The University of Melbourne
Department of Electrical Engineering

ELEN90066 - Workshop Report

Embedded control for MyRio Robot

Group 6 - Workshop 1 (Monday 10:00am)

Tuan Khoi Nguyen (1025294)
Quoc Bao Pham (1086263)

Contents

Stage 1

Introduction

This report presents the process of designing an algorithm used in an embedded system. Specifically, the algorithm is used to control the Kobuki robot so that it can navigate through the obstacle while keeping the trajectory toward a predetermined direction, as well as it must be able to detect the terrain's tilting angle (being on the hill) and keep the robot being straight with the incline. The system's hardware components include a myRio controller, which has a dual-core ARM Cortex-A9 process and an onboard accelerometer and wifi hotspot; a Kobuki robot which is an autonomous mobile robot with bump sensors, wheel-drop sensors, wheel speed sensors (encoder), and cliff sensors installed in it. The algorithm can be developed on the LabVIEW (or Eclipse) platform and embedded into the myRio controller via wifi protocol. The work in this report includes the project's background, the strategies of the controlling algorithm, and the final assessment of the algorithm's capabilities to accomplish the task requirements. Lastly, the report would bring out suggestions for future work.

1.1 Background & Devices

Cyber Physical Systems - the combination between programming and hardware [1], has becoming more common in modern autonomous applications. Autonomous robots are intelligent machines that can accomplish determined tasks in an environment independently, without support of human control. They are capable of helping workers to perform their delegated hazardous occupations. Such examples include Roomba by the company iRobot, which has a set of sensors that allow it to navigate in-house to clean the floor, by moving across the area while avoiding furniture & walls.

(a) Component annotations of the Kobuki from LabVIEW simulation (b) Kobuki simulation in group's visualisation

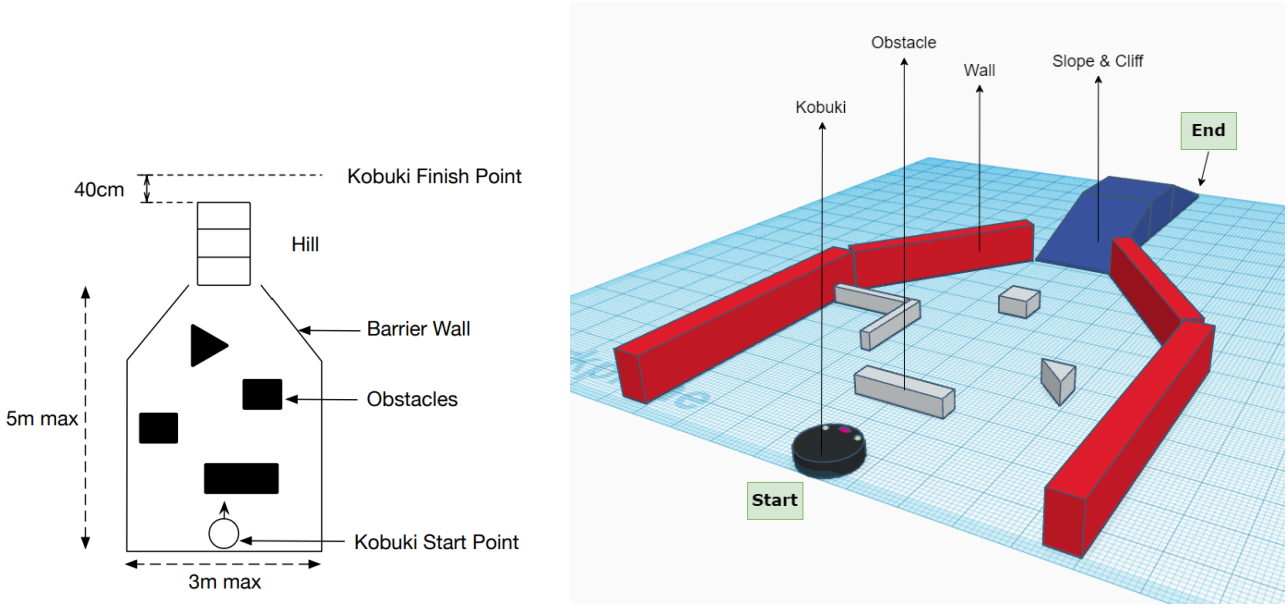
Figure 1.1: The Kobuki model that will be used in this Course Project

Kobuki is a disk-shaped mobile robot designed for research and education at a low cost. The onboard sensors of a Kobuki robot are pretty similar to those of a Roomba robot; hence it can perform the navigation tasks the same as the Roomba. As mentioned above, a Kobuki robot is coupled with a myRio processor in this course. The built-in sensors on myRio and Kobuki enable the user to accomplish all the moving of a Roomba without adding any sensor to the system. The goal of this project is to introduce the learners to embedded programming with a practical example in which the learners would have the chance to apply the knowledge in the lectures in a hands-on project. Mainly the project focused on designing algorithms based on the ground from lectures rather than trial-and-error.

MyRio is a processor used for education and research purpose which is developed by National Instrument. With multiple port, it is compatible to a wide range of devices for mechatronics applications. In this project, the myRio is connected with the Kobuki robot via a DB25 port through a screw-terminal connector. The user can get access to the data of sensors on the Kobuki and also the sensor modules of the myRio processor.

1.2 Requirements

In the Course Project, a system will be designed, which will control the Kobuki system (Figure 1.1) to navigate through an unknown course, where an example is shown in Figure 1.2b. While navigating, it has to successfully avoid obstacles or cliffs, and go through a hill terrain.



(a) Layout of a sample project course

(b) 3D simulation of a sample project course

Figure 1.2: The Kobuki course type that will be used in this Project

From the project description and the course information, the requirements are summarized in Table 1.1 as follows:

Name	Objective
Startup & Run	Kobuki moves if and only if 'Play' button is pushed
Obstacle Avoidance	Is able to avoid objects on the path, without hugging them
Cliff Avoidance	Is able to avoid falling off edges or wheels losing ground contact
Hill Climb	Oriented to go towards ascending direction, not encountering any edges
Downhill Stop	Must stop within 40cm after finish descending
Timeliness	Reaches the end point within a specified time
Performance	No chattering, no rotation of over 180^o, no sudden termination

Table 1.1: Requirements needed in this project

As the exact course layout is not specified, the system design will need to reach a specific level of robustness, in order to satisfy all requirements in different courses. Therefore, careful testing must be carried out in order to detect edge cases, as well as measuring the versatility of the system design in different terrains.

However, it is given that the course will contains 2 parts like in Figure 1.2b: the ground (section covered with red walls) and the hill (blue section at the end). The hill will not have obstacles, and the ground, being flat, will have obstacles randomly distributed in the area. This will be able to summarize down the system needs to:

- Successfully avoid all obstacles on the ground.
- Be able to go up the hill, avoiding cliffs, then descend and stops.

Stage 2

Control & Modelling Framework

2.1 Finite State Machines

A finite-state machine (FSM) is a model of discrete dynamics in which the valuations of the input are mapped to the valuations of the outputs in each reaction with a finite number of states. The behavior of the Kobuki, according to task requirements, is compatible with a hierarchical state machine. There are two main types of FSM: Moore and Mealy. Moore FSM is the model where the output of the state machine is purely dependent on the state variables, and Mealy FSM is the model where the output can depend on the current state variable values and the input values. In this project, a Mealy FSM is utilized since it reduces the number of necessary states to a minimum, and it is also feasible to perform any adjustment (simply adding more transitions with respect to the change). Also, the Mealy machine is convenient to be able to produce an output that instantaneously responds to the input [1].

2.2 Tools and Utilities

A number of software tools were chosen for our main process development, based on how quick they are for continuous integration and deployment, as well as how efficient the choices and steps can work with each other.

Programming: LabVIEW 2018 In this project, the programming for Kobuki can be done on either LabVIEW, which uses Statecharts to represent the algorithms, or Eclipse, which uses C code to describe the FSM behaviors. Having tested on both methods, we decided to choose LabVIEW for developing our main algorithm. First, LabVIEW's Statecharts have a compact representation that captures all state hierarchies, refinements and guards within a small window, instead of having to specify lengthy code lines in C. Second, testing with available myRio devices shows that through many runs, the majority of them have their network misconfigured, hence unable to deploy and build the C code on. Moreover, the debugging tool of LabVIEW can assist in detecting inappropriate guards, transitions, or abnormal states. The user interface of LabVIEW allows the developer to implement the algorithm visually, according to its graphical nature. Additionally, the LabVIEW platform can communicate with the myRio processor through a wifi connection while the Kobuki is executing the task and display the data optically on the PC.

Simulation: CyberSim We chose to use CyberSim, not only because this was the main simulator taught in the workshop demonstrations, but also the fact that it can compile with `.xml` file format, which can be modified using LabVIEW's graphical user interface. Through easy modification, it will be more convenient to configure the simulated course, such as adding obstacles, or form dead-ends, etc.

2.3 Sensor Components & Allocated Variables

While there are a large variety of components in Kobuki, only a few among them were chosen to retrieve input from, given that one of our aims for the state machine is to be compact for the simplification of debugging process. This section lists out the sensors where their input is essential to the algorithms, and how those inputs are named.

