

ENEE408I - Spring 2018

Capstone Design Project: Autonomous Robotics

Personal Care Robot



Team 5: Jesse Pai, Vladimir Mokrushin, Tewodros Mengistu
Professor Gilmer Blankenship

Table of Contents

Introduction	2
Links to Repository	2
Design and Construction	2
<i>Basic Construction and Wiring</i>	2
<i>Adding Ping Sensors</i>	3
<i>Mounting a Camera</i>	5
Introduction to Daisy's Subsystems	6
Subsystems	7
<i>Low Level Controls</i>	7
<i>High Level Controls and Services</i>	9
<i>Human Interaction Interface</i>	13
Features	18
<i>Person Following</i>	19
<i>Text Messages and Calls</i>	19
<i>Memory Game</i>	20
<i>Exercise Monitoring</i>	21
<i>Database</i>	22
<i>Daisy Dashboard</i>	24
Conclusion	25
Appendix	26

Introduction

The purpose of this capstone project is to design, construct, and program a personal assistant robot, primarily for the elderly. The capabilities for this robot include facial recognition, tracking, following, responding to voice commands, as well as any other additional features we believed to be useful to aid or monitor the health of an elderly person. Some of these features we decided to include were a memory game and video monitoring. With the implementation of a memory game, our robot will be more geared towards the assistance of a person with alzheimer's. For this report, we will be explaining all the steps of our design process, including early builds, discarded and additive changes, and any challenges we have faced. We will also discuss what we have been able to achieve and what we couldn't due to constraints.

Links to Repositories

Daisy Embedded - <https://github.com/J-Pai/DaisyEmbedded>

This repository contains the code that should be loaded onto the Arduino microcontroller. The README contains the requirements and instructions on how to setup the Arduino.

Daisy Jetson - <https://github.com/J-Pai/DaisyJetson>

This repository contains the code that should be executed on the Jetson. The README contains the directions, requirements, and links on how to set up the services and processes.

Daisy Alexa - <https://github.com/tewodros88/DaisyAlexa>

This repository contains the code that also should be executed on the Jetson. It is the code that interfaces with the Alexa/Echo Dot service.

Design and Construction

Basic Construction and Wiring

The basic design of our robot involved a plexiglass base with two motors on the side, each with a wheel attached to it. We were provided an Arduino Uno which we connected to our motors to control them. The power supply we were provided was a 11.1V, 2200mAh LiPo battery which was sufficient to power all our devices. We also attached a third, non-motor

controlled, office chair wheel to the front to add support so the robot does not topple over the front. The process of wiring all our devices was lengthy due to soldering, fitting, and securing in the proper wires to keep everything in place and powered. To ensure equal distribution of voltage to all our components, we installed a terminal block to power up to three devices in parallel. For our case, these devices were the Arduino Uno, the Jetson carrier board, and the breadboard for any additional components such as the motors. The physical implementation on our robot can be seen in *Figure 1* and a rough circuit topology can be seen in *Figure 2*.

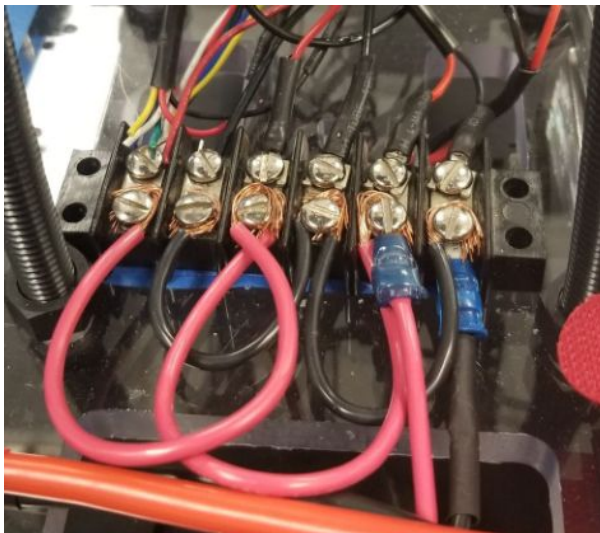


Figure 1: Parallel Power Distribution via Terminal Block

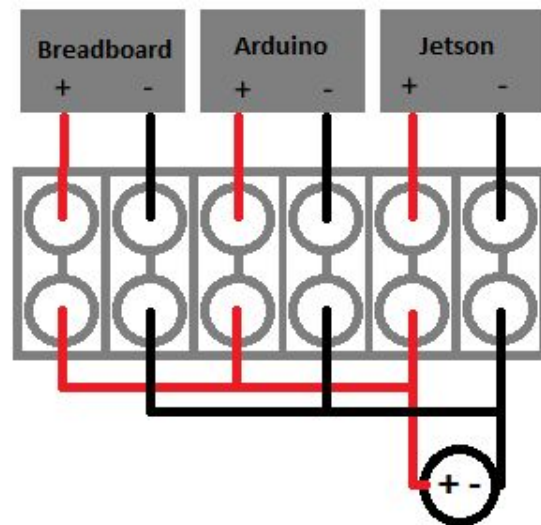


Figure 2: Parallel Power Distribution Rough Topology

Adding Ping Sensors

The next step for our design involved adding ping sensors to our robot. The purpose of these sensors was to prevent our robot from crashing into obstacles by sending a signal to the Arduino which we programmed to halt or turn a specific direction depending on which ping signal was received.. To add the ping sensors to our robot, we got holes drilled into our plexiglass base that allowed us to add mounting brackets. We added these three brackets to the very front of our robot and roughly 45 degrees both left and right from the front. We chose

these locations for our ping because they were optimal to detect all obstacles that could restrict the robots forward movement. These ping sensors can be connected to any available pins on the Arduino as long as the pin isn't dedicated to a different function, in our case we used pins 3, 5 and 7.

