**Team 3: Marie Curie Head**

Intelligent Robotics II

Winter 2020

by

**Robert Holt**

**David Yakovlev**

### 

[**Project Overview**](#_rwvjv9p191vc) **3**

[**Initial State of Marie Curie Head**](#_nsxxrf1rmlbh) **4**

[**Repairs**](#_fnnpxv4wg5oa) **6**

[Skull Repair](#_mb7fyrm8wd0) 6

[Latex Repair](#_fxxoxpofzv15) 6

[Other Repairs](#_a63s4wk8dmax) 7

[Outcome](#_ilcajbpffsfr) 9

[Additional Damage and Repairs During Project](#_wsdh6p2f9a9) 9

[**Testing Physical Components**](#_k6319a5mzo7o) **10**

[**Expression and Gesture Framework**](#_2dbngwaq7rbp) **12**

[Python and Pololu based Development](#_sqv4l54y74s5) 12

[Framework Oriented Approach](#_sgkz9o7uy3su) 12

[Simplified Representation of Expressions](#_mcqj01k253qr) 13

[Simple GUI for Testing Expressions](#_ree4m9jddd8) 14

[**Expression Learning via Genetic Algorithm**](#_cp38srenapec) **14**

[Overview](#_90tk7axmxeeh) 14

[Algorithm](#_90tk7axmxeeh) 15

[Scoring Algorithm](#_jg3co8o1e2eq) 17

[Results](#_e7rvj4jdgguw) 19

[Learning a happy expression](#_v2dgsrh3raot) 20

