# Development of the JetBot: An Open-Source, AI-Powered Robot

Husam Hamu, Weilin Han, Yiyang Wang, Qi Li and Xi Chen
Code on GitHub: https://github.com/husamhamu/ps_robotik

***Abstract*— The primary objective of this study is to present the development of an AI-driven vehicle, JetBot AI, leveraging the Nvidia Jetson Nano module. The focus is on implementing crucial functionalities, including autonomous obstacle detection utilizing camera inputs, navigation path planning, and vehicle motion control. Through the analysis of environmental data, a safe navigation path is generated to avoid obstacles effectively, and the JetBot vehicle is accurately directed to execute the necessary actions. The research incorporates advanced methodologies such as map creation, obstacle identification, and obstacle avoidance algorithms. The paper elaborates on the methods and procedures employed to facilitate object detection, obstacle avoidance, and path planning for the JetBot vehicle, employing mapping, tracing, and sorting objects [1].**

## I. Introduction

With the rapid advancement of artificial intelligence and machine learning, there is a growing demand for intelligent robotics. The JetBot, a competent and scalable platform, is an ideal experimental platform for researchers, students, and developers to explore and innovate in robotics.

The successful completion of this project will enable students to gain a comprehensive understanding of the fundamental principles underlying robotics, computer vision, and artificial intelligence. It will equip them with the knowledge and skills to comprehend the intricacies of intelligent path planning, object recognition, and autonomous navigation. Additionally, students will become proficient in utilizing the ROS robot operating system framework, empowering them to drive advancements in the development and application of intelligent robotics [2].

The development of JetBot encompasses several vital technologies and areas. Firstly, it is built upon the NVIDIA Jetson platform, which provides powerful embedded computing capabilities for executing complex computational tasks and deep learning inference. The JetBot leverages camera-based object recognition to perceive and understand its environment. Moreover, it enables users to quickly develop and execute custom robot control and deep learning algorithms by utilizing the ROS robot operating system and a comprehensive suite of software libraries and development tools [3].

This paper will delve into the core technologies and methodologies employed in developing JetBot's path planning, object recognition, and autonomous navigation capabilities. Specifically, we will elaborate on integrating sensor data, maps, and algorithms to realize effective path planning, as well as utilizing deep learning and computer vision techniques for accurate and robust object recognition. Data processing and integration techniques will be employed to achieve real-time system responsiveness and precision. A visual interface will present crucial information, including the JetBot's status, environment maps, recognition results, path planning, and execution trajectories, facilitating real-time monitoring and analysis of the JetBot's operation and performance. Rigorous testing and optimization will be conducted to ensure optimal results in task completion.

## II. JetBot Development

### A. *Deep learning based approach for object recognition*

In this project, we employ deep learning-based object detection to classify and detect experimental blocks and spheres. Over the years, significant progress has been made in the domain of object detection, with the mainstream algorithms broadly falling into two categories: The first category embodies two-stage methods, such as the R-CNN series of algorithms. The central concept behind these algorithms is using heuristic techniques like selective search or CNN networks such as Region Proposal Network (RPN) to generate a sparse set of candidate boxes, which are subsequently subjected to classification and regression. The primary advantage of two-stage methods is their high precision.

The second category comprises single-stage methods, exemplified by YOLO (You Only Look Once) and SSD (Single Shot MultiBox Detector). The basic idea here is uniformly sampling dense locations across the image, incorporating various scales and aspect ratios. Features are then extracted using CNN and directly used for classification and regression in a single step. The critical advantage of single-stage methods lies in their rapid processing speed. However, due to the severe imbalance between positive samples and the background (negative samples), uniform, dense sampling presents challenges in training, resulting in slightly reduced model accuracy.

A partial comparison of the advantages and disadvantages of the YOLO and SDD methods is shown in Table 1. Our primary focus in this study is to explore the single-stage methods for object detection, specifically the SSD and YOLO approaches.

#### 1) *SSD (Single Shot MultiBox Detector)*

Our SSD method consists of the following steps:

- Setting up the environment with PyTorch, TensorRT, and other dependencies (referring to the tutorial [4]).
- Setting up the label file and categorizing the classes into 7 categories: ball, bluesquare, greensquare, redsquare, yellowsquare, orangesquare, and purplesquare.

- Placing the Jetson Nano device inside the experimental area, activating the monocular camera, and using the camera-capture tool to capture and classify the data.
- Training the collected data.
- Obtaining the trained model and converting the file format by converting the PyTorch model to ONNX.
- Loading the converted model and performing testing.

During the model training process, we encountered frequent lag issues due to the limited memory capacity of the Jetson Nano. To address this problem, we employed Swap to establish an additional 8GB of virtual memory, effectively resolving the performance issue [4]. With the implementation of this solution, the Jetson Nano was able to handle memory-intensive training tasks more efficiently, alleviating the occurrence of lag and improving the overall performance.

Based on the steps mentioned above, we carried out multiple rounds of sampling and model training. Initially, we collected 300 training images and trained the model. However, the detection results were unsatisfactory, as shown in Figure 2. Upon observation, the model failed to calibrate when the detection distance increased to 20 centimeters. Consequently, we expanded the dataset by adding 300 training images, 100 testing images, and 100 validation images. Although the detection distance increased, the improvement in detection performance was not significant. Therefore, we decided to employ the YOLO algorithm for object detection.

For the improvement of the SSD method, we can enhance its performance through various preprocessing techniques using data augmentation. Some key techniques include:

- Horizontal Flip: Randomly flipping the image horizontally to increase the diversity of training data, thereby improving the model's robustness.
- Random Crop & Color Distortion: Randomly cropping a portion of the image and applying color distortions such as brightness, contrast, and saturation changes. This helps to increase the diversity of samples and improve the model's ability to detect objects of different scales and viewpoints.
- Randomly Sample a Patch: Randomly selecting a small patch area within the image as a training sample, particularly beneficial for training on small objects. This helps the model better learn the features and contextual information of small objects. By applying these data augmentation techniques, we can increase the diversity of the training data, alleviate the risk of overfitting, and improve the model's generalization ability and detection accuracy. These anticipated improvements are expected to yield better performance in the SSD method.

