# THE NON-REMARKABLE-MUSICIAN'S REMARKABLE CHORD CLASSIFIER

Jonathan Rotner

Digital Signal Processing Final Project

ELEN 4810

Fall 2006

Professor Dan Ellis

## CONTENTS

## ABSTRACT

This program attempts to identify which chord has been played by an acoustic guitar. The chord is read in, analyzed via digital signal processing techniques, and then compared both to previously stored data and a table of musical frequencies. These two subsystems of analysis will hereby be referred to as the *static* and *dynamic* sections of the application. The static system was compiled with 178 different samples from 4 different guitars giving rise to a total of 35 distinct chords. Thy dynamic system works by comparing frequencies of the unknown chord with a table of known values. The hope is that a combination of these two subsystems will allow for an element of universality in instrument and a reasonable flexibility with regards to tuning.

## SIGNAL PROCESSING

The file ver1.m was used to analyze and compile the important frequencies of a chord so that the actual .wav files could be discarded. We will now go through a step by step analysis of the program with detailed explanation. The program begins by reading in the wav file. The request is that the filename be put in single quotes, so that Matlab recognizes it as a string.

```
chord = input('Input name of file for importing wrapped in single
quotes: ');
[x, Fs] = wavread(chord);
```

*x* is the 1D vector of the time signal, while *Fs* corresponds to the sampling rate, in samples/second. For a list of all variables, please refer to Appendix B. Next, we take the DFT (using the FFT function in Matlab). This converts the signal into the frequency domain, making a frequency analysis much easier.

```
NFFT = 2^nextpow2(l);
X = abs(fft(x,NFFT));
```

The function `nextpow2` is used to pad the data with enough zeros to make its length a power of two for the fastest possible FFT. The `abs` is used to convert the sequence to real numbers by taking its magnitude.

Converting the sequence to a dB scale allows for a more direct comparison since the signal can be now judged in terms of relative loss.

```
XdB = 20*log10(X);
```

Plotting the dB values of the sequence is commented out, but the lines of code are still included for analyzing and debugging purposes.

```
plot(f,XdB(1:NFFT/2));
```

The plot only contains half of the frequency because we are restricted by half of the Nyquist sampling rate. An example of a dB plot is illustrated below:

