VEX Robot Final Report

Student Name: Alireza Nouri, Armaandeep Singh, Ayush Gandhi, Sharvesh Kandavel

Student ID: 21203771, 21176574, 21218866, 21226122

2nd December 2025

Word count: 6135

Table of contents

1 – Introduction **1**

2 – Scope **1**

2.1 – Changes in Scope **2**

3 – Constraints and Criteria **2**

3.1 – Navigation **2**

3.1.1 – Constraints **3**

3.1.2 - Criteria **3**

3.2 – Cube Detecting and Sorting **3**

3.2.1- Constraints **3**

3.2.2 - Criteria **4**

3.3 – Elevator and Structural Components **4**

3.3.1 – Constraints **4**

3.3.2 – Criteria **5**

4 – Mechanical Design and Implementation **5**

4.1 – Overall Design Description **5**

4.1.1 – Original Design **5**

4.1.2 – Second Design **6**

4.1.3 – Final Design **8**

4.2 – Chassis Design **9**

4.3 – Motor Drive Design **9**

4.4 – Sensor Attachment Design **10**

4.5 – Overall Assembly **11**

5 – Software Design and Implementation **12**

5.1 – VEXcode and Organization **12**

5.2 – Task Alterations **13**

5.2.1 – Pathfinding to Home Plate **13**

5.2.2 – Varying Bin Heights **14**

5.2.3 – Error Correction **14**

5.2.4 – Color Detection **14**

5.3 – Code **15**

5.3.1 – Program Initiation **15**

5.3.2 – Predefined Path Following and Obstacle **16**

5.3.2.1 – Testing **17**

5.3.3 – Returning to Home Plate and Drop-Off Plate **17**

5.3.3.1 – Testing **18**

5.3.4 – Color Sorting Cubes into Bins **18**

5.3.4.1 – Testing **19**

5.3.5 – Preparing For Next Cycle **20**

5.3.5.1 – Testing **20**

6 – Verification **21**

7 – Project Plan **21**

7.1 – Task Distribution **22**

7.2 – Revisions to the Project Plan **22**

7.3 – Planned vs Actual Timeline **22**

7.4 – Deadlines and Timings **23**

8 – Conclusions **23**

9 – Recommendations **24**

9.1 – Mechanical Improvements **24**

9.2 – Software Improvements **24**

10 – Bibliography and Appendix **25**

10.1 – Bibliography **25**

10.2 – Appendix A – Complete Program **26**

List of Figures

Figure 1: **1**

Figure 2: **4**

Figure 3: **6**

Figure 4: **7**

Figure 5: **8**

Figure 6: **9**

Figure 7: **10**

Figure 8: **11**

Figure 9: **12**

Figure 10: **15**

Figure 11: **16**

Figure 12: **18**

Figure 13: **19**

Figure 14: **20**

1 – Introduction

The goal of the project was to create a robot that actively and autonomously searches for colored cubes obstructing its path to collect and sort by color. The robot was able to perform all tasks with the help of motor encoders, Proportional-Integral-Derivative (PID) and sensors, which will be discussed in more detail later. In terms of mechanical design, the robot was able to complete core tasks with the use of an elevator mechanism implemented with a guided rail and a grabber, both functioning via sets of gears connected to motors.

2 – Scope

The main functionality of the robot is to color sort multiple objects along the predefined path followed by the robot. The robot had to go through a rectangular serpentine path and scan for cubes obstructing its way. After detecting a cube, the robot detects its colour. The predefined path is shown in Figure 1 below.

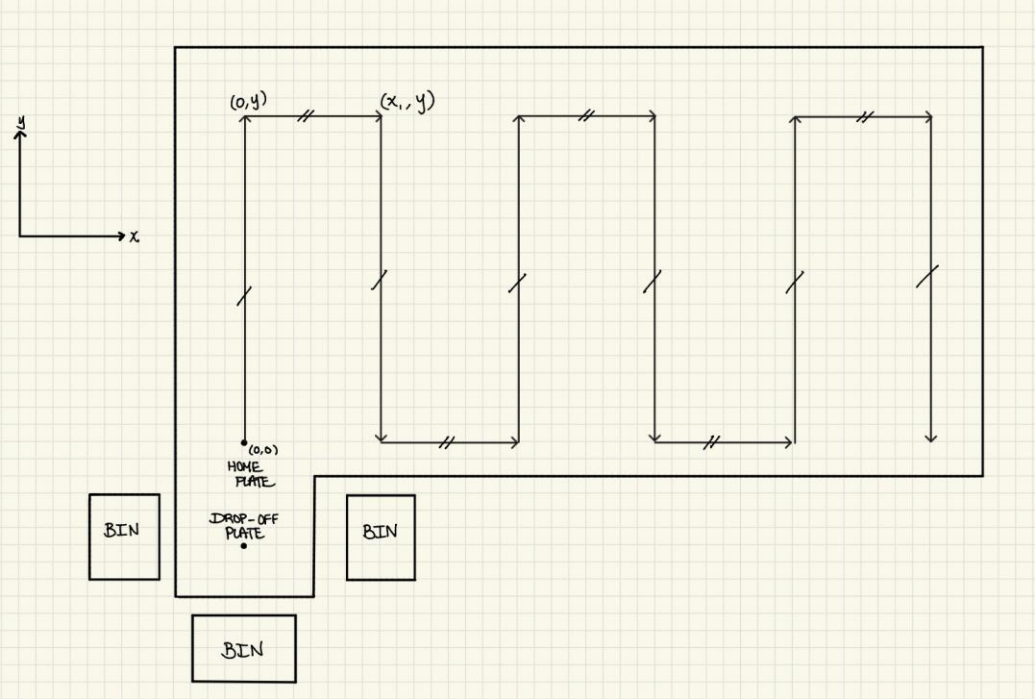


Figure 1 – Digital drawing showing the predefined path the robot followed

After picking up the cube, the robot initializes a return home procedure, where the robot navigates back to where it started. Since the robot moves in both the x and y direction, motor encoders were used to keep track of wheel rotations for both directions.

Once it reaches the home plate, it initializes the drop off sequence, allowing the robot to understand which bin to put the cube in, according to its color.

After detecting every cube within its path, the robot will continue until the end of the predefined path. Once the end is reached, the program will shut down.

The robot is expected to successfully travel the full path while maintaining accuracy, handling the objects encountered, and complete the task by reaching the end of the course.

2.1 – Changes in Scope

At first, the design included bins of different heights. The elevator system was expected to stop at the appropriate height for each bin to store the object. Due to time constraints, the drop off system was simplified to the use of bins with the same height. Moreover, a recalibration procedure was added to correct the misalignments and inaccuracies accumulated.

3 – Constraints and Criteria

The development of the robot required constraints and criteria to be set in place to ensure that it performs adequately in both the programming and the mechanical design side. These constraints and criteria are required for repeatable and reliable operations of the robot. Certain elements such as navigation, object detection and structural components are critical constraints and criteria set in place for optimal functionality.

3.1 – Navigation

3.1.1 – Constraints

The major constraint on the robot's operation was the limitation of its area. The specified fixed rectangular arena is defined by a maximum and minimum vertical range and a maximal horizontal displacement. This restricts the robot's motion to only the two axes. Any deviation from its path would introduce inaccuracies when picking up objects and errors when returning to the home plate.

Another constraint would be the predefined path the robot must follow. It must pass on the path that the coloured cubes will be placed on. Any deviation from its path will impact its ability to detect, pick up, sort the objects. This deviation also has an impact on its process to returning to the home plate. Once it picks the object, it must return back home using the net x and y distances it travelled. Once all objects have been found, the robot continues till the end of the path and shuts down at the end of the arena.

3.1.2 – Criteria

A primary criterion for the robot journey across the arena is that it must maintain accuracy while tracking the changes in x position and y position. This appropriate tracking allows for the return home process to be executed with precision. Vertical changes in position are regarded as changes in the y-axis and horizontal changes are regarded as changes in the x- axis.

Additionally, the robot must begin correctly aligned with the path. The proper orientation ensures that the robot does not start to deviate off course from the beginning of the run. This could cause malfunctions when picking up cubes and errors while returning to the home plate. Proper starting conditions make for better runs as it allows for the robot's logic to perform reliably by minimizing the path deviations and malfunctioning cubes detection.

Moreover, when turning the robot must make exactly 90 degree turns in both counterclockwise and clockwise. This ensures the robot to stay on the course and maintain proper cube detection as well as proper return home process.

3.2 – Cube Detecting and Sorting

3.2.1- Constraints

A primary mechanical constraint was the design of the grabber system. To ensure a tight handle on the cube once it was grabbed, the cube was designed with holes on the sides to have space to place the grabbers shafts through. This required for the robot to align itself directly facing the object and at an appropriate distance for the shafts to enter properly, without any slippage. The stopping distance is not only a criterion but also a constraint as the robot must position itself effectively to have the grabber's arm pick it up. Any errors would cause the robot to fail picking up the cube steadily and securely.

Moreover, another constraint was that the object should not fall during the return home process but reliably fall when at the appropriate bins. For this to function, the grip should not be too tight nor too loose. The holes and shafts going through must have substantial clearance. The release and grabbing speed of the grabber motor had to be precisely controlled to ensure a secure deposit into the bins.

3.2.2 – Criteria

A criterion for the performance of the robot is to stop precisely at the specified distances from the cubes as well as the bins. When approaching a cube, the robot must stop at a specific distance for the grabber's arm to be able to reach the holes on the side of the cube. Similar logic goes into place when the robot approaches the bins for the cube to be dropped in the center. This allowed for proper task execution and reliable runs.