[Learning a blank expression](#_dlfha5j4x0rt) 20

[**Code**](#_agkhmbllzpfz) **21**

[Semantic Control Software](#_valnv5dl0bs1) 21

[core.py](#_j1meo8xy0iyv) 21

[face.py](#_nyt3dvn73lok) 23

[servo.py](#_g4yvwsbx0zxn) 25

[maestro.py](#_351b8ts1o6ze) 27

[marie\_servo\_descriptions.yaml](#_ej0n5fyxb74l) 32

[expressions\_gestures.yaml](#_ko0c1q0xo2c) 34

[Genetic Algorithm Software](#_43i8wj6tru1g) 35

[Ga.py](#_8awinnrjnnml) 35

[Population.py](#_xz04hs86vyxv) 37

[Candidate.py](#_7xrbs5vgefgu) 43

[Ga\_utils.py](#_j0aylsxtr7is) 45

## 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.

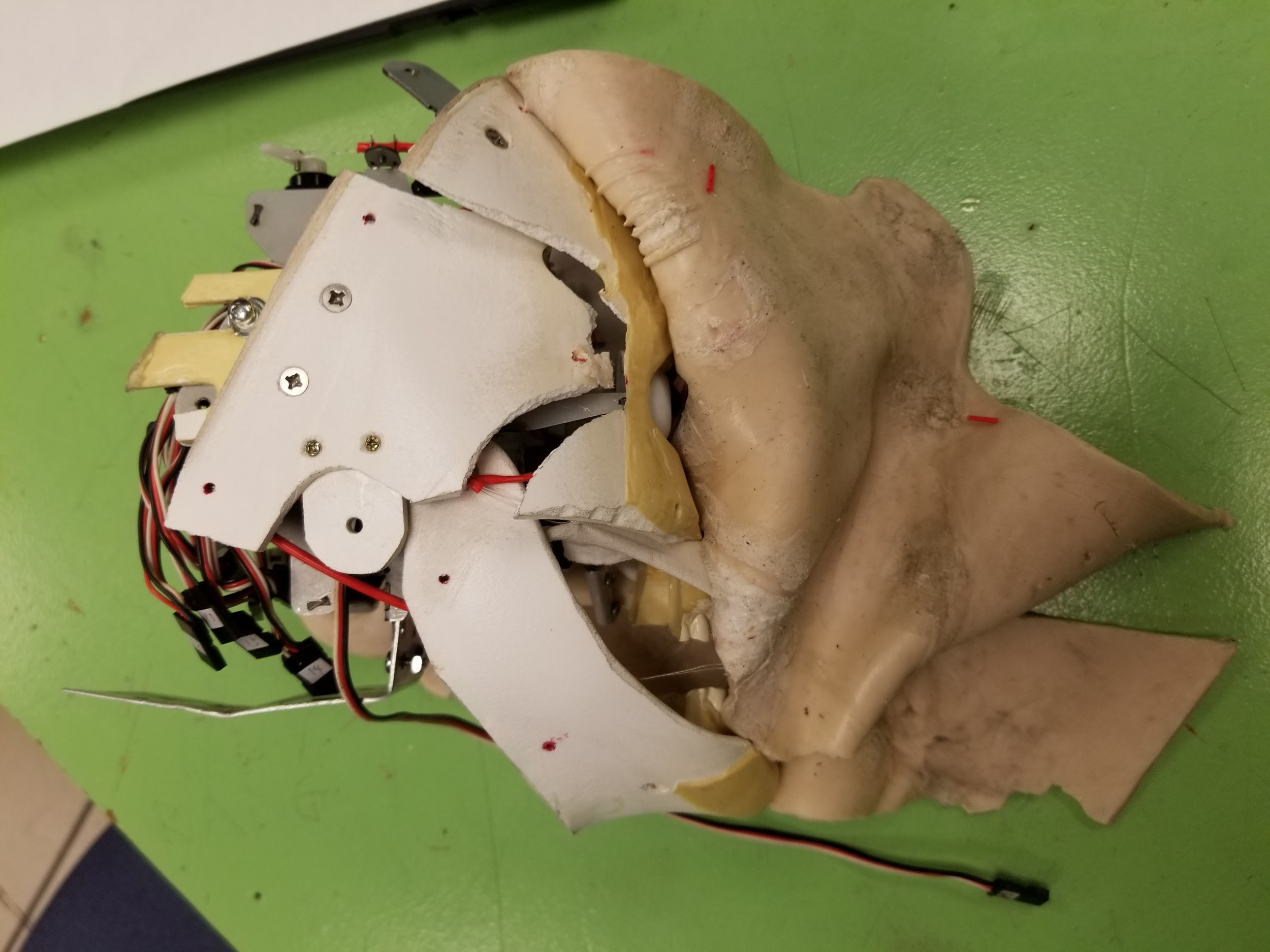
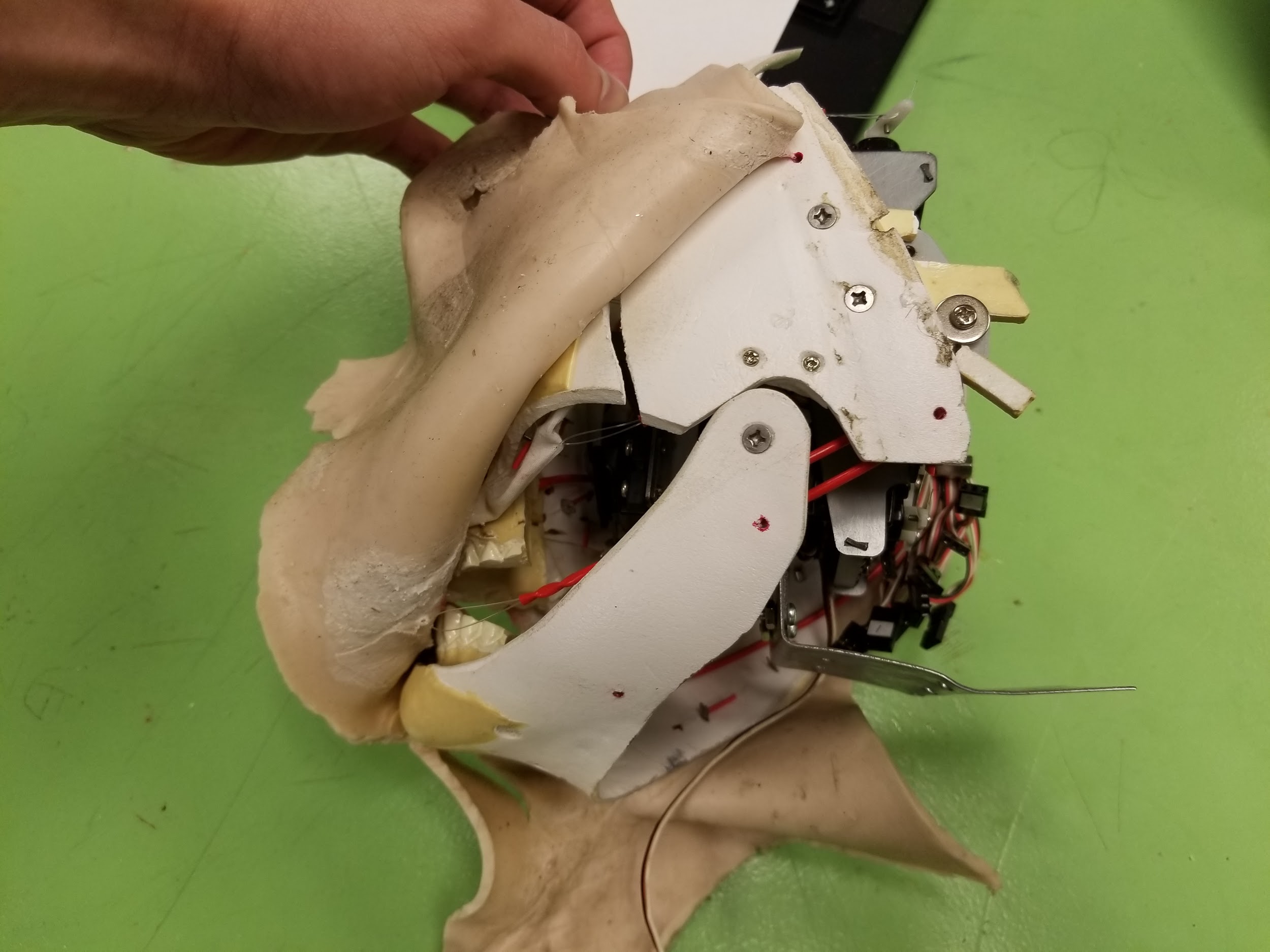
Second, after verifying that the physical components are in working order, the next step was to understand the computational platform included. This involved understanding how control is distributed to various modules and how the actuators / servos are controlled. Investigations into the architecture 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