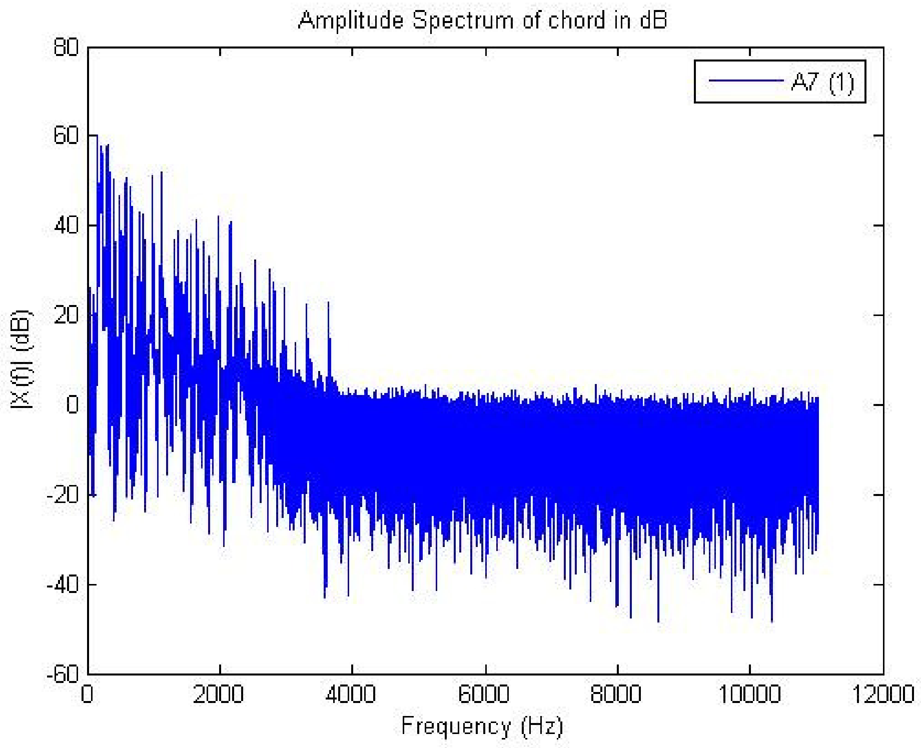

Figure 1: Amplitude Spectrum (dB) of A7 (1).wav

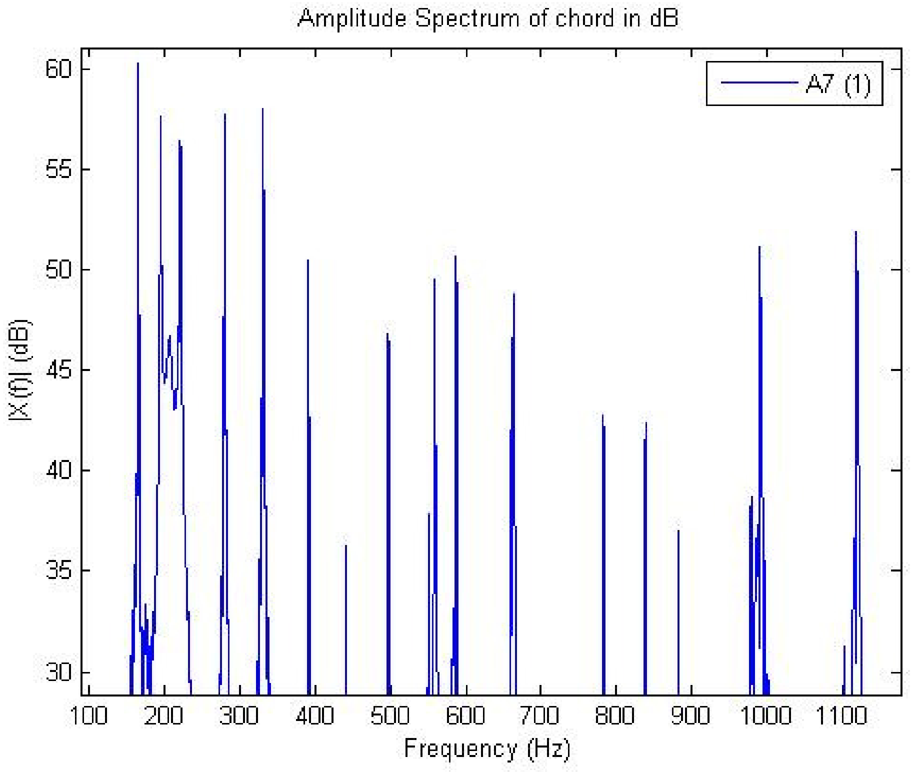

Figure 2: Zoomed in Amplitude Spectrum (dB) of A7 (1).wav

## SIGNAL ANALYSIS & STORING DATA

The next step is to look for the peak amplitudes that correspond to the frequencies of the notes found in the chord. First we search for the largest peak and its location in terms of sample number:

```
[val, idx] = max(XdB(floor(100*len/Fs):floor(len/2)));
idx=idx+100*len/Fs - 1; %idx is in units of length(XdB)/sample
idx=idx*Fs/len;
```

The search is restricted from 100 Hz to half the sequence (since it is an even function and we can disregard the second half). *idx* is normalized and thus converted to frequency (Hz).

In a for loop, we look through the samples in the first half of the sequence, starting at 100 Hz. We further restrict the search-set to within 20 dB of the maximum peak. Then we look for local maxima by comparing any given point to its previous and following one; if it is larger then both, we have a local max and store its amplitude and frequency in the arrays, *peaks* and *indices*, respectively.

```
j=1;
for i = floor(100*len/Fs):floor(len/2),
    if(XdB(i)>(val-20))
      if(XdB(i)>XdB(i-1) & XdB(i)>=XdB(i+1))
            peaks(j)=XdB(i);
            tmp=find(XdB==XdB(i));
            indices(j)=(tmp(1)-1)*Fs/len;
            j=j+1;
end; end; end;
```

Unfortunately, the peaks are not perfect; it is often the case that a peak is really composed of multiple jagged edges all in close proximity, as illustrated by Figure 3.

