LINEAR-QUADRATIC REGULATOR DESIGN FOR THE CONTROL OF A DIFFERENTIAL DRIVE GROUND ROBOT: QUANSER QBOT 2

_____

A Thesis

Presented to the

Faculty of

San Diego State University

_____

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Mechanical Engineering

_____

by

Vittorio Longi

Fall 2018

SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Vittorio Longi:

Linear-Quadratic Regulator Design for the Control of a Differential Drive Ground Robot: Quanser QBot 2

Kaveh Akbari Hamed

Kaveh Akbari Hamed, Chair
Virginia Polytechnic Institute and State University

Peiman Naseradinmousavi
Department of Mechanical Engineering

Ping Lu
Department of Aerospace Engineering

9/13/2018

Approval Date

DEDICATION

This work is dedicated to Jack and Marianne Moore. Without their loving support, inspiring encouragement and endless generosity, this thesis, as well as my entire studies at San Diego State University, would not have been possible.

ABSTRACT OF THE THESIS

Linear-Quadratic Regulator Design for the Control of a
Differential Drive Ground Robot: Quanser QBot 2
by
Vittorio Longi
Master of Science in Mechanical Engineering
San Diego State University, 2018

Autonomous ground vehicles have gained considerable popularity in recent years for a wide variety of applications, in fields such as the industrial, logistics, and agricultural. One class of these vehicles in particular has gained traction for its simple yet robust structure – the differential drive robot.

This study investigates the development of optimal control of differential drive ground robots using linear quadratic regulators (LQR) to track predetermined circular paths. To implement this type of controllers, the system is first modeled using the state-space approach, linearized and then proven to be controllable. Simulations are performed using MATLAB and Simulink to show the effectiveness of the controller.

Because the time-varying LQR gains are computed offline at a fixed discrete time interval by solving the Riccati equation, their use in an environment with faster sample rates is compromised. To overcome this issue, a set of artificial neural networks (ANNs) is implemented to learn the pattern of each LQR gain. The networks, trained via backpropagation, are then used to generate gain values at any sample rate.

Finally, the developed controller is experimentally evaluated on an actual robot, a Quanser QBot 2. The LQR controller created for the simulation is adapted for use in real-time. To measure the position and orientation of the vehicle in real-time, six high-speed infrared cameras (OptiTrack Flex 3) track the reflective markers mounted on the robot and communicate the data to the controller. During the experiments, data is recorded and later compared with simulations to prove the successful performance of the controller.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Dr. Kaveh Akbari Hamed for his astounding guidance and patience, both of which were essential for me to complete this work.
Special thanks go to Julian Espinoza and to Alex Lee for their invaluable advice and help in the lab, as well as to all faculty members of the College of Engineering who imparted me the knowledge I needed to develop this thesis.

I am grateful to my family, my mother Cristina, my father Paolo and my sister Federica, for their love and continuous encouragement throughout the years. I am also thankful to Josephine, who lovingly supported me and put up with me during the hard times. Thank you all for always believing in me.

CHAPTER 1

INTRODUCTION

Differential drive robots are mobile ground vehicles characterized by two independently powered wheels mounted symmetrically along a central axis, and a number of castor wheels for balancing [1]. They are generally small in height, as shown in Fig. 1, allowing for several sensors or actuators to be mounted onto them. Their structural simplicity, sturdiness and reliability, paired with the innovative technologies from the field of artificial intelligence, has made them increasingly popular for a wide variety of applications.

In the recent years, their use as autonomous ground vehicles has gained striking popularity in many different fields. Primarily operated indoors, these robots are becoming widely used for applications in logistics automation, such as in large warehouses, distribution centers and factories, to transport items, containers or stacked shelves. The rise in popularity is reflected in the copious investments that companies such as Amazon, DHL and UPS have made in the last years for expanding this technology. The increasing need for automation in warehouses, combined with the fact that only 5% of current warehouses in the U.S. are automated [2], suggests that there is a strong market for this type of robots, as companies are striving to improve efficiency while minimizing costs.

Figure 1. Example differential-drive robot for warehouse automation: The OTTO 1500. Source: [3].

Autonomous ground vehicles seem to therefor have a relevant role in the developing "Industry 4.0" trend and in the Internet of Things (IoT) technology. Organizations in fields other than the industrial and logistics are channeling resources in developing smart ground vehicles for use in outdoor environments, for applications such as: agriculture (e.g. up-keeping and harvesting of produce [4]), space (e.g. planetary surface exploration [5]), consumer (e.g. building surveillance and vacuum cleaning [6]), postal (e.g. delivery of packages or food [7]), medical (e.g. automated wheelchairs [8]), recreational (e.g. intelligent toys or utility tools [9]) and educational (e.g. robots to introduce programming to the young [10]).

Swarm robotics, a new approach for coordinating the simultaneous movement of multiple inter-connected robots, is also being explored to perform tasks that a single robot would not be able to accomplish.

Despite their simplicity, the state feedback control of differential drive robots is not trivial, as this thesis will demonstrate. Note that the term "differential drive" has a different meaning in automotive engineering, where it is used to indicate that the power is split between two or more wheels through the use of a mechanical differential.

Fundamentally, autonomous ground robots have the potential to greatly contribute to society. Consider a scenario where traditional indoors (i.e. warehouses and factories) transportation methods, such as human-operated machinery or vehicles with internal-combustion engines, were completely replaced or at least partially complemented by these robots. Consider the case where autonomous ground robots can be fully adapted for short or medium range outdoors transportation, ideal for urban or suburban environments. Their impact on society would be highly beneficial. Lower energy consumption guaranteed not only by the emission-free electric motors, but by the use of efficiency-maximizing and time-minimizing control and motion planning algorithms, would be an essential step for the shift towards more environmentally sustainable societies.

LITERATURE REVIEW

Autonomous ground vehicles, in particular those with differential drive structure, have been studied and developed for decades, making their documentation vast and detailed. Drastic innovations achieved in recent years in the fields of computer science and artificial

intelligence have allowed for better performance of these robots and for their application in numerous new fields. In this section, some of the recent innovations regarding this type of robots are presented, to have a better understanding of the current stage of development of this technology. The analysis will focus on the areas of nonholonomic modeling, control, motion planning and artificial intelligence, all with a particular focus on ground vehicle applications.

Before developing a controller, determining the model of the system is usually a prerequisite. For nonholonomic mobile vehicles, such as the differential drive ground robots, this task has been well accomplished and studied. Complete kinematic models that simplify the motion of the robot and assume that the wheels do not slip [11], as well as dynamic models that include nonholonomic and dynamic constraints [12], have been developed. Autonomous ground vehicles are generally manufactured with either two driving wheels with castors or with four driving wheels. The kinematic models of both setups have been studied and compared [13]. While the no-slip assumption is commonly used for the development of controller, as in the case of this thesis, the effect of wheel slip and traction forces in relation to the static and kinetic friction coefficients have also been examined for this type of robots. To complete this study, techniques to avoid wheel slippage by finely modulating the torque applied to the wheels have also been developed [14]. Some studies, on the other hand, focus on the more challenging task of modeling the system with a more detailed and realistic approach, such as, for example, a recent study in which the modeling of the nonholonomic robot included both longitudinal and lateral slip [15]. The approaches described so far base the system model on the coordinates of the vehicles. This is not the only way of describing the system. Some studies use alternative methods such as, for example, using the travelled distance measured from the wheel rotation to develop the model [16].

Once the model of the system is determined, the options for developing a controller are numerous. The well-known proportional-integral-derivative (PID) controller has been successfully applied to all kinds of ground robots. One example is the control of an all-terrain vehicle, achieved through nonlinear modeling and subsequent linearization of the model followed by the development of the PID controller [17]. Because the challenging task of using PID controllers is determining the values for its three gains, different studies have been conducted to determine effective methods for tuning these values. In one study, genetic

algorithms, which mimic the natural selection and evolution of successful traits in living beings, has shown to be successful at finding near-optimal gains for a ground vehicle PID controller [18]. Fuzzy logic has also been explored as an option for integrating human-like decision-making, expressed through the definition of a set of rules, into the control method for ground robots [19]. Alternatively, fuzzy logic has been shown to be successful when applied to PID controllers. The resulting controllers, called fuzzy-PID controllers, are able to accomplish the same task as regular PID controllers by using a more human-like reasoning process [20]. Fuzzy logic has also been successfully used for tuning the PID gains, as an alternative to the genetic algorithm method, in a speed controller for ground vehicles [20]. The method of model predictive control (MPC), an alternative to the LQR method, has been studied to optimally control ground vehicles. When dealing with nonlinear systems such as ground vehicles, the nonlinear MPC has been developed for dealing with the higher complexity of the model [21]. Alternatively, the model can first be linearized, as was done in a study where a controller based on Kautz functions was developed for improved trajectory tracking and disturbance rejection [22]. Finally, the recently-thriving field of machine learning has also been involved in the control of ground vehicles. For example, in a study focused on using reinforcement learning (RL), the controller produced system inputs that were continuously evaluated and rated to provide feedback to the controller, which in turn adjusted itself to become better at performing its task [23]. A comparison of some of the above-mentioned control methods, as well as additional ones, has also been constructed in order to identify which controller is most suitable for different scenarios [24]. The stability of nonholonomic vehicles has also been well studied using different approaches, such as with Lyapunov analysis [25] and port-Hamiltonian systems theory [26].

