Spring 2022

Interval Driven Melodic Mutation Using A Genetic Algorithm

Jack A. Carson
Bard College

Interval Driven Melodic Mutation Using A Genetic Algorithm

Senior Project Submitted to

The Division of Science, Math, and Computing

of Bard College

by

Jack Carson

Annandale-on-Hudson, New York

May 2022

Dedicated to:

Chris and Judy Carson for always supporting me.

David Stoltz for teaching me everything I know about guitar and making me the musician I am today.

Acknowledgements:

I would like to thank Robert McGrail for leading me through this process. A senior project is a daunting undertaking, but his confidence became my confidence, and I could not have finished this without him.

Everything I know about Max/MSP is due to Matthew Sargent. His classes have inspired me to think differently about not only music, but coding as well. I am grateful for all the help he has given me over the last year, as well as all the advice he has given me about the future.

Keith O'Hara made it possible for me to stick through my computer science degree, even when I was faltering. I am thankful for all the ways he kept programming fun for me.

Finally, I would like to thank everyone I have met at Bard, faculty and friends alike, for helping me along my undergraduate journey.

Table of Contents

Introduction:

Music has been composed algorithmically long before the computer was invented (Collins & D'Escrivan 2007). Once computers were better equipped to handle the processes necessary for composition, the field of computer music emerged. One result of this was the advancement and automation of algorithmic composition. The ability of a computer program to create music is a fascinating step towards encountering art that was created without any biological or neurological input, human or otherwise.

As a musician and avid music listener, I find the songs I enjoy most are ones that surprise me with compositional ideas unlike anything else I've heard. This typically correlates to pieces of music with a significant level of dissonance, which became a key idea throughout the entire project. Creating a program that can replicate these surprises and producing melodic ideas that a human composer might never think of was my main objective. In this paper I will describe how I approached this problem by creating a genetic algorithm using the programming language Max/MSP.

To do this project, an understanding of basic music theory was necessary. Specifically, the project was built around levels of consonance within different intervallic relationships of notes, as well as utilizing common chord forms. I have been playing guitar for 16 years with a focus on more technical styles, so I already had a working knowledge of all the theory necessary. The first chapter will explain the way I define consonance between intervals and in other aspects of music, and how I chose to represent that in my code.

Max/MSP is a visual, multimedia handling programming language. It is a very powerful compositional tool that is used by many artists in many different disciplines, but it does have a

focus on music. The second chapter will give an overview of the language, consisting of a brief history of the language's development, ways it is utilized, its idiosyncrasies, and why I chose to program in this language over more common ones such as Python.

The rest of the paper will go over genetic algorithms, and what it took to code one into Max/MSP. In chapter three I will define genetic algorithms, detail how they work, and explain why I chose to approach this project in this way instead of with the more commonly used model of Markov Chains. The next chapter will detail my own code, walking through how a melody is handled and subjected to the processes of the genetic algorithm.

Chapter 1: Defining Consonance and Dissonance

A central part of my senior project is the relationship between musical consonance and dissonance. However, many other aspects of music were considered as well, such as different chord forms, tempo or timing, and general music convention. In this section I will define these terms and provide all the musical information necessary to understand my project.

Music theory itself is nothing more than a diagnostic tool for interpreting music. It helps create a language that musicians and composers can speak to each other without having to meticulously define what they mean. It is much easier to say “let’s play a 12 bar blues in C” than “Let’s play over four measures made up of four quarter notes at 120 quarter notes per minute over a chord that’s made up of the notes C, E, G, and Bb…”. Music theory is made up of a set of conventions that make understanding what is happening in a song easier, but it provides no answers to anything outside of those conventions. You can hear a piece of music and really like it, but music theory has nothing to do with it. It can tell you the song follows a standard pop chord progression at a certain speed, and it can tell you that those chords in that order are considered to be very consonant, but it cannot tell you if the song is in any way “good” or why you like it, only what is happening. These conventions can be broken as well. Modern western music is almost always in the 12 tone equal temperament tuning system, but this is a tuning system that has evolved over thousands of years, and isn’t even the only tuning system in use today. My project aims to focus on a specific aspect of music theory, namely how songs can be defined as consonant or dissonant, and see how this can be modeled and manipulated if you take the composer out of the equation, leaving only the computer.

Consonance and dissonance are very easy to identify in music with just the ear, but to quantify that in ways a computer can also understand requires defining those terms. I am mostly

using intervals, or the space between any two notes, as my main method of defining consonance or dissonance. Musically, consonance is a very stable sound that does not have any need to resolve. In western music we divide octaves into 12 tones that are roughly the same distance apart from each other. This distance, i.e. the distance between a C and a C#, is also called a semitone. These make up a chromatic scale, but the vast majority of written music is based off of other scales, the most common two being *Major* and *Minor*. These, along with other common scales, are built using only 7 tones instead of 12. All major scales are constructed using the same intervallic distances in order, no matter what the tonic, or root, of the scale is. For example, the first three notes in a C Major scale are C, the tonic, D, the major second and two semitones away from the root, and E, the major third which is four semitones away from the root. The first three notes in a G major scale are G, A, and B, all with the same relative distance to each other that is present in the first three notes of the C major scale. Intervals can be defined by the mathematical relationship between frequencies, such as a perfect fifth interval always having a 2 to 3 ratio between the two notes. For example, the note A, oscillating at 440 hz, has a perfect fifth of E, which oscillates at 660 hz. The simpler these ratio relationships are, the more consonant the notes sound to the listener. The more complex the relationship between the two frequencies are, such as the 32 to 45 ratio of the infamous tritone, the more dissonant they sound.

These interval relationships can be broken up into five groups, as shown in the table below (Toro & Crespo 2017):

Interval Evaluation	**Interval Name**	**Interval Ratio**
Absolute Consonance	Unison	1:1
	Octave	1:2
Perfect Consonance	Fifth	2:3

	Fourth	3:4
Medial Consonance	Major Sixth	4:5
	Major Third	3:5
Imperfect Consonance	Minor Third	5:6
	Minor Sixth	5:8
Dissonance	Major Second	8:9
	Major Seventh	8:15
	Minor Seventh	9:16
	Minor Second	15:16
	Tritone	32:45

Chords can also be consonant or dissonant, although this is harder to define as there are many ways to construct the same chord. A C Major Triad will always be made up of the notes C, E, and G, but they do not necessarily have to be in that order. Playing an E, G, C (when writing chords out like this, assume the notes are being played in ascending order and without octave skips. E would be the lowest note, followed by the next G higher in pitch, followed by the next C) or G, C, E are still C major triads, namely the 1st and 2nd inversions. All chords are made up of intervals, however, and can thus be defined by the notes' intervallic relations within the chord. C, E, G contains a major third interval (medial consonance) followed by a minor third interval (imperfect consonance), with a perfect fifth interval (perfect consonance) between the first and last note. E, G, C is a minor third followed by a perfect fourth, an imperfect and a perfect consonance with a minor sixth imperfect consonance between the first and last notes. G, C, E is a perfect fourth followed by a major third, a perfect and medial consonance, with a major sixth medial consonance from G to E. To summarize:

- A Major Triad has one perfect consonance, one medial consonance, and one imperfect consonance.
- The 1st inversion has one perfect consonance and two imperfect consonances.
- The 2nd inversion has one perfect consonance and two medial consonances.

From this, it can be concluded that the 2nd inversion of a major triad is the most consonant form of a major triad.

This still is not a perfect solution to labeling a chord's level of consonance. An augmented triad, one that is made up of two major thirds stacked on top of each other, contains two medial consonances and one imperfect. One would imagine this chord to be somewhat consonant, but it is widely regarded as being dissonant. This has less to do with the harmonic ratios of the notes within the chord, and more to do with other aspects of music theory. For one, an augmented triad does not naturally exist in any major or minor scale, or any modes derived from them. It is also impossible to invert, due to the nature of the three notes each being a fourth of an octave away from each other; if you were to try and move the root note to the top of the chord instead of the bottom, the ratios defining the chord would not change at all. I also mentioned earlier that musical consonance implies no real need for resolution, while an augmented chord can be said to "want" to resolve. A chord is typically resolved once it changes in some way to be a more consonant chord, typically one that is derived from the tonic or fifth of the key of the song (it is worth noting that while the terms "key" and "scale" are different in the world of music theory, that difference is a lot less pronounced in my project. I do not use the terms interchangeably, but you can think of them as roughly the same thing). To resolve an augmented triad, all one would need to do is move the note(s) that exist outside of the key to nearby ones that do fit the key. Lastly, augmented triads are a lot less common than major or

minor ones. While they definitely still appear in popular music, general audiences are a lot less exposed to them, which could result in an unconscious bias against them. From looking at the augmented triad in this way, it should be clear that the difference between a consonant and dissonant chord is very delicate.

