Midterm Research Project Report

Youtube link: https://www.youtube.com/watch?v=tCTHA47poxo

Ethan Mandel

Mandel.e@northeastern.edu

Submit date: 4/1/2021

Due Date: 4/2/2021

1.0 Project Description

The goal of the project was to take audio into the DE1-SoC using the microphone port, generate a Discrete Fourier Transform of the input signal, analyze the DFT for local maxima, and determine a musical chord from the maxima. With the musical chord determined by the algorithm, the 7-segment display displayed the chord name to the user and chords relative to the identified chord were played using the keypad and speakers.

2.0 Hardware Used

The audio-in port was used to capture the sound from either a piano or online piano that plays a chord that was identified by the algorithm. The buttons were used to begin capturing audio or to stop the program entirely. The keypad was used once the chord was identified to pick other chords to play that were related to the identified chord. Once a selection was made on the keypad, the speakers played the selected chord for .5 seconds.

3.0 Algorithm Used

3.1 The Name of the Algorithm/Method

Algorithm 1: Discrete Fourier Transform

Algorithm 2: Note Identification

Algorithm 3: Chord Identification

3.2 The IEEE/ASEE Paper where the Algorithm/Method is located

Algorithm 1: Soo-Chang Pei, Min-Hung Yeh and Tzyy-Liang Luo, "Fractional Fourier series expansion for finite signals and dual extension to discrete-time fractional Fourier transform," in IEEE Transactions on Signal Processing, vol. 47, no. 10, pp. 2883-2888, Oct. 1999, doi: 10.1109/78.790671.

https://ieeexplore-ieee-org.ezproxy.neu.edu/document/790671

Algorithm 3: V. Zenz and A. Rauber, "Automatic Chord Detection Incorporating Beat and Key Detection," 2007 IEEE International Conference on Signal Processing and Communications, Dubai, United Arab Emirates, 2007, pp. 1175-1178, doi: 10.1109/ICSPC.2007.4728534.

https://ieeexplore-ieee-org.ezproxy.neu.edu/document/4728534

Algorithm 3: B. Lin and T. Yeh, "Automatic Chord Arrangement with Key Detection for Monophonic Music," 2017 International Conference on Soft Computing, Intelligent System and Information Technology (ICSIIT), Denpasar, Indonesia, 2017, pp. 21-25, doi: 10.1109/ICSIIT.2017.29.

https://ieeexplore-ieee-org.ezproxy.neu.edu/document/8262537

3.3 Explanation of the Algorithm

Algorithm 1 (DFT):

The Discrete Fourier Transform takes an input signal which is expressed in the time domain and displays the signal in the frequency domain. This allows for easy frequency spectrum analysis as signal amplitude is compared to frequency in a DFT plot as opposed to time in a standard signal amplitude graph. By creating a complex sinewave at each point in time and then taking the dot product of the signal and the sine wave a Fourier coefficient (amplitude) is found for each frequency value. The term FFT (Fast Fourier Transform) is often used synonymously with DFT but I have chosen to express it as a DFT for the sake of clarity. The DFT is highly effective at clearly expressing the frequency spectrum of a signal, but it is a very processor demanding function transformation that requires many individual calculations. The calculation is extremely processor demanding. It takes a very noticeable amount of time for a 2 second signal to be fully transformed. The DFT reduction factor implemented reduces the number of Fourier coefficients processed drastically reducing processing time.

Algorithm 2 (Note and Chord Identification):

The algorithm to determine a note from a frequency is relatively simple. Each note corresponds with a specific set of frequencies that are integer multiples of each other. Each note of the same type, for example “A”, is separated musically by an octave. An octave in frequency terms implies the doubling of the initial frequency. For example, A4 = 440 Hz and A5 = 880 Hz. Each semitone in an octave is 2^x/12 * reference frequency away from the reference frequency. For example, B4 is 2 semitones away from A4 so it has a frequency of 440* (2^2/12). With an instrument that is perfectly in tune while also playing in perfect temperature, acoustic, and humidity conditions these relations hold perfectly true. Assuming imperfect conditions I factored in some error. This error was approximately half of the distance between adjacent semitones. The distance between semitones grows as frequency increases so the error does as well. If a note is within the low and high bounds of the error, for all intents and purposes it is that note.

Algorithm 3: (Chord Identification)

With each frequency assigned a note value, a chord can be discerned from a set of frequencies. For this project I chose to only focus on major and minor triads as the chords investigated for a total of 24 possible chords. For a list notes the algorithm I designed determines how many times a specific note, regardless of octave, appears in the list. More appearances in the list are causally linked to greater probability of that note being in the played chord. Initial position in the list is of secondary importance as lower frequencies, which are found earlier in the list, are always more likely to be the first note of the chord as the structure of chords in western music revolves around picking a note and building up from that note. Once the 3 notes that either have the greatest number of appearances and secondarily appear earlier in the list are identified, these three notes are compared to a list of all major and minor chords. If the 3 notes are the same 3 notes as a chord than the chord is correctly identified, and that result is output to the user. The algorithm was designed to overcome the problem of overtones interfering with note identification. Whenever a string is plucked or hit, a frequency is generated but other frequencies that are integer multiples of that frequency are generated as well. The priority of note appearance part in the algorithm was included to use the overtones as confirmation of a note's appearance as the majority of the overtones a string produces are just higher octaves of the same note.

3.4 Implementation of the Algorithm

Algorithm 1:

In the code below the frequency spectrum of the signal is being created by getting the Fourier coefficients from the dot product of a complex sine wave and the original signal. These coefficients are then being squared so there are only positive frequency values present as negative frequency values are not important in this context. The method for generating the Fourier coefficients was inspired by the MATLAB program created by Ron Fredericks [1].

```
double fs = 8000;//Sampling frequency of De1soc
double L = 10000; //RawDatasize
double dft_loop_reduction = DFTreduction; //set to 1 for no reduction
double f[RawDatasize/2+1];

for(int i=0; i<=RawDatasize/2; i++){
    f[i] = fs* (i/L);//Makes frequency array half of the size of raw audio
file as negative frequencies are discared
}

complex<double> fourTime[RawDatasize];
double inttodouble =0; //Used to prevent errors in comparing ints to doubles
for(int i=0; i<=RawDatasize-1; i++){
    inttodouble=i;
    fourTime[i] = (inttodouble/L); //Fills fourTime with 10000 divisions of
audio data size
    //cout << fourTime[i] << endl;
}
inttodouble=0;

complex<double> fCoefs[RawDatasize];
complex<double> csw[RawDatasize];
complex<double> multipliedfourTime[RawDatasize];
```

```
complex<double> signaltimescsw[RawDatasize];

for (int i=1; i<=RawDatasize; i= i +dft_loop_reduction){//Calculates fourier coeficients
    inttodouble=i;
    scalarProductMat(fourTime,(-1.0*Imaginarynumber*2.0*pi*(inttodouble-1)),RawDatasize,multipliedfourTime);
    for(int j=0; j<RawDatasize;j++){
        csw[j]= multipliedfourTime[j];
    }
    for(int j=0; j<RawDatasize;j++){
        csw[j] = pow(e,csw[j]);//Forms complex sine wave
    }
    for(int j=0; j<RawDatasize;j++){
        signaltimescsw[j] = csw[j]*RawAudioData[j];
    }
    fCoefs[i] = SumElementsInArray(signaltimescsw,RawDatasize); //Fourier coefficients are the sum of complex sinewaves
}

double DFT[RawDatasize];

for(int i=0;i<=RawDatasize;i++){
    DFT[i]= pow(abs(fCoefs[i]/L),2); //Takes magnitude of frequency specturm so there are no negative values
}
```

After the DFT was created it needed to be analyzed for local maxima. This was accomplished by finding the max value for every 50 samples in the DFT array and then creating an array containing all the maximums and another array containing their indices. From this I determined at which frequency a max value occurred and I chose to use the top 6 frequencies as all frequencies beyond that proved to be of little value. These 6 frequencies were then sorted from least to greatest.

```
for (int i =0; i<=RawDatasize/DFTsegment-Highfrequencyvalues; i++){//Finds maximum value for every 50 data points
    indexAtMaxY=0;
    for(int j=0; j<DFTsegment; j++){ //Checks each 50 point segment and marks max and the index the max occurs at
        Psegment[j] = DFT[(i*DFTsegment)+j+1];
        indexAtMaxY= indexatmax(Psegment,DFTsegment);
        indexAtMaxY = indexAtMaxY + DFTsegment*i +1;
        xValueAtMaxYValue = DFT[indexAtMaxY];
        ArrayofMaximums[i] = xValueAtMaxYValue; //Array that holds all maximum values
        ArrayofMaxIndicies[i] = indexAtMaxY; // Array that holds the indexes of all maximum values
    }
}
int FrequencyIndexAtMax =0;
```

```
for (int i =0; i<NumberofNotesCompared; i++){ //Takes the number of maximums
equivalent to value of NumberofNotesCompared
    indexAtMaxY= indexatmax(ArrayofMaximums,RawDatasize/DFTsegment-
Highfrequencyvalues+1);
    FrequencyIndexAtMax =ArrayofMaxIndicies[indexAtMaxY];
    ArrayofMaximums[indexAtMaxY] =0;
    if(f[FrequencyIndexAtMax] < 2000 && f[FrequencyIndexAtMax] >1){ //If
frequency is in accpetable range add it to chord frequencies
        ChordFrequencies[i] =f[FrequencyIndexAtMax];
    }else{
        ChordFrequencies[i] =0;
    }
}
int arraysortsize = sizeof(ChordFrequencies)/sizeof(ChordFrequencies[0]);
sort(ChordFrequencies, ChordFrequencies +arraysortsize); //Sorts chord
frequencies from least to greatest
//This sort is done because lower frequencies are given priority as the
lowest frequency is always the root of the chord
```

These 6 frequencies were all that were required to determine the chord in the first iteration of my design so from here algorithm 2 was implemented.

Algorithm 2:

The frequency values of each note were first defined using a formula similar to what I mentioned above with the addition of allowing for each note to be present across 7 octaves.

```
for (int i=0; i<=6; i++){
    inttodouble = i;
    note = 440*pow(2,(((1+(12*inttodouble))-49)/12));//Formula to determine
value of A
    Afrequencies[i]= note;
    note = 440*pow(2,(((2+(12*inttodouble))-49)/12));
    BflatAsharpfrequencies[i]= note;
    note = 440*pow(2,(((3+(12*inttodouble))-49)/12));
    Bfrequencies[i]= note;
    note = 440*pow(2,(((4+(12*inttodouble))-49)/12));
    Cfrequencies[i]= note;
    note = 440*pow(2,(((5+(12*inttodouble))-49)/12));
    DflatCsharpfrequencies[i]= note;
    note = 440*pow(2,(((6+(12*inttodouble))-49)/12));
    Dfrequencies[i]= note;
    note = 440*pow(2,(((7+(12*inttodouble))-49)/12));
    EflatDsharpfrequencies[i]= note;
    note = 440*pow(2,(((8+(12*inttodouble))-49)/12));
    Efrequencies[i]= note;
    note = 440*pow(2,(((9+(12*inttodouble))-49)/12));
    Ffrequencies[i]= note;
```

```
    note = 440*pow(2,(((10+(12*inttodouble))-49)/12));
    GflatFsharpfrequencies[i]= note;
    note = 440*pow(2,(((11+(12*inttodouble))-49)/12));
    Gfrequencies[i]= note;
    note = 440*pow(2,(((12+(12*inttodouble))-49)/12));
    AflatGsharpfrequencies[i]= note;
}
```

Note error was then established as mentioned above to ensure that notes within an acceptable range were correctly identified.

```
//Acceptable note error increases with frequnecy as higher notes are farther
apart frequency wise
NoteError =0;
if (ChordFrequencies[i] > 0 && ChordFrequencies[i]< 63) { //Acceptable error
of pitch at each octave in Hz
    NoteError = 2;
}else if (ChordFrequencies[i] > 63 && ChordFrequencies[i]< 127) {
    NoteError = 4;
}else if (ChordFrequencies[i] > 127 && ChordFrequencies[i]< 255) {
    NoteError = 8;
}else if (ChordFrequencies[i] > 255 && ChordFrequencies[i]< 511) {
    NoteError = 16;
}else if (ChordFrequencies[i] > 511 && ChordFrequencies[i]< 1023) {
    NoteError = 32;
}else if (ChordFrequencies[i] > 1023 && ChordFrequencies[i]< 2047) {
    NoteError = 64;
}else if (ChordFrequencies[i] > 2047 && ChordFrequencies[i]< 10000) {
    NoteError = 128;
}
```

Each frequency identified by the DFT algorithm was then put through the identification process. Below I'll display just what occurred for the note "A" so I can explain the identification process in more detail.

```
if ((Afrequencies[j]-NoteError <ChordFrequencies[i] &&
Afrequencies[j]+NoteError >ChordFrequencies[i]) && (abs(Afrequencies[j]-
ChordFrequencies[i])< abs(BflatAsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Afrequencies[j]-ChordFrequencies[i])< abs(AflatGsharpfrequencies[j]-
ChordFrequencies[i]))){
    ChordNotes[i]="A";
}
```

The identified frequency (ChordFrequencies[i]) must lie between the frequency value of "A" for octave 1 – error for octave 1 as well as "A" for octave 1 + error for octave 1. The absolute value of the difference between the frequency of the semitone above – ChordFrequencies[i] and the semitone below must also be less than the absolute value of A frequency- ChordFrequencies[i]. This ensures correct identification of the note.

With each note identified algorithm 3 was put into use.

