ECE 579 - Intelligent Robotics II

Winter 2020

InMoov Torso with Pioneer Base

MESHAL ALBAIZ
TRISTAN CUNDERLA
BRIAN HENSON
DANIELLE NICHOLAS
ARON SCHWARTZ

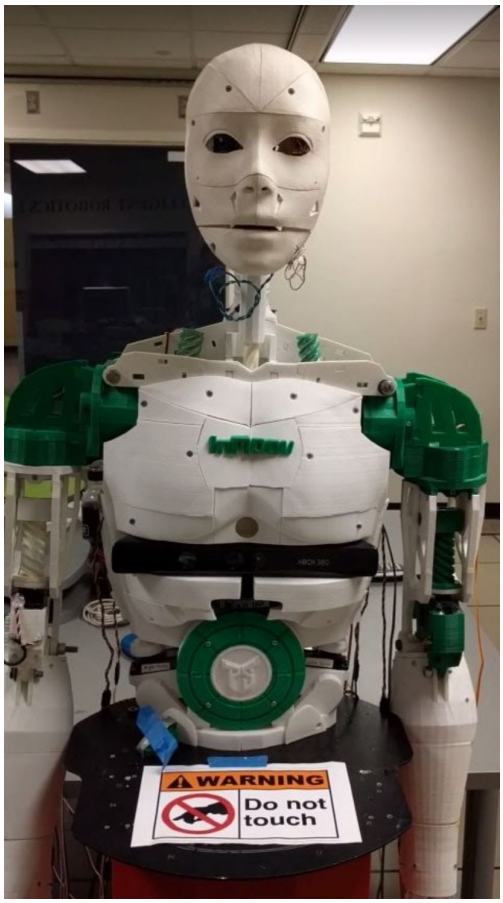

Table of Contents

Project Description

The project for this term involved working with the InMoov robot torso and the Pioneer base. InMoov is a 3D-printed robot that includes a full torso with a head, neck, and arms. It has 25 servos in total. The Pioneer base is a 3-wheeled metal chassis that has 2 DC motors and 3 12V batteries, and can be controlled through a serial port. Some ideas about what could be done with the robots were discussed early on during the quarter, but as the team discovered the status of the robot, it was determined that getting full mechanical functionality back was going to be the main task at hand.

The robots were found with InMoov torso clamped to the Pioneer base. The InMoov torso had a Raspberry Pi but no SD card or servo HATs. There was some legacy Python code, including drivers, created by the group that last worked on the InMoov robot. The servos were all disconnected since there were no HATs, and mostly unlabeled. The Pioneer base was missing a NUC, needed battery replacements, and body adjustments.

The goals of this project are mainly to prepare the robots for future use by upcoming groups. Here is a list of some of the tasks the group set out to complete:

- The InMoov robot needed a stable operating system on a dedicated SD card, as it needs to have the most recent version of the developed source code on it.
- The servo cables needed to be labeled.
- PWM servo hats are required, in addition to stackable headers, right-angle 3x4 headers and servo extensions.
- A power distribution solution is required to power all the PWM servo hats.
- The Pioneer base requires two of its three batteries to be replaced, and a NUC with ROS installed and operational.
- A second Raspberry Pi could be required if vision is going to be implemented. This is mainly due to how busy the main Raspberry Pi is with the PWM servo hats and how short the camera ribbon is, but processing power or bandwidth bottlenecks are also a concern.
- Create natural gestures and a GUI-based posing program with the ability to save poses for future use in animation.

