

Devin Wolfe

12/10/2012

ECE478

Final Project: Robot Facial Expressions

Table of Contents

Introduction	3
Description of Genetic Algorithm	3
Software Implementation	5
Hardware Implementation	11
Control Software	12
Results	14
Discussion and Conclusions	16
Source Code	17

Introduction

This project was originally supposed to be a collaborative effort between myself and another teammate. From our discussions, the scope of the project was going to include speech recognition that would trigger facial expressions. Unfortunately, my partner dropped out of the class with only about two weeks left, and took all of his work with him

As a result, I had to redefine the scope of the project, omitting the speech recognition portion. What was left was a genetic algorithm to iteratively develop facial expressions for the Niels Bohr robot. The data from the algorithm would also be used to set the parameters for a randomized expression generator. I also intended to interface with the robot by writing the output from my algorithm software directly to a file that could be used by the control application. Even without the speech recognition, this feature set turned out to be a handful in itself, as I had to develop, program and test the software, repair the robot, and integrate the two somehow.

Ultimately, the only thing missing was a closed feedback loop, as the output from the algorithm program needed to be entered into the control application manually. I will explain the progress of my work and the reasons for the lack of this or any other features later on.

Description of Genetic Algorithm

The algorithm I used is based on genotypes consisting of a set of 6 genes, each gene representing the position of a servo in the robot's face. The servos that I chose for the genotypes were left and right eyebrow up/down, eyes up/down, left and right corner of mouth up/down, and mouth open/closed. I chose these particular servos because they are all, with the exception of the eye position, required to form the basic static facial expressions: happiness, sadness, anger, and surprise. The position of the eyes can add an extra layer of subtlety or direction to these expressions, but are not directly necessary to convey the expressions.

The genotypes made up of these genes are grouped into an array called a generation, and they start out with completely random values. In every generation, there is a feedback phase, during which the genotypes are displayed on the robot's face, and one or more human observers rates the expression generated by the genotype for its accuracy or appropriateness at conveying the desired expression. I decided to only evaluate one expression per generational line, because trying to evaluate multiple expressions per generation becomes very complex to program, and also slows the process by which the genotypes increase in fitness over the generations. The slowing is due to the limited number of genotypes per generation

- the default value is 20, because the time it takes to evaluate an expression is very large compared to a genetic algorithm where the computer evaluates fitness based on a pre-determined fitness function. To evaluate more than one expression in a generation, the generation would have to be effectively split into two sub-generations, one per expression, because the expressions evaluated become mutually exclusive as their fitness increases. That is to say, a happy face becomes a poorer sad face as it becomes more happy, and vice versa; although the expressions can start out muddled, perhaps looking both sad and angry, as they become more accurate in portraying one emotion the expressions become less and less appropriate examples of the other emotions.

In order to put selective pressure on the genotypes to improve their fitness, three processes are used - elitism, crossover, and mutation. Elitism in this case means that a variable number of the most fit expressions per generation are allowed to continue without modification into the next generation. The number of genotypes that are allowed to continue in this way can be defined by the user, but is set by default to 2. Genotypes above a certain level of fitness that are not selected as "elite" are allowed to mingle their genes through the crossover function, producing composite "children" of the next generation. For each two genotypes being crossed over, the exact genes that are swapped will be different. The total number of genes that are swapped, and the particular genes that are swapped are both randomly chosen. Following the crossover process, the child genotypes are also subjected to mutation, where a random number of randomly selected genes are changed in value by a random number, which is either added or subtracted from the original gene value. The remainder of the unfit genotypes simply have all of their genes randomized, and better luck next time.