What I have described about the tonal aspect of music is in no way stagnant. Compositions can, and often do, change tonal elements within the piece. Modulation is a musical technique that changes the tonal center of a piece. This not only changes what notes would be considered consonant or dissonant, but the modulation itself can hold those properties. Some modulations are done between two keys that share notes and chords in common, making the transition relatively seamless and otherwise consonant. Other modulations can happen abruptly and jarringly, with seemingly nothing in common between the original and new keys, sounding very dissonant. A composer might also choose to write a piece that has no tonal center, not adhering to any conventions of key or mode. This is where atonal music comes from, which is prevalent in experimental genres such as free jazz.

Lastly, there is one concept in music that can completely undercut the idea of dissonance: repetition. The more the human ear hears something, the more familiar it becomes, and the more consonant it will sound. Using a few dissonant notes that might even exist outside the key of a song will grab the listener's attention at first, but the more it happens the more natural it feels to the song. This works especially well with chord progressions or lead melodies, parts of a song that are repeated a lot. It is also utilized in music that has a lot of improvisation, where a soloist might play a scale that juxtaposes the key of the song. Some composers will use a simple repeated idea throughout a piece to provide a foundation that they can build on top of in any way they want, and always have something solid to come back to that is consonant to the listener.

Other composers have written music with as little repetition as possible, even going so far as to have none whatsoever, which sounds extremely dissonant.

Ultimately, the ideas of consonance and dissonance in music are extremely context dependent. A single stray note might sound jarring in a simple melody, but sound perfectly reasonable in one with more complex tones. Focussing solely on intervallic relationships is a very simplistic way of thinking about consonance, but it is very easy to model. My program does not account for the key of a song or an underlying chord progression in any way, and completely ignores any temporal aspects of music. This causes my program's output to be biased towards atonal melodies, but I do not consider this to be a detriment.

Chapter 2: An Overview of Max/MSP

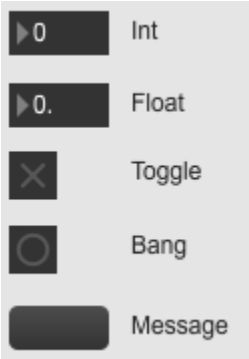
Rather than use a more conventional programming language for my senior project, such as Python or Java, I opted to use Max Signal Processing (Max/MSP). I learned how to use the language over the course of doing this project, both by working on it and by taking a class this semester. In this section I will give a brief history of the development of Max/MSP, what it is and how it works, as well as why I choose to use this language over any other.

Miller Smith Puckette began work on Max in the 1980s at the Institute for Research and Coordination in Acoustics/Music (IRCAM). It was originally written in C and could not synthesize sound, instead only sending Midi data. In 1989, Max Faster Than Sound (Max/FTS) was developed for the NeXT computer by IRCAM, adding the ability to synthesize sound in real time using the computer's digital signal processor (DSP). IRCAM also licensed Max to Opcode Systems that same year, leading to a 1990 comercial release of Max, developed by David Zicarelli. In 1997, Zicarelli bought the rights to Max to prevent Opcode from canceling it, and founded his own company Cycling ‘74 who still develops Max to this day. The next year, Cycling ‘74 released Max/MSP, which allows for the manipulation of digital audio signals in real time without a dedicated DSP. This gave composers/programmers the ability to create their own custom synthesizer instruments and effect processors. In 2011, Max/MSP was made able to integrate with the Ableton Live digital audio workstation (DAW), and in 2016 Ableton acquired Cycling ‘74.

Max/MSP itself is a visual based programming language for multimedia, but with a focus on music. It has a few basic data types, and more complex objects that can be connected to each other. The program works right to left and top to bottom, although delays can be built into the code to get around this if necessary. Max/MSP can handle both Midi data and soundwaves,

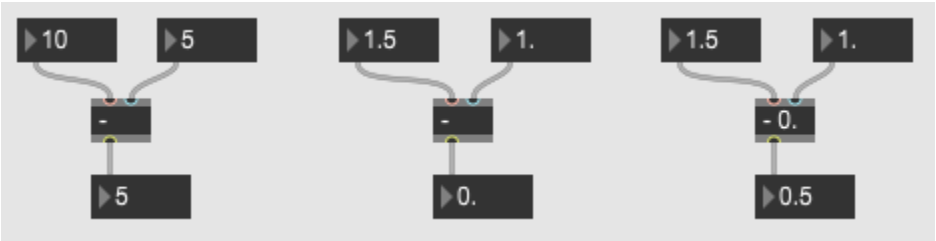
lending itself to a wide variety of uses. Users can create their own instruments or effects that can be brought into Ableton Live for creating music. Users can also use Max/MSP without any external DAW, and create compositions wholly within the program itself. It can be used to generate procedural scores for performers to follow. Users can also create effect pedals for instruments such as electric guitars if they have a way to port their program to hardware. Max's extendable API allows users to create and share packages to be implemented by other users. There is a dedicated community around the program, and Cycling '74 even showcases user made packages on their website, as well as hosts an online forum for users to learn from and share with each other.

Max/MSP has a few basic building blocks: ints, floats, toggles, bangs, and messages. Ints and floats behave similar to other programming languages; ints being integer numbers and floats being numbers with decimals. Toggles and bangs are how the program knows what functions to run and when. Toggles are an on/off button, continuously sending a signal to do something when on, and not when off. Bangs are like buttons in that they can be clicked for a one time activation of any connected functions. Other objects might output bangs as well. Messages hold and give instructions, sometimes numbers like ints or floats, and other times specific functions an object may need to be called, such as a message that opens a file to be put into an object.



All objects in Max/MSP, including the building blocks mentioned above, will have either some amount of inputs that show up on top of the object box, outputs that are on the bottom, or in most cases, both. Wires connect objects together, an output to an input. Some basic objects are math operators. These operators also showcase a key aspect in how Max/MSP treats integers and

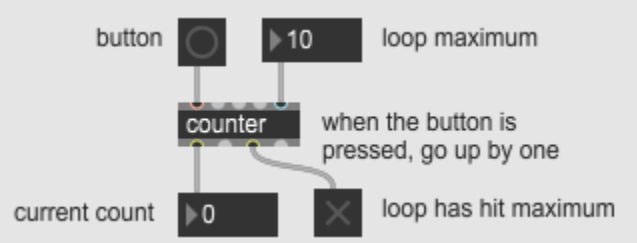
floats. The first example shows two integers, 10 and 5, being subtracted. The 5 is going into the blue, or cold, inlet of the subtraction box. Cold inlets store information to the object, but do not tell the object to do anything with that information yet. In this case, the subtraction object is being told that when a number goes into the orange, or hot, inlet on the right, 5 should be subtracted from it. The 10 going into the hot inlet is what activates the mathematical operation, outputting 5. If the 5 that goes into the cold inlet was not set, or was changed after the 10 was already put in, the outcome would be 10 as the subtraction object was not told what to subtract from 10.



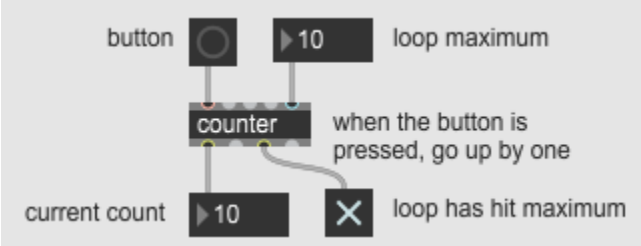
The middle example shows 1.5 being subtracted by 1.0, but it outputs 0.0 despite all the number boxes being floats, and the equation being set up correctly, with the cold inlet being set before anything came in through the hot inlet. This is because all math operators in Max/MSP default to integers, and are not overloaded with float operations. The rightmost example shows the fix for this, in which the user must input some float value into the subtraction object to signal that it should be expecting floats. Math objects can also be given default values instead of having to use the cold inlet. For example, if nothing was in the cold inlet for the correct float subtraction, the object would default to subtracting any value that came in through the hot inlet by 0.0. This value can be anything, if it were written as "- 1.0", and 1.5 came in through the hot inlet, it would still output 0.5 if there was nothing connected to the cold inlet.

