INTERNSHIP REPORT

Object Detection Robot with GPS Navigation and Collision Avoidance

Naval Science & Technological Laboratory

Defense Research & Development Organization

Ministry of Defense, Government of India

Visakhapatnam

1 August 2022

Sai Srinivas Tatwik Meesala
Robotics Intern
tatwikroy@gmail.com

Table of Contents

NAVAL SCIENCE & TECHNOLOGY LABORATORY

CERTIFICATE

This is to certify that Sai Srinivas Tatwik Meesala, Bachelor in Robotics, IV Year student at Arizona State University, USA has successfully completed his project work on **"Design and Development of Object Detection/Tracking Robot with Autonomous Navigation and Collision Avoidance**" during 17-05-2022 to 21-07-2022.

ACKNOWLEDGEMENT

Naval Science & Technological Laboratory (NSTL) is one of the premier establishments of the Defense Research and Development Organization (DRDO). This establishment is committed to providing state-of-the-art products and services to the Indian Navy. It gives Sai Srinivas Tatwik a great pleasure to complete his research training at NSTL.

Sai Srinivas Tatwik is highly obliged to Dr. Y. Sreenivas Rao, Director NSTL, for allowing me to associate with this esteemed establishment as a summer intern.

Sai Srinivas Tatwik takes the privilege to express his sincere thanks to Technology Director (HRD), Mr. Kushal Bhattacharya, Scientist 'F' for allowing him to work in this prestigious institution and for entrusting me and permitting him to carry out project work at NSTL.

Most importantly, Sai Srinivas Tatwik expresses his deepest gratitude to Mr. B Suresh Kumar Scientist ‘E’, and Mr. Mahesh Gopi Scientist 'C' for their unflagging guidance throughout the progress of this project as well as for their valuable contribution to the preparation and compilation of the text.

Sai Srinivas Tatwik is thankful to all those who have helped him with the successful completion of the training at NSTL.

ABSTRACT

The goal of this project is to create an autonomous robot that detects objects of interest and uses GPS navigation and image processing to avoid collisions. A self-navigating unmanned vehicle that travels to a user-specified destination coordinates wirelessly. The navigation of the unmanned vehicle uses GPS and magnetometer, and a vision-based collision avoidance system has also been built. Utilizing image processing from the camera feed of the vehicle's front view to prevent collisions. The research uses YOLOv5 (You Only Look Once), which is a real-time object detection system, to detect and track the object of interest.

I. INTRODUCTION

The goal of this project is to create an autonomous robot that detects objects of interest and uses GPS navigation. Image processing is employed to avoid collisions. A self-navigating unmanned vehicle travels to user-specified destination coordinates wirelessly. The navigation of the unmanned vehicle uses GPS and magnetometer, and a vision-based collision avoidance system is being used. Utilizing image processing from the camera feed of the vehicle's front view to prevent collisions. The research uses YOLOv5 (You Only Look Once), which is a real-time object detection system to detect and track the object of interest. In this paper, Sai Srinivas Tatwik presents the design and implementation of the autonomous vehicle's algorithm. The created algorithm may be extended to a number of tasks, such as monitoring, research, and navigating to areas that may be unsafe for people.

The robot used during the course of the project is the JetBot, which was developed by NVIDIA. JetBot uses the Jupyter Notebook programming interface. The Jupyter Notebook uses text, code, and graphical display to interact with the robot. Jupyter Notebook supports more than 40 programming languages, including Python, R, Julia, etc. For this project, the author used the Python programming language to design the algorithm.

The sensors used are Ublox NEO 6M, MPU-6500, and Raspberry Pi Camera V2.1. The Global Positioning System and magnetometer modules help to track the current position and the heading direction of the robot. The camera module is used to detect collisions and detect and track objects of interest.

The author provides requirements for important topics like object detection, obstacle avoidance, as well as algorithm design for the research. and concluding with the data gathered following the system's implementation.

II. PROBLEM STATEMENT

The Naval Science & Technological Laboratory (NSTL) wants an unmanned vehicle that navigates to a given location and detects and tracks any object of interest. Sai Srinivas Tatwik hopes to solve this issue by creating an autonomous ground robot that navigates to a designated location and detects the object without any human intervention. The purpose of this project is to create an autonomous ground vehicle that can self-navigate using GPS coordinates, avoid collisions, and detect any object of interest using a camera.

III. PREREQUISITES

The components used in this project are:

- JetBot
- Ublox NEO 6M
- MPU-6500
- Raspberry Pi Camera V2.1

The JetBot is the robot utilized in this project. Based on the NVIDIA Jetson Nano, the JetBot is a free and open-source robot. JetBot comes with a series of Jupyter notebooks that cover fundamental robotics concepts like programmable motor control and more complex subjects like developing a custom AI for collision avoidance [1].

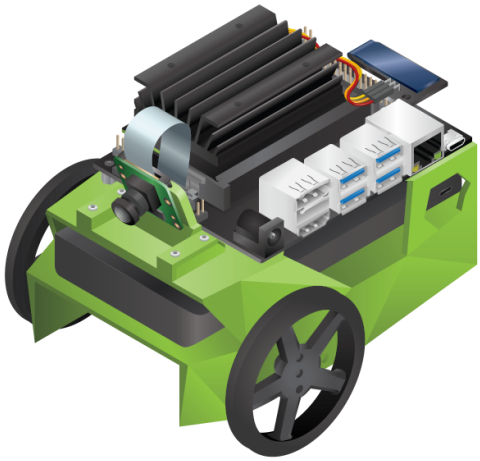


Figure 1. Animated View of JetBot [1]

The research uses Ublox NEO 6M as the Global Positioning System (GPS) module to determine the latitude and longitude location of the robot. The MPU-6500, a System in Package (SiP) with a 3-axis gyroscope, 3-axis accelerometer, and 3-axis magnetometer, is one of the two

chips included in the MPU-9250. This project uses MPU-9250's magnetometer as a compass to track the robot's heading.

The following are the dependencies used for the algorithm:

- sys
- os
- time
- argparse
- numpy
- cv2
- tensorrt
- pycuda.driver
- pycuda.autoinit
- jetbot
- tool.utils
- absl
- IPython.display
- PIL
- Processor
- Visualizer
- Jetson.GPIO

IV. GLOBAL POSITIONING AND HEADING

Compass

The magnetic field or magnetic dipole moment around the Earth is measured via a 3-axis magnetometer.

To determine the robot's facing orientation, one has to know the magnetometer values in the X and Y directions. When the magnetometer reading exceeds 0,

$$heading = 90 - arctan\left(\frac{magX}{magY}\right) \quad (1)$$

In the above equation, the current heading of the robot, *magX* is magnetometer values in the X direction, and *magY* is magnetometer values in the Y direction. When *magY* is less than 0,

$$heading = 270 - arctan\left(\frac{magX}{magY}\right) \quad (2)$$

If *magY* is 0 and *magX* is less than 0, then the current heading is 180° and when *magX* is more than 0°. This outputs the data in degrees, allowing the robot to navigate to the target location [2]. Please refer to Appendix A for the current heading code.

Steering Angle

The GPS module outputs the latitude and longitude in terms of degrees. Both the GPS module and compass data are needed to get the steering angle towards the target location concerning the robot's current location and heading. To calculate the steering angle,

$$Steering\ Angle = arctan\left(\frac{X}{Y}\right) - Current\ Heading \quad (3)$$

Where *"x"* and *"y"* are

$$x = cos(targLat) * sin(currLon - targLon) \quad (4)$$

$$y = cos(currLat) * sin(targLat) - sin(currLat) * cos(targLat) * cos(currLon - targLon) \quad (5)$$

where *targLat* and *targLon* represent the latitude and longitude of the target location and currLat and currLon are the current location of the robot [3]. The output, in this case, ranges from 0° to 360°.

The steering angle should be output between -180° and 180°. So, if the angle is greater than 180°,

$$Steering\ Angle = Steering\ Angle - 360 \quad (6)$$

Please refer to Appendix B for Steering Angle code.

