Team 3: Marie Curie Head

Intelligent Robotics II
Winter 2020

by

Robert Holt
David Yakovlev

Project Overview

For the first half of the project, we had three major goals to complete. First was to accumulate and rebuild the Marie Curie head on a base of some kind. As the head was initially disassembled in different pieces, mechanical work to get it rebuilt was necessary. Once rebuilt, we needed to verify that the curie head was in a working state, which was done by powering the head and testing basic functionality such as movement and gestures.

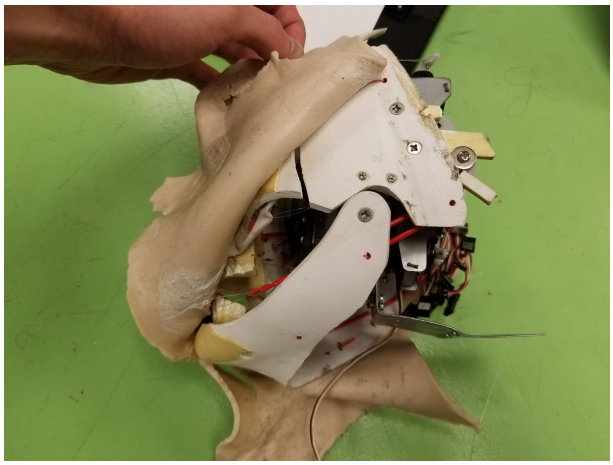
Second, after verifying that the physical components are in working order, the next step was to understand the computational platform included. This involved understanding how control is distributed to various modules and how the actuators / servos are controlled. Investigations into the architecture of the operating system and how the main computational platform interact with any servo driver were of interest in this phase. If the base computational platform is outdated, broken, or otherwise troublesome, this will be an opportunity to update it to something more modern and fitting of the programming skillset which this team has.

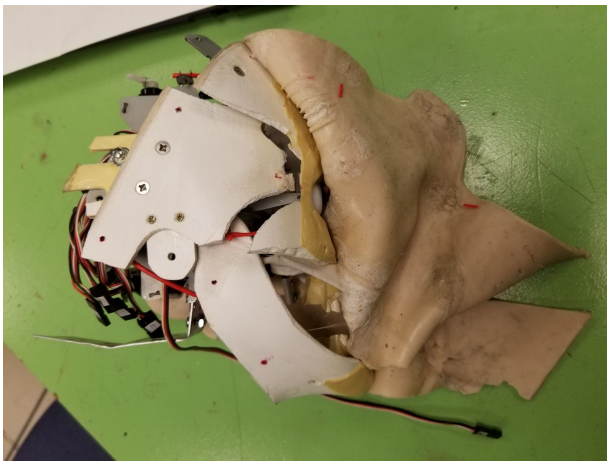
Finally, the first half of the project concluded by creating a code base which allows for the easy inclusion of new gestures. Gestures are represented in a form that allows for them to be easily tweaked and modified. Data structures were built around their representation so that a future user can theoretically add a new gesture without modifying core code to the bot. Several new gestures were developed and tested.

For the second half of the project, we mainly focused on the Marie Curie head replicating a facial expression that we give it. The given expression is either in real time through a webcam, or an image of a face through a jpeg file. We used a genetic algorithm to implement the learning of a new facial expression as we found it to be the most efficient method. The method included finding an initial population, fitness function, selection, crossover, and finally mutation. These will be talked about later in the report.

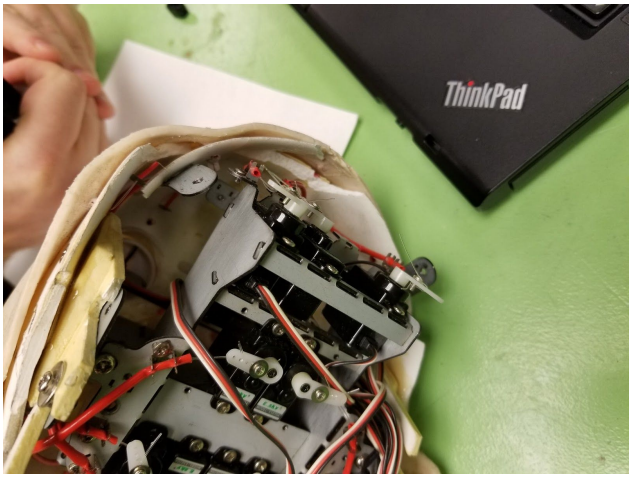
Initial State of Marie Curie Head

The initial state of the Marie Curie head was in a very bad shape, much more than we expected. The skull was completely broken, detached from the skull, on both sides of the face, as can be seen from the two images below. While the pieces were still attached to the metal bracket holding the servos, there was no connection to the main part of the face.

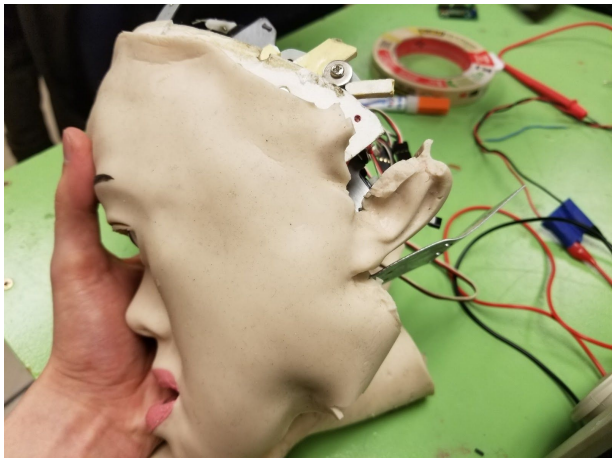


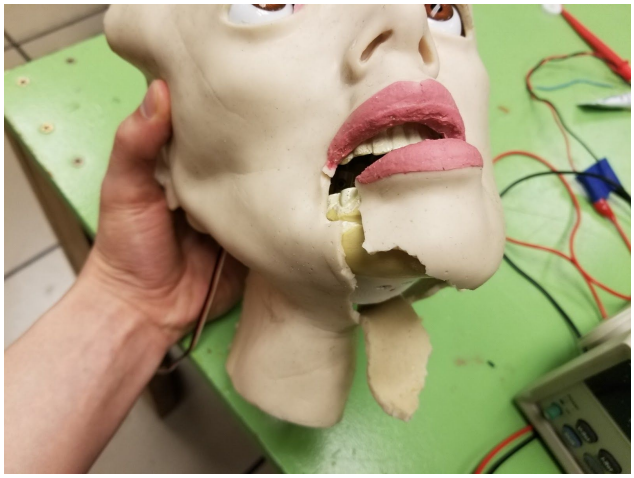


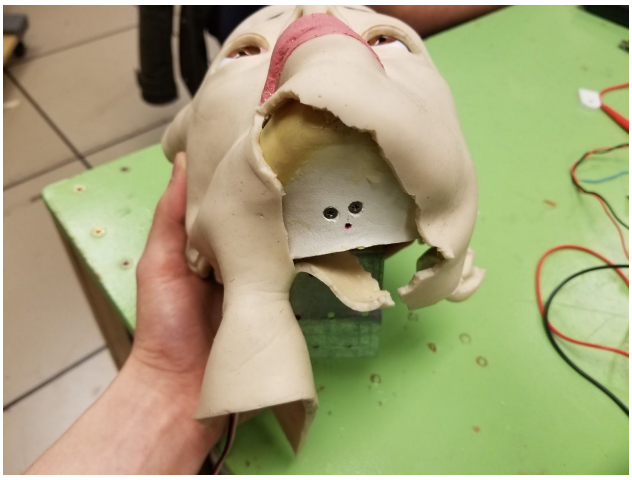
At the top of the head, there was a piece broken off from the back of the skull as can be seen in the image below. Along with the piece hanging on it's own there was a part of that piece that was missing, so when it was eventually repaired, there is still a missing chunk of skull that is not where it should be.



Then there was the case of the latex. First, the left ear was halfway detached from the face. Second, there was a major tear that split the latex all the way down from the right corner of the mouth to the very bottom. There appeared to be a missing piece of latex as well right where the tear begins at the corner of the mouth, which can be seen after it was repaired later in this report. Images of the damages can be seen below.







Along with the skull and latex damages, two servos were not functioning at all, the jaw joint connecting the jaw to a servo was broken, there was no Pololu board to interface with the servos, and a lot of the strings connecting the servos to the latex were either not connected or were connected very loosely and needed tensioning to properly create gestures on the face.

Repairs

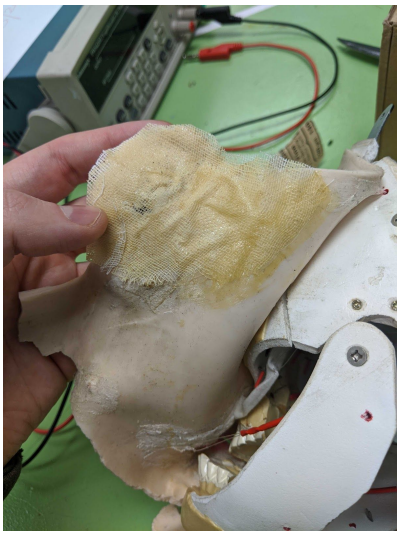
Skull Repair

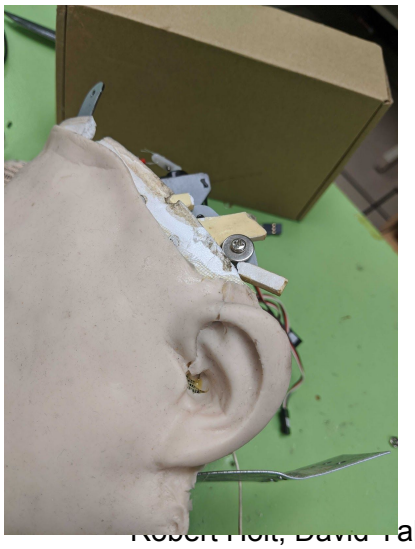
The top of the head was not able to be fully repaired as there was a missing piece from the initial skull damage. With the broken piece we had, we used epoxy to glue the broken piece to the skull, held it in place for about 5 minutes, then put a clamp on the center of the crack / damaged area, and let that sit overnight for at least 24 hours. We did make sure to clean off some of the excess glue on the skull before placing the clamp on so the clamp would not stick to the skull.

