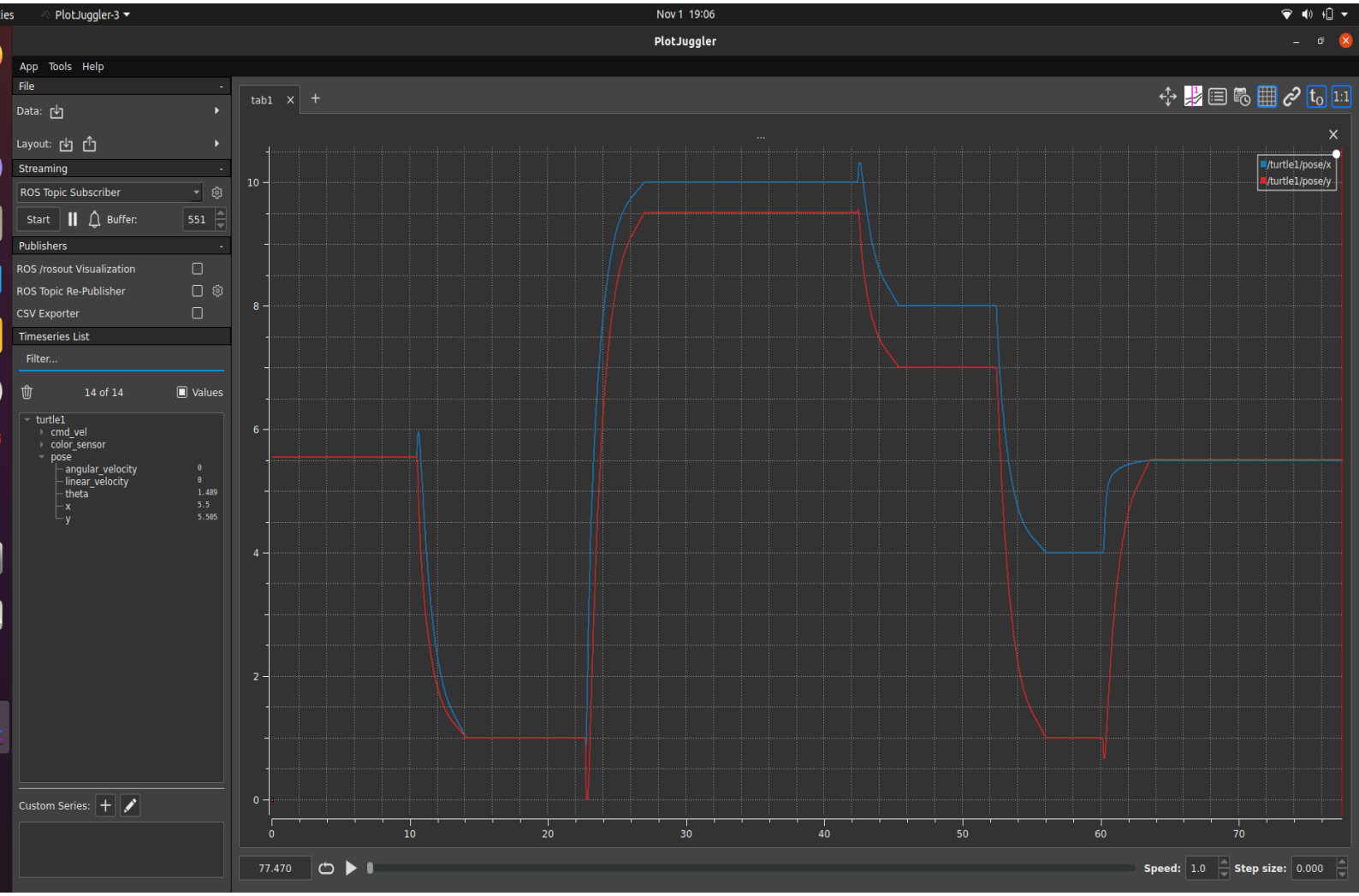
Assignment - Robotics Engineer
by Aman Vyas

Goal 1 : Controlling turtle:

To achieve this goal, I defined specific constants for proportional, derivative, and integral gains. These constants were used to calculate errors (referred to as p, d, and i) within iterative loops, and the robot's velocity was continuously updated and published. This process continued until the robot's self-position closely aligned with the target location, with a predefined tolerance. By adjusting this tolerance value, which is currently set at 0.5, I was able to enhance the precision of goal achievement. In this context, the linear velocity parameter was determined as the distance between the robot's current position and the goal position. Alternatively, it could be computed as the difference between the current and goal coordinates, accommodating both forward and reverse velocities. However, I resolved this issue by appropriately setting angular velocities.