Distance between Current and Target Location

For the robot to know how far it is from the target location, calculating the distance between two points' latitude and longitude is done using the Haversine Formula, a geodetic measuring function, represented as

$$Distance = R * c \quad (7)$$

Where "*R*" is the radius of the earth, 6378.137 KM or 6378.137*1000 m, and *"c"* is

$$c = 2 * arctan(\frac{sqrt(a)}{sqrt(1-a)}) \quad (8)$$

And *"a"* is

$$a = sin(dLat/2) * sin(dLat/2) + cos(curLat * pi / 180) \quad (9)$$

$$* \ cos(lat2 \ * \ pi \ / \ 180) * sin^2(dLon/2)$$

where

$$dLat \ = \ lat2 \ - \ curLat \tag{10}$$

$$dLon \ = \ lon2 \ - \ curLon \tag{11}$$

here, latitude and longitude differences between the current and target locations are represented by *dLat* and *dLon* [4].

Way Point Generation

A waypoint can also be generated if the target heading and distance are given from the robot's current location. To calculate the target position coordinate,

$$targLat \ = (asin(sin(currLat) \ * cos(Ad) \ + cos(currLat) \ * sin(Ad) \ * cos(targHead))) \tag{12}$$

$$targLong \ = \ currLon \ + (atan(sin((targHead)) \ * sin((Ad)) \ * cos((currLat))/cos((Ad)) \ - sin((currLat)) \ * sin((targLat)))) \tag{13}$$

where Ad is the angular distance

$$\text{Ad} = \text{dist/R} * 180/\pi$$

dist is the distance from the robot's current position to the target location, and *R* is the radius of the earth. And *targHead* is the angle between the robot and the target location [3].

Please refer to Appendix C for the algorithm for the distance between the two locations and waypoint generation.

V. MOVING ROBOT TO TARGET LOCATION

In the JetBot, the right and left wheels are controlled individually. The algorithm incorporates differential drive to make the robot efficiently move toward the target location. A two-wheeled drive system has individual actuators for each wheel; the differential drive. The robot doesn't need to steer at all since it can change its direction simply by altering the relative pace at which its wheels rotate. To prevent the vehicle from tipping, castor wheels are often used on robots with this type of propulsion [5].

Here, the logic is to change the robot's direction by changing the wheels' speed in real-time. When the left wheel speed is less than the right wheel speed, the robot tends to go to the left and vice-versa. To make this possible, initially, the speed of the wheel should be remapped depending on the steering of the robot

$$mapping\ function\ = \frac{Steering\ Angle}{180} * Drive\ Speed \tag{14}$$

where Drive Speed is the desired speed of the robot, and Steering Angle is calculated from the formula mentioned previously.

If the robot desires to move left, i.e., the Steering Angle is less than 0

$$Left\ Wheel\ Speed\ =\ Drive\ Speed\ -\ mapping\ function \tag{15}$$

$$Right\ Wheel\ Speed\ =\ Drive\ Speed \tag{16}$$

Here, only the left wheel speed is reduced to make the robot steer towards the left.

If the robot desires to move right, i.e., the Steering Angle is greater than 0

$$Left\ Wheel\ Speed\ =\ Drive\ Speed \tag{17}$$

$$Right\ Wheel\ Speed\ =\ Drive\ Speed\ -\ mapping\ function \quad (18)$$

Here, only the right wheel speed is reduced to make the robot steer towards the right.

The Jetbot's wheels are bidirectional; they can turn in opposite directions to each other to make faster turns. To create aggressive moments when the robot is facing the opposite direction toward the target location,

```
if (steeringAngle < -90) and (steeringAngle >= -180):
    Left Wheel Speed = Drive Speed -2*mapping function
    Right Wheel Speed = Drive Speed
if (steeringAngle > 90) and (steeringAngle <= 180):
    Left Wheel Speed = Drive Speed
    Right Wheel Speed = Drive Speed-2*mapping function
```

Please refer to Appendix D for the Differential Drive code. Appendix E is the algorithm that navigates the robot based on its current location to the target location's latitude and longitude.

VI. COLLISION AVOIDANCE

Rather than using Light Detecting and Raigin (LIDAR) or ultrasound sensors, the robot uses vision-based collision avoidance. The single-camera setup used thresholding to detect any obstacle in front of the robot. When the robot detects an obstacle, it steers away from it.

First, the camera feed is converted to grayscale. Then a dark threshold should be declared, *Dark_TH*. The dark threshold is used to compare the number of obstacles that are on the left, right, and in front of the robot. There is another variable named *row_start*. This is to ignore any dark zones that are located below the declared level.

From that grayscale image, the robot extracts and divides the area into three segments:

- *dark_area*: This area is defined by the number of dark particles that are located in the center of the camera feed.
- *port_area*: This area contains the dark particles that are located on the left (port) of the camera feed.
- *stbd_area*: This area contains the dark particles that are located on the right (starboard) of the camera feed.

The robot takes the product of rows and columns of the areas of one of the sides, and then subtracts the resulting area from the corresponding area by any nonzero pixels. *Image.shape()* returns the number of rows (height), number of columns (width), and channel of the image. Zero pixels denote the black color and *countNonZero()* is used to find the number of black or white pixels. The resultant of the equation gives the required dark area in the respective zone.

Then a center threshold is declared, *CEN_TH*. If the robot detects more dark areas are present than the center threshold, the robot moves back and takes a left to avoid the obstacle in front of it. The robot may also take the right path depending on the programmer's preference.

The robot turns to the right or left depending on whether the port region (left side) or starboard region (right side) respectively of the robot has more dark patches than the Dark Threshold.

If all paths are clear (above *row_start*), the robot may move toward its desired destination. Please refer to Appendix F for the algorithm for collision avoidance.

VII. YOLOv5 INTEGRATION

You Only Look Once (YOLO) is one of the best real-time object detection systems. YOLOv5 is the fifth iteration of the family of compound-scaled object identification models trained on the COCO dataset. It has minimal features for Test Time Augmentation (TTA), model ensembling, hyperparameter evolution, and export to ONNX, CoreML, and TFLite. From YOLOv1 to YOLOv4, there are four variations of YOLO that are a component of the Darknet infrastructure. YOLOv5 is the next version in the YOLO family.

Glenn Jocher created the project on GitHub under the Ultralytics group. The framework was utilized as PyTorch, and the language used was Python. It is a collection of object detection models on its own. Starting with extremely tiny models that can deliver real-time FPS on edge devices and moving all the way up to very big and precise models designed for cloud GPU installations [6].

Training New Models

To train a new model, important steps include:

- The dataset preparation.
- Training of the models.
- Performance comparison.
- Inference on images and videos.

The researcher trained the model using the instructions available on “Custom Object Detection Training using YOLOv5” [6].

After training the model, it produces a .yaml file. YAML is frequently used as a format for configuration files, but its support for object serialization makes it a competitive alternative

to JSON [7]. Then, it is converted to the ONNX model. Open Neural Network Exchange (ONNX) is a widely used open-sourced format created to express machine learning models. To allow AI developers to use models with a variety of frameworks, tools, runtimes, and compilers, ONNX defines a common set of operators—the fundamental components of machine learning and deep learning models—and a single file format [8].

Then the ONNX file is transferred to JetBot and then converted to a .engine file using TensorRT, which is already available on JetBot's system. TensorRT is NVIDIA's SDK for the high-performance inference that may be achieved by optimizing learned deep learning models. For trained deep learning models, TensorRT includes a deep learning inference optimizer and an execution runtime.

TensorRT executes a deep learning model faster and with less latency after the model is trained in the framework of the user's choice [9].

VIII. OBJECT DETECTION

YOLO predicts the use of bounding boxes. In the Anchor Boxes with dimensions, YOLO predicts the width and the height as offsets from the cluster centroid. YOLO predicts the center coordinates of the box relative to the location of the filter application using a sigmoid. Each model that is trained using YOLO is stored as a class. For this project, "Object1" and "Object2" have been trained using the instructions stated previously. The names "Object1" and "Object2" may not be disclosed as part of the Non-Disclosure Agreement signed by the author with NSTL.