Max/MSP has many objects, but most are created with the purpose of dealing with sound waves or Midi data. This leads to Max/MSP not having many functions that are considered very

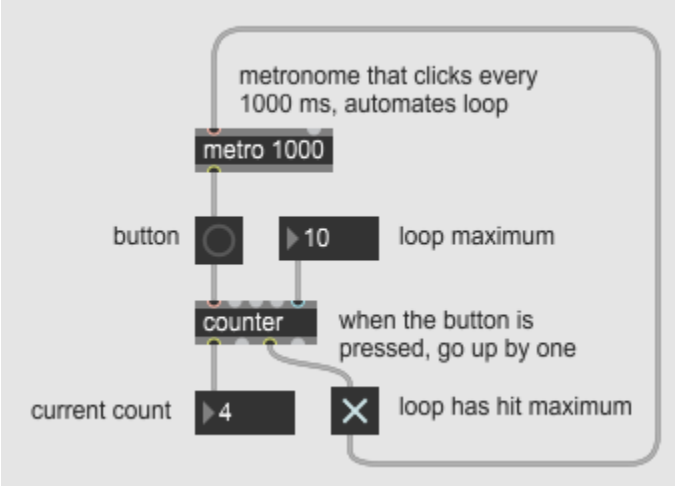
basic for other programming languages, such as any kind of loop. It is still possible to create these functions, but they must be built from scratch. Here is a very basic example of a for loop built using Max/MSP. The button, or bang, is connected to an object “counter”. Counter will



increment the value being output by 1 every time it is activated, in this case when bang is clicked. An integer goes into a cold inlet telling the counter object what the maximum value it can output is. There is another integer that is receiving the current count from the counter object, and a toggle that will activate once the counter has reached its maximum value of 10. The



leftmost outlet of counter sends integer data, but other functions might treat that as a bang, or activation, which is why this is a complete for loop. However, right now the for loop only iterates when a user manually clicks the bang. To automate this, all that would need to be done is attaching a metronome object, called “metro”, that sends a bang every click. In this



case I’ve written the speed of the metronome into it, 1000 milliseconds, but this could also be done with an input into the rightmost inlet. The toggle is now connected to the hot inlet of the metronome, telling it to count continuously as long as the toggle remains on. Once the counter object has hit the maximum count

of 10, it will send a signal to the toggle, telling it to turn off, ending the now fully automated loop.

Max/MSP also handles local variables in a very specific way. Each file, which is analogous to a class in other languages, is considered to be global by default within itself. Subpatches, similar to class functions, can still send and receive data from the main file, or even other subpatches. This led to a complication where I had multiple of the same subpatches that were meant to be called at different times. Every instance of the same subpatch would receive the same data meant to be sent to only one of them, causing unwanted output from subpatches that should not have been accessible. To solve this problem I had to take the code from my subpatches, put them into their own files, and rename everything that acted as a variable to fit Max/MSP's syntax for local variables: #0_.

If Max/MSP doesn't even have built in for loops or local variables, what advantages does it have that would make me choose to use it for my senior project? The most obvious benefit is built in Midi functionality. The point of my project is to be able to mutate a melody, and being able to immediately play the end result without needing to port the data to another software sped up the development process. It also allows for possible extensions to the project, such as a user being able to input a custom melody via a Midi keyboard, or the ability to directly export the melody to Ableton and play it with those instruments. When I first began working on the project, I was using Python, and had a very guitar focussed idea what to do. After only a bit of coding, I realized that trying to perform the math that I wanted to do on the notes had extra levels of complexity that Max/MSP would be able to alleviate. Specifically, figuring out distances of notes on a guitar fretboard. I have since moved away from the guitar centric approach towards one

more focussed purely on the intervallic relationship between the notes in a melody, which has made the math easier, but also took away some of the need to use Max/MSP specifically.

Chapter 3: Genetic Algorithms

The name “genetic algorithms" makes them sound much more complicated than they actually are. Inspired by the work of Charles Darwin, genetic algorithms seek to solve a problem by evolving to the correct answer using an iterative process, the same way an organism might evolve to fit a niche over several generations and years upon years of genetic mutation. In this chapter I will explain how a genetic algorithm works, and how I approached the creation of the genetic algorithm used in my senior project.

The idea of a genetic algorithm is based on Darwin’s theory of evolution, specifically the idea of “survival of the fittest.” Genetic algorithms begin with a user inputting some population of data, and deciding a measure of fitness. Each data point is evaluated against the fitness measure, and assigned a fitness score, contributing to the larger evaluation of fitness for the entire population. The genetic algorithm will then iterate through the population, each iteration being called a generation. While iterating through each generation, some data of the population is selected, and then subjected to different mutation functions. These mutation functions change the data in some way, typically based on what data was selected to be mutated. The mutations are then evaluated according to the measure of fitness, and if the mutations are more fit, they are put into the next generation. This process is then repeated many times, either until some large number of new generations have been created, or until the fitness measure is sufficiently reached.

The paper “Sorting the Sortable from the Unsortable” (McGrail and McGrail 2006) outlines a simple problem where use of a genetic algorithm is an appropriate solution. Consider that a professor might drop some number of grades from assignments in the consideration of a final class grade. This is rudimentary in the case that all grades are weighted equally; simply eliminate the grades with the lowest values. A complication occurs when the grades are

weighted. The paper outlines how there is no readily available solution with the use of sorting algorithms, and instead turns to a genetic algorithm to find the solution.

The algorithm takes an array **A** of lists of scores. Each score is defined as a tuple of a grade and a weight. Initially, the array only consists of copies of the same list. The fitness the algorithm is looking for is a maximum average of some number **k** of the grades in the list, with the average being found by dividing the sum of the first **k** grades by the sum of the first **k** weights. For the first half of the lists in **A**, the algorithm replaces a corresponding list in the second half with two of the scores in the current list randomly switched. The algorithm then finds the fitness of each list, and reorders **A** to be in descending order. The algorithm then repeats the replacement, measurement, and reorder processes some amount of times, and outputs the first list in **A** as the solution.

Another very common approach to computer aided composition is the implementation of Markov chains. Markov chains consider a set of states, and how likely each one is to occur after another. This approach is so popular due to the idea of music being probabilistic and operating under a certain set of rules (Bill 2011), making it useful for trying to replicate human sounding composition. This is not how I want the melodies my program produces to sound, which is why I opted to use a genetic algorithm instead.

Chapter 4: A Walkthrough of the Code

In this section I will go over my code, and explain how my genetic algorithm works.

The first feature a user should take note of when using my program is the consonance slider, as shown in figure 4.1. The user simply clicks or drags on the bar between the words “dissonant” and “consonant”, and the slider calculates the fitness function for the rest of the program to reference. The slider defaults to midi values, which are typically between 0 and 127. The slider outputs an integer within that range (0 being at “consonant”, 127 at “dissonant”) to a scale function. This function takes numbers within a range of values, and returns the relative value in a different range. I wrote the code block as [scale 0 127 0. 1.] so that when given a midi value, it outputs the normalized value from 0 to 1. This value is sent to a float box for two reasons. The first simply being so that the user can see what fitness value they are inputting on the slider. The second reason is that if a user has a very specific fitness measure in mind, they can manually enter it into the float box instead of needing to find it on the slider. Everytime the value in the float box is changed, whether manually or via the slider, it is sent as a global variable “fitness”. The “s” in the code block is a “send” command, and any “receive fitness” or “r fitness” code blocks will output the value in the floatbox.

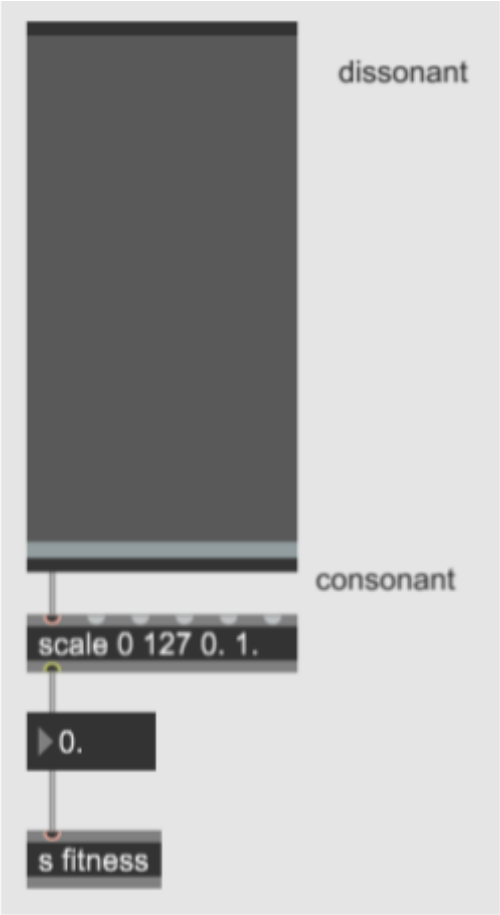


Figure 4.1

The main loop for iterating through the melody is shown in figure 4.2. The counter object starts at 0, and each time it increments it sends its current count into the table holding the melody. The table then returns the midi pitch value of the note at the index of the current counter value. The counter also goes to a variable to keep track of which index is being accessed, cnt1, and also accesses the next two notes in the melody. Due to the counter looking ahead, the maximum value the counter can reach is set to be two less than the length of the melody. The tables output the pitch of the notes being looked at to an integer box, which goes to variables storing the current value of the notes, as well as mods them by 12 so that all notes are compared as if they existed within the same octave. These residues are also stored as variables, and are put together into one list by the "pack" function. The first note is delayed by 100 ms via the "pipe" function, because it is going into the active inlet of "pack", and I wanted to make sure the other values had enough time to be put into the cold inlets. Once the three residues of the notes are put

