Deep reinforcement learning(SLAM)

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Rajendra Singh
(111601017)

under the guidance of

Dr. Chandra Shekar

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Deep reinforcement learning(SLAM)**" is a bonafide work of **Rajendra Singh** (**Roll No. 111601017**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

Dr. Chandra Shekar

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

I would like to express my special thanks of gratitude to my mentor **Dr. Chandra Shekar** as well as our project coordinator **Albert sunny** who gave me the golden opportunity to do this wonderful project on the topic **Deep reinforcement learning(SLAM)**, which also helped me in doing a lot of Research and I came to know about so many new things I am really thankful to them. Secondly I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Contents

List of Figures

Chapter 1

Introduction

During my summer internship at UST Global (2019), I studied various SLAM and swarm algorithm. Now my attempt is to use reinforcement learning based algorithm for robot control, path planning, mapping and localization.

1.1 Problem Statement

Study Simultaneous localization and mapping(SLAM) and its various type and application. Study quadcopter control and path planning. Improve same using reinforcement learning based algorithms. Study multi robot system and their various coordination algorithms(swarm algorithms).

1.2 Goal for this semester

For this semester my main focus is to study SLAM and Swarm behaviour of robots. I'll writing the efficient reinforcement learning based algorithms. Test and analysis these algorithms with simulated robots using ros. I will be learning to design the custom environment for simulations. By the end this semester I'll start implementing these algorithms on the

real hardware.

1.3 Organization of The Report

Code written for slam package is large and is in form of ros packages, hence this report contain pseudo codes and their documentation and algorithms. Full detailed for each of this can be found git.[1]

chapter 1 : We introduced the problem statement, discussed goal for this semester and organisation of this report.

chapter 2 : We'll introduce SLAM problem and discuss its common terminology, algorithms and my prior work in it using gmapping and RTABMAP slam.

chapter 3 : We'll discuss about the various Reinforcement learning algorithm and it implementation on mountain car problem using the openAI gym.

chapter 4 : We'll discuss about control and planning in drones firstly using pid controller and later using the reinforcement learning.

chapter 5 : We'll discuss about the sub-field of AI which swarm intelligence and its various application in robotics.

chapter 6 : In the end, we'll conclude my work and discuss future work.

Chapter 2

Simultaneous localization and mapping (SLAM)

2.1 Motivation

Mobile robots are gaining significant relevance and starting to enter our daily life. For example, Vacuum cleaner,robotic lawnmower. Besides, they become more popular in the industry. Many companies started to use automated guided vehicles or mobile manipulators in their warehouses. Moreover, mobile robots are used in explorations of caves, pyramids, reefs or similar things. Another possible application of mobile robots in military operations where they could be used for transportation, reconnaissance or even fighting. The main request to a mobile robot obviously is mobility. The robot has to be capable of moving from its current pose, defined by its position and orientation, to a desired target configuration. In order to achieve this, it needs to find a valid collision-free path connecting the corresponding configurations. Moreover, there may be other demands to the path like to be as short, as fast or as safe as possible. If the robot wants to find such a path it needs access to several information. Firstly it needs information about itself such as its own size and its manoeuvrability. Furthermore, the environment including the current pose and the target

pose have to be known. For this purpose mobile robots usually use a map. Now given this information path can be found. These paths can be calculated with several path planning algorithms. Planning algorithms usually work in discrete domains, which is why often trajectory optimization methods are used in the second step in order to smoother these paths. SLAM methods can create a map of an unknown environment and localize the robot in this map at once. That means they allow the robot to deal with any unknown environment. Therefore they are very popular and often used in mobile robotics. With SLAM there is no more need to create a map and hand it to the robot in order to navigate it since the robot develops its own.[2]

2.2 Introduction

Human beings get the information about their surrounding through vision, hearing the voice through ears, smelling through the nose and they do feel the strength of objects through touch. In general human beings get information about reality through senses. A robot cannot explore an unknown environment unless it is provided with some sensing sources to get information about the environment. There are different kinds of sensors used to make a robot capable of sensing a wide range of environments: Odometers, Laser range finders, Global Position System (GPS), Inertial Measurement Units (IMU), Sound Navigation and Ranging (Sonar) and cameras. The map of the environment is a basic need for a robot to perform actions like moving room to room, picking an object from one place and taking it to another one. To perform such actions, the robot should not only know about the environment but while it is moving it should also be aware of its own location in that environment. The aim to achieve is the navigation of the robot in a new and unknown environment by using the ROS. Robot builds the map, localizes itself on the map and performs navigation. Simultaneous localization and mapping is a problem where a moving object needs to build a map of an unknown environment, while simultaneously calculating its position within this map. There are several areas which could benefit from

having autonomous vehicles with SLAM algorithms implemented. Examples would be the mining industry, underwater exploration, and planetary exploration.[2]

2.2.1 Formulation

The SLAM problem, in general, can be formulated using a probability density function denoted [2]

p(xt, m—z1: t, u1:t)

' where, xt is position of the vehicle at time t, m - map, z1:t - vector of all measurements(Observations), u1:t - vector of the control signals of the vehicle(control commands or odometry "'

2.2.2 Data association

Basically, the concept data association is to investigate the relationship between older data and new data gathered. In a SLAM context, it is of necessity to relate older measurements to newer measurements. This enables the process of determining the locations of landmarks in the environment, and thus this also gives information regarding the robot's position within the map. Simultaneous localization and mapping.[2]

2.2.3 Loop closure

The concept of loop closure in a SLAM context is the ability of a vehicle to recognize that a location has already been visited. By applying a loop closure algorithm, the accuracy of both the map and the vehicle's position within the map can be increased. However, this is not an easy task to perform, due to the fact that the operating environment of the vehicle

could contain similar structural circumstances as previously visited locations. In that case, if the loop closure algorithm performs poorly, it could lead to a faulty loop closure, which could corrupt both the map as well as the pose of the vehicle within the map.[2]

2.2.4 Full SLAM and online SLAM

There are two different types of SLAM problems. The online SLAM problem of which only the current pose xt(x at time t) and the map m are expressed, given the control input u1:t and measurements z1: t. As well as the full SLAM problem which expresses the entire trajectory of the robot. The PDF of the full SLAM problem is denoted as p(x0: t, m—z1:t , u1:t), where all the poses of the robot are considered, including its initial pose x0.

2.2.5 General models for SLAM problem

Challenge of SLAM is to compute the robots state xt and the map m depending on previous observations z0:t and control commands u0:t-1. For this to happen a motion model returning the current state xt depending on the previous state xt-1 and the last control command ut-1 is needed. The motion model is typically derived from the odometry data of the robot. Since odometry is sensitive to errors like non-round contours of the wheels, belt slip or inaccurate calibration this model is not deterministic. However, it can describe the belief of the state p(xt) as conditional probability p(xt) := P(xt —xt-1, ut-1) Furthermore an observation model returning the current expected sensor output zt depending on the state xt and the map m has to be known. Due to inaccuracies of the sensors, this model is also not deterministic, but can be described as conditional probability p(zt) := P(zt —xt , m) These models being non-deterministic means that everything we calculate using them will not be absolute. That implies that the SLAM problem can not be solved absolutely. It is not possible to compute the actual state or map. But what can be estimated is a belief over state and map represented by a probability. Assuming we know the motion and transition model of a certain robot and all previous states x0:t , the belief over the map

m could be easily computed as p(m) := P(m—x0:t , z0:t , u0:t-1) Vice versa, if the map m and the models are known, the state xt can be estimated as p(xt) := P(xt —m, z0:t , u0:t-1) But in the SLAM problem the robot knows neither its location nor the surrounding environment. And it is not just that both are unknown, in fact they depend on each other, which makes SLAM a chicken-egg-problem. The challenge is to compute the belief over the map m and the state xt simultaneously since they can not be estimated one after each other. Moreover, this has to be done carefully because wrong data association can lead to divergence. In fact there exist two different definitions of the SLAM problem. The first only seeks to recover the current pose xt and is formulated as p(xt , m) := P(xt , m—z0:t , u0:t-1) This description is known as online SLAM. The other definition, which is called full SLAM, estimates the whole state trajectory x0:t as p(x0:t , m) := P(x0:t , m—z0:t , u0:t-1) Above two equations may seem impossible to solve at first. But since the solution of the SLAM problem is very significant for mobile robotics, it is a well-studied problem and there were several methods developed to solve it.[2]

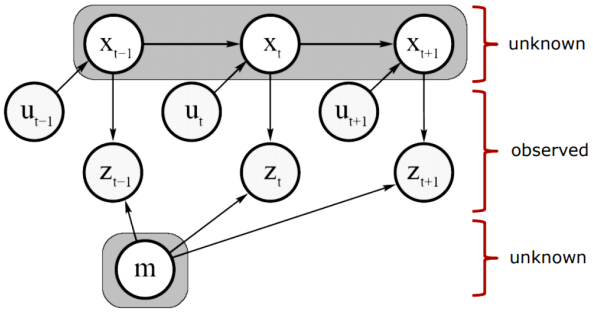

Figure 2.1 SLAM

2.3 Implementation

2.3.1 Gmapping slam using lidar

I used RPLidar A208 model to create map the environment. I implemented gmapping slam to create the map and tested the navigation on turtlebot burger model in gazebo

simulation. Detailed description of which is as below:[2]

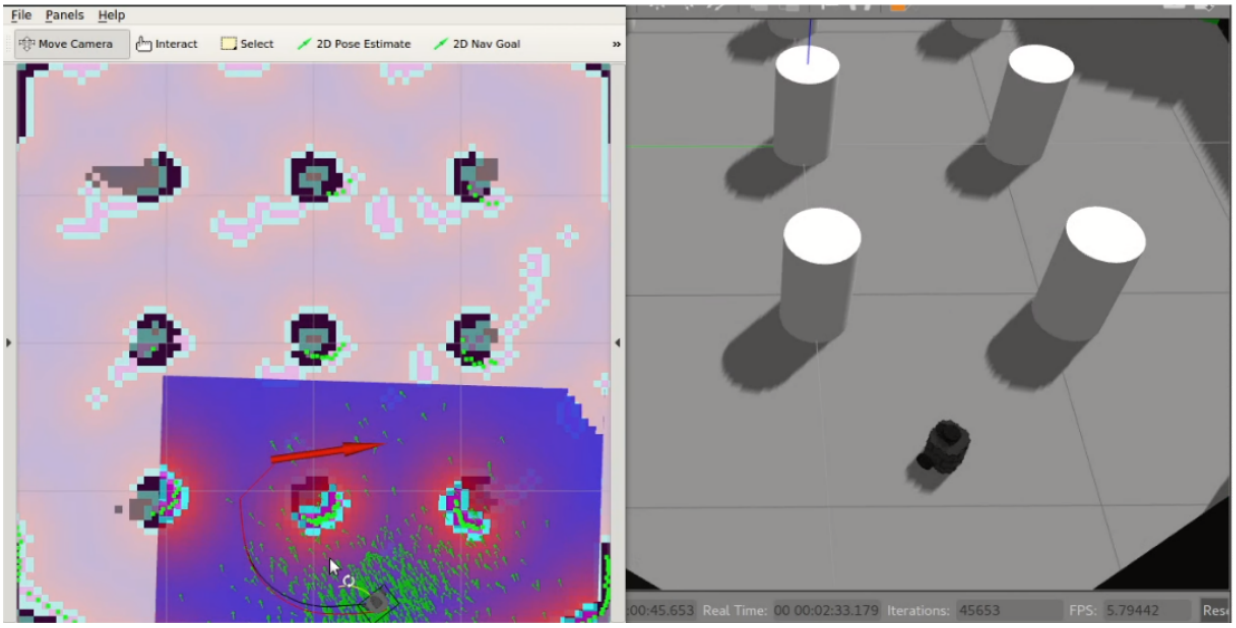

Figure 2.2 Gmapping using lidar

- Firstly I create ros node to read the lidar data and publish scan message on /scan topic.
- Here we use the turtlebot burger model in gazebo, to get the relevant tf of robot.
- Instead of subscribing to the scan topic generated by gazebo model we subscribe our real robot lidar sensor.
- Now let create launch file for mapping where we do the following: demo
  - Launch the turtlebot world and spawn the burger model.
  - Now we run the instance of the robot state publisher for getting relevant transforms.
  - Load the urdf to robot description param.