Bump sensor The bump sensors are located at the left, right and center of the front side of the robot, right under the edge surfaces. They are simply switches (bump switches) with a default value of `state` variable as 0 (8-bit integer). If the robot hits an obstacle at any of these positions, the switch at that one would be pressed, and the digital signal of the `state` variable of the sensor would be turned to 1. Besides that, the kobuki also generated a message whenever a particular bumper was pressed or released, with 0, 1, and 2 (for `bumper` variable) corresponding to left, center and right sensors being triggered. In that way, the state of each sensor is informed to the processor. For example, if the left sensor is pressed, the message would be `bumper = 0` and `state = 1`. The bump sensors is useful for detecting the collision with obstacles, after then the robot will go slightly back then rotate to another direction.

Cliff sensor The cliff sensors apply the infra-red light to measure the distance between the robot base and the floor by emitting a wave pulse and receiving the reflecting pulse, then calculating the distance from the light traveling time. There are 3 cliff sensors on the kobuki, which are located at the left, center and right of the robot's bottom base. Similar to bump sensors, a message would be generated whenever a particular cliff sensor signals, indicating the kobuki approached or moved away from a cliff. There is a predetermined variable used for setting the vacuum detecting, which is the `bottom` variable (16-bit integer); if the distance between the sensor and floor is larger than the decided range, the system will assign it as a cliff. The message is the combination of two variables (`sensor` & `state`). The `sensor` variable will have a value of 0, 1, 2 with respect to the left, center, right sensors, while the `state` variable will have a value of 0, 1 corresponding to floor & cliff. If the robot encounters a cliff on the left hand, the message would be `sensor = 0` and `state = 1`.

Odometer The odometer is a combination of gyroscope & encoders attached to kobuki's wheel. The encoder would record the ticks while the wheel rotates and accumulate them to get the result of the traveled distance as well as rotated angle. However, the drifting error of rotated angle can accumulate by this method; thus, to lower the error accumulating through time, a gyroscope is implemented.

A gyroscope is used to measure angular velocity of an object. With gyroscope in action, the result of the rotated angle is much more reliable since even if the robot slips during rotation (the encoder would not record this motion since the wheel did not rotate), the gyroscope can still capture this motion. Even though without any reference coordinate system (usually a magnetometer is used for this purpose), the error can still accumulate; therefore, in some experiences, the robot moves slightly out of the direction determined from the start even though the recorded rotated angle of the system is still 0. To access the data, the value `Input.distance` and `Input.angle` are two variables recorded the traveled distance and rotated angle of the robot in LabVIEW environment.

Accelerometer The accelerometer is an on-board sensor of the myRio processor, which can measure the acceleration on the axes of the sensor as shown in Figure 2.1. The sensor data in 3 axes can be accessed in LabVIEW environment from the variables `Input.acc.x`, `Input.acc.y`, `Input.acc.z`. These data are mainly used to detect incline of the terrain, for example the guard used to detect hill presence is: $|accX| + |accY| > 0.24$. The value 0.24 is determined based on the measurement of the inclination of the ramp used in testing course, which will be explained in the next stages. Additionally, the left side of the equation is the sum of acceleration on two horizontal axes (with respect to the body of kobuki) since the robot may not start going onto the ramp with the direction towards the incline.

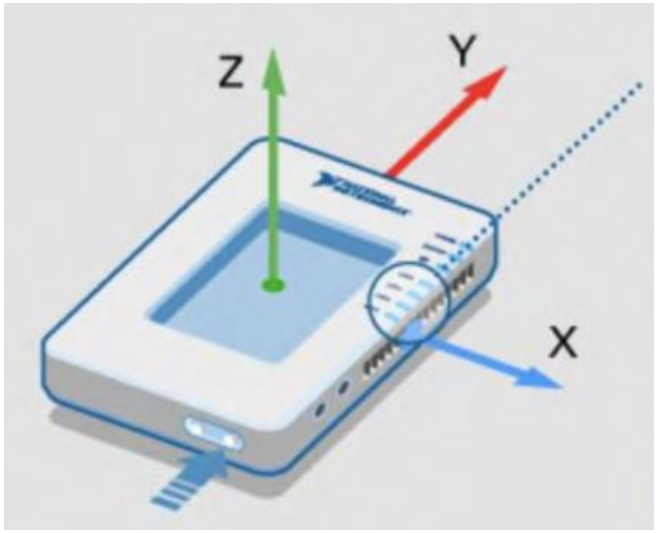


Figure 2.1: The accelerometer axes of the myRio

Stage 3

System Design & Development

Given that the control is based on state machines, it is essential to ensure everything works on a case-by-case basis, which covers as most scenarios as possible. Overall, the implementation will attempt to cover obstacle bumps, edge cliffs and dead-ends, which are common cases that Kobuki may encounter on the course. Note that this stage will assume the x-axis being kobuki's forward direction, in conjunction with real life hardware placement.

3.1 Signal processing

This section deals with signal uncertainties found during in-person testing, due to sensors' and actuators' limitation.

3.1.1 Multi-signal handling

When coming to contact with an obstacle, there might be cases where both the front bump sensor and any side bump sensor will send a signal simultaneously, just like Figure 3.1. This may cause wrong actions to be taken, if each guard only relies on one bump sensor state. Therefore, handling will need to be done for cases where multiple sensor signals are detected.

Our implementation handles this, by setting priority order for which sensor signal to follow first: left-right-center. For example, if left bump sensor and right bump sensor signals, then Kobuki will turn right to the left's signal, or if center and right sensor signals, then it will turn left according to the right's signal.

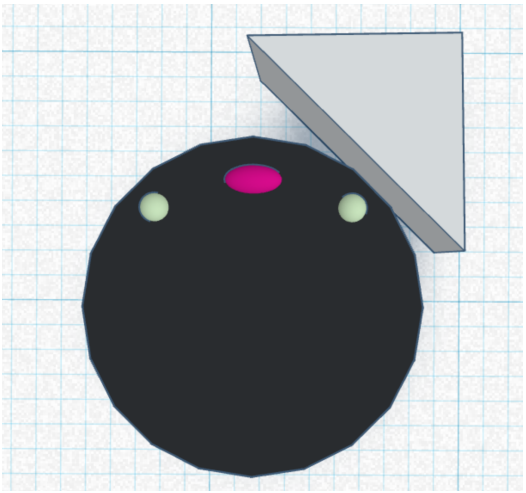


Figure 3.1: A situation where front bump and right bump sensor will signal simultaneously

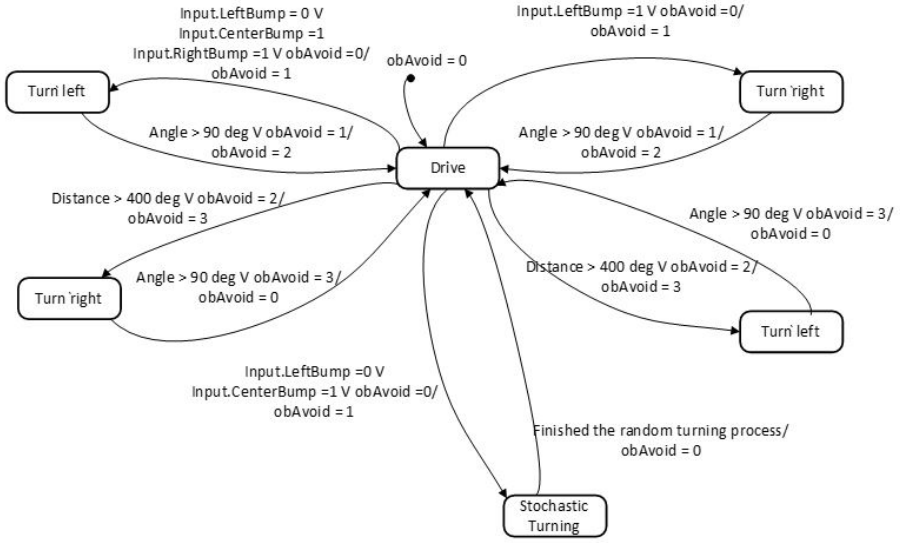


Figure 3.2: Detailed FSM of priority setup for multi-signal handling process

3.1.2 Low-pass filter

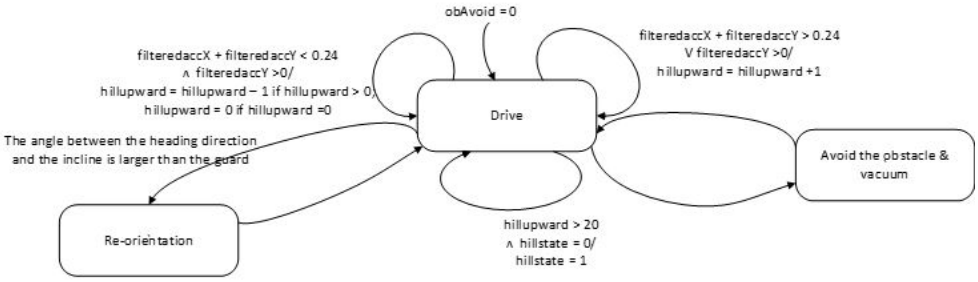
We noted that during the operation, the accelerometer signal is affected by white noise continuously and sometimes spike due to bump or reorientation. Therefore, a low-pass filter is implemented to filter out these unwanted signals. In this project, the low-pass filter was chosen for its feasibility in programming and reliability against white noise (which mainly concentrate at high frequencies). The values of the filtered accelerometer data are calculated as:

$$filteredAccX[n] = (1-\alpha).filteredAccX[n-1] + \alpha.Input.accX[n]$$
$$filteredAccY[n] = (1-\alpha).filteredAccY[n-1] + \alpha.Input.accY[n]$$
$$filteredAccZ[n] = (1-\alpha).filteredAccZ[n-1] + \alpha.Input.accZ[n]$$

