Hexapod Report

Patrick Gmerek, Ammar Khan, Emma Smith, and Charles Stoll

Fall 2018

Table of Contents

Hardware

Introduction

The hardware of this project was fairly straightforward and involved CAD modeling, soldering, and wiring. The main theme of the hardware design was to not modify the hexapod irreversibly; all changes and additions can be easily undone to return the hexapod to how we initially received it.

Goals

1. Attach humanoid robot torso to the top of the hexapod.
2. Replace previous microcontroller with a Raspberry Pi.
3. Provide a means to power the servos safely with either a battery or a wired power supply.
4. Provide adequate ventilation or active cooling to the Raspberry Pi.
5. Securely fasten all screws and nuts to prevent the loosening of hexapod legs when walking.
6. Attach a bobble head to the top of the torso for comedic effect.

Design

Since we were told what was expected in terms of hardware for our robot at the beginning of the project, we didn't have too much freedom with the design. However, below are descriptions of the robots when we first got them.

Hexapod Robot

This robot has six legs, each having three servos. The legs are mounted around of circular frame 60 degrees apart. The body of the hexapod consists of two metal plates separated by long standoffs. In between these plates are 6 of the 18 servos, and the microcontroller. All 18 servos present at the start of the project were the HS-485HB Deluxe by HiTec. At the time of this writing, these servos could be source online for about $15 a piece. The principle design of the hexapod body was not modified, but the torso of the humanoid robot was attached to the top using an acrylic adapter.

Humanoid Robot

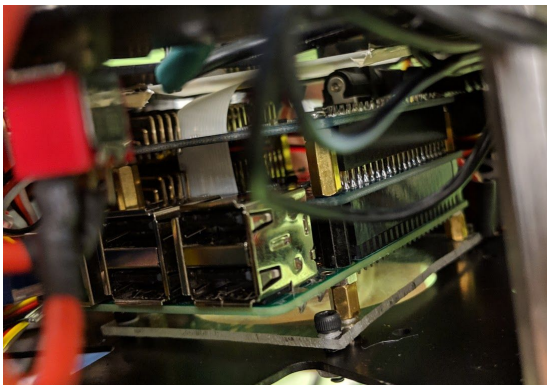
This robot had two arms, two legs, and a servo to mimic turning at the waist. After disassembling this robot, the torso and legs were separated. The torso was mounted atop the hexapod but the legs were not use in this project.

Implementation

At the start of the project, the hexapod was barren. The aside from the servos, the only other component attached to the robot was the old microcontroller within the cage. We decided right away that we didn't want to try to reuse this microcontroller, and instead opted for a Raspberry Pi 3 Model B+. This is because we were unsure if the old microcontroller could run Python scripts and we were more familiar with Raspberry Pi. However, the Raspberry Pi cannot drive servos by itself; separate expansion boards, called *hats*, are required to generate Pulse Width Modulation (PWM) signals in order to control the servos. The hats we chose are the Adafruit's 16-Channel PWM/Servo hat. Two of these hats are required to drive the 25 servos present on the combined robot. These hats communicate via I2C and up to 62 hats can be stacked on a single Raspberry Pi. More information about the Adafruit 16-Channel PWM/Servo hat can be found on their website.

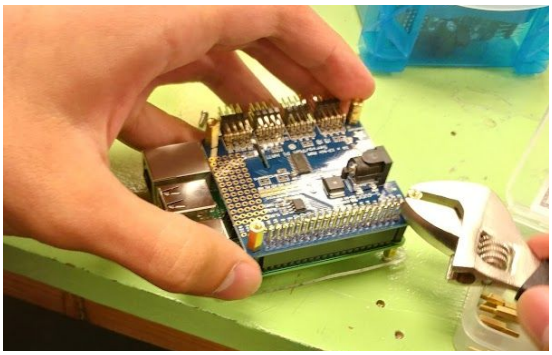
The Adafruit hats ship with straight headers for the servo connections. This kind of header is suitable if only one hat is used, but since we were using two hats, right-angle headers were required. To let their customers choose what sort of header to use, Adafruit ships the boards without the headers soldered on. As such, for each board we had to solder 48 pins for all four angled headers. This wasn't too difficult but a fine-tipped soldering iron is required. We initially soldered the headers for two boards, but ended up soldering an additional three more.

The hats include a place to solder an optional capacitor to maintain a constant voltage when a high current load is placed on the power supply. We did not use this feature because we didn't have high capacity, yet physically small capacitors that could fit in between the hats.