The only thing to know is that "Object2" is situated on top of "Object1," yet "Object2" is no longer visible as the robot approaches the object of interest due to the location of the robot's camera.

First the .engine file is loaded by:

```
opt={'eng':'(Location of ".engine
file")','classes':2,'conf':0.3,'img_size':640,'name':'data/objects.names','save':'T
rue','show_out':'True','search':'True','col_avoid':'True'}
```

here, the location of the .engine file is stored inside *"eng"*. From it, both the classes (trained models) are loaded and the *save, shoe_out, search* and *col_avoid* are set to True to save the camera feed, save the output in the disease location, and spiral search for the object, and establish the collision avoidance algorithms respectively. Then a new function is called "*main()"*. Here we input the engine path, image size, processor of the classes, and camera feed.

Then the image from the camera is extracted and each new frame number is stored inside the variable *"frame_id"*. Then, the information about each class is stored. The position, area, center, and dimensions (height and width) of the box in each class are extracted.

Then the area thresholds for boundary restrictions of Object1 and Object2 are declared as "*BOUNDARY_TH*=0.2" and "*BOUNDARY_TH_2*=0.05", respectively.

To navigate the robot towards the object of interest, the algorithm is written in the following way:

- If the object of interest is in the frame and the Area of the "Object1" is less than "*BOUNDARY_TH"* then move the robot forward towards the object.
- Once the object of interest is detected and is lost in the frame of the image then, the previous Yaw angle should be extracted from the magnetometer and the robot should direct itself towards the previous Yaw angle until the object is back in the frame.
- If the Area of the "Object1" is greater than *BOUNDARY_TH*, that means the robot has reached the object of interest and the robot sends a message to the user that the destination has been reached.
- If the Area of the "Object2" is greater than *BOUNDARY_TH_2*, that means the robot has reached the object of interest and the robot sends a message to the user that the destination has been reached.

The live camera feed can be displayed by:

```
retimg=visualizer.draw_results(img, boxes, confs, classes,class_names,BOUNDARY_TH)
retimg=cv2.cvtColor(retimg,cv2.COLOR_BGR2RGB)
display(Image.fromarray(retimg))
time.sleep(0.5)
clear_output(True)
```

To save the camera feed:

```
retimg=visualizer.draw_results(img, boxes, confs, classes,class_names,BOUNDARY_TH)
cv2.imwrite('det-output/det-YoloImg_'+str(image_num)+'.jpg',retimg)
image_num = image_num + 1
```

Please refer to Appendix G for Box Position Algorithm swing YOLO

IX. SOFTWARE DESIGN

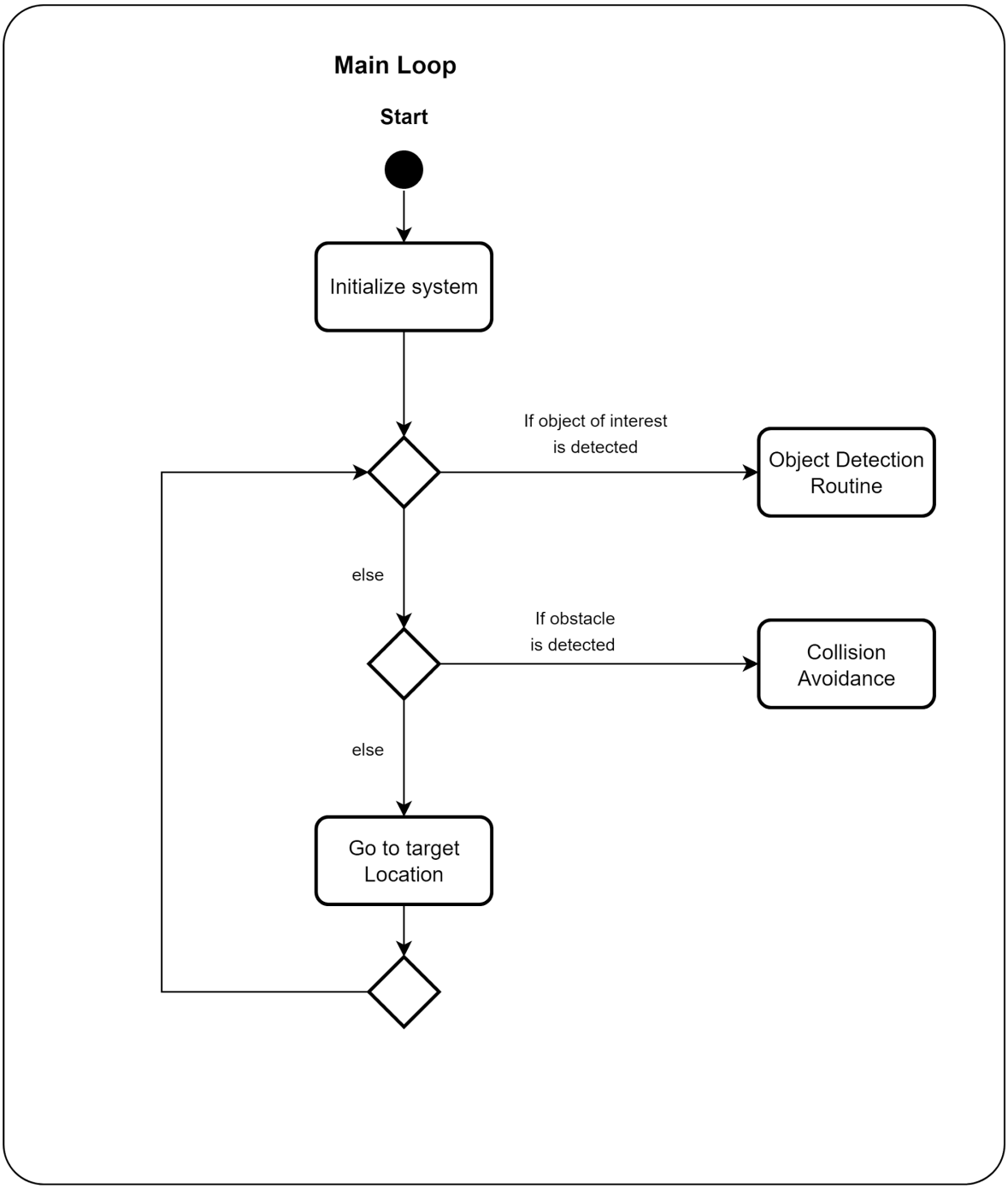


Figure 2. Main Loop flowchart

Please refer to Appendix H for Main Routine Function.