The next step after modeling the system and developing a controller is to make a ground robot become autonomous. This usually involves the integration of some kind of path planning algorithm that informs the robot on which way to navigate to. When the map of the environment is already known, and the robot has a way of precisely knowing its position within that environment, the main challenge is determining a feasible path to reach a goal while minimizing the travelled distance (or other cost functions, such as energy expenditure). A popular algorithm for determining such trajectory is known as the A* algorithm. This technique can be modified and combined with other methods, such as in a recent study where

a hybrid A* algorithm was used together with potential fields to determine the best path in an environment with obstacles of different relevance [27]. Because the search algorithms that determine the best plan are often computationally expensive, entire studies have been focused on improving the performance of the algorithms, such as a recent study that was able to cut down the search process by half [28]. More complex methods for determining optimal paths involve the combination of several techniques, such as in a study where probabilistic roadmaps were combined with hierarchical genetic algorithms to produce the optimized path [29]. Other studies focused on simply improving the availability and accuracy of positioning data from built-in sensors or outside positioning sources, such as in a study where neural networks were used to intelligently help the robot know its position in environments with poor GPS signal [30].

Recent studies have also focused on the scenario in which the robot does not know its location or environment beforehand, a problem known as simultaneous localization and mapping (SLAM). The goal is for the robot to map its environment, locate itself within it and safely navigate through it to reach a certain goal [31]. Several methods to tackle this problem exist, each with its strengths and drawbacks. Machine learning techniques such as deep reinforcement learning have been explored to guide a robot to a goal while avoiding collisions with other objects without providing the robot with any prior knowledge [32]. Finally, an additional level of difficulty is represented by an unknown environment in which objects are continuously changing their position. One solution was proposed by a study in which a fuzzy logic algorithm was implemented to allow the robot to switch between different control behaviors in order to determine the best course of action when facing various situations [33].

GOALS AND OBJECTIVES

The goal of this thesis is to design a Linear Quadratic Regulator (LQR) to control a differential-drive robot and demonstrate its correct functioning by testing it on an actual vehicle. This type of controller guarantees that the control input is optimal and that a specified performance index is satisfied. The implementation of the LQR for differential drive ground robots, in particular for the Quanser QBot 2, has only scantly been explored and deserves therefor to be investigated more in detail.

Moreover, solving the Riccati equation using standard computational methods, such as MATLAB's ODE solvers, yields discrete-time values for the LQR gains. When running the controller in real-time on an actual robot, the sample time might be faster than the time increments used to compute the LQR gains, which is a computationally expensive task. To overcome this issue, several solutions are possible. One solution consists of approximating the gain values with a polynomial. Another solution, which this thesis will focus on, is to train a neural network to learn the LQR gains. Once trained, the network can be used to generate accurate gain values for any given time (within the valid interval), which allows the use of the LQR in applications with fast sample rates.

Before developing the LQR, the system first needs to be mathematically modeled. Then, after a nominal trajectory is chosen as the desired path that the robot must follow, the system is linearized and expressed using the state-space representation. All of these steps are implemented in a MATLAB program and in Simulink models, in order to numerically evaluate and analyze the system's response. Finally, the developed controller is tested on an actual robot, a Quanser QBot 2, to verify that its response is in agreement with the theory and computational simulations.

Modeling

The motion of the QBot 2 can easily be modeled using the kinematics approach by relating the wheels' angular velocities to the robot's three coordinates on the ground plane: its position measured along the x and y axes of the Cartesian plane, and its orientation θ measured counter-clockwise from the x-axis. The differential drive model will be used to derive the governing kinematic equations of the system and will then be translated into the more useful and simple unicycle model for designing the controller.

Using kinematics instead of the dynamics approach implies that a number of simplifications and assumptions are made. Friction between the wheels and the ground is not considered (known as pure rolling without slipping assumption). No forces or torques are included in any of the modeling. Inertia is also ignored, thus assuming that the moving robot immediately stops after cutting the power from the wheels. These assumptions are all valid as they do not have any significant impacts on the results under standard operating conditions.

LQR Design Problem

The control of differential drive robots has been thoroughly studies with traditional methods, such as with proportional-integral-derivative (PID) controllers. While these techniques successfully stabilize and guide the robot, they lack to consider the energy expenditure element. For this reason, a great candidate controller for differential drive robots is the Linear Quadratic Regulator, an important subject in optimal control theory based on the state space representation.

This controller implements a quadratic cost function J, also known as performance index, whose minimum is attempted to be found by solving a Riccati equation. The cost function contains three tweakable parameters, the weighting control matrices F, Q and R, which penalize the difference in the system's state from the desired state and the input control signals. By modulating their weight inside the cost function, they ultimately affect the system's response.

The output of the LQR is the time-varying optimal gain vector k_{lqr}, which contains the two coefficients that multiply the robot's forward velocity and angular velocity before the signals are sent to the actuators. The objective of this thesis is to make use of the gains produced by the LQR in real-time to guide the Quanser QBot 2 robot around a pre-defined circular path.

Neural Network

Once the time-varying LQR gains are computed offline by solving the Riccati equation for the entire run time, they can be used for simulations and during the actual testing on the robot. To be used in fast sample rate environments, the curves obtained by plotting the LQR gains over time can each be used to train an equal number of backpropagation neural networks (BPNN).

For this application, the networks have a single input (the current time) and a single output (the corresponding LQR gain value). By having enough layers and neurons, the networks can very closely replicate the original curves. The advantage is that the networks are not restricted to producing pre-computed (time discretized) gain values but can generate an approximate value for any given time. That is, as long as the given time is within the range that the networks were trained on. Unfortunately, because of the high computational

power required for using neural networks, which is beyond what the lab computers are capable of, the neural networks are only implemented for the simulation part of this thesis and not for the real-time experiments. Nonetheless, the principles and code setup are still valid for real-time applications.

The advantage of using a neural network is that the time-dependent LQR gains can be computed with a lower sample rate, thus speeding up computation time, while still allowing to be used in environments with much faster sample rates, such as for real-time applications.

Real-Time Controller

To test the system on an actual robot, a real-time controller, developed in the Simulink environment, is constructed. The controller uses the actual position of the robot and the nominal position at the current time to first calculate the positioning error, which is a vector of three values, containing the error of each position (x and y) and orientation (theta) variables. The positioning error is then multiplied by the LQR gains, previously computed offline, and the resulting vector is subtracted from the nominal input values. The new vector, consisting of two values, corresponds to the inputs signals to be sent to the plant. This is, in short, the overall functioning of the controller. Its input and output parameters are visualized in Fig. 2 and its structure is analyzed in more detail in the Experimental Setup section.

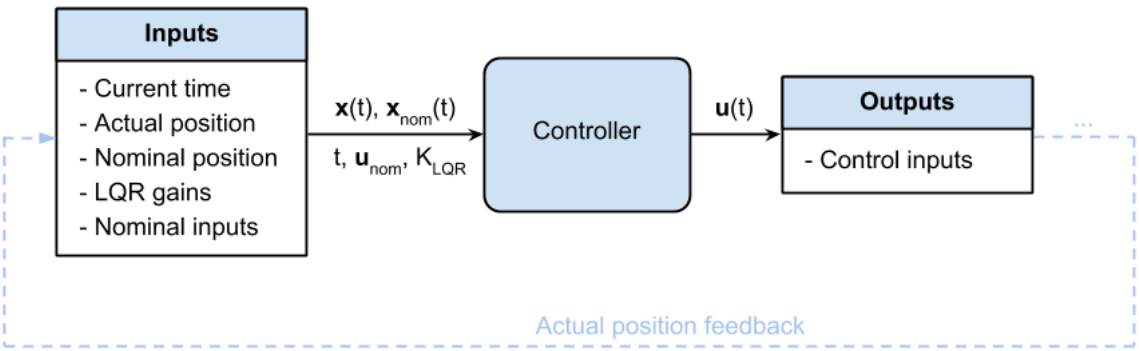

Figure 2. Inputs and outputs to the real-time controller.

Hardware Description

The theoretical content of this thesis is in general applicable to the entire family of differential-drive robots. However, for the experimental part, one specific device had to be

selected and the developed conclusion may therefore be only applicable to this device specifically.

