ROS Based Husky Humanization Report

Cameron Pepe, Jack Lafiandra, Tyler Lee
CS 2699
Cedric Pradalier
5.5.17

Table of Contents

Overview

The goal of the project was to integrate three separate phases on the Husky robot to create a human interactive demo. The three entities are face detection/camera movement, Konova arm movement, and speech recognition/voice response which are implemented and connect on the open source ROS framework. The idea behind the demo would be to give a voice command, have the camera provide input, and the arm to gesture in response.

Initially, the three group members gained familiarity with the ROS ecosystem and then starting using pre-built packages to finally build individual scripts that communicate via ROS topics with publishers and subscribers. The commands were fleshed out among the group members which are identified by number (See list below). For example, stop is the base orientation - robot stops all execution, goes to default position for instruments, and any further command can be executed from this state.

In designing commands, three levels were created - beginner, intermediate, advanced - with the goal being to successfully execute all beginner commands, most of intermediated, and test the waters with advanced. With a semester to accomplish these tasks, the results were positive overall.

Running the System

To execute all three components of the project at once, simply run the following command with the ROS master running:

```
roslaunch ~/husky_dream/src/face_rec/launch/husky_demo.launch
```

The launch file is a bit confusing as one terminal doesn't show much debugging, recommendations is to run each part separately.

