18-551 Final Report
Group 7 – Fall 2009

Robust Song Identification on a DSK

Marinos Bernitsas (marinos@cmu.edu)

Faraz Chowdhury (frc@andrew.cmu.edu)

Jason Ma (jasonm1@andrew.cmu.edu)

Contents

1. Introduction

The goal of this project is to identify songs based on recorded segments using the DSK. We used a customized algorithm created from scratch that used Short Time Fourier Transforms, a fingerprinting procedure described in [2] and hashing described in [1] to make queries very fast. Total time for song recognition is 1.875 seconds with level 3 optimization.

2. The Problem

As the number of songs being released every day increases, it is very common to hear a song on the radio, in a restaurant, or the mall but not know the title or artist. Our goal is to allow users to identify a song even from a 15-second recording of the song. Our solution is functional even when background noise and speech are present and even in loud recording environments such as clubs and bars. Our solution has to be able to run on devices like mobile phones, so we used 8,000KHz as our sampling rate, which is feasible for all current mobile phones, and is also the sampling rate of the telephone network.

3. Novelty

While commercial solutions for song recognition do exist (Shazam and Midomi SoundHound), they run of large server farms with large song databases of at least 100 million songs. The algorithms are proprietary, and while the overall procedure has been publicized in conferences such as the International Symposium on Music Information Retrieval, the details are omitted.

To the best of our knowledge, this procedure has never been implemented in C in a non-commercial setting before and has never been implemented on a DSK. Our goal is to reengineer these algorithms and optimize them given our limited processing power. This project has not been attempted in previous 551 projects. Its closest match is “Name That Tune: Content Searching for Music Databases” (S03, G11), which performs ‘Query by Humming’. The latter has several problems that our system overcomes: it only works for specific “notable” segments of the song, it fails to identify remixes or covers of the same song, and requires the user to commit to memory a part of the song.

4. Our Solution

The basic procedure is very straightforward. Once the DSK has been set up, the user records a 15-second segment of sound. The DSK then processes the recording and matches it with the one in our database. A GUI on the PC-side displays the song title, artist and album cover.

Our solution has two steps: the training phase and the testing phase (or identification). During the training phase we process a library of songs to generate each one's fingerprint (see glossary), which we then add to our database that is stored using a hash table (see glossary) for increased speed. During the testing phase we record the 15-second segment on the DSK and then process it using the same algorithm to generate its fingerprint. We then compare the fingerprint for the recording to our database and based on the number of matches we either identify, guess or not match the recording to a song.

Our solution for both training and testing uses a three step procedure. Training phase runs on the PC, testing phase runs on the DSK, using the PC GUI.

Step 1: Spectrogram – We generate the spectrogram for the song in question (either the 15-second recording in the testing phase or the full song in our library in the training phase)

Step 2: Find Landmarks – We use our algorithm to process the spectrogram and identify the list of landmarks (see glossary). The landmarks are the unique identifying characteristics of the recording, and are invariant when the song is played again and again.

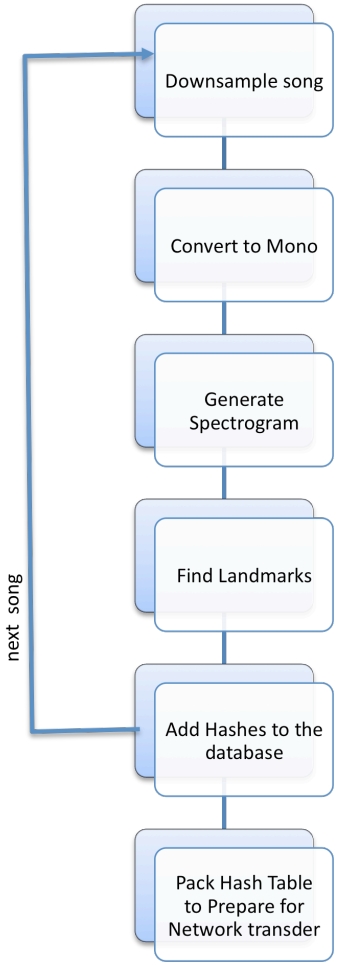
Step 3 (training): Hashing Database - When we are in the training phase we need to find the hash value (see glossary) for each landmark in the song and add it to the database

Step 3 (testing): Identification – When we are in the testing phase we need to go through our landmark list for the 15-second song and for each landmark in our recording find how many songs contain that landmark. According to the distribution of landmarks we output an integer which classifies the song as certain match, guess or no match (see glossary)

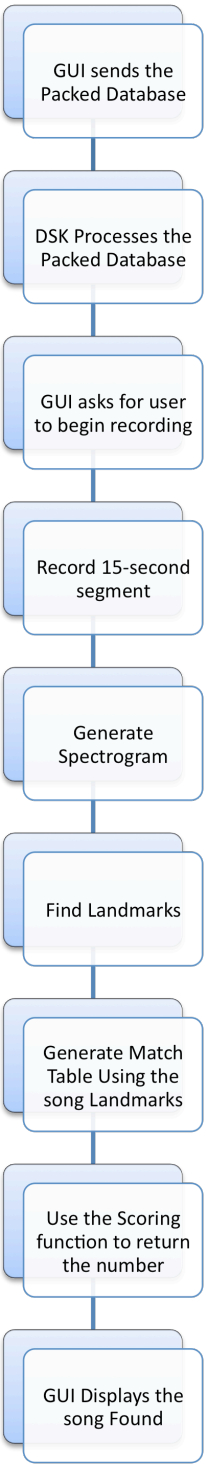
In the following sections we detail our algorithm and the process followed.

5. Flowchart

Training Flowchart



Testing Flowchart



6. DSK vs. PC Breakdown

DSK:

- Records the song through a microphone.
- specgram – computes the FFT of the input audio and generates a spectrogram
- findlandmarks – creates fingerprints
- landmark2hash – creates hash of each fingerprint
- matchLandmarks – looks for a match in our Library

PC:

- Preprocesses song database
    - Downsample and convert to mono
    - Create hash table of hash values for each song
- Receive songID from DSK after match
- Display song name, artist name, and album cover in GUI

7. Spectrogram

The first step to produce a Spectrogram is to first take the Short Time Fourier Transfer (STFT) of the signal. STFTs are used to determine the frequency content of local sections of a signal as it changes over time. We decided to use a 64-millisecond window for each FFT after testing in Matlab with various window lengths [Figure 1]. Using shorter windows causes the algorithm to find and match significantly less landmarks; using a longer window increases the hash values by 1 bit, doubling the size of the hash table, which makes it impossible to store on the DSK. We also used a 50% overlap for each FFT so we won't lose data at the boundaries of the 512 samples. In Matlab, we tested matching results with no overlap and 75% overlap. Zero overlap gives inconsistent landmark matching results due to alignment and 75% only gives a minor improvement in the number of landmarks found. Although 75% gives a slight improvement, it greatly increases memory usage when creating the spectrogram. We used 50% overlap because it gave a good balance between accuracy and performance and memory usage. Since we are sampling our song library and the 15-second segment at 8 KHz, 64ms window contains 512 samples.

$$8 \text{ KHz} / 1000 = 8 \text{ samples per ms}$$

8 * 64 = 512 samples per window

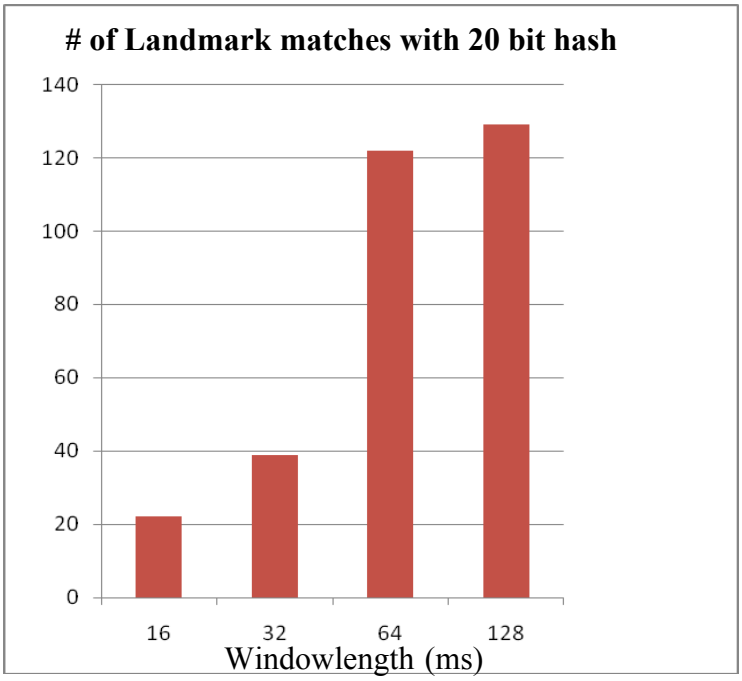
We use the DSPF_sp_cfftr2_dit function from the TS320C67x DSP Library. We chose this FFT function because it needs to be radix-2 for the 512 samples and it is assembly optimized for the DSK. When the function finishes reading in the samples from each song, our function first checks to make sure the number of samples is a multiple of 512. If not, we pad the end with zeros until it is. A hamming window is applied over each 512 samples before it is passed into the FFT function along with the twiddle factors. The hamming window smoothes out the side lobes so the FFT output won't have sharp changes or drops. We tested hamming window against a square window function and it returned more landmark matches. Because the FFT function is radix-2 and complex, the output will be bit reversed and real values will be interleaved with complex values. The next step is to bit reverse and square the magnitude of the output to produce the spectrogram.

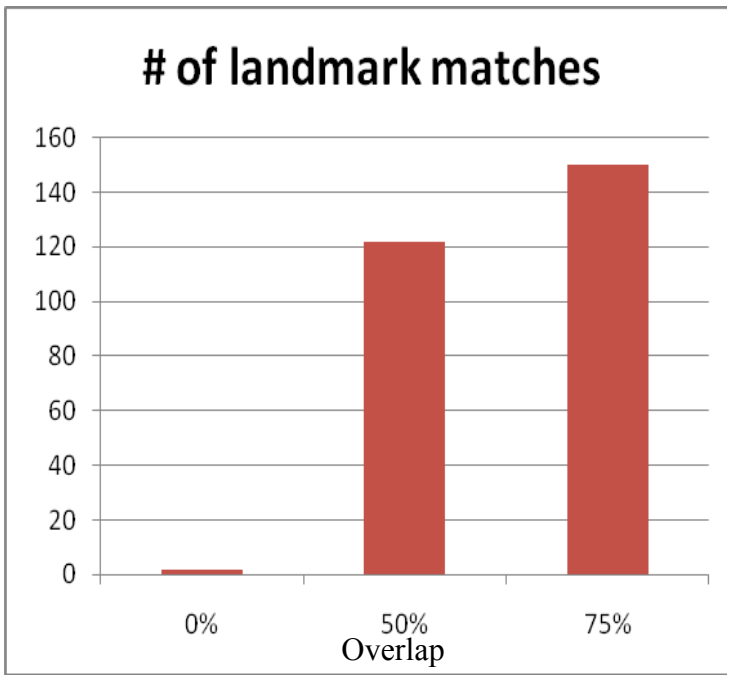
$$\text{spectrogram}(t,\omega) = |\text{STFT}(t,\omega)|^2$$

Number of FFTs we have to take is based on the number of samples. We also have take into account the 50% overlap.

numFFT =(numSample / 256) – 1

We store the entire spectrogram into one array to pass onto the findLandmarks function.