Algorithm 3 begins by taking the identified notes and ordering them by appearance. Notes are first ranked by appearance and secondarily by frequency for example if A = 440 is in the list and B = 494 is also in the list and 3 instance of A occur as well as 3 instances of B occur the list would read A,B as the lowest A is lower than the lowest B. The part of the sort by appearance formula that just ranks the list by appearance was created by GeeksforGeeks[2]. This is shown below.

```
int ChordNoteNumberSize = sizeof(ChordNotes) / sizeof(ChordNotes[0]);

bool check[ChordNoteNumberSize];
int counterarray[ChordNoteNumberSize];
for(int i= 0; i <ChordNoteNumberSize; i++){
    counterarray[i] =0;
}

string UniqueChordNotes[ChordNoteNumberSize];
for(int i= 0; i <ChordNoteNumberSize; i++){//Makes an empty array
    UniqueChordNotes[i] = " ? ";
}

for(int i=0;i<ChordNoteNumberSize;i++){
    check[i] = 0;
}
for(int i=0; i<ChordNoteNumberSize; i++){//Determines the number of times
each note appears
    if(check[i]== 1){
        continue;
    }
    int count = 1;

    for(int j = i+1; j<ChordNoteNumberSize; j++){
        if (ChordNotes[i].compare(ChordNotes[j]) == 0){
            check[j] = 1;
            count++;
        }
    }
    counterarray[i] = count; //Count of each note is filled into this array
    UniqueChordNotes[i] = ChordNotes[i];

    cout<<ChordNotes[i]<<" appears: " << count << " time(s)"<< endl;
}
cout << endl << endl;
```

```
int comparisoncounterarray[ChordNoteNumberSize];
for(int i =0; i <ChordNoteNumberSize; i++){
    comparisoncounterarray[i] = counterarray[i];
}
sort(counterarray, counterarray + ChordNoteNumberSize, greater<int>());
//Count array is sorted from least to greates

string SortedUniqueChordNotes[ChordNoteNumberSize];

for(int i=0; i<ChordNoteNumberSize; i++){ //Using the sorted count array the
note that lines up with the number of appearence is brought to the top
    for (int j=0; j<ChordNoteNumberSize; j++){
        if(counterarray[i] == comparisoncounterarray[j] &&
UniqueChordNotes[j].compare(" ? ") != 0 && UniqueChordNotes[j].compare("NaN")
!= 0 && counterarray[i] !=0){
            comparisoncounterarray[j] =0;
            SortedUniqueChordNotes[i] = UniqueChordNotes[j];
            break;
        }
    }

}
```

From here all different types of chords were defined as well as being split into their component notes.

```
//Definition of all chords
    string ListofChords[24] = {"A major", "A minor", "Bflat/Asharp major",
"Bflat/Asharp minor", "B major", "B minor","C major", "C minor","Dflat/Csharp
major", "Dflat/Csharp minor","D major", "D minor","Eflat/Dsharp major",
"Eflat/Dsharp minor","E major", "E minor","F major", "F minor","Gflat/Fsharp
major", "Gflat/Fsharp minor","G major", "G minor","Aflat/Gsharp major",
"Aflat/Gsharp minor"};
    string Amajorcomponents[3] = {"A","Dflat/Csharp","E"};
    string Aminorcomponents[3] = {"A", "C", "E"};
    string BflatAsharpmajorcomponents[3] = {"Bflat/Asharp", "D", "F"};
    string BflatAsharpminorcomponents[3] = {"Bflat/Asharp", "Dflat/Csharp",
"F"};
    string Bmajorcomponents[3] = {"B", "Eflat/Dsharp", "Gflat/Fsharp"};
    string Bminorcomponents[3] = {"B", "D", "Gflat/Fsharp"};
    string Cmajorcomponents[3] = {"C", "E", "G"};
    string Cminorcomponents[3] = {"C", "Eflat/Dsharp", "G"};
    string DflatCsharpmajorcomponents[3] = {"Dflat/Csharp", "F",
"Aflat/Gsharp"};
    string DflatCsharpminorcomponents[3] = {"Dflat/Csharp", "E",
"Aflat/Gsharp"};
    string Dmajorcomponents[3] = {"D", "Gflat/Fsharp", "A"};
    string Dminorcomponents[3] = {"D", "F", "A"};
    string EflatDsharpmajorcomponents[3] = {"Eflat/Dsharp", "G",
"Bflat/Asharp"};
    string EflatDsharpminorcomponents[3] = {"Eflat/Dsharp", "Gflat/Fsharp",
```

```
"Bflat/Asharp"};
    string Emajorcomponents[3] = {"E", "Aflat/Gsharp", "B"};
    string Eminorcomponents[3] = {"E", "G", "B"};
    string Fmajorcomponents[3] = {"F", "A", "C"};
    string Fminorcomponents[3] = {"F", "Aflat/Gsharp", "C"};
    string GflatFsharpmajorcomponents[3] = {"Gflat/Fsharp", "Bflat/Asharp",
"Dflat/Csharp"};
    string GflatFsharpminorcomponents[3] = {"Gflat/Fsharp", "A",
"Dflat/Csharp"};
    string Gmajorcomponents[3] = {"G", "B", "D"};
    string Gminorcomponents[3] = {"G", "Bflat/Asharp", "D"};
    string AflatGsharpmajorcomponents[3] = {"Aflat/Gsharp", "C",
"Eflat/Dsharp"};
    string AflatGsharpminorcomponents[3] = {"Aflat/Gsharp", "B",
"Eflat/Dsharp"};
```

A 2D array was then created with each column containing all 3 notes for each chord. The 3 notes identified to be the most likely to occur in the chord were compared against each column of the 2D array. If all 3 components aligned, then the chord was identified and sent as an output from the function.

```
int isthisthecorrectchord=0;
string CorrectlyIdentifiedChord ="Unidentified Chord";
for(int i=0; i<24; i++){
    isthisthecorrectchord= 0;
    for(int j=0; j<3; j++){ //Checks comparison chord with all possible
chords
        if(comparisonchord[j]== ChordComponents[i][0] || comparisonchord[j]==
ChordComponents[i][1] || comparisonchord[j]== ChordComponents[i][2]){
            isthisthecorrectchord =isthisthecorrectchord+1;
        }else{
            isthisthecorrectchord= 0;
        }
    }
    if(isthisthecorrectchord ==3){//If all three notes align with a chord
then the chord is identified
        CorrectlyIdentifiedChord=ListofChords[i];
    }
}

cout << "Identified Chord" << endl;
return CorrectlyIdentifiedChord;
```

4.0 Improvement Evaluation

4.1 Goal/Target of the Improvement

After successfully identifying a chord for the first time, my goal was to improve the accuracy of chord identification process. My initial design for the chord identification portion of the program was to simply take a list of the 6 frequencies with the largest power values and then take the first 3 unique notes in that list as the notes that make up the chord.

4.2 Current State of your Improvement

To improve the chord identification accuracy, two processes were implemented. The first process was the arrangement of the notes into frequency order before analysis. The second process was the prioritization of note appearance. Note appearance in this case implies how many times a specific note appeared regardless of frequency. This change ensured that noise in the signal was more likely to be ignored as the overtones of the chord tones would appear multiple times and take priority.

4.3 Method of Improvement

Improvement regarding chord identification accuracy can be evaluated simply through percent accuracy at various DFT reduction factor values. 24 text files containing the audio data for each major and minor chord were created and run through the program with all DFT reduction factors ranging from 1 to 100. The accuracy of the both the initial and improved program was tested in this manner.

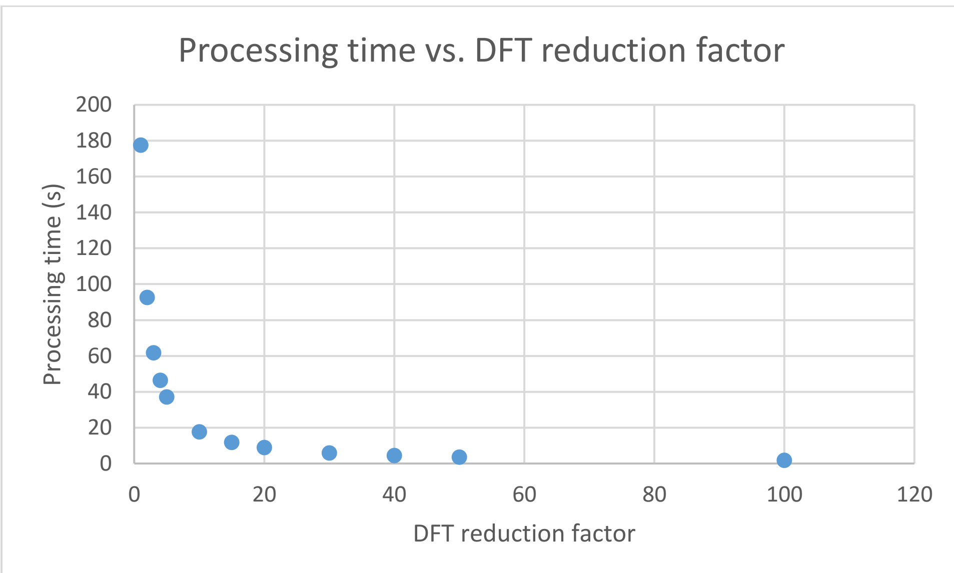
4.4 Result of Improvement

The reducing the processing time required to generate a DFT was one of the key goals of this project. Below is a table displaying the time to generate a DFT based on the reduction factor. Although the processing times for factors above 40 were under 5 seconds too much of the signal was destroyed in this simplification that it was near impossible to correctly identify frequencies from so few data points. From many observations of many trials the best balance between identification accuracy occurred at a DFT reduction factor of 10. Improvements to the project did not involve altering DFT calculation speed beyond this.

Table 1: Processing time vs. DFT reduction factor

DFT Reduction Factor	Processing time (s)
1 (no reduction)	177.48
2	92.63
3	61.72
4	46.35
5	37.07
10	17.68
15	11.8
20	8.85
30	5.91
40	4.41
50	3.53
100	1.76

Figure 1: Processing time vs. DFT reduction factor



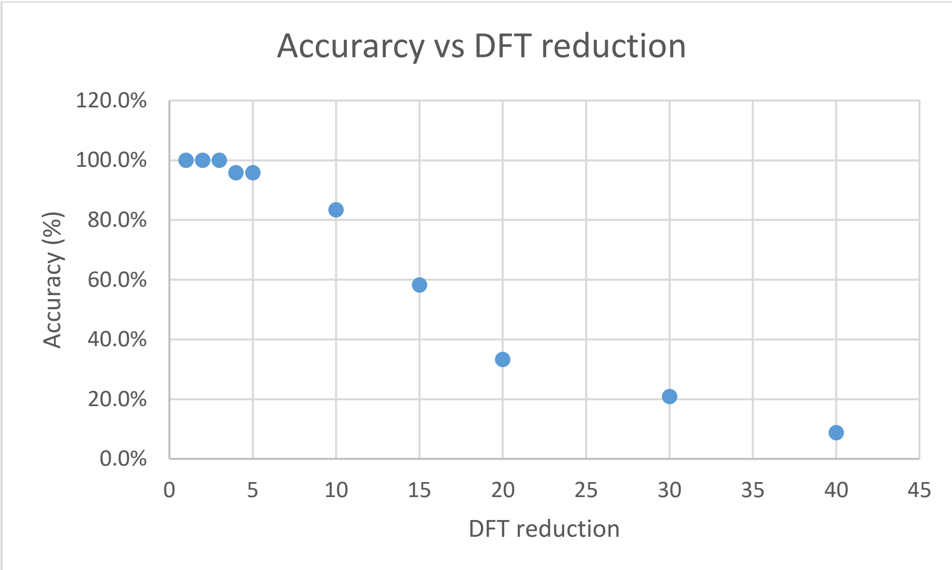
Proper note identification was also a focus of this project. As notes have a direct relationship with frequency, note identification was accurate 100% of the time as all notes falling within the acceptable bounds were identified to be that note. The reason for this perfect accuracy is due to the fact that notes are defined by their frequencies and assigning note values is essentially just giving a name to the frequency. If a C key on a piano is played but it happens to be very sharp due to the weather, acoustics, poor tuning, or other factors the note played may have been intended to be a C but it is a C sharp. This is an example of how frequency is the determinant of the note and not the key pressed.

Proper chord identification was the other key focus of the project. All 24 chords in the fourth octave were fed into program at key DFT reduction factors to determine how DFT reduction affects the accuracy of identifying the chord. To determine how accurate the algorithm is at determining a chord without conflict from DFT reduction the trials with a DFT reduction factor of 1 was used. The tables below present accuracy at various levels of DFT reduction.

Table 2: Percent correctly identified chords vs DFT reduction factor

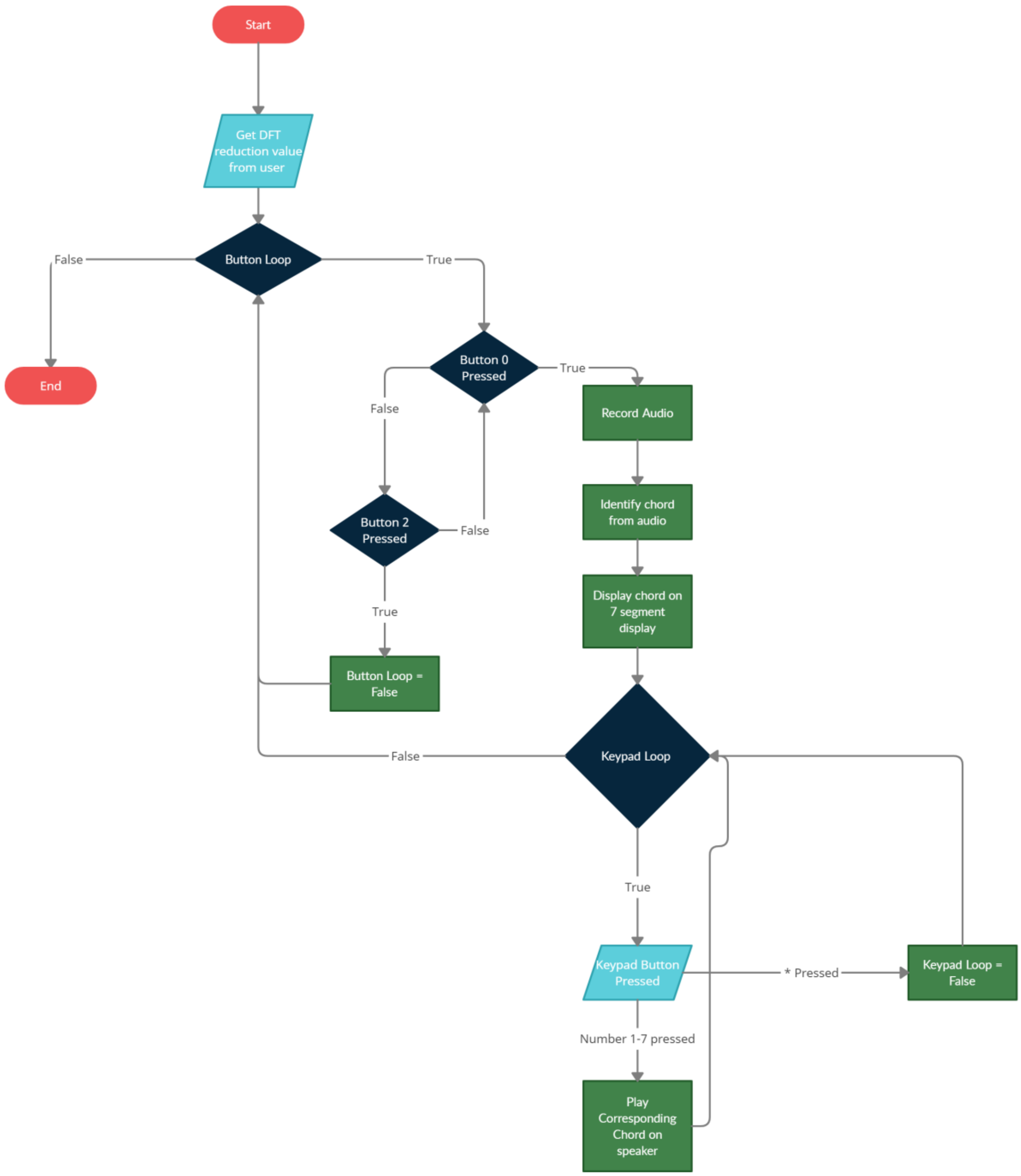
DFT Reduction Factor	Percent correctly identified
1	100.0%
2	100.0%
3	100.0%
4	95.8%
5	95.8%
10	83.3%
15	58.2%
20	33.3%
30	20.8%
40	8.7%
50	<1%
100	<1%