The two broken sections on the side of the skull were repaired in a similar fashion, though repaired is used very loosely. We used epoxy to glue the pieces to the skull, at separate times, held it together for 5 minutes and then let the skull sit for a minimum of 48 hours. We were not able to use the clamp for these pieces because of the angles at which the pieces were broken. Every time we tried to put the clamp on, it would move the skull out of position. With this method that we used, we were able to get the skull repaired well enough for the project but there are still some major cracks in the skull that we anticipate will probably break off in the not too distant future if it isn't handled very carefully.

Latex Repair

We found a link online (https://www.ehow.com/how_8050806_repair-latex-halloween-mask.html) that had instructions on how to repair latex so we used their method to repair the left ear. We recommend going to the link if you wish to replicate this repair but here is a brief summary: First we cleaned the area that was to be repaired with soapy water. Then brushed the torn area, and the area around it, with contact cement. Cut a piece of cheese cloth that covers the area you've applied the contact cement to and press that cloth firmly onto the latex. Let it sit for 24 hours. The results can be seen in the two images below.



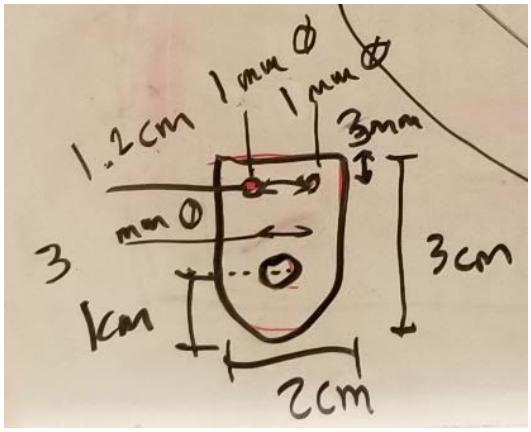


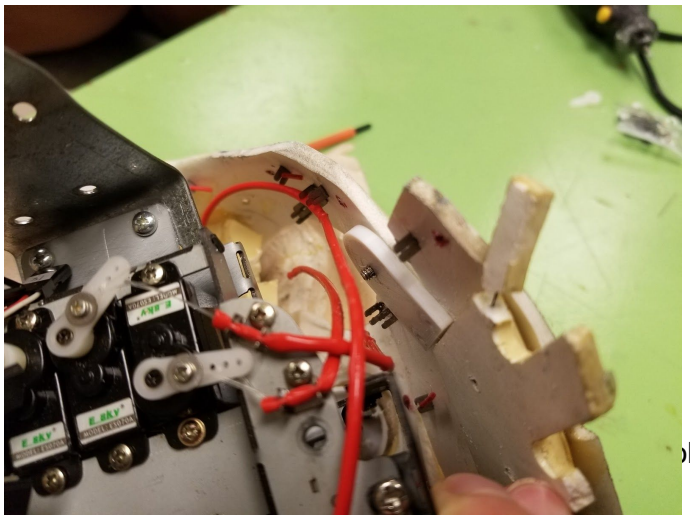
Unfortunately we could not use the same technique on the tear on the lower part of the face because the latex at the chin was connected to the skull using string that would have been much too difficult to remove and and reconnect. Since we could not reach enough of the backside of the latex to apply the same method as with the ear, we instead put small styrofoam blocks that we cut out behind the latex and used pins to hold the latex together. Then we applied superglue inside and around the tear in the latex and let it sit for 48 hours. Images of this method can be seen below. It doesn't look that great as there are still some gaps in the tear but it sufficed for what we need to do the project. If any future groups have to repair this same tear, we would recommend a different method if possible. (note: all materials used for repair were purchased on our own at Home Depot, none of it is found in the Robotic lab)

Other Repairs

- Initial Servos: we found two of the servos were completely dead at the start of the project through testing and needed to be replaced. One was the servo that controlled the eye moving left and right, which we purchased from: https://www.sparkfun.com/products/9065 (Generic (Sub-Micro Size) Servo). The other servo was the Futaba S148 Precision Servo: https://www.amazon.com/dp/B0015GZ8VE?psc=1&ref=ppx_pop_dt_b_asin_image which controlled the jaw.
- Additional Servo: Halfway through the project, we had Servo 11 spontaneously die and had to replace it with one of the Generic Sub-Micro Servos. To replace a servo, you need to remove the two screws that mount the servo to the metal bracket, connect the plastic servo horn to the new servo, tie the string to the horn, and clamp it down with a screw and washer into the second hole on the servo horn. You can always look at the other servos to see how it was done for reference.

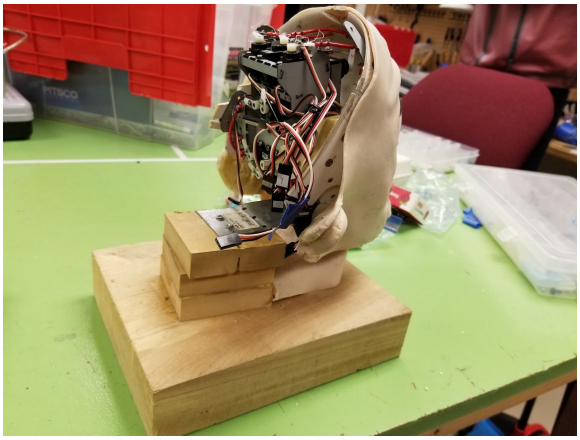
- Jaw joint linked to servo: There is a metal piece that connects a servo to the jaw to move it during a facial gesture. It was not connected because the metal piece was broken. We had to purchase a metal piece from Home Depot that fit the connectors on the servo and the jaw itself.
- Re-attach fabric to skin for oris control: On the backside of the latex, there are strings that connect to servos for almost all facial gestures. The ones that control the oris (near the cheekbone) have a piece of fabric that the strings are connected to for better control. This fabric had loosened up and needed to be glued back using superglue.
- Servo Controller Board: The head did not come with any kind of board to control the servos so we had to purchase our own. We went with the Mini-Maestro 18-channel: https://www.pololu.com/product/1354
- Tensioning strings: After we connected the controller board to each servo, we found that some of the strings needed to be tightened to give a wider range of control over the latex. This was done by untying the string on the servo and retying it a bit tighter.
- Jaw to skull attachment: There is a piece connected on both sides of the face to connect the jaw to the skull. The piece on the left side of the face was broken and we were not able to attach the jaw to that side of the face. What we did was take measurements of the piece on the right side (which can be seen below from an image we drew on a white board), 3D print a new piece using those measurements, remove the old broken piece, and finally attached the new piece to the side of the face. Images of the new 3D printed piece and it being attached to the side of the face can be seen below.





Outcome

After everything was repaired to the best of our abilities, we created a small wood mount to place the head on. The mount was three wooden blocks glued on top of each other, on a larger wooden platform. The final result wasn't necessarily beautiful, but it got the job done for what we need to accomplish for this project. Images below.



Additional Damage and Repairs During Project

While working on the Marie Curie head this term, there were additional damages that occurred to the head through the "wear and tear" of servos constantly moving the latex and wires simultaneously every which way. The first issue occurred after lengthy usage of the latex was that the corner of the mouth started to rip in a different direction than the other tear as can be seen in the image below.

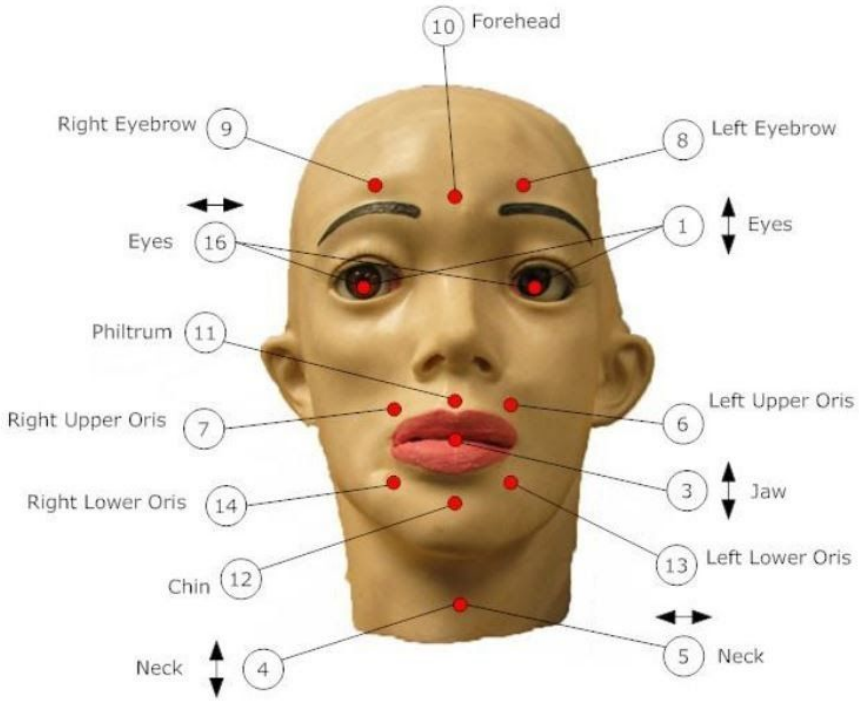
To stop the tear from getting worse, we used super glue to glue it back together. While it did prevent the tear from getting worse, it looked far from good or restored.

