MIRO Robot Manual for Scheduler

Written by Ellie Sona in April 2019

# Introduction

This manual explains the scheduler code for the MIRO robot. The scheduler receives input from the user about where in the room it wants the robot to end up and in what position and uses a webcam to get the current position and orientation of the robot. It then sends a direction to the robot to make it move from its current position and orientation to the final position and orientation.

# Robot Information

The RASL lab has 2 robots – one has a collar and one does not. When connected to the vuDevice network, the robot with the collar has an IP address of 10.68.1.92, and the robot without the collar has an IP address of 10.68.1.94. the vuDevice password used with both robots is pbPv)-X3aGOMB. The MiRo app should be used to connect to Wi-Fi, though the robots will automatically connect to the vuDevice network. With the new, reflashed SD cards (currently installed on the robot without a collar), the root password for the robot is raslMIRO. With the old SD cards (currently installed on the robot with a collar), the root password is !amMIRO which is the default password for the robots. With both, root should be used as the SSH username. The robot with a collar is currently setup to run with the scheduler code.

# Preparation

The workstation, robot, and room must be prepared properly to run the scheduler program.

### Workstation

The workstation computer must have Python 2.7 installed. OpenCV, Paramiko, and SciPy must be installed using pip. The files from <https://github.com/ellieasona/MIRO/tree/master/Scheduler> must be downloaded into the same folder. Finally, you must have a webcam connected and available in USB position 1, though this can be easily changed in CalculatePosAndDir.py and CameraCalibration.py. For example, to change to using the default computer webcam, you would use camera = cv2.VideoCapture(0) instead of camera = cv2.VideoCapture(1)in both those files. Using the code that is currently commented directly above those lines, you can set the programs to run using an image that was already taken rather than capturing the current webcam frame.

### Robot

The robot must be connected to Wi-Fi, and it must be set to run code on-board by changing that in the ~/.profile file. You should not run code automatically when bridge\_control.sh toggle\_running is called as you would when running code automatically on-board. You do need to run roscore automatically on start up by putting roscore at the end of your ~/bin/start\_user.sh file. An example of this is included at <https://github.com/ellieasona/MIRO/tree/master/on%20bot>. All the other files at that link must be loaded onto the robot at the /home/root directory.

The robot also needs to have colored dots placed around its base like shown in the images below. If the colors of the dots are changed, the color ranges must be updated in getlocation.py. This can be aided using OpenCV’s range detector functionality to get the HSV ranges. Be sure to run range detector with dots at many different locations around to room to make sure they can be detected with the robot at any position. They also must be large enough that the camera can pick them up even when the robot is in the back of the room.

 

### Room

The room must have a table or other way to place a webcam above ground level. The higher the webcam, the more accurate the program will be, though it is important that the image has the features needed for CameraCalibration.py as described below. The portion of the room visible from the webcam must be empty except for the robot. Finally, the room cannot have anything in it that is the same color as the dots on the robot.

# Code Structure

To code used to run the scheduler program is described in the sections below.

### Workstation

There are several files needed on your workstation to successfully run the scheduler. They are all available at <https://github.com/ellieasona/MIRO/tree/master/Scheduler>.

1. CameraCalibration.py – This program must be run to calibrate the webcam when you move the camera or move it to a different room. It requires the user to click on several points on the image as directed by the program: the top left corner of the floor, the top right corner of the floor, one foot from the top and left of the wall, where the left side of the wall ends in the photo, where the right side of the wall ends in the photo, and one foot from the front on the frame in the center. As an example, these points are shown in the lab room in the image below. This code only has to be run once when the camera is placed and stores the configuration in the file calibration.txt. A sample calibration file is also available at the link above.



1. MoveToLocation.py – This is the program that is run to actually start the scheduler. It calls CalculatePosAndDir.py to get the current x and y coordinates of the robot along with its orientation. It then prompts the user for their desired final x, y, and rotation of the robot. The x and y are measured in meters from the top left corner of the room and are always positive. The orientation is measured in degrees according to the unit circle. With that information, it calculates an initial rotation needed to get the robot in the correct orientation to drive straight and end up at the correct point, calculates the distance the robot needs to drive straight, and calculates the final rotation needed to get the robot in its correct final orientation. It then prompts the user for the username, password, and IP of the robot they want to send the command to and connects to that robot using Paramiko. Once connected, it sends commands over Paramiko to run the bridge, run the start\_user.sh file on that robot with the proper parameters for initial angle, distance, and final angle, and stops the bridge.
2. CalculatePosAndDir.py – This file is called by MoveToLocation.py that gets the current x, y, and rotation of the robot. It first calls getlocation.py to get the pixel locations of any of the colored dots on the bot visible to the webcam. It gives those pixel locations to CalculatePointLocations.py to get the real-world x and y of each of the dots. With that information, it uses the known orientation of the dots and the angles between them to calculate the position and orientation of the robot in meters and degrees.
3. getlocation.py – This file is called by CalculatePosAndDir.py to get the pixel locations of the colored dots on the robot that are visible to the robot. It uses OpenCV to take a picture using the webcam, looks for places in the image that are in the color range for each of the dots, makes a mask of those places that are in range, finds the contour of each of those regions, and finds the center of each contour. A list of points of the center of the contours is returned from this.
4. CalculatePointLocations.py – This file is called by CalculatePosAndDir.py to convert the pixel locations of the dots from getlocation.py to real world locations. It begins by loading the information from calibration.txt in and performing calculations to get the pixel locations of various points in the image. It then does a series of calculations converting pixel distance to real-world distance taking into effect camera distortion and limited view. Because the camera calibration happens in feet, it then converts the feet to meters, and returns an ordered pair of the actual x and y of the dot in meters.
5. geterror.py – This file can be used to help calculate the difference between the instructed rotation and distance sent to the robot and the actual rotation and distance moved by the robot to help eliminate error. Because of wheel slip, the robot moves less than instructed, so this program instructs the robot to move x amount allowing the user to measure how far it actually moved to compare. Currently, it instructs it to move straight 1.5 to 6 feet incrementing by .5 each time, but this could easily be changed for a different sized room or to be in degrees (be sure to convert degrees to radians when passing the command, though). Note that you should convert the feet to meters in your plot to get the correct intercept. The actual vs. instructed amounts can be plotted to get a trendline that can replace the current equations in MoveToLocation.py. The plots used for the current equations are below.