Figure 2: Chord identification accuracy vs. DFT reduction factor

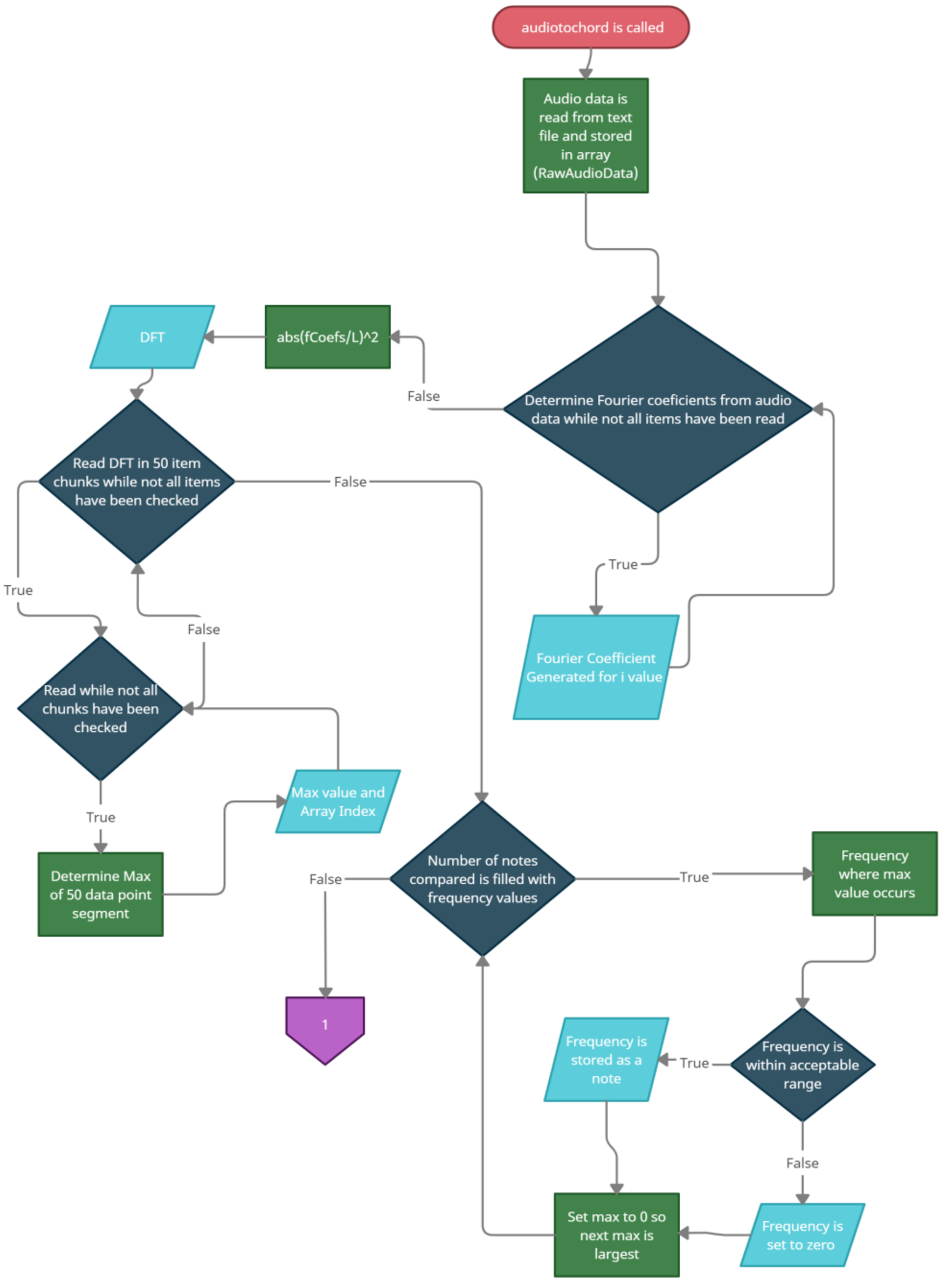


5.0 C++ Code Explanation

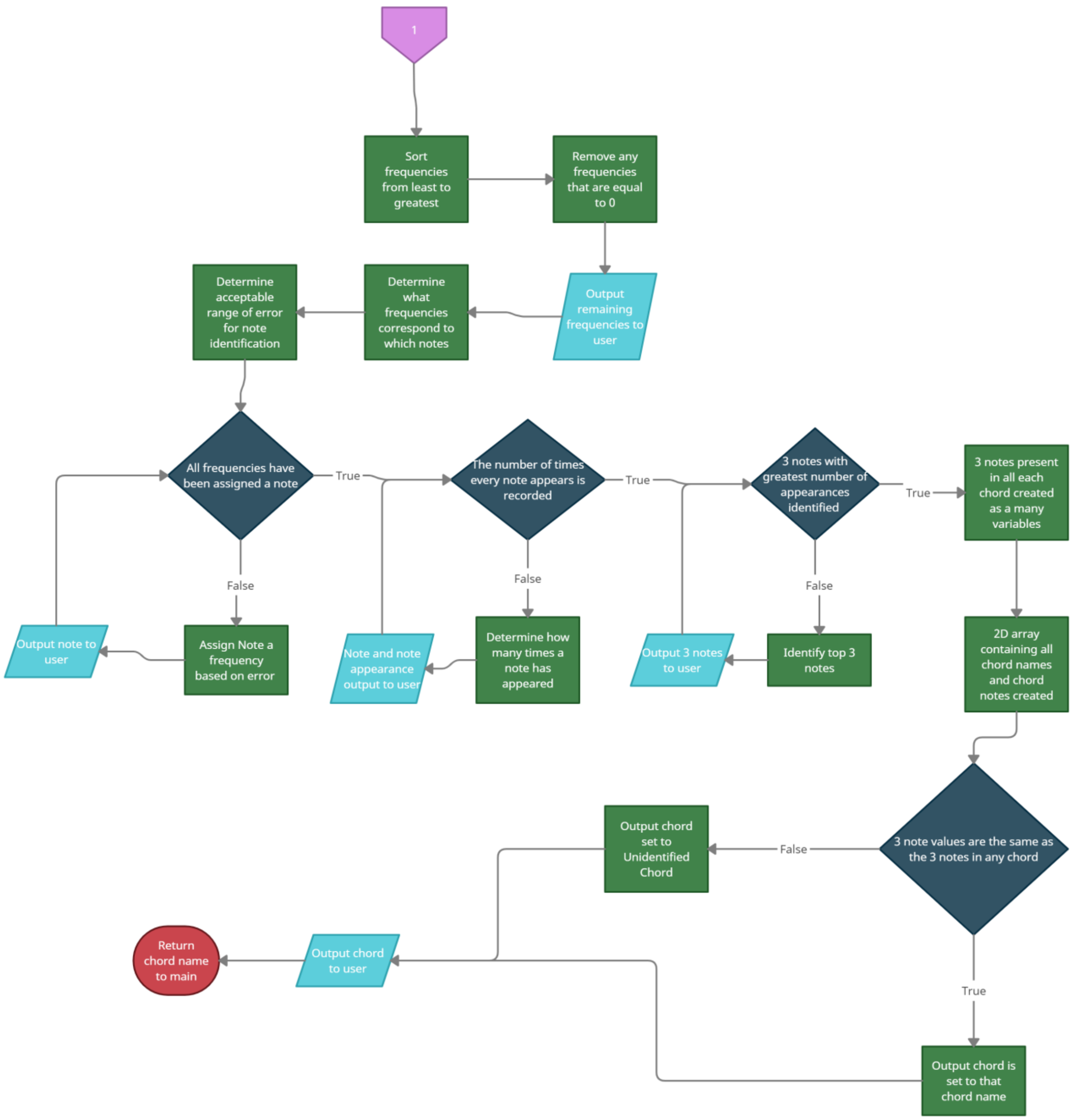
Flowchart of entire program:



Flowchart to turn audio into a chord 1:


audiotochord is called
Audio data is read from text file and stored in array (RawAudioData)
Determine Fourier coeficients from audio data while not all items have been read
True
Fourier Coefficient Generated for i value
False
abs(fCoefs/L)^2
DFT
Read DFT in 50 item chunks while not all items have been checked
True
Read while not all chunks have been checked
True
Determine Max of 50 data point segment
Max value and Array Index
False
False
Number of notes compared is filled with frequency values
False
1
True
Frequency where max value occurs
Frequency is within acceptable range
True
Frequency is stored as a note
False
Frequency is set to zero
Set max to 0 so next max is largest

Part 2:



5.2 Explanation of the Code found in Section 6.0

The code for the audiotochord() function was thoroughly explained in the section above regarding implementation of the algorithms. The program housing this key function is located in the audiotochord.cpp file. This program takes user input to determine what DFT reduction factor will be used for the calculation of the DFT. With this information the program checks for buttons being pressed and if button 0 is pressed the DE1-SoC beings recording audio. The audiotochord()

function converts the audio input data stored in a text file into a string containing the name of the chord. That string is divided into the note name and the chord type for example “C” and “major”. The last 2 displays then display the note name and the first 3 display the chord type. If the chord identified is marked as “Unidentified chord” two – symbols will appear on the 7 segment displays. The user is now granted the ability to interact with the keypad. A loop constantly is checking for keypad inputs. If a keypad button numbered 1-7 is being pressed that number chord relative to the identified chord will be played through the speakers. If the * is pressed the keypad loop ends. If any other button is pressed nothing will happen and a message will show that the button has no effect. The user is then reentered into the button loop where they may press button 0 to record more audio or button 2 to end the program entirely.

The Audio.cpp program records audio data fed in through the microphone jack and converts it into a text file labeled “Chord”. The usage of a text file was chosen as much of the formulation of the DFT algorithm was initially done in MATLAB. The text file allowed for data recorded on the DE1-SOC to still be analyzed by MATLAB. The initialize function in the audio program clears the data stored in the FIFO (first in, first out) memory of the DE1-SOC while also setting the FIFO to write instead of read so the audio is recorded and not played. A buffer of 10000 data points is created so the board takes in about 2 seconds of audio. The DE1-SoC beings recording until the number of data points is equal to the buffer size. At this point an array storing the data writes it to a text file.

The DE1-SoC.cpp program was created by Dr. Marpaung. It allows for C++ to read and write to the DE1-Soc. This program is the parent class to all other programs regarding access to the board’s functionality with various devices such as the jp1, audio ports, and buttons.

The SevenSegment.cpp program allowed for alterations to the 7-segment display. Functions within the class provided the ability to write to all 7 segment displays, write to a specific 7 segment display, clear all the 7 segment displays, and clear a specific 7 segment display. Two functions created specifically for this project allowed for the display of a specific note and chord determined by a string input. Given a string input the function ChordNoteDisplay() assigned a value to a variable which corresponded with an array of Hexadecimal numbers that would display specified note names on the displays 4 and 5. Given another string input the function ChordTypeDisplay() assigned a value to a variable which corresponded with that same array and would display the chord name on displays 0 through 2.

The Jp1.cpp program allowed for the program to both receive input as well as output signals through the JP1 ports. This class also utilized the clock as well. The function AllOutput() set every port as an output. This configuration was ideal for using speakers on any port. The function InitializeKeypad() set every port as an output with the exception of the row headers for the keypad. The jp1_WriteSpecific() function allowed for a low or high value to be written to any pin. The identifykeypadpressed() function set all columns to low with the exception of 1 and checked the value for each row. If certain hex values were returned when a button was pressed a character value corresponding to what appears on the keypad was set. The playchord() function utilized the 3 speakers to play a chord. With a given note, chord type, and keypad value pressed the function determined the frequencies of the 3 notes that needed to be played. By knowing the frequency of each root note the function used ratios between note frequencies to determine the frequencies of the major 3rd , minor 3rd , perfect 5th, and diminished 5th. With these ratios determined the clock began to cycle and every set number of cycles each speaker would go from low to high generating the correct tone to complete the triad.

5.3 Polymorphism or Linked List

I chose to use a linked list in support of my program. The linked list was used to clear out unwanted frequencies before assigning note values to them. This drastically reduced the number of calculations needed in determining what note each frequency was as all frequencies above 2000 Hz or below 16 Hz were simply removed from memory.