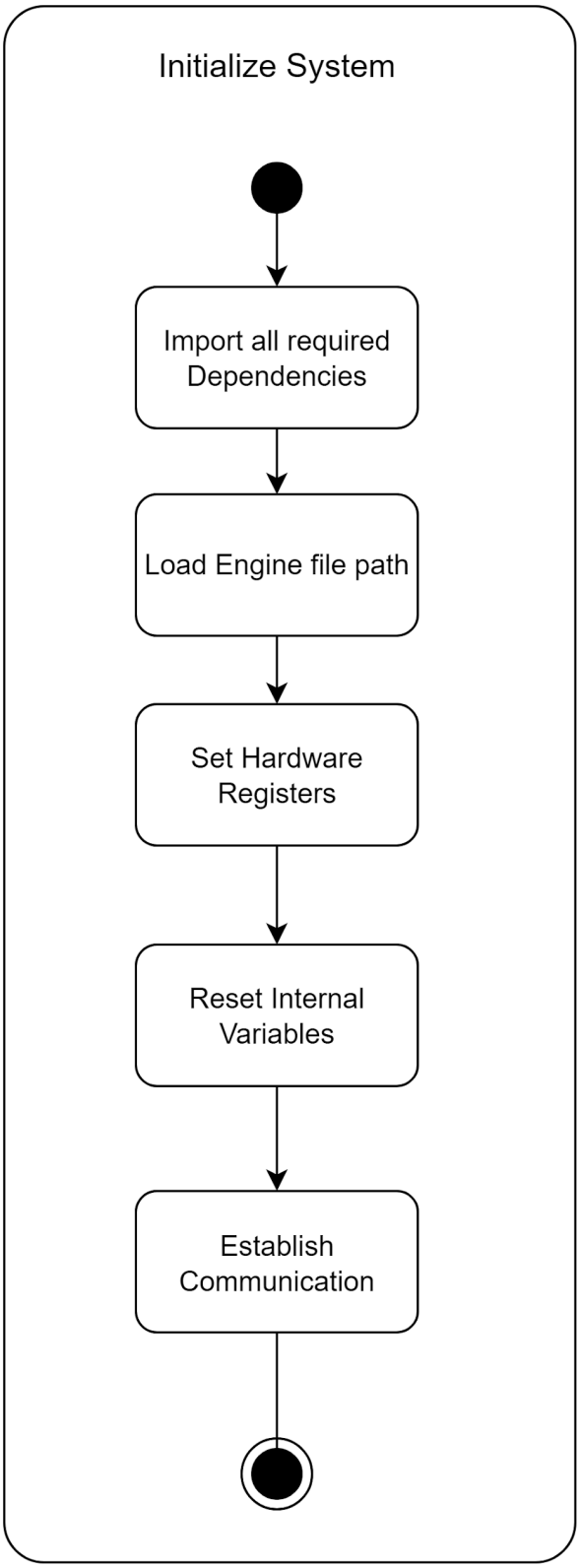


Figure 3. Flowchart for System Initialization

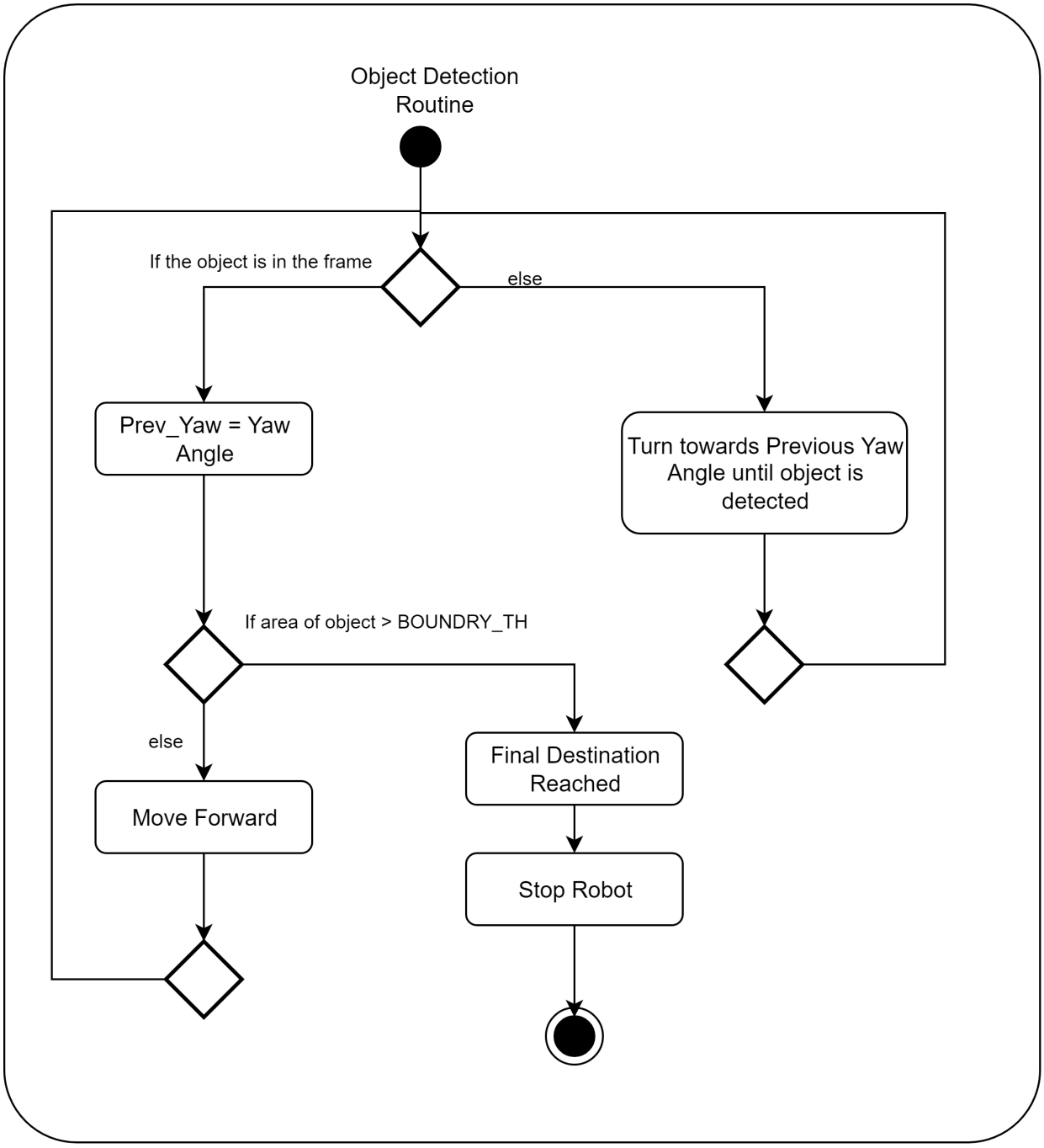


Figure 4. Subroutine for Object Detection

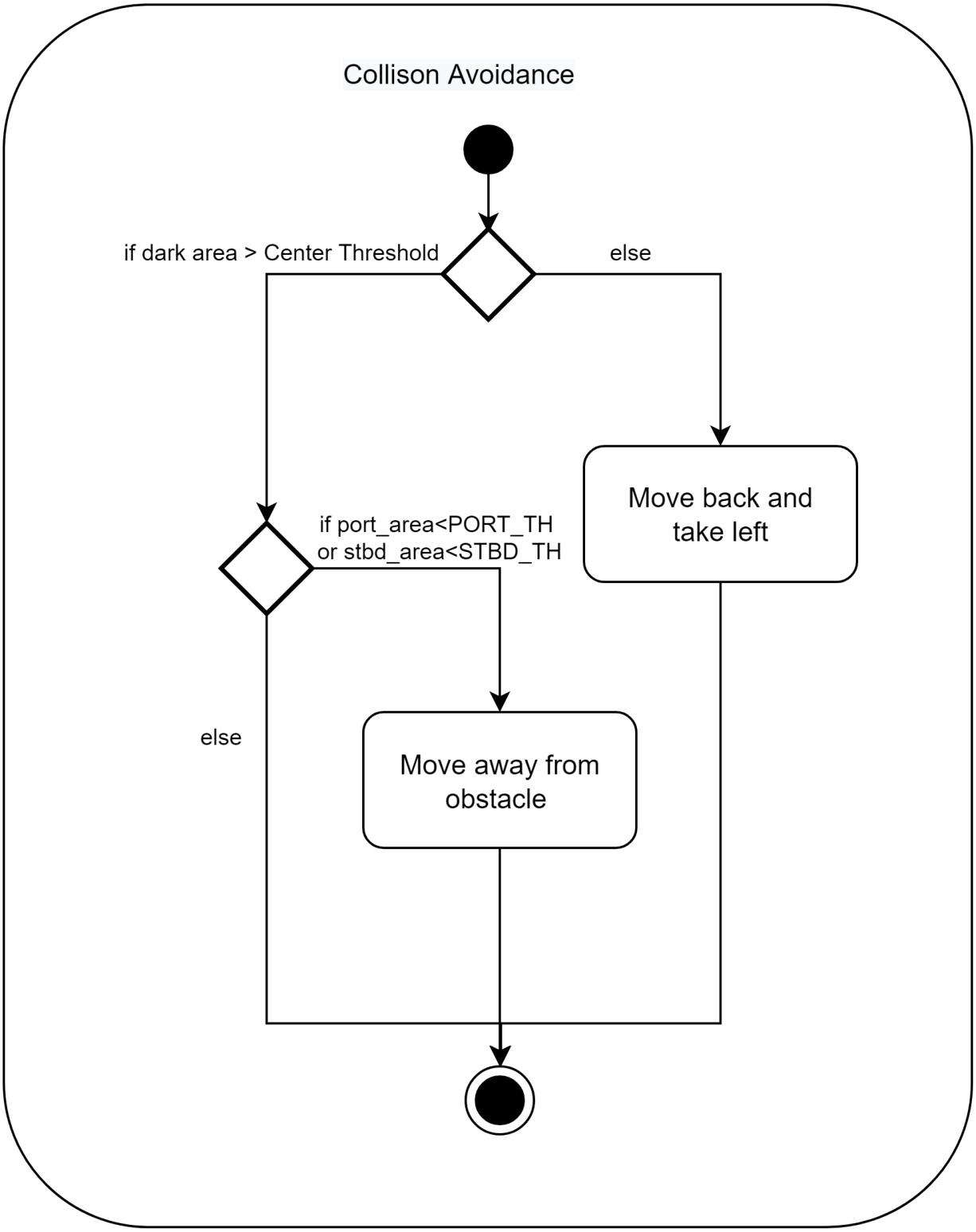


Figure 5. Subroutine for Collision Avoidance

X. TESTING

Collision Avoidance

The following are the images collected after running the collision avoidance algorithm:

In Figure 6 below, the robot was able to detect the obstacle that was located towards its port side (left).

Figure 6 (a & b). Original and Extracted Images after running collision avoidance

In Figure 7 below, the robot was able to detect the obstacle that was located towards its port side (left).

Figure 7 (a & b). Original and Extracted Images after running collision avoidance