Using the Arduino software, we wrote a program that would cause the robot to turn left if there's an obstacle detected by the right sensor and vice versa. To power the sensors, we connected them to 7805 voltage regulators on our breadboard because the sensors were limited to a 5V input while our battery outputted 11.1V. Despite this entire process of adding the ping sensors, we removed them in our final design. When the robot is tracking and following an individual, it will not crash into obstacles in controlled runs. If there is an obstacle restricting the robot's movement and requires turning, it may lose track of the individual and will stop following anyway. *Figure 3* shows the voltage regulator used and the pin layout.

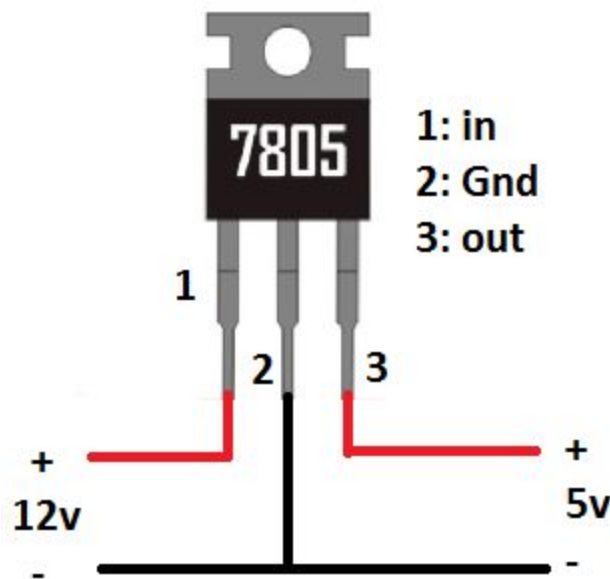


Figure 3: 7805 Voltage Regulator Pin Wiring

The ping sensors were eventually removed from the front of the robot because our team utilized a Kinect2 for our tracking and movement. This meant that the ping sensors served very little purpose in terms of controlling the robot.

Mounting a Camera

The camera we were provided in the lab was a small, fish-eye lens camera. While this camera was sufficient for basic video and tracking, we wanted a better camera. We chose to add an Xbox Kinect camera to our robot. The benefit of this camera included better video quality, which can improve tracking, and distance data. The distance data opened up more opportunities for tracking and made it easier for us to tell the robot to halt at a certain distance away from the person it is following. More information about this usage of distance information will be described in the High Level Controls section.

However, due to the camera's large size compared to the smaller, fish-eye lens camera, we had to find extra space to mount it. On top of this required space, we also needed the camera to be at an elevated height so it could have a better view of the person it is tracking, otherwise it would track our backs when it is too close. We solved this by adding four approximately 1 ½ foot rods secured into our base holding an extra platform at the top. This gave us the much needed space and height for our camera. Additionally, a small box was placed in front of the platform to angle the camera upwards to help it keep track of an individual's face.

Mounting a platform on some rods came with a disadvantage; the rods were not totally rigid and had a tendency to bend and wobble when the robot moved. As a result, this caused our camera platform to wobble, which made tracking more difficult. To mildly alleviate the problem, we tightened zip ties connecting each of the rods near the center. This gave the rods more tension and reduced some of the wobbling. On top of this, we tied and tightened two lengths of twine on the right and left side connecting the mounted platform to the plexiglass base. This reduced most of the major side-to-side wobbling of the platform which was adequate for our purposes. *Figure 4* illustrates the final camera mount with the twine and zip ties securing it.

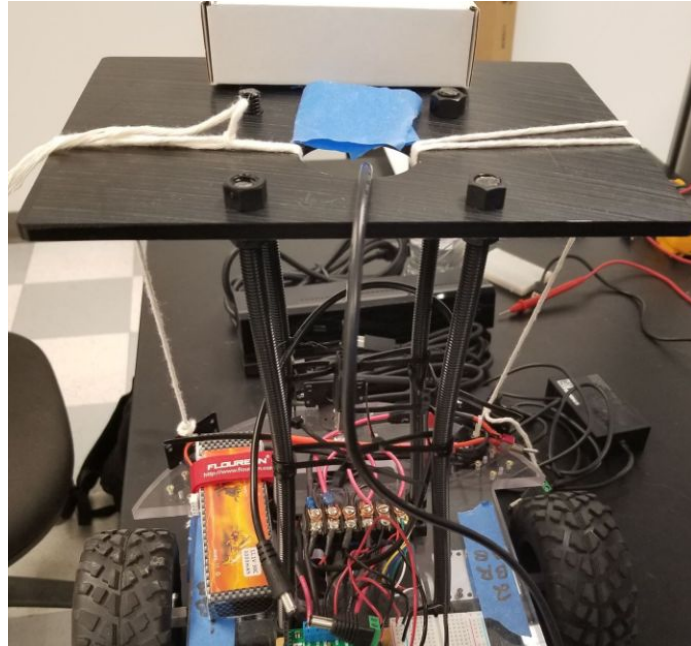


Figure 4: 7805 Voltage Regulator Pin Wiring

Introduction to Daisy's Subsystems

In order to complete the task of tracking and interacting with a person, development of the robot was split into three subsystems or categories. These categories consisted of the Low Level Communication System, the High Level Controls and Services, and finally the Human Interaction Interface. The Low Level Communication (LLC) system is focused on providing an interface to control the motors of the robot. The High Level Controls and Services (HLC) system is focused on setting up the communication between all the subsystems, handling the camera inputs and ultimately doing the decision making on how the robot will move or do something. The Human Interaction Interface (HII) system is focused on providing visuals and handling user input so that the robot can understand what the user wants the robot to do. The HII subsystem also consisted of code that collected data and generated visualizations of the data. In general, the HII took both user input and provided the user with information about the status of the robot.

In general, the code that was written for all modules was designed to be separate and independent. A scheme (which will be outlined in the section for the HLC system) was created to enable communication between the various processes and subsystems. The reason why such a

scheme was designed was because we wanted to avoid running all the processing on the same thread and isolating asynchronous web services from the iterative programs/processes.

