Carnegie Mellon University

16-681

MRSD Project I

Task 8 Progress Review 2
Team C - COBORG

Jonathan Lord-Fonda

Teammates: Husam Wadi, Feng Xiang, Yuqing Qin, Gerry D'ascoli

March 18, 2021

Table of Contents

1. Individual Progress 1
2. Challenges 3
3. Teamwork 3
4. Plans 4
5. Appendices 4
   - 5.1. Appendix 1 – V3 ROS Node Map Walkthrough 4
   - 5.2. Appendix 2 – Goal_Setter Intuitive Draft 5
   - 5.3. Appendix 3 – Frame_Transforms Intuitive Draft 8
   - 5.4. Appendix 4 – Move_It Intuitive Draft 8
   - 5.5. Appendix 5 – Path_Shifter Intuitive Draft 10
   - 5.6. Appendix 6 – Motor_Controls Intuitive Draft 11

1. Individual Progress

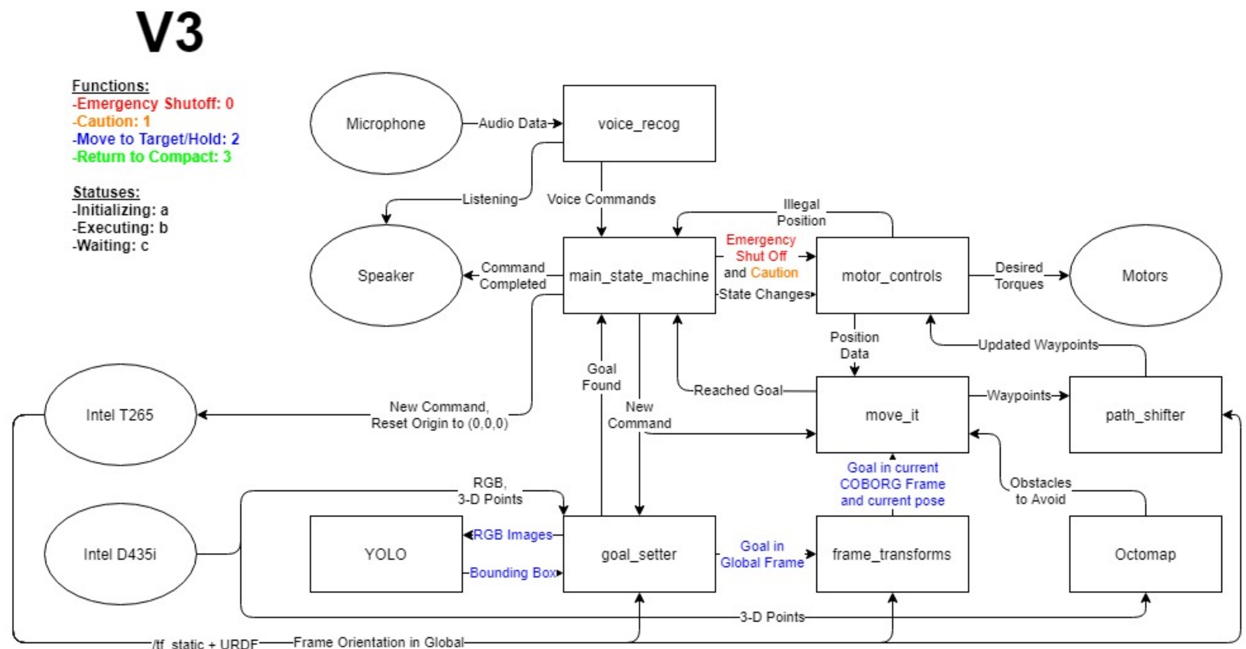
My primary tasks since the last progress review included implementing the main_state_machine node and connecting it with the voice_recog node, updating the ROS Node Map with a proposed structure, writing out semantic versions of each of the nodes, working with Jason on the Actuated Manipulation subsystem, and reading up about Elastic Bands.