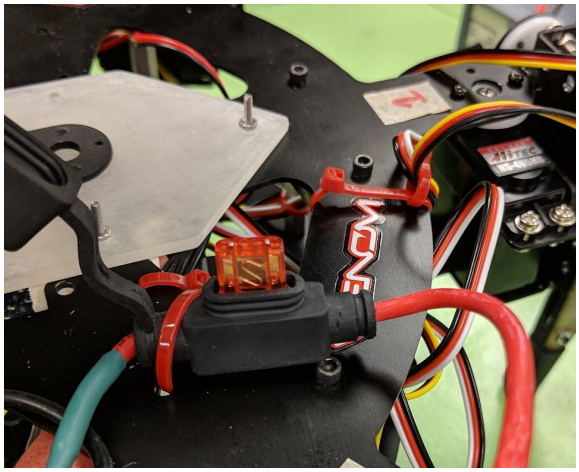


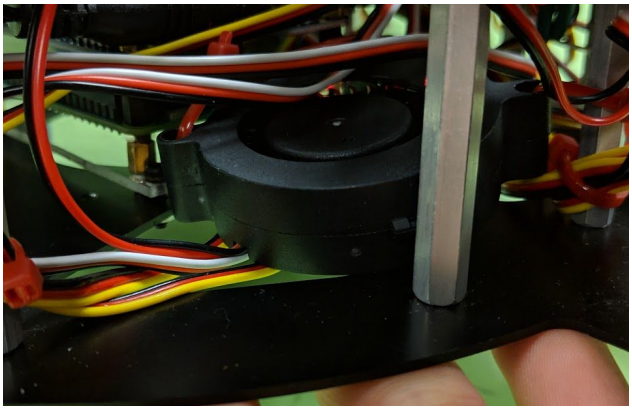
After removing the old microcontroller, we discovered that the mounting hole patterns between the previous microcontroller and the Raspberry Pi were different. This meant we had no easy way to secure the Raspberry Pi to the robot. Instead of drilling more holes in the hexapod frame to mount the Raspberry Pi, we opted to fabricate an adapter to reuse the existing mounting holes. This was modelled in Autodesk Fusion 360 with the intent using the laser cutter in the EPL to fabricate the adapter. However, all suitable laser cutters in the EPL were broken at the time so we opted to cut acrylic by hand and use a drill press to make our adapter. The CAD files for the design are in the project Git Repository so that it may be reproduced with a laser cutter if needed.

The Raspberry Pi, Adafruit Hats, and acrylic adapter were fastened together using M2.5 standoffs and screws. The whole assembly is mounted within the hexapod with M2.5 screws and nuts. We didn't need to remove the top portion of the hexapod to mount the Raspberry Pi within. Some of the servo wires required an extension to reach the servo header on the hat. These extensions were connected in such a way that the black (ground) wires from the servo and the extension were together.



The servo hats are powered independently of the Raspberry Pi and operate at six volts. Each board is powered with 2.5 mm barrel connectors. The outside of the barrel connection is ground and the inside is power. Two of these connectors were soldered in parallel to a Tamiya connector commonly used in Remote Controlled Aircraft. An inline mini ATC fuse holder was spliced in between the barrel connectors and the Tamiya connector on the power side. This fuse was added to prevent the Adafruit Servo Hats from burning out if there was a short circuit. There's currently a 10 amp fuse mounted, but the inline fuse holder is rated for 25 amps. Both a NiCad battery and a wired power supply were used to power the hats throughout the project. If using a power supply, try to set the current to at least 8 amps. The voltage output of power supplies not capable of supplying this much current may drop to a point where it interferes with the logic on the servo hats; this can lead to erratic behavior of the servos. To charge the NiCad batteries, simply plug them into the charger and plug the charger into a power outlet. You can choose the charging current using a switch on the charger.





After running a real-time, openCV face detection script, the Raspberry Pi began to thermal throttle and reduce the efficacy of the script. To counteract this, we added heat sinks to the processor, RAM, and LAN controller on the Raspberry Pi. Furthermore, we placed a 5 volt blower fan inside the hexapod directed across the top of the Raspberry Pi. This fan is powered by a servo hat's 6 volt rail, but is not PWM-controlled; if the

main power switch is on, the fan will turn on. Since adding the heat sinks and fan, we have yet to encounter thermal throttling of any sort.

The torso was attached to the top of the hexapod using an adapter designed with CAD. Like the Raspberry Pi adapter, we reused existing holes in the body of the robot instead of drilling our own. The Raspberry Pi camera (used for object detection) was mounting to the chest of the torso using twine.

Challenges

1. The servos on the hexapod were not suitable for the added weight of the torso.
    a. The tip servo on leg 5 operates poorly now and impedes the walking. Before the torso was added, this servo operated fine.
2. The Phillips head screws used on the hexapod are made of very soft metal and strip easily.
3. The nuts used throughout the hexapod are smaller than any standard sized wrench and placed in difficult to access places.

Possible Improvements

1. Stronger servos - for project 2, we intend to look for replacement servos that provide more torque while keeping the same physical size
2. Better tools
3. Better cable management

Movement

Introduction

The movement in this project was most certainly one of the more challenging aspects due to the complexity of the robot (25 individual servos) coupled with the weight bearing nature of the legs. This meant that the movement for the legs had to be precisely calibrated and controlled or the robot would either fall over or injure itself. Couple these issues with servos that are barely capable of holding up the weight of the robot, and movement becomes non-trivial. The below sections outline how movement was designed and accomplished.

Goals

The primary goals for movement were four-fold: the code needed to be expandable, reusable, allow for re-configuration, and be safe. In this context, expandable means that new leg positions, gaits, or movements could be added without complicating the code further. Reusable meant that if a position was used during one walk cycle, it should be reusable in another without having to copy and paste code--eliminating any need for maintaining multiple copies of the same lines of code. Re-configurable means that each servo needs to be easily re-calibrated and the code should be (relatively) easy to adapt if the leg design or body design changes. Safety has two meanings here. One, the code needs to have basic safety checks on servos so that they are not able to try and push past their physical limit. Two, there should be some mechanisms in place which prevent operation which would drop the robot or damage it. For example, if the robot is standing with its front legs forward and its rear legs back, it should not be allowed to move straight into a normal standing position since this would mean the servos are dragging the legs on the ground and over stressing themselves.