Subsystems

Low Level Communication System

A vital part of our project involved the serial communication between our Jetson TX2 computer and our Arduino Uno. This was important because it determined the movement of our robot through the commands implemented for tracking, following, and our Echo device. To do this, we used the PySerial module to import the 'serial' package which gave us access to functions for sending bytes of data from our Jetson to the Arduino. PySerial can be installed using the command "pip install pyserial" from Git Bash. For additional information such as documentation and help on PySerial, visit the following link:

<https://pythonhosted.org/pyserial/>

We added movement functions in Python code such as left, right, halt, forwards, and backwards which were assigned to a specific byte that would be sent the Arduino. We also sent movement speed data so we can control how fast we want our robot to go, with varying speeds for turning and going forward and backward. The Arduino software would receive these bytes and use its own serial communication functions to decode the data into a number that was assigned to specific movement commands. From there, we were able to use that Python script to conveniently call any movement function we needed to implement following and other commands.

The Arduino code also provides functionality for obtaining the measurement for the 3 ping sensors originally on the front of the robot. We did not bother to implement the python side of this functionality since the ping sensors ended being removed from the final design of the robot. They did not serve any useful purpose for the controls of the robot and we were looking to remove as much load from the batter and motors as possible.

The code that executed on the Arduino and handled the bits sent from the Jetson can be found in the following repository: <https://github.com/J-Pai/DaisyEmbedded>. The main Arduino file being DaisyEmbedded.ino.

The code that was used to send bits to the Arduino can be found in the following repository: <https://github.com/J-Pai/DaisyJetson>. The main file that was used to send commands to the Arduino is daisy_spine.py. The python script/code can be executed as a standalone which allows for command line input of movement commands or it can be imported into a file and used by calling the functions in the class DaisySpine. Read up on how to instantiate classes in Python in order to utilize the code or take a look at daisy_brain.py to see how it was utilized in the project.

The following is a table of the integers that are sent to the Arduino by the python script:

Integer Code	Command
0	HALT
1	FORWARD Speed 80
2	TURN Clockwise Speed 50
3	TURN Counter Clockwise Speed 50
4	BACKWARD Speed 80
5 <M1 Speed> <M2 Speed>	Move with M1 Speed and M2 Speed
253	Return Left Ping Distance
254	Return Right Ping Distance
255	Return Middle Ping Distance
Other	HALT

Integer code 5 was a special code that allowed for multiple bytes to be passed to the Arduino. By sending code 5 to the Arduino first, the Arduino code would then wait to receive 4 additional bytes. The first 2 bytes were the left motor's integer speed and the next 2 bytes were the right motor's integer speed. It is important to note that integers on the Arduino are 2 bytes each. Normally integers are 4 bytes (32-bits). Keep this in mind when trying to pass integers to the Arduino.

High Level Controls and Services

The next primary subsystem is the High Level Controls and Services. This subsystem contains the code that enables identification and tracking of a person and handles message/data passing between the various modules.

Human Tracking

For us, the first step to achieving human tracking was human identification. In order to achieve this, we decided to use a Python library written by ageitgey. This Python library utilized DLIB library and also had a portion of the code optimized for CUDA. In general, the performance of the facial tracking was about 2 - 3 frames per second running on a 1080p camera stream. The library can be obtained from the following link:

https://github.com/ageitgey/face_recognition.

We originally intended on utilizing the fisheye lens camera as our primary camera, but due to the way the fisheye lens warped the objects on the image, the performance/accuracy of the facial recognition algorithm was not great. Since our team was not particularly good at Machine Learning, we did not try to modify the code to improve performance. Some experiments determined that it was the facial detection (not identification) algorithm that took the most amount of time and processing power. An optimization on this part of the identification process would certainly help with performance.

With facial identification working, it was noted that this approach would only work if the user was facing the camera at a specific angle. In addition, due to its slow performance (2 - 3 FPS) there needed to be a faster way of following the individual through the scene. For this task, we utilized a tracker that was implemented in the OpenCV Extra Modules code base. See the following links for more information: https://github.com/opencv/opencv_contrib and <https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/>. After initializing the tracker on an individual's face/head, the tracker could then take over and follow the head of the individual while the tracking algorithm was active. The tracker provides a lot more flexibility in terms the object/individual movement throughout the scene and was somewhat able to follow the head of an individual when the individual turned around or was not facing the robot. Testing around for the trackers, we determined that the CSRT tracker performed the best under the

widest environment and conditions. A list of all potential trackers implemented by OpenCV can be found in the documentation page for OpenCV trackers

(https://docs.opencv.org/trunk/d0/d0a/classcv_1_1Tracker.html).

Despite using trackers, it was quickly determined that due to the low placement of the camera (about 1.5 feet off the ground) and the resulting 45 degree angle necessary for the camera to see an individual's face, when the user turned around, the user's back would cover up the head that the tracker was tracking and as a result, fail to continue tracking the individual. This meant that the initial bounding box for the tracker needed to be initialized on a larger portion of the person's body in order to enable more reliable and accurate tracking. To do this, we decided to leverage the Microsoft Xbox Kinect2.

The Kinect2 provides both a better, less warped, view of the individual. In addition, it provided an excellent auto mode (balancing of whites and colors) and the all important distance data. The main advantage the Kinect2 had over the Xbox 360 Kinect is that the depth data can be obtained in a 1 to 1 (pixel to pixel) pairing with the RGB data. That means RGB pixel (100,100) and distance data at (100, 100) represents the same point in space. Using this information, I was able to effectively obtain an outline of the individual we wanted to track and use that to initialize the tracker. The code that does all of this is in the daisy_eye.py script found here: (https://github.com/J-Pai/DaisyJetson/blob/master/daisy_eye.py).

Even with a larger tracking box, the tracker will still eventually drift away. As such, it became necessary to weave the facial recognition algorithm into the tracking code every other frame. To further boost performance, we needed to crop out a small part of the 1080p video stream generated by the Kinect2 and feed it into the facial recognition algorithm. For this, we created a function called `__crop_frame()` which allowed us to cleanly crop out a section of the 1080p image. This helped triple our framerate and ensured that the facial tracking algorithm only worked when the individual's face was in the center of the image. To boost the performance of the tracker, we also scaled the image of from the Kinect2 down to a lower resolution so that the tracker's performance and accuracy could be improved. The function to do this is called `__scale_frame()`. The following is a flowchart that shows how the image data is created and processed every iteration.

