CSCE 274 Section 1 Fall 2016 – Project 02 – Boyd Compton, Jose Tadeo, Timothy Senn

Project Description:

The random walk program is designed to divide the major task into different threads. For this program, there were three major tasks which were the high level controller, the actuator controller, and the reading of sensor data. Of these three tasks, both controllers are found in main.py while the other is located in sensor_inf.py. This program structure was mainly adopted to modularize the program's codes for future projects or applications. With the chosen structure of the project, the section of code responsible for some bug or unwanted behavior is easier to identify. For example, if the robot rotated left on the left bumper press, then we can identify that the bug is mostly likely within the actuator controller's code.

The high level controller's goals are to set the state of the robot, control the state of all other threads, initialize shared resources, and maintain a handle to the shared resources. In this context, shared resources are any resource used in multiple threads such as the robot's connection, the log file or, the handle for the sensor interface. On top of its goals, the high level controller will always be the first thread spawned, and the last thread to be stopped. Since this thread maintains all other threads, the most noticeable action preformed in this section is the response to the clean button press. When such an event happens, the high level controller will spawn or stop the random walk's actuator controller as well as logging the datum 'BUTTON'.

The actuator controller's goal is to perform the random walk. The exact specifications for the random walk algorithm are as follows (clockwise = CW, counter-clockwise = CCW):

Movement	**Condition**
Stop all movement	If either wheel drop is down
Randomly rotate either CW or CCW	If both conditions for rotate CW and rotate CCW are true.
Rotate CW	If the left bump is down or any of the left cliff sensors detect a cliff
Rotate CCW	If the right bump is down or any of the right cliff sensors detect a cliff
Drive Forward	All other conditions are false.

On top of these movement conditions, the actuator controller is responsible for logging the robot's movements and any unsafe events such as a cliff sensor detecting a cliff or a wheel in a dropped state. The datum's format for each event is as follows:

Event	**Datum Format**
Forward movement	<*distance*> mm
Rotation (either CW or CCW)	<*angle in degrees*> deg
Unsafe event	UNSAFE <*event*>

With an exception for the forward movement, only one log statement will be generated per action. The forward movement statement will be generated on an interval whose maximum length is the maximum wait time for a rotation. Such an exception was made to eliminate the possibility of an encoder turning over multiple times during a single drive forward. Additionally, each wait interval is subdivided into discrete smaller intervals where the rotate and stop conditions are checked each iteration. To avoid adding noticeable additional time to the specified wait interval, the execution time of the code block was also factored in when updating the amount of remaining wait time.

The reading of sensor data happens in a thread spawned by an instance of Sensor class found within the sensor_inf.py. Any sensor data requested by the other two sections will essentially read the specific data member(s) within the Sensor instance via a getter. All the locking mechanism required to prevent race conditions are within the getters and sensor updater thread. At first, a simple locking mechanism was chosen, however, we later identified that this interfaces structure should only lock getters when updating the internal variables. So, a reader-writer lock replaced the simple locking mechanism to prevent race conditions for the internal instance variables. Furthermore, this interface increases readability of the main program by simplifying the detection of more complex events such as a button's initial press or release. This interface was developed to the extent required to complete this project, however, the depth of this interface can be easily be modified to accomplish a desired purpose without altering the overlaying structure of this interface.

Finally, the robot path plotter was created to reconstruct the robot's path based solely on the generated log file. To do so, this program begins by parsing the selected log file. While parsing the file, the state of the robot will be updated by applying forward kinematics to the robot's position and rotation. During reconstruction, the supplied log file will additionally be divided into different runs where the robot's position is known. If the robot's position was to ever become unknown, then the new run would begin with the initial point at the origin of the plot. Furthermore, each run parsed from the log file will all be displayed on the opened plot window with differing colors.

Another additional design decision was to check for turnover per encoder before calculating the difference between the two encoder values. This was mainly done because the different cases were not easily identifiable. This also made a universal turnover detection method that could be used for both distance and angle calculations. Additionally, we wanted to avoid locking mechanisms scattered all around our threaded environment. Thus, we wanted to place the locking mechanism in a logical place. For example, in serial_inf.py, contains the locks for reading from a serial connection and writing to a serial connection. This decision helped keep an understandable race free code

Project Evaluation:

Both the random walk program and the path plotter program constantly fulfill their necessary requirement, so these programs work reasonably well. Through all our testing, we only noticed a few minor problems. First, the robot may not respond to an extremely short button press. The issue's most probable source is the wait time added from the locking mechanism for the sensor interface's internal variable updater. The amount of time before execution can differ based on the order and number of received read requests. This additional wait time can in turn slightly alter the sampling speed of the internal variable updater. Another rare minor problem is the fact that the calculated angle will return an astatically huge angle. We have not been able to pinpoint the exact events that result with such a return due to how rarely it occurs. However, we believe it is due to one or both encoders turning over. Since working on this project, we have only seen the angle problem twice. Finally, the robot's wheel sometimes drop off the cliff while trying to rotate itself to a safe direction. The robot has to be approaching the cliff head on for this event to even have a possibility of occurring. At this time, we believe that this event is caused by the robot getting too close to the ledge before it begins to rotate away from the ledge.

To evaluate other calculations such as distance and angle, we would solely test the calculation in a controlled environment where we already knew the answer. For example, when testing the distance, we would measure the distance between points with a ruler and then compare it to the returned calculation. This process was then repeated multiple times to ensure it was not a coincidence. A similar process repeated for angle and the different conditions in the random walk algorithm. For instance, when testing the turning conditions for the random walk algorithm, the forward movement was disabled by setting its velocity to 0. After this point, we would trigger a rotation by either tapping the bumper or activating a cliff sensor. We also determined that the wheel velocity of 100 mm/sec gave the robot an ample amount of time to respond to stimuli for most situations. There still exist outlaying situations where the robot may not respond quickly enough like the wheel dropping off the cliff during rotation. If you wanted to further reduce the amount of outlaying situations, you could reduce the wheel velocity more. However, we found that 100 mm/sec was a good balance between speed and response time.

Allocation of Effort:

Boyd Compton:

- Augmented the project 1's interface
- Wrote the sensor interface and project write up
- Created the initial code and structure for each file.
- Helped finish the random walk's controller.
- Helped finish the path plotter's code
- Helped test the robot's functionality.
- Aided in identifying critical sections of code
- Helped convert the project to a threaded environment by implementing the necessary changes

Jose Tadeo:

- Helped write the random walk's controller logic
- Helped write the path plotter's code
- Added the path plotter's entry to the README.txt
- Performed tests to determine the robot's functionality.

Timothy Senn:

- Helped identify potential race conditions and critical sections
- Aided in converting the project to a threaded environment by looking for missed race conditions and critical sections
- Helped maintain an understandable and clear code structure for each file.
- Ensured the project's code remained well clear and well documented
- Wrote the README.txt