Phil Ngo
Tyler Clites
Peter Olson

# Design Decisions

Contents:



## Overview:

Here are a few thoughts outlining some of the constraints we were working under before we jump into the details of our design decisions. We needed a reliable Fast Fourier Transform to perform a Discrete Fourier Transform on our audio data. We decided not to spend time learning the mathematics that would allow us to write our own FFT, and chose instead to use a reliable 3rd party FFT so that we could focus on data interpretation. We ended up using an FFT written in C, distributed as part of the Gnu Science Library. This tied us to C, which was fine, because we didn't really need to be web activated for full functionality; in fact, we have greater speed and flexibility in C because we have direct access to the host computer. Our C code actually depends on 2 third-party libraries – GSL and LibXML2, for the FFT code and XML parsing code, respectively. However, we found that a C-based GUI would be significantly more difficult to write than Java-based GUI, so we decided to write a Java GUI and execute our C code from a Java.

Furthermore, we found that our C code for processing harmonies and reading wav files and writing music xml was getting quite large, so we decided to make it into a library, which we called "musicxml." Gaining access to the functions therein is now as simple as putting a *#include* at the top of a .c file with a "main" function. This is how we implemented the "import" program, which imports a wav file, processes it, harmonizes it, and writes it to an XML file.

## Usability and Customizability – Java/C interfacing

The Java GUI was designed in Netbeans to help the user configure the settings for harmonization before submitting WAV files to the C code for processing. Java runs a process started by the method Runtime.getRuntime().exec("…"), which executes the command in quotes. This is where we passed the arguments to the C program, telling it where to find the input file, what the filename of the output should be, how to harmonize, and other useful information.

Netbeans is a convenient Integrated Development Environment (IDE) for several reasons: First of all, it makes arranging the GUI components on the page much easier. Coding layout specifics is a nightmare without a GUI editor. Second, it recognizes when variables or libraries are created or imported and not used, and recommends removing them; it identifies when methods are called without the necessary libraries' having been inputted, and suggests the import files. "Smart" features of this sort proved invaluable, especially given that we had very prior experience in Java and were learning as we went.

Java was a helpful language to use because its being object-oriented makes organizing GUIs very intuitive. Here, we created a jFrame, and placed four panels on that. The "Select File" button took a little bit of work, because we wanted to restrict the users from selecting anything other than WAV files, but we found that setting up file filters for the jFileChooser we used in the "Select File" button meant that every time the button was clicked, an additional filter was added. To correct for this, we inserted code to reset the file filter settings and rebuild them every time the "Select File" button is clicked. The "Play" button was not overly difficult to code—There were examples of JAVA audio players online, so all we had to do was remove some of the bells and whistles from those.

"Click to the Beat" took a little bit of thought—We wanted to have a moving average of the BPM, but we didn't want someone to start clicking the button, pause for 10 seconds, and then click again, only to ruin that average. We decided to fix that problem by only incorporating distances of time less than 200 BPM, since BPM is the highest value that could be submitted to begin with.

Working with the "Time Signature," "Original Key," etc., drop-down boxes took a little while to figure out, but once we gained a better understanding of how combo boxes functioned, this was not hard. Netbeans makes it easy to change the settings for drop-down boxes.

At first, we considered only accepting strings without spaces in the "File Name" and "Title" text boxes, because passing those to the command line for our C code to input would be difficult for the C code (we would have to compensate for variation in the number of arguments passed, etc.), but we decided in the end to replace the spaces in those strings with underscores as part of the GUI code, and put the spaces back with our C code.

We decided to create a slew of error messages that should cover the full panoply of possible problems with the GUI input specifications, and also request a response from the System to inform the user that the harmonies were properly generated by the C code.

**Algorithms for Audio Processing**

We use a Fast Fourier Transform to extract frequency information for each note that we detect in a piano melody recorded in a WAV file. This transform takes n discrete points in the time domain and gives n discrete frequency domain features. Our program analyzes these features and stores the information contained therein. The following a more specific explanation of each of the program's functions, along with the rationale that governed their design.

