Queensland University of Technology	QUT Systems Engineering UAVPAYG19	Doc No: UAVPAYG19 -TAIP-FD-01 Issue: 1.0 Page: 1 of 45 Date: 28 October 2022

Final Design

Target Acquisition & Image Processing

Prepared by	A.S C.H___________________ Alex Switala & Connor Harvey Target Acquisition & Image Processing Co-Leads	Date	28/10/2022
Checked by	R.B_________________________ Ryan Brooker, Web Interface Design Lead	Date	28/10/2022
Approved by	M.B_________________________ Marissa Bowen, Project Manager & Enclosure Design Lead	Date	28/10/2022
Authorised for use by	______________________________ Dr. Felipe Gonzalez, Project Coordinator	Date	_______________

Queensland University of Technology
Gardens Point Campus
Brisbane, Australia, 4001.

Revision Record

Document Issue/Revision Status	Description of Change	Date	Approved
1.0	Initial Issue	28/10/2022	A.S

Table of Contents

List of Figures

List of Tables

Definitions

Acronym	Definition
TAIP	Target Acquisition and Image Processing
FPS	Frames Per Second
QUT	Queensland University of Technology
UAV	Unmanned Aerial Vehicle
AI	Artificial Intelligence
ArUCO	Augmented Reality University of Cordoba
Wi-Fi	Wireless Fidelity
WVI	Web Visualisation and Interfaces
IP	Image Processing
YOLOv3	You Only Look Once, Version 3

1. Introduction

This final design document provides an overview of the design choices and components of the Target Acquisition & Image Processing (TAIP) subsystem. An outline of the system architecture is provided within this document with a focus on ensuring that the customer needs and system requirements are met. The descriptions for the interfaces between the TAIP subsystem and the other subsystems of the UAVPAYG19 project are provided in this document. This is to be paired with a collection of final design documents of the remaining subsystems to address all the functional requirements and to facilitate a coherent integration between all the subsystems of the UAVPAYG19 project.

1.1. Scope

This subsystem design document contains the following, a description of the Target Acquisition and Image Processing subsystem which includes descriptions of the integration of the TAIP with the rest of the system, diagrams of the TAIP's integration showing details such as the input and outputs of the subsystem, details of the operation, functions and interfaces described within the subsystem architecture as well as designs of the hardware, software, algorithms and all other specifications pertaining to the subsystem. Inclusive, when relevant, within each of these sections are also fully detailed descriptions and justifications backed by evidence on why particular design choices were made.

1.2. Background

The Queensland University of Technology's Airborne System Lab (ASL) has commissioned the group UAVPAYG19 to design and develop a payload capable in detecting specific objects, recording air quality data to be displayed on a web interface and to pierce a ground sample. This payload is to be attached to a S500 UAV which completed an automated flight path. The payload is mounted on the bottom of the UAV using a provided bracket. This payload must contain all components to complete its required tasks. These components are:

- Raspberry Pi 3B+
- Raspberry Pi Camera
- Pimoroni enviro sensor
- DF15RSMG 360 Degree Motor

The project is required to identify three targets, a valve (In open or closed position), a fire extinguisher and an ArUCO marker. The Pimoroni sensor is to be used to record air temperature, pressure humidity, light and potentially hazardous gas level data. This data along with a live feed of the Raspberry Pi Camera is to be visualized on a Web Interface. Lastly a soil sample must be obtained using a sampling mechanism.

2. Reference Documents

2.1. QUT Avionics Documents

	Document Title	Document Description
RD/1	UAV - Customer Needs	Advanced Sensor Payload for UAV Target Detection and Air Quality Monitoring in GPS Denied Environments
RD/2	UAV - System Requirements	UAVPayloadTAQ System Requirements
RD/3	UAVPAYG19-PM-PMP-03	PMP
RD/1	UAVPAYG19-ICD-01	ICD
RD/6	UAVPAYG19-ED-PD-02	ED Preliminary Design
RD/7	UAVPAYG19- AQS-PD-02	AQS Preliminary Design
RD/8	UAVPAYG19-TAIP-PD-02	Target Acquisition and Image Processing Preliminary Design
RD/10	UAVPAYG19-WVI-PD-02	WVI Preliminary Design
RD/23	UAVPAYG-19-TAIP-TR-01	Target Acquisition and Image Processing Test Report
RD/25	UAVPAYG-19-AQS-TAIP-WVI-TR-01	Air Quality Sensor, Target Acquisition and Image Processing, Web Visualization Interface Integration Report
RD/26	UAVPAYG-19-ED-AQS-TAIP-ST-TR-01	Enclosure, Air Quality Sensor, Target Acquisition and Image Processing, Sampling Tube Test Report
RD/27	UAVPAYG-19-ED-AQS-TAIP-WVI-ST-TR-01	Enclosure, Air Quality Sensor, Target Acquisition and Image Processing, Sampling Tube Integration Report
RD/28	UAVPAYG-19-TR-AT-01	Acceptance Test Report
RD/29	UAVPAYG-19-VV-01	Verification and Validation Report
RD/35	UAVPAYG-19-ED-AQS-TAIP-TR-01	Enclosure, Air Quality, Target and Image processing Test Report

2.2. Non-QUT Documents

	Document Title	IEEE Reference
Doc1	OpenCV	"Home - OpenCV", OpenCV, 2022. [Online]. Available: http://opencv.org/. [Accessed: 22- Aug- 2022].
Doc2	YoloV3	J. Redmon, "YOLO: Real-Time Object Detection", Pjreddie.com, 2022. [Online]. Available: https://pjreddie.com/darknet/yolo/. [Accessed: 22- Aug- 2022].
Doc3	YoloV3 Deep Learning	V. Meel, "YOLOv3: Real-Time Object Detection Algorithm (Guide) - viso.ai", viso.ai, 2022. [Online]. Available: https://viso.ai/deep-learning/yolov3-

		overview/#:~:text=YOLOv3%20AI%20models-,What%20is%20YOLOv3%3F,network%20to%20detect%20an%20object. 22/8/22. [Accessed: 22- Aug- 2022].
Doc4	Mini-YoloV3	Q. Mao, H. Sun, Y. Liu and R. Jia, "Mini-YOLOv3: Real-Time Object Detector for Embedded Applications", Ieeexplore.ieee.org, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/8839032. [Accessed: 22- Aug- 2022].
Doc5	What's new in Yolo V3	A. Kathuria, "What's new in YOLO v3?", towardsdatascience, 2022. [Online]. Available: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b. [Accessed: 22- Aug- 2022].
Doc6	Deeplearning	"Deep Learning Darknet 53", Mathworks, 2022. [Online]. Available: https://au.mathworks.com/help/deeplearning/ref/darknet53.html#:~:text=DarkNet%2D53%20is%20a%20convolutional,%2C%20pencil%2C%20and%20many%20animals. [Accessed: 22- Aug- 2022].
Doc7	Yolo V3 overview	V. Meel, "YOLOv3: Real-Time Object Detection Algorithm (Guide) - viso.ai", viso.ai, 2022. [Online]. Available: https://viso.ai/deep-learning/yolov3-overview/. [Accessed: 01- Sep- 2022].
Doc8	Raspberry PI Camera	R. Megapixels, "Raspberry Pi Camera Board v2 - 8 Megapixels", Core-electronics.com.au, 2022. [Online]. Available: https://core-electronics.com.au/raspberry-pi-camera-board-v2-8-megapixels-38552.html. [Accessed: 01- Sep- 2022].
Doc9	LabelImg	J. Nelson, "LabelImg for computer vision annotation", Roboflow Blog, 2022. [Online]. Available: https://blog.roboflow.com/labelimg/#:~:text=What%20is%20LabelImg%3F,your%20next%20object%20detection%20project. [Accessed: 01- Sep- 2022].

3. Subsystem Introduction

The TAIP subsystem is responsible for the image capturing and processing of a live video feed from the camera attached to the payload. This subsystem as per the requirement 'HLO-M-2 – Target Identification' must be able perform three major tasks:

3.1. Identify targets

The three different types of targets that need to be identified are: fire extinguishers, open and closed valves. The approach taken to complete this task is image detection using machine learning, where a dataset of images is collected and processed. An algorithm was then trained on the dataset using a separate machine and run to identify these targets during operation. Unlike other approaches such as image masking and colour detection, this machine learning algorithm took time to train as thousands of images must be identified for the algorithm to be accurate. This approach however, unlike others, will yield better results hence its selection. All image processing was performed on board the system using the Raspberry pi in which the target identification program was run on.

3.2. Detect ArUCO markers and provide position estimates

Due to the nature of the drone and how it has no GPS tracker, our team needed to create our own local coordinate system to 'stand in' for this. This was done by using data gathered from ArUCO markers. Once detected the program was run through a position estimation algorithm that was able to provide the position in (x, y, z) coordinates along with the orientation of the payload. Similarly, to the target identification program this position estimation algorithm was also run on the Raspberry pi.

3.3. Data Transmission to web interface and LCD interface

Once the Target identification video feed and position estimation has been gathered, it needed to be sent to the web interface so that it can be viewed. This was solved using the Wi-Fi capabilities of the on-board Raspberry pi by sending this data to a web server, which displayed this information on the web interface. In addition to this the LCD interface displayed the target identification video feed as part of requirement [REQ-M-12] alongside the IP and temperature.

3.4. Subsystem Requirements

The subsystem must be able to adhere to a set of verifiable requirements as listed below in Table 1. These requirements are provided by the client and are imperative to the development of the subsystem as they must be met for the project to be considered a success.

Table 1: Subsystem Requirements

Requirement	Description	Verification
REQ-M-03	The UAVPayloadTAQ shall communicate with a ground station computer to transmit video, target detection and air quality data.	Demonstration
REQ-M-04	The target identification system shall be capable of alerting the GCS of a target's type.	Demonstration/ simulation
REQ-M-06	The Web Interface is required to display the images of the targets that are taken directly from the UAVPayloadTAQ and updated every time a new picture is taken.	Demonstrated
REQ-M-12	The LCD screen should display live feed of target detection as well as temperature readings from the Pi and the Enviro sensor board.	Demonstration
REQ-M-14	Developed solution shall conform to the systems engineering approach.	Submitted documentation
REQ-M-16	The UAVPayloadTAQ shall process all imagery on-board via the on-board computer	Demonstration
REQ-M-17	The processing must be able to analyse all data acquired from the camera and sensors while the UAV moves at a maximum speed of 2 m/s.	Demonstration
REQ-M-18	The processing must be able to analyse all data acquired from the camera and sensors while the UAV operates at an altitude of between 1 to 3m.	Demonstration
REQ-M-19	Live data from the UAV must be made available through the web server within 10 seconds of capture.	Demonstration

4. Subsystem Architecture

The entirety of the subsystem architecture is displayed in the Figure 1 below with the relevant components of the TAIP subsystem highlighted in a green box. From the Figure 1, power is provided to the raspberry pi camera and the image stream is sent back to raspberry pi for image processing. The image processing handles the image data from the camera and feeds it into the image recognition algorithm to identify the relevant targets. The processed image stream is sent to the web server for the WVI subsystem to handle and display. In addition to the data being sent to the WVI, a live video stream is sent to the LCD screen for users to view.