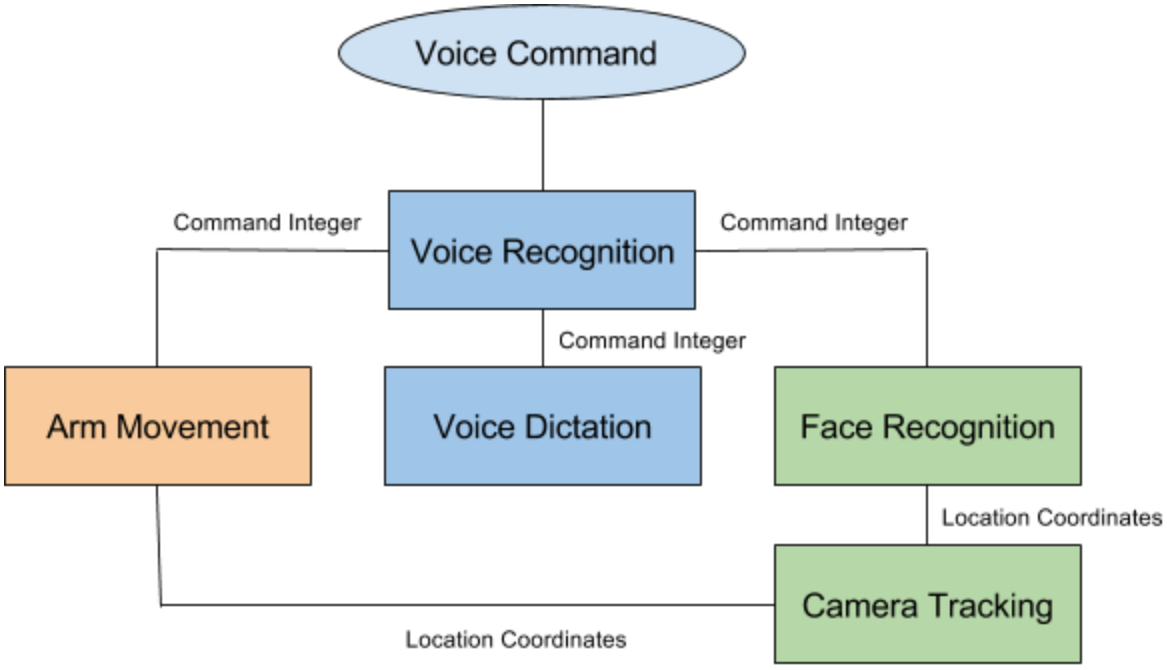
Commands List

#	Audio Command	Audio Response	Camera	Arm Response	Robot Movement
0	Stop	“Stopped”	Stop all computing	Stop moving	Stop moving
1	Home	“Going home”	Move to default position, Look for faces	Move to default position	Stop moving
2	Wave	“Hello”	Output coordinates to closest face	Wave (generic arm movement)	Turn to face user
3	Point	“Pointing…”	Output coordinates to closest face	Point at user	Turn to face user
4	Greeting	“What is your name?” “Hello <name>”	Output coordinates to closest face		Turn to face user
5	Knuckles	“Knuckles	Output coordinates to closest face	Orient for knuckles	Turn to face user
6	High five	“High five”	Output coordinates to closest face	Orient for high five	Turn to face user
7	Turn Left	“Turning left”			Turn 90 degrees left
8	Turn Right	“Turning right”			Turn 90 degrees right
9	Look Left	“Camera turned left”	Hard code turn to angle on left side		
10	Look Right	“Camera turned right”	Hard code turn to angle on right side		
11	Handshake	“Shake my hand”	Align robot to user	Handshake, slight grip, up and down	Turn to face user

				movement	
12	Come to me	“Here”	Tracks where user is		Turn to face user, move to user, stop in front of user
13	Follow me	“Following…”	Tracks where user is	Point at user that we’re following	Follows user, continuously turning to face them
14	Hand me something	“Give it to me” “Thanks”	Align robot to user	Extend open arm, grasp object	Turn to face user
15	Come take this	“Coming…” “Give it to me” “Thanks”	Track where user is, align robot to user	Extend open arm, grasp	Turn to face user, move to user, stop in front of user
16	Find Color (“R/G/B”)	“Searching for color <RGB>” “Pointing…”	Find Color, align robot to color	Point Arm	Turn to face color
17	Pick something up? (spooky hard one)		Find object, orient robot to put object in position	Pick up object	
18	Say Hello	“Hello <name>”	Facial recognition of known individuals	Wave	Turn to face user

*Not all boxes in the table are filled in

Node Plot


Voice Command
Command Integer
Voice Recognition
Command Integer
Command Integer
Arm Movement
Voice Dictation
Face Recognition
Location Coordinates
Camera Tracking
Location Coordinates

Speech Recognition/Response

The speech recognition component of the project relies on the open source Carnegie Mellon University codebase known as Sphinx[1]. Sphinx utilizes a user built dictionary (*.dic) of individual word pronunciations alongside a timing file (*.kwlist) for each word/phrase. Carnegie Mellon provides a web resources for word look up in English[2]. Dictionaries can also be created in other languages although command line resources will have to be downloaded to build the language models. I was personally able to obtain timings for the commands from our list based on examples for Sphinx usage. Finally, Sphinx requires a language model object, the wide majority of which can be found here[3]. These files and objects will need to be referenced in the ROS script, however I set default paths in my script to allow ease of switching between languages via command line arguments.

Sphinx is designed to work with any project in mind, and implementing the system on ROS requires two main packages. The first is Pocketsphinx which is a lightweight sphinx-based voice recognition system meant for smaller, less powerful devices. The second is the python pocketsphinx, a python wrapper which I utilized for ease of use with ROS. Both packages can be found in the cmusphinx Github repository[1].

User input via the onboard microphone also required the pyaudio package which is installable via pip:

```
pip install pyaudio
```

I encountered a major error at this point due to a missing “portaudio.h” file which can sometimes be fixed via

```
sudo apt-get install portaudio19-dev
```

Or online sources[4] as was required for me.

There were quite a few other packages required for execution. Gstreamer is required by sphinx and packages can be installed via the Aptitude package manager.

```
sudo apt install gstreamer<latest version><plugins>
```

In my implementations, I found the good and bad plugin packages were also necessary for sphinx to compile, your mileage may vary. Swig is a translating package written to allow C files to be accessed in python and other scripting languages. It comes with most version of linux although I had to install it from source. The latest release can be found here[5]. Lastly, I was required to install Bison[6] for language parsing which can be installed via the following commands:

```
wget http://ftp.gnu.org/gnu/bison/bison-2.3.tar.gz
tar -xvzf bison-2.3.tar.gz
cd bison-2.3
./configue
make
sudo make install
```

To keep things organized I created a folder to house source code for packages in case reinstallation became necessary as well as to house the language model and files. The folder I created is named “audio_ros_research” and can be found in the home directory of the Husky robot. For a multi-user system it would be better to place the model and files in the “/share” folder. My script’s current defaults point to this location. The model can also be placed in the ROS package which I avoided to ensure ROS would still make in all future versions.