**Part* read(char* wavfile, int bpm, int divspermeasure)**

The read function takes as input a string containing the path to the desired .wav file, an integer representing the number of beats per minute, and an integer representing the number of divisions per measure (governed by the time signature). It begins by opening the WAVE file and reading the file header, using a function called openWavFile to store the information from the file header into a struct called info, which contains vital information for use in later processing. Once the information is loaded, the function performs a quick check for compatibility, since the current version only supports 16-bit samples. In future versions we hope to include the capability to process other file types, but for now the function returns NULL if the file type is not supported. The function then uses arithmetic to determine how many averages will be taken, creates an array of appropriate size, and calls the findAvgs function, which returns an array containing the average of the absolute value of every [AVG_WINDOW] samples of the data from the .wav file. This provides an "envelope" of sorts, which traces the magnitude of the signal from the .wav file. The array of averages is then passed to the diff function, which uses simple arithmetic to find the discrete derivative of the averages, and returns a pointer (differences) to those values. A threshold is set at [THRESHOLD_FACTOR] times the maximum value of the differences. This threshold will be used to determine the start of each note.

Once the threshold has been set, read begins to assemble a linked list of the sequence of notes and their durations. The function begins by iterating through the differences until it finds a value that is greater than the threshold. If said value is the first, the head pointer is set to point to a Part in the list. All subsequent notes are added to the end of the linked list, which was a design decision made in order to preserve the order of the list. The "start" value is then set to the sample number of the beginning of the note. The function then skips ahead the length of a 16th note, to avoid double-counting, and then continues to search for the next such instance of a super-threshold difference. Once such an instance is found, the note_length values is set to contain total number of samples in the note. The data arrays are then reallocated as necessary, if the number of samples to be stored is greater than the size of the array. This allows us to allocate only the memory necessary to store the longest note in the piece. It is important to note that we currently store data from both channels (if two channels are present in the .wav file), but that only one of these channels is analyzed. This decision was made so that future versions could add the ability to process the second channel and cross-check it with the first to improve accuracy. The file position indication then seeks to the beginning of the note, and makeWindow is called to store the appropriate data into the data arrays. Once that data is stored the analyzeData does most of the "heavy lifting" and processes the data information, eventually resetting the data arrays to zero and returning an integer representing the note value, which is stored in the appropriate Part. If this number is -1, indicating an error, read returns NULL. If it is 0, the segment is considered noise, and the start indicator is reset so that the value

will be overwritten. If it is a valid note, the duration is calculated and stored to the Part. If the duration is greater than zero, indicating that the segment was not simply a blip, the Part is completed, and a new Part is created and added to the end of the linked list. The loop then jumps the length of a sixteenth note and continues searching.

Once the loop reaches the end of the differences array, the last note is added to the end of the linked list, by a process similar to that described above. The only difference is that the duration is cut in order to ensure that the printed piece contains only enough total durations to complete an integer number of measures. The memory is then freed and the files are closed, and read returns the head of the linked list.

**int findAvgs(wavFileInfo* info, double avg[], int num_avg)**

findAvgs takes as input a pointer to the struct containing the information from the WAVE fileheader, a pointer to the array in which the averages will be stored, and an integer value representing the total number of averages. The function begins by initializing the avg array to zeros. It then enters a loop in which it reads the data from the .wav file, sums the absolute values of each sample from the left channel [AVG_WINDOW] samples at a time , and stores the average into the avg array. It then returns once it reaches the end of the file.

**int makeWindow(wavFileInfo* info, double* data_left, double* data_right, int note_length)**

makeWindow takes as input the struct containing the information from the WAVE file header, pointers to the two data arrays, and an integer representing the number of samples in the note. The function then populates the data array, first reading each sample into the buffer as an int16_t and then casting it to a double and storing it. It is important to note that it is incredibly improbable for makeWindow to completely fill the array, which is why the array is initialized to contain all zeros, and on each iteration of the analyzeData it is reset to contain all zeros. The array is padded with zeros because values of zero do not affect the FFT output. It is important, however, that the array size be a power of two, so that the output of the FFT is cleaner and more precise.

**int analyzeData(double data[], FILE* out, wavFileInfo* info, int current_size)**

analyzeData takes as input an array containing the data from the current note, the struct containing the WAVE file header info, and an int representing the current size of the data array. The function first runs the gsl FFT, which overwrites the data in the array with the FFT output. It then searches the array for the maximum value, which represents the "strongest" frequency in the given segment. This is taken to represent the diatonic note of the segment, but because of overtone interference, more processing in required to determine the exact pitch. The maximum value is used to

calculate the frequency of the note, which is then mapped to a key on the piano, and stored as max_key. Again, because of overtones, this is still not enough information to determine the exact pitch or key on the piano. In plainer terms, often the FFT returns the correct note an octave too high as the strongest frequency (eg. C5 was played but C6 had the strongest frequency). This problem is solved using histogram analysis, as described below.