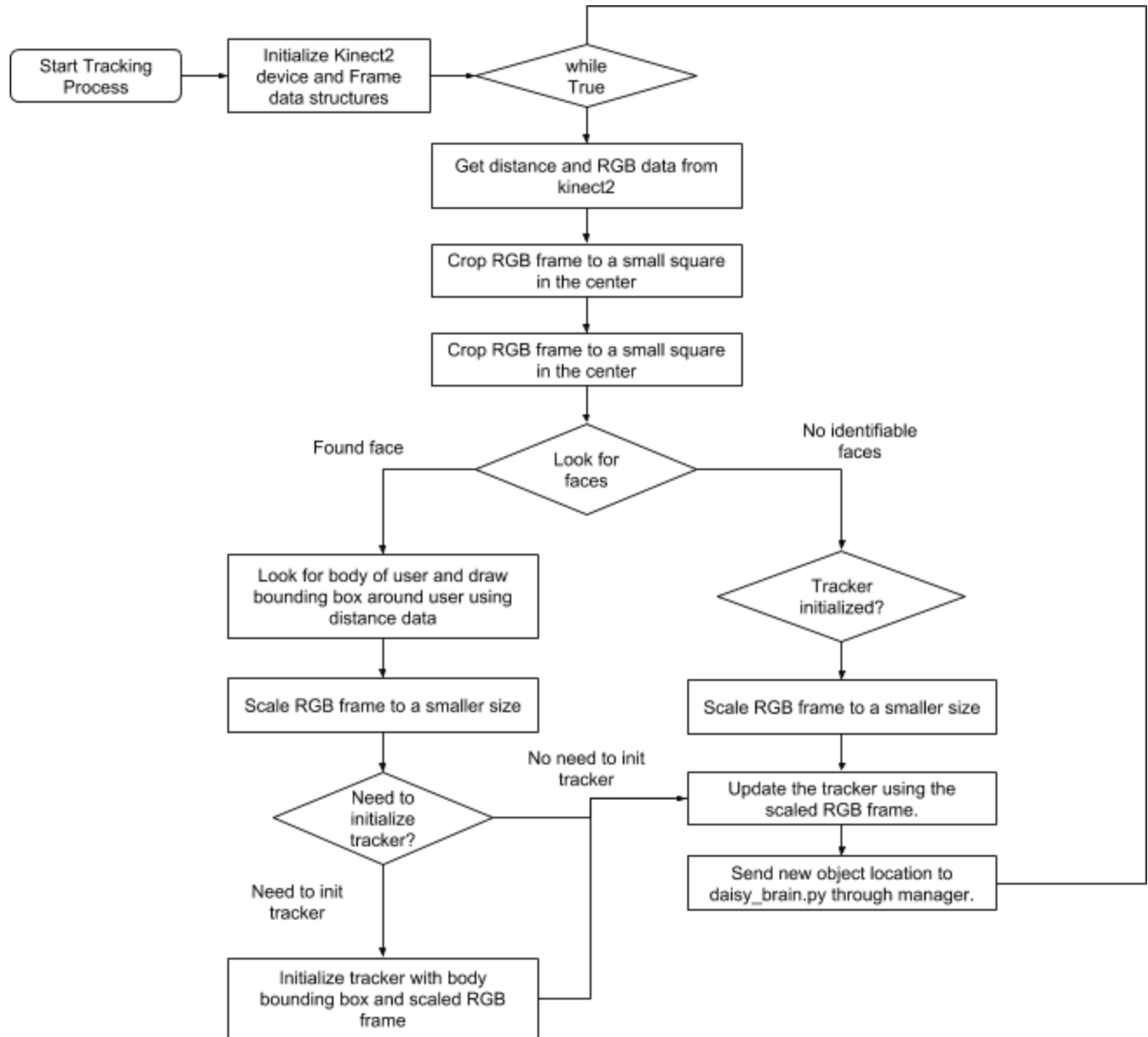


Figure 5: *find_and_track_kinect* control flow

The primary control code for the Daisy robot can be found in the python script `daisy_brain.py`. As the name suggests, this script does the primary decision-making process for the entire robot. An important feature of `daisy_brain` is the fact that both the control algorithm and tracking algorithm noted previously run on separate threads. This means that the tracking algorithm will not impact the control algorithm and the control algorithm is free to run as fast as it possibly can. The tracker algorithm is simply generating data for the control algorithm to

consume and make decisions with. The design is also helpful when it came to handling Alexa commands which effectively act as interrupts in the standard control and tracking of an individual.

Interprocess Communication (Neurons)

Many of the algorithms and applications that support Daisy's capabilities are ran as separate processes or applications. This separation ensures that if one service goes down or crashes, the other processes will not be impacted. Unfortunately, this also means that passing information and data to different services becomes a non-trivial task. An example of data being passed to different services for our implementation can be seen in our video streaming feature that can be found on the web based dashboard we created for Daisy. As the feature name suggests, Daisy has a web application dashboard that allows others to remotely see what Daisy sees through her Kinect2 camera. Since camera output cannot be shared across multiple applications (once an application binds to the Kinect2, other applications are locked out from it), an advanced scheme will need to be set up in order to pass image data to the web server that hosts the dashboard. This task cannot simply be solved by creating a shared text file in the filesystem as this approach would be incredibly time consuming and stressful on the disk/SSD.

To solve this problem we leveraged a multiprocessing feature in Python called managers. What effectively occurs is a manager (which acts like a local webserver) stores a system global data structure. In our case, we used a manager based implementation of a python dictionary. From here, different applications which need access to certain pieces of information and data subscript to the data structure and the data structure is updated/read from as if it was a normal object in the application. The implementation of the manager server can be found in the python script `daisy_neuron.py` found at the following link:

https://github.com/J-Pai/DaisyJetson/blob/master/daisy_neuron.py.