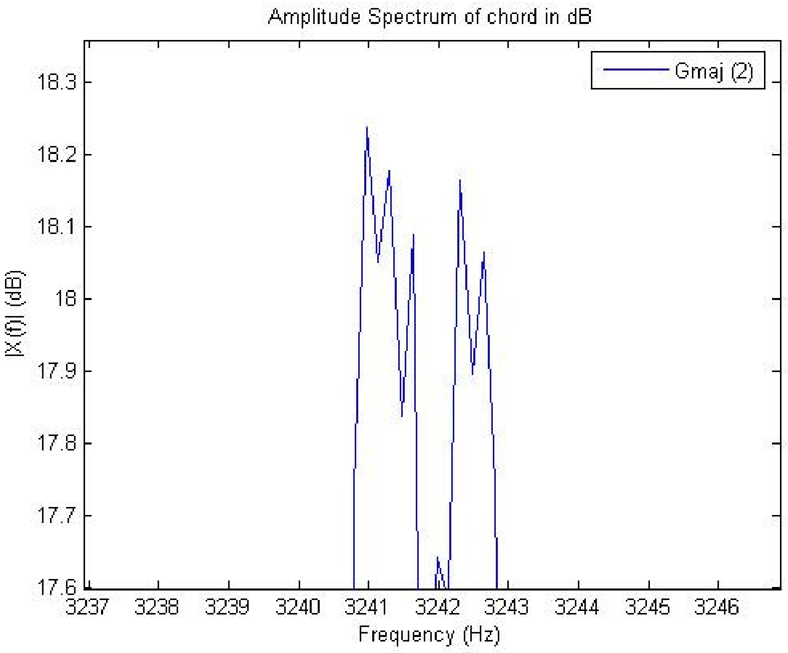

Figure 3: A Close Collection of Local Maxima

Thus, we must search through all the indices of local maxima and remove any duplicates within 5 Hz, while keeping the index with the maximum amplitude. The new amplitudes and frequencies are stored in *pks* and *inds*, respectively. 5 Hz was used as a comparison because on examination of a list of all musical frequencies, an average distance of 12-13 Hz separated two notes, thus a 5 Hz range on both sides allows for flexibility of out-of-tune guitars while not infringing on another note's boundaries.

```
j=1; i=1;
while(i<=length(peaks)),
    k=i+1;
    while(k<length(peaks) & (indices(k)-indices(i))<5)
        k=k+1;
    end
    s=max(peaks(i:k-1));
    pks(j)=s;
    tmp=find(XdB==s);
    inds(j)=(tmp(1)-1)*Fs/len;
    j=j+1;
    i=k;
end
```

In the final txt file we wish to store the frequencies at which the 5 maximum amplitudes are reached. The variable, *final_freq* will eventually store all local maxima remaining. But before that step, we have to remove any remaining data within 5 Hz. But wait, didn't the last section just do this? Yes, but upon recombination, there may still exist two notes within 5 Hz of each other that were not originally compared in the previous section, i.e. if 330 Hz is the target frequency, imagine we have the frequencies {325, 327, 329, 331}. 325 Hz would have been the initial basis of comparison (*k* in the code) and may have removed 325, 329, and kept 327 as the max frequency. But 331 would not be compared to the 325 (as it is 6 Hz away). However, 331 still refers to the same note, so 327 and 331 must be compared, as it is done in the next section of code. Also in this section of code, *final_freq* will store the resorted frequencies by amplitude.

```
final_freq=[idx];
[sorted,j]=sort(pks);
for i=1:length(sorted)-1,
    if(abs(inds(j(end-i)) - final_freq) > 5)
        final_freq(i+1)=inds(j(end-i));
    else
        tmp=find(not((abs(inds(j(end-i)) - final_freq)>5)));
        if(pks(j(end-i)>pks(tmp)))
            final_freq(i+1)=inds(j(end-i));
end; end; end
final_freq(final_freq==0)=[];
```

The last line removes any zeroes due to eliminating any frequency within 5 Hz.