My first task was to implement the main_state_machine node on the Coborg's computer and get it to talk with the voice_recog node and publish states out to a topic. I worked with Gerry and wrote up the node and we tested it. It successfully interpreted the voice commands published to it and published out corresponding commands. This was one of the first pieces of software implemented in ROS on the Coborg system.

After this I set out to update the ROS Node Map to include localization corrections for a person shifting their position while the Coborg was in use, as seen in Figures 1 and 2, below. The most important change was adding the path_shifter node between the move_it and motor_controls nodes. The idea behind this node is that it could take the waypoints generated by move_it, when the robot was at a certain pose, and shift them so that they'd end up in the same

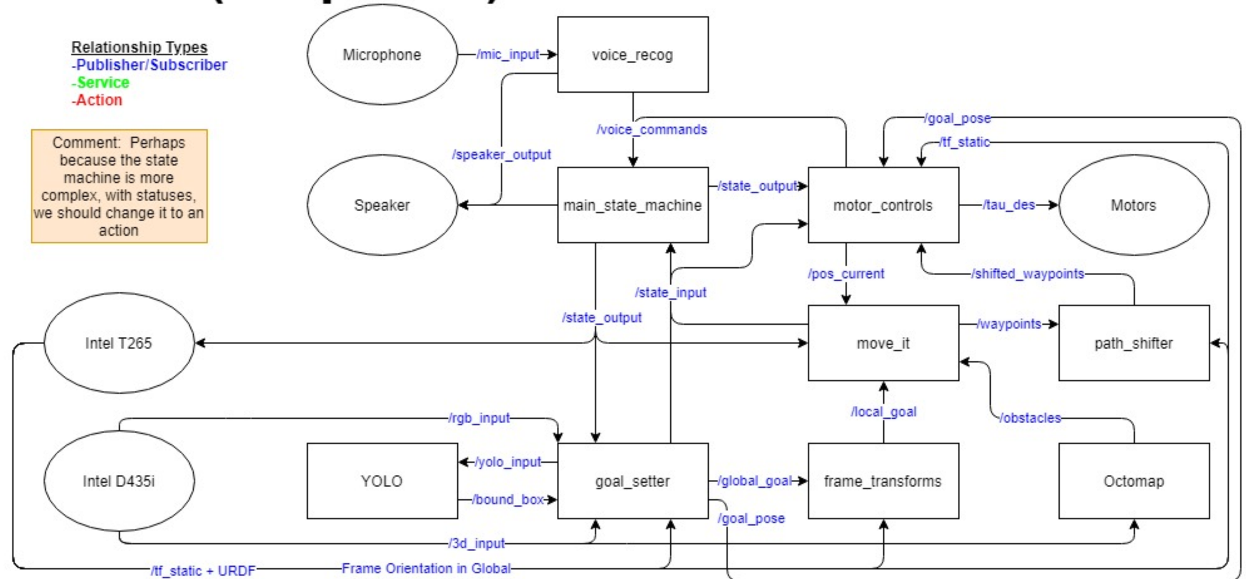
global position as expressed in the current local frame. Additionally, discussion with Jason and Husam helped me understand the required inputs for move_it and led me to add a frame_transforms node that transforms the goal coordinates from the camera from the global frame into the local frame. Further consideration of each node's function led me to have far more nodes feed into and read from the main state topics of the Coborg, so that they could make more informed decisions. One good example of this is the motor_controls node which chooses which control policy to follow based on the current state of the system.

Figure 1 - Updated ROS Node Map, Semantic Descriptions



Changes to the ROS Node Map not discussed above include feeding the robot's global pose into more nodes, as well as having the motor_controls node feed information on whether the robot has achieved an illegal pose into the main_state_machine node.

Figure 2 - Updated ROS Node Map, Node and Topic Descriptions



As described in the caption for Figure 1, a number of nodes and topics were added and named. Additionally, which is unique to this graph, the specific topics were rearranged to accommodate the increased connections.

