Brad Pitney

12/11/2013

ECE 578

Perkowski

Homework #1: IGA-based Motion Generation with Countess Quanta Robot

# Introduction

An interactive genetic algorithm (IGA) was created to evolve new motion sequences to be used by the Countess Quanta robot in playing the existing harp instrument. The IGA chromosome is used to encode a sequence of hand motions, which defines a “song” that can be played by the robot. The available hand motions include the ability to lift and reposition the hand, which offers an improvement over the existing rhythm files. During fitness evaluation, the robot performs each song for a human user, who then rates the relative quality of each song. Along with the IGA development process, this report includes a review of the song rating system and a performance analysis of the selected GA parameters.

# Problem Description

The existing software for the Countess Quanta robot allows the robot to play from a selection of seven rhythm files. Each of these rhythm files is relatively simple, consisting of a series of 3 to 5 moves using only the wrist servo (Servo 0). When I initially began looking at how to apply an IGA to evolve some behavior in the Countess Quanta robot, I thought that I might use the existing rhythm files and explore different combinations of these files played in series. However, after spending some time testing these files with the robot, I began to see that it would be very hard to evaluate combinations of these rhythms in any sort of meaningful way. The files provide a good demonstration of basic sound generation using the instrument, but the similarity between each rhythm would make it difficult to arrange them in a sequence and decide whether the song is ‘good’ or ‘bad’. For a human listener, each sequence would sound very similar and it would be hard to define any sort of rating system. Without being able to meaningfully rate a sequence, it wouldn’t be possible to use an IGA to generate an improved sequence, or to evaluate the performance of such a GA.

Instead of recombining the existing rhythms, I began to look into how I might use an IGA to generate improved rhythm files. One of the main limitations of the existing rhythms is that they only utilize a single servo. The robot moves the arm into position with the hand pressed against the instrument strings, and each rhythm file moves the hand left and right in different patterns to strum the strings. This places some large restrictions on what kind of sounds the robot is able to create compared to what a human would be able to do playing the same instrument. For instance, when a human strums an instrument, they might lift their hand at the end of the motion to change the sound. They might also position their hand at different locations above the strings, before lowering their hand against the strings and strumming. To capture this additional complexity, I decided to add a degree of freedom to the allowed robot motions, and let the robot lift its hand while playing.

# Move Representation

To capture this behavior, I defined four types of moves:

1. Robot lifts hand off of the strings.
2. Robot lowers hand onto the strings.
3. Robot moves raised hand above the strings, to a new position.
4. Robot moves lowered hand across the strings, strumming the instrument.

The robot can only perform move types 1 and 4 while the hand is in the ‘lowered’ state, and can only perform moves 2 and 3 while the hand is in the ‘raised’ state. Additionally, the actual servo motions for move types 3 and 4 can be identical, and only change functionally depending on whether the hand is currently raised or lowered. Because of these features, the moves can be simplified to two types:

1. A hand-state toggling move, which raises the hand off of the strings if it is currently lowered, or lowers the hand onto the strings if it is currently raised.
2. A positioning/strumming move, which strums the hand across the strings if the hand is currently lowered, or repositions the hand above the strings if the hand is currently raised.

The figure below is a simple state machine showing this behavior:



# IGA Design

To create the genetic algorithm, I referred back to a GA that I had written in Matlab for a previous ECE 559 Genetic Algorithms class, which was used to solve an instance of the Santa Fe Trail Problem. Since the Countess Quanta servo control code is all written in C++, much of this GA code had to be re-implemented, due mainly to differences in how arrays are handled between C++ and Matlab. I was able to adapt the basic algorithm for the roulette wheel parent selection and 2-point crossover that I had used in this previous project. The crossover and mutation chances that I selected for the Countess Quanta GA (60% and 1%, respectively) were typical GA values that had worked successfully in this previous class project.

One feature that I had used in this previous project and had opted to leave out the Countess Quanta GA was the implementation of ‘elitism’. In my previous project, I had included logic which would ensure that the current best solution in any generation would always be passed on to the next generation unmodified, to ensure that this solution was never lost. This concept doesn’t make much sense in the context of the Countess Quanta GA, due to the subjective nature of the song quality. It’s not really expected that the GA would discover a single ‘optimal’ song, so much as it would generate a set of songs with similar desired qualities. However, it’s still possible that a user might encounter a particular song that they would like to immediately preserve, to ensure that it’s not lost during subsequent GA activities. To address this, I added a ‘save’ feature that can be used to store the current song to a file.