The result of successive generations being subjected to these processes are genotypes of an increasing fitness, as well as a large amount of data, as the values of every gene in the genotypes of a generation can be written to file at the end of every generation. Part of the algorithm program are two functions that search the data from a generation, and for a given level of fitness, determine the minimum and maximum values for each gene in the genotypes that meet or exceed the chosen level of fitness. In this way, the range of servo values are found which correlate strongly with the expression being evaluated. In the second function, the ranges generated by the first function are used to generate a genotype with random gene values that fall between the ranges. This is, in effect, a random expression generator, which will generate random variations of the expression that was evaluated by the generation of genotypes from which the ranges were taken. Because all of the possible gene values correspond to a high level of fitness, each genotype generated in this way should also have a fitness that meets or exceeded the chosen level.

Software Implementation

The software implementing this algorithm was coded entirely in C++, using object oriented programming techniques. The data structure used to store the generation data is a two dimensional array, a matrix, with columns equal to the number of genes plus a field for fitness to be stored, and rows equal to the number of genotypes per generation. This matrix is stored in the private data of a class, the public members of which are the various functions of the genetic algorithm. The following is a complete list of the functions implemented, and a description of how they operate.

Basic genetic algorithm functions -

`int Init_Generation()` - This function is called once at the beginning of a new line of generations. It initializes the arrays used to store the genotype data, and generates the defined number of genotypes and genes with random values between 0 and 254.

`int Randomize(int Gtype)` - This function randomizes the genes of a single genotype, specified by the generation array member `Gtype`. While I was writing the code, I was thinking about the way that my algorithm would attempt to increase the fitness of the genotypes from generation to generation. I decided that only using elitism - carrying the most fit genotypes over to the new generation unchanged - to improve the fitness of the group might be insufficient. Instead of crossing over and mutating the remaining genotypes, it might be desirable to instead cross over and mutate only those genotypes that got a fitness rating above a certain level. Genotypes of insufficient fitness, instead of passing their poor genes on, could then be randomized for a chance at a better combination.

`int Crossover(int GtypeA, int GtypeB)` - This crossover function swaps the genes from the genotypes identified by the generation array members `GtypeA` and `GtypeB`. It functions in the same way that I originally designed it, crossing over a random number of random genes up to a defined number of genes, set with `NUM_CROSSOVER` in the header. It does this by incrementing a for loop that steps through each gene position for the genotype arrays. For each gene position, if the number of genes that have been selected is less than the specified maximum, the function either selects that gene for crossover or leaves it alone. For the selected gene positions, the genes from the parents are swapped. This actually changes the

values of the parent genes, because only a single two-dimensional array is used to store genotypes. The transition from one generation to the next takes place by changing the genotypes in the current generation, not by creating child genotypes in a new array.

`int Mutate(int Gtype)` - This mutation function mutates the genotype identified by the generation array member `Gtype`. It functions in the same way as originally specified, mutating a random number of random genes within a predefined range, which is set with `NUM_MUTATE`. I did however change the way the actual mutation occurs; instead of increasing the gene value by a percentage, I decided to instead change the gene value by a percentage of the maximum, 254. This eliminates situations where genes with low values do not change by an appreciable amount, basically becoming stuck low over generations.

`int Feedback(int Gtype, char Expression[])` - This function collects and stores feedback for a single genotype. The feedback data is stored into a parallel array that mirrors the generation array, with the same number of "rows" representing genotypes. For every genotype, there are a set number of fields for feedback in the form of integers. The variable controlling the number of fields is `FEEDBACK_ARRAY_BYTES`, in the header. The feedback function gives the user a text prompt telling them to enter a value between 1 and 10, rating the appropriateness of the expression generated by the genotype at conveying the desired emotion. The array of characters 'Expression' passes the name of the expression being evaluated. This function can be called up to the number of times defined in the array length variable for a single genotype. Each successive call will store its data in the next available array location, if any.

`int Calc_Fitness(int Gtype)` - This function converts feedback data for a single genotype into an integer representing the fitness of the genotype. The fitness integer is the rounded result of a floating point average of all the entered feedback for the genotype. If there are less than the maximum number of feedback datum, only those data that have been entered are used to calculate the fitness, but in normal operation of the algorithm all data should be entered.

