

# Design and Development of AI Robot

## IIT Kanpur ERA ICRA Technical Report

Ashok Kumar Chaudhary, Suraj Hanchinal, Suryansh Agarwal,  
Naishadh Parmar, Soumya Ranjan Dash, Karan Vaish, Shobhit Jagga, Nitish Deshpande,  
Aviral Khare, Harsh Khandelwal, Yadu Sharma

### I. INTRODUCTION

The problem statement of the AI Challenge is to build the software pipeline of an AI powered autonomous robot. The robot should be able to localize itself in the known arena, to create a map of the surrounding and use it to detect obstacles and for path planning which is explained later in the report. The sensor data obtained is used for localization and mapping. The computer vision system acts as the visual cortex of the robot to help it identify enemy robots and their armor modules. It also tracks the enemy robot when it is not stationary or when it may not be visible by registering the last visible movements. The brain of the robot relies on a reinforcement learning-based decision making process to formulate strategies of actions in the arena. With state-of-the-art hardware components and software paradigms, this robot will definitely succeed in the AI Challenge.

### II. MECHANICAL STRUCTURE

We made the following mechanical modifications on the official robot platform.

- Mounts for installation of LIDAR
- Mounts for placement of computing board
- Mounts for installation of 3 depth cameras
- Mounts for installation of gimbal camera

The Controller Area Network (CAN bus) [1] is the nervous system, enabling communication between all parts of the body. Nodes - or electronic control units (ECU) - are connected via the CAN bus, which acts as a central networking system.

1) *CAN protocol*: Some features of the CAN bus protocol are summarized as follows:

- Bus arbitration: This refers to which node gets priority or access over the bus to send a message. The philosophy around which this bus arbitration technique is built around is that:
  - All nodes are equally important in the bus (multi-master).
  - The priority over the bus is decided by the message not the node.

- Messages have priority, nodes do not.
- Each message has a unique ID which decides what priority it has over other messages.
- Broadcast concept:
  - In the CAN protocol a message is not targeted to a specific node. It is just broadcasted on the bus by the node transmitting it. The transmitting node behaves like a radio station temporarily broadcasting the message.
  - Those nodes which need the information message subscribe to it while other nodes filter it out. This concept is known as message filtering.

Because of the above advantages of CAN bus protocol, we are using it to control the chassis motors and gimbal motors.

### III. SENSOR TYPE AND USAGE

Since the size of the arena is (5m × 8m), the maximum distance would be the diagonal distance from one corner to the opposite corner ~9.43m. The total weight of the robot should be less than 20kg, as mentioned in the rulebook. Each round is 3 minutes long and comparing the dimensions of the arena to the dimensions of the robots, the number of robots and their motions, the arena can be considered to be relatively small. This requires the sensors to measure at a relatively high frequency and with high accuracy and precision.

#### A. LiDAR Sensor

We are using a Light Detection and Ranging (LiDAR) sensor for localization, obstacle avoidance and path-planning. These functionalities were achieved using a 2D LiDAR sensor. A 3D LiDAR sensor could give better inputs for us to map obstacles in the environment but as such better inputs will not better the performance of the robot to a great extent. Instead, a 3D LiDAR sensor would have flooded the onboard computer with a lot of data. So although a 3D LiDAR sensor had been beneficial for the robot, we used a 2D LiDAR sensor to save computing cost.