To structure the GA, I defined a chromosome consisting of ten genes, with each gene representing a single movement in a sequence. Each gene contains a ‘toggleHandState’ Boolean, to store whether this is a move to raise/lower hand, or whether this will move the hand side-to-side. A ‘wristPosition’ integer within each gene stores the new side-to-side hand position, which is used if ‘toggleHandState’ is false. When a gene is created or mutated, the ‘toggleHandState’ value is randomized with a 50% chance of either state. If this value is false (i.e. ‘wristPosition’ will be used), then the ‘wristPosition’ value is set to a random integer within the allowed 600 to 1300 wrist position range.

The actual servo motions corresponding to a single movement were determined by experimenting with the Countess Quanta robot. Before running any motions, the existing motion script file called ‘MoveArm’ is executed, to move the robot’s right arm into instrument playing orientation. Once in position, the hand is raised and lowered by changing the position of ‘servo::Elbow\_Extension’ (Servo 2). Moving this servo to position 720 raises the hand off of the instrument strings, and setting this servo to position 884 lowers the hand so that it touches the strings. The strumming and repositioning moves are executed by changing the position of servo::Wrist\_right (Servo 0). Integer values in the range 600 to 1300 are allowed, since this range keeps the robot’s hand on or above the strings of the instrument. Prior to executing a move sequence, the hand is always raised and the wrist position is set to 950, for consistency.

# Fitness Evaluation Process

During execution, the GA first randomly generates a population of ten chromosomes. The fitness of each individual in the population is then evaluated by converting each gene in the chromosome into a servo move command. The sequence of move commands is sent to the robot servo controller with 100ms intervals between each motion. The resulting robot performance is then evaluated by a human viewer, who rates the quality of the song on a scale from 1 to 9. The viewer inputs their rating value into the program terminal window, and this value is stored as the fitness of the corresponding individual.

One time-saving feature that I included in the fitness evaluation routine was to skip re-evaluation of any individuals that had already been evaluated in a previous generation. That is, if a particular move sequence was rated by the user in Generation 1 and this sequence happens to be passed on unchanged into Generation 2, then we simply use the previous fitness value for this individual during the Generation 2 evaluations, rather than requiring the user to rate this song again. This both saves time during the evaluation process and reduces error due to the user possibly rating this song differently in each generation.

# Parent Selection, Crossover, and Mutation

Once the viewer has rated all ten individuals, the GA uses roulette wheel selection to select ten parents for the next generation. For each pair of parents, there’s a 60% chance that 2-point crossover is then applied to create two offspring. The remaining 40% of the time, the parents are passed along as offspring, unmodified. After the crossover process, each gene in the offspring population is subjected to a 1% mutation chance. If a gene is selected for mutation, then it is replaced with a new, randomly generated move (i.e. new random ‘toggleHandState’ and ‘wristPosition’ values). The evaluation process then repeats for this new population, and the GA progresses until the specified number of generations is reached.

# Design Considerations

One question that arose early on was whether to blindly perform all moves described in a chromosome, or to include some logic for pruning moves that don’t affect the actual strumming of the instrument. For instance, the move list might include a move to raise the hand above the strings, followed by a series of multiple wrist moves that just wiggle the hand in the air, followed by a move to lower the hand onto the strings. In this case, only the last wrist move actually matters, since this move affects the position that the hand will be in when it is finally lowered onto the strings – every other wrist move prior to this is superfluous. To prevent this situation, I added some logic to ignore all but the last wrist move prior to lowering the hand.

Another scenario where I had considered ignoring extra moves is the case where multiple ‘hand-toggle’ moves occur in series. For instance, the move list might include a move to lift the hand off of the strings, immediately followed by a move to lower the hand back onto the strings. In this case, the hand hasn’t been repositioned, so it would seem that this move was pointless. However, in this case, the move sequence does cause the hand to impact the strings, which would have an effect on the song. Similarly, a human playing a stringed instrument might strum the instrument and then touch the strings again to dampen the sound. I decided to allow this ‘raise then lower’ move sequence, since it might allow for interesting behavior to appear.

# Song Rating System

One of the challenges of this project was in determining how to rate the quality of a given motion sequence. This required experimental testing with the robot, in order to get a sense for what kinds of sounds this system allows. Once I had some experience with the range of sound quality that the instrument and robot was able to produce, I was able to provide a basic rating of how pleasant or unpleasant I found a song, compared to others that were being produced. Below, I’ve included some examples of generated motion sequences that resulted in songs that I found to be either good or bad. The sounds produced by a motion sequence are not obvious from reading the servo coordinates, so I’ve included descriptions of the performance and explanations for why I liked or disliked each song.