With the neuron script running, other scripts can then connect to the neuron to access its global data structures and make updates to the data structure's state. Examples of the implementation can be found in the following files:

- `daisy_server.py`:

https://github.com/J-Pai/DaisyJetson/blob/master/daisy_server.py

- daisy_brain.py:
https://github.com/J-Pai/DaisyJetson/blob/master/daisy_brain.py
- daisy_eye.py:
https://github.com/J-Pai/DaisyJetson/blob/master/daisy_eye.py
- DaisyAlexa.py:
<https://github.com/tewodros88/DaisyAlexa/blob/master/DaisyAlexa.py>

For more information on Managers check out the documentation at the following link:

<https://docs.python.org/2/library/multiprocessing.html#managers>

As described before, the most interesting interaction/usage of managers can be found in daisy_eye.py and daisy_server.py. The daisy_eye.py process/thread acts as a producer of image data which updates a data structure (dictionary) stored in the manager and the daisy_server.py acts as a consumer of the image data which reads the data stored in the data structure (dictionary) and outputs the image onto the web based dashboard. These two applications run on separate threads and are effectively independent of one another other than the relationship/connection through the manager.

When using managers it is important to use properly synchronized data structures. The manager library in Python contains reimplemented versions of standard python data structures that protect the data structure from concurrent access while ensuring that a thread cannot starve out other threads when attempting to access the data stored on the manager. We consider the neuron/manager implementation of Daisy to be the biggest achievement of this group since it opens up a lot of new options for future groups to pass data between different workflows/applications and potentially different computer systems.

Human Interaction Interface

Using Amazon Echo

Initially, we started out by connecting the Echo Dot to the internet and our Amazon account. This was done using the Amazon Alexa app on our phones. There are four buttons located on the Echo dot and the one with a dot is referred to as the action button. This button needs to be pressed down for 5 seconds for the Echo dot to be in its set up mode. Once the Echo

dot is in setup mode you will see an orange light coming from the device and you can connect it to the UMD wifi through the Amazon Alexa app.

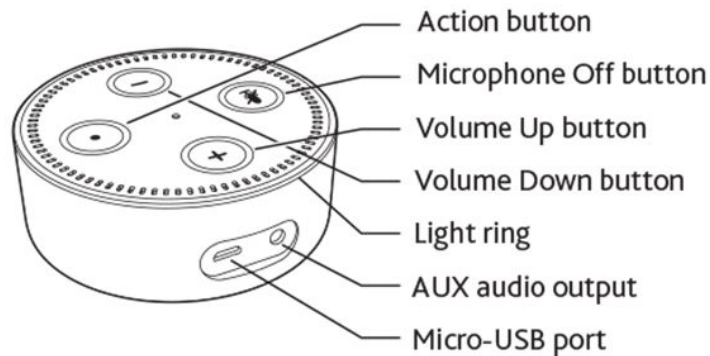


Figure 6: Amazon Echo Dot Basics

After having the Echo dot completely set up, we wanted to be able to send voice commands to our robot and perform the stated command. This would be accomplished by running a Python script on the Jetson that could receive the voice commands from the Alexa and trigger some action after a command was stated. There are multiple Python libraries that can accomplish this and we found Flask-Ask to be one of the best options out there.

Finally, we wanted to be able to connect our python script with the commands we built on the Alexa skills Kit website. This would allow us to grab data that was stated to the Echo dot and execute code off of what was said. We used an application known as Ngrok to provide a web tunnel that links our Python script to the skills made on the Alexa Skills Kit. Once we executed Ngrok on a local port it provides an https link that we use as an Endpoint for the Alexa skills kit.

Alexa Skills kit

When working with the Alexa skills kit there are four steps required to build a skill. They are the following, an invocation name, Intents, building the model, and providing an Endpoint.

1. Invocation Name

- The invocation name is a name that will start an interaction between your robot skills and the Alexa. In our case, we used the name of our Robot which was Daisy. The skill would activate when we stated “Alexa start Daisy.”

2. Intents, Samples, and Slots

- Next, we wanted to create an intent, this would be the commands that are robot would execute. An example of one of our created intents would be the MoveIntent shown in the Figure 7 below. The first thing you need is to provide an utterance or phrase to invoke the intent. In our case we state “ move {direction}” or just “{direction}.” The reason "directions" is in a curly bracket is because it is a custom intent slot that we have created. Slots are useful because they allow us to parse the data in our python script.

Intents / MoveIntent

Sample Utterances (2) ⓘ

What might a user say to invoke this intent?

move (direction)

(direction)

Intent Slots (1) ⓘ

ORDER ⓘ	NAME ⓘ	SLOT TYPE ⓘ
1	direction	LIST_OF_DIRECTIONS

Figure 7: MoveIntent from Alexa Skills Kit

- In order to create a slot go to Slot Types and click the Add button. Once you have provided a name you want to start to provide slot values. In our case, we created a slot type known as LIST_OF_DIRECTIONS and the values for this slot are left, right, forward, and backward. This is shown in the Figure 8 below.

Slot Types / LIST_OF_DIRECTIONS

Slot Values (8)

VALUE	ID (OPTIONAL)	SYNONYMS (OPTIONAL)		
halt	<input type="text" value="Enter ID"/>	<input type="text" value="Add synonym"/>	+	
move	<input type="text" value="Enter ID"/>	<input type="text" value="Add synonym"/>	+	
backward	<input type="text" value="Enter ID"/>	<input type="text" value="Add synonym"/>	+	
forward	<input type="text" value="Enter ID"/>	<input type="text" value="Add synonym"/>	+	
right	<input type="text" value="Enter ID"/>	<input type="text" value="Add synonym"/>	+	
left	<input type="text" value="Enter ID"/>	<input type="text" value="Add synonym"/>	+	

Figure 8: Slot type LIST_OF_DIRECTION created on Alexa Skills Kit

- Lastly, going back to the Move intent, you can see under Intent Slots we have "directions" which is given the Slot Type we created known as LIST_OF_DIRECTIONS. In the case someone states “move left,” the MoveIntent will be invoked and we can parse direction in our Python script which will be equal to left in this case.