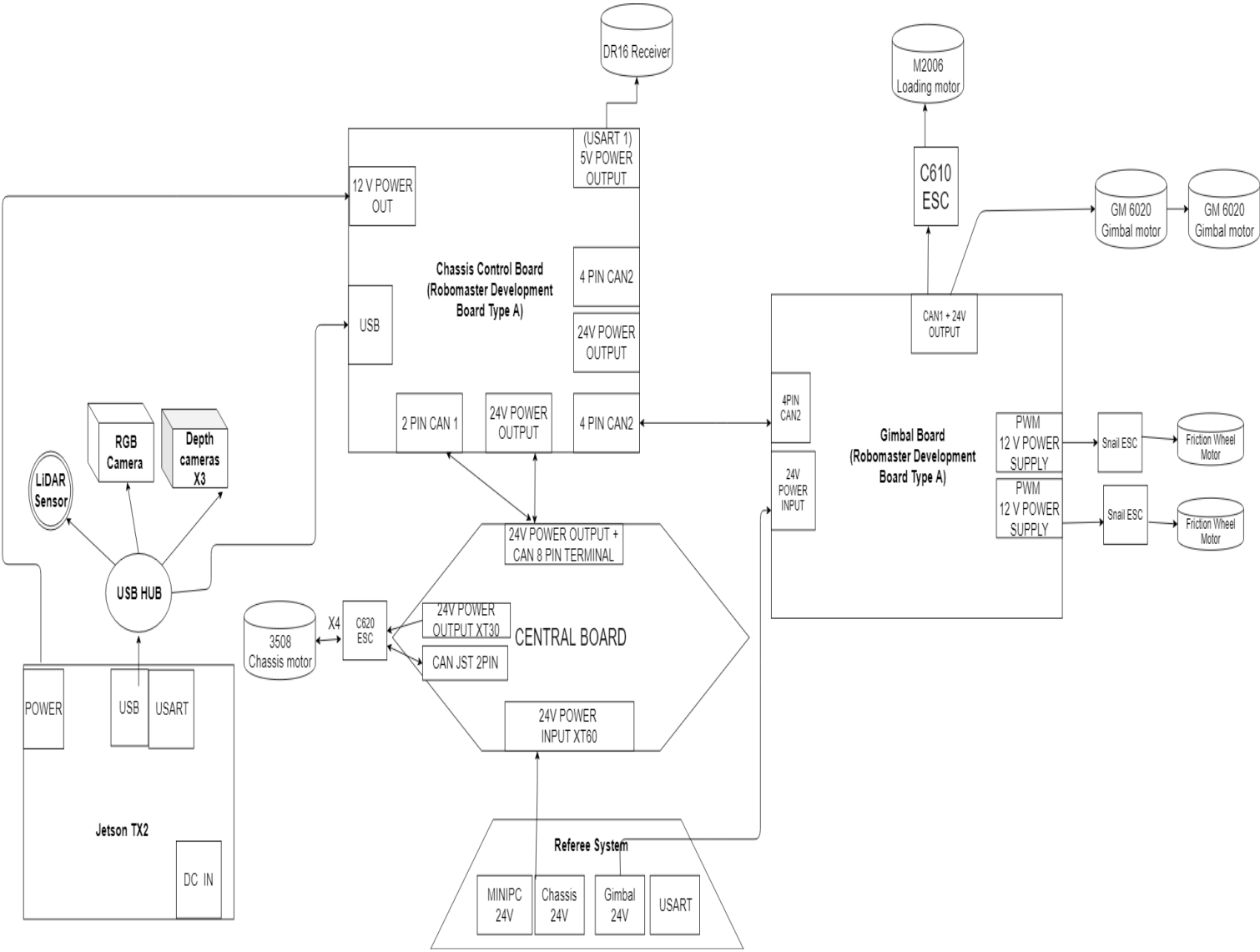


Fig. 1. Communication and Hardware Analysis

We are using a *Slamtec RPLIDAR A2*. It is adequate to be used in the arena, based on its dimensions. The horizontal field of view of the *RPLIDAR A2* makes the identification of obstacles easier; without rotating in its place, the robot can sense the entire environment. The frequency of measurements makes it suitable for our use-case, since the arena is relatively small and the measurements of the environment should be quick. The *RPLIDAR A2* also employed the *RPVision 2.0* range engine that gave a much more accurate laser scan than the traditional range engine.

### B. Cameras

The robot is equipped with depth cameras in order to detect and track enemy robots. Depth cameras are preferred over monocular cameras since depth cameras also provide the distance of the enemy robots from the robot in addition to a 2D picture to apply computer vision algorithms on. A camera was installed on the shooter as well, to accurately align in order to shoot

the armor module.

Keeping all these requirements in mind, we chosed to use the *Intel RealSense D435i*. A lower resolution was adequate to detect only the armor module, but the robot needed to have the capability of tracking the enemy robot. These processes required a greater resolution of the image, to sense more intricate details. With its frame rate the robot is visually always in sync with the environment, which is essential since the arena is relatively small. It is able to operate in different lighting environments which is essential if the camera gets flashed by the LEDs on the armor module of an enemy robot that suddenly turns up in front of the camera. 3 such cameras are installed on the bot to be able to visually sense in 360° at all times. The range of depth is suitable according to the dimensions of the arena.

The depth cameras provides the initial goal for the gimbal to align itself with and the camera attached to the shooter is used only to fine-tune the alignment. The




Hardware	Specification
 <p>Fig. 2. Slamtec RPLIDAR A2</p>	<ul style="list-style-type: none"> <li>• Depth Range: 0.15m-12m</li> <li>• Depth Resolution: 0.5mm</li> <li>• Horizontal Field-of-View: 360°</li> <li>• Measurement frequency: 4000 Hz</li> </ul>
 <p>Fig. 3. Intel RealSense D435i Camera</p>	<ul style="list-style-type: none"> <li>• Resolution: 1280 × 720</li> <li>• Operating Range (Min-Max): 0.11m - 10m</li> <li>• Depth Field of View: 85.2 × 58</li> <li>• Frame rate: 30 frames per second</li> </ul>
 <p>Fig. 4. Logitech C615</p>	<ul style="list-style-type: none"> <li>• Resolution: 720p</li> <li>• Frame rate: 30 frames per second</li> <li>• Field-of-View: 78°</li> </ul>

TABLE I  
Sensor specifications

resolution required for this camera is not so high since only armor detection is the task at hand. The frame rate is also moderate. The camera attached to the shooter is a *Logitech C615*. Its resolution and frame rate satisfies the requirements as stated earlier.