Alongside updating the ROS Node Maps I also wrote out a walkthrough of the system's primary function and pseudo-code for every node present on the map. The outline and full pseudo-code can be found in Appendices 1 through 7 in order of their operation according to the outline. Writing out the pseudo code helped me understand everything required in the nodes' function which led to more topic connections on the ROS Node Map; they were symbiotic activities.

After updating the system architecture I coordinated with Jason to work on the actuated manipulation subsystem. We went into the lab and he explained how the whole system worked currently, how we were currently planning and executing paths, how to boot up the system, etc. We also discussed how to solve the localization issue for actuated manipulation. In a conversation, Dr. Kroemer had mentioned the use of "Elastic Bands" so we researched that concept for our system. Upon digging into it I found out that the system was developed for mobile robot bases, not manipulators, but that there was at least one subsequent paper that focused on implementation for manipulators.

Other activities that I accomplished included setting up my personal Linux and ROS systems on a persistent USB. I also met with Kelvin Kang to have him review my ROS Node Map and advise on whether he saw any problems or issues, given his experience with the software.

2. Challenges

One of my primary goals for this review cycle was to finalize the ROS Node Map so that it could successfully handle the localization of our target for the actuated manipulation system. However, after creating the ROS Node Map and semantic node drafts I realized a major problem. Move_it provides waypoints for mobile robot bases, but joint angles for manipulators. Since we are using a manipulator we cannot simply take a series of waypoints provided by move_it, perform a homogeneous transform on them, and send them forward in the current local frame. We have a number of possible solutions. The first is to rapidly run forward kinematics on these joint angles to turn them into waypoints, convert them as planned with a homogeneous transform, and then perform inverse kinematics to return them to joint angles. Unless this is extremely fast it won't be a solution for us. A more likely solution is to implement some form of Elastic Bands to continually update our trajectory based on our current position and shifting obstacles. Our mentor, Julian Whitman, recommended that we look into using CHOMP because it operates similarly to Elastic Bands and is included in the move_it library. Additional challenges included a number of bugs while trying to implement the main_state_machine node in ROS.

3. Teamwork

Since the last progress review, Jason was able to tie the T265 and D435i cameras to the URDF model, allowing Rviz to update and display Coborg's global position. We also spent time discussing the actuated manipulation subsystem.

Since the last progress review, Gerry integrated a new microphone with our system to improve voice recognition. He also helped me develop and implement the main_state_machine node and get it functioning with the voice_recog node that he created a ROS wrapper for. Gerry also updated our project's website and developed the conceptual design for our Coborg PCB with Husam.

Since the last progress review, Yuqing implemented 3d YOLO in ROS and was able to output bounding boxes of multiple hands, publishing the bounding boxes to a topic. She also performed post-processing for the average 3d position of the bounding boxes, setting up the goal_getter node so that YOLO could function within our system. Additionally, Yuqing also combined the launch files for the d435i and the t265 cameras.

Since the last progress review, Husam helped implement the main_state_machine node in ROS. He also assisted Jason with the t265 tracking camera output and ensured that the team used Github.

4. Plans

Before the next progress review, I would like to finish researching the Elastic Bands method for our path planning and implement either Elastic Bands or CHOMP for the Coborg robot. Doing that would require working alongside Jason. Additionally I'd like to perform a number of updates on work that I previously did. This would include updating the main_state_machine node with added functionality now that it has been tested and going through our system's current requirements and validation plans. The speaker needs to be added to the requirements and validation plans. I also need to go through each of the validation plans for each subsystem with their owners to ensure that we're on track to accomplish what we set out to do. This is important because if, for example, we test the cameras and they prove to be highly inaccurate, we need to determine solutions and begin implementing them now while we still have some time before our final demonstrations. It's also important to update the validation plans now that we as a group better understand how our system is shaping up so that we can start finalizing details and preparing the necessary externalities and measurement equipment.