- Launch lidar node.
- Now we run the slam_gmapping node from gmapping package with appropriate base, odom and map frame.
- Launch the rviz node.
- Now we move the robot around using the teleop node, create the map.
- Once we're happy with the generated map, we save it using the map_saver node from map_server pkg.
- Now once we have the good map of environment, we navigate around the space: demo
- Here also we run the similar launch file as in mapping, just that we do not launch slam_gmapping node.
- We launch the map_server node for publishing the already saved map, as global map.
- We also run the instance of the acml node which will localise the robot in the surrounding environment, publish robot pose as covariance vector, which will improve over time.
- Then we run the move_base node which is responsible for robot path planning. Here we used 2 types of DWAPlanner:
  * Global planner : It is responsible for giving the global optimised path between start and destination pose.
  * Local planner: It is responsible for planning the path locally for instance, when there also people moving in the same environment. In this case we can not navigate based on the global plan is based out one time static map.
- Here our case we rviz gui tool to initial state estimation, and giving the goal.

2.3.2 RTAB-MAP slam using 3D Camera

demo I used intel realsense D435 to get the pointcloud data of real environment. This data was used to create the 3D map to environment using RTABMAP slam algorithm. Real-Time Appearance-Based Mapping(RTAB-Map) is a RGB-D, Stereo and Lidar Graph-Based SLAM approach based on an incremental appearance-based loop closure detector. The loop closure detector uses a bag-of-words approach to determinate how likely a new image comes from a previous location or a new location. When a loop closure hypothesis is accepted, a new constraint is added to the map's graph, then a graph optimizer minimizes the errors in the map. A memory management approach is used to limit the number of locations used for loop closure detection and graph optimization, so that real-time constraints on large-scale environnements are always respected. RTAB-Map can be used alone with a handheld Kinect, a stereo camera or a 3D lidar for 6 DoF mapping, or on a robot equipped with a laser rangefinder for 3 DoF mapping. I followed these steps for creating 3d map of the environment:[2]

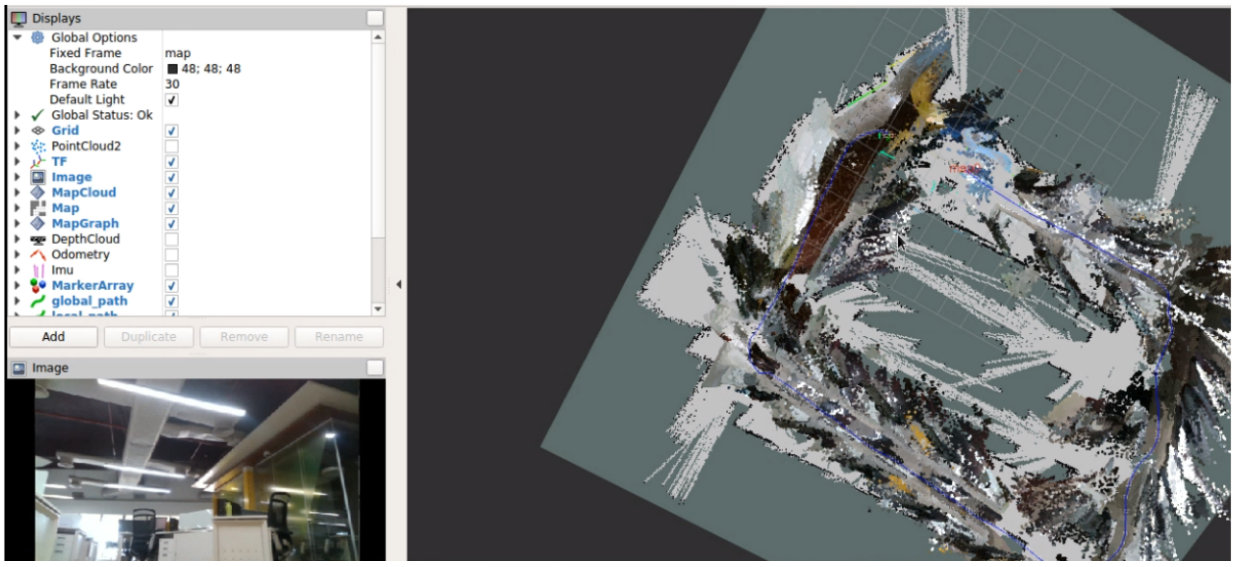

Figure 2.3 RtabMap on AGV using lidar and 3D camera

- First of we launch node which read the rgb and depth image from the 3d camera and publish as rostopic.
- We run rgbd_sync node to sync these images if they aren't already.
- We can run the rtabmap is these two modes:
  - Visual odom : This is feature based odometry, where we use the image feature to localise the robot in the environment. An RGBD odometry finds the camera movement between two consecutive RGBD image pairs. The input are two instances of RGBDImage. The output is the motion in the form of a rigid body transformation.
  - Icp_odom : Its use iterative closest point(icp) algorithm to get the odom data.
- We use the lidar data along with pointcloud2 data to get more accurate mapping of the environment.
- Then run the rtabmap node with appropriate sensor topics, frames and parameter.
- For visualisation of the mapping we can either use rtabmapviz or rviz.
- Once we are happy with the obtained map, we stop the mapping. This will save the 3d map,poses and camera images .db file which can be used later during the navigation.
- We can also further do post processing on the obtained to map to further refine it.

Similar to the mapping we can also run the rtabmap in localisation mode where its will use previously generated map to localise in environment. Also we can give the goal pose directly through rviz path planning.

2.4 Conclusion

In this chapter studied various type of slam and its related terminology. We also implemented gmapping and RTABMAP slam. This chapter establish the base for future work where we will improve this algorithms usign reinforcement learning based algorithms.

Chapter 3

Reinforcement Learning on Mountain car

There are various RL algorithms. Here we'll focus on Q learning, sarsa and DQN.

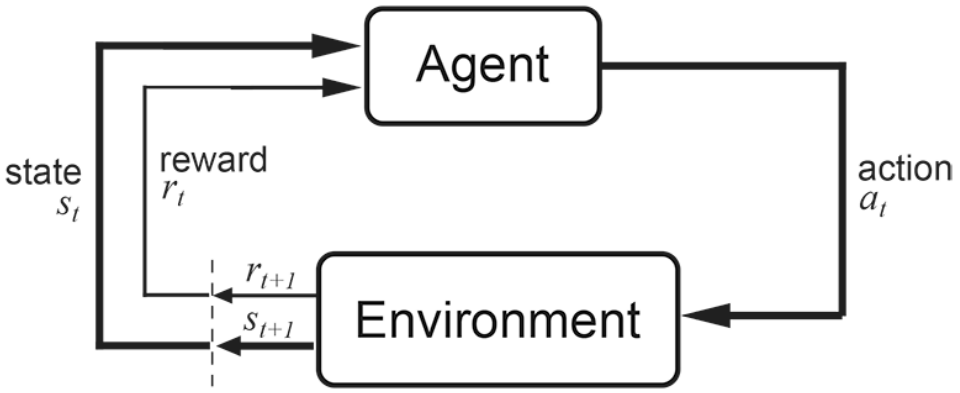

Figure 3.1 Agent and environment

3.1 Q-learning

Lets see Q learnning first.

3.1.1 Algorithm

Pseudo code

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

3.1.2 Implementation

Openai gym is open source library for implementing the RL algorithm.[3] Let train the mountain car using the q learning as below pseudo code. Full code of the same can be found here.

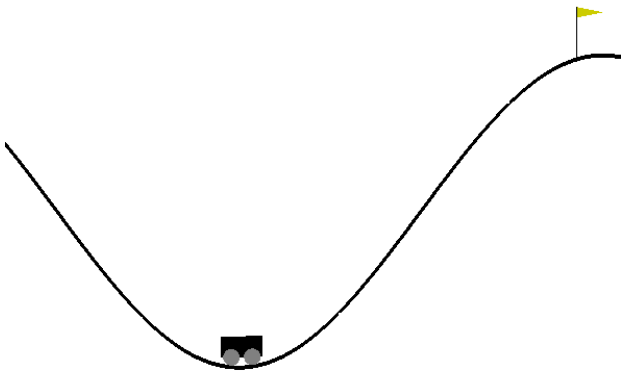

Figure 3.2 Mountain Car

Pseudo Code

```
def Qlearning(environment, learningRate, gamma, epsilon, n_Episodes):
    find_Number_state_in_x_and_y()
```

```
    initialise qTable and epsilon decay rate
    Run for n_Episodes
        state= resetEnvironment();
        state = discretiseState(initState)
        while not done
            action = epsilonGreedy()
            newstate, Reward, done, info = env.step(action)
            s_d = discret(s_)# Discretize
            update_qTable()  #as below
            '''
            Q[sd[0], sd[1],a] =(1-lr)*Q[sd[0],sd[1],a]  +   lr*(R +
gamma*np.max(Q[s_d[0], s_d[1]]))
            '''
            state=newstate
        epsilionDecayUpdate()
        printAvgRper100episodes()
    env.close()
    return
```

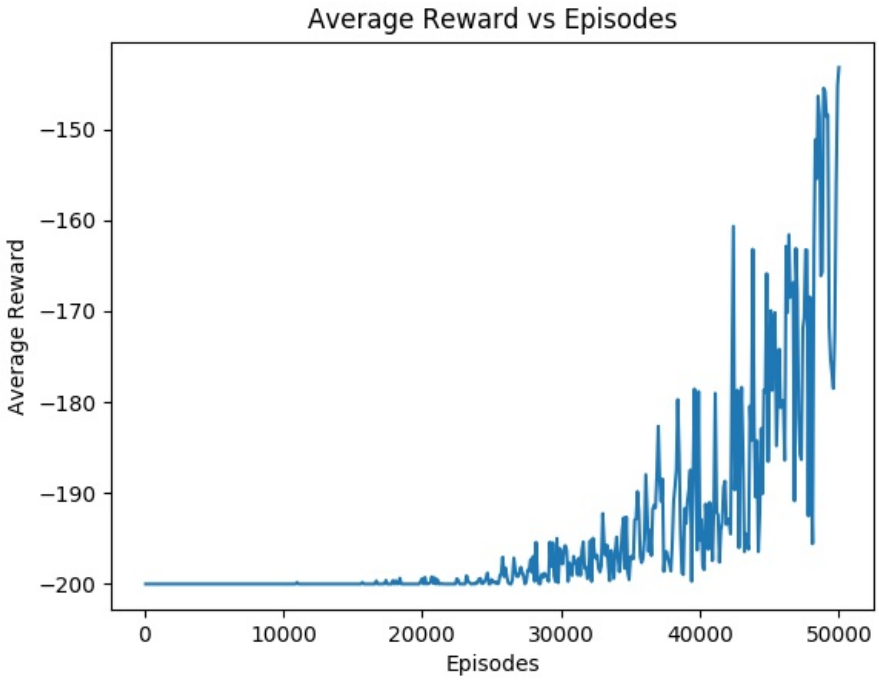

Figure 3.3 Result - average reward for 100 episodes

3.2 Sarsa

Here we try to improve the result of above implementation by using sarsa. Fully detailed code of the same can be found here.