3. Build Model

- After creating your intents and slots you want to simply click the Save and Build model buttons at the top of the page.

4. Endpoint

- As stated before, we want to link our Amazon skills with our Python script so that our robot can execute these commands. We used Ngrok and after running the application on our machines localhost it provides us with an https link, shown in the Figure 9 below.


```
Tunnel Status      online
Version            2.1.3
Region             United States (us)
Web Interface      http://127.0.0.1:4040
Forwarding          http://95e26af4.ngrok.io -> localhost:3000
                   https://95e26af4.ngrok.io -> localhost:3000

Connections        ttl    opn    rt1    rt5    p50    p90
                   0      0      0.00   0.00   0.00   0.00
```

Figure 9: Ngrok screenshot showing https link

- Copy this link and paste it into Default Region and select HTTPS. It is also required to state that the endpoint is a subdomain under Default Region as is shown in Figure 10 below. Once you click Save Endpoints you have completed creating your Amazon Skill.

Endpoint



The Endpoint will receive POST requests when a user interacts with your Alexa Skill. The request body contains parameters that your service can use to perform logic and generate a JSON-formatted response. Learn more about Lambda endpoints here. You can host your own HTTPS web service endpoint as long as the service meets the requirements described here.

Service Endpoint Type

Select how you will host your skill's service endpoint.

☐ AWS Lambda ARN (Recommended)

☒ HTTPS

Default Region (Required)

My development endpoint is a sub-domain of a domain that has a wildcard certificate from a certificate authority...

Figure 10: Endpoint field on Alexa Skills Kit

Flask-Ask

As stated before, we used the Python library Flask-Ask to allow us to trigger commands for our robot after certain phrases or words were said to the Echo Dot. To install Flask-Ask we simply typed “pip3 install flask-ask” and imported the library to our python script. The full documentation is present at http://flask-ask.readthedocs.io/en/latest/getting_started.html.

In terms of our project, we used Flask-Ask to execute commands based off of the skills we created using the Amazon skills kit. One of the Intents we created was the MoveIntent

shown in Figure 7 above. In Figure 11 below we show the code written to have the robot move in a certain direction. In line 33 we use the `@ask.Intent` decorator to map the `MoveIntent` to a view function `move`. The `move` function takes `direction` as an input, which is a slot type of all the possible directions our robot will move. With this you can execute another function such as `TurnRight()` under the 'if' statement that tells your robot to start moving right. In Figure 11 below we look at the value of `direction` and set the value of 'msg' which will be a returned statement that Alexa will say. In the case that a person only says "move," we can have the Echo Dot return a question as shown in line 44 of Figure 11 that asks the user "in what direction?" Returning a question triggers the Echo Dot for a response and, if nothing is stated, it will timeout. In our case, we re-prompt the user, if they don't state anything, to "please give a direction." This is how we utilized Flask-Ask to execute voice commands given to the Echo Dot. Running Ngrok in the same directory will allow your Python script to connect with the Skills you created and the Alexa. The Python script we used to communicate with the Alexa can be found on the following Github link: (<https://github.com/tewodros88/DaisyAlexa>).

```
33 @ask.intent("MoveIntent")
34 ▼ def move(direction):
35     if direction == 'left':
36         msg = "moving left"
37     elif direction == 'right':
38         msg = "moving right"
39     elif direction == 'forward':
40         msg = "moving forward"
41     elif direction == 'backward':
42         msg = "moving backward"
43     elif direction == 'move':
44         return question("In what direction?").reprompt("Can you please give a direction?")
45     return statement(msg)
```

Figure 11: Code for MoveIntent using Flask-Ask

Features

Person Following

One of the basic functions of Daisy is how the robot is able to track and follow a person. After starting up the robot and Alexa, the user simply starts the Alexa interaction for Daisy (Alexa, start Daisy) and tells the interaction to begin tracking a person. It is important that the person's face and name is registered with the `AlexaDaisy.py` and `daisy_brain.py` code. From

here, the tracking algorithm (daisy_eye.py) will begin looking for the tracked individual's face around the middle of the image stream collected from the Kinect2. This process was already described in great detail above.

With a bounding box tracking the individual's upper body, the bounding box data is then passed to daisy_brain.py's control thread/loop. The control loop is very simple. If the bounding box moves to the right of the view of the robot, the robot will turn counter-clockwise to center the bounding box in the middle of the view of the robot. If the bounding box moves to the left of the view of the robot, the robot will turn clockwise to center the bounding box in the middle of the view of the robot. Using the distance data pulled from the Kinect2, the robot will move forward if the individual moves away from the robot and will move backward if the individual moves towards to the robot. The code for this interaction can be found in the daisy_eye.py and daisy_brain.py scripts (https://github.com/J-Pai/DaisyJetson/blob/master/daisy_eye.py and https://github.com/J-Pai/DaisyJetson/blob/master/daisy_brain.py).

Text Messages and Calls

We were able to have the robot send text messages and make calls using Twilio. Twilio is a developer platform for communications. Many companies such as Lyft use the Twilio API to add capabilities like voice, video, and messaging to their applications. In our case, we used the free trial version to simply make either a call or a text stating "Hello from Daisy."

In order to set up Twilio, we imported the library by typing in the command "pip3 install Twilio." After creating an account, we are provided with a personal phone number, a SID number, and Authentication Token. The code needed to execute text messages and phone calls is very simple with an example shown in Figure 12, this example and much more are also available in the documentation found on the Twilio website and the link in the appendix section below. For our robot, we created a Text Intent on the Amazon Alexa Skills kit to execute the code shown in Figure 12 below. After a user stated "text Teddy," the code would send the message "Hello from Daisy" to the specified user. This is how we managed to add text messaging and call features to our Python script and robot.

```

51 client = Client(account_sid, auth_token)
52
53 message = client.messages.create(
54     body="Hello from Daisy!",
55     from_="+15555555555",
56     to="+12316851234")
57
58 print(message.sid)

```

Figure 12: Code for texting using Twilio

Memory Game

The purpose of the project was to create a companion robot with an elderly user in mind. Something we wanted to add to our robot was a memory game feature that would be triggered by the Echo dot. The game would state 5 numbers and ask the user to repeat the number sequence backwards. After playing the game, the Echo Dot would return a message stating whether the user won or lost the game. This would test the user's memory capabilities and either help further sharpen their memory or monitor their memory over time.