Linked list can be found on pages: 51 and 52

6.0 C++ Code

6.1 Original Program

Audio.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
#include <fstream>
#include "audio.h"


using namespace std;

const unsigned int AUDIO_OFFSET = 0x3040;
const unsigned int AUDIO_FIFOSPACE_OFFSET = 0x3044;
const unsigned int AUDIO_LEFT_OFFSET = 0x3048;
const unsigned int AUDIO_RIGHT_OFFSET = 0x304C;




Audio :: Audio(){ //Constructor
    cout << "Audio Created" << endl;

}

Audio ::~Audio(){ //Destructor
    cout << "Audio Destroyed" << endl;
}

void Audio :: InitializeRecording(){
    //Clear audio-in
    RegisterWrite(AUDIO_OFFSET,0x4);
    RegisterWrite(AUDIO_OFFSET,0x0);
    cout << "Recording..." << endl;
}

void Audio :: InitializePlayback(){
    //Clear audio-out
    RegisterWrite(AUDIO_OFFSET,0x8);
    RegisterWrite(AUDIO_OFFSET,0x0);
    cout << "Playing" << endl;
}

void Audio :: Record(int record){ //Sends Audio in data to text file
    const static int BUF_SIZE= 10000; // about 2 seconds of buffer (@ 8K
samples/sec)
    int BUF_THRESHOLD= 96; // 75% of 128 word buffer
    int buffer_index= 0;
```

```
    double left_buffer[BUF_SIZE];
    double right_buffer[BUF_SIZE];
    int fifospace;
    int createrecording = record;
    while(createrecording==1) { //begin recording
        RegisterWrite(LEDR_OFFSET, 0x1); // turn on LEDR[0]
        fifospace = RegisterRead(AUDIO_FIFOSPACE_OFFSET); // read the audio
port fifospace register
        if ((fifospace & 0x000000FF) > BUF_THRESHOLD) // check RARC
        {
// store data until the the audio-in FIFO is empty or the buffer
// is full
            while ((fifospace & 0x000000FF) && (buffer_index < BUF_SIZE)) {
                left_buffer[buffer_index] = RegisterRead(AUDIO_LEFT_OFFSET);
//Fills left_buffer with audio data
                right_buffer[buffer_index] =
RegisterRead(AUDIO_RIGHT_OFFSET);
                ++buffer_index;
                if (buffer_index == BUF_SIZE) { //when buffer is full
// done recording
                    createrecording = 0;
                    cout << "Done recording" << endl;
                    cout << "Calcuating..." << endl;

                    RegisterWrite(LEDR_OFFSET, 0x0);
                    ofstream audiofile;
                    audiofile.open("Chord.txt");
                    for (int i = 0; i < BUF_SIZE; i++) { //Audio data written
to text file
                        audiofile << left_buffer[i] << "\n";
                    }
                    audiofile.close();
                }
                fifospace = RegisterRead(AUDIO_FIFOSPACE_OFFSET); // read the
audio port fifospace register
            }
        }
    }
}
```

Audio.h

#ifndef AUDIO_H
#define AUDIO_H

#include "de1socfpga.h"

class Audio : public DE1SoCfpga{
public:

```
    Audio();
    ~Audio();
    void InitializeRecording();
    void InitializePlayback();
    void Record(int record);
};

#endif
```

JP1.cpp

```
#include <iostream>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
#include <string>
#include "jp1.h"


using namespace std;


jp1 :: jp1(){//Constructor
    cout << "JP1 connected" << endl;
    dataValue =0;
    directionValue=0;
    initialvalueLoadMPCore = RegisterRead(MPCORE_PRIV_TIMER_LOAD_OFFSET);
//Sets up timer
    initialvalueControlMPCore =
RegisterRead(MPCORE_PRIV_TIMER_CONTROL_OFFSET);
    initialvalueInterruptMPCore =
RegisterRead(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET);
    RegisterWrite(JP1_DIRECTION_REGISTER_OFFSET,directionValue);
}

jp1 ::~jp1(){//Destructor
    RegisterWrite(MPCORE_PRIV_TIMER_LOAD_OFFSET, initialvalueLoadMPCore);
    RegisterWrite(MPCORE_PRIV_TIMER_CONTROL_OFFSET,
initialvalueControlMPCore);
    RegisterWrite(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET,
initialvalueInterruptMPCore);
    cout << "JP1 disconnected" << endl;
}

void jp1 :: AllOutput(){//Sets all jp1 pins to output
```

```
    directionValue = directionValue | 0xFFFFFFFF;
    RegisterWrite(JP1_DIRECTION_REGISTER_OFFSET,directionValue);
}

void jp1 :: InitializeKeypad(){//Sets all jp1 pins to correct configuration
for keypad usage
    directionValue = 0;
    directionValue = directionValue | 0xFFFF95FF;
    RegisterWrite(JP1_DIRECTION_REGISTER_OFFSET,directionValue);

}

void jp1 :: jp1_WriteSpecific(int index, int state){//Write specific value to
a specific pin
    unsigned int bitmask = 1;
    int bitflip = 0xFFFFFFF;
    if(state == 0){
        bitmask = bitmask << index;
        bitmask = bitmask ^ bitflip;
        dataValue = dataValue & bitmask;
    }else{
        bitmask = bitmask << index;
        dataValue = dataValue | bitmask;
    }
    RegisterWrite(JP1_DATA_REGISTER_OFFSET,dataValue);

}

char jp1 :: identifykeypadpressed(){//Program to identify which keypad button
is pressed
    char keyout= '~';
    //By setting column values on and off and then checking the output of the
rows the button pressed is determined
    jp1_WriteSpecific(15,0);//Output pins on keypad
    jp1_WriteSpecific(17,1);
    jp1_WriteSpecific(19,1);
    jp1_WriteSpecific(21,1);
    if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x2A6800){
         keyout = '1';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x2A6200){
         keyout = '4';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x2A4A00){
         keyout = '7';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x2A2A00){
         keyout = '*';
    }

    jp1_WriteSpecific(15,1);
    jp1_WriteSpecific(17,0);
    jp1_WriteSpecific(19,1);
    jp1_WriteSpecific(21,1);
    if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x28E800){
         keyout = '2';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x28E200){
         keyout = '5';
```

```
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x28CA00){
         keyout = '8';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x28AA00){
         keyout = '0';
    }

    jp1_WriteSpecific(15,1);
    jp1_WriteSpecific(17,1);
    jp1_WriteSpecific(19,0);
    jp1_WriteSpecific(21,1);
    if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x22E800){
         keyout = '3';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x22E200){
         keyout = '6';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x22CA00){
         keyout = '9';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0x22AA00){
         keyout = '#';
    }

    jp1_WriteSpecific(15,1);
    jp1_WriteSpecific(17,1);
    jp1_WriteSpecific(19,1);
    jp1_WriteSpecific(21,0);
    if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0xAE800){
         keyout = 'A';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0xAE200){
         keyout = 'B';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0xACA00){
         keyout = 'C';
    }else if(RegisterRead(JP1_DATA_REGISTER_OFFSET) == 0xAAA00){
         keyout = 'D';
    }

    return keyout;
}

void jp1 :: playchord(string Note, string chordtype, char Keypad_value){
//Plays chord
    const unsigned int frequency_table[12]=
{440,466.16,493.88,523.25,554.37,587.33,622.25,659.25,698.46,739.99,783.99,83
0.61}; //All notes from A4 to Aflat5

    int countlow = 0; // timeout = 1/(200 MHz) x 200x10^6 = 1 sec
    RegisterWrite(MPCORE_PRIV_TIMER_LOAD_OFFSET, countlow); //Offset occurs
every count low
    RegisterWrite(MPCORE_PRIV_TIMER_CONTROL_OFFSET, 3);

    //AllOutput();

    int i = 0;
    int highlow1=0;
    int highlow2 =0;
    int highlow3=0;
    int ratio1 =0;
```

```
    int ratio2=0;
    int ratio3=0;
    int chordratiolcm;

    //Checks what chord has been input and sets value according to that
    int value =12;
    if(Note.compare("A") == 0){
       value =0;
    }else if(Note.compare("Bflat/Asharp")== 0){
        value=1;
    }else if(Note.compare("B")== 0){
        value=2;
    }else if(Note.compare("C")== 0){
        value=3;
    }else if(Note.compare("Dflat/Csharp")== 0){
        value=4;
    }else if(Note.compare("D")== 0){
        value=5;
    }else if(Note.compare("Eflat/Dsharp")== 0){
        value=6;
    }else if(Note.compare("E")== 0){
        value=7;
    }else if(Note.compare("F")== 0){
        value=8;
    }else if(Note.compare("Gflat/Fsharp")== 0){
        value=9;
    }else if(Note.compare("G")== 0){
        value=10;
    }else if(Note.compare("Aflat/Gsharp")== 0){
        value=11;
    }else{
        value =12;
    }
//Determines what chord needs to be played relative to either major or minor
chord
    if(chordtype.compare("major")== 0){
        if(Keypad_value == '1'){
            ratio1=60; //Ratio of clock cycles needed to play major chord
            ratio2=48;
            ratio3=40;
            chordratiolcm = 60;
        }else if(Keypad_value == '2'){
            ratio1=60; //Ratio of clock cycles needed to play minor chord
            ratio2=50;
            ratio3=40;
            chordratiolcm = 60;
            value = value+2;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '3'){
            ratio1=60;
            ratio2=50;
            ratio3=40;
            chordratiolcm = 60;
```

```
            value = value+4;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '4'){
            ratio1=60;
            ratio2=48;
            ratio3=40;
            chordratiolcm = 60;
            value = value+5;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '5'){
            ratio1=60;
            ratio2=48;
            ratio3=40;
            chordratiolcm = 60;
            value = value+7;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '6'){
            ratio1=60;
            ratio2=50;
            ratio3=40;
            chordratiolcm = 60;
            value = value+9;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '7'){
            ratio1=60; //Ratio of clock cycles needed to play diminished
chord
            ratio2=50;
            ratio3=42;
            chordratiolcm = 60;
            value = value+11;
            if(value>11){
                value=value-12;
            }

        }
    }else if(chordtype.compare("minor")== 0){
        if(Keypad_value == '1'){
            ratio1=60;
            ratio2=50;
            ratio3=40;
            chordratiolcm = 60;
        }else if(Keypad_value == '2'){
            ratio1=60;
            ratio2=50;
            ratio3=42;
            chordratiolcm = 60;
            value = value+2;
```

```
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '3'){
            ratio1=60;
            ratio2=48;
            ratio3=40;
            chordratiolcm = 60;
            value = value+3;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '4'){
            ratio1=60;
            ratio2=50;
            ratio3=40;
            chordratiolcm = 60;
            value = value+5;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '5'){
            ratio1=60;
            ratio2=50;
            ratio3=40;
            chordratiolcm = 60;
            value = value+7;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '6'){
            ratio1=60;
            ratio2=48;
            ratio3=40;
            chordratiolcm = 60;
            value = value+8;
            if(value>11){
                value=value-12;
            }
        }else if(Keypad_value == '7'){
            ratio1=60;
            ratio2=48;
            ratio3=40;
            chordratiolcm = 60;
            value = value+10;
            if(value>11){
                value=value-12;
            }
        }
    }else{
        cout << "error" << endl;
    }
    Hex_ClearAll();
    Hex_WriteSpecific(4, value); //Writes played chord root to 7seg
    if(ratio3 ==42){ //depending on note ratios writes chord type to 7 seg
```

```
        Hex_WriteSpecific(0,15);
    }else if(ratio2 == 48){
        Hex_WriteSpecific(0,13);
    }else if(ratio2 ==50){
        Hex_WriteSpecific(0,14);
    }





    countlow = 200000000/frequency_table[value]/2/chordratiolcm; //Number of
clock cycles before offset
    RegisterWrite(MPCORE_PRIV_TIMER_LOAD_OFFSET, countlow);
    RegisterWrite(MPCORE_PRIV_TIMER_CONTROL_OFFSET, 3);

    while (i < frequency_table[value]*chordratiolcm) {
        if (RegisterRead(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET) != 0) {
            RegisterWrite(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET, 1);
            i++;
            if (i % ratio1 == 0) {//Every ratio 1 clock cycles speaker 1 goes
from high to low or low to high
                if(highlow1==0){
                    highlow1++;
                    jp1_WriteSpecific(0,0);
                }else{
                    jp1_WriteSpecific(0,1);
                    highlow1=0;
                }
            }
            if (i % ratio2 == 0) {//Every ratio 2 clock cycles speaker 2 goes
from high to low or low to high
                if(highlow2==0){
                    highlow2++;
                    jp1_WriteSpecific(1,0);
                }else{
                    jp1_WriteSpecific(1,1);
                    highlow2=0;
                }
            }
            if (i % ratio3 == 0) {//Every ratio 3 clock cycles speaker 3 goes
from high to low or low to high
                if(highlow3==0){
                    highlow3++;
                    jp1_WriteSpecific(2,0);
                }else{
                    jp1_WriteSpecific(2,1);
                    highlow3=0;
                }
            }

        }
    }
```

```
cout << "chord playing complete" << endl;
}
```

JP1.h

```
#ifndef JP1_H
#define JP1_H

#include "de1socfpga.h"
#include "sevensegment.h"
#include <string.h>

using namespace std;

const unsigned int JP1_DATA_REGISTER_OFFSET = 0x60; // Points to JP1 DATA
const unsigned int JP1_DIRECTION_REGISTER_OFFSET = 0x64; // Points to JP1 DIRECTION
const unsigned int MPCORE_PRIV_TIMER_LOAD_OFFSET = 0xDEC600; // Points to LOAD
const unsigned int MPCORE_PRIV_TIMER_COUNTER_OFFSET = 0xDEC604; // Points to COUNTER
const unsigned int MPCORE_PRIV_TIMER_CONTROL_OFFSET = 0xDEC608; // Points to CONTROL
const unsigned int MPCORE_PRIV_TIMER_INTERRUPT_OFFSET = 0xDEC60C; // Points to INTERRUPT



class jp1 : public SevenSegment{
private:
    unsigned int dataValue;
    unsigned int directionValue; //0s are inputs 1s are outputs
    unsigned int initialvalueLoadMPCore;
    unsigned int initialvalueControlMPCore;
    unsigned int initialvalueInterruptMPCore;
```

```
public:
   jp1(); //Constructor
   ~jp1(); //Destructor
   void AllOutput();
   void jp1_WriteSpecific(int index, int state);
   void InitializeKeypad();
   char identifykeypadpressed();
   int readspecificindex(int index);
   void playchord(string chordnote, string chordtype, char Keypad_value);
};
#endif
```

DE1-SoC.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
#include "de1socfpga.h"
using namespace std;



DE1SoCfpga :: DE1SoCfpga(){
    // Open /dev/mem to give access to physical addresses
    fd = open( "/dev/mem", (O_RDWR | O_SYNC));
    if (fd == -1)         //  check for errors in openning /dev/mem
    {
        cout << "ERROR: could not open /dev/mem..." << endl;
        exit(1);
    }

    // Get a mapping from physical addresses to virtual addresses
    char *virtual_base = (char *)mmap (NULL, LW_BRIDGE_SPAN, (PROT_READ |
PROT_WRITE), MAP_SHARED, fd, LW_BRIDGE_BASE);
    if (virtual_base == MAP_FAILED)        // check for errors
    {
        cout << "ERROR: mmap() failed..." << endl;
        close (fd);        // close memory before exiting
        exit(1);         // Returns 1 to the operating system;
    }
    pBase= virtual_base;
    cout << "DE1 created." << endl;
}
```

```
DE1SoCfpga :: ~DE1SoCfpga(){
    if (munmap(pBase, LW_BRIDGE_SPAN) != 0) {
        cout << "ERROR: munmap() failed..." << endl;
        exit(1);
    }
    cout << "DE1 destroyed" << endl;
    close(fd);    // close memory
}

void DE1SoCfpga :: RegisterWrite(unsigned int offset, int value){
    *(volatile unsigned int *) (pBase + offset) = value;
}

int DE1SoCfpga :: RegisterRead(unsigned int offset){
    return *(volatile unsigned int *) (pBase + offset);
}
```

DE1-SoC.h

#ifndef DE1SOCFPGA_H
#define DE1SOCFPGA_H

// Important Message from Prof. Marpaung
// Read the PDF to find the real ADDRESS of LEDR, SW and KEY.
// End Important Message