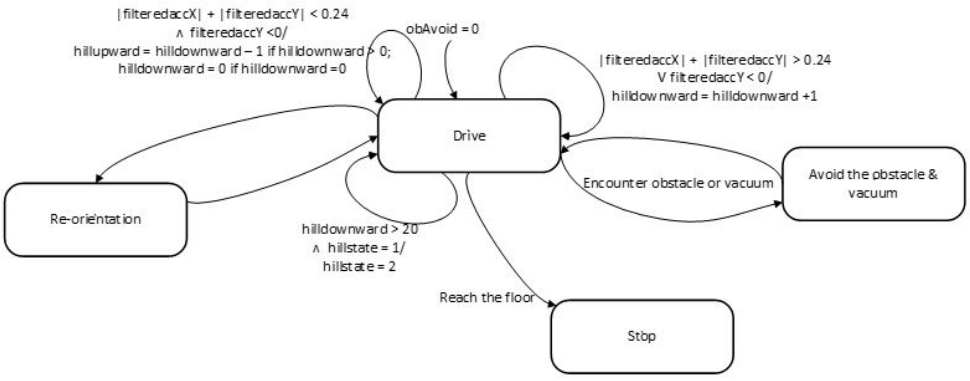
The variables *filteredAccX*, *filteredAccY*, *filteredAccZ* are added to record the filtered values of acceleration on the axes. The value of $y[n-1]$ is the previous filtered acceleration value while $Input.acc[n]$ is the input of the accelerometer. The α value is a predetermined parameter in the range of [0,1]. If the α value is relatively high, the system would be too sensitive to any change, which means the ability to resist against noise is lowered. In the other hand, if the α is too low, the smoothing effect will happen, making the system to extremely insensitive. We decide the value of $\alpha = 0.6$ by experimental approach, however we still encounter the cases where the jitters of the robot during rotating or driving causes the spikes which is mistake by the system as cliff. Therefore, we implement a hill detector algorithm in the next section to solve this issue.

3.1.3 Jitter handling

A notable issue that may cause the robot to detect the hill incorrectly is the sudden moving or stopping, causing acceleration values to surge. The algorithm therefore must be able to distinguish between the real states where Kobuki is moving on an incline, and the state where large values of acceleration signal is generated due to the Kobuki passing over sharp bumps. We observed that when the Kobuki is moving uphill, the $|filteredAccX| + |filteredAccY|$ value is almost larger than 0.24 at any time (theoretical analysis in Section 4.2.6), while $filteredAccY$ is larger than 0. If the $|filteredAccX| + |filteredAccY|$ surpass the guard of 0.24 due to the spikes of signal, $filteredAccY$ will oscillate drastically. Thus, we set up a counter for the going upward the hill event as variable `hillupward`. The value of this variable would increment by 1 each time the condition of acceleration is identical to the characteristics of acceleration while the robot going upward the hill, and would be subtracted by 1 each time this condition is not satisfied as shown in Figure 3.3a.



(a) Kobuki is going upward the hill



(b) Kobuki is going downward the hill

Figure 3.3: FSM of jitter handling

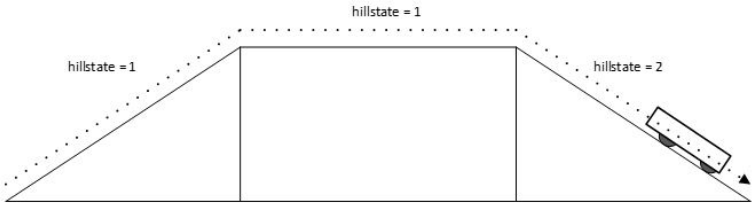


Figure 3.4: The states of hillstate value with respect to each part of the ramp

Note that the value of `hillupward` counter will not be lower than 0. If the event where the acceleration matches the features of the robot going upward the hill happens repeatedly, with the guard as 20, the system will switch the `hillstate` from 0 to 1. In this state, besides dodging the obstacles and the vacuums, Kobuki robot also has to navigate to keep it aligned towards the incline.

Similarly, the algorithm for detecting the going downward of hill state is designed with a counter & trigger. If the counter for detecting downward of the hill is filled (the guard is 20 as the same with upward detector), the `hillstate` is switched to 2. The states of `hillstate` with respect to each part of ramp is shown in Figure 3.4

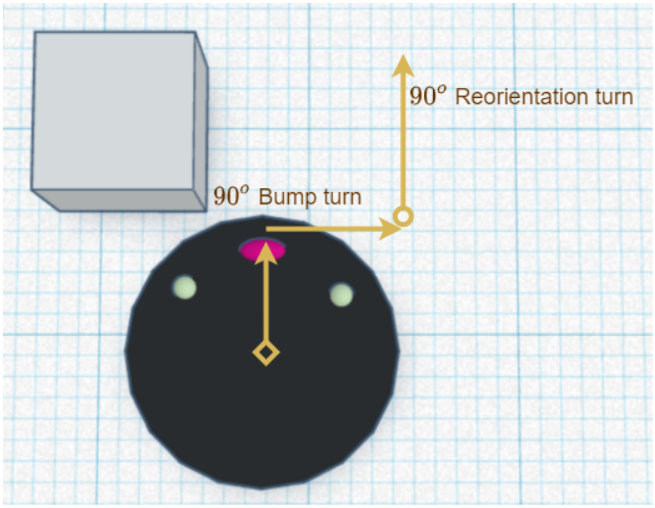
3.2 Obstacle & Cliff Avoidance

3.2.1 Goal-Oriented Movement

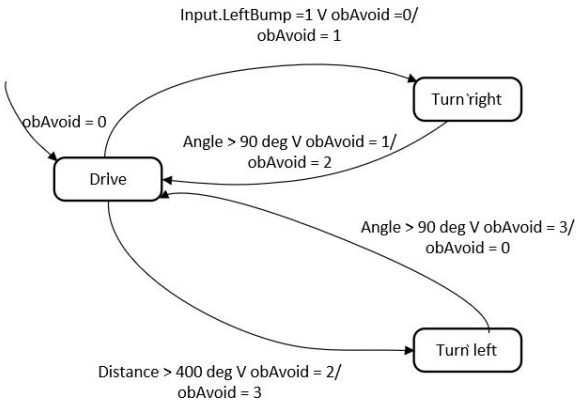
The only information that operators know about the map, is the direction to destination. Therefore, the robot is set to keep the same orientation as when it starts. Not only that it will satisfy the constant heading requirement, this will also ensure that Kobuki will always move towards the finish line. Specifically, the logic is done as follows:

1. Upon bumping or cliff, it will turn 90^o. Its turning direction will depends on which side it bumped into.
2. After turning, the robot will run for a short distance, then turn back to original orientation and moving on.

The turning angle is always set to 90^o in order to prevent the Kobuki from hugging the obstacles and make as few turns as possible to avoid the obstacle. By default, as specified in Section 3.1.1, the robot will turn right when a left sensor signals, or vice versa, and when multiple signals are raised at the same time, multi-signal handling logic will be done. For bumps of cliffs only at the center, another handling method will be described in the next part.



(a) Turning path visualisation



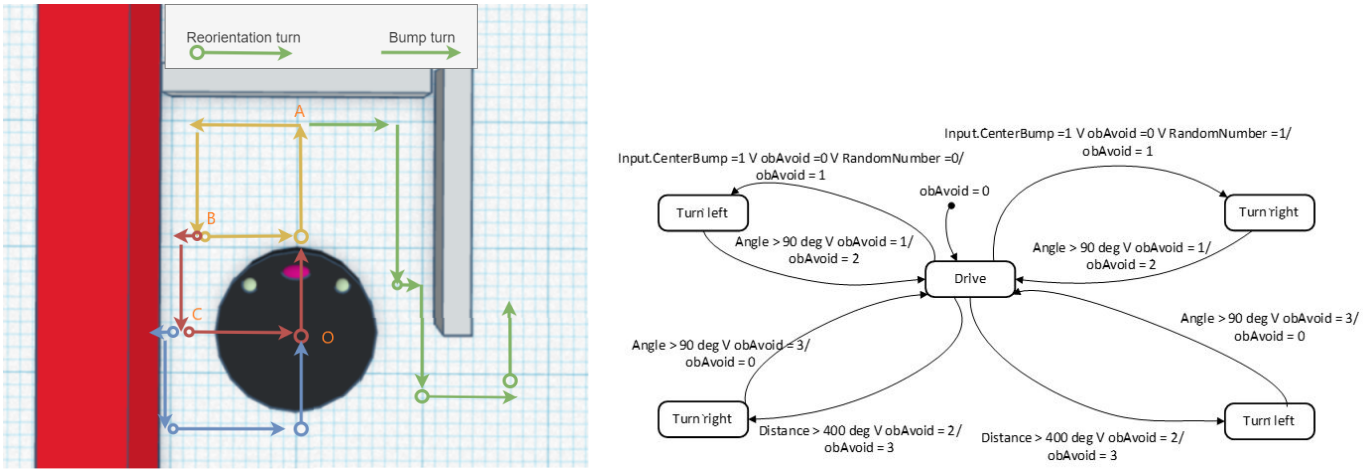
(b) FSM of turning process

Figure 3.5: How turning algorithm is implemented for Kobuki

3.2.2 Map Uncertainty: Stochastic Turning

Given that the robot itself will not have any prior knowledge about the map, nor able to retrieve any information at the start, being able to cover all dead-end cases will require very complicated implementation, which can be hard to set up due to time and memory constraint. To overcome this, a reinforcement learning based method is used. Monte-Carlo Tree Search (MCTS) technique finds an optimal sequence of actions for a player, by playing out all possible moves until it eventually samples enough possible sequences and their likelihoods [2].