### 2) *YOLO(You Only Look Once)*

*Yolov7*

The You Only Look Once (YOLO) object detection algorithm performs both object localization and classification in a single step. It regresses the bounding box coordinates and predicts the class label for each bounding box in a unified output layer. Initially, we chose YOLOv7 as the method for object detection, and the main implementation steps are as follows:

TABLE I
YOLO VS SSD

| | YOLO [5] | SSD [6] |
|---|---|---|
| Processing Speed | Expected to be very fast, allowing for real-time object detection | Fast but usually slower than YOLO |
| Accuracy | Great improvements in the new version | Traditionally good, especially with small objects |
| Ease of Use and Flexibility | Generally easier to use and quicker to train | More flexible in detecting objects of various shapes and sizes |
| Small Object Detection | Slightly less accurate than SSD | Very small objects can be detected |

- Network Architecture: Designing the YOLOv7 network architecture, which typically consists of convolutional layers, downsampling layers, and the final detection layer , set the environment [7].
- Data Preprocessing: Preprocessing the training dataset by resizing the images to a specific input size 640*480, normalizing the pixel values, and converting the ground truth bounding box annotations into the required format. For the data collection, we took 853 photos with the monocular camera of the jetson nano and subsequently processed and calibrated these photos on a private computer using the software roboflow [8].
- Anchor Box Clustering: Clustering the ground truth bounding box sizes to determine a set of anchor boxes that represent different scales and aspect ratios of objects in the dataset.
- Training: Training the YOLOv7 model using the annotated dataset. This involves optimizing the model's parameters through techniques like gradient descent and backpropagation, with the objective of minimizing the detection loss. In this step, we also train in an ubuntu environment built with docker on a personal computer, which also greatly reduces the training time compared to training on jetson nano.
- We implemented YOLOv7 (You Only Look Once version 7) on a Jetson Nano platform. In order to facilitate the deployment, a specific virtual environment named 'Yolov7' was created. The process of environment configuration and creation of the virtual environment was guided by the comprehensive tutorial [9].
- Inference: Using the trained model to perform object detection on new images. This involves passing the input image through the network, predicting the bounding box coordinates and class probabilities, and applying post-processing techniques such as non-maximum suppression (NMS) to filter out overlapping detections.

*Yolov3*

When deploying YOLOv7 on the Jetson Nano for image detection, we observed that the detection time for each image is approximately 2 seconds (as shown in Figure 1), which does not meet the requirements for real-time detection. We suspect that the issue may be due to the limited computational capabilities of the Jetson Nano. Therefore, we have decided to try using the computationally lighter YOLOv3. By transitioning to YOLOv3, we expect to achieve faster detection speeds on the Jetson Nano while meeting the real-time detection requirements. YOLOv3 offers a lower computational complexity compared to YOLOv7, which may help improve the performance of the Jetson Nano.

The implementation steps of yolov3 are similar to those of yolov7 (referring to tutorial [10]).

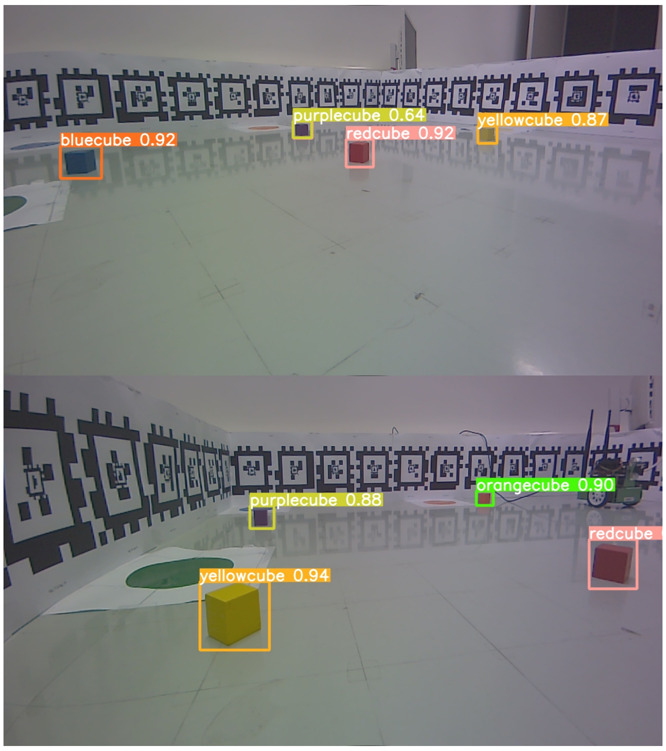


Fig. 1. Recognition of YOLOv3 objects

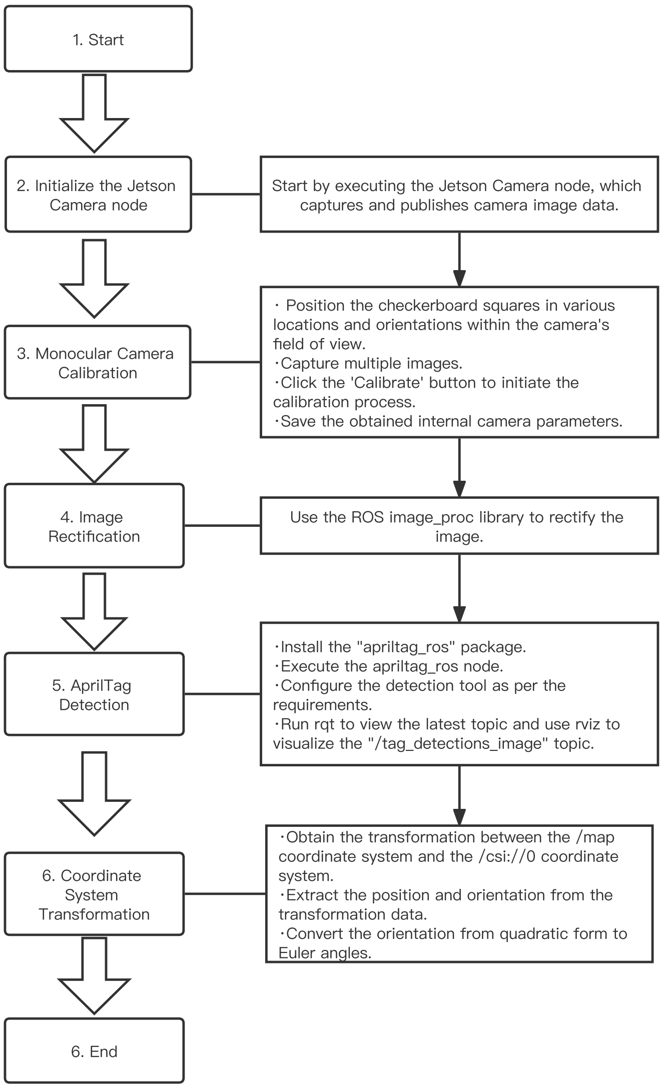

1. Start
2. Initialize the Jetson Camera node
Start by executing the Jetson Camera node, which captures and publishes camera image data.
3. Monocular Camera Calibration
· Position the checkerboard squares in various locations and orientations within the camera's field of view.
·Capture multiple images.
·Click the 'Calibrate' button to initiate the calibration process.
·Save the obtained internal camera parameters.
4. Image Rectification
Use the ROS image_proc library to rectify the image.
5. AprilTag Detection
·Install the "apriltag_ros" package.
·Execute the apriltag_ros node.
·Configure the detection tool as per the requirements.
·Run rqt to view the latest topic and use rviz to visualize the "/tag_detections_image" topic.
6. Coordinate System Transformation
·Obtain the transformation between the /map coordinate system and the /csi://0 coordinate system.
·Extract the position and orientation from the transformation data.
·Convert the orientation from quadratic form to Euler angles.
6. End

Fig. 2. Build Apriltag flow chart

## B. Build Apriltag

Before calibrating the camera and constructing the AprilTag, it is necessary to execute the Jetson Camera node, which initiates the camera's image capture and publishing functions. By utilizing the image_view package, the Jetson Camera node publishes the image data obtained from the camera, enabling real-time visualization of the camera's video feed through a designated window [11]. The process for building an Apriltag is shown in Figure 2.