// Physical base address of FPGA Devices
const unsigned int LW_BRIDGE_BASE = 0xFF200000; // Base offset

// Length of memory-mapped IO window
const unsigned int LW_BRIDGE_SPAN = 0x00DEC700; // Address map size

// Cyclone V FPGA device addresses
const unsigned int LEDR_OFFSET = 0x0;//real ADDRESS of RED LED - LW_BRIDGE_BASE ;
const unsigned int SW_OFFSET = 0x40;//real ADDRESS of SWITCH - LW_BRIDGE_BASE ;
const unsigned int KEY_OFFSET = 0x50;//real ADDRESS of PUSH BUTTON - LW_BRIDGE_BASE ;

```
class DE1SoCfpga
{
public:
    char *pBase;
    int fd;
    DE1SoCfpga(); //Constructor
    ~DE1SoCfpga(); //Destructor
    void RegisterWrite(unsigned int offset, int value);
    int RegisterRead(unsigned int offset);

};

#endif
```

Sevensegment.cpp

```
#include <iostream>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
#include <string>
#include "sevensegment.h"

using namespace std;
const unsigned int HEX3_HEX0_OFFSET = 0x20;
const unsigned int HEX5_HEX4_OFFSET = 0x30;


const unsigned int bit_values[16]={119,0x7F7C,127,57,0x397E,63,0x797C,
121,113,0x717E,61,0x777C, 1, 0x15770E , 0x153037,0x5E3015};
//Contains all notes and chord types displayed on the board




SevenSegment :: SevenSegment(){
    reg0_hexValue =0;
    reg1_hexValue =0;
    RegisterWrite(HEX3_HEX0_OFFSET,reg0_hexValue);
```

```
    RegisterWrite(HEX5_HEX4_OFFSET,reg1_hexValue);
    cout << "SevenSegment Created" << endl;

}
SevenSegment::~SevenSegment(){
    Hex_ClearAll();
    cout << "SevenSegment Destroyed" << endl;

}
int SevenSegment :: Read1Switch(int switchNum){
    int value = RegisterRead(SW_OFFSET); //sets value to switch values
    int bitmask=1;

    bitmask = bitmask << switchNum; //left shifts bitmask to the switch of
interest
    value = value & bitmask; //checks if bitmask and value are both 1 or not
    value = value >> switchNum; //right shifts value back as many times as
the number of the switch
    return value; //returns 1 or 0 depending on if switch is on or off
}

void SevenSegment :: testvalues(){//Just a test
    reg0_hexValue =bit_values[15] + (bit_values[6] << 8) + (bit_values[1] <<
16) + (bit_values[9] << 24);
    reg1_hexValue =bit_values[8] + (bit_values[2] << 8);
    RegisterWrite(HEX3_HEX0_OFFSET,reg0_hexValue);
    RegisterWrite(HEX5_HEX4_OFFSET,reg1_hexValue);
}
void SevenSegment:: Hex_ClearAll(){//Clears display
    reg0_hexValue =0;
    reg1_hexValue =0;
    RegisterWrite(HEX3_HEX0_OFFSET,reg0_hexValue);
    RegisterWrite(HEX5_HEX4_OFFSET,reg1_hexValue);
}
void SevenSegment :: Hex_ClearSpecific(int index){//Clears specific display
    int bitmask = 0x7F;
    int bitflip1 = 0xFFFFFFFF;
    int bitflip2 = 0xFFFF;
    if(index<4){//Shifts FFFF to correct spot and then reverses it so 7 seg
shows 0
        bitmask = bitmask << (index*8);
        bitmask = bitmask ^ bitflip1;
        reg0_hexValue= reg0_hexValue & bitmask;
    }else{
        bitmask = bitmask << ((index-4)*8);
        bitmask = bitmask ^ bitflip2;
        reg1_hexValue = reg1_hexValue & bitmask;
    }

    RegisterWrite(HEX3_HEX0_OFFSET,reg0_hexValue);
    RegisterWrite(HEX5_HEX4_OFFSET,reg1_hexValue);

}
void SevenSegment :: Hex_WriteSpecific(int index, int value){//Writes to
specific 7 seg display
```

```
    Hex_ClearSpecific(index);
    unsigned int bitmask = 0x7F;
    if(index<4){
        if (value >=0){
            bitmask = bit_values[value];
            bitmask = bitmask << (index *8);
            reg0_hexValue = reg0_hexValue | bitmask;
        }else{
            bitmask = bit_values[value+16];
            bitmask = bitmask << (index *8);
            reg0_hexValue = reg0_hexValue | bitmask;
        }
    }else{
        if (value >=0){
            bitmask = bit_values[value];
            bitmask = bitmask << ((index-4) *8);
            reg1_hexValue = reg1_hexValue | bitmask;
        }else{
            bitmask = bit_values[value+16];
            bitmask = bitmask << ((index-4) *8);
            reg1_hexValue = reg1_hexValue | bitmask;
        }
    }
    RegisterWrite(HEX3_HEX0_OFFSET,reg0_hexValue);
    RegisterWrite(HEX5_HEX4_OFFSET,reg1_hexValue);
}
void SevenSegment :: Hex_WriteNumber(int number){
    Hex_ClearAll();
    int bitmask =0xF;
    int display[6];
    if (number>=0){
        display[0] = number & bitmask;
        display[1]= (number >> 4) & bitmask;
        display[2]= (number >> 8) & bitmask;
        display[3]= (number >> 12) & bitmask;
        display[4]= (number >> 16) & bitmask;
        display[5]= (number >> 20) & bitmask;
        cout << endl <<endl;
        for(int j=0; j <6; j++){
            bool continuecheck =true;
            for(int i =0; i <16; i++){
                if(display[j] == i && continuecheck == true){
                    display[j] = bit_values[i];
                    continuecheck =false;
                }
            }
        }
    }else{
        number = number + 16777216;
        display[0] = number & bitmask;
        display[1]= (number >> 4) & bitmask;
        display[2]= (number >> 8) & bitmask;
        display[3]= (number >> 12) & bitmask;
        display[4]= (number >> 16) & bitmask;
        display[5]= (number >> 20) & bitmask;
```

```
        cout << endl <<endl;
        for(int j=0; j <6; j++){
            bool continuecheck =true;
            for(int i =0; i <16; i++){
                if(display[j] == i && continuecheck == true){
                    display[j] = bit_values[i];
                    continuecheck =false;
                }
            }
        }
    }
    reg0_hexValue = reg0_hexValue | display[0];
    reg0_hexValue = reg0_hexValue | (display[1] << 8);
    reg0_hexValue = reg0_hexValue | (display[2] << 16);
    reg0_hexValue = reg0_hexValue | (display[3] << 24);
    reg1_hexValue = reg1_hexValue | display[4];
    reg1_hexValue = reg1_hexValue | (display[5] << 8);
    RegisterWrite(HEX3_HEX0_OFFSET,reg0_hexValue);
    RegisterWrite(HEX5_HEX4_OFFSET,reg1_hexValue);

}

void SevenSegment :: ChordNoteDisplay(string Note){ //Based on the string
that enters the corresponding 7 seg appears
    int value =12;
    if(Note.compare("A") == 0){
       value =0;
    }else if(Note.compare("Bflat/Asharp")== 0){
        value=1;
    }else if(Note.compare("B")== 0){
        value=2;
    }else if(Note.compare("C")== 0){
        value=3;
    }else if(Note.compare("Dflat/Csharp")== 0){
        value=4;
    }else if(Note.compare("D")== 0){
        value=5;
    }else if(Note.compare("Eflat/Dsharp")== 0){
        value=6;
    }else if(Note.compare("E")== 0){
        value=7;
    }else if(Note.compare("F")== 0){
        value=8;
    }else if(Note.compare("Gflat/Fsharp")== 0){
        value=9;
    }else if(Note.compare("G")== 0){
        value=10;
    }else if(Note.compare("Aflat/Gsharp")== 0){
        value=11;
    }else{
        value =12;
    }


    Hex_WriteSpecific(4, value);
```

```
}

void SevenSegment :: ChordTypeDisplay(string Type){ //Depending on chord the
corresponding 7 seg turns on
    int value =12;
    if(Type.compare("major") == 0){
        value =13;
    }else if(Type.compare("minor")== 0){
        value=14;
    }else if (Type.compare("diminished")==0){
        value =15;
    }
    else{
        value =12;
    }


    Hex_WriteSpecific(0, value);

}
```

Sevensegment.h

```
#ifndef SEVENSEGMENT_H
#define SEVENSEGMENT_H

#include "de1socfpga.h"
#include <string.h>



using namespace std;

class SevenSegment : public DE1SoCfpga{
private:
   unsigned int reg0_hexValue;
   unsigned int reg1_hexValue;

public:
   SevenSegment(); //Constructor
   ~SevenSegment(); //Destructor
```

```
    int Read1Switch(int switchNum);
    void testvalues();
    void Hex_ClearAll();
    void Hex_WriteSpecific(int index, int value);
    void Hex_ClearSpecific(int index);
    void Hex_WriteNumber(int number);
    void ChordNoteDisplay(string Note);
    void ChordTypeDisplay(string Type);
};

#endif
```

Audiotochord.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <complex> //https://www.geeksforgeeks.org/complex-numbers-c-set-1/
#include <math.h>
#include <bits/stdc++.h>
#include "chordtokeypad.h"



using namespace std;

void filereader(double array[],int size);
int Arraylength(complex<double> array[]);
void scalarProductMat(complex<double> mat[], complex<double> k,int
size,complex<double> returnedarray[]);
complex<double> DotProduct(complex<double> arr1[], complex<double> arr2[],int
size);
complex<double> SumElementsInArray(complex<double> arr[],int size);
double maxofarray(double arr[], int n);
int indexatmax(double arr[], int n);
string audiotochord();
string SplitString(string str, int wordnumber);


int main(void) {
```

```
    Audio *audio = new Audio;
    SevenSegment *sevensegment = new SevenSegment;
    jp1 *keypad = new jp1;
    int record = 0, play = 0;
    record = 0;
    play = 0;
    int buttonloop= 1;
    int KEY_value=0;
    char Keypad_value= '~';
    string Identifiedchord;
    while (buttonloop == 1) {
        while (KEY_value ==0){
            KEY_value = audio->RegisterRead(KEY_OFFSET);
        }

        if (KEY_value == 0x1){
            audio->InitializeRecording();
            record =1;
            audio->Record(record);
            Identifiedchord =audiotochord();
            cout << Identifiedchord << endl;
            cout << "\n" << "\n";

            string ChordNote;
            string ChordType;
            ChordNote = SplitString(Identifiedchord,0);
            ChordType = SplitString(Identifiedchord,1);
            sevensegment-> Hex_ClearAll();

            sevensegment->ChordNoteDisplay(ChordNote);
            sevensegment->ChordTypeDisplay(ChordType);
            keypad->InitializeKeypad();

            cout << "Enter Key:" << endl;
            while (Keypad_value == '~'){
                Keypad_value = keypad->identifykeypadpressed();
            }
            cout << Keypad_value << endl;
            Keypad_value = '~';


            record =0;
        }

        if(KEY_value == 0x4){
            buttonloop = 1;
        }


        while(KEY_value !=0){
            if(KEY_value == 0x4){
                buttonloop = 0;
                audio->RegisterWrite(LEDR_OFFSET,0x0);
            }
            KEY_value = audio->RegisterRead(KEY_OFFSET);
```

```
        }


    }
    delete audio;
    delete sevensegment;
    delete keypad;
}

void filereader(double array[],int size){
    ifstream file("Chord.txt");
    int count = 0;
    double x;

    while (count < size && file >> x) {
        array[count++] = x;
    }


    // display the values stored in the array
    /*for (int i = 0; i < count; i++) {
        cout << array[i] << "\n";
    }*/

}

int Arraylength(complex<double> array[]){
    int x= *(&array +1)-array;
    return x;
}

void scalarProductMat(complex<double> mat[], complex<double> k,int
size,complex<double> returnedarray[]){
    // scalar element is multiplied by the matrix
    for (int i = 0; i < size; i++)
        returnedarray[i] = mat[i] *k;
}

complex<double> DotProduct(complex<double> arr1[], complex<double> arr2[],int
size){
    complex<double> DotProduct(0,0);
    for(int i; i< size; i++){
        DotProduct= DotProduct + (arr1[i] *arr2[i]);
    }
    return DotProduct;
}

complex<double> SumElementsInArray(complex<double> arr[],int size){
    complex<double> SumElementsInArray(0,0);
    for(int i=0; i< size; i++){
        SumElementsInArray+= arr[i];
    }
    return SumElementsInArray;
}
```

```
double maxofarray(double arr[], int n){
    double max = arr[0];
    for (int i = 0; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

int indexatmax(double arr[], int n){
    double max = arr[0];
    int indexatmax=0;
    for (int i = 0; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
            indexatmax =i;
        }
    }
    return indexatmax;
}

string audiotochord(){
    const int RawDatasize = 10000;
    const int DFTsegment= 50;
    const int NumberofNotesCompared = 5;
    double pi =3.14159265358979;
    double e =2.7182818284;
    double RawAudioData[RawDatasize];
    complex<double> Imaginarynumber (0,1);
    filereader(RawAudioData,RawDatasize);


    double fs = 8000;
    double L = 10000; //RawDatasize
    double dft_loop_reduction = 10; //set to 1 for no reduction
    double f[RawDatasize/2+1];

    for(int i=0; i<=RawDatasize/2; i++){
        f[i] = fs* (i/L);
    }

    complex<double> fourTime[RawDatasize];
    double inttodouble =0;
    for(int i=0; i<=RawDatasize-1; i++){
        inttodouble=i;
        fourTime[i] = (inttodouble/L);
        //cout << fourTime[i] << endl;
    }
    inttodouble=0;

    complex<double> fCoefs[RawDatasize];
    complex<double> csw[RawDatasize];
    complex<double> multipliedfourTime[RawDatasize];
    complex<double> signaltimescsw[RawDatasize];
```