Another issue that came up through constantly using the servos was that some of the wires connecting the servos to the latex came undone on the servo side. Luckily this was an easy fix as all we had to was to tie the wires back to the servo.

Finally, there was an issue with the CV algorithm incorrectly detecting the face for the GA program we had written. The problem was that the ears were not ‘pinned back’ far enough, meaning they were sticking out from the head further than what is normal for an actual human, and that was causing the CV algorithm to detect the face farther out that what is normal. The solution to this was gluing cheese cloth to the back of the ears, which had already been done to one ear for a different repair. Then use a wire/string to tie one end through the cloth, and the other end goes through a hole in the skull and tied on that end as well. This pulls the ears closer to the skull, making it look more like a human face.

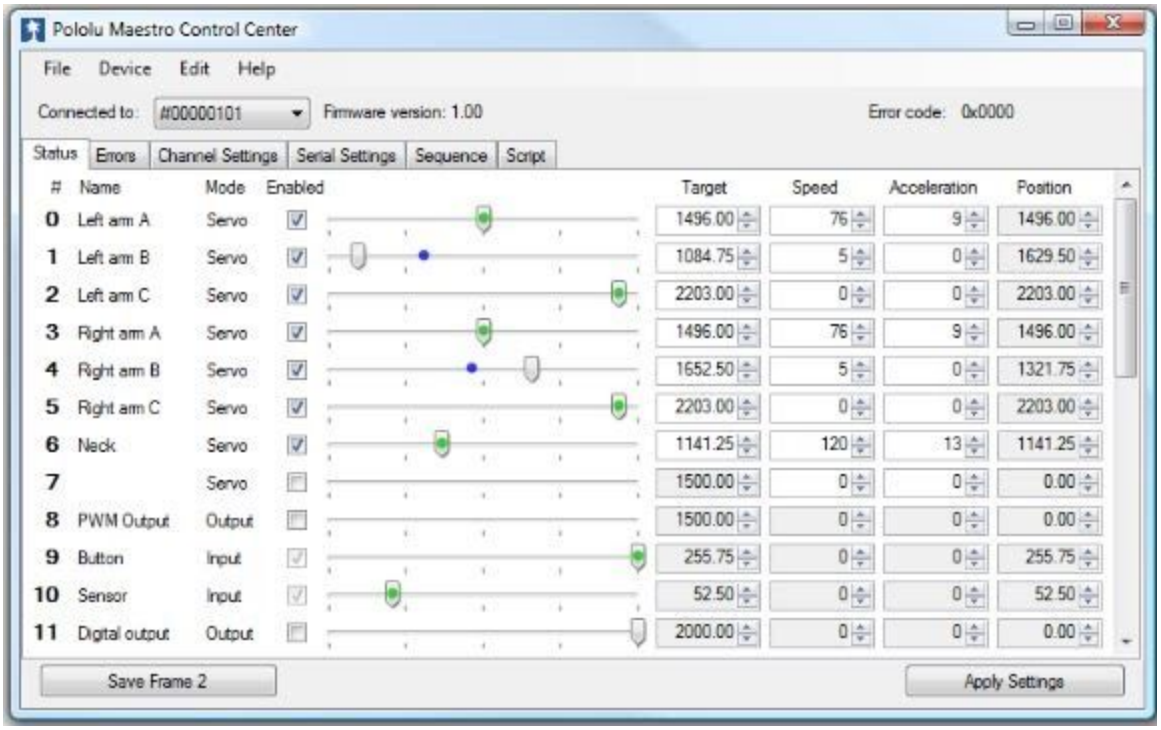
Testing Physical Components

Each of the servo connections are labeled as to which servo controls which part of the face. The image below, courtesy of another group’s report, shows how they are connected.



We took each connection, connected it to the Pololu board, and powered the board using the Tektronix PWS4205 power supply found in the lab at 5V / 2A. Then, using the Pololu Maestro Control Center, which can be seen in the image below, we tested each servo to verify that it was

not only working properly, but to see what its full range of motion is. Note: If the current ever goes near 2A while testing each servo, there is something wrong. Turn off the power supply and double check how you have everything connected. (eg. voltage and ground aren't swapped).



When running the scripts/code, we connected each servo to its own individual channel on the Pololu board so we could control them all at the same time for a full gesture. These are hard coded into the code but can easily be changed as the code is very readable. The servos were connected in the following order:

Channel 0 - Servo 0
Channel 1 - Servo 1
Channel 2 - Servo 3
Channel 3 - Servo 8
Channel 4 - Servo 9
Channel 5 - Servo 13
Channel 6 - Servo 14
Channel 7 - Servo 10
Channel 8 - Servo 6
Channel 9 - Servo 7
Channel 10 - Servo 11

Expression and Gesture Framework

Python and Pololu based Development

This project elected to use a Pololu Maestro 18 channel servo controller in order to actuate and interface with the Marie Curie servos. This choice was made because the Pololu Maestro has a convenient serial interface working over USB, and the team found an open source library which provided the necessary Python based API. The code for this library can be found at https://github.com/FRC4564/Maestro courtesy of Steven Jacobs with First Robotics Team #4564.

This API represents the Pololu as an object which has member functions for setting target positions (specified in quarter-microseconds), velocities, and accelerations for a given channel on the Maestro board. It also allows for software defined limits on the individual channel positions. For the Marie Curie project, the API was extended to support functionality for setting multiple targets in a single serial command. When setting positions for 18 or more servos, there is significant overhead if set target command are issued individually over the serial bus. Instead, the Maestro has an interface which allows a longer command for setting a contiguous range of channels simultaneously. Pololu states in their documentation that this functionality allows for servos in this contiguous range to be set 3x faster than with consecutive writes.

Documentation regarding the Pololu Maestro commands can be found here: https://www.pololu.com/docs/0J40/5.e.

Framework Oriented Approach

The primary goal in developing a method to represent expressions with Marie was to have a robust base that allowed for extensibility without changing said base software. An object oriented approach where a class called “Face” had various methods that manipulated a list of attributes which were instances of the newly created “Servo” class. When these instances are instantiated they access values via human readable YAML files which define their functionality.

An instance of Servo determines the following information by looking up the definition in the marie_servo_descriptions.yaml file which corresponds to the name passed to it in the constructor:

- **Controller Channel** - The constructor for a Servo instance provides the controller instance which the servo should use when setting a position, but the Servo instance relies on a numerical channel assignment in the marie_servo_descriptions file to determine which channel to request actuation through on the controller object.

- **Default Position** - The position which defines the neutral position of the servo. This is used when an expression is requested through the Face instance which does not define a position for this Servo instance.
- **Positions** - A dictionary where keys are intuitive names of the position (i.e. “open”, “closed”, “up”, “down”, “pulled back” etc) with values which are the numerical value of the position in quarter-microseconds.

An example can be seen below:

```
eye_l_r:
  channel: 0
  default_position: center
  positions: {center: 5604, left: 8000, right: 1984}
```

A developer can simply add the definition of a new servo to the YAML file when they add this servo physically to the Marie Curie head. When instantiating the Face instance, this new servo will be recognized and it can immediately be used in the definition of expressions.

Simplified Representation of Expressions

As was seen above, servos are defined in simple markup language (i.e. YAML) which is parsed by the framework in order to create objects and instances within the core algorithms to control the Marie Curie robot. Additionally, a similar approach was desired for representing an expression.

An expression is defined as a pose or fully defined list of positions for each servo. An expression has no time component (i.e. is not performed as a sequence of poses). This led to another YAML file which contains a human readable, easily modified, machine parsable definition for these expressions. Expressions are defined in expressions_gestures.yaml and are composed as in the example below:

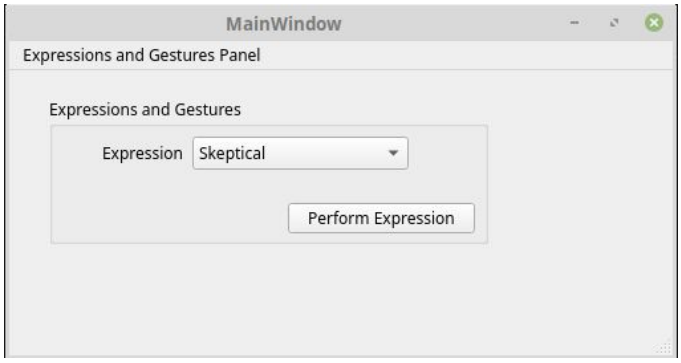
```
skeptical:
    eyebrow_l: up
    eye_l_r: right
```

A few things are immediately noticeable about the expression defined above. The first is that the expression is defined by an intuitive name which people can understand. The second is that elements defined underneath the name are indented and are the names of servos which are defined in marie_servo_descriptions.yaml. Associated with the servo names are the positions which these servos need to achieve in order to make this expression. As with the servo names, these positions are defined in marie_servo_descriptions.yaml. The final thing to notice is that not all servos have defined positions for a given expression. An expression must have at least one servo’s position defined, but not all must be defined. The behavior of the Face object, when actuating these expressions is defined when the request to perform the expression is called. If

the caller passes a value true for the default_positions parameter, then the undefined servos in the expression will be set to their default positions. Alternatively, a value of false for the default_positions parameter will cause these servos to not be set, thus they will remain at whatever position they were previously at before the “perform_expression” call was made.

Simple GUI for Testing Expressions

The final software goal for the first half of this project was to create a simple graphical user interface (GUI) which allowed for expressions to be actuated easily. It is easy enough to request expressions to be actuated via an interactive Python terminal, but we wanted to ensure that there was a minimal barrier to entry for quickly iterating on new expressions. Currently the functionality is limited to a simple combo box which lets the user select from the list of possible expressions (populated from expressions_gestures.yaml), and then execute the expression. There were plans to allow the user to actuate servos independently to make a kind of “expression builder” to generate and write new expressions to expressions_gestures.yaml, but this functionality was at a lower priority and was not accomplished.



