Reference Point Tracking

Arba Shkreli and Elianne Sacher

September 8, 2022

Abstract

In this report we summarize ROS2 reference point tracking control projects that we have completed in the CoNG-Li lab. The controllers used in our implementations were PID and LQR. The programs were completed both on single robot and multiple robot systems. We have proved the effectiveness of our code both in simulation (performed in Gazebo) and in practice (performed on Turtlebot3 robots in the lab).

The goal of completing these projects was to get acquainted with ROS2 so that we can proceed to create more complex programs. During this process, we learned how to use the ROS2 framework of interacting nodes to construct a simple robotic system of one or more robots to fulfill the control goal of reaching a given reference point or trajectory. This was done on both a simulation platform and in a lab environment with real robots.

Contents

1 Introduction

This report summarizes the different reference point tracking control projects that we have completed while working in the CoNG-Li lab. The goal of these projects was to help introduce us to ROS2, Turtlebot3, and the OptiTrack systems. The first step we took was performing proportional–integral–derivative (PID) control in 2D on a single simulated robot simulated on Turtlesim, ROS2's learning platform simulator. After accomplishing this step, we moved to perform the control objective of reaching the reference with two different methods:

1. Our first method was to gradually arrive at a reference point by reaching intermediate points, where each waypoint is approached by proportional velocity control. Each waypoint is generated by gradient descent of the superposition of 2D potential functions assigned to all objects in the system, where the system is taken to be all the robots (that know each other's positions under a common coordinate system) and the reference point. Each robot is attracted to the reference point by one potential function and repelled to the other robots by the use of another potential function. The associated potential function for the attraction is set to have a minimum point at the reference itself, while the associated potential function for the repulsion has a maximum point at the location of the other robots.

2. Our second method was to arrive at a reference point by generating a full path of waypoints by gradient descent of the same reference potential functions mentioned before, and the points are approached by linear-quadratic regulator (LQR) velocity control. In order to find the waypoint path, we had to assign 2D potential functions to all objects in the system, where the system is taken to be all the robots (that know each other's positions under a common coordinate system) and the reference point. Each robot is attracted to the reference point by a potential function, which is set to have a minimum point at the reference itself. The robots iterate through the waypoints following them individually using LQR control. To avoid collision, when the robots are within a certain radius of each other, a second potential function is introduced (same as the one mentioned in the first method) which repels the robots away from the others. This repulsive function has a maximum point at the location of the other robot and superposing its gradient of the next waypoint successfully prevents collision.

The codes were written in an adaptable way enabling the of any number of robots desired. Our codes worked in both simulation (using the Gazebo environment) and in a lab setting using Turtlebot3 robots and the OptiTrack system (which provides nearly real-time coordinates of our robot).

Note: In this project we did not test the systems' robustness to measurement and communication noise/delay. Since this was our first project using ROS2, we used it to learn the basics first. We must note that in the simulation of the robots, the `.urdf` and `.sdf` files that describe them as objects in the simulation environment we realized already had noise applied to their actuation mechanisms. This is by virtue of us having made use of a description file for the Turtlebot that was already included in the installation of Gazebo.

1.1 Relevant ROS2 Terminology

To implement our model on Tutlebot3 we used Robot Operating System (ROS). ROS2 is a set of software libraries and tools for building robot applications [Rob21]. In this report, we will be using some ROS2 terms, which include:

- ROS2 Graph: At the heart of any ROS 2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate, which includes messages, topics, and discovery.

- ROS2 Node: A node is a participant in the ROS graph. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to Topics. Nodes can also provide or use Services and Actions. There are configurable Parameters associated with a node. Connections between nodes are established through a distributed discovery process. Nodes may be located in the same process, in different processes, or on different machines. These concepts will be described in more detail in the sections that follow.

- ROS2 Topics: ROS2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.
- ROS2 Messages: ROS data type used when subscribing or publishing to a topic.
- ROS2 Discovery: The automatic process through which nodes determine how to talk to each other.