### 1) Camera calibration

In the first step of the project, the monocular camera needs to be calibrated. Camera calibration for obtaining internal camera parameters.

To begin, the camera_calibration package should be installed. The calibration target is set as the checkerboard grid. The image theme chosen for calibration is image_raw. It is essential to position the checkerboard squares at various locations and orientations within the camera's field of view. This includes tilting the board to the left, right, up, and down to fill the entire field of view and acquire multiple images [12].

Once the images have been acquired, the 'Calibrate' button is clicked to initiate the calibration process. Upon completion, the results are displayed, and the calibrated image should appear rectified. The internal parameters obtained from the calibration can be saved in a YAML file for future reference and usage [13].The calibration results are shown in Table II.

Furthermore, the ROS image_proc library can be utilized to rectify the image image_raw. This process involves demosaicking and de-warping the original camera image and applying color correction techniques.

### 2) Apriltag

AprilTag is a 2D code system valuable tool for machine vision and target tracking. It enables rapid and accurate detection and localization of markers within camera images, providing essential position and poses information relative to the camera. Within the framework of ROS (Robot Operating System), AprilTag can be applied to various robotic applications, including navigation, SLAM (Simultaneous Localization and Mapping), target tracking, and more. In this project, the

TABLE II
CAMERA CALIBRATION RESULTS

```
image_width: 1280
image_height: 720
camera_name: narrow_stereo
camera_matrix:
 rows: 3
 cols: 3
        [790.759221        0.         690.730757
 data:        0.       790.416682   326.233959
              0.           0.           1.          ]
distortion_model: plumb_bob distortion_coefficients:
 rows: 1
 cols: 5
 data:  [−0.330602   0.101843   −0.000086   −0.000752   0.000000]
rectification_matrix:
 rows: 3
 cols: 3
        [1.   0.   0.
 data:   0.   1.   0.
         0.   0.   1.]
projection_matrix:
 rows: 3
 cols: 4
        [599.871155        0.         708.99075    0.
 data:        0.       728.773682   320.50153    0.
              0.           0.           1.        0.]
```

path planning methodology involves utilizing AprilTag.

To facilitate the integration of AprilTag with ROS, the "apriltag_ros" package is employed, which serves as a ROS wrapper for the AprilTag 3 visual fiducial detection algorithm. Before compiling AprilTag, it is necessary to clone and install the "apriltag_ros" package into the designated catkin workspace. Once installed, the apriltag_ros node can be executed to initiate AprilTag's detection and tracking capabilities [11].

To view the latest topic, you can refer to the topic list. You can run rqt, a graphical tool in ROS, to accomplish this. Within rqt, you can open rviz and visualize the "/tag_detections_image" topic. Additionally, you can display the pose of the detected tags and tag packets in rviz.

By executing the steps mentioned above, you can observe the most recent topic data, view the corresponding image on the "/tag_detections_image" topic, and visualize the pose information associated with the detected tags and tag packets in rviz.

To detect AprilTag tags in the image or camera data and save the location information of the detected tags, you can utilize the detection tool provided by the AprilTag package. You can use the configuration files in the "apriltag_ros/config" directory for parameter configuration [14].

By adjusting the settings in the "settings.yaml" file and specifying the tags to be detected in the "tags.yaml" file, you can configure the detection tool according to your requirements. The "settings.yaml" file allows you to define parameters such as the camera calibration information, detection thresholds, and other settings related to tag detection [15]. The "tags.yaml" file allows you to specify the IDs and sizes of the AprilTag tags that you want to detect.

#### 3) *Robot & World Coordinate System*

The position of the robot in the world is obtained via AprilTag, along with the global coordinate system (or world coordinate system), which describes the robot's position and orientation in the entire environment, and the robot coordinate system (or local or body coordinate system), which describes the position and orientation of objects in the robot's perspective [16].

When a robot needs to reach a target location or interact with an object in its environment, it needs to know the position and orientation of the target or object relative to itself. Therefore a transformation from the global coordinate system (describing the target position) to the robot's coordinate system is required [17]. In addition, the robot also needs to know its position and orientation in the global coordinate system in order to perform path planning and navigation.

The code implementation aims to transform position and orientation data between different coordinate systems using the tf library in ROS, using the loop method to obtain the transformation between the /map coordinate system (global coordinates) and the /csi://0 coordinate system (robot coordinates). The transformation data includes translations (trans) and rotations (rot). Next, the position (x, y, z) and direction are extracted from the transformation data. The orientation is converted from quadratic form to Euler angles (roll, pitch, yaw) [18]. This gives the robot pose (position and orientation).

The target point is initially set, and then the PID controller is invoked to calculate the left and right velocities of the wheels based on the current robot pose and target position.

### C. *Obstacle Location*

The localization of obstacles is a crucial component of this project. Accurate measurement of obstacle distances is essential for enabling successful path planning.

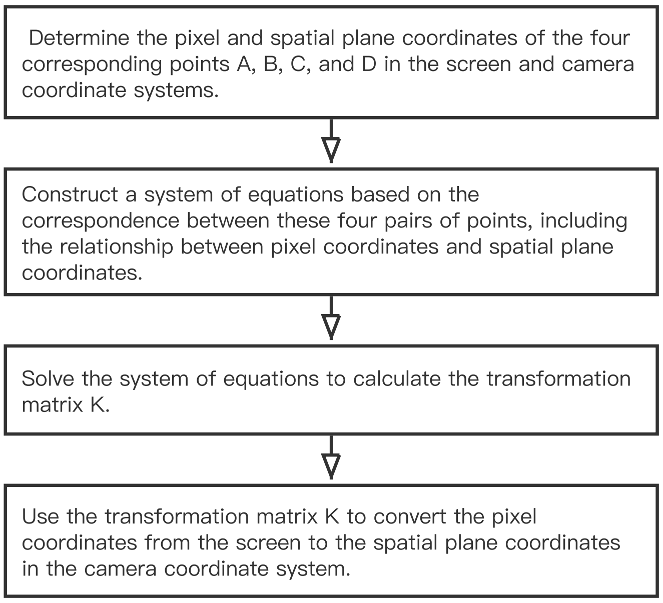
#### 1) *Coordinate transformation*

When performing the coordinate transformation, common approaches include using similar triangles and the PnP (Perspective-n-Point) method. However, these methods require obtaining various information about the cube, such as pixel coordinates, pixel lengths, and multiple corner points. These are used to reduce errors through edge detection.