Moreover, another criterion is to ensure reliable colour detection before picking up the object. The robot has to reliably identify the appropriate colour under various lighting conditions so that the cubes can be appropriately sorted in the respective bins. Given that the bins can be moved around, the colour detection allows for the cubes to be accurately sorted.

3.3 – Elevator and Structural Components

3.3.1 – Constraints

A mechanical constraint for the elevator system was limitations on the vertical displacements. Using motor encoders, there were maximum and minimum rotations set for the elevator motor in order to prevent the elevator mechanism from detaching and braking. This constraint prevented the elevator's mechanical failures.

Moreover, the dimensions of the bins used was another constraint. These boxes needed dimensions large enough to allow a large clearance when the coloured cubes were dropped from the grabber. They must also be tall enough for them the cubes to not bounce out when dropped into the bins. Figure 2 below shows the bins selected.



Figure 2 – Image showing the bins that were used. The cubes are also shown in the center

3.3.2 – Criteria

A criterion was to ensure the elevator's vertical motion to be smooth during cube pick up process and while accomplishing the return home process. After a cube was detected and picked up, the elevator motor had to maintain a hold to not drag the coloured cubes on the floor as the returning home process was running.

Furthermore, an important criterion was the smooth operation transition when grabbing the cube, lifting it up and finally dropping it off. After grabbing the object, the elevator had to raise it high enough to ensure that the cube does not interfere with the bins while not exceeding the mechanical limits. This process had to be had to repeatable and stable to ensure accuracy and prevent misalignments. The cubes are shown in Figure 2.

4 – Mechanical Design and Implementation

4.1 – Overall Design Description

4.1.1 – Original Design

Originally, the team chose a chain drive system with a spur gear and a heavy duty offset sidebar roller chain. Within the gaps of the chains, the group attached the entire grabber mechanism by inserting a shaft through a hole. While this was a functional method, the VEX chains introduced a lot of slack even though the chains were made as tight as possible. This original design did not include the use of a guided rail system, as it was not thought of in the time. The slack of the chain prevented the robot from grabbing objects steadily and confidently transporting them across the arena. Moreover, the hand shape used in this model lacked traction, making the objects picked up fall immediately after picking them up. This slack introduced many unpredictable errors, including robot tilting, which lead to torque imbalances as well as incorrect object grabbing. Additionally, the provided amount of VEX chain links limited the height of the chain drive system. Figure 3 below provides an image of the first design.

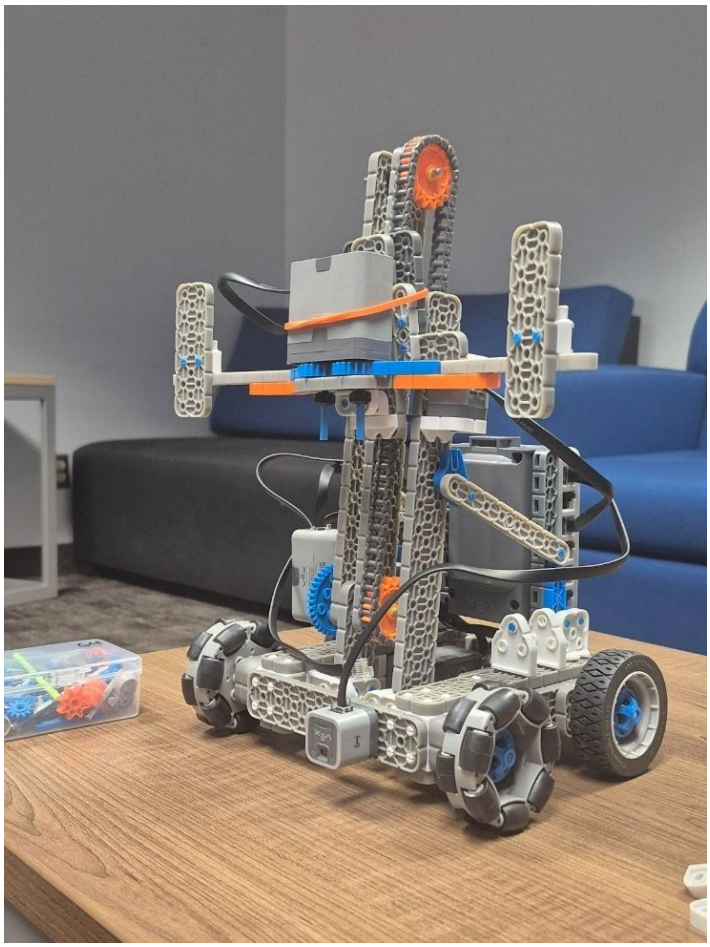


Figure 3 - First constructed design of the robot

4.1.2 – Second Design

Afterwards, for the second Design, a pulley design was chosen to lift the grabber. To enhance the pulley's accuracy and stability, a guided rail system was implemented. Sliders that surround the rails were placed for the grabber to have smoother vertical displacement. The general shape of our grabber mechanisms changed to increase stability when picking up an object in the path. The larger surface area of the grabber allowed it to be constrained itself evenly along the guided rails. The hands of the robot were made smaller, and rubber elastics were added to enhance the grip that was lacking in the previous model. Figure 4 shows the second design.

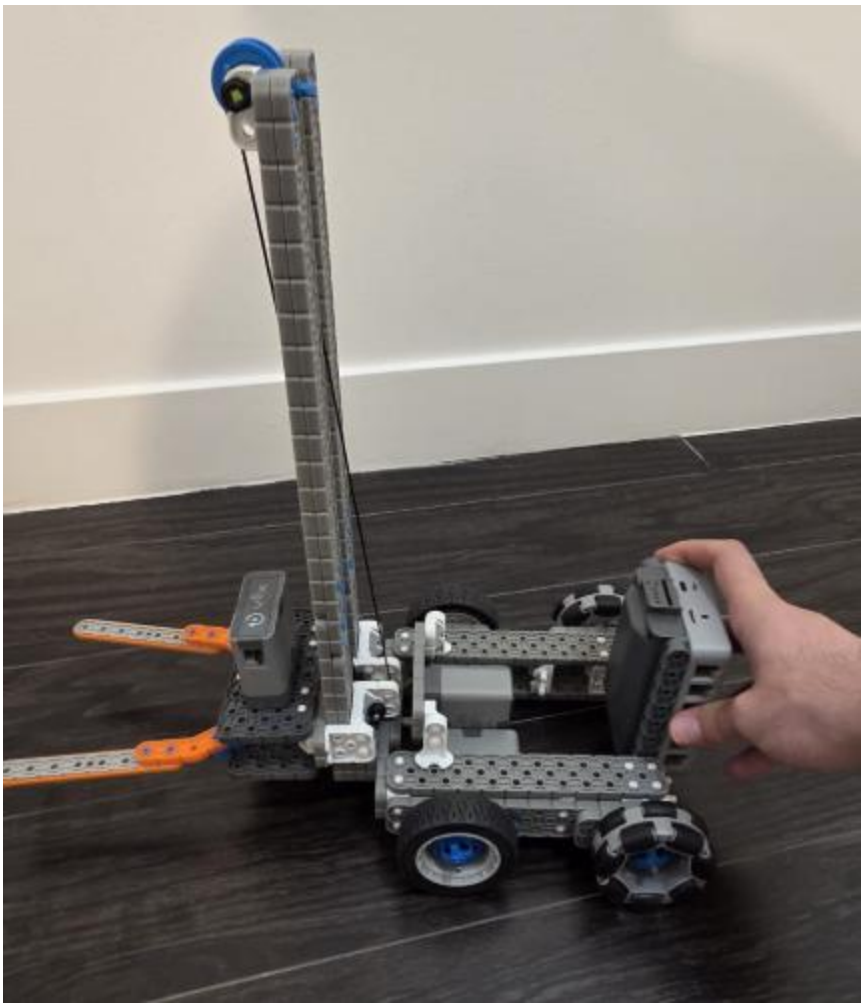


Figure 4 - Second constructed design of the robot

However, even though it was a sturdier model, there were certain complications that arose with the changes that were made. There was an uneven weight distribution towards the front of the robot even while the brain was acting as a counterweight. Additionally, the provided strings did not allow for the pulleys used in the elevator mechanism to go to the desired heights. Furthermore, the elevator mechanism couldn't go as low as it needed to go to pick up objects.

4.1.3 – Final Design

For the final Design, the team combined what was learned from the first two designs. The first design's chain drive idea was incorporated and the guided rails and wider grabber mechanisms from the second design were used. The robot's overall design involved the usage of mechanical components such as a grabber system, a guided rail system and an elevator system. The elevator system consisted of a chain drive that allowed the vertical displacement of the grabber. The elevator system's motor was attached with a VEX heavy duty offset side bar chain surrounding a spur gear. This gear was attached to a shaft that consequently spun a LEGO gear attached with its appropriate chain. This LEGO chain was attached to the grabber with a shaft connected through a hole in the VEX piece. The grabber mechanism was also attached to a slider component which surrounded the guiding rails. This piece allowed a smoother vertical displacement of the entire grabber piece.