2 Theoretical Concepts Used

2.1 Discrete System Modeling

In order to implement any one of our control methods we first needed to generate a systems of equations modeling our system [AM21a]. After discretizing the physical system and using basic Newtonian mechanics we receive the following models:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} \dot{x}_t \\ \dot{y}_t \\ \dot{\theta}_t \end{bmatrix} \Delta t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} \cos\theta_t \cdot v_t \\ \sin\theta_t \cdot v_t \\ \omega_t \end{bmatrix} \Delta t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} \cos\theta_t & 0 \\ \sin\theta_t & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_t \\ \omega_t \end{bmatrix}^T \Delta t \tag{1}$$

Where x and y are the X and Y coordinate (assuming a Cartesian model), θ is the yaw angle calculated about the Z axis (assuming the positive X axis direction as $\theta = n \cdot 2\pi$ where $n = 0, 1, ..., \infty$), t is a given time-step (assuming $t \geq 1$), Δt is the time-span between each step, v is the linear velocity at a given time-step, and ω is the angular velocity at a given time-step. Adopting it to a state space model our state vector, $s(t)$, will be equal $\begin{bmatrix} x_t & y_t & \theta_t \end{bmatrix}^T$, our state matrix A will be the identity matrix, our input matrix will equal $B = \begin{bmatrix} \cos\theta_t & 0 \\ \sin\theta_t & 0 \\ 0 & 1 \end{bmatrix}$, and our control vector would equal $u(t) = \begin{bmatrix} v_t & \omega_t \end{bmatrix}$.

2.2 PID Control

The proportional–integral–derivative (PID) controller is the most common form of feedback [AM21b]. The "textbook" version of the PID algorithm is described by:

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{2}$$

Where y is the measured process variable, r the reference variable, u is the control signal and e is the control error, also known as the set point, $(e = r - y)$ The control signal is thus a sum of three terms: the P-term (which is proportional to the error), the I-term (which is proportional to the integral of the error), and the D-term (which is proportional to the derivative of the error). The controller parameters are proportional gain K, integral time T_i, and derivative time T_d. The integral, proportional and derivative part can be interpreted as control actions based on the past. Upon discretizing the system and applying it to our model, we will have the following computation:

$$\begin{bmatrix} v_x(t) \\ v_y(t) \end{bmatrix} = k_P \cdot \begin{bmatrix} x_{ref} - x(t) \\ y_{ref} - y(t) \end{bmatrix} + k_I \sum_0^t \begin{bmatrix} x(\tau) \\ y(\tau) \end{bmatrix} + k_D \cdot \begin{bmatrix} x(t) - x(t-1) \\ y(t) - y(t-1) \end{bmatrix} \tag{3}$$

Or using angular and linear velocity as the control vector, we will have the equivalent:

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = k_P \cdot \begin{bmatrix} d_{err}(t) \\ \theta_{ref}(t) - \theta(t) \end{bmatrix} + k_I \sum_0^t \begin{bmatrix} d_{err}(\tau) \\ \theta_{ref}(\tau) - \theta(\tau) \end{bmatrix} + k_D \cdot \begin{bmatrix} \Delta d(t) \\ \theta(t) - \theta(t-1) \end{bmatrix} \tag{4}$$

Where $d_{err}(t) = \sqrt{(x_{ref} - x(t))^2 + (y_{ref} - y(t))^2}$ and $\Delta d(t) = \sqrt{(x(t) - x(t-1))^2 + (y(t) - y(t-1))^2}$

2.3 LQR Control

In optimal control theory, the goal is to operate a dynamic system at a minimum cost. The dynamic system is described by linear differential equations and the cost is described by a quadratic function. This is called the LQ problem, which the solution of which is provided by the linear-quadratic regulator (LQR) controller [SP08]. The infinite-horizon discrete-time LQR can be described with a linear system (in our case the linear system is as provided in section 2.1):

$$s(t+1) = As(t) + B(s_t)u(t), \text{ where matrix B is time varying depending on } s_t \text{ (notation for } s(t)). \tag{5}$$

As denoted in the equation above, since matrix B is time varying (see section 2.1 for the state space model breakdown), when implementing, we will have to calculate the matrix at every time step with our current state (s_t) before moving to applying the LQR controller.

