector for a high-dimensional space. Random lines are generated in this space and the point is projected onto the lines. The points is then binned by its position along each line. This effectively splits the high-dimensional space into polyhedra regions

and points close enough to one another are cast to the same hash value.[2] However, clips that match songs are expected to only have a few matching points of interest. Therefore, matches are not similar enough to be reliably detected through the Euclidean distance hashing function as they will almost certainly not occupy the same polyhedric region.

The next tested algorithm was SimHash. SimHash iterates through each point of interest in a recording, represents the value as a binary number, and adds one to the corresponding index in a running total at the location of each one and subtracts one at the location of each zero. When it is done iterating, every element in the running total with a positive value is set to one and every value with a negative value is set to zero. The resulting binary number is the calculated hash. While the SimHash algorithm was created for a similar purpose as the minHash algorithm, namely detecting similar web-pages, the SimHash algorithm does a poor job of detecting low similarity sets.[3] For short clips, the SimHash algorithm does not correctly associate clips with the corresponding songs. SimHash does not begin to become effective until the clip length is nearly the full duration of the song.

The minHash algorithm remained superior to other tested algorithms. Its main advantage over the alternatives is that it is capable of detecting and distinguishing between recordings with no similarities and those with only a few matches.

Matching

During the development process, we realized that the low level matching algorithm is ultimately dependant on the structure of the hashing mechanism. That being said, we did consider higher-level randomized search approaches. As the database table grows to large proportions, this could allow for significant convergence speed ups. Randomized search methods could greatly improve this process. This is something we would have liked to try given more time.

Additionally, parallelized searching could speed up the identification process. If the table were constructed with more information on popularity of searches, music genre, and other interesting labels, on-server search time could be significantly reduced. There would have to exist an additional estimation process to preempt a brute force search as we have implemented here.

Testing Architecture

The following script demonstrates how we tested our algorithm. Songs are pulled from our curated folder of 50, resampled, truncated to form clips, tested, and verified. The variable $testOption$ tells our main function whether or not to regenerate the database hashmap for all of the songs. It takes roughly 50 milliseconds to test a 5 second clip, and more duration testing data can be seen in figure 3.

```
1  testOption = 0;
2  names = [];
3  duration = 5; % Duration of sample clip
4  for i = 1:50
5      % Resample the clip data
6      fileName = ['../Shazam_minhash/songDatabase/',num2str(i,'%02.f'),'.mat'];
7      sample = load(fileName,'-mat');
8      Fs = 8000;
9      y_og = resample(sample.y(:,1),Fs,sample.Fs);
10     q=length(y_og);
11
12     % Calculate the time frame to generate clips
13     clip_start = 0.5*q;
14     clip_end = clip_start + duration*Fs;
15     if clip_end > q
16        clip_end = q;
17     end
18
19     % Truncate to create clip, add noise for robust demonstration
20     y  = y_og(ceil(clip_start): floor(clip_end));
21     y = y + randn(size(y))*1e-1;
22     clip.y = y;
23     clip.Fs = Fs;
24
25     % run the search function on the clip
26     tic
27     [songName] = main_minhash(testOption,clip);
28     names = [names; songName];
29     toc
```

```
30      if songName == i
31          disp(songName)
32      end
33      testOption = 0;
34  end
35  correct = sum([1:50].' == names)/50;
36  disp('done');
```

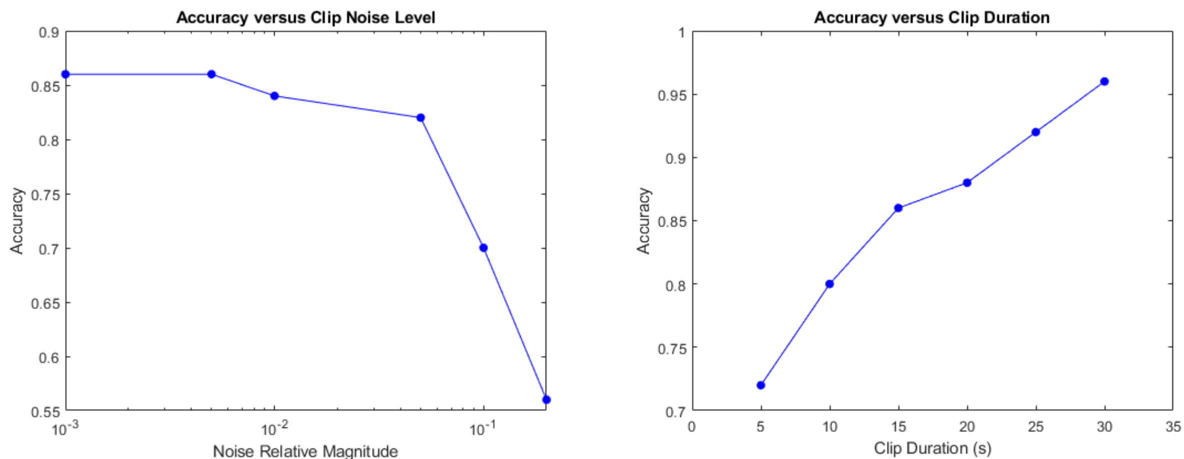
The tested database of music was composed of royalty-free music spanning a variety of genres and a variety of run times. The uncompressed music is about two gigabytes in size. The above testing script allowed several parameters such as the clip's length, position within the song, and artificial noise level to be adjusted to quickly measure the impact of changes on the algorithm's accuracy.