The GUI was implemented using PyQt5, which has the same benefits as using Qt5 in C++ or other environments. A core reason for selecting PyQt5 was that it enabled us to use the Qt Designer which is a drag and drop tool for generating the Qt “.ui” files. The generated “.ui” file was then converted to python using the pyuic5 command line utility. The resulting Python file contains a class with member functions and attributes to get UI signals.

Expression Learning via Genetic Algorithm

Overview

Having a robot face make many different expressions can be useful for many different things so how one goes about implementing the way the robot learns these expressions is important to save on time and money. One way to teach the face to learn expressions is by using the Maestro Control Center to figure out which position each servo needs to be in to make a certain expression. This is obviously a very time consuming method as one has to go through each servo individually and figure out it’s best position.

The method we used instead is a genetic algorithm, which is a search heuristic that is inspired by Charles Darwin’s theory of natural evolution. The algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found. For this project, when we talk about a 'selection of fittest' individuals', we are referring to the best servo position for that specific servo to best replicate a given expression.

This method allows us to show the algorithm any face an expression for the Marie Curie head to replicate, and let the program take over.. We consider a set of solutions for a problem and select the set of best ones out of them.

Five phases are considered in a genetic algorithm, which will be discussed later in the report:

1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

Algorithm

The algorithm is described below, and starts with "Algorithm 1 Marie Curie Genetic Algorithm Overview" which calls Algorithm 2 and Algorithm 3 in a loop. The returned value from the algorithm is the optimal candidate which when implemented, was an instance of a class "Candidate" with members for that candidate's chromosome, score, and image which the scoring was performed on. Several meta-parameters are defined and used in the algorithms below. These meta-parameters are:

- **Number of Generations** - The number of generations to perform the evolution across. This can be improved in the future to be a threshold for convergence in the scores of the candidates.
- **Population Size** - Overall size of the population. At the end of each generation, the population will cull candidates which are the worst scoring until the population has this size.
- **Number of Candidates Bred Per Generation** - This is the amount of new candidates to breed, score, and merge into the population before culling the worst scoring.
- **Number of Candidates Bred Per Pair of Parents** - This is the number of candidates that a given set of parents should breed. For low values, this may cause problems as the best scoring parents may not be able to crossover their chromosomes enough times to find more optimal solutions. For high values there may also be problems because local minima may be found as most candidates may be coming from a set of only a few parents, thus promoting inbreeding.

- **Candidate Mutation variance** - variation in quarter microseconds to vary each gene (servo timing/position value) inherited from the parent chromosome. High amounts of variation allows new candidates to break out of potential local minima, but does not allow for tight refinement when trying to approach the optimal candidate (successive candidates are significantly different from the parents). We performed our experiments with a standard deviation of 50 us, which is typically around 1% - 5% of the total range of the servo position.

Algorithm 1 Marie Curie Genetic Algorithm Overview

Let $NUM\_GENS = 100$ represent the number of generations to evolve
Let $POP\_SIZE = 100$ represent the size of the total population
Let $NBPG = 50$ represent the number of new candidates to breed every generation
Let $\boldsymbol{S_{min}}$ and $\boldsymbol{S_{max}}$ represent the vectors of servo limits
Let L_R represent the landmarks in the reference image
Let $P = \{C_i, i \in \{0, 1, .., POP\_SIZE\}\}$ where each C_i is a candidate expression
for $i \in \{0, 1, .., POP\_SIZE - 1\}$ **do**
    Initialize $\mathbf{C_i}$**.chromosome** to $\mathcal{U}(\boldsymbol{S_{min}}, \boldsymbol{S_{max}})$
    Score each candidate s.t. $C_i.score = mc\_score()$
end for
for $g \in \{0, 1, .., NUM\_GENS\}$ **do**
    Let $N = \{N_i, i \in \{0, 1, .., NBPG\}\} = mc\_breed\_new(NBPG)$ be the set of new candidates for this gen
    Score N s.t. $N_i.score = mc\_score(\mathbf{N_i}$**.chromosome**$, L_R)$
    Set $C = C \cup N$
    Remove the $NBPG$ elements with the worst score from C
end for
return $C_{opt} = C_i$ where $i = \underset{i}{\text{argmin}}\{C_i.score\}$

Algorithm 2 MC Breed New (mc_breed_new(N_C, N_{CpP}))

Let N_C be the number of new candidates to breed
Let N_{CpP} be the maximum number of candidates to breed for a single set of parents
Calculate the number of pairs of parents N_{pP}
Get the mating pool M which is comprised of the best (lowest) scoring $2N_{pP}$ candidates in C
Create N_{pP} pairs of parents, P_p, by pairing the highest scoring parents together (i.e. $P_p = \{\{M_0, M_1\}, \{M_2, M_3\}, \dots\}$)
Initialize $N = \emptyset$ where N is the set of all new candidates bred this generation.
for $p \in P_p$ **do**
    for $j \in \{0, 1, \dots, N_{CpP} - 1\}$ **do**
        Let C_N represent the new candidate
        Randomize order of p
        Randomly select location i in which to perform crossover of chromosomes
        Set $\mathbf{C_N}$**.chromosome** $= [p_0[:i]^T, p_1[i:]^T]^T$ representing the concatenation of the two parent chromosomes sliced at index i
        Set $\mathbf{C_N}$**.chromosome** $= \mathcal{N}(\mathbf{C_N}$**.chromosome**$, 50^2)$ representing a mutation of the inherited chromosome with each gene varying in a normal distribution parameterized by $\boldsymbol{\mu} = \mathbf{C_N}$**.chromosome** and $\sigma^2 = 50^2$ where 50 is the standard deviation in quarter microseconds
        Saturate values in $\mathbf{C_N}$**.chromosome** to safe limits for servos
        Append C_N to N by $N = N \cup C_N$
    end for
end for
return N

Algorithm 3 MC Score Algorithm (mc_score(X, L'_R))

Let $M = 68$ be the number of landmarks recognized in an image (constrained by dlib landmark recognition algorithm)
Let $D = 2$ be the dimensionality of a single landmark
Let $L \in R^{D,M}$ represent the raw pixel valued landmarks
Let L'_R be the reference landmarks to score against
Let X be the chromosome to score
Actuate X on the face using the Maestro Controller
Capture an image, I_C, of the face
Get the bounding box B of the face
Detect the center of a rectangular bounding box, $B_C = [x, y]^T \in \mathbb{R}^2$ of the face
Detect raw landmarks for cand. image $L_C = dlib\_predictor(I_C).landmarks$
Calculate new "centered" landmarks $L_C^C = L_C - (B_C \otimes \mathbf{1}^T)$
Calculate the horizontal scaling by $x_s = B_{Right} - B_{C,x}$
Calculate the vertical scaling by $y_s = B_{Bottom} - B_{C,y}$
Get normalized landmarks $L'_C = L_C^C \oslash ([x_s, y_s]^T \otimes \mathbf{1}^T)$
Calculate difference in corresponding landmarks between the reference image and the candidate image $L_D = L'_C - L'_R$
return $\|L_D\|_F = \sqrt{tr(L_D^T L_D)}$ where $L_D^T L_D \in \mathbb{R}^{(M,M)}$ is the Gramian matrix of L_D and the diagonal elements of $L_D^T L_D$ represent the euclidean distance error of each landmark. This can also be expressed as the square root of the sum of the squared errors.

Scoring Algorithm

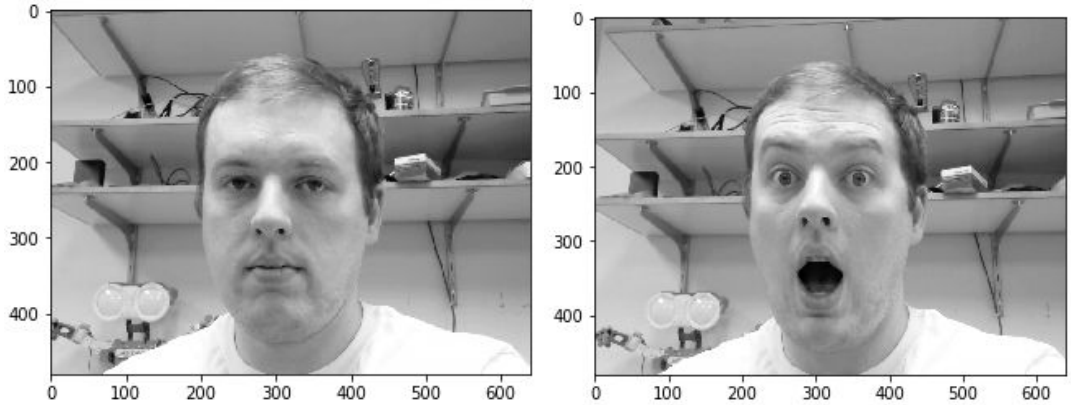
The scoring algorithm needed to be capable of comparing an image to another image to determine how well correlated the two images were. Instead of doing an intensity based image registration it was instead determined that a landmark approach would be preferable. This is because the robot only had an ability to actuate certain facial attributes such as eyebrows, forehead, mouth, eye position, and thus it did not make sense to compare things this whether the hair and background in the reference image correlated with the image of the robot.
We looked for an existing implementation of a facial landmark detection algorithm which would perform the landmark recognition and found a library called dlib (http://dlib.net) which had a Python implementation of a worthy landmark detector. The implementation is a face detector made using the Histogram of Oriented Gradients (HOG) feature combined with a linear classifier, an image pyramid, and sliding window detection scheme. We used a pretrained model found on the dlib website at http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2, which was pre trained to recognize 68 landmarks on people. These landmarks are detected with correspondence which is important for the algorithm to accurately compare the positions of the landmarks in the reference image with those from the image of each candidate face.