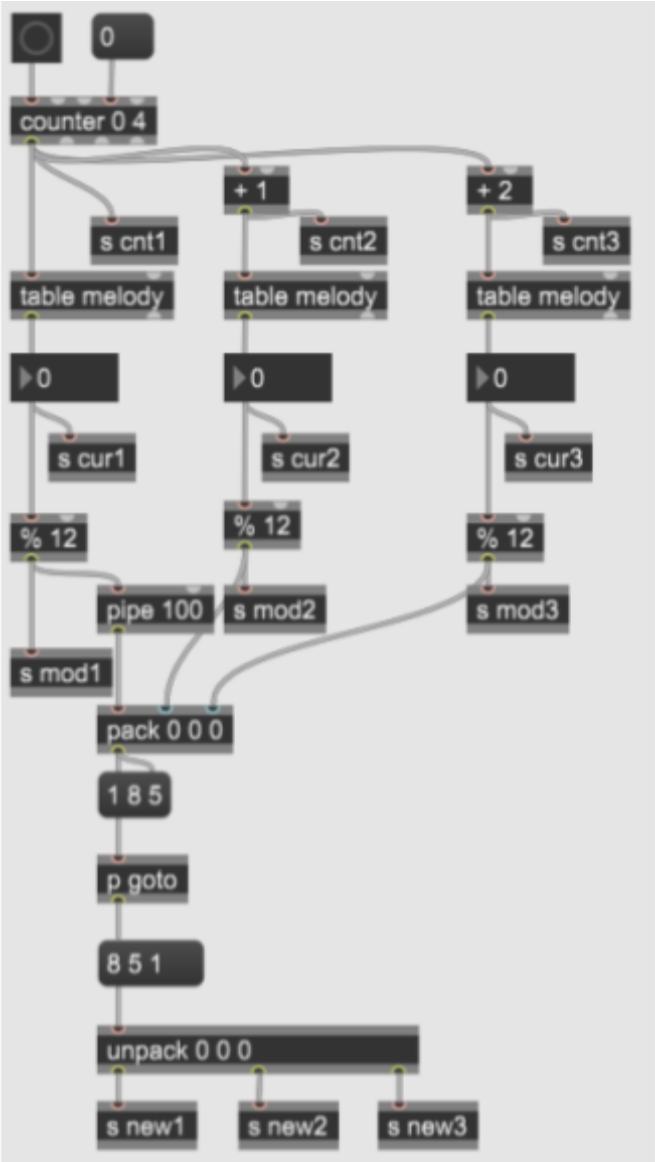


Figure 4.2

together, they are sent to the “goto” subpatch, which returns a list of three mutated notes that are unpacked and set as variables.

The “goto” subpatch I created takes the three notes given to it, and then based on their intervallic relationships, decides which mutation functions to send them to. The square with the 1 on it in figure 4.3 is the input of the function, and sends the information to other functions. Firstly, it takes the three given notes and computes their fitness score using the subpatch

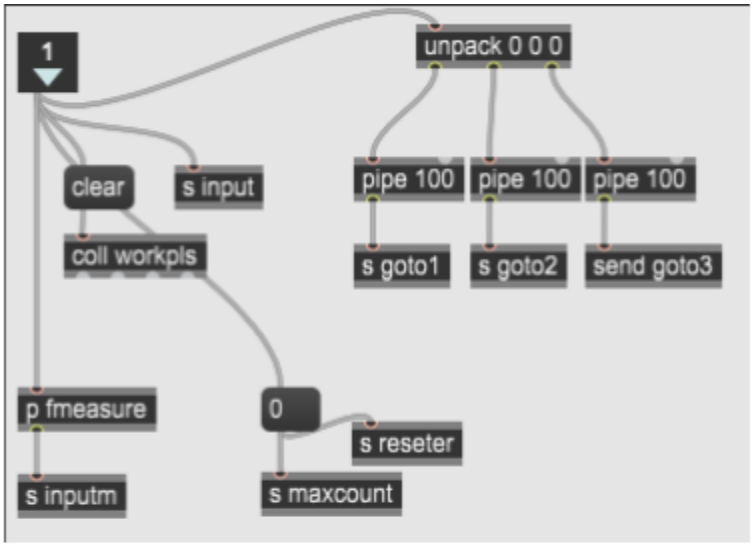


Figure 4.3

“fmeasure”, which is then assigned to the variable “inputm”. The list of three notes are also stored as a variable “input”. When receiving information, the function also activates a couple of messages, “clear” and “0”. The clear message resets a “coll”, or collection of data, and the 0 resets other things in the patch, as well as the variable “maxcount”. Lastly, the three notes are unpacked and sent to different variables after a small delay to make sure everything has enough time to be properly reset.

The three variables storing the note values (goto1-3) are compared against each other to find the absolute value of their intervallic relationships, as shown in figure 4.4. These differences are then sent to a selecting function, which ultimately determines what mutation functions are available. The chords I have decided to use as the basis for my mutation functions never have intervallic distances of 1, 2, or 9, which is why they are not able to be selected. The number that

goes into the selection function is sent to the outlet corresponding to the placement of the numbers after “sel” (if 0 is inputted, it goes to the leftmost outlet, 3 goes to the second leftmost outlet, etc.). These numbers are sent to another selection function that adds 1 to the variable “maxcount” for every valid intervallic relationship, to a possible maximum of three.