The chosen subject of this thesis is the QBot 2, an "open-architecture autonomous ground robot" made by Canadian-based Quanser. It is built on the iClebo Kobuki platform, a mobile battery-powered robotic base with several built-in features [34]. It is the same platform used for certain products by the parent company, Yujin Robot, based in South Korea, which specializes in the development of autonomous vacuum cleaners and assembly lines [35]. It includes a variety of sensors, such as gyroscope, bumper, cliff, and wheel-drop sensors. The base is driven by two differential-drive wheels with built-in encoders (left and right) and is balanced by two caster wheels (front and rear) [36]. The maximum linear velocity achievable by the robot (i.e., the maximum linear velocity of each driving wheel) is 0.7 m/s.

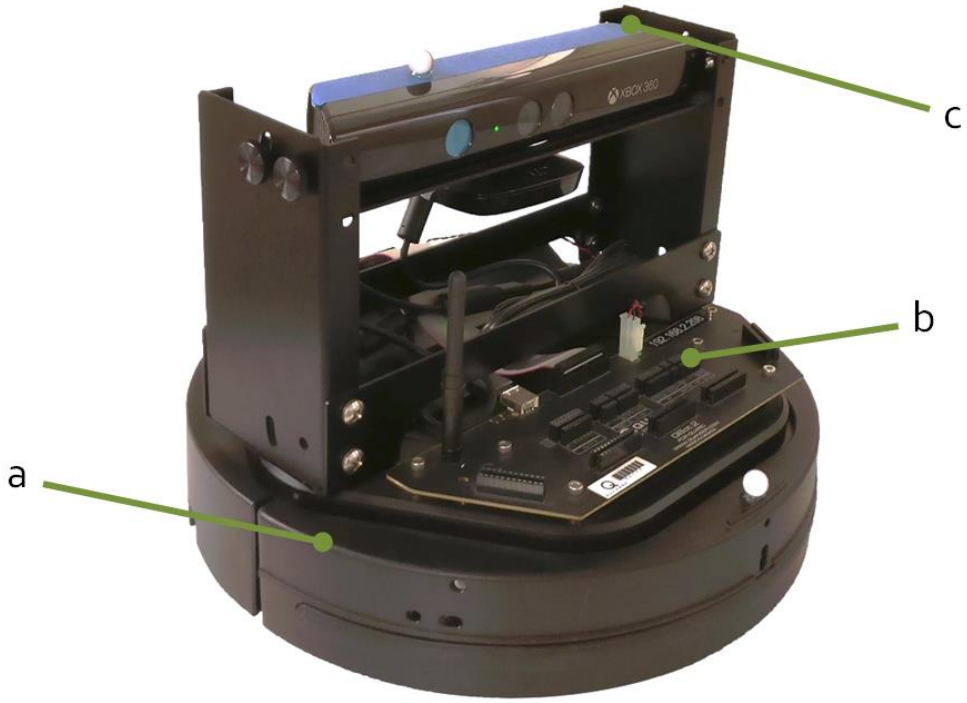

Figure 3. QBot 2 robot with its main components: (a) Kobuki base, (b) on-board computer, (c) Kinect sensor.

Quanser expanded this device's capabilities with the addition of a low power on-board Linux embedded computer (Gumstix DuoVero), which allows the robot to wirelessly communicate in real-time with a host computer. This computer must be preloaded with

Quanser's QUARC Real-Time Control Software, which integrates seamlessly with MATLAB and Simulink.

The QBot 2 is also equipped with a Microsoft Xbox 360 Kinect sensor, containing an RGB camera and a 12-bit depth sensor that give the robot the ability to perceive its surroundings in great detail. A data acquisition card (DAQ) supporting both digital and analog I/O ports is also present on the robot, for connecting additional sensors or actuators. The entire system is powered by a Lithium-ion battery pack, making its operation completely wireless.

Table 1. Critical Dimension of the QBot 2

Symbol	Dimension	Measurement
r_{wheel}	Wheel radius	0.035 m
D	Robot diameter	0.35 m
L	Distance between wheels	0.235 m

Source: [37].

CHAPTER 2

THEORETICAL BACKGROUND

The QBot 2 robot is a nonlinear dynamical system. It has three outputs (position in two axes and orientation) and two inputs (left and right wheel velocities), making it a multiple-input multiple-output (MIMO) system. Because of the presence of kinematic constraints on velocity, the system is non-holonomic [38]. Physically, this can be explained by the fact that the robot cannot move freely in any arbitrary direction (e.g. it cannot move laterally), but it is limited to rotational and forward/backward motion, as opposed to omnidirectional motion. Also, we note that the number of controllable degrees of freedom (2 inputs) is less than the total degrees of freedom (3 outputs).

DIFFERENTIAL-DRIVE MODEL

As the name suggests, differential-drive robots are able to move in various directions by modulating the difference between the wheel velocities. The most physically-accurate way of describing the motion of such robots is the differential-drive model, based on the kinematics of the two wheels [39]. The model is grounded on the simple equation that relates linear velocity v and angular velocity ω of a wheel with radius r_{wheel}.

$$\omega = \frac{v}{r_{wheel}} \tag{1}$$

As shown in Table 1, $r_{wheel} = 0.035\ m$ for the QBot 2 robot. When solved for v, this equation allows to not only know an object's current linear (ground) velocity, but also, indirectly, to know how far it has travelled since some initial reference time.

One important assumption is necessary before continuing the description of this model, that is, the ground on which the robot travels is taken to be perfectly flat. In other words, the robot's motion is limited to a 2D Cartesian plane and its location is only

determined through x and y coordinates. The third-dimensional elevation coordinate z is assumed to be constant at zero, and thus is not taken into account.

With this assumption in mind, the linear velocity of the robot can now be split into its two components, measured with respect to the x and the y coordinates respectively.

$$\dot{x} = \frac{r_{wheel}}{2} (\omega_R + \omega_L) \cos\theta \tag{2}$$

$$\dot{y} = \frac{r_{wheel}}{2} (\omega_R + \omega_L) \sin\theta \tag{3}$$

For the rotation of the vehicle, the standard convention of measuring angles counter-clockwise from the x –axis is used. The angular velocity of the robot can then be determined.

$$\omega = \dot{\theta} = \frac{r_{wheel}}{L} (\omega_R - \omega_L) \tag{4}$$

Note that, for example, when ω_L is rotating at a faster rate than ω_R, the angular velocity becomes negative, which is in agreement with the chosen angle-measurement convention. Also note that when the angular velocities of the wheels are the same, $\dot{\theta}$ becomes zero, thus indicating that the robot is moving in a straight line (or stationary).

UNICYCLE MODEL

A useful simplification of the differential-drive approach is known as the unicycle model [40]. This representation is more natural and intuitive to visualize since it is based directly on the linear (center) velocity v_C and the angular velocity ω of the entire vehicle.

$$\dot{x} = v_C \cos\theta \tag{5}$$

$$\dot{y} = v_C \sin\theta \tag{6}$$

$$\dot{\theta} = \omega \tag{7}$$

The separate wheel velocities are now unnecessary for representing the dynamics of the robot. However, while this model is useful for visualization and designing of a controller, it cannot be used to directly send control signals to the vehicle (the physical motion of the robot still depends on the two wheel velocities). Note again that the system is nonholonomic because there exist velocity constraints, such as, for example, the relation between $\dot{x}$ and v_C.

In fact, $\dot{x}$ is constrained by the v_C through $\cos\theta$ and thus they cannot be considered independent from each other.

It is then necessary to have a way to convert from unicycle model back to the differential-drive model velocities.

$$\omega_R = \frac{2v_C+\omega L}{2r} \tag{8}$$

$$\omega_L = \frac{2v_C-\omega L}{2r} \tag{9}$$

Due to the benefits of using the unicycle model, this representation will be the primary one used throughout this text, unless otherwise noted, and its symbols are summarized in Fig. 4.