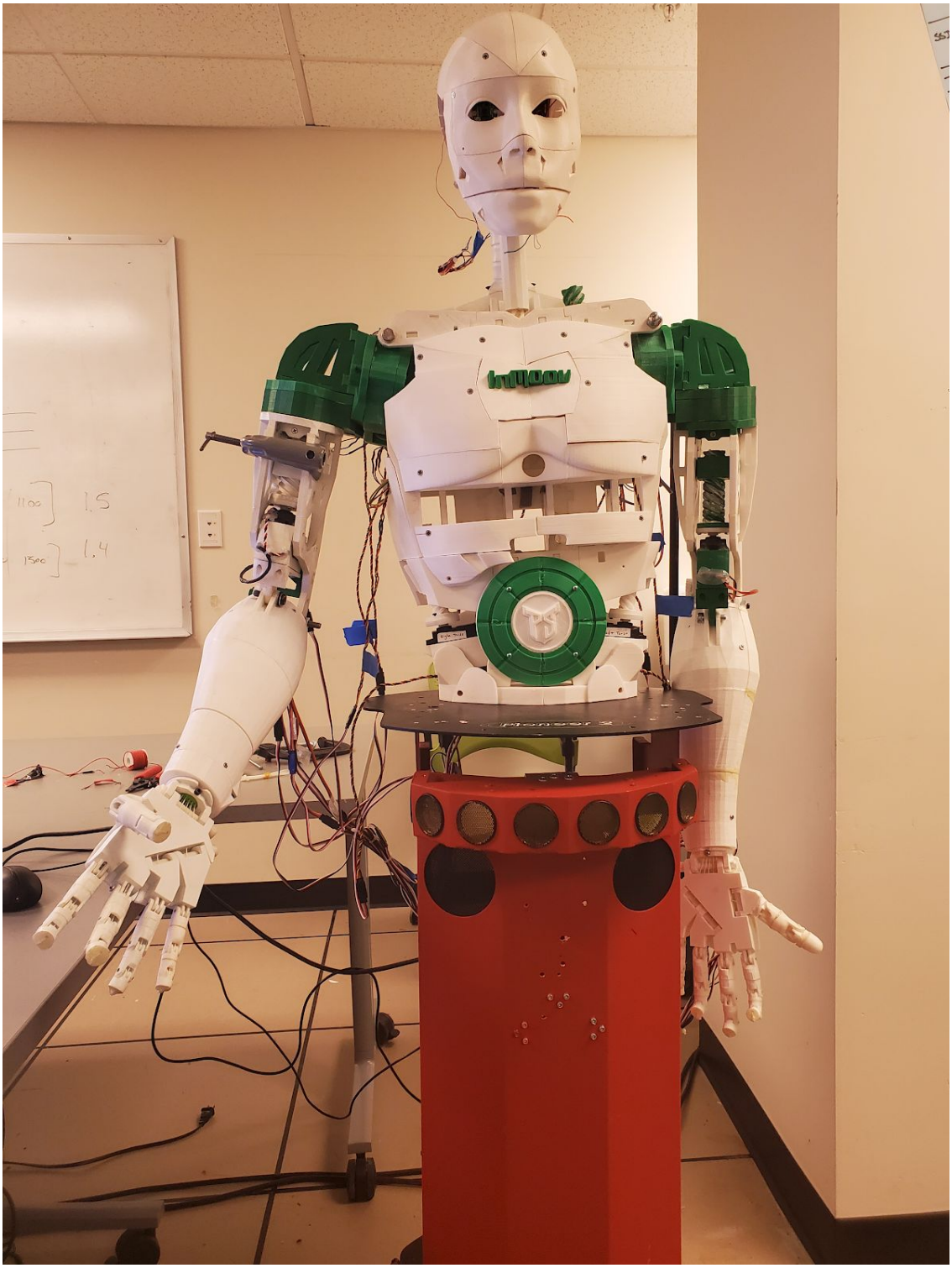

Figure 1: Integrated torso and Pioneer base

InMoov Hardware

To evenly distribute power across the body, the group purchased 3 Raspberry Pi stackable servo hats, with 16 channels each. Stacking hats and right-angle headers were then soldered onto the hats. The team also purchased 20 1m-long servo extension cables. The servos were tested individually using a power supply, with the rough current draw estimates being used to decide which servos were connected to which servo hat, to avoid overloading one hat with too much current.

Each servo hat uses solder jumpers to set a different address: hats in position bottom=0/ middle=1/top=2 are labeled with A, C, B and their corresponding addresses are 0x4A, 0x4C, and 0x4B. The software drivers use a JSON file *inmoov_servo.json* to define the names, IDs, and channel numbers of each servo. This helps in maintaining structure and re-usability. The ID numbers for the servos go from 0 to 47: the hat number is (ID / 16) rounded down, and the channel number is (ID % 16). A chart of each of the servo connections to each of the HATs can be seen on the next page.

All servos are now labeled and none are burnt out. One issue that still needs to be addressed is the loose servos for both wrists, one of the wrists barely moves and the other can be heard moving but doesn't move the wrist. The team needed to improve the connection between the elbow servos and the attached long worm-gear to support the weight of the arm; a screw was inserted down the barrel of the servo, through the white plastic adapter, to prevent the worm gear from simply popping off. It is worth explicitly noting that many of the servos (like the elbows) are "free-spinning", meaning they have been modified to use external feedback potentiometers and remove the limits on their rotation. They will keep turning until their feedback pot reaches the target value.

InMoov's head has a Pi camera looking through its right eye, mounted in place with hot glue. The camera is perfectly functional. The group wrote code that uses ROS to capture and transmit a stream of images from this camera, but the project ended up not being able to use this visual input. It could be used for vision-assisted gestures, interaction, or recognizing people in future projects.

One of the gestures the team intended on implementing is grasping and improving the grasping abilities of the robot. The 3D-printed material is pretty slippery and doesn't have much grip, the fingers and hands could be improved by adding a layer of a more grippy material and possibly rubber finger tips.

Servo information:

- **HS-805BB Hitec (12):** Elbow, arm rotate, arm front, arm lift out, torso tilt, head, neck
- **S3003 Futaba (3):** Jaw, wrist
- **HK15298B HKing (10):** Fingers

Servo HAT Connections

Servo Name	Hat address	Hat channel #	JSON ID #	
left_shoulder_lift_out	A	0	0	
right_shoulder_lift_front	A	1	1	
jaw	A	2	2	
head_x	A	3	3	A is bottom HAT
head_y	A	4	4	
right_thumb	A	5	5	
left_thumb	A	6	6	
left_shoulder_lift_front	C	0	16	
left_arm_rotate	C	1	17	
left_elbow	C	2	18	
left_index	C	3	19	
left_mid	C	4	20	
left_ring	C	5	21	C is middle HAT
left_pinky	C	6	22	
left_wrist	C	7	23	
left_torso	C	8	24	
right_shoulder_lift_out	B	0	32	
right_arm_rotate	B	1	33	
right_elbow	B	2	34	
right_index	B	3	35	
right_mid	B	4	36	B is top HAT
right_ring	B	5	37	
right_pinky	B	6	38	
right_wrist	B	7	39	
right_torso	B	8	40	

Table 1: Servo connection distribution on HATs, colors are estimates of current draw
Red: High current draw Yellow: Medium current draw Green: Low current draw

	Open	Close
Right Thumb	X	✓
Right Index	✓	✓
Right Middle	✓	✓
Right Ring	✓	X
Right Pinky	✓	X
Left Thumb	✓	✓
Left Index	X	X
Left Middle	X	✓
Left Ring	X	✓
Left Pinky	X	✓