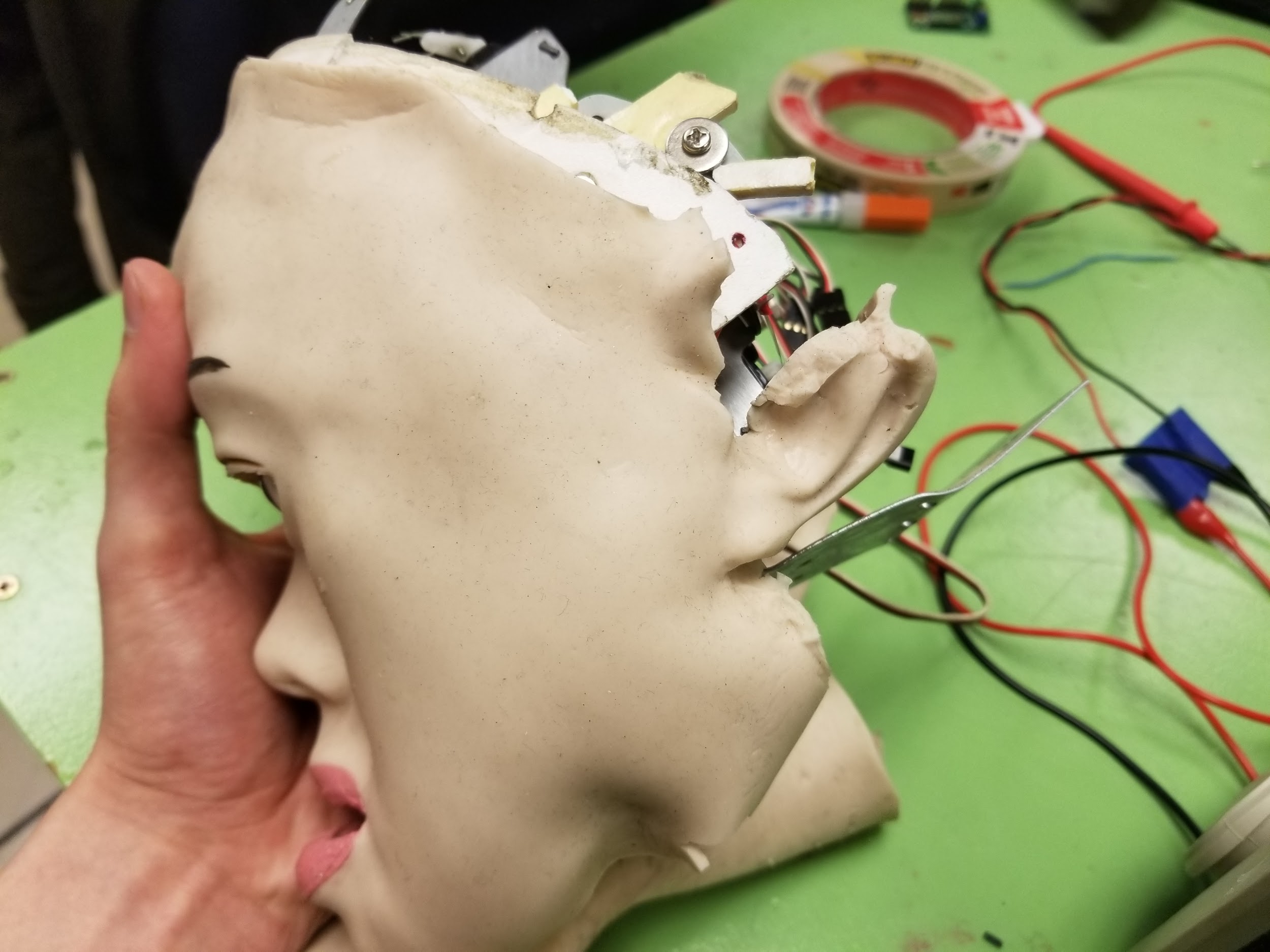
The initial state of the Marie Curie head was in a very bad shape, much more than we expected. The skull was completely broken, detached from the skull, on both sides of the face, as can be seen from the two images below. While the pieces were still attached to the metal bracket holding the servos, there 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