The last portion of ver1.m exports the maximum 5 frequencies to ‘database.txt’ and then closes all files in case any were opened during debugging.

```
dlmwrite('Database.txt', final_freq(1:5), 'delimiter', '\t', '-
append');
ST = fclose('all');
```

## RUNNING THE PROGRAM & STATIC ANALYSIS

After storing the unknown chord as a .wav file, we run the program, *ver2.m* to try to identify it. The first part of the program is the same as *ver1*, except with different variable names. The first line is different: it reads in the .txt file of all stored data:

```
data = dlmread('database.txt');
```

Next, we extract all maximum peaks within 20 dB of the global maxima, and store it in the variable, *max_freq*. The top 5 are again stored in *final_freq*.

The first identifier portion of the code is the static program. We begin by looking for a match between the maximum 5 frequencies, sorted by amplitude, and the 5 frequency points stored on each line of the database. For a match, the program compares each of the 5 frequencies in turn and asks if they are within 5 Hz of each other. If a direct match is found, the row in *database.txt* is stored in the variable *hash* (a hash-table will later be used to convert row number to note). Also, *confidence*, a variable which will store how confident the result is, is set to 5 (like a 5-star rating system).

```
hash = 0; confidence = 0;
N = size(data);
for i=1:N(1)
    if(abs(final_freq - data(i,:)) < 5) %if each peak is within 5 Hz
        hash = i;
        confidence = 5;
        break;
end; end
```

If no match is found, do not dismay. Next, the program sorts the 5 maximum peaks by frequency, rather than amplitude, of the current unknown chord as well as *all* the stored data in *database.txt*. If a match is found now, *hash* takes on the appropriate value, and *confidence* is set to 4.

```
if(confidence == 0)
    sorted_data=sort(data,2);
    sorted_ff=sort(final_freq);
    for i=1:N(1)
        if(abs(sorted_ff - sorted_data(i,:)) < 5)
            hash = i;
            confidence = 4;
            break;
end; end; end
```

If we still have no luck, we try a second run through on the original, unsorted, stored data, this time just looking for 4 out of 5 matches. Only if the variable, *cnt*, has a value of 4, for 4 matches, does *hash* = *i* and *confidence* = 3.

```
if(confidence == 0)
    for i=1:N(1)
        cnt=0;
        for k=1:5
            if(abs(final_freq(k) - data(i,k)) < 5)
                cnt = cnt+1;
            end; if (cnt==4)
                hash = i;
                break
        end; end;
        if(cnt==4)
            confidence = 3;
            break
end; end; end
```

One last attempt if we are still unsuccessful. We look for 4/5 matches on the sorted-by-frequency path. *confidence* is now set to 2.

```
if(confidence == 0)
    for i=1:N(1)
        cnt=0;
        for k=1:5
            if(abs(sorted_ff(k) - sorted_data(i,k)) < 5)
                cnt = cnt+1;
            end; if (cnt==4)
                hash = i;
                break
        end; end;
        if(cnt==4)
            confidence = 2;
            break
end; end; end
```

Now the static program is complete and it is time to display the results. Using the file *hash_table.m* (see Appendix A), we retrieve which chord corresponds to the row stored in *hash*. Using the file *conf.m* in a similar manner, we retrieve a string holding our confidence level. The last line displays the results. If no matches were found, the user is informed.

```
if(hash~=0)
    name = hash_table(hash);
    rating = conf(confidence);
    sprintf('The chord you played was %s \n%s according to data
matching', name, rating)
else
    disp('No matches found')
end
```

## DYNAMIC ANALYSIS

Music and chords follow beautiful, mathematical patterns. The dynamic part of the program aims to capitalize on these relationships in major and minor chords. The patterns will be reviewed explicitly a little later on.