In Figure 8 below, the robot was not able to detect the obstacle that was located in front of it. As the color of the pot and the wall behind were similar, the algorithm was not able to

detect any shade difference. As a result, it created a false negative instance and failed to detect the obstacle in front of it.

Figure 8 (a & b). Original and Extracted Images after running collision avoidance

In Figure 9 below, the robot was able to detect the obstacle that was located toward its starboard side (right).

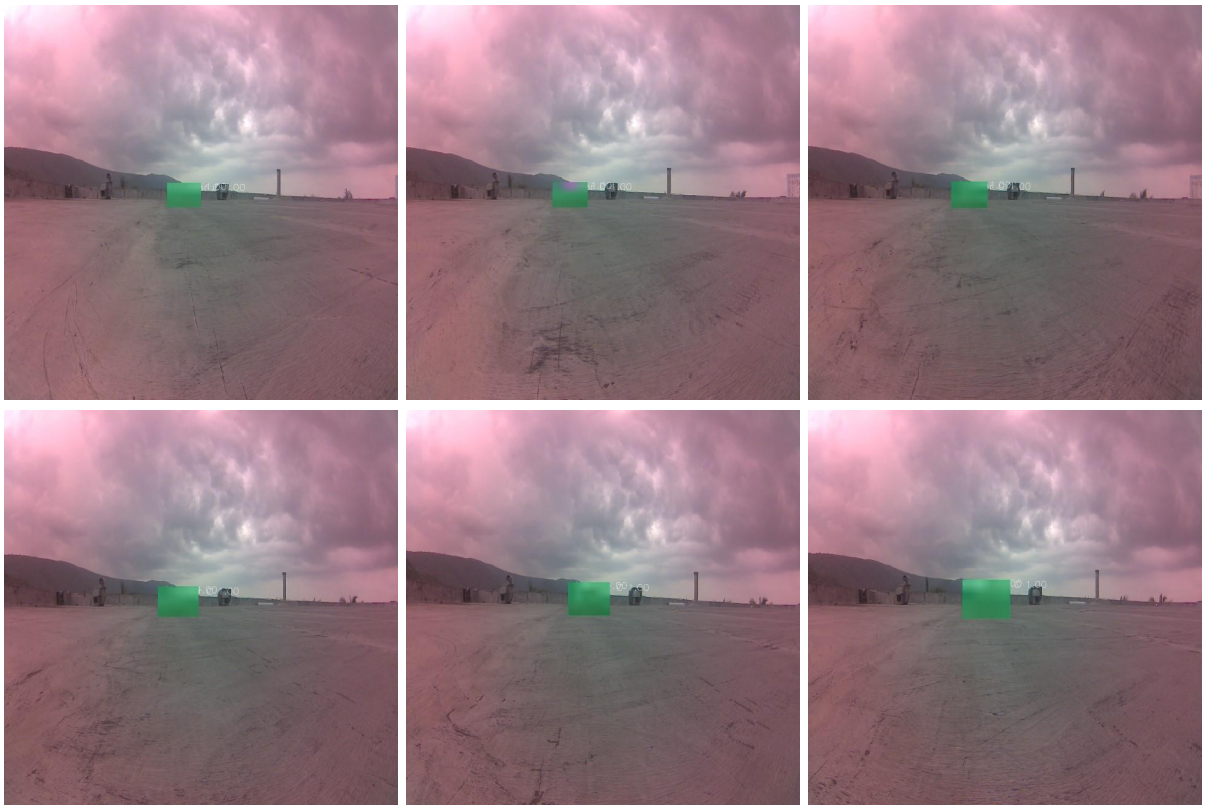
Figure 9 (a & b). Original and Extracted Images after running collision avoidance

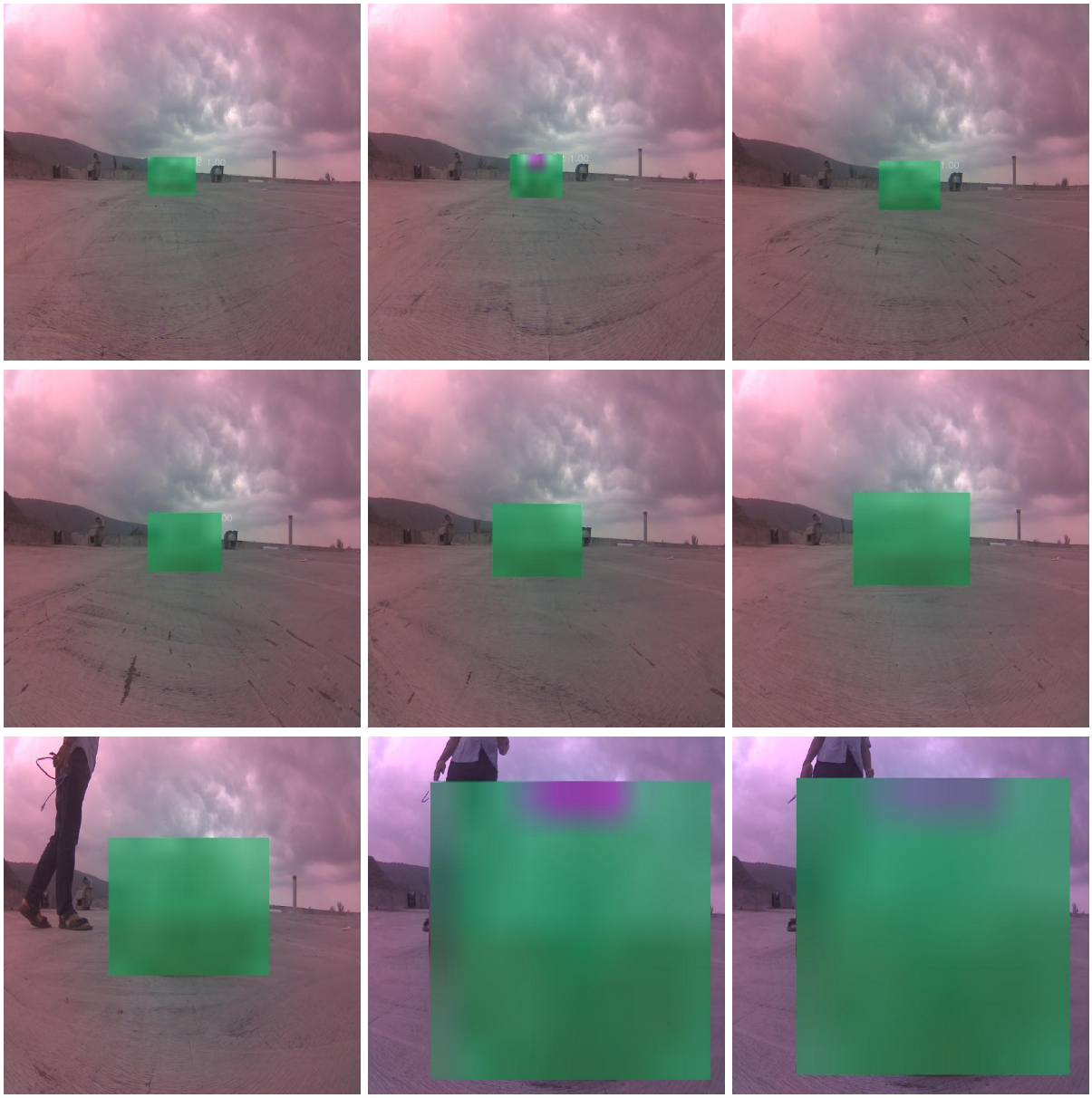
In Figure 10 below, the robot was able to detect the obstacle in front of it but it did not avoid it as it was less than the dart threshold.