To prepare for the histogram analysis, the function first creates the histogram's bins by calculating bin ranges using a formula that maps frequencies to piano keys, and then storing them in an array called ranges. analyzeData then creates the histogram using the gsl histogram function. Once the histogram has been created, the function iterates over the data array to find the top thirty "strongest" frequencies and increments the histogram bins that contain those frequencies. Those top thirty are then printed to a file called visual.txt, which eventually gives a visual representation of the FFT data. analyzeData then calls findClumps, which is designed to use the histogram data to find the correct pitch, moving either up or down by octaves. It is described in detail below, but its primary function is to search the histogram for "clumps" of data around the correct note. In this version, a clump is defined as one strong frequency in the correct range, but the function is still helpful in case the piano is slightly out of tune or the FFT returns values near the boundary of two different notes.

Once the correct pitch is analyzed, the histogram is printed to visual.txt for further analysis. This is no longer necessary in the finished product, but it was incredibly useful in designing the program, and it may prove useful in understanding how the FFT works. analyzeData then frees the histogram, clears the data arrays (sets them to zeros) and returns.

**int openWavFile(wavFileInfo* info)**

openWavFile takes as input a pointer to the struct that will eventually contain the WAVE file header information. The function reads the .wav file header 4 bits at a time, storing the bits into a buffer array called chunk. Each value in the header is read in using bit-shifting techniques (many are stored as little-endian), checked for validity, and then stored into the wavFileInfo* struct.

**int findClumps(gsl_histogram* h, int max_key)**

findClumps takes as input a pointer to the histogram created in analyzeData and an integer representing the "strongest" note picked up by the FFT. The function iterates through each octave of the given note, searching the two bins around it for a "clump" until it finds one. With pure notes, such as those from a piano, it is acceptable to define a clump as a single strong frequency because all of the overtones come at frequencies higher than the note that was actually played. When analyzing more convoluted signals, it may be appropriate to more strictly define a clump as containing more than one strong frequency, which is why the capability is still included

in the program. The size variable keeps track of the size of the clump. If size is greater than the threshold value, the function returns the value that needs to be added to max_key_number. If findClumps is unable to find any notes that fit the above criteria, the function returns 0 and the max_key_number remains unchanged.

**double* diff(double data[], int n)**

diff takes as input an array of data and its size n, and iterates through it subtracting each entry from the next, which gives the discrete derivative of the input function. It returns a pointer to an array containing the derivative values.

**double max(double data[], int n)**

max takes as input an array of data and its size n, and iterates through until it finds the maximum value in the array. It then returns the max value.

**int powerOfTwo(int v)**

powerOfTwo is taken from the bit-twiddling hacks website, and it uses bitwise functions to find the next highest powerOfTwo of the input int v.

## 3) Data structures for internal and external representation of musical notes and harmonies

We use linked lists to sequentially store information about the notes in a melody of arbitrary length. We create this list as it is read in from raw WAV data. Each node stores information about the frequency of the note, its duration, how it should be annotated, what it should sound like as its played back and a pointer to the next node. Similarly, we have specialized linked lists for harmonic patterns and rhythmic patterns, which work in tandem to fully describe a piece of music. The lists are short and most operations on them (transposition, harmonization) are sequential, so we have no need for random access. We eventually traverse these linked lists and translate them node by node into musicxml, which creates a tree structure containing all of the necessary information for the third-party notation editor to read. This XML file includes header information, such as the title, the composer, the number of parts in the piece, branches for each measure, and many notes within each measure.

Other information about harmony types (i.e. Major and minor triads, $7^{th}$ chords, scales, so on) is stored in other useful structs for organization. The purpose of these structs is primarily to standardize information transfer between functions, keeping it fairly lightweight and organized.