Table 2: Functionality of finger movement

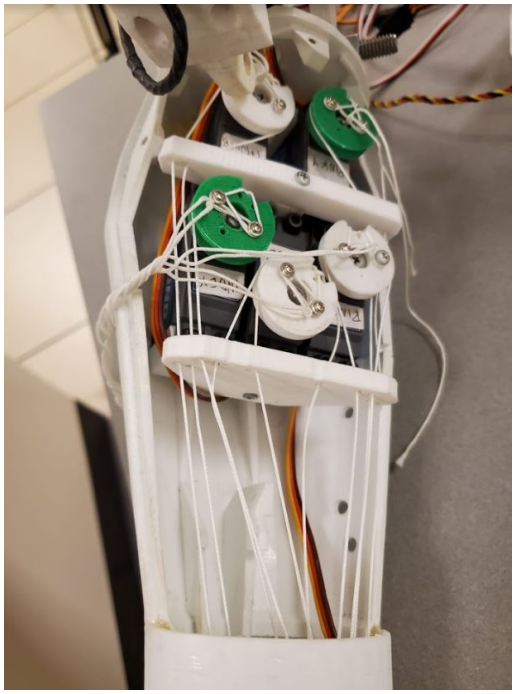

Figure 2: Fingers are actuated via string pulleys attached to servos

Figure 1 shows how the fingers are actuated. Each finger has two strings that correspond to either opening or closing the finger. “Open” refers to the extension movement of the fingers, while “Close” refers to the grasping movement of the fingers. If a string is not taut when the finger is fully extended (“Open”), or when the finger is grasping (“Close”), and this is indicated in the above table with an “X”. There is a similar 3D-printed InMoov arm somewhere in the lab that has much red plastic in it; its finger-strings contain springs on the “open” strand to aid in opening the fingers. Much could be learned from examining its design and combining the best elements of both.

InMoov Software

Drivers

In order to run within ROS Kinetic, all of the Python code in the system has been designed to run on both Python2.7 and Python3.5. Certain features like function argument type annotation are not supported in Python2.7, so they needed to be removed from the project.

The primary module in the system is *Inmoov.py* which defines the *Inmoov* class. When instantiated, it reads the *inmoov_servo.json* file and builds a servo object for each entry there. This happens whether the servo hats are attached or not, and serves as a way for the group's GUIs to get the parameters of the servos even when they aren't controlling them. This also builds a hierarchy of objects from these servo objects, where the finger servo is part of a Finger object, which is in turn part of a Hand object, which is part of an Arm object, etc. This hierarchical system is defined in *Structures_new.py* but is mostly unused. The group's code more frequently uses the *find_servo_by_name()* function to access specific servos. At the end of the class initialization, all the servos are explicitly de-powered for safety reasons. This class also defines the function *inmoov_ros_command()*, which is the entry point for controlling the Inmoov bot over ROS using several specially-formatted command strings.

The file *Servo.py* defines a *Servo* class which represents one servo in the bot. This holds config values such as the angle-space limits, the PWM-space values that correspond to those endpoints, the PWM hat # and channel #, the default angle when *initialize()* is called, and whether the servo is disabled or not (if a servo is disabled in the JSON, then any attempt to move that servo will immediately return and have no physical effect). These objects are instantiated when the Inmoov class parses the config JSON. The function *rotate()* will set the servo to a specific angle, and *rotate_thread()* will set the servo to move to a specific angle over a specific duration. Angles are floats. This is accomplished via a background daemon thread defined in *Frame_Thread.py* that exists for each Servo object, and interpolation logic that will gradually move the servo setpoint between its current position and its destination. The group's project did not make use of this gradual motion, because unlike the Hexapod project where the idea was originally conceived, the Inmoov servos are all quite sluggish and don't need to be artificially speed-limited. However the code is written and works perfectly well.

The file *Pwm_Interface.py* serves as the interface between the code that the group wrote with the drivers that control the Adafruit hats. This is where the order (bottom-middle-top) of the servo hats is correlated to their addresses (0x4A-0x4C-0x4B). The code is designed so that if the imports fail (such as when running the GUI on a computer for testing reasons) it will still seem to run without sending any actual I2C messages. It also implements a locking mechanism to make the PWM control thread-safe.

The drivers for the InMoov bot include *Adafruit_I2C.py* and *Adafruit_PWM_Servo_Driver.py*, two files which presumably came from Adafruit and manage the low-level communication between the Pi and the servo hats. This group did not modify these files at all, except to change print statements and make them Python3 compatible.

GUI Pose Maker

Purpose

The pose-maker GUI lets a user control the servos on the bot one at a time, in a clear and intuitive manner. This also illustrates the maximum range of motion of any given servo, as well as the numeric values that correspond to given angles. It also allows for taking a snapshot of the robot's current servo values and saving that as a "pose" object, for use as part of an animation sequence.

Code Construction

The file for this GUI is *body_poser_GUI.py*. All the GUI elements are constructed with the Tkinter library that is built in with a Python installation; no extra libraries are needed. To give the illusion of having 3 non-interacting tabs of servos, it maintains several lists-of-lists with structure identical to the variable *names_all*. It tracks the last value the slider was set to, the on/off state of the servo, and the checked/uncheck state of its checkboxes. When changing tabs, it overwrites the values of the existing GUI elements to match the values in these lists-of-lists. These are also used to detect which slider or checkbox changes when one is changed; they all have the same callback, but only one has a current value that doesn't match the behind-the-scenes tracker. That is the one that changed.

GUI Components

The GUI has 3 groups of buttons along the top: tab switching, init/off, and save/load.