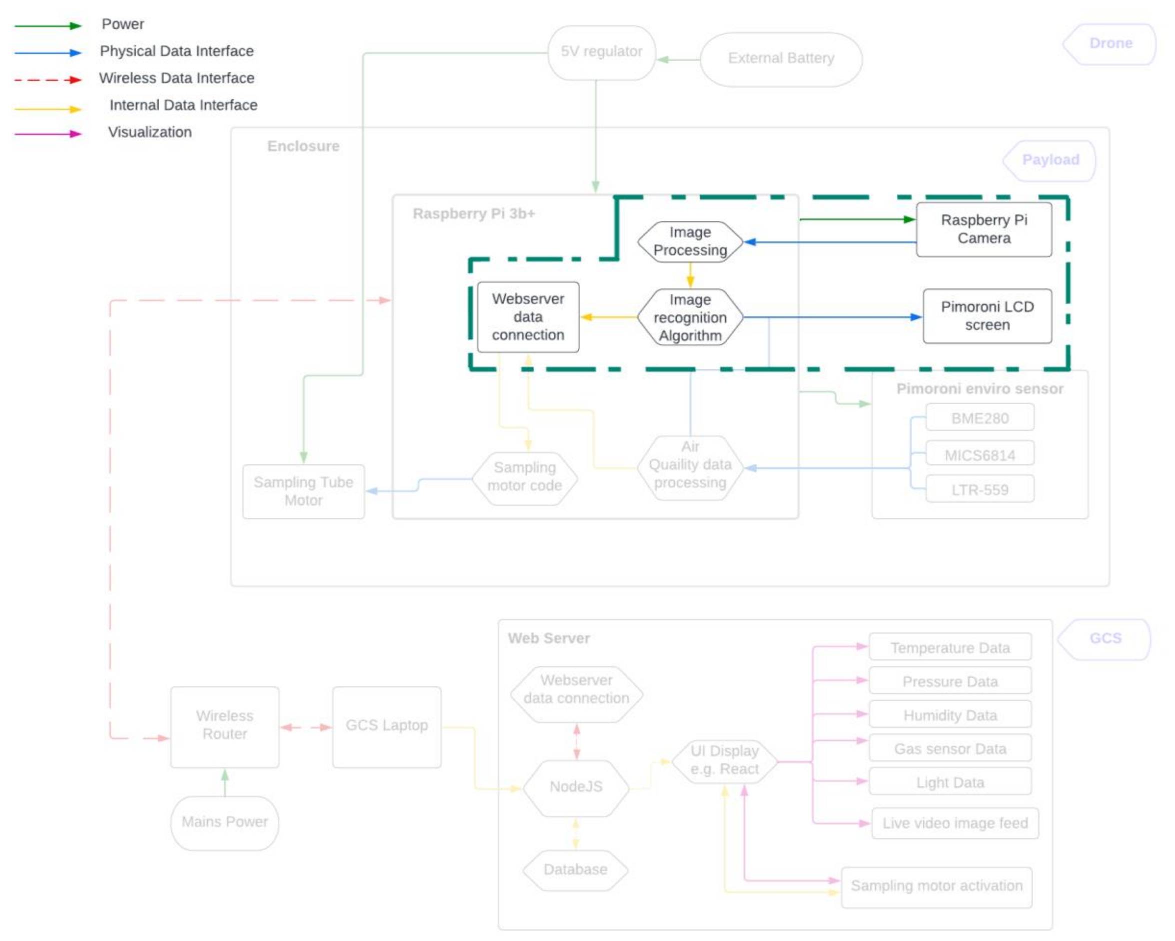


Figure 1: Subsystem Architecture

4.1. Subsystem Interfaces

Figure 2 displays a diagram of this subsystem and its interfaces with the other subsystems.

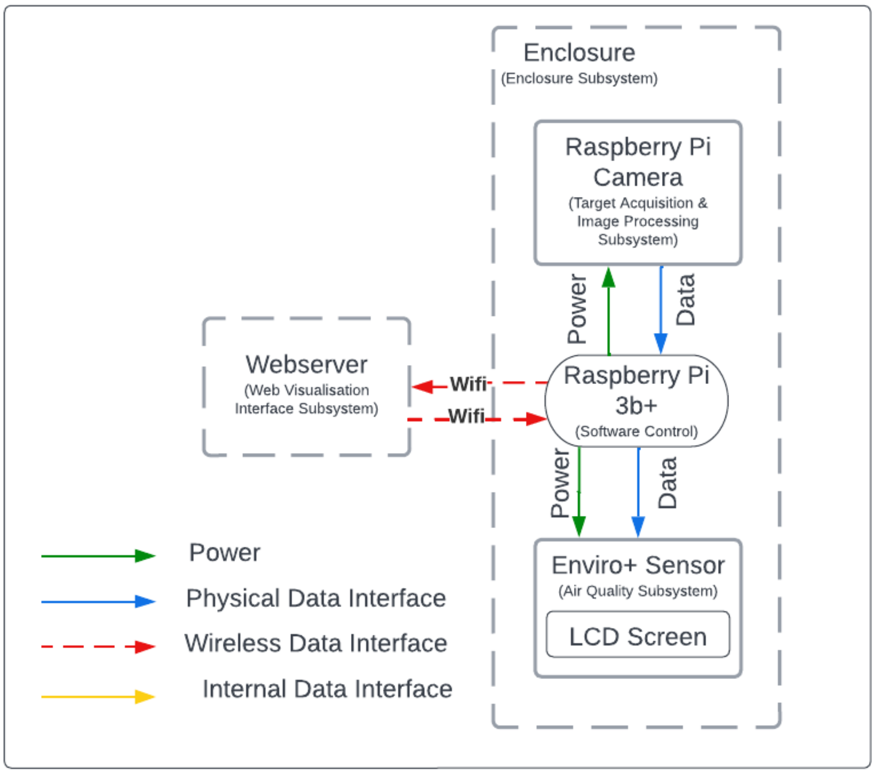


Figure 2: Subsystem Interfaces

Table 2 below provides a detailed description of the interfaces of the TAIP subsystem with the other subsystems and components of the payload.

Table 2: Subsystem Interface Descriptions

Interface	Component/System	Interface Description
I1	Raspberry Pi Camera V2	The raspberry Pi communicates with the Raspberry Pi Camera V2 by powering the camera and receiving a live video feed for the image processing. This will be connected through a ribbon cable from the Raspberry Pi to the Camera. Type: Wired Connection Data Type: Video Stream 480*640p 90FPS
I2	Enclosure Sub-System	The Raspberry Pi camera is connected to the Enclosure subsystem through four mounting screws at the base of the enclosure. There is an additional hole cut for the camera lens to have allow the TAIP subsystem to have vision beneath the payload.
I3	LCD Screen (Air Quality Sub-System)	The TAIP subsystem interfaces with the Air Quality subsystem through the LCD screen located on the Enviro+ sensor. Processed imagery will be passed along to the LCD screen to display a live processed image feed. Type: Wired Connection Data Type: Processed video stream 480* 640 90FPS (Likely Downscaled)

I4	Web Server	The TAIP subsystem interfaces with the web server subsystem through sending a live processed video feed, processed imagery and a JSON data containing drone positioning, ArUco marker IDs and target identification. Type: Wireless Connection Data Type: TCP/IP Packer, Processed video stream and position estimation JSON.

5. Design

For the design of this subsystem, each item was provided to the group thus making preliminary research into other items for the design irrelevant. Table 3 below contains the items provided to the group.

Table 3: System Components

Relevant Item	What it was for
1 Raspberry PI Camera V2	Image capture
1 Raspberry Pi 3B +	On board processing and Wi-Fi connectivity
1 Raspberry Pi 3B + Case	Protecting the internal circuitry
1 Ribbon Cable	Connecting the Camera to the Pi
1 32 GB Micro SD card	Imaging the Pi and Storage
1 Pimoroni Enviro+ Air Quality Sensor	This was inclusive of the LCD Screen
1 SD Card Reader	Imaging the Pi

There are two components from this list that are of importance to this subsystem, the Raspberry Pi Camera and the Raspberry Pi 3B + running on Raspbian. YOLOv3 is used as the main software for the image processing on the Pi and OpenCV is used as an additional tool for adding information to these images. Examples are inclusive of labels and boxes that show where and what the targets are. The final design of the TAIP system was based on these components.

5.1. IP Camera (Raspberry Pi Camera V2)

The image processing section of the subsystem was comprised of two main components, the hardware component being the raspberry pi camera shown in Figure 3 for image capture and the software component for the image processing. Table 4 below contains the hardware specifications for the camera.

Figure 3: Raspberry Pi Camera V2 (Doc8 Raspberry PI Camera)

Table 4: Raspberry Pi Camera V2

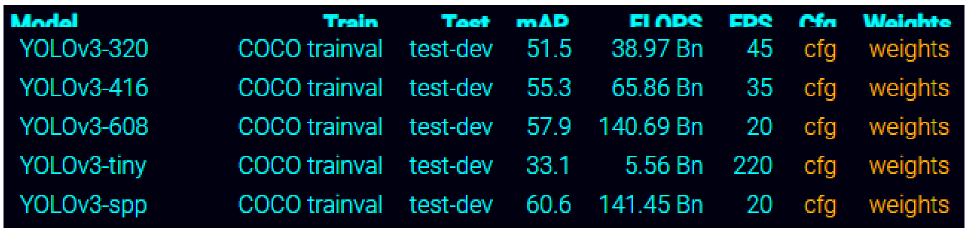
Raspberry Pi Camera V2	Board Powered Image Processing Camera
Image resolution	Photographs 3280x2464, Video 1080p,720p,640*480p
Frame Rate	30 FPS at 1080p, 60FPS at 720p, 90FPS at 640*480
Dimensions	25mm x 23mm x 9mm
Weight	3.4g (Board + Cable)
Minimum Focus Distance	75cm
Sensor	Sony IMX219
Sensor Resolution	3280 x2464 pixels
Sensor image area	3.68 x 2.76 mm (4.6 mm diagonal)
Depth of field	Adjustable with supplied tool
Focal length	3.04 mm
Field of view	Horizontal 62.2 degrees, Vertical 48.8 degrees
Linux Integration	V4L2 driver available
Image Type	Full Colour

5.2. Software

The software architecture of this sub-system was based around using a trained YOLOv3 machine learning model for object detection to identify the targets and OpenCV to process the images by creating boundary boxes and labels. The Machine learning model was trained on a separate machine using data collected from the sessions within the demonstration area. This data contains a collection of photos and videos taken of the targets. The collection was then manually labelled using 'labelImg' for the algorithm to train the weights for classifying the targets. Once trained, OpenCV running in python was used to capture a live camera feed from the Raspberry Pi camera V2 and use the trained weights from the machine learning model to detect and segment the targets on the live image stream. The result of the image processing is a live video feed with clearly identifiable and labelled targets.

5.2.1 YOLOv3:

YOLOv3 (You Only Look Once, Version 3) is a popular real time object detection algorithm that identifies items in videos, live feeds, or images. This machine learning algorithm uses features from deep convolutional networks to achieve its image detection. This algorithm was created by Joseph Redmon and Ali Farhadi with YOLOv3 being the final official version of the YOLO algorithm. This project is to use version of v3 rather than the later "unofficial" versions of YOLO such as YOLOv4 (which was based on a different framework) or YOLOv5 (which was published by the company Ultralytics). The main benefit of this algorithm for this project is the high level of accuracy paired with its ability to detect object in real time which will allow the subsystem to achieve a high rate of performance. For this project there are five different machine learning models of YOLOv3 that had been identified as potential options that can run on the raspberry pi, with YOLOv3-tiny being chosen after running several tests. (Doc3,YOLOV3 Deep Learning)



Model	Train	Test	mAP	FLOPS	FPS	Cfg	Weights
YOLOv3-320	COCO trainval	test-dev	51.5	38.97 Bn	45	cfg	weights
YOLOv3-416	COCO trainval	test-dev	55.3	65.86 Bn	35	cfg	weights
YOLOv3-608	COCO trainval	test-dev	57.9	140.69 Bn	20	cfg	weights
YOLOv3-tiny	COCO trainval	test-dev	33.1	5.56 Bn	220	cfg	weights
YOLOv3-spp	COCO trainval	test-dev	60.6	141.45 Bn	20	cfg	weights

Figure 4: YOLOV3 Versions (Doc2, YOLOV3)

5.2.2 How YOLOV3 works:

The basic premise of YOLO that differentiates it from other object detection software is that it “Only Looks Once” at images. Compared to other machine learning models, YOLO performs the detection procedure as a regression task, increasing the speed of detection and allowing for an assorted sizes of images to be used as an input. In typical models, features learned by convolutional layers are passed onto classifiers to make a detection prediction. However, YOLO instead uses a convolutional layer based on 1x1 convolutions for the detection prediction. The 1x1 convolutions that YOLO uses means that the size of the prediction map is the same size as the feature map before it. YOLO uses Darknet-53 for performing feature extraction; Darknet-53 being a 53 layer deep convolutional neural network; along with this it uses multi scale prediction which helps to improve the accuracy of the target detection.

Convolutional layers within a neural network are layers where filters are applied to the original image and feature maps. The size of the filters is normally specified by a user. In this case it is 5 convolutions alternating from 1x1 kernels to 3x3 kernels finishing with a 1x1 kernel convolution. This also uses three scale predictions found by down-sampling the dimensions of the input image by 32, 16 and 8 respectively, the process is shown in Figure 5 below.

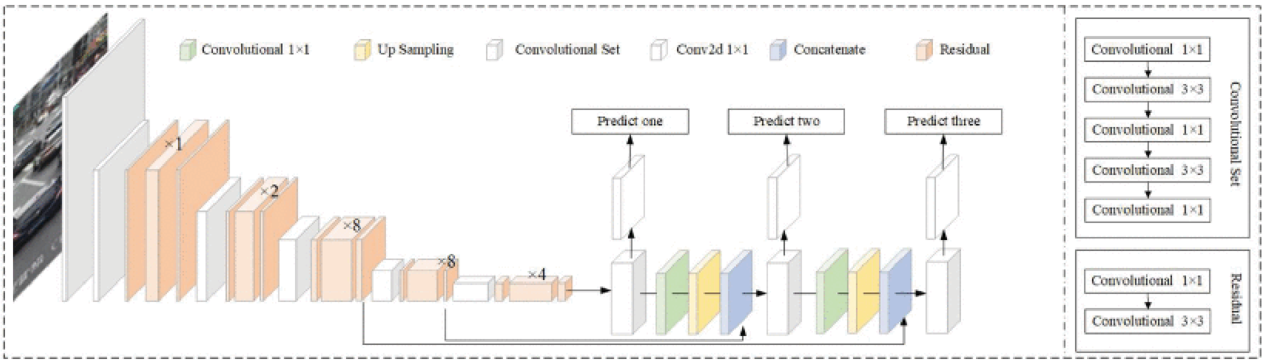


Figure 5:YOLOV3 Convolution Diagram (Doc4, Mini-YoloV3)

The shape of the detection kernel is found using the following equation $n \times n \times (B \times (5 + c))$, where ‘n×n’ is the resolution of the kernel and $(B \times (5 + c))$ is used to calculate the depth of the feature map. ‘B’ being the number of bounding boxes that each cell on the feature map can predict and ‘5+C’ representing the attributes of the boundary boxes. These attributes are the centre coordinates, dimensions, object confidence and finally the number of ‘C’ class confidences. The class confidences are a representation of the probability or ‘confidence scores’ that a detected object belongs to a particular class, i.e. a car, dog, apple etc. (Doc5,Whats new in Yolo V3)

Figure 6 is a visual depiction of a what YOLO would see when looking at the images provided by the data set. Within this Figure, the yellow square is responsible for detecting the fire extinguisher and is added into the prediction feature map.