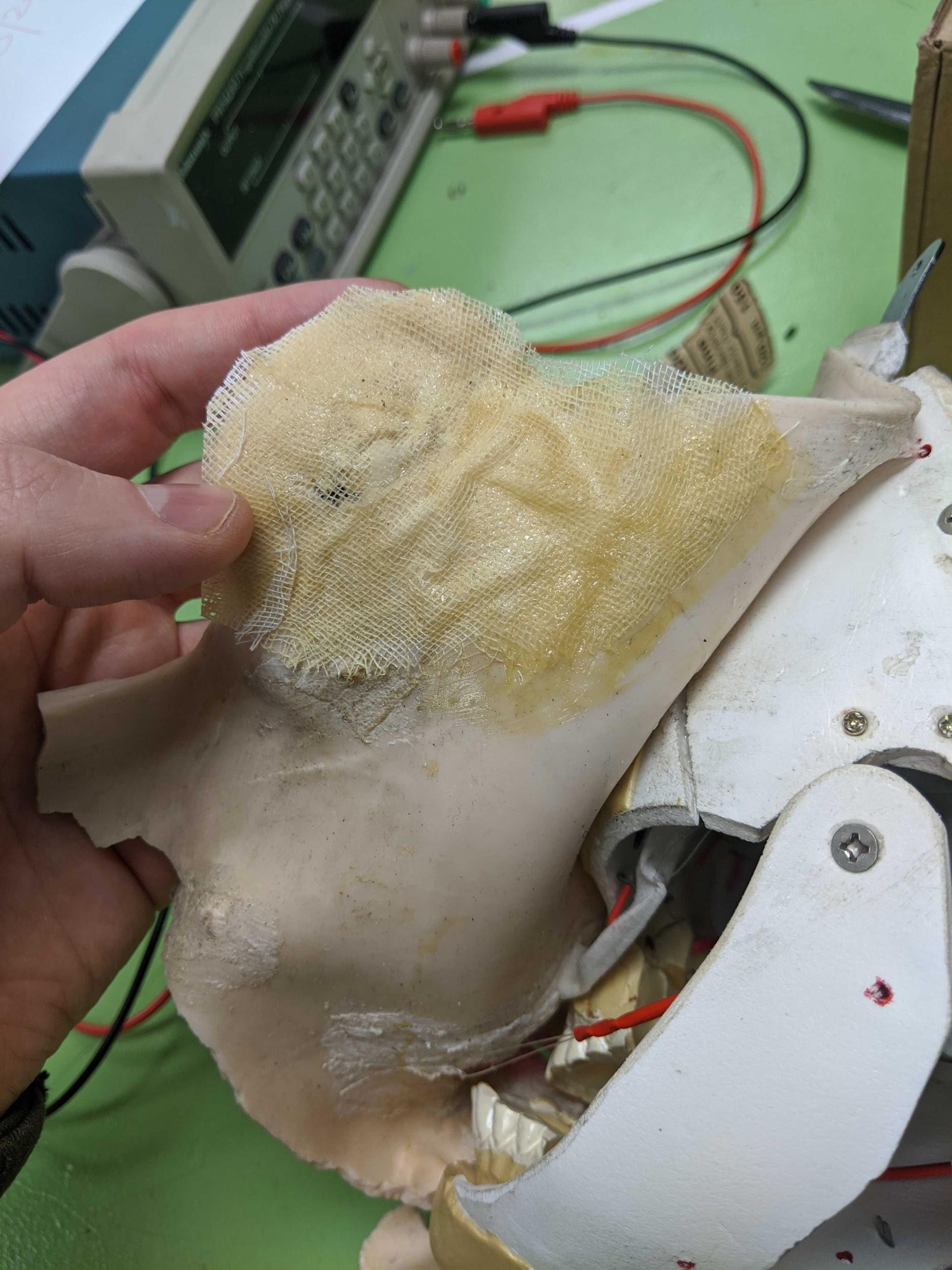