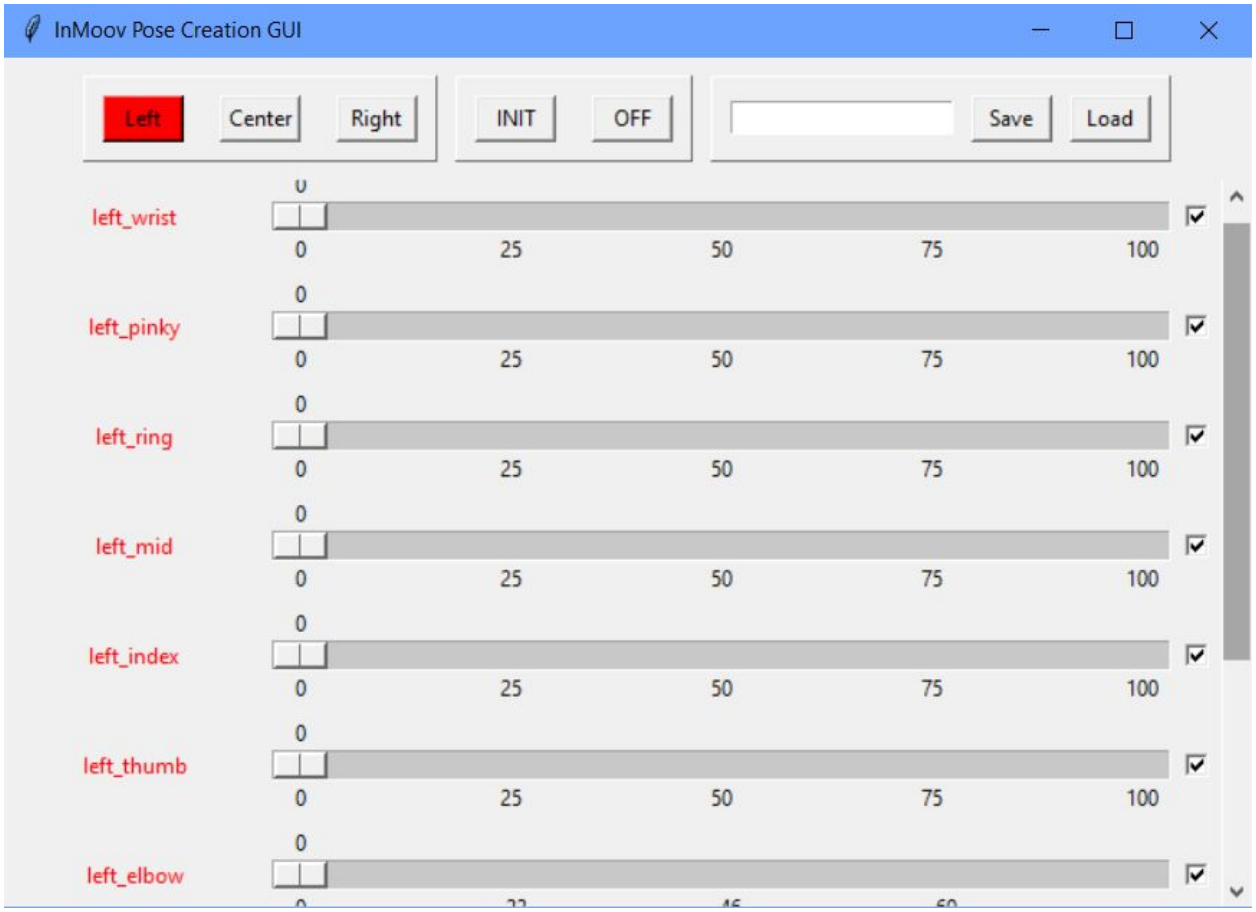

Figure 3: Body Poser GUI

The tab-switching buttons change which servos are displayed and controllable. This was implemented to reduce scrolling and visual clutter, and to make it easier to see symmetry when switching between the left and right tabs. The vertical scroll position won't change when changing tabs. The button for the currently active tab is red.

The "init" button tells the Inmoov object to execute its "*initialize()*" function, which sets each servo to its default values, but it does so in a specific order and with delays between sections.

The "off" button depowers all servos in the bot, causing them to go limp and malleable.

The "save" button will add a new pose to the file *pose.json* using the current values of the sliders in the GUI (including sliders in the hidden tabs), and the name of the new pose is what is entered in the text-entry box.

The "load" button reads the file *pose.json* and searches for a pose with the name in the text-entry box; when it finds it, it sets the sliders in the GUI to match, and commands the Inmoov bot to move to that pose.

The scrollable region in the rest of the window is the space controlled by the tab buttons. Each name is the name of a servo in the bot, and its slider shows the full angle-space range of motion for that servo. Moving the slider will cause the servo to correspondingly move in real time. If the slider for a servo is missing, then that servo is disabled in the configuration JSON file. The color of the name indicates the on/off state of that servo: red=off and black=on. If a servo is off, then since it is depowered and malleable, the position of its slider doesn't necessarily correspond to the position of its physical servo. Moving a slider will cause an off servo to turn on.

A pose does not need to include all the servos in the bot; it supports masking. The checkboxes indicate whether or not a servo will be included in a pose that is saved with this GUI. If a pose contains only some servos, then loading or executing that pose will only change the values/positions of those servos it contains. The "init" button will check the checkboxes of all servos. The "off" button will uncheck the checkboxes of all servos. When loading a pose, each checkbox is checked if it is included in the pose, or unchecked if it is not.

Animation Creator GUI

Purpose

The animation creator GUI provides users with the ability to create, edit and save animations that can be used on the InMoov humanoid robot. An animation is created from stringing multiple static poses together with time delays in between to give the sense of movement.

Code Construction

The code for the animation creator GUI can be found in the file path inmoov/gui/ and is entitled *animation_creator_GUI.py*. Just like the Body Poser GUI, this is built with Tkinter. Though the code is heavily commented there are a few main details that need to be touched on. The most important variable within the GUI is the class variable PoseDict. This dictionary holds the information of the current animation within the GUI workspace: each pose, their respective hold times, and the order of the poses within the sequence. The PoseDict uses integer values, starting at 1, as keys to indicate execution order and a list containing the pose name in index 0 and hold time in index 1 as the value.

GUI Components

The GUI has two different states, showing the loading functionality and hiding the load functionality, which can be seen in figures 4 and 5. How to operate the GUI will be explained in detail below.

