# Team A10: Vex Project Part Two

## Introduction

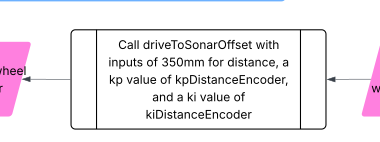
In part two of the VEX Project, each group was tasked with programming an automated forklift to navigate a warehouse facility, to move stock from one location to another, in less than 120 seconds. To easily simulate this task, our solution is to be demonstrated by a VEX robot travelling along a layout map of the warehouse.

As the warehouse facility contains stock shelves, and other obstacles, as well as the payload being fragile, it is required that the robot navigates with precision and accuracy. However, from our previous experience in part 1 of the VEX project, we found that achieving precise and accurate movement with a basic (open loop) robot is difficult, as even supposedly identical motors and systems differ in performance when given identical inputs.

To develop a fully autonomous solution for Dairy Industry Cooperative that is both accurate and precise, we used a closed loop control system, using built in encoders and sensors in the robot to make decisions and keep track of the exact direction and position of the robot.

## Solution

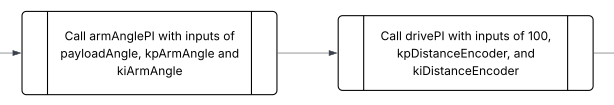
For our solution, we will go through our program step by step, looking into the features and logic the code, to show how the full run is put together.



Our solution starts by placing the robot inside the starting zone, while ensuring the turning pivot of the robot starts behind the brown line. This is to account for any backlash in turning and will be explained further later. Then, using our driveToSonarOffset function (above) which uses a sonar sensor to detect the robot’s distance from the payload, and then calls our function DrivePI, which drives it the required distance to the payload. The distance of 350mm from the payload is to ensure the hook of the arm can be lowered without hitting the payload eye.

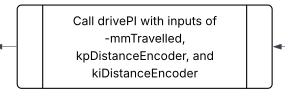
Our drivePI function uses two PI controllers, one to control the distance travelled, and one to ensure both wheels travel at the same angular velocity. As a PI controller integrates error over time, the accumulated error grows even when the actuators are saturated, leading to large overshoot, as the accumulated error takes long time to decrease, even when our robot has overshot its reference. To account for this, we decided to limit the accumulated error to an absolute value of 3000 mm∙ms, which, when maxed out, gives our integral controller enough power to overcome steady state error, while still being small enough to cause no windup or overshoot

Secondly, as the error at the start was large, our actuators would instantly saturate, leading to a jerking/twitching motion at the beginning of acceleration. By starting an internal timer every time, the function is called, and for the first second of acceleration, multiplying the power sent to the motors by the fraction of the second which has passed.



To not interfere with the sonar sensor readings, we intentionally drive 100 mm short of the detected distance to the payload (above), before lowering the arm to the height of the payload eye, using the armAnglePI function. The armAnglePI function works by using a PI controller to set the arm angle, by resetting the arm at the maximum angle, and using a conversion of encoder counts to degrees to determine the exact position of the arm

After the arm is lowered to the height of the payload eye, the robot drives forward 100mm, and then using the armUp function raises the arm to its max height, picking up the payload.



To travel back to our starting position, we recorded the distance travelled towards the payload and then use our DrivePI function to travel backwards the same distance, so we return exactly to our starting point (above).

We decided to travel along the line following path first, as our line following function was consistently able to line the robot up with the drop zone, allowing us to more accurately place the load in the intended drop zone compared to using our rotateAnglePI and drivePI functions alone.

To first reach the brown line, once we have driven back to our starting point, we rotate the robot 90 degrees clockwise using our rotateAnglePI function. This function consists of a simple PI controller and uses the same anti-windup feature as in the DrivePI function, along with an offset of 5 degrees in the direction of rotation to account for the backlash when turning. This backlash turned out to be a major issue when dealing with our robot, and we found that this offset was accurate enough for this robot, however we acknowledge that this problem is unavoidable with the hardware available.

As we knew turning a certain angle would not be perfectly accurate with the hardware available, we opted to place our robot slightly left of the brown line, by starting it in a position where the pivot is behind the brown line, we ensure that when it turns anticlockwise, it will face slightly left of the brown line.

We then use our driveUntilBlackPI function which uses the original drivePI logic to keep the robot travelling in a straight line. Alongside this, the function uses additional logic which detects for any light sensor changes which indicates that the black line is underneath the robot. This detection will then trigger the exit condition for the loop and stop the robot right on the line.

After stopping on the line, we then use drivePI to drive forwards 10mm to get away from the black line before using our line following function. This is to avoid an accidental trigger of the black detection logic within the line following function, which will be explained in more detail in the next section. We then start our lineFollowingPI function to follow the line to the other side of the area.