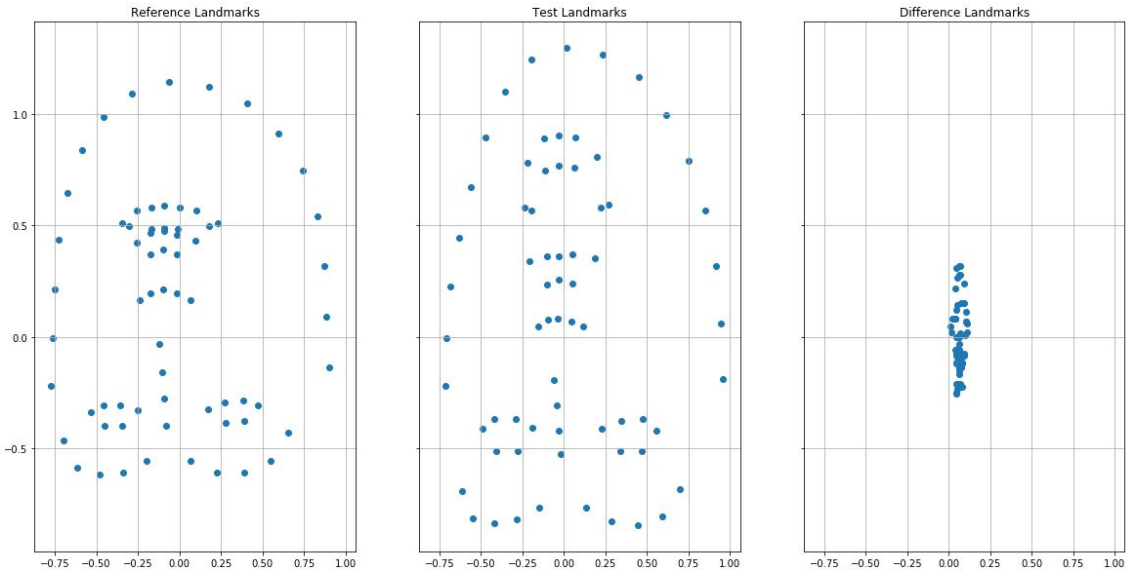
The output of the dlib landmark classifier is simply pixel values within the image. If we wanted to use these algorithms in their raw format, we would have had to ensure that the image captured had the same resolution as the reference image, the face would need to be in the same location within the image, the face would need to be the same size as the face in the

reference image, and the face would need to have the same orientation within the image. These would have been very error prone assumptions and very difficult to satisfy, so we implemented a normalization function within the scoring algorithm. After the dlib algorithm detects the face, it gives a bounding box and matrix of the raw pixel facial landmarks. Using the bounding box, we were able to find a “center” of the face, and de-mean the raw pixel landmarks around this new “center” such that a new landmark at the very center of the face would have a location (0, 0). While this does solve the problem of the face possibly not being in the same location within the image as the reference face, it doesn’t solve the problem of the different sizes of the face. To solve this, the algorithm next scaled each landmark based on the edges of the landmark, such that a landmark located on the right edge of the bounding box would have an x component of 1.0.

After normalizing these landmarks, the difference was taken resulting in a matrix where each column represents the error of the landmark. Calculating the frobenius norm on the matrix of errors effectively is a square root of the sum of the squared errors and was the metric used as the score for a given image.

The following plots show the landmarks normalized for both a flat expression face and a surprised face. Note the landmarks are upside down because of the way in which pixel values in images from openCV are indexed. The plot titled “Difference Landmarks” shows a plot of the difference in the landmarks. It shows that most of the error can be seen in the vertical direction, which makes sense given the second image has an elongated appearance due to the mouth being open. The slight error in the lateral x direction is probably due to slight variation in the orientation of the face within the image, which is a source of uncompensated error. The frobenius norm of this matrix of errors results in the score of 1.677.





Results

The algorithm was ran on two reference images, one being a blank expression, and the other being a very challenging happy expression. The images were chosen because they are different in terms of skin tone, hair style, background and other attributes which the algorithm is designed to be independent of.

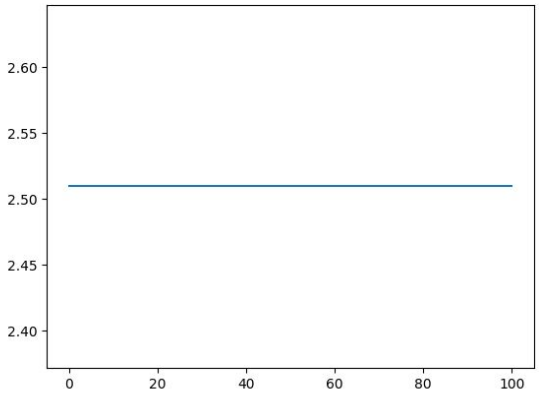
The actual performance of the algorithm was found to be somewhat questionable. For faces such as the very happy face, the robot is not capable of actuating in a way that squints or allows for very wide opening of the mouth. Nevertheless, it can be seen that the resulting expression consistently evolved to an expressing with the mouth open essentially as far as possible. Similarly, for the blank expression, the robot evolved a face that was flat in affect.

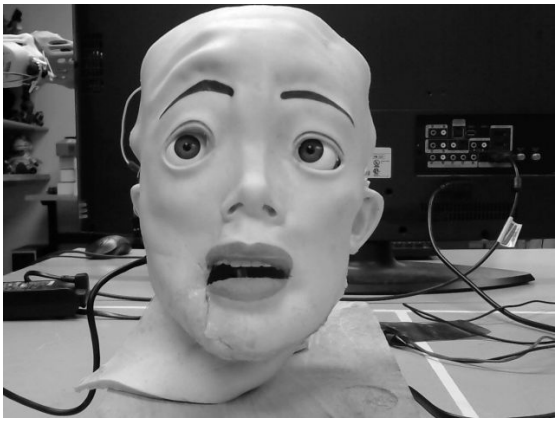
A major source of error for this project was in using a pretrained model for the dlib classifier. The classifier works very well on people, but continually had difficulties estimating the contour of the eyes and eyebrows on the Marie Curie robot. The results of the two trials are outlined below. Videos of the evolution of the faces are available here.

Software can be found here.

Learning a happy expression

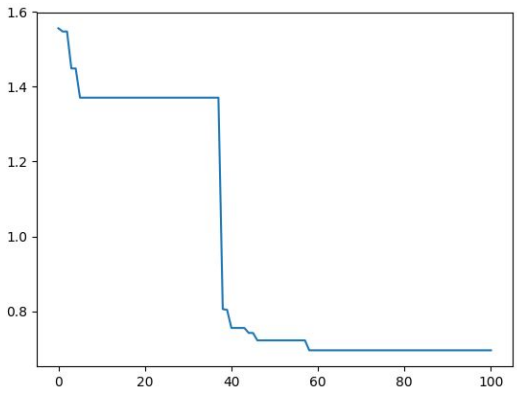
The following plot unfortunately doesn't have labelled axes, but represents the best score for any candidate across generations. It can be seen that the best score was found in the first initial population. This is a somewhat unlikely outcome which represents mistuning of the parameters, settling on a local minima, or perhaps an inability of the face to really actuate in a way that would cause a more steady convergence.





Learning a blank expression

Here we saw a more expected curve which indicates a reduction in the best score from the initial population by more than 50%. The resulting expression does indeed look "blank". In both this and the above expression, the right eyebrow evolved to be raised, which was an interesting artifact of the learning in general. This team did not explore the rationale for this.



Code

Semantic Control Software

core.py

```
import maestro
import servo
import face
from PyQt5.QtWidgets import QMainWindow, QApplication
from marie_curie_gui import Ui_MainWindow

class MarieGui(Ui_MainWindow):
    def __init__(self, controller, window):
        Ui_MainWindow.__init__(self)
        self.setupUi(window)
        self.controller = controller
        self.face = face.Face(self.controller)
        expression_names = [name.capitalize() for name in
self.face.get_expression_names()]
        self.expressionComboBox.addItems(expression_names)
        self.pushButton.clicked.connect(self.user_rq_expression_perf)

    def user_rq_expression_perf(self):
        print('in callback')
        exp_name = self.expressionComboBox.currentText().lower()
        print(exp_name)
        self.face.perform_expression(exp_name)


def main():
    app = QApplication([])
    controller = maestro.Controller()
    window = QMainWindow()
    ui = MarieGui(controller, window)
    window.show()
    app.exec_()

if __name__ == '__main__':
    main()
```

face.py

```
import servo
import yaml

class Face:
    def __init__(self, controller):
        # Parse the YAML file to get the actual servo descriptions
        servo_desc_fname = "marie_servo_descriptions.yaml"
        with open(servo_desc_fname, 'r') as f:
            self.servo_descriptions = yaml.load(f, Loader=yaml.FullLoader)
        facial_expressions_fname = "expressions_gestures.yaml"
        with open(facial_expressions_fname, 'r') as f:
            exp_gestures = yaml.load(f, Loader=yaml.FullLoader)
            self.expressions = exp_gestures['expressions']

        self.servos = {}
        for name in self.servo_descriptions.keys():
            self.servos[name] = servo.Servo(name, controller)

    def __str__(self):
        all_servo_names = ", ".join(self.servos.keys())
        expressions = ", ".join(self.expressions.keys())
        return f"Face has servos: {all_servo_names}.\nFace has expressions:
{expressions}"

    def __repr__(self):
        all_servo_names = ", ".join(self.servos.keys())
        expressions = ", ".join(self.expressions.keys())
        return f"Face has servos: {all_servo_names}.\nFace has expressions:
{expressions}"

    def get_expression_names(self):
        return list(self.expressions.keys())

    def perform_expression(self, exp_name, default_positions=True):
        staged_movements = {}

        # if default positions are desired for unspecified servos, then stage
all resets
        if default_positions:
            for name in self.servos.keys():
                staged_movements[name] =
self.servo_descriptions[name]['default_position']
```

```
        #Update staged movements with those explicitly set in the expression
        for s_name, pos in self.expressions[exp_name].items():
            staged_movements[s_name] = pos

        # Execute all movements
        for s_name, pos in staged_movements.items():
            self.servos[s_name].set_sem_pos(pos)

if __name__ == "__main__":
    import maestro
    face = Face(maestro.Controller())
    face.perform_expression('meh')
    print(face)
```