### Robot

Four files are needed in the /home/root folder of the robot to successfully run the scheduler. They are all available at <https://github.com/ellieasona/MIRO/tree/master/on%20bot>. SchedulerInitiator.py, SchedulerHandler.py, and interfaces.py operate very similarly (and in the case of interfaces.py, identically) to those used to run the default behavior described in the operating manual located at <https://github.com/ellieasona/RASL-MIRO_Sensor_React/blob/master/README.pdf>.

1. start\_user.sh – This file is run using Paramiko’s exec\_command. It accepts initial rotation in radians, distance in degrees, and final rotation in radians as parameters. First, it sources the proper files and sets the proper environment variables to run python code. Then, it runs SchedulerInitiator.py with the same parameters.
2. SchedulerInitiator.py – This file is called by start\_user.sh and works the same way as the Initiator.py file used in the default behavior except with the 3 parameters. It creates a rospy node for the robot and calls SchedulerHandler.py.
3. SchedulerHandler.py – This file is called by SchedulerInitiator.py. It uses interfaces.py to instruct the robot to rotate the initial amount, move the designated amount, and rotate the final amount. Because there is no instruction option to move or robot a certain amount, it instructs it to rotate/move at a certain rate and sleeps for the correct amount of time to make it rotate/move the correct amount.
4. interfaces.py – This file is used by SchedulerHandler.py to give the robot instructions more easily. It is identical to the file used for the default behavior and a full description is in the user manual linked above.

You must also edit the user\_ready.sh as indicated in the preparation section, so a proper user\_ready.sh file is available in the folder linked above.

# Running the Scheduler

Once your workstation, robot, and room are prepared, run the code by simply opening a terminal in the directory where your code is located and running the command python MoveToLocation.py.

# Conclusion

This program allows the robot to move to a designated point and orientation autonomously. Moving forward, the program could be improved by expanding it to be more practically functional and by reducing the error in image processing. A table of errors with the robot in different positions and orientations is below. The bolded error is an especially difficult situation where when the robot is on the side of the room, only one dot may be visible to the camera despite the robot being positioned at a 45-degree angle. Email Ellie Sona at [ellie.a.sona@vanderbilt.edu](mailto:ellie.a.sona@vanderbilt.edu) with any questions about this program.

|  |  |  |
| --- | --- | --- |
| Actual Position:  x (ft), y (ft), r (degrees) | Calculated Position:  x (ft), y (ft), r (degrees) | Difference between actual and calculated:  x (ft), y (ft), r (degrees) |
| 1, 1, 90° | 0.8, 0.7, 65° | 0.2, 0.3, 25° |
| 1, 1, -90° | 0.7, 1, -90° | 0.3, 0, 0° |
| 1, 1, -45° | 1, 1, -35° | 0, 0, 10° |
| 3, 1, 0° | 3, 0.7, 0° | 0, 0.3, 0° |
| 3, 1, 135° | 3, 0.9, 160° | 0, 0.1, 25° |
| 5, 1, -90° | 5, 0.7, -90° | 0, 0.3, 0° |
| 5, 1, 20° | 5, 0.6, 10° | 0, 0.4, 10° |
| 1, 3, 30° | 0.6, 2.5, 25° | 0.4, 0.5, 5° |
| 1, 3, 170° | 1.1, 2.6, 180° | 0.1, 0.4, 10° |
| 3, 3, -45° | 3, 3, -45° | 0, 0, 0° |
| 3, 3, -135° | 2.5, 3, -145° | 0.5, 0, 10° |
| 3, 5, 30° | 2.8, 5, 27° | 0.2, 0, 3° |
| 5, 3, 130° | 5.2, 2.5, 140° | 0.2, 0.5, 10° |
| **5, 5, -135°** | **4.6, 4.7, -90°** | **0.4, 0.3, 45°** |