8. Find Landmarks

Find Landmarks is the most crucial step in the process of identifying a song. The Find Landmarks algorithm inputs the spectrogram of the song and outputs a set of Landmarks. Below, we will explain the series of steps that we follow to identify the landmarks of our song. We basically first identify the peaks to create the constellation map and then generate the landmarks to create the fingerprint. These steps are based on [1], [2] and [3] and are all necessary, i.e. the algorithm fails if any of these steps are not present.

1. Input Spectrogram
2. Convert to Log Domain
3. Remove the Mean
4. High Pass Filter Data to remove slowly varying terms
5. Forward Pruning using a thresholding envelope
6. Backward Pruning using a thresholding envelope
7. Landmark Generation

Steps 1-6 are necessary to find local maxima in order to create the constellation map of the song. Step 7 then generates the landmarks.

a. Input Spectrogram

The input to the find landmarks function is the spectrogram of the song, and its length in floats. The number of frequency bins is parametrizable, so the function can handle any type of spectrogram. The spectrogram will be used to identify local maxima on it, which will be pruned to result to the final constellation points.

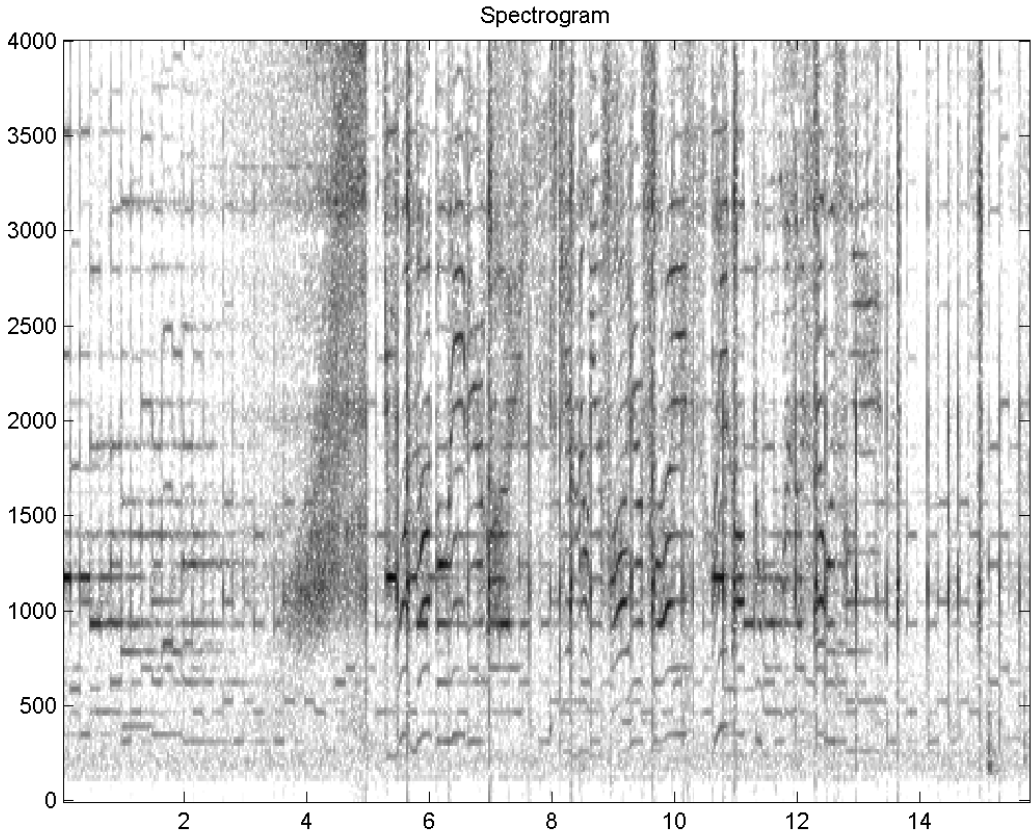


Figure 2

b. Convert to Log Domain

We need to convert to the log domain, because of large exponential differences in the intensity of the spectrogram across a song. Since songs have loud (high energy and frequency content) parts as well as quiet parts (usually vocals and a few instruments), we want to make sure that we can identify segments in each piece of the song, regardless of the intensity. For this reason, taking logs of our spectrogram values is necessary.

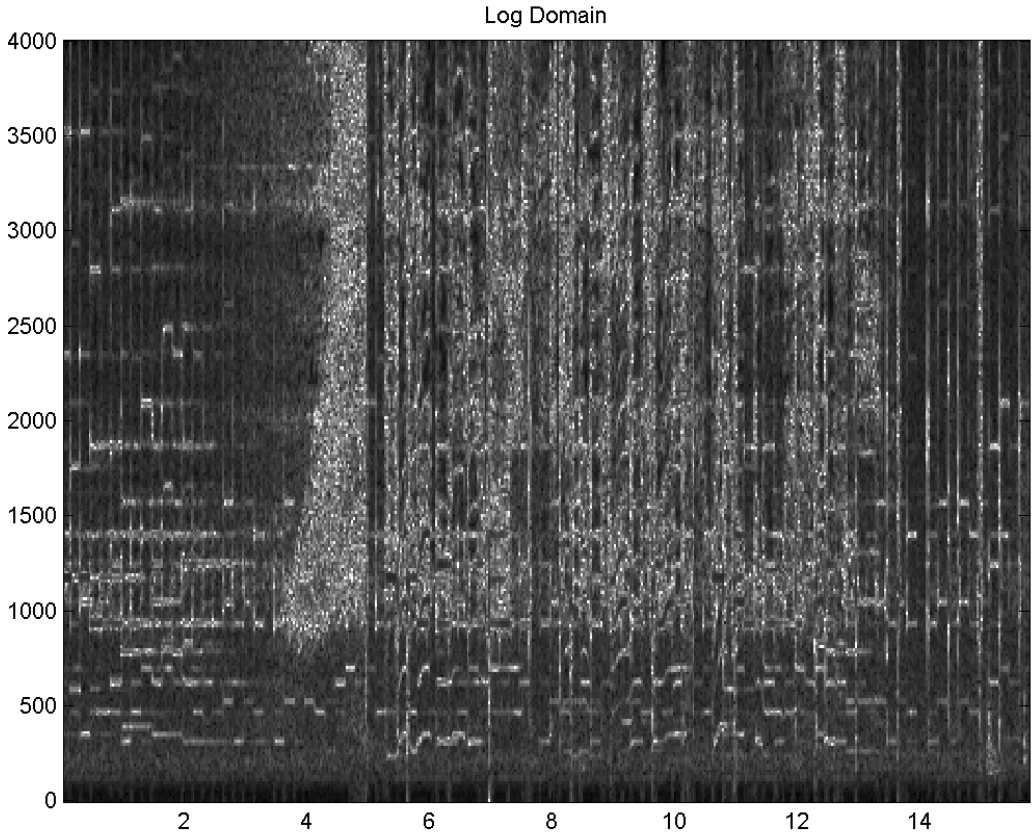


Figure 3

c. De-Meaning

The next step is to remove the mean from the log spectrogram. The mean is calculated by taking the mean of all data points in our spectrogram. We use this because the spectrogram log has shifted the values, and we want to make sure that we have a good distribution of both positive and negative values, for reasons which will become apparent later on, when we filter and apply the thresholding envelope.

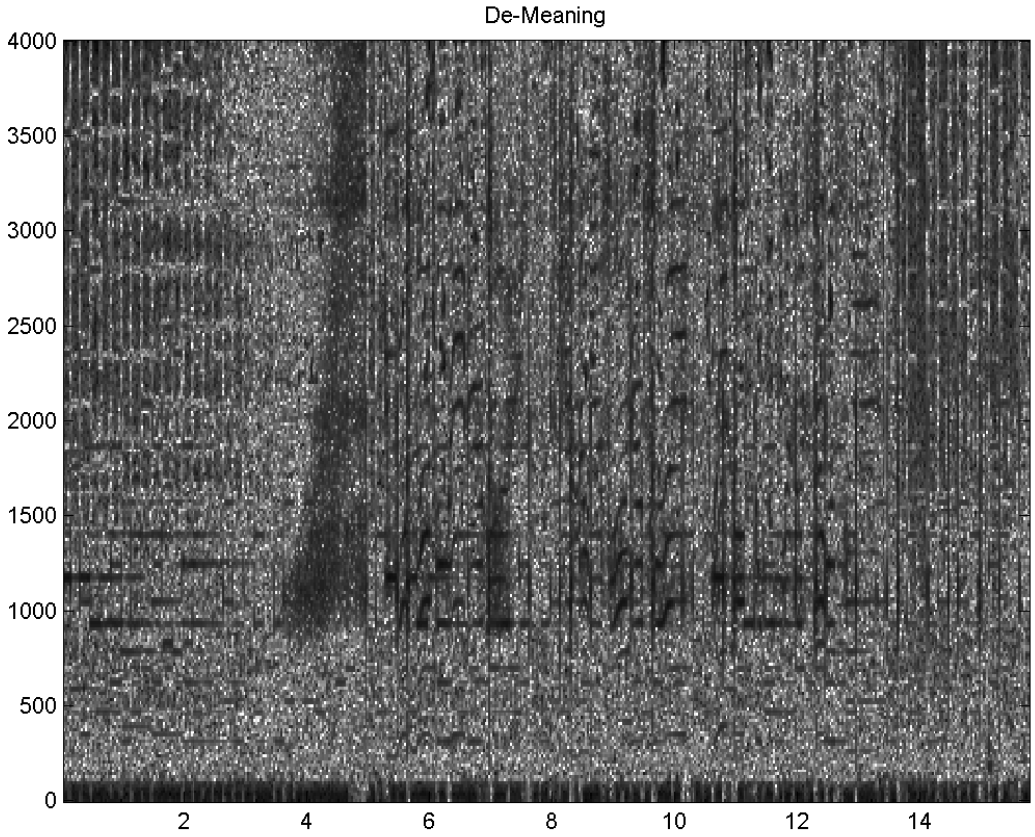


Figure 4

d. High Pass Filtering

This is a high-pass filter, applied to the de-meaned log spectrogram values. It blocks slowly-varying terms, which are problematic for us, because it is impossible to correctly identify their peaks each time the song is played. It also emphasizes onsets and increases granularity, generating more peaks. The filter is basically an ordinary high pass filter with a parametrizable pole of 0.98. The C function we wrote for high pass filtering is `hpf`, which is identical to matlab's `filter([1 -1], [1 -0.98],Specgtrogram')`.