servo.py

```
import yaml

class Servo:
    """ Implements a class for interacting and dealing with servos at a
semantic level
    :param name: This is the name given to a servo and must match those defined
in marie_servo_config.py
    :type str:
    :param maestro_controller: Used for actually issuing the control commands
    :type maestro_controller: maestro.Controller
    :attr last_semantic: Description of semantic position last requested (i.e.
up, down)
    """
    def __init__(self, name, maestro_controller):
        self.name = name
        servo_desc_fname = "marie_servo_descriptions.yaml"

        # Parse the YAML file to get the actual servo descriptions
        with open(servo_desc_fname, 'r') as f:
            self.servo_descriptions = yaml.load(f, Loader=yaml.FullLoader)

        # Try to identify which logical channel the servo is connected to on
the controller
        try:
            self.channel = self.servo_descriptions[name]['channel']
        except:
            print(f'Could not find channel for this controller.  Double check
the name and marie_servo_config')
            raise

        self.controller = maestro_controller
        self.last_semantic = 'unknown'
        return

    def set_sem_pos(self, semantic_pos):
        """ Set a target position based on a name in
marie_servo_config.servo_positions
        :param semantic_pos: Named position (i.e. 'open' in case of jaw servo)
        :type semantic_pos: str
        """
        pos = self.servo_descriptions[self.name]['positions'][semantic_pos]
```

```
        self.controller.setTarget(self.channel, pos)
        self.last_semantic = semantic_pos

    def set_num_pos(self, numerical_pos):
        """ Set a target position based on a raw numerical value
        :param numerical_pos: number of microseconds for the PWM pulse
        :type numerical_pos: float
        """
        self.controller.setTarget(self.channel, numerical_pos)
        self.last_semantic = 'unknown'

    def get_allowed_sem_pos(self):
        return self.servo_descriptions[self.name]['positions'].keys()

    def __repr__(self):
        return f"{self.name} servo is on channel {self.channel} in position
{self.last_semantic}"

    def __str__(self):
        return f"{self.name} servo is on channel {self.channel} in position
{self.last_semantic}"
```

maestro.py

```
import serial
from sys import version_info

PY2 = version_info[0] == 2   #Running Python 2.x?

#
#----------------------------
# Maestro Servo Controller
#----------------------------
#
# Support for the Pololu Maestro line of servo controllers
#
# Steven Jacobs -- Aug 2013
# https://github.com/FRC4564/Maestro/
#
# These functions provide access to many of the Maestro's capabilities using
the
# Pololu serial protocol
#
class Controller:
    # When connected via USB, the Maestro creates two virtual serial ports
    # /dev/ttyACM0 for commands and /dev/ttyACM1 for communications.
    # Be sure the Maestro is configured for "USB Dual Port" serial mode.
    # "USB Chained Mode" may work as well, but hasn't been tested.
    #
    # Pololu protocol allows for multiple Maestros to be connected to a single
    # serial port. Each connected device is then indexed by number.
    # This device number defaults to 0x0C (or 12 in decimal), which this module
    # assumes.  If two or more controllers are connected to different serial
    # ports, or you are using a Windows OS, you can provide the tty port.  For
    # example, '/dev/ttyACM2' or for Windows, something like 'COM3'.
    def __init__(self,ttyStr='/dev/ttyACM0',device=0x0c):
        # Open the command port
        self.usb = serial.Serial(ttyStr)
        # Command lead-in and device number are sent for each Pololu serial
command.
        self.PololuCmd = chr(0xaa) + chr(device)
        # Track target position for each servo. The function isMoving() will
        # use the Target vs Current servo position to determine if movement is
        # occuring.  Upto 24 servos on a Maestro, (0-23). Targets start at 0.
        self.Targets = [0] * 24
        # Servo minimum and maximum targets can be restricted to protect
```

```
components.
        self.Mins = [0] * 24
        self.Maxs = [0] * 24

    # Cleanup by closing USB serial port
    def close(self):
        self.usb.close()

    # Send a Pololu command out the serial port
    def sendCmd(self, cmd):
        cmdStr = self.PololuCmd + cmd
        if PY2:
            self.usb.write(cmdStr)
        else:
            self.usb.write(bytes(cmdStr,'latin-1'))

    # Set channels min and max value range.  Use this as a safety to protect
    # from accidentally moving outside known safe parameters. A setting of 0
    # allows unrestricted movement.
    #
    # ***Note that the Maestro itself is configured to limit the range of servo
travel
    # which has precedence over these values.  Use the Maestro Control Center
to configure
    # ranges that are saved to the controller.  Use setRange for software
controllable ranges.
    def setRange(self, chan, min, max):
        self.Mins[chan] = min
        self.Maxs[chan] = max

    # Return Minimum channel range value
    def getMin(self, chan):
        return self.Mins[chan]

    # Return Maximum channel range value
    def getMax(self, chan):
        return self.Maxs[chan]

    # Set channel to a specified target value.  Servo will begin moving based
    # on Speed and Acceleration parameters previously set.
    # Target values will be constrained within Min and Max range, if set.
    # For servos, target represents the pulse width in of quarter-microseconds
    # Servo center is at 1500 microseconds, or 6000 quarter-microseconds
    # Typcially valid servo range is 3000 to 9000 quarter-microseconds
```

```
    # If channel is configured for digital output, values < 6000 = Low ouput
    def setTarget(self, chan, target):
        # if Min is defined and Target is below, force to Min
        if self.Mins[chan] > 0 and target < self.Mins[chan]:
            target = self.Mins[chan]
        # if Max is defined and Target is above, force to Max
        if self.Maxs[chan] > 0 and target > self.Maxs[chan]:
            target = self.Maxs[chan]
        #
        lsb = target & 0x7f #7 bits for least significant byte
        msb = (target >> 7) & 0x7f #shift 7 and take next 7 bits for msb
        cmd = chr(0x04) + chr(chan) + chr(lsb) + chr(msb)
        self.sendCmd(cmd)
        # Record Target value
        self.Targets[chan] = target

    def setMultipleTargets(self, start_chan, targets):
        """ Sets multiple servos to values.  Performs 3x faster than multiple
individual writes.
        This requires that the targets are provided contiguously.
        (i.e. start_chan = 3, targets = [1500, 1500, 1500] will write channels
3, 4, 5.

        Arguments:
            start_chan {int} -- The first channel number out of the contiguous
channels to be set
            targets {list} -- List of target values each in quarter
microseconds.  Index 0 is
                 target value from channel start_chan.
        """
        # Initial command string.  0x1f corresponds to multiple target writes
        cmd = chr(0x1f) + chr(len(targets))
        # Iterate through targets and append to command
        for chan, target in enumerate(targets, start=start_chan):
            lsb = target & 0x7f #7 bits for least significant byte
            msb = (target >> 7) & 0x7f #shift 7 and take next 7 bits for msb
            cmd += chr(chan) + chr(lsb) + chr(msb)
            # Record target value
            self.Targets[chan] = target
        # Send the actual command
        self.sendCmd(cmd)
```

```
    # Set speed of channel
    # Speed is measured as 0.25microseconds/10milliseconds
    # For the standard 1ms pulse width change to move a servo between extremes,
a speed
    # of 1 will take 1 minute, and a speed of 60 would take 1 second.
    # Speed of 0 is unrestricted.
    def setSpeed(self, chan, speed):
        lsb = speed & 0x7f #7 bits for least significant byte
        msb = (speed >> 7) & 0x7f #shift 7 and take next 7 bits for msb
        cmd = chr(0x07) + chr(chan) + chr(lsb) + chr(msb)
        self.sendCmd(cmd)

    # Set acceleration of channel
    # This provide soft starts and finishes when servo moves to target
position.
    # Valid values are from 0 to 255. 0=unrestricted, 1 is slowest start.
    # A value of 1 will take the servo about 3s to move between 1ms to 2ms
range.
    def setAccel(self, chan, accel):
        lsb = accel & 0x7f #7 bits for least significant byte
        msb = (accel >> 7) & 0x7f #shift 7 and take next 7 bits for msb
        cmd = chr(0x09) + chr(chan) + chr(lsb) + chr(msb)
        self.sendCmd(cmd)

    # Get the current position of the device on the specified channel
    # The result is returned in a measure of quarter-microseconds, which
mirrors
    # the Target parameter of setTarget.
    # This is not reading the true servo position, but the last target position
sent
    # to the servo. If the Speed is set to below the top speed of the servo,
then
    # the position result will align well with the acutal servo position,
assuming
    # it is not stalled or slowed.
    def getPosition(self, chan):
        cmd = chr(0x10) + chr(chan)
        self.sendCmd(cmd)
        lsb = ord(self.usb.read())
        msb = ord(self.usb.read())
        return (msb << 8) + lsb

    # Test to see if a servo has reached the set target position.  This only
provides
```

```
    # useful results if the Speed parameter is set slower than the maximum
speed of
    # the servo.  Servo range must be defined first using setRange. See
setRange comment.
    #
    # ***Note if target position goes outside of Maestro's allowable range for
the
    # channel, then the target can never be reached, so it will appear to
always be
    # moving to the target.
    def isMoving(self, chan):
        if self.Targets[chan] > 0:
            if self.getPosition(chan) != self.Targets[chan]:
                return True
        return False

    # Have all servo outputs reached their targets? This is useful only if
Speed and/or
    # Acceleration have been set on one or more of the channels. Returns True
or False.
    # Not available with Micro Maestro.
    def getMovingState(self):
        cmd = chr(0x13)
        self.sendCmd(cmd)
        if self.usb.read() == chr(0):
            return False
        else:
            return True

    # Run a Maestro Script subroutine in the currently active script. Scripts
can
    # have multiple subroutines, which get numbered sequentially from 0 on up.
Code your
    # Maestro subroutine to either infinitely loop, or just end (return is not
valid).
    def runScriptSub(self, subNumber):
        cmd = chr(0x27) + chr(subNumber)
        # can pass a param with command 0x28
        # cmd = chr(0x28) + chr(subNumber) + chr(lsb) + chr(msb)
        self.sendCmd(cmd)

    # Stop the current Maestro Script
    def stopScript(self):
        cmd = chr(0x24)
```

```
        self.sendCmd(cmd)
```