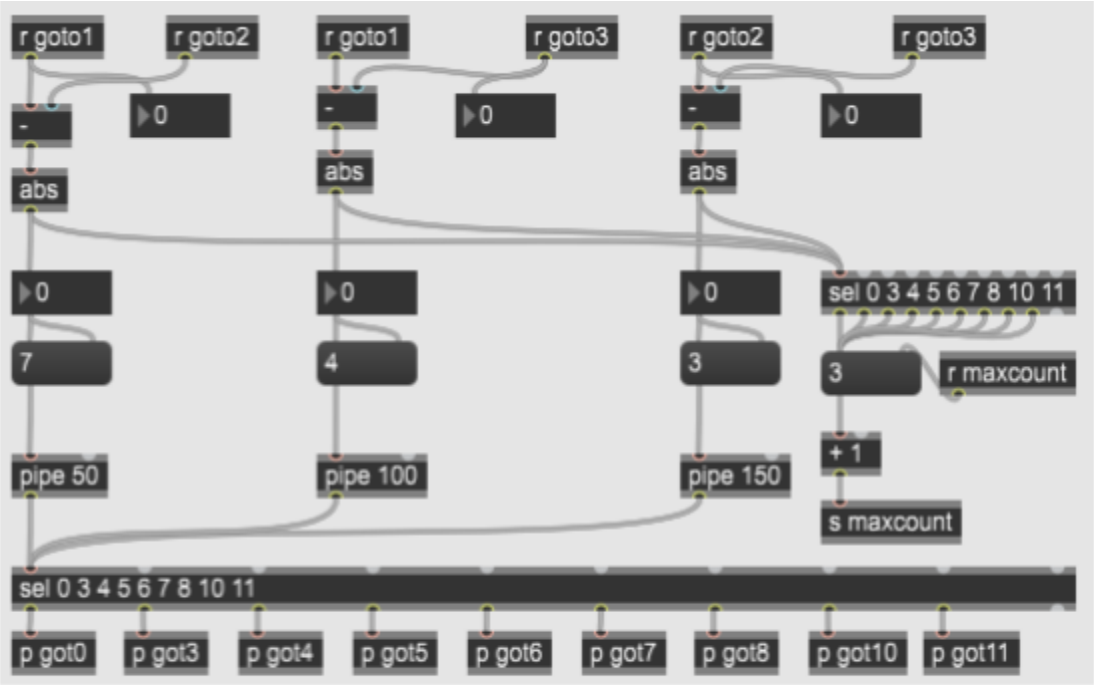


Figure 4.4

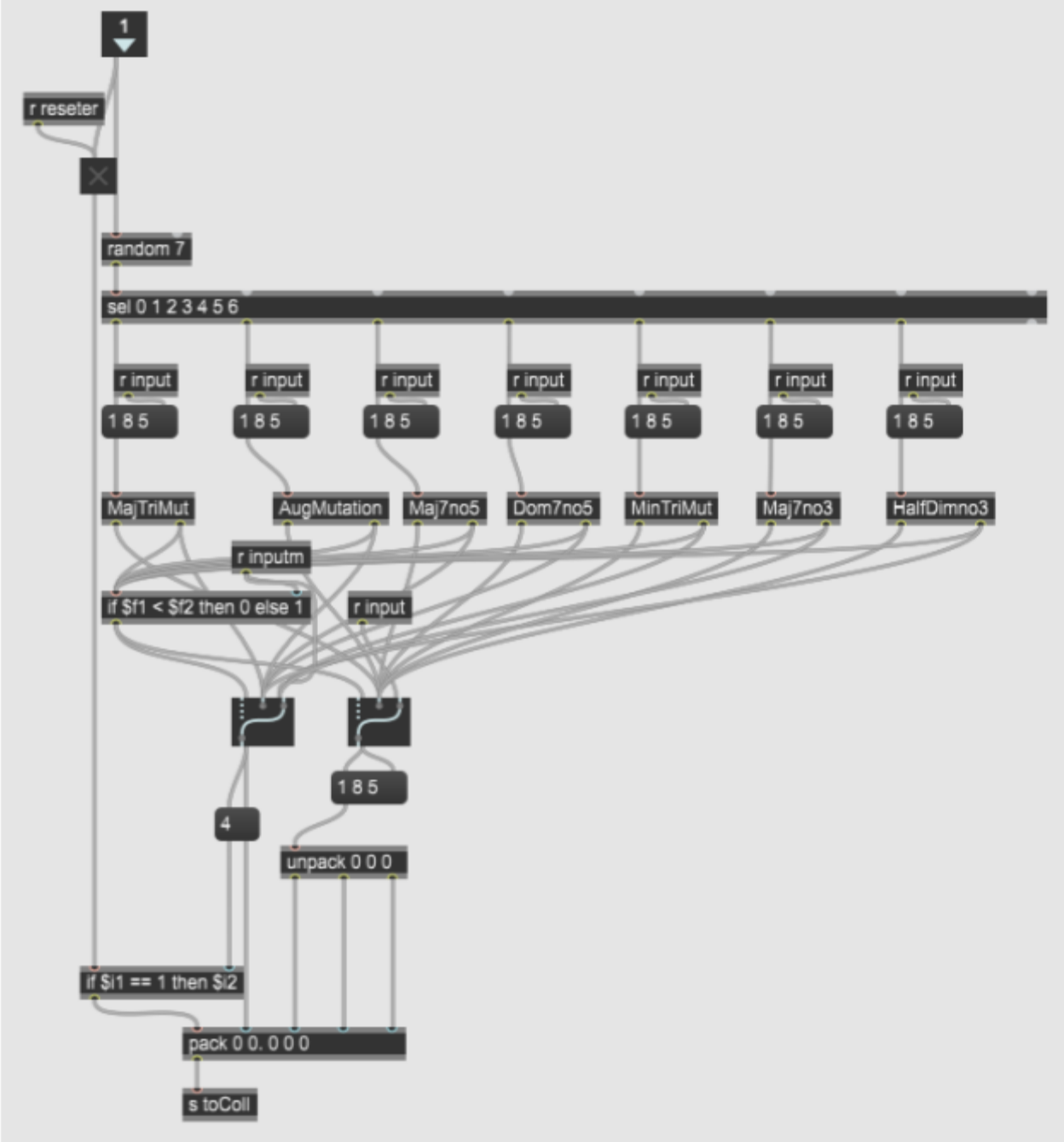


Figure 4.5

Figure 4.5 shows the entire subpatch “got4”, which is only called if there is at least one intervallic distance of 4 between the notes, corresponding to a major third. Before anything is sent to this subpatches inlet, the reseter message first turns off a toggle, which acts as a gate so

that these functions are not mistakenly called or left accessible after every time they are called. When given an input, the toggle is turned on, opening the gate, and a random number between 0 and 6 is generated and sent to a selection function. Based on what number is generated, the three notes that were inputted to the “goto” function (accessed by “r input”) are sent to a mutation function. As this is the “got4” subpatch, the notes can be sent to any chord mutation that has a major third in it: a major triad, augmented triad, two kinds of major 7 chords, a dominant 7th chord, a minor triad, and a half diminished chord. These mutation functions take three notes as an input, and return mutated three notes and their fitness measure. The fitness measure from the mutation is then compared to the fitness of the original three notes via the “if” statement code block. The if statement makes it so that whatever has the lowest fitness score, either the mutation or original three notes, are then added to the coll as a list of the index in the coll, the fitness measure, and the three notes. All of the “gotx” subpatches behave essentially the same, the only difference being how many mutation functions they are able to send notes to. None of these subpatches return anything.

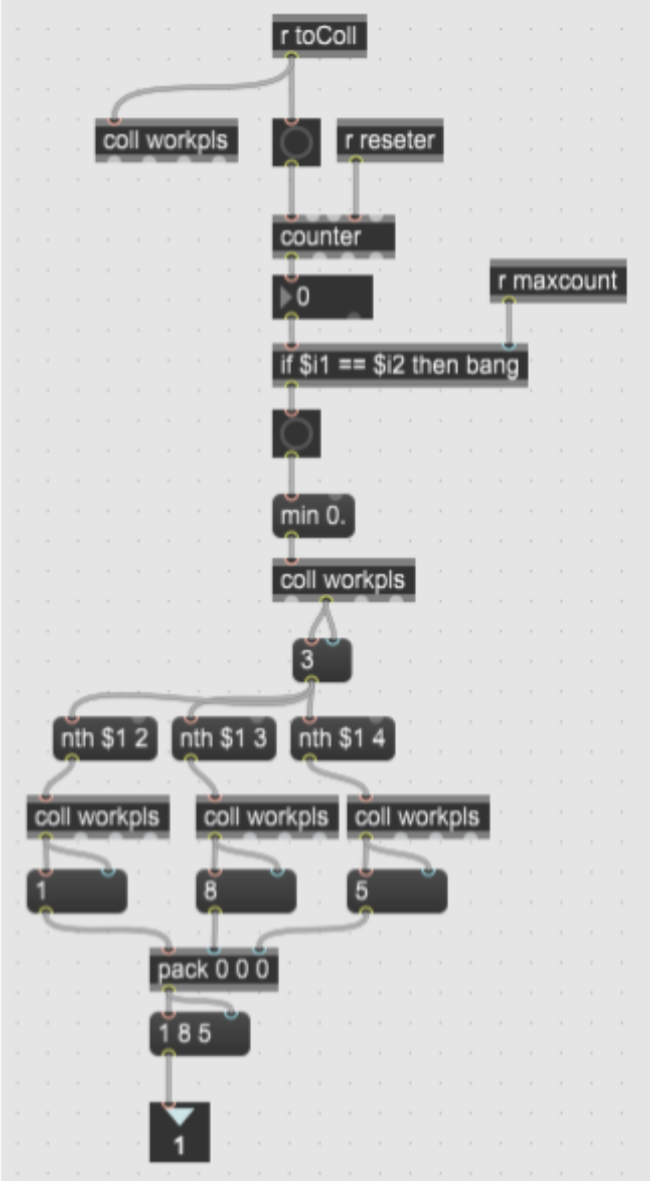


Figure 4.6

The last section of the "goto" subpatch determines its output and is shown in figure 4.6. The "gotx" subpatches within goto all end by sending data to "toColl", which is received here. The data is immediately put into the coll, and a counter is activated once each time that happens. Once the max count is reached by the counter, that means everything that would be added to the coll has been, and the coll is then searched for the index of minimum float value within. This index is then sent as a variable input to the "nth" messages, which search the coll for the three

notes that share the index the float value is stored in. These note values are then packed into a list, which is outputted from the goto function.

After the goto subpatch returns the new notes, they are sent to the last pieces of code in the main function. The code in figure 4.7 changes the values of the melody to be the new, mutated values. The new note is compared to the residue of the note that was determined when the table holding the melody was first accessed. The new note is sent to the active inlets of the operator functions immediately below it, and the residue of the old note is sent to the cold outlets. If the new note is greater than the residue, it sets a message to 1, or 0 if it is less than or equal to. This message is only accessed if the two notes are different. If the notes are the same, then there is no reason for the table to be changed and the code stops there. If the new note was greater than the residue, the difference between the two is added to the original value of the current note in the table, making that now be the same as the new note returned by the mutation function, but in the octave of the original note. If the new note is lower than the residue of the old note, it is subtracted instead. Once the current note is given its new value, it is packed with the corresponding count variable, and put back into the melody table at the correct index. The code

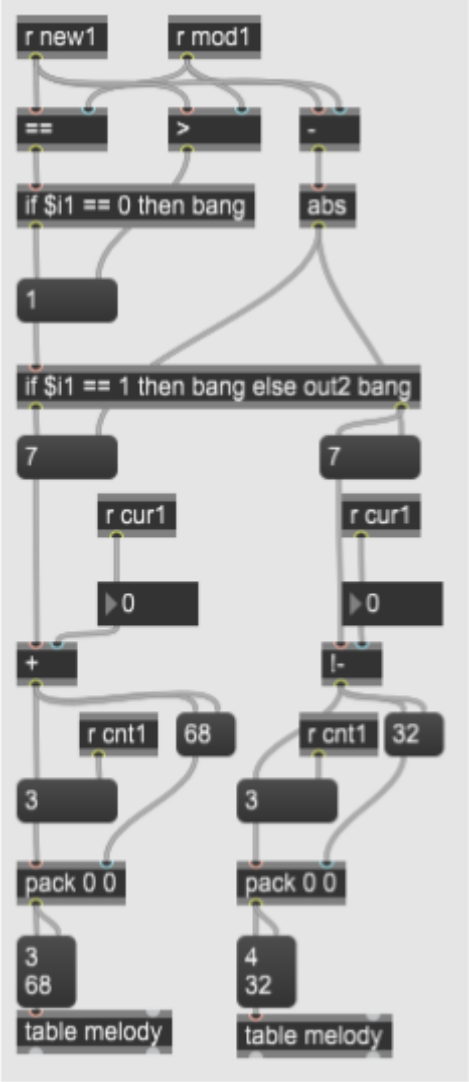


Figure 4.7

shown only does this for the first note, but is the same for the other notes as well, the variables are just changed from "___1" to 2 or 3.