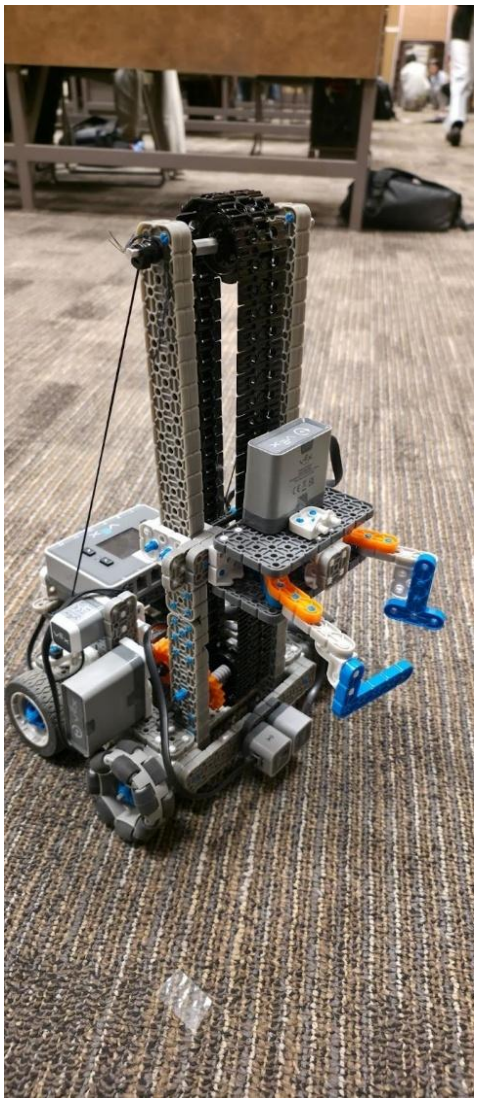


Figure 5 - Final constructed design of the robot

4.2 – Chassis Design

For the chassis design, the robot's width was increased to fit the entire elevator mechanism. The elevator's motor, used for the vertical motion of the grabber, interfered with the rotation of the wheels, so plates were added to increase the space of the wheels from the chassis.

4.3 – Motor Drive Design

The front wheels used were omni-directional wheels. Turning was made easier and more accurate since the omniwheels were placed in front. Because the omniwheels reduce friction with the ground, the robot's pivoting motion was smoother and more precise. The robot was a two-wheel drive design since the rear wheels used motors.

The elevator mechanism used a motor that had a set maximum limit and ensured that the elevator mechanisms do not go past this limit. Gear slippage was minimized by adding extra gear reinforcement, which helped to maintain a consistent lifting motion under the load. Figure 6 below shows the extra gear.



Figure 6 - Two adjacent orange gears to ensure minimal gear slippage

The grabber mechanism used a motor that gripped objects. Using a set of 2 linked gears, spinning in opposite directions, controlling the opening and closing of the grabber arms, allowing the robot to securely pick up and release the colored object. For a reliable grip, the motor applied a steady torque so the grabber could hold objects without crushing them or slipping. Figure 7 below showcases the grabber mechanism.

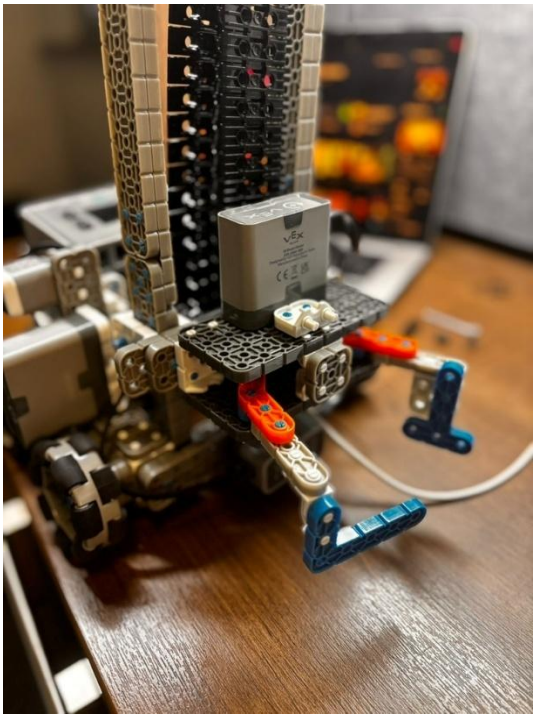


Figure 7 – The grabber mechanism of the robot, located at the front

4.4 – Sensor Attachment Design

The robot utilized 3 distinct sensors for inputs. The touch sensor is positioned for easy access on the side of the robot. It was used for starting the robot's search through the predefined arena. The distance and optical sensors were placed at the bottom of the chassis, under the grabber mechanism. This placement of the two sensors allowed for easy access to obtain inputs from the object. The distance sensor was used to measure how far the robot should stop to pick up the object with the hands of the grabber. The optical sensor was used to capture the hue of the object. Figure 8 below displays all three sensors.



Figure 8 – Color sensor (top-left) [1], distance sensor (top right) [1] and the touch sensor (bottom) [2]

4.5 – Overall Assembly

The final version incorporates the chassis, grabber mechanism and the elevator system into a full stable structure. The elevator mechanism is mounted in the front center of the chassis to ensure a balanced position. The grabber is attached to the elevator's chain system with a shaft and stopper assembly, which allows a secure attachment. The grabber is also connected to a slider device, which surrounds the rails to ensure that the grabber does a consistent smooth vertical motion. The front omniwheels allow for precise and smooth turning, helping the robot accurately perform its turns and maintain its path. The distance and optical sensors were positioned directly under the entire grabber and elevator system, to measure the distance and sense the color of the obstructing objects and allow the grabber to accurately pick up the object. Lastly, the Brain component was implemented at the back of the robot in order to balance the weight of the robot. The elevator and grabber mechanisms created heavy loads in the front and the brain acted as a counterweight in the rear. The whole assembly was incorporated in a fashion that minimized interference between parts to ensure reliability.

5 – Software Design and Implementation

5.1 – VEXcode and Organization

The project was programmed entirely in VEXcode, a C++-based environment that incorporates the VEX application programming interface (API). All source files were written and tested within the VEXcode integrated development environment (IDE).

Organization and planning of the code were primarily done using a flowchart which illustrated the choices and decisions that the robot can take from the beginning to the end of the program. This helped for the program to be simplified to clear and legible blocks.

As the project progressed, the flowchart was updated to more accurately represent the robot's actions and decision-making process. A portion of the finalized flowchart is shown below for demonstration.

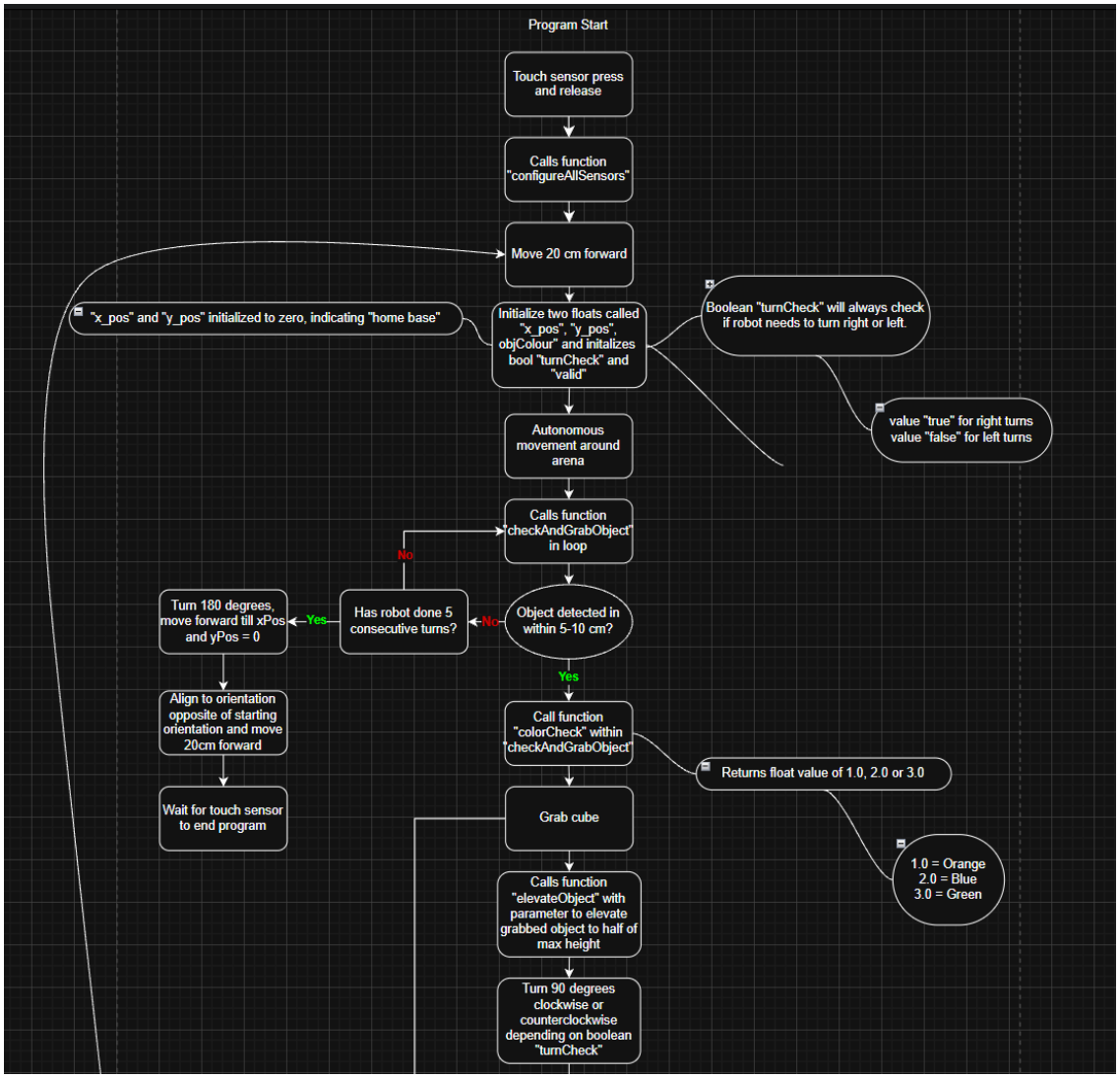


Figure 9 - A portion of the finalized flowchart. The entire flowchart could not be included due to the size

Additionally, a task list was created to provide a summary of the robot's actions throughout the program. Like the flowchart, the tasks list went through several iterations before being finalized to the list below:

1. Press touch sensor to initiate program
2. Wait for 2 seconds to configure all sensors
3. Robot moves 20 cm forward, marks position as Home Plate, and then follows a predefined path.
4. The robot constantly scans for objects in front of its path.
5. When an object is detected, the robot stops and identifies the colour of the object.
6. Pick up the object and raise it to half of the elevator's height.
7. Calculate the shortest non-diagonal path back to the **Home Plate** and follow it
8. Face the orientation opposite of starting orientation and move 20cm forward
9. Calibrate robot to face perfectly straight
10. Check the colour of each bin and place the object in the correctly coloured bin
11. Face initial orientation and move 20 cm forward, and then repeat process again from step 3
12. If no object detected for 5 consecutive turns, stop and play a siren sound before ending the program

The predefined path mentioned in step 3 is the same as the path shown in figure 1.

