# Flytbase – Robotics Engineer Assignment Documentation

Pranav Shirgur  
Email: [pranav.shirgur@gmail.com](mailto:pranav.shirgur@gmail.com)  
Ph: +91 9769920710

## Introduction

This document describes the approach each programming problem provided, highlights assumptions, reasoning and tools used to come up with the most equitable solution to each problem. Goal from one to three were solved to highly competent execution standards, drawbacks for these as well as from goal four to six have been mentioned as well. **Video comparison and documentation have been embedded into this document itself in case they aren’t visible please check the zip file provided with the subsequent email.**

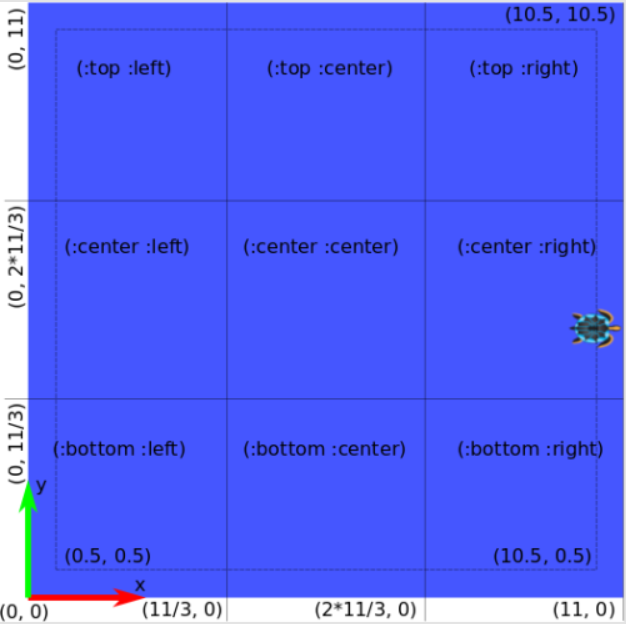
## General Information

Initially the environment in which ROS was being configured was Ubuntu Jammy – 23.04, this version of Ubuntu has had several installation and dependency conflicts with ROS these issues have been documented since [May 2022](https://github.com/ros2/ros2/issues/1287) and [June 2022](https://github.com/ros2/ros2/issues/1287).

Hence, ROS noetic was used with a fresh install of Ubuntu Focal - 20.04.6, all packages were compiled manually with catkin with equitable package.json and CMakeLists.text configs.   
These implementations were done parallel to learning how to program using roscpp. My basic operational knowledge of computer science and embedded systems propelled me to these solutions below, hence the ROS documentation was used extensively.