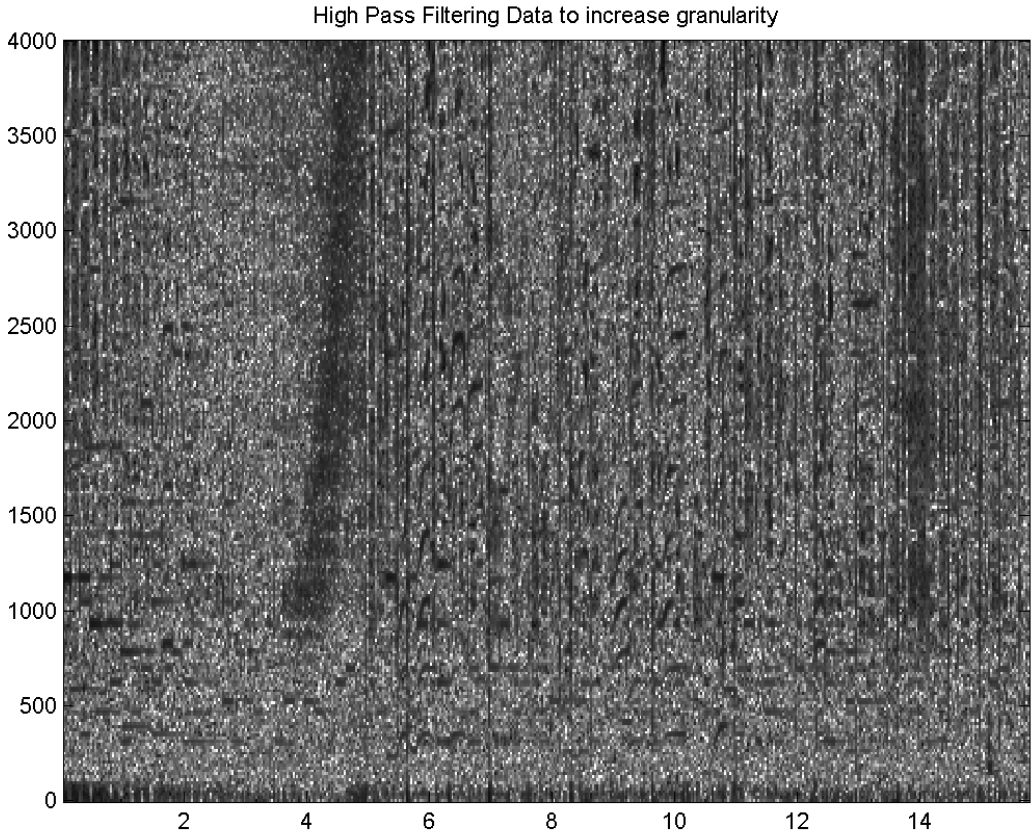


Figure 5

e. Forward Pruning

The forward and backward pruning steps are the workhorses of our find landmarks algorithm. They are necessary for identifying 2-dimensional local maxima, because our points have to be local maxima in the frequency and time domain simultaneously. Forward pruning operates on one window at a time. Each window (a 64ms FFT with 50% overlap) contains 256 frequency bins. Forward pruning is gets its name because it starts from the first time window (t=0) to the last time window in the spectrogram.

Frequency Axis Local Maxima

To identify local maxima on the frequency axis, the `locmax` algorithm inputs the current time window in the spectrogram (256 frequency bins), and returns the frequency bins of the local maxima on the window, sorted by descending magnitude. The functions involved in this step are

`locmax` and `sortfloat`, which is a modified version of quick sort to sort by indexes (frequency bins) and is assembly-optimized.

Time Axis Local Maxima

To identify local maxima on the time axis, a more intricate procedure is necessary, because we want the local maxima to be maxima in two dimensions and for their identification to be clear each time (i.e. they have to be significantly higher than their neighbors to be identifiable each time the song is run. For this reason, we use a threshold envelope.

The threshold envelope is a set of 256 thresholds, one for each frequency in out window. The identification of peaks is then simple: if a point has a higher value (intensity) than our threshold envelope (at that particular frequency) it is identified as a peak. Once a peak is found, the threshold envelope is updated, to reflect the higher value of the peak. In order not to overwhelm ourselves with peaks, we pick the '`maxpksperframe`' first peaks with the highest magnitude. The latest value of `maxpksperframe` used was 5, although it is rarely reached.

Updating the Threshold Envelope

Every time a new peak is found, we have to update the threshold envelope for that peak. Since we want the point to be the highest in a region, we apply a Gaussian around that frequency, weighted by the value of the peak, with a standard deviation of 50 frequency bins. Since the envelope is updated at every step, we take the pointwise max of the threshold and value generated by the Gaussian around the peak. In addition, the envelope decays by 0.01 whenever the time window is advanced. This decay is necessary because essentially it specifies the time range around which the max will live.

When we first run forward pruning we build the threshold envelope based on the first 10 time windows. The threshold envelope is continuously updated after each peak and decayed every time we proceed to a new time window. The use of the threshold envelope is paramount for identifying peaks across the frequency bins because the intensity of the song varies significantly from bin to bin and from one time window to another.

After we have successfully pruned the spectrogram, from time=0 to the end, we are left with a set of peaks stored as <time, frequency, value> pairs and we do not need the spectrogram any

more. This is a big optimization which reduces both access time for the spectrogram and saves storage space.

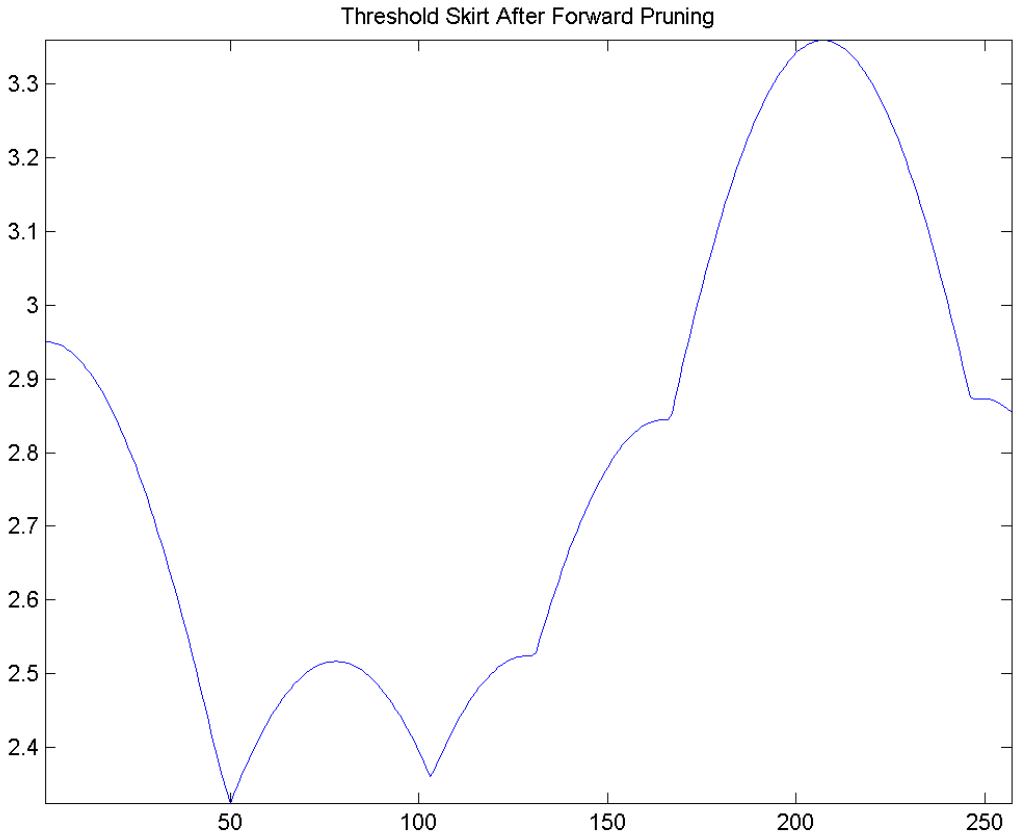


Figure 6

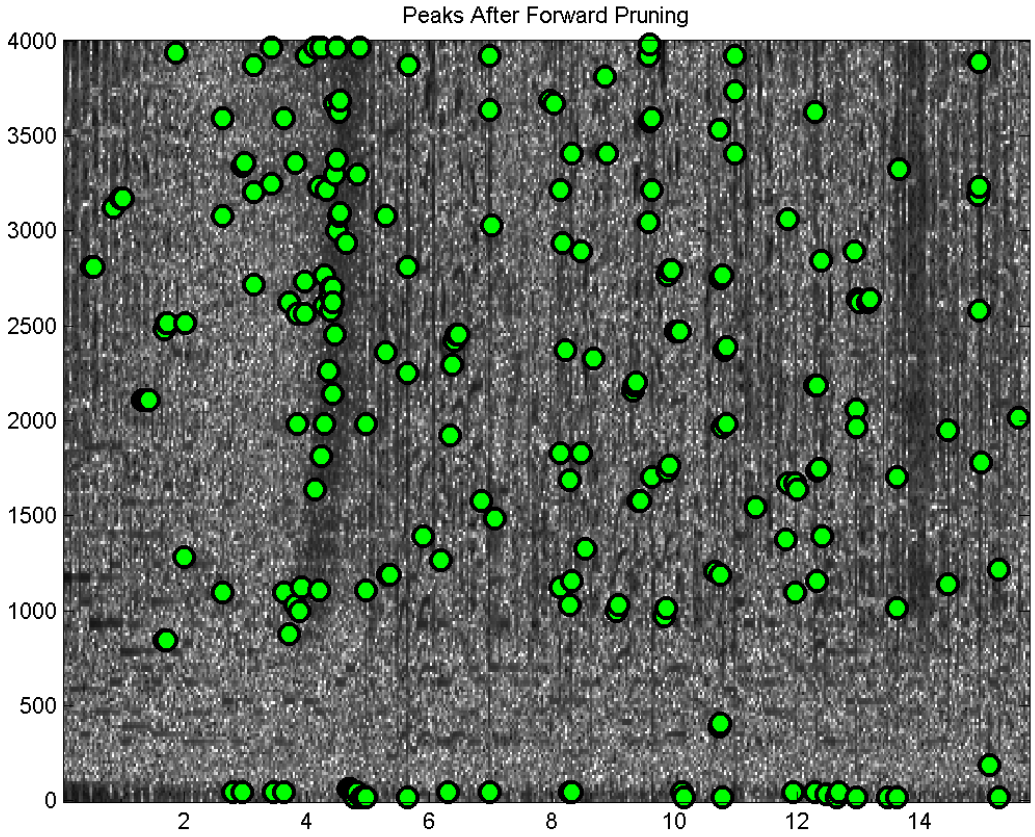


Figure 7

f. Backward Pruning

Backward pruning is essentially the reverse of forward pruning. We start from the last time window and advance backwards to the first time window at t=0. Backward pruning is essential because after forward pruning we obtain a set of points that is a larger set than the set of local maxima. This is because forward pruning only looks for points that are maxima with respect to the time windows it has already seen before the current time window (i.e. for time windows occurring at an earlier time than the current time window's time). To correctly identify the local maxima we need to prune these forward-pruned maxima backwards (i.e. they also have to be local maxima with respect to time windows occurring at a later time than the current time window's). That way each point is a local maximum with respect to both the points to the left of it (looking at the spectrogram) and to the right of it.