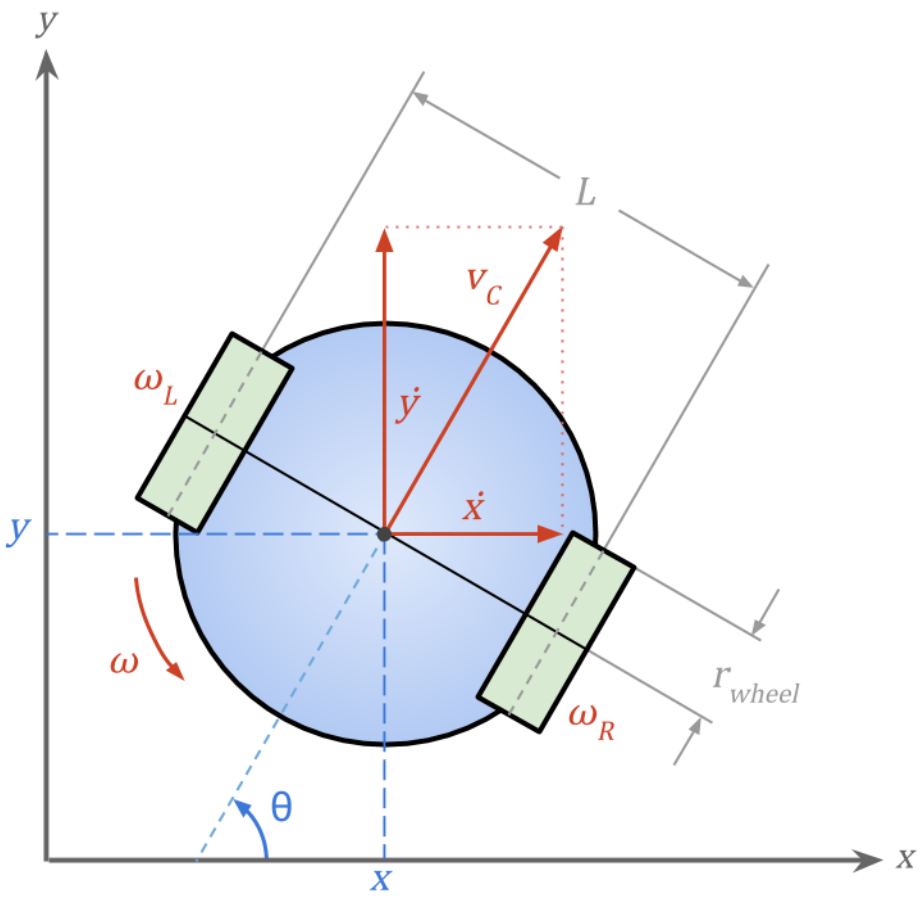

Figure 4. Coordinate frame and symbols used for the QBot in the unicycle model.

NOMINAL TRAJECTORY

Before developing the state-space representation of the system, it is necessary to convert it to a linear time varying (LTV) system by linearizing it around a specific nominal (desired) trajectory. The latter could be any arbitrarily-chosen path, describable by a continuously-differentiable and continuous function.

For this thesis, a counterclockwise (CCW) circular path was chosen, as it is simple yet representative of a wide range of applications. For example, this path shape can be used for initial scanning and mapping of an unknown environment, using the RGB and depth sensors of the QBot 2, by sweeping 360 degrees around the origin. The trajectory is chosen as a circle centered at the origin, with radius r, and is given by:

$$x^2 + y^2 = r^2 \tag{10}$$

A robot moving along such path at constant velocity will have constant tangential velocity v_C and constant angular velocity ω_0. The angular velocity depends on the value chosen for the period T, which represents the time it takes for the robot to complete one full rotation around the path (expressed in seconds), as described by the formula:

$$\omega_0 = \frac{2\pi}{T} \tag{11}$$

Then, the two velocities can be related by the following equation:

$$v_C = r\omega_0 = \frac{2\pi r}{T} \tag{12}$$

To obtain the total distance travelled by the robot on the x-y plane, the tangential velocity is multiplied by the elapsed time t.

$$Travelled\ distance \coloneqq \frac{2\pi r}{T} t = r\omega_0 t \tag{13}$$

The total distance can then be split into the total distances travelled along each axis, $x(t)$ and y(t). These components will be used as the nominal positions tracing the desired circular trajectory and will therefore be represented by $x^*(t)$ and $y^*(t)$ from now on.

$$x^*(t) = r\cos(\omega_0 t) \tag{14}$$

$$y^*(t) = r\sin(\omega_0 t) \tag{15}$$

A desired angular position for the robot must also be specified. In general, to compute the angular position of an object along a circular trajectory, measured from the positive x-axis, this formula is used:

$$\theta(t) = \omega_0 t \tag{16}$$

In this case, when the robot is driving counterclockwise around the circle, its heading direction (also measured from the positive x-axis) should always be tangent to the circle, which corresponds to 90° more than its angular position. Thus, the nominal angular position is set to:

$$\theta^*(t) = \frac{\pi}{2} + \omega_0 t \tag{17}$$

Now that the equations for $x^*(t)$, $y^*(t)$ and $\theta^*(t)$ have been obtained, their derivatives with respect to time can be taken to find the nominal velocities, $\dot{x}^*$, $\dot{y}^*$ and $\dot{\theta}^*$.

$$\dot{x}^* = v_C \cos\left(\theta + \frac{\pi}{2}\right) \tag{18}$$

$$\dot{y}^* = v_C \sin\left(\theta + \frac{\pi}{2}\right) \tag{19}$$

$$\dot{\theta}^* = \omega_0 \tag{20}$$

STATE-SPACE REPRESENTATION

Now that the equations of motion and the nominal trajectory have been defined, the state-space representation can be derived. This representation is useful as it allows to model the system as sets of state variables, inputs and outputs with first-order differential equations, and, later on, to develop the linear quadratic regulator.

First, the state variables x_1, x_2, x_3 are chosen. Three is the minimum number of them needed to fully describe the system (two for position, one for orientation). They are collected in the state vector $\boldsymbol{x}(t) \in \mathbb{R}^3$.

$$\boldsymbol{x}(t) \coloneqq \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} \tag{21}$$

Similarly, the nominal state variables are collected in the nominal state vector $\boldsymbol{x}^*(t)$.

$$\boldsymbol{x}^*(t) \coloneqq \begin{bmatrix} x_1^*(t) \\ x_2^*(t) \\ x_3^*(t) \end{bmatrix} = \begin{bmatrix} x^*(t) \\ y^*(t) \\ \theta^*(t) \end{bmatrix} = \begin{bmatrix} r\cos(\omega_0 t) \\ r\sin(\omega_0 t) \\ \frac{\pi}{2} + \omega_0 t \end{bmatrix} \tag{22}$$

The input (or control) vector $\boldsymbol{u}(t) \in \mathbb{R}^2$ contains the parameters that allow the robot to be actuated. Using the unicycle model, the input vector $\boldsymbol{u}$ and the seemingly identical but constant nominal input vector $\boldsymbol{u}^*$ are defined as:

$$\boldsymbol{u}(t) \coloneqq \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} v_C \\ \omega \end{bmatrix} \qquad \boldsymbol{u}^* \coloneqq \begin{bmatrix} u_1^* \\ u_2^* \end{bmatrix} = \begin{bmatrix} r\omega_0 \\ \omega_0 \end{bmatrix} = \begin{bmatrix} v_{C,0} \\ \omega_0 \end{bmatrix} \tag{23}$$

The output vector $\boldsymbol{y}(t) \in \mathbb{R}^3$ represents the measurable quantities output by the system. In this case, it is defined the same way as the state vector.

$$\boldsymbol{y}(t) \coloneqq \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} \tag{24}$$

Recall that the equations of motions described in the Unicycle model section. They can be combined in the vector $\boldsymbol{f}(t)$, also symbolized by $\dot{\boldsymbol{x}}(t)$, and its variables are replaced by the state variables:

$$\boldsymbol{f}(t) \coloneqq \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} v_C\cos\theta \\ v_C\sin\theta \\ \omega \end{bmatrix} = \begin{bmatrix} u_1\cos x_3 \\ u_1\sin x_3 \\ u_2 \end{bmatrix} \tag{25}$$

To derive the state matrix $A(t) \in \mathbb{R}^{3x3}$ and the input matrix $B(t) \in \mathbb{R}^{3x2}$, the Jacobian matrix of $\boldsymbol{f}(t)$ with respect to the state vector and to the input vector respectively are first required.

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\big(\boldsymbol{x}(t), \boldsymbol{u}(t)\big) \coloneqq \begin{bmatrix} \frac{\partial \dot{x}}{\partial x_1} & \frac{\partial \dot{x}}{\partial x_2} & \frac{\partial \dot{x}}{\partial x_3} \\ \frac{\partial \dot{y}}{\partial x_1} & \frac{\partial \dot{y}}{\partial x_2} & \frac{\partial \dot{y}}{\partial x_3} \\ \frac{\partial \dot{\theta}}{\partial x_1} & \frac{\partial \dot{\theta}}{\partial x_2} & \frac{\partial \dot{\theta}}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 0 & 0 & -u_1\sin x_3 \\ 0 & 0 & u_1\cos x_3 \\ 0 & 0 & 0 \end{bmatrix} \tag{26}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\big(\boldsymbol{x}(t),\boldsymbol{u}(t)\big) \coloneqq \begin{bmatrix} \frac{\partial \dot{x}}{\partial u_1} & \frac{\partial \dot{x}}{\partial u_2} \\ \frac{\partial \dot{y}}{\partial u_1} & \frac{\partial \dot{y}}{\partial u_2} \\ \frac{\partial \dot{\theta}}{\partial u_1} & \frac{\partial \dot{\theta}}{\partial u_2} \end{bmatrix} = \begin{bmatrix} \cos(x_3) & 0 \\ \sin(x_3) & 0 \\ 0 & 1 \end{bmatrix} \tag{27}$$

The A and B matrices are then found by evaluating the Jacobian matrices at the nominal state and input variables.