**Good Song 1:**

Repositioning hand to 1174.

Lowering hand onto strings.

Strumming strings to 1113.

Strumming strings to 1288.

Strumming strings to 740.

Strumming strings to 1201.

Strumming strings to 685.

Raising hand off of strings.

Lowering hand onto strings.

Strumming strings to 806.

When playing this song, the robot makes several large strumming motions in sequence, lifting its hand at the end. It then lowers its hand and makes one last strumming motion. I found that the vigorous strumming provided a sense of enthusiasm. Lifting the hand before the last strum added variety, and seemed more like the kind of motion a human would make if they were playing the instrument.

**Good Song 2:**

Lowering hand onto strings.

Strumming strings to 1251.

Raising hand off of strings.

Skipping superfluous move to 1074.

Skipping superfluous move to 1211.

Skipping superfluous move to 769.

Skipping superfluous move to 1151.

Repositioning hand to 775.

Lowering hand onto strings.

Strumming strings to 1088.

In this song, the robot makes two large strumming motions from different locations. I thought that the strumming sounded very deliberate, and simulated how a human might use the instrument.

**Bad Song 1:**

Lowering hand onto strings.

Raising hand off of strings.

Repositioning hand to 1154.

Lowering hand onto strings.

Raising hand off of strings.

Lowering hand onto strings.

Raising hand off of strings.

Repositioning hand to 1052.

Lowering hand onto strings.

Strumming strings to 1136.

In this song, the robot pats the strings repeatedly, and strums a small distance at the end. I thought that this kind of motion would work better for a drum than a stringed instrument.

**Bad Song 2:**

Lowering hand onto strings.

Raising hand off of strings.

Skipping superfluous move to 1214.

Skipping superfluous move to 632.

Skipping superfluous move to 1168.

Skipping superfluous move to 671.

Skipping superfluous move to 1146.

Repositioning hand to 1015.

Lowering hand onto strings.

Strumming strings to 763.

This song appeared while evolving decent songs with many strumming motions. It shows an example of a potentially good song that was greatly reduced in quality by the placement of an extra hand state toggle move just prior to the strumming sequence. The hand is raised off of the strings before the sequence, so the large strumming motions are skipped entirely. The resulting song is a single strumming motion, which sounded very boring compared to the other rapid strumming songs that the GA had been evolving.

After viewing and evaluating many songs, I was able to identify some specific features of the motions that often determined whether I rated a song as ‘good’ or ‘bad’. Of course, this rating system is very subjective, and another user might have a very different set of criteria, depending on what kind of song they are trying to evolve. Here are some of the criteria I found myself using:

**Features of a ‘good’ song:**

* The song involves enthusiastic strumming, due to sequences of wrist moves with large changes in wrist position.
* The song involves a sequence of strumming, raising and repositioning the hand, and then strumming again.

**Features of a ‘bad’ song:**

* The song involves very few moves, due to much of the sequence being superfluous positioning moves.
* The song involves little motion, due to strumming moves having little position change between wrist coordinates.
* The song consists mostly of ‘patting’ the strings, due to sequences of hand state toggle moves.

# IGA Performance Comparison

From experimenting with the IGA over several trials, I found that songs from later generations tended to sound similar to the highly rated songs of earlier generations. From this experience, it seemed that the IGA was generally successful in evolving a particular kind of song. To perform a more quantitative analysis of the IGA performance, I gathered the song rating data from three trials. Each trial consisted of evolving a population of ten individuals over a period of five generations, using the “Standard GA” properties described earlier (60% crossover rate, 1% mutation rate).

Note that this collected data for each generation excludes any individuals that were brought over unmodified from the previous generation. The ratings within each generation (after the first generation, since this population is random) are based only on new songs that were created from the GA’s crossover and mutation routines. In this way, the collected data correctly represents the GA’s capability for evolving improved individuals based on the user’s feedback.

The figure below shows a Microsoft Excel plot of the average song rating of each of the five generations, for the three trials. This chart includes best-fit linear trend lines for the data of each trial.