To explain the mutation functions, I will walk through the major triad mutation. Similar to the goto subpatch, the mutation functions are given a list of three notes, which are unpacked and sent as variables, as well as activating a reset method, shown in figure 4.8. Notice that these send methods all begin with with "#0_". This is how Max/MSP handles local variables. Originally, I had the mutation functions as other subpatches as well, but because the variables inside the mutations share the same names (i.e. all major triad mutations had the variables n1-n3), they were being called whenever another instance of the mutation function was, which lead to bad data. I had to copy the code from the subpatches to be new, separate patches outside of the main, and rename the send and receive methods to be local.

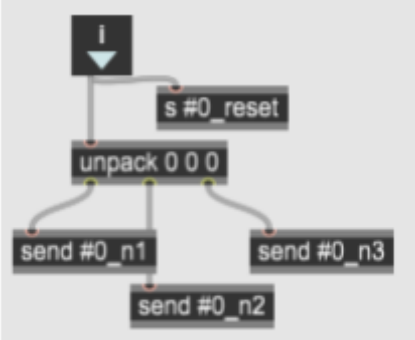


Figure 4.8

The notes are then compared against each other, shown in figure 4.9, to find what interval they should be mutating on. In the example shown, the code is looking for an intervallic distance of a major third, the first interval in a major triad. If it finds one, it adds 3 to the higher of the two notes being compared, adding a minor third interval on top of the major third interval, which completes the triad. The changed note is then packed with the notes that had the major third relation, set as a variable of a possible option for the mutation function to return, and put into the fitness measure function, the output of which is also set as a variable. If there is no major third relationship, the fitness measure is set as 1, the highest it can be, which is to say not fit. There are

2 other sets of similar code, one looking for the minor third interval that exists in a major triad, and the other looking for a perfect fifth.

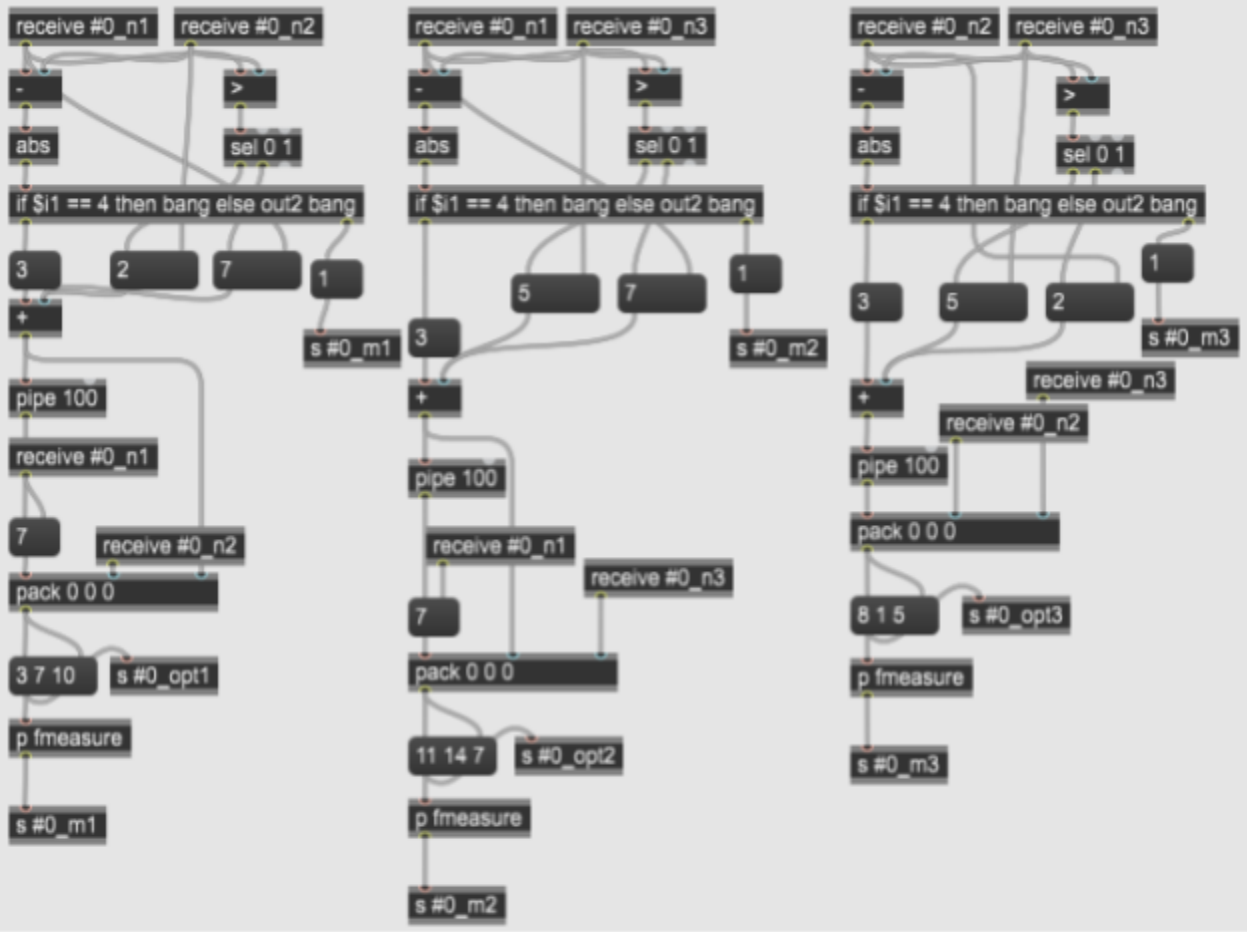


Figure 4.9

Once the possible options of new notes and their fitness scores are found, they are sent to the code in figure 4.10. The code shown is only comparing the fitness measures of a major third being found between notes one and two, and a major third being found between notes one and three, although there are a total of 18 similar pieces of code. The code only compares fitness measures found via the same intervallic relationship, so the fitness measure found from a major third is not compared to one found from a minor third or perfect fifth. If the first fitness measure passed to the if statement is smaller, the first index of the results table is accessed, which

corresponds to the fact that the first fitness measure (m1) is more fit than the second (m2). The value at this index is then incremented, keeping a tally of how many times the first fitness measure beat out another, and sent as a list to the “res”, or result, method.