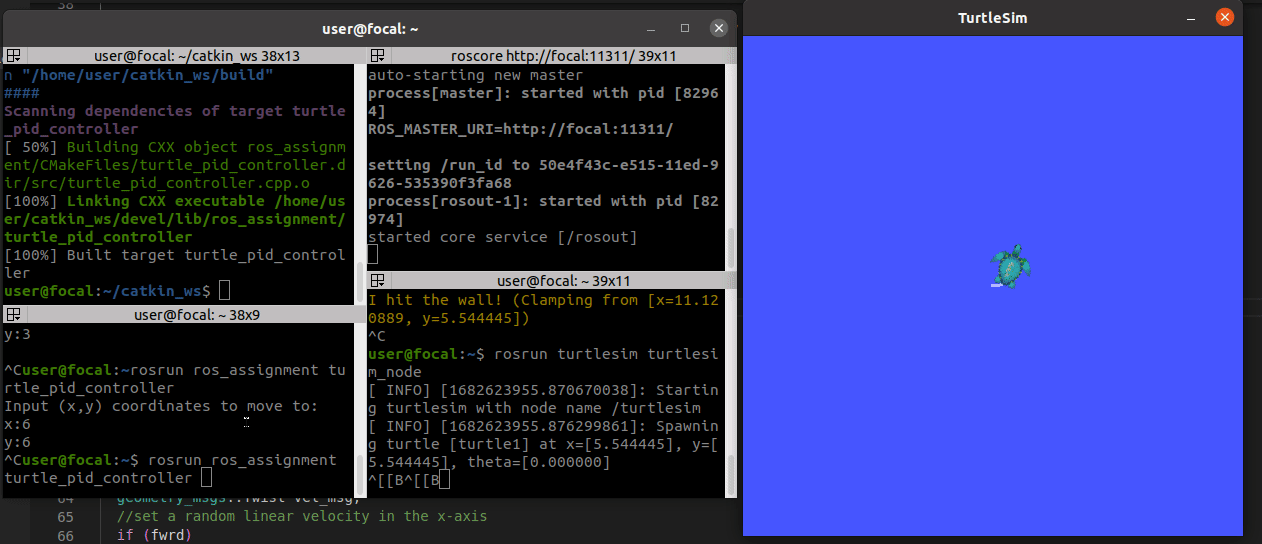
## Goal 1: Control Turtle

Here a turtle must spawn at a random location and must move specified coordinates by leveraging a PID controller and minimize overshooting. Coordinates here are provided by the user. The coordinate mapping for turtlesim ranges form (0,0) on the bottom left of the screen to (11,11) on the top right.



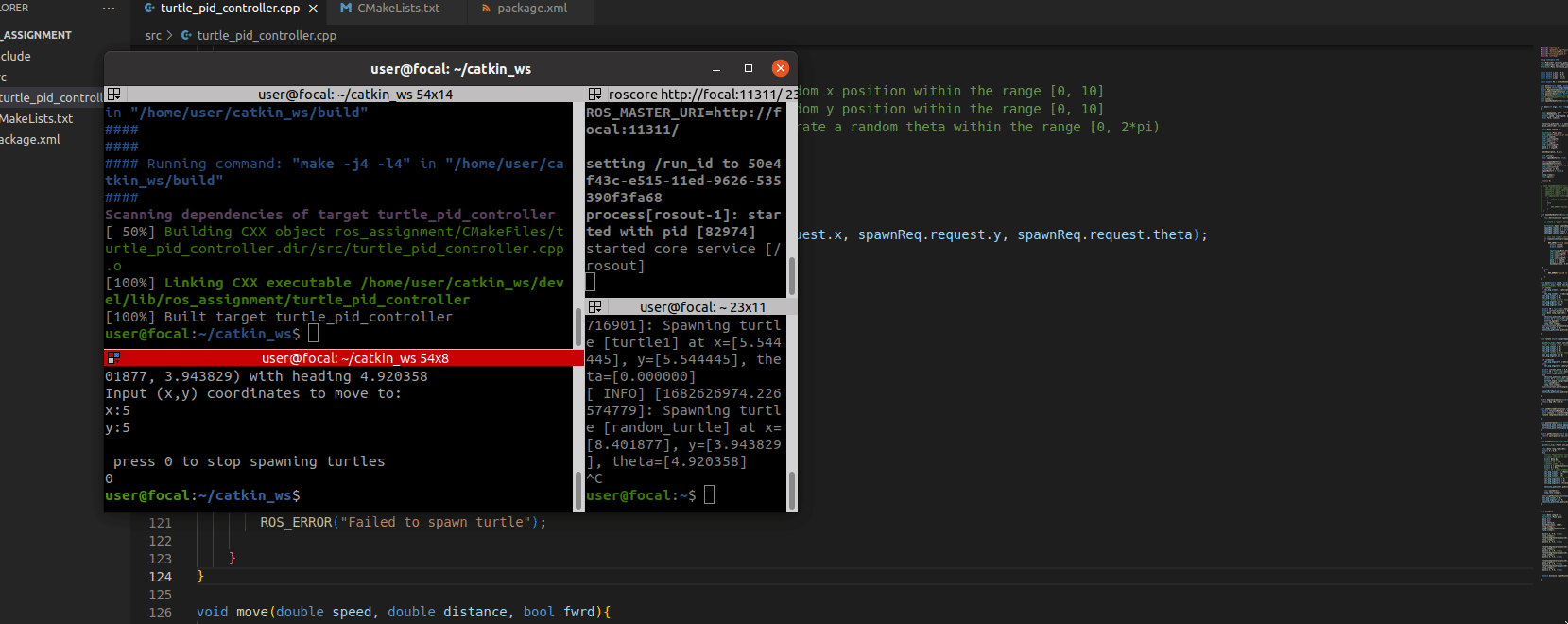
*2D Cartesian Coordinates for turtlesim*

Here is the execution for moving the default turtle to user specified coordinates.