## 4) Algorithms for Harmonization

One of the most interesting problems that needed to be solved for this project to be effective was to determine an algorithm harmonizing arbitrary

melodies. The algorithm we use allows some flexibility in choice of harmonic rhythm (i.e. how often the base chord changes), key, number of parts, and other variables such as the range of the notes used, the staff on which these notes are displayed, etc. Some of the more technical options have not been made accessible to the user in the GUI, but can easily be added to the GUI later or configured manually by the programmer. The algorithm uses a number of custom structures to store information. It uses primarily Parts, Rhythms, and Harmonies, which are all implemented as linked lists. The harmony algorthim has the following iterative structure:

1) Traverse a linked list, called a "Part", which contains a melody stored note by note (node by node), and compile a list of chords which contain the melody note.
2) Traverse this list of allowed chords, but this time, group them by harmonic rhythm (which is, in turn, stored in a structure called a "Rhythm") and apply a weight to them depending on whether or not they come at the same time as a rhythmically emphasized note, such as a downbeat.
3) Traverse this list of weighted chords, select a few of the most highly weighted chords, and choose randomly from among them, creating a "Harmony" linked list with the harmonic rhythm provided. At this point, no specific harmonizing notes have been chosen, only the chords from which they will be chosen in the next step.
4) For each part desired, traverse this Harmony and, for each node in the target Rhythm (which can be different from the one chosen before – not available to the GUI) pick a note which falls within the allowed range. The range of allowed notes is determined by the Harmony and by a range of notes specified by the programmer.

This basic algorithm is used to harmonize arbitrary melodies. It is highly dependent on the quality of the melody that it recieves. Melodies with strange harmonic patterns and rhythms, particularly those which do not fall into traditional modes and scales, sound quite foreign to our ears, and attempts to harmonize these certainly don't bring them any closer to home. For this reason, the GUI requires the user to contribute some helpful information about the algorithm. For example, the GUI requires the user to specify the key of the music and the beats/min (BPM) of the recording. To make this easier on them, we developed a small tool to help estimate the speed of the piece in beats per minute, a standard unit of measure for music, which helps keep the musical notation accurate: as the user plays back the file they uploaded, they tap a button to the beat of the music, and the program suggests an appropriate bpm. Within the program, note durations are stored in units called divisions. A division is the smallest possible duration of a note. In our case, we decided arbitrarily to set 1 quarter note equal to 96 divisions, which is useful because it is 2^5 * 3, which allows a great number of notes to be represented. The divisions unit comes directly from Music XML and is used when transcribing notes.

Phil's Functions for Harmony Writing:

Part* addPickup(Part* part, int beats, int meter);

Adds rest at the beginning of the part to make a pickup of the given # of beats to a measure of the given meter.

Returns a pointer to the head node of the new Part.

int allowedChordProgression(int first, int second);

Checks to see if the second diatonic chord generally follows the first chord in harmonic progressions. first, second = 0 to 6.

Returns 1 if it does, 0 if it doesn't.

int alterOf(char* note, int key);

Checks to see if the note (note = 1 to 88) falls within the given key and returns its accidental if not.

Returns -1 if flat, 1 if sharp, 0 otherwise.

Rhythm* copyPartRhythm(Part* melody);

Returns a Rhythm with the same rhythm as the Part

int* determineHarmonicScaleDegrees(Part* part, int key);

This is used in step (1) of the algorithm described above.

Returns a pointer to an array of scale degree chords that contains each note in the part and NULL on error.

Harmony* determineHarmony(Part* part_head, Rhythm* rhythm_head, int key, int beats);

This incorporates steps (1) to (3) of the algorithm above. The provided harmonic Rhythm must be longer than the provided melody Part.

Returns a pointer to the first node of a Harmony or NULL on error

Part* getCounterpointPart(Harmony* harmony, Rhythm* rhythm, Part* other_parts[], int num_parts, int key, int staff);

This is used in setp (4) of the algorithm. Writes a counterpoint part given a Harmony, a Rhythm and some other parts.

Returns a pointer to the head node of that

Note getNote(int piano_key, int key_sig);

This is a helper function that takes a piano key (1 to 88) and a key signature (-7 to 7) and returns a struct with these values:

.octave = octave of the key # (middle C starts octave 4)

.alter is sharp = 1, natural = 0, flat = -1 depends on the provided key signature

.step is the alphabetical note name A = 0, B = 1 ...

.duration is the duration in whatever units are provided (quarter note, usually)

int getNotes(int notes[], int alters[], Section section, int key);

Fills the notes and alters arrays according to the properties supplied in section/key. Section is an alternative storage paradigm which complements the Rhythm and Part structures, and carries different types of information.