#### IV. COMPUTING DEVICE

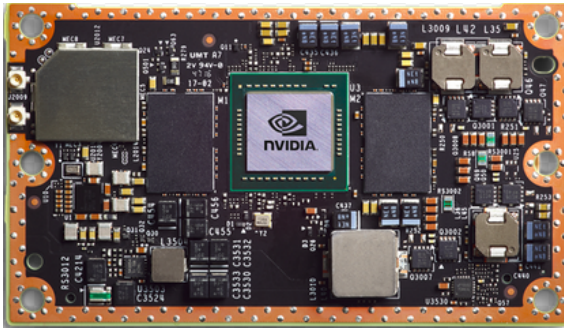


Fig. 5. NVIDIA Jetson TX2

The *NVIDIA Jetson TX2* is a full Linux computer designed to help make robots, drones and other devices that rely on computer vision applications. The board's main attraction is a GPU based on *NVIDIA's* latest Pascal architecture. The Pascal GPU brings computer vision to robots and drones, allowing them to recognize objects and navigate around obstacles. The *Jetson TX2* runs Ubuntu and ROS (Robot Operating System), which is built on top of the Linux OS. It has 32GB of

storage, 8GB of LPDDR4 memory and 802.11ac Wi-Fi which could be used to set up communication between the 2 robots in the arena. As of now we are using a single robot only. The *TX2* has two *Denver 2* CPU and four Cortex-A57 CPUs. *Jetson TX2* accelerates cutting-edge deep neural network (DNN) architectures using the *NVIDIA cuDNN* and *TensorRT* libraries, with support for Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and online reinforcement learning. We prefer to use *Jetson TX2* because it is twice as energy efficient than its predecessor, and therefore can take the load of heavy computations coming from the vision, localization, motion planning and decision making sections. The weight of the *Slamtec RPLIDAR A2* is 190g. The total weight of the three *Intel RealSense Camera* is 72g. The weight of the *NVIDIA Jetson TX2* is 85g. The weight of the *Logitech C615* is 103g. The total added weight of the equipment is 879g. This will maintain the constraint on the weight of the robot.

#### V. SOFTWARE SYSTEM

##### Automatic Recognition

The robot is using computer vision algorithms to detect the armor modules and to determine their real world position and heading relative to the robot. This information is passed to the gimbal control along with the pitch the yaw, the velocity, the distance and the

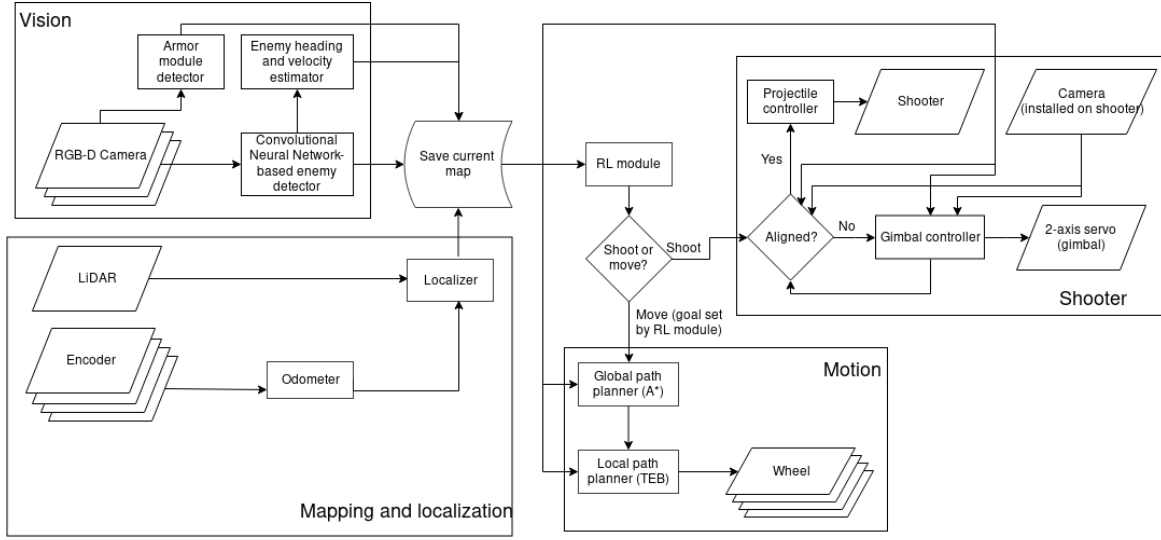


Fig. 6. Overall structure of software system

angle with which the launching mechanism needs to align itself to be able to shoot the respective armor module. Apart from this, the other two major tasks under this section are to provide the real-time pose of the enemy robot in-order to determine its direction of travel for future reference, to track the enemy robots after detecting them.

