ECE4078: Final Demonstration Report
Group 2_05

First Name	Last Name	Email address	Student ID
Maitri	Justitian	mjus0002@student.monash.edu	29438330
Evan	Tan	etan0008@student.monash.edu	27401995
Lucas	Tobar	ltob2@student.monash.edu	27860221

Function we are most proud of:
Our object detection algorithm uses three methods to estimate object pose; nonlinear regression, scale factor multiplication, and the intrinsic camera properties (focal length). This resulted in the maximum number of estimations allowed for each individual object detected of three. A YOLOv4-Tiny darknet implementation was used to detect objects and their respective bounding boxes. In all methods, the object position was only recorded when it was positioned close to the exact centre of the camera frame. The three implementations returned an estimated distance between the robot and the object, and then the pose of the object in the world frame was calculated by using the current robot pose as described by the following equations:

$$x_{obj} = dist * \cos(\theta_{robot}) + x_{robot}$$
$$y_{obj} = dist * \sin(\theta_{robot}) + y_{robot}$$

For the nonlinear regression estimation, polynomial coefficients were found by fitting a curve in MATLAB to bounding box height in pixels against Euclidean distance in Gazebo as seen in the **figure 1**. The coefficients were then input into a python function to estimate distance based on the bounding box height during the demonstration.

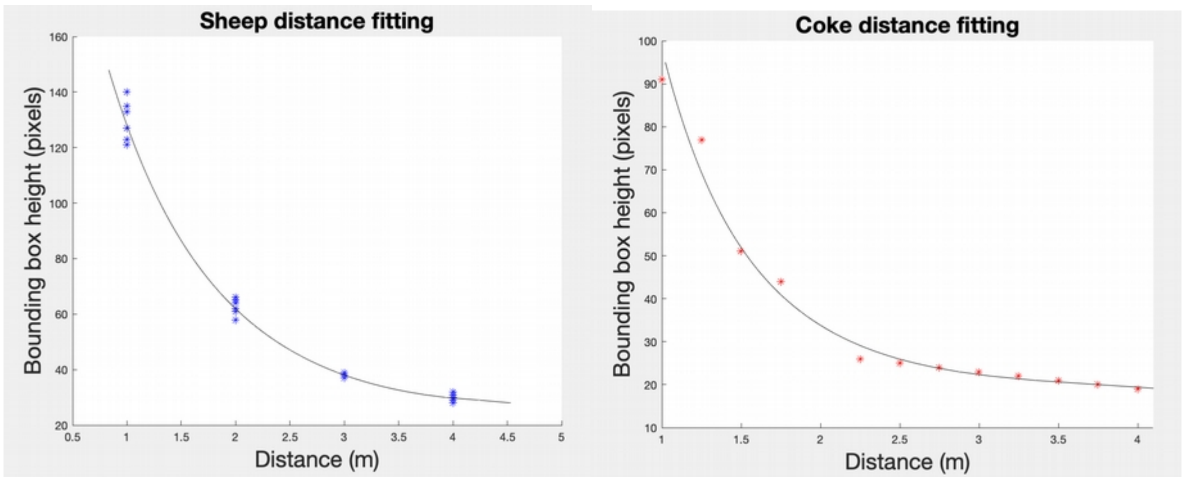


Figure 1: *Nonlinear regression fitting of bounding box height vs distance for sheep (left) and coke cans (right).*

For the scale factor multiplication approach, linearity is assumed in using the similar triangles concept. Using bounding box height at 1m, the distance to object could be determined based on similar triangles i.e. how much larger or smaller a bounding box is compared to that at 1m, to determine if the object is closer or further away respectively, as described by the following equation:

$$dist = scale\ factor / height\ of\ bounding\ box$$

For the focal length approach, a pinhole camera model was assumed, and focal length was calculated on the fly. Using the bounding box dimensions from Gazebo, object distance was calculated using both width and height, and the average of the two was taken. This is described by the following equations:

$$LH_{focal} = frame\ height / (2 * \tan(FOV_{vertical}/2))$$
$$LW_{focal} = frame\ width / (2 * \tan(FOV_{horizontal}/2))$$
$$dist_H = true\ object\ height * LH_{focal} / height\ of\ bounding\ box$$
$$dist_W = true\ object\ width * LW_{focal} / width\ of\ bounding\ box$$
$$dist = (dist_H + dist_W)/2$$

