Robot

ROSE is designed to be a mobile system to replace a waiter. In order to accomplish such tasks, our team attempted to find a solution that was flexible and reliable, mobile in many common situations, and has the ability to interact with items a waiter normally would interact with.

Our group used VEX 393 motors at each joint, often in parallel for higher torque. The wheels on the base have motors driving each of them and the arm has 13 motors adjusting its degrees of freedom. These motors are controlled via an Arduino UNO v3 microcontroller coupled with 2 Adafruit motorshields to accept power from 7.2V batteries. The distributed Arduinos are connected to a 12V powered 9-port USB3.0 hub for communication to a central computer. They are powered by a 12V power source. The central computer is a Jetson TK1, an NVIDIA system-on-chip that contains an Arm Processor and a Tegra CUDA-capable GPU. In addition, there is a PS3 Eye as the camera and multiple sensors that all send back feedback to the TK1, often through the arduinos.

In order to control the positions of the arm and legs, there are potentiometers attached to every angular joint and encoders attached to every joint (making 13 potentiometers and 4 shaft encoders in total). Each potentiometer is calibrated to their maximal angles, and then PID coefficients are calculated on each Arduino for their respective controlled joints so that they can try to reach the desired steady-state position. The angles and motor speeds, along with current and voltage readings, are sent back to the central processor via a single pre-formatted serialized string, along with an ID for initial connection identification. The strings are then read and stored in buffers, which are then deserialized and analyzed for information. When sending out commands from the central computer, each Arduino receives a serialized string which contains both desired positions or velocities and an enable number to enable either position, velocity, or neither.

Pose of the robot is determined through the forward kinematic calculation of the end-effectors and the inverse kinematic solvers from a given target position. The angles of the joints and the lengths of the links are stored within a definition file, which is used during the forward kinematic calculation to determine the positions of the wheel end-effectors and all interim effectors. The lengths and any updated angles are solved by going down the chain of rotation translation matrices.

When the program specifies a desired pose for the robot, it sends the positions of the end-effectors into the inverse kinematic solver, which takes the calculated pose from

forward kinematics and compares it to the desired positions. The arm has multiple configurations which depend on the normal vector of the end effector. In order to calculate the end-effector's configuration, we find valid normals via 2-depth binary search, and then the mean is chosen as the most feasible configuration. If a feasible position has indeed been found, it sent to the arm to assume its desired angles. However, if there are no feasible positions, no angles are sent.

In order to move the robot from one location to another, it must be able to localize its position. Many industry-wise robotic systems use kalman filters or variants of it. However upon analysis we agreed that given the complexity and mechanical flexibility of our system, deriving the motion and observation matrices (which are required by the filter) would be too extensive to ascertain.

We needed a robust localizer, which we accomplished this by implementing a particle filter algorithm. We chose the particle filter algorithm because of its high fidelity in dynamic environments. Dynamic environments are those which have incomprehensible events from the robot’s perspective.

First, a normal distribution of position hypotheses were generated around the starting position of the robot. We then use a grid-based representation of our map, with landmarks marked in discrete locations. The landmarks we first used are chilitag fiducial markers, which provide us a ground truth landmark identifier. The measurements are weighed with a gaussian probability distribution around the last mean coordinate of the robot. The hypotheses, or particles, are then updated using a motion model provided via the readings of the encoders from the wheels of the robot. The particles are then resampled, where each particle is weighted with a bivariate normal polar product so that measurements that are more probable are resampled more often. After about .1 second, 1000 particles coalesce to the current position of the robot if landmarks are sensed.

Server:

The server communication consists of 2 segments, web app to database communication and ROSE to database communication. Web app communication uses PyMongo to interface the Python code of the web app to the MongoDB server. In turn, ROSE will access the database using the MongoDB C++ Legacy driver. Both the web app and the server will have to create a connection to a MongoDB database. In this regard, the database's name is "rosedb" and for any machine that has MongoDB installed, the database "rosedb" will automatically be created the first time that the program runs. On subsequent calls to the program, the database will not be recreated, but simply reinitialized with starting values. In addition, MongoDB collections and documents will work in a similar fashion, only being created if the corresponding collection or document does not exist yet.