For armor-module detection, the approach used is to use the LED lights mounted on the enemy robot as a reference to approximately determine the center, height, width, coordinates of corners and orientation of the armor module. By tracking the bounding box around the armor module using the camera mounted on the gimbal, we obtain the yaw angle and pitch angle with respect to the gimbal according to the mount location of the camera. We have used image processing techniques such as contour detection, erosion, dilation, thresholding, masking, color spaces and camera intrinsic functions. After successfully detecting the modules, through the contour area function, the largest area armor module (which should be thought of as the one nearest to the robot and capable of doing maximum damage to the enemy) is selected and its information is processed (coordinates of corners, center and area) to determine real world coordinates and after calculating the appropriate parameters, send them to the gimbal control to take necessary actions.

We decided to go with the image processing approach of detecting lights owing to its high precision in detection of armor modules. On being set carefully, the hue value allows us to reject the reflections of light on armor module as non-armor module contours helping us to focus on the actual armor module in the frame. This approach also works for sufficiently large distances, convenient enough to detect and hit the enemy's armor module. The algorithm iterates at

a frame rate of 45-50 fps on a video captured by a camera of 60 fps, which very well serves our purpose of a fast detection algorithm.

Armor module tracking using KCF tracker is applied after it is detected to make the detection of armor module continuous and noise free, thus reducing the jerks on the gimbal and leading to greater precision and accuracy in shooting.

For the task of detecting and tracking the enemy robot, object detection using SSD mobilenet CNN architecture and tracking using KCF(Kernelized Correlation Filter) is used. [2] The CNN architecture is trained on visual images of the whole robot structure which the tracking algorithm uses to track the robot in the arena. It also provides the bounding box to the tracker in case the enemy bot goes out of frame. The 3 depth cameras used cover the entire 360° field of vision to detect and track the enemy robot in any direction to send data to and align the robot so as to be able to shoot the enemy. The dataset to train this CNN comprises of a large collection of images taken under different lighting conditions and divided into two classes, red and blue. Transfer learning is applied on the CNN architecture on a pretrained model of the pets dataset, wherein we decided through tests which of the last layers of the CNN architecture need to be modified as per the arena settings. SSD mobilenet is superior compared to other pre-trained models due to its speed of computation.

After detecting the enemy robot a kcf(kernelized correlation filter) tracker is deployed on the detected window to determine the direction of travel of the robot. Recovery from full occlusion is of paramount importance because the robots are subject to leaving the frame of camera every now and then. After losing

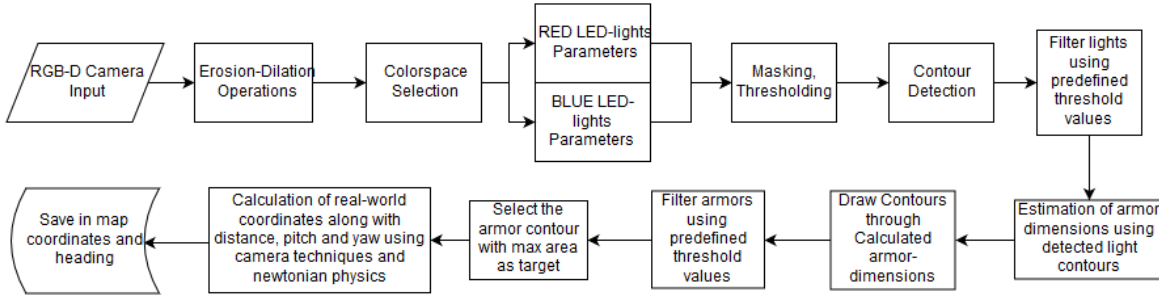


Fig. 7. Armor detection pipeline

a dummy target due to some obstacle in the path, kcf tracker tracks the object in both forward and backward directions in time and measures the discrepancies between these two trajectories. Minimizing this forward-backward error enables them to reliably detect tracking failures and select reliable trajectories in video sequences. The application of a tracker simultaneously in sync with detection also speeds up the computation, as once the object is detected the tracker stores these set of points and tracks their movement between two consecutive frames until they go out of the frame, after which the frame is again detected for enemy robots and so on the process continues and reduces the need for applying the more time consuming detection algorithm on every frame.

### Localization

It is necessary for the robot to be aware of its location. We already know the map of the arena. Hence, we get the exact location of the robot on the map. This information is used to navigate the robot as well as making better decisions on dueling with the enemy robot. The Robomaster SDK provides the robot odometry but that is not enough to accurately localise the robot. Odometry by design is error-prone due to the sensors involved and the data drifts from the ground truth over time. Odometry is only accurate in short intervals of time. We use another source of data that is the laser scan from the LiDAR sensor which is far more accurate. We correlate the laser scan to the *a priori* map and localize the robot. We can fuse the location data from the LiDAR sensor and the odometer to accurately determine the location of the robot. We make use of the `amcl_localization` package which contains the optimized implementations of these features.