marie_servo_descriptions.yaml

The following is a yaml formatted description of the positions of the servos. The servo channel is the physical channel that the servo is connected to on the Pololu board. The positions are defined by the length of the PWM pulse imarie_servo_descriptions.yamln quarter microseconds. The position of the servo is a function of the length of the PWM pulse, therefore the value in quarter microseconds corresponds to a position. These values are determined empirically using the Maestro Control Center.

```
eye_l_r:
  channel: 0
  default_position: center
  positions: {center: 5604, left: 8000, right: 1984}
eye_u_d:
  channel: 1
  default_position: center
  positions: {center: 5792, down: 8000, up: 3584}
jaw_u_d:
  channel: 2
  default_position: closed
  positions: {closed: 7056, halfway: 6360, open: 5212}
eyebrow_l:
  channel: 3
  default_position: resting
  positions: {resting: 5832, up: 8000}
eyebrow_r:
  channel: 4
  default_position: resting
  positions: {resting: 7680, up: 6000}
mouth_l:
  channel: 5
  default_position: resting
  positions: {back: 5640, resting: 8000}
mouth_r:
  channel: 6
  default_position: resting
  positions: {back: 6532, resting: 3968}
middle_forehead:
  channel: 7
  default_position: resting
  positions: {clinched: 6000, resting: 3968}
oris_l:
  channel: 8
```

```
  default_position: resting
  positions: {clinched: 8000, resting: 3584}
oris_r:
  channel: 9
  default_position: resting
  positions: {resting: 8000, right: 4000}
upper_lip:
  channel: 10
  default_position: resting
  positions: {clinched: 8000, resting: 3968}
```

expressions_gestures.yaml

The following is a yaml description of five example expressions and one gesture. Servos which are not defined for a given expression may or may not be set to their default values depending on how the perform_expression function in face.py is called by the user.

```
expressions:
  skeptical:
    eyebrow_l: up
    eye_l_r: right
  resting:
    upper_lip: resting
  angry:
    middle_forehead: clinched
    upper_lip: clinched
  meh:
    eye_l_r: left
    eye_u_d: down
  surprised:
    eyebrow_l: up
    eyebrow_r: up
    jaw_u_d: open
gestures:
  Something:
  - !!python/tuple
    - meh
    - 0.0
  - !!python/tuple
    - surprised
    - 1.0
  - !!python/tuple
    - meh
    - 2.0
```

Genetic Algorithm Software

Ga.py

```
import Population
import ga_utils
import cv2
import os
from datetime import datetime
import matplotlib.pyplot as plt

NUM_GENS = 100
POPSIZE = 100
NUM_BRED_PER_GEN = 50

# TODO: make the next two lines inputs via argument parser
ref_img_path = "reference_images/happy.jpg"
expression_name = 'test'
# Get some information about the current datetime in order
# to create a new folder for this run
dt_str = datetime.now().strftime(format="%Y_%b_%d_%H_%M_%S")
rel_run_name = f"ga_{expression_name}_{dt_str}"
abs_run_name = os.path.join(os.getcwd(), rel_run_name)
os.mkdir(abs_run_name)
print(f"Calculating landmarks on reference image")
ref_landmarks = ga_utils.get_ref_img_landmarks(ref_img_path)
print(f"Initializing population")

best_scores = []

pop = Population.Population(POPSIZE)
for i, candidate in enumerate(pop.new_candidates):
   score, scored_image = ga_utils.get_score(candidate.chromosome,
ref_landmarks)
   cv2.imwrite(os.path.join(abs_run_name, f"gen{0:04}candidate{i:04}.jpg"),
scored_image)
   candidate.set_score(score, scored_image)
   print(f"Candidate {i}: {score}")
pop.merge_and_drop_candidates()

best_scores.append(pop.get_best_candidate().score)

print(f"Beginning evolution")
```

```
for gen_num in range(1, NUM_GENS + 1):
   print(f"Now evolving generation {gen_num}")
   pop.breed_new(NUM_BRED_PER_GEN)
   print(f"Now scoring {len(pop.new_candidates)} new candidates")
   for i, candidate in enumerate(pop.new_candidates):
       score, scored_image = ga_utils.get_score(candidate.chromosome,
ref_landmarks)
       cv2.imwrite(os.path.join(abs_run_name,
f"gen{gen_num:04}candidate{i:04}.jpg"), scored_image)
       candidate.set_score(score, scored_image)
   pop.merge_and_drop_candidates(NUM_BRED_PER_GEN)
   this_gen_best_img = pop.get_best_candidate().get_image()
   best_scores.append(pop.get_best_candidate().score)
   cv2.imwrite(os.path.join(abs_run_name, f"best_img_gen{gen_num:04}.jpg"),
this_gen_best_img)

print("Done!  Getting best candidate")
winning_candidate = pop.get_best_candidate()
print(winning_candidate)
cv2.imwrite(os.path.join(abs_run_name, "best_run_img.jpg"),
winning_candidate.get_image())
print("Displaying best candidate on robot")
ga_utils.actuate_chromosome(winning_candidate.chromosome)

# Scores over generations
print(f"Scores over gen's: {best_scores}")

plt.plot(best_scores)
plt.show()

ga_utils.CAP.release()
ga_utils.CONTROLLER.close()
print("Adding expression to expressions list")
#ga_utils.add_exp(expression_name, winning_candidate.chromosome)
```