In contrast, we only need to obtain the pixel coordinates of the cube's midpoint when using a perspective transformation. For close distances, the detection results have minimal error (around 0.6 centimeters). For longer distances, the error is approximately 3-4 centimeters, which is sufficient for the close-range detection requirements of the task [19].

Perspective transformation simplifies the process by relying on the pixel coordinates of the cube's midpoint, making it suitable for accurate detection at close distances. While there may be some errors at longer distances, it still meets the needs of the task.

After obtaining the pixel coordinates of the cube on the screen, you can use the perspective transformation method to convert them into spatial plane coordinates in the camera coordinate system. Perspective transformation is achieved by solving a coefficient matrix, which can be computed using known coordinates in both coordinate systems. The specific coordinate conversion process is shown in the Figure 3.



Determine the pixel and spatial plane coordinates of the four corresponding points A, B, C, and D in the screen and camera coordinate systems.

Construct a system of equations based on the correspondence between these four pairs of points, including the relationship between pixel coordinates and spatial plane coordinates.

Solve the system of equations to calculate the transformation matrix K.

Use the transformation matrix K to convert the pixel coordinates from the screen to the spatial plane coordinates in the camera coordinate system.

Fig. 3. Perspective transformation flow chart

Figure 4 illustrates the correspondence, where A, B, C, and D represent the four known points in both coordinate systems, and the transformation matrix K can be calculated from these points.

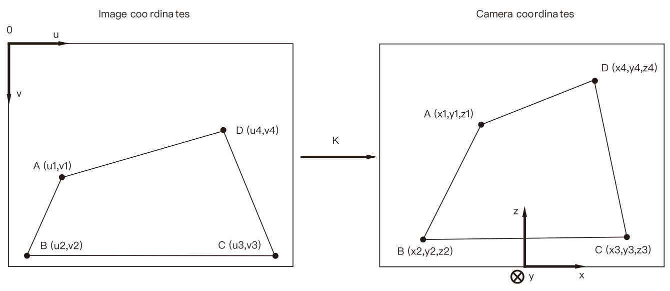


Fig. 4. Perspective transformation

The basic principle is to solve the transformation relationship matrix between the pixel coordinates in the image and the camera coordinate system. As shown in the following equation:

$$K = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{2}$$

where $u$, $v$ represent the pixel coordinates where the object blocks in the image are located. $x$, $y$, $z$ represent are the new coordinates in the camera coordinate system after the transformation. Here, the transformation matrix is of a 3*3 form. We can obtain the system of equations:

$$\begin{aligned} x &= a_{11}u + a_{12}v + a_{13} \\ y &= a_{21}u + a_{22}v + a_{23} \\ z &= a_{31}u + a_{32}v + a_{33} \end{aligned} \tag{3}$$

Next, we locate the midpoints of the four blocks in the camera coordinate system and use YOLOv3 to identify them, obtaining their corresponding pixel coordinates in the image. By substituting these coordinates into the equations mentioned above, we can solve for all the unknowns in the coefficient matrix.

Once the transformation matrix is obtained, we can get the position of the pixel in the camera coordinate system by entering its coordinates.

Figure 5 depicts the actual coordinates of obstacles. Upon measurement, it was found that the estimated position coordinates, as shown in the Figure, deviate slightly from the real coordinates. Moreover, this deviation progressively increases with distance. The maximum discrepancy obtained from the comparison varies based on how accurate is the bounding box estimated by the neural network model, but it is approximately 8 centimetres.

#### 2) *Update obstacle locations*

All obstacles are detected through YOLO (You Only Look Once), and their locations are updated from the perspective of the camera. Taking the variance of the object location into account, when obtaining a new measurement, we match a new measurement with an old measurement, if the distance is less than 0.2 meter. The average of the matched points is then taken and used as the new object location. Unmatched new measurements are considered to be new obstacles in the arena.

Visualization of the positional changes of the obstacles is done, including previously identified obstacles, newly discovered obstacles, and obstacles whose positions have been updated. These obstacles are plotted as scatter points in a two-dimensional space, with different colors distinguishing them. The labels of the updated obstacles are then displayed near the corresponding points(as shown in Figure 6).

### D. *Path Planning*

The JetBot mobile robot vehicle is tasked with planning a path from an initial position to a target position within a working field. This task requires the robot to navigate around static and dynamic obstacles. Initially, the team considered implementing Voronoi diagram algorithms for this purpose. However, after careful analysis and evaluation, this approach was not adopted and the Rapid Exploration Random Tree (RRT) algorithm was chosen instead.The path planning for Task 2 is implemented through this section, as shown in the Figure7.

#### 1) *Voronoi diagram*

The Voronoi diagram algorithm is a classical path planning method that ensures that a robot is as far away from an obstacle as possible by guiding it along a median between obstacles.