Before starting on the analysis of the peak frequencies, a little clean up is needed. We do not wish to check any frequencies over 1000 Hz because the 5 Hz range is no longer valid in that region. Also, since we are dealing with real-world data, we get some unwanted and unexpected data. There are always certain notes that blend in with the chord whether due to the body of the guitar, the sound box or inaccurate strumming. Upon review of the maximum peaks, I empirically deduced that the unwanted frequencies were often represented with at most 2 harmonics, while the notes that belonged had 3 to 4 harmonics. Mathematically speaking, if a note is at 440 Hz (A4), then its closest harmonics should appear at half and double that frequency (at 220 Hz is A3, and at 880 Hz is A5). Thus, to eliminate unwanted frequencies, the program sorts *max_freq*, (the new variable is *sorted_max*) and finds and keep only the notes that have 1 harmonic above and 1 harmonic below (within a range of 7 Hz and 5 Hz respectively). The array, *check*, stores the old, unsorted location from *max_freq*.

```
sorted_max=sort(max_freq);
check = zeros(1,length(sorted_max));
for i = 1:length(sorted_max)
    if(sorted_max(i)==1)
        continue;
    else
        h=find(abs(sorted_max-sorted_max(i)*2)<7);
        l=find(abs(sorted_max-sorted_max(i)/2)<5);
        if(h&l)
            check(i)=1;
end; end; end
sorted_max(find(~check))=0;
sorted_max(sorted_max==0)=[];
```

We limit the frequencies to one particular octave to make comparisons easier and eliminate any repeats that may remain.

```
if(find(sorted_max<320))
   sorted_max(find(sorted_max<320))=sorted_max(find(sorted_max<320))*2;
end
if(find(sorted_max>630))
   sorted_max(find(sorted_max>630))=sorted_max(find(sorted_max>630))/2;
```

```
end
%eliminating repeats
sorted_max=sort(sorted_max);
for i=1:length(sorted_max)-1,
    if(sorted_max(i+1)-sorted_max(i)>0 & sorted_max(i+1)-
sorted_max(i)<7)
        sorted_max(i)=0;
    end
end
sorted_max(sorted_max==0)=[];
```

A new array, *notes*, is initialized, which will store the actual notes of the chord as strings. We use the file, *freq_table.m,* to map frequencies to 5-character strings of the appropriate note. *freq_table.m* also returns the variable, *add*, which will later be used to determine how many half-steps away the individual notes are from each other. One last line establishes a long string array, *list*, which is hard-programmed to hold strings of just fewer than 2 octaves, starting with 'E'.

```
notes=[];
for i=1:length(sorted_max)
    [note,add]=freq_table(sorted_max(i));
    notes=[notes; note];
    adder(i)=add;
end

list = ['E    ';'F    ';'F#/Gb';'G    ';'G#/Ab';'A    ';'A#/Bb';'B
';'C    ';'C#/Db';'D    ';'D#/Eb';'E    ';'F    ';'F#/Gb';'G
';'G#/Ab';'A    ';'A#/Bb';'B    ';'C    ';'C#/Db';'D    '];
```

At this point all the recognized notes are stored and labeled by frequency and by string. It is time to match them to a particular pattern. If three notes are recognized, we can map the chord to a major or a minor progression. A major chord is defined by a Root-3rd-5th triad. Mathematically, given a starting note, the middle note should be 4 half-steps away, and the last member of the triad should be 7 half-steps away (a half-step is just the next note in *list*). A minor chord is defined by a R b3 5 triad (root, flatted-third and fifth). We set *list(adder(i))* as the root, and *i* cycles through all known notes, that way every combination of notes is tried. If the three notes fit into the major or minor pattern, the variable, *answer*, is set to 1 and a confidence string is outputted.

```
answer=0;
for i=1:size(notes,1)
    if(size(notes,1)==3)
        major=[list(adder(i),:);list(adder(i)+4,:);list(adder(i)+7,:)];
        if(sort(major,1) == sort(notes,1))
            sprintf('Your chord is %s %s \nwith a confidence rating of
                  VERY CONFIDENT according to frequency
                  matching',num2str(list(adder(i))),'major')
            answer=1;
            break;
        end
        %minor pattern: R b3 5
        minor=[list(adder(i),:);list(adder(i)+3,:);list(adder(i)+7,:)];
        if(sort(minor,1) == sort(notes,1))
            sprintf('Your chord is %s %s \nwith a confidence rating of
                  VERY CONFIDENT according to frequency
                  matching',num2str(list(adder(i))),'minor')
            answer=1;
            break;
        end
```