We built the memory game using an example shown on the Alexa blog for Flask-Ask, the link for this is given in the appendix section. In the given example, three random integers are generated and stored in the list known as numbers. The list is then reversed using `session.attributes` and stored in a new list known as `winning_numbers`. These numbers are then compared to the stated numbers that the user provides to the Echo Dot and responds with either the winning or losing message.

In terms of the Alexa skills kit, two intents are created: `GameIntent` and `AnswerIntent`. `GameIntent` generates the three random number and the reversed list of these number. This intent is invoked by stating “play memory game.” Once the number list is made, the function returns a question that asks the user if they can repeat all the number sequence in reverse order. The `Answer Intent` is then invoked when the user responds with a number. This is done by specifying ‘first’, ‘second’, and ‘third,’ in line 27 in Figure 13, as a slot type known as `Amazon.Numbers`. `Amazon.Numbers` is a built-in slot type that can parse integers stated to the Alexa. After converting the strings into integers we can compare it to the `winning_numbers` list and return a statement for winning or losing the game as seen in Figure 13 below. In terms of

our project, we edited the code in Figure 13 below to state five numbers instead of three and record the performance of our group members. We used this data create a plot showing the individual's performance over time. This can be seen on our Github page:

(<https://github.com/tewodros88/DaisyAlexa>).

```
12 @ask.intent("GameIntent")
13
14 def next_round():
15     numbers = [randint(0, 9) for _ in range(3)]
16
17     round_msg = render_template('round', numbers=numbers)
18
19     session.attributes['numbers'] = numbers[::-1] # reverse
20
21     return question(round_msg)
22
23
24
25 @ask.intent("AnswerIntent", convert={'first': int, 'second': int, 'third': int})
26
27 def answer(first, second, third):
28
29     winning_numbers = session.attributes['numbers']
30
31     if [first, second, third] == winning_numbers:
32
33         msg = render_template('win')
34
35     else:
36
37         msg = render_template('lose')
38
39     return statement(msg)
```

Figure 13: Code for Memory Game

Exercise Monitoring

Although our robot was initially targeted towards elderly persons with declining memory, either through old age or Alzheimer's, we added an exercise monitoring feature to our robot as a bonus. The way the monitoring access stores data is similar to the memory game. That said, the exercise monitoring feature leverages the Kinect2 to track how many squats an individual can do. The algorithm is quite simple and can be found inside of daisy_brain.py (https://github.com/J-Pai/DaisyJetson/blob/master/daisy_brain.py).

The way the algorithm works is that it simply starts tracking an individual and as the individual does squats, the tracking box follows the individual down. The tracking code is not modified in any way when compared to the standard tracking/following mode of the robot. With the tracking bounding box following the individual's upper body, after the individual's chest moves past a certain point, a squat count is incremented. After the exercise session is complete, the data/count is then passed to the Alexa code and the data is stored into the database.

Database

As we added features such as the memory game and exercise mode, we wanted to record the user's performance in these features. To accomplish this we used mLab which is a cloud database service. This application allowed us to store up to 500mb of data online for free and is where we stored the records for our memory game and exercise mode. This allowed us to keep track of how many times a user had played the memory game, their overall scores on the game, and the number of squats performed in the exercise mode. This data was critical when creating a plot that showed the overall performance of the user over each time played.

In order to set up the database, we first created an account on <https://mlab.com>. The next step was to choose the free 500mb AWS cloud service, this will create the database and provide us with a MongoDB URI that can be used to access the database in our python script. We then create a collection in the database known as `memory_records` and `exercise_records`, these store performance data in the form of a dictionary as shown in Figure 14 below.

Once we had completed creating the database we created three functions known as `getRecord()`, `pushRecord()`, and `updateRecord()`, this is shown in Figure 15 below. The `getRecord()` function takes in an `id_num` which is a specific number that identifies a member of our group, in the case of Jessie, his `id_num` is equal to 0 as shown in Figure 14 below. Once we have identified and accessed Jessie's records, the function returns a dictionary with all the user data stored. The next function is `pushRecord()` which simply pushes a given record to our database. Lastly, we have the `updateRecord()` function that updates the specified fields in a dictionary. This is accomplished by creating a new dictionary known as "updates" that stores the new values you want to be entered into records. Once the fields in the record dictionary have

been updated with the function `updateRecord()`, we can use the `pushRecord()` function to send the updated dictionary to our database.

Overall, these three functions allowed us to keep track of performance data after each iteration of our memory game and exercise mode. The functions also allowed us to make edits to the performance data once the games had been played and update our records. The complete code for how this was utilized in our project can be found on our GitHub page:

(<https://github.com/tewodros88/DaisyAlexa>).

```
1 {
2   "_id": {
3     "$oid": "5af1033f6c385277a2bb54f2"
4   },
5   "score": 0.8,
6   "overall_score": 180,
7   "user": "Jessie",
8   "id_num": 0,
9   "count": 2,
10  "overall_performance": 90.00,
11  "data": [
12    100,
13    80
14  ]
15 }
```

Figure 14: Stored record in mLab

```
25 MONGODB_URI = "mongodb://User:password@ds123889.mlab.com:12389/records"
26 client = MongoClient(MONGODB_URI, connectTimeoutMS=30000)
27 db = client.get_default_database()
28 records = db.records
29
30 def getRECORD(id_num):
31     record = records.find_one({"id_num":id_num})
32     return record
33
34 def pushRECORD(record):
35     records.insert_one(record)
36
37 def updateRecord(record, updates):
38     records.update_one({'_id': record['_id']},{
39         '$set': updates
40     }, upsert=False)
41
```

Figure 15: Code for receiving, pushing and updating records in database.