My speech recognition script publishes two topics, “voice_command_int” and “voice_command_str” which correlate directly to the command lists numbers (..._int) and command name (..._str) documented above. When a phrase is understood, the result is sent to the both the arm and the camera via the integer topic to execute the proper response, the string is largely for ease of debugging and testing.

I ran tests on a voice response system using the audio common ROS package which includes a say.py wrapper script that takes command line string arguments and has a speech output. It works as developed with any string, but the response is extraordinary robotic due to the nature of concatenating sounds to create words. In my tests, I attempted to use modified strings in order to have the proper sound output for the desired word. I began modifications to work via subscriber access as well as searching for an alternative voice response system, but was unable to finish due to time constraints.

My catkin package is named “voice_controller” and is located at

```
~/husky_dream/src/voice_controller
```

with scripts specifically in the “scripts” folder and launch files in the “launch” folder within the package.

The main script can be run via the following command once the catkin make has been performed.

```
rosrun voice_controller husky_voice_controller_v2.0.py
```

Speech recognition is a tricky task even with the well build libraries I utilized. Initial tests realized about 80% accuracy on my personal device with a quality built in microphone and minimal optimization. After completing the setup on the husky, I was seeing around 60% accuracy, although with adjustments to the microphone this dramatically improved as well as additionally removing some of the background noise in the lab. I also found revising some of the commands for clarity amongst the words helped; for instance “Halt” was changed to “Stop” to avoid confusion with “Home”. I also spent time adjusting the phrase timing values with a net result of about 90% accuracy. My biggest dilemma is that constantly listening results in attempts to find words/phrases is regular conversation not meant to be robot commands, although with a more directional microphone this problem could also be mitigated.

The codebase for my implementation can be found on my Github[7].

Sources

[1] https://github.com/cmusphinx
[2] http://www.speech.cs.cmu.edu/cgi-bin/cmudict
[3] https://sourceforge.net/projects/cmusphinx/files/Acoustic%20and%20Language%20Models/
[4] http://www.portaudio.com/docs/v19-doxydocs/portaudio_8h_source.html
[5] http://www.swig.org/download.html
[6] https://www.gnu.org/software/bison/
[7] https://github.com/tlee753/husky-robotics-research

Face Tracking/OpenCV/Camera

The program can be run by calling

```
roslaunch face_rec face_rec_bag.launch
```

The facial tracking and camera control work primarily with two libraries: *OpenCV* for image processing and facial detection and *Axis* for camera input and output. The OpenCV2 and Axis packages are installed in the ROS libraries on the Husky robot.

There are two scripts that I wrote that works with these two libraries: face-rec-axis.cpp and output-action.cpp.
Face-rec-axis subscribes to `/axis/image_raw` (the output image of the Axis camera), and publishes to `/faceROI` (the region of interest in the image designating a face, in pixels). The program uses OpenCV to scan the video image for faces and stores the locations of the faces in a vector. It then publishes the region of interest of the largest face (by pixel size) to the topic /faceROI. The biggest issue run into with this script is the low frame rate and low image resolution of the camera. The lack of sophistication of the facial detection through OpenCV also posed an issue because, for example, a face turned at an angle would not be detected and sometimes an inanimate object would be recognized as a face for a single frame. Due to the low frame rate and low resolution of the camera, I was unable to avoid this issue altogether, only mitigate its effects on the rest of the program.

Output-action subscribes to:

- `/faceROI` (published by face-rec-axis.cpp)
- `/axis/camera_info` (the technical and computer vision information of the Axis camera including the intrinsic camera matrix)
- `/axis/state` (pan, tilt, and zoom information from the Axis camera)
- `/voice_command_int` (published by Tyler's speech recognition program containing which command was given by voice)

The program publishes to

- `/axis/cmd` (the command for the Axis camera including the pan, tilt, and zoom (among other) command).
- `/faceLoc` (the information about the location of the face in terms of angles for Jack to use for arm operations.