Note that it is not necessary that we first forward-prune and the backward-prune. We would have obtained the same set of points if the procedure was repeated, or even if we run both pruning steps simultaneously and obtained the intersection of both steps. Essentially, we need each point to be a local max at the frequency domain, and at the time domain before and after the point. This is exactly what we have achieved after the backward pruning step. We now have the set of true local maxima, which is our constellation map. At this point we do not care about the value of the points, we only care about the time and frequency at which they occur – they have already been identified as peaks.

The algorithm is robust, so every time we have the same song or a piece of the song we identify the exact same constellation points. Now that we have the constellation map, we move on to the last step, the generation of the landmarks.

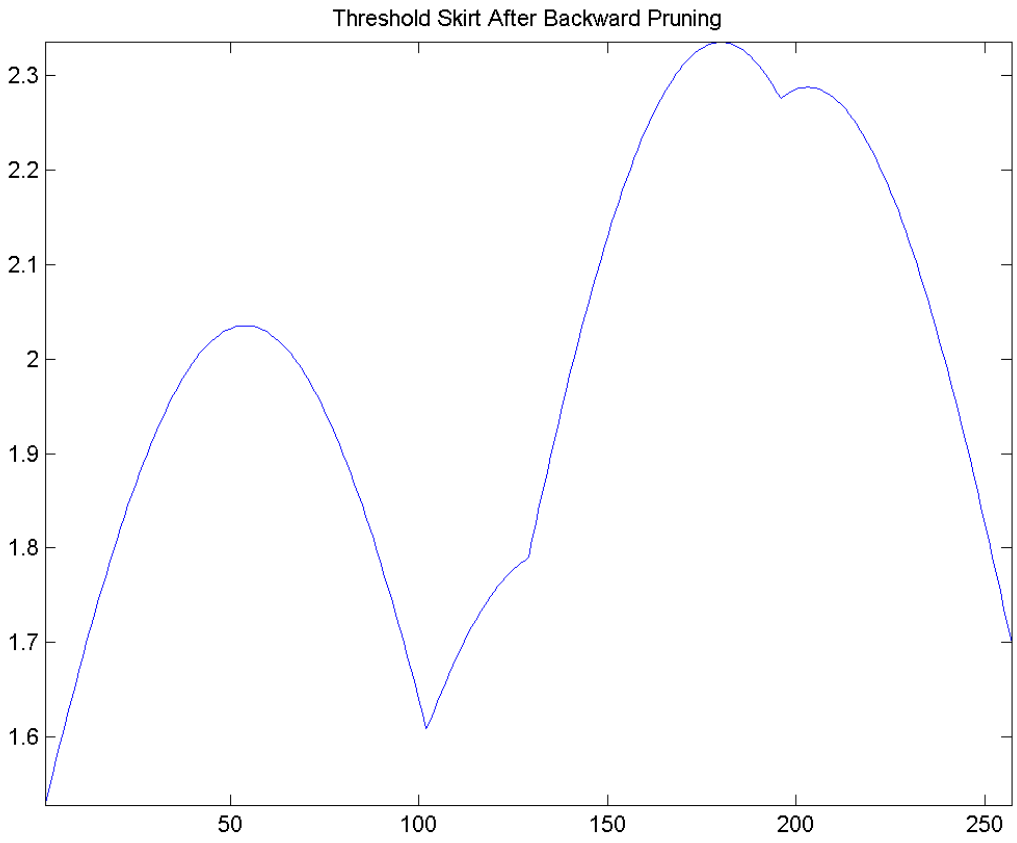


Figure 8

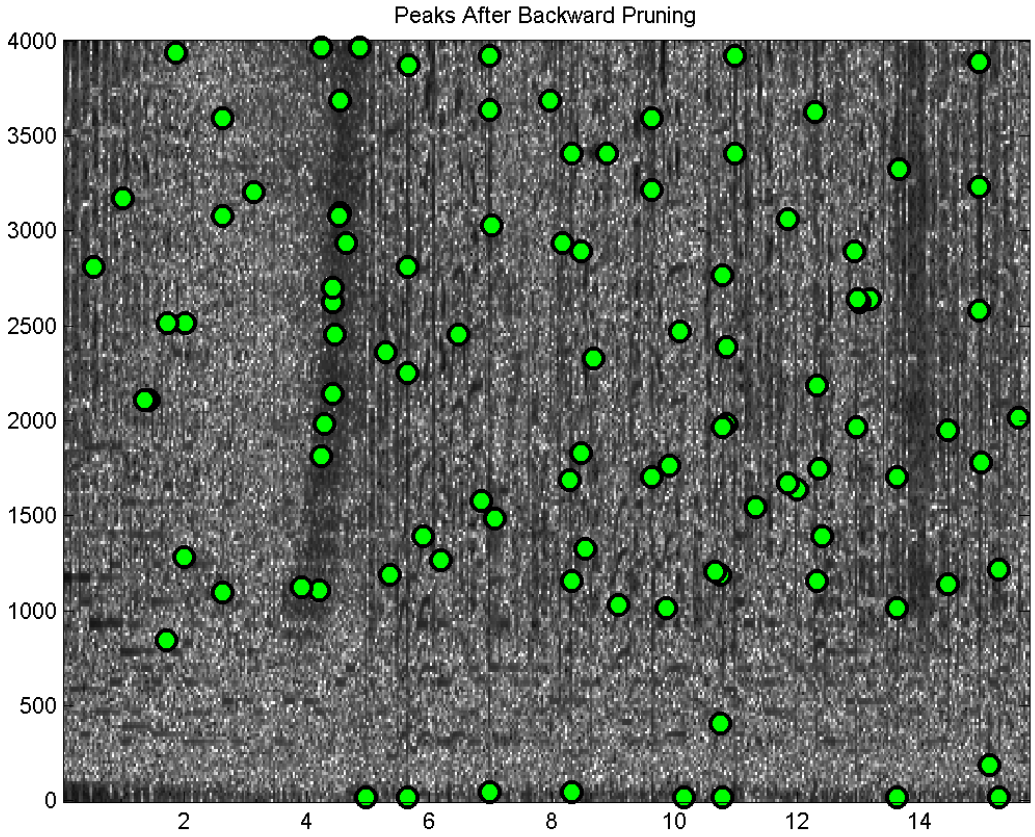


Figure 9

g. Landmark Generation

The landmark generation step converts the constellation map to the set of landmarks, creating the fingerprint for the song. A landmark is a pair of constellation points (<t1,f1,t2,f2> - see the Glossary section for more information). Since the constellation map is unique for each song, so is the landmark, hence the name fingerprint. In fact, since we create multiple landmarks for each constellation point, landmark generation has enough redundancy so we can identify landmarks even if some get lost due to noise of poor recording conditions. While we could have used the constellation points themselves to map the 15-second sample on the recording, we use landmarks because they statistically increase the entropy of the hash function, since there is lower probability that a pair of points has the same values compare to single points. With a small overhead induced by generating the landmarks we significantly reduce the amount of

computation at the identification of the recording, because we don't have to deal with time differences between points.

To generate the Landmarks, we step through the constellation map, selecting each point (referred to as 'the current point' later on) and searching for eligible points to be paired with it. Since we need landmarks to be generated for points that are close enough, we impose the following restrictions on the range (or target zone) and number of landmarks, abbreviated by the parameter name in the program.

- `targetdt`: the maximum look-ahead number of time frames for identifying a pairing point. This identifies the time-dimension range of a landmark. Thus, the only points that are eligible to be paired with the current point are points which belong to a time window greater than the current time window, and less than the current time window+targetdt. The latest value for tagetdt is 63 time windows
- `targetdf`: the number of frequency bins below and above a point for identifying a pairing point. This identifies the frequency-dimension range of a landmark. Thus, the only points eligible to be paired with the current point are points which belong to a frequency bin from f+targetdf to f-targetdf where f is the frequency of the current point. The latest value for targetdf is 31 frequency bins.
- `maxpairsperpeak`: the maximum number of pairs that a constellation point may have. This imposes a cap in case there are more than maxpairsperpeak points in the range. The latest value for maxpairperpeak is 8.

After the landmark generation step completes, it generates a set of landmarks (the fingerprint for the song) which is stored as <t1,f1,t2,f2> 16-byte entries. Landmarks are the red edges in Figure 10.

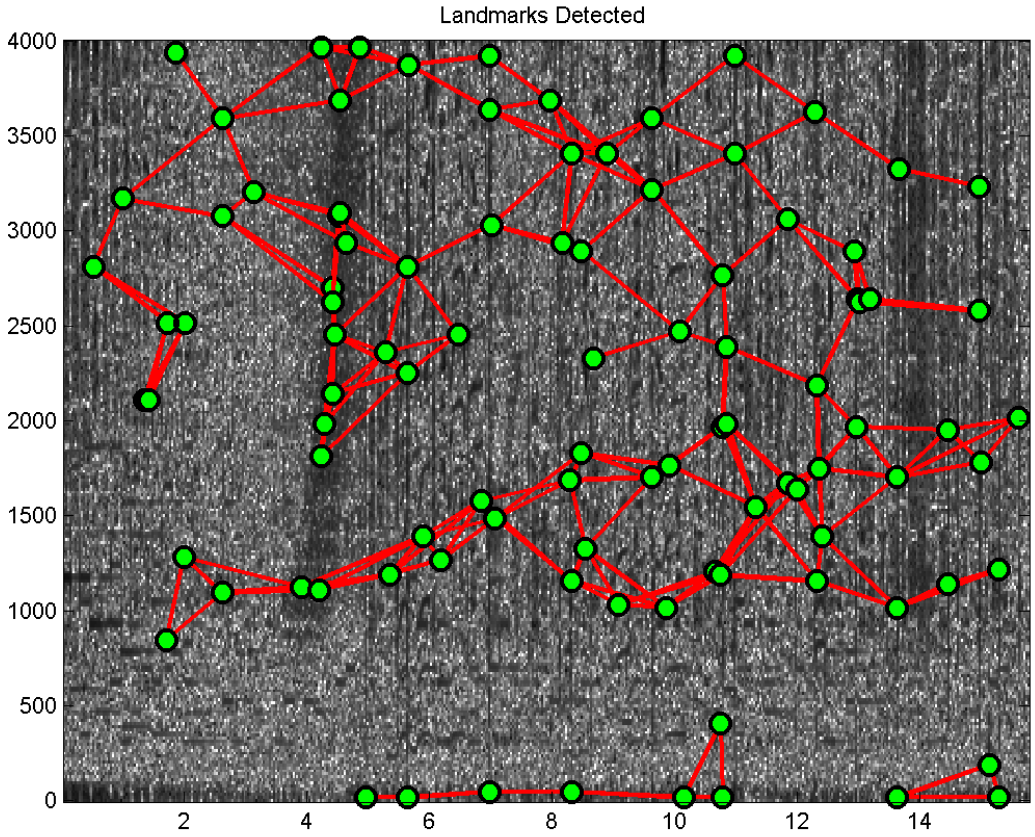


Figure 10

h. Conclusion

Through the series of steps described above, the find landmarks algorithm generates a unique set of points, which are then paired to create a unique set of pair for each song. The redundancy in the pairing process (we have multiple pairs for each point) allows for us to correctly identify a song even if some of the landmarks in the recording are not present, because of noise or bad recording conditions.