3.2.1 Algorithm

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

3.2.2 Implementation

Pseudo Code

```
def Sarsa(environment, learningRate, gamma, epsilon, n_Episodes):
    find_Number_state_in_x_and_y()
    initialise qTable and epsilon decay rate
    Run for n_Episodes
        state= resetEnvironment();
        state = discretiseState(initState)
        while not done
            action = epsilonGreedy()
            newstate, Reward, done, info = env.step(action)
            s_d = discret(s_)# Discretize
            update_qTable() #as below
            '''
            if np.random.random() < 1 - ep:#epsilon greedy exr vs expt
                    a_ = np.argmax(Q[s_d[0], s_d[1]])
```

```
                else:
                    a_ = np.random.randint(0, env.action_space.n)
            Q[sd[0], sd[1],a] =(1-lr)*Q[sd[0],sd[1],a]  +   lr*(R +
gamma*Q[s_d[0], s_d[1],a_])
            '''
            state=newstate
        epsilionDecayUpdate()
        printAvgRper100episodes()
    env.close()
    return
```

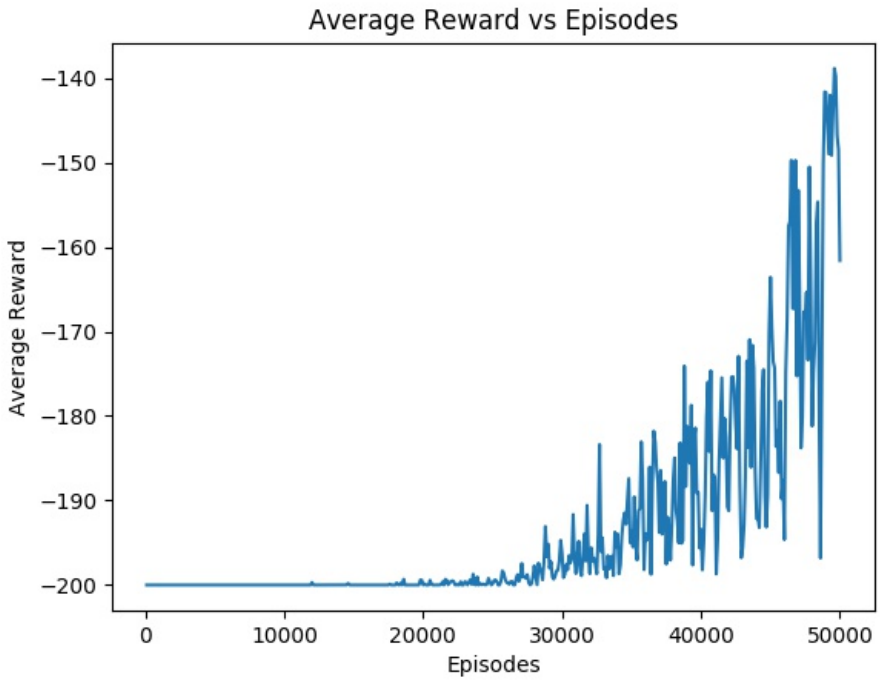

Figure 3.4 Result - average reward for 100 episodes

3.3 Deep Q-Network (DQN)

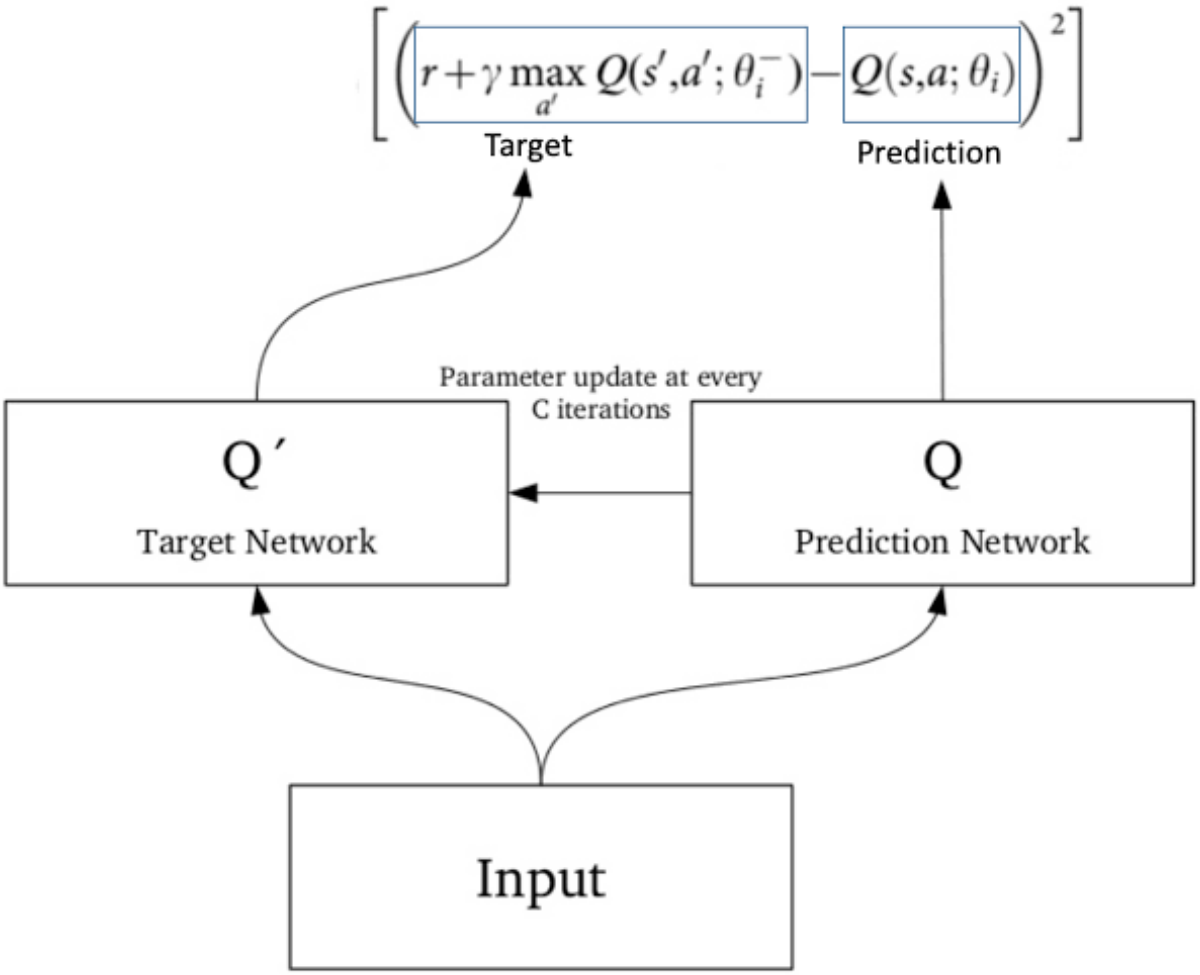

Figure 3.5 Flow chart of DQN

3.3.1 Algorithm

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 3.6 Deep Q-Network algorithm