Web app to database communications:

The web app operates with bidirectional communication to and from the server. The collection created by the web app to send controls information is currently called "mycollection" and will be created if it does not exist on the host server yet. The information that the web app sends to the server is information on how ROSE should move. These movement commands include

- Translational motion
- Rotational motion
- Change in velocity

The web app also is able to retrieve information from the robot, using the database as a medium, currently the web app retrieves the current voltage used by the batteries of the robot.

ROSE to database communications:

ROSE also has the capability of bidirectional communication with the server. As the web app pushes commands to the database based on user input, the robot will extract those commands and interpret them. This process begins by first storing the information of each command into a variable by type casting the variable from a the MongoDB BSON format to defined types in standard C++. These variable will then be stored within a struct that is associated with the database communication. Once the information has been stored within the struct, the information will proceed to be interpreted by functions that reside within the robot's code. As far as when ROSE will read from the database, this is done through polling the information from the database, checking on every millisecond whether or not a new command has been sent to the database by the web app. In regards to sending data, the robot will be able to send the

data to the database by storing the variables that need to be sent into a BSON format in which MongoDB can read. The robot will create a collection called “robo_info” that will hold the information about the conditions of the robot. We store these variables into documents that will be used to update the currently existing documents within the MongoDB collection.

Web Application

The manager interface for the web application was implemented using a combination of HTML, Javascript, and Python using the Flask microframework. The following illustrates how this application was designed:

- Button functionality: HTML <Button> tags with appropriate “onmousedown” and “onmouseup” attributes, as well as short Javascript functions, were used.
- Key press functionality (arrow keys for movement, A/S keys for rotation, PageUp/PageDown keys for speed adjustment): Javascript functions were written to handle these events; values are stored using <Input> tags with the hidden style attribute, modified and sent by these functions using a POST.
- Information Display: Using curly brackets for code snippets and JS functions for on-event element modification, the web application is designed to show the values that are in the database, both upon visiting the web application and on changing said values.

For the above mentioned functions, requests were made via the HTTP protocol through XMLHttpRequest() to the locally hosted server to perform appropriate functions, most commonly to store corresponding data to the database. Initially, the page was rerouted back to the index page after calling a function in Python. However, we eliminated the need for rerouting by implementing asynchronous actions using XHTTP, which allowed us to affect the database without having to POST a form element.

Computer Vision

In order for the robot to pick up objects and interact with the environment, it must be able to detect objects and feature spaces. We have a couple of different objects to detect this time around (different types of soda cans). Initially, we decided to try doing object detection in a way in which it can be used to detect anything. The first object we tried to detect was chilitags. These tags are similar to enlarged barcodes and can be detected very quickly and efficiently. We started with these because they are very useful for a particle filter that we will be implementing for our final demo. The chilitags library is very scalable solution in our situation. You can print out a chilitag and place it on any object and let the robot know what the chilitag represents. Once we found a solution we decided to look for better ways to detect objects. Next, we tried various libraries in matlab and opencv and different algorithms to detect objects and people (brute force, SVM, haar cascades). In matlab, we used SVM, by finding the surf(Speeded-Up Robust Features) features of an object(coke can) and comparing those with each frame from the camera feed. This worked very well for comparing between images but there were problems while updating frames in the webcam feed. Next, I tried brute force algorithms

in python by getting critical points of a soda can and comparing with an image with multiple soda cans. This didn't work really well because a lot of critical points matched up with other soda cans as well. Next, I tried making haar cascades to determine objects. I downloaded 2,000 negative images, superimposed a positive image on all those negative images, and then ran the opencv cascade training command. The resulting cascade file wasn't as good as expected, but the method with haar cascades was very good for detecting facial features. Next, I found an open source robotics computer vision library. I downloaded the necessary dependencies and added various objects to the testing application to detect all the objects I needed. The application uses SVM. The application finds the SURF features of the object you choose and compares them with the SURF features of each frame to detect the object. This is very similar to the code I tested in matlab, but it was a lot faster. Since, there was a 1 second delay, I am trying to make it faster by writing my own c++ application that can detect a limited number of objects in a more efficient way. Currently, the application can detect the surf features on an object and compare them with other features on other images to detect objects. I am also testing neural networks (CAFFE), higher quality haarcascades (more testing phases in training) and more efficient SVM algorithms to come up with the most efficient and scalable way to detect objects.