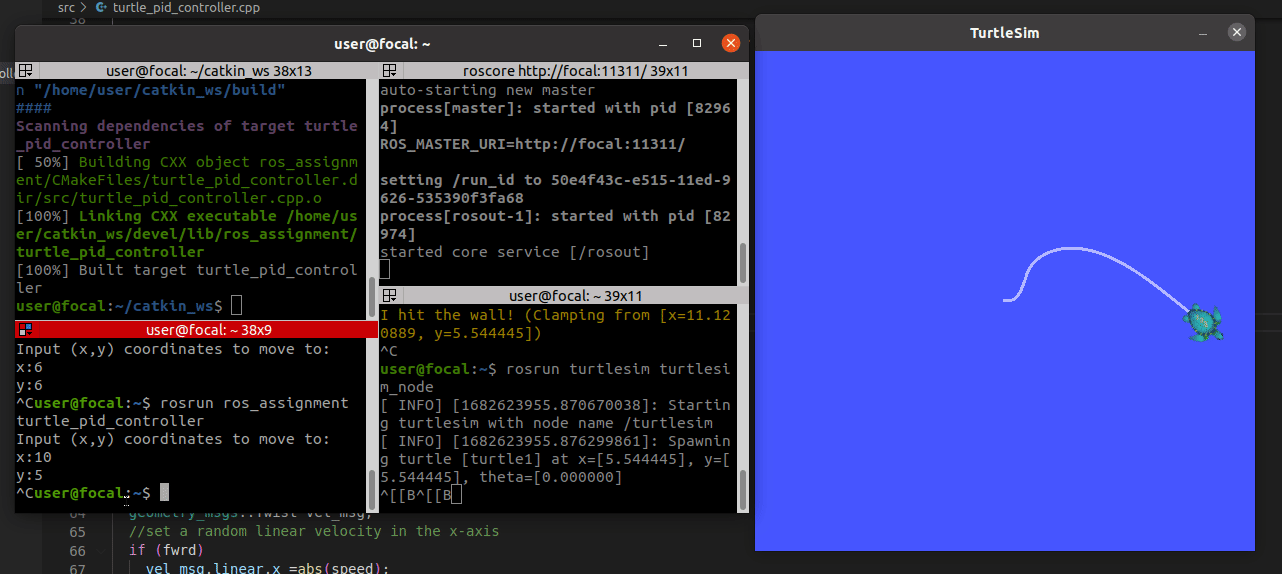
*Moving turtle to (10,5) by providing user input*

Now this implementation isn’t without limitations, the program must be run before executing the turtlesim node, this will prevent conflict between moving the default turtle and a newly randomly generated one, refer to *General Information* section of this documentation to understand how the code interacts and compiles with a subsequent Linux environment.



*Spawning a turtle at random pos, however movement is locked to the original*

Another limitation is that handling for wall collision isn’t provided, the default clamping isn’t optimal for real world executions.



*Turtle hits the wall, and values are auto-clamped by the turtlesim node*

While attempting to plot performance comparisons for various gain inputs, Rqt Multiplot would throw installation dependency conflicts with the build environment. To get around this angular velocity profiles were printed out to the console and those values were graphed against execution time, these implementations are illustrated in the goal 2 description.

### Understanding the moveGoal function

This function is responsible for moving the turtle to a specific inputted coordinate. This is a PD controller and not a PID controller – it has no integral component, there were some issues with executing a PID controller causing the turtle to move erratically. Here’s how the function works:

The function takes in two arguments: the goalPose, which is a turtlesim::Pose object representing the x and y coordinates and the orientation of the turtle's goal location, and the distTolerance, which is a double representing the distance tolerance within which the turtle is considered to have reached its goal.