Design

Designing to above criteria proved to be somewhat challenging and took a few iterations. In the end, an object oriented, table-driven approach was utilized. The reason for this is mostly to cut down on the amount of “working code” that exists while keeping each piece of the code modular. The specific implementation is described below in the implementation section but it seemed prudent to outline some of the reasoning behind using this implementation in order to warn future users against thoughtless re-design.

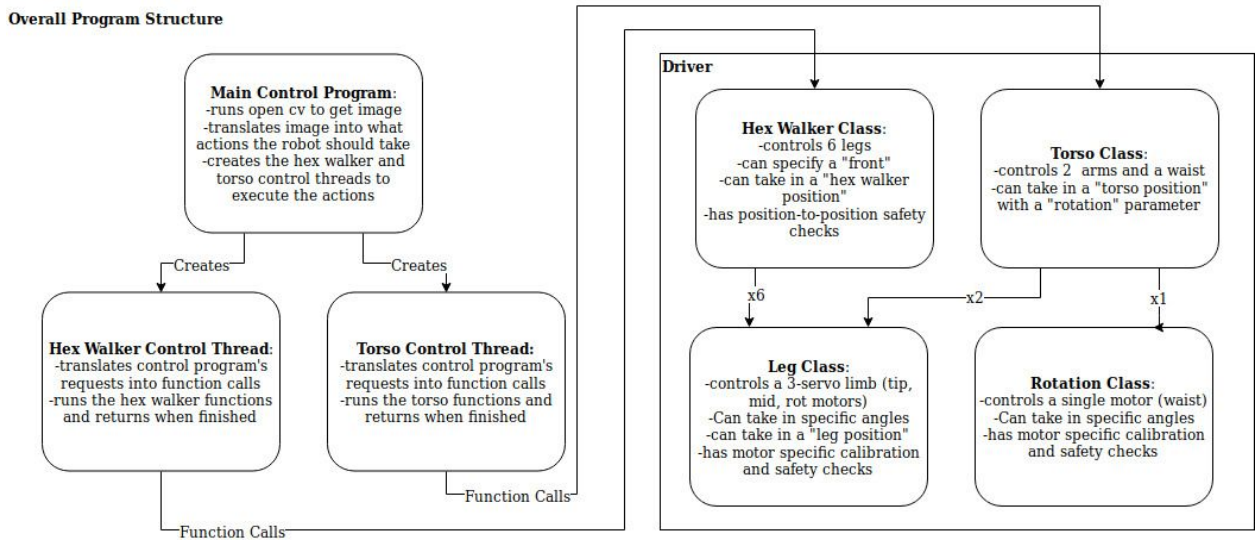
Using an object oriented approach was a relatively obvious decision due to the nature of the project. Each leg is identical to each other leg but needs to be individually calibrated and will have its own values for safe operation. Creating a leg object meant that these parameters could simply be set individually and re-use the same code rather than needing separate code paths for each leg. Furthermore, it enabled an easier use and creation of tables which could contain “leg position” objects and it becomes easy to do leg1.set_position(LEG_TABLE[“neutral”]) for all

six legs. Procedurally, this would become a nightmare as each servo would need to be individually set for each leg. Once an object oriented approach was decided upon however, the exact method of creating movement was still unclear.

Originally, the plan was to have action functions which could write out to individual leg objects what to do in order to create motions. However, this ran into a number of difficulties. It meant that each movement had to be programmed in to a function which eliminates any chance for reuse of these positions since they are tangled in with running code. It also meant that singular positions could not be tested and only the entire action could be tested. Furthermore, it meant that code maintenance would be much more difficult since running code would need to be modified rather than just static tables. Using static tables for positions, however, solves all of these problems at the only cost of readability. Trading readability for reusability, maintainability, a smaller code base, and less bug prone code though is certainly a win.

Implementation

The specific implementation is fairly straightforward though it may be confusing to some. There are 4 classes total: a leg class, a hex walker class, a torso class, and a rotation class as shown in right side of the diagram below.



The leg class is somewhat poorly named and should be refactored to something more accurate. Essentially, the leg class is used to control any 3-servo limb. They each get set with their own calibrated values for each servo upon initialization as well as the pwm addresses and channels which the object controls. They can be moved by either calling a function to directly change one of the tip, mid, or rot(ational) servos to a specific angle or by passing in a leg position which specifies an angle for each servo. It needs to be noted and understood that ***LEG OBJECTS HAVE NO UNDERSTANDING OF ANYTHING OUTSIDE OF THEMSELVES.*** This means that leg objects and leg positions should NEVER have values which reference “forward” or “backward” since these have no meaning to a leg. A leg understands only itself and nothing about the world around it. Legs can have left, right, up, down, in, or out since all of these directions are identical for all other leg objects. Likewise, an arm should have in, out, up, and

down since these movements will be identical for all arms. As an example, let us specify the position “forward” for a leg. For a leg on the left side of the robot, the rotational motor of the leg would need to be at a value >90, say 120. While a leg on the right side of the robot would need a value of <90, say 60. However, this means that now we will need code checks before writing this position out since we need to see *where* the leg is before deciding if we want this position. However, if we instead call the position “left”, all legs would use the same value of 60 since it does not matter where the leg is located. “Left” of a leg is “left” of ALL legs. Note that “left” here does NOT mean to the left of the robot but simply left of the leg being discussed.