The performance index is defined as:

$$J_i = \sum_{t=0}^{\infty} ((s_i^* - s_t)^T Q (s_i^* - s_t) + u_t^T R u_t) \tag{6}$$

Where s_i^* denotes the the i-th waypoint in the waypoint path and J_i denotes the performance index for approaching that point.

As denoted in the equation above, since we approach every waypoint in the path one waypoint at a time, both s_i^* and J_i will remain constant for each LQR implementation.

The optimal control sequence minimizing the performance index is given by:

$$u_k = -F(s_i^* - s_t), \text{ where } F = (R + B(s_t)^T P B(s_t))^{-1}(B(s_t))^T PA) \tag{9}$$

P is the unique positive definite solution of the discrete time algebraic Riccati equation (DARE):

$$P_{t-1} = A^T P_t A - (A^T P_t B(s_t))(R + B(s_t)^T P_t B(s_t))^{-1}(B(s_t)^T P_t A) + Q \tag{10}$$

Applying these equations to our model (with the A and B matrices provided in subsection 2.1) while applying the different cost matrices will produce our control objective.

2.4 Waypoint Generation by Gradient Descent on Potential Functions

Reaching a possibly far-away reference point becomes much more tractable and realistic if instead the problem is broken down into reaching closer points (waypoints) approaching the reference. This is especially useful when accounting for obstacles. We decided to use a common approach in control that involves assigning objects in the robot's environment potential functions, and the robot generates waypoints by computing gradient descent on the superposition of the potential functions associated with all the objects in its environment. [LF01] In other words, robots move in the direction where the net potential decreases the most.

We conceptualize the environment as being a field in which all objects (including the reference point), depending on where they are relative to each other, have an associated potential. In this field, each Turtlebot has its own potential, and will seek to go in the direction in which its potential decreases the most. The robot is attracted to the reference because the potential function associated with that point has an absolute minimum there, and is steadily decreasing to that point. Obstacles such as other robots and objects have a potential which has a maximum at their position, so the robot would feel a repulsion force from them. Thus, as a result, the robots will get near the reference, and at the same time not collide.

We needed to choose the potential functions so that they were differentiable everywhere for the gradient descent to be possible, though it is not necessary for the center point of a robot when repelling another robot. The potential any robot in position (x, y) experiences from the reference point (x_{ref}, y_{ref}) we chose to be of the following form:

$$P_{ref}(x, y) = k((x - x_{ref})^2 + (y - y_{ref})^2) \tag{11}$$

Where k is a positive, real constant. This function is at a global minimum at the reference point. The potential any robot experiences from another robot i in position (x_i, y_i) is:

$$P_i(x, y) = \frac{k}{(x - x_i)^2 + (y - y_i)^2} \tag{12}$$

As described, each robot j has a total potential which is a superposition, or addition, of potentials from all objects.

$$P_{total}(x) = k((x - x_{ref})^2 + (y - y_{ref})^2) + \sum_{i \neq j} \frac{k}{(x - x_i)^2 + (y - y_i)^2} \tag{13}$$

What is convenient about our choice of potential functions is that their gradient has a relatively simple analytic form. For Equation 11, the gradient is:

$$\nabla P_{ref} = \langle 2k(x - x_{ref}), 2k(y - y_{ref}) \rangle \tag{14}$$

For Equation 12, the gradient is:

$$\nabla P_i = \langle \frac{-2k(x - x_i)}{((x - x_i)^2 + (y - y_i)^2)^2}, \frac{-2k(y - y_i)}{((x - x_i)^2 + (y - y_i)^2)^2} \rangle \tag{15}$$

By the linearity of the gradient, the gradient of the total potential a robot j experiences is

$$\nabla P_{total} = \nabla P_{ref} + \sum_{i \neq j} \nabla P_i \tag{16}$$

The waypoints are then calculated by discrete gradient descent of the net potential function:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \delta \cdot \nabla P_{total}(\mathbf{x}_t) \tag{17}$$

where $\mathbf{x}_t$ is the robot's current position and $\delta > 0$ is an arbitrary step size

Upon the implementation of the potential functions for the waypoint generation, we moved to choosing the different methods of approaching them:

1. PID waypoint tracking - calculating the next waypoint at each time step by gradient descent and approaching it with PID velocity control.