Given a map of the environment, `amcl` estimates the position and orientation of a robot as it moves and senses the environment. The algorithm uses a particle filter to represent the distribution of likely states, with each particle representing a possible state, i.e., a hypothesis of where the robot is. The algorithm typically starts with a uniform random distribution of

particles over the configuration space, meaning the robot has no information about where it is and assumes it is equally likely to be at any point in space. Whenever the robot moves, it shifts the particles to predict its new state after the movement. Whenever the robot senses something, the particles are resampled based on recursive Bayesian estimation, i.e., how well the actual sensed data correlate with the predicted state. Ultimately, the particles should converge towards the actual position of the robot.

The belief, which is the robot's estimate of its current state, is a probability density function distributed over the state space. In the AMCL algorithm, the belief at a time (t) is represented by a set of M particles  $X_t = \{x_t^{[1]}, x_t^{[2]}, \dots, x_t^{[M]}\}$ .

Each particle contains a state, and can thus be considered a hypothesis of the robot's state. Regions in the state space with many particles correspond to a greater probability that the robot will be there—and regions with few particles are unlikely to be where the robot is.

The algorithm assumes the Markov property that the current state's probability distribution depends only on the previous state (and not any ones before that), i.e.,  $X_t$  depends only on  $X_{t-1}$ . This only works if the environment is static and does not change with time. Typically, on start up, the robot has no information on its current pose so the particles are uniformly distributed over the configuration space.

We don't need random initial locating in our environment as the starting point at every round is given to us, and even if a rough estimate is given to the localization process it converges very fast. High speed motion locating due to fast convergence on a rough estimate makes it superior to other algorithms.

The locating precision of the algorithm is 0.1 metres and a frame rate of 45 fps.

```

Algorithm MCL( $X_{t-1}, u_t, z_t$ ):
   $\bar{X}_t = X_t = \emptyset$ 
  for  $m = 1$  to  $M$ :
     $x_t^{[m]} = \text{motion\_update}(u_t, x_{t-1}^{[m]})$ 
     $w_t^{[m]} = \text{sensor\_update}(z_t, x_t^{[m]})$ 
     $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
  endfor
  for  $m = 1$  to  $M$ :
    draw  $x_t^{[m]}$  from  $\bar{X}_t$  with probability  $\propto w_t^{[m]}$ 
     $X_t = X_t + x_t^{[m]}$ 
  endfor
  return  $X_t$ 

```

Fig. 8. Localization structure

### Motion planning

#### Obstacle Avoidance

The robot continuously detects obstacles in the arena using the LiDAR sensor. Since the layout of the arena has already been provided, the map is initialized with these static obstacles. Mapping is needed because the robots need to know the location of dynamic obstacles in the arena (the other robots).

The mapping framework we are using is Gmapping because of its faster and efficient algorithm and strong ros support. The map we are using is a static map and the only dynamic obstacles are the enemy robots. We are able to remove the static map subtracting it from the realtime costmap extracted from the lidar. This method proved to be very efficient in providing the information of the location of the enemy robot in the frame of reference of our robot.

#### Path Planning

The whole arena is divided into grid cells assigning one cell as the ‘start’ and another as the goal. Global and local cost maps of the arena are built from the probabilistic occupancy map of the arena and these cost maps are used to plan the global and local path, respectively. The global path defines the path through which the robot has to go in order to reach the destination having the static map. The local path defines the exact trajectory that the robot has to go through in real-time, depending on the obstacles in the immediate locality. Since the map has static obstacles, the only obstacles that may obstruct the robot during the traversal of the global path is an enemy robot crossing its path. This will be handled by the local path planner.

*Global Planning:* The algorithm employed for global path planning is A\* [3]. A\* is similar to Dijkstra’s algorithm in finding the shortest path. It uses

the global cost map and comes up with the shortest path using a greedy approach. It maintains a tree of paths originating at the start node and extending those paths by one edge at a time until the goal is reached. Each grid cell of the grid map is treated to be a node of the graph and is assigned a particular value using the function  $f(n)$  where  $f(n) = g(n) + h(n)$ .  $g(n)$  is the exact cost of the path from the starting point to the node  $n$ .  $h(n)$  is the heuristic estimated cost of the path from the node  $n$  to the goal. Thus the algorithm takes both the factors into consideration to determine the shortest path for a given start point and the goal.

The algorithm gives us costs of many paths and the length of the shortest path after complete exploration of all possible paths without considering the requirement of any local changes. The actual sequence of edges leading to the path is extracted by keeping track of the predecessor of every node encountered. After the algorithm is run, each node, including the end node, points to its predecessor all the way to the start node.