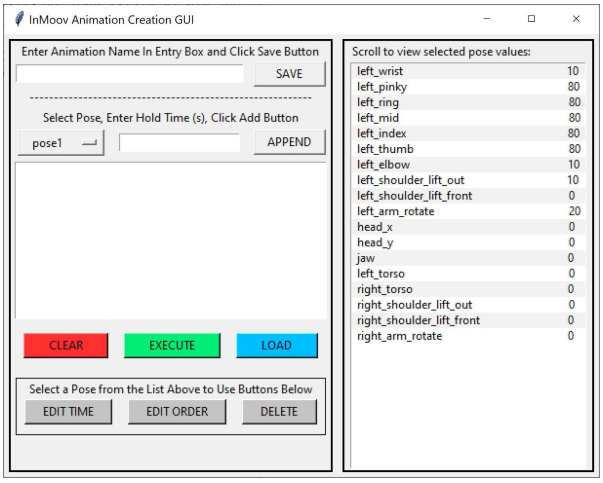

Figure 4: Animation Creator GUI not showing load functionality

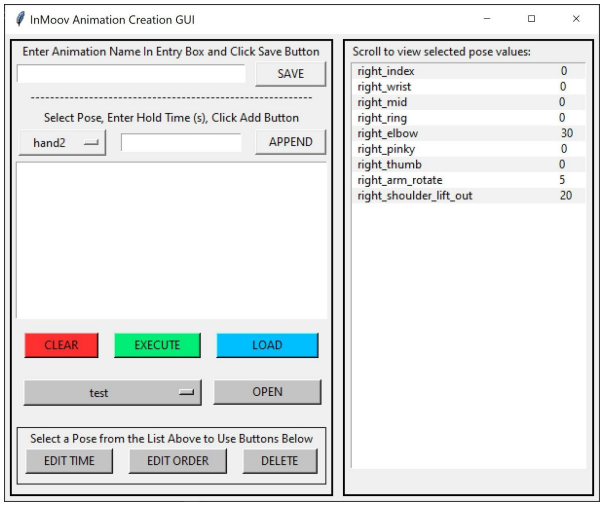

Figure 5: Animation creator GUI showing load functionality

The GUI will also produce popup windows to change animation parameters as well as indicate errors. These popup windows can be seen below.

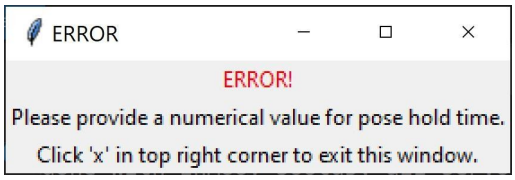

Figure 6: Popup window that is displayed when no animation save name is given

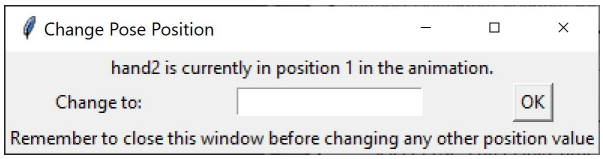

Figure 7: Popup window that is displayed when changing pose order

Capabilities

- Create animations using poses and hold times: provides users with ability to make animations from static poses with a delay in between the poses to simulate motion
- Display current pose: provides users with a visual (this view will change dynamically based on which pose is selected by the user)
- Load animations: allows users to load already saved animations in order to edit or execute them
- Change pose time in animations: allows users to change an already set hold time for a pose
- Change pose order within animation: allows users to rearrange poses dynamic instead of having to open the JSON file or restart the animation from the beginning
- Delete pose from animation: allows users to delete any unneeded poses from the current animation

- Save animations: once happy with an animation a user can save an animation to the animation.JSON file which can be loaded and edited later
- Execute animations: allows users to run animations in real time

Procedures

- How to Add a Pose to an Animation
  1. Use the provided option menu on the left side of the GUI to select a saved pose from the pose.JSON file, which was created using the body poser GUI
  2. Enter a time value in seconds into the entry box to the right of the option menu in step 1
  3. Click the ‘APPEND’ button to add the pose to the animation
  4. The pose will appear within the text box underneath the items list in steps 1-3 and will indicate the position in the animation the pose is, the pose and how long the pose will be held for
- How to Save an Animation
  1. After adding a pose or several poses to the animation via the ‘How to Add a Pose to an Animation’ procedure enter a name for the animation in the provided entry box located at the top of the GUI
  2. Click the ‘SAVE’ button to save the animation to the animation.JSON under the name provided in the entry box
- How to Load an Animation
  1. Click the blue ‘LOAD’ button to show the animation loading functionality of the GUI
  2. Select a saved animation using the option menu provided on the left side of the newly exposed area of the GUI
  3. Click the ‘OPEN’ button to load animation into GUI
  4. The animation can now be altered the same way as one would edit a newly created animation (change hold times, delete poses, reorder poses and saving an animation)
- How to Change a Pose Time
  1. Select a pose from the text box located on the left hand side of the GUI and oriented in the middle (below the section to append a pose and above the colored ‘CLEAR’, ‘EXECUTE’ and ‘LOAD’ buttons)
  2. Once pose is highlighted (typically in blue), click the ‘EDIT TIME’ button located towards the bottom of the screen
  3. A popup window will appear prompting for a new hold time to be entered
  4. After entering a valid time into the entry box click the ‘OK’ button to confirm the change
  5. The pop up window will disappear and the pose within the animation will be updated with the new hold time

  NOTE: if user wishes not to change hold time of selected pose make sure to exit the popup window before trying to change another hold time or using other GUI functionality
- How to Change the Pose Order
  1. Repeat step 1 of the ‘How to Change a Pose Time’ procedure outlined above
  2. Once pose is highlighted (typically in blue), click the ‘EDIT ORDER’ button located towards the bottom of the screen

3. A popup window will appear prompting for a new position for the selected pose to be entered (position 1 is at the top of the text box and will be the first pose to be executed)
4. After entering a valid position into the entry box click the ‘OK’ button to confirm the change
5. The pop up window will disappear and the pose within the animation will be updated with the new animation order