5. Appendices

5.1. Appendix 1 – V3 ROS Node Map Walkthrough

This is a walkthrough of how I envision the ROS Node Map V3 acting when a command of "Move to TARGET" is issued, with all optional features included and marked with a *. Note: The optional features can just be not coded or commented out if we don't like them and it will still work fine.

main_state_machine sends out a command
goal_setter pulls RGB, 3d, and pose data, wrapping them together
goal_setter sends the RGB images to YOLO
YOLO returns the bounding boxes of the hands
goal_setter transforms the bounding boxes and 3d points into the world frame
goal_setter determines the global location of the hand
*if accuracy/intersection is low enough, goal_setter will add more images or replace and repeat
goal_setter determines global target and passes it to frame_transforms

frame_transforms takes the global target, transforms it into a point in the COBORG frame, and passes that to move_it
frame_transforms also passes the frame of the goal to move_it to maintain the data connection
move_it runs RRT Connect to create a path from the base of the robot to the local point
*move_it runs the last step of RRT Connect one more time to update the goal
move_it sends the path (with its "old" pose) to path_shifter
*path_shifter checks the current pose and the "old" pose and shifts the waypoints accordingly
*path_shifter scales the shift to smooth disruptions
path_shifter sends the updated waypoints to motor_controls
*motor_controls picks the best control scheme to use based on # of waypoints left or distance to last waypoint
motor_controls determines the motor torques based off of the control scheme, waypoints, and current position data
motor_controls applies a limit to the torque
motor_controls publishes the torques to the motors
There may be a HEBI module between motor_controls and Motors

5.2. Appendix 2 – Goal_Setter Intuitive Draft

Goal_Setter Intuitive Draft

```
On Start-up{
    Initialize publisher to /state_input
    Initialize publisher to /goal
    Initialize publisher to /yolo_input
    Initialize subscriber to /state_output
    Initialize subscriber to /rgb_input
    Initialize subscriber to /3d_input
    Initialize subscriber to /tf_static
    Initialize subscriber to /bound_box
}

If receiving a 2 from state_output (COBORG Hold command){
    //////////////////// This could be a first step, essentially, can we pull in a set of data?
    Create rgb vector
    Create 3d points vector
    Create /tf_static vector
    Create bound_box vector
```

Add the current rgb image to rgb vector
Add the current 3d points to 3d point vector
Add the current pose to /tf_static vector
Publish rgb to /yolo_input
Create index i to iterate through all vectors
////////////////////
}

On receiving an image from YOLO{
//////////////////// This could be a second step, testing if we can evaluate YOLO's results well and tuning the system.
//////////////////// If YOLO ends up being really accurate here, we can throw out most of the complexity later
Check to see whether the bounding box's accuracy passes a threshold.

If it doesn't, either throw the results out or simply add another data set to each vector and publish the rgb to /yolo_input.

If the bounding box (or boxes) achieve a certain threshold of accuracy or intersection, run Goal Finding Program (below)
////////////////////
}

Goal Finding Program (runs after good enough results are achieved){
Create bounding box midpoints vector (Maybe create a "left" and "right" one?)
Create Goal Midpoints vector
Create Goal Locations vector
Create Goal Poses vector

//////////////////// This could be a third step, can we transform the images to a global reference?
//////////////////// If we're doing this I guess we might have to grab the 3-d coordinates of the bounding boxes
Use /tf_static vector to transform bounding boxes into the global frame of reference
////////////////////

//////////////////// This could be a fourth step
Find intersection of leftmost bounding boxes

Find midpoint of leftmost bounding boxes (H/2,W/2), add it to the bounding box midpoints vector

Find intersection of rightmost bounding boxes

Find midpoint of rightmost bounding boxes (H/2,W/2), add it to the bounding box midpoints vector

Find Goal Midpoint between bounding box midpoints, add it to the Goal Midpoints vector

////////////////////

//////////////////// This could be a fifth step, potentially done at the same time as the third step?

Use /tf_static vector to transform 3-d point clouds into the global frame of reference

////////////////////

//////////////////// This could be a sixth step, similar to the fourth step, but using multi-dimensional splines because the 3-d point cloud is sparse

Change 3-d point clouds to splines

Find the depth of each point corresponding to Goal Midpoint in each of the 3-d point clouds, add the 3-d point to Goal Locations vector

////////////////////

//////////////////// This could be a seventh step, can we get the pose of the plate from the 3-d point cloud? Again, we can use the spline.