3.3.2 Implementation

I used the the keras and gym to implement DQN on mountain Car problem. Detailed code for the same can be found here.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 24)                72
_________________________________________________________________
dense_2 (Dense)              (None, 48)                1200
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 147
=================================================================
Total params: 1,419
Trainable params: 1,419
Non-trainable params: 0
_________________________________________________________________
```

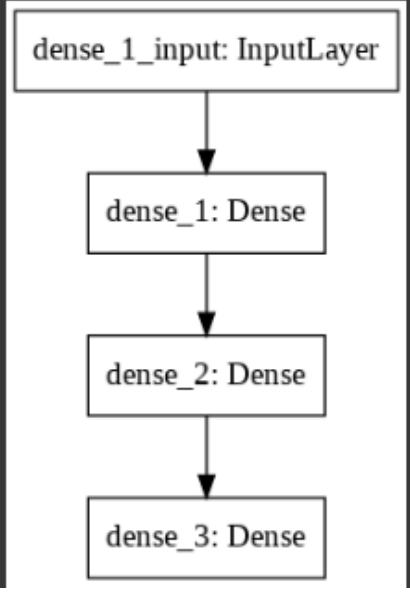

Figure 3.7 DQN model 1(keras)

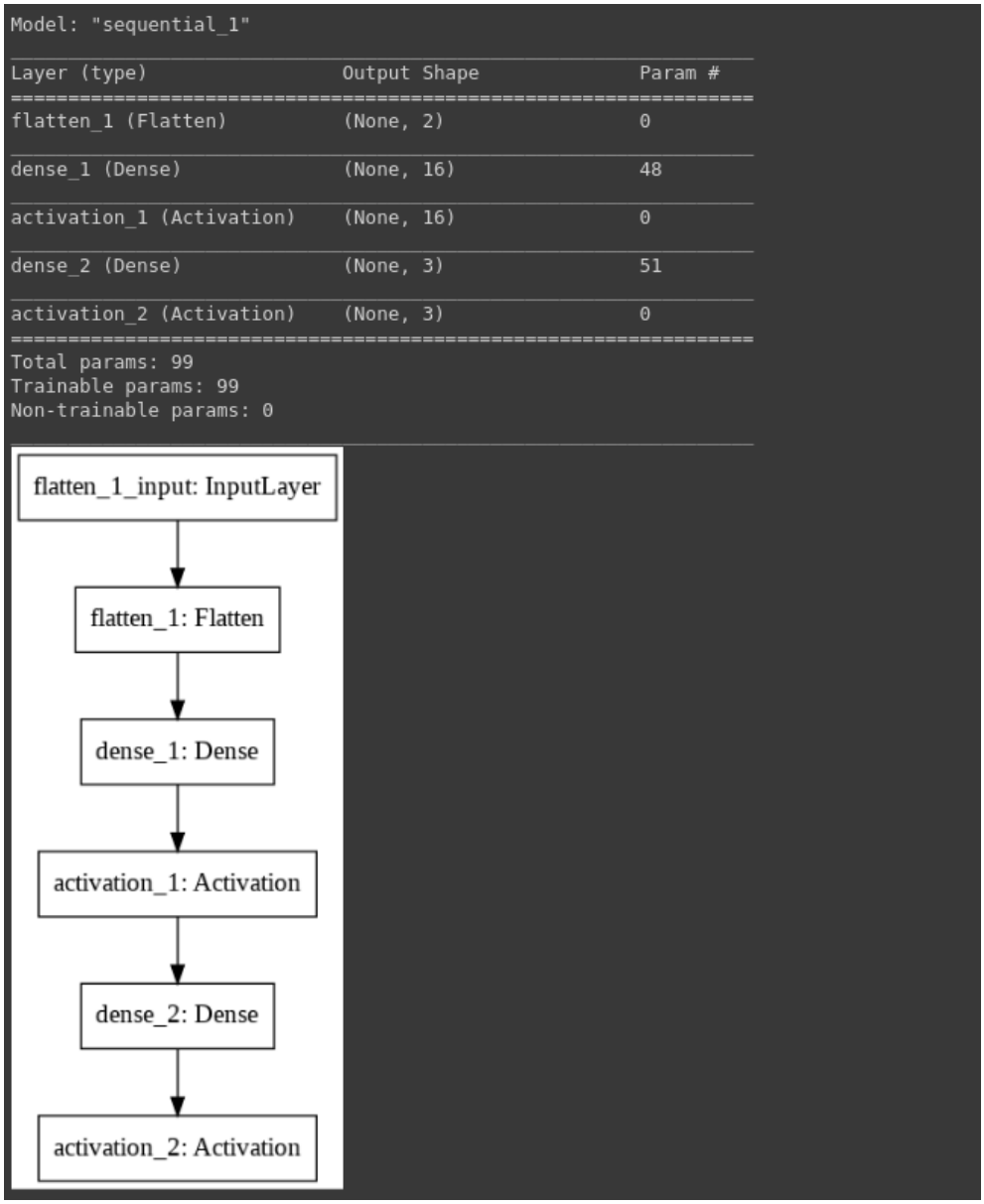

Figure 3.8 DQN model 2(keras-rl)

3.4 Conclusion

In this chapter, we studied and implemented various algorithms like Q-learning, Sarsa, DQN on the mountain car problem using the openai gym library. Here DQN out performs both q learning and sarsa which in turn out perform q learning. Average reward for q-learning is (-143), Sarsa is (-132) and DQN is (-110).

Chapter 4

Controlling drones

4.1 PID controller

My first sub-task of BTP was to train a drone in simulation to go from point A to point B without much deviations, jerks and oscillations. It can be achieve in two ways, one by pid controller and other by training a drone by reinforcement learning. Its important to understand PID controller approach first, later will discuss the RL based approach.

4.1.1 Introduction

PID

PID stands for Proportional, Integral, Derivative, it's part of a flight controller software that reads the data from sensors and calculates how fast the motors should spin in order to retain the desired rotation speed of the aircraft.[4]

The goal of the PID controller is to correct the "error", the difference between a measured value (gyro sensor measurement), and a desired set-point (the desired rotation speed). The "error" can be minimized by adjusting the control inputs in every loop, which is the speed of the motors.[5]

There are 3 values in a PID controller, they are the P term, I term, and D term:

"P" looks at present error – the further it is from the set-point, the harder it pushes

"D" is a prediction of future errors – it looks at how fast you are approaching a set-point and counteracts P when it is getting close to minimize overshoot

"I" is the accumulation of past errors, it looks at forces that happen over time; for example if a quad constantly drifts away from a set-point due to wind, it will spool up motors to counteract it

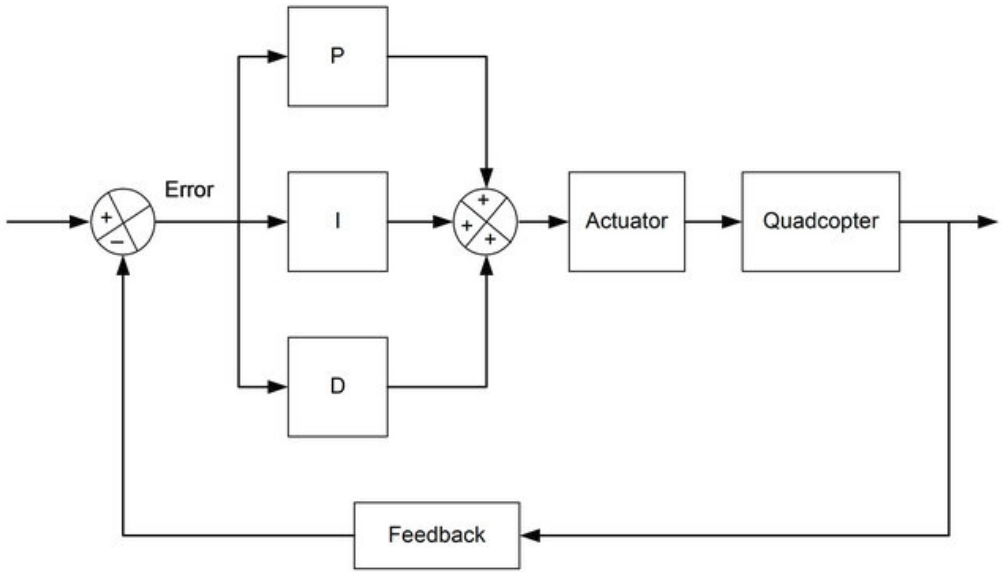

Figure 4.1 PID Controller Diagram

ROS

Robot Operating System(ROS) is a meta-operating system for a robot. It provides services that one would expect from an operating system, including hardware abstraction, device drivers, libraries, visualizers, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple

computers. It is named as a meta-operating system because it is something between an operating system and middleware. It provides not only standard operating system services (like hardware abstraction) but also high-level functionalities like asynchronous and synchronous calls, a centralized database, a robot configuration system, etc. ROS can be interpreted also as a software framework for robot software development, providing the operating system. ROS is based on a Unix-like philosophy of building many small tools that are designed to work together. ROS grows out of a collaboration between industry and academia and is a novel blend of professional software development practices and the latest research results.[6]

Pluto drone

Drone used for this experiment is pluto drone, this is light weight drone. It have the support for ROS.[7]

Figure 4.2 Pluto drone

V-REP

V-REP is simulation software which can be used with ROS to simulated various robots.[8]

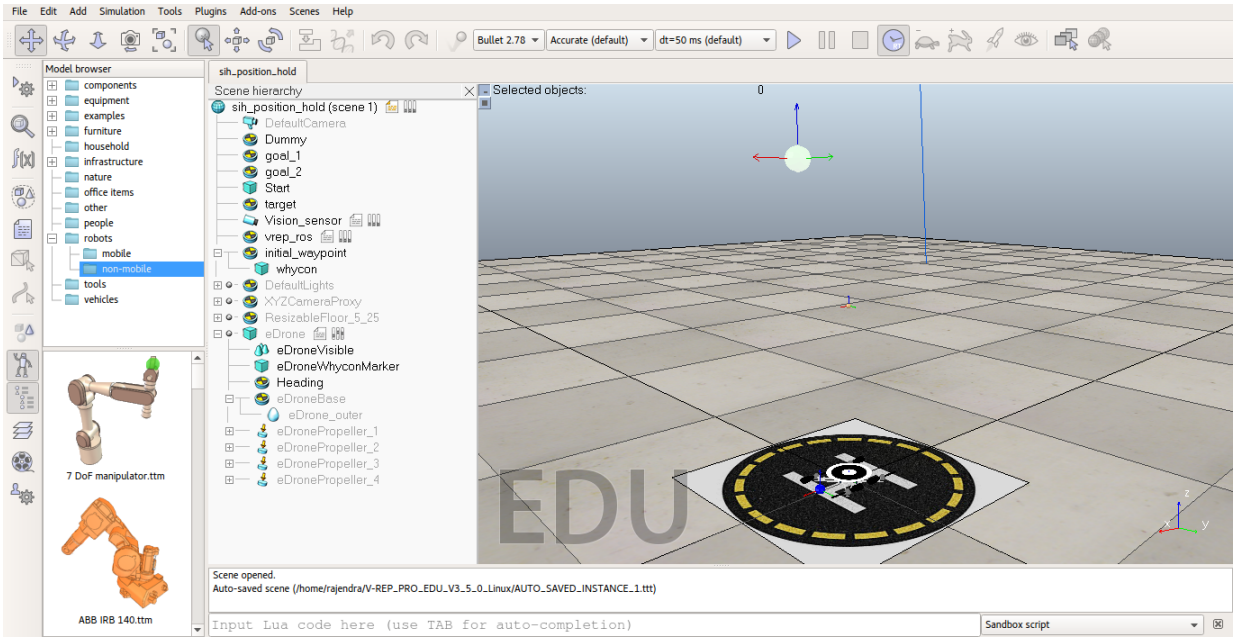

Figure 4.3 V-Rep simulation of pluto drone

4.1.2 Implementation

Here we will implement the position hold algorithm for drone using ROS in vrep software.

Algorithm

Lets describe the algorithm and pseudo code in brief. Full detailed code of the same can be found here. Here we uses the position of drone we get by detecting the whycon marker on the drone and publish the drone command to hold to the given position.