9. Hashing Song Database

Hashing is the third step in our program, when running at the training phase, i.e. when we are generating the song database. The hashing procedure accepts a list of Landmarks (or fingerprint) and the songID to which the Landmarks belong, adding each landmark to the Hash Table.

Hash Values for Each Landmark

We are using 20-bit hash values, therefore we have a 2^20 entry Hash Table (4MB) – see Glossary section. Given a Landmark <t1, f1, t2, f2>, our Hash Function uses the 8 bits of the frequency of the first point in the landmark (f1) as bits 19-12 of the hash value. It uses 6 bits of the frequency difference between the two points f2-f1 to as bits 11-6 of the hash value. Finally, it uses the 6 bits of the time difference t2-t1 as the last 6 bits of the hash value, bits 5-0.

The reason we hash time difference dt instead of t1 (which is the number of time windows from the beginning of the song to the first constellation point in the landmark) is that during the identification phase t1 is different, since it refers to the time since the beginning of the recording, so we would never be able to find any landmarks.

There is also a reason for the bit assignments in the Hash Value. We use 8 bits for frequency because frequency is a value between 0-255 (256-bins in the spectrogram). We use 6 bits for time difference and frequency difference because given our criteria for determining which points are eligible peaks (targetdt and targetdf) we know that these numbers will each be from 0-63. This is also the reason why we manually set targetdf to be +-31 frequency bins and targetdt to be 0-63.

As a result, our hash function does not throw away any redundant information. In addition, hashing by pairs creates high entropy which gives a good distribution of points across our hash table.

Hash Elements

Our Hash Table, as mentioned in glossary, does not store any information – it is simply a set of pointers to linked lists which have the same hash values as the indexes to the Hash Tables. The data is stored in the form of Hash Elements, which are basically elements in the linked lists of the form <t1, songID, next>. Note that the hash value for the Hash Element is never stored. Instead, it is inferred by the index of the Hash Table that was used to obtain the particular landmark. t1 is the offset from the beginning of the song to the first point in the landmark. It is used in identification, to make sure that the landmarks in the recording and in the song are in the right order. songID is the ID of the song. It is used in identification to determine which song(s) a particular landmark belongs to. Last, next is the pointer to the next Hash Element, in case there

are more than one Hash Elements with the same hash value. This basically creates a linked list hanging from each index of the hash table. If there is no next, next gets a value of NULL (0).

Generating the Database

To generate the database, our add hash function runs after landmarks, processing the list of landmarks or fingerprint of the song and the song ID. It then uses `landmark2hash` to convert a landmark to a hash element and places it in the index of the Hash Table given by the hash value of the landmark. After this procedure is repeated for every single song, our database is ready.

Querying the Database

Using a hash table makes searches O(1). Given a landmark, we can check whether the landmark belongs to our hash table, by evaluating its hash value through using the hash function. Then all we need to do is access the index in the hash table returned from the hash function and traverse the list. If there are no elements in the list, that landmark does not exist. Otherwise, traversing the list gives us all of the songIDs that contain that landmark.

Packing the Hash Elements

Since it would be a misuse of the DSK to have it process each song in our library and generate the hash table, we have assigned that function to the PC. Once the Hash Table for our song database is created on the PC side, we "pack it" in a CSV file (Comma-Separated Values). All this processing can be done in the training phase, so the packed hash table is generated beforehand and stored. This makes our testing phase PC client really simple, because it only has to transfer the packed hash table to the DSK. The CSV file simply contains an image of the hash table, so for each hash element it generates a triplet <hashvalue, t1, songID>. The triplets are then transferred to the DSK in the training phase. Note that we are not transferring the 4MB hash table, we are transferring the data it points to.

Unpacking the Hash Elements

When the DSK receives the packed hash elements from the PC client, it basically has a set of triplets of the form <hashvalue, t1, songID>. To unpack the Hash Elements, it converts each triplet to a hash element and adds it to the Hash Table in the same way the PC side code initially added it to the hash table. Having the DSK accept a packed hash elements significantly reduces computation time, since receiving the triplets is interleaved with unpacking each triplet,

converting it to a hash element and adding it to the DSK hash table. After the packed hash elements have been successfully transferred and added to the hash table we are ready to start identifying songs on the DSK side.

10. Song Identification

Song Identification is the third step of our program when running at the testing phase, i.e. when we are trying to match a 15-second recording of the song to a song in our database. It runs right after we find landmarks on the recorded segment, and inputs the list of landmarks or fingerprint for the recording. It outputs the songID found, according to our scoring function.

a. Matching

The matching function inputs a list of landmarks and outputs a match table. A match table is an array of size equal to the number of songs in the database, where each row corresponds to the number of matches for the specific song ID (the song ID is implicitly the index of the array, since the song IDs are contiguous and in ascending order).

For each landmark in the landmark list, we find its hash value by using the hash function (landmark2hash), we query the Hash Table, and for each Hash Element we find, we update the count for the Hash Element's song ID in the match table by 1. After we have processed all landmarks in our landmark list we have the completed match table with all the matches for each song.

b. Scoring

The scoring function accepts a match table and outputs a number which describes the certainty of the identification and the song ID (see glossary). It identifies the song IDs for the first and second highest numbers of matches. The output is determined from the following 3 cases.

1. If the number of matches for the first hit is higher than 2 times the number of matches for the second hit and the number of matches for the first hit is greater than 8, we have a Certain Match, so we output the positive of the song ID of the first hit
2. If the number of matches for the first hit is higher than 1.4 times the number of matches for the second hit and the number of matches for the first hit is greater than 8, we have a Guess, so we output the negative of the songID of the first hit.

3. In all other cases we output 0 (no match)

Note that if the number of landmarks for the 15-second recording is less than 100, then we consider it 'noise' or 'too quiet' so the find landmarks procedure returns an empty landmark list. Then by following the identification algorithm we can see that we get a 0 output returned in this special case.

The integer returned after the song identification step is then transferred across the network as a message and used to display the song information on the GUI.

11. Data Structures

A number of data structures are used for the needs of our code. Below, we detail each one and note its size in bytes. For more information about their use, please look at the glossary, as well as the respective sections in which they are discussed.

HASH: The hash value. It is an integer, whose first 20 bits are only used (4 bytes)

LANDMARK <t1, f1, t2, f2>: A struct of four integers holding the coordinates (t1,f1),(t2,f2) of a pair of constellation points (16 bytes). To generate a landmark list we use LANDMARK *

HASHELEMENT <t1, songID, next>: A struct of two shorts for t1 and songID and a pointer for next, which forms the linked list pointed to by an element in the hash table (8 bytes). We were able to get this down to 8 bytes from 12 by fitting t1 into a short.

PEAK <t,f,v>: As struct of two ints for time and frequency of the peak and a float v for its value (12 bytes).

TRIPLET <HV,t1, songID>: A triplet is an entry in the packed hash table, storing the hashvalue, time and songID for each Hash Element in the database. It uses an int for the hash value HV and two shorts for t1 and song ID, minimize the number of bytes that need to be transferred over the network. (8 bytes)

12. Storage

Our program is very demanding in terms of storage, because of the database that resides on the DSK side. Below, we detail our storage allocation for maximum use of the 16MB of the DSK.

Hash Table: Global Variable in external memory which is paged into internal memory to speed up identification. It has 1M entries so it is 4MB long and is of type HASHELEMENT *.

Song Database: The Hash Elements, which are the actual data for our songs. For 58 songs (maximum DSK storage allocation with no cap for network buffer) it takes 4.91MB. For 116 songs (maximum DSK storage allocation with capped network buffer of 100KB) it takes 9.81MB.

Spectrogram: Two global variables in external memory which holds the spectrogram and the filtered spectrogram (from the high pass filter). The spectrogram demands a total of 0.92MB.

Network Buffer: The network buffer stores the triplets from the packed hash table that is transmitted through the network. It resides in external memory. We have two options: either use a network buffer which is malloced to be equal to the size of the hash table or use a capped network buffer of 100KB. The former consumes a lot of capacity, during the transfer, which is OK since we are freeing all that heap space right after the transfer is complete. The latter is statically allocated at compile-time and causes slower network transfers, because we have to process the 100KB of triplets before we can overwrite the buffer. Its main advantage is that we can get a lot more songs on the database of the DSK, at the cost of slower speeds. For our demo we used a large network buffer, malloced to the size of the hash table in order to speed up network transfers. For 58 songs (maximum DSK storage allocation with no cap for network buffer) it takes 4.91MB.

Code (.text): Our code size was 70KB.

The table below summarizes the storage allocation for the faster case of 58 songs (maximum DSK storage allocation with no cap for network buffer):

	Storage Usage (Bytes)	Storage Usage (MB)
.Text	71,424	0.07 MB
Hash Table	4,194,304	4.00 MB
Landmarks	5,144,832	4.91 MB
Network Buffer	5,144,832	4.91 MB
Spectrogram	480,256	0.46 MB
Spec_filtered	480,256	0.46 MB
Recording Int	481,280	0.46 MB
Recording Float	481,280	0.46 MB
Findlandmarks Static	10,240	0.01 MB
Findlandmarks Heap	120,000	0.11 MB
Miscellaneous	168,512	0.16 MB

13. Data Flow

Our system starts out by setting up a server on the PC side. A network connection is created between the PC and DSK. Then, the PC side creates the hash table of the song database on the PC. PC sends a message that contains the length of the song database and says it is ready to transfer the hash table. The song database is then transferred to the DSK. To indicate the end of the song database transfer, the PC will send a message to the DSK. Next, the DSK sends a message to the PC/GUI telling the user that system is ready to start the recording process. The user now can either to choose to exit the system or begin recording. If the latter is chosen, the PC sends a message to the DSK that begins the 15 second recording process. DSK then sends a message to the PC to indicate the end of the recording process and the start of the landmark finding and matching process using the algorithm. Once the matching process is over, the DSK returns the results to the PC/GUI where it is displayed to the user. The user will again have the choice to exit or start another recording process.