Although A\* is a simple and efficient path planning algorithm, the re-planning of the global path when the map is updated with new obstacles is computationally heavier than other alternative algorithms discussed in the next paragraph. As per the arena described in the problem statement, the global map will not change other than the motion of the other robots which will be taken care of by the local path planner.

We have also tested Field D\* [4] algorithm which takes care of the environment including dynamic obstacles treating it as a dynamic environment. It had a low computational cost for the given problem as it updates the cost of the nodes closely related to the nodes that get blocked due to the changing environment. Although the algorithm very well handled the changing environment, the response of the algorithm to sudden obstacles wasn’t accurate as it took time to update the global map for the arena. Thus it is preferable to use a computationally simpler global path planning algorithm and use a local path planner that can handle dynamic obstacles.

*Local planning:* The algorithm employed for local path planning is Timed Elastic Bands trajectory optimization [5]. A configuration  $s_i$  is defined by the  $x$  coordinate,  $y$  coordinate and the angle of the robot with respect to the global frame.  $\Delta T_i$  is the time interval between two consecutive configurations  $s_i$  and  $s_{i+1}$ . The sequence of configurations is  $Q = \{s_i\}_{i=0\dots n}$  and the sequence of time intervals is  $\tau = \{\Delta T_i\}_{i=0\dots n-1}$ . Timed Elastic Band (TEB) is a tuple of the sequences



of configurations and time intervals  $B := (Q, \tau)$ .

$$f(B) = \sum_k \gamma_k f_k(B)$$

$$B^* = \underset{B}{\operatorname{argmin}} f(B)$$

$B^*$  is the optimized TEB, determined by optimizing  $f(B)$  which is a summation of scalarized component objective functions  $f_k$ . These functions belong to two basic types: constraints such as velocity and acceleration limits formulated in terms of penalty functions and objectives functions with respect to the trajectory such as shortest path, fastest execution time or clearance from obstacles. These functions rely on parameters that only depend on a subset of neighboring configurations of the band.

This property of locality of TEB results in a sparse system matrix which is represented by a hyper-graph, where the nodes correspond to the configurations and time intervals. The nodes containing parameters that contribute to the same objective function are connected by a corresponding multi-edge. Each objective functions depends on a subset of TEB-states (configurations and time differences), and a hyper-edge represents the objective function  $f_k$  and connects nodes which correspond to the configurations and time differences that occur as parameters in its evaluation. To finally optimize this graph, the general (hyper-)graph optimization "g2o" algorithm is used, which yields  $B^*$ , the ultimate optimized local path that the robot will follow.

The experimental results of this algorithm are promising since the local paths are planned keeping in mind the robot's footprint and the obstacles in the path (which will be the enemy robots in the arena). The obstacles in the path are circumvented exactly, meaning the extra time it takes to deviate from the planned global path is optimized. This is essential because every round in the competition is of 3 minutes and so every second counts.

Other local path planner choices are Dynamic Window Approach (DWA) [6] and Model Predictive Control (MPC) [mpc]. The Dynamic Window Approach deviates more from the global path than TEB based on experimental results and does not give time-optimized results. MPC produces local paths that stay close to the global path but show a lot of oscillations in doing so, meaning it is not the most robust local path planner. Keeping these factors and the time-optimality requirement in mind, TEB is the best choice.

The frequency of local planner is 40Hz, frequency of global planner is 3Hz, tolerance of path planner

is 0.05 metres. The planner has a maximum obstacle avoidance speed of 2.5 m/s.

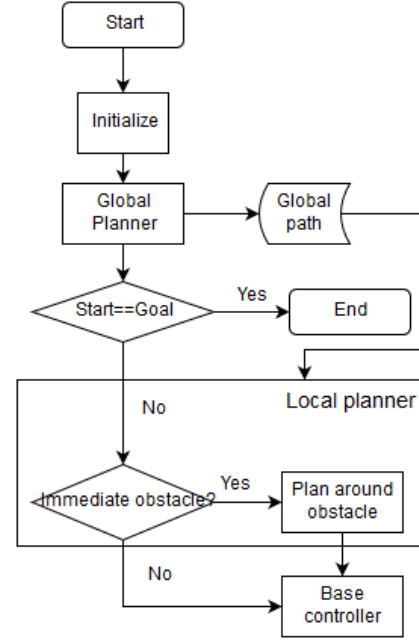


Fig. 9. Path planning structure

### Automatic Firing

The gimbal's main task is to align the launching mechanism with the detected target after the armor is detected by cameras deployed for detection task. Once aligned, the firing task desires the gimbal to rotate to the correct direction and the shooter to translate to the correct angle in order to shoot the enemy's armor with high precision and accuracy. Pitch rate and yaw rate parameters are required to position the gimbal in the direction of the enemy robot, which are provided by the tracking algorithm. The tracking algorithm stores the center coordinates of the bounding box generated around the enemy robot. With the reference of these coordinates the pitch rate and yaw rate is calculated.