$$A(t) \coloneqq \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\big(\boldsymbol{x}^*(t),\boldsymbol{u}^*(t)\big) = \begin{bmatrix} 0 & 0 & -u_1^* \sin x_3^* \\ 0 & 0 & u_1^* \cos x_3 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -\sin\left(\frac{\pi}{2}+\omega_0 t\right) \\ 0 & 0 & \cos\left(\frac{\pi}{2}+\omega_0 t\right) \\ 0 & 0 & 0 \end{bmatrix} =$$

$$= \begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix} \tag{28}$$

The input matrix $B(t) \in \mathbb{R}^{3x2}$ can be similarly obtained.

$$B(t) \coloneqq \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\big(\boldsymbol{x}^*(t),\boldsymbol{u}^*(t)\big) = \begin{bmatrix} \cos(x_3^*) & 0 \\ \sin(x_3^*) & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{\pi}{2}+\omega_0 t\right) & 0 \\ \sin\left(\frac{\pi}{2}+\omega_0 t\right) & 0 \\ 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} -\sin(\omega_0 t) & 0 \\ \cos(\omega_0 t) & 0 \\ 0 & 1 \end{bmatrix} \tag{29}$$

The A and B matrices will be essential for developing the LQR. One important consideration has to be made at this point. While the system is nonlinear by nature, it can now be treated as Linear Time Varying (LTV) as long as the position and orientation of the robot are close to the specified nominal circular trajectory.

The QBot 2 system can now be expressed using the state-space representation with the following state equation, which is the matrix notation for a set of first-order ordinary differential equations (ODEs).

$$\dot{\boldsymbol{x}}(t) = A(t)\boldsymbol{x}(t) + B(t)\boldsymbol{u}(t), \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{30}$$

This expression can be more explicitly written out as

$$\begin{bmatrix}\dot{x}(t)\\ \dot{y}(t)\\ \dot{\theta}(t)\end{bmatrix} = \begin{bmatrix}0 & 0 & -\cos(\omega_0 t)\\ 0 & 0 & -\sin(\omega_0 t)\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}x(t)\\ y(t)\\ \theta(t)\end{bmatrix} + \begin{bmatrix}-\sin(\omega_0 t) & 0\\ \cos(\omega_0 t) & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}v_C\\ \omega\end{bmatrix},\ \boldsymbol{x}(0) = \begin{bmatrix}x_0\\ y_0\\ \theta_0\end{bmatrix} \quad (31)$$

Writing it out in algebraic form, this matrix equation is equivalent to the system of three equations:

$$\Sigma = \begin{cases}\dot{x}(t) = -v_C \sin(\omega_0 t) - \theta(t)\cos(\omega_0 t),\ \ x(t_0) = x_0\\ \dot{y}(t) = v_C \cos(\omega_0 t) - \theta(t)\sin(\omega_0 t),\ \ y(t_0) = y_0\\ \dot{\theta}(t) = \omega,\ \ \theta(t_0) = \theta_0\end{cases} \quad (32)$$

Note that the output equation $\boldsymbol{y}(t) = C(t)\boldsymbol{x}(t) + D(t)\boldsymbol{u}(t)$, typically included in addition to the state equation $\dot{\boldsymbol{x}}(t)$ to fully describe the system Σ, is omitted as it is not relevant for the development of this controller.

CONTROLLABILITY ANALYSIS

Before proceeding with the development of the LQR, it is necessary to verify that the system is controllable. Controllability is a property that guarantees the existence of a control function that transfers the system from any initial state x_0 at time t_0 to any final state x_f at an arbitrary and finite time t_f. Verifying the controllability of an LTI systems is fairly short and straightforward, as it only involves checking whether or not the controllability matrix is full rank: if it is, the system is controllable [41]. For LTV systems, such as the QBot 2, the procedure is substantially more involved and is covered in the next sections.

State Transition Matrix

In general, there is no closed-form solution to the system described with the state-space representation. However, if a number of properties regarding the so-called "state transition matrix" hold, a general solution can indeed be determined. More specifically, for any control input $\boldsymbol{u}(t)$ and initial conditions $\boldsymbol{x}(t_0)$, the solution of a linear system can be determined with the use of the state transition matrix $\Phi(t, t_0)$ [42]. First, we must define this matrix as

$$\Phi(t, t_0) \coloneqq X(t)X^{-1}(t_0) \quad (33)$$

where $X(t)$ is any fundamental matrix of $\dot{x} = A(t)x$.

The state transition matrix, which will later be used to construct the Controllability Gramian, must satisfy the following two conditions [43]:

$$\begin{cases} \frac{\partial}{\partial t}\Phi(t,t_0) = A(t)\Phi(\mathrm{t},\mathrm{t}_0) = \begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix}\Phi(\mathrm{t},\mathrm{t}_0) \\ \Phi(\mathrm{t}_0,\mathrm{t}_0) = \mathrm{I}_{3\mathrm{x}3} \end{cases} \quad (34)$$

For LTV systems such as the QBot 2, if the property

$$A(t) * \int_{t_0}^{t} A(\tau)\, d\tau = \int_{t_0}^{t} A(\tau)\, d\tau * A(t) \quad (35)$$

is satisfied, then $\Phi(t,t_0)$ can be computed as

$$\Phi(\mathrm{t},\mathrm{t}_0) = \exp\left[\int_{t_0}^{t} A(\tau)\, d\tau\right] \quad (36)$$

where τ is the current time variable.

Thus, we first compute the integral of the state matrix, which we denote as A_{int}, using the $A(t)$ matrix computed earlier for the QBot.

$$A_{int} = \int_{t_0}^{t} A(\tau)\, d\tau = \int_{t_0}^{t} \begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix} d\tau = \frac{1}{\omega_0}\begin{bmatrix} 0 & 0 & \sin(\omega_0 t_0) - \sin(\omega_0 t) \\ 0 & 0 & \cos(\omega_0 t) - \cos(\omega_0 t_0) \\ 0 & 0 & 0 \end{bmatrix} \quad (37)$$

Note how the property described in equation (35) holds, as the multiplications yield a zero matrix both ways.

$$A(t) * A_{int}(t) = A_{int}(t) * A(t) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (38)$$

Equation (36) is now applied to obtain the state transition matrix:

$$\Phi(\mathrm{t},\mathrm{t}_0) = \exp[A_{int}] = \exp\left[\frac{1}{\omega_0}\begin{bmatrix} 0 & 0 & \sin(\omega_0 t_0) - \sin(\omega_0 t) \\ 0 & 0 & \cos(\omega_0 t) - \cos(\omega_0 t_0) \\ 0 & 0 & 0 \end{bmatrix}\right] \quad (39)$$

By noting that any triangular matrix with zeros along the main diagonal is nilpotent, the matrix A_{int} can be said to be nilpotent with degree 2 (i.e. elevating it to any power A_{int}^{k} for some positive $k \geq 2$ always returns a zero matrix). This property allows to greatly simplify the matrix exponential operation, which is computed as:

$$\exp(A_{int}) = I_{3x3} + A_{int} + \frac{A_{int}^2}{2!} + \frac{A_{int}^3}{3!} + \cdots + \frac{A^n}{n!} \tag{40}$$

Because all terms higher than $n \geq 2$ are equal to zero, this operation is reduced to just:

$$\exp(A_{int}) = I_{3x3} + A_{int} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \frac{1}{\omega_0}\begin{bmatrix} 0 & 0 & \sin(\omega_0 t_0) - \sin(\omega_0 t) \\ 0 & 0 & \cos(\omega_0 t) - \cos(\omega_0 t_0) \\ 0 & 0 & 0 \end{bmatrix} \tag{41}$$

Thus, the state transition matrix is found to be:

$$\Phi(\mathrm{t}, \mathrm{t}_0) = \begin{bmatrix} 1 & 0 & -\frac{1}{\omega_0}[\sin(\omega_0 t_0) - \sin(\omega_0 t)] \\ 0 & 1 & \frac{1}{\omega_0}[\cos(\omega_0 t) - \cos(\omega_0 t_0)] \\ 0 & 0 & 1 \end{bmatrix} \tag{42}$$

Its partial derivative with respect to time is equal to the $A(t)$ matrix and results in:

$$\frac{\partial}{\partial t}\Phi(\mathrm{t}, \mathrm{t}_0) = \begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix} \tag{43}$$

The conditions that the state transition matrix must satisfy, as stated earlier in equation (34), can now be verified, yielding:

$$\begin{cases} \frac{\partial}{\partial t}\Phi(\mathrm{t}, \mathrm{t}_0) = \begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 & -\frac{1}{\omega_0}[\sin(\omega_0 t_0) - \sin(\omega_0 t)] \\ 0 & 1 & \frac{1}{\omega_0}[\cos(\omega_0 t) - \cos(\omega_0 t_0)] \\ 0 & 0 & 1 \end{bmatrix} = \\ \qquad = \begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix} \\ \Phi(\mathrm{t}_0, \mathrm{t}_0) = \begin{bmatrix} 1 & 0 & -\frac{1}{\omega_0}[\sin(\omega_0 t_0) - \sin(\omega_0 t_0)] \\ 0 & 1 & \frac{1}{\omega_0}[\cos(\omega_0 t_0) - \cos(\omega_0 t_0)] \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathrm{I}_{3\mathrm{x}3} \end{cases} \tag{44}$$

Note that these calculations are further simplified when $t_0 = 0$. This will in fact be done in this thesis during the simulation and experimental phases.