From the trend lines, we can see that each trial showed some increase in average song rating as the generations progressed. Trial 3 showed the least increase, but it also happened to start with the highest average rating of the initial populations. From the individual data points, we can see that the average rating sometimes varied greatly between generations, as in the case of the Trial 2 and Trial 3 value at Generation 4, which are both below the adjacent generations. This variation is likely an impact of the relatively small GA population, which results in collecting a small number of data points within each generation.

To get a better idea of how well these “Standard GA” parameters performed compared to other potential IGA settings, I tried running the software in two other configurations, and collected trial data for each of these. I defined a “Mutation GA” by setting the crossover rate to 0% and raising the mutation rate to 30%. This configuration relies entirely on occasional mutations of individual genes to generate new motion sequences. I also defined a “Random Motions” configuration by setting the crossover rate to 0% and the mutation rate to 100%. With this configuration, every generation consists entirely of new randomly generated individuals. This random configuration acts as a kind of control group, and should show some baseline song rating with no expected change over each generation.

The figure below shows the results of these new trials. The “Standard GA” plot consists of an average of the three trials discussed above. The “Mutation GA” and “Random Motions” plots are each the results of similar trials, with populations of ten individuals evolved over five generations. The best-fit linear trend lines for each data set are also included.

From the trend lines, we can see that the “Standard GA” parameters tended to result in the best overall increase in average song rating across the generations, compared to the other methods. The “Mutation GA” also showed an increase in song rating over the generations, although this wasn’t as pronounced as with the “Standard GA”. I found that the “Mutation GA” generated a greater variety of songs in later generations, compared to the “Standard GA” which tended to focus in on a single type of song. Since the user can save songs to a file at any point, the “Mutation GA” might be useful for exploring different types of songs and saving off any that are appealing.

The “Random Motion” configuration shows a roughly horizontal line, as we would expect from a method that doesn’t adapt to user feedback. These “Random Motion” results also indicate that my average rating for a completely random song within this framework tended to be around 3. It’s interesting to note that the “Random Motion” average ratings also showed high variability, ranging from an average rating of 2.0 for Generation 3 to a value of 4.2 for Generation 4. Since all generations should ideally show the same average with this method, this range reflects the kind of variation that we would expect to see in the other trial data, most likely due to the small number of individuals in the population.

# Conclusion

An IGA was developed to evolve new motion sequences for playing an instrument with the Countess Quanta robot. Moves types were defined to allow the robot to lift and lower its hand, to support strumming motions across the strings, and repositioning moves above the strings. IGA fitness evaluation was implemented by performing songs for the user and collecting song quality rating data from the user. The rating system was reviewed and possible elements of high and low quality songs were discussed. The IGA performance was evaluated by analyzing rating data from three trials to see how the average rating of newly evolved songs changed over each generation. The IGA with standard crossover-based parameters was compared to both a purely mutation-based GA, and a random motion generator, and was found to show higher average ratings that these alternate methods.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Standard GA Trial 1** |  |  |  |  |
| **Generation:** | 1 | 2 | 3 | 4 | 5 |
|  | 2 | 3 | 6 | 2 | 5 |
|  | 2 | 3 | 2 | 5 | 5 |
|  | 3 | 4 | 2 | 1 | 5 |
|  | 5 | 2 | 5 | 7 | 3 |
|  | 6 | 1 |  | 5 | 6 |
|  | 2 | 2 |  | 5 | 6 |
|  | 3 | 5 |  | 6 | 6 |
|  | 2 |  |  |  |   |
|  | 1 |  |  |  |   |
|  | 5 |   |   |   |   |
| **Average:** | 3.10 | 2.86 | 3.75 | 4.43 | 5.14 |
|  |  |  |  |  |  |
| **Standard GA Trial 2** |  |  |  |  |
| **Generation:** | 1 | 2 | 3 | 4 | 5 |
|  | 5 | 7 | 8 | 2 | 8 |
|  | 4 | 2 | 3 | 8 | 7 |
|  | 2 | 4 | 2 | 8 | 7 |
|  | 8 | 4 | 8 | 1 | 8 |
|  | 3 | 3 | 8 | 2 | 7 |
|  | 3 | 7 | 2 | 2 | 8 |
|  | 6 | 7 |  |  |   |
|  | 4 | 3 |  |  |   |
|  | 2 |  |  |  |   |
|  | 5 |   |   |   |   |
| **Average:** | 4.20 | 4.63 | 5.17 | 3.83 | 7.50 |
|  |  |  |  |  |  |
| **Standard GA Trial 3** |  |  |  |  |
| **Generation:** | 1 | 2 | 3 | 4 | 5 |
|  | 4 | 6 | 7 | 1 | 5 |
|  | 7 | 4 | 5 | 2 | 8 |
|  | 5 | 4 | 5 | 5 | 5 |
|  | 4 |  | 7 | 6 | 8 |
|  | 8 |  |  |  | 7 |
|  | 5 |  |  |  | 1 |
|  | 7 |  |  |  | 6 |
|  | 3 |  |  |  | 3 |
|  | 3 |  |  |  | 6 |
|  | 1 |   |   |   |   |
| **Average:** | 4.70 | 4.67 | 6.00 | 3.50 | 5.44 |
|  |  |  |  |  |  |
| **Mutation-based GA** |  |  |  |  |
| **Generation:** | 1 | 2 | 3 | 4 | 5 |
|  | 2 | 3 | 4 | 8 | 4 |
|  | 7 | 5 | 7 | 2 | 8 |
|  | 2 | 3 | 1 | 4 | 3 |
|  | 2 | 3 | 3 | 6 | 2 |
|  | 7 | 4 | 6 | 1 | 3 |
|  | 3 | 3 | 3 | 3 | 6 |
|  | 6 | 7 | 5 | 4 | 1 |
|  | 3 | 6 | 4 | 2 | 7 |
|  | 3 | 3 | 1 |  | 7 |
|  | 3 | 2 | 4 |   |   |
| **Average:** | 3.80 | 3.90 | 3.80 | 3.75 | 4.56 |
|  |  |  |  |  |  |
| **Random Motions** |  |  |  |  |
| **Generation:** | 1 | 2 | 3 | 4 | 5 |
|  | 3 | 2 | 2 | 4 | 3 |
|  | 5 | 6 | 1 | 2 | 3 |
|  | 4 | 2 | 1 | 5 | 2 |
|  | 3 | 3 | 1 | 4 | 5 |
|  | 2 | 4 | 4 | 1 | 2 |
|  | 3 | 4 | 3 | 7 | 3 |
|  | 3 | 2 | 4 | 5 | 1 |
|  | 4 | 2 | 1 | 4 | 2 |
|  | 2 | 4 | 1 | 7 | 1 |
|  | 3 | 3 | 2 | 3 | 3 |
| **Average:** | 3.20 | 3.20 | 2.00 | 4.20 | 2.50 |