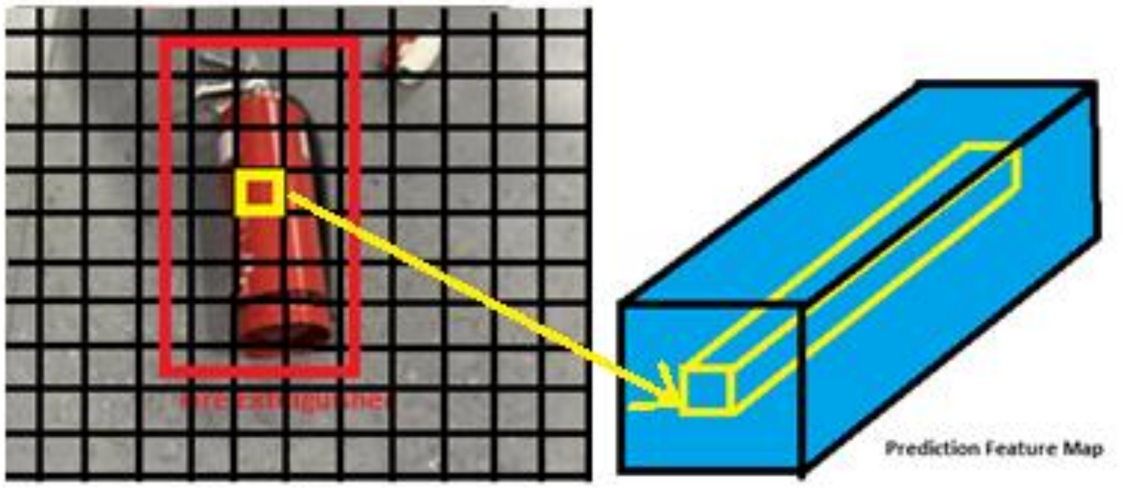


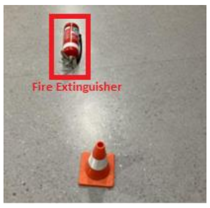
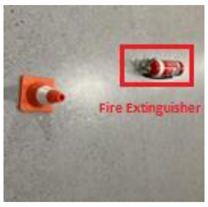
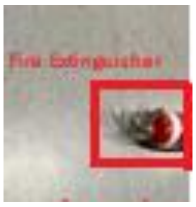
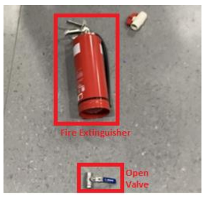
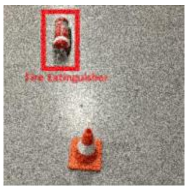
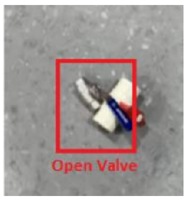
Figure 6: Pixel Bounding Boxes and Prediction Feature Map

As discussed previously, the prediction feature map contains boundary boxes consisting of attributes. These feature maps are what the convolutional layers perform operations on, where object detections are made at different layers to help detect smaller objects with unsampled layers concatenated with previous layers. This allows for the finer details and features of the images that are usually ignored by other machine learning models to be preserved.

Finally, when the model attempts to determine the objects within the image, the classes found using the feature prediction map are used. The class scores are calculated using logistic regression and a threshold value given to determine the labels for an object. The highest scoring classes compared to the threshold value are assigned as labels to the boundary box of an object, with YOLOV3 allowing for multiple labels to be assigned to each object. (Doc7, YoloV3 overview)

5.2.3 Example Dataset with target identification, and image training process

Table 5: Example Targets with idealised object detection

Raw Image:	Un-augmented Labelled Image:	Rotated Labelled Image:	Cropped Labelled Image:	False positive Image:
				
Raw Image:	**Un-augmented Labelled Image:**	**Noised Labelled Image:**	**Occluded Labelled Image:**	**Blurry Image:**
				

Images were augmented with different operations applied to them such as rotation, noise addition and cropping with occluded data along with this there were false positives fed through the dataset as well as unlabelled images which the model would see. This data is then labelled then converted into a readable format for the YOLOV3 algorithm. The data, once in a readable format, is fed through and processed to create a trained model.

5.2.4 LabelImg:

LabelImg is a free, open-source tool for graphically labelling images. It is python based and will be used for labelling the images in training the YOLOV3 model.

This program is a simple to use tool that allows for boxes to be drawn around the images with labels shared across each image within a folder.

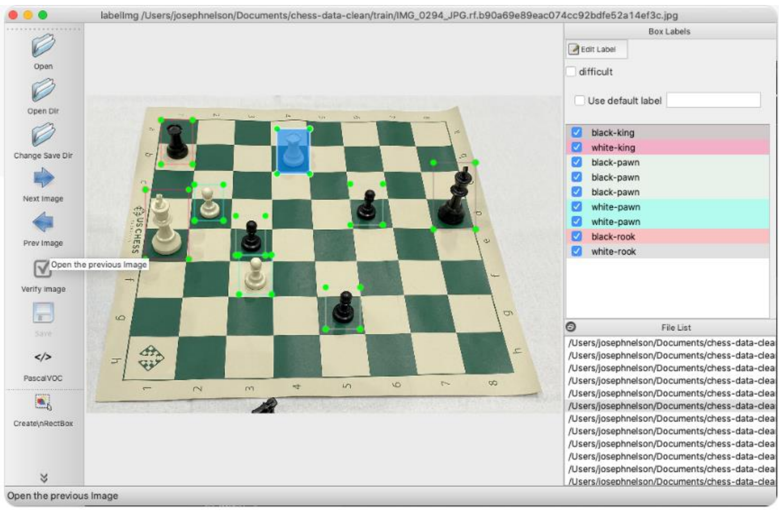


Figure 7: LabelImg interface (Doc9 LabelImg Tutorial)

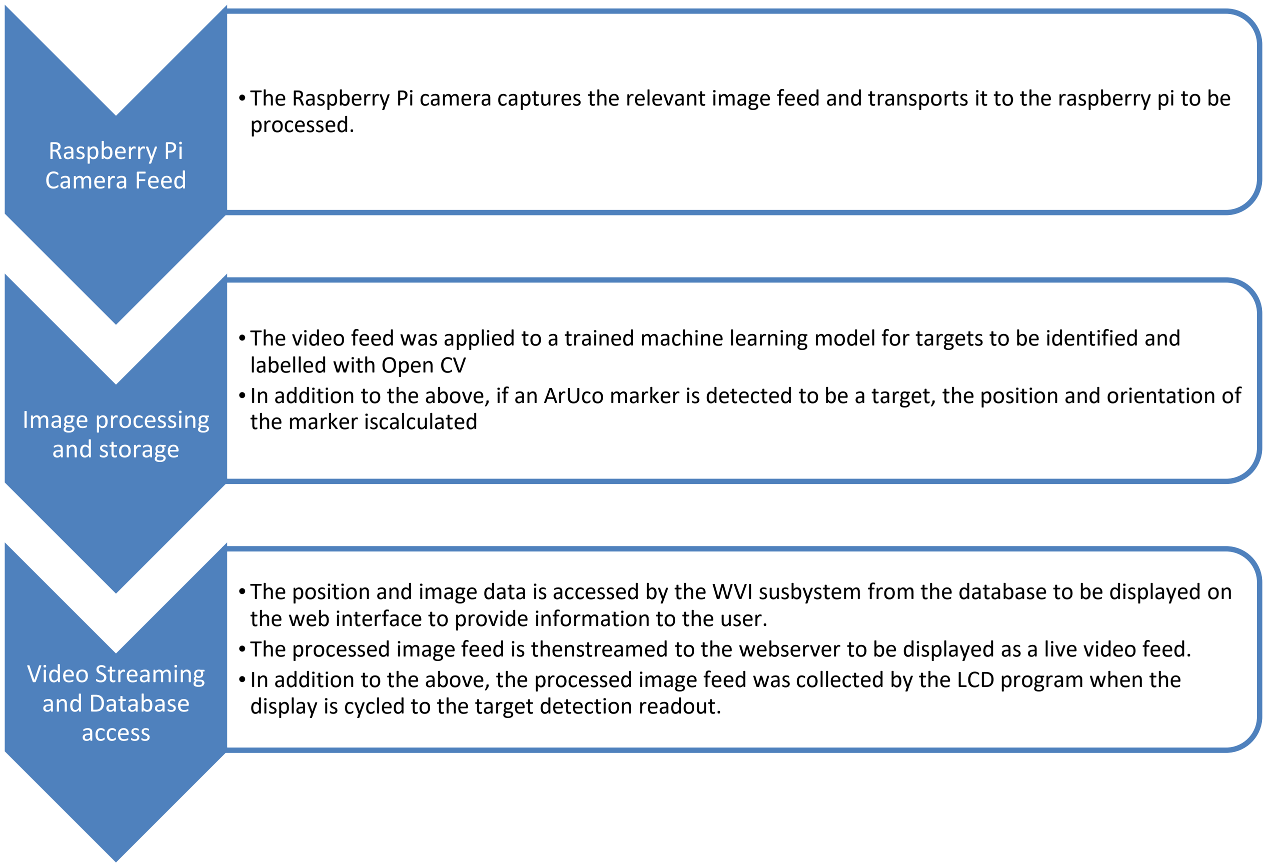
5.2.5 OpenCV:

OpenCV is an open-source computer vision library used for many real time applications due to its optimisation and was built to provide a common infrastructure for computer vision applications. In this project the OpenCV library was used with the python programming language for image processing. More specifically, it was used to draw boxes and labels around the identified images from the live image feed. The library also helped expand the training of the dataset for the machine learned model as it allows for rotating, flipping, warping, stretching, scaling, cropping and other such forms of image manipulation. This allows for the expansion of the dataset by turning one image into many. (Doc1, OpenCV)

An ArUco library module from OpenCV was used to reference a dictionary of ArUco marker types and identify the position of the drone along with the unique identifier of the marker as seen in Appendix item B. Using OpenCV, the position and orientation of the marker is calculated using the known unique identifier values and exact size of the ArUco markers which are 200mm x 200mm in physical size. The resulting calculated position is inserted into a JSON file and sent to the webserver to be displayed.

5.3. Software Flow Diagram

All programs for this subsystem was written in the python programming language. This has been selected due to its simplicity and the wide support available if there are any issues in implementing these sections of code. Multi-threading of programs was managed by the raspberry pi OS and code that is integrated with other subsystems will be discussed and developed with those subsystem leads. Figure 8 describes the basic flow of the software, indicating where the data will be sent once collected, and how it is used.


Raspberry Pi Camera Feed
Image processing and storage
Video Streaming and Database access

- The Raspberry Pi camera captures the relevant image feed and transports it to the raspberry pi to be processed.
- The video feed was applied to a trained machine learning model for targets to be identified and labelled with Open CV
- In addition to the above, if an ArUco marker is detected to be a target, the position and orientation of the marker iscalculated
- The position and image data is accessed by the WVI susbystem from the database to be displayed on the web interface to provide information to the user.
- The processed image feed is thenstreamed to the webserver to be displayed as a live video feed.
- In addition to the above, the processed image feed was collected by the LCD program when the display is cycled to the target detection readout.