5.2 – Task Alterations

Throughout the project, several modifications were made to the strategies and actions that the robot would make. Reasons for these changes include limitations with provided sensors, motors, and time constraints. Major changes made are provided below.

5.2.1 – Pathfinding to Home Plate

It was initially planned for the robot to identify its location on the arena with the use of vectors. Home Plate would be defined by the zero vector:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The robot would constantly update its position with reference to the Home Plate by using vectors, which would be calculated by taking its current position in a 2-dimensional x-y plane and its magnitude according to its location on the coordinate plane.

This was ultimately changed due to unnecessary complications. After discussions with the group, it was decided that an x-y coordinate plane would be the best option moving forward. This will later be explained in more detail.

5.2.2 – Varying Bin Heights

Initially, the bins were meant to have different heights so that the robot faced the additional challenge of knowing how high to move the block up before dropping it in the bin. Halfway through the project, the proposal was scrapped due to time constraints. It was then decided that all the bins remain at the same level.

5.2.3 – Error Correction

It was planned that the robot was going to be autonomous throughout the demonstration. It was quickly evident that small errors in the moving and turning of the robot resulted in it drifting off the desired path.

Despite using PIDs for turns, the robot would consistently veer off the path slightly with every turn, making autonomous activity challenging. As a result, over two or more cycles of the program, the robot would be unable to return precisely to its Home Plate. The group proposed that after every complete cycle of the program, the robot would await touch sensor input before repeating the entire program. This would allow the team to manually realign the robot by hand to make sure it is straight, and then press the touch sensor to allow the robot to repeat the program. This ensured the small errors in turning would not build up over the duration of the demonstration.

Despite the idea working, the group decided to find a workaround that would revert the robot back to being fully autonomous. This was done by implementing code into the program that would let the robot automatically correct its orientation once it returns to Home Plate. To do this, the robot would make a slow 360-degree turn and stop when it is at the desired angle, which in the program is at about 0 degrees (Appendix A, lines 515 - 529).

5.2.4 – Color Detection

The group intended to use **opticalSensor.color()** to detect the color of the bins and cubes. When testing this method, it became evident that the colors detected were inaccurate. As a result, the team had to create our own function to detect the color.

Unlike **opticalSensor.Color()**, this function would check the hue of the color detected and calculate which color the hue represents most accurately. Most ranges of hues were sorted into being blue, green or orange (Appendix A, lines 190 – 208).

5.3 – Code

5.3.1 – Program Initiation

To first initiate the program, a user had to press and release the touch sensor on the side of the robot. The robot would then call **void configureAllSensors()**[1] and initialize variables **float x_pos** = 0 , **float y_pos** = 0, **float objColour** = 0, bool **turnCheck** = true and **bool valid** = true (Appendix A, lines 324 - 331).

Float x_pos and **y_pos** are used as the coordinates of the robot. For instance, if the robot moves one unit upwards, then **float y_pos** = 0 gets updated **to y_pos** = 1. **bool turnCheck** lets the robot know whether to turn left or right, and **bool valid** makes it so that the robot only requires the touch sensor to be pressed once at the start of the program, setting it to false after the first cycle of the program.

After initializing all variables, the robot will start moving on its predefined path, shown in figure 1. The flowchart describing the process described above is given below in figure 10.

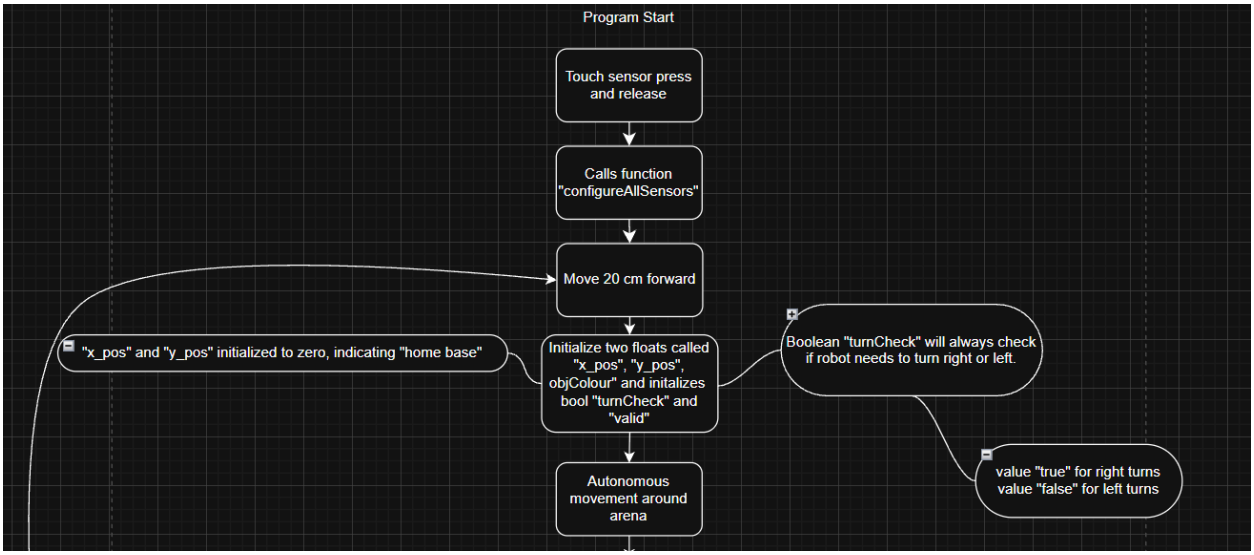


Figure 10 - Section of the flowchart showing the process described in the text above

[1]Provided by Professor Divya

5.3.2 – Predefined Path Following and Obstacle

The robot uses **void turnPID()**[2] to make precise turns when required [3], [4], [5]. This aids in reducing how far the robot veers off the intended path. Additionally, for straight paths, the robot uses motor encoders to move an accurate amount forward (Appendix A, line 365, 379, 381).

As the robot moves on the predefined path, it will constantly call the **bool checkAndGrab()**[2], which will check for objects within 5 cm of the robot with the use of the distance sensor. If nothing is detected and the number of total consecutive turns is less than 5, the function returns false and the robot continues its defined path as usual (Appendix A, lines 365 to 399). Figure 11 given below shows the decisions described above and will also be used to demonstrate later parts of the code.

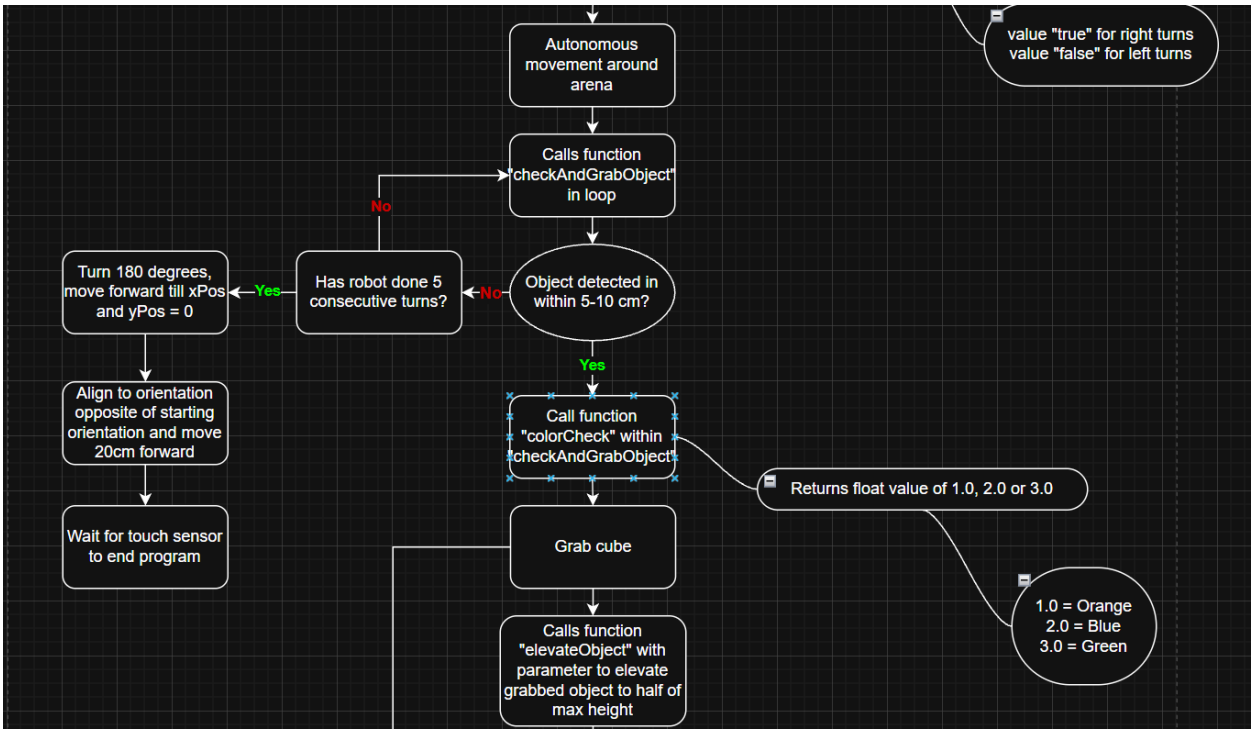


Figure 11 - Flowchart describing the steps above

The nested if condition of whether the robot has performed 5 consecutive turns will be returned to in later portions of the report. When the robot detects an object in front of it, it calls the **float colorCheck()**[3] function within the **checkAndGrab()** function, which returns a **float 1.0, 2.0 or 3.0**[4] which is then defined into **float objColour**. The float values represent different colours. Afterwards, the robot grabs the cube in front and returns true, exiting **checkAndGrab()**.