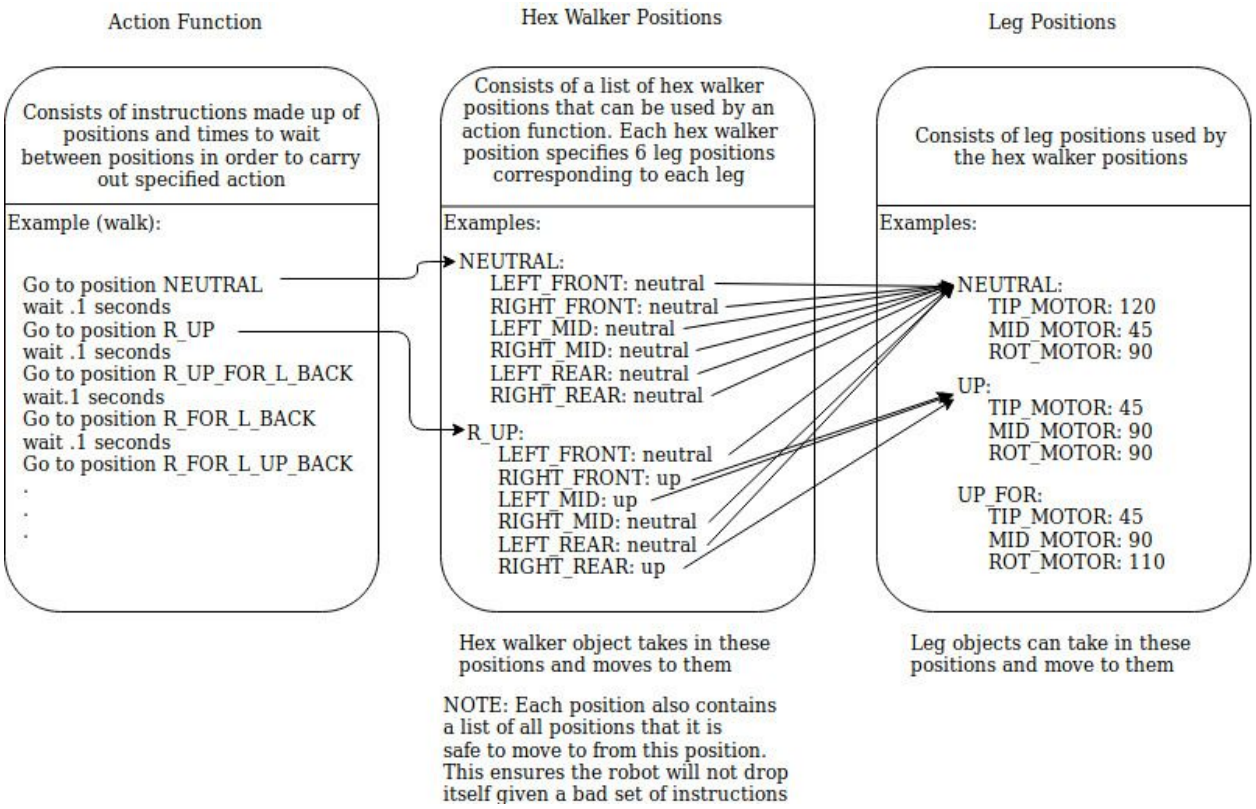
The hex walker class controls the base of the robot. It is made up of 6 leg objects and some extra meta-data such as what position it is currently in and where the front of the robot is currently defined as. It is able to be moved by sending it a hex walker position. It will then set each leg to the position specified by the hex walker position. It should also be noted that the hex walker is unable to walk backwards. Instead, it is only able to define its “front” as behind it and then walk forwards. This should be the preferred method of making rotationally-identical movements such as walking forward, bowing, or anything of this sort.

The rotation class is a very simple class that is used only to control the abdominal rotation of the robot. It is able to be set by sending it a specific angle to move to.

The torso class is similar to the hex walker class in that it instantiates other pieces in order to define itself. It holds two leg objects (repurposed to be arms) and one rotation object. It is able to be set to a new position by giving it a torso position in addition to a rotation amount. The reason for having the rotation amount separate from the torso position is that it is highly likely that a user would want to be able to wave while looking in different directions without wanting to create a whole new set of torso positions. Therefore, rotation is passed in as an extra parameter in order to cut down on unnecessary duplication of code.

Code Flow and Adding New Actions

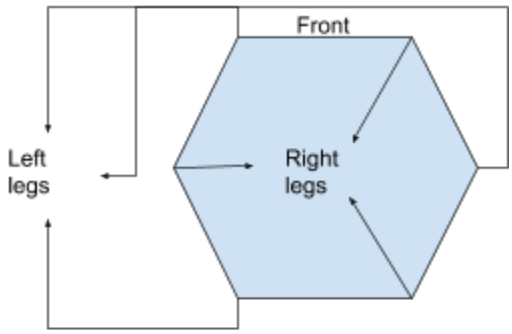
The diagram below shows the basic logical flow of how movement works. A more detailed description of the diagram will follow.



The code flow starts on the left with an action function. Here, we use a pseudo code “walk” example.

Programming a new action function: found in hex_walker_driver.py

The action function itself is quite simple. It is simply a list of positions through which the robot should move in order to complete an action. In this example, we are showing the beginning of a simplistica tripodal walk cycle. The diagram below shows which legs should be considered the “right” legs and which should be considered the “left” legs.