Functions that did not work as expected:

In general, our final demonstration went very well, and all the aruco markers and objects were detected within 1m of their actual pose, resulting in 100% accuracy. Thus, almost nothing went wrong, however, something that we didn't expect from our functions was for the non-linear regression model to be on average, just as accurate as our other two models for estimating object pose. As seen in **figure 2**, the average absolute error for both the sheep and coke can estimates are fairly similar for the non-linear regression, scale factor, and focal length estimations. In addition, it was found that the scale factor estimate was actually much better at detecting objects from a large distance than the other two estimates. This is surprising as the non-linear regression model should, in theory, be more accurate than all the other models, however it seems to have only been slightly better on average. This could be due to a range of factors, including overfitting and small random bounding box height variations causing large changes in the distance estimations. The best way to combat this would be to first increase the resolution of the images, which would make the bounding boxes more precise. Another solution would be to take the average bounding box height of the object from multiple scans, thus making the heights more accurate, and in turn the distance estimates more accurate.

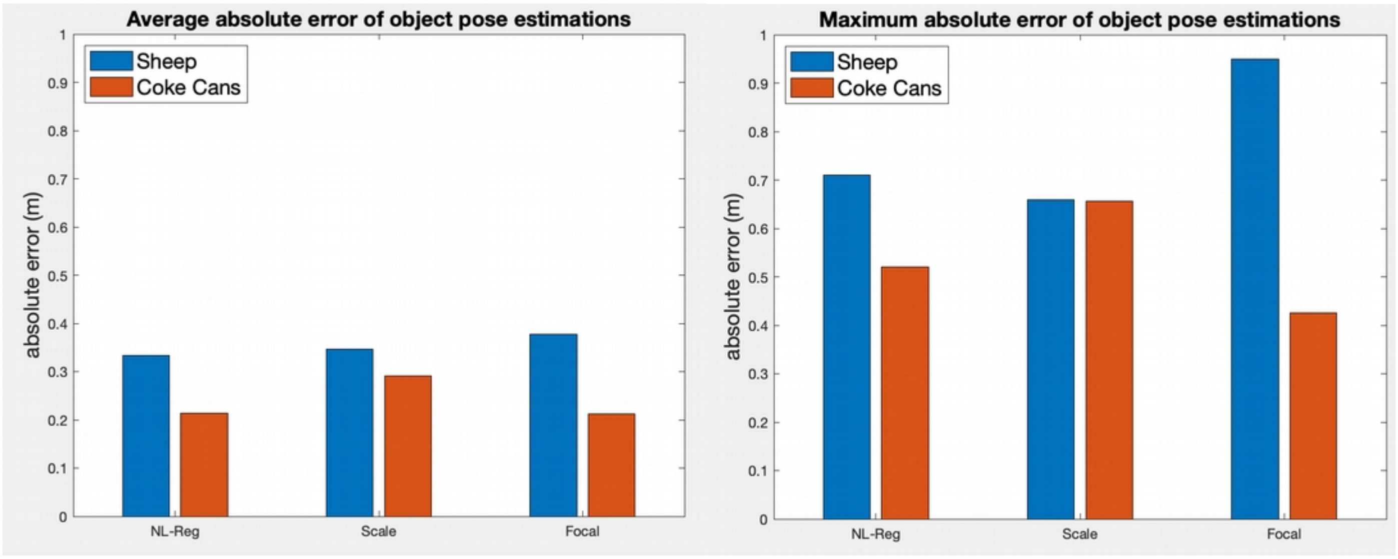
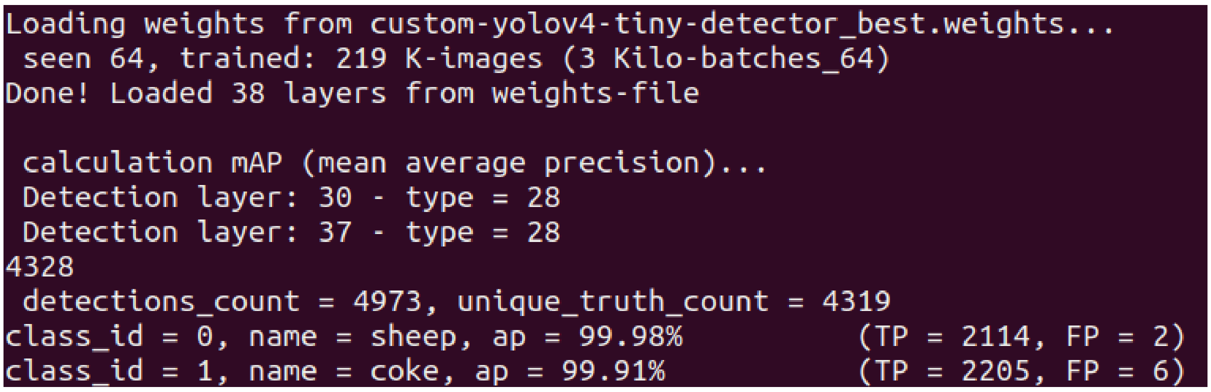


