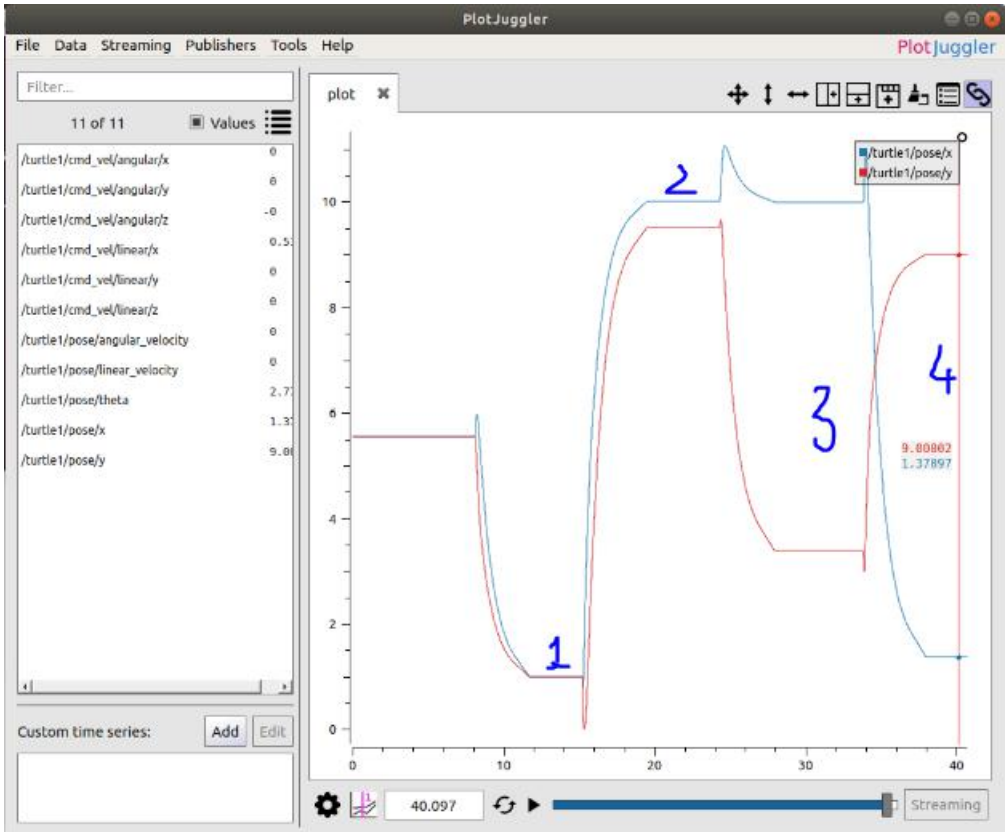
Goal 1:

To reach the input goal, constants of proportional, derivative and integral gains are defined. The respective errors(p,d,i) are calculated in loops and velocity is published, until the self position differs from the goal by a specified tolerance. Adjusting this value (currently 0.5), we can achieve more precise goals. Here, the parameter for setting linear velocity is taken as distance (always positive) from goal. It could also be (current_coordinate-goal_coordinate), hence including reverse velocities also. However, this can also be resolved by setting appropriate angular velocities, which is what has been done here. Also, the derivative error has not been divided by time as the derivative constant would account for that.
The launch file turtle_nav.launch was used for this goal.
Link to video: Video

Goals were given in order : (1,1), (10,9.5), (10,3.4), (1.4,9)
Tolerance was 0.5
Result plot:



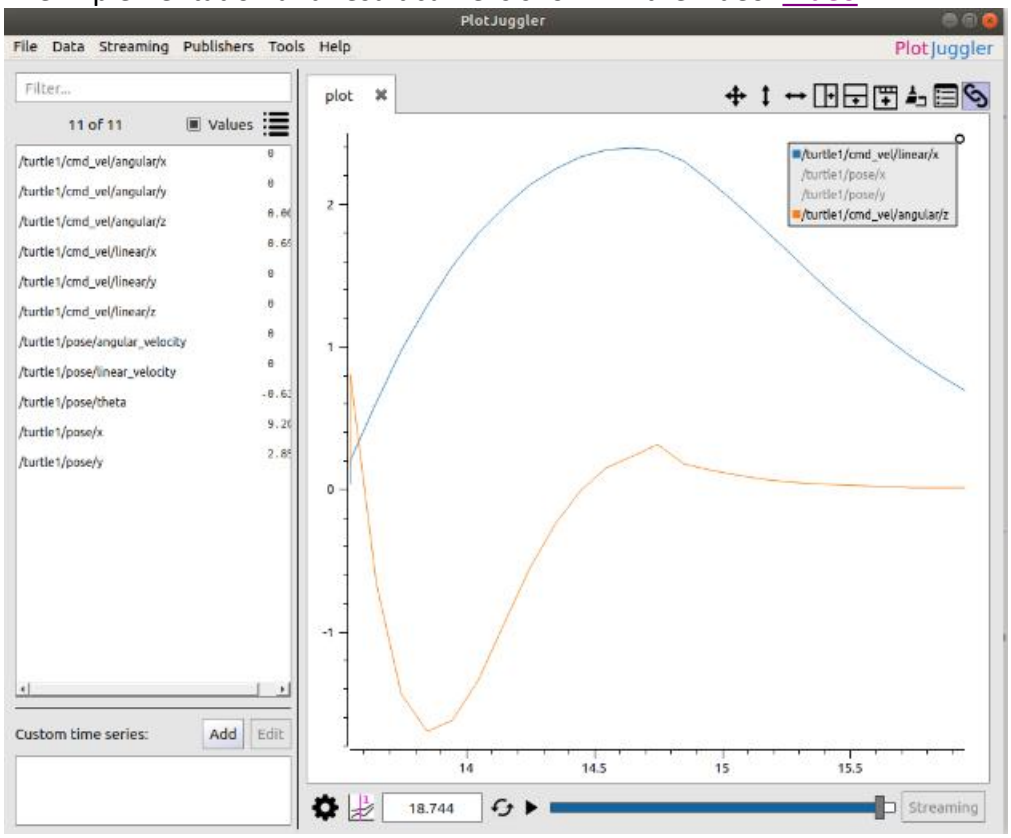
(Link for video of results: Video)

Goal 2:

To limit the accelerations, a function step_vel has been used. In this, the target velocity is sent, and it increases the velocity in steps. Before publishing this velocity, however, the step is checked for its value.

The maximum acceleration limits are defined, and can be changed. The delta(time) is recorded for each stage of velocity increment. If the velocity step/delta(time) is greater than the maximum value, the step size is reset to maximum allowable value w.r.t. that particular delta(time). This process is done for both linear and angular acceleration/deceleration.

The implementation and result curve is shown in the video: Video



The gradual rise of the velocities (linear,angular) are shown in the figure.

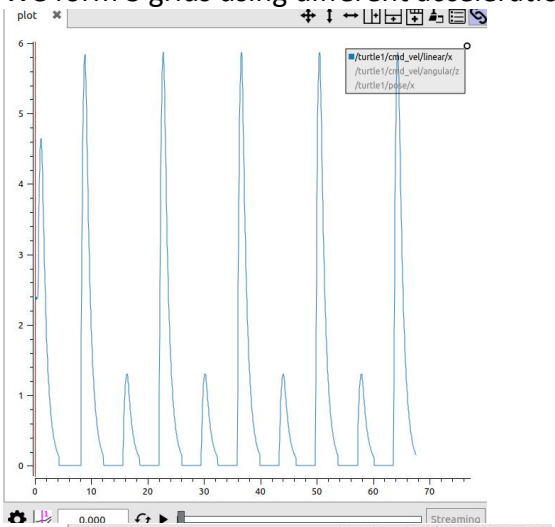
The node for this is goal_accel_decel

Grid

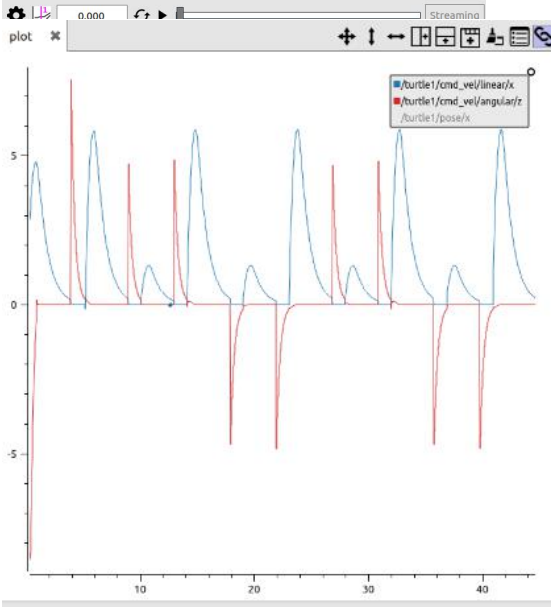
The grid task can be though as discrete sections.

1. From any random location, go to the starting position in fastest way possible.
2. Thereafter, grid points are defined which have to be followed. Hence, fastest way would give a curvature and not follow lines. Thus, grid corners are defined.
3. However, at the end of a corner, the orientation of the turtle is perpendicular to next line that has to be followed. Directly commanding to go the next corner would again give a curvature. (Note: The lines are given importance, if that wasn't the case, subsequent calls to go_to_goal(target) would suffice.)