Following the action program then, the first thing that happens is all legs go to neutral. Then, the right legs lift up, then the right legs go forward (while in the air) and the left legs go back which causes the robot to go forward. Then the right legs come down, etc… It is simply the series of positions which a robot needs to go through in order to do the action. This then links into the hex walker position table because that is where each one of these positions is defined. So, to define a new action function, one would simply need to get a list of positions through which their action needs to move and write these out one after the other with small sleeps in between. Note that this is process is the same for creating actions for the hex walker as well as for the torso.

Programming new positions: found in hex_walker_data.py or torso_data.py

Programming a new position is also quite straightforward. For this example, we will focus on the hex walker since the torso is simpler than the hex walker. A hex walker position consists of three things: 6 leg positions (one corresponding to each leg), a list of safe positions to move to from this position, and a description. The description is the simplest and is just a string consisting of a written explanation of the position for debugging purposes. The list of safe positions to move into is just that. A list of integers (each one of which maps to a position) that indicate acceptable positions to move to from this position. The 6 leg positions are then the positions that each leg needs to be moved to in order to actually make the position. R_UP position from the example, it shows each leg assigned to a specific position. Each “right” leg is in the “up” position while the “left” legs remain in the “neutral” position. Note that in the actual python code, the order of these positions DOES matter since each corresponds to a specific leg (right front, right mid, right rear, left rear, left mid, or left front). These individual leg positions will then be written out to the correct legs when the hex walker is set to that position. These leg positions will then be defined in leg position tables. This keeps the leg positions separate from the hex walker positions

Programming new leg positions: found in leg_data.py

In leg_data.py there are numerous tables. Each one of these tables will describe every leg position required for a specific action to take place. For walking, each leg needs to have an out/in and neutral since each leg needs to be able to push the robot forward when it is required. As for the leg positions themselves, they are very simple and only hold 3 values corresponding to the angles each servo needs to move to. In order to program these positions, first all needed

positions should be written out. Then, using the leg position creator tool, the exact values needed to create that leg position should be recorded. Next, create a new table which will hold all the leg positions and enter this data in. Finally, make sure that the leg position names are descriptive and that some leg position is safe to get into from a neutral stance.

In practice, it is most likely easier to program from the bottom up by starting with creating the leg positions, then making the hex walker positions, and finally stringing these actions together into a sequence. However, this is entirely up to the programmer.

Challenges

The challenges for movement can be broadly broken into two categories: physical challenges and coding challenges.

Physical Limitations

-The robot is not perfect and much of the movement was not as exact as one would hope

-measuring exact angles on the robot is difficult and often led to more guessing than there should have been

-LOTS of servos meant that there is a lot of coordination and far too much movement to just do everything procedurally

-Weak servos meant that what “should” happen did not necessarily mean that it “will” happen since the robot would sag during movement or be unable to support itself

-Leg movements are functional, not expressive which means that the amount of tolerable error during movement is MUCH smaller than with the torso

Coding Related

-Naming positions for a hexapod is difficult since all positions of every leg need to be included in the title some how

-Avoiding code length explosion was problematic since it rules out many “simpler” coding options

-Two “separate” robots (hex walker and torso) meant that the solution for movement had to be general enough to work for two completely unrelated robots

-Safety checking is not easy and coming up with a method to safety check through positional movement was difficult

Possible Improvements

The main improvements that could be done would include:

-refactoring the code to replace the “leg” object with a “limb” object

-adding a turn function which takes in a number of degrees to turn

-adding a walk function which takes in a number of inches to move forward

-add more gaits to the robot

-add a way for the robot to get back to neutral from any position
-come up with a more formulaic, readable naming scheme for hexapod positions

Vision

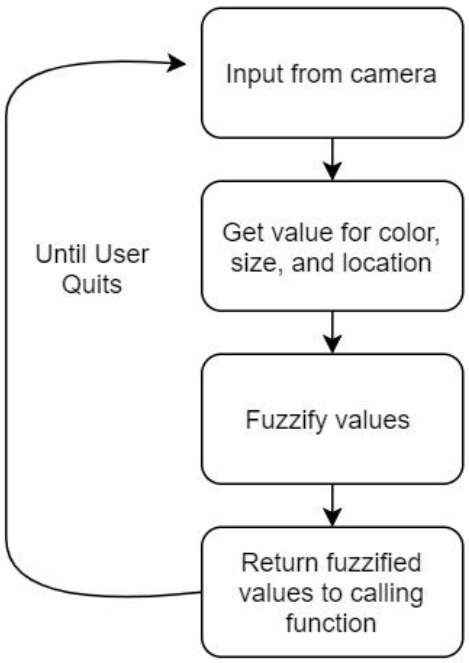
Introduction

The vision module allows the hexapod to take information from its environment and process it. This is written in python3 and uses numpy, opencv3, and imutils. Because the robot does not have precise movements, the program returns fuzzified values, instead of exact values for color location, and size. Users are able to choose which colors should be detected, and the numerical definitions for different sizes and different locations, buy defining parameters at the beginning of the object_detection.py file.