With the same intuition to MCTS, the robot is set to make a random turn when bump or cliff happens at the front. Assuming all wrong direction will have road blocks (no infinite paths), the robot would eventually make the right sequence of turnings, and reach the destination. It is notable here that the process timing is dependent of the map complexity, where it will take longer for maps with more dead-ends.



(a) Simulated situations of the Kobuki robot in dead-end

(b) FSM of random turn

Figure 3.6: The random turn implementation

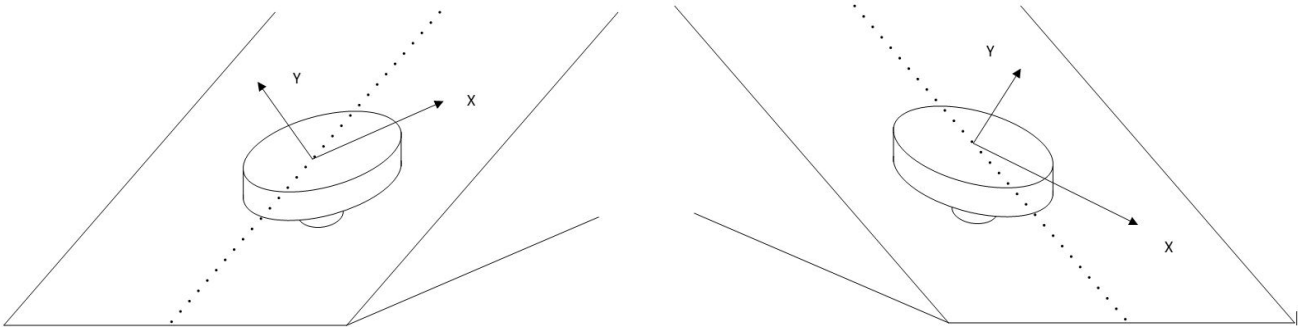
Figure 3.6a demonstrate an example case where Kobuki encounters a dead-end, with few small cases being omitted as they will eventually return to the same position: Starting at O, when bumping the wall at A, it will either turn right and exit correctly according to the logic, or turn left. If it turns left, it will reach B, and will again make a random decision on which side to turn, after having moved a specific distance. If it turns left, it will eventually reorient (due to being goal-oriented) and reach A again, then the process restarts. If it turns right, it will move to C, which again, will have a 50% chance of going back to A, or continue going down. Overall, when run enough number of times, Kobuki will eventually return to A, and will eventually make the right turning decision after a finite number of approaches.

3.2.3 Slope Re-orientation

After the state of being on the ramp is detected from the above algorithm, the Kobuki will navigate to keep it align with the ramp. The re-orientation movement would only be performed if the Kobuki is not avoiding any obstacle or cliff. According to the position of the accelerometer in Figure 3.7a, while the Kobuki is moving upward of the hill and the heading direction of the Kobuki revolves to the right, the acceleration value on the y-axis will be positive and vice versa, it would be negative if the Kobuki revolve to the left.

However, it is impossible to reach the state when the value of acceleration on the y-axis is 0 due to noisy measurements, and even if it happens momentarily, a noise or any jitter would easily break that state. Therefore, we need to set a range of angle where if the angle between the heading direction and the incline is still in that range, we consider that state as the Kobuki robot being aligned to the ramp. By experimental approach, we decide to set the range of $|filteredAccY| < 0.03g$ as the desired range.

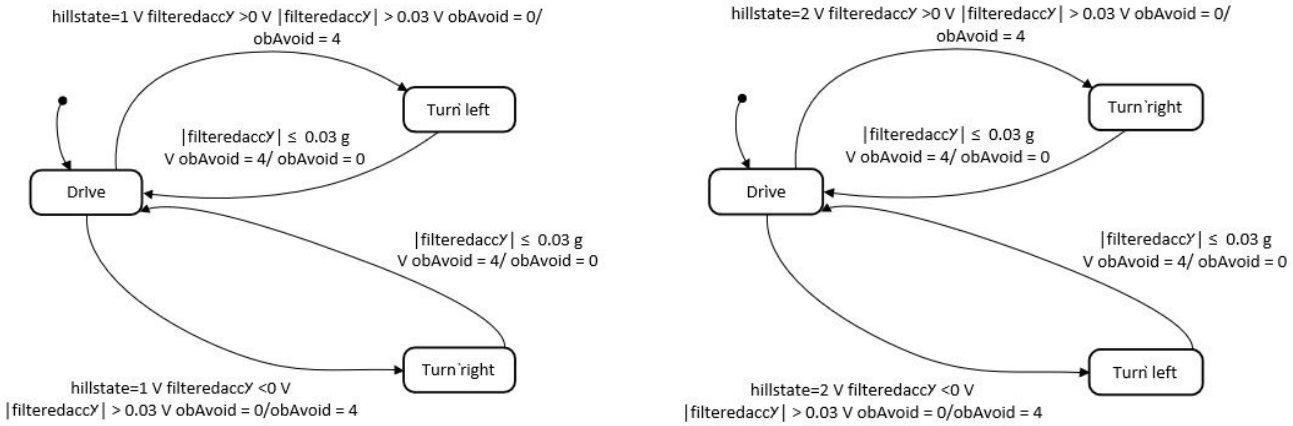
The similar transition's guard would be applied for the downward heading of the hill process. The only difference is that the acceleration value on the y-axis will be positive if the Kobuki revolves to the left and vice-versa (Figure 3.7b).



(a) Kobuki is going upward the hill (b) Kobuki is going downward the hill

Figure 3.7: The accelerometer axes while the Kobuki is moving on the ramp, assuming x-axis is forward direction

We decided to perform the re-orientation by simple rotation. As discussed above, the guard for re-orientation is set at $|filteredAccY| < 0.03$, the FSM or re-orientation for upward and downward to the hill are shown in Figure 3.8a and 3.8b



(a) Kobuki is going upward the hill (b) Kobuki is going downward the hill

Figure 3.8: The FSM for self-orientation while the robot is on the ramp

3.2.4 Edge-Handling: Reverse

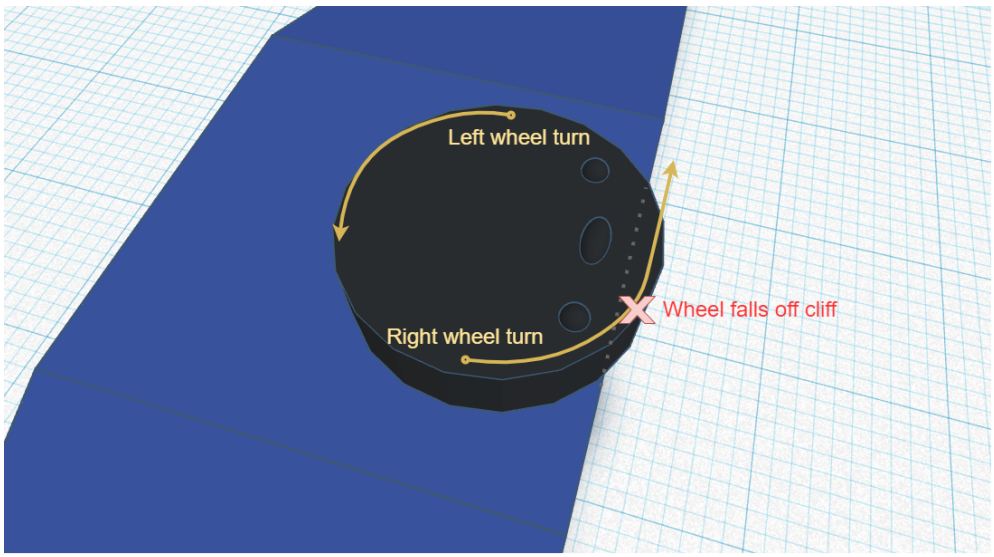
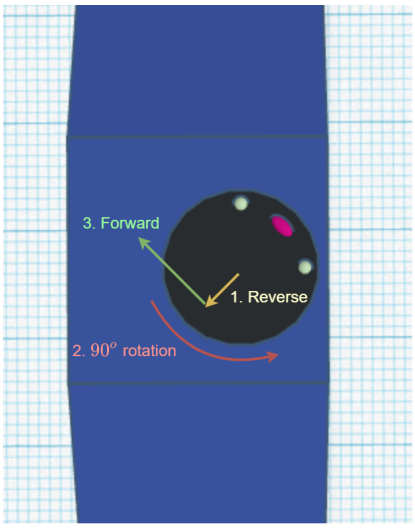