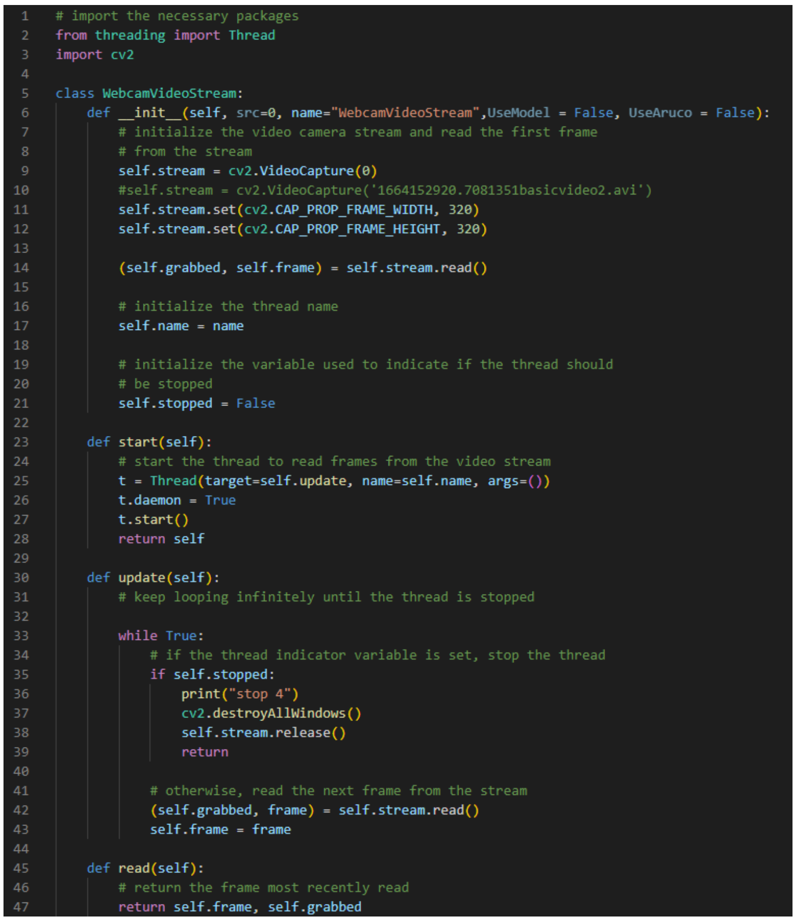
Figure 8: Software Flow

5.4. Implemented Code

This section describes the software that was implemented to identify the three distinct types of targets, as well as process the imagery with the relevant target information. During the development of the code, all function were initially developed as a stand-alone script in order to test their individual functionalities. The code was later refactorized using object orientated programming to operate as a class allowing it to be accessed as an object during the integration stage of the project.

5.4.1 Multi-Threaded video stream

In order to speed up the subsystem, the reading of the camera's video frames was separated from the image processing through the use of multi-threading. A separate class was created containing a function to start the thread which is constantly updating a variable containing latest read frame. By turning this class into an object, the code pertaining to the image processing can easily read the latest frame from memory instead of relying on opening the camera to take an image. This marginally speeds up the frame rate and reliably provides a faster response time to the LCD screen on the payload. The class can seen below in Figure 9.

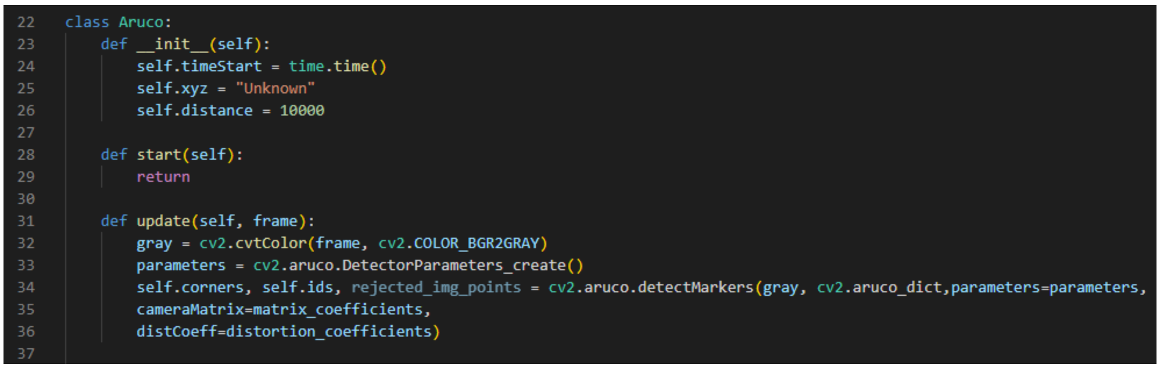


```python
# import the necessary packages
from threading import Thread
import cv2

class WebcamVideoStream:
    def __init__(self, src=0, name="WebcamVideoStream",UseModel = False, UseAruco = False):
        # initialize the video camera stream and read the first frame
        # from the stream
        self.stream = cv2.VideoCapture(0)
        #self.stream = cv2.VideoCapture('1664152920.7081351basicvideo2.avi')
        self.stream.set(cv2.CAP_PROP_FRAME_WIDTH, 320)
        self.stream.set(cv2.CAP_PROP_FRAME_HEIGHT, 320)

        (self.grabbed, self.frame) = self.stream.read()

        # initialize the thread name
        self.name = name

        # initialize the variable used to indicate if the thread should
        # be stopped
        self.stopped = False

    def start(self):
        # start the thread to read frames from the video stream
        t = Thread(target=self.update, name=self.name, args=())
        t.daemon = True
        t.start()
        return self

    def update(self):
        # keep looping infinitely until the thread is stopped

        while True:
            # if the thread indicator variable is set, stop the thread
            if self.stopped:
                print("stop 4")
                cv2.destroyAllWindows()
                self.stream.release()
                return

            # otherwise, read the next frame from the stream
            (self.grabbed, frame) = self.stream.read()
            self.frame = frame

    def read(self):
        # return the frame most recently read
        return self.frame, self.grabbed
```

Figure 9: WebcamVideoStream Class for multithreading

5.4.2 ArUco detection

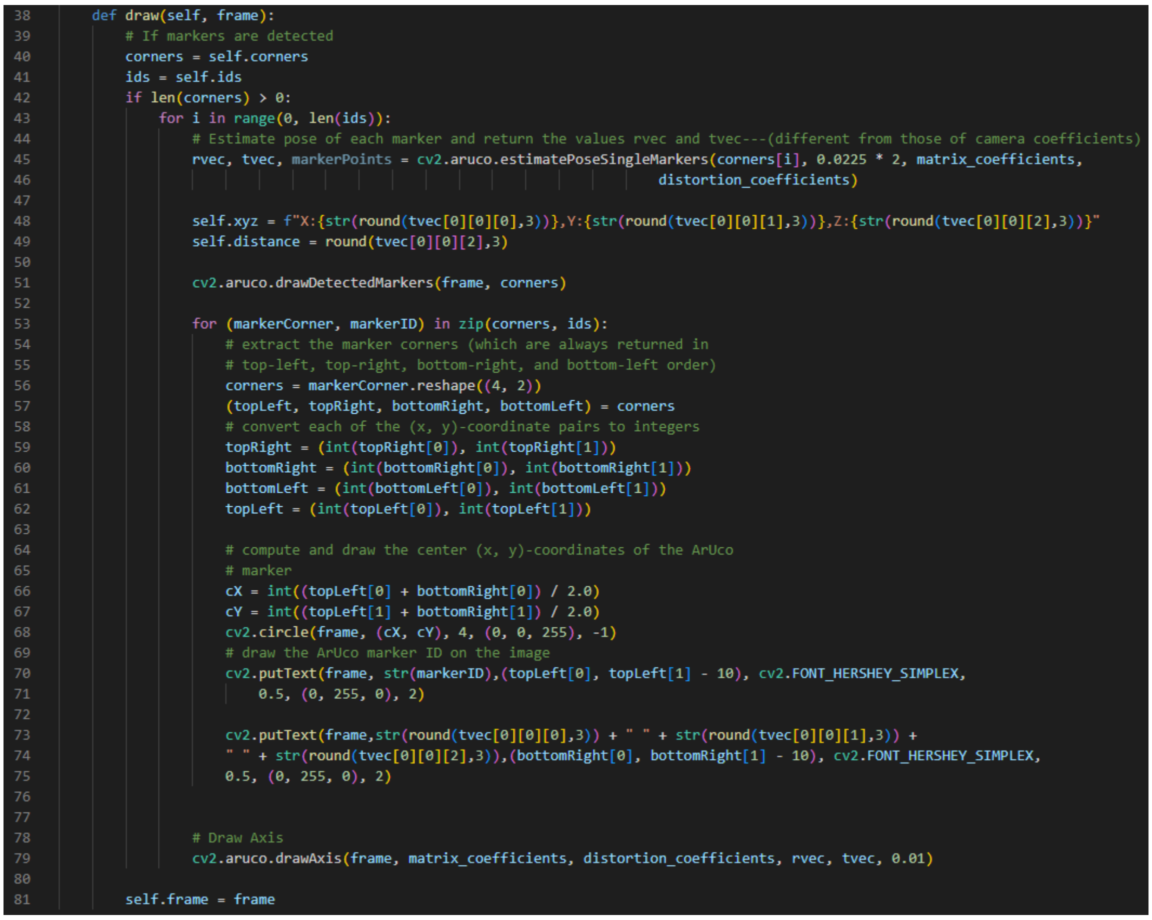
Similar to the previous code, the ArUco identification and coordinate calculation was developed within an individual class. The class itself contains two main functions that was used to process the ArUco marker. The first function is 'Update' and is used to first identify any existing ArUco marker within a frame and internally assign the IDs of each marker to a list and can be seen in the figure 10 below. This function also assigns the XY coordinates of each marker's corners within the image frame into a list. A global camera matrix and distortion coefficient values are inserted into the function to identify the ArUco markers and is further discussed in section 5.4.3.



```python
class Aruco:
    def __init__(self):
        self.timeStart = time.time()
        self.xyz = "Unknown"
        self.distance = 10000

    def start(self):
        return

    def update(self, frame):
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        parameters = cv2.aruco.DetectorParameters_create()
        self.corners, self.ids, rejected_img_points = cv2.aruco.detectMarkers(gray, cv2.aruco_dict,parameters=parameters,
        cameraMatrix=matrix_coefficients,
        distCoeff=distortion_coefficients)

```

Figure 10: Aruco class with Update function

The second function is 'Draw' and is used to calculate the XYZ coordinates of the marker compared to the drone and display the information on the live video feed. This function also draws a boundary box around the marker which perfectly outlines the synthetic square regardless of the orientation, as well as the unique ID of each marker. The XYZ coordinates are saved internally as a variable inside the class object which is accessed and sent to the WVI subsystem during the integration. The 'Draw' function can be seen in Figure 11 below.



```python
    def draw(self, frame):
        # If markers are detected
        corners = self.corners
        ids = self.ids
        if len(corners) > 0:
            for i in range(0, len(ids)):
                # Estimate pose of each marker and return the values rvec and tvec---(different from those of camera coefficients)
                rvec, tvec, markerPoints = cv2.aruco.estimatePoseSingleMarkers(corners[i], 0.0225 * 2, matrix_coefficients,
                                                                           distortion_coefficients)

                self.xyz = f"X:{str(round(tvec[0][0][0],3))},Y:{str(round(tvec[0][0][1],3))},Z:{str(round(tvec[0][0][2],3))}"
                self.distance = round(tvec[0][0][2],3)

                cv2.aruco.drawDetectedMarkers(frame, corners)

                for (markerCorner, markerID) in zip(corners, ids):
                    # extract the marker corners (which are always returned in
                    # top-left, top-right, bottom-right, and bottom-left order)
                    corners = markerCorner.reshape((4, 2))
                    (topLeft, topRight, bottomRight, bottomLeft) = corners
                    # convert each of the (x, y)-coordinate pairs to integers
                    topRight = (int(topRight[0]), int(topRight[1]))
                    bottomRight = (int(bottomRight[0]), int(bottomRight[1]))
                    bottomLeft = (int(bottomLeft[0]), int(bottomLeft[1]))
                    topLeft = (int(topLeft[0]), int(topLeft[1]))

                    # compute and draw the center (x, y)-coordinates of the ArUco
                    # marker
                    cX = int((topLeft[0] + bottomRight[0]) / 2.0)
                    cY = int((topLeft[1] + bottomRight[1]) / 2.0)
                    cv2.circle(frame, (cX, cY), 4, (0, 0, 255), -1)
                    # draw the ArUco marker ID on the image
                    cv2.putText(frame, str(markerID),(topLeft[0], topLeft[1] - 10), cv2.FONT_HERSHEY_SIMPLEX,
                        0.5, (0, 255, 0), 2)

                    cv2.putText(frame,str(round(tvec[0][0][0],3)) + " " + str(round(tvec[0][0][1],3)) +
                    " " + str(round(tvec[0][0][2],3)),(bottomRight[0], bottomRight[1] - 10), cv2.FONT_HERSHEY_SIMPLEX,
                    0.5, (0, 255, 0), 2)


                # Draw Axis
                cv2.aruco.drawAxis(frame, matrix_coefficients, distortion_coefficients, rvec, tvec, 0.01)

        self.frame = frame
```

Figure 11: Aruco Class containing Draw function

5.4.3 Camera Calibration

In order to obtain accurate location data of the payload's coordinates, a calibration matrix and distortion coefficients of the Raspberry Pi camera were required to be calculated. The calibration script was retrieved from a GitHub Repository provided by the teaching team named 'ArUCo-Markers-Pose-Estimation-Generation-Python' created by the user GSNCodes. The calibration.py file requires several pictures of a checkerboard pattern in various positions and orientations to be taken from the Raspberry Pi. The script finds a desired area of squares in each image and calculates the distortion of frame which is used for accurate ArUco marker tracking. An example including the found square grid is included in figure 12 below.