`int Find_Elites()` - After the feedback data has been converted into fitness values, this function can be called to find the most fit members of the current generation, and the values identifying those members are stored in a separate

array. The number of genotypes that are selected and stored this way can be set with the variable NUM_ELITES in the header.

int Find_Exp_Ranges(int Fitness) - For a given value of fitness, this function searches through the generation array and keep track of the highest and lowest values for each gene from the members of the generation whose fitness value is higher than the argument. So if the integer 6 is passed into this function, the ranges found will represent all members of the generation with a fitness of 6 or above. 3 values are stored in different arrays for each gene - High, Low, and Absolute. High and Low are self explanatory, and the Absolute value is just the difference between the High and Low values for that gene. The function finds these values by stepping through each genotype in the generation array, and comparing its fitness value to the function argument. If the fitness is greater than or equal to the argument, the function steps through each gene in that genotype. For each gene, the value is compared with High and Low. If the gene value is greater than High or less than Low, the gene value replaces High or Low, respectively. High starts at 0 for each gene, and Low starts at 0. It is possible for this function to generate erroneous values if either no feedback or feedback on only a single genotype had been entered when Calc_Fitness() function was called. If no feedback had been entered, Calc_Fitness would have failed, and all of the genotypes would have the default value of 0 for fitness. This would be easy to spot, because the range arrays would remain at their default values of 0 for High and 255 for Low. If only a single genotype had feedback, then both the High and Low ranges would have the same value for each gene, and the absolute values would be all zeros.

int Gen_Expression(char Filename[]) - Once the Find_Ranges function has been called for the current generation, this function can be called to generate, print, and write out to file a single random expression, with values that fall into the ranges determined by Find_Ranges(). The filename to write the expression to is specified with the argument Filename, and it is assumed to be a new file, as the file pointer starts at the beginning of the file and will overwrite any existing data. In order to generate the value for each gene, the random number generator finds a number between 1 and the absolute range for that gene. Then, the low range value for that gene is added, which yields a number in between the high and low range for that gene.

Read/Write functions -

These functions all do more or less the same thing, which is to read or write data from the generation array to file. The data can either be stored in its raw form, to load back into the generation array later, or converted into usable servo values. The read function only reads raw data, one generation at a time, but the write functions can output converted or unconverted data, and write either a whole generation or a single genotype at one time.

The write functions that convert the data first do so by using the minimum and maximum servo values entered with the `Set_Servo_Ranges` function to convert the data by scaling the raw value by the ratio between the raw total range, 254, and the real servo value range, servo maximum minus servo minimum. To this scaled value, the minimum value is added, completing the conversion.

The filename to write the genotype or generation to is specified with the argument `Filename`, and it is assumed to be a new file, as the file pointer starts at the beginning of the file and will overwrite any existing data.

Print functions -

There are several print functions, which all do the same basic thing - print data to the console. These are mostly for debug purposes, and can print the contents of a single genotype in raw or converted form, and entire generation in raw form, the array numbers of the current elites, if any, the entered feedback for each genotype, if any, and the ranges found from the last `Find_Ranges` function, if any.

Interface Functions -

`int Set_Servo_Ranges()` - Necessary for interfacing with the robot, this function is used to set the minimum, maximum and default servo values for each servo that is controlled. From the minimum and maximum values, the total range is also calculated and stored.

`int Write_Servo_Ranges(char * filename)` - Writes the current servo range values out to a file.