2. LQR trajectory tracking - calculating the entire waypoint path by gradient descent of these potential functions, iteratively approaching each successive waypoint with LQR control.

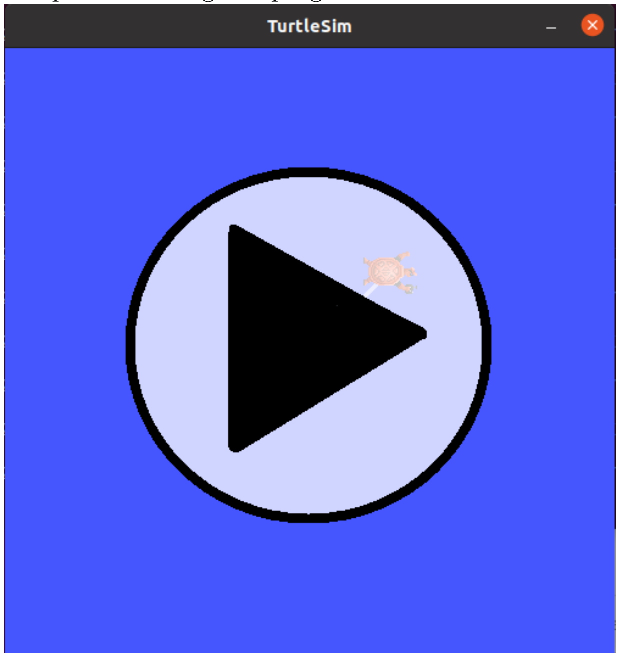
3 Implementation

3.1 Simple PID Control

First, we implemented reference tracking by using a simple PID controller. We implemented this in the most rudimentary simulation platform offered by ROS2, Turtlesim. Turtlesim is a simulation learning platform, which enabled us to get acquainted to our system.

When approaching the implementation, we had to establish our ROS2 graph. In our code, we had two nodes. The first, the simulation window, which is a predefined ROS2 node. The second, the turtle object's node, which is implemented by our code. The starting point, reference point, and constants were defined as user inputs and provided to the turtle object's node as configuration inputs. The turtle object's node received the objects x and y coordinates by subscribing to the Pose topic, which was published by the simulation window's node. Our code generated the movement of the simulated turtle by publishing the velocity in the x and y directions (as calculated by the PID control function's output which uses the theoretical logic provided in equation 3) to the Twist topic every clock-cycle (spanning a predefined Δt in seconds).

The following is an example of running our program:



Click the figure above to see the simulation video

Our code is located in the ros_orientation GitHub repository under turtlesim_pid_ctrl. Additional information regarding the code is provided in GitHub with the use of `.md` files as well as in-code comments.

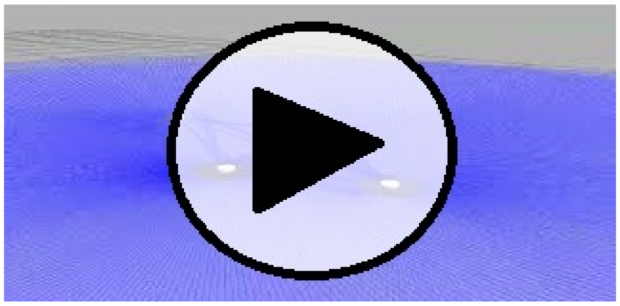
3.2 PID Waypoint Tracking

Initially we implemented single and multi-robot reference position tracking on a waypoint by waypoint basis using PID control. The simulation platform changed from the basic tutorial Turtlesim introduced in the ROS2 starter guide to Gazebo, a program that allows for more realistic modeling. We have already discussed the model we chose for the system, as well as the method we use to generate waypoints. The code for this approach can be found on the ros_orientation GitHub repository under gazebo_multisim_basic.