It starts by defining a geometry\_msgs::Twist object, which is used to control the linear and angular velocities of the turtle. It then sets up a loop that continues until the turtle has reached its goal within the specified distance tolerance. Within the loop, the function calculates the current distance between the turtle's position (stored in a turtlesim::Pose object called turtlesim\_pose) and the goalPose using the getDistance() function. **The getDistance() function is a regular Euclidean distance calculating function.**

The PD controller is implemented using two constants, Kp and Ki, which control the proportional and integral components of the controller. The proportional component is calculated by multiplying the distance error (e) by the proportional constant Kp. The derivative component is calculated by computing the difference between the current and previous distance errors (Integral not implemented) and multiplying it by the derivative constant Kd.

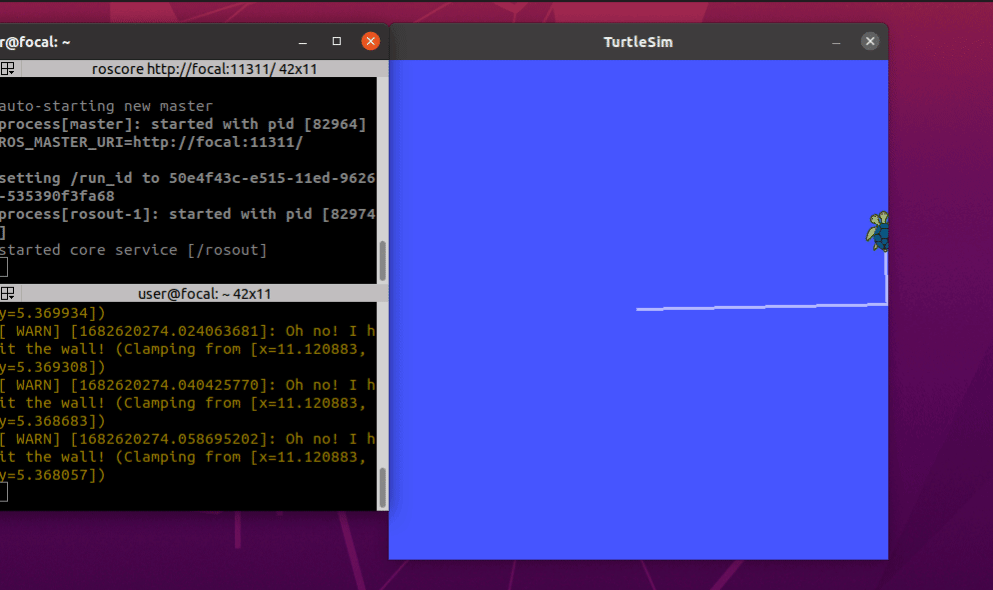
The function then calculates the angular velocity of the turtle using the atan2() function, which computes the angle between the turtle's current position and its goal location. This angle is then used to set the angular velocity of the turtle.

Finally, the function publishes the computed linear and angular velocities to the turtle using a ROS velocity\_publisher object. Once the turtle reaches its goal, the function sets the linear and angular velocities to zero and publishes them to stop the turtle.

## Goal 2: Make a Grid

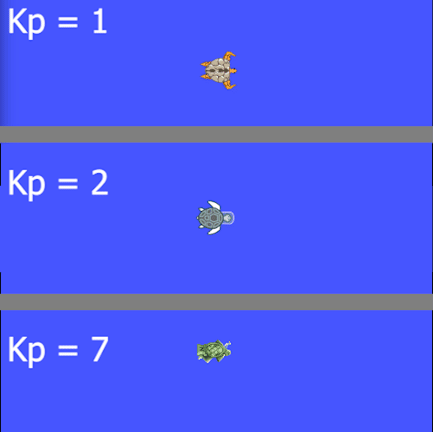
Here we must program acceleration and deceleration profiles to our turtle, this is done by manipulating the gains on the PD controller, namely Kp.

This builds on the already existing turtle described above, hence the initial functionality will not be compromised – the user will still give coordinate inputs for the turtle to move after which a sweep () function is called to draw the grid.