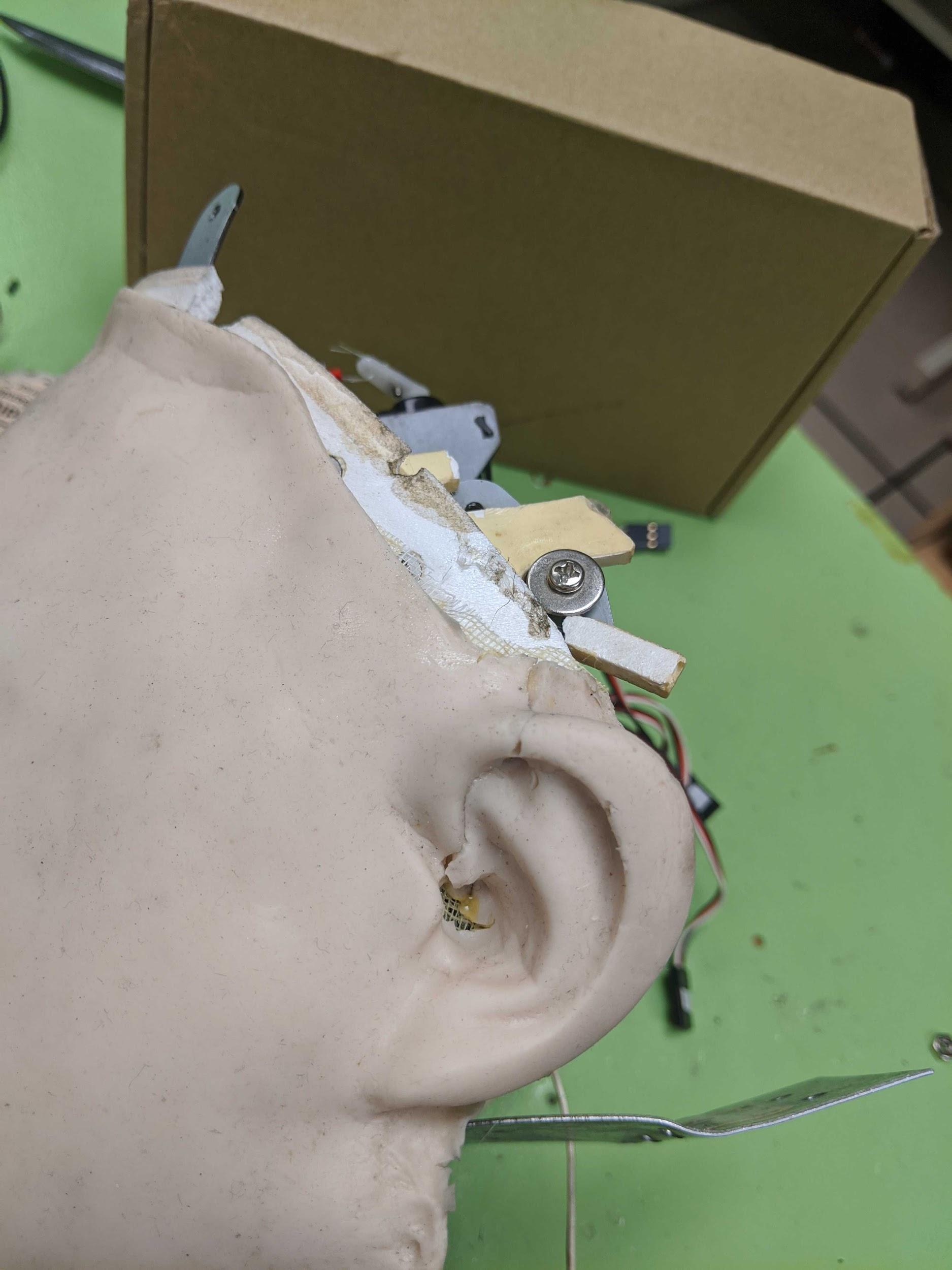
### Skull Repair