[2]Created by Alireza Nouri
[3]Created by Armaandeep Singh
[4]A float value of 1.0 represents blue, 2.0 represents green, and 3.0 represents orange

Returning true lets the program know that it has detected and grabbed a cube, signalling the robot to to stop pathfinding return to **Home Plate**. Before returning, the robot calls **void elevObject(1)**[5] with a parameter of 1 to elevate the cube to half the maximum height of the elevator (Appendix A, line 405). This prevents the cube from dragging on the floor.

5.3.2.1 – Testing

During the testing phase, the functions were isolated and run independently to prevent interference from other code.

The **colorCheck()** function was tested by placing coloured sheets of paper in front of the colour sensor and having the function output what colour it detected in real time. The colour sheets consisted of the colour's orange, blue and green as these were the only colours the robot would see.

The **checkAndGrab()** function was tested by placing a coloured cube at varying distances in front of the distance and colour sensor of the robot. Correct behaviour of the function meant that the robot detects and picks up the cube at about 5 centimetres away from the sensor. Additionally, since the **colorCheck()** function resided in the **checkAndGrab()** function, this allowed the team to also test the **colorCheck()** function in standard conditions.

The **elevObject()** function was tested by calling the function with several different parameters ranging from 0 to 2. Correct operation meant that the robot would accurately move to the position specified by the parameter.

5.3.3 – Returning to Home Plate and Drop-Off Plate

Once a cube is grabbed and elevated, the program calls **void returnHome()**[6]**.** The robot will first turn 90 degrees clockwise or counterclockwise depending on **bool turnCheck**. Afterwards, the robot takes the total change in **float x_pos** and travels the same distance forward. Then, the robot performs a 90-degree counterclockwise turn independent of **bool turnCheck** before moving forward the total change in **float y_pos**. The robot ends up at **Home Plate** after completing its movements. Before moving to **Drop-Off Plate**, the robot will first realign itself to the orientation exactly opposite of its starting orientation. This ensures that the robot is accurately lined up with the bins. To move to **Drop-Off Plate**, the robot will then move 20 cm forwards. Figure 12 provides the flowchart of the described code above.

[5]Created by Ayush Gandhi
[6]Created by Sharvesh Kandavel

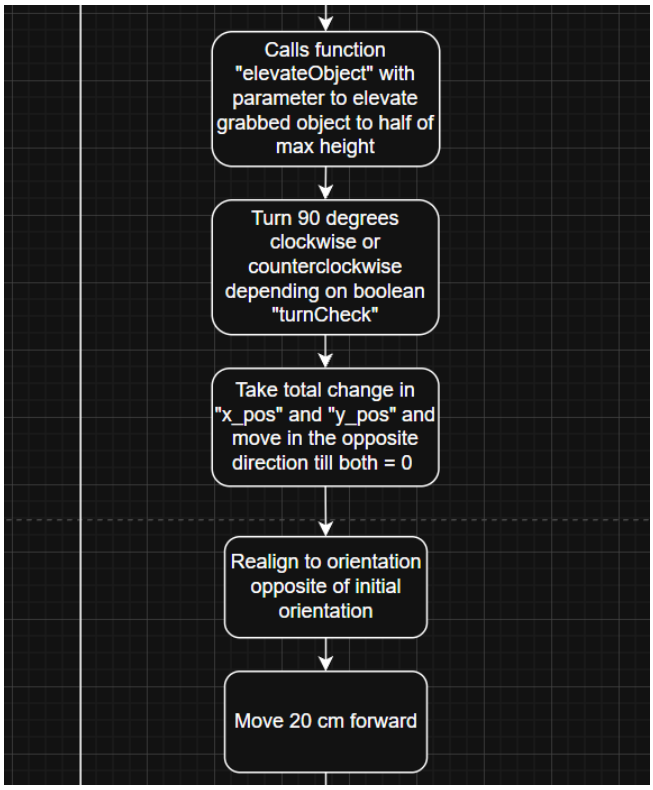


Figure 12 – Flowchart describing the above code

5.3.3.1 – Testing

returnHome() was a very challenging function to perfect. Initial major problems included the robot moving forward significantly more than the value in **float x_pos**. It was discovered that the issue arose due to a stray **wait()** in the code.

After fixing this issue, another issue arose where the robot would not accurately return to **Home Plate** and **Drop-Off Plate**. After several reiterations of the code, it was decided that the best solution would be to allow the robot to make a slow 360-degree turn and stop at the angle it was required to stop at, which, in this case, was 180 degrees opposite of its initial orientation. This solution was not the most efficient, but other alternatives included their own issues and were therefore scrapped.

5.3.4 – Color Sorting Cubes into Bins

Once the robot reaches the **Drop-Off Plate**, it places the cube into the correct bin. The robot first checks the cube's colour using the **colorCheckBin()** function, then uses the **dropping()**[6] function to release it safely. The grabber and elevator motors control the cube's movement, making sure it is placed accurately. After dropping the cube, the robot moves back slightly and resets its position to get ready for the next cube. The code uses sensors and PID turns to guide the robot, ensuring each cube is sorted correctly and the process runs smoothly.

[6]Created by Sharvesh Kandavel

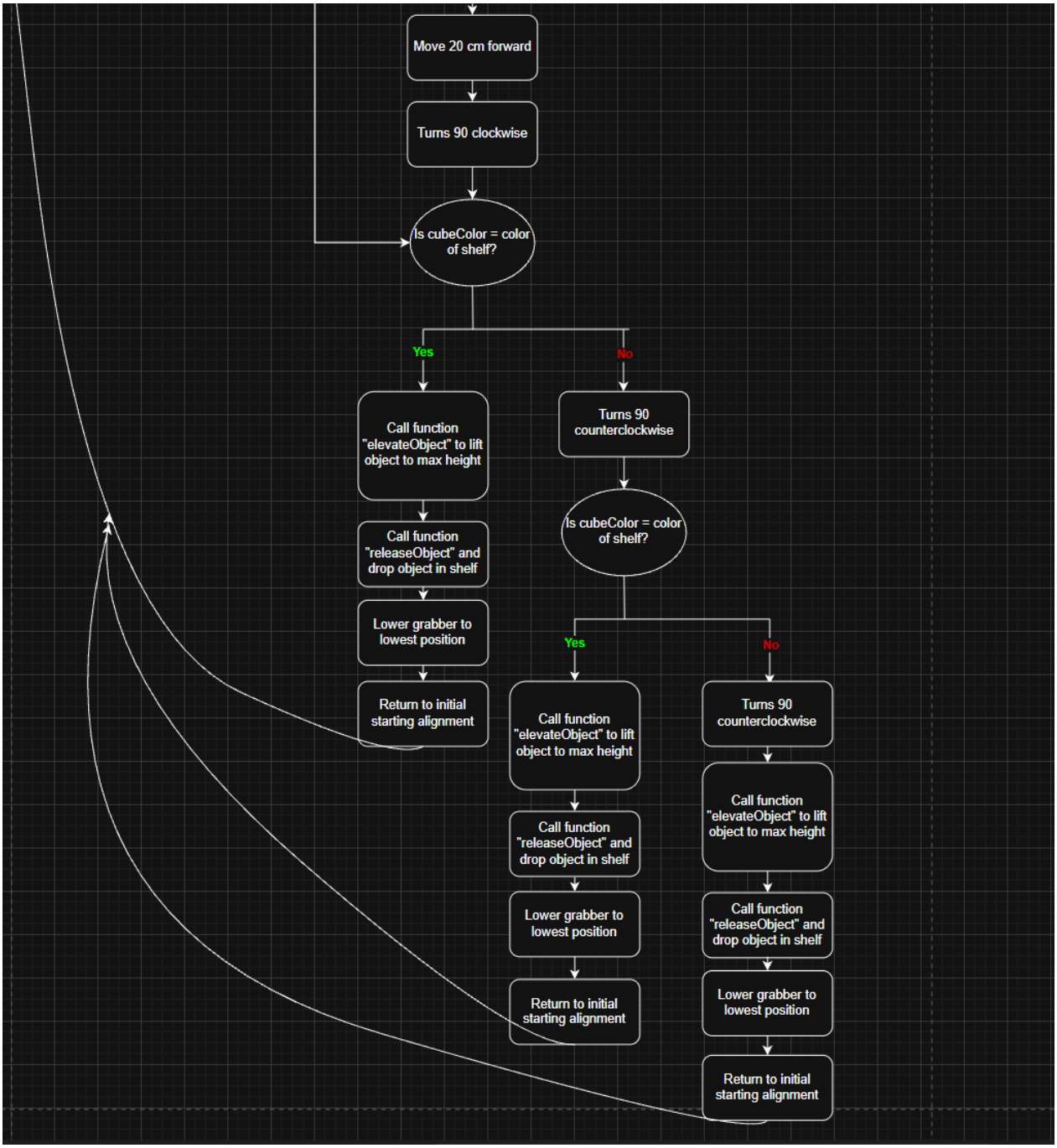


Figure 13 - Portion of the flowchart explaining the code described above

5.3.4.1 – Testing

This portion of the program was tested to ensure cubes were sorted into the correct bins. The team changed the order of the bin colors and tested multiple times while the robot detected the cube color using **colorCheckBin()** and released it with **dropping()**. The elevator height, grabber timing, and turning angles were adjusted during testing to improve accuracy. These tests confirmed that the robot could correctly identify each cube's color and drop it into the right bin even when the bin layout changed.