Figure 10 (a & b). Original and Extracted Images after running collision avoidance

Object Detection and Tracking

Figures 11 to 26 are collected after the robot was able to detect Object1 and Object2. The images “Object1” and “Object2” are blurred as part of the Non-Disclosure Agreement signed by the author with NSTL.





Figures 11 - 26. Objects detected using YOLO and tracked by the robot

As the image approaches the boundary threshold (shown in Figure 27), the robot stops and informs the user of the information.

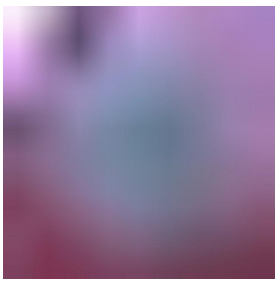


Figure 27. Image of the Object that exceeds border threshold

XI. CONCLUSION

The Naval Science & Technological Laboratory (NSTL) was looking for an autonomous vehicle that could find its way to a specific location while also detecting and following any object of interest. Sai Srinivas Tatwik seeks to find a solution to this problem by building an autonomous ground robot that finds its way to a predetermined area and detects the object without human assistance. The goal of this project was to build a self-navigating ground vehicle that uses GPS coordinates to prevent collisions, self-navigates, and uses a camera to detect any object of interest.

The autonomous unmanned vehicle wirelessly traveled to the user-specified destination coordinates. The unmanned vehicle uses a vision-based collision avoidance system in addition to GPS and a magnetometer for navigation. utilizing visual processing to avoid collisions from the front-facing camera feed of the vehicle. A real-time object detection system called YOLOv5 (You Only Look Once) was utilized in the study to find and follow the object of interest. The design and execution of the algorithm for the autonomous vehicle were presented. The developed algorithm may be used for a variety of activities, including research, monitoring, and traveling to potentially dangerous places for people.

The research lays out the prerequisites for critical areas like object detection and obstacle avoidance with algorithm design. Finishing with the data gathered once the system was put into place.

The robot was able to successfully navigate to the specified waypoint. With the help of the differential drive, the robot smoothly moved to the final location. As seen in the Testing section, the robot was able to identify most obstacles in its path with only one false negative instance. With the help of the algorithm, the robot successfully avoids most collisions. For best

results, depending on the location's lighting conditions, the dark threshold should be adjusted accordingly. When the object of interest was detected by the camera, the robot was able to move toward the object and inform the user about its location.

You may refer to the following GitHub link to access the robot code, https://github.com/Tatwik19/Object-Detection-Robot-with-GPS-Navigation-and-Collision-Avoidance.

Recommendations for Future Work

Two cameras along with a distance-measuring sensor or LIDAR module may be used for better collision avoidance. By this method, any inaccuracies can be easily eliminated. A night vision system can be used. This enables the robot to navigate in darkness or poor weather conditions.

In this research, only two models have been trained for object detection; more objects can be trained for sophisticated use. Real-time multi-person keypoint detection software like MediaPipe or OpenPose may be used for the implementation of gesture control for the robot.

When the GPS module fails to connect to the satellite(s), the code generates an error. Better GPS modules, in conjunction with an IMU sensor, can be used to accurately measure the robot's position. The robot can be paired with another ground or aerial robot for swarming. This approach can more efficiently detect any object of interest.

References

[1] C. Yato and John, “Jetbot,” *GitHub*. [Online]. Available: https://github.com/NVIDIA-AI-IOT/jetbot/blob/master/docs/index.md. [Accessed: 22-Jul-2022].

[2] Honeywell, “Compass heading using magnetometers,” *AN203_Compass_Heading_Using_Magnetometers*. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/AN203_Compass_Heading_Using_Magnetometers.pdf. [Accessed: 22-Jul-2022].

[3] A. Upadhyay, “Formula to find bearing or heading angle between two points: Latitude longitude,” *IGISMAP*, 31-May-2019. [Online]. Available: https://www.igismap.com/formula-to-find-bearing-or-heading-angle-between-two-points-latitude-longitude/. [Accessed: 26-Jul-2022].

[4] A. Upadhyay, “Haversine formula - calculate geographic distance on Earth,” *Haversine formula - Calculate geographic distance on earth*, 20-Jun-2018. [Online]. Available: https://www.igismap.com/haversine-formula-calculate-geographic-distance-earth/. [Accessed: 22-Jul-2022].

[5] Klančar Gregor, Zdešar Andrej, Blažič Sašo, and Škrjanc Igor, *Wheeled Mobile Robotics from fundamentals towards Autonomous Systems*. Oxford: Butterworth-Heinemann, 2017.

[6] S. Rath, “Custom object detection training using yolov5,” *LearnOpenCV*, 25-May-2022. [Online]. Available: https://learnopencv.com/custom-object-detection-training-using-yolov5/. [Accessed: 25-Jul-2022].

[7] Erik, “YAML tutorial: Everything you need to get started in minutes,” *CloudBees*, 11-Dec-2018. [Online]. Available: https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started. [Accessed: 25-Jul-2022].

[8] ONNX, “Open Neural Network Exchange,” *ONNX*. [Online]. Available: https://onnx.ai/. [Accessed: 25-Jul-2022].

[9] NVIDIA, “NVIDIA TensorRT,” *NVIDIA Documentation Center*, Jul-2022. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/quick-start-guide/index.html. [Accessed: 25-Jul-2022].

APPENDICES

Appendix A

Current Heading Algorithm

```
def currentHeading(): #Get current Heading/Facing direction from
Magnetometer (0 to 360 deg)

    imu.readSensor()
    imu.computeOrientation()

    magX = imu.MagVals[0]
    magY = imu.MagVals[1]
    M_PI = math.pi
    offSet = 30 # Correction Angle

    if (magY > 0):
        heading = 90 - math.atan(magX/magY)*(180/M_PI)

    elif (magY < 0):
        heading = 270 - math.atan(magX/magY)*(180/M_PI)

    elif (magY == 0 and magX < 0):
        heading = 180

    elif (magY == 0 and magX > 0):
        heading = 0

    heading = heading + offSet
    if heading > 360:
        heading = heading - 360;
    elif heading < 0:
        heading = heading + 360;

    return heading
```

Appendix B

Steering Angle Algorithm

```
def getSteering(currHead, currLat, currLon, targLat, targLon): # Steering
angle towards target location wrt our current location and heading
    # currHead must be from -360 to 360;
    x = math.cos(np.deg2rad(targLat)) *
math.sin(np.deg2rad(currLon-targLon));
    y = math.cos(np.deg2rad(currLat)) * math.sin(np.deg2rad(targLat)) -
math.sin(np.deg2rad(currLat))* math.cos(np.deg2rad(targLat)) *
math.cos(np.deg2rad(currLon-targLon));
    mag = currHead;
    steering = -np.rad2deg(math.atan2(x,y)) - mag;
    if steering >= 180:
        steering = steering - 360;
    elif steering <= -180:
        steering = steering + 360;
    return steering
```