Controllability Conclusion

Now that the state transition matrix has been defined and its conditions satisfied, the Controllability Gramian can be determined. It is denoted by W_C and is defined as [44]:

$$W_C(t_0, t_f) \coloneqq \int_{t_0}^{t_f} \Phi(\mathrm{t}, \mathrm{t}_0)\, B(t)\, B^T(t)\, \Phi^{\mathrm{T}}(\mathrm{t}, \mathrm{t}_0)\, dt \tag{45}$$

The state transition matrix $\Phi(\mathrm{t}, \mathrm{t}_0)$ found in the previous section and the input matrix $B(t)$ are plugged into this equation. The integrand is first computed separately for clarity.

$$\Phi(\mathrm{t}, \mathrm{t}_0)\, B(t)\, B^T(t)\, \Phi^{\mathrm{T}}(\mathrm{t}, \mathrm{t}_0) =$$

$$= \begin{bmatrix} 1 & 0 & -\frac{1}{\omega_0}[\sin(\omega_0 t_0) - \sin(\omega_0 t)] \\ 0 & 1 & \frac{1}{\omega_0}[\cos(\omega_0 t) - \cos(\omega_0 t_0)] \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -\sin(\omega_0 t) & 0 \\ \cos(\omega_0 t) & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} -\sin(\omega_0 t) & \cos(\omega_0 t) \\ 0 & 0 \\ 0 & 1 \end{bmatrix} *$$

$$* \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{\omega_0}[\sin(\omega_0 t_0) - \sin(\omega_0 t)] & \frac{1}{\omega_0}[\cos(\omega_0 t) - \cos(\omega_0 t_0)] & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} W'_{C,11} & W'_{C,12} & W'_{C,13} \\ \star & W'_{C,22} & W'_{C,23} \\ \star & \star & W'_{C,33} \end{bmatrix}$$

$$W'_{C,11} = \sin^2(\omega_0 t) + \frac{1}{\omega_0^2}[\sin(\omega_0 t) - \sin(\omega_0 t_0)]^2$$

$$W'_{C,12} = -\cos(\omega_0 t)\sin(\omega_0 t) - \frac{1}{\omega_0^2}[\cos(\omega_0 t) - \cos(\omega_0 t_0)][\sin(\omega_0 t) - \sin(\omega_0 t_0)]$$

$$W'_{C,13} = -\frac{1}{\omega_0}[\sin(\omega_0 t) - \sin(\omega_0 t_0)]$$

$$W'_{C,22} = \cos^2(\omega_0 t) + \frac{1}{\omega_0^2}[\cos(\omega_0 t) - \cos(\omega_0 t_0)]^2$$

$$W'_{C,23} = \frac{1}{\omega_0}[\cos(\omega_0 t) - \cos(\omega_0 t_0)]$$

$$W'_{C,33} = 1 \tag{46}$$

Because the resulting matrix is symmetric, the elements (2,1), (3,1) and (3,2) have been omitted. Computing the integral of this matrix yields the large and, again, symmetric matrix:

$$W_C(t_0, t_f) = \int_{t_0}^{t_f} \Phi(\mathrm{t}, \mathrm{t}_0)\, B(t)\, B^T(t)\, \Phi^{\mathrm{T}}(\mathrm{t}, \mathrm{t}_0)\, dt = \begin{bmatrix} W_{C,11} & W_{C,12} & W_{C,13} \\ \star & W_{C,22} & W_{C,23} \\ \star & \star & W_{C,33} \end{bmatrix}$$

$$W_{C,11} = \frac{1}{4\omega_0^3}\big[3\sin(2\omega_0 t_0) + \sin(2\omega_0 t_f) + 6\omega_0 t_0 - 6\omega_0 t_f + 2\omega_0^3 t_0 - 2\omega_0^3 t_f$$
$$- 8\cos(\omega_0 t_f)\sin(\omega_0 t_0) - \omega_0^2 \sin(2\omega_0 * t_0) + \omega_0^2 \sin(2\omega_0 t_f)$$
$$- 4\omega_0 t_0 \cos^2(\omega_0 t_0) + 4\omega_0 t_f \cos^2(\omega_0 t_0)\big]$$

$$W_{C,12} = \frac{1}{4\omega_0^3}\big[3\cos(2\omega_0 t_0) - 4\cos\big(\omega_0(t_0 + t_f)\big) + \cos(2\omega_0 t_f) - \omega_0^2 \cos(2\omega_0 t_0)$$
$$+ \omega_0^2 \cos(2\omega_0 t_f) + 2\omega_0 t_0 \sin(2\omega_0 t_0) - 2\omega_0 t_f \sin(2\omega_0 t_0)\big]$$

$$W_{C,13} = -\frac{1}{\omega_0^2}\big[\cos(\omega_0 t_0) - \cos(\omega_0 t_f)\big] - \frac{1}{\omega_0}(t_0 - t_f)\sin(\omega_0 t_0)$$

$$W_{C,22} = \frac{1}{4\omega_0^3}\big[3\sin(2\omega_0 t_0) + \sin(2\omega_0 t_f) - 2\omega_0 t_0 + 2\omega_0 t_f - 2\omega_0^3 t_0 + 2\omega_0^3 t_f$$
$$- 8\cos(\omega_0 t_0)\sin(\omega_0 t_f) - \omega_0^2 \sin(2\omega_0 t_0) + \omega_0^2 \sin(2\omega_0 t_f)$$
$$- 4\omega_0 t_0 \cos^2(\omega_0 t_0) + 4\omega_0 t_f \cos^2(\omega_0 t_0)\big]$$

$$W_{C,23} = \frac{1}{\omega_0}\big[\cos(\omega_0 t_0)(t_0 - t_f)\big] - \frac{1}{\omega_0^2}\big[\sin(\omega_0 t_0) - \sin(\omega_0 t_f)\big]$$

$$W_{C,33} = t_f - t_0 \tag{47}$$

Note again that the calculations become greatly simplified when setting $t_0 = 0$. The result of the Controllability Gramian is not actually relevant on its own, but its rank is. In fact, for the system to be controllable, W_C must be full rank (i.e. its rank must equal the number of state variables, in this case three). The following conclusion, as well as all other calculations in this section, was proven using the MATLAB code shown in Appendix A.

$$rank(W_C) = 3 \tag{48}$$

Thus, W_C is shown to be full rank for $t_0 < t_f$, proving that the LTV model of the QBot 2 around the circular path is indeed controllable.

LOCALIZATION TECHNIQUES

For determining the current state of the robot, two techniques exist. The first one is based on odometry (also known as dead reckoning), which computes the current location and orientation from the wheel velocities over time. By knowing the initial state of the system, the robot can continuously update its state by estimating how far it has travelled and how much it has rotated. This approach is vulnerable to errors, as noise in the signals or slippage of the wheels can greatly affect the accuracy of the calculations. After some simulations with MATLAB and V-REP, this option was promptly discarded.

The alternative is to use some external positioning system that allows the robot to know its exact position and orientation in space. In virtual simulations, this is easily accomplished as the software knows exactly the state of the robot as it runs the simulation. In reality, having a precise positioning measurement is much harder. Using satellite-based global positioning system (GPS) is by far the most commonly used method. However, this method is only accurate to about 3 meters and, more importantly, does not work indoors.

Luckily, the Robotics and Controls Laboratory at San Diego State University (located in room E-300A) is equipped with OptiTrack, a motion capture and 3D tracking system. This is a type of local positioning system (LPS), as it only works within a finite rectangular area of approximately 12 by 11 feet in the lab, shown in Fig. 5.