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)



#### 

#### 

#### 

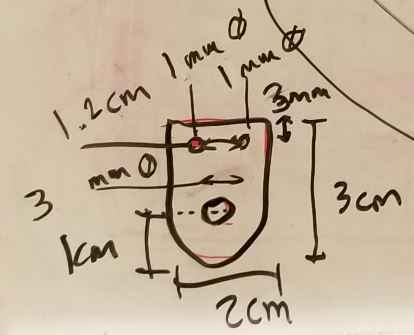
#### 

#### 

#### 

### 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.



#### 

#### 

#### 

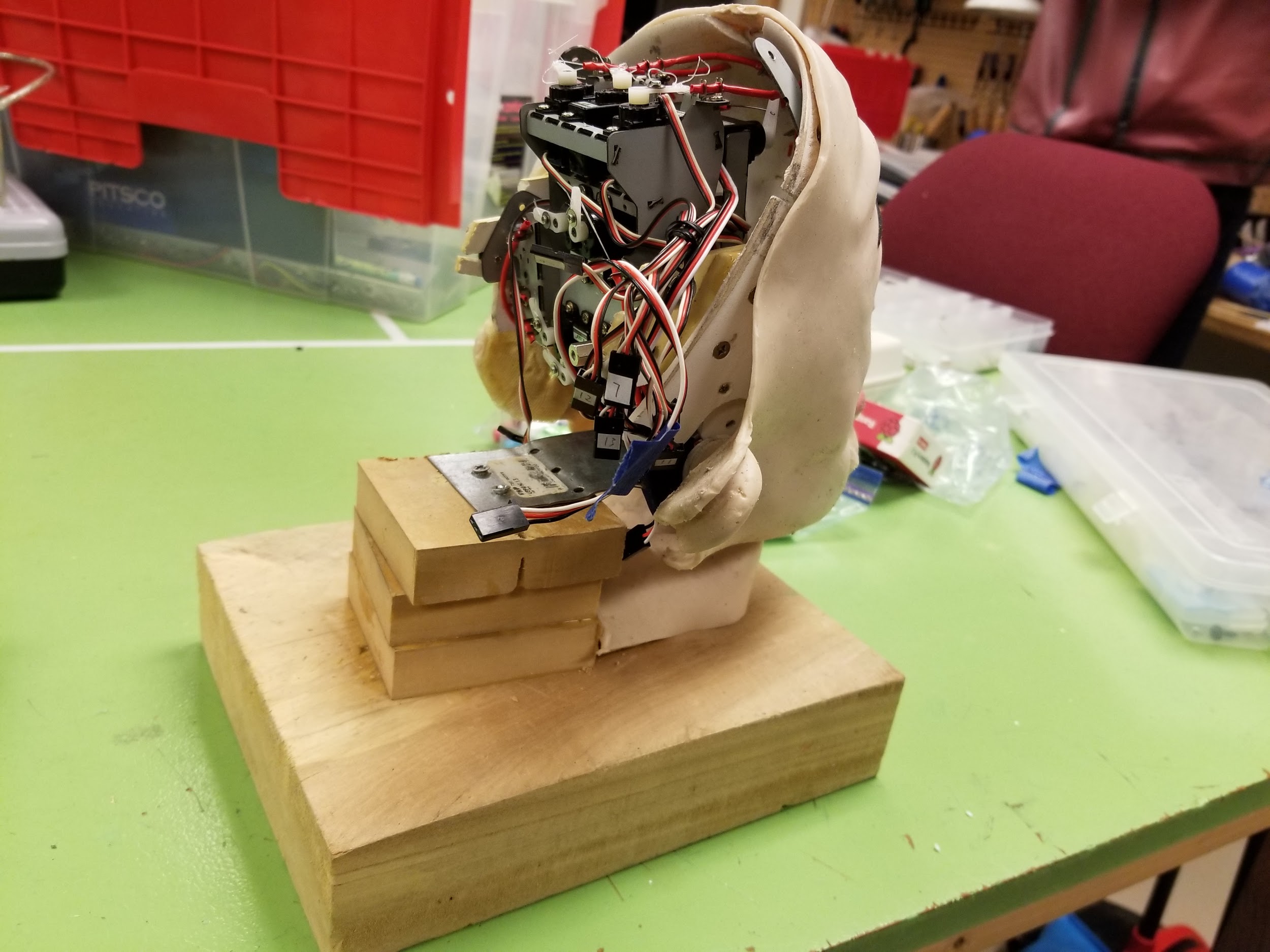
#### 

#### 

#### 

### 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

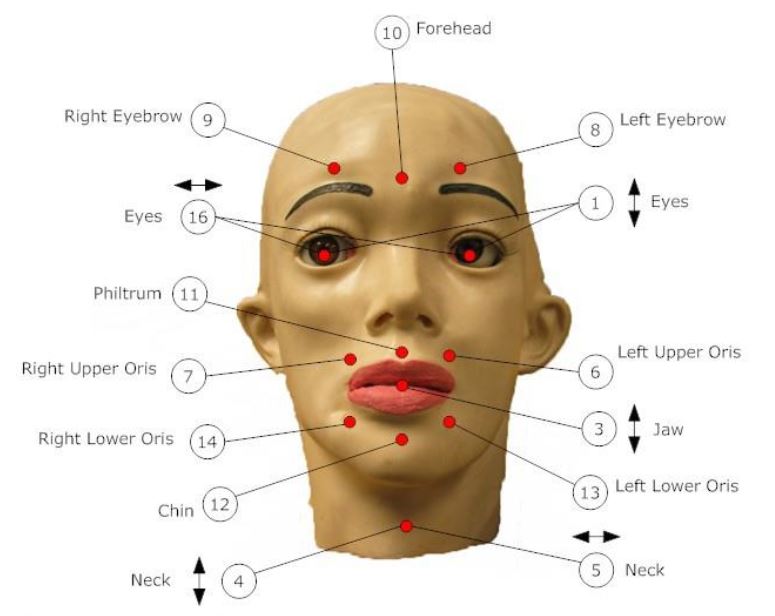
While working on the Marie Curie head this term, there were additional damages that occurred to the head through the “wear and tear” of servos constantly moving the latex and wires simultaneously every which way. The first issue occurred after lengthy usage of the latex was that the corner of the mouth 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.