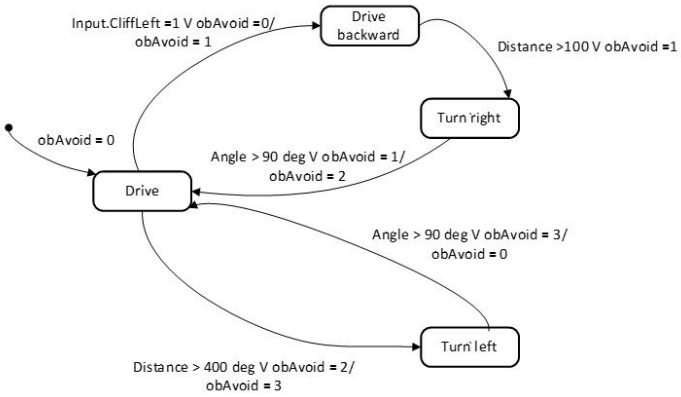
Figure 3.9: An edge situation where the Kobuki's wheel may fall off the course

The robot is set to turn if it detects either bump or cliff. However, as the cliff sensors are located further inside the robot, it can only signal when it is closer to the cliff than it is to the bump. As a result, the robot is more prone to sliding off the cliff when turning.

To handle this problem, the robot is specified an additional step: it has to reverse back slightly before making a turn. This way, the turning wheel is guaranteed to not falling of a cliff.



(a) Demonstration of the Kobuki reverse action



(b) FSM diagram of action when Kobuki approaches a left cliff

Figure 3.10: The Kobuki cliff reverse-and-turn process

3.2.5 Self-orientation if Going Backwards

During the navigation process of this project, the Kobuki is supposed to keep track with the heading direction which is determined from the start. During operation, there can be the case when the Kobuki encounter the U-shaped dead end or some combinations of obstacles that force the Kobuki to go backward to escape. Therefore, there must be an algorithm that re-orient the Kobuki to the predetermined direction. We designed it simply by using the *odomAngle* variable, which record the rotation angle respect to the heading from the start. Noted that it is different with $Angle$ variable which is the angle with respect to each turning. While the Kobuki goes toward the opposite direction for a certain distance ($distance > 1000$), the Kobuki robot will rotate by 180^o to get it back to the predetermined direction. The FSM diagram for this algorithm is presented in Figure 3.11.

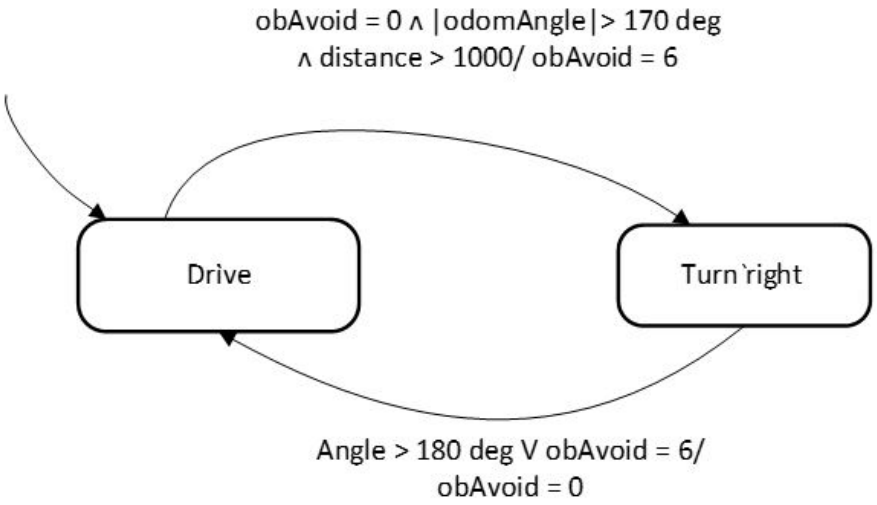


Figure 3.11: The FSM diagram for Self orientation while going backward too far

3.2.6 Stop after reaching the floor

Similar to the hill detector, the Kobuki may mistakes the momentary flat acceleration of the X-axis and Y-axis combined as the floor, therefore we set up a counter to detect the real floor. We notice that when the robot is running on the floor, the sum of $|filteredAccX|$ and $|filteredAccY|$ is usually lower than 0.03, thus we set this as the guard. This algorithm would only take action after the downward to the hill movement has been detected, if the sum acceleration on X-axis and Y-axis is lower than the guard, the counter `floor` would add 1 to its value and if the sum of acceleration is larger than this value, the `floor` value is subtracted by 1 (`floor`'s lower bound will be limited by 0). After reaching the floor, the $|filteredAccX| + |filteredAccY|$ will has the value being lowered than 0.03 repeatedly, when the counter reach 20, the Kobuki will stop. The FSM diagram of floor detection is shown in Figure 3.12

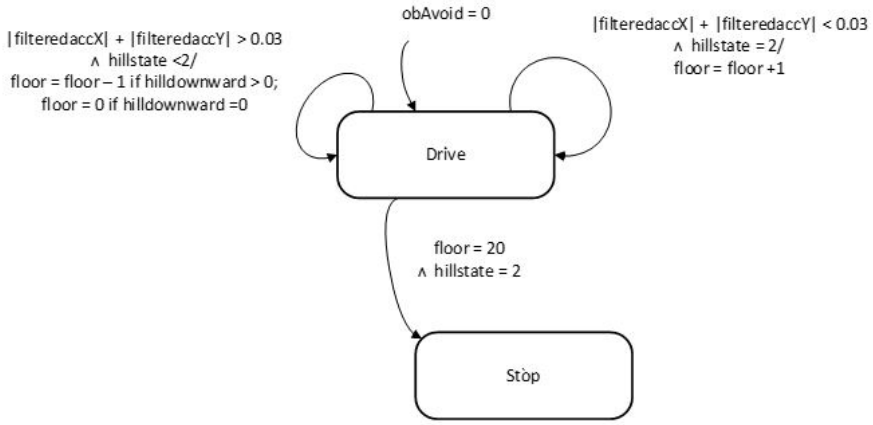


Figure 3.12: The FSM diagram for stopping after the Kobuki reaches the floor

3.3 Combining Algorithms

Overall, all necessary cases have been maximally covered, in order to maximize kobuki's chance of reaching the course destination. These ranged from how to avoid cliffs or obstacles, how dead-ends can be tackled, to how kobuki can filter out irrelevant signal inputs to determine its states in slopes. A final, fully composed version that combines all FSM figures in this section is shown in Appendix A.3. The next step after forming algorithms, is to select parameters analytically, as well as testing their robustness to different course types.

Stage 4

Testing, Parameter Validation & Calibration

The testing procedure was done in order to test the developed methods, algorithms and variables to see if they can really work efficiently, as well as looking for any potential problems or edge cases that were not detected during idea formulation. The process is done in a mixture of CyberSim simulations and in-person hardware test.

4.1 Method Comparison

4.1.1 Simulation test

Simulation is a quick way to test if all the implemented logic works as expected. Given that a simulated course is given, this is a convenient method, where the result can be seen directly on computer screen after deployment, and no complicated set up is required. While it can be used to check for logical flaws, it cannot cover cases where there is difference in Kobuki configurations. Examples of these includes the coordinate axis (x-y axis changed positions between simulation and hardware), the different outputs that actuators can produce due to manufacturing variation, or potential problems caused by sensor inaccuracies.

4.1.2 In-person hardware test

Hardware test is the best way to check if everything works as expected, given that the actual assessment run will be done in person. It will be able to check on logics, a Kobuki's specific component bias, and any potential inaccuracies of the actuators. However, the procedure will requires more complicated set up to get the Kobuki running. First, the Kobuki needs to be connected to the device, then connection configurations must be satisfied, and finally the program is deployed into Kobuki to be run. Not only these procedures require a long time to work, it is not a guarantee that the same Kobuki will be given through each runs, thus calibration cannot be done efficiently. However, the process would still be useful to look for possible faults and uncertainties that a Kobuki might potentially have.

4.2 Results & Adjustments

4.2.1 Overall

All logics implemented in the algorithm worked as expected. However, the procedure has proven to be useful, by pointing out a number of inaccuracies within the Kobuki hardware.

4.2.2 Heading

It was expected that the Kobuki would go in a straight direction by default. However, different test runs showed something else - Kobuki will have drifting in heading, where it rotates slightly to a constant angle as it moves. We tried to measure this, by keeping track of Kobuki's trajectory as shown in Figure 4.1. As shown in Table 4.1, the drift constant is different for each Kobuki - some headed to the left and some headed to the right, with different levels of drifting. The tracking result of this is noted, and will be investigated in the discussion stage.