```
'''-------------------- PID algorithm here --------------'''
# Steps:
#   1. Compute error in each axis. eg: error[0] =
self.drone_position[0] - self.setpoint[0] ,where error[0] corresponds
to error in x...
#  2. Compute the error (for proportional), change in error (for
derivative) and sum of errors (for integral) in each axis.
```

```
#  3. Calculate the pid output required for each axis. For eg: calcuate
self.out_roll, self.out_pitch, etc.
#  4. Reduce or add this computed output value on the avg value ie
1500.EXPERIMENT AND FIND THE CORRECT SIGN
#  5. Don't run the pid continously. Run the pid only at the a sample
time. self.sampletime defined above is for this purpose.
#  6. Limit the output value and the final command value between the
maximum(1800) and minimum(1200)range before publishing.
#  7. Update previous errors.eg: self.prev_error[1] = error[1] where
index 1 corresponds to that of pitch
#  8. Add error_sum

'''--------------- PID algorithm code here---------------'''
def pid(self):
  error = [0.0, 0.0, 0.0, 0.0] #reset every time
  change_in_error = [0.0, 0.0, 0.0, 0.0] #reset every time
  for i in range (0, 4):
    error[i] = self.setpoint[i] - self.drone_position[i] #find error
    self.error_pub[i].publish(error[i]) #publish error
    change_in_error[i] = error[i] - self.prev_values[i] #change in error
    self.sum_of_error[i] = self.sum_of_error[i] + error[i] #sum of error
    self.iterm[i] = self.iterm[i] + (error[i] * self.Ki[i]) #integral
    term
    self.output[i] = (self.Kp[i] * error[i]) + self.iterm[i] +
    (self.Kd[i] * change_in_error[i]) #all together
    self.prev_values[i] = error[i]
```

```
# set drone command based on current output error
self.cmd.rcPitch = 1500 - self.output[0]
self.cmd.rcRoll = 1500 - self.output[1]
self.cmd.rcThrottle = 1500 - self.output[2]
self.cmd.rcYaw = 1500 + self.output[3]

self.setRange() #check if in range

self.command_pub.publish(self.cmd) #publish command to drone
```

4.1.3 Conclusion

I was able to hold the drone to given position but it required a lot and very precise pid tuning to get there. Hence, PID control based algorithm is easy and intuitive to understand but hard to improve and tune.

4.2 Reinforcement Learning

4.2.1 Introduction

Here we will try to implement the q learning based controller for drone in ros. Lets try to send drone from point A to B.

ARdrone

ARdrone most popular drone often used with the simulation using the ros.[9]

Figure 4.4 Ardrone

ROS development studio

I used the online ros development studio by **theconstruct** for simulating RL algorithm on the drone as my local system was not powerful enough to take heavy load.[10]

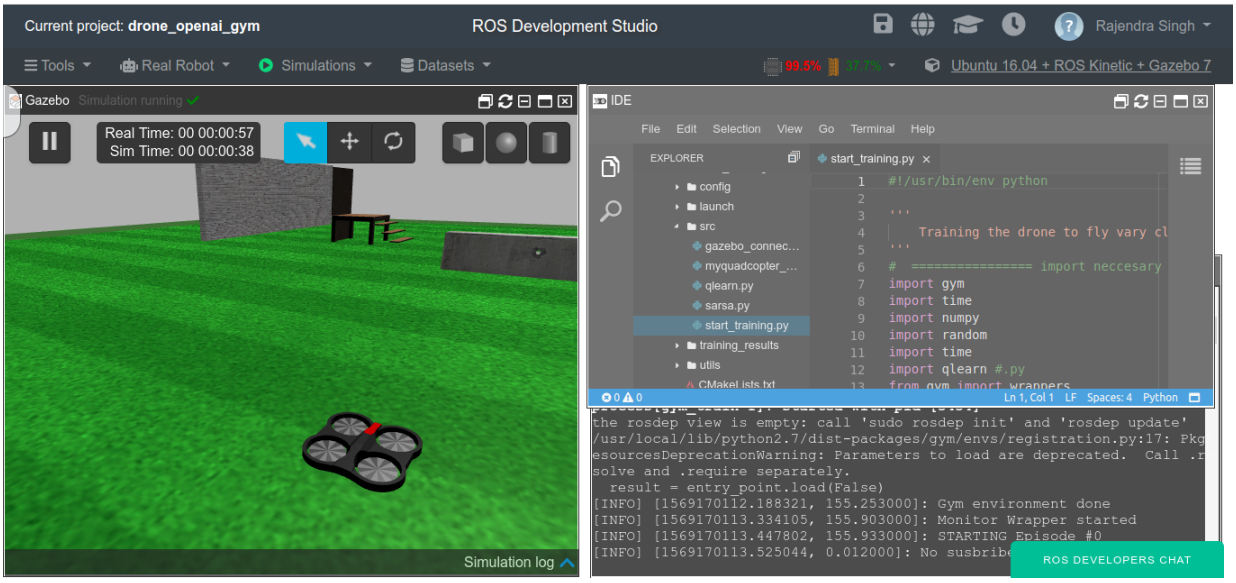

Figure 4.5 Simulation of ARdrone on ROS Development studio

4.2.2 Q-learning

Below I present the pseudo code of Q-learning algorithm on ARdrone to training it to fly from point A to point B. Full detailed code of the same can be found here.[11]

Pseudo Code

```
'''
    Training the drone to fly from point A to point B.
'''
#  ================ import necessary gym and ros libraries =============
#
if __name__ == '__main__':
    # ======= Create the Gym environment ===#
    # ============== Set the logging system ==================#
    # load param form yaml file
    # Initialises(class) the algorithm that we are going to use for
    learning
    # run for n episodes
        # ==== For n steps ====#
            action = qlearn.chooseAction(state) # Pick an action based on
            the current state
            observation, reward, done, info = env.step(action)# Execute
            the action in the environment and get feedback
            cumulated_reward += reward
            if highest_reward < cumulated_reward:#update the
            highest_reward
                highest_reward = cumulated_reward
```

```
            nextState = ''.join(map(str, observation)) #next state
            qlearn.learn(state, action, reward, nextState)# Make the
            algorithm learn based on the results
            if not(done):
                state = nextState
            else:
                break #done
    #publish ros log
    env.close() #close env
```

Note:

1.) In above code we import the myquadcopter-env.py, this is a openai gym environment made using ARdrone gazebo simulation and It is not written by me.

4.2.3 Sarsa

Implementing sarsa is also similar to q learning but here we will have different update function. Full detailed code of the same can be found here.

Pseudo Code

```
'''
    Here update rule also similar to that in q learning but a'(action to
be taken from nextstate s' in each step in any episode) is chosen based
epsilon greedy algorithm which is as  below s
'''
def chooseAction(self, state):
        if random.random() < self.epsilon:
            action = random.choice(self.actions)
```

```
else:
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)
    count = q.count(maxQ)
    if count > 1:
        best = [i for i in range(len(self.actions)) if q[i] ==
        maxQ]
        i = random.choice(best)
    else:
        i = q.index(maxQ)

    action = self.actions[i]
return action
```

4.2.4 Conclusion

In this chapter, we proposed another algorithm to control the drone, which is based on the Q learning and Sarsa. As this task not just required the good understanding of Q learning but also required very good understanding of ROS and gazebo, ARdrone simulation and ros message and topics. Hence, It was not possible for me to write whole code myself and therefore I used libraries for some part, e.g. myquadcopter-env.py as mentioned above. In future, I'll learn and try to write the env also myself.

Chapter 5

Swarm Intelligence (SI)

5.1 Motivation

In real world most of time we aren't simply working with just one robot. Indeed many a time, several robots have to cooperate to do some task. For the same reason we want to study the behaviour of robots when they will be working together. This is swarm behaviour. Lets understand it in more depth as below.

5.2 Introduction

Swarm intelligence is the collective behavior of decentralized, self-organized systems, natural or artificial. The concept is employed in work on artificial intelligence.[12] SI systems consist typically of a population of simple agents interacting locally with one another and with their environment. The inspiration often comes from nature, especially biological systems. The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, local, and to a certain degree random, interactions between such agents lead to the emergence of "intelligent" global behavior, unknown to the individual agents. Examples of swarm intelligence in natural systems include ant colonies, bird flocking, hawks hunting, animal herding, bacterial growth, fish

schooling and microbial intelligence.[13]

5.3 Example of swarm behaviour

5.3.1 Foraging

Foraging is searching for food resources. A group of birds/animals can make their search for food efficient by adopting some common behaviour(will explain below in detail). Their behaviour will help the group as whole sustain their life.[14]

5.3.2 Flocking

Flocking behavior is the behavior exhibited when a group of birds, called a flock, are foraging or in flight. It is when, several birds fly in group for their common advantage e.g. protection from predator.

5.3.3 Schooling

It is similar to flocking but for fishes.

5.3.4 Ant Colony

In ant colony, when group of ant found a food source then always tend to travel to/fro from it to their home in shortest path possible. This possible because secretes/put a chemical on the path when move through it. Eventually, shortest path tend to get more of this chemical and hence ant always follow the shortest path.

5.4 Algorithms

5.4.1 Particle swarm optimization (PSO)

PSO is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a

population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particles position and velocity. Each particles movement is influenced by its local best known position, but is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.[15]

A basic variant of the PSO algorithm works by having a population of particles. These particles are moved around in the search-space according to a few simple formulae. The movements of the particles are guided by their own best known position in the search-space as well as the entire swarms best known position. When improved positions are being discovered these will then come to guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered.[15]

Pseudo Code

```
'''
Let f: Rn -> R be the cost function which must be minimized. The
function takes a candidate solution as an argument in the form of a
vector of real numbers and produces a real number as output which
indicates the objective function value of the given candidate solution.
The gradient of f is not known. The goal is to find a solution a for
which f(a) <= f(b) for all b in the search-space, which would mean a is
the global minimum.
```

```
Let S be the number of particles in the swarm, each having a position xi
belongs to Rn in the search-space and a velocity vi belongs to Rn. Let pi
be the best known position of particle i and let g be the best known
position of the entire swarm. A basic PSO algorithm is then:
 '''


 for each particle i = 1, ..., S do
   Initialize the particles position with a uniformly distributed random
   vector(xi)
   Initialize the particles best known position to its initial
   position(pi)
   if f(pi) < f(g) then
       update the swarms best known  position(g)
   Initialize the particles velocity