After detecting the armor module in the camera frame, using the solvePNP algorithm we apply the transformation given below to convert the image frame coordinates into the real world coordinates of the center of the detected armor module.

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Fig. 10. Transformation matrix

Once the real world coordinates were obtained, we aligned the median line of the camera frame with the x coordinate of the detected point. Thus we were able to obtain the yaw angle. Neglecting air resistance and considering the y coordinate of the armor module being fixed with respect to real world, we used the formula

to obtain the pitch angle.

$$\theta = (1/2) * \arcsin(gy/v^2) \quad (1)$$

After the target is detected and aligned, we had to maximize the trajectory range. The optimization methods we are using the Method of Golden Ratio that required one new run each time and a Three-Points Plan. During the maximization, the tolerance of the angle is reduced stepwise by comparing the trajectory ranges to obtain an optimal alignment. The Method of the Golden Ratio allows us to select the intermediate points  $x$  and  $y$  so that we are able to use one of the two values from the previous step again. The intermediate points divide the interval in a certain fixed ratio.

The farthest strike distance of the shooting algorithm is 2.5 metres currently, with a strike precision of 7 out of 9 clear hits on the armor module.

### Automatic Supply

Automatic supply to replenish balls whenever the number of balls in the shooter gets below the lower limit is accomplished using location obtained by lidar, depth obtained from the depth camera and the AprilTag used to fine tune with the projectile fall of the balls from the supplier. The AprilTag is detected using Aruco library of OpenCV giving us the id of the tag to align with the supplier properly and prevent the robot from going into the supplier zone of the enemy robot saving us from the penalty.

### Smart Decision

#### Need for using Reinforcement Learning

We have implemented reinforcement learning algorithms for strategy planning and taking intelligent decisions in the arena. For the purpose of this competition, conventional supervised learning methods are not suitable because the actual arena conditions and scenarios that the bot would face will vary from the training zone for the robot. The strategies used by the opponent team will be unknown to us and the bot will have to make decisions on the spot. The number of strategies that can be followed by the enemy robot is practically huge and it is not feasible to hardcode for all possible strategies. But the strategies used by the robots can be generalized in several ways. One of the strategies that an enemy robot would follow would be to keep the armor module constantly in motion which makes it harder for our robot to shoot it. An effective counter would be to predict the trajectory of the armor modules and "pre-fire" in that predicted trajectory. Another strategy could be to orient the bot at around  $45^\circ$  to the viewing direction of enemy bot

on seeing it such that it sees two armor modules but partially so that it cannot make a good hit. An effective counter could be to orient our bot so that the armor module appears completely in its field of view. Other such strategies can be discovered and implemented which are not as evident but might be rewarding in a certain scenario. Hence, have simulated dueling of robots using Q learning to discover and implement other such strategies.

### Methodology

1) *Q learning*: Reinforcement learning models can be considered to have sparse and time-delayed labelled datasets. Reinforcement learning approaches can be broadly categorised as model-free and model-based. We are pursuing the popular model-free Q-learning approach. Q-Learning is an RL algorithm that creates and maintains a table of values which estimate the utility of taking an action in a state. The agent is given a function associating states with a predetermined reward. The algorithm then feeds rewards back to state-action pairs that lead to reward states creating gradually improving utility estimates. The update formula of Q-learning is:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} (Q(s', a')))$$

where  $Q(s, a)$  is the Q value of the current state-action pair,  $Q(s', a')$  is the Q value of the successor state-action pair,  $r$  is the reward associated with the successor state,  $\alpha$  is the learning rate parameter and  $\gamma$  is the discount factor parameter.

2) *Behaviour Tree*: We have made our RL model as modular as possible by using a Behavior Tree (BT) [7] to structure the switching between different tasks in our autonomous robot. The main advantage of using a behaviour tree is that individual behaviors can easily be reused in the context of another higher-level behavior, without needing to specify how they relate to subsequent behaviors.

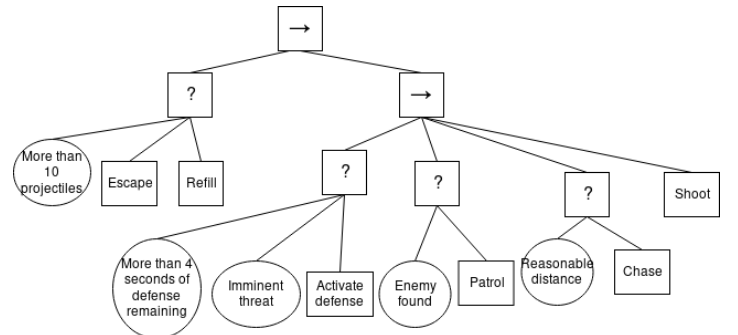


Fig. 11. Behaviour Tree as input

In the standard form, BTs are composed of nodes that either dictate tree traversal logic or execute behaviours. Behaviour nodes contain various status codes