Goals

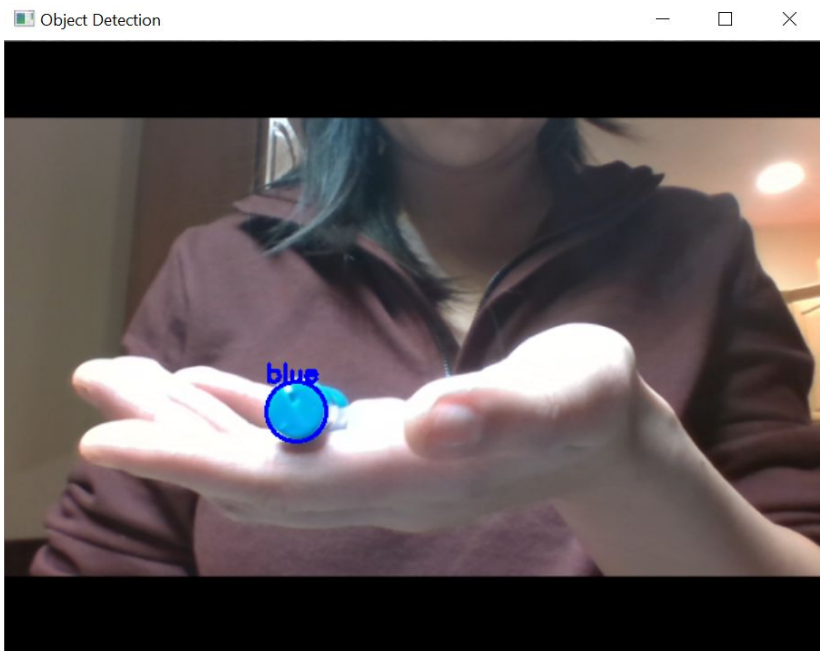
1. Detect objects
2. Detect colors of objects
3. Fuzzify the object size and location
4. Write program so that it is easy to change what colors are detected
5. Write the program so it is easy to change the parameters for a large object
6. Write the program so it easy to change the parameters for where the object is located
7. Add face detection - in progress

Design



Implementation

Before using this program, users should ensure that they have python3, opencv v3, imutils, picamera-array, and numpy installed on their machines. Users who use the module as a standalone program have the option to use it on a laptop webcam or on the pi camera. There is a function called raspi_camera to get input from the raspberry pi camera and a function called laptop_camera for implementing a laptop camera. Simply choose the appropriate function and call it at the end of the file. Users also have the option to view the feed and objects that being detected. Just uncomment the imshow() functions before executing. This feature is usually disabled on the pi because it takes processing power and it is not that useful when it is running in the program that is used to integrate all of the robot capabilities. The program also automatically prints to the command line when it is running, so users can monitor what is being picked up by the feed. It prints “none” for all fields when it processes a frame that does not have an object in it and prints the color, size, and location of an object, when an object is detected.



When the vision module is being used for the robot, it uses the picamera-array library to access the camera and get the RGB array for each pixel. The module takes a frame from the camera stream, translates the RGB array into the HSV color space, and gets a color array, size, and location. The size is given as a pixel area, but is then fuzzified, based off of some parameters, and will be either small, medium, or large. The location goes through a similar process and will be either left, middle, or right, depending on where the object is in the frame. If no objects were detected in the frame, it returns none.

Challenges

- The pi has a weaker processor than a laptop, so algorithms that ran easily on a laptop were slow and inconsistent on the pi.
    - Solution: Use less taxing algorithms. This did compromise some capability, but the current algorithm works well enough for our current use.

- The vision program also intermittently picks up reflections off of objects and processes them as objects, so the vision program might see a small yellow object, when really it is just a reflection off of a large blue object.
    - So far, the best solution is to use this program in a room that doesn't have intense overhead lighting, but ultimately the goal is to refine the object detection so that the reflections aren't picked up as frequently
- Sometimes objects were picked up in the background that caused unexpected returns
    - Solution: Make sure that there is a neutral background or define the allowed object size, so that background items are not seen as valid objects

Possible Improvements

- Add face detection
- Refine the object detection so that the reflections aren't picked up as frequently

Speech

Introduction

The text to speech aspect of the hexapod was relatively straight forward. There are various ways to perform text to speech, and for this project the Espeak and num2words library were used. The main theme of the text to speech was to control the speed and tone of speech, along with being able to modify the speech, and also have a small but powerful speaker.

Goals

1. Have the robot speaking with a default voice
2. Be able to take in text input from the user and output speech from the board
3. Have control over the speed of the robot's speech
4. Have speakers that are small, light, but still powerful
5. Be able to modify the type of the voice

Design

For the text to speech the requirements are a Raspberry Pi 3 Model B, a raspian SD card, speakers, and a power supply. For the software aspect, two libraries known as espeak and num2words need to be installed in order to write the text to speech code.

Implementation

In order to implement text to speech, a text to speech engine is required. The one used for this project is the Espeak engine.

To download and install espeak type:
sudo apt-get install espeak

Once Espeak is installed, you can test for speech by typing this command in the terminal:
Espeak "Enter text here" 2>/dev/nullk

You will also need to install the num2words library in order to be able to take in numerical inputs to convert them to speech.
To download and install num2words type:
sudo pip3 install num2words
Once those libraries are installed python code should now function.

Challenges

1. One of the challenges for the text to speech was determining the correct method to type in the code from the python script to the shell. The way Espeak works with python is that it writes code to the terminal in the script, and then takes in user input. One of the difficulties was syntactically interfacing the python code with the shell. This was merely a syntactical error and was debugged after more research on the internet.

2. The speakers obtained for this raspberry pi were very weak. They were light, but not very powerful, and as a result the sound quality was poor. In order to fix this issue, other more powerful but also small speakers need to be obtained.