5.3.5 – Preparing For Next Cycle

After dropping the cube into the bin with the appropriate colour. The robot will then realign itself to the initial orientation and then use the **elevObject(0)** with a parameter of 0 to lower the arm to the lowest position. Afterwards, the program defines **float x_pos** and **float y_pos** to zero and sets **bool turnCheck** to **true**.

Once defining the variables, the robot moves 20 centimetres forward before stopping and realigning itself once more, this time to the initial orientation.

Finally, the robot repeats the entire program from the first while loop, as seen in line 340 of appendix A. For this and the following cycles, touch sensor input is no longer required, and the robot will perform all steps autonomously. Figure 14 shows how the program restarts process.

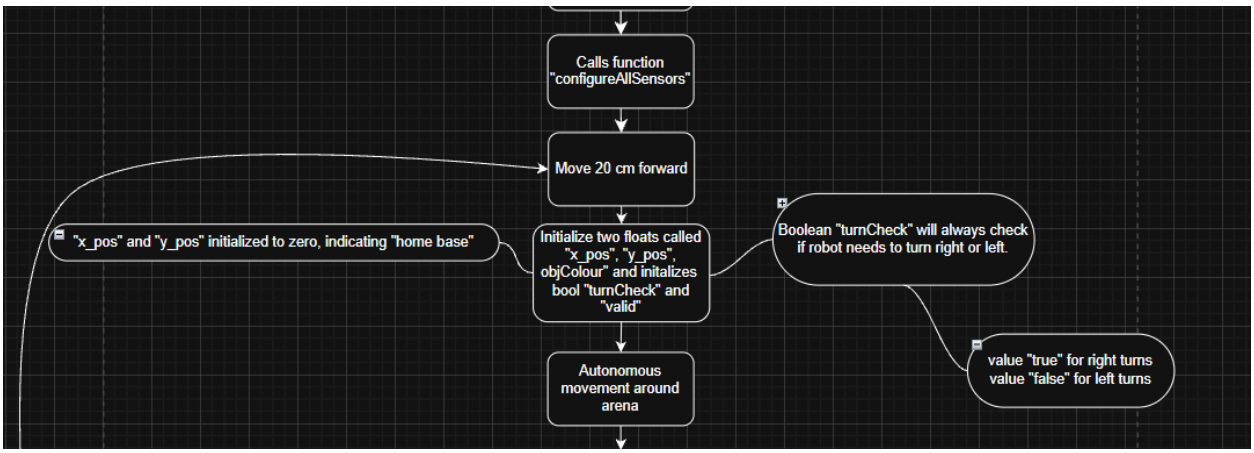


*Figure 14 – The long arrow to the left represents what occurs after the robot is realigned and at **Home Base** after dropping a cube in its appropriate bin*

5.3.5.1 – Testing

To ensure that the robot would properly restart the cycle, it was allowed to continuously pick up and return cubes up to 5 times in a row. Proper functionality of the program meant that the robot would accurately and consistently perform the desired actions in all cycles.

6 – Verification

The team performed in-depth tests in the rectangular serpentine area to evaluate the robot's performance in demonstrations. The tests benchmarked all major aspects of the robot and program, including navigation, sensing, lifting and sorting.

When testing navigation, the robot remained stationary until the touch sensor was activated. Once pressed and released, it followed its encoder-based x and y axis trajectory, consistently returning to the **Home Plate** and **Drop-Off Plate** before restarting its tasks its tasks.

Detecting and sorting the cubes involved placing all cubes on the straight segments of the arena. Furthermore, the location of the bins was changed after every cycle to ensure that the robot would still perform the necessary tasks. As mentioned in section 5, the functions were initially tested independently to avoid interference from other portions of the code.

Mechanical durability was assessed by running the program for prolonged durations. The lift remained within its programmed height limits, and the gears did not skip throughout the entire benchmark. The chassis and grabber remained aligned throughout testing and did not drop or lose cubes during transportation back to **Drop-Off Plate**.

Despite the use of PID for turning, the robot occasionally veered off the intended path. However, these errors did not prevent the robot from accurately returning to **Drop-Off Plate** due to the use of heading calibration. Overall, the system met all criteria established in the project demonstration.

7 – Project Plan

The project was structured to allow mechanical and software development to progress in tandem, with coordination between the two teams to ensure compatibility. While mechanical components were being assembled, software development focused on creating the control logic and required functions for autonomous operation.

Regular communication between the members allowed for adjustments to be made in real time, ensuring that hardware constraints were considered in the programming and that software needs could guide minor mechanical modifications. This parallel approach was aimed to maximise efficiency and maintain steady progress across both major aspects of the project. Deadlines for all the processes were noted and were used as a guideline to ensure that all requirements will be meant for the appropriate dates.

7.1 – Task Distribution

The group divided the work so that every important part of the project was completed properly. Two members worked on the mechanical parts of the robot. One made sure all the components were mounted correctly, the wheels were aligned, and the chassis was strong and easy to fix. The other took care of the electronics, including the sensors, wiring, and power, to make the robot safe and organised.

The other two members focused on coding. They wrote the program that controlled the robot and created the flowcharts, algorithms, and pseudocode to plan how the robot would behave. They also measured and set up the path the robot had to follow, ensuring it was accurate and met the project goals.

One member also helped by designing the outside parts needed for the task in SolidWorks, such as the bin and target objects. These designs were prepared for 3D printing so they would be the right size, strong enough, and able to work smoothly with the robot.

7.2 – Revisions to the Project Plan

Although the intention was for mechanical and software development to progress in parallel, the team soon recognised that the mechanical structure needed to be fully reliable before effective programming and testing could take place. Elevator Stability, chassis stability, and accurate positioning of components required more attention than initially expected. Because of this, the plan was adjusted so that the majority of the group temporarily shifted focus to the mechanical build until it reached a dependable state.

Once the hardware was complete and operating smoothly, the full team moved together into the software phase. This proved beneficial, as coding involved several complicated elements such as precise movement control using PID's, sensor integration and adjustment, as well as task recognition. The revised approach ensured that both the mechanical and coding challenges were addressed thoroughly, ultimately improving the quality and reliability of the finished robot.

7.3 – Planned vs Actual Timeline

The original plan aimed for mechanical and software development to progress in parallel. In practice, the mechanical phase required more dedicated time than expected. This led the team to shift temporarily to a more sequential approach, concentrating on mechanical completion before advancing fully into coding.

This modification in the plan proved beneficial even though it altered the order and timing of the work. Early hardware completion prevented errors brought on by continuous mechanical changes and offered a reliable and consistent platform for software testing. Later, when the team concentrated on programming as a unit, progress was quicker and better organized, enabling the team to successfully complete the project by the deadline.

7.4 – Deadlines and Timings

While the project timeline changed throughout the process, overall, the updated strategy enhanced teamwork, enhanced component integration, and ultimately contributed to a timely and successful outcome.

The team had defined a timeline for events scheduled in line with the MTE100 course expectations and key deadlines. Each milestone was clearly outlined so that all members understood their responsibilities and the required completion dates. This structured approach enabled the group to stay focused, avoid unnecessary delays, and maintain consistent progress.

8 – Conclusion

This project developed a colour-sorting robot that operates on its own, using VEX robotics parts and SolidWorks for custom design. The main goal was to detect coloured objects along a set path, pick them up with a motorized grabber, and place them into the appropriately coloured containers. The system functioned fully on its own by using a distance sensor for detecting objects, a colour sensor for identifying them, and a touch sensor for starting up, along with motors for movement, navigation, elevating, and gripping.

Custom 3D-printed parts and redesigned containers improved the robot's mechanical alignment and overall accuracy in handling objects. Working within a 2 m × 1 m area, the robot reliably placed objects with an accuracy of about ±1 cm, showing good repeatability in controlled conditions. The software used PID-based turning, sensor-driven decision-making, and basic error-recovery routines to improve reliability.

While the robot worked well, some minor issues were noted. About one in every five PID-controlled turns sometimes had slight inaccuracies, usually overshooting or undershooting the target angle. This inconsistency did not affect the robot's ability to place the first object correctly. However, it could shift the alignment during later cycles, causing the robot to stay off its intended path.

Overall, the project showed effective integration of mechanical design, embedded systems, and autonomous control. It met the main functional goals and demonstrated a solid understanding of mechatronics principles, while also revealing clear areas for improvement in future versions such as adding one more colour sensor.

9 – Recommendations

9.1 – Mechanical Improvements

Building a set arena using cardboard or wooden boards with a clearly taped path is strongly recommended. This would create a controlled testing environment, allowing objects to be placed in consistent positions and orientations. A physically defined arena would also make it easier to check the robot's performance, reduce unpredictability during testing, and enhance demonstration accuracy.