Figure 2: *Absolute error summary from the final demonstration for our object pose estimates, using three different estimation methods; Non-linear regression model (NL-Reg), scale factor model (Scale) and focal length model (Focal). Average error (left) and maximum error (right).*

Possible Improvements:

Improvement 1: YOLOv4-Tiny Accuracy



```
Loading weights from custom-yolov4-tiny-detector_best.weights...
 seen 64, trained: 219 K-images (3 Kilo-batches_64)
Done! Loaded 38 layers from weights-file

 calculation mAP (mean average precision)...
 Detection layer: 30 - type = 28
 Detection layer: 37 - type = 28
4328
 detections_count = 4973, unique_truth_count = 4319
class_id = 0, name = sheep, ap = 99.98%          (TP = 2114, FP = 2)
class_id = 1, name = coke, ap = 99.91%           (TP = 2205, FP = 6)
```

Figure 3: *darknet recall of our trained weights, without a test/validation set.*

The YOLOv4-Tiny model was trained using 2211 images of coke, 2119 images of sheep, with a 15.14% false positive rate as seen in **figure 3**. At large distances (approximately 4m and greater) sheep are sometimes labeled as coke cans. Coke cans are also not detected at a certain mid-range distance from the robot. These situations could be avoided by providing a larger dataset with more backgrounds from the M5 arenas, and creating test/validation datasets to evaluate performance of trained weights in order to accurately determine true average precision values (ap) unlike those seen in **figure 3** which inaccurately measure performance since the training dataset is used to test mean average precision (mAP). Using the M5 arenas for data collection would also bridge the gap between predictions of objects located from 5.5m to 12m away from the robot, seeing as the M2 arena (which was used to collect the images of the objects) only allowed up to approximately 5.5m of distance between the robot and the object. Additionally the 38 layer YOLOv4-Tiny model could be tuned by using different activation functions and thresholds, or adding an extra YOLO layer in order to improve the accuracy of detecting small objects, effectively increasing object detection range.