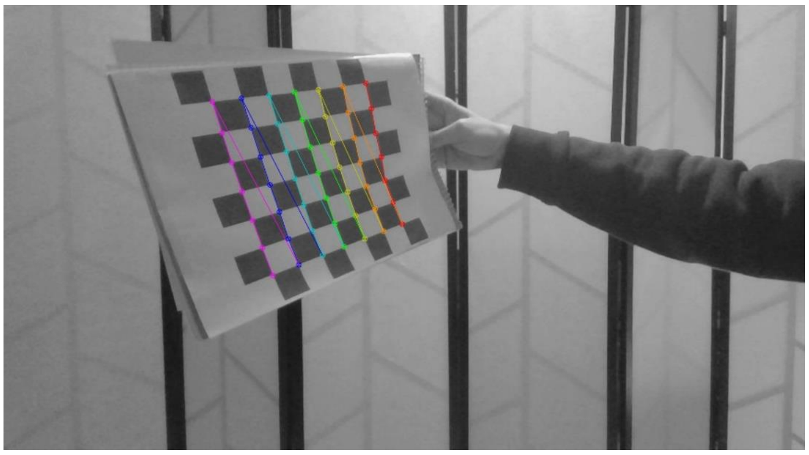
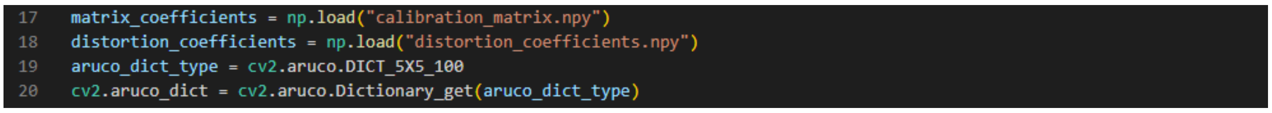


Figure 12: Example camera calibration identifying checkerboard pattern

After the calibration matrix and distortion coefficients are calibrated, they were loaded at the top of the Aruco class within the ArucoDetect.py file, along with the dictionary of the required 5X5_100 markers which were needed to be identified. The included code is seen in figure 13 below.


```
matrix_coefficients = np.load("calibration_matrix.npy")
distortion_coefficients = np.load("distortion_coefficients.npy")
aruco_dict_type = cv2.aruco.DICT_5X5_100
cv2.aruco_dict = cv2.aruco.Dictionary_get(aruco_dict_type)
```
Figure 13: Aruco Class including calibration matrix and distortion coefficients

5.4.4 ArUco Tracking Results

The TAIP was able to successfully track ArUco markers irrespective of position and orientation and calculate the local coordinate frame of the drone. This can be seen in the figure 14 below, where a boundary box is drawn around the ArUco markers including an ID of the marker and the xyz coordinates. Further examples of ArUco identification and processing can be found in the TAIP test document RD/23.

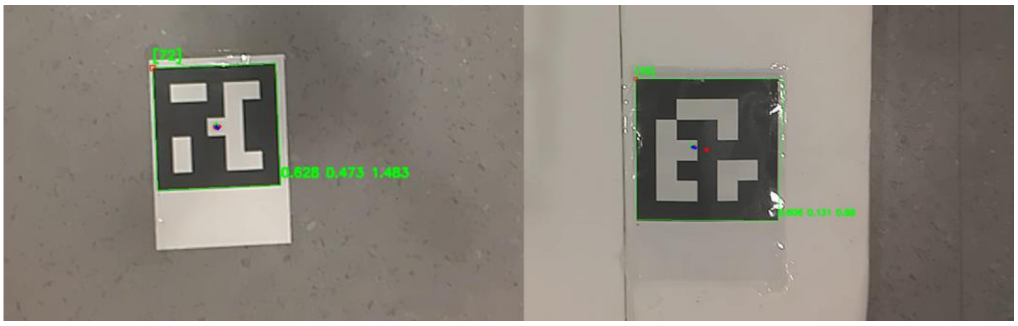
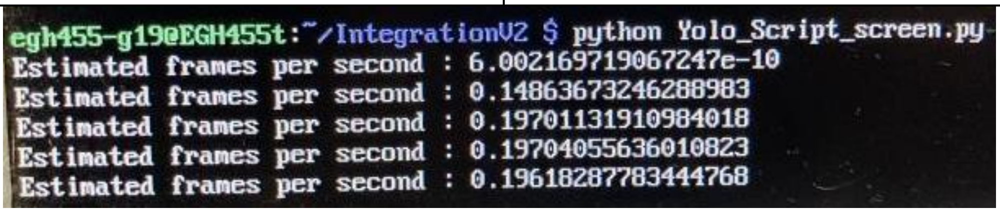
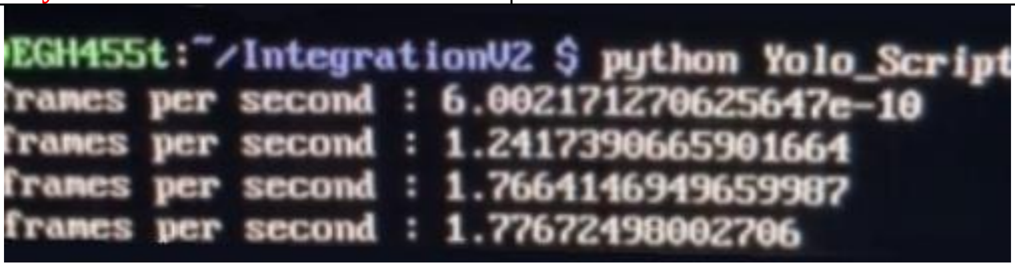
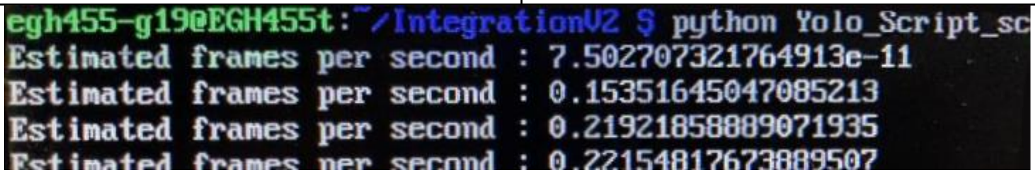


Figure 14: Working ArUco tracking with image processing

5.4.5 YOLOv3 Model

To determine which YoloV3 model to use, tests were performed with three candidate models tested and one chosen. YoloV3 320 was the first candidate as it was the lowest resolution YoloV3 model which could be run on the dataset with a recorded mean Average Precision (mAP) value of 51.5 (Redmon, J. 2012). The YoloV3 tiny Model which is a lightweight version of YoloV3 that uses only 2 convolutional layers one 1x1 and one 3x3 layer. This lowered its precision with its mAP value of 33.1, (Redmon, J. 2012) but allowed it to be less taxing on the systems processing power. Finally, a newer version of the lightweight Yolo model YoloV5 Nano which had a mAP value of 45.7 (Jocher, G. 2020). For a more detail comparison of the models see RD/23 'Test Report Target Acquisition and Image Processing, software test 1: Comparison of different yoloV3 models'. The results of the tests can be seen in Table 6 below:

Table 6: Framerates of Viable Models

Model	**Frames per second**
YoloV3 320	**0.197**
	
YoloV3 Tiny	**1.77**
	
YoloV5 Nano	**0.22**
	

```
egh455-g19@EGH455t:~/IntegrationV2 $ python Yolo_Script_screen.py
Estimated frames per second : 6.002169719067247e-10
Estimated frames per second : 0.14863673246288983
Estimated frames per second : 0.19701131910984018
Estimated frames per second : 0.19704055636010823
Estimated frames per second : 0.19618287783444768
```

```
EGH455t:~/IntegrationV2 $ python Yolo_Script
rames per second : 6.002171270625647e-10
rames per second : 1.2417390665901664
rames per second : 1.766414694965998
frames per second : 1.77672498002706
```

```
egh455-g19@EGH455t:~/IntegrationV2 $ python Yolo_Script_sc
Estimated frames per second : 7.502707321764913e-11
Estimated frames per second : 0.15351645047085213
Estimated frames per second : 0.21921858889071935
Estimated frames per second : 0.22154817673889507
```

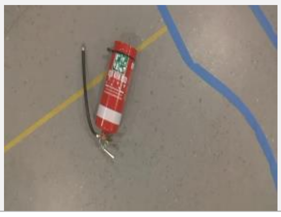
Yolo TinyV3 was the model selected to be used as the performance measured in the framerate of the model was the limiting factor of the system and not the accuracy of the model.

Once the model was determined it was trained on darknet, a machine learning program for training Image Processing models. See below in Figure 15 for the final average loss values as the model was trained over a 3-hour period.

5.4.6 Dataset Training

There were three different data collection sessions over the duration of the project, with an initial count of 2000 images collected within the first session. The first available session allowed for an in-person image capturing of each target. This was before the demonstration field was finalised with no confirmation of the shape of the valves, obstacles, and ground condition. The second session was images taken from a drone flight test which the payload was mounted to record a video. Images was sliced from the video using OpenCV with each unique frame being used as an images for training. The final session consisted of using 2 supplied videos which consisted of the drone flying within a room where the ground was covered in yellow and blue lines. Obstacles were placed to intentionally trigger false positives and consisted of a high vis orange bag and a red valve. Example images from each collection session can be seen in Table 7 below.

Table 7: Key features of collected Image Dataset

Item	Example images from first collection session	Example images from second collection session	Example images from final collection session
Open Valve			
Closed Valve			
Fire Extinguisher			
Example False positive			No False positive examples were gathered during this session.
Ground			

After collecting the images, they were then processed using the OpenCV library and its suite of tools with the following types of image augmentation being applied to each (for images see Table 5). This ballooned the number of total images to be around 5437 images which was then divided via a 70%:30% split into training and testing data.

Once the model was determined, it was then trained using the training data on darknet, a machine learning program for training Image Processing models. Figure 15 below shows the final average loss values; which are values that show how accurately the model is being fitted to the dataset. The value is indicative of its error as it trains; with the model trained over a 3-hour period.

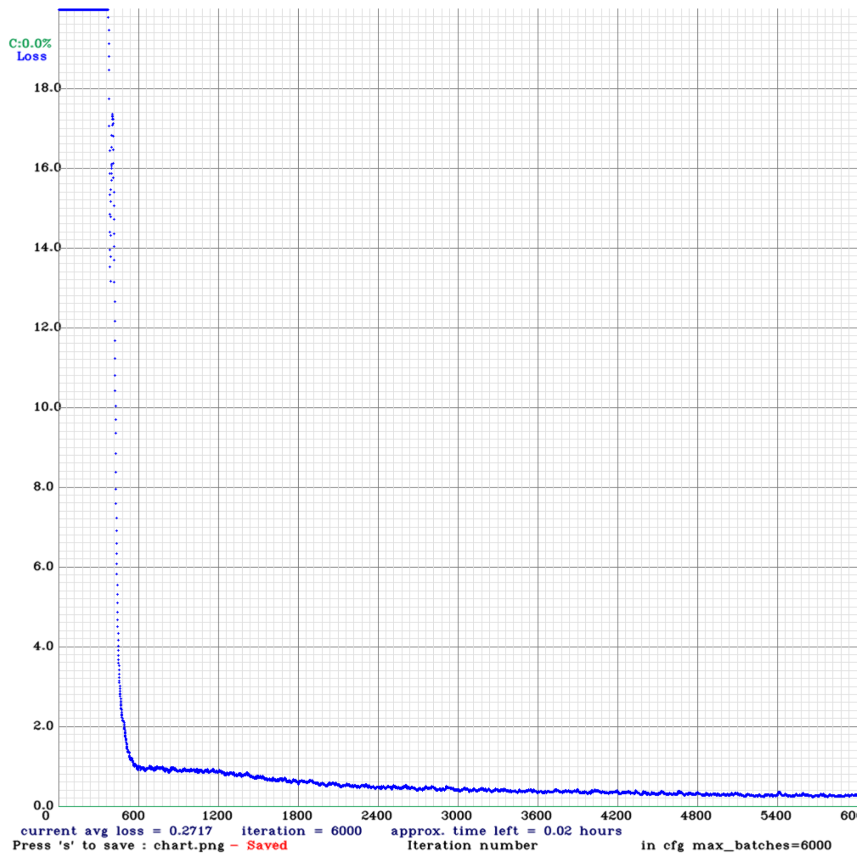


Figure 15 Average Loss Graph Generated during training

The .cfg file for the yolo model, yolov3-tiny.cfg was downloaded with the installation of darknet from (Alexy (2020)) with the following values within being changed to suit the dataset. Within the yolov3-tiny.cfg file under [net] and # Testing batch and subdivisions was commented out which set the flags that the model wasn't testing, and then below that under #Training batch and subdivisions were uncommented with subdivisions being set to be equal to 16. This is the number of times that the image batch is divided into smaller parts, with the memory that is necessary to run the application increasing each time. A maximum value of 64 could theoretically be used, however, the GPU a RTX 2070 SUPER and the 8GB Video Ram that was the training computer was only able to handle 16 subdivisions. Although, this only impacted the speed at which the model would train. The steps were changed to be 4800 and 5400 rather than 400000 and 450000, which is the learning rate that was recommended to be used for a model. That is only needed to identify 3 classes with higher rates being for more classes. The classes variable under [yolo] was changed to be 3 as there were only 3 classes to be identified, along with this, the filters variable under each [yolo] section of the .cfg file was changed according to the following formula: $filters = (classes + 5) * 3$ in the [convolutional] before each [yolo] layer. Thus as there were only 3 classes, the filters before each yolo layer was changed to be $filters = 24$.