The simulation is fairly similar to real lab conditions, since in the lab the Turtlebots have knowledge of the whereabouts of other robots in almost real time, making use of the OptiTrack motion capture system. In that sense, the system is centralized given that it is not the robots that calculate the whereabouts of neighbors by processing sensor input. However, the robots are afterwards capable of making their own calculation of needed velocity. The transition to a decentralized approach would be to equip the robots with sensors to see incoming obstacles themselves.

The robots need to know each other's "pose" information, which includes their position on their shared coordinate system, as well as orientation. The way this was done in the ROS framework was by making the naming convention of the topics be `/robot_name/attribute`. For example, if a robot is named `turtle1`, then their pose is given in the topic `/turtle1/pose`. A `Turtles` node was created which acts as a storage and access point for the names of all robots. A robot's functionalities are encapsulated in the `Turtle` node, of which many can be instantiated with different names. A robot will post its name to a topic `/new_turtle`, which will be read by the `Turtles` node, which will subsequently update the `/all_turtles` topic containing a list of the names of all the robots online, which is accessed by all the robots. Thus, the simulation is capable of structurally supporting as many robots as can be generated by our computer's resources.

The next video shows 4 simulated Turtlebots using this method approaching a given reference `(5, 5)`. It can be seen that the robots get close to this point and remain in the vicinity without colliding. They remain in the regime of clustering around the reference, but they do not yet take on an tight, stationary formation, as we have not implemented that behavior yet.



Click the figure above to see the simulation video

The next version of this report will also include a video with the actual Turtlebots in the lab using this code. To make testing of current and future code on Turtlebots easier in the lab, we wrote a script that made sending commands of starting up and stopping the robots faster. The process of making a Turtlebot's resources available for use under ROS2 is known as "bringup," which as described on the Turtlebot website, is fairly cumbersome for multiple robot experiments. Initially, the process involved finding the IP of each robot on the WIFI network, and SSH-ing into them, resulting in many terminal instances on the user's side. Now, with the script one just needs the username and password of the host machine to send commands to them. The application is not limited to use in our lab, and the code and documentation can be found on the cntcn_utils repository on GitHub.

The code in simulation and for the one for the real robots remains relatively the same, except for there being no longer a need for generating the objects in a platform with their resources available for access and manipulation for ROS 2, as was done in the simulation, but instead one begins to communicate with the robots once their "bringup" script is run on them. The main challenge in the transition from simulation to physical robots rested on learning how to use and troubleshoot the Optitrack motion capture system which we use for determining the position and orientation information of the robots.

3.3 LQR Changing Reference Tracking

Upon the implementation of the PID Gradient Descent controller, we moved to a different method. In this project, our goal was both to experiment with linear-quadratic regulator (LQR) trajectory tracking as well as to ultimately have the robots arrive at a reference point. First, we implemented this on a single robot and later, we implemented this controller on multiple robot system.

3.3.1 Single Robot System

The single robot system included two nodes, one for the environment and one for the Turtlebot. The Turtlebot's node, "Turtle", was created which handled the subscription to the Turtlebot's position (the pose topic), the publication of the Turtlebot's calculated velocity (the cmd_vel topic), the generation of the waypoints path, and approaching the waypoint path's waypoints.

The function that calculated the waypoint path, "calc_waypoint_path", did so by calculating the discrete gradient descent of the reference potential function (see equation 11). This function provided the turtle node's class with a waypoint list attribute, which included the coordinates as well as angle of each waypoint in the path to the reference. Approaching the waypoints was completed by the "approach_waypoint" function, which was called by the "approach_waypoint_timer" callback function every dt. This function used the location of the robot provided by subscribing to the pose topic and changed the robot's velocity by publishing to the cmd_vel topic. The velocity was determined by calling an LQR function which calculated the velocities according to predetermined cost matrices, the system's model (see equation 1), and the current location of the robot. Upon handling a waypoint, the waypoint was discarded (by removing it from the waypoint list) leaving the next waypoint as the next objective.