Kobuki Number	5	6	8	9
Offset angle α	24^o	15^o	6^o	9^o
Side	Left	Left	Right	Left

Table 4.1: Drifting measures on different Kobuki robots

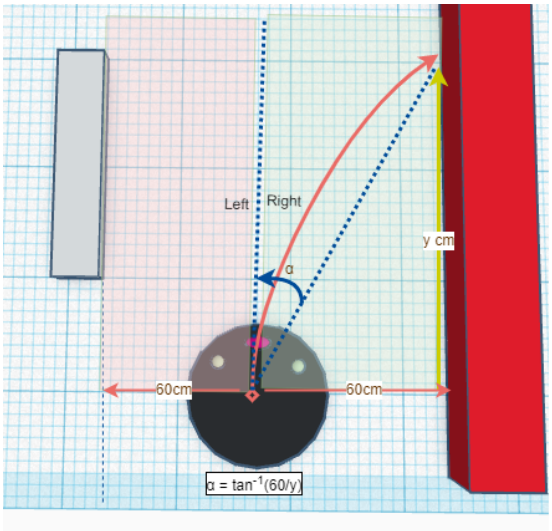


Figure 4.1: How offset angle is measured

4.2.3 Reverse-and-turn distance

One parameter needed to be considered when performing reverse-and-turn on cliff approaching, is the distance to reverse before making a turn. We want this to be large enough to prevent wheels from sliding off the cliff when turning, but also minimal so that the process is time-efficient, and not going to other side's cliff.

Initially, we put the reverse distance to be equal to the distance from the cliff sensor to the outer rim, which was measured to be 8cm. However, several in-person runs show that Kobuki still have the wheel tipped over the edge, proving that a close measurement may not be sufficient enough due to component inaccuracies, which varies between different Kobuki robots. To prevent this, we slowly increased the reverse distance by 2cm each time, until we no longer see tipping happens within 5 consecutive runs. As a result, we made the final reverse distance to be 16cm.

4.2.4 Movement Speed

The movement speed of Kobuki also needs consideration, as theoretically, it will be inversely proportional to the course run time, which means the faster the Kobuki, the quicker it will be able to reach the finish line. We set the initial speed to be the maximum speed that the Kobuki can have according to the datasheet - 70cm/s [3]. However, we realised that it also increases the risk of control, where it might have slight heading change after bumping an obstacle, or just finished going uphill. This is because speed is proportional to momentum, and when the Kobuki was supposed to be hold in place, momentum may affect on the weighting of Kobuki, making an unpredictable spin on the heading as the wheel lost contact with the ground. This may cause the loss of predefined goal orientation, and Kobuki may not be able to head towards the goal consequentially. Therefore, we have to reduce the speed gradually, until the heading change is acceptable after bumping and exiting hill ascension, which were found to be 50cm/s.

4.2.5 Turning Speed

Similar to movement speed, turning speed also decides how quick Kobuki is capable of traversing the course, but too large velocity can cause heading sliding off due to momentum drag. Furthermore, as rotation depends on internal gyroscope to guard, too fast turning can cause the gyroscope to exhibit drifting, and the final heading might be wrong as a result. Therefore, we decreased the turning speed gradually from $110^o/s$ (the maximum safe velocity recommended in the datasheet [3]), and found the best turning speed to be $90^o/s$.

4.2.6 Slope Detection

For slope detection and reorientation, there are 2 essential validations that needs to be done, directly relating to the extensive use of measurements in multiple axes for slope indication.

Axis Swap As mentioned in section 4.1.1, due to different myRio placement on Kobuki between simulation and real trials, the axes x and y needs to be swapped when switching the testing environment. In simulation, y is the heading forward axis, but in person, the myRio is placed so that x-axis is heading forward. This is important in the slope detection steps, given that these procedures use axial accelerometer measurements.

Angle Indication Given that the final course will use the same slope as testing, it can be useful to tune the slope indicators according to the testing terrain. In Figure 4.2 below, h and w are measured to be approximately 40cm and 200cm respectively. The slope is then determined as $s = tan^{-1}(\frac{40}{200}) \approx 0.245\ rad$. The algorithms relating slopes such as uphill reorientation will then have their thresholds determined from this value, using trigonometry and the given axes allocation (assuming $g = 9.8m/s^2$).

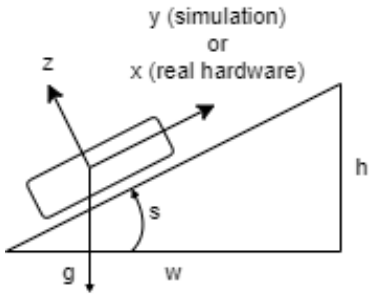


Figure 4.2: Diagram of Kobuki on an uphill slope

Stage 5

Final Run

5.1 Set up

The final run assessment is done in person on the Week 12 of the course semester. Each team will get one turn in running the robot through the course. The specified time limit to run the course is 180 seconds. The final Statechart in LabVIEW that were used in the run is also included in Appendix A.4.

5.2 Results

The Kobuki's movement during the final run was tracked in Figure 5.1. Overall, the result for our run was satisfactory for the majority of the tasks and requirements, but did not finish fully due to an unknown equipment fault. In each stage, the performance were recorded:

Start The Kobuki did not move on its own. After pushing the `B0` button corresponding to 'Play', Kobuki started moving immediately.

Ground Course Kobuki was able to avoid all the obstacles it encountered in the course. We were lucky that the Kobuki did not fall into any dead-ends on the ground, hence the time for Kobuki to reach the hill from start took only less than 40 seconds in total. It is noticed again here that Kobuki did not move in an exactly straight direction. Its heading angle slightly headed over to the left as it moved.

Uphill & Plateau At the start of the upward slope, Kobuki was able to self-orient so that its movement direction will be towards the plateau. After reaching the plateau, Kobuki's heading slightly leaned to the right, and approached an edge cliff at the plateau. This was a lucky chance, as we get to observe Kobuki successfully performing the reverse-then-turn action (Section 3.2.4) as expected. Kobuki then proceeds to go downhill towards the finish line.

Downhill & Finish Kobuki was able to go downhill with a constant speed, but when it was about 10cm before finishing the slope, the robot suddenly stopped operating, which never happened during test runs. While it was unknown what have caused the problem, we were not considered to have finished the task. If Kobuki did not stop operating mid-way, we would have completed the course in less than 100 seconds.

Overall, the requirements from Table 1.1 were checked, and we have satisfied most of them:

Name	Objective	Achieved?
Startup & Run	Kobuki moves if and only if 'Play' button is pushed	Yes
Obstacle Avoidance	Is able to avoid objects on the path, without hugging them	Yes
Cliff Avoidance	Is able to avoid falling off edges or wheels losing ground contact	Yes
Hill Climb	Oriented to go towards ascending direction, not encountering any edges	Yes
Downhill Stop	Must stop within 40cm after finish descending	N/A
Timeliness	Reaches the end point within a specified time	N/A
Performance	No chattering, no rotation of over 180^o, no sudden termination	Sudden termination

Table 5.1: Project requirements and their completeness checklist

The next stage will investigate and discuss further details into these results. We will also look more into smaller problems that happened during the runs.

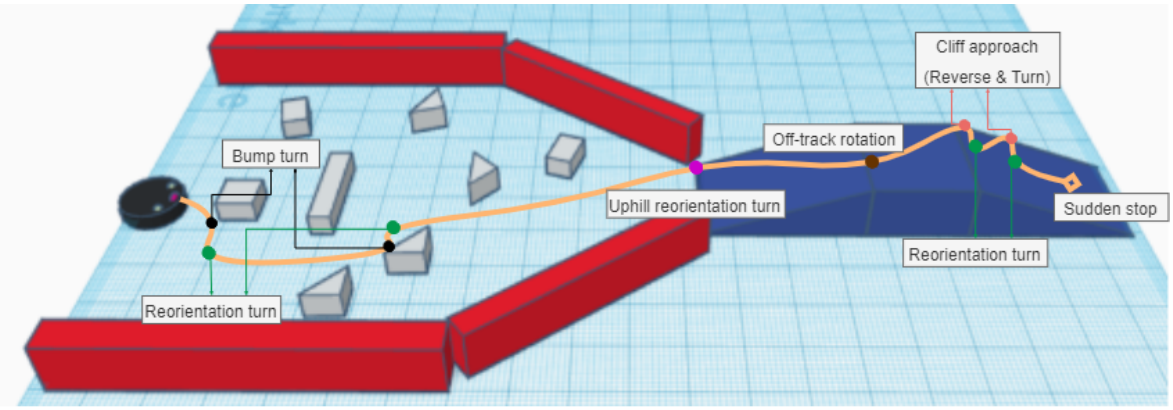


Figure 5.1: Kobuki's path of movement during Final Run Assessment

Stage 6

Result Discussion

6.1 Achievements

Overall, through test runs and the final assessment run, all case-specific prevention measures that were developed for our Kobuki worked successfully. The robustness of these algorithms have been verified in the final run, where we were able to get through the ground course and reach the hill plateau within just a short amount of time.

6.2 Downfalls & Problems

6.2.1 Heading Drifting

As mentioned, although we have set the wheel motor speed to be constant across both side wheels, the heading of the Kobuki still had a slight constant change to the left as it moved. As the amount of change was different for each Kobuki, we conclude that this heading drifting phenomenon happened due to manufacturing variations between each Kobuki's wheel motors. The variance have caused the Kobuki to have slightly different velocity, and Kobuki could not move in a straight line as a result. This is inevitable, as Kobuki devices are allocated randomly in the final run, making any means of manual calibration unfeasible due to non-intervention rules.