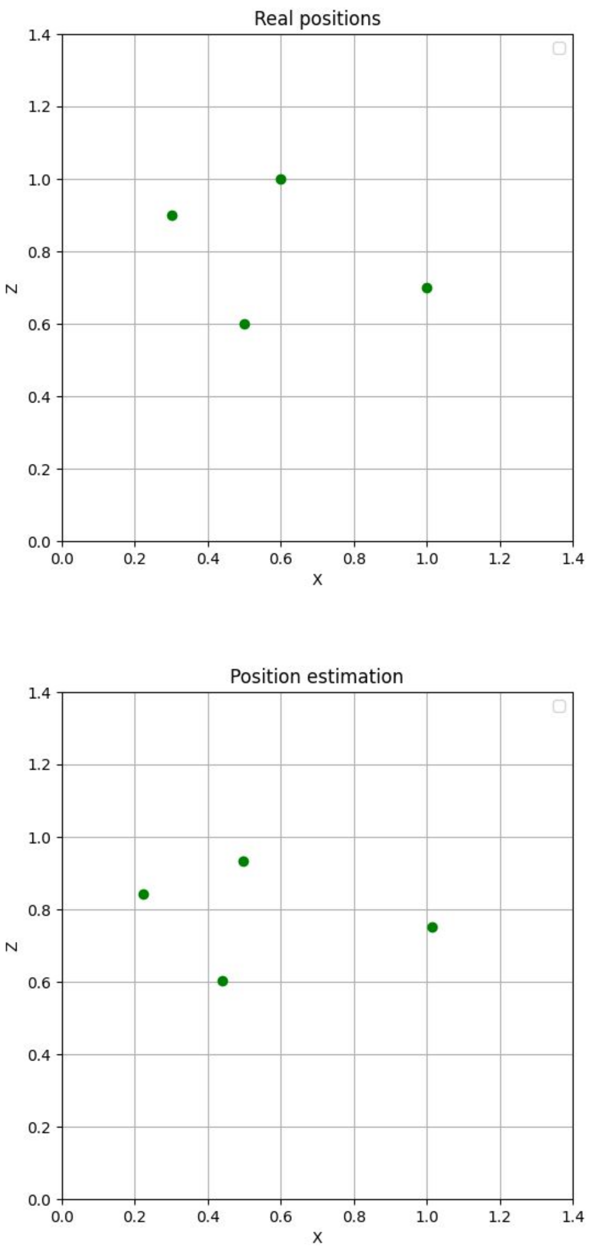


Fig. 5. Real Position & Position Estimation

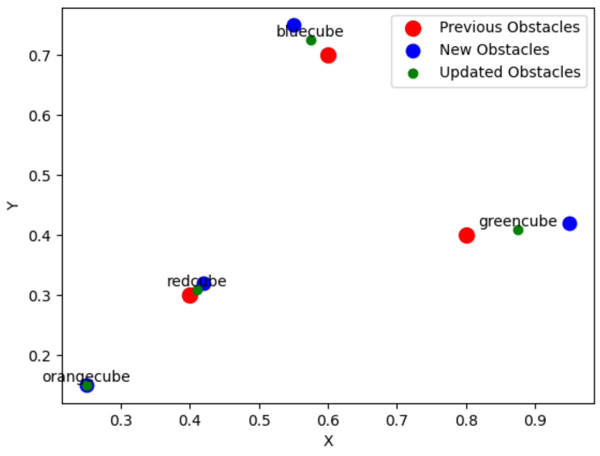


Fig. 6. Update the obstacle positions

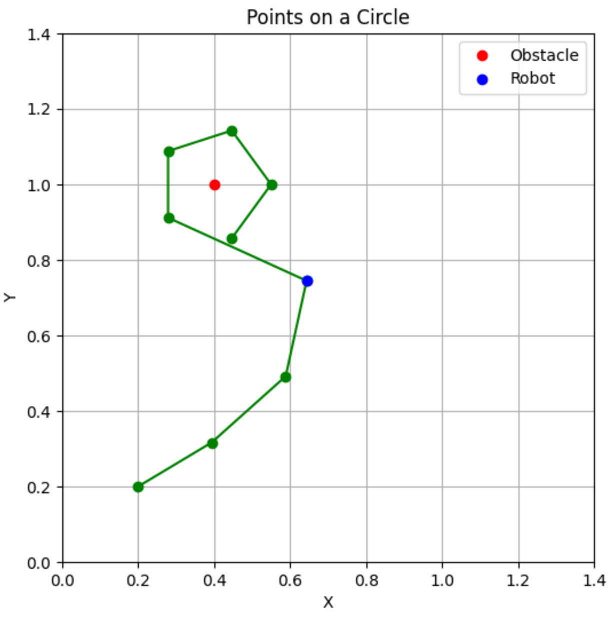


Fig. 7. Path planning

The specific implementation method includes the following steps [20]:

- First step: For each point in free space, calculate its distance to the nearest obstacle.
- Second step: Express the distance of each point to the nearest obstacle as a height on an axis perpendicular to the two-dimensional space, similar to creating a histogram.
- Third step: When a point is at the same distance from two or more obstacles, a peak appears at that distance point. By connecting these peak points, a Voronoi diagram is constructed.

### 2) *RRT*

However, its applicability in dynamic environments is limited. The Voronoi diagram algorithm requires an available map of the environment, and the computational cost of recalculating the diagram in response to environmental changes can be high. The team members did not find a suitable method for controlling the cart's movement path due to the distance to the obstacle, so an alternative algorithm was chosen.The specific implementation flow of RRT path planning is shown in the Figure 8.

In contrast, the Rapid Exploration Random Tree (RRT) algorithm was chosen to explore the environment by progressively building a path tree, expanding randomly to unexplored areas. The RRT algorithm provides a robust solution for the

JetBot that can adapt to changing obstacles while aiming at the target and controlling the cart's advance.

The Rapid Exploration Random Tree (RRT) algorithm is a sampling-based path planning method that creates efficient navigation maps and identifies feasible paths. Its primary modus operandi consists of a forward search towards a given target point, achieved by generating random points in each iteration. This technique allows the algorithm to efficiently bypass obstacles, prevent trajectories from falling into local minima, and ensure rapid convergence [21].

We chose to implement the RRT path planning algorithm using a Python script, where the main class is designed to store and manipulate key RRT information. This information includes various parameters, including the initial position, the target position, the number of obstacles, and the size of the environment. The implementation uses the extend_tree(self)

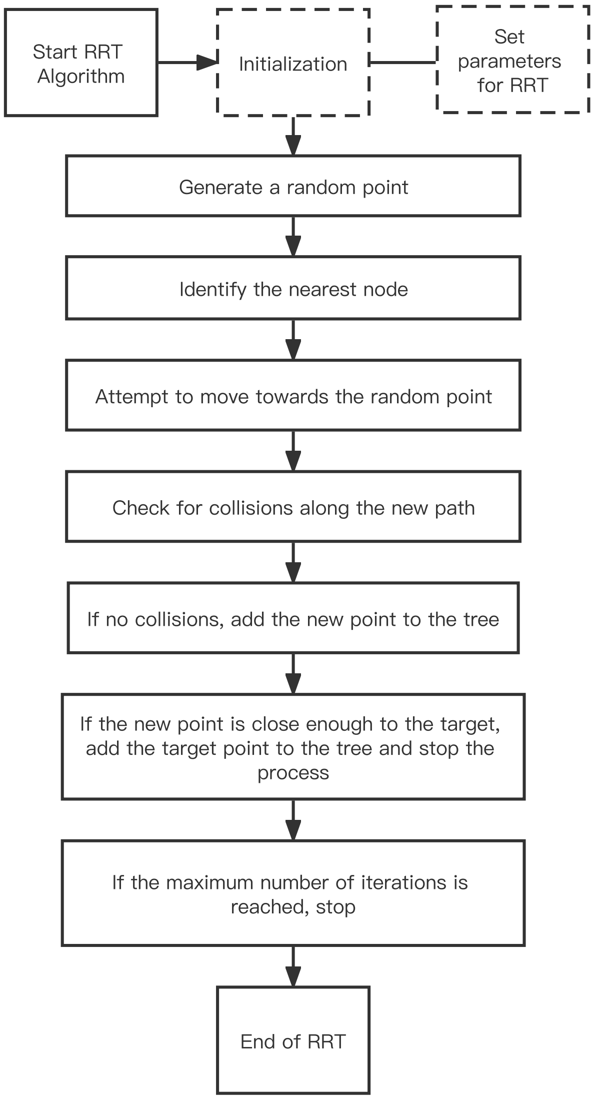

Start RRT Algorithm
Initialization
Set parameters for RRT
Generate a random point
Identify the nearest node
Attempt to move towards the random point
Check for collisions along the new path
If no collisions, add the new point to the tree
If the new point is close enough to the target, add the target point to the tree and stop the process
If the maximum number of iterations is reached, stop
End of RRT

Fig. 8. Path planning RRT flow chart

method, generating a random point and then identifying the nearest node. It then attempts to move towards the identified point and, if no collisions occur, adds the new point to the tree. If this new point is close enough to the target, the method adds the target point to the tree and stops the process. In addition, the process also terminates if the maximum number of iterations allowed is reached.