while a termination criterion is not met do:
   for each particle i = 1, ..., S do
      for each dimension d = 1, ..., n do
         Update the particles velocity(vi)
      Update the particles position: xi <- xi + vi
      if f(xi) < f(pi) then
         Update the particles best known position: pi <- xi
         if f(pi) < f(g) then
            Update the swarms best known position: g <- pi
```

5.4.2 Flocking

In the natural world, organisms exhibit certain behaviors when traveling in groups. This phenomenon, also known as flocking, occurs at both microscopic scales (bacteria) and macroscopic scales (fish). Using computers, these patterns can be simulated by creating simple rules and combining them. This is known as emergent behavior, and can be used in games to simulate chaotic or life-like group movement.[16]

Basic models of flocking behavior are controlled by three simple rules:

1. **Separation** - avoid crowding neighbours (short range repulsion)
2. **Alignment** - steer towards average heading of neighbours
3. **Cohesion** - steer towards average position of neighbours (long range attraction)

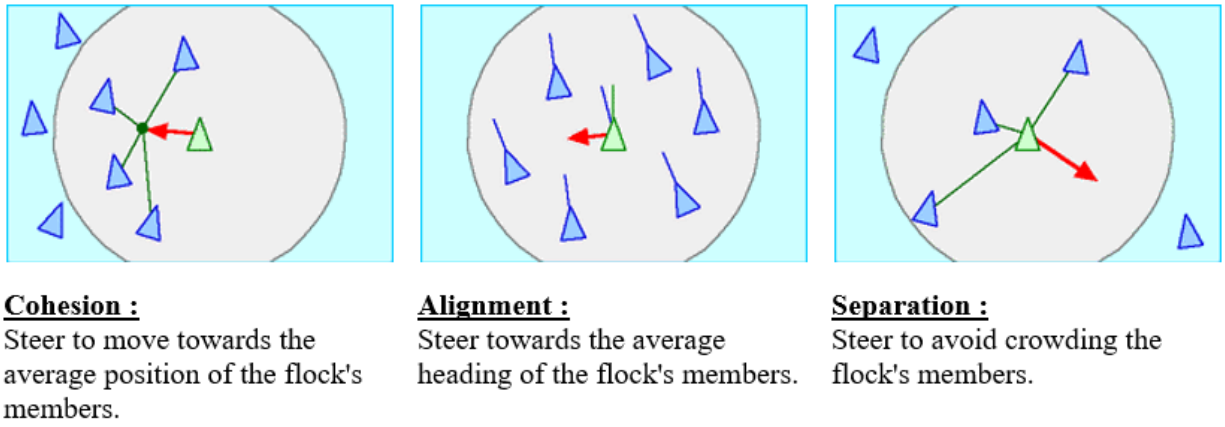

Figure 5.1 Flocking rules

With these three simple rules, the flock moves in an extremely realistic way, creating complex motion and interaction that would be extremely hard to create otherwise.

5.5 Implementation

5.5.1 Particle swarm optimization (PSO)

We use the PSO algorithm(described above) to solve some basis problem.

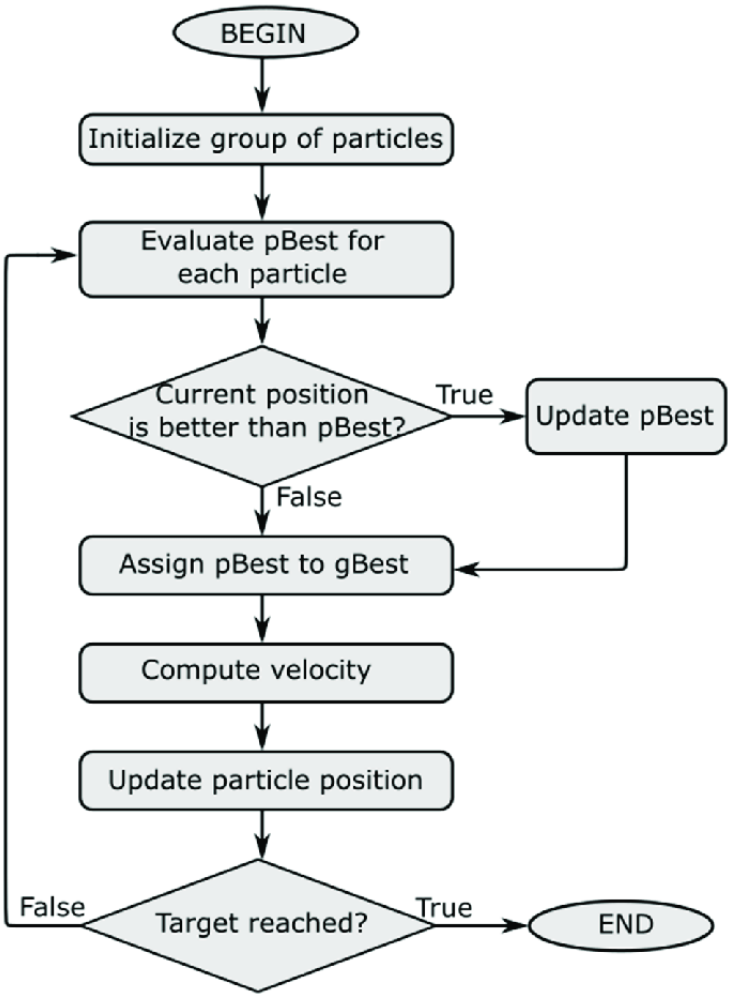

Figure 5.2 PSO flow chart

Finding the global minimum of function

Here I created python class for particle and environment. Multiple particle altogether tries to find the minimum value of the given function. Here function is

$$cost = x^n + y^n + 1$$

here x and y are coordinate of the particles. Hence origin with be the only minima. Particle finds this point by cooperating in best possible way by follow PSO algorithm. Detailed code of the same can be found here.

Pseudo code

```
Run for n iterations:
    # update personal and global best
    for ith particle:
        fitness_cadidate = fitness_function(particle_position_vector[i])

        # Personal update
        if(pbest_fitness_value[i] > fitness_cadidate):
            pbest_fitness_value[i] = fitness_cadidate
            pbest_position[i] = particle_position_vector[i]

        # global update
        if(gbest_fitness_value > fitness_cadidate):
            gbest_fitness_value = fitness_cadidate
            gbest_position = particle_position_vector[i]


    #break if required constrained met
    if(abs(gbest_fitness_value - target) < target_error):
        break


    # calculate new velocity and hence new position for each particle
```

```
for ith particle:
    # update velocity
    new_velocity = (W*velocity_vector[i]) + (c1*random.random()) *
    (pbest_position[i] - particle_position_vector[i]) +
    (c2*random.random()) *
    (gbest_position-particle_position_vector[i])

    # update position
    new_position = new_velocity + particle_position_vector[i]
    particle_position_vector[i] = new_position
iteration++
```

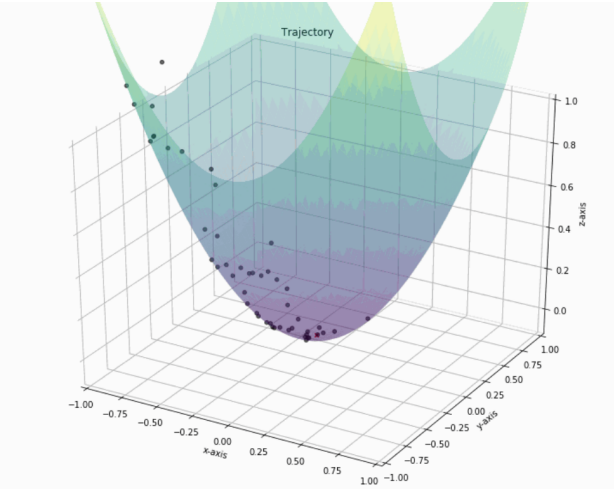

Figure 5.3 PSO : Random initialisation of particles

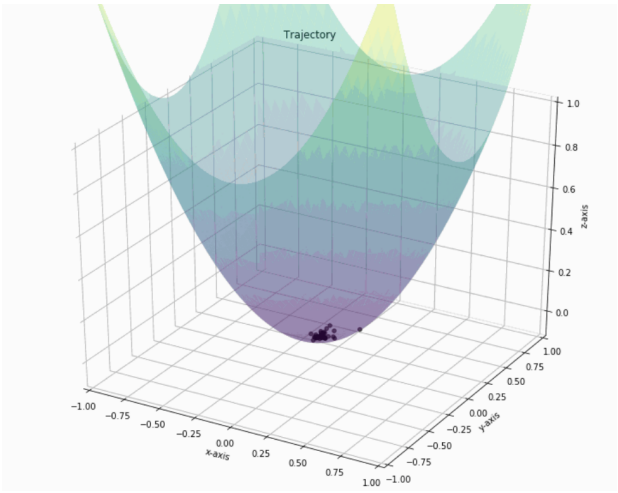

Figure 5.4 PSO : Particles converging to global minima

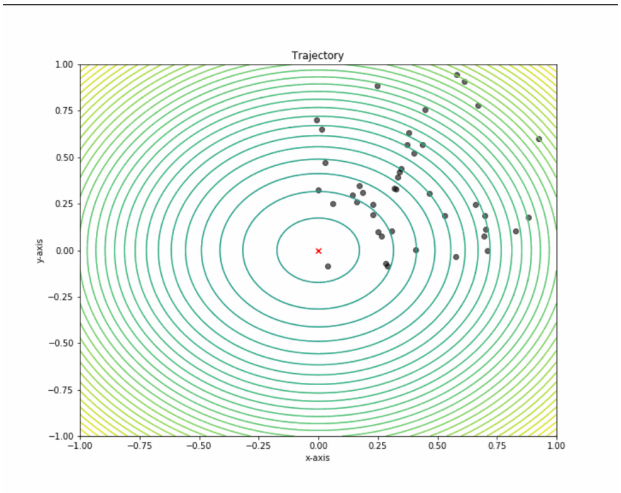

Figure 5.5 PSO : Random initialisation of particles(Top view)

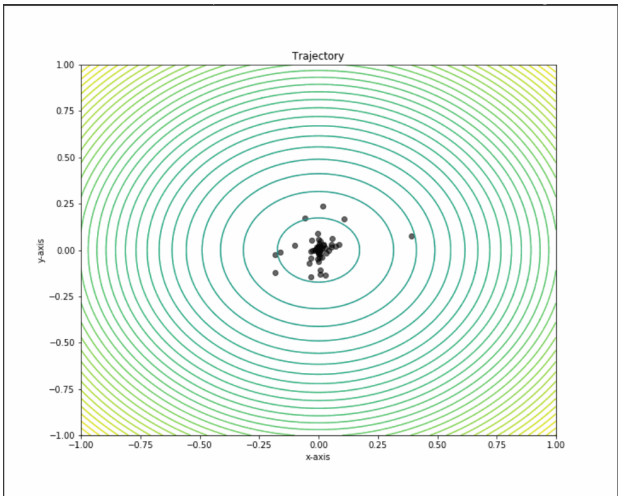

Figure 5.6 PSO : Particles converging to global minima(Top view)

Using PSO to solve Inverse Kinematics(IK) problem

I was surprised when I came to know that same PSO can be applied to even solve the Inverse Kinematics problem in robotic manipulator. In general we use DH method to solve IK.[17] Inverse Kinematics is the inverse function/algorithm of Forward Kinematics. The Forward Kinematics function/algorithm takes a target position as the input, and calculates the pose required for the end effector to reach the target position. While in inverse kinematics we find the joint value required for arm to reaches the given coordinate. I use PSO (pyswarms library) to solve the IK, and detailed code for the same can be found here.

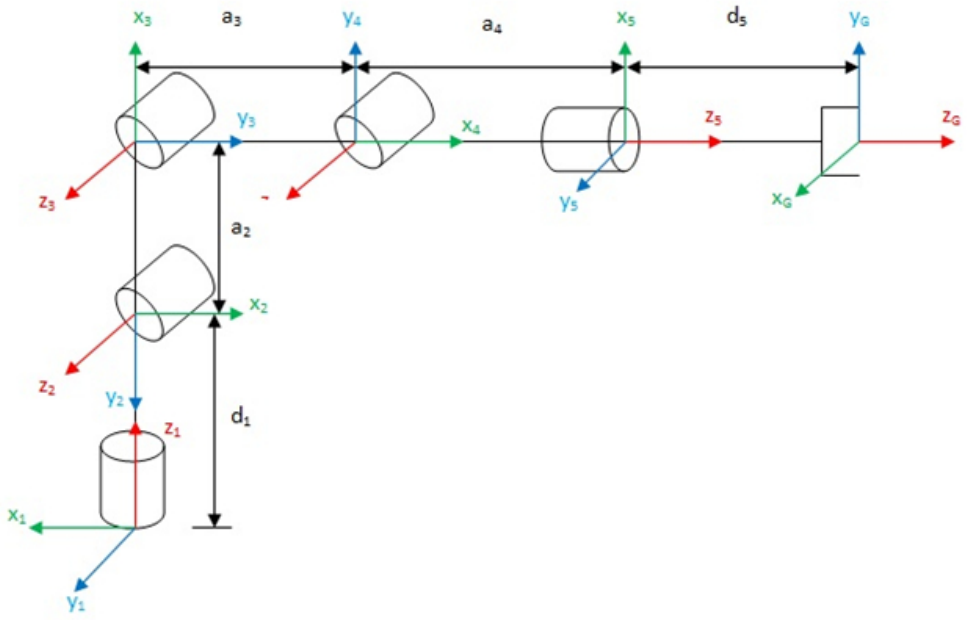

Figure 5.7 IK : Finding inverse kinematics solution for 6 dof robotics manipulator

Pseudo Code