6.2.2 Limited Knowledge

Our Kobuki was able to walk through the ground course in short time, this is because the obstacle distribution on the ground was sparse, and Kobuki luckily chose easier paths. But we believe that in more complex courses such as mazes, Kobuki will not perform within satisfactory time, nor even reach the destination. This is because the robot does not have any prior information about the map, and it does not have the capability to retrieve any mapping information.

6.2.3 Unknown operation termination

When Kobuki stopped working suddenly, the cause could not be investigated, as the device that imported the code to Kobuki was disconnected due to weak wireless connection. With the fact that this did not happen before during test runs, and noise cancelling did not show any problem, we came up with two hypotheses that may have caused this incident:

- The Kobuki may have ran out of battery and disconnected all the power sources to the motors. This can happen if the battery charger did not work as expected.
- Something may have caused the orientation sensors in MyRio to give a constantly false flat measurement, such as the MyRio sliding off from original position, or MyRio sensor returns a null (unmeasurable) value, which might be read as 0 in the program.

Both reasons above are related to the fact that manufacturing defects or age-affected dysfunction in any component of Kobuki or MyRio can make the whole system become unknowingly unreliable. Unfortunately, as we did not have any evidence of these problems, our Kobuki's final run was not considered completed.

6.3 Potential Improvements

While our implementation worked well in the final course, there are still many potential developments that could not be integrated, due to limited software access, restrictions and time limits. If these were to be applied as part of the algorithm, the Kobuki would overall perform significantly better, and become more adaptable to a larger variety of courses.

6.3.1 Autonomous heading calibration

Our implementation had the problem of heading drifting, which did not have any solutions developed to handle due to the time constraint. This might not work well for larger courses, as the heading of Kobuki might change significantly during the run. This can be prevented or minimized by developing a method where the Kobuki can calibrate itself. Such examples include keeping track of the heading, and reorient every time it exceeds a threshold, or a feedback control model that calculates the offset between the wheel velocities, and self-adjust accordingly.

6.3.2 Input guards

With the hypotheses mentioned when the Kobuki terminates suddenly, they all relates to the fact that the inputs could not be checked throughout, such as battery remaining, or if the sensor is returning erroneous value. In the future, guards that are able to device-specific problems can also be investigated and implemented in order to prevent any in-progress breakdown.

6.3.3 Case-specific logic

With limited access to the in-person course, it is hard to identify all possible corner cases. When more timing allows, more edge cases can be tested, such as small pathways, or narrow dead-ends. With careful testing, the state machine implementation can be expanded further to handle more specific problems that Kobuki might meet on its path.

6.3.4 Sensor addition & fusion

In this project, Kobuki could only rely on its own & MyRio sensor systems. This will limit the Kobuki into being able to handle problems before it happens. For example, integration of a LIDAR or stereo cameras might help the Kobuki to detect an obstacle ahead, which can help it calculate and make a decision ahead of time. Similarly, localization can be done by further combine the available sensors of Kobuki, such as gyroscope and accelerometer, to keep track of where Kobuki is with respect to the starting point and the destination.

6.3.5 Heuristics & Path-finding Algorithms

A problem that may prolong the operation time of a Kobuki, or make it unable to reach the destination, is that it does not know any prior information about the course, and does not have 'memory' to store its knowledge about the course, making it susceptible to repeat its mistakes. When further hardware are allowed to be integrated with Kobuki, implementations that help the robot getting prior information about the map, as well as storing information, becomes possible. With that being possible, Kobuki can be able to plan the intended route ahead, before starting its moves. An example includes installing cameras, and use their inputs to perform Simultaneous Localisation and Mapping (SLAM), which will work out the partial map of the course, as well as keeping track of where Kobuki is within that map. Once being able to know the map, possible paths can be worked out using Voronoi diagram, and path finding algorithms such as A-Star or Dijkstra can be used to work out the optimal path to get Kobuki to reach its destination as quick as possible.

Stage 7

Conclusion

Overall, our design & implementation of the Kobuki was largely successful in getting it to navigate itself through the obstacle course. With the material provided from the workshops and course lectures, as well as implementation instructions for the Kobuki, the logical design was well created and was able to handle the basic requirements. This has been further refined through a rigid process of testing, validation and calibration, which helps the Kobuki become more optimal in performing its tasks.

The final course run further demonstrates the effectiveness of our implementation, shown through the fact that the Kobuki was able to traverse through all obstacles in less than 40 seconds, and was able to perform consecutive successful turnings and reorientations when meeting various obstacles or cliffs. The only failure that happened was that the Kobuki stopped working unexpectedly just right before the finish line, which was believed to be an unexpected equipment fault. Potential prevention measures for cases like this have been mentioned in the discussions above.

While the major logic can be developed with ease, more challenges arise as more edge cases and noisy or inaccurate measurements are introduced, and need to be handled. As there are countless possible edge cases, and it is hard to prevent sensors from providing erroneous data, careful considerations were taken, in order to keep the operation effective, but not making the logic too complicated with addition handlers. Testing and calibration procedures were carried out continuously in order to check the feasibility of the system. However, it is hard to cover all possibles in the real world, which highlights the difficulty of autonomous vehicle programming.

The case by case handling difficulties also shows the problem of working with embedded systems, specifically state machines. To counter the unpredictability of real life situations, everything has to take careful consideration so that all system affections are noted and expected. This is even harder, with the problem of sensors or actuators providing different measurements and outputs, due to manufacturing defects or noisy processing, which have proved to cause many problems, even for a small scale application like this project.

Teamwork is also considered an important aspect of this work. Having many people can provide different views to the procedures, coming up with more ideas and possibilities, as well as identifying more potential problems or logical flaw within an algorithm or handling method. This have helped the team to implement an efficient algorithm that worked well in general, which could have been harder to proceed with only 1 person doing the work.

Common problems for this project surround the fact that the system has limited measurement, and has no prior information about the course. Therefore, potential improvements mostly focused on improving the perception of the Kobuki, by the addition of sensors that can provide it with online information and 'memory', as well as using modern path finding algorithms that can enhance the planned path. Regardless, the available resources were sufficient enough for courses with the same scale as this project.

In conclusion, the project have displayed the complexity of embedded programming. Therefore, careful analysis was taken throughout each process to ensure that problems are covered as much as possible. This has introduced the basic concepts of autonomous design, and has various potentials to be applied in the future, when more resources and time allow.

Bibliography

[1] Edward A Lee and Sanjit A Seshia. *Introduction to embedded systems*. The MIT Press. MIT Press, London, England, 2 edition, December 2016.

[2] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, pages 72–83. Springer Berlin Heidelberg, 2007.

[3] Yujin Robot. Online user guide, Mar 2017.

Appendix A

Additional Resources

A.1 Kobuki Component Breakdown

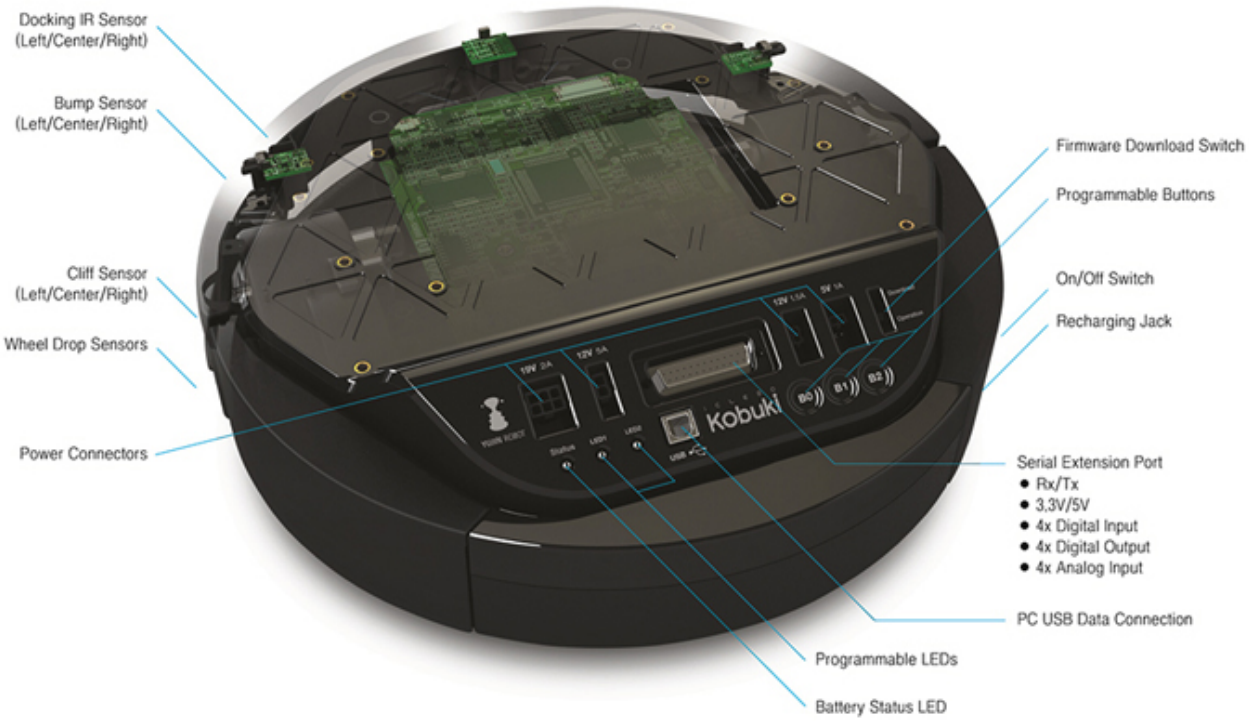


Figure A.1: Annotation of Kobuki components in the datasheet [3]