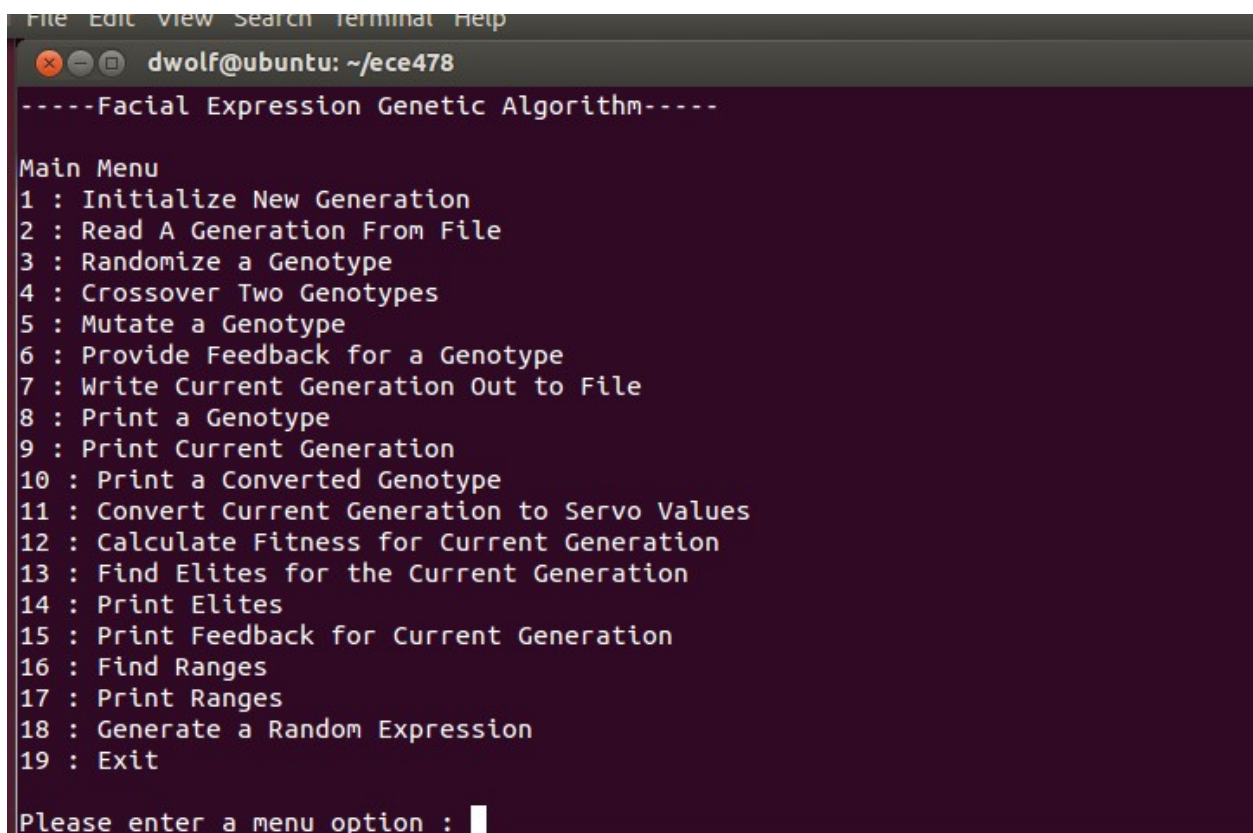
`int Read_Servo_Ranges(char * filename)` - Reads servo range values from a file.

`int Send_Gtype(int Gtype)` - This function takes the stored gene values for the selected genotype, converts them to raw servo values according to the entered servo ranges, and outputs them to be displayed directly on the robot's face.

int Display_Elites() - This function works like the Send_Gtype function, except that instead of outputting a specified genotype to the robot, the elite genotypes of the current generation are displayed, one after the other, switching on a keystroke.

int Reset_Servos() - Resets all of the servos on the robot that are being controlled to their default values, as entered in the Set_Servo_Ranges function.