Additionally, the robot's turning is off by about ±0.5°. This can be improved by strengthening the chassis to reduce shaking and wheel slippage. Using wheels with better grip and tightening the motor mounts can help the robot move more steadily. The grabber could also be improved by adding rubber or redesigning its shape so that objects do not slip while being carried.

9.2 – Software Improvements

Adding a second colour sensor directed at the floor to detect a taped or painted grid/path could enhance performance. This would allow the robot to use a line-following or grid-following approach along with its current distance-based navigation. By combining floor colour input with object detection, the robot would have better control of its position and would be less likely to stray off the set path.

Furthermore, further tuning of the Proportional-Integral-Derivative (PID) values is recommended to enhance movement accuracy and stability. Adjusting the PID parameters would limit overshooting and undershooting during turns, helping the robot respond more smoothly. Together, improved sensing and better tuning would make the system more reliable and consistent.

If this project were adapted for an industrial setting, the software would change so the robot first scans the boundaries of its working area instead of relying on a pre-defined arena. Using its distance sensors the robot would identify the edges of its workspace and set its own virtual boundaries before starting the sorting process. This would allow the system to adjust to different environments while still maintaining safe and controlled movement. By automatically defining its working limits, the robot would become more robust, flexible, and easier to use in changing conditions without needing manual setup.

10 – Bibliography and Appendix

10.1 – Bibliography

[1]

"Using the IQ Distance Sensor (2nd gen)," VEX Library, Apr. 18, 2025. https://kb.vex.com/hc/en-us/articles/4407296307348-Using-the-IQ-Distance-Sensor-2nd-gen (accessed Dec. 03, 2025).

[2]

"IQ Touch LED," Vexrobotics.com, 2025. https://www.vexrobotics.com/228-3010.html (accessed Dec. 03, 2025).

[3]

"Reddit - The heart of the internet," *Reddit.com*, 2025. https://www.reddit.com/r/vex/comments/fjnjro/what_exactly_is_pid_and_how_do_you_program_it/ (accessed Dec. 03, 2025).

[4]

Youtube.com, 2025. https://www.youtube.com/watch?v=KMSJviT2jI8 (accessed Dec. 03, 2025).

[5]

VEX Robotics, "PID Turning," *VEX Forum*, May 15, 2020. https://www.vexforum.com/t/pid-turning/81196 (accessed Dec. 03, 2025).