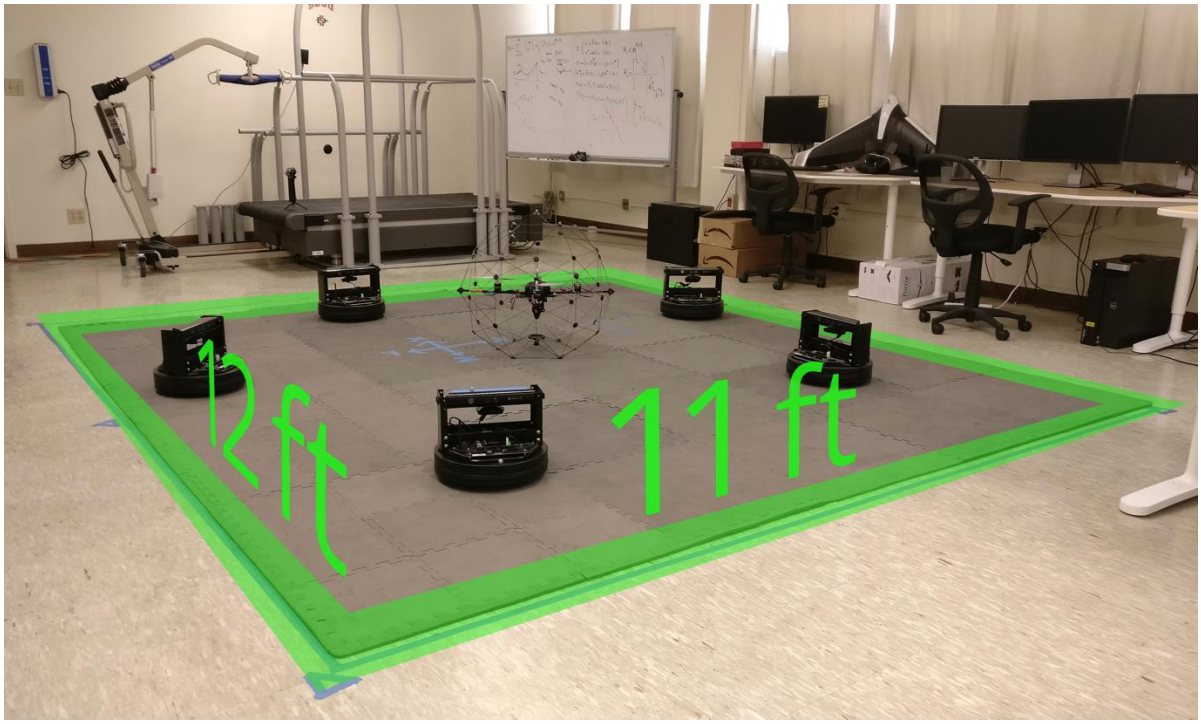

Figure 5. Working area for the OptiTrack camera system in the lab.

The six ceiling-mounted Flex 3 high-speed infra-red (IR) cameras, shown in Fig. 6, together with the Motive software, can track objects with reflective markers to a high degree of accuracy (less than 1 mm). The approximate disposition of the cameras is displayed in Fig. 6, an image taken directly from Motive.

Figure 6. Close-up of one of the six OptiTrack Flex 3 ceiling-mounted IR camera.

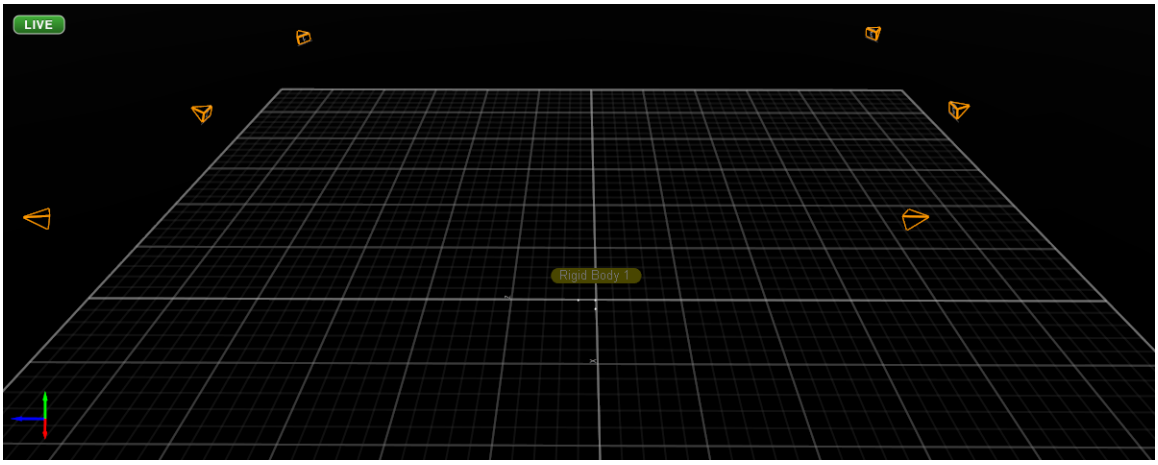

Figure 7. Camera layout as shown in Motive.

Each camera has a ring of 26 LEDs emitting 850 nm wavelength light around the lens and can capture up to 100 FPS [45]. The software seamlessly communicates with Simulink, allowing a continuous feed of object coordinates to be used in simulations. For the QBot 2 robot to be tracked by the OptiTrack system, three small IR-reflective balls are mounted asymmetrically on top of it, as shown in Fig. 8. The exact positioning of the markers is not

relevant, as the Motive software automatically measures and records their positions relative to each other.

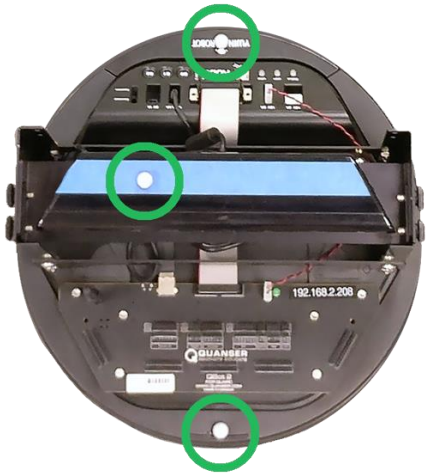

Figure 8. Reflective marker locations, highlighted in green, mounted on the QBot 2.

CHAPTER 3

CONTROLLER DESIGN

At this stage, the system is fully modeled using the state-space representation and has been proven to be controllable. The next objective is to formulate the Linear Quadratic Regulator problem for a linear time varying system such as the QBot 2. This type of controller guides the system state or, in this case, its deviation from a nominal trajectory to zero while minimizing the control effort. Three different parameters can be adjusted to obtain the desired response. After solving for the optimal control law, the optimal feedback system is constructed.

LQR PROBLEM

The LQR provides a solution for the optimal control problem concerned with the minimization of a cost function (or performance index). This technique makes use of the state space representation and three tweakable design parameters to compute the optimal control input $\boldsymbol{u}(t)$ [46]. The latter is then used to form an optimal closed-loop feedback system. To begin, consider the again the QBot 2 LTV system Σ described by the state equations:

$$\dot{\boldsymbol{x}} = A(t)\boldsymbol{x} + B(t)\boldsymbol{u} \tag{49}$$

$$\dot{\boldsymbol{x}} \in \mathbb{R}^3, \boldsymbol{x} \in \mathbb{R}^3, \boldsymbol{u} \in \mathbb{R}^2, A \in \mathbb{R}^{3\times3}, B \in \mathbb{R}^{3\times2}$$

The specified initial conditions are, in general,

$$\boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{50}$$

and the final time is

$$t_f = T_f \geq t_0 \quad with \;\; T_f < \infty \tag{51}$$

Note how this LQR problem belongs to the Finite-Horizon category, as the final time T_f is specified and finite. This also implies that there is no control constraint or terminal constraint on the system.

The cost function (or performance index) J is defined as

$$J = \phi\big(\boldsymbol{x}(T_f), T_f\big) + \int_{t_0}^{T_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\, dt \tag{52}$$

The objective is to minimize this function, in order for the control input to be optimal.

The first scalar function $\phi\big(\boldsymbol{x}(T_f), T_f\big)$ is concerned only with the final state of the system, at time T_f, and thus penalizes the cost function based on the error between the final state and zero (i.e., the origin). The function is defined as

$$\phi = \tfrac{1}{2}\boldsymbol{x}^T(T_f) F \boldsymbol{x}(T_f) \tag{53}$$

where F is the first design parameter, a matrix with the following characteristics:

$$F \in \mathbb{R}^{3\times 3}, \qquad F \geq 0\ (positive\ semi-definite)$$

The second scalar function $L(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ is integrated from initial time t_0 to final time T_f, and thus takes the entire process into account, including both the system's state and the applied control input. The function is defined as

$$L = \tfrac{1}{2}\boldsymbol{x}^T(t) Q(t) \boldsymbol{x}(t) + \tfrac{1}{2}\boldsymbol{u}^T(t) R(t) \boldsymbol{u}(t) \tag{54}$$

where Q and R are the remaining two design parameters, matrices with the following characteristics:

$$Q(t) \in \mathbb{R}^{3\times 3},\ Q \geq 0\ (positive\ semi-definite)$$

$$R(t) \in \mathbb{R}^{2\times 2},\ R > 0\ (positive\ definite)$$

for every $t \geq 0$. The cost function can then be expressed explicitly as

$$J = \tfrac{1}{2}\boldsymbol{x}^T(T_f) F \boldsymbol{x}(T_f) + \tfrac{1}{2}\int_{t_0}^{T_f} \boldsymbol{x}^T(t) Q(t) \boldsymbol{x}(t) + \boldsymbol{u}^T(t) R(t) \boldsymbol{u}(t) dt \tag{55}$$

Note that because all three design parameters F, Q and R, often referred to as penalty weightings, are at least positive semi-definite matrices (except R), this cost function tries to guide $\boldsymbol{x}(T_f) \to 0$ while maintaining $\boldsymbol{x}(t)^2$ and $\boldsymbol{u}(t)^2$ as small as possible throughout the

trajectory. While the Q and R matrices can be chosen to be time-varying, they are set to constant values for this application (as is, in general, commonly done).