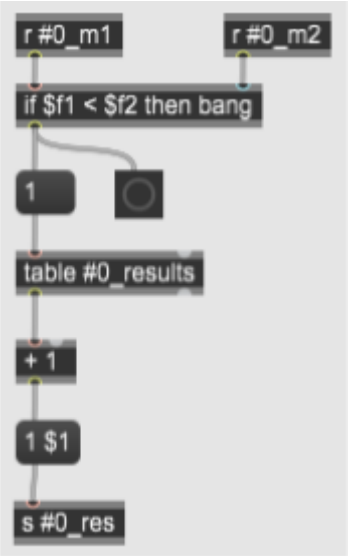
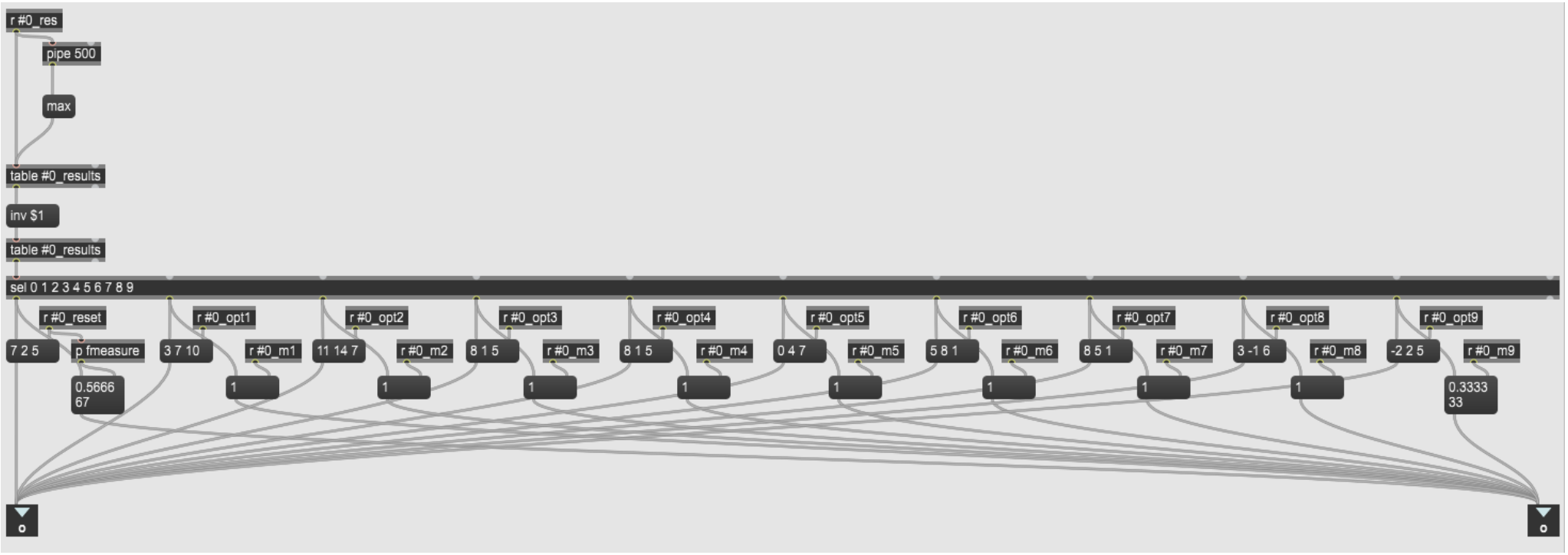


Figure 4.10

The result method (figure 4.11) both passes the list given to it by the last code into the results table, and finds the maximum value of the results table after a short delay so that all results can be given to the table. The maximum value is then used to find its index in the table, which is sent to a select function that returns the corresponding list of notes and their fitness score. The idea is that the index of the table with the highest value must be the most fit.

Results method:



The last piece of code to look at is the fitness measure subpatch, or "fmeasure" (figure 4.13). This function is given a list of three notes, finds the differences between them, and compares that to a table of interval scores that is initialized when the program first opened (figure 4.12). The index of the table represents an intervallic relationship, and the value represents the weight of consonance. The 0th index corresponds to a unison interval, which is perfectly consonant, giving it a score of 0. The 1st index is a minor second interval, which is dissonant, giving it a score of 10. The scores of the intervals between the three notes given to the function are summed, and then normalized from a range of 0 to 30 (the highest possible sum of three dissonances) to a range of 0 to 1. This is then compared to the fitness decided by the user using the slider, and the absolute value of their difference is found. The smaller this value is, the more fit the three notes are, as it means there is not a large difference between their intervallic relationships and the desired level of consonance.

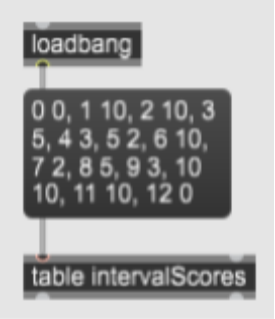


Figure 4.12

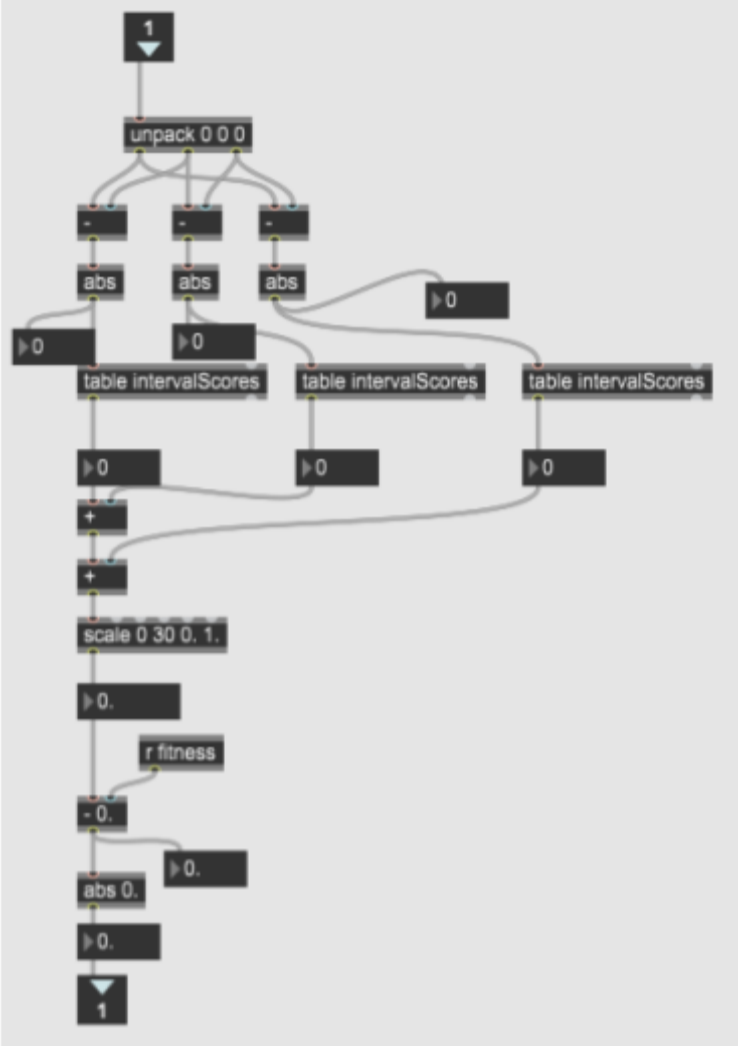


Figure 4.13

Conclusion:

The objective of my senior project was to create a program that would take a melody, and transform it along the spectrum of consonance and dissonance as defined by intervallic relationships. I created a genetic algorithm in Max/MSP, a visual coding language with built in Midi functionality, to achieve this. I have tested the algorithm on multiple melodies, and overall it successfully accomplishes what I want it to do.

Consonance and dissonance, the foundation of my project, are very broad terms in the world of music. They are largely context dependent based on many aspects of a composition, and are also informed by a listener's subjective opinion of what sounds "correct" to their ear. In my project, I chose to define consonance and dissonance based on the intervallic relationships between notes. These relationships can be mathematically expressed as ratios, which removes much of the need for context or subjectivity, as well as makes them easily comparable. The way my program changes notes in melodies, however, is based on different three note chords. These are more difficult to assign values of consonance to, and are often much more context dependent than individual notes. I solved this problem by evaluating chords based on the intervals that make them up, but this still leaves out ideas of key or chord progressions. As a result of this, the algorithm behaves atonally regardless of if it is trying to make a melody more consonant or more dissonant, causing a bias towards dissonance.

The entirety of my project is coded in the visual language Max/MSP. This was useful for handling midi data, but led to other challenges. Max/MSP feels much more stripped down than other coding languages like Python or Java, as even basic functions must be built manually by the programmer. This allows for a wide variety of solutions for any problem, but it also makes it easy to get lost when programming anything complicated. The main benefits of Max/MSP are

how the programs can be extended. Due to the inherent Midi functionality, it is easy to direct the output of a Max/MSP program into a D.A.W. like Ableton. The multimedia capabilities of Max/MSP also make it a great language for any coding that might be necessary for art installations, or even an entire installation. Aside from handling Midi data, I do not know if there was any aspect of Max/MSP as a language that was necessary for my senior project. However, it is a language that I fully plan on using in the future, and coding my entire senior project in Max/MSP was a great way of learning it.

The approach I chose to take for my senior project was a genetic algorithm. Over many different iterations, a genetic algorithm gradually finds a solution to the problem given to it based on some metric of fitness. My program does this by continuously going through the melody given to it, and applying three note mutations that are found to be more fit than what is already there. It decides what is fit based on the user's desired level of consonance. I chose this approach over the use of markov chains, a more commonly implemented model used in computer composition, because I did not want my algorithm to make decisions based on probability. One of my goals was to create a program that could create melodies a human composer might never think of, and markov chains are used to predict most likely outcomes, which is the opposite of what I was looking to accomplish.

My genetic algorithm is coded to look at three sequential notes in the given melody, and mutate them based on what intervallic relationships exist between the three notes. Different intervals apply different mutations to fit common three note chords. The mutations and original notes are compared against the fitness measure given by the user, and the closer of the two is put back into the melody. There are not mutations for every possible interval. Specifically, any kind of second interval or a major sixth have no associated mutation functions. Additionally, the

intervals of a unison, perfect fourth, and minor sixth only have one mutation function each, whereas every other interval randomly selects from multiple possible mutations. Increasing the amount of mutation functions to fill in these gaps would add variety to my program's output. Most of my mutation functions are based on triads or seventh chords, so I would need to look to other chord forms to do this, or create mutation functions that are not based on chords.