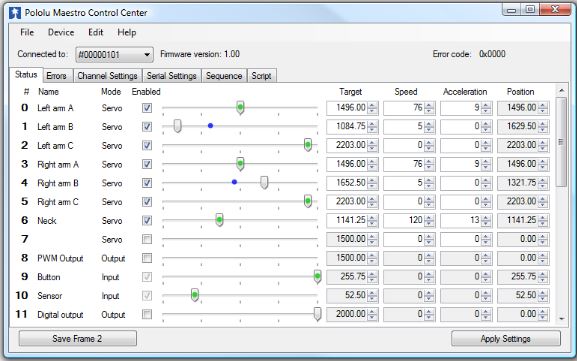
Another issue that came up through constantly using the servos was that some of the wires connecting the servos to the latex came undone on the servo side. Luckily this was an easy fix as all we had to was to tie the wires back to the servo.

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

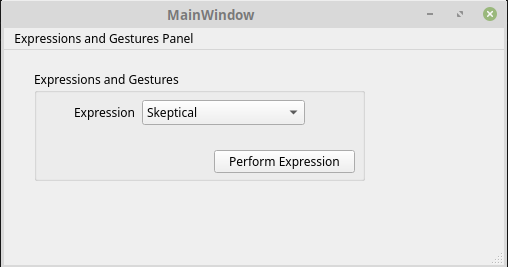
As was seen above, servos are defined in simple markup language (i.e. YAML) which is parsed by the framework in order to create objects and instances within the core algorithms to control the Marie Curie robot. Additionally, a similar approach was desired for representing an expression.

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.

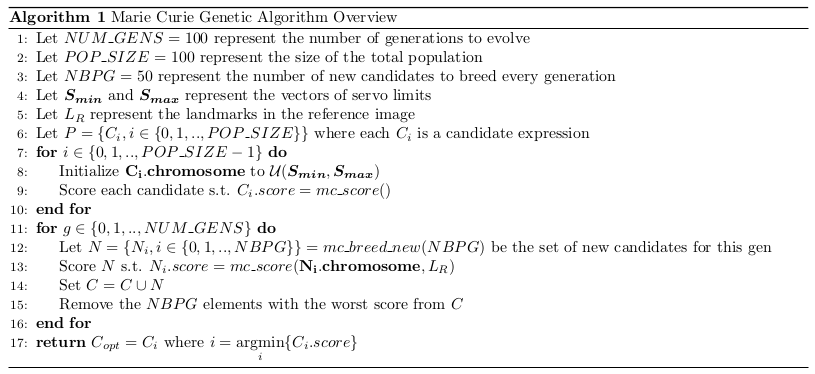
Five phases are considered in a genetic algorithm, which will be discussed later in the report:

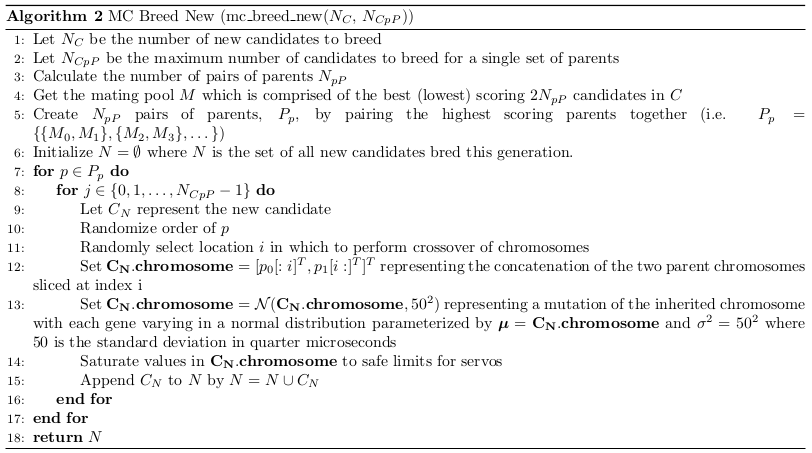
1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