to indicate the current Behaviour's state ("success", "fail" or "running"). Traversal logic nodes are called Composite behaviours where, depending on the results returned from one or more of their children, they can succeed or fail. The most common examples of Traversal logic nodes are the Sequence and Selector nodes. Another node commonly used in BTs is the Condition node. These are usually located as part of a Sequence Node's children and can be used to check the state of an agent or the environment. Early checks within a Sequence node can indicate whether any of the node's children are likely to succeed or fail and inhibit later behaviours in the sequence from running. This is particularly useful if the behaviour to be executed is computationally expensive and the cost could be avoided.

3) *QL-BT*: One of the common obstacles encountered in Reinforcement learning is that the size of the solution space of a problem grows exponentially with each additional feature describing the state. This problem, described by Bellman as "the curse of dimensionality," has limited the application of reinforcement learning in many domains. The point of combining Behaviour Trees with Reinforcement Learning is to address the curse of dimensionality, and do RL on smaller state spaces, while the BT connects these subproblems in a modular way.

Q-learning systems are comprised of states, actions, and a reward function. Each agent's internal state values are combined with percept values to provide a single state for use with the Q-learning algorithm. In a Q-learning context, the deepest level Sequence nodes of a BT can be seen as actions, because they group together lower level actions and execute them consecutively without interruption. Each Sequence node also contains condition nodes towards the beginning of the sequence. We split the lowest level behaviours into atomic actions and apply hierarchical RL to them.

#### Overview of the algorithm

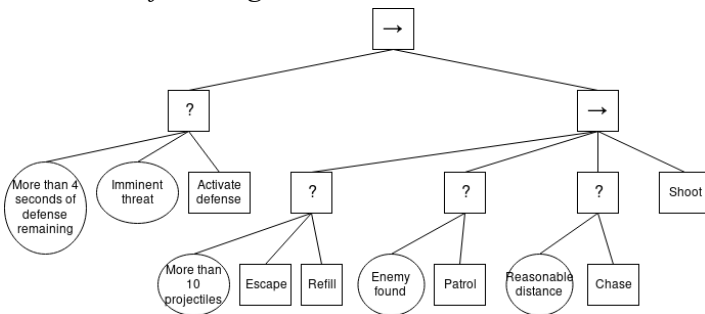


Fig. 12. Reordered Behavior Tree (With Max Q-values)

The algorithm begins with a BT as an input. The tree is analyzed to find the deepest Sequence nodes. These

nodes are identified as actions for the RL stage. The table is then divided into sub-tables by action and the highest valued states for the action are extracted into the Q-Condition nodes within the BT. These actions are used in an offline Q-learning phase to generate a Q-value table. The Condition nodes in the input BT are then replaced with the Q-Condition nodes. Finally, the BT's topology is reorganized by sorting each node's child by their maximum Q-value, which provides us with a more optimized permutation of the BT.

#### Video Link Address

The video showing the implementation of the mentioned algorithms and strategies can be found at <https://youtu.be/-CWaQt43hYU>.

#### References

- [1] R. De Andrade et al. "Analytical and Experimental Performance Evaluations of CAN-FD Bus". In: *IEEE Access* 6 (2018), pp. 21287–21295. issn: 2169-3536. doi: 10.1109/ACCESS.2018.2826522.
- [2] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (Apr. 2017).
- [3] M. S. Ganeshmurthy and G. R. Suresh. "Path planning algorithm for autonomous mobile robot in dynamic environment". In: *2015 3rd International Conference on Signal Processing, Communication and Networking (ICSCN)* (2015), pp. 1–6.
- [4] David Ferguson and Anthony (Tony) Stentz. *The Field D\* Algorithm for Improved Path Planning and Replanning in Uniform and Non-Uniform Cost Environments*. Tech. rep. CMU-RI-TR-05-19. Pittsburgh, PA: Carnegie Mellon University, 2005.
- [5] Christoph Rösmann et al. "Efficient trajectory optimization using a sparse model". In: *2013 European Conference on Mobile Robots* (2013), pp. 138–143.
- [6] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. "The Dynamic Window Approach to Collision Avoidance". In: *Robotics Automation Magazine, IEEE* 4 (Apr. 1997), pp. 23–33. doi: 10.1109/100.580977.
- [7] Rahul Dey and Chris Child. "QL-BT: Enhancing behaviour tree design and implementation with Q-learning." In: *CIG. IEEE*, 2013, pp. 1–8. isbn: 978-1-4673-5308-3. url: <http://dblp.uni-trier.de/db/conf/cig/cig2013.html#DeyC13>.