NOTE: if user wishes not to change the position of selected pose make sure to exit the popup window before trying to change another pose position or using other GUI functionality

- How to Delete a Pose
  1. Repeat step 1 of the ‘How to Change a Pose Time’ procedure outlined above
  2. Once pose is highlighted (typically in blue), click the ‘DELETE’ button located towards the bottom of the screen
  3. Animation will be updated within GUI with the selected pose now deleted
- How to Clear an animation
  1. Click the red ‘CLEAR’ button
  2. Animation properties within the GUI will be cleared and now the GUI is in the state as if it were just opened
- How to Execute an Animation
  1. Click the green ‘Execute’ button
  2. Animation will be executed using the poses displayed within the text box above the execute button
  3. Current animation process will be displayed within the terminal window
  4. Once an animation is complete the terminal window will indicate that it is done and execute button will no longer be pressed

ROS Work and ROS Nodes

The first hurdle towards getting the group's systems working with ROS was installing some version of ROS onto the group's Raspberry Pis. This was accomplished by finding an existing Ubuntu 16.04 image specifically for use on Raspberry Pis, with RosPi already installed and set up, graciously provided by Ubiquity Robotics. Even with this foothold, it was a long and difficult process to get the system to a state where the group wanted it. The group needed to install many additional packages and configure obscure settings. To save future groups from going through this process, the group captured an image of the group's final OS, compressed it, and uploaded it to Google Drive for other groups to use. 6GB image, 1.1GB download. The password to login is "ubuntu". This OS image does not have the final version of the group's Inmoov code, but other groups would be using it for other projects anyways. If you simply flash it onto an 8GB+ SD card, it should boot when inserted into a Raspberry Pi. There are several .txt files on the desktop that explain various things about the OS or about ROS, these are also included in the group's Github repo.

The last lingering issue the group couldn't resolve with this OS was that the group couldn't figure out how to set up DHCP for automatic name resolution. As a result, each node was unable to refer to itself or others by the hostnames, and instead needed to use the specific IP addresses they had on the network. This was disappointing and annoying, but didn't impact the function of the ROS network communication at all.

The thing that did impact the group's final product was the absence of a suitable core node, or a suitable user-input node. The NUC which was going to control the Pioneer2 bot and perform SLAM algorithm never got working, and none of the group's members had a Linux-based laptop to run ROS with, and the group were unable to get ROS for Windows working. The ideal planned topology would be to have one Pi in the head, connected to the camera and sending a constant stream of images; one Pi in the Pioneer tower connected to all the servos of the Inmoov torso; a NUC connected to the Pioneer over serial and the RGBD camera, running SLAM and controlling the Pioneer with RosAria, and acting as the core node; and a portable Linux laptop for user input and control.

Having said that, the group did create several useful and working ROS nodes for the group's system:

- The file *body.py* is a ROS node launched with *rosrun inmoov body.py* which should be running on the Pi physically connected to the Inmoov servos. It simply listens to a certain ROS topic for string-type messages, decodes them, and executes the commands they contain with the function *inmoov_ros_command()*. Supported commands include initialization of all servos, deactivation of all servos, single-servo control (either direct set or using the threading approach), pose execution, or animation execution.

- Both GUIs (*body_poser_GUI.py* and *animation_creator_GUI.py*) are designed to be runnable either locally (on the actual Pi connected to the servos) or over ROS. To execute them over ROS, if the *body* node is already running, the GUI nodes are launched with *rosrun inmoov animation_creator_GUI_ros.py* or *rosrun inmoov body_poser_GUI_ros.py*. They behave in exactly the same way, except that communication goes over the ROS network to the *body* node.

- The group devised a crude automatic file transfer system to synchronize the files across the network. There are two nodes, a sender (*file_xfr.py*) and a receiver (*file_rcv.py*). Each pair will serve to synchronize a single file by transmitting the whole file if it is modified. The file must have the same name on both ends, but can live at a different location. This is launched with *rosrun file_transfer file_xfr.py _file:=./whatever.txt*, and the receiver is launched the same way. This does not work with binary files. This is a separate ROS package (file_transfer) from the group's Inmoov code (inmoov), but both are on the Github together.

- The group also devised a pair of nodes for image capture, compression, transmission, and receiving. The sender is launched with *rosrun inmoov camera_capture.py*, and uses a PiCam to send a small compressed image roughly twice per second. This image is received and used by the node *rosrun inmoov camera_receive.py*, and it works, but it doesn't currently do anything with the image it receives.

- The group installed a ROS package called Image Viewer as a way to see a live feed of what the camera_capture node is publishing. This is launched with the command *rosrun image_view image_view image:=/inmoov/head/image _image_transport:=compressed*

- Lastly, the group got RosAria installed and runnable on the Pis. RosAria is a package specifically designed to interface with and control Pioneer robots. The group didn't have much time to explore all of its features, but the remote operation (teleop) works perfectly. The computer connected to the Pioneer's serial port runs the RosAria node with *rosrun rosaria RosAria _port:=/dev/ttyUSB0*. Then the computer which will serve as manual input (can be the same computer) launches the teleop interface with *rosrun rosaria_client interface*.

Pioneer Hardware

The Pioneer2 base is a 3-wheeled metal chassis that has 2 DC motors and 3 12V batteries, which can be controlled through a 9-pin serial port connected via USB of the controlling computer. It also has an array of ultrasonic sonar range sensors (which are confirmed to be operational, but their quality is unknown) and several bump sensors (which are untested). The base has a built-in controller that includes self-testing moves. The initial idea to utilize the base was to allow the InMoov torso to drive around and implement SLAM (Simultaneous Location And Mapping) algorithm running on a NUC board, using an RGBD camera that was provided to us.