For each of the Goal Midpoints in each of the 3-d point clouds, check a small area around the Goal Midpoint to determine the surface pose, add it to Goal Poses vector

^-This could be as simple as + and - 1 pixel vertically and horizontally to determine the x and y components of change in depth

////////////////////

//////////////////// This could be an eighth step, if everything else works, can we smush it together and publish our final answer?

Average Goal Locations

Average Goal Poses

Publish the averaged Goal Locations /global_goal

Publish the averaged Goal Poses to /goal_pose

Publish "b" to /state_input to let the Main_State_Machine node know that the command is done initializing and is now executing

////////////////////

}

5.3. Appendix 3 – Frame_Transforms Intuitive Draft

frame_transforms Intuitive Draft

On Start-up{
    Initialize publisher to /local_goal
    Initialize subscriber to /global_goal
    Initialize subscriber to /tf_static
}

On receiving a global goal from /global_goal{
    Overwrite the previous global goal
}

On receiving a global COBORG frame from /tf_static{
    Overwrite the previous global COBORG frame as the one received from /tf_static
    Apply a homogeneous transformation on the stored global point to re-write it in terms of the COBORG's current frame
    Publish the local point and the current global COBORG frame in /local_goal
}

rospy.spin()

5.4. Appendix 4 – Move_It Intuitive Draft

move_it Intuitive Draft

On Start-up{
    Initialize publisher to /waypoints
    Initialize publisher to /state_input
    Initialize subscriber to /state_output
    Initialize subscriber to /local_goal
    Initialize subscriber to /obstacles
    Initialize subscriber to /pos_current
}

On receiving a local goal and COBORG global frame from /local_goal{
    Update the current local goal
    Update the current COBORG global frame
    // The COBORG global frame isn't used anywhere in this code. It's passed on so that the next node knows how "old" the path is.

}

On receiving an obstacle list from /obstacles{
    Update current local obstacles
}

On receiving a position from /pos_current{
    Update current local pose of robot (joint angles)
    If the current position of the robot is within x distance from the stored goal, publish status WAITING in /state_input
    // This basically just says that we've reached our location. It lets us switch our control type and also readies the system to receive another command
}

On receiving a HOME command from /state_output{
    Store state value as HOME
    Create an RRT Connect path to achieve the desired pose
    Publish waypoints to /waypoints
    // Include variable in the /waypoints message to designate that the waypoints shouldn't be changed
    Publish EXECUTING status to /state_input
    // We only need to publish once and we don't need any other information because it's just a local pose that's always the same
}

On receiving a TARGET command from /state_output{
    Store state value as TARGET
    Set status to PLANNING
}

While rospy isn't shut down (Constant loop){
    If state value is TARGET{
        Store the current local goal, COBORG global frame, local obstacles, and local pose of robot in a separate set of variables that won't constantly update
        // The above variables will update once per path, not constantly like the ones defined above
        Create an RRT Connect path using the above variables (local goal, local obstacles, and local pose of robot)
        // Optionally, when we finish making the path we can update the local goal so that it is current and do one more step of RRT Connect to update it.

// If we do the optional procedure above, we'd also send an updated COBORG global frame

Publish waypoints and the stored COBORG global frame to /waypoints

If status is PLANNING, set to EXECUTING and publish EXECUTING to /state_input

// We just don't want to spam /state_input with executing commands, especially if the main_state_machine has moved onto other functions

}

}

// Another optional feature we can add to this node is pre-planned paths. Basically, if we're sent a command to move to TARGET we can just check which rough area our target location is in (>45 degrees above horizontal? Between -45 and 45?, etc.). We can then immediately publish a path to move it to that general area before starting on our first "real" path. This will get the COBORG arm moving as soon as it detects where to go. It wouldn't have obstacle detection until RRT Connect plans the first path though, but it would basically save us the runtime of one loop of RRT Connect each time we called TARGET.