The training resulted in a weights file called yolov3-custom-tiny_last.weights which held the model. This in conjunction with the yolov3-custom-tiny.cfg file could be used to run the model, which was done so using OpenCV and darknet itself during testing.

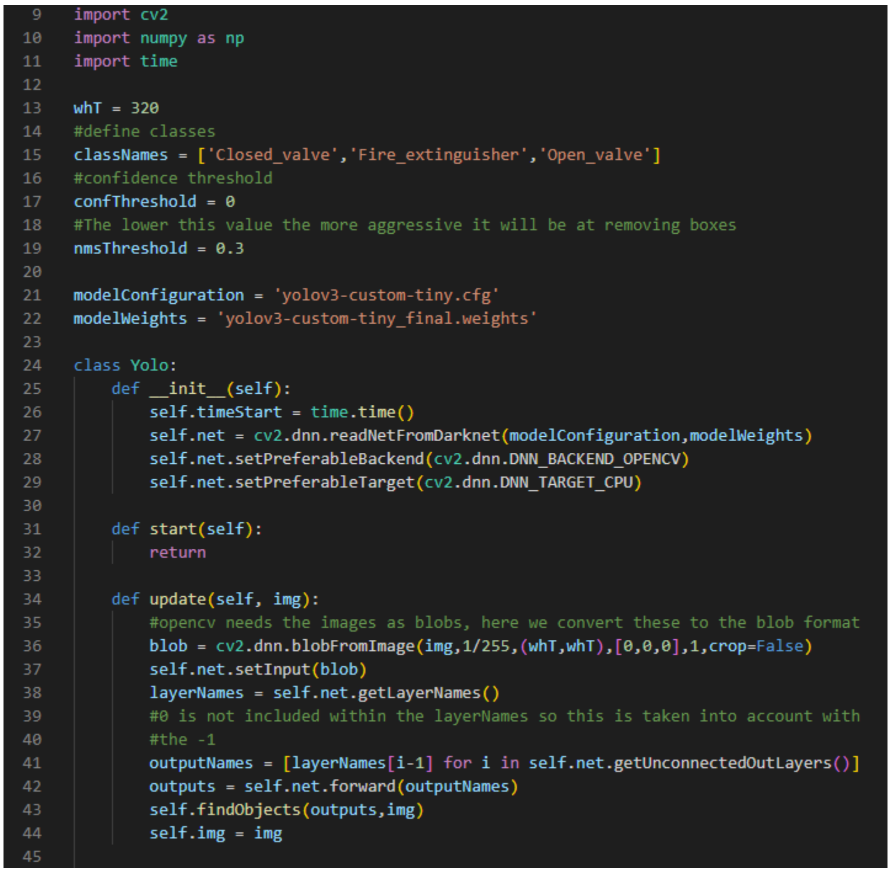
The model was then tested on the testing data, which consisted of images and videos. Below being an example of a video where the final model was tested, to confirm its viability. For more information on the testing see RD/23 Target Acquisition and Image Processing Test Report; section 4.1.3 Software Test 3: Accuracy of yoloV3 model in realistic conditions.

Testing Image processing Link: Result of Test flight V3 1.8m

Result of Test flight V3 2 5m

5.4.7 YoloModel Class

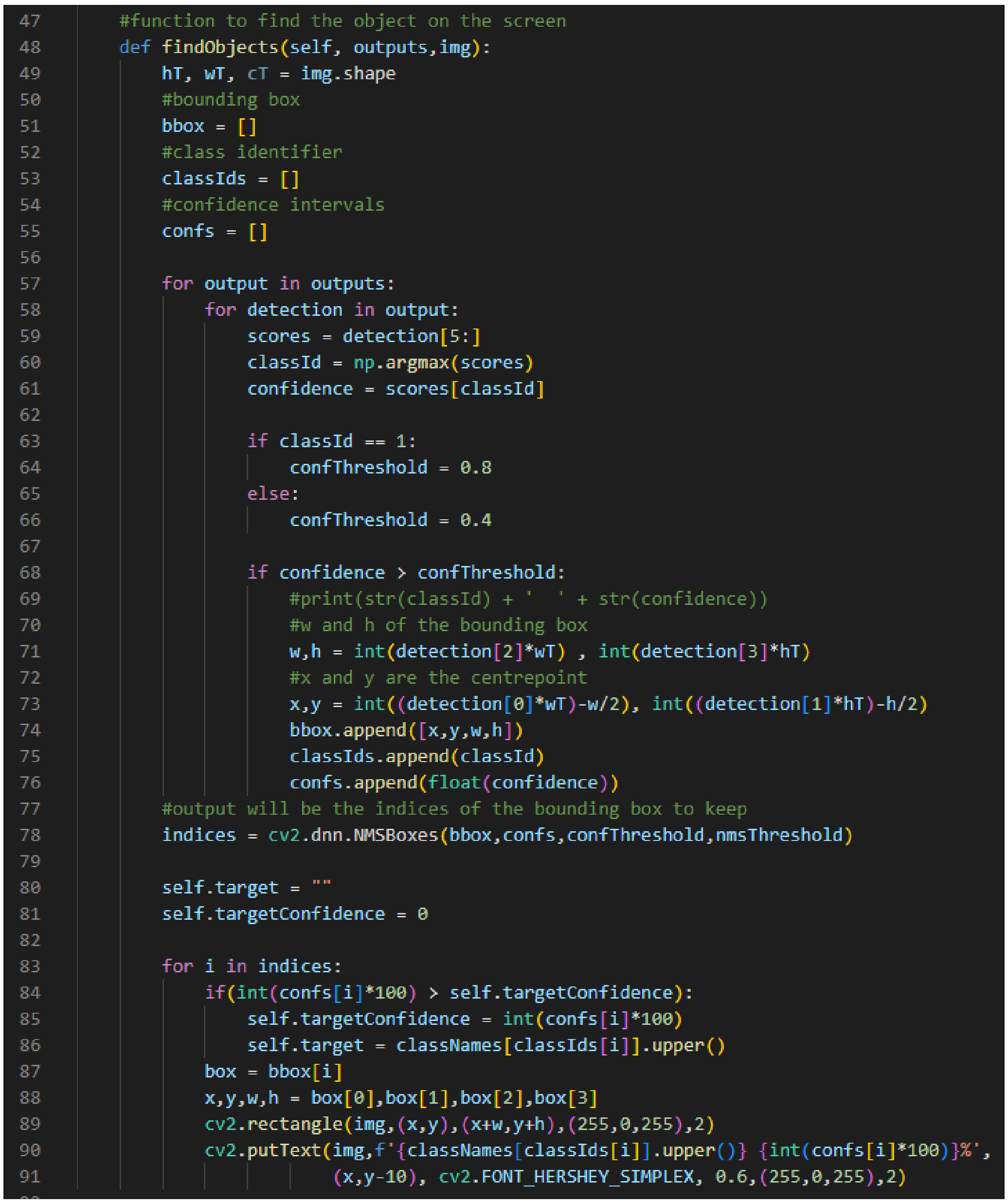
Similar object orientated coding practices were using when implementing the code for the Yolov3 model. A Yolo class was created which used the weights gained from the training the model in order to find the trained targets within an image frame. The class contains an initialisation function that runs when the class object is created in the integration python file which loads the configuration, weights and desired processing framework. After initialization, an image frame is continuously sent to the update function that converts the image into a blob format and runs a separate find object function. The following class and update function can be seen in figure 16 below.



```python
import cv2
import numpy as np
import time

whT = 320
#define classes
classNames = ['Closed_valve','Fire_extinguisher','Open_valve']
#confidence threshold
confThreshold = 0
#The lower this value the more aggressive it will be at removing boxes
nmsThreshold = 0.3

modelConfiguration = 'yolov3-custom-tiny.cfg'
modelWeights = 'yolov3-custom-tiny_final.weights'

class Yolo:
    def __init__(self):
        self.timeStart = time.time()
        self.net = cv2.dnn.readNetFromDarknet(modelConfiguration,modelWeights)
        self.net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
        self.net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)

    def start(self):
        return

    def update(self, img):
        #opencv needs the images as blobs, here we convert these to the blob format
        blob = cv2.dnn.blobFromImage(img,1/255,(whT,whT),[0,0,0],1,crop=False)
        self.net.setInput(blob)
        layerNames = self.net.getLayerNames()
        #0 is not included within the layerNames so this is taken into account with
        #the -1
        outputNames = [layerNames[i-1] for i in self.net.getUnconnectedOutLayers()]
        outputs = self.net.forward(outputNames)
        self.findObjects(outputs,img)
        self.img = img
```

Figure 16: Yolo class containing Initialization and update functions

The find objects function identifies all the target objects within the image and assigns a confidence value based on how accurately the yolo model believes it to be the object. The objects are then compared against a confidence threshold to reduce the number of false positives within the image. Each object has an index corresponding to the class name which is both assigned to the object and drawn onto the image if the confidence of the object is higher than the threshold. The code for the object detection is located in figure 17 below.

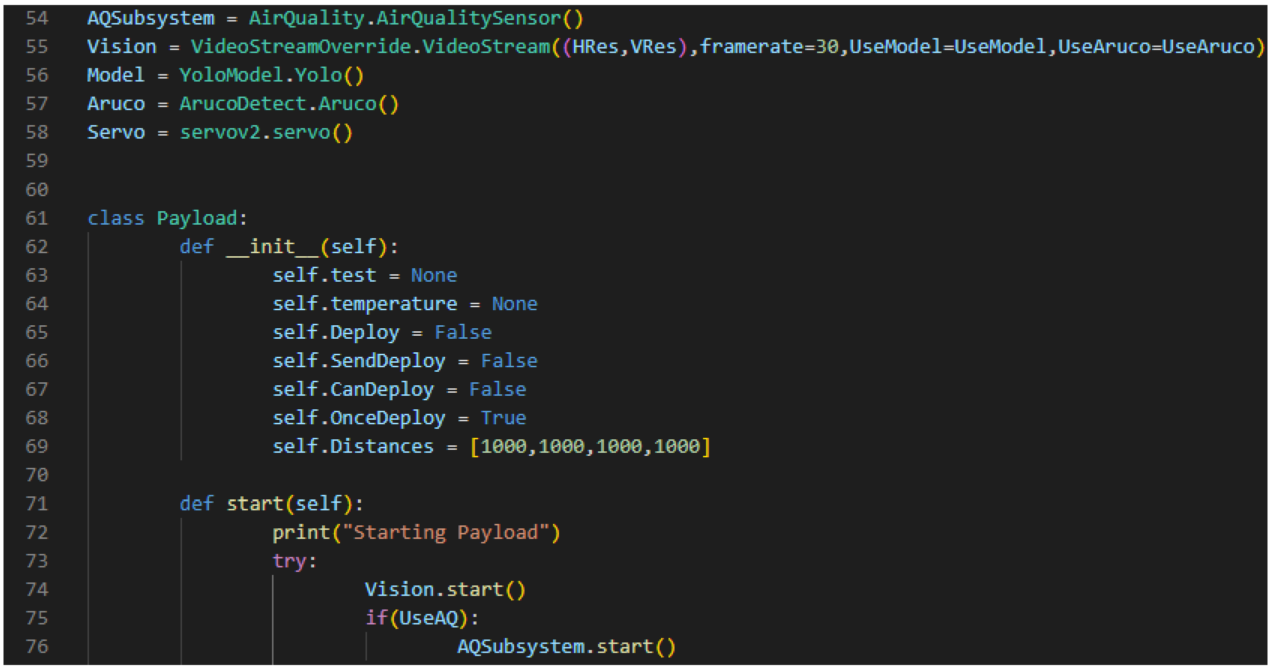


```python
    #function to find the object on the screen
    def findObjects(self, outputs,img):
        hT, wT, cT = img.shape
        #bounding box
        bbox = []
        #class identifier
        classIds = []
        #confidence intervals
        confs = []

        for output in outputs:
            for detection in output:
                scores = detection[5:]
                classId = np.argmax(scores)
                confidence = scores[classId]

                if classId == 1:
                    confThreshold = 0.8
                else:
                    confThreshold = 0.4

                if confidence > confThreshold:
                    #print(str(classId) + '  ' + str(confidence))
                    #w and h of the bounding box
                    w,h = int(detection[2]*wT) , int(detection[3]*hT)
                    #x and y are the centrepoint
                    x,y = int((detection[0]*wT)-w/2), int((detection[1]*hT)-h/2)
                    bbox.append([x,y,w,h])
                    classIds.append(classId)
                    confs.append(float(confidence))
        #output will be the indices of the bounding box to keep
        indices = cv2.dnn.NMSBoxes(bbox,confs,confThreshold,nmsThreshold)

        self.target = ""
        self.targetConfidence = 0

        for i in indices:
            if(int(confs[i]*100) > self.targetConfidence):
                self.targetConfidence = int(confs[i]*100)
                self.target = classNames[classIds[i]].upper()
            box = bbox[i]
            x,y,w,h = box[0],box[1],box[2],box[3]
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,255),2)
            cv2.putText(img,f'{classNames[classIds[i]].upper()} {int(confs[i]*100)}%',
                        (x,y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.6,(255,0,255),2)
```

Figure 17: Yolo Class' findObjects function

5.4.8 TAIP Integration code

The TAIP subsystem played a major role in the integration of the software between the subsystems of the payload. A Payload class was created within the G19-Integration.py file which included the objects for the other python classes that were initialised when the Payload object was created. Figure 18 below shows the Payload class including the initialisation of the other subsystems.