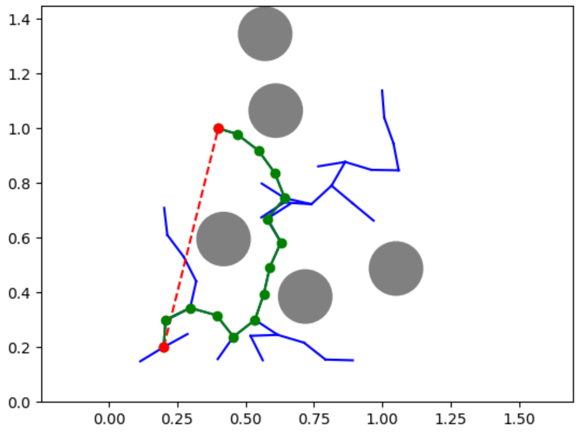


Fig. 9. Path planning RRT process

### 3) RRT optimization

Although the Rapid Exploration Random Tree (RRT) algorithm is relatively efficient and robust for path planning problems with incomplete constraints, it possesses certain limitations which, although not addressed in this project, can be mitigated by optimization to RRT* (RRT-Star). A key drawback of RRT is that it does not guarantee the optimality of the paths found. Specifically, RRT can only identify a feasible path but cannot guarantee that this path is the shortest or most efficient.

When RRT* incorporates a new node into the tree, it does not merely identify the nearest node to the new node. Instead, it also searches for all nodes within a specified radius and tries to optimize the path by interconnecting these nodes. Thus, as more nodes are included, the paths identified become closer to the optimal solution [22].

In practice, it was found that the RRT algorithm takes a significant amount of time to generate the paths, and the RRT* algorithm computes paths at a cost significantly lower than that of RRT. This efficiency can be attributed to two critical processes included in the RRT* algorithm: reselecting the parent node and rewiring. These two processes are mutually reinforcing, with parent node reselection aiming to minimize the path cost of the newly generated nodes and random tree rewiring aiming to generate new nodes with fewer redundant paths, thereby reducing the path cost.

## E. JetBot Control

An essential aspect of guiding the Jetbot along a predetermined route is the management of the motor control, which directs the movement of the Jetbot. The cart's movement is mainly powered by two motors located at its base, which drive the left and right wheels of the robot. By manipulating the state of these motors, we control the movement of the trolley.

The initial phase consisted of verifying the effective operation of the motors by following the tutorial and implementing the official NVIDIA drive motor script. We specified the direction of the camera as forward and the direction of the OLED screen as backward. In this case, we call the form statements related to the carriage, such as " forward()," " backward()," "left()," “right()” and "stop()" [11]. Alternatively, we can have direct control of the motor for the same purpose. The Adafruit Motor HAT can control both DC and stepper motors.

The speeds of the two motors are configured with speed values in the range [0 0.2]Each motor’s pulse width modulation (PWM) value is set to control its speed. This function initially adjusts the input speed value and converts it into a PWM value, which subsequently determines the speed of the motor. Next, the function specifies the direction in which the motor is to run. The motor advances forward if the input speed value is less than zero. If it exceeds zero, the motor goes backward. Convert the input angle to an angle starting on the negative y-axis with a range of [-180, 180] degrees. Then determine whether the robot should turn to the left or right to align its direction with the target direction. If the difference between the target direction minus the current direction is greater than 0, then it should turn left. Otherwise, it should turn right. In the code, we implement the logic of a Proportional Controller, which calculates a ’control signal’ to minimize specific errors, proportional to the error, used for fast response systems. P controller is one part of PID controller. We have only used the Proportional controller in this project.

Proportional (P), Integral (I), and Derivative (D) components are combined to form the output of the PID controller, which is used to set the speed of the robot’s left and right wheels. During each control cycle, new control inputs are calculated based on the current error (i.e., the difference between the robot’s current and target positions) and past errors, allowing the robot to gradually approach the target position while continuously adjusting its driving speed [23]. Receives the current position and direction of the robot and the target’s position and then calculates the input values for the PID controller. First, it calculates the difference in angle between the current direction and the vector pointing to the target. It then uses the P controller to calculate a control signal based on the angular difference. Next, it calculates the distance to the target and the desired speed of the robot based on the distance. Finally, it calculates the speed of the robot’s left and right wheels based on the control signal and the desired speed, causing the robot to turn in the direction of the target.

Initially, the function calculates the angle and distance of the robot to the target and then uses a P controller to calculate the control signal. Subsequently, the desired speed of the robot is calculated based on the distance to the target. If the robot is already close to the target (distance less than 4.0), the robot will stop.

Subsequently, the speed of the left and right wheels is calculated based on the target’s relative position to the robot’s current direction. The function then limits the speed to a reasonable range. Finally, the function returns the calculated velocities of the left and right wheels. If the speed of both the left and right wheels of the robot is 0, the robot has reached the target point, and the function returns the robot’s current position.