If four notes are recognized, we can map the chord to a major7 or a minor7 progression. A major7 progression follows a R 3 5 7 pattern while a minor7 progression follows a R b3 5 b7 pattern. This portion is still within the same for loop (i.e. the root note is cycled through all known notes).

```
    elseif(size(notes,1)==4)
        %major-7 pattern: R 3 5 b7
        major7=[list(adder(i),:);list(adder(i)+4,:);list(adder(i)+7,:);
            list(adder(i)+10,:)];
        if(sort(major7,1) == sort(notes,1))
            sprintf('Your chord is %s %s \nwith a confidence rating of
                  VERY CONFIDENT according to frequency
                  matching',num2str(list(adder(i))),'major7')
            answer=1;
            break;
        end
        %minor-7 pattern: R b3 5 b7
        minor7=[list(adder(i),:);list(adder(i)+3,:);list(adder(i)+7,:);
            list(adder(i)+10,:)];
        if(sort(minor7,1) == sort(notes,1))
            sprintf('Your chord is %s %s \nwith a confidence rating of
                  VERY CONFIDENT according to frequency
                  matching',num2str(list(adder(i))),'minor7')
            answer=1;
            break;
        end
```

If two notes are recognized, we can try and guess what that third note could be, utilizing the beauty of the musical patterns. We find at what index in *list* the two known notes are and take the difference; thus, we can conclude if we are missing the 3rd or 5th part of the triad. It is assumed that we are not missing the root, because the root note of the chord is often played on 2 to 3 strings of the guitar, and thus is most often represented. After taking the difference, we can pin the missing note to a model. If the difference = 3, we are most likely missing the 5th in a minor chord. If the difference = 4, we are most likely missing the 5th in a major chord. If the difference = 7, we are missing the 3rd. As the third may or may not be flatted, we do not know if this is a major or minor chord. Another assumption is that we do not have a major7 or minor7 chord since not capturing 2 notes is far less likely than not capturing 1 note. A different confidence level is also outputted.

```
elseif(size(notes,1)==2)
        tempor=find(list(:,1)==notes(1,1));
        tempor(find(list(tempor,2)==notes(1,2)));
        tempor2=find(list(:,1)==notes(2,1));
        tempor2(find(list(tempor2,2)==notes(1,2)));
        diff=tempor2(1)-tempor(1);

        if (diff==3)
            notes(3,:)=list(adder(1)+7,:);
            sprintf('The 2 notes that match are %s %s;\n The third is
most likely %s to complete the minor chord\n with a confidence level of
MOST LIKELY according to frequency matching', num2str(notes(1,:)),
num2str(notes(2,:)),num2str(notes(3,:)))
            answer=1;
        elseif (diff==4)
            notes(3,:)=list(adder(1)+7,:);
            sprintf('The 2 notes that match are %s %s;\n The third is
most likely %s to complete the major chord\n with a confidence level of
MOST LIKELY according to frequency matching', num2str(notes(1,:)),
num2str(notes(2,:)),num2str(notes(3,:)))
            answer=1;
        elseif (diff==7)
            notes(3,:)=list(adder(1)+4,:); %maj
            notes(4,:)=list(adder(1)+3,:); %min
            sprintf('The 2 notes that match are\n %s %s\n The third is
most likely\n %s to complete the major chord or \n %s to complete the
minor chord\n with a confidence level of LIKELY according to frequency
matching', num2str(notes(1,:)), num2str(notes(2,:)),
num2str(notes(3,:)), num2str(notes(4,:)))
            answer=1;
        end
        break;
```