Improvement 2: Driving Velocity

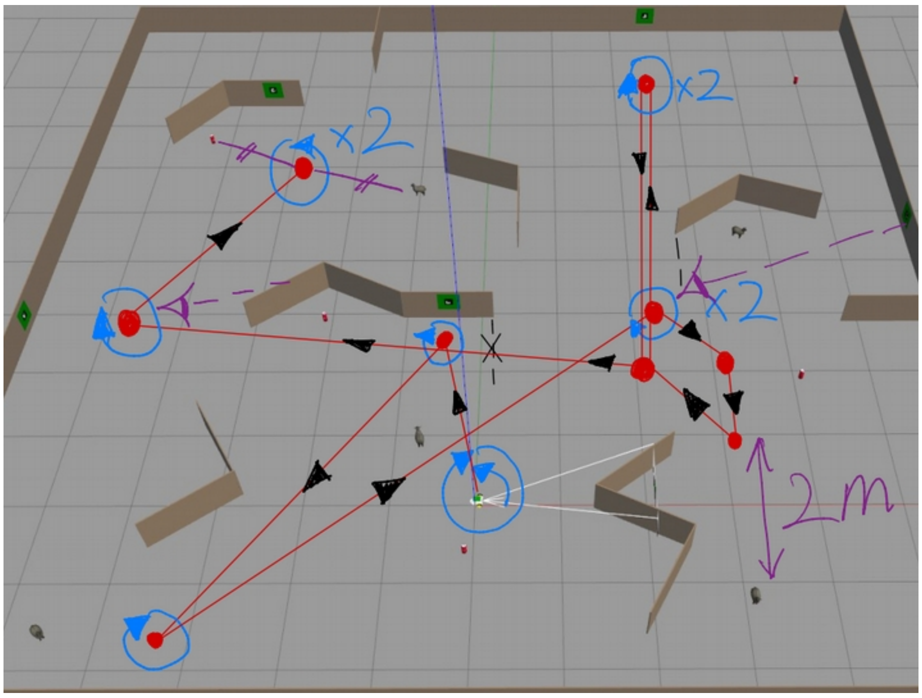


Figure 4: *Path taken during final live demo*

Currently, the robot has been calibrated for a linear velocity of 60ticks/s and angular velocity of 28ticks/s, using friction values of 50 for mu and 200 for mu2 in the SDF file, taking 57.6s for each full 360° turn. These velocities resulted in a completion time of 16m31s for the path taken in **figure 4**. An improvement would be to increase these velocities, while pose estimates are still maintained within the error threshold of 1m for both markers and object estimates, ensuring optimum speed while not sacrificing too much accuracy. The linear compensation could also be scaled in order to counteract drift in Gazebo. Improved speed would be beneficial in increasing score by 3 marks, quicker iteration towards an ideal path, as well as quicker restart in the rare case of the Python requests module or Gazebo failing. Furthermore, a ramp up controller could be added into sending keyboard commands in order to reduce any chance of wheel slip.

Possible Additions:

Addition 1: Semi-auto navigation option

A possible function that could be added to the robot implementation would be a key press that would activate a semi automatic drive. The function will run the robot automatically, by doing a 360 degree scan with YOLO turned off and then drive to the nearest detected aruco marker. This is a common action taken by the robot in any run, thus would be a handy function to have. The robot would start by turning 360 degrees, scanning all the surrounding aruco markers visible from the robot's position and recording the marker with closest distance. The robot will also improve the estimated position of the robot in the world map using SLAM. YOLO is turned off as we would not want to record any object position when the robot is positioned poorly in the world map, because detected objects' positions are calculated with respect to the robot's pose. The robot will then proceed to drive to the nearest aruco marker, unless there are no aruco markers visible. In that case, the robot will remain in position and a text will be printed "Scanning done, no aruco markers visible". If the same key is pressed twice, then the function will be stopped immediately (in case where the nearest aruco marker isn't ideal). This was seen in many other groups implementations during the M5 demonstration. This brings many benefits, with the main one being an extra 5 marks for semi-autonomous driving. In addition, this would ease the pressure on the user to drive in the most optimal direction towards an aruco marker, as the precise path to the next aruco marker would be accurately pre-calculated.

Addition 2: Auto Navigation

The current design of the robot requires manual path planning (i.e. not by implemented algorithm) and multiple test runs to obtain the optimal pathing which would result in accurate map positions (up to 100% accuracy with margin of error less than 1 meter). Implementing auto navigation (automatic path-planning and navigation) would make the robot more robust to different maps. It would be easier to run as there would be no path planning needed, and it would not require any hands on control. In addition, this would result in an extra 10 marks on top of our current implementation. As a safety mechanism, there would be an option to revert to either the semi-auto navigation option described above, or full teleoperation, if for some reason the path planning and autonomous driving failed.