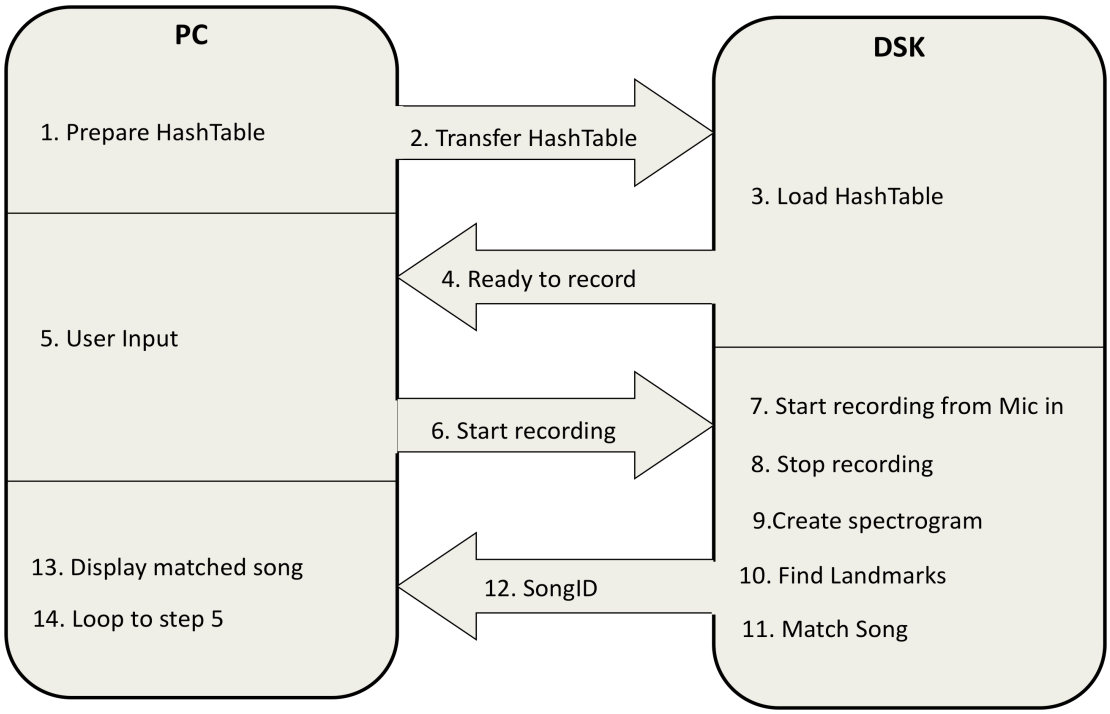


Figure 11

14. Network Transfers

The PC and DSK communicate in two ways over a TCP/IP connection: they can send and receive either a message or data. Four functions for transfers are on both side: sendMsg, recvMsg, sendData, recvData.

The purpose of the send and receive message functions are to send codes to make sure the PC the DSK are sending the correct amount of data at the right time. The code number is to let the each side know the progress on the other side.

```
#define EXIT_MSG                    2
#define START_HASH_TRANSFER_MSG     3
#define END_HASH_TRANSFER_MSG       4
#define START_RECORD_MSG            5
#define READY_RECORD_MSG            6
#define END_RECORD_MSG              7
#define SONGID_MSG                  8
```

The sendMsg function always sends 2 ints (8 bytes). The first int is the code number and the second contains the value. The value is always zero unless the code is `START_HASH_TRANSFER_MSG` or `SONGID_MSG`. When sending `START_HASH_TRANSFER_MSG`, we also send the hash table length as the value. When sending `SONGID_MSG`, we also send song

ID as the value. The purpose of the send and receive data function is just to send a payload. The only time this function is used is to transfer the hash table from the PC to the DSK. We transfer 1000 bytes at the time. We tested this with transferring the entire payload at the time and transferring less bytes. In our code, we used 1000 because it gave the highest speed out of the numbers we tested.

15. Training Data

Training data is the set of songs that the recording from the DSK will try to match to. The songs are first downsampled to 8kHz and converted to mono. The songs are all downsampled to 8kHz because most recordings done using cell phones are in 8kHz, and our program simulates a cell phone application. Next they are preprocessed on the PC into a database based on their hash values. This song database is then transferred over to the DSK for the testing phase to commence.

To compile our final training database of 30 songs we tested different songs to create a compilation of different types of songs. The database contains songs from many genres, ranging from rock and pop to country and ballads. We wanted to see if there is a correlation between genre and recognition accuracy. Also, we picked some loud and high energy songs such as "Poker Face" by Lady Gaga and also some quiet songs using very few instruments such as "Love Song" by Sara Bareilles, to see how they affect recognition. Other determination categories used were: songs that sound similar, different songs by the same artist, etc. The following is a list of the 30 songs used in our final database:

All Star - Smash Mouth
Alright - Darius Rucker
Beautiful Day - U2
Beautiful Girls - Sean Kingston
Bye ByeBye - N'Sync
Dirty Little Secret - All American Rejects
Down (feat. Lil' Wayne) - Jay Sean
Fly Away - Lenny Kravitz
Friends in Low Places - Garth Brooks
Gotta Be Somebody - Nickelback
Hanging By A Moment - Lifehouse
Hips Don't Lie - Shakira

I'm On A Boat (feat. T-Pain) - Lonely Island
It's my life - Bon Jovi
It's Not My Time - 3 Doors Down
It's the End of the World - REM
Kryptonite - 3 Doors Down
Living on a Prayer - Bon Jovi
Love Song - Sara Bareilles
Poker Face - Lady Gaga
Second chance - Shinedown
Sleep Now In The Fire - Rage Against The Machine
Such great heights - The Postal Service
Sweet Home Alabama - LynyrdSkynyrd
Tearin' Up My Heart - N'Sync
The Remedy - Jason Mraz
The Seeker - The Who
Use Somebody - Kings of Leon
Whatcha Say - Jason Derulo
Wonderwall - Oasis

16. Testing Data

Testing data consists of the 15 seconds recording done by using the DSK. The recorded segment is at 8kHz sampling frequency so can be matched with the training database. The recording is then matched with the algorithm. The testing data was recorded under different noise conditions. We tested the algorithm in normal lab hours with considerable amount of noise in the background. We also used white noise generator [6] along with the song to simulate different levels of noise. Finally we tested the algorithm by minimizing noise as much as possible, to see how behaves under perfect conditions. Also to vary the testing data, songs that are not in the training database were played. Finally we used multiple parts of the same song for different tries in the testing phase.

17. Test Results

To test our algorithm we used a database of 30 songs and thoroughly tested each song to determine recognition accuracy. For each song we tested the algorithm starting at 0 seconds of the song, 30 seconds, and 60 seconds of the song. We performed 2 separate, independent tries for each starting point to determine the precision of the algorithm. In the table above, the first six columns of data represent the outcome of the algorithm for the 6 tries (2 starting at 0 sec, 2 at 30 sec, and 2 at 60 sec). For each table entry, a + indicates a certain match (see glossary), a - indicates a correct guess (see glossary), and a 0 indicates no match (see glossary).

The columns labeled "Certain Matches for all 6 tries" displays the total number of +'s for the song. The column labeled "Certains andGuesses for all 6 tries" displays the total number of +'s and -'s for the particular song. On average, for all 30 songs if we count all 6 tries then the recognition accuracy for certain matches only is 55% and the recognition accuracy for certains and guesses is 81%. However, we have discovered that we achieve much better results if we ignore the first 2 tries starting at 0 seconds of the song. The reason behind this is, our algorithm cannot really find many landmarks and thus cannot find many matches if the song is too quiet. The program always returns a no match, if the recording has less than 100 landmarks. It is easier for the algorithm to find more landmarks and thus more matches for song segments that have high energy content, i.e. strong and loud vocals, different instruments, etc. For many of our songs, the beginning is usually very quiet since it sometimes has a silent pause, or a quiet single instrument, or just really slow and quiet vocals before the song picks up the pace. Therefore, trying to recognize the song after the first 15-20 seconds really improves the recognition accuracy. The 11^{th} and 12^{th} columns in the table above show the number of certain matches and number of certains and guesses, respectively, for each song in the database. The average recognition accuracy now improves to 60% for certain matches and 90% for certains and guesses.

All the testing was done in lab with varying amount of noise in the background. As can be seen from our test results, there are a few cases where we matched a song with certainty for one try and then the next try, playing the same part of the song, failed to match the song. (Ignoring the beginning part of the songs and the first two tries, there are 3 cases where one try results a + and

another results a 0, for the same section of the song). We attribute this anomaly to the different levels of background noise in lab, and most probably in these 3 cases the noise was loud enough to overpower the song causing the algorithm to fail. As I have mentioned earlier, the algorithm cannot match songs properly if the recorded segment is too quiet. If there is too much noise present, the noise overpowers the song and makes it seem quiet to the algorithm even if it is not in reality. In an ideal case, if there is no noise present whatsoever, then the algorithm will return a certain match 100% of the time. We tested the songs trying to reduce noise as much as possible, and the algorithm works fine. However, it is hard to completely suppress all noise as some noise is always present from the microphone and speakers. We used a regular headset microphone to do our recordings and played the music from laptop speakers. As these instruments are not of great quality noise is already introduced by the speakers and the microphone and affects our algorithm even if we did not have much noise in the lab. The microphone and speaker noise were troublesome for some of the quiet parts of the song. But without any other lab noise we always have either a certain match or correct guess.

We also tested the algorithm by recording just white noise and not playing an actual song. This resulted in very few landmarks and thus outputted a no match every time (since less than 100 landmarks is detected in the white noise segment). We also tried playing a different song not in the database, and while that generates a lot of landmarks from the recordings it does not create that many landmark matches with the songs in the database. If the highest number of landmarks matched is less than 8, then it means it is a different song and no match is outputted by the algorithm.

We have tested different genres of songs, different songs by the same artist, and same song by different artists, to test the robustness of the algorithm. Before choosing our final database of 30 songs, we tested a variety of songs to determine a good assortment of songs for our database. We discovered many interesting correlations between song types and recognition accuracy. Songs from the rock, pop, metal, etc. genre are more easily identified by the algorithm than songs belonging to country, ballads, classical or soft rock genre. Rock and pop songs have very high energy content as they usually have loud vocals and lots of instruments playing in the background. This means there are more landmarks identified in these songs making the

matching easier. On the other hand, country songs and ballads are usually quieter in general and sometimes have low monotonous vocals and sometimes single or no instrument. Songs like these have lower number of landmarks making it hard for the algorithm to find matches. Often, we received guesses instead of certain matches for country songs or ballads. For example, "Hips don't lie" by Shakira (a pop song) was matched with certainty every time, where as "Friends in low places" by Garth Brooks (a country song) was sometimes matched as a guess or no match, as the noise overpowered the already quiet song. We also tested different songs by the same artist and discovered that the algorithm can correctly differentiate between the two songs. Next, we tried testing the same song by two different artists, and that worked as well as long as both the songs were in the database. We tested "Don't stop Believin" by Journey and a cover of the same song done by Glee, and the algorithm could easily distinguish one song from the other.