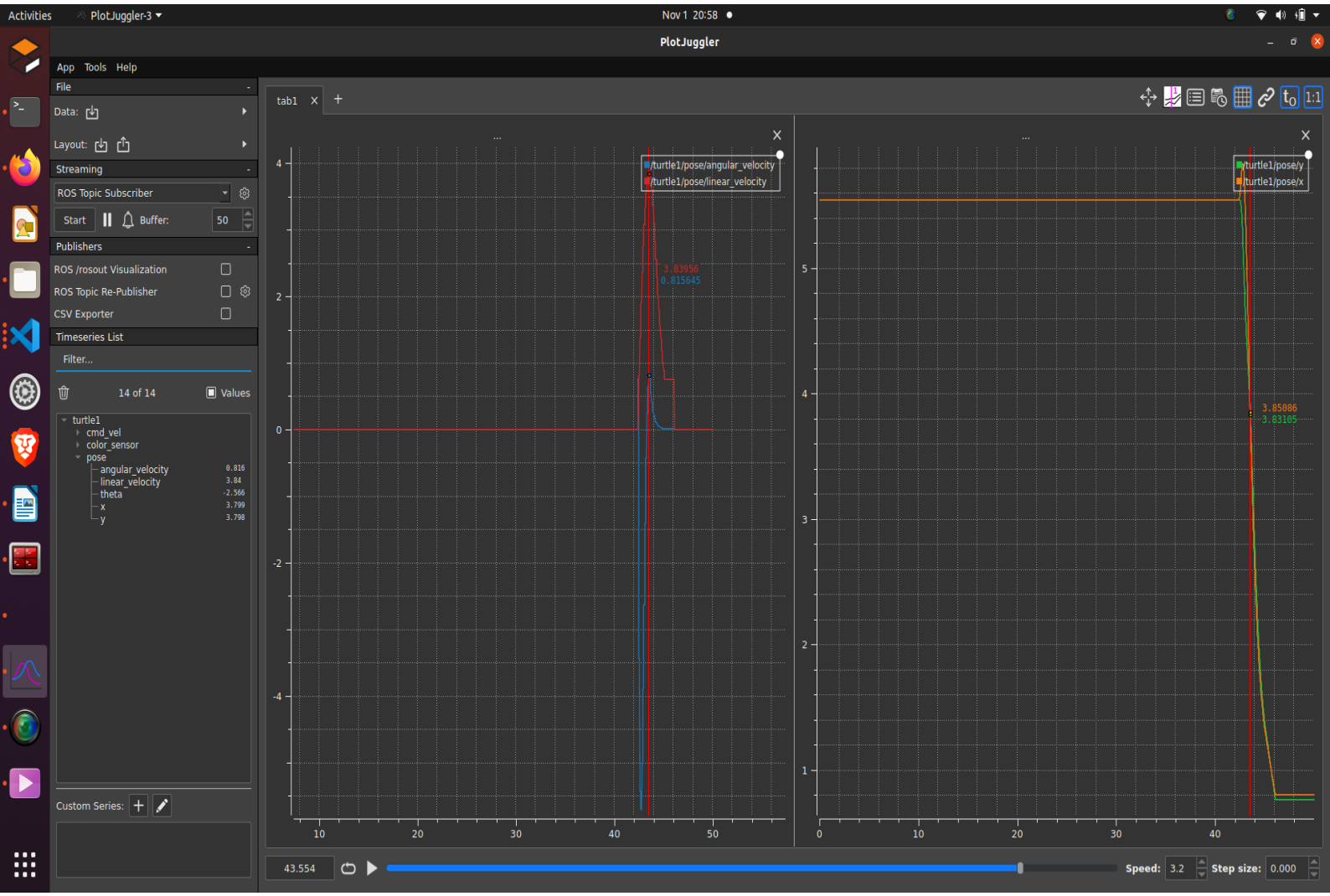
This entire process was orchestrated using the 'turtle_precise_navigation.launch' launch file.position tolerence here is 0.5

Goal points:(1,1),(10,9.5),(8,7),(4,1),(5.5,5.5)