Returns 0 on success, 1 otherwise.

Part* getPart(const char* filename, int key);

Useful for prototyping and for greater storage efficiency.
Imports a melody from a text file in a csv with two columns:
1) key number (1-88)
2) duration

Range getRange(int function, int type_id, int key);
Returns a struct wich allowed notes according to the type of harmony provided

Rhythm* getRhythm(int divisions, int beats, int style);
Returns the head pointer to a Rhythm with given specs

Type getType(int id);
Returns a struct with information about the type
0: Major – major 3rd, minor 3rd
1: Minor – minor 3rd , major 3rd
2: … more

int isInRange(Range range, int note_num);
A bit of encapsulation for the Range: see if a note is in Range
Returns 1 if the note is in range, 0 if not

int rmHarmony(Harmony* head);
int rmRhythm(Rhythm* head);
int rmPart(Part* head);
These free the memory associated with the Harmony/Rhythm/Part pointed to by head.
Returns 0 on success.

int transpose(Part* part, int old_key, int new_key, int shift_direction);
Transposes the part from the old key to the new key in the specified direction (1 = up, -1 = down).
Returns the new key

Phil's Functions for XML writing:

xmlNodePtr writeHeader(xmlDocPtr doc, char* composer, char* title);
Writes a file header for a music xml file.
Returns an xmlNodePtr (a data type from the libxml2 library) which points to the xml node representing a single musical part, which will have a number of child measures.

int writeMeasureAttributes(xmlNodePtr measure, MeasureAttribute measure_attributes);
Writes a measure attributes such as meter and key signature at the beginning of each measure.
Returns 0 on success.

int writeNote(xmlNodePtr measure, Note note, int num_divs, int tie_type, int beam_pos, int chord, int staff, int numeral, int type, int rest);
Writes a single note in xml according to specs in the arguments
Returns 0 on success.

int writePart(const char* filename, Part* part[], int num_parts, int beat, int key, char* composer, char* title);
Writes an array of Parts to a xml file of the name specified.

Returns 0 on success.

## 5) Other functions

The other functions in the musicxml library will be used for various other tasks that a more advanced version of Harmonizer might offer. For instance, a user can specify an exact harmony and generate xml notation for a variety of arpeggios, chord patterns and rhythms. There is also a rudimentary function for generating a random harmony, random rhythms and random arpeggio patterns. These were used in testing and may also be used in future versions of Harmonizer. A list of these functions is below:

Phil's Other Utility Functions:

int createRandomInput(char* filename);

Create a totally random input file:

1) key is somewhere between -7 and 7 inclusive
2) chord progression starts on 1 and ends on 1
3) length is somewhere bewteen 5 and 40 sections

Returns 0 on success

int* determineMeter(Part* part);

Determine the meter and number of pickup beats of a part. Note: this function is still in beta. The algorithm doesn't always work. That's why we still have the user input.

Returns a a pointer to an array containing the meter and number of pickup beats

int generateArpeggioPattern(int pattern[], int length);

Generates a random arpeggio pattern of a certain length, filling the array with ints representing the pattern.

Returns 0 on succes.

int generateRhythmPattern(int rhythm[], int num_notes, int duration);

Generates a random rhythm with the number of notes specified and over the duration specified, filling the array rhythm with the generated data

Returns 0 on success.

Composition getUserInput(char* filename);

If the filename supplied is invalid, it prompts user for input from the command line. This allows users to be very specific about the harmony they would like generated.

Returns a struct with compostion attributes supplied by a user

int writeHarmony(const char* filename, Harmony* harmony_head, int key);

Writes a harmony to a file of the name specified

Returns 0 on success.

Phil's Other XML Functions:

int writeArpeggio(xmlNodePtr measure, Section section, int key, float note_dur);

Writes an arpeggio in the measure provided

int writeChord(xmlNodePtr measure, Section section, int key);

Writes a chord (should be the first in the sequence - no backup tag

int writeMelody(xmlNodePtr measure, Section sect, int key, float num);

Writes a melody in the measure provided

int writeMusic(const char* filename, Composition composition);

Writes the music according to the specifications of the user

xmlNodePtr writeSection(xmlNodePtr part, xmlNodePtr last_measure, Section section, int key);

Writes a section (which may not be exactly a measure long) in the part provided