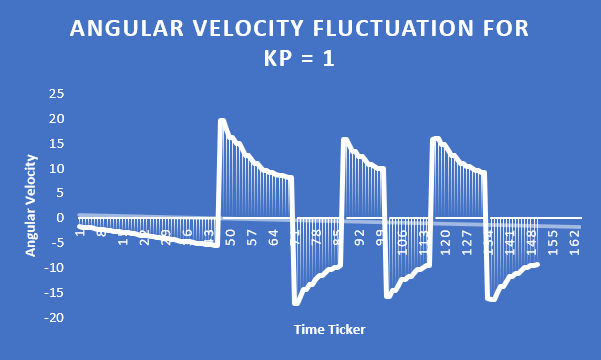
*Turtle tracing the desired grid path*

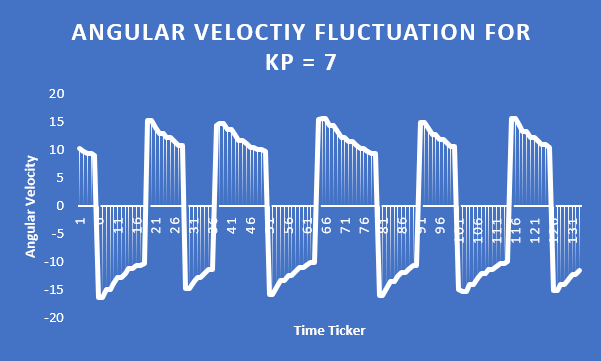
As expected, the turtle stops abruptly see how by adding a deceleration profile for varying values of Kp the movement of the turtle’s change:



*Varying Acceleration and Deceleration*

Let us now compare how the angular velocity of the turtles vary with an increase in Kp:





*Graphs displaying angular velocity fluctuations for different Kp*

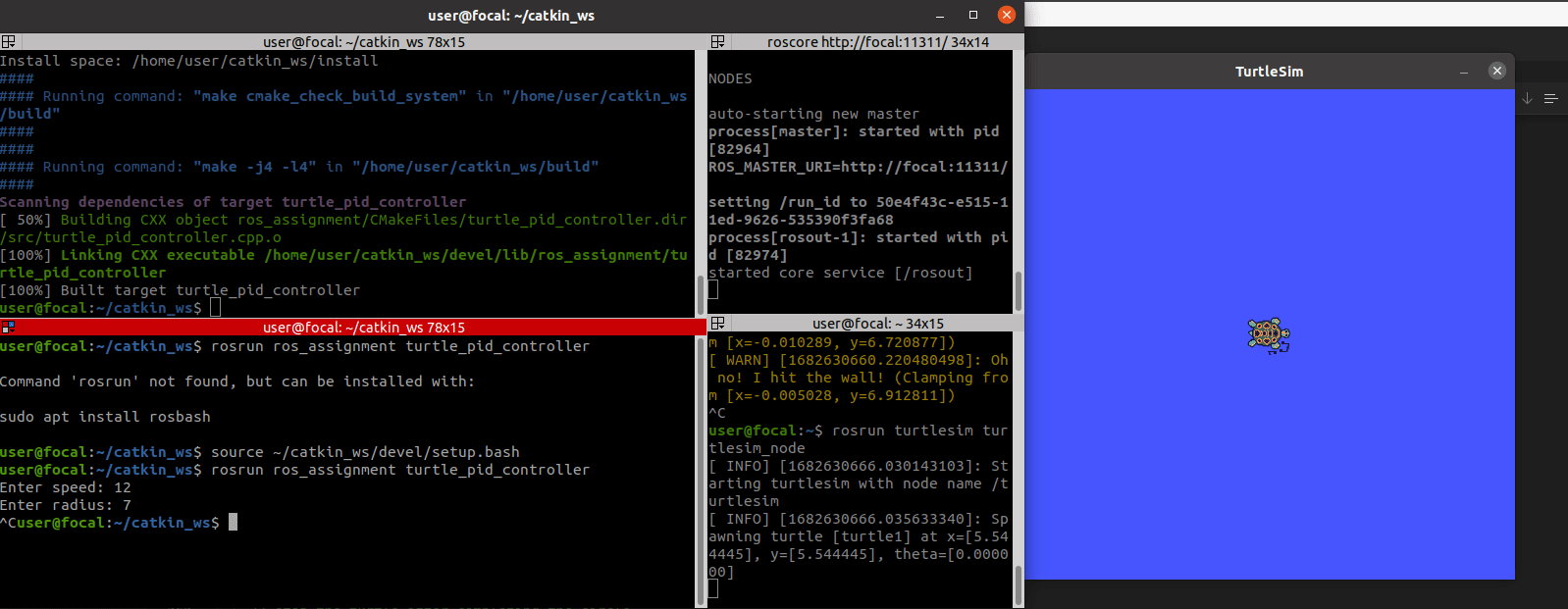
These graphs were rendered by exporting the ROS\_INFO angular velocity to an excel file, as rqt\_multiplot was triggering installation conflicts, as mentioned before. We can now **hypothesize** with the given data that the proportional component of PD controller in this case is directly proportional to the change in angular velocity of the turtle. Please note that this has been hypothesized and not concluded, a further deal of control testing would be required to draw out robust logical conclusions in context of this data.