```
    for (int i=1; i<=RawDatasize; i= i +dft_loop_reduction){
        inttodouble=i;
        scalarProductMat(fourTime,(-1.0*Imaginarynumber*2.0*pi*(inttodouble-
1)),RawDatasize,multipliedfourTime);
        for(int j=0; j<RawDatasize;j++){
            csw[j]= multipliedfourTime[j];
        }
        for(int j=0; j<RawDatasize;j++){
            csw[j] = pow(e,csw[j]);
        }
        for(int j=0; j<RawDatasize;j++){
            signaltimescsw[j] = csw[j]*RawAudioData[j];
        }
        fCoefs[i] = SumElementsInArray(signaltimescsw,RawDatasize);
    }

    double P1[RawDatasize];

    for(int i=0;i<=RawDatasize;i++){
        P1[i]= pow(abs(fCoefs[i]/L),2);
    }


    double Psegment[DFTsegment];
    int indexAtMaxY =0;
    double maxYValue=0.0;
    double xValueAtMaxYValue =0.0;
    double ArrayofMaximums [RawDatasize/DFTsegment-50+1];
    int ArrayofMaxIndicies [RawDatasize/DFTsegment-50+1];
    double ChordFrequencies [NumberofNotesCompared];




    for (int i =0; i<=RawDatasize/DFTsegment-50; i++){
        indexAtMaxY=0;
        maxYValue =0;
        for(int j=0; j<DFTsegment; j++){
            Psegment[j] = P1[(i*50)+j+1];
            indexAtMaxY= indexatmax(Psegment,DFTsegment);
            indexAtMaxY = indexAtMaxY + 50*i +1;
            xValueAtMaxYValue = P1[indexAtMaxY];
            ArrayofMaximums[i] = xValueAtMaxYValue;
            ArrayofMaxIndicies[i] = indexAtMaxY;
        }
    }
    int FrequencyIndexAtMax =0;


    for (int i =0; i<NumberofNotesCompared; i++){
        maxYValue = maxofarray(ArrayofMaximums,RawDatasize/DFTsegment-50+1);
        indexAtMaxY= indexatmax(ArrayofMaximums,RawDatasize/DFTsegment-50+1);
        FrequencyIndexAtMax =ArrayofMaxIndicies[indexAtMaxY];
        ChordFrequencies[i] =f[FrequencyIndexAtMax];
```

```
        ArrayofMaximums[indexAtMaxY] =0;
    }

    int arraysortsize = sizeof(ChordFrequencies)/sizeof(ChordFrequencies[0]);
    sort(ChordFrequencies, ChordFrequencies +arraysortsize);

    cout << "ChordFrequencies" << endl;
    for (int i=0; i <NumberofNotesCompared; i++){
        cout << ChordFrequencies[i] << " ";
    }
    cout << "\n" << "\n";

    double Afrequencies[7];
    double BflatAsharpfrequencies[7];
    double Bfrequencies[7];
    double Cfrequencies[7];
    double DflatCsharpfrequencies[7];
    double Dfrequencies[7];
    double EflatDsharpfrequencies[7];
    double Efrequencies[7];
    double Ffrequencies[7];
    double GflatFsharpfrequencies[7];
    double Gfrequencies[7];
    double AflatGsharpfrequencies[7];

    double note=0.0;

    for (int i=0; i<=6; i++){
        inttodouble = i;
        note = 440*pow(2,(((1+(12*inttodouble))-49)/12));
        Afrequencies[i]= note;
        note = 440*pow(2,(((2+(12*inttodouble))-49)/12));
        BflatAsharpfrequencies[i]= note;
        note = 440*pow(2,(((3+(12*inttodouble))-49)/12));
        Bfrequencies[i]= note;
        note = 440*pow(2,(((4+(12*inttodouble))-49)/12));
        Cfrequencies[i]= note;
        note = 440*pow(2,(((5+(12*inttodouble))-49)/12));
        DflatCsharpfrequencies[i]= note;
        note = 440*pow(2,(((6+(12*inttodouble))-49)/12));
        Dfrequencies[i]= note;
        note = 440*pow(2,(((7+(12*inttodouble))-49)/12));
        EflatDsharpfrequencies[i]= note;
        note = 440*pow(2,(((8+(12*inttodouble))-49)/12));
        Efrequencies[i]= note;
        note = 440*pow(2,(((9+(12*inttodouble))-49)/12));
        Ffrequencies[i]= note;
        note = 440*pow(2,(((10+(12*inttodouble))-49)/12));
        GflatFsharpfrequencies[i]= note;
        note = 440*pow(2,(((11+(12*inttodouble))-49)/12));
        Gfrequencies[i]= note;
        note = 440*pow(2,(((12+(12*inttodouble))-49)/12));
        AflatGsharpfrequencies[i]= note;
    }
    inttodouble=0;
```

```
    double NoteError=0;
    string ChordNotes[5];

    for(int i =0; i<NumberofNotesCompared;i++){
        NoteError =0;
        if (ChordFrequencies[i] > 0 && ChordFrequencies[i]< 63) {
//Acceptable error of pitch at each octave in Hz
            NoteError = 2;
        }else if (ChordFrequencies[i] > 63 && ChordFrequencies[i]< 127) {
            NoteError = 4;
        }else if (ChordFrequencies[i] > 127 && ChordFrequencies[i]< 255) {
            NoteError = 8;
        }else if (ChordFrequencies[i] > 255 && ChordFrequencies[i]< 511) {
            NoteError = 16;
        }else if (ChordFrequencies[i] > 511 && ChordFrequencies[i]< 1023) {
            NoteError = 32;
        }else if (ChordFrequencies[i] > 1023 && ChordFrequencies[i]< 2047) {
            NoteError = 64;
        }else if (ChordFrequencies[i] > 2047 && ChordFrequencies[i]< 10000) {
            NoteError = 128;
        }

        for (int j=0; j<=6; j++){
            if ((Afrequencies[j]-NoteError <ChordFrequencies[i] &&
Afrequencies[j]+NoteError >ChordFrequencies[i]) && (abs(Afrequencies[j]-
ChordFrequencies[i])< abs(BflatAsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Afrequencies[j]-ChordFrequencies[i])< abs(AflatGsharpfrequencies[j]-
ChordFrequencies[i]))){
                ChordNotes[i]="A";
            }
            if ((BflatAsharpfrequencies[j]-NoteError <ChordFrequencies[i] &&
BflatAsharpfrequencies[j]+NoteError >ChordFrequencies[i])&&
(abs(BflatAsharpfrequencies[j]-ChordFrequencies[i])< abs(Bfrequencies[j]-
ChordFrequencies[i]) && abs(BflatAsharpfrequencies[j]-ChordFrequencies[i])<
abs(Afrequencies[j]-ChordFrequencies[i]))){
                ChordNotes[i]="Bflat/Asharp";
            }
            if ((Bfrequencies[j]-NoteError <ChordFrequencies[i] &&
Bfrequencies[j]+NoteError >ChordFrequencies[i]) && (abs(Bfrequencies[j]-
ChordFrequencies[i])< abs(Cfrequencies[j]-ChordFrequencies[i]) &&
abs(Bfrequencies[j]-ChordFrequencies[i])< abs(BflatAsharpfrequencies[j]-
ChordFrequencies[i]))){
                ChordNotes[i]="B";
            }
            if ((Cfrequencies[j]-NoteError <ChordFrequencies[i] &&
Cfrequencies[j]+NoteError >ChordFrequencies[i]) && (abs(Cfrequencies[j]-
ChordFrequencies[i])< abs(DflatCsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Cfrequencies[j]-ChordFrequencies[i])< abs(Bfrequencies[j]-
ChordFrequencies[i]))){
                ChordNotes[i]="C";
            }
            if ((DflatCsharpfrequencies[j]-NoteError <ChordFrequencies[i] &&
DflatCsharpfrequencies[j]+NoteError >ChordFrequencies[i])&&
(abs(DflatCsharpfrequencies[j]-ChordFrequencies[i])< abs(Dfrequencies[j]-
ChordFrequencies[i]) && abs(DflatCsharpfrequencies[j]-ChordFrequencies[i])<
```

```
abs(Cfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="Dflat/Csharp";
            }
            if ((Dfrequencies[j]-NoteError <ChordFrequencies[i] &&
Dfrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Dfrequencies[j]-
ChordFrequencies[i])< abs(EflatDsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Dfrequencies[j]-ChordFrequencies[i])< abs(DflatCsharpfrequencies[j]-
ChordFrequencies[i])))){
                ChordNotes[i]="D";
            }
            if ((EflatDsharpfrequencies[j]-NoteError <ChordFrequencies[i] &&
EflatDsharpfrequencies[j]+NoteError >ChordFrequencies[i])&&
(abs(EflatDsharpfrequencies[j]-ChordFrequencies[i])< abs(Efrequencies[j]-
ChordFrequencies[i]) && abs(EflatDsharpfrequencies[j]-ChordFrequencies[i])<
abs(Dfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="Eflat/Dsharp";
            }
            if ((Efrequencies[j]-NoteError <ChordFrequencies[i] &&
Efrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Efrequencies[j]-
ChordFrequencies[i])< abs(Ffrequencies[j]-ChordFrequencies[i]) &&
abs(Efrequencies[j]-ChordFrequencies[i])< abs(EflatDsharpfrequencies[j]-
ChordFrequencies[i])))){
                ChordNotes[i]="E";
            }
            if ((Ffrequencies[j]-NoteError <ChordFrequencies[i] &&
Ffrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Ffrequencies[j]-
ChordFrequencies[i])< abs(GflatFsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Ffrequencies[j]-ChordFrequencies[i])< abs(Efrequencies[j]-
ChordFrequencies[i])))){
                ChordNotes[i]="F";
            }
            if ((GflatFsharpfrequencies[j]-NoteError <ChordFrequencies[i] &&
GflatFsharpfrequencies[j]+NoteError >ChordFrequencies[i])&&
(abs(GflatFsharpfrequencies[j]-ChordFrequencies[i])< abs(Gfrequencies[j]-
ChordFrequencies[i]) && abs(GflatFsharpfrequencies[j]-ChordFrequencies[i])<
abs(Ffrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="Gflat/Fsharp";
            }
            if ((Gfrequencies[j]-NoteError <ChordFrequencies[i] &&
Gfrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Gfrequencies[j]-
ChordFrequencies[i])< abs(AflatGsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Gfrequencies[j]-ChordFrequencies[i])< abs(GflatFsharpfrequencies[j]-
ChordFrequencies[i])))){
                ChordNotes[i]="G";
            }
            if ((AflatGsharpfrequencies[j]-NoteError <ChordFrequencies[i] &&
AflatGsharpfrequencies[j]+NoteError >ChordFrequencies[i])&&
(abs(AflatGsharpfrequencies[j]-ChordFrequencies[i])< abs(Afrequencies[j]-
ChordFrequencies[i]) && abs(AflatGsharpfrequencies[j]-ChordFrequencies[i])<
abs(Gfrequencies[j]-ChordFrequencies[i]))) {
                ChordNotes[i] = "Aflat/Gsharp";
            }
        }
    }
```

```
    cout << "ChordNotes" << endl;
    for(int j=0; j<=4;j++){
        cout << ChordNotes[j] << " ";
    }
    cout << "\n" << "\n";

    string ListofChords[24] = {"A major", "A minor", "Bflat/Asharp major",
"Bflat/Asharp minor", "B major", "B minor","C major", "C minor","Dflat/Csharp
major", "Dflat/Csharp minor","D major", "D minor","Eflat/Dsharp major",
"Eflat/Dsharp minor","E major", "E minor","F major", "F minor","Gflat/Fsharp
major", "Gflat/Fsharp minor","G major", "G minor","Aflat/Gsharp major",
"Aflat/Gsharp minor"};
    string Amajorcomponents[3] = {"A","Dflat/Csharp","E"};
    string Aminorcomponents[3] = {"A", "C", "E"};
    string BflatAsharpmajorcomponents[3] = {"Bflat/Asharp", "D", "F"};
    string BflatAsharpminorcomponents[3] = {"Bflat/Asharp", "Dflat/Csharp",
"F"};
    string Bmajorcomponents[3] = {"B", "Eflat/Dsharp", "Gflat/Fsharp"};
    string Bminorcomponents[3] = {"B", "D", "Gflat/Fsharp"};
    string Cmajorcomponents[3] = {"C", "E", "G"};
    string Cminorcomponents[3] = {"C", "Eflat/Dsharp", "G"};
    string DflatCsharpmajorcomponents[3] = {"Dflat/Csharp", "F",
"Aflat/Gsharp"};
    string DflatCsharpminorcomponents[3] = {"Dflat/Csharp", "E",
"Aflat/Gsharp"};
    string Dmajorcomponents[3] = {"D", "Gflat/Fsharp", "A"};
    string Dminorcomponents[3] = {"D", "F", "A"};
    string EflatDsharpmajorcomponents[3] = {"Eflat/Dsharp", "G",
"Bflat/Asharp"};
    string EflatDsharpminorcomponents[3] = {"Eflat/Dsharp", "Gflat/Fsharp",
"Bflat/Asharp"};
    string Emajorcomponents[3] = {"E", "Aflat/Gsharp", "B"};
    string Eminorcomponents[3] = {"E", "G", "B"};
    string Fmajorcomponents[3] = {"F", "A", "C"};
    string Fminorcomponents[3] = {"F", "Aflat/Gsharp", "C"};
    string GflatFsharpmajorcomponents[3] = {"Gflat/Fsharp", "Bflat/Asharp",
"Dflat/Csharp"};
    string GflatFsharpminorcomponents[3] = {"Gflat/Fsharp", "A",
"Dflat/Csharp"};
    string Gmajorcomponents[3] = {"G", "B", "D"};
    string Gminorcomponents[3] = {"G", "Bflat/Asharp", "D"};
    string AflatGsharpmajorcomponents[3] = {"Aflat/Gsharp", "C",
"Eflat/Dsharp"};
    string AflatGsharpminorcomponents[3] = {"Aflat/Gsharp", "B",
"Eflat/Dsharp"};

    string ChordComponents[24][3];
    for (int j=0; j<3; j++){
        ChordComponents[0][j]=Amajorcomponents[j];
        ChordComponents[1][j]=Aminorcomponents[j];
        ChordComponents[2][j]=BflatAsharpmajorcomponents[j];
        ChordComponents[3][j]=BflatAsharpminorcomponents[j];
        ChordComponents[4][j]=Bmajorcomponents[j];
        ChordComponents[5][j]=Bminorcomponents[j];
```