5.5. Appendix 5 – Path_Shifter Intuitive Draft

path_shifter Intuitive Draft

On Start-up{

Initialize publisher to /shifted_waypoints

Initialize subscriber to /waypoints

Initialize subscriber to /tf_static

}

On receiving a set of waypoints and COBORG global frame from /waypoints{

Update the current waypoints

Update the stored COBORG global frame

If the message indicates that there's no need to shift the points (because it's just going home), set Shift to 0, otherwise set it to 1

// This COBORG global frame corresponds to the RRT Connect waypoints

}

On receiving a COBORG global frame from /tf_static{

Create and store a homogeneous transformation from the stored COBORG global frame to the current COBORG global frame

}

```
While rospy isn't shut down (Constant loop){
      If Shift is equal to 1{  // Basically, if we're supposed to shift it
            Use the homogeneous transformation stored above to transform the waypoints, shifting them so that they're up to date
            // Optionally, we can also apply less transformation to the earlier points to smooth the transition
            Publish the shifted waypoints to /shifted_waypoints
      }
      Otherwise, publish the normal waypoints to /shifted_waypoints
}
```

5.6. Appendix 6 – Motor_Controls Intuitive Draft

```
motor_controls Intuitive Draft

// We should consider writing this code in C++ because it's so low level and needs to be FAST

On Start-up{
      Initialize publisher to /voice_commands
      Initialize publisher to /tau_des
      Initialize publisher to /pos_current
      Initialize subscriber to /state_output
      Initialize subscriber to /state_input
      Initialize subscriber to /shifted_waypoints
      Initialize subscriber to /goal_pose
      Initialize subscriber to /tf_static
      function = HOME
      status = initializing
      waypoints = the current position (i.e. don't move)
}

On receiving a new function from /state_output{
      Store the current function as function
}

On receiving a new status from /state_input{
      Store the current status as status
}

On receiving a new set of waypoints from /shifted_waypoints{
```

```
        Store the current waypoints as waypoints
}

On receiving a new goal pose from /goal_pose{
        Store the current goal_pose
}

On receiving a new COBORG global pose from /tf_static{
        Store the current COBORG global pose
}

While rospy isn't shut down (Constant loop){
        Publish the current joint angles to /pos_current
        if the current joint angles violate the standard "core" of the assumed
operator{
                // This function is optional, particularly if it causes false positives
                Set function to E-STOP
                Publish all zeros to /tau_des
                Kill power to motors
                Publish E-STOP in /voice_commands
                // This is so that the system stops trying to send commands.  Perhaps
we can include a way to pull out of this emergency state without having to reboot
        }

        // Consider adding a CAUTION state in which the motors just hold their
location

        if function is NOT E-STOP{
                // Use if statements to select ideal control method.  This is optional,
but I think it will be really important for stabilization.
                if function is TARGET and (the robot is within x distance or status =
WAITING){
                        We've reached our location, we'll use impedence control with
high force along the perpendicular axis and loose control over the parallel axes.
                        This should let the arm "give" easily to help account for the
user's motion, while still holding the panel up.
                        We can determine the axes from goal_pose and the
COBORG global pose.  We do this by creating a vector perpendicular to the
pose and then transforming it into the COBORG's current reference frame.
```

Use control function, waypoints (desired position), and current position to determine motor torques

Apply bounds to motor torques (limit the max)

Publish motor torques to /tau_des

}

else if function is HOME and (the robot is within x distance or status = WAITING){

We can create a set of gains optimal for maintaining its 'compact' position

Use control function, waypoints (desired position), and current position to determine motor torques

Apply bounds to motor torques

Publish motor torques to /tau_des

}

// We can add as many else if's in here as we want to optimize for particular states, or we can have no if's and just use one controller for every state

else{

Use some "typical" canned function for all other cases

Use control function, waypoints (desired position), and current position to determine motor torques

Apply bounds to motor torques

Publish motor torques to /tau_des

}

}

}