Alpha Test Plan:

Mission Control: Rouleau, Luke; Shi, Xuanhao

Computer Engineering Senior Design

University of Florida, Department of Electrical and Computer Engineering

Gainesville, Florida

Introduction

Team *Mission Control's* project is tough to classify. Since we have no prior robotics experience, our project feels like a full-blown personal research project, as well as a technical attempt to build a competition-ready deliverable. We also do not function alone in a vacuum: since we represent the software element of the IEEE Hardware Design Team, we are subject to delays and redesigns from our team. This has affected our ability to develop tests that can be deployable to the robotic hardware since there is no assembled robotic hardware yet. The robot's platform arrived from a fabrication facility today, and the arm will be fully assembled soon; however, as far as the alpha build deadline goes, that is too late. Thus, we ask for understanding in our adaptation of this unique project to the more generic specifications laid out by the test plan document.

Expected Behavior

We will build the test plan from the bottom up:

- **First**, we need to ensure hardware communication. Starting from the Jetson, upon robot boot-up, we must check that the cameras are feeding in input via USB. Then we need to verify via handshake that the Teensy microcontroller can communicate coherently with the Jetson.
    - We expect that all hardware components can properly communicate to the Nvidia Jetson via USB connections
- **Second,** we need to guarantee that the Bobert's ROS node's for controlling wheel movement is operational and tuned.

  - We expect the robot to move an accurate distance in real life based on the command issued in ROS
- **Third**, we need to guarantee that Bobert's ROS nodes are properly instructing the arm.
  - We expect that, after performing the inverse kinematics and path planning, that the robot's hardware interface for the arm servos can instruct the servos to move to the correct final position.
- **Fourth**, we need to verify that the robot can identify target items in its field of vision.
  - We expect the robot to be able to identify target items in its field of vision with an accuracy of 97% or greater.
- **Fifth**, we need to be able to navigate towards the item
  - Assuming that the target has been correctly identified and located, we expect the robot to be able to plan a path to the item that avoids obstacles and stays on the road.

Present Test Procedures (visible under the "Testing" heading on the team repo)

Testing the *bobert_control* package

- Currently, the test is embedded inside the src files for the bobert_control package.
- To launch the project with tests, do the following:
  - source the workspace with `source ./devel/setup.bash`
  - launch the main hardware interface

    `roslaunch ./src/bobert_control/launch/bobert_HW_main.launch`
- The main hardware interface will load the arm into the RVIZ and then subscribe to the `/teensy/bobertTelemetry` topic and publish the current arm angles to `/teensy/armCmd`

- This interface will also load the testing node within `./src/bobert_control/src/bobert_sim_echo.cpp`
- This testing node will simulate a node that works like a teensy controller.
- This testing node will subscribe to the `/teensy/armCmd` topic and echo the angle positions to `/teensy/bobertTelemetry`
- And thus we can use `rostopic info` to see the message about the joint controller between the hardware interface node and the teensy node.
- As such, inside RVIZ, we can add whatever planning group we want to modify on the arm, and then plan the path to communicate the information through the terminal in the same format as the `./src/bobert_control/msg/`
- The expected testing results from the `/teensy/armCmd` or `/teensy/bobertTelemetry` topic would be a set of six angle messages in the format of
  - `angle: [-68.1826..., 23.0462..., 127.3962..., -167.1223..., 5.9455..., 88.7630...]`
- Every time we plan and execute a specific path on a planning group in RVIZ, the terminal message for the six values will change accordingly. And we can thus see if these messages match our path roughly.
- In the future, we can add more complex msg structures and inputs from the simulated teensy node to be sent to the main hardware interface.

<u>**Testing the *RealSense* camera drivers**</u>

- On a Jetson Nano with the Intel RealSense SDK installed, run `realsense-viewer`

  - If prompted to update the drivers on the cameras, do so

- Turn on the camera feed toggle for the D435 and T265 cameras on the left-hand menu of the realsense-viewer application
  - If you can see the camera streams, the camera drivers have been properly installed.

<u>Testing the *jetson_dev* environment</u>

- Navigate to the catkin workspace inside of `OS_Melodic_Implementation/jetson_dev/catkin_ws`
- Source the workspace with `source ./devel/setup.bash`
- Start roscore with `roscore`
- Start the camera node launch files with `roslaunch realsense2_camera rs_d400_and_t265.launch`
- Open the ROS visualization GUI by `rqt` in another terminal
- Navigate to *Introspection > Node Graph* from the GUI dropdown menu
- Verfiy that the node graph looks like the "ROS Perception Interface" under the “Usability” heading in the team repo.
- If it passes visible verification, then run `roslaunch realsense2_camera rs_rtabmap.launch` to perform 3D slam mapping, visible in Rviz.
- Verify, by moving your RealSense camera array around your surroundings, that the SLAM functionality appears to be working.
  - If so, you can definitively say image data is being passed through ROS messages properly.

Future Test Procedures

1. Ensuring hardware communication:
    a. **Automated Unit Test**: Once the robot components are connected on the robotic platform, we will test that hardware components are functional by running a system-verification routine upon power up. This routine will run the main ROS launch file to start up all the required nodes, then check that the camera feed topics are feeding information from the cameras, then send a string-message type like "test" via to a test node, wait for the Teensy to register the message, and respond "test heard" (or "test failed" if no message received). If the Jetson can recognize the camera feed and communicate to and from the Teensy, then the communication protocols are functional.
2. Testing and tuning ROS node's for controlling wheel movement:
    a. **Teleoperation Test**: To verify the proper translation of velocities on the cmd_vel topic to servo movement, we can perform a teleoperation test. We have composed a teleoperation keyboard controller node that can emulate the messages that will be passed from the (SLAM and planning) Node Network onto the cmd_vel node. Once the robot is assembled, we can use this tool to measure how far the robot moves in reality *when it thinks it moves one meter (i.e., we issue a teleoperation command to move forward one meter)*. By measuring the error, we can tune our custom hardware interface to make sure that we scale the commands properly for the servos.

3. Testing and tuning Bobert's arm movement:
    a. **Simulation Verification:** At present we are working on developing a URDF so we can simulate the arm moment. Once we have a proper model, we can test the hardware interface for the arm via simulation. This is done using MoveIt and we have composed a test echo node that acts as the Teensy board to receive messages as we simulate. Once we have an assembled robot platform, we will create an automatic test routine that will routine that moves the arm through its full range of motion.
4. Verify that the robot can identify target items in its field of vision:
    a. **Classification Test:** We need to be able to test the model we develop to identify target items via transfer learning. The models provided by Nvidia are pre-trained, so once we transfer-train on our target items, we need to determine a threshold of classification accuracy. We aim for an accuracy of >97% and we will test this using a large data set of images collected inside of the training course.
5. Navigation towards an identified item:
    a. **SLAM Testing:** Once we know we can identify target items with sufficient accuracy, then we need to make sure Bobert can navigate to within operable range of the target. This is done via SLAM and path planning. We will use Rviz to verify that Bobert's mapping of the course is functional and can effectively navigate towards a target location.

6. Error Logging:

    a. **ROS Logs**: ROS handles logging automatically, making debugging generally simple. For each ROS session, a log file is recorded that we can parse through in the case of an error. We can easily see the origin of an error, the type of error, and when the error occurred (in the middle of which routine?) from the log files. These will serve as a major debugging tool through the next design phase.