Challenges arose while attempting to use the Pioneer base. First, two of the three batteries were dead and had to be replaced, which slowed down progress on the power solution and testing the base. Second, the group was unable to get a NUC to control the base until the last 3 weeks of the course, and then the NUC had hard-drive issues which made it unusable for this group's project. The NUC delays and problems made progress on the Pioneer goal a dead end. Future groups would be able to pursue the SLAM goal if they are able to procure a working NUC with a working hard drive. An Ubuntu OS would need to be installed on the NUC drive, and it would be running Gazebo or whatever is needed to execute SLAM along with RosAria to control the base.

As a last-ditch effort, the group attempted to run RosAria on a Raspberry Pi as a method of controlling the Pioneer, and the group were astonished to find that it worked. Due to time constraints the group didn't explore this much beyond finding how to manually control the driving of the bot, however the group are confident the Pi doesn't have the compute power for complex algorithms or SLAM. To use this manual control, see the ROS section above.

Upgraded Power Solution

A major functionality upgrade was switching to battery centric power solution in contrast to the power supply solution that was relied upon by the last group, as there is no point attaching the InMoov to a driveable Pioneer base if it is still reliant upon an external power supply. This was accomplished by drawing power from the lead-acid batteries that power the Pioneer base and connecting them to the servo hats power inputs. A picture of the power solution fully integrated can be seen below.

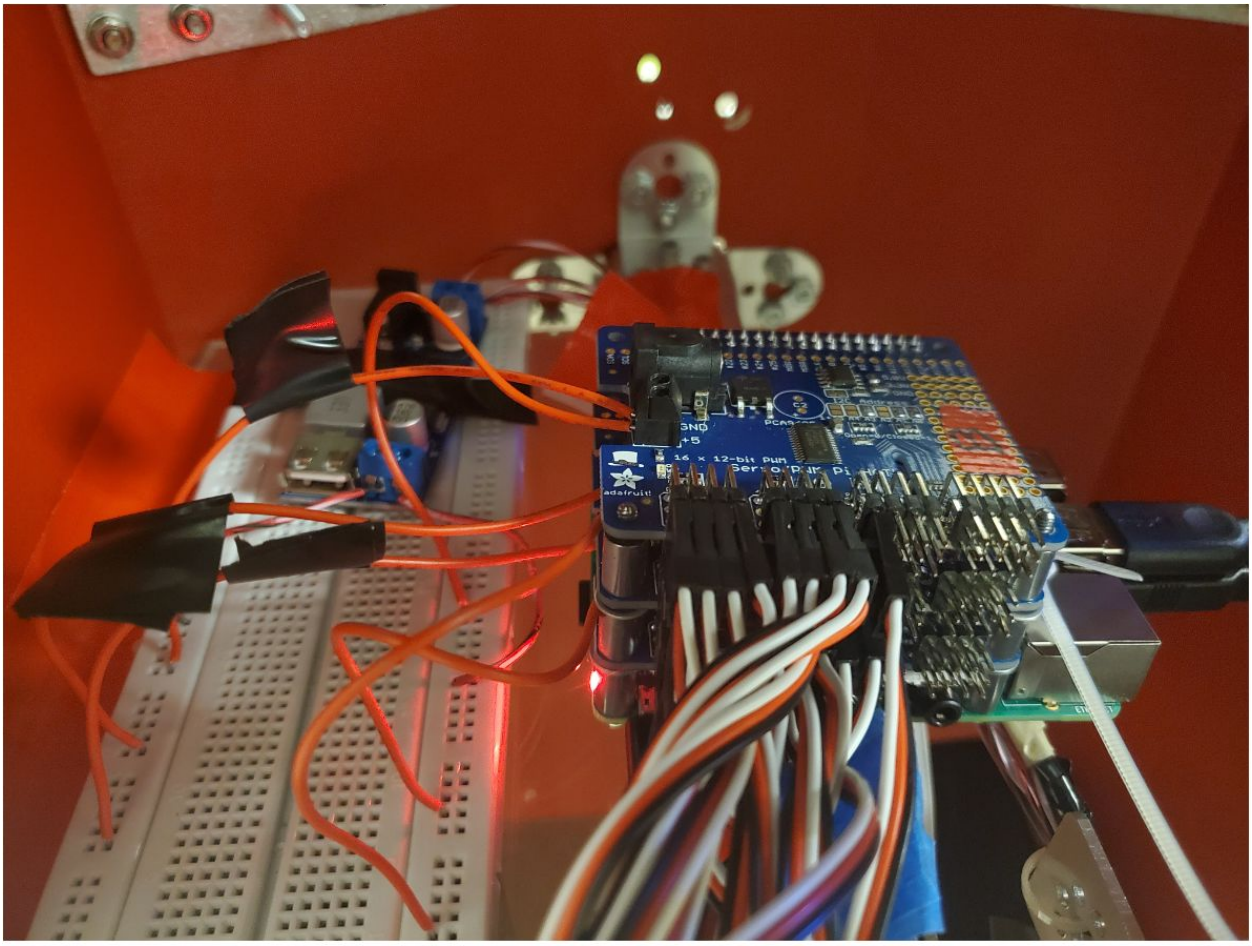

Figure 8: Fully integrated power solution utilizing Pioneer batteries

Before making the modifications, the three 12V batteries were first tested to ensure they were still functional. While one was fine, the other two measured under 5V indicating an internal chemistry failure. Luckily two extra 12V batteries existed in the back of the lab, and these were swapped out to result in three functional batteries. Once the battery functionality was verified, the ability to charge them was verified with a 12V AC adapter.

Once the fundamentals of the new solution were checked, the pre-existing power cables attached to the power nodes were soldered together with some additional wire to allow power access in the torso area. To reduce the 12V battery power to the needed 5V servo power, the group purchased a step-down buck converter which worked very well and promised a maximum current output of 5A. It is theoretically possible for the servos to pull more than 5 amps of current at a time, but only if almost all servos are straining against a load, so this should be perfectly adequate. The group had planned to use a portable USB battery bank to power the Pis, for a more stable and guaranteed power source, but never implemented this plan.