Here are some screenshots from my algorithm program, showing some of the functionality.



```
File Edit View Search Terminal Help
dwolf@ubuntu: ~/ece478
-----Facial Expression Genetic Algorithm-----

Main Menu
1 : Initialize New Generation
2 : Read A Generation From File
3 : Randomize a Genotype
4 : Crossover Two Genotypes
5 : Mutate a Genotype
6 : Provide Feedback for a Genotype
7 : Write Current Generation Out to File
8 : Print a Genotype
9 : Print Current Generation
10 : Print a Converted Genotype
11 : Convert Current Generation to Servo Values
12 : Calculate Fitness for Current Generation
13 : Find Elites for the Current Generation
14 : Print Elites
15 : Print Feedback for Current Generation
16 : Find Ranges
17 : Print Ranges
18 : Generate a Random Expression
19 : Exit

Please enter a menu option : █
```

The main menu

```
dwolf@ubuntu: ~/ece478
Please enter a menu option : 9
Genotype 0:
24 192 137 15 185 168 0
Genotype 1:
66 245 174 32 185 63 0
Genotype 2:
152 251 11 59 157 166 0
Genotype 3:
34 164 111 173 11 67 0
Genotype 4:
220 204 155 69 23 138 0
Genotype 5:
3 47 202 12 190 4 0
Genotype 6:
180 128 250 99 160 180 0
Genotype 7:
35 185 48 46 244 77 0
Genotype 8:
84 150 113 196 69 125 0
Genotype 9:
135 161 74 35 102 97 0
Genotype 10:
45 105 16 120 118 206 0
Genotype 11:
124 170 79 246 142 112 0
Genotype 12:
171 177 169 91 95 30 0
Genotype 13:
169 179 181 27 247 122 0
Genotype 14:
152 127 28 98 34 130 0
Genotype 15:
67 207 108 84 199 98 0
Genotype 16:
162 68 13 114 187 155 0
Genotype 17:
98 230 204 12 67 44 0
Genotype 18:
42 236 96 223 8 215 0
```

A new generation

```
LibreOffice Writer
Please enter the number of the genotype to print.
[Integer from 0 - 19] : 0
Genotype 1: 24 192 137 15 185 168 0
```

A genotype

```
Please enter a menu option : 5

Please enter the number of the genotype to mutate.
[Integer from 0 - 19] : 0

Selected genes for mutation: 0, 12, 3,
Mutating genes...
Mutating gene 0 by 5 percent.
Mutation = -12.700000
Mutated from 24 to 11.
Mutating gene 1 by 14 percent.
Mutation = -35.560001
Mutated from 192 to 156.
Mutating gene 2 by 1 percent.
Mutation = -2.540000
Mutated from 137 to 134.
Mutating gene 3 by 13 percent.
Mutation = -33.020000
Mutated from 15 to 0.
```

Same genotype being mutated

```
Please enter a menu option : 8

Please enter the number of the genotype to print.
[Integer from 0 - 19] : 0

Genotype 1: 11 156 134 0 185 168 0
```

```
Please enter a menu option : 8

Please enter the number of the genotype to print.
[Integer from 0 - 19] : 1

Genotype 2: 66 245 174 32 185 63 0
```

Two genotypes before crossover

```
Please enter a menu option : 4

Please enter the number of the first genotype to cross over.
[Integer from 0 - 19] : 0

Please enter the number of the second genotype to cross over.
[Integer from 0 - 19] : 1

Selected genes for crossover: 0, 4,
Performing crossover...
```

Crossover procedure

```
LibreOffice Calc menu option : 9
Genotype 0:
66 156 134 0 185 168 0
Genotype 1:
11 245 174 32 185 63 0
Genotype 2:
```

Same genotypes after crossover

Hardware Implementation

The robot used in this project is the Niels Bohr robot from the PSU robot theatre. It consists of considerably more than just a face, having wheels to move around, a fully movable neck, various sensors and an arm with a gripping hand. However, in my case the face was the only part of interest. The robot's face is a rubber Halloween mask, with various attachment points for the servos that are used to generate expressions. When the servos move, the mask is deformed, moving the facial feature at the location where the servo is attached. The eyebrows and mouth are moved with servos that pull on wire lines, dragging the eyebrows up and the mouth down from their resting positions. The cheek servos have X-shaped attachments, one arm each of which is attached directly to the back of the mask at a point roughly corresponding to the cheek/corner of the mouth. The eyes have two degrees of freedom, able to look up/down and left/right. However, in the genotype of the algorithm, I only took into account the servo controlling the vertical movement of the eyes, as the horizontal movement is not as important at conveying emotion. One main shortcoming is that all of the servos are attached to the mask with small circles of Velcro, one circle on the servo and one on the mask. This means that after a certain level of tension, the servos simply detach from the mask,