Appendix C

Distance between Two Points Algorithm and Waypoint Generation

```
def getDistance(curLat, curLon, lat2, lon2):  # generally used geo
measurement function
    R = 6378.137; # Radius of earth in KM
    dLat = lat2 * math.pi / 180 - curLat * math.pi / 180
    dLon = lon2 * math.pi / 180 - curLon * math.pi / 180
    a = math.sin(dLat/2) * math.sin(dLat/2) + math.cos(curLat * math.pi /
180) * math.cos(lat2 * math.pi / 180) * math.sin(dLon/2) * math.sin(dLon/2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
    d = R * c;
    return d * 1000; # meters



def getWayPoint(currHead, targHead, currLat, currLon, dist): # Get Waypoint
when target heading, distance and current location is given
    R = 6378.137 * 1000  # Radius of earth in M

    Ad = dist/R * 180/math.pi # Angular distance

    targHead = currHead + targHead # previous targHead is w.r.t North
(Bearing Angle)

    targLat = np.rad2deg(math.asin(math.sin(np.deg2rad(currLat)) *
math.cos(np.deg2rad(Ad)) + math.cos(np.deg2rad(currLat)) *
math.sin(np.deg2rad(Ad)) * math.cos(np.deg2rad(targHead))))
    targLong = currLon +
np.rad2deg(math.atan2(math.sin(np.deg2rad(targHead)) *
math.sin(np.deg2rad(Ad)) * math.cos(np.deg2rad(currLat)),
math.cos(np.deg2rad(Ad)) - math.sin(np.deg2rad(currLat)) *
math.sin(np.deg2rad(targLat))))

    return [targLat,targLong]
```

Appendix D

Differential Drive Algorithm

```
def mapf(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def differentialDrive(steeringAngle, driveSpeed):
    # steeringAngle must be from -180 to 180.

    leftDriveSpeed = driveSpeed;
    rightDriveSpeed = driveSpeed;
    if (steeringAngle <= 0) and (steeringAngle > -90):
        L = leftDriveSpeed - mapf(abs(steeringAngle), 0,180, 0,driveSpeed)
# slow left down
        R = rightDriveSpeed

    if (steeringAngle < -90) and (steeringAngle >= -180):
        L = leftDriveSpeed - 2*mapf(abs(steeringAngle), 0,180,
0,driveSpeed) # slow left down
        R = rightDriveSpeed


    if (steeringAngle > 0) and steeringAngle <= 90:
        L = leftDriveSpeed
        R = rightDriveSpeed - mapf(abs(steeringAngle), 0,180, 0,driveSpeed)
# slow right down

    if (steeringAngle > 90) and steeringAngle <= 180:
        L = leftDriveSpeed
        R = rightDriveSpeed - 2*mapf(abs(steeringAngle), 0,180,
0,driveSpeed) # slow right down

    L_R_velocity = [L, R,mapf(abs(steeringAngle), 0,180, 0,driveSpeed)]
    return L_R_velocity
```

Appendix E

Navigating the Robot to the Target Location Algorithm

```
def goToTarget(Speed,targetLat,targetLon): # Go to target location based on
your input lat and lon

    imu.readSensor() # Read IMU Sensor
    imu.computeOrientation()

    received_data = (str)(ser.readline()) #read NMEA string received
    GPGGA_data_available = received_data.find(gpgga_info)   #check for NMEA
GPGGA string

    if (GPGGA_data_available>0): # when GPS is active
        GPGGA_buffer = received_data.split("$GPGGA,",1)[1]  #store data
coming after "$GPGGA," string
        NMEA_buff = (GPGGA_buffer.split(','))
        nmea_time = []
        nmea_latitude = []
        nmea_longitude = []
        nmea_time = float(NMEA_buff[0]) + 53000                    #extract
time from GPGGA string
        nmea_latitude = NMEA_buff[1]                #extract latitude from
GPGGA string
        nmea_latitude_dir = NMEA_buff[2]                #Lat direction
        nmea_longitude = NMEA_buff[3]               #extract longitude from
GPGGA string
        nmea_long_dir = NMEA_buff[4]                #Long direction

        GPS_Fix = NMEA_buff[5]
        satellite_count = NMEA_buff[6]
        Alt = NMEA_buff[8]

        lat = float(nmea_latitude)
        lat = convert_to_degrees(lat)
        longi = float(nmea_longitude)
        longi = convert_to_degrees(longi)

        currentHeading1 = currentHeading()
        print("currentHeading:",currentHeading1)
        currentLat = lat
        currentLon = longi

        print('Time: ',nmea_time)
        print('Current Location: ',currentLat,',',currentLon)

        driveSpeed = Speed
```

```
        steerAng = getSteering(currentHeading1, currentLat, currentLon,
targetLat, targetLon)
        distFromTraget = getDistance(currentLat, currentLon, targetLat,
targetLon)

        print('Steering Angle: ',steerAng)
        print('Distance from Target: ',distFromTraget)



        if distFromTraget > 3:
            drive = differentialDrive(steerAng, driveSpeed)
            LWSpeed = drive[0]
            RWSpeed = drive[1]
            print('Speed:\t',LWSpeed,'\t,',RWSpeed)
            robot.set_motors(LWSpeed,RWSpeed)
#             clear_output(wait=True)

        else:
            print("Location Reached")
            robot.set_motors(0,0)
```

Appendix F

Collision Avoidance Algorithm

```
BOUNDARY_TH=0.2# Area Threshold for boundary restriction
BOUNDARY_TH_SEN=0.05# Area threshold for sensor
PREV_MOVE='' # Previous move
#opt = parser.parse_args()

HEADING_SPEED=0.18

targLat = <Specify lat>
targLon = <specify longitude>

#Function for collision avoidance

def col_avoid(img,frame_id,lft,THRESH=250,row_start=470,DARK_TH=700): #
THRESH=80,row_start=470,DARK_TH=1500
    img_gray=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    #ret,th=cv2.threshold(img_gray,5,255,cv2.THRESH_BINARY)
    img_gray[img_gray>THRESH]=255
    img_gray[img_gray!=255]=0
    th=img_gray
    cv2.imwrite('Col-output/cam-out_'+str(frame_id)+'.jpg',th)

    CEN_TH=DARK_TH
    STBD_TH=DARK_TH
    PORT_TH=DARK_TH

    th_new=th[row_start::,100:500]
    th_port=th[row_start::,0:100]
    th_stbd=th[row_start::,500::]
# Area in the respective zone need to be subtracted by the non zero pixles
which gives dark zone
    dark_area=th_new.shape[0]*th_new.shape[1]-cv2.countNonZero(th_new)
    port_area=th_port.shape[0]*th_port.shape[1]-cv2.countNonZero(th_port)
    stbd_area=th_stbd.shape[0]*th_stbd.shape[1]-cv2.countNonZero(th_stbd)
#     print(f'port atea:{port_area},dark area:{dark_area},stbd
area{stbd_area}')


#     if frame_id%3==0 and frame_id>1:
    if dark_area<CEN_TH:
        print('forward path clear',dark_area)
        goToTarget(HEADING_SPEED,targLat,targLon)

    elif port_area<PORT_TH or stbd_area<STBD_TH:
        if port_area>stbd_area:
```

```
            print('Dark zone.. moving RIGHT',dark_area)
            robot.right(0.14)
            time.sleep(0.5)
            robot.stop()
        else:
            print('Dark zone.. moving LEFT',dark_area)
            robot.left(0.15)
            time.sleep(0.5)
            robot.stop()
    else:
        print('Dark zone.. moving BACK',dark_area)
        robot.backward(HEADING_SPEED)
        time.sleep(0.5)
        robot.stop()
        robot.left(0.15)
        time.sleep(0.5)
        robot.stop()


#             time.sleep(0.5)
#             robot.stop()
#             mov_circle_b(0.4,delay=1)
#     #elif port_area<PORT_TH or stbd_area<STBD_TH:


#     elif lft==1:                    #elif stbd_area<STBD_TH and lft==1:
#         #print('obstacle port side')
#         robot.right(0.14)
#         time.sleep(0.5)
#         robot.stop()
#         lft=0
#     elif lft==0:#elif port_area<PORT_TH and lft==0:
#         #print('obstacle stbd side')
#         robot.left(0.15)
#         time.sleep(0.5)
#         robot.stop()
#         lft=1

    return lft
```