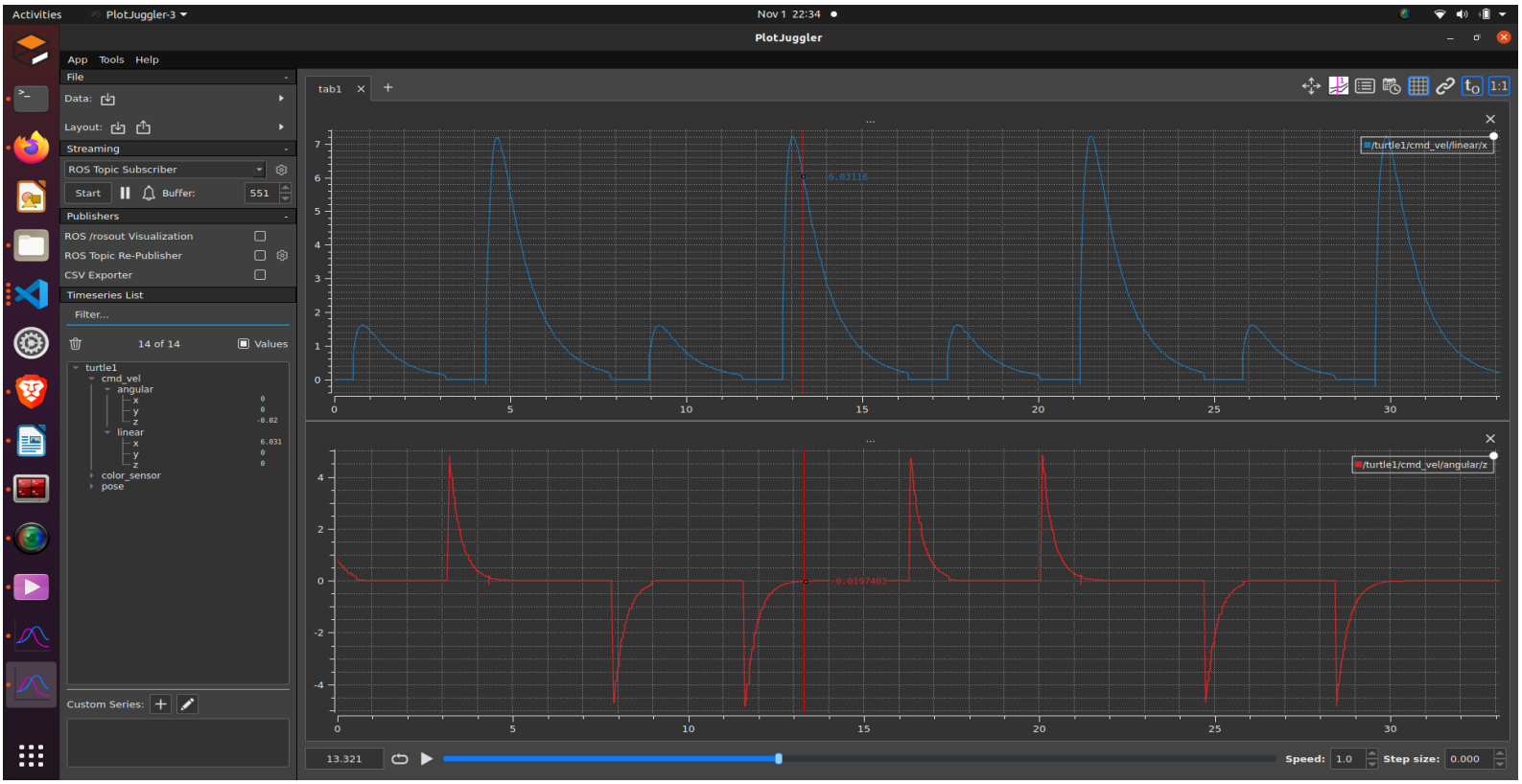
[video link for results of goal_1]

Goal 2: Making a grid pattern:
To constrain accelerations, a function "step_vel" has been employed. Within this approach, the desired velocity is transmitted and is progressively incremented in discrete steps. Prior to the publication of this modified velocity, a thorough assessment of the step's value is conducted.
The maximum acceleration limits are set and can be adjusted as needed. In each stage of velocity increment, the corresponding time interval, denoted as delta(time), is recorded. If the ratio of the velocity step to the delta(time) exceeds the maximum permissible value, the step size is reset to the maximum allowable value for that specific delta(time).
[Implementetion video]