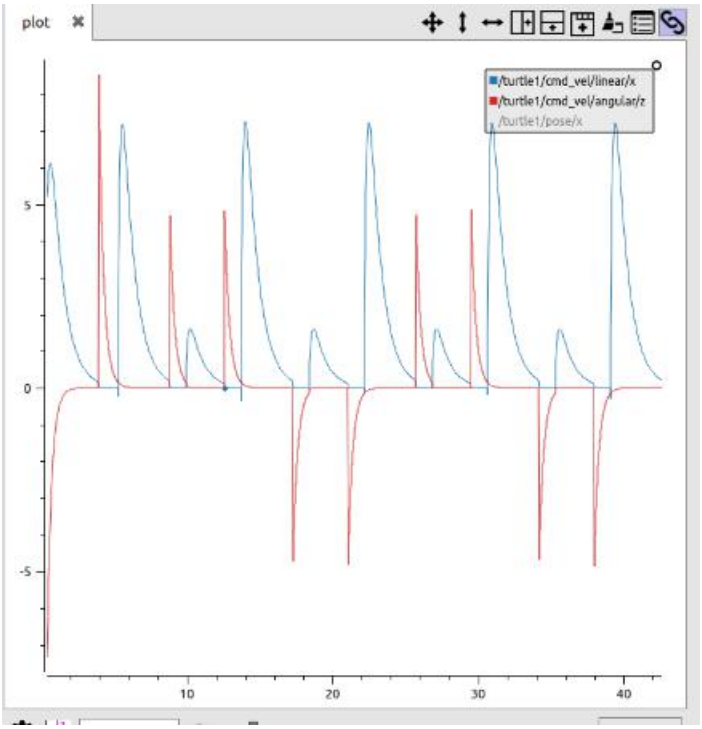
4. To avoid curvature at the end, a rotate() function has been defined. This takes in an angle and rotates it with proportional control.
5. Finally, the grid function is defined. Grid_corners contain x,y coordinates, as well as angles to be rotated at the end of traversal. Go_to_goal() is then called.
The acceleration parameters are varied and trajectory recorded for them.
Link: Video
We form 3 grids using different accelerations. The curves of these are as follows:



Grid 1
Max_accel_linear: 1500
Max_accel_angular:6000



Grid 2
Max_accel_linear: 2500
Max_accel_angular:8000



Grid 3
Max_accel_linear: 12000
Max_accel_angular:12000

The linear velocity curves become sharper as the acceleration constants are increased.

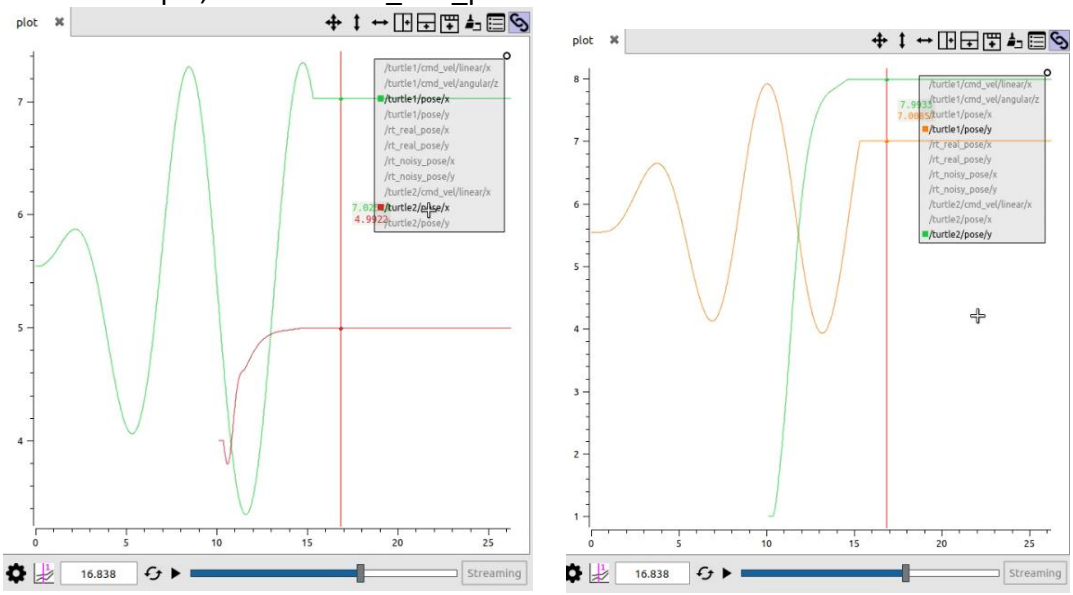
Goal 3:

The circles() function prompts the user for input of radius and velocity, accordingly calculating angular velocity.
Once determined, the velocity of the turtle is increased in steps using step_vel() function.
When launched, it starts a time variable. Every 5 seconds, this variable causes the node to publish two poses: rt_real_pose, which is the actual pose of the turtle, and rt_noisy_pose, which contains random gaussian noise added. The mean and standard deviation of this noise is a major parameter in determining whether it can be chased down in further goals.
Link to video: Video

Goal 4:

The chase is executed as follows:
First, the turtlesim_node is launched with one turtle.
Then the circle function from Goal 3 is used to start circling motion of this 'Robber Turtle'. Then another node turtlespawn is used. This first gives a delay of 10 seconds. After this, it calls the service spawn and gives random values to spawn another turtle, which gets named 'turtle2'. However, the name could be taken as a return argument of the service also, but assuming that only one of the launch files are run at a time, there won't be any name mismatch. The launch file then spawns the chase_limit_accel.

This node subscribes to the rt_real_pose. Hence, it has to wait for 5 seconds before receiving its first goal position. The speed of this is greater than that, and as such no limit on speed has been imposed at this stage (will be done later).
However, the success of this program is contingent on various factors. The chase was said to be completed when the turtles were 10 units apart. However, this resulted In no motion most of the times and were already less than 10 units apart. Hence the new threshold was taken as 3 units.
Due to the increased speed, PT can reach RT's last location quickly. But the difference of the speeds determine exactly how far apart will they be when the next rt_real_pose comes in. Till then the PT will rest on the last known location of RT if it reaches in less than 5 seconds. In this case, there are several possibilities. If the circle is large enough, the RT might be on the other end after 5 seconds. While PT again reaches the new location, RT could be>3 units far. Depending on how much it is catching up every 5 seconds, the chase would then be completed in finite time. However, if PT is very fast, it might also catch up in one go. Radius of circle will play a role, as a smaller one finishes the task quickly.
Link: Video
Here, the radius of the circle is 2 units, and velocity of RT is 2 units. PT has limit on acceleration, not on speed.
In this example, it took one rt_real_pose to reach within 3 units of RT.



Dx=2 units, dy=0.9 units.

Goal 5:

In the last example, due to greater velocity, PT would always gain on RT, even if in very small increments, as it would reach its last location quickly and RT would not have left the last location as far behind. So in the next position message, it could shorten their distance even more. Thus we could be sure that it will catch up in future.
However, if the velocity of RT is double that of PT, whenever PT reaches a location RT has been, RT would have traversed more than what PT could catch. Hence we cannot be sure of catching up in this case.

However, this is true for a generic path. For a circle, as the RT would at regular intervals come at the same location it had previously been, the gradient of their velocities have the same effect of closing the gap between them.
Below is a video of this. Here, the velocity of RT is 2, and that of PT is limited at 1.
Link: Video
But this is not an ideal way as we are not explicitly controlling the chase
Planning:.
We can leverage the fact that RT is moving in circles. The chase_limit_accel_vel_plan is the node which uses this. In it, we use a planning function.
The node waits for first three inputs from the rt_real_pose. Once it obtains three poses, then there will be a unique circle defined. Thus, running define_circle function gives us the radius and center of circle. Further, we calculate where the RT will be next time it sends the signal. Once we have that position, we check if we can reach that in 5 seconds. If yes, then we command the PT to go to the position. If not, then we find the position RT will be at 10 seconds from now. We repeat this process in steps of 5 seconds, till we obtain a point reachable in the time slot. Then we proceed to the goal_predicted position. During this whole process, as the rT_real_poses arrive, we first check if they are in the range of catching (3 units) and end the task accordingly. Below is the link of this method:
Video
Possible improvement: Instead of predicting only at 5 second intervals, an exact position-time curve can be mapped. Then we would just have to stay the RT until a rt_real_pose is sent out, which will end the task immediately as we would be in required proximity (3 units or some other value).

Goal 6:

If the RT emits pose with gaussian noise, the planning becomes difficult. The uncertainty of reaching near the target is dependent on the mean and standard deviation of the gaussian noise. If high values of standard deviation are chosen, with the PT half as fast as RT, it will not be able to catch RT. However, again due to circular motion, this may be the case, as even with gaussian noise, and a threshold of 3 units, the PT would get close enough to end the chase. As random gaussian noise fluctuates quickly, it might destabalize PT.
Here is a video where PT was able to approach RT (std. dev=1) and get in proximity to end the chase.
Video
Here is another video where planning algorithm failed to pinpoint location due to innacurate pose. Thus it took multiple poses to get in proximity and end the chase.
Video
But here, the PT could not approach the RT due to excessive gaussian noise (std. Dev=2)
Video
In such situations, better estimates of position is required to proceed and finish the chase.