Appendix G

Box Position Algorithm

```
def box_position(boxes,classes,robot,class_names):
    areas={}
    cen={}
    pos={}
    box_co={}
    h,w =640.0,640.0
    for cl in class_names:
        areas[cl]=0.0
        cen[cl]=0.0
        pos[cl]=''
        box_co[cl]=[0.0,0.0,0.0,0.0]

    for box,cls in zip(boxes, classes):
        x1, y1, x2, y2 = box
        c=(x1+x2)/2.0
        box_area_per=(x2-x1)*(y2-y1)/(h*w)
        areas[class_names[cls]]=box_area_per
        cen[class_names[cls]]=c
        box_co[class_names[cls]]=x1, y1, x2, y2

        if c<(0.3*w):
            box_pos='P'
        elif c>(0.7*w):
            box_pos='S'
        else:
            box_pos='C'
        pos[class_names[cls]]=box_pos

    return areas,cen,pos,box_co

def box_mov(area_per,cen,box_pos):
    global PREV_MOVE
    #x1, y1, x2, y2 = box
    h,w =640.0,640.0
    cmnd=0.0

    if box_pos=='P' and area_per<BOUNDARY_TH:
        cen_t=cen/w
        #print('Area % is:',0.31-cen_t)
        if abs(0.28-cen_t)>0.2:
            cmnd=0.2
            #robot.left(0.25)
        elif 0.28-cen_t>0.18:
            cmnd=0.25-cen_t
            #robot.left(0.28-cen_t)
        else:
            cmnd=0.28-cen_t
        if cmnd<0.15:
            cmnd=0.14
```

```
        robot.left(cmnd)
        time.sleep(0.4)
        robot.stop()
        PREV_MOVE='L'
        print(f'{box_pos} command:{cmnd}')
    elif box_pos=='S' and area_per<BOUNDARY_TH:
        #robot.right(0.3)#
        cen_t=cen/w
        if abs(cen_t-0.78)>0.22:
            cmnd=0.19
            #robot.right(0.25)
        elif cen_t-0.78>0.18:
            cmnd=cen_t-0.78
            #robot.right((cen/w)-0.78)
        else:
            cmnd=cen_t-0.74
        if cmnd<0.15:
            cmnd=0.13
        robot.right(cmnd)
        time.sleep(0.4)
        robot.stop()
        PREV_MOVE='R'
        print(f'{box_pos} command:{cmnd}')

    elif box_pos=='C' and area_per<BOUNDARY_TH:
        robot.set_motors(0.18, 0.18)
        time.sleep(1)
        robot.stop()
        PREV_MOVE='F'
```

Appendix H

Main Function

```
def main(engine_path, image_size,opt,processor,visualizer,camera):
    ##To create camera instance

    det_count=0  #count of consecutive detections
    Object1_bound=0 # Object1 boundary reached flag
    Object2_bound=0 # Object2 boundary reached flag
    Object1_search_count=0
    conf=opt['conf']
    input_size = opt['img_size']#FLAGS.size

    frame_id = 0

    lft=0
    image_num = 100000
    #fps = FPS().start()
    try:
        while True:
            image = camera.value #change['new']
            frame_size = image.shape[:2]

            # Press Q on keyboard to  exit
            if cv2.waitKey(25) & 0xFF == ord('q'):
                camera.stop()

            image_src = image #cv2.imread(image_path)
            num_classes = opt['classes']   # mahesh  from 80
            namesfile = opt['name']   # names file
            class_names = load_class_names(namesfile)

            img = cv2.resize(image_src, (opt['img_size'], opt['img_size']))
            output = processor.detect(img)
            boxes, confs, classes =
processor.post_process(output,origin_w=opt['img_size'],origin_h=opt['img_size'],
conf_thres=conf)

            areas,cen,pos,box_co=box_position(boxes,classes,robot,class_names)
            #print('Center positions:',cen['Object1']/640,cen['Object2']/640)
            #print(areas)
            if areas['Object1']>0 and Object1_bound==0:
                GPIO.output(11,GPIO.HIGH)
                det_count+=1
                yaw_prev=imu_yaw()  #computing the yaw

                if areas['Object1']<BOUNDARY_TH:
                    print('Moving towards Object1...')
                    box_mov(areas['Object1'],cen['Object1'],pos['Object1'])
                else:
                    print('Object1 boundary reached..')
                    Object1_bound=1
```

```
            elif Object1_bound==1:
                if areas['Object2']>0:
                    if areas['Object2']<BOUNDARY_TH_2 and sen_bound==0:
                        box_mov(areas['Object2'],cen['Object2'],pos['Object2'])
                    else:
                        robot.set_motors(0,0)
                        print('Final destination reached..')
                        sen_bound=1
                    GPIO.output(12,GPIO.HIGH)
                else:
                    print('Object1 boundary reached..')
                    sen_bound=1
                    robot.set_motors(0,0)
                    print('Final destination reached..')


            elif det_count>0:
                yaw_present=imu_yaw()
                yaw_diff=yaw_prev-yaw_present
                if abs(yaw_diff)>0.5:
                    if yaw_diff<0:
                        mov_right(0.15,0.5)
                        time.sleep(0.15)
                    else:
                        mov_left(0.15,0.5)
                        time.sleep(0.15)

            else:
                det_count=0 if det_count<=0 else det_count-1
                GPIO.output(11,GPIO.LOW)
                GPIO.output(12,GPIO.LOW)
                if opt['search']=='spiral':
                    spiral_search(frame_id)       #spiral search
                else:
                    if opt['col_avoid']=='True':
                        lft=col_avoid(img,frame_id,lft,THRESH=60,row_start=470)
                    else:
                        lft=st_search(frame_id,lft)  #standard search
            if opt['show_out']=='True':
                retimg=visualizer.draw_results(img, boxes, confs,
classes,class_names,BOUNDARY_TH)
                retimg=cv2.cvtColor(retimg,cv2.COLOR_BGR2RGB)
                display(Image.fromarray(retimg))
                time.sleep(0.5)
                clear_output(True)
            if opt['save']=='True':
                retimg=visualizer.draw_results(img, boxes, confs,
classes,class_names,BOUNDARY_TH)
                cv2.imwrite('det-output/det-YoloImg_'+str(image_num)+'.jpg',retimg)
                image_num = image_num + 1
            frame_id += 1
            #print("Elasped time: {:.2f}".format(fps.elapsed()))
            #print("FPS: {:.2f}".format(fps.fps()))
```

```
        #fps.stop()
    except KeyboardInterrupt:
        GPIO.output(11,GPIO.LOW)
        GPIO.output(12,GPIO.LOW)
        print('keyboard interrupt & stopping the camera')
        robot.stop()
        camera.stop()
        pass
```

Appendix I

Establishing Paths and Starting the Robot

```
opt={'eng':'(Location of ".engine
file")','classes':2,'conf':0.3,'img_size':640,'name':'data/objects.names','save':'T
rue','show_out':'True','search':'True','col_avoid':'True'}
#print(opt.weights)
engine_path = opt['eng']
image_size=((opt['img_size']),(opt['img_size']))
print('Engine loading')
processor = Processor(engine_path,opt['classes'])
print('Engine loaded....')
visualizer = Visualizer()
camera = Camera.instance(width=640, height=640)
main(engine_path, image_size,opt,processor,visualizer,camera) #Starting Main
Function
```