Daisy Dashboard

Daisy contains a dashboard that allows the user to access visualizations of the collected data from the memory game and exercise modes. Another feature of the dashboard is the fact that the user can also view/see what the robot is seeing or doing at that time. It provides a live video stream of the current camera's view. The dashboard was created as a way to allow family members to visibly see their loved ones remotely and allow the tracking and following aspect of the robot to serve more than just keeping the robot close to the user at hand. The implementation of the dashboard can be found in the Python script `daisy_server.py`. On the initial access of the webpage, the remote user will need to type in a username and password to login. The username and password can be found embedded in the source code for the webserver. We acknowledge that this is not secure, but we did not spend too much time building and developing a user account system as it was not a major focus of the project. The following is a screenshot of the dashboard and what can be found on the page.

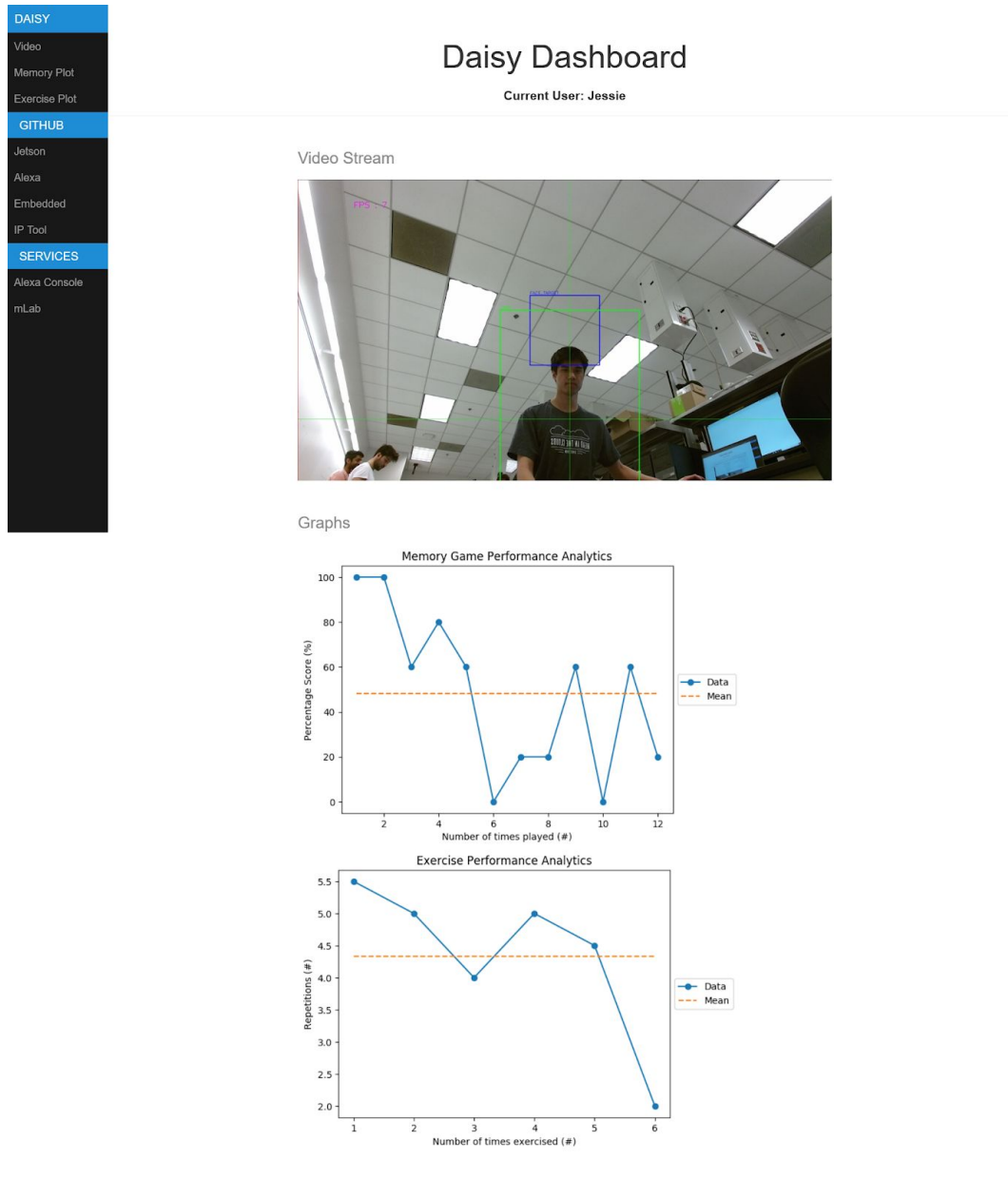


Figure 16: Screenshot of the Dashboard page

Conclusion

Overall, we were satisfied with the final result of our personal care robot. Our initial goals for this project involved getting tracking and following working, implementing texts and calls through the Echo, and adding a memory game. The majority of our time was spent on employing tracking and following, but after that was finished, we realized we had a lot more

time to add extra features such as exercise monitoring, a database, and a dashboard for our robot. Given the time and hardware constraints of this project, we have accomplished everything we have wanted to do and more. We hope this report serves well as a guide to future groups seeking to implement similar features to their robots and that our explanations and solutions listed throughout the report are concise and helpful.

Appendix

<https://www.twilio.com/docs/sms/quickstart/python?code-sample=code-send-an-sms-using-twilio&code-language=py&code-sdk-version=6.x> - Twilio Documentation on Texts

<https://developer.amazon.com/blogs/post/Tx14R0IYYGH3SKT/Flask-Ask-A-New-Python-Framework-for-Rapid-Alexa-Skills-Kit-Development> - Using Flask-Ask

<https://docs.mlab.com/> - mLab Documentation for creating database