# Source Code

The code below was added to the existing ‘GuidebotupperPlayMusic’ project to support the new IGA-based music generation. This includes a new (temporary) main() function, used for testing, and new RhythmGenerator.h and RhythmGenerator.cpp files, which contain the new RhythmGenerator class.

## Modifications to main.cpp

// This is a new main() function, for testing the RhythmGenerator IGA.

int main()

{

 srand(time(NULL));

 speech \*newspeak = new speech;

 Gestures \*thisgestures = new Gestures;

 thisgestures->ServoSetup();

 //move arm to right position and ready for playing

 Readfile \*newfilereader = new Readfile("script/MoveArm");

 newfilereader->process(thisgestures,newspeak);

 RhythmGenerator generator(thisgestures);

 generator.RunIGA();

 return 0;

}

## RhythmGenerator.h

// Brad Pitney

// ECE 578

// Fall 2013

// The RhythmGenerator class implements an interactive genetic algorithm, which is used to evolve

// movement sequences for playing a stringed instrument attached to the front of the Countess Quanta

// robot.

#ifndef RHYTHMGENERATOR\_H

#define RHYTHMGENERATOR\_H

#include <ctime>

#include <cstdlib>

#include <iostream>

#include <fstream>

#include <string>

#include "gestures.h"

using namespace std;

// IGA parameters

////////////////////////////////////////////////////////

// The size of the move sequence stored in each individual of the population.

const int numberOfMoves = 10;

// The number of individuals in the population. Note: this value must be an even number.

const int populationSize = 10;

// Total GA iterations.

const int totalGenerations = 5;

// Chance of applying crossover to each pair of parents.

const double recombinationProbability = 0.6;

// Chance of applying mutation to each gene.

const double mutationProbability = 0.01;

// Standard GA parameters

//const double recombinationProbability = 0.6;

//const double mutationProbability = 0.01;

// Mutation-based GA parameters

//const double recombinationProbability = 0;

//const double mutationProbability = 0.30;

// Random generation parameters

//const double recombinationProbability = 0;

//const double mutationProbability = 1.0;

////////////////////////////////////////////////////////

// Motion parameters

////////////////////////////////////////////////////////

// Range of allowed positions for servo::Wrist\_right.

const int minAllowedWristPosition = 600;

const int maxAllowedWristPosition = 1300;

// Starting location of servo::Wrist\_right, prior to evaluating a move sequence.

const int defaultWristPosition = 950;

// Positions of servo::Elbow\_Extension which result in raising and lowering the hand onto the intrument strings.

const int handRaisedPosition = 720;

const int handLoweredPosition = 884;

// The delay in microseconds between each move in a sequence, during evaluation.