This table is continued in the next page

		Returned Values					
ID	Landmarks	Start at 0 sec Try 1	Start at 0 sec Try 2	Start at 15 sec Try 1	Start at 15 sec Try 2	Start at 30 sec Try 1	Start at 30 sec Try 2
1	All Star - Smash Mouth.wav.txt	+	-	+	+	0	+
2	Alright - Darius Rucker.wav.txt	+	-	-	+	+	-
3	Beautiful Day - U2.wav.txt	+	0	+	+	-	+
4	Beautiful Girls - Sean Kingston.wav.txt	+	+	+	+	+	+
5	Bye Bye Bye - N'Sync.wav.txt	0	-	-	0	+	0
6	Dirty Little Secret - All American Rejects.wav.txt	+	-	-	+	+	+
7	Down (feat. Lil' Wayne) - Jay Sean.wav.txt	0	+	0	-	+	+
8	Fly Away - Lenny Kravitz.wav.txt	0	+	+	+	+	+
9	Friends in Low Places - Garth Brooks.wav.txt	0	0	-	+	0	-
10	Gotta Be Somebody - Nickelback.wav.txt	0	-	-	0	+	+
11	Hanging By A Moment - Lifehouse.wav.txt	+	-	+	+	+	+
12	Hips Don't Lie - Shakira.wav.txt	+	+	+	+	+	+
13	I'm On A Boat (feat. T-Pain) - Lonely Island.wav.tx	-	+	-	+	0	-
14	It's my life - Bon Jovi.wav.txt	-	+	0	+	+	+
15	It's Not My Time - 3 Doors Down.wav.txt	-	0	-	0	-	-
16	It's the End of the World - REM.wav.txt	+	+	+	+	+	+
17	Kryptone - 3 Doors Down.wav.txt	0	0	+	-	-	+
18	Living on a Prayer - Bon Jovi.wav.txt	0	0	-	+	+	-
19	Love Song - Sara Bareilles.wav.txt	+	0	-	+	-	-
20	Pker Face - Lady Gaga.wav.txt	-	0	+	+	+	+
21	Second chance - Shinedown.wav.txt	0	-	-	+	-	0
22	Sleep Now In The Fire - Rage Against The Machin	+	-	-	-	-	-
23	Such great heights - The Postal Service.wav.txt	+	+	+	+	+	+
24	Sweet Home Alabama - Lynyrd Skynyrd.wav.txt	+	+	-	+	-	-
25	Tearin' Up My Heart - N'Sync.wav.txt	+	+	+	+	+	+
26	The Remedy - Jason Mraz.wav.txt	+	-	-	+	+	+
27	The Seeker - The Who.wav.txt	0	0	+	-	+	+
28	Use Somebody - Kings of Leon.wav.txt	+	+	-	-	0	+
29	Whatcha Say - Jason Derulo.wav.txt	+	+	0	-	+	+
30	Wonderwall - Oasis.wav.txt	0	0	+	-	+	+

Legend:
if + means song matched correctly with certainity (first song's hits > 2 x second's hits)
if - means song guessed correctly (2 x second's hits > first song's hits > 1.4 x second's hits)
if 0 means no match with a song

Figure 12

		For all 6 tries		For tries 2 - 6		For tries 2 - 6	
ID	Landmarks	Certain Matches Total # of +'s	Certains and Guesses Total # of +'s and -'s	Certain Matches Total # of +'s	Certains and Guesses Total # of +'s and -'s	Certain Match Accuracy	Certain and Guess Accuray
1	All Star - Smash Mouth.wav.txt	4	5	3	3	75%	75%
2	Alright - Darius Rucker.wav.txt	3	6	2	4	50%	100%
3	Beautiful Day - U2.wav.txt	4	5	3	4	75%	100%
4	Beautiful Girls - Sean Kingston.wav.txt	6	6	4	4	100%	100%
5	Bye Bye Bye - N'Sync.wav.txt	1	3	1	2	25%	50%
6	Dirty Little Secret - All American Rejects.wav.txt	4	6	3	4	75%	100%
7	Down (feat. Lil' Wayne) - Jay Sean.wav.txt	3	4	2	3	50%	75%
8	Fly Away - Lenny Kravitz.wav.txt	5	5	4	4	100%	100%
9	Friends in Low Places - Garth Brooks.wav.txt	1	3	1	3	25%	75%
10	Gotta Be Somebody - Nickelback.wav.txt	2	3	2	3	50%	75%
11	Hanging By A Moment - Lifehouse.wav.txt	5	6	4	4	100%	100%
12	Hips Don't Lie - Shakira.wav.txt	6	6	4	4	100%	100%
13	I'm On A Boat (feat. T-Pain) - Lonely Island.wav.tx	2	5	1	3	25%	75%
14	It's my life - Bon Jovi.wav.txt	4	5	3	3	75%	75%
15	It's Not My Time - 3 Doors Down.wav.txt	0	3	0	3	0%	75%
16	It's the End of the World - REM.wav.txt	6	6	4	4	100%	100%
17	Kryptone - 3 Doors Down.wav.txt	2	4	2	4	50%	100%
18	Living on a Prayer - Bon Jovi.wav.txt	2	4	2	4	50%	100%
19	Love Song - Sara Bareilles.wav.txt	2	5	1	4	25%	100%
20	Pker Face - Lady Gaga.wav.txt	4	5	4	4	100%	100%
21	Second chance - Shinedown.wav.txt	1	3	1	3	25%	75%
22	Sleep Now In The Fire - Rage Against The Machir	1	6	0	4	0%	100%
23	Such great heights - The Postal Service.wav.txt	6	6	4	4	100%	100%
24	Sweet Home Alabama - Lynyrd Skynyrd.wav.txt	3	6	1	4	25%	100%
25	Tearin' Up My Heart - N'Sync.wav.txt	6	6	4	4	100%	100%
26	The Remedy - Jason Mraz.wav.txt	4	6	3	4	75%	100%
27	The Seeker - The Who.wav.txt	3	4	3	4	75%	100%
28	Use Somebody - Kings of Leon.wav.txt	3	5	1	3	25%	75%
29	Whatcha Say - Jason Derulo.wav.txt	4	5	2	3	50%	75%
30	Wonderwall - Oasis.wav.txt	3	4	3	4	75%	100%
					AVERAGE	60%	90%

Figure 13

18. Optimizations

We used a number of optimizations in our code. Some are algorithm optimizations (i.e. modifying algorithms so they can run faster and with more efficient data structures). Others are DSK-specific optimizations, such as parallelizing loops and paging.

Speed Optimizations

- Using list of peaks instead of spectrogram in backward pruning. This significantly reduced storage overhead and processing time.
- Paging Hash elements (song database) from external to internal memory. Since the hash elements were created at the same time, we were able to page sections of our database from external to internal memory in our matchlandmarks function, speeding up the identification step.
- Maintained consistent double-word alignment. We made sure that both our FFT and our database had double-word alignment. In the case of the FFT, double-word alignment was necessary for it to work correctly. In the case of the database, we found that it led to speed improvement, because of L1 caching.
- Reduced HASHELEMENT size from 12 bytes to 8 bytes. This led to both storage reduction by 33% and also faster access times, because of L1 caching.
- L2 cache was turned off, not only to preserve storage space, but also because there is no need for it - we do not access the same elements twice, unless they are in internal memory, in which case we would use the L1 cache. This also gives more space for our songs.
- L1 cache was on, because the elements in internal memory are frequently accessed, so we determined there would be a big storage improvement.
- Rewrote processing-intensive loops in spectrogram and find landmarks functions to create parallelism. We hand optimized the most demanding loop in spectrogram and find landmarks (forward pruning). We were able to reduce speed by a few 1,000's of cycles, but due to the number of on-chip variables we were not able to get phenomenal improvements.

Quality Optimizations

- Once we had our algorithm working we worked to fine-tune the parameters of the findlandmarks algorithm. In order to better identify songs with a few landmarks (quiet songs) and songs with quiet sections we increased maxlandmarksperpeak to 8 from 5, which it was initially. This gave us an increased recognition in noisy environments for quieter parts of the songs.

19. Profiling

During the testing phase, after it is done recording the whole algorithm takes 1.875 seconds to complete under optimization level 3. The algorithm can be broken down into three main phases: spectrogram, find landmarks and create and match hashes. Creating the spectrogram takes 124,892,584 cycles, or 0.55 seconds. Find landmarks is 170,539,367 cycles – 0.75 seconds. Finally, creating and matching hashes (called matchsongdsk) takes 129,349,607 cycles or .57 seconds.

For level 0 optimization, the algorithm takes 4.16 seconds to complete. The spectrograms takes 259,361,475 cycles or 1.13 seconds. In theory it should take around 235,200,000 cycles. The find landmarks function takes 368,724,632 cycles – 1.62 seconds to complete. In theory it should take 330,000,000 cycles. Finally, the matchsongDSK takes 321,769,782 cycles or 1.41 seconds, whereas theoretically it should take 290,500,000 cycles.

20. The Graphical User Interface

All the network transfers are written in C to communicate with the DSK. The Graphical User Interface (GUI) was written in Matlab and communicates with the C code to tell the DSK to start recording and to display the final song information after the matching is completed. Once the hash table has been transferred to the DSK and it is ready to record, the Start button GUI is enabled allowing the user to start recording. Once the button is pressed both the start and exit buttons are disabled and the DSK starts recording for 15 seconds. After the recording it automatically searches for a match and returns the songID. The songID is the position of the song in an array, which has been sorted alphabetically in ascending order based on song names.

Once the GUI receives the songID from the DSK it displays the song name, artist name and album cover associated with that songID. If the songID is returned as a certain match then the GUI displays the song and artist as portrayed below. The GUI also plays the particular song that was matched and the stop song button is then enabled. The start and exit buttons are then re-enabled.

Figure 14

If a song is not a certain match and the DSK returns a guess, then the GUI still displays the song name, artist, and album cover and plays the song, but also displays a warning sign. If the algorithm matches the song as a guess, then it returns a negative songID value to the GUI. The GUI displays the song with the warning message, "WARNING: Not sure but the closest match I can find is the above", which means that the song displayed was a guess.

Figure 15

If the algorithm fails to match a song then it results in a no match and it will return the songID zero to the GUI. The GUI then displays “NO MATCH” as song and artist name. The user can record another 15 second segment by pressing start or quit the program by pressing exit.