```
        ChordComponents[6][j]=Cmajorcomponents[j];
        ChordComponents[7][j]=Cminorcomponents[j];
        ChordComponents[8][j]=DflatCsharpmajorcomponents[j];
        ChordComponents[9][j]=DflatCsharpminorcomponents[j];
        ChordComponents[10][j]=Dmajorcomponents[j];
        ChordComponents[11][j]=Dminorcomponents[j];
        ChordComponents[12][j]=EflatDsharpmajorcomponents[j];
        ChordComponents[13][j]=EflatDsharpminorcomponents[j];
        ChordComponents[14][j]=Emajorcomponents[j];
        ChordComponents[15][j]=Eminorcomponents[j];
        ChordComponents[16][j]=Fmajorcomponents[j];
        ChordComponents[17][j]=Fminorcomponents[j];
        ChordComponents[18][j]=GflatFsharpmajorcomponents[j];
        ChordComponents[19][j]=GflatFsharpminorcomponents[j];
        ChordComponents[20][j]=Gmajorcomponents[j];
        ChordComponents[21][j]=Gminorcomponents[j];
        ChordComponents[22][j]=AflatGsharpmajorcomponents[j];
        ChordComponents[23][j]=AflatGsharpminorcomponents[j];
    }
    string comparisonchord[3];
    for(int i=0;i<3;i++){
        comparisonchord[i] = "Q";
    }

    for (int i=0; i<3; i++){
        comparisonchord[i] = ChordNotes[i];
    }




    for(int j=0; j<5; j++){
            if (comparisonchord[1] == comparisonchord[2] ||
comparisonchord[1] == comparisonchord[0]){
                if (comparisonchord[1] !=ChordNotes[j] && ChordNotes[j]
!=comparisonchord[0] && ChordNotes[j] !=comparisonchord[2]){
                    comparisonchord[2] = ChordNotes[j];
                }
            }
        if (comparisonchord[2] == comparisonchord[0] || comparisonchord[2] ==
comparisonchord[1]){
            if (comparisonchord[2] !=ChordNotes[j] && ChordNotes[j]
!=comparisonchord[0] && ChordNotes[j] !=comparisonchord[1]){
                comparisonchord[2] = ChordNotes[j];
            }
        }
    }

    cout << "comparisonchord" << endl;
    for(int j=0; j<=2;j++){
        cout << comparisonchord[j] << " ";
    }
    cout << "\n" << "\n";

    int isthisthecorrectchord=0;
```

```
    string CorrectlyIdentifiedChord ="Unidentified Chord";
    for(int i=0; i<24; i++){
        isthisthecorrectchord= 0;
        for(int j=0; j<3; j++){
            if(comparisonchord[j]== ChordComponents[i][0] ||
comparisonchord[j]== ChordComponents[i][1] || comparisonchord[j]==
ChordComponents[i][2]){
                isthisthecorrectchord =isthisthecorrectchord+1;
            }else{
                isthisthecorrectchord= 0;
            }
        }
        if(isthisthecorrectchord ==3){
            CorrectlyIdentifiedChord=ListofChords[i];
        }
    }

    cout << "Identified Chord" << endl;
    return CorrectlyIdentifiedChord;
}

string SplitString(string str, int wordnumber){
    stringstream ss(str);
    string split;
    string output;
    int counter =0;
    while(ss >> split){
        if(counter == wordnumber){
            output =split;
        }
        counter++;
    }
    return output;
}
```

Audiotochord.h

#ifndef AUDIOTOCHORD_H

#define AUDIOTOCHORD_H

#include "audio.h"

#include "de1socfpga.h"

#include "sevensegment.h"

#include "jp1.h"

Makefile

```
# audiotochord

runaudiotochord: audiotochord.o de1socfpga.o jp1.o audio.o sevensegment.o
		g++ audiotochord.o audio.o de1socfpga.o jp1.o sevensegment.o -o runaudiotochord

de1socfpga.o: de1socfpga.cpp de1socfpga.h
		g++ -g -Wall -c de1socfpga.cpp

audio.o: audio.cpp audio.h
		g++ -g -Wall -c audio.cpp

sevensegment.o: sevensegment.cpp sevensegment.h
		g++ -g -Wall -c sevensegment.cpp

jp1.o: jp1.cpp jp1.h
		g++ -g -Wall -c jp1.cpp

audiotochord.o: audiotochord.cpp audiotochord.h
		g++ -g -Wall -c audiotochord.cpp

clean:
		rm audiotochord.o de1socfpga.o sevensegment.o jp1.o audio.o runaudiotochord


#endif
```

6.2 Improved Program (The only function that changed between iterations was the audiotochord() function so I will display only it here)

Audiotochord() (Linked list is highlighted in blue)

```
string audiotochord(double DFTreduction){
    const int RawDatasize = 10000;//Number of rows in the data file
    const int DFTsegment= 50;//Number of data points analyized at a time to
find local maxes of DFT graph
    const int NumberofNotesCompared = 15;//Number of notes used to determine
chord
    double pi =3.14159265358979;
    double e =2.7182818284;
    double elapsedtime; //Clock related variables
    double RawAudioData[RawDatasize];//Array that contains audio data from
text file
    complex<double> Imaginarynumber (0,1);//This is the value i
    filereader(RawAudioData,RawDatasize);//Reads txt file and fills
RawAudioData with each row


    double fs = 8000;//Sampling frequency of De1soc
    double L = 10000; //RawDatasize
    double dft_loop_reduction = DFTreduction; //set to 1 for no reduction
    double f[RawDatasize/2+1];

    for(int i=0; i<=RawDatasize/2; i++){
        f[i] = fs* (i/L);//Makes frequency array half of the size of raw
audio file as negative frequencies are discared
    }

    complex<double> fourTime[RawDatasize];
    double inttodouble =0; //Used to prevent errors in comparing ints to
doubles
    for(int i=0; i<=RawDatasize-1; i++){
        inttodouble=i;
        fourTime[i] = (inttodouble/L); //Fills fourTime with 10000 divisions
of audio data size
        //cout << fourTime[i] << endl;
    }
    inttodouble=0;

    complex<double> fCoefs[RawDatasize];
    complex<double> csw[RawDatasize];
    complex<double> multipliedfourTime[RawDatasize];
    complex<double> signaltimescsw[RawDatasize];

    //Begin DFT clock
    clock_t start =clock();


    for (int i=1; i<=RawDatasize; i= i +dft_loop_reduction){//Calculates
fourier coeficients
        inttodouble=i;
        scalarProductMat(fourTime,(-1.0*Imaginarynumber*2.0*pi*(inttodouble-
1)),RawDatasize,multipliedfourTime);
        for(int j=0; j<RawDatasize;j++){
            csw[j]= multipliedfourTime[j];
        }
        for(int j=0; j<RawDatasize;j++){
```

```
            csw[j] = pow(e,csw[j]);//Forms complex sine wave
        }
        for(int j=0; j<RawDatasize;j++){
            signaltimescsw[j] = csw[j]*RawAudioData[j];
        }
        fCoefs[i] = SumElementsInArray(signaltimescsw,RawDatasize); //Fourier
coefficients are the sum of complex sinewaves
    }

    double DFT[RawDatasize];

    for(int i=0;i<=RawDatasize;i++){
        DFT[i]= pow(abs(fCoefs[i]/L),2); //Takes magnitude of frequency
specturm so there are no negative values
    }

    //End DFT clock
    clock_t end = clock();
    elapsedtime = double(end - start)/CLOCKS_PER_SEC;
    cout << "Time to create frequency spectrum: "<< elapsedtime << " seconds"
<< endl << endl;


    double Psegment[DFTsegment];
    int indexAtMaxY =0;
    double xValueAtMaxYValue =0.0;
    int Highfrequencyvalues=DFTreduction*5; //Removes highest frequencies
from analysis as they are higher than necessary
    double ArrayofMaximums [RawDatasize/DFTsegment-Highfrequencyvalues+1];
    int ArrayofMaxIndicies [RawDatasize/DFTsegment-Highfrequencyvalues+1];

    //Linked list
    int linkedlistcount = NumberofNotesCompared;
    int linkedlistsize = NumberofNotesCompared;
    double PreLinkedChordFrequencies[linkedlistsize];




    for (int i =0; i<=RawDatasize/DFTsegment-Highfrequencyvalues;
i++){//Finds maximum value for every 50 data points
        indexAtMaxY=0;
        for(int j=0; j<DFTsegment; j++){ //Checks each 50 point segment and
marks max and the index the max occurs at
            Psegment[j] = DFT[(i*DFTsegment)+j+1];
            indexAtMaxY= indexatmax(Psegment,DFTsegment);
            indexAtMaxY = indexAtMaxY + DFTsegment*i +1;
            xValueAtMaxYValue = DFT[indexAtMaxY];
            ArrayofMaximums[i] = xValueAtMaxYValue; //Array that holds all
maximum values
            ArrayofMaxIndicies[i] = indexAtMaxY; // Array that holds the
indexes of all maximum values
        }
    }
    int FrequencyIndexAtMax =0;
```

```
    for (int i =0; i<NumberofNotesCompared; i++){ //Takes the number of
maximums equivalent to value of NumberofNotesCompared
        indexAtMaxY= indexatmax(ArrayofMaximums,RawDatasize/DFTsegment-
Highfrequencyvalues+1);
        FrequencyIndexAtMax =ArrayofMaxIndicies[indexAtMaxY];
        if(FrequencyIndexAtMax <RawDatasize/2){
          if(f[FrequencyIndexAtMax] < 2000 && f[FrequencyIndexAtMax] >16 &&
ArrayofMaximums[indexAtMaxY] > 1000000000000.00){ //If frequency is in
accpetable range add it to chord frequencies
              PreLinkedChordFrequencies[i] =f[FrequencyIndexAtMax];
          }else{
              PreLinkedChordFrequencies[i] =0.0;
          }
      }else{
        PreLinkedChordFrequencies[i] =0;
        }
        ArrayofMaximums[indexAtMaxY] =0;
    }



    int arraysortsize =
sizeof(PreLinkedChordFrequencies)/sizeof(PreLinkedChordFrequencies[0]);
    sort(PreLinkedChordFrequencies, PreLinkedChordFrequencies
+arraysortsize); //Sorts chord frequencies from least to greatest
    //This sort is done because lower frequencies are given priority as the
lowest frequency is always the root of the chord

// Removes all zeros from chord frequency list and then shortens array to
save on processing time and memory
//Linked list portion
    double *ChordFrequencies;
    ChordFrequencies = new double[linkedlistsize];

    for(int i=0; i<linkedlistsize;i++){
        ChordFrequencies[i] = PreLinkedChordFrequencies[i];
    }

    cout << "ChordFrequencies" << endl;
    for (int i=0; i <NumberofNotesCompared; i++){
        cout << ChordFrequencies[i] << " "; //Print chord frequencies
    }

    int zerocounter=0;

    for(int i=0; i <linkedlistsize; i++){
        if(ChordFrequencies[i] ==0){
            zerocounter++;
        }
    }
    cout << endl << endl;

    for(int i=0; i<linkedlistsize; i++){
```

```
        if(i+zerocounter <linkedlistsize){
            ChordFrequencies[i] = ChordFrequencies[i+zerocounter];
        }else{
            ChordFrequencies[i] =0;
        }
    }

    if (linkedlistsize -zerocounter >=3){
        linkedlistcount = linkedlistsize -zerocounter;
    }else{
        linkedlistcount =3;
    }

    cout << "There are " << linkedlistcount << " viable frequencies" << endl
<< endl;


    double *nv;
    nv = new double[linkedlistcount];
    for(int i=0; i<linkedlistcount; i++){
        nv[i] = ChordFrequencies[i]; //New array filled with non zero
frequency entries
    }
    delete[] ChordFrequencies;
    ChordFrequencies = nv;

    const int PostLinkedlistNumberofNotesCompared = linkedlistcount;



    cout << "ChordFrequencies" << endl;
    for (int i=0; i <PostLinkedlistNumberofNotesCompared; i++){
        cout << ChordFrequencies[i] << " "; //Print chord frequencies
    }

    cout << "\n" << "\n";




    //create all values for all notes on piano
    double Afrequencies[7];
    double BflatAsharpfrequencies[7];
    double Bfrequencies[7];
    double Cfrequencies[7];
    double DflatCsharpfrequencies[7];
    double Dfrequencies[7];
    double EflatDsharpfrequencies[7];
    double Efrequencies[7];
    double Ffrequencies[7];
    double GflatFsharpfrequencies[7];
    double Gfrequencies[7];
    double AflatGsharpfrequencies[7];

    double note=0.0;
```

```
//Each row below is each note halfstep calculated for 7 octaves
    for (int i=0; i<=6; i++){
        inttodouble = i;
        note = 440*pow(2,(((1+(12*inttodouble))-49)/12));//Formula to
determine value of A
        Afrequencies[i]= note;
        note = 440*pow(2,(((2+(12*inttodouble))-49)/12));
        BflatAsharpfrequencies[i]= note;
        note = 440*pow(2,(((3+(12*inttodouble))-49)/12));
        Bfrequencies[i]= note;
        note = 440*pow(2,(((4+(12*inttodouble))-49)/12));
        Cfrequencies[i]= note;
        note = 440*pow(2,(((5+(12*inttodouble))-49)/12));
        DflatCsharpfrequencies[i]= note;
        note = 440*pow(2,(((6+(12*inttodouble))-49)/12));
        Dfrequencies[i]= note;
        note = 440*pow(2,(((7+(12*inttodouble))-49)/12));
        EflatDsharpfrequencies[i]= note;
        note = 440*pow(2,(((8+(12*inttodouble))-49)/12));
        Efrequencies[i]= note;
        note = 440*pow(2,(((9+(12*inttodouble))-49)/12));
        Ffrequencies[i]= note;
        note = 440*pow(2,(((10+(12*inttodouble))-49)/12));
        GflatFsharpfrequencies[i]= note;
        note = 440*pow(2,(((11+(12*inttodouble))-49)/12));
        Gfrequencies[i]= note;
        note = 440*pow(2,(((12+(12*inttodouble))-49)/12));
        AflatGsharpfrequencies[i]= note;
    }
    inttodouble=0;
    double NoteError=0;
    string ChordNotes[PostLinkedlistNumberofNotesCompared];

    for(int i =0; i<PostLinkedlistNumberofNotesCompared;i++){//Determines
acceptable note error for each note
        //Acceptable note error increases with frequnecy as higher notes are
farther apart frequency wise
        NoteError =0;
        if (ChordFrequencies[i] > 0 && ChordFrequencies[i]< 63) {
//Acceptable error of pitch at each octave in Hz
            NoteError = 2;
        }else if (ChordFrequencies[i] > 63 && ChordFrequencies[i]< 127) {
            NoteError = 4;
        }else if (ChordFrequencies[i] > 127 && ChordFrequencies[i]< 255) {
            NoteError = 8;
        }else if (ChordFrequencies[i] > 255 && ChordFrequencies[i]< 511) {
            NoteError = 16;
        }else if (ChordFrequencies[i] > 511 && ChordFrequencies[i]< 1023) {
            NoteError = 32;
        }else if (ChordFrequencies[i] > 1023 && ChordFrequencies[i]< 2047) {
            NoteError = 64;
        }else if (ChordFrequencies[i] > 2047 && ChordFrequencies[i]< 10000) {
            NoteError = 128;
        }
```