```
# find forward kinematics pos of end effector
def get_end_tip_position(params):
    # Create the transformation matrices for the respective joints i.e
    find t_00, t_12, t_23, t_34, t_45,t_56

    # Get the overall transformation matrix
    end_tip_m =
    t_00.dot(t_01).dot(t_12).dot(t_23).dot(t_34).dot(t_45).dot(t_56)
    # The coordinates of the end tip are the 3 upper entries in the 4th
    column
    pos = np.array([end_tip_m[0,3],end_tip_m[1,3],end_tip_m[2,3]])
    return pos
```

```
# function to be optimized
def opt_func(X):
    n_particles = X.shape[0]  # number of particles
    target = np.array([-2,2,3])
    dist = [distance(get_end_tip_position(X[i]), target) for i in
    range(n_particles)]
    return np.array(dist)


# Call an instance of PSO
optimizer = ps.single.GlobalBestPSO(n_particles=swarm_size,
                                    dimensions=dim,
                                    options=options,
                                    bounds=constraints)


# Perform optimization
cost, joint_vars = optimizer.optimize(opt_func, iters=1000)
```

5.5.2 Flocking

I used the ARGOS simulator to simulate the flocking behaviour(as described above) of ground robots. ARGOS is a open source simulation software, mostly popular for swarm robotics.[18] Here I spawn several robots in environment. Somewhere in the environment there is source of light. Each of the robot have the sensor to sense the light intensity. Based on this, flock of this robots tries to move towards light source. The detailed code for the same can be found here and video of simulation can be found here.

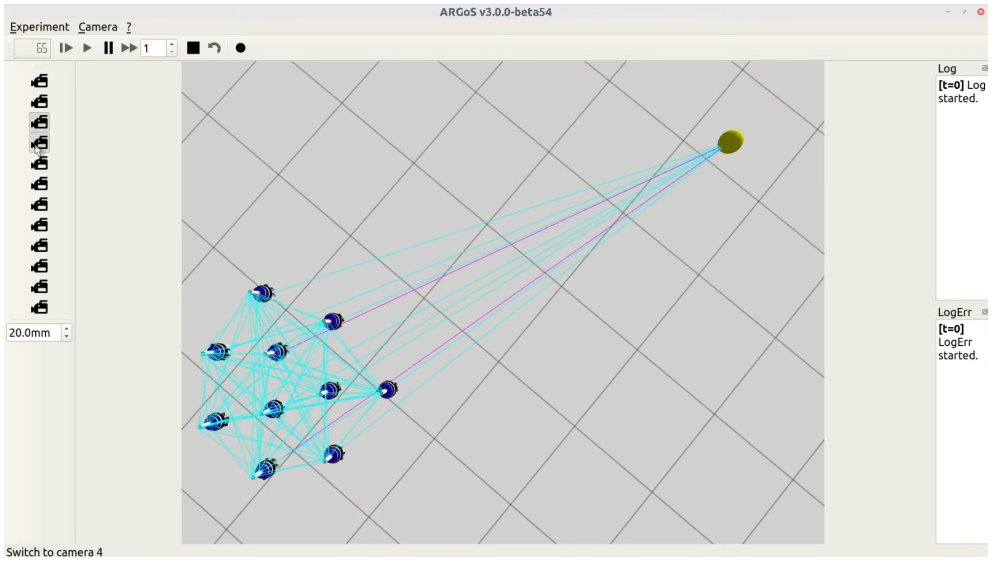

Figure 5.8 Robots started flocking towards light source

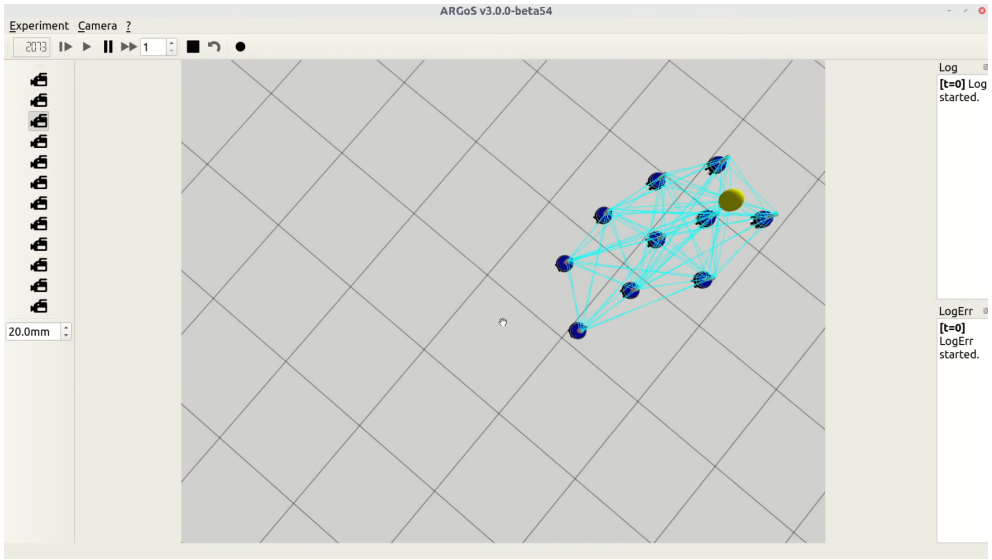

Figure 5.9 Robots flock almost reach light source

Pseudo code

```
/* Get the camera readings */
/* Go through the camera readings to calculate the flocking interaction
vector */
```

```
{
   for all blobs{

       /*
         * The camera perceives the light as a yellow blob
         * The robots have their red beacon on
         * So, consider only red blobs
         * In addition: consider only the closest neighbors, to avoid
         * attraction to the farthest ones. Taking 180% of the target
         * distance is a good rule of thumb.
         */

       if color red && dist<target {
           /*
            * Take the blob distance and angle
            * With the distance, calculate the Lennard-Jones interaction
force
            * Form a 2D vector with the interaction force and the angle
            * Sum such vector to the accumulator
            */
           /* Calculate LJ */
           /* Sum to accumulator */
           /* Increment the blobs seen counter */
       }
   }

   if(unBlobsSeen > 0) {
```

```
        /* Divide the accumulator by the number of blobs seen */
        /* Clamp the length of the vector to the max speed */
       /*Finally return flocking vecctor */
    }
}
```

5.5.3 Foraging

Similar to flocking here we spawn several robots in environment in their nest(in some part of environment). Environment also contain food at several place. Now robot have to leave their nest, go out, find the food and pick it and bring it back into nest. The detailed code for the same can be found here and video of simulation can be found here.

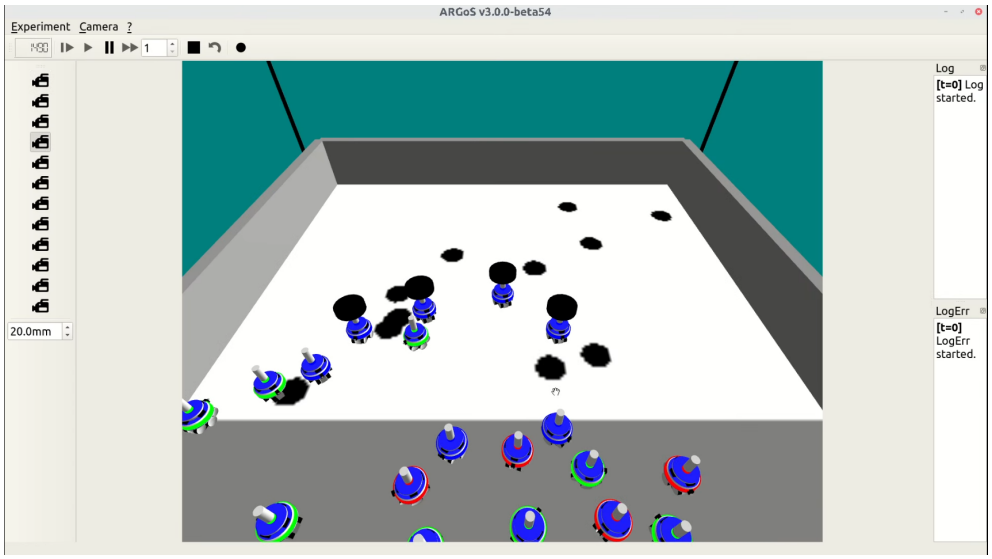

Figure 5.10 Robots foraging for food source

Pseudo code

```
// ======================== UpdateState() ========================//
   /* Reset state flags */
```

```
   /* Read stuff from the ground sensor */
   /*
    * You can say whether you are in the nest by checking the ground
sensor
    * placed close to the wheel motors. It returns a value between 0 and
1.
    * It is 1 when the robot is on a white area, it is 0 when the robot
    * is on a black area and it is around 0.5 when the robot is on a gray
    * area.
    * The foot-bot has 4 sensors like this, two in the front
    * (corresponding to readings 0 and 1) and two in the back
    * (corresponding to reading 2 and 3).  Here we want the back sensors
    * (readings 2 and 3) to tell us whether we are on gray: if so, the
    * robot is completely in the nest, otherwise it's outside.
    */

// ==================== CalculateVectorToLight() ==================//
   /* Get readings from light sensor */
   /* Sum them together */
   /* If the light was perceived, return the vector */
   /* Otherwise, return zero */

// ==================== DiffusionVector() ====================//
   /* Get readings from proximity sensor */
   /* Sum them together */
   /* If the angle of the vector is small enough and the closest obstacle
```

```
      is far enough, ignore the vector and go straight, otherwise return
it */
// ==================== SetWheelSpeedsFromVector() ==================//
   /* Get the heading angle */
   /* Get the length of the heading vector */
   /* Clamp the speed so that it's not greater than MaxSpeed */
   /* State transition logic */
   /* Wheel speeds based on current turning state */
   switch{
      case /* Just go straight */
      }
      case /* Both wheels go straight, but one is faster than the other
      */
      }
      case /* Opposite wheel speeds */
      }
   }
   /* Apply the calculated speeds to the appropriate wheels */
   /* Finally, set the wheel speeds */


// ==================== Explore() ==================//
/* We switch to 'return to nest' in two situations:
    * 1. if we have a food item
    * 2. if we have not found a food item for some time;
    *    in this case, the switch is probabilistic
    */
```