// This value should be in increments of 100000, since this is the increment used when storing moves to a file.

const int usecMoveDelay = 100000;

////////////////////////////////////////////////////////

// Move sequences that are saved are stored in this file.

const string saveFileName = "script/IGA\_Rhythms.txt";

// Enables automatic storage of the rating values that the user selects when evaluating each individual.

const bool storeRatings = true;

const string ratingListFileName = "script/IGA\_RatingList.txt";

class RhythmGenerator

{

public:

 RhythmGenerator(Gestures \*objGestures);

 virtual ~RhythmGenerator();

 void RunIGA();

private:

 struct Gene

 {

 bool toggleHandState;

 int wristPosition;

 };

 struct Chromosome

 {

 Gene gene[numberOfMoves];

 int fitness;

 int rouletteValue;

 };

 bool m\_handRaised;

 Chromosome m\_population[populationSize];

 Gestures \*m\_objGestures;

 ofstream m\_saveFile;

 int m\_totalMoveDelay;

 void GenerateRandomMove(Gene &move);

 void PerformMove(servo::servo\_name servoID, unsigned short position, bool writeToFile);

 void PlaySong(int individual, bool writeToFile);

 int EvaluateFitness(int individual, bool reviewMode);

};

#endif

## RhythmGenerator.cpp

// Brad Pitney

// ECE 578

// Fall 2013

// The RhythmGenerator class implements an interactive genetic algorithm, which is used to evolve

// movement sequences for playing a stringed instrument attached to the front of the Countess Quanta

// robot.

#include "RhythmGenerator.h"

RhythmGenerator::RhythmGenerator(Gestures \*objGestures)

{

 m\_objGestures = objGestures;

 m\_handRaised = true;

}

RhythmGenerator::~RhythmGenerator()

{

}

// This method randomly generates either a hand state toggle move or a wrist move, with a 50% chance of each.

// If a wrist move is selected, then the new wrist position is randomly generated within the allowed wrist range.

void RhythmGenerator::GenerateRandomMove(Gene &move)

{

 bool toggleState = ((rand() % 2) == 1);

 move.toggleHandState = toggleState;

 if (!toggleState)

 {

 move.wristPosition = minAllowedWristPosition + (rand() % (maxAllowedWristPosition - minAllowedWristPosition + 1));

 }

 else

 {

 move.wristPosition = 0;

 }

}

// This method makes the call to actually move a servo to the specified position. If writeToFile is set,

// it instead writes the move description to the save file.

void RhythmGenerator::PerformMove(servo::servo\_name servoID, unsigned short position, bool writeToFile)

{

 if (writeToFile)

 {

 m\_saveFile << "%" << servoID << "\_" << position << "\_" << m\_totalMoveDelay << "%";

 m\_totalMoveDelay += (usecMoveDelay / 100000);

 }

 else

 {

 m\_objGestures->SetServo(servoID, position\*4); // SetServo positions are 4x those displayed on the Maestro Control Center.

 usleep(usecMoveDelay);

 }

}

// This method executes the move sequence specified by the selected individual (i.e. chromosome).

void RhythmGenerator::PlaySong(int individual, bool writeToFile = false)

{

 if (writeToFile)

 {

 cout << "Saving song to file '" << saveFileName.c\_str() << "'.\n";

 m\_saveFile.open(saveFileName.c\_str(), std::ofstream::app);

 m\_totalMoveDelay = 0;

 }

 else

 {

 cout << "Playing song number " << individual << ".\n";

 usleep(usecMoveDelay);

 }

 // Move to initial position.