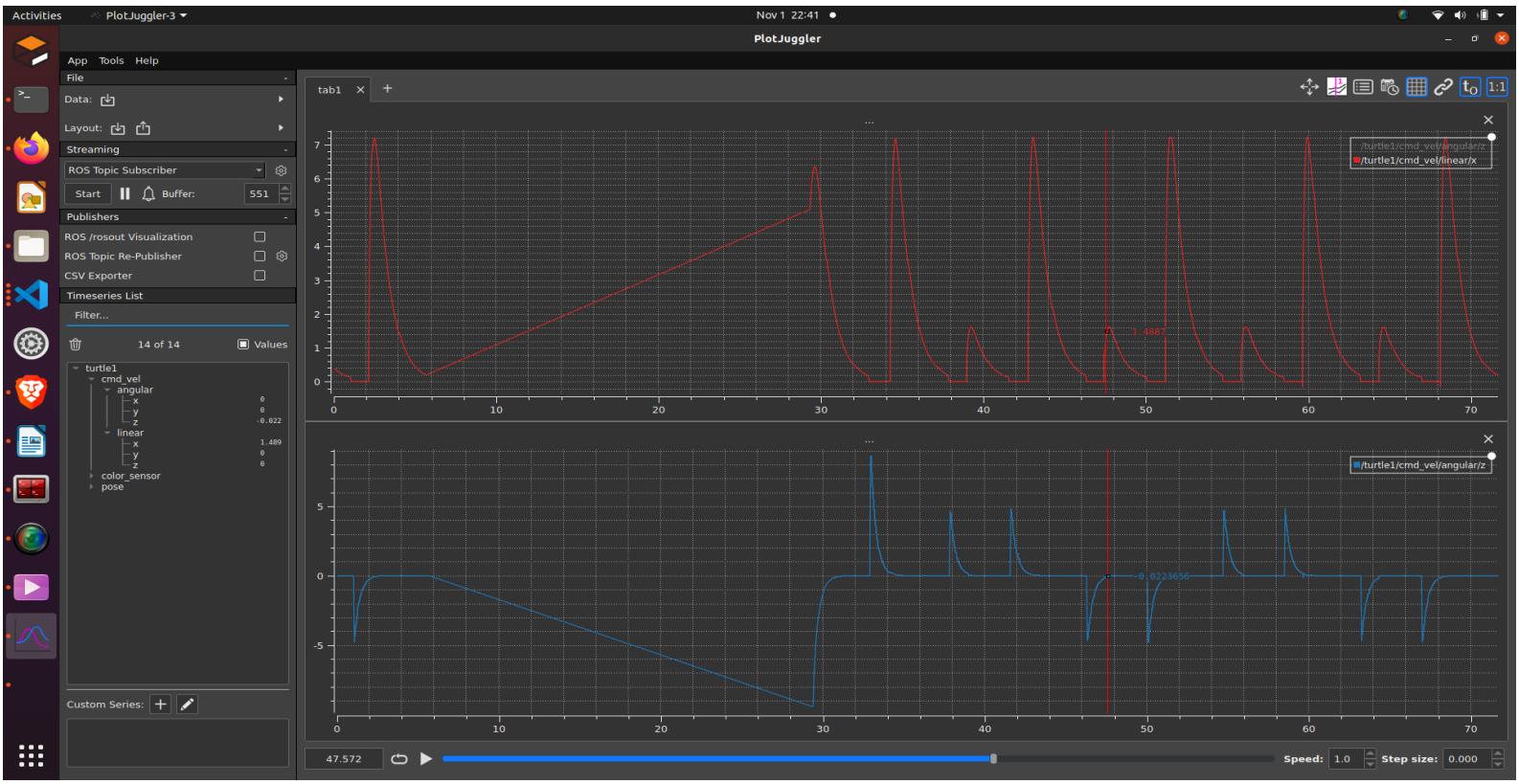


for above goal, the node is goal_acceleration_deceleration.py and the launch file is turtle_nav_accel.launch.
Grid
The grid task can be though as discrete sections.