Possible Improvements

1. Obtain more powerful speakers so that audio can be heard more easily. At the same time, the speakers should be of minimal weight, so they don't overburden the servos of the hexapod.
2. Another potential improvement would be adding emotions to the text to speech in some manner. This would require additional research.

Integration

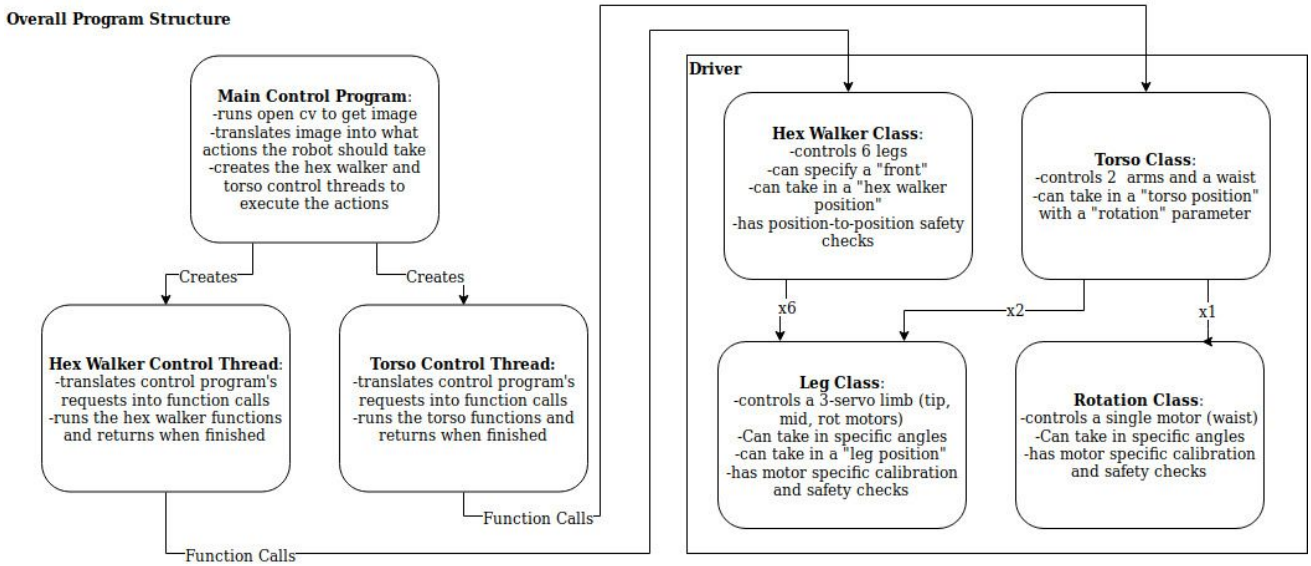
Introduction

After we had the infrastructure for getting the hexapod walking and object recognition using OpenCV, the next step was to integrate these so that the robot could behave depending on what it sees. The key principle to integration was that threading to allow the torso and legs to move at the same time, but also independently of each other.

Goals

1. Allow the robot to respond to visual feedback without stalling program
2. Have independent control over torso, legs, and vision processing

Design



Implementation

In addition to a command line based menu to showcase all the different dancing and walk cycles we developed, we wrote a simple controller to orchestrate a vision-driven demo. This controller would take input from the Raspberry Pi camera, categorize what objects it sees, and return a colored, valid object to the calling routine. If there were no objects detected, the robot will do nothing.

After a frame has been processed, two threads are spawned; one for the torso, and one for the legs. If not object is detected, these threads join the calling routine right away without commanding the servos to change position. After the threads have spawned but before they join, image retrieval and processing is halted.

Currently, the controller is setup to recognize three colors: pink, blue, and yellow. Upon seeing a pink object, the hexapod will bounce up and down while the torso does the “King Kong” dance. A yellow object causes the hexapod to wave its legs while the torso waves. A small pink object will cause the robot to step closer, while a large one will make the robot step away.

Challenges

Implementing the threading was new for all group members, but the final controller work well.

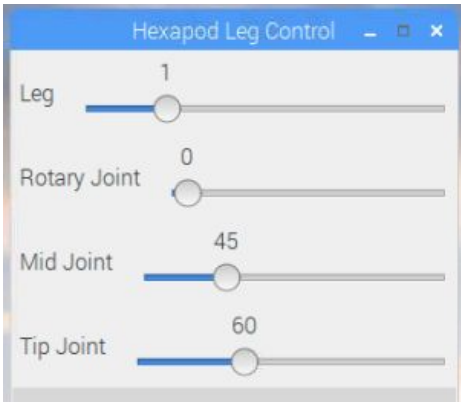
Possible Improvements

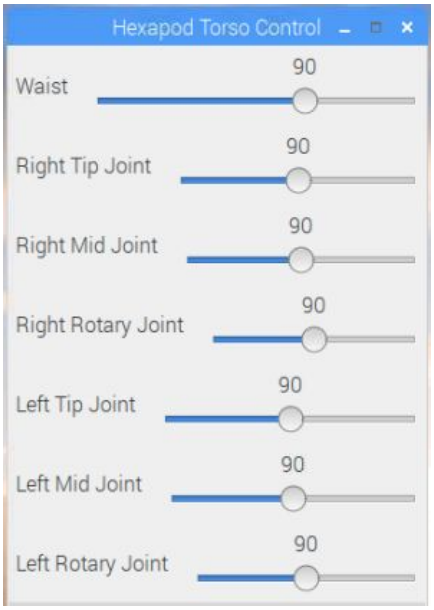
1. Use ROS (required) instead of using Python threads

Additional Features

User Interface for Servo Testing

Using openCV, two scripts were written to make gathering leg positions easier and faster: one for the torso, one for the legs. These scripts used openCV's track bars so that a user can set the desired angle for each servo. The scripts reused the leg object meaning the user was not able to damage the servos by overextending them.