The following is an example of running our program in the lab on an actual robot:

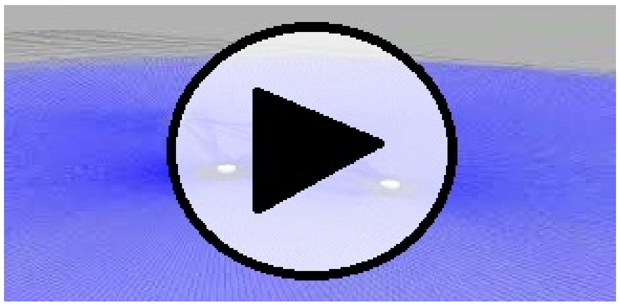
Click the figure above to see the video

Our code is located in the ros_orientation GitHub repository under gazebo_single_lqr_gradient_path (for the Gazebo simulation code), and real_single_lqr_gradient_path (for the in lab on a Turtlebot code). Additional information regarding the code is provided in GitHub with the use of `.md` files as well as in-code comments.

3.3.2 Multi Robot System

In a multi robot system, we must add collision avoidance. To do so, each robot must know the location of the other robots in the system. The same method which was used in the "PID Waypoint Tracking" project (section 3.2) was used in this project. The naming convention was maintained and a "Turtles" node was created which acts as a storage and access point for the names of all robots. A robot will post its name to a topic "/new_turtle", which will be read by the "Turtles" node, which will subsequently update the "/all_turtles" topic containing a list of the names of all the robots online, which is accessed by all the robots. Therefore, every robot can access the other robot's position by subscribing to the topic "/<turtle_name>/pose". The "Turtle" node's code will be similar to the single robot's system. Each robot's movement will still be handled by a "Turtle" type node. The generation of waypoints will stay exactly the same; however, when approaching the waypoints, if multiple robots are close to each other, a repulsive potential function will be introduced (see equation 10) and the the waypoint path of the relevant robots will be modified to be the discrete gradient descent of the net potential function of both the reference and the repulsive potential functions (see equation 11).

The following is an example of running our program:



Click the figure above to see the simulation video

Our code is located in the ros_orientation GitHub repository under gazebo_multi_lqr_gradient_path. Additional information regarding the code is provided in GitHub with the use of `.md` files as well as in-code comments.

4 Next Steps

There are many possibilities for the next project for us to expand upon. Some of these include:

1. For multi-robot systems, in addition to collision avoidance, designing and implementing algorithms for how the robots should arrange themselves closely around a specific point. It should adapt based on the number of robots working together.
2. Making the conditions faced by the robots more realistic. Instead of being given the positions of other objects, they could use sensors or computer vision techniques to understand their surroundings independently.
3. Giving specific path shapes for the LQR controller to follow. This includes describing more complex paths programmatically (not ones generated by potential function gradient descent), as well as concretely evaluating how well the robot follows the path. Currently the implementation is promising but there are some bugs in the code that need to be sorted before moving on.
4. Adding on to the previous goal, we would also like to expand the implementation to multiple robots. That is, study methods for robots to react appropriately when encountering obstacles in their path (be it other robots or immovable obstacles). We propose to apply some more potential function ideas to generate a reasonable deviation from the path (if that is deemed the proper course of action), and then get back on it.

References

[AM21a] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2021. Chapter 2, http://www.cds.caltech.edu/~murray/amwiki/index.php?title=System_Modeling.

[AM21b] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2021. Chapter 10, http://www.cds.caltech.edu/~murray/amwiki/index.php?title=PID_Control.

[LF01] Naomi Ehrich Leonard and Edward Fiorelli. Virtual leaders, artificial potentials and coordinated control of groups. *2001 IEEE*, pages 2968–2970, 2001.

[Rob21] Open Robotics. Ros2 concepts. 2021. https://docs.ros.org/en/foxy/Concepts.html.

[SP08] A.J Shaiju and Ian R. Petersen. Formulas for discrete time lqr, lqg, leqg and minimax lqg optimal control problems. 41, Issue 2:8773–8778, 2008.