```
        for (int j=0; j<=6; j++){ //Determines what note each frequency is using note error
            if ((Afrequencies[j]-NoteError <ChordFrequencies[i] && Afrequencies[j]+NoteError >ChordFrequencies[i]) && (abs(Afrequencies[j]-ChordFrequencies[i])< abs(BflatAsharpfrequencies[j]-ChordFrequencies[i]) && abs(Afrequencies[j]-ChordFrequencies[i])< abs(AflatGsharpfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="A";
            }
            if ((BflatAsharpfrequencies[j]-NoteError <ChordFrequencies[i] && BflatAsharpfrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(BflatAsharpfrequencies[j]-ChordFrequencies[i])< abs(Bfrequencies[j]-ChordFrequencies[i]) && abs(BflatAsharpfrequencies[j]-ChordFrequencies[i])< abs(Afrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="Bflat/Asharp";
            }
            if ((Bfrequencies[j]-NoteError <ChordFrequencies[i] && Bfrequencies[j]+NoteError >ChordFrequencies[i]) && (abs(Bfrequencies[j]-ChordFrequencies[i])< abs(Cfrequencies[j]-ChordFrequencies[i]) && abs(Bfrequencies[j]-ChordFrequencies[i])< abs(BflatAsharpfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="B";
            }
            if ((Cfrequencies[j]-NoteError <ChordFrequencies[i] && Cfrequencies[j]+NoteError >ChordFrequencies[i]) && (abs(Cfrequencies[j]-ChordFrequencies[i])< abs(DflatCsharpfrequencies[j]-ChordFrequencies[i]) && abs(Cfrequencies[j]-ChordFrequencies[i])< abs(Bfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="C";
            }
            if ((DflatCsharpfrequencies[j]-NoteError <ChordFrequencies[i] && DflatCsharpfrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(DflatCsharpfrequencies[j]-ChordFrequencies[i])< abs(Dfrequencies[j]-ChordFrequencies[i]) && abs(DflatCsharpfrequencies[j]-ChordFrequencies[i])< abs(Cfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="Dflat/Csharp";
            }
            if ((Dfrequencies[j]-NoteError <ChordFrequencies[i] && Dfrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Dfrequencies[j]-ChordFrequencies[i])< abs(EflatDsharpfrequencies[j]-ChordFrequencies[i]) && abs(Dfrequencies[j]-ChordFrequencies[i])< abs(DflatCsharpfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="D";
            }
            if ((EflatDsharpfrequencies[j]-NoteError <ChordFrequencies[i] && EflatDsharpfrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(EflatDsharpfrequencies[j]-ChordFrequencies[i])< abs(Efrequencies[j]-ChordFrequencies[i]) && abs(EflatDsharpfrequencies[j]-ChordFrequencies[i])< abs(Dfrequencies[j]-ChordFrequencies[i])))){
                ChordNotes[i]="Eflat/Dsharp";
            }
            if ((Efrequencies[j]-NoteError <ChordFrequencies[i] && Efrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Efrequencies[j]-ChordFrequencies[i])< abs(Ffrequencies[j]-ChordFrequencies[i]) && abs(Efrequencies[j]-ChordFrequencies[i])< abs(EflatDsharpfrequencies[j]-
```

```
ChordFrequencies[i]))){
                ChordNotes[i]="E";
            }
            if ((Ffrequencies[j]-NoteError <ChordFrequencies[i] &&
Ffrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Ffrequencies[j]-
ChordFrequencies[i])< abs(GflatFsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Ffrequencies[j]-ChordFrequencies[i])< abs(Efrequencies[j]-
ChordFrequencies[i]))){
                ChordNotes[i]="F";
            }
            if ((GflatFsharpfrequencies[j]-NoteError <ChordFrequencies[i] &&
GflatFsharpfrequencies[j]+NoteError >ChordFrequencies[i])&&
(abs(GflatFsharpfrequencies[j]-ChordFrequencies[i])< abs(Gfrequencies[j]-
ChordFrequencies[i]) && abs(GflatFsharpfrequencies[j]-ChordFrequencies[i])<
abs(Ffrequencies[j]-ChordFrequencies[i]))){
                ChordNotes[i]="Gflat/Fsharp";
            }
            if ((Gfrequencies[j]-NoteError <ChordFrequencies[i] &&
Gfrequencies[j]+NoteError >ChordFrequencies[i])&& (abs(Gfrequencies[j]-
ChordFrequencies[i])< abs(AflatGsharpfrequencies[j]-ChordFrequencies[i]) &&
abs(Gfrequencies[j]-ChordFrequencies[i])< abs(GflatFsharpfrequencies[j]-
ChordFrequencies[i]))){
                ChordNotes[i]="G";
            }
            if ((AflatGsharpfrequencies[j]-NoteError <ChordFrequencies[i] &&
AflatGsharpfrequencies[j]+NoteError >ChordFrequencies[i])&&
(abs(AflatGsharpfrequencies[j]-ChordFrequencies[i])< abs(Afrequencies[j]-
ChordFrequencies[i]) && abs(AflatGsharpfrequencies[j]-ChordFrequencies[i])<
abs(Gfrequencies[j]-ChordFrequencies[i]))) {
                ChordNotes[i] = "Aflat/Gsharp";
            }
            if(ChordFrequencies[i] < 16){
                ChordNotes[i] = "NaN";
            }
        }
    }



/* This portion of the code below from here until the definition of chords is
the improvement on the prior method to determine the chord
 * The number of times the note appears is now taken into account
 * The more the note appears the higher priority it has and is then placed
higher up in the chord
 * The 3 top notes contain the chord
*/

    delete[] ChordFrequencies;
    int ChordNoteNumberSize = sizeof(ChordNotes) / sizeof(ChordNotes[0]);

    bool check[ChordNoteNumberSize];
    int counterarray[ChordNoteNumberSize];
    for(int i= 0; i <ChordNoteNumberSize; i++){
        counterarray[i] =0;
    }
```

```
    string UniqueChordNotes[ChordNoteNumberSize];
    for(int i= 0; i <ChordNoteNumberSize; i++){//Makes an empty array
        UniqueChordNotes[i] = " ? ";
    }

    for(int i=0;i<ChordNoteNumberSize;i++){
        check[i] = 0;
    }
    for(int i=0; i<ChordNoteNumberSize; i++){//Determines the number of times
each note appears
        if(check[i]== 1){
            continue;
        }
        int count = 1;

        for(int j = i+1; j<ChordNoteNumberSize; j++){
            if (ChordNotes[i].compare(ChordNotes[j]) == 0){
                check[j] = 1;
                count++;
            }
        }
        counterarray[i] = count; //Count of each note is filled into this
array
        UniqueChordNotes[i] = ChordNotes[i];

        cout<<ChordNotes[i]<<" appears: " << count << " time(s)"<< endl;
    }
    cout << endl << endl;

    int comparisoncounterarray[ChordNoteNumberSize];
    for(int i =0; i <ChordNoteNumberSize; i++){
        comparisoncounterarray[i] = counterarray[i];
    }
    sort(counterarray, counterarray + ChordNoteNumberSize, greater<int>());
//Count array is sorted from least to greates

    string SortedUniqueChordNotes[ChordNoteNumberSize];

    for(int i=0; i<ChordNoteNumberSize; i++){ //Using the sorted count array
the note that lines up with the number of appearence is brought to the top
        for (int j=0; j<ChordNoteNumberSize; j++){
            if(counterarray[i] == comparisoncounterarray[j] &&
UniqueChordNotes[j].compare(" ? ") != 0 && UniqueChordNotes[j].compare("NaN")
!= 0 && counterarray[i] !=0){
                comparisoncounterarray[j] =0;
                SortedUniqueChordNotes[i] = UniqueChordNotes[j];
                break;
            }
        }

    }
```

```
    for (int i=0; i <3; i++){
        ChordNotes[i] = SortedUniqueChordNotes[i]; //Chord notes are now
sorted by rate of appearence
    }

    cout << "\n" << "\n";
//Definition of all chords
    string ListofChords[24] = {"A major", "A minor", "Bflat/Asharp major",
"Bflat/Asharp minor", "B major", "B minor","C major", "C minor","Dflat/Csharp
major", "Dflat/Csharp minor","D major", "D minor","Eflat/Dsharp major",
"Eflat/Dsharp minor","E major", "E minor","F major", "F minor","Gflat/Fsharp
major", "Gflat/Fsharp minor","G major", "G minor","Aflat/Gsharp major",
"Aflat/Gsharp minor"};
    string Amajorcomponents[3] = {"A","Dflat/Csharp","E"};
    string Aminorcomponents[3] = {"A", "C", "E"};
    string BflatAsharpmajorcomponents[3] = {"Bflat/Asharp", "D", "F"};
    string BflatAsharpminorcomponents[3] = {"Bflat/Asharp", "Dflat/Csharp",
"F"};
    string Bmajorcomponents[3] = {"B", "Eflat/Dsharp", "Gflat/Fsharp"};
    string Bminorcomponents[3] = {"B", "D", "Gflat/Fsharp"};
    string Cmajorcomponents[3] = {"C", "E", "G"};
    string Cminorcomponents[3] = {"C", "Eflat/Dsharp", "G"};
    string DflatCsharpmajorcomponents[3] = {"Dflat/Csharp", "F",
"Aflat/Gsharp"};
    string DflatCsharpminorcomponents[3] = {"Dflat/Csharp", "E",
"Aflat/Gsharp"};
    string Dmajorcomponents[3] = {"D", "Gflat/Fsharp", "A"};
    string Dminorcomponents[3] = {"D", "F", "A"};
    string EflatDsharpmajorcomponents[3] = {"Eflat/Dsharp", "G",
"Bflat/Asharp"};
    string EflatDsharpminorcomponents[3] = {"Eflat/Dsharp", "Gflat/Fsharp",
"Bflat/Asharp"};
    string Emajorcomponents[3] = {"E", "Aflat/Gsharp", "B"};
    string Eminorcomponents[3] = {"E", "G", "B"};
    string Fmajorcomponents[3] = {"F", "A", "C"};
    string Fminorcomponents[3] = {"F", "Aflat/Gsharp", "C"};
    string GflatFsharpmajorcomponents[3] = {"Gflat/Fsharp", "Bflat/Asharp",
"Dflat/Csharp"};
    string GflatFsharpminorcomponents[3] = {"Gflat/Fsharp", "A",
"Dflat/Csharp"};
    string Gmajorcomponents[3] = {"G", "B", "D"};
    string Gminorcomponents[3] = {"G", "Bflat/Asharp", "D"};
    string AflatGsharpmajorcomponents[3] = {"Aflat/Gsharp", "C",
"Eflat/Dsharp"};
    string AflatGsharpminorcomponents[3] = {"Aflat/Gsharp", "B",
"Eflat/Dsharp"};

    string ChordComponents[24][3]; //2D array that holds all chord componets
for all 24 maj/min chords
    for (int j=0; j<3; j++){
        ChordComponents[0][j]=Amajorcomponents[j];
        ChordComponents[1][j]=Aminorcomponents[j];
        ChordComponents[2][j]=BflatAsharpmajorcomponents[j];
        ChordComponents[3][j]=BflatAsharpminorcomponents[j];
```

```
        ChordComponents[4][j]=Bmajorcomponents[j];
        ChordComponents[5][j]=Bminorcomponents[j];
        ChordComponents[6][j]=Cmajorcomponents[j];
        ChordComponents[7][j]=Cminorcomponents[j];
        ChordComponents[8][j]=DflatCsharpmajorcomponents[j];
        ChordComponents[9][j]=DflatCsharpminorcomponents[j];
        ChordComponents[10][j]=Dmajorcomponents[j];
        ChordComponents[11][j]=Dminorcomponents[j];
        ChordComponents[12][j]=EflatDsharpmajorcomponents[j];
        ChordComponents[13][j]=EflatDsharpminorcomponents[j];
        ChordComponents[14][j]=Emajorcomponents[j];
        ChordComponents[15][j]=Eminorcomponents[j];
        ChordComponents[16][j]=Fmajorcomponents[j];
        ChordComponents[17][j]=Fminorcomponents[j];
        ChordComponents[18][j]=GflatFsharpmajorcomponents[j];
        ChordComponents[19][j]=GflatFsharpminorcomponents[j];
        ChordComponents[20][j]=Gmajorcomponents[j];
        ChordComponents[21][j]=Gminorcomponents[j];
        ChordComponents[22][j]=AflatGsharpmajorcomponents[j];
        ChordComponents[23][j]=AflatGsharpminorcomponents[j];
    }
    string comparisonchord[3];
    for(int i=0;i<3;i++){
        comparisonchord[i] = "Q";
    }

    for (int i=0; i<3; i++){
        comparisonchord[i] = ChordNotes[i];//First 3 notes from ChordNotes is
used to compare to list
    }


//This part was removed as the sort by frequency removed the need to prevent
duplicates from appearing in chords

    /*for(int j=0; j<5; j++){
            if (comparisonchord[1] == comparisonchord[2] ||
comparisonchord[1] == comparisonchord[0]){
                if (comparisonchord[1] !=ChordNotes[j] && ChordNotes[j]
!=comparisonchord[0] && ChordNotes[j] !=comparisonchord[2]){
                    comparisonchord[2] = ChordNotes[j];
                }
            }
        if (comparisonchord[2] == comparisonchord[0] || comparisonchord[2] ==
comparisonchord[1]){
            if (comparisonchord[2] !=ChordNotes[j] && ChordNotes[j]
!=comparisonchord[0] && ChordNotes[j] !=comparisonchord[1]){
                comparisonchord[2] = ChordNotes[j];
            }
        }
    }*/

    cout << "comparisonchord" << endl;
    for(int j=0; j<=2;j++){
        cout << comparisonchord[j] << " ";
```

```
    }
    cout << "\n" << "\n";

    int isthisthecorrectchord=0;
    string CorrectlyIdentifiedChord ="Unidentified Chord";
    for(int i=0; i<24; i++){
        isthisthecorrectchord= 0;
        for(int j=0; j<3; j++){ //Checks comparison chord with all possible 
chords
            if(comparisonchord[j]== ChordComponents[i][0] ||
comparisonchord[j]== ChordComponents[i][1] || comparisonchord[j]==
ChordComponents[i][2]){
                isthisthecorrectchord =isthisthecorrectchord+1;
            }else{
                isthisthecorrectchord= 0;
            }
        }
        if(isthisthecorrectchord ==3){//If all three notes align with a chord 
then the chord is identified
            CorrectlyIdentifiedChord=ListofChords[i];
        }
    }

    cout << "Identified Chord" << endl;
    return CorrectlyIdentifiedChord;
}
```

References

[1] https://www.mathworks.com/matlabcentral/answers/544352-wav-file-dft-without-fft

[2] https://www.geeksforgeeks.org/frequency-of-a-string-in-an-array-of-strings/