Appendix

Project Log

Ammar

Week 1: Installed OpenCV and python on ubuntu.
Week 2: Researched ROSS and realized we weren't going to use it for Project 1.
Week 3: Helped Charles record leg positions.
Week 4: Initially started writing an arm class, but then realized it was redundant as the arm class was merely going to work the same way the leg class worked. Started measuring the angles for the arm servos.
Week 5: Helped with the torso code. Set up the second raspberry pi to test the arm and torso servos positions and capabilities. Tested the shoulder servos for the torso. A problem that I found was that one of the servos didn't work and had to unscrew and replace the servo with Melih's help.
Week 6: Bought another Raspberry Pi and began working on text to speech. Finished working on text to speech with the third raspberry pi. Faced a few issues with being able to read numbers but not text. Upon further debugging and making a few minor modifications text to speech began working.

Charles

Week 0: Ordered Raspberry Pi Servo/PWM Hats
Week 1: Test legs and individual servos
Week 2: Test legs together. Problem: need a way to bundle leg movements together because each leg acting individually isn't going to work. Left legs have different requirements than rights legs, and motion require legs to move in tandem.
Week 3: Hand-calibrated each servo individually. Discovered that the servos were not strong enough to support the robot in our original walking height. Created new walking position to compensate for how shitty the servos are.
Week 4: Got hexapod roughly walking. Problem: the robot is too heavy for the servos. Movement is impaired due to weight.
Week 5: Revised leg positions and made the torso work. Wrote the script that will be run during the demonstration for part 1

Emma

OpenCV development throughout (need to figure out dates and capabilities)

10 October 2018: Soldered one of the servo hats

16 October 2018: Soldered more servo hats for the robot
17 October 2018: Soldered a servo hat for dev testing of the arms

19 October 2018: Interfaced with pi wirelessly, went through potential robot positions, troubleshooted pi camera issues, assembled pi, servo hats and robots

29 October 2018: Worked on object tracking with opencv and discussed structure for the main robot controller

31 October 2018: Worked on opencv and combining the torso with the legs

01 November 2018: Opencv detects objects by color and creates aliases for returned data. Needs to be tested on the robot to determine size and location in reference to the robot

03 November 2018: Tested vision code on the robot and changed necessary parameters, so that they were adjusted for the robot. Also integrated vision code so that the robot was responding to visual feedback. Finished my portion of the PowerPoint

05 November 2018: Last minute adjustments to the parameters for the demo, so that it was configured correctly for the demo .Demoed the robot and presented powerpoint.

06 November 2018: Created the outline and formatting for the report. Worked on writing the vision and integration sections.

Patrick

Week 1: Play with openCV for reading and displaying images.
Week 2: Soldered angled servo headers to a PWM Raspberry Pi hat.
Week 3: Fabricated a power harness for the Servo hat with an inline fuse. Two servo hats were previously destroyed and the fuse was added in an effort to prolong the life of our components.
Week 4: Helped fine tune the walking positions. Tightened all of the servos and attempted to clean up the wires. Fabricated an adapter to attach the Raspberry Pi within the robot where the old microcontroller once sat. Attached fan to body to keep the Raspberry Pi cool.
Week 5: Created user interface to control each leg individually. Used openCV for UI. Mounted torso on hexapod. Wired each servo to both Raspberry Pi Hats. Tightened all nuts and screws on hexapod to prevent the legs from falling off mid walk

Week 6: Prepared robot for demo. Created a user interface to control the torso. Added bobble head to torso.
Week 7: Continued preparing for demo. Working on presentation and rep
ort.

Photos: Everyone add any photos you've taken with a date and description

Date: 19 October 2018
Description: Assembling pi, hat, and base (probably don't need this but it's here if he wants it "log style"

Useful Links

Code

Link to Git Repository: https://github.com/pgmerek/ECE478_Project1

Videos

Single Leg Testing: https://youtu.be/N0eHbJYXwFQ
Single Leg Sweep Testing: https://youtu.be/VY68gqJXkeo
Three Legs Moving Together: https://youtu.be/jQpbgYSgK04
First Time all the Legs Move Together: https://youtu.be/DE-JMa9TeWQ
Early "Crawling": https://youtu.be/HXO4nm1dM3k
Tall Walking Test: https://youtu.be/jcKxYJbV9DI
Early Walking Test: https://youtu.be/zUr2h63fz3M
Torso Imitating Johnny Bravo: https://youtu.be/QEe9Pc3HD80
Torso Waving: https://youtu.be/JneKdxrjVBo
Torso Hugging Another Bobblehead: https://youtu.be/N9r3G8sU7TM
Torso Moving and Hexapod Rising Up: https://youtu.be/IMFu28ybqwE
Demo Program Running showing Action based off Vision: https://youtu.be/fufKx_GRvGg
Hexapod Turning Itself Off: https://youtu.be/zuXpRv3HBoo
Walking with Torso: https://youtu.be/N9Qlhjh_op4
Torso Shaking Charles's Hand: https://youtu.be/LX-VvIhiCs0