If the number of notes we recognize is less than 2 or more than 4, an error message is displayed. Also, if no pattern was found, the user is informed that the dynamic program did not succeed. The last 2 lines of code displays the dynamic notes found and closes all files.

```
    else
        disp('error with length of notes array')
        break;
end; end

if(answer~=1)
    disp('No dynamic match')
end
notes
ST = fclose('all');
```

## APPENDIX A: INCLUDED FILES

Attached, please find the following files:

conf.m

freq_table.m

hash_table.m

ver1.m

ver2.m

database.txt

as well as the following 3 sample files:

1Cmaj.wav

1D7.wav

1F#m.wav

## APPENDIX B: LIST OF VARIABLES

| Variable | Function | M-file |
|---|---|---|
| *add* | integer that stores index of a note in *list* | ver2, freq_table |
| *adder* | array of all *add*s | ver2 |
| *answer* | if 0, then no dynamic match; if 1, dynamic match | ver2 |
| *check* | if 1, then note has harmonic an octave higher and lower, else 0 | ver2 |
| *chord* | input string of filename | ver1 |
| *confidence* | integer that stores static confidence rating. Score is out of 5 | ver2 |
| *data* | database of stored chord data | ver2 |
| *f* | x-dim for plot of *X* | ver1, ver2 |
| *final_freq* | the frequencies of all maximum peaks within 20 dB | ver1 |
| *final_freq* | the frequencies of the top 5 maximum peaks | ver2 |
| *Fs* | sampling rate (samples/sec) | ver1, ver2 |
| *hash* | row in *data* that matches *final_freq*; argument to *hash_table.m* | ver2 |
| *idx* | frequency at which global maximum resides | ver1, ver2 |
| *indices* | array of frequencies of ALL local maxima within 20 dB | ver1, ver2 |
| *inds* | indices with close frequencies (within 5 Hz) removed | ver1, ver2 |
| *j* | index mapping between sorted and pks | ver1, ver2 |
| *l* | number of points in sample | ver1 |
| *len* | length of *XdB* | ver1, ver2 |
| *list* | 23x5 char array with just less than 2 octaves of notes | ver2 |
| *major* | 3x5 char array of major pattern with a given root note | ver2 |
| *match* | returns string holding name of chord as stored by static program | hash_table |
| *max_freq* | the frequencies of all maximum peaks within 20 dB; *final_freq* in *ver1.m* | ver2 |
| *minor* | 3x5 char array of minor pattern with a given root note | ver2 |
| *N* | amount of rows stored in *data* | ver2 |
| *name* | string holding name of chord as stored by static program | ver2 |
| *NFFT* | Next power of 2 from length of sample | ver1, ver2 |
| *note* | name of note played as mapped by frequency value | ver2, freq_table |
| *notes* | char array holding the name of all notes found in dynamic program | ver2 |
| *peaks* | array of amplitudes of ALL local maxima, sorted by frequency | ver1, ver2 |
| *pks* | amplitudes with close frequencies (within 5 Hz) removed | ver1, ver2 |
| *rating* | string holding confidence value for static program | ver2, conf |
| *sorted* | *pks* sorted from lowest to highest amplitude | ver1 |
| *sorted* | sorted version of *pks* | ver2 |
| *sorted_data* | database sorted by frequency, rather than by amplitude | ver2 |
| *sorted_max* | frequencies of all dynamically found notes within one octave, sorted by frequency | ver2 |
| *unknown* | input string of filename; *chord* in *ver1.m* | ver2 |
| *val* | value of global maxima | ver1, ver2 |
| *x* | original sequence | ver1, ver2 |
| *X* | the FFT sequence | ver1, ver2 |
| *XdB* | the FFT sequence in dB | ver1, ver2 |