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 goal_reach(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 for the video implementation
I form 2 grids using different accelerations. The curves of these are as follows:



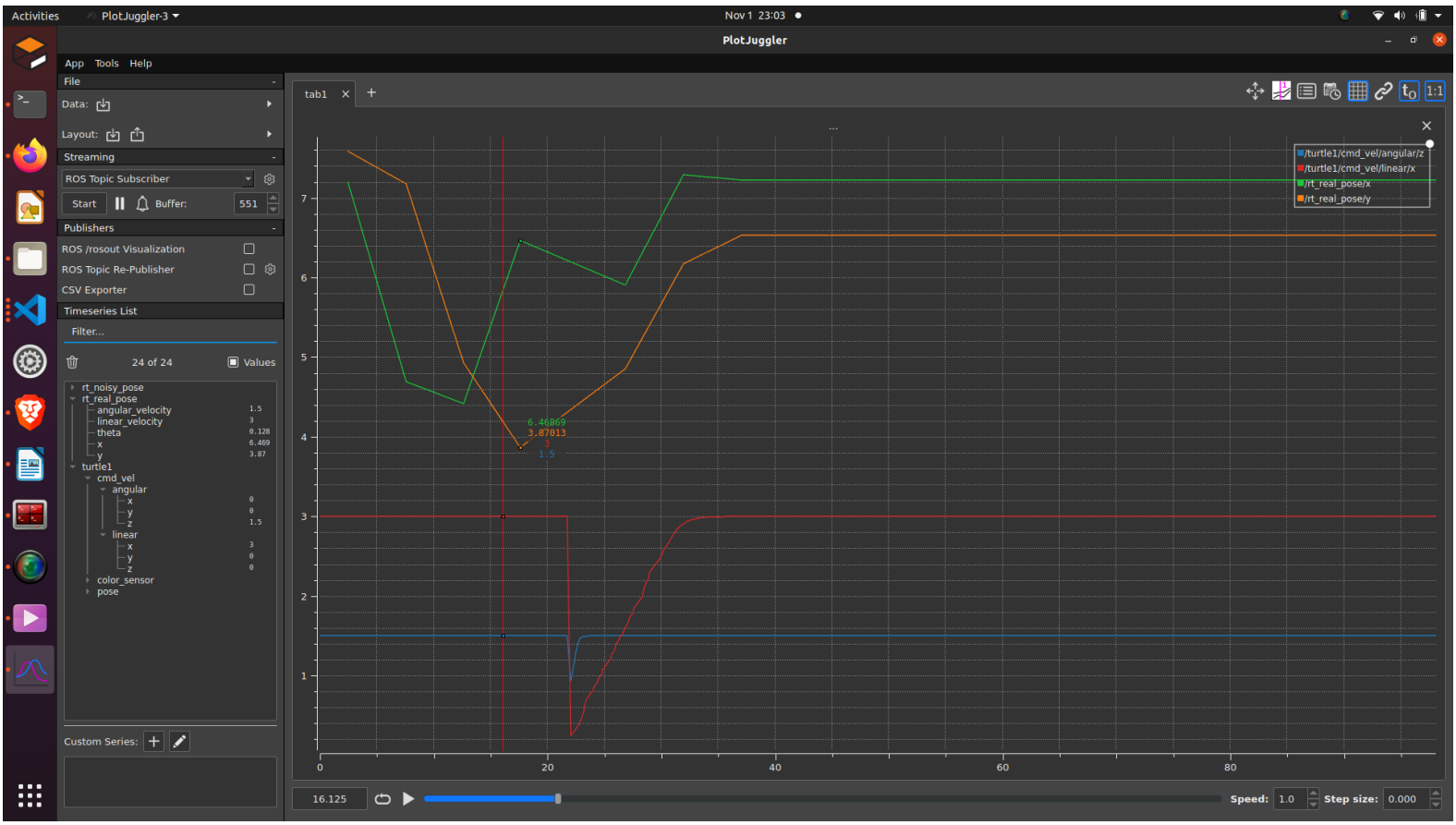
Grid 1
Max_accel_linear: 20000
Max_accel_angular:12000



Grid 2
Max_accel_linear: 600
Max_accel_angular:1200
The linear velocity curves become sharper as the acceleration constants are increased

Goal 3:Rotate turtle in circle

The `circles()` function initiates user input for radius and velocity, leading to the computation of angular velocity. Once these values are established, the `step_vel()` function is employed to incrementally increase the turtle's velocity. Upon launch, a time variable is initialized, and every 5 seconds, this variable triggers the publication of two poses: `rt_real_pose`, representing the precise turtle pose, and `rt_noisy_pose`, which introduces random Gaussian noise. The parameters governing the mean and standard deviation of this noise play a pivotal role in subsequent goal pursuit, impacting the feasibility of tracking the turtle.