Final Results and Analysis

The testing script is run under a variety of conditions to analyze the algorithms efficacy under various stresses. Primarily the impact of the length of the clip and the magnitude of the added noise is investigated. For the former, the testing script is run for series of clip duration times. A plot of the resulting algorithm accuracy is shown in figure 3(b). The plot shows a near linear relationship between the algorithm accuracy and the clip duration ranging from 72% when the clip length is five seconds and achieving 96% accuracy with thirty second clips.

When noise is added in to the clips the results are unaffected so long as the noise is generated with a standard deviation below 0.005 times the maximum possible signal value. Beyond this point noise begins leading to inaccurate associations. Around 0.05, the noise dramatically reduces the algorithm's accuracy. Interestingly, the algorithm is still capable of making accurate predictions for around 56% of the songs with noise standard deviation at 0.2 times the maximum signal value. At this point the music cannot be heard above the noise and the algorithm continues to make a substantial number of accurate associations. The results for this test are shown in figure 3(a).

Additionally, the majority of our song database consisted of instrumental music, which turns out to be the adversarial case for similar algorithms according to literature. Therefore, we expect our algorithm to yield higher identification accuracy with shorter, lyrical, and more temporally diverse music.



a) Predictably, identification accuracy decreased with the addition of noise. However, the algorithm was still relatively robust for many songs.

b) Predictably, as the duration of the sample clip provided increased, so did the identification accuracy.

Overall, given the difficulty of the problem, our algorithm was surprisingly accurate, and we think this formulation could be useful to a plethora of applications.

Obstacle Discussion

The most difficult part of this project was determining the structure and performing feature selection. Tuning parameters to get the right balance of local and interesting features was difficult, but made the difference in the end. There are a plethora of different ways randomized algorithms could be applied to this problem, but most of the time a simple solution worked so well that the impetus vanished. An example of this is resampling versus sketching or random sampling. Had we built an extensive list of songs, and a large database, we could have certainly employed many more advanced algorithms.

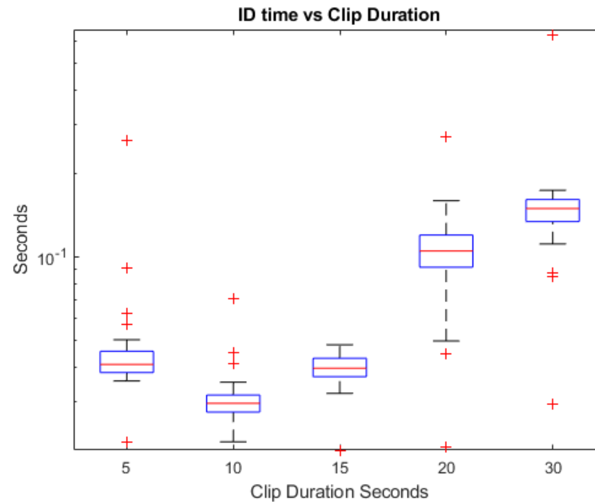


Figure 3: Box and whisker plot showing the spread of execution times for duration of clip. Note the y-axis is logarithmic.

Conclusion

Ultimately, our randomized hashing algorithm allowed us to significantly compress songs, utilize locale sensitivity, and provided deterministic output dimensions. This last point is important in that makes matching easier, provides significant time saving in searching, and makes addressing much simpler during iterative operations. The algorithm accurately associated clips with a database of music and performed the task efficiently in terms of both time and storage requirements with the essential features of two gigabytes of music compressed into a little over 100 kilobytes. The algorithm was fairly accurate even for brief, five second clips as well as clips dominated by noise.

This kind of algorithm is powerful and we already see direct applications to aerospace sensor systems like star trackers. Given a locally stored star catalog, an image, like a spectrogram, can be used to determine a current attitude with respect to reference vectors. Additionally, this methodology could be used for pulsar navigation, allowing autonomous navigation during deep-space operations required for starting an interplanetary colony.

References

[1]Broder, A. Z., "On the resemblance and containment of documents," *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, IEEE, 1997, pp. 21–29.

[2]Wang, J., Shen, H. T., Song, J., and Ji, J., "Hashing for similarity search: A survey," *arXiv preprint arXiv:1408.2927*, 2014.

[3]Henzinger, M., "Finding near-duplicate web pages: a large-scale evaluation of algorithms," *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2006, pp. 284–291.