The Hamiltonian matrix for LQR problem is then constructed using the integrand from the cost function, as well as the state equations and the costates of the system.

$$H(\boldsymbol{x},\boldsymbol{u},\boldsymbol{p},t) \coloneqq L(\boldsymbol{x},\boldsymbol{u},\boldsymbol{p},t) + \boldsymbol{p}^T\dot{\boldsymbol{x}} = \frac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + \frac{1}{2}\boldsymbol{u}^T R\boldsymbol{u} + \boldsymbol{p}^T(A(t)\boldsymbol{x} + B(t)\boldsymbol{u}) \quad (56)$$

where $\boldsymbol{p}$ is the vector of costates. By applying the optimality condition

$$\frac{\partial H}{\partial u} = R(t)\boldsymbol{u} + B^T(t)\boldsymbol{p}(t) = 0 \quad (57)$$

the control $\boldsymbol{u}_0$ that minimizes the performance index can be solved for as

$$\boldsymbol{u}_0(\boldsymbol{x},\boldsymbol{p},t) = -R^{-1}(t)B^T(t)\boldsymbol{p} \quad (58)$$

Note that since

$$\frac{\partial^2 H}{\partial u^2} = R(t) > 0 \quad (59)$$

the control $\boldsymbol{u}_0$ is proven to be the control that minimizes H. Now, the Hamiltonian from equation (56), the optimality condition from equation (57), the state equation from equation (49) and the vector of costates $\boldsymbol{p}$ can be used in the maximum principle to obtain the optimal solution. The following set of equations are used to apply this principle.

$$H(\boldsymbol{x},\boldsymbol{u},\boldsymbol{p},t) = \frac{1}{2}\boldsymbol{x}^T Q\boldsymbol{x} + \frac{1}{2}\boldsymbol{u}^T R\boldsymbol{u} + \boldsymbol{p}^T(A(t)\boldsymbol{x} + B(t)\boldsymbol{u}) \quad (60)$$

$$\dot{\boldsymbol{x}} = \left(\frac{\partial H}{\partial \boldsymbol{p}}\right)^T = A(t)\boldsymbol{x} + B(t)\boldsymbol{u} \qquad \boldsymbol{x}(0) = \boldsymbol{x_0} \quad (61)$$

$$-\dot{\boldsymbol{p}} = \left(\frac{\partial H}{\partial \boldsymbol{x}}\right)^T = Q(t)\boldsymbol{x} + A(t)^T\boldsymbol{p} \qquad \boldsymbol{p}(T_f) = F\boldsymbol{x}(T_f) \quad (62)$$

$$\frac{\partial H}{\partial u} = R(t)\boldsymbol{u} + B^T(t)\boldsymbol{p}(t) = 0 \qquad u = -R^{-1}(t)B^T(t)\boldsymbol{p} \quad (63)$$

The optimal solution can be found either by solving a two-point boundary value problem or, as done here, by guessing the form of the solution. In this case, the guessed form of the solution is $\boldsymbol{p}(t) = K(t)\boldsymbol{x}(t)$ [47]. Taking the derivative by applying the product rule and expanding the result with equations (49) and (63), we obtain

$$\dot{\boldsymbol{p}} = \dot{K}\boldsymbol{x} + K\dot{\boldsymbol{x}} = \dot{K}\boldsymbol{x} + K(A\boldsymbol{x} + B\boldsymbol{u}) = \dot{K}\boldsymbol{x} + K(A\boldsymbol{x} - BR^{-1}B^T\boldsymbol{p}) =$$

$$= \dot{K}\boldsymbol{x} + K(A\boldsymbol{x} - BR^{-1}B^TK\boldsymbol{x}) = \dot{K}\boldsymbol{x} + K(A - BR^{-1}B^TK)\boldsymbol{x} \tag{64}$$

Next, equation (62) is used to replace the left-hand-side of this equation before rearranging and simplifying some terms.

$$-Q\boldsymbol{x} - A^T\boldsymbol{p} = \dot{K}\boldsymbol{x} + K(A - BR^{-1}B^TK)\boldsymbol{x}$$

$$-Q\boldsymbol{x} - A^TK\boldsymbol{x} = \dot{K}\boldsymbol{x} + K(A - BR^{-1}B^TK)\boldsymbol{x}$$

$$-Q - A^TK = \dot{K} + KA - KBR^{-1}B^TK \tag{65}$$

Equation (65) is satisfied if a $K(t)$ can be found such that

$$\dot{K} = KBR^{-1}B^TK - KA - A^TK - Q \tag{66}$$

with $K(T_f) = F$. This ODE is known as the matrix Riccati equation and, together with its final condition, is expressed as

$$\begin{cases} \dot{K} = KBR^{-1}B^TK - KA - A^TK - Q \\ K(T) = F \end{cases} \tag{67}$$

Note that the solution of the Riccati equation, which can be solved by backwards numerical integration, is guaranteed and independent of initial conditions $\boldsymbol{x}_0$ (meaning it can be used no matter what the initial state of the system is). Also, the resulting $K(t)$ is inevitably a positive semi-definite matrix.

The solution to the Riccati equation can then be used to form the optimal control law, which becomes

$$\boldsymbol{u}(t) = -R^{-1}B(t)^TK(t)\boldsymbol{x}(t) \tag{68}$$

By letting the LQR gain $K_{lqr}(t) = R^{-1}B(t)^TK(t)$, we obtain the more compact form

$$\boldsymbol{u}(t) = -K_{lqr}(t)\,\boldsymbol{x}(t) \tag{69}$$

This information can be used to construct an optimal feedback system (closed-loop) in general. Because the goal is to steer the QBot around a nominal trajectory rather than simply towards the origin, the term $\boldsymbol{x}(t)$ in the above equation is replaced by $[\boldsymbol{x}(t) - \boldsymbol{x}^*(t)]$. This

way, rather than having the system state itself approach zero, the error between the system state and the desired state, called tracking error, is guided towards zero. This idea is visualized in Fig. 9.

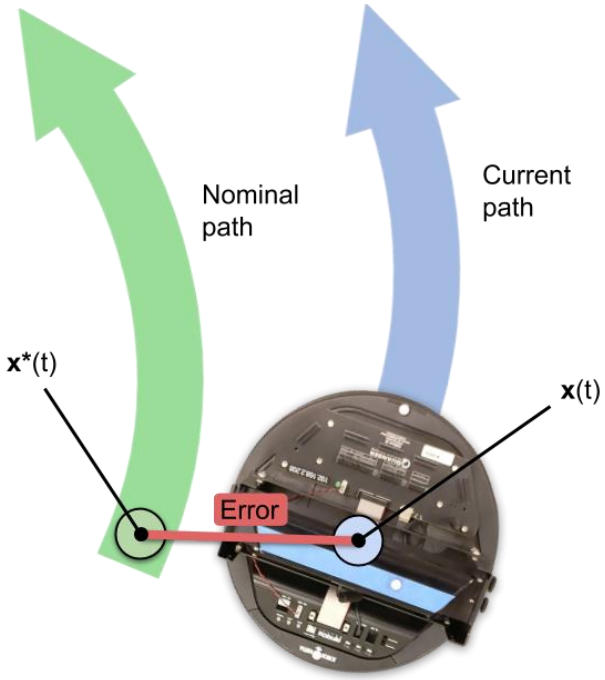

Figure 9. Representation of the error between nominal and current state (tracking error).

As described in the previous chapters, the trajectory of the QBot is described by constant forward and angular velocities, contained in the vector $\boldsymbol{u}^*$. This term must be added to the right-hand side of the $\boldsymbol{u}(t)$ equation. This way, the corrections made by the LQR to the control inputs are around the nominal inputs rather than around zero. In conclusion, the optimal feedback control law for the QBot robot becomes

$$\boldsymbol{u}(t) = -K_{lqr}(t)\,[\boldsymbol{x}(t) - \boldsymbol{x}^*(t)] + \boldsymbol{u}^* \tag{70}$$

This control law can now be used to construct an optimal feedback system that tracks a nominal state (a stationary location or a moving path). The diagram of this controller is shown in Fig. 10. In the next chapters, it will be implemented in MATLAB and Simulink to simulate the system's response and, later, to test the controller on the actual system.