 PerformMove(servo::Elbow\_Extension, handRaisedPosition, writeToFile);

 PerformMove(servo::Wrist\_right, defaultWristPosition, writeToFile);

 m\_handRaised = true;

 // Step through the chromosome, performing each move.

 for (int j = 0; j < numberOfMoves; j++)

 {

 if (m\_population[individual].gene[j].toggleHandState)

 {

 // Raise or lower hand.

 if (m\_handRaised)

 {

 // Lower hand onto strings.

 cout << "Lowering hand onto strings.\n";

 PerformMove(servo::Elbow\_Extension, handLoweredPosition, writeToFile);

 }

 else

 {

 // Raise hand off of strings.

 cout << "Raising hand off of strings.\n";

 PerformMove(servo::Elbow\_Extension, handRaisedPosition, writeToFile);

 }

 m\_handRaised = !m\_handRaised;

 }

 else

 {

 // Move hand left or right.

 if (m\_handRaised)

 {

 // If hand is above strings, check the next move to ensure that we're not wasting time moving the hand multiple times

 // without touching the strings. Skip this move if the next move will just move the hand to another coordinate.

 if ((j+1 >= numberOfMoves) || !m\_population[individual].gene[j+1].toggleHandState)

 {

 cout << "Skipping superfluous move to " << m\_population[individual].gene[j].wristPosition << "\n";

 }

 else

 {

 // Move hand.

 cout << "Repositioning hand to " << m\_population[individual].gene[j].wristPosition << "\n";

 PerformMove(servo::Wrist\_right, m\_population[individual].gene[j].wristPosition, writeToFile);

 }

 }

 else

 {

 // If hand is on the strings, always move hand.

 cout << "Strumming strings to " << m\_population[individual].gene[j].wristPosition << "\n";

 PerformMove(servo::Wrist\_right, m\_population[individual].gene[j].wristPosition, writeToFile);

 }

 }

 }

 if (writeToFile)

 {

 m\_saveFile << "\n";

 m\_saveFile.close();

 }

}

// This method evaluates the fitness of an individual by first playing the song, and then prompting the user to

// rate the performance. Options to repeat the song or to save the song to a file are also presented. if reviewMode

// is set, then the method just plays the song and offers to repeat/save, without requesting a rating.

int RhythmGenerator::EvaluateFitness(int individual, bool reviewMode = false)

{

 bool writeFile = false;

 // If this individual's fitness was already evaluated during a previous generation, then just return this fitness.

 if (m\_population[individual].fitness != 0 && !reviewMode)

 {

 return m\_population[individual].fitness;

 }

 while (true)

 {

 PlaySong(individual, writeFile);

 writeFile = false;

 bool repeatPrompt = false;

 do

 {

 repeatPrompt = false;

 if (reviewMode)

 {

 cout << "Type 'c' to continue to the next song.\n";

 }

 else

 {

 cout << "Please type 1-9 to rate the song (1 = very bad song, 9 = very good song):\n";

 }

 cout << "Type 'r' to repeat the song.\n";

 cout << "Type 's' to save the song to the '" << saveFileName << "' file.\n";

 char inputChar;

 cin >> inputChar;

 int inputValue = atoi(&inputChar);

 if (inputChar == 'r' || inputChar == 'R')

 {

 // Repeat the song.

 continue;

 }

 else if (inputChar == 's' || inputChar == 'S')

 {

 // Save the song to the file.

 writeFile = true;

 continue;

 }

 else if (inputValue >= 1 && inputValue <= 9)

 {

 if (reviewMode)

 {

 // Ignore ratings while in review mode.

 repeatPrompt = true;

 }

 else

 {

 // Store the selected rating, if this option is enabled.

 if (storeRatings)

 {

 ofstream ratingListFile;

 ratingListFile.open(ratingListFileName.c\_str(), std::ofstream::app);

 ratingListFile << inputValue << "\n";

 ratingListFile.close();

 }

 // Return the selected rating as the fitness evaluation.

 return inputValue;

 }

 }

 else if (inputChar == 'c' || inputChar == 'C')

 {

 // 'continue' is only used in review mode.

 if (reviewMode)

 {

 return 0;

 }

 else

 {

 repeatPrompt = true;

 }

 }

 else

 {

 repeatPrompt = true;

 }

 } while(repeatPrompt);

 }

}

// This method manages the interactive genetic algorithm process by creating a random population and evolving

// the population for the selected number of generations. During each generation, the fitness of each individual

// in the population is first determined by performing each move sequence and gathering fitness ratings from the

// user. These user-selected fitness values are then used in roulette wheel parent selection. Recombination and

// mutatation are applied to create the population for the next generation.

void RhythmGenerator::RunIGA()

{

 // Generate random population.

 for (int i = 0; i < populationSize; i++)

 {

 for (int j = 0; j < numberOfMoves; j++)