limiting the range of motion. Unfortunately, the servos cannot be attached permanently to the mask, for two reasons. First, any adhesive strong enough to outperform the Velcro would damage the mask if someone were to attempt to remove it, and second, the mask must be removed in order to access any part of the face, for instance to replace a servo or adjust one of the control wires.

When I began to integrate the robot into my project, I found that time and neglect had left it in fairly sorry shape. Most of the Velcro tabs attaching the servos had peeled off of the mask, since they had been attached with the weak adhesive that they were supplied with on their backs. The mask itself had several tears in it, and was only held in position on the head by a single clip in the back. Once I had completed the basic repair work needed to reattach all of the servos, I went about trying to connect to the robot.

Control Software

Originally, this project was to have used the program Visual Show Automation in order to interface with the robot. I thought at the time that this was necessary, because I did not know how to either output on a serial COM port, or communicate the proper information to the robot's servo control chip. The system that I designed to use Visual Show was slow and clumsy. Instead of interfacing directly with the genetic algorithm software that I wrote myself, it would be necessary to generate servo values, print them out, and then enter them by hand into the VSA interface.

Ultimately, I found what I needed to have my genetic algorithm program communicate directly with the robot. The two required tools were a datasheet for the MiniSSC II servo control chip, and a free library that allows C and C++ programs to easily output bytes one at a time or in a string on serial COM ports, including emulated USB serial ports. The library is included in my program from 'Algorithm.h', titled 'rs232.c' and 'rs232.h'. The functions it defines are used in all of my functions that communicate directly with the robot. The functions in the library include excellent documentation, but basically they consist of functions to open and close a specified serial port, and to output either a single byte, or an array of bytes to that port. Only the latter is important for this project.

The MiniSSC II servo controller operates at 9600 baud, and accepts commands consisting of 3 bytes. The first byte is always equal to 0xFFFF, or decimal 255. The second byte is address of the servo on the chip, out of a total of 8 servos with one chip, or 16 with both chips connected. The third byte is the servo value, a number between 0 and 255, which is converted by the controller to reflect either a 90 or 180 degree range of motion. The range of motion depends on the position of a jumper on the chip.

So in lieu of VSA, I made functions to output a single genotype to the robot, to display the elite genotypes on the robot, and to reset the values of each servo to the set defaults. This is the basic functionality that allows each generated expression to be displayed in turn on the robot so that feedback can be entered as quickly as possible. The rotation for a generation would go like this:

1. Display expression on robot
2. Enter feedback
3. Repeat 1 and 2 for each expressions
4. Generate fitness values for each expression

Following this loop, the fitness values are used to generate the elites, and to determine which genotypes are crossed over and mutated, as opposed to simply randomized.

Results

I started by generating some random expressions using my algorithm program, and entering the servo values into VSA. What I found was that the range of the expressions generated was minimal; there was not a dramatic difference between the minimum and maximum positions. After fiddling with the servos and the attachment points on the mask, I was able to improve the expressions generated somewhat, but this system has some fundamental limitations, which I go into further in the discussion.

After playing with the program for a while, I ran 3 test generations in my program to try to generate some expressions. The expressions I evaluated for where happy, sad, and surprised. I found that I was able to get usable expressions after about 10 generations. On the following page are 3 of the best expressions that I was able to generate with my algorithm.

These results were obtained, laboriously, before I developed the new software for directly communicating with the robot. Transferring the generated expressions of a generation from the console program into VSA, entering feedback, and completing crossover and mutation took approximately ten minutes per generation. Accordingly, in order to run the number of generations it took to obtain these faces, about two hours were required. Using the new protocol, the time necessary to complete a generation should be closer to 3-5 minutes, a large improvement.



