popular cartpole gym environment [8]. Furthermore, the two arguments of the class constructor correspond to the minimum and maximum values respectively.

Step Function

The required functionality of the `step` function was implemented next. Firstly, in order to take the desired joint incrementing actions using the action server decided upon in section 5.4.3, the ROS `actionlib` was used. This was decided due to the fact that ROS actions are much more robust than just using topics to command positions. Actions give the user the ability to cancel trajectories on command and wait for successful completion of trajectories. Additionally, when commanding joint trajectories in quick succession using topics, the system can sometimes exhibit undesired behaviour such as freezing. To implement the action client for moving the joints a `move` function was created, which takes in a desired joint position as a goal:

```
self.action_server_proxy =
↪  actionlib.SimpleActionClient('arm/arm_controller/follow_joint_trajectory',
↪  FollowJointTrajectoryAction)
self.action_server_proxy.wait_for_server()

def move(self, pos):
        msg = FollowJointTrajectoryActionGoal()
        msg.goal.trajectory.joint_names = self.joint_names
        point = JointTrajectoryPoint()
        point.positions = pos
        point.time_from_start = rospy.Duration(1.0/60.0)
        msg.goal.trajectory.points.append(point)
        self.action_server_proxy.send_goal(msg.goal)
```

Listing 3: ROS action based joint command implementation

In order to store the current commanded joint positions an instance variable was used. This instance variable is updated by first adding the desired action to the elements of the actual joint angles, and then clipping it to the allowable joint angles using the numpy `clip` function (to avoid robot malfunction).

```
self.joint_pos = np.clip(self.joint_pos+action,
                         ↪  a_min=self.joint_pos_low,a_max=self.joint_pos_high)
```

Listing 4: Implementation of joint position clipping for actions

Once the clipping has been performed, the new desired joint angles are sent to the action server via the aforementioned `move` function. Following this, a delay is called in order to wait for the arm to reach its new position and stop. This is done to improve the state's accuracy such that it can be considered Markov as described in section 3.1.1. This delay was called using the `rospy.sleep(Duration)` function from the `rospy` ROS client library.

Once the arm has reached its new position, the updated state information is received. In order to abstract this process, another function, `get_state`, was created which returns the current joint angles, a timeout boolean and an arrived boolean.

The joint angles are received via subscribing to the `'arm/arm_controller/state'` topic and setting a joint state instance variable. This can be done by defining a `rospy` subscriber and then creating a callback function:

```
self.joint_state_subscriber =
↪  rospy.Subscriber('arm/arm_controller/state',
↪  JointTrajectoryControllerState, self.joint_state_subscriber_callback,
↪  queue_size=1)
 def joint_state_subscriber_callback(self, joint_state):
        self.joint_state = np.array(joint_state.actual.positions)
```

Listing 5: `rospy` subscriber setup for joint states

Note the use of the queue_size argument which limits the subscriber to only sending one (the latest) value to the callback function, as opposed to a 2D array of values. The timeout boolean was calculated by accessing a `sim_time` instance variable that was set by subscribing to the `'/clock'` topic and following the same basic process as mentioned above.

Following the reception of the updated state values, a function is called which returns the reward received as a result of the last action. The reward function includes all reward

shaping, and is given as follows:

```
def get_reward(self, time_runout, arrive):
        reward = -1*self.distance_reward_coeff*self.get_goal_distance()
        if(self.hit_floor):
            reward = reward - 50.0
        if(time_runout and not arrive):
            reward = reward - 25.0
        if(arrive):
            print("Arrived at goal")
            reward += 25.0
        return reward
```

Listing 6: `get_reward()` function implementation

As the reward is the only way that an RL agent has any knowledge of what is considered good or bad behaviour, it is necessary to define any goal states here. As can be seen in the above function, the reward that it receives regardless of if it has neither hit the floor nor reached the goal is equal to the negative of the distance from the goal at the current time, multiplied by some positive coefficient used for scaling purposes. Finally, the agent is penalised for hitting the floor in order to discourage this potentially dangerous behaviour, and is rewarded for reaching the goal. Many other reward shaping methods were used throughout the implementation process, and the final reward shape was found after iterating through them.

As mentioned above, the distance to the goal needs to be found in order to calculate the reward. This distance was calculated as the euclidean distance between the arm's end-effector, and the goal's center. As mentioned in section 5.4.3, ROS's tf2 was used to find the position of the end-effector in the world frame. The implementation of tf as well as the goal distance function is shown below:

```
self.tf_buffer = tf2_ros.Buffer()
self.tf_listener = tf2_ros.TransformListener(self.tf_buffer)
def get_goal_distance(self):
        trans = self.tf_buffer.lookup_transform('world', 'dummy_eef',
        ↪  rospy.Time())
        x = trans.transform.translation.x
        y = trans.transform.translation.y
        z = trans.transform.translation.z
        trans = np.array([x,y,z])
        self.eef_pos = trans
        goal_distance = distance.euclidean(trans,self.goal_pos)
        return goal_distance
```

Listing 7: tf listener setup and get_goal_distance() function implementation

Exception handling was also implemented to account for possible lookup errors, but is not included here. Additionally, the `distance` function from the popular `scipy` library was used to calculate the euclidean distance. A test program to verify the functionality of the `get_goal_distance()` function was created and its output can be seen below:

```
(RL_arm_noetic) devon@devon-Aspire-VX5-591G:~/RL_Arm_noetic
↪  /home/devon/miniconda3/envs/RL_arm_noetic/bin/python
↪  /home/devon/RL_Arm_noetic/src/arm_bringup/scripts/config_test.py
[INFO] [1605095527.165041, 7.019000]: Defining a goal position...
[INFO] [1605095527.166402, 7.021000]: Goal position defined
[INFO] [1605095528.443261, 0.317000]: Defining a goal position...
[WARN] [1605095528.684392, 0.558000]: Detected jump back in time of
↪  6.738000s. Clearing TF buffer.
Goal Distance:  0.40027990206854
Enter an action:1 1 1 1
Goal Distance:  0.3820581396102969
```

Listing 8: `get_goal_distance()` implementation test

Finally, the state is returned from the step function as a concatenation of the previously received joint angles and an instance variable containing the current goal position. The reward float as well as the done boolean are also returned, as per the requirements given