#### Line Following Flowchart:

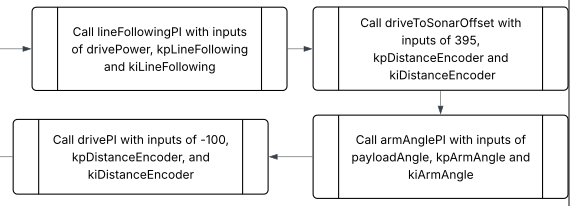
Our lineFollowingPI function uses an edge following technique, with additional logic to find the line if it has been lost. The advantage of using a PI controller for line following is that the accuracy vs a basic controller is increased significantly. The PI controller smoothly follows the line in straight sections and helps to line up the robot with the payload area.

Additionally, we added basic logic for when the line is lost entirely, which tracks the last side of the line which was detected and then determines which way to turn the robot to find the line once again. This logic adds a level of robustness to the function to help avoid scenarios where the robot is misguided and is not able to find the line again as well as when the robot needs to turn corners and angles, making the run more consistent.

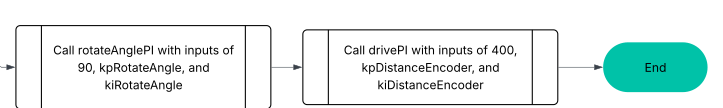
Directly after starting our line following function, we set the value of our basic logic to “Has not seen the line yet” which starts the robot’s logic to find the line initial. This simply uses a mix of driving forwards and turning slightly right until the line is found. Finally, we used a looping check for black function, which is used each cycle to detect a change in the light sensor that indicates the presence of the black indicator line, which indicates the end of the line following sequence.

#### Flowchart for checkForBlack:

After successfully detecting black, our robot is in a position directly in line with the target bullseye, allowing us to simply travel forwards to drop off the payload. To do this, we call the driveToSonarOffset (below), with an offset which lands the payload directly in the middle and then proceeds to the drop off zone. As we approach the zone, the robot starts slowing down due to the PI controller and stops in the position to drop off the payload.



We then employ our armAnglePI function (above), which sets the robots arm to the payload drop off angle and then proceeds with our clockwise route by using the drivePI function to reverse 100mm. After the robot has reversed, for safety reasons, we call the armUp function to set the robots arm to the up position, this being set to remove any collisions with staff or objects whilst traversing the floor which is required for both the real-life scenario and scale model.



Following this, we employ the rotateAnglePI function to turn ninety degrees, then use the drivePI function to move the robot's position to be directly in line with the finish area.

Finally, we once again use the rotateAnglePI function to turn ninety degrees, this time directly towards the finish zone, and employ drivePI for the last time, moving directly into the finish zone, where the robot stops and the main routine ends.

### Main Function Flowchart

## Reflection

Looking back at the demonstration day, our robot performed to the level we expected, with a few minor issues and difficulties which ultimately stopped us from achieving the time bonus.

During our final hands-on session before observation day, we managed to complete the full program whilst battling a few bugs in our code which had been preventing us from completing our runs. Our main issue involved a bug with the drive until black function which prevented our robot from self-correcting, thus stopping the robot from consistently finding the black line for line following and occasionally making the robot fall off the mat whilst finding the line.

For our first observation session, our goal was to complete as many runs as possible with the original code whilst aiming to understand where we could reduce time loss to achieve the time goal in the second session. We managed to get a good run at both the start and end which gave us a good overall score to then go for the time bonus in the second session.

In between the sessions, we increased the speed of most functions which surprisingly created a more consistent program and was ultimately just under the time limit depending on how long the robot took to find the brown line after stopping at the black line.

In the second session, we ultimately did not achieve the time bonus and only finished the course one time due to an unfortunate error in the positioning of the robot in certain areas of the line following. The robot seemed to be very consistent at hitting the line on a strange angle, thus inducing a full 180-degree rotation to where the robot would begin travelling backwards on the line. This was quite unfortunate, however ultimately, we were happy with our score and the robot’s performance.

#### Key Improvements:

Key changes we could have made involve an improved version of the line following function, tuning values for a higher saturation speed of the motors and adding in a drive function to align the robot with the payload target better.

By creating an improved version of the line following, we would have been able to speed up the robot more through the line following section and decrease the overall time of the run. This improvement would have also been designed to account for all eventualities and therefore remove the weird bug we had in the second observation session.

During our observations, we had consistent accuracy however due to the line following being on the left edge, we constantly were slightly left of the bullseye. By adding in another drive function in-between stopping at the black after line following and the drive forward to drop payload, we would have been able to account for following the line on the left edge.

In conclusion, whilst we did have a few potential updates and changes we could have made, we are confident that our designed solution fit the brief and would be able to be up scaled to fit the needs of the client.