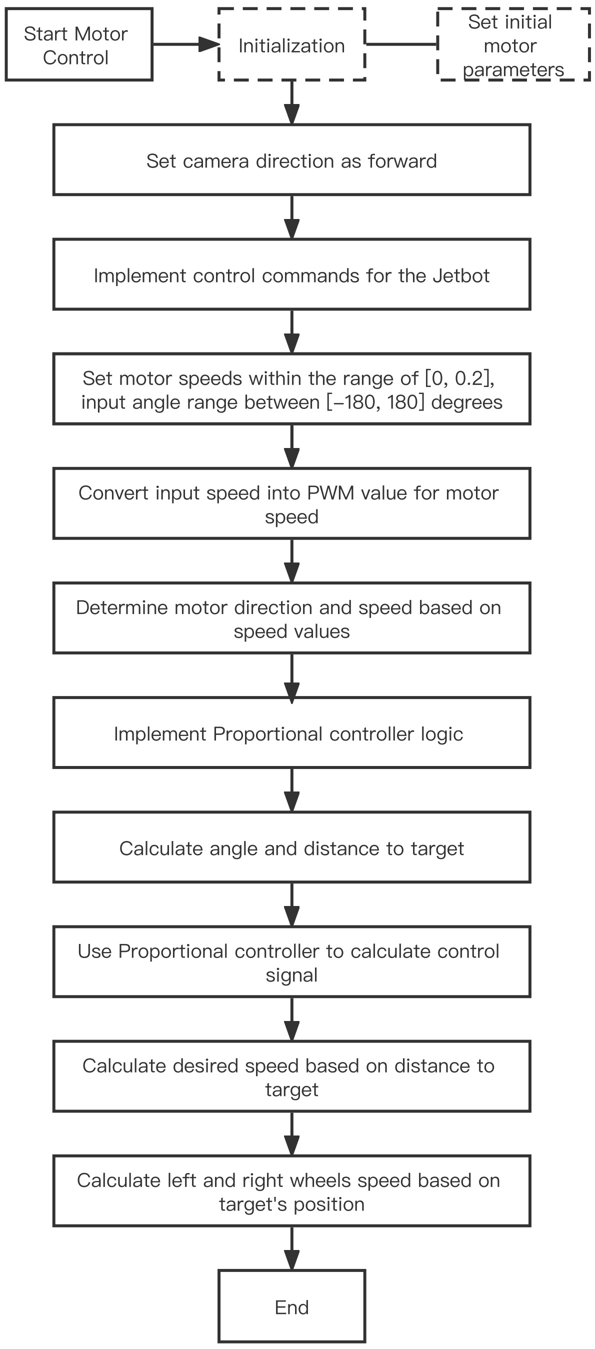

Start Motor Control
Initialization
Set initial motor parameters
Set camera direction as forward
Implement control commands for the Jetbot
Set motor speeds within the range of [0, 0.2], input angle range between [–180, 180] degrees
Convert input speed into PWM value for motor speed
Determine motor direction and speed based on speed values
Implement Proportional controller logic
Calculate angle and distance to target
Use Proportional controller to calculate control signal
Calculate desired speed based on distance to target
Calculate left and right wheels speed based on target's position
End

Fig. 10. Motor drives flow chart

## III. Task realization process

### A. Task 1

#### 1) Requirements for implementation

- To achieve the first task, Jetbot should be initialised at the "Start-Goal" base before starting the task, including calibrating the camera (as Section II-B.1 mentioned), starting the system, etc.
- Jetbot should use its front camera and deep learning algorithms to identify objects in the environment. Object

recognition can be done using the pre-trained convolutional neural network (CNN) model YOLO (as II-A mentioned). The color of the object can be obtained using image processing techniques.
- Jetbot should be able to obtain the object's position by perspective transformation and the object's position in the world coordinate system by transforming the matrix. The coordinates of the object position should be based on the arena coordinate system.
- With object recognition and Apriltag (as Section II-B mentioned) positioning, Jetbot can generate a two-dimensional map of the environment. On the map, the position and color of each object should be marked.
- Jetbot shall display the generated 2D map and list of object center coordinates via a laptop or monitor.

*2) Implementation methods*

By using the RRT (Rapidly-exploring Random Tree) algorithm, Jetbot will plan the following paths sequentially for the given points A and B as final points:

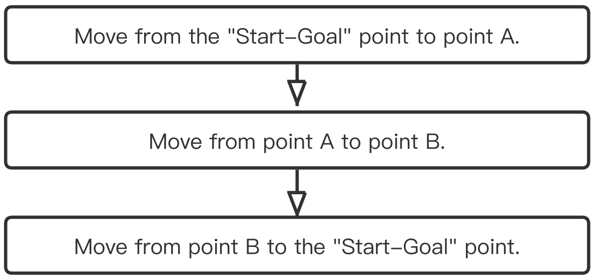

Move from the "Start–Goal" point to point A.
Move from point A to point B.
Move from point B to the "Start–Goal" point.

Fig. 11. Points in order

At the starting point, we will take a photo and detect the coordinates of the blocks. Then, during the subsequent movements, as we generate the path, we will obtain the nodes in the path (such as nodes a, b, c, d, and e in the figure for movement from A to B). Each time Jetbot reaches a node, it will capture a photo using YOLOv3 and perform a coordinate transformation to obtain the world coordinates of the blocks. Using this approach, we will have a collection of images when Jetbot returns to the "Start-Goal" point. Next, we will reduce the error by averaging the coordinates of the same blocks in each image. Finally, we will output the colors and position coordinates of the blocks. Through this process, Jetbot can accurately determine the colors and a fair estimate of positions of the blocks based on the captured images and the coordinate transformations.

## B. Task 2

*1) Requirements for implementation*
- The basic functionality modules required in Task 1 should be implemented.
- Input of the known positions of three red and three blue blocks should be obtained from the console.
- The motion algorithm should be implemented to rotate clockwise around the red blocks and counterclockwise around the blue blocks.

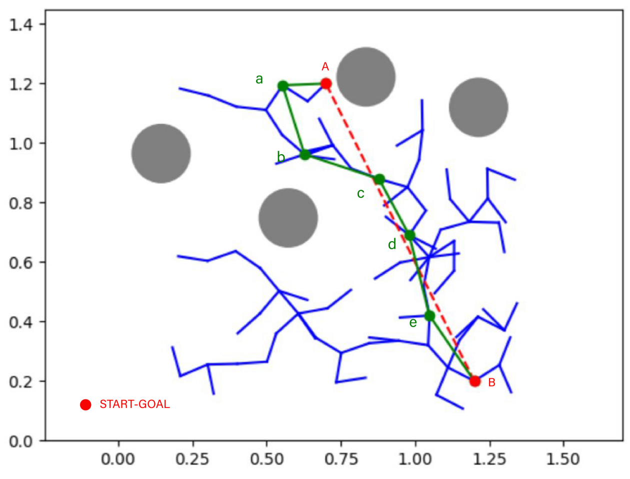


Fig. 12. Path planning: Movement from point A to point B

- Calculate a trajectory that avoids the wooden ball based on the input cube positions and object detection. The red cubes should rotate clockwise, while the blue cubes should rotate counterclockwise. The RRT (Rapidly-exploring Random Tree) path planning algorithm can achieve this goal (as Section II-D mentioned).
- After completing the task, JetBot should return to the "Start-Goal" base after bypassing all the cubes.

*2) Implementation methods*

The implementation approach involves using the RRT algorithm similar to Task 1. Each of the three red and three blue blocks is named red1, red2, red3, blue1, blue2, and blue3, respectively. These blocks are set as final points in the RRT algorithm in sequential order, allowing JetBot to reach each block's position in a sequence. Then, the algorithm logic is implemented to rotate clockwise around the red blocks and counterclockwise around the blue blocks. Finally, when the rotation around blue3 is completed, the "Start-Goal" point is set as the final point. JetBot will continue generating a path using the RRT algorithm to return to the "Start-Goal" point.

Through this implementation, JetBot can sequentially reach the positions of the red and blue blocks and perform the required rotations. After bypassing all the blocks, JetBot will return to the "Start-Goal" base.

## C. Task 3

*1) Requirements for implementation*

The same basic functional modules are required as in Task 1.

*2) Implementation methods*
- JetBot's camera and object recognition algorithm YOLO are used to identify and locate coloured dice and their colour-corresponding bases in the arena. The dice and pedestals can be identified using colour recognition or deep learning methods.
- The nearest colored dice are then found, and the distance between the robot's current position and each dice is

calculated using an algorithm such as the nearest neighbor algorithm. The closest colored dice are selected as the next target.

- Using the robot's motion control method, such as differential drive or omnidirectional wheel drive, the robot is moved to the position of the target dice. Push the dice into the corresponding bases using the pusher provided with the project.
- After all, the dice have been processed, return the robot to the starting base. The path planning algorithm RRT can calculate the shortest path from the robot's current position to the starting base.