This goal was achieved using the sweep() function, which by itself makes extensive use of move() moveGoal() rotate() and setDesiredOrientation() functions. Let us explore how these function work, in context of sweep():

1. A turtlesim::Pose object named pose is created and initialized with an x, y, and theta value of 1, 1, and 0 respectively. This represents the starting position and orientation of the turtle robot.
2. The moveGoal() function is called with pose as the input argument and a distance of 10.0. This function moves the turtle to the desired goal position using the ros::Publisher and geometry\_msgs::Twist messages.
3. The loop is put to sleep using loop.sleep() to allow time for the turtle to move to the goal position.
4. The setDesiredOrientation() function is called with an input argument of 0. This function sets the desired orientation of the turtle to 0 degrees.
5. The loop is put to sleep again.
6. The move() function is called with input arguments of 2.0 and 9.0, representing the distance and speed of the turtle's movement. The third input argument, true, indicates that the turtle should move in a straight line until it reaches the desired distance.
7. The loop is put to sleep again.
8. The rotate() function is called with input arguments of degrees2radians(10), degrees2radians(90), and false. This function rotates the turtle by 10 degrees in a counter-clockwise direction until it reaches the desired angle of 90 degrees.
9. The loop is put to sleep again.
10. The move() function is called again with input arguments of 2.0 and 9.0 to move the turtle to a new position.
11. Steps 10-12 are repeated two more times with different rotation angles and movement distances to create the sweep-like motion.
12. The getDistance() function is a standard Euclidean distance calculator