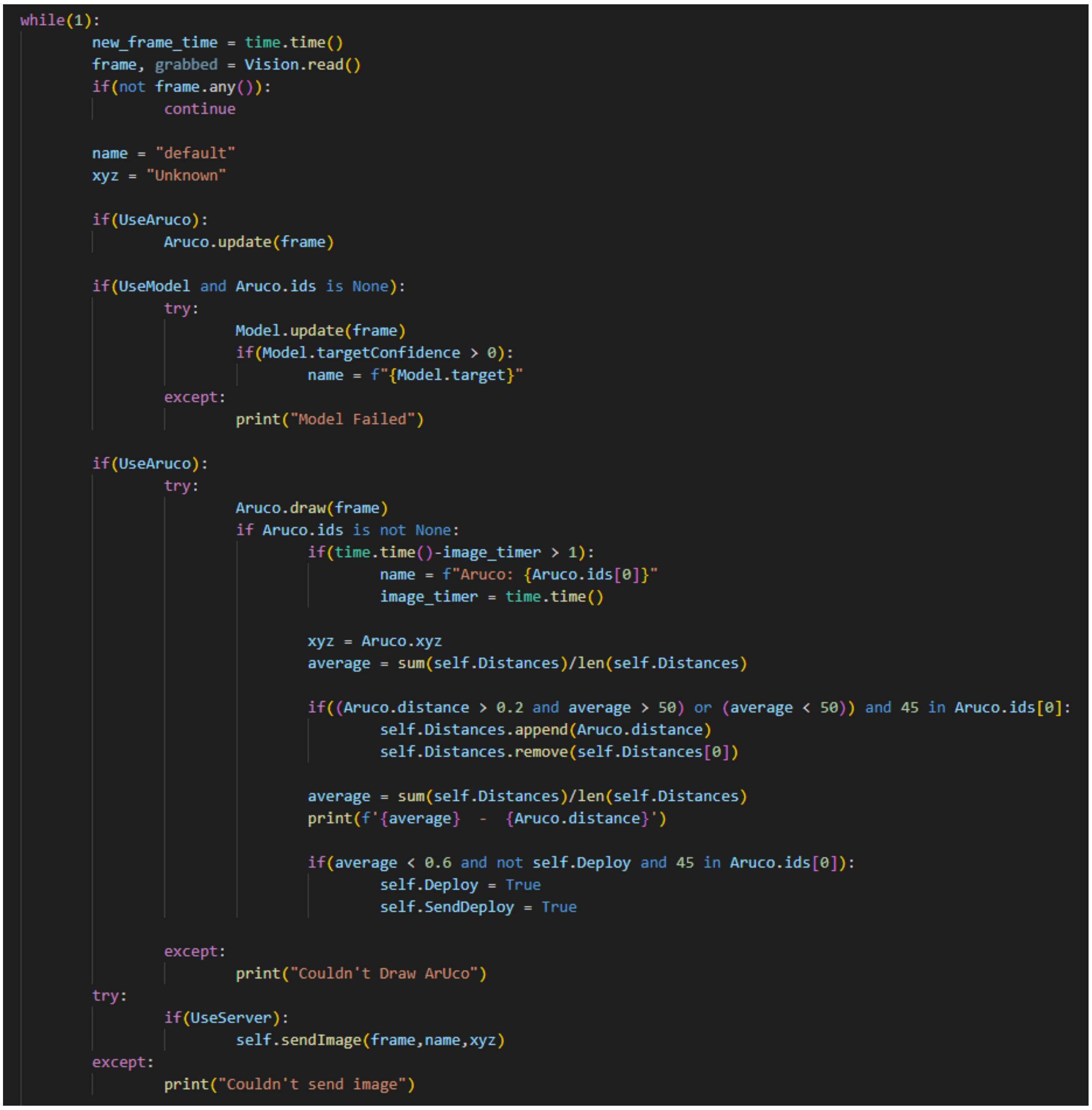


```python
AQSubsystem = AirQuality.AirQualitySensor()
Vision = VideoStreamOverride.VideoStream((HRes,VRes),framerate=30,UseModel=UseModel,UseAruco=UseAruco)
Model = YoloModel.Yolo()
Aruco = ArucoDetect.Aruco()
Servo = servov2.servo()


class Payload:
        def __init__(self):
                self.test = None
                self.temperature = None
                self.Deploy = False
                self.SendDeploy = False
                self.CanDeploy = False
                self.OnceDeploy = True
                self.Distances = [1000,1000,1000,1000]

        def start(self):
                print("Starting Payload")
                try:
                        Vision.start()
                        if(UseAQ):
                                AQSubsystem.start()
```

Figure 18: G19-Intergration.py Payload Class

After the Payload start function is run, the vision system is started which opens the thread and saves each concurrent frame to an internal Vision object. A loop is entered that continuously reads the Frame variable of the Vision object which provides the frame to the ArUco class to identify any ArUco markers, followed by the Yolo Model class to identify any remaining targets. If an ArUco marker is identified, the function skips the Yolo model which significantly increases the framerate of the subsystem.

A running average of the distance to the ArUco marker containing the ID of 45 is taken throughout the loop. This is to provide an accurate distance measurement that is used to enable the sampling tube payload if the marker is within 700mm. Finally, the image is sent to the server containing MetaData about the most recent captured frame. The process can be seen in figure 19 below.



```python
while(1):
        new_frame_time = time.time()
        frame, grabbed = Vision.read()
        if(not frame.any()):
                continue

        name = "default"
        xyz = "Unknown"

        if(UseAruco):
                Aruco.update(frame)

        if(UseModel and Aruco.ids is None):
                try:
                        Model.update(frame)
                        if(Model.targetConfidence > 0):
                                name = f"{Model.target}"
                except:
                        print("Model Failed")

        if(UseAruco):
                try:
                        Aruco.draw(frame)
                        if Aruco.ids is not None:
                                if(time.time()-image_timer > 1):
                                        name = f"Aruco: {Aruco.ids[0]}"
                                        image_timer = time.time()

                                xyz = Aruco.xyz
                                average = sum(self.Distances)/len(self.Distances)

                                if((Aruco.distance > 0.2 and average > 50) or (average < 50)) and 45 in Aruco.ids[0]:
                                        self.Distances.append(Aruco.distance)
                                        self.Distances.remove(self.Distances[0])

                                average = sum(self.Distances)/len(self.Distances)
                                print(f'{average}  -  {Aruco.distance}')

                                if(average < 0.6 and not self.Deploy and 45 in Aruco.ids[0]):
                                        self.Deploy = True
                                        self.SendDeploy = True

                except:
                        print("Couldn't Draw ArUco")
        try:
                if(UseServer):
                        self.sendImage(frame,name,xyz)
        except:
                print("Couldn't send image")
```

Figure 19: TAIP System logic in G19-Integration.py file

6. Conclusion

The Target Acquisition and Image Processing subsystem that is used in the final design has been designed to meet the requirements described in the system requirements table. Using both hardware and software connections, the Raspberry Pi Camera V2 was be used to capture a stream of images and processed on board the raspberry pi. The real time object detection algorithm of YOLOv3 was used to detect the required targets with OpenCV used for image labelling. The images were both displayed onto the enviro+ LCD screen and sent to the web server for the WVI subsystem to handle and display. There were no issues with the integration of this subsystem with the rest of the project and the subsystem was verified by the individual and integration tests performed. However, when demonstrating the payload, the image processing requirement of the subsystem was unable to be validated as working due to a false positive and missing detection of a closed valve. For more detailed information see RD/29 Verification and Validation Report.

The Table 8 below shows how the final design meets the system requirements described by the TAIP subsystem requirements and the HLO2 requirements.

Table 8: Requirements met by final design

Requirement	Description	Requirement Met
REQ-M-03	The UAVPayloadTAQ shall communicate with a ground station computer to transmit video, target detection and air quality data.	**Met: -** The TAIP subsystem was designed to send processed image frames as a stream to the ground station computer hosting the web interface.
REQ-M-04	The target identification system shall be capable of alerting the GCS of a target's type.	**Met: -** The TAIP subsystem packaged the identified target type into a JSON file.
REQ-M-06	The Web Interface is required to display the images of the targets that are taken directly from the UAVPayloadTAQ and updated every time a new picture is taken.	**Met: -** The TAIP subsystem packaged a BASE64 encoded jpg image into a JSON file.
REQ-M-12	The LCD screen should display live feed of target detection as well as temperature readings from the Pi and the Enviro sensor board.	**Met: -** The subsystem streamed the processed video feed from the YoloV3 model onto the LCD screen.
REQ-M-14	Developed solution shall conform to the systems engineering approach.	**Met: -** All the design decisions of the TAIP subsystem are a direct result from the system requirements and HLO requirements.
REQ-M-16	The UAVPayloadTAQ shall process all imagery on-board via the on-board computer	**Not met: -** The TAIP subsystem used the YoloV3 machine learning model to detect targets and OpenCV to open the video feed and draw boundary boxes and labels onto the video feed. However, it was not successful in identifying all the targets correctly

		with an erroneous detection of the fire extinguisher.
REQ-M-17	The processing must be able to analyse all data acquired from the camera and sensors while the UAV moves at a maximum speed of 2 m/s.	**Met: -** The TAIP subsystem is designed to run multiple threads along with a lightweight machine learning model to have enough resources to process images at a fast rate of movement.
REQ-M-18	The processing must be able to analyse all data acquired from the camera and sensors while the UAV operates at an altitude of between 1 to 3m.	**Met: -** The TAIP subsystem is designed to detect targets at a range between 1 to 3m by training the machine learning model of images taken within that distance.
REQ-M-19	Live data from the UAV must be made available through the web server within 10 seconds of capture.	**Met: -** The subsystem will ensure that the data sent to the web server will be handled by a separate thread to ensure data will be sent as fast as possible.