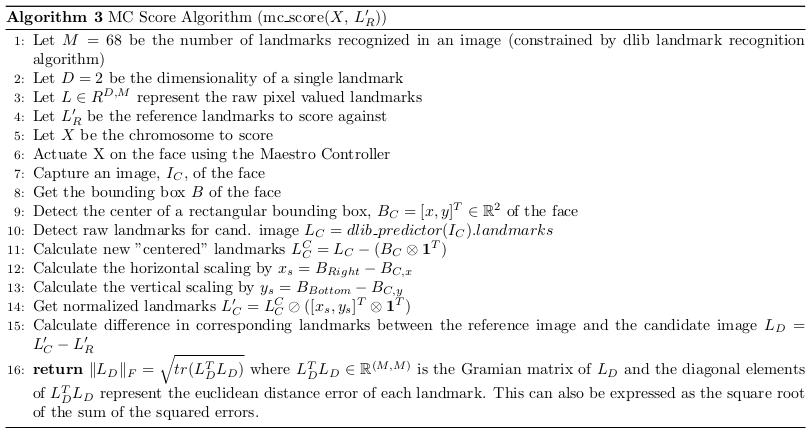
### Algorithm

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.







#### Scoring Algorithm

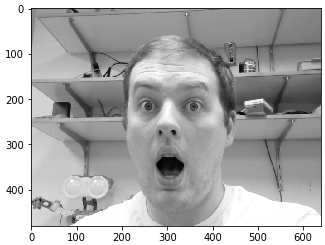
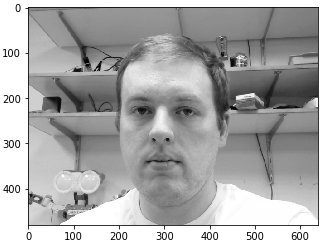
The scoring algorithm needed to be capable of comparing an image to another image to determine how well correlated the two images were. Instead of doing an intensity based image registration it was instead determined that a landmark approach would be preferable. This is because the robot only had an 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.

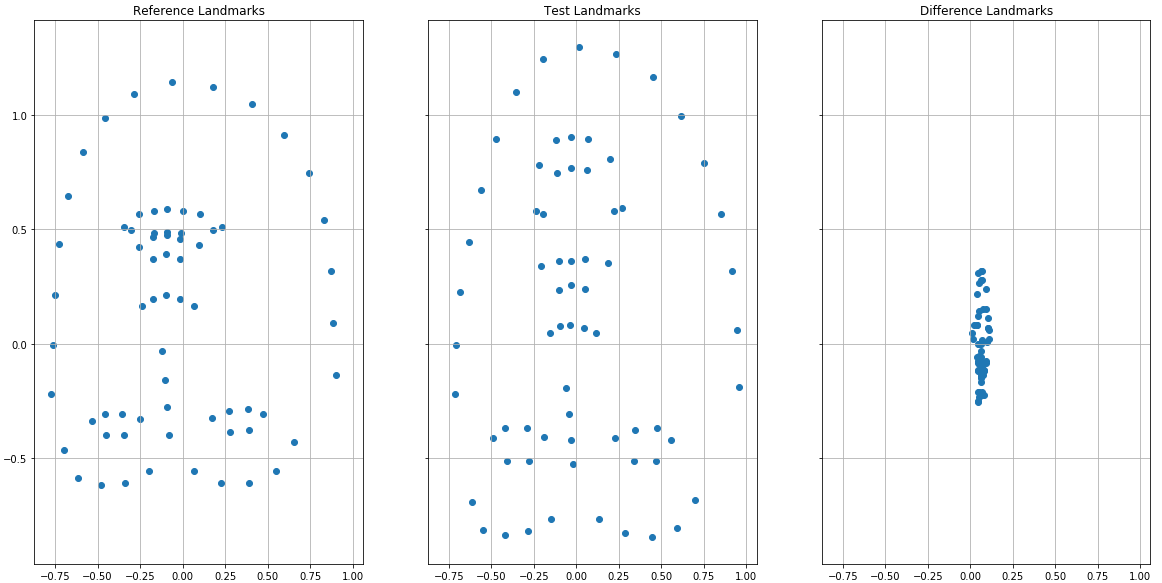
We looked for an existing implementation of a facial landmark detection algorithm which would perform the landmark recognition and found a library called dlib (http://dlib.net) which had a Python implementation of a worthy landmark detector. The implementation is a face detector made using the Histogram 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](https://drive.google.com/drive/folders/1B9JeaWgeQZYPb63GznIJPSqqim_qkPpS?usp=sharing).

Software can be found [here](https://github.com/holtrob/ece579_marie_curie).

#### 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}") |