[Link to video:]

Goal 4:Chase turtle fast

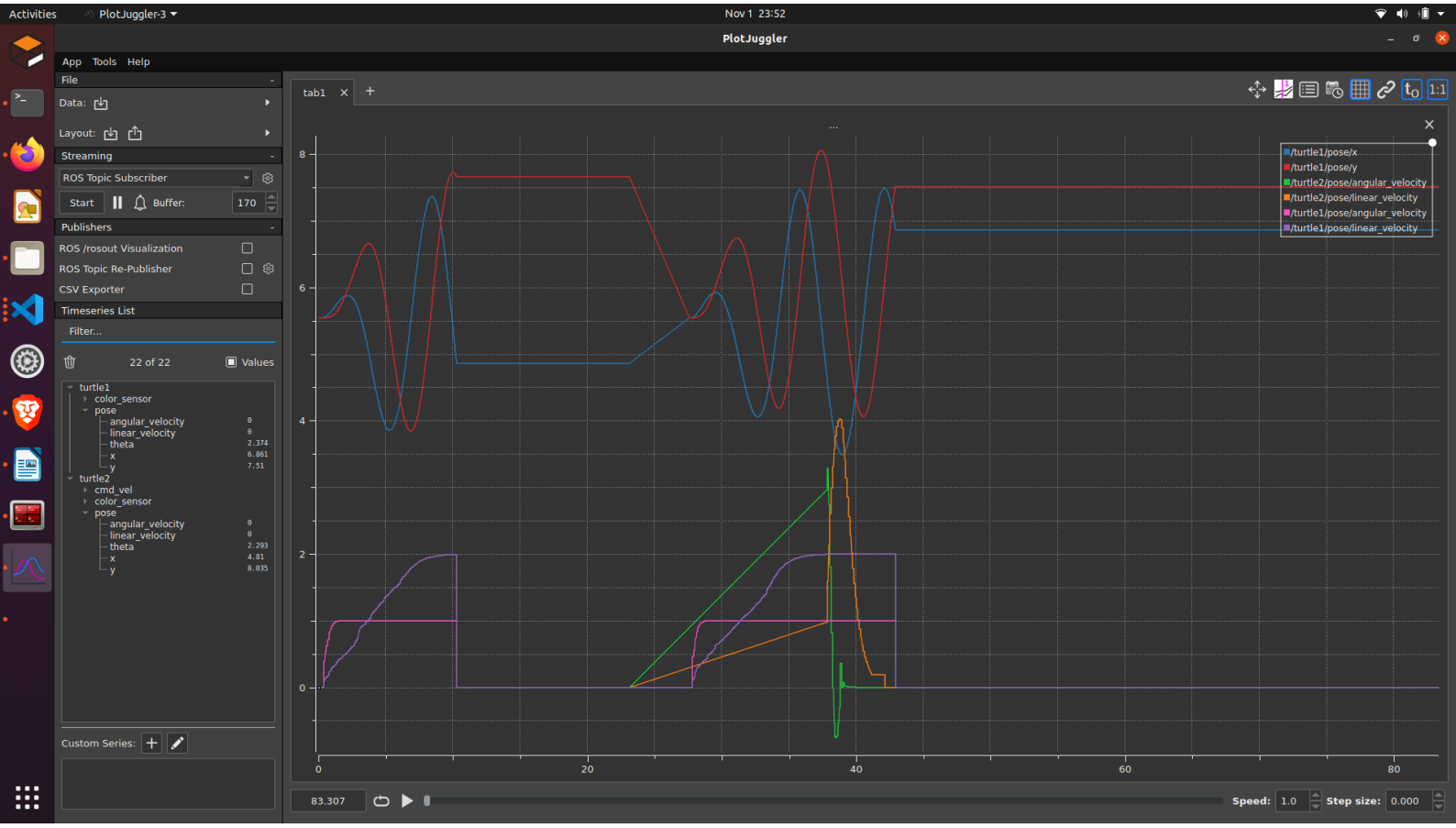
The sequence of actions to achieve this objective is as follows:

Initially, the `turtlesim_node` is initiated with a single turtle.
Subsequently, the circling motion of this 'Robber Turtle' is initiated using the circle function from Goal 3.
To introduce another turtle, the `turtlespawn` node is employed, which involves a 10-second delay. Following this delay, it invokes the 'spawn' service, assigning random values for the creation of another turtle, referred to as 'turtle2'. It's worth noting that the service could also return the name, but considering that only one of the launch files is typically executed at any given time, there shouldn't be any naming conflicts.
The launch file then triggers the `chase_limit_accel` with the following parameters from Grid 3:

- Max_accel_linear: 12000
- Max_accel_angular: 12000

This node subscribes to `rt_real_pose` and necessitates a 5-second waiting period before receiving its initial goal position. No speed limitations are imposed at this point, although they may be imposed later.
However, the success of this program depends on several factors. Initially, the chase was defined as completed when the turtles were 10 units apart, but this often resulted in no motion, as they were already closer than 10 units. Consequently, the new threshold was set at 3 units.

The increased speed enables the 'PT' (Pursuer Turtle) to rapidly reach the last known location of 'RT' (Robber Turtle). However, the difference in their speeds determines the distance between them when the next 'rt_real_pose' update is received. If 'PT' arrives within 5 seconds, it will pause at the last known location of 'RT'. Several scenarios are possible in this situation. If the circle is large, 'RT' might be on the opposite side after 5 seconds. When 'PT' reaches the new location, 'RT' could be more than 3 units away. The time it takes for the chase to be completed will depend on how quickly 'PT' catches up every 5 seconds. If 'PT' is exceptionally fast, it might catch up in a single attempt, and the size of the circle also influences the task duration, with a smaller circle leading to a quicker completion.
Link:



Goal 5:Chase turtle slow
In the preceding example, the discrepancy in velocities played a crucial role in the pursuit dynamics. With PT moving at a higher speed, it consistently gained ground on RT, albeit in incremental increments. This happened because PT swiftly reached its most recent location, and RT, with its slower pace, lagged behind. Consequently, the subsequent position update further reduced the separation between them, instilling confidence in their eventual convergence.

However, when RT's velocity was twice that of PT, a different scenario emerged. Whenever PT reached a location that RT had previously occupied, RT had already covered a distance greater than PT's catching capacity. This posed uncertainty about the possibility of PT catching up under these circumstances.

Nonetheless, it's essential to note that these dynamics hold true for a general path. In the case of a circular path, where RT cyclically revisited its prior locations, the relative velocities of PT and RT exerted a consistent effect in narrowing the gap between them.

For a visual demonstration, please refer to the video provided below, where RT's velocity is set at 2 while PT's speed is capped at 1.
Link: Video
However, this approach has some limitations since it lacks explicit control over the chase planning. An alternative strategy can be employed by taking advantage of the circular motion pattern exhibited by the real-time (RT) system. The key component for this is the `chase_limit_accel_vel_plans` node, which utilizes a planning function.

In this approach, the node initially waits for the receipt of the first three inputs from the `rt_real_pose` source. Once these three poses are obtained, a unique circle is defined. Subsequently, the `define_circle` function is executed, providing us with crucial information such as the radius and center of the circle. With this data, we can predict the RT's future position at the time of its next signal transmission.

The system then calculates whether it is possible to reach the predicted position within a 5-second timeframe. If the calculated position is reachable within this window, the system commands the PT (presumably a control system) to move to that position. If the 5-second timeframe is insufficient for reaching the position, the system then estimates the RT's position at 10 seconds in the future and repeats the process in 5-second intervals until a reachable point within the allotted time frame is determined. Once this point is identified, the system proceeds towards the `goal_predicted` position.

Throughout this entire process, the system continually monitors the incoming `rt_real_poses`. If any of these poses come within the defined catching range of 3 units, the system terminates the task accordingly.

For more details on this method, you can refer to the following link:
Video

Goal 6:Chase turtle noisy
In the pursuit of our objective, when the Real-Time (RT) system introduces pose data with Gaussian noise, it introduces a level of complexity to the planning process. The extent of uncertainty in successfully reaching the target is contingent on both the mean and standard deviation of the Gaussian noise. Opting for higher standard deviation values while keeping the Planning-Time (PT) half as fast as RT can result in PT being unable to keep up with RT. Even in the presence of Gaussian noise and a specified threshold of 3 units, PT can approach RT closely enough to conclude the pursuit. The rapid fluctuations introduced by random Gaussian noise may potentially destabilize PT. In a provided video example, PT managed to approach RT when the standard deviation was set to 1, eventually reaching proximity and ending the pursuit.
Video