7. References

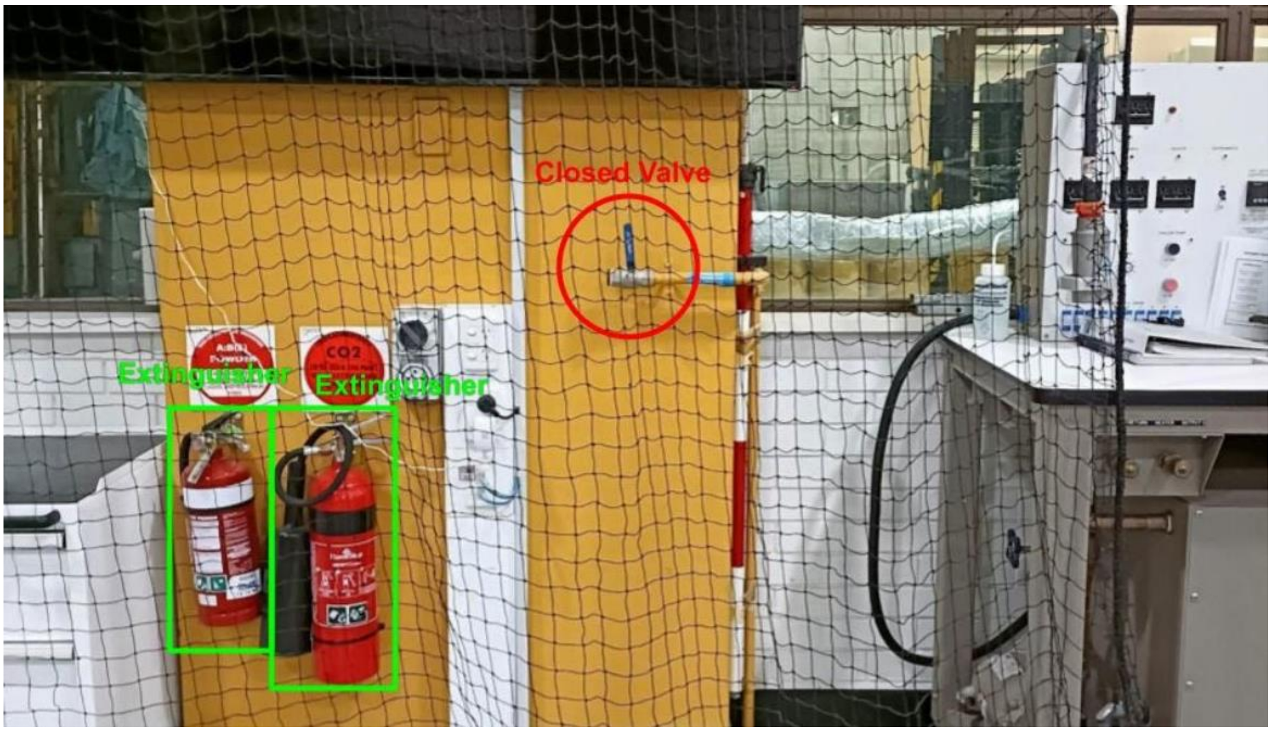
Alexey (2020). *AlexeyAB/darknet*. [online] GitHub. Available at: https://github.com/AlexeyAB/darknet.

Jocher, G. (2020). *ultralytics/yolov5*. [online] GitHub. Available at: https://github.com/ultralytics/yolov5.

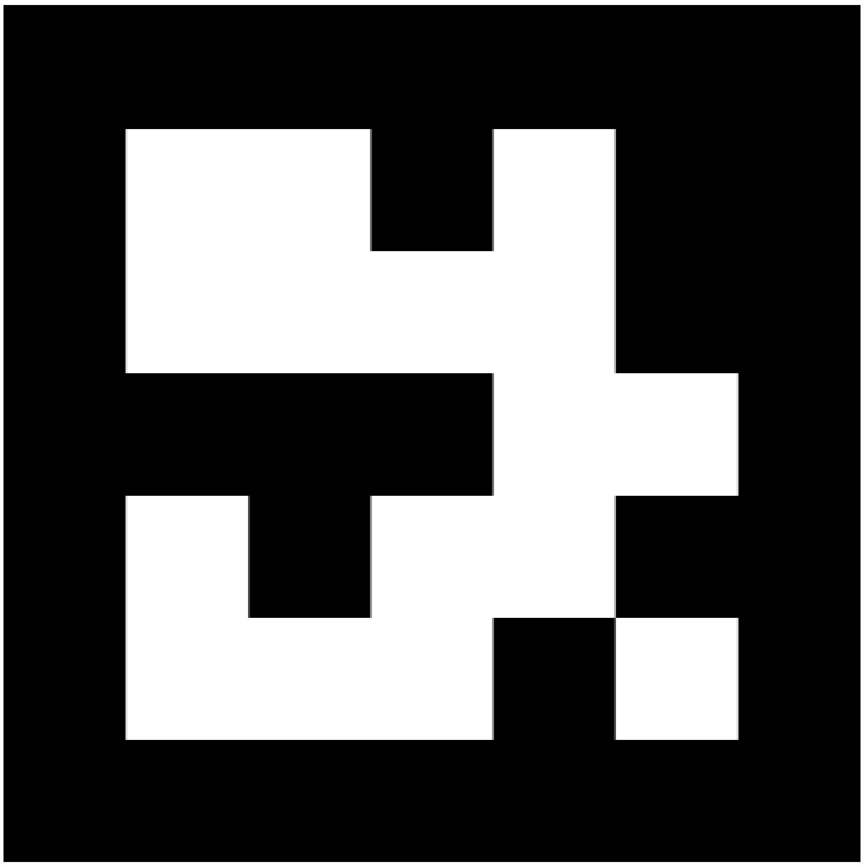
Redmon, J. (2012). *YOLO: Real-Time Object Detection*. [online] Pjreddie.com. Available at: https://pjreddie.com/darknet/yolo/.

8. Appendix

7.1. Appendix A – Targets: Extinguisher and Valve



7.2. Appendix B – Targets: ArUco Marker



7.3. Appendix C – YoloV3 & OpenCV Logos

7.4. Appendix D – WebcamVideoStreamOverride.py (Used for multi-threaded frame reading)

```
# import the necessary packages
from threading import Thread
import cv2

class WebcamVideoStream:
    def __init__(self, src=0, name="WebcamVideoStream",UseModel = False, UseAruco
= False):
        # initialize the video camera stream and read the first frame
        # from the stream
        self.stream = cv2.VideoCapture(0)
        #self.stream = cv2.VideoCapture('1664152920.7081351basicvideo2.avi')
        self.stream.set(cv2.CAP_PROP_FRAME_WIDTH, 320)
        self.stream.set(cv2.CAP_PROP_FRAME_HEIGHT, 320)

        (self.grabbed, self.frame) = self.stream.read()

        # initialize the thread name
        self.name = name

        # initialize the variable used to indicate if the thread should
        # be stopped
        self.stopped = False

    def start(self):
        # start the thread to read frames from the video stream
        t = Thread(target=self.update, name=self.name, args=())
        t.daemon = True
        t.start()
        return self

    def update(self):
        # keep looping infinitely until the thread is stopped
        while True:
            # if the thread indicator variable is set, stop the thread
            if self.stopped:
                print("stop 4")
                cv2.destroyAllWindows()
                self.stream.release()
                return

            # otherwise, read the next frame from the stream
            (self.grabbed, frame) = self.stream.read()
            self.frame = frame

    def read(self):
        # return the frame most recently read
        return self.frame, self.grabbed

    def stop(self):
        # indicate that the thread should be stopped
        print("stop 3")
        self.stopped = True
        self.close()
```

7.5. Appendix E – ArucoDetect.py Used for ArUco Detection

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 29 09:00:00 2022

@author: Alex
Image Detection with yolov3 and opencv
"""

import numpy as np
import cv2
import sys
#from utils import aruco_display
#from utils import ARUCO_DICT
import argparse
import time

matrix_coefficients = np.load("calibration_matrix.npy")
distortion_coefficients = np.load("distortion_coefficients.npy")
aruco_dict_type = cv2.aruco.DICT_5X5_100
cv2.aruco_dict = cv2.aruco.Dictionary_get(aruco_dict_type)

class Aruco:
    def __init__(self):
        self.timeStart = time.time()
        self.xyz = "Unknown"
        self.distance = 10000

    def start(self):
        return

    def update(self, frame):
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        parameters = cv2.aruco.DetectorParameters_create()
        self.corners, self.ids, rejected_img_points = cv2.aruco.detectMarkers(gray,
cv2.aruco_dict,parameters=parameters,
        cameraMatrix=matrix_coefficients,
        distCoeff=distortion_coefficients)

    def draw(self, frame):
        # If markers are detected
        corners = self.corners
        ids = self.ids
        if len(corners) > 0:
            for i in range(0, len(ids)):
                # Estimate pose of each marker and return the values rvec and
tvec---(different from those of camera coefficients)
                rvec, tvec, markerPoints =
cv2.aruco.estimatePoseSingleMarkers(corners[i], 0.0225 * 2, matrix_coefficients,
                                                                                   distortion
_coefficients)

                self.xyz =
f"X:{str(round(tvec[0][0][0],3))},Y:{str(round(tvec[0][0][1],3))},Z:{str(round(tve
c[0][0][2],3))}"
                self.distance = round(tvec[0][0][2],3)

                cv2.aruco.drawDetectedMarkers(frame, corners)

                for (markerCorner, markerID) in zip(corners, ids):
                    # extract the marker corners (which are always returned in
                    # top-left, top-right, bottom-right, and bottom-left order)
                    corners = markerCorner.reshape((4, 2))
                    (topLeft, topRight, bottomRight, bottomLeft) = corners
                    # convert each of the (x, y)-coordinate pairs to integers
                    topRight = (int(topRight[0]), int(topRight[1]))
                    bottomRight = (int(bottomRight[0]), int(bottomRight[1]))
                    bottomLeft = (int(bottomLeft[0]), int(bottomLeft[1]))
                    topLeft = (int(topLeft[0]), int(topLeft[1]))

                    # compute and draw the center (x, y)-coordinates of the ArUco
                    # marker
                    cX = int((topLeft[0] + bottomRight[0]) / 2.0)
                    cY = int((topLeft[1] + bottomRight[1]) / 2.0)
                    cv2.circle(frame, (cX, cY), 4, (0, 0, 255), -1)
                    # draw the ArUco marker ID on the image
                    cv2.putText(frame, str(markerID),(topLeft[0], topLeft[1] - 10),
cv2.FONT_HERSHEY_SIMPLEX,
                        0.5, (0, 255, 0), 2)

                    cv2.putText(frame,str(round(tvec[0][0][0],3)) + " " +
str(round(tvec[0][0][1],3)) +
                    " " + str(round(tvec[0][0][2],3)),(bottomRight[0],
bottomRight[1] - 10), cv2.FONT_HERSHEY_SIMPLEX,
                    0.5, (0, 255, 0), 2)


                # Draw Axis
                cv2.aruco.drawAxis(frame, matrix_coefficients,
distortion_coefficients, rvec, tvec, 0.01)

        self.frame = frame
```

7.6. Appendix F – YoloModel.py used to identify objects with the Yolov3 Model

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 29 09:00:00 2022

@author: Alex, Connor
Image Detection with yolov3 and opencv
"""

import cv2
import numpy as np
import time

whT = 320
#define classes
classNames = ['Closed_valve','Fire_extinguisher','Open_valve']
#confidence threshold
confThreshold = 0
#The lower this value the more aggressive it will be at removing boxes
nmsThreshold = 0.3

modelConfiguration = 'yolov3-custom-tiny.cfg'
modelWeights = 'yolov3-custom-tiny_final.weights'
```

```
class Yolo:
    def __init__(self):
        self.timeStart = time.time()
        self.net = cv2.dnn.readNetFromDarknet(modelConfiguration,modelWeights)
        self.net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
        self.net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)

    def start(self):
        return

    def update(self, img):
        #opencv needs the images as blobs, here we convert these to the blob
format
        blob = cv2.dnn.blobFromImage(img,1/255,(whT,whT),[0,0,0],1,crop=False)
        self.net.setInput(blob)
        layerNames = self.net.getLayerNames()
        #0 is not included within the layerNames so this is taken into account
with
        #the -1
        outputNames = [layerNames[i-1] for i in
self.net.getUnconnectedOutLayers()]
        outputs = self.net.forward(outputNames)
        self.findObjects(outputs,img)
        self.img = img


    #function to find the object on the screen
    def findObjects(self, outputs,img):
        hT, wT, cT = img.shape
        #bounding box
        bbox = []
        #class identifier
        classIds = []
        #confidence intervals
        confs = []

        for output in outputs:
            for detection in output:
                scores = detection[5:]
                classId = np.argmax(scores)
                confidence = scores[classId]

                if classId == 1:
                    confThreshold = 0.8
                else:
                    confThreshold = 0.4

                if confidence > confThreshold:
                    #print(str(classId) + '   ' + str(confidence))
                    #w and h of the bounding box
                    w,h = int(detection[2]*wT) , int(detection[3]*hT)
                    #x and y are the centrepoint
                    x,y = int((detection[0]*wT)-w/2), int((detection[1]*hT)-h/2)
                    bbox.append([x,y,w,h])
                    classIds.append(classId)
                    confs.append(float(confidence))
        #output will be the indices of the bounding box to keep
        indices = cv2.dnn.NMSBoxes(bbox,confs,confThreshold,nmsThreshold)

        self.target = ""
        self.targetConfidence = 0

        for i in indices:
            if(int(confs[i]*100) > self.targetConfidence):
                self.targetConfidence = int(confs[i]*100)
                self.target = classNames[classIds[i]].upper()
            box = bbox[i]
            x,y,w,h = box[0],box[1],box[2],box[3]
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,255),2)
            cv2.putText(img,f'{classNames[classIds[i]].upper()}
{int(confs[i]*100)}%',
                        (x,y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.6,(255,0,255),2)
```

7.7. Appendix G – Custom yolov3-tiny.cfg

```
[net]
# Testing
#batch=1
#subdivisions=1
# Training
batch=64
subdivisions=16
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 6000
policy=steps
```

```
steps=4800,5400
scales=.1,.1

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
```

```
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

###########

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=24
activation=linear



[yolo]
mask = 3,4,5
```

```
anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
classes=3
num=6
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1

[route]
layers = -4

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 8

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=24
activation=linear

[yolo]
mask = 0,1,2
anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
classes=3
num=6
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
```