The three tasks mentioned above entail using color recognition, the deep learning object detection algorithm YOLOv3 (Section II-A), the path planning algorithm RRT (Section II-D), and the control theory PID controller (Section II-E). Furthermore, obstacle avoidance techniques are employed through object recognition and distance estimation (Section II-B, Section II-C). The successful implementation of these tasks necessitates extensive programming efforts, encompassing Python and the ROS operating system. Moreover, hardware configuration and troubleshooting are also integral components of the process.

## IV. Conclusion

This project has provided us with a valuable understanding of AI and allowed us to learn about robot control through the development of Jetbot. We learned how to program the robot's motion, camera, and sensor data acquisition and processing. We also studied autonomous navigation and path planning algorithms during the development process.

The execution of the first task involved learning computer vision techniques. We acquired knowledge on capturing image data using the camera and applying image processing and computer vision algorithms for object detection, recognition, and localization.

Furthermore, we delved into the application of machine learning and deep learning. By utilizing machine learning and deep learning models, we enabled Jetbot to perform tasks such as object classification, behavior recognition, and object tracking.

This project also deepened our understanding and application of ROS (Robot Operating System). It will be instrumental in future module development for robots. We also gained insights into hardware platform development and deployment, explicitly learning and applying embedded systems.

Throughout the Jetbot development process, we encountered various technical challenges and problems. For example, we made several attempts at object ranging. By solving these issues, we cultivated problem-solving and innovative thinking abilities while enhancing our analytical, design, and debugging skills. Reading and studying other contributors' code and projects on GitHub also gave us a wealth of programming knowledge.

The following milestones have been successfully implemented and tested within the scope of this project:

- Detection of objects in the form of cubes and spheres using a neural network model.
- Localization of these objects with the help of Perspective Transform method.
- Path planning for the robot trajectory and obstacle avoidance using RRT algorithm and a Proportional controller logic.

To improve on these results we obtained, using a stereo camera should be considered, as object localization was a real challenge with a monocular camera.

## References

[1] Waveshare, "Jetbot ai kit - waveshare wiki," 2022. [Online]. Available: https://www.waveshare.com/wiki/JetBot_AI_Kit

[2] CSDN, "Jetbot learning," 2022. [Online]. Available: https://blog.csdn.net/qq_45019121/article/details/124270540

[3] Belochka, "Controlling the jetbot robot from ros," 2021. [Online]. Available: https://www.hackster.io/belochka/controlling-the-jetbot-robot-from-ros-014620

[4] D. Franklin, "Depolying deep learning," 2023. [Online]. Available: https://github.com/dusty-nv/jetson-inference/blob/master/docs/pytorch-collect-detection.md

[5] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, "A review of yolo algorithm developments," *Procedia Computer Science*, vol. 199, pp. 1066–1073, 2022.

[6] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37.

[7] CSDN, "Yolov7," 2023. [Online]. Available: https://blog.csdn.net/weixin_45921929/article/details/126448031

[8] Roboflow, "Roboflow: Go from raw images to a trained computer vision model in minutes." 2023. [Online]. Available: https://roboflow.com/

[9] J. Speh, "Deploy yolov7 to jetson nano for object detection," 2022. [Online]. Available: https://www.hackster.io/spehj/deploy-yolov7-to-jetson-nano-for-object-detection-6728c3

[10] CSDN, "yolov3 training," 2022. [Online]. Available: https://blog.csdn.net

[11] NikHoh, "Project seminar ris | technical university of darmstadt," 2023. [Online]. Available: https://github.com/NikHoh/jetbot_ros

[12] Y. Zhang, Z. Zhang, and J. Zhang, "Camera calibration using 2d-dlt and bundle adjustment with planar scenes," 武汉大学学报 *(信息科学版)*, vol. 27, no. 6, pp. 566–571, 2002.

[13] J. Bowman, "camera_calibration," 2011. [Online]. Available: http://library.isr.ist.utl.pt/docs/roswiki/camera_calibration.html#

[14] D. Malyuta, "apriltag_ros - ros wiki," 2022. [Online]. Available: http://wiki.ros.org/apriltag_ros

[15] ROS.Wiki, "apriltag_ros/tutorials/detection in a video stream - ros wiki," 2019. [Online]. Available: http://wiki.ros.org/apriltag_http://wiki.ros.org/apriltag_ros/Tutorials/Detection%20in%20a%20video%20stream

[16] J. M. S. Motta, G. C. De Carvalho, and R. McMaster, "Robot calibration using a 3d vision-based measurement system with a single camera," *Robotics and Computer-Integrated Manufacturing*, vol. 17, no. 6, pp. 487–497, 2001.

[17] S. Lei, L. Jingtai, S. Weiwei, W. Shuihua, and H. Xingbo, "Geometry-based robot calibration method," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, vol. 2. IEEE, 2004, pp. 1907–1912.

[18] T. E. Lee, J. Tremblay, T. To, J. Cheng, T. Mosier, O. Kroemer, D. Fox, and S. Birchfield, "Camera-to-robot pose estimation from a single image," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 9426–9432.

[19] S. Tuohy, D. O'Cualain, E. Jones, and M. Glavin, "Distance determination for an automobile environment using inverse perspective mapping in opencv," 2010.

[20] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 345–405, 1991.

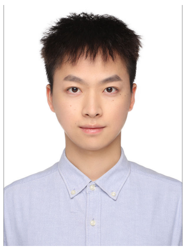
[21] Y. Gan, B. Zhang, C. Ke, X. Zhu, W. He, and T. Ihara, "Research on robot motion planning based on rrt algorithm with nonholonomic constraints," *Neural Processing Letters*, vol. 53, pp. 3011–3029, 2021.

[22] Wikipedia, "Rapidly-exploring random tree," 2020. [Online]. Available: https://de.wikipedia.org/wiki/Rapidly-exploring_random_tree#:~:text=Rapidly%2Dexploring%20random%20tree%20(RRT,zuf%C3%A4llig%20nach%20m%C3%B6glichen%20Pfaden%20absucht.

[23] C. Knospe, "Pid control," *IEEE Control Systems Magazine*, vol. 26, no. 1, pp. 30–31, 2006.
[24] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

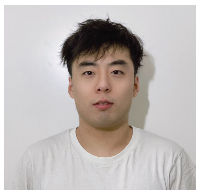
**Husam Hamu** M.Sc. Mechatronics at TU Darmstadt Interests in robotics and control

**Weilin Han** b. 1998, MSc in Computational Engineering. Convolutional Networks for Biomedical Image Segmentation in MEC Tu Darmstadt. Computergrafiken f ü r die Bohrmaschinenbearbeitung im PTW Tu Darmstadt. Interessen K ü nstliche Intelligenz, Deep Learning.



**Yiyang Wang** M. Sc. Computational Engineering at TU Darmstadt Interests in AI and robots

**Qi Li** b.1997, M. Sc. Computational Engineering (Robotik) at TU Darmstadt Interests in AI and Machine learning



**Xi Chen** M. Sc. Computational Engineering Computational Robotics