Happy" face



"Sad" Face



"Surprised" face

The randomized expression generator was somewhat disappointing when used with the actual robot. Although it works exactly as designed, the range of facial expressions generated using actual algorithm data was minimal, and although I could see and hear the servos moving from one expression to the next, I often had a hard time telling the expressions apart.

Discussion and Conclusions

The part of this project that worked the best was definitely the software implementation of the algorithm and random expressions generator. I was very happy to get some proof that the algorithm could evolve usable expressions, instead of a constant stream of random faces. And, based on testing the random expression generator with data from a generation containing some fit genotypes, I know from comparing the resulting generated expressions to the more fit examples in the generation that the random generator was doing its job. However, as I mentioned in the results, the difference between the expressions generated in this fashion were minimal when actually displayed on the robot.

The main limiting factor for good facial expressions and the optimal function of the facial servos is the mask used, and the way in which the servos are attached to the mask. The mask is made of a light rubber, but it is still not pliable enough to be dramatically deformed by the servos. This is due to three factors: actual mechanical resistance from the mask, the limited torque of the servos, and the sub-optimal method of attaching the servos. The first two factors could be dealt with easily, by replacing either the mask, the servos, or both. The mask should be replaced with one that is either thinner or made of a more stretchy material, and the servos should be upgraded to a model that can apply more force. As for the method of attaching the servos to the mask, this will be more difficult, and will require engineering an entirely new system. One possibility would be to use metal clasps on the servos, connected to rings attached permanently with strong adhesive to the back of the mask. This would allow the mask to be removed when necessary, but also allow the servos to exert their full force on the mask without detaching the connections as the Velcro is prone to do.

If I had more time to work on this robot, it would also be helpful to rethink the way in which the facial servos are oriented. For example, in their current orientation, the eyebrow servos and the mouth servo do not exert force in exactly the direction that the mask should deform. The eyebrow servos should pull the mask upwards, and the mouth servo should pull it downwards at its point of attachment, but all of these servos instead pull the mask inwards as well. This causes the eyebrows not to move as much as they could, and causes the lower lip

of the mask's mouth to pull inwards noticeably when it opens wider than about half an inch. A solution to this problem would be to mount pulleys to redirect the angle at which the wire control lines pull on the mask. Two pulleys could be mounted at the eyebrow level, one above each eye, and one pulley could be mounted in the space where the chin is.

Another improvement that I would make would be to add something like a rigid skeletal structure to the robot's face. Right now, the shape of the robot's head is roughly cuboid, and the lack of internal structure supporting the mask severely limits the realism of the robot's face.

The improvement to the algorithm program allowing direct communication with the robot was a big step forward for the project, but the time needed to evaluate a generation of 20 genotypes is still too long. A future implementation of this program would benefit from a GUI, allowing users to rate an expression with a single click, for instance having a slider acting like a number line from 0 to 10 that would generate a floating point value depending on where the user clicked. It does not take long for a user to evaluate an expression, because the only important metric is the overall emotional gestalt that the expression conveys. In short, the face either works or it doesn't, and it is not feasible or necessary to examine or rate each individual feature of the expression on how it contributes to the whole. The expression cannot be broken down into something like "happy right eyebrow, sad left eyebrow, surprised cheek, angry mouth." It is not the individual parts of the face that are happy, or sad, or surprised, it is the integrated whole. So, because a user can take in the face in an instant, it makes sense to allow them to commit their impression just as instantaneously.

If a genotype can be rated in a few seconds, then a generation could be rated inside of a minute. Because the elites, crossover, mutation and randomization are handled automatically, dealing with them takes less than a second. With a hypothetical one minute generation, usable expressions and data could be obtained in a few minutes, and refined results in under a half hour. A days work could yield commercial animatronic grade results, with a full set of static expressions.