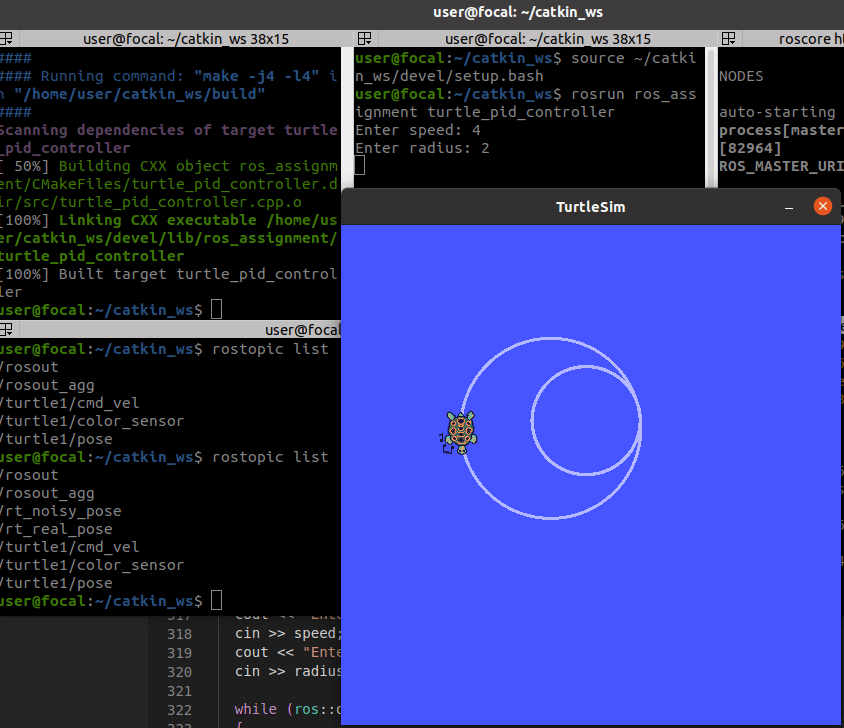
### Goal 3: Rotate Turtle in Circle

For goal 3 we have to rotate the turtle in a circle, this is relatively straightforward, we take user inputs for the speed and radius while aggregating topics /rt\_real\_pose for standard pathing and /rt\_noisy\_pose for pathing with random gaussian noise Find the implementation below:



*Turtle rotating with a speed of 12, high speed*

The above turtle moves with a speed of 12 units and it’s path has a radius of of 1.2 units, as mentioned above the bounds of rotation are from (0,0) to (11,11) there is no handling for collision with walls either. Now lets see a turtle rotating at a slower speed, the turtle starts its rotation from its current position, also notice the topics below:



*Turtle rotating at slower speed starting from its current position, topics also displayed on screen*

This implementation is run by a completely different program, here is how the program works:

The program creates three publishers to publish messages to three different topics: /turtle1/cmd\_vel, /rt\_real\_pose, and /rt\_noisy\_pose.

It start by initializing a ROS node with the name "move\_circle" and creating a node handle object nh to communicate with the ROS system. It then creates the publishers for the three topics mentioned earlier. The user is prompted to enter a speed and radius, which are used to calculate the linear and angular velocities of the turtle. The program then enters a loop where it continuously publishes velocity commands to move the turtle in a circle.

The turtlesim\_pose object is used to set the position and orientation of the turtle at each iteration of the loop. The vel\_msg object is used to set the linear and angular velocities of the turtle. The current\_angle variable is calculated by subtracting the current time (t1) from the starting time (t0) and multiplying by the angular velocity. This angle is then used to calculate the new position and orientation of the turtle.

At each iteration of the loop, the current position of the turtle (turtlesim\_pose) is published to the /rt\_real\_pose topic. A noisy version of this pose, with added Gaussian noise, is also published to the /rt\_noisy\_pose topic.

Gaussian noise is added to the turtle's pose to simulate the effects of measurement errors or sensor noise. The amount of noise added is generated randomly and follows a normal distribution, with a mean of zero and a specified standard deviation. The effect of Gaussian noise is to add random variations to the turtle's pose, which can cause the turtle to deviate slightly from its intended path. This is a common problem in real-world robotic systems, where sensor measurements are often noisy and prone to error. By simulating these effects, the code can test the robustness of the system and help improve its performance.

Once the turtle has completed a full circle, the program sets the linear and angular velocities to zero, stopping the turtle. The loop then continues to publish the final turtle pose to the /rt\_real\_pose and /rt\_noisy\_pose topics.

### Goals 4,5 and 6:

The solution to these problems was programmed, however it was not up to equitable standards and did not produce the desired output, hence here is the attempted procedure for each Goal.

#### The Approach:

* Define the turtles: Create two turtles using the turtlesim spawn service/library - one for the robber and one for the police. Set the initial position and velocity of the robber turtle according to the user-defined radius and speed.
* Set up the publishers and subscribers: Create a publisher for the robber turtle's position data, publishing it to the topic /rt\_real\_pos. Create a subscriber for the same topic, which the police turtle will subscribe to track the robber turtle.
* Implement the police turtle's movement: In the police turtle's movement loop, calculate the distance between the police and robber turtles using their respective position data. If the distance is less than 3 units, the police turtle has caught the robber turtle and the program can end. Otherwise, the police turtle should adjust its velocity to move towards the robber turtle's position, considering its acceleration and deceleration limits.
* Run the program: Start the ROS node and allow the robber turtle to spin in circles for 10 seconds. After 10 seconds, spawn the police turtle at a random location and start its movement loop. The police turtle should chase down the robber turtle until it is caught.

For Goal 5 and 6 speed and gaussian noise must be added respectively.