```
/*
 * Test the first condition: have we found a food item?
 * NOTE: the food data is updated by the loop functions, so
 * here we just need to read it
 */
   /* Apply the food rule, decreasing ExploreToRestProb and increasing
    * RestToExploreProb */
   /* Store the result of the expedition */
   /* Switch to 'return to nest' */
}
/* Test the second condition: we probabilistically switch to 'return
to
 * nest' if we have been wandering for some time and found nothing */
      /* Store the result of the expedition */
      /* Switch to 'return to nest' */
   }
   else {
      /* Apply the food rule, increasing ExploreToRestProb and
       * decreasing RestToExploreProb */
   }
}
/* So, do we return to the nest now? */
if{
   /* Yes, we do! */
}
else {
   /* No, perform the actual exploration */
```

```
      /* Get the diffusion vector to perform obstacle avoidance */
      /* Apply the collision rule, if a collision avoidance happened */
         /* Collision avoidance happened, increase ExploreToRestProb and
          * decrease RestToExploreProb */
     /*
       * If we are in the nest, we combine antiphototaxis with obstacle
       * avoidance
       * Outside the nest, we just use the diffusion vector
       */
         /*
          * The vector returned by CalculateVectorToLight() points to
          * the light. Thus, the minus sign is because we want to go away
          * from the light.
          */
      }
      else {
         /* Use the diffusion vector only */


// ==================== ReturnToNest() ==================//
   /* As soon as you get to the nest, switch to 'resting' */
   /* Are we in the nest? */
     if{
      /* Have we looked for a place long enough? */
         /* Yes, stop the wheels... */
         /* Tell people about the last exploration attempt */
         /* ... and switch to state 'resting' */
```

```
        }
        else {
            /* No, keep looking */
        }
    }
    else {
        /* Still outside the nest */
    }
    /* Keep going */
// ==================== END ==================//
```

5.5.4 Synchronisation in the drone

Here I want constrain two drones to fly place A to B in synchronisation. For this I use VREP and ROS. I spawn to two pluto drone in the VREP world. I used markers for localization of checkpoints of the path, obstacles and on the drones to locate them in the frame of an overhead camera which is watching this whole setup from the top. We used Aruco markers with different ids for obstacles and checkpoints of path. We also placed whycon markers on the drones to detect their position. Since the overhead camera is able to locate each of the markers in its frame of reference, both drones can now navigate on the predefined path synchronously while avoiding obstacles. PID control algorithm is used for drones to navigate between any two coordinates. We have to find such a path which can avoid obstacles and also navigate both drones maintaining their relative distance and attitude. For the same, I used Open Motion Planning Library(OMPL) on ROS and simulated the same in VREP. Here, we grouped both the drones and considered them as an object where only drones and obstacles were set as collidable. I use the OMPL algorithm to plan a path for midpoint of the group avoiding obstacles. Now, to get the individual path for both drones, firstly divided the midpoint path in some point called way-points.

Now we found the initial offset of both drones from the mid-point in vector form, added it to each coordinate of midpoint path way-point. So, this way we are able to find the two separate paths for both the drones where distance between the drones is being maintained. So the only thing to be done now is to run both the drones on their corresponding path synchronously. We will give the way-points to the drone simultaneously. Suppose we have given the first pair of corresponding way-points to the drones. The next pair of way-points will be given only after both the drones reach their corresponding previous way-points properly. A point is said to be properly reached only if its error in all three axes is less than some given constant value.

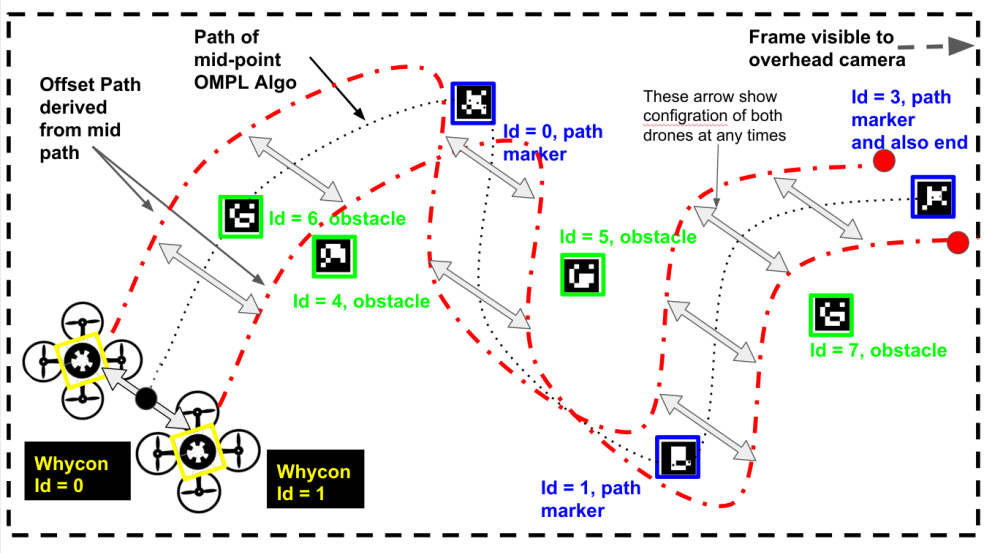

Figure 5.11 Algorithm flowchart for pluto drone synchronisation

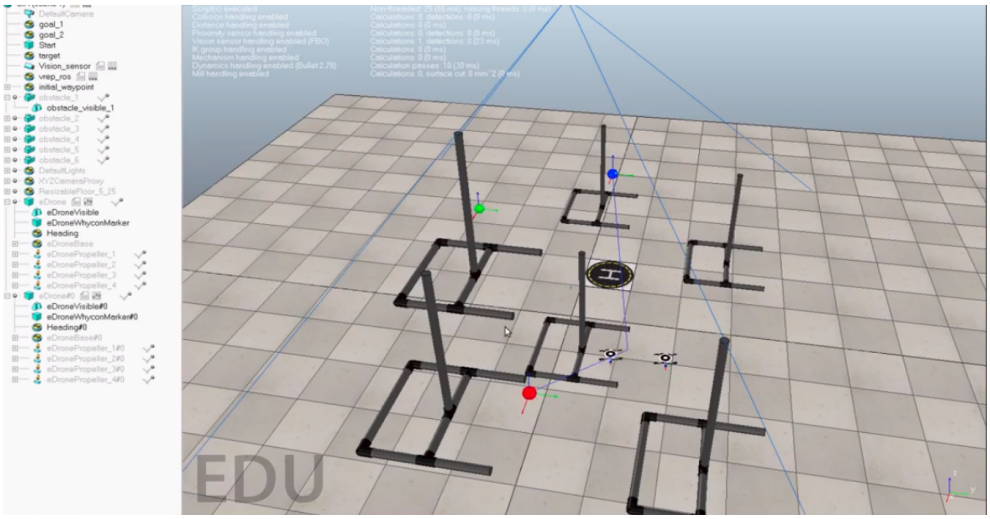

Figure 5.12 VREP Simulation of above Algorithm

5.5.5 Pseudo Code

Once we get the list of set point(using above algorithm) for both the drone rest code very much similar to one used in section 4.1.2.

```
'''
Code only here as below
'''
if __name__ == '__main__':
    pid_class = PID() #pid class as described in previous pid controller
    code
    error = []
    while True:
        error = pid_class.pid(pid_class.initialSetPoint)
        if (abs(error[0]) <= 0.5 and abs(error[1]) <= 0.5 and
        abs(error[2]) <= 0.5):
            break
    for i in range (0, 3):
```

```
    if i != 0:
        print("next_target")
    pid_class.next.publish('1')
    rospy.sleep(0.5)
    length = len(pid_class.setPoint.poses)
    for j in range (0, length):
        temp = [0.0, 0.0, 0.0, 0.0]
        temp[0] = pid_class.setPoint.poses[j].position.x
        temp[1] = pid_class.setPoint.poses[j].position.y
        temp[2] = pid_class.setPoint.poses[j].position.z
        temp[3] = 0.0
        while (True):
            error = pid_class.pid(temp)
            if (error[0] <= 0.5 and error[1] <= 0.5 and error[2] <=
            0.5):
                break
# pid_class.disarm()

#land at start point
while True:
    error = pid_class.pid(pid_class.start)
    if (abs(error[0]) <= 0.5 and abs(error[1]) <= 0.5 and
    abs(error[2]) <= 1):
        pid_class.disarm()
        break
```

5.6 Conclusion

This chapter we studied swarm behaviour of robot in depth and how it can help a group as whole to benefit from each other. We also implemented flocking, foraging and pso in code.

Chapter 6

Conclusion and Future Work

In this report, we successfully learnt about SLAM and Swarm Intelligence. We begin with understand various slam method. Then, we implement various RL algorithm(Q-learning, Sarsa, DQN) to solve the mountain car problem to understand these algorithm in depth. Later, we also used some of these algorithms to control the quad-copter. We show that how these method ease the process control compered to conventional method of pid tuning.
In the later part, studied various swarm algorithm and shown that how multiple robot together can perform most of the task better than any of individual robot in group can.

It seems we have build sufficient theoretically background and also have tested and varified our algorithm on simulated robots. Its high time we start implementing algorithm on hardware and further improve them. Hence in future, I will be working to implement this algorithms on hardware.

Bibliography

[1] "Btp github repo." [Online]. Available: https://github.com/iamrajee/Slam_and_RL_BTP

[2] R. Singh, "Ust global internship report," 2019. [Online]. Available: https://drive.google.com/file/d/1RTBjH5_nVITV9RjlLy2PGjsJbRWrLrKR/view?usp=sharing

[3] "Openai gym." [Online]. Available: https://gym.openai.com/

[4] "Pid controller." [Online]. Available: https://en.wikipedia.org/wiki/PIDcontroller

[5] "The art of quadcopter pid tuning." [Online]. Available: https://oscarliang.com/quadcopter-pid-explained-tuning/

[6] "Robot operating system." [Online]. Available: http://wiki.ros.org/

[7] "Pluto drone ros." [Online]. Available: http://wiki.ros.org/pluto_drone

[8] "V-rep simulation." [Online]. Available: http://www.coppeliarobotics.com/

[9] "Ardrone simulation." [Online]. Available: https://github.com/AutonomyLab/ardrone_autonomy

[10] "Ros development studio." [Online]. Available: https://rds.theconstructsim.com/

[11] "My rosject." [Online]. Available: http://www.rosject.io/l/ca1685c/

[12] "Swarm intelligence." [Online]. Available: https://en.wikipedia.org/wiki/Swarm_intelligence

[13] "Swarm behaviour." [Online]. Available: https://en.wikipedia.org/wiki/Swarm_behaviour

[14] K. Roy E. Plotnick, "Theoretical and experimental ichnology of mobile foraging," 2007.

[15] "Particle swarm optimization." [Online]. Available: https://en.wikipedia.org/wiki/Particle_swarm_optimization

[16] "Flocking behaviour." [Online]. Available: https://en.wikipedia.org/wiki/Flocking_(behavior)

[17] "Denavit–hartenberg parameters." [Online]. Available: https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

[18] "Argos simulation." [Online]. Available: https://www.argos-sim.info/