 {

 GenerateRandomMove(m\_population[i].gene[j]);

 m\_population[i].fitness = 0;

 }

 }

 // Evolve the population for the selected number of generations.

 for (int numGen = 0; numGen < totalGenerations ; numGen++)

 {

 cout << "\n\n\*\*\* Beginning evaluation of Generation " << numGen << " \*\*\*\n\n";

 if (storeRatings)

 {

 ofstream ratingListFile;

 ratingListFile.open(ratingListFileName.c\_str(), std::ofstream::app);

 ratingListFile << "\nRatings for Generation " << numGen << ":\n";

 ratingListFile.close();

 }

 // Step through population, evaluating the fitness of each individual.

 for (int i = 0; i < populationSize; i++)

 {

 m\_population[i].fitness = EvaluateFitness(i);

 }

 cout << "\n\*\*\* Creating new songs \*\*\*\n\n";

 // Select parents for next generation.

 // Create roulette wheel.

 int totalFitness = 0;

 for (int i = 0; i < populationSize; i++)

 {

 totalFitness += m\_population[i].fitness;

 m\_population[i].rouletteValue = totalFitness;

 }

 // Create parent list.

 int parentList[populationSize];

 for (int i = 0; i < populationSize; i++)

 {

 int randValue = (rand() % (totalFitness + 1));

 for (int j = 0; j < populationSize; j++)

 {

 if (randValue <= m\_population[j].rouletteValue)

 {

 parentList[i] = j;

 break;

 }

 }

 }

 // Apply 2-point crossover.

 Chromosome newPopulation[populationSize];

 // Step through parent pairs.

 for (int i = 0; i < populationSize; i += 2)

 {

 if (((double)rand() / RAND\_MAX) < recombinationProbability)

 {

 // If recombination chance succeeds, then apply crossover.

 // Select 2 random crossover points in the chromosome.

 int point1 = rand() % numberOfMoves;

 int point2 = rand() % numberOfMoves;

 // Reorder these points, if necessary, so that point1 comes first.

 if (point2 < point1)

 {

 int temp = point1;

 point1 = point2;

 point2 = temp;

 }

 // Apply crossover to create offspring.

 for (int j = 0; j < numberOfMoves; j++)

 {

 if (j < point1 || j > point2)

 {

 newPopulation[i].gene[j] = m\_population[parentList[i]].gene[j];

 newPopulation[i+1].gene[j] = m\_population[parentList[i+1]].gene[j];

 }

 else

 {

 newPopulation[i+1].gene[j] = m\_population[parentList[i]].gene[j];

 newPopulation[i].gene[j] = m\_population[parentList[i+1]].gene[j];

 }

 }

 // Set fitness to 0 to mark these individuals for re-evaluation.

 newPopulation[i].fitness = 0;

 newPopulation[i+1].fitness = 0;

 }

 else

 {

 // If recombination chance fails, then copy over parents unchanged.

 for (int j = 0; j < numberOfMoves; j++)

 {

 newPopulation[i].gene[j] = m\_population[parentList[i]].gene[j];

 newPopulation[i+1].gene[j] = m\_population[parentList[i+1]].gene[j];

 }

 // Individuals are unchanged, so use the same fitness as before.

 newPopulation[i].fitness = m\_population[parentList[i]].fitness;

 newPopulation[i+1].fitness = m\_population[parentList[i+1]].fitness;

 }

 }

 // Apply mutation.

 for (int i = 0; i < populationSize; i++)

 {

 for (int j = 0; j < numberOfMoves; j++)

 {

 if (((double)rand() / RAND\_MAX) < mutationProbability)

 {

 GenerateRandomMove(newPopulation[i].gene[j]);

 // Set fitness to 0 to mark this individual for re-evaluation.

 newPopulation[i].fitness = 0;

 }

 }

 }

 // Copy newPopulation to population.

 for (int i = 0; i < populationSize; i++)

 {

 for (int j = 0; j < numberOfMoves; j++)

 {

 m\_population[i].gene[j] = newPopulation[i].gene[j];

 m\_population[i].fitness = newPopulation[i].fitness;

 }

 }

 }

 // Review final population.

 cout << "\n\n\*\*\* Finished Evolving \*\*\*\n";

 cout << "\*\*\* Now reviewing final population. Last chance to save songs! \*\*\*\n";

 for (int i = 0; i < populationSize; i++)

 {

 EvaluateFitness(i, true);

 }

}