I have run my algorithm on three different melodies: Shave and a Haircut, Never Meant (American Football 0:11), and Party Smasher (Dillinger Escape Plan 1:01). Shave and a Haircut is very short, only being seven notes, most of which are either a major or minor second apart. This makes it a bad melody for my program to run on, and not much changes within the melody. I realized that second intervals are very common in melodic lines, which is another reason to create mutation functions based on them. I next ran the melody of Never Meant through my program because I consider it to be both angular (a melody with larger leaps between notes than typical) and consonant, which made it ideal for testing. I found that my algorithm was able to produce more interesting results from this melody, but still left some sections largely untouched regardless of what the fitness measure was. I believe this to be due to the fact that my melody looks at three notes at a time, and in the part of the melody that remained unchanged, every other note is the same. This would cause that part of the melody to always go through the single mutation function correlated to unison, leading to a lack of variation. I found the most success with the last melody. Party Smasher is extremely dissonant, mostly consisting of tritones and minor seconds. It consistently produced the most changes on the original melody regardless of the fitness score being more consonant or dissonant. The melody ends with a cluster of notes all within a minor second of each other, which never get changed, but the rest of it worked great.

From this I conclude that my genetic algorithm was a success. It is able to evaluate the consonance level of a collection of notes, and mutate them accordingly. My program works best on melodies that are longer and more angular, but this is solvable by adding more mutation functions for interval relationships underrepresented by the chords I chose to base mutations off of. The results do not often sound like a melody a human composer would write, which is what I set out to accomplish. Currently, I do not see much use for my project outside of looking for inspiration when writing original melodies, but it is a proof of concept that genetic algorithms can be used in computer driven composition.

Bibliography

American Football. “Never Meant.” *Spotify*. https://open.spotify.com/track/51R5mPcJjOnfv9lKY1u5sW?si=9f6006cf0208475c

Bill, Chip. “Consortium for Computing Sciences in Colleges.” Consortium for Computing Sciences in Colleges, *Algorithmic Music Composition Using Dynamic Markov Chains and Genetic Algorithms*, 2011.

Collins N. and D’Escrivan J., *The Cambridge Companion to Electronic Music,* New York, NY: Cambridge University Press, 2007.

Dillinger Escape Plan. “Party Smasher.” *Spotify.* https://open.spotify.com/track/7D96FRFBAzdjokigZD7RtD?si=ec1c0961fdef46a2

McGrail, Tracey Baldwin, and Robert W. McGrail. “American Association for Artificial Intelligence.” American Association for Artificial Intelligence, *Sorting The Sortable From The Unsortable*, 2006.

Toro, Juan & Crespo, Paola. (2017). Consonance Processing in the Absence of Relevant Experience: Evidence from Nonhuman Animals. Comparative Cognition & Behavior Reviews. 12. 33-44. 10.3819/CCBR.2017.120004.

Appendix:

These are the three note chord forms the mutation functions were derived from, as shown on a guitar fretboard. The vertical lines are strings, incrementing by one semitone every horizontal line. The horizontal lines are frets, all having a perfect fourth interval, or five semitones, between the strings used. The diminished triad does show two notes being on the same string, but the interval relationships are correct. The half diminished chord is only shown as having a no 3rd form. This is because the no 5th form is the same as the minor 7th no 5th form. Similarly, the minor 7th no 3rd form is the same as the dominant 7 no 3rd..

Augmented Triad

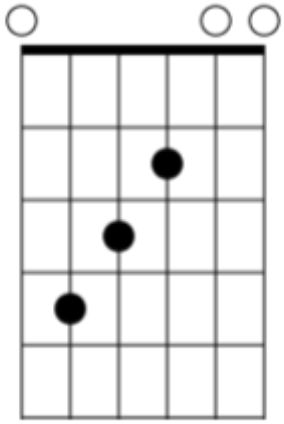


Figure 5.1

Diminished Triad

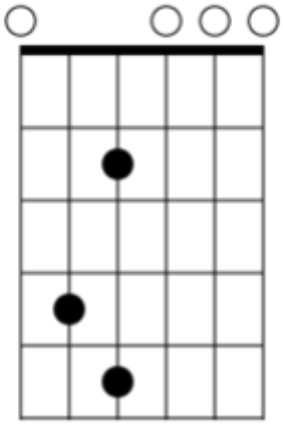


Figure 5.2

Dominant 7 no 3rd

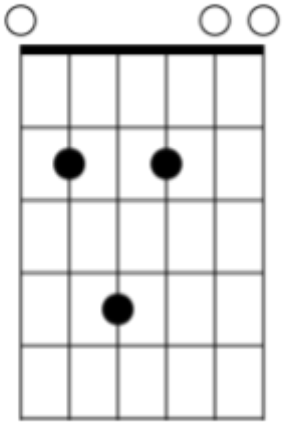


Figure 5.3

Dominant 7 no 5th

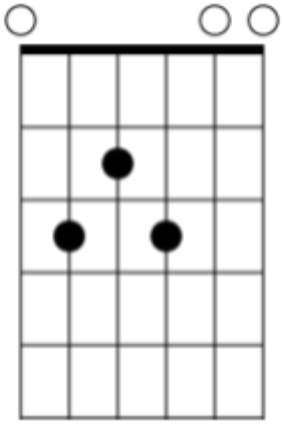


Figure 5.4

Half Diminished no 3

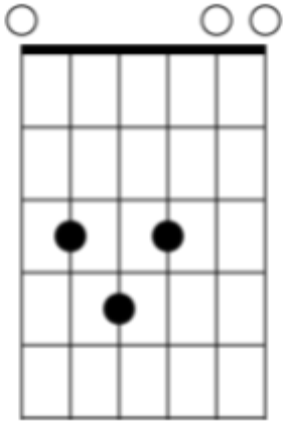


Figure 5.5

Major 7 no 3rd

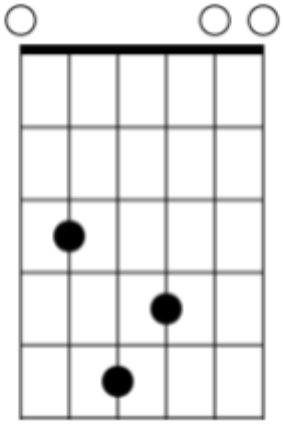


Figure 5.6

Major 7 no 5th

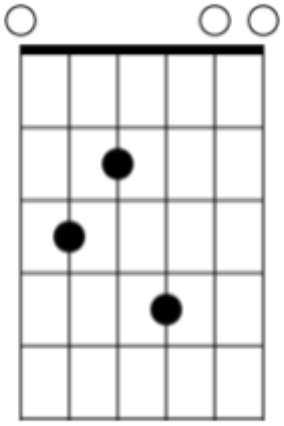


Figure 5.7

Major Triad

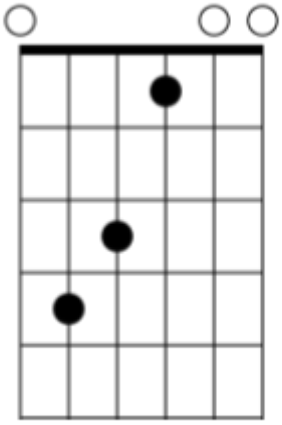


Figure 5.8

Minor 7 no 5th

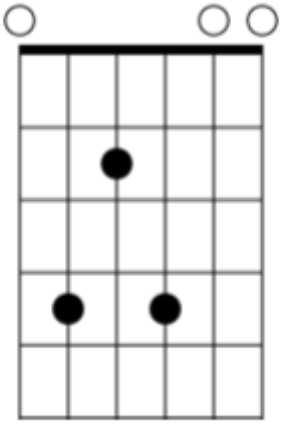


Figure 5.9

Minor Triad

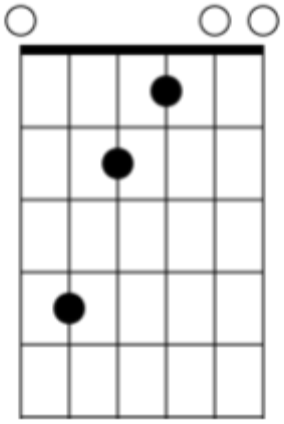


Figure 5.10

Power Chord

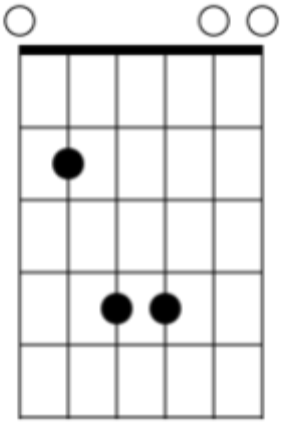


Figure 5.11