Challenges

There were constant mechanical issues throughout the duration of the project, software was not as much of a challenge as the hardware and physical state of the bot. While these are mentioned in various places throughout this report, a brief summary of the major challenges can be seen below.

1. *Robot initial state:*
   The largest global issue with getting the InMoov robot up to speed was the initial state that it was in. The torso and Pioneer were not connected in any way, no global servo control solution existed, and no power solution existed. The Pioneer base was also completely non-functional. The team was required to re-architect the Python control from the ground up. In addition, the team needed to construct a board-level solution by which all the servos could be controlled from a central Pi. While these efforts were ultimately successful, there was essentially no completed work that could be utilized from the previous group and the majority of the effort was spent on getting the bot to a place where intelligent solutions could be focused on.

2. *Right arm finger issues:*
   During servo testing, an issue arose in the right forearm that was and continues to be challenging to explain. When applying PWM signals to the forearm and finger servos, the servos twitch erratically and uncontrollably, but periodically (video link at pg. 23). It was undetermined whether this was due to faulty servos, faulty servo hats, a power bottleneck, or any other source. The main concern is that this problem seems to be sporadic and is not reproducible every time these servos are used.

3. *Left elbow issues*
   During servo testing, the left arm fell off due to an installation error from the previous group. When this occurred, the potentiometer for the left elbow was damaged and the elbow could no longer be adjusted accurately. Fortunately, there was a spare unused arm that had a potentiometer that was then swapped to the left elbow of the robot. Although the potentiometer appears to be working the overall range of functionality is quite puzzling. The PWM range is merely 25 ranging from about 550 to 575. This range does not allow for a high degree of precision and should be looked into by a future group

4. *“Down” but not “Up” Arm Issue*
   There was an intermittent issue in which an arm could be moved down, but not back up again. The issue would phase in and out, and it was challenging to determine if the root cause was a lack of power, I2C bus clobbering, a python control issue, or a combination of the three. Ultimately as the code was cleaned up and the solution was optimized, the problem went away indicating that cleaner Python control could have been the source of the problem.

5. *NUC issues*
   There were several issues with regards to bringing up the NUC and utilizing it to control the Pioneer Base. For starters, there were delays in obtaining the NUC until the project was significantly underway, thus reducing the time frame to handle any issues. When attempting to install Ubuntu on the NUC, a storage issue arose due to a lack of proper power connection with the SSD storage, thus preventing installation. The connection was fixed and Ubuntu was able to be installed. The SATA connection wasn’t stable though and kept losing write permission due to errors. The group eventually ended up not using the NUC for SLAM and used a Raspberry Pi to control the Pioneer base.

Required Tasks for Future Groups

Broken Components

This is a list of broken components and mechanical issues that need to be fixed, in order to have a functional robot and be able to focus on software:

- 3D print right mid finger and plastic pin, as these pieces are broken and missing
- 3D print broken structures in right arm
- Repair the wrists
- Place springs on string pulleys for better motor control of fingers (see red Inmoov arm)
- More securely attach the head
- Servo hat C seemed to have issues providing sufficient current to servos, resulting in slow and strained movements. Confirm whether the hat is damaged or not and replace if necessary.

Improvement Paths

While significant mechanical integration was completed, full control integration still needs to be completed by the next groups. Here are some of the tasks that need to be taken in order to fully integrate the two robots.

- If SLAM is desired, procure a NUC without SATA issues or an M.2 drive for the NUC.
- Install Ubuntu and ROS on the NUC. (it might be possible to use the Ubuntu RosPi image, but that is untested)
- Use ROS to communicate between 2 Pis and NUC (Pi for body, Pi for vision).
- Power bottlenecks may occur if too many servos are activated at once on the same hat. Potentially distribute load on additional hats? There is no downside to extra hats beyond cost and physical size.
- Add more intelligent behaviors such as facial recognition or object recognition. This might be helped by the *brain/* folder in the Github repository; this was created by the group before us, and the group didn't touch or modify these files at all.

Project Log and Member Contribution

Everyone:

- Soldered stacking headers and right-angle headers to servo HAT.
- Tested servo functionality and power draw.
- Fixed mechanical issues with InMoov robot.
- Assisted with robot calibration.

Meshal Albaiz:

- Procured servo HATS, right-angle headers, stacking headers.
- Modified the Pioneer base to allow for serial port access.
- Procured WiFi Adapter for NUC.

Tristan Cunderla:

- Procured SD card for Raspberry Pi.
- Initially setup and configured Raspberry Pi.
- Created editing and saving mechanism of GUI-based pose maker.
- Created GUI-based animation maker.

Brian Henson:

- Modified and improved up servo drivers.
- Created GUI-based pose maker.
- Procured second Raspberry Pi and SD card.
- Found, installed, and configured Ubuntu for running on the Raspberry Pis.
- Created all ROS nodes and ROS communication.

Danielle Nicholas:

- Worked on power solution to drive 5V servo HATs from the 12V Pioneer base battery.
- Procured servo extension cables and performed cable management.
- Procured and installed Pioneer fuse replacement.

Aron Schwartz:

- Worked on power solution to drive 5V servo HATs from the 12V Pioneer base battery.
- Replaced 2 dead batteries on Pioneer base.
- Modified the Pioneer base to allow for serial port access.
- Created animation executer that uses saved poses from GUI pose maker.

Videos and Repository

Videos

Wave and handshake: https://youtu.be/i33iTY_VzCA

Twitching hand: https://youtu.be/CgvM8TiMXPs

GitHub Repository

https://github.com/mish3albaiz/Robotics_ECE579

Open Source Resource for InMoov

http://inmoov.fr/build-yours/