Population.py

```
"""
ECE 579 Intelligent Robotics II
Team 3 - Marie Curie Robot
R. Holt
D. Yakovlev
Population.py
"""
from Candidate import Candidate
from ga_utils import num_children_gen, clip_chromosome_limits
import random
import numpy as np

class Population:
    def __init__(self, popsize):
        self.popsize = popsize
        self.candidates = []
        self.new_candidates = []
        for _ in range(popsize):
            self.new_candidates.append(Candidate.create_random())
        return

    def breed_new(self, num_to_breed, num_per_parents=2):
        """
        breed_new breeds num_to_breed candidates from the candidate
        pool using built in methods to define the mating pool and
        actually perform the mating.  num_per_parents represents how
        many of the total num_to_breed candidates should come from each
        pair of parents.  All new candidates are added to
self.new_candidates.

        :param num_to_breed: total number of new candidates to breed
        :type num_to_breed: int
        :param num_per_parents: number of candidates to have each group
        of parents produce, defaults to 2
        :type num_per_parents: int, optional
        """
        print(f"Breeding {num_to_breed} candidates with {num_per_parents}
per parents")
        q, r = divmod(num_to_breed, num_per_parents)
        num_pairs = q + int(r > 0)
```

```
        print(f"Need {num_pairs} pairs of parents")

        all_parents = self._get_mating_pool(num_pairs)
        parent_pairs = self._pair_off_parents(all_parents)

        for parent1, parent2, num in num_children_gen(parent_pairs,
num_to_breed, num_per_parents):
            self.new_candidates += self._mate(parent1, parent2, num)
        print(self.new_candidates)

    def merge_and_drop_candidates(self, num_merged_dropped=None):
        """
        merge_and_drop_candidates concatenates the self.new_candidates with
        the previous self.candidates.  It then drops the worst
num_merged_dropped
        candidates from the pool.

        :param num_merged_dropped: quantity of Candidates to remove from
the pool
        :type num_merged_dropped: int
        """
        print(f"Merging {len(self.new_candidates)} into the previous
population of {len(self.candidates)}")
        self.candidates += self.new_candidates
        self.new_candidates = []
        print(f"Now dropping {num_merged_dropped} candidates")
        self._drop_worst(num_merged_dropped)

    def get_best_candidate(self):
        """
        get_best_candidate simply returns the most fit candidate
        as deteremined based on the scores of all candidates in
self.candidates.

        :return: most fit candidate
        :rtype: Candidate
        """
        self.candidates = self._sort_population()
        return self.candidates[0]

    def _pair_off_parents(self, parents):
        """
```

```
        _pair_off_parents Creates a list of lists where each
        sublist is a pair of two parents.  CUrrently this pairs
        the most fit parents together.

        :param parents: flat list of candidates which are the parents
        :type parents: list of Candidates
        :return: list of list of paired candidates
        :rtype: list of list of Candidates
        """
        # TODO: improve this to pair off parents in other statistical ways
        # zip together into a list pairs of parents sequentially
        return list(zip(parents[::2], parents[1::2]))

    def _get_mating_pool(self, num_pairs):
        """
        _get_mating_pool identifies the parents which should make up the
        entire mating pool based on the number of pairs of parents.
        Currently it only takes the most fit 2 * num_pairs candidates.

        :param num_pairs: Quantity of pairs of parents
        :type num_pairs: integer
        :return: candidates which represent the mating pool
        :rtype: list of Candidates
        """
        # TODO: Improve this to select candidates with some probability
        # such that sometimes lesser fit candidates are matched in
        # order to introduce variety into gene pool
        self._sort_population()
        # This only returns the top scoring candidates
        return self.candidates[:2 * num_pairs]

    def _sort_population(self, high_score_is_better=False):
        """
        _sort_population Function to sort member candidates from the
population.
        Puts the "best" candidates in low indices of returned list.

        :param high_better: indicates whether high candidate scores are
better, defaults to False
        :type high_better: bool, optional
        :raises Exception: when not all Candidates have scores to sort by
        :return: list of candidates
```

```
        :rtype: list of instances of Candidate
        """
        # If not all candidates have a score, then raise an error
        if not all([True if candidate.score else False for candidate in
self.candidates]):
            raise Exception("Cannot sort population, not all candidates are
scored")

        return sorted(self.candidates, key=lambda x: x.score,
reverse=high_score_is_better)

    def _mate(self, candidate1, candidate2, num_new=2, mut_std=50):
        """
        _mate Combine parents into a number of new child candidates.
        Contiguous sections of the individual parent chromosomes are
selected for
        each child.  The parent for lower/upper indices of the child
chromosome
        is randomized for each child.

        :param candidate1: First parent candidate
        :type candidate1: instance of Candidate
        :param candidate2: Second parent candidate
        :type candidate2: instance of Candidate
        :param num_new: number of children to generate from these parents,
defaults to 2
        :type num_new: int, optional
        :param mut_std: Standard deviation of the individual gene mutations
in quarter microseconds
        :type mut_std: int
        :return: Child Candidates
        :rtype: list of Candidates
        """
        chrom_len = len(candidate1.chromosome)
        parents = [candidate1, candidate2]
        new_candidates = []
        for _ in range(num_new):
            # Randomize the order of the parents
            random.shuffle(parents)

            # Determine how much to take from the parents
            # This can be improved by biasing the location in
```

```
            # favor of taking more genes from the more fit parent.
            slice_loc = random.randint(1, chrom_len - 1)

            # Crossover the parent's chromosomes
            new_cand = Candidate(parents[0].chromosome[:slice_loc] +
parents[1].chromosome[slice_loc:])

            # Apply variation to each gene in the chromosome.
            # The mutation is based on a multivariate normal distribution
with means centered
            # on the inherited genes via crossover.  The variance controls
how severe the mutation is
            # a standard deviation (mut_std) indicates that ~68% of the
time, the new gene will be within
            # 50 quarter microseconds of the inherited gene.
            new_cand.chromosome =
list(np.random.normal(loc=new_cand.chromosome,
scale=mut_std).astype('int'))

            # Saturate the genes at their allowed limits to ensure that the
hardware is not damaged.
            new_cand.chromosome =
clip_chromosome_limits(new_cand.chromosome)

            new_candidates.append(new_cand)
        return new_candidates

    def _drop_worst(self, num_to_drop=None):
        """
        drop_worst Removes the worst num_to_drop candidates from
self.candidates
        using the self._sort_population function.

        :param num_to_drop: Quantity of candidates to remove from the
population
        :type num_to_drop: int
        """
        if not num_to_drop:
            return
        else:
            self.candidates = self._sort_population()[:-num_to_drop]
        return
```

```
if __name__ == "__main__":
    pop = Population(100)
    for cand in pop.new_candidates:
        cand.set_score(random.random())
        print(cand)
    pop.merge_and_drop_candidates()

    print("Finding best...")
    best = pop.get_best_candidate()
    print(best)

    print("Testing mating...")
    pop._mate(pop.candidates[0], pop.candidates[1])
```

Candidate.py

```
"""
ECE 579 Intelligent Robotics II
Team 3 - Marie Curie Robot
R. Holt
D. Yakovlev
Candidate.py
"""
from ga_utils import LIMITS
from random import randint
# Throughout all of this, servo positions and chromosomes are used
interchangably to mean the same thing

class Candidate:
   def __init__(self, chromosome):
       self.chromosome = chromosome
       self.score = None
       self.image = None


   def set_score(self, score, image=None):
       self.score = score
       self.image = image

   def get_image(self):
       return self.image

   def __repr__(self):
       return f"Candidate chromosome: {self.chromosome} / Score: {self.score}"

   def __str__(self):
       return self.__repr__()

   # Classmethod is a special type of method that doesnt receive "self"
   # since it doesnt have a concept of an instance of Candidate.  Instead
   # it receives the class Candidate, and functions as an alternate
constructor.
   @classmethod
   def create_random(cls):
       # Calc random servo_positions here
       servo_positions = []
       for limit in LIMITS:
           servo_positions.append(randint(limit[0], limit[1]))
```

```
        return cls(servo_positions)

if __name__ == "__main__":
    print(Candidate.create_random())
```

Ga_utils.py

```
import maestro
import yaml
import dlib
from imutils import face_utils
import time
import cv2
import numpy as np

try:
   CONTROLLER = maestro.Controller()
except:
   print("Controller couldnt be initialized, expect problems")

try:
   CAP = cv2.VideoCapture(0)
   CAP.set(cv2.CAP_PROP_BUFFERSIZE, 1)
except:
   print("Video device couldnt be initialized, expect problems)")

DETECTOR = dlib.get_frontal_face_detector()
PREDICTOR = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

def get_servo_limits(yaml_filename="marie_servo_descriptions.yaml"):
   """
   get_servo_limits parses the yaml file defining servo limits for the robot.
   The resulting list of tuples has the lower limit in index 0 and the upper
limit
   in index 1 for each tuple.

   :param yaml_filename: filename for the yaml file with appropriate structure,
   defaults to "marie_servo_descriptions.yaml"
   :type yaml_filename: str, optional
   :return: list of tuples which indicate the lower and upper limits for each
   :rtype: list of tuples of ints
   """
   # Open up the yaml file and bring in the servo descriptions
   with open(yaml_filename, 'r') as f:
       servo_descs = yaml.load(f, Loader=yaml.FullLoader)
   # Sort the servo descriptions by channel number
   servo_desc_list = list(servo_descs.values())
   servo_desc_list.sort(key=lambda x: x['channel'])
```

```
    limits = []
    # Iterate through the descriptions and build a min/max limit tuple
    for servo_desc in servo_desc_list:
        limits.append((servo_desc['min_pos'], servo_desc['max_pos']))
    return tuple(limits)

LIMITS = get_servo_limits()

def clip_chromosome_limits(chromosome):
    np_limits = np.array(LIMITS)

    clip_l = np_limits[:,0]
    clip_h = np_limits[:,1]

    return list(np.clip(chromosome, clip_l, clip_h))

def actuate_chromosome(chromosome):
    servo_pos = chromosome
    CONTROLLER.setMultipleTargets(0, servo_pos)
    return

def get_score(chromosome, ref_norm_landmarks):
    # Actuate the face and wait for it to be actuated
    actuate_chromosome(chromosome)
    time.sleep(0.5)

    # Throw away a capture because the minimum buffersize is 1
    _, _ = CAP.read()
    # Take the actual image
    ret, frame = CAP.read()

    # Our operations on the frame come here
    # Make the frame grayscale
    img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    adjusted_landmarks = _get_normed_landmarks(img)

    diff_landmarks = ref_norm_landmarks - adjusted_landmarks
    score = np.linalg.norm(diff_landmarks @ diff_landmarks.T, ord='fro')

    return score, img

def _get_normed_landmarks(img):
    # Detect bounding box for faces
```

```
    dets = DETECTOR(img, 1)

    # Assume only one face is going to be found
    shape = PREDICTOR(img, dets[0])
    raw_landmarks = face_utils.shape_to_np(shape, dtype='float')

    bbox_l, bbox_r, bbox_t, bbox_b = shape.rect.left(), shape.rect.right(),
shape.rect.top(), shape.rect.bottom()

    center = [shape.rect.center().x, shape.rect.center().y]
    x_factor = float(bbox_r - center[0])
    y_factor = float(bbox_b - center[1])

    adjusted_landmarks = raw_landmarks - center
    adjusted_landmarks[:,0] = adjusted_landmarks[:, 0] / x_factor
    adjusted_landmarks[:,1] = adjusted_landmarks[:, 1] / y_factor
    return adjusted_landmarks

def get_ref_img_landmarks(filename):
    # Throw away a capture because the minimum buffersize is 1
    frame = dlib.load_rgb_image(filename)

    return _get_normed_landmarks(frame)

def add_exp(name,
            servo_posns,
            expressions_filename="expressions_gestures.yaml",
            servo_positions_fname="marie_servo_descriptions.yaml"):
    with open(servo_positions_fname, 'r') as f:
        servo_descs = yaml.load(f, Loader=yaml.FullLoader)

    # Create reverse lookup table to turn a channel number into a servo name
    servo_name_lut = {value['channel']: key for key, value in
servo_descs.items()}

    # Create a new dictionary which represents the expression
    new_exp_dict = {servo_name_lut[channel]: servo_posns[channel] for channel in
range(len(servo_posns))}

    with open(expressions_filename, 'r') as f:
        exp_ges = yaml.load(f, yaml.FullLoader)
        exp_ges['expressions'][name] = new_exp_dict
    with open(expressions_filename, 'w') as f:
        yaml.dump(exp_ges, f)
```

```
    return

def num_children_gen(parent_pairs, total_num, each):
    num_remaining = total_num
    for parent1, parent2 in parent_pairs:
        num_to_yield = min(num_remaining, each)
        yield parent1, parent2, num_to_yield
        num_remaining -= num_to_yield
    return

if __name__ == "__main__":
    print(f"Limits are:")
    for i, limit in enumerate(LIMITS):
        print(f"Servo {i}: {limit}")
```