A.2 Kobuki Operating Specifications

Taken from Kobuki's online user guide [3]:

- Maximum translational velocity: 70 cm/s
- Maximum rotational velocity: $180^o/s$ ($> 110^o/s$ gyro performance will degrade)
- Payload: 5 kg (hard floor), 4 kg (carpet)
- Cliff: will not drive off a cliff with a depth greater than 5cm
- Threshold Climbing: climbs thresholds of 12 mm or lower
- Rug Climbing: climbs rugs of 12 mm or lower
- Expected Operating Time: 3/7 hours (small/large battery)
- Expected Charging Time: 1.5/2.6 hours (small/large battery)
- Docking: within a 2mx5m area in front of the docking station

A.3 Completed FSM of Kobuki

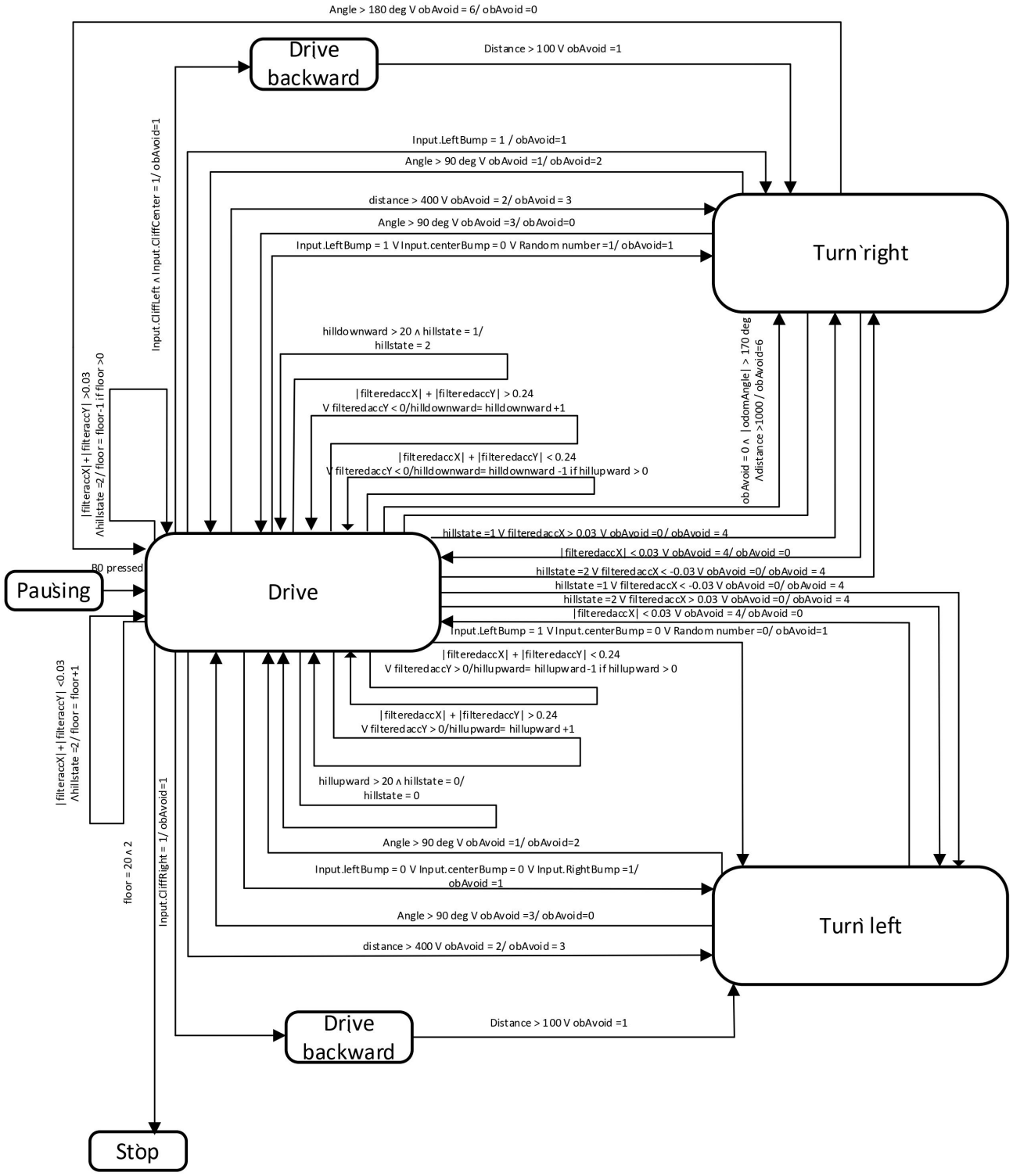


Figure A.2: Completed FSM consisting of all algorithms for the final course run

A.4 LabVIEW Statechart of Running Mode part

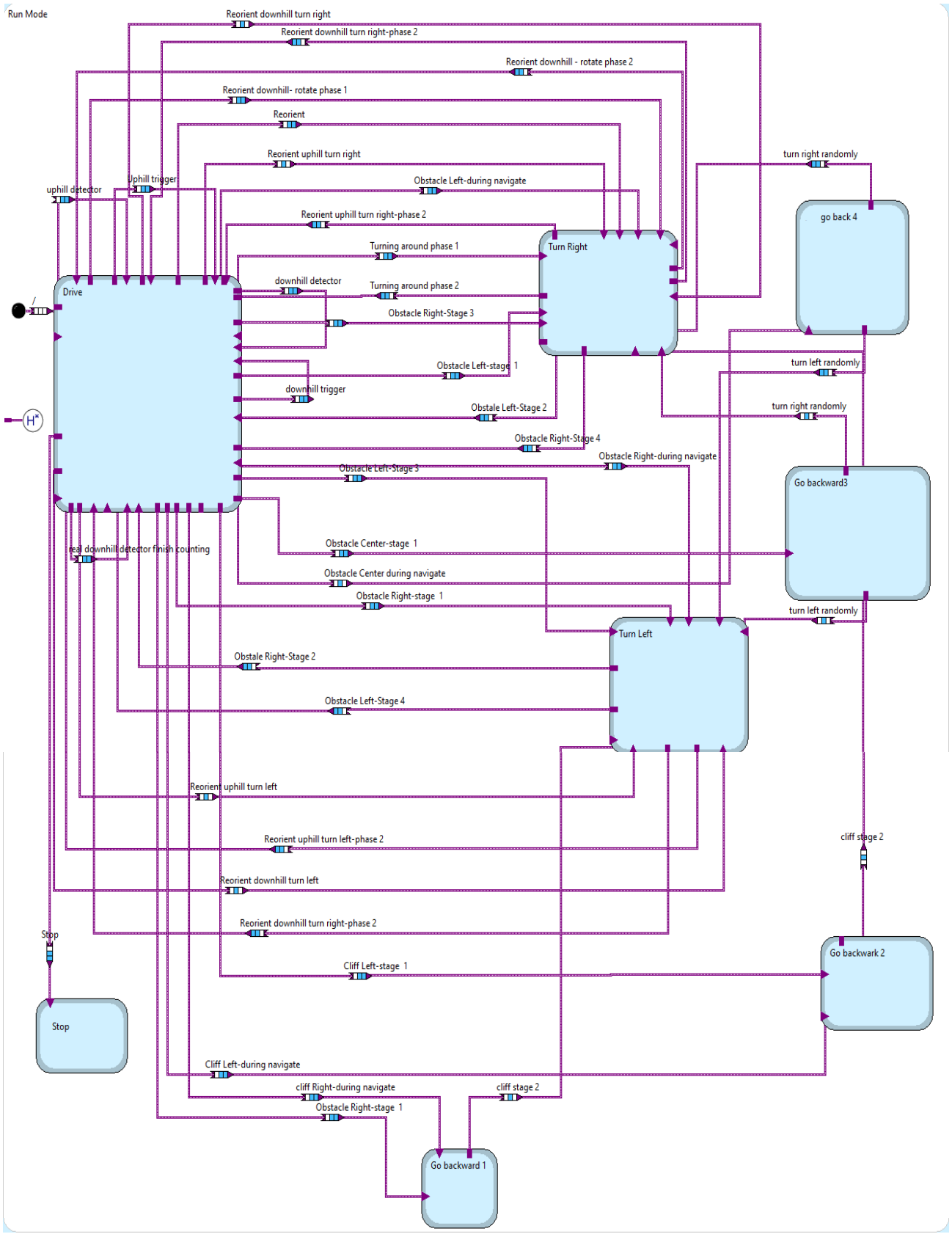


Figure A.3: LabVIEW Statechart for the final course run