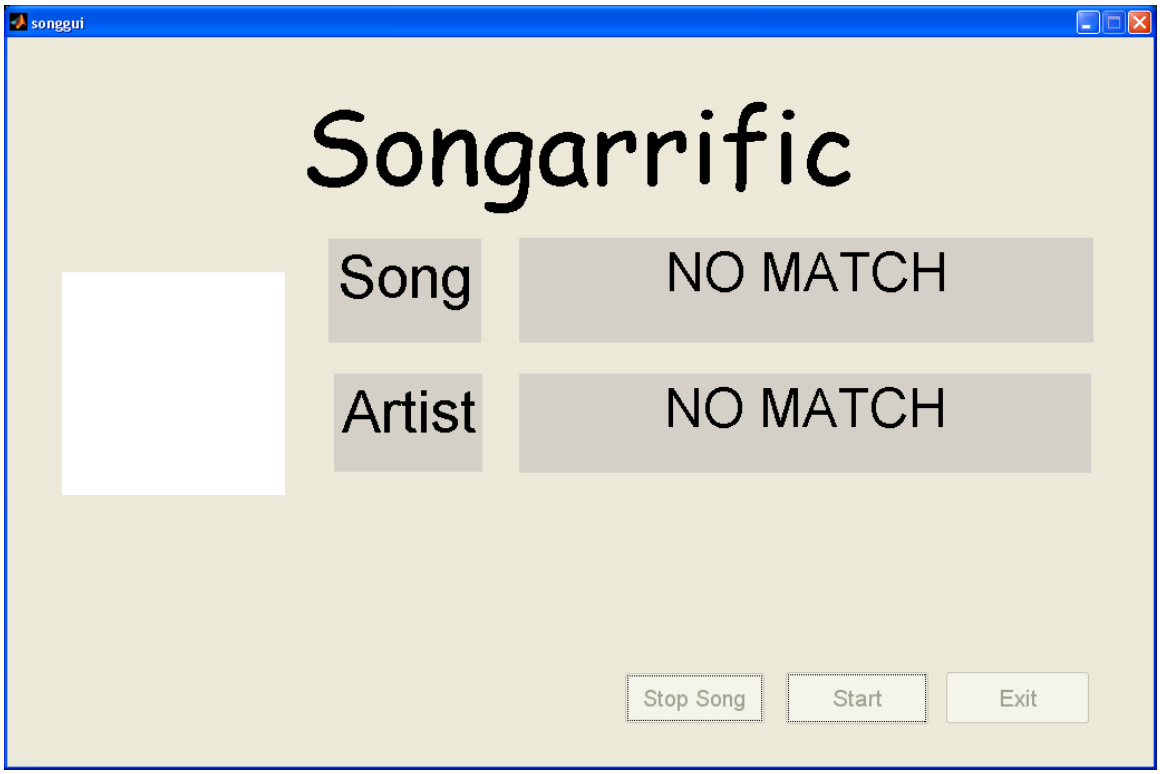


Figure 16

21. Semester Schedule

week 1	5-Oct	Online research. Understanding theory and algorithms. (Everyone)
week 2	12-Oct	Matlab implementation. Understanding theory and algorithms. (Everyone)
week 3	19-Oct	Matlab testing.(Everyone)
week 4	26-Oct	Create hashmap.(Marinos) Create song library.(Faraz) STFT and spectrogram C code.(Jason)
week 5	2-Nov	Set up MIC in on DSK.(Jason) Generate constellation map.(Marinos)
week 6	9-Nov	Finish all C code.(Jason, Marinos) Test algorithm on PC.(Faraz)
week 7	16-Nov	Implement C code to work on DSK.(Jason, Marinos)
week 8	23-Nov	Network Transfers.(Jason) Debugging.(Everyone) GUI.(Faraz)
week 9	30-Nov	Final presentation and demo. Documentation.(Everyone)

22. Future Improvements and Recommendations

We are very satisfied with the performance of our solution as well as the algorithm used. However, when testing the same recording with our solution and Shazam, the leading commercial music recognition software, we noticed that our solution had lower tolerance to noise. This is attributed to a number of reasons, most of which represent necessary restrictions in order to fit everything in the DSK. Below, we list a few recommendations and improvements for anyone attempting to do this project on a non-DSK environment or on a DSK with higher storage capacity.

- Use more landmarks. We had to limit the number of peaks per time frame and the number of landmarks per peak. Increasing these constants increases the number of landmarks and thus increases the probability that landmarks that survive heavy noise are found. This results in better recognition in louder noise.
- Increase the sampling rate for the songs stored in the database. We downsample our library songs to 8,000 Hz before adding them to the library. This deteriorates the quality of the recording significantly, and thus we obtain less landmarks. We are assuming Shazam uses CD-quality sampling rate, so it is able to extract more landmarks very accurately. Even if the recording is sampled at 8,000Hz, the landmarks accuracy is increased at the matching phase, resulting in better recognition in louder noise.
- Equalize the number of landmarks per song. When songs do not all have the same number of landmarks we run into cases where the more the landmarks, the easier it is to confuse a recording with low landmarks with a recording with a higher number of landmarks. Equalizing the number of landmarks per song is therefore essential. We would recommend re-processing quiet songs, if an adequate number of landmarks are not found.
- Increasing the size of the FFT. While increasing the sampling rate scales the time axis of the spectrogram, increasing the size of the FFT increases the frequency axis of the spectrogram. In our code, every time we double the Sampling Rate or double the FFT size we increase our hash table size by a factor of 2. This limited us to 20-bit hash values, and thus an FFT size of 512 and a Sampling Rate of 8,000. Given more storage,

increasing the size of the FFT would increase hash collisions and increase the accuracy of our algorithm.

The topic of song identification continues to interest us and we hope to continue this project, as there is currently no alternative to an open-source song identification scheme.

23. Glossary

Certain Match: After the recorded segment is matched with database, if the highest number of landmarks matched is twice or more as many as the next highest number of landmarks matched, then the song with the most matches is called a certain match.

Constellation Map: The set of constellation points of a song. According to theory [2], a segment of the song should have the exact same constellation map as the song, at that time interval. Thus a constellation map is a unique characteristic for each song.

Constellation Point: A point of high interest within the constellation map of the song, which is identified by two values <time,frequency>. While the intensity (energy) of the point in the spectrogram is used to identify the constellation point, it is not stored.

Database: The combined system of Hash Tables and Hash Elements to store the data obtained through processing the songs in our library at the training phase.

Fingerprint: A set of all Landmarks in a song. Essentially, since Fingerprints are derived from the constellation map, the same song should always have the same constellation map and therefore the same set of Landmarks, and thus the same Fingerprint every single time.

Guess: After the recorded segment is matched with database, if the highest number of landmarks matched is greater than 1.4 times but less than 2 times the next highest number of landmarks matched, then the song with the most matches is called a guess.

Hash Element: An element of the linked list hanging from each row of the Hash Table. It is of the form <t1,songID,next> where t1 is the time window at which the first point of the hashed landmark occurs in time, songID is the ID of the song to which the particular Hash Element belongs and next is the pointer to the next Hash Element in the Linked list, or NULL if there is no other element in the linked list. A Hash Element is 8 bytes long, after optimization: t1 and songID are of type short, while next is of type address.

Hash Function: A function which coverts a Landmark (<t1,f1,t2,f2>) to a 20-bit Hash Value. To do that, our Hash Function uses the 8 bits of the frequency of the first point in the landmark (f1) as bits 19-12 of the hash value. It uses 6 bits of the frequency difference between the two points

f2-f1 to as bits 11-6 of the hash value. Finally, it uses the 6 bits of the time difference t2-t1 as the last 6 bits of the hash value, bits 5-0.

Hash Table: An array of pointers to linked lists that have a hash value equal to the index of the array. The hash table needs to have an entry for each possible hash value, which forces its size to be equal to 2^(# of bits in the hash value). Elements with the same hash values (hash collisions) are appended to the linked list pointed to by each row in the Hash Table array. In our case, we are using 20-bit hash values, so our hash table has 2^20 entries (1M entries) each of which is of type pointer to linked list (address). Since addresses in the DSK are 4 bytes long, our hash table is 4MB long. Note that the hash table is simply an array of addresses, so it does not hold any data whatsoever. The data is in the linked lists that are hanging from each index in the array. The use of hash table in our program is extremely important, because it makes search O(1).

Hash Value: A unique 20-bit value for each landmark generated by the hash function.

Identification: The process of matching the landmarks of the recording to the landmarks of the songs in our database. This is done by generating the hash values for each of the landmarks of the recording and querying the hash table for song ID's at those specific hash values.

Landmark: A pair of constellation points, indentified as <t1, f1, t2, f2> where (t1,f1) and (t2,f2) are constellation points. We use landmarks to increase hashing entropy and to filter the results we would get if looking at constellation points, because there is lower probability that two neighboring points in different songs have the same frequencies and times. Landmarks are limited by three parameters:

Library: A folder with songs that have been preprocessed with Matlab (downsampled to 8,000Hz and converted to mono).

- maxpairsperpeak: the maximum number of pairs that a constellation point may have (if there exist any in its range)

No Match: No match refers to not being able to identify the recorded segment with any song in the database. If a song is not returned as a certain match or a guess, then it is a no match.

Peak: A casual term for constellation point, since constellation points are points of high intensity (peaks in the spectrogram)

Recording: Refers to the 15-second segment to be identified, which is recorded using the DSK and a microphone attached to it. Also referred to as recorded segment.

Scoring Function: A function that returns a number based on the distribution of matches across our songs. A 0 is returned for a 'No Match', the negative of the Song ID for a 'Guess' and the positive of the song ID for a 'Certain Match'.

- targetdf: the number of frequency bins below and above a point for identifying a pairing point. This identifies the frequency-dimension range of a landmark
- targetdt: the maximum look-ahead number of time frames for identifying a pairing point. This identifies the time-dimension range of a landmark

24. References

[1] JaapHaitsma, Antonius Kalker, “A Highly Robust Audio Fingerprinting System”, International Symposium on Music Information Retrieval (ISMIR) 2002, pp. 107-115.

- Information on how to create the hash table

[2] Avery Li, Chun Wang, “An Industrial-Strength Audio Search Algorithm”, Storage and retrieval methods and applications for multimedia 2004: (San Jose CA, 20-22 January 2004)

- The shazam algorithm the project based on

[3] Robust Landmark-Based Audio Fingerprinting http://labrosa.ee.columbia.edu/~dpwe/resources/matlab/fingerprint, Dan Ellis

- Matlab code similar to our algorithm, using fingerprints to match audio files

[4] Cheng Yang, “MACS: Music Audio Characteristic Sequence Indexing For Similarity Retrieval”, in IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, 2001

- Different algorithm, indexing raw-audio music files for content-based similarity retrieval. Talks about Hashing technique

[5] TI code: http://focus.ti.com/lit/ug/spru657b/spru657b.pdf.

-TI code for FFT radix 2. Single-precision floating-point radix-2 FFT with complex input

[6] http://simplynoise.com

- White noise generator