The program reads the information provided by face-rec-axis on /faceROI (the location of the face in the current image) and, using the intrinsic matrix of the camera, calculates the angle that the person's face is relative to the center of the image. The angle in the x (horizontal) and y (vertical) directions are relative to the standard cartesian coordinate axis of the image. These angles become the *pan* and *tilt* variables (with some variated scaling based on the current zoom of the camera) in the command message to the camera. If both the *x* and *y* variables are within a threshold (10 degrees), then the zoom variable is increased. This message is published to /axis/cmd. The camera reads this topic and adjusts its pan, tilt, and zoom variables accordingly. This works to move the camera so that the person's face is in the

center of the image. When the face is within 10 degrees of the center of the camera, the zoom is increased until the face is roughly 1/7 the screen's width. Once the camera is zoomed in on the person's face, it continues to track the person's face and adjust the pant and tilt variables accordingly. The size 1/7 was found to be a good mix of size and ability to continue to track faces with the low frame rate.
If the camera is zoomed in and a person's face cannot be found in the image for over 1 second, the camera begins to zoom out. The speed at which it zooms is scaled by the time since the last face was found. After 3 seconds without finding a face (receiving a message on the /faceROI topic), the camera resets to a pseudo-base position. It maintains the same pan angle but adjusts the tilt angle to 10 degrees and resets the zoom to 1 (all the way zoomed out). After 5 seconds without finding a face, the camera begins to scan its area for faces. The camera pans from -60 degrees to 60 degrees (0 degrees being directly in front of the husky) at roughly 5 degrees a second. Once a face is found, the scan state is exited and focuses on the face it found (going back to the first step).
If the voice command "stop", "home", "look right", or "look left" is detected (Tyler's part):

- Stop - Stop the camera where it is, (keep the same pan, tilt, and zoom) until another voice command of any type is detected
- Home - Reset the camera's position to face directly in front of the husky
- Look Right - set the camera's position to face 45 degrees to the right of the husky
- Look Left - set the camera's position to face 45 degrees to the left of the husky

Sources

- http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html
- https://github.com/cedricpradalier/axis_camera

Arm Movement

The main program can be run by calling

```
rosrun ArmController controller.py
```

ArmController subscribes to:

- `/faceLoc (published by Cameron)`
  - `Face = whether a face is present`
  - `Theta x`
  - `Theta y`
- `/voice_command_int (published by Tyler)`

The arm controller works primarily with the Kinova drivers, so the program will never run correctly without first launching the drivers, which can be done by doing this command in the command line:

```
roslaunch kinova_bringup kinova_robot.launch
kinova_robotType:=j2n6a300
```

The entire Arm Movement works through a series of python scripts located under ArmController/scripts. The primary script is controller.py which use a robot object from robot.py, which has a method called command which determines which command needs to be executed. The command function takes in two parameters, the first being an integer from the Speech Recognition that corresponds to a key phrase, while the other parameter consists of theta x from Cameron's Face tracking. The command function will then activate one of the scripts either *jointvel.py(wave functionality)* and *jointpos.py*(point functionality). The point functionality will locate a face and then point directly towards the person, while the wave functionality will wave in front of the robot. Another important function is the service call for the arm, used with

```
rosservice call /in/j2n6a300_driver/home_arm
```

Be careful when using this though, because it can run into any obstacles on the way home. The problems that I ran into when working on this project are a few, the first one that comes to mind is that the Kinova-ros drivers and the VRep drivers are quite different in actuality; Vrep gives you a good idea for proof of concept, but the actual execution is much different in how they are coded, with Kinova using JointVelocity while Vrep using jointcommand. Another large problem while coding was that quite a bit of functionality in the kinova-ros package is still being developed from what I understand. Particularly, absolute joint_angles would cause the arm to often flail randomly, and often times it would have to be shut off. Furthermore, Cartesian control did not work very well, it did work, but it was incredibly slow and somewhat glitchy in the movement sector.

Sources

- https://github.com/xwu4lab/kinova-ros
- https://github.com/xwu4lab/jaco_husky_demo

Conclusion

Collectively each of our individuals nodes connects and accomplishes the task set out at the beginning of the semester. We have learned quite about about the ROS framework, publishing and subscribing, the Husky hardware, simulation, speech libraries, OpenCV image processing, and manipulating the Konova arm. We have all created ROS packages which run scripts we have each made.

As with any project, the horizon has grown and there is definitely improvements and upgrades to make. While we accomplished our goal of performing all the beginner tasks and well into the intermediate, there is plenty of knowledge to be learned from the advanced category which we were only able to discuss theoretically. Movement of the robot would also be a huge addition and while it didn't fall under any of our jurisdictions, it would greatly contribute to the humanization element of the objective.