10.2 – Appendix A – Complete Program

```
#pragma region VEXcode Generated Robot Configuration
// Make sure all required headers are included.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <string.h>


#include "vex.h"

using namespace vex;

// Brain should be defined by default
brain Brain;


// START IQ MACROS
#define waitUntil(condition)                                                   \
      do {                                                                        \
              wait(5, msec);                                                          \
      } while (!(condition))

#define repeat(iterations)                                                     \
      for (int iterator = 0; iterator < iterations; iterator++)
// END IQ MACROS


// Robot configuration code.
inertial BrainInertial = inertial();
motor MotorLeft = motor(PORT7, false);
motor MotorRight = motor(PORT12, true);
optical OpticalSensor = optical(PORT3);
motor MotorGrabber = motor(PORT2, false);
motor MotorElevator = motor(PORT6, false);
distance Distance1 = distance(PORT1);
touchled touchSensor = touchled(PORT8);


// generating and setting random seed
void initializeRandomSeed() {
      wait(100,msec);
      double xAxis = BrainInertial.acceleration(xaxis) * 1000;
      double yAxis = BrainInertial.acceleration(yaxis) * 1000;
      double zAxis = BrainInertial.acceleration(zaxis) * 1000;
      // Combine these values into a single integer
      int seed = int(
                      xAxis + yAxis + zAxis
                );
      // Set the seed
      srand(seed);
}

// Converts a color to a string
const char* convertColorToString(color col) {
      if (col == colorType::red) return "red";
      else if (col == colorType::green) return "green";
      else if (col == colorType::blue) return "blue";
      else if (col == colorType::white) return "white";
      else if (col == colorType::yellow) return "yellow";
      else if (col == colorType::orange) return "orange";
      else if (col == colorType::purple) return "purple";
      else if (col == colorType::cyan) return "cyan";
      else if (col == colorType::black) return "black";
      else if (col == colorType::transparent) return "transparent";
      else if (col == colorType::red_violet) return "red_violet";
```

```
67      else if (col == colorType::violet) return "violet";
68      else if (col == colorType::blue_violet) return "blue_violet";
69      else if (col == colorType::blue_green) return "blue_green";
70      else if (col == colorType::yellow_green) return "yellow_green";
71      else if (col == colorType::yellow_orange) return "yellow_orange";
72      else if (col == colorType::red_orange) return "red_orange";
73      else if (col == colorType::none) return "none";
74      else return "unknown";
75 }
76
77
78 // Convert colorType to string
79 const char* convertColorToString(colorType col) {
80      if (col == colorType::red) return "red";
81      else if (col == colorType::green) return "green";
82      else if (col == colorType::blue) return "blue";
83      else if (col == colorType::white) return "white";
84      else if (col == colorType::yellow) return "yellow";
85      else if (col == colorType::orange) return "orange";
86      else if (col == colorType::purple) return "purple";
87      else if (col == colorType::cyan) return "cyan";
88      else if (col == colorType::black) return "black";
89      else if (col == colorType::transparent) return "transparent";
90      else if (col == colorType::red_violet) return "red_violet";
91      else if (col == colorType::violet) return "violet";
92      else if (col == colorType::blue_violet) return "blue_violet";
93      else if (col == colorType::blue_green) return "blue_green";
94      else if (col == colorType::yellow_green) return "yellow_green";
95      else if (col == colorType::yellow_orange) return "yellow_orange";
96      else if (col == colorType::red_orange) return "red_orange";
97      else if (col == colorType::none) return "none";
98      else return "unknown";
99 }
100
101
102 void vexcodeInit() {
103
104     // Initializing random seed.
105     initializeRandomSeed();
106 }
107
108 #pragma endregion VEXcode Generated Robot Configuration
109
110 //----------------------------------------------------------------------------
111 //
112 //    Module:       main.cpp
113 //    Author:       Group 20
114 //    Created:      {date}
115 //    Description:  Color Sorting
116 //
117 //----------------------------------------------------------------------------
118 #include "iq_cpp.h"
119 using namespace vex;
120
121
122 void configureAllSensors() {
123     BrainInertial.calibrate();
124     wait(2,seconds);
125     BrainInertial.setHeading(0,degrees);
126     BrainInertial.setRotation(0,degrees);
127     MotorLeft.setPosition(0,turns);
128     MotorRight.setPosition(0,turns);
129     Brain.Screen.clearScreen();
130     Brain.Screen.setFont(mono15);
131     OpticalSensor.setLight(ledState::on);
132 }
133
134 void turnPID(float relativeAngle) { // Author : Alireza Nouri
135     const float kP = 1.2;
```

```
136     const float kI = 0.0;
137     const float kD = 0.15;
138
139     const float dt = 0.02; // 20 ms
140     const float tolerance = 1.0;
141     const float settleTime = 0.20;
142
143     float startAngle = BrainInertial.rotation(degrees);
144     float target = startAngle + relativeAngle;
145
146     float error = 0;
147     float prevError = 0;
148     float integral = 0;
149
150     float stableCounter = 0;
151     bool done = false;
152
153     while (!done) {
154             float current = BrainInertial.rotation(degrees);
155             error = target - current;
156
157             integral += error * dt;
158             float derivative = (error - prevError) / dt;
159             prevError = error;
160
161             float power = kP * error + kI * integral + kD * derivative;
162
163             if(power > 60) power = 60;
164             if(power < -60) power = -60;
165
166             float minPower = 8;
167             if (fabs(error) > 3.0) {
168                     if(power > 0 && power < minPower) power = minPower;
169                     if(power < 0 && power > -minPower) power = -minPower;
170             }
171
172             MotorLeft.spin(forward,  power, percent);
173             MotorRight.spin(forward, -power, percent);
174
175             if (fabs(error) <= tolerance)
176                     stableCounter += dt;
177             else
178                     stableCounter = 0;
179
180             if (stableCounter >= settleTime)
181                     done = true;
182
183             wait(dt, seconds);
184     }
185
186     MotorLeft.stop(brake);
187     MotorRight.stop(brake);
188 }
189
190 float colorCheck() { // Author : Armaandeep Singh
191     int hue = OpticalSensor.hue();
192     // Get the color detected
193     if (hue >= 70 && hue <= 130) { //hue values for green
194             return 3.0;
195     }
196     //hue value: Blue = (140 - 250)
197     else if (hue >= 140 && hue <= 250) {
198             return 2.0;
199     }
200     //hue value: Orange = (0 - 40)
201     else if (hue >=0 && hue <=40) {
202             return 1.0;
203     }
204     // Wrong Color Detected
```

```
205     else {
206             return 67.0;
207     }
208 }
209
210 bool checkAndGrab(float& ObjColour) { // Author : Alireza Nouri
211     if(OpticalSensor.isNearObject() && Distance1.objectDistance(mm)< 55) {
212             ObjColour = colorCheck();
213             MotorLeft.stop();
214             MotorRight.stop();
215             wait(1,seconds);
216             MotorGrabber.spin(forward);
217             wait(1,seconds);
218             MotorGrabber.stop(brake);
219             MotorGrabber.stop(hold);
220             return true;
221
222     } else {
223             return false;
224     }
225 }
226
227 void elevObject (int motorpower, int position) {// Author : Ayush Gandhi
228     MotorElevator.setPosition(0,turns);
229     MotorElevator.setVelocity(motorpower, percent);
230     MotorElevator.spin(forward);
231
232     while (MotorElevator.position(turns)<position) {}
233
234     MotorElevator.stop(hold);
235     MotorGrabber.setVelocity(motorpower,percent);
236     MotorGrabber.spin(reverse);
237 }
238
239 void returnHome(float X_pos,float Y_pos,bool turnCheck) { // Author : All Group Members
240     if (turnCheck == true) {
241             turnPID(-90);
242     } else if (turnCheck == false) {
243             turnPID(90);
244     }
245
246     MotorLeft.setPosition(0,turns);  // Moves forward the base
247     MotorRight.spin(forward);
248     MotorLeft.spin(forward);
249
250     while(MotorLeft.position(turns) < X_pos)
251     {}
252
253     MotorLeft.stop(brake);
254
255     if (turnCheck == true) {
256             turnPID(-90);
257     } else if (turnCheck == false) {
258             turnPID(-90);
259     }
260
261     MotorLeft.setPosition(0,turns);    // To reach the Home Plate
262     MotorRight.spin(forward);
263     MotorLeft.spin(forward);
264
265     while(MotorLeft.position(turns) < Y_pos) {}
266
267     MotorRight.stop(brake);
268     MotorLeft.stop(brake);
269
270     wait(1, seconds);
271 }
272
273 void dropping (int motorpower, float &x ) { // Author : Sharvesh Kandavel
```

```
274     float dis = MotorLeft.position(turns);
275     MotorLeft.spin(forward);
276     MotorRight.spin(forward);
277
278     while(OpticalSensor.isNearObject() == 0 || Distance1.objectDistance(mm) >15) {}
279
280     MotorLeft.stop(brake);
281     MotorRight.stop(brake);
282
283     MotorGrabber.spin(reverse);
284     wait(1.5, seconds);
285     MotorGrabber.stop(brake);
286     MotorGrabber.stop(hold);
287
288     MotorLeft.spin(reverse);
289     MotorRight.spin(reverse);
290     while(MotorLeft.position(turns) > dis) {}
291     MotorLeft.stop();
292     MotorRight.stop();
293
294 }
295
296 float colorCheckBin(float &ObjColor, float &x) { // Author : Sharvesh Kandavel
297     float BinColor = 0;
298
299     BinColor = colorCheck();
300
301     if(BinColor == ObjColor) {
302             Brain.Screen.print("orange");
303             return 1.0;
304     }
305
306     turnPID(90);
307     BinColor = colorCheck();
308
309
310     if (BinColor == ObjColor) {
311             Brain.Screen.print("green");
312             return 2.0;
313     }
314
315     BinColor = colorCheck();
316     return 3.0;
317 }
318
319
320 int main() {
321     // Initializing Robot Configuration. DO NOT REMOVE!
322     vexcodeInit();
323
324     configureAllSensors();
325     // Begin project code
326     //Touch Led :: To start
327     bool turnCheck = true;
328     bool valid = true;
329     float x_pos = 0;
330     float y_pos = 0;
331     float ObjColour = 0;
332
333
334     MotorLeft.setVelocity(70,percent);
335     MotorRight.setVelocity(70,percent);
336     MotorElevator.setVelocity(70,percent);
337     OpticalSensor.setLight(ledState::on);
338
339
340     while (valid == true) {
341             while(!touchSensor.pressing()) {}
342
```

```
343             while(touchSensor.pressing()) {}
344
345             MotorLeft.setPosition(0,turns);
346             MotorLeft.spin(forward);
347             MotorRight.spin(forward);
348
349             while(MotorLeft.position(turns) < 2) {}
350
351             MotorLeft.stop(brake);
352             MotorRight.stop(brake);
353
354             wait(1,seconds);
355
356             while (x_pos < 7.0 && y_pos>-0.1) {
357
358                     bool grabbed = false;
359                     MotorRight.setPosition(0,turns);
360                     MotorLeft.setPosition(0,turns);
361                     MotorLeft.spin(forward);
362                     MotorRight.spin(forward);
363
364
365                     while (MotorLeft.position(turns) <= 3.95 && grabbed==false) {
366                             grabbed = checkAndGrab(ObjColour);
367                     }
368
369                     float distanceTraveled_y = MotorLeft.position(turns);
370
371                     if (turnCheck) y_pos += distanceTraveled_y;
372                     else y_pos -= distanceTraveled_y;
373                     //keep track of y_position in these three lines
374                     if (grabbed == false) {
375                             MotorLeft.stop();
376                             MotorRight.stop();
377
378                             if (turnCheck) {
379                                     turnPID(90);
380                             } else if (!turnCheck) {
381                                     turnPID(-90);
382                             }
383
384                             MotorRight.setPosition(0,turns);
385                             MotorLeft.spin(forward);
386                             MotorRight.spin(forward);
387
388                             while(MotorRight.position(turns) < 1.5) {}
389
390                             float distanceTraveled_x = MotorRight.position(turns);
391                             x_pos += distanceTraveled_x ;
392                             //keep track of x_position in these above two lines
393                             if(turnCheck) {
394                                     turnPID(90);
395                                     turnCheck = false;
396                             } else if(!turnCheck) {
397                                     turnPID(-90);
398                                     turnCheck = true;
399                             } else return 99999;
400
401                     } else {
402
403
404
405                             elevObject(1);
406
407
408
409                             MotorLeft.stop(hold);
410                             MotorRight.stop(hold);
411                             float temp = x_pos*0.80;
```

```
412                             Brain.Screen.print("%f ", temp);
413                             returnHome(temp,y_pos,turnCheck);
414                             MotorLeft.setVelocity(10,percent);
415                             MotorRight.setVelocity(10,percent);
416
417                             if(BrainInertial.rotation(degrees) < 180) {
418                                     MotorRight.spin(reverse);
419                                     MotorLeft.spin(forward);
420                                     while(BrainInertial.rotation(degrees) < 180 ) {}
421
422                                     MotorLeft.stop();
423                                     MotorRight.stop();
424                             } else if(BrainInertial.rotation(degrees) > 180) {
425                                     MotorRight.spin(forward);
426                                     MotorLeft.spin(reverse);
427
428                                     while(BrainInertial.rotation(degrees) > 180 ) {}
429
430                                     MotorLeft.stop();
431                                     MotorRight.stop();
432                             }
433
434                             MotorLeft.setVelocity(70,percent);
435                             MotorRight.setVelocity(70,percent);
436                             MotorLeft.stop(brake);
437                             MotorRight.stop(brake);
438
439                             wait(0.5,seconds);
440
441                             MotorLeft.setPosition(0,turns);
442                             MotorLeft.spin(forward);
443                             MotorRight.spin(forward);
444
445                             while(OpticalSensor.isNearObject()!=1  ||  Distance1.objectDistance(mm)  >
80) {}
446
447                             MotorLeft.stop(brake);
448                             MotorRight.stop(brake);
449                             Brain.Screen.print("ooga booga");
450                             BrainInertial.rotation(degrees);
451                             float max = 0.5;
452                             int s = MotorElevator.position(turns);
453                             MotorElevator.spin(forward);
454
455                             while (MotorElevator.position(turns)< max+s) {}
456
457                             MotorElevator.stop(brake);
458                             MotorElevator.stop(hold);
459                             float Colour = colorCheckBin(ObjColour,x);
460
461                             wait(2,seconds);
462
463                             MotorLeft.spin(forward);
464                             MotorRight.spin(forward);
465
466                             while(OpticalSensor.isNearObject()!=1) {}
467
468                             MotorLeft.stop(brake);
469                             MotorRight.stop(brake);
470
471                             dropping (50,x);
472
473                             Brain.Screen.newLine();
474                             Brain.Screen.print("We dropped");
475                             Brain.Screen.clearLine(1);
476                             MotorLeft.stop(brake);
477                             MotorRight.stop(brake);
478
479                             if (Colour ==1.0) {
```

```
480                                     Brain.Screen.newLine();
481                                     Brain.Screen.print("Orange");
482                                     turnPID(180);
483                             } else if (Colour==2.0) {
484                                     Brain.Screen.newLine();
485                                     Brain.Screen.print("Blue");
486                                     turnPID(90);
487                             } else if (Colour == 3.0) {
488                                     Brain.Screen.newLine();
489                                     Brain.Screen.print("Green");
490                                     turnPID(-90);
491                             } else return 9999;
492
493
494
495                             eleveObject(0);
496
497                             turnCheck = true;
498                             MotorLeft.setPosition(0,turns);
499                             MotorLeft.spin(forward);
500                             MotorRight.spin(forward);
501
502                             while(MotorLeft.position(turns) < 2) {}
503
504                             MotorLeft.stop(brake);
505                             MotorRight.stop(brake);
506
507                             wait(2,seconds);
508
509                             x_pos = 0;
510                             y_pos=0;
511                             MotorLeft.setVelocity(10,percent);
512                             MotorRight.setVelocity(10,percent);
513
514
515                             if(BrainInertial.rotation(degrees) > 0 ) {
516                                     MotorRight.spin(reverse);
517                                     MotorLeft.spin(forward);
518
519                                     while(BrainInertial.rotation(degrees) > 2 ) {}
520
521                                     MotorLeft.stop();
522                                     MotorRight.stop();
523                             } else if(BrainInertial.rotation(degrees) > 180) {
524
525                                     MotorRight.spin(forward);
526                                     MotorLeft.spin(reverse);
527                                     while(BrainInertial.rotation(degrees) < 2 ) {}
528                                     MotorLeft.stop();
529                                     MotorRight.stop();
530                             }
531                             MotorLeft.setVelocity(70,percent);
532                             MotorRight.setVelocity(70,percent);
533                     }
534                     Brain.playSound(siren);
535                     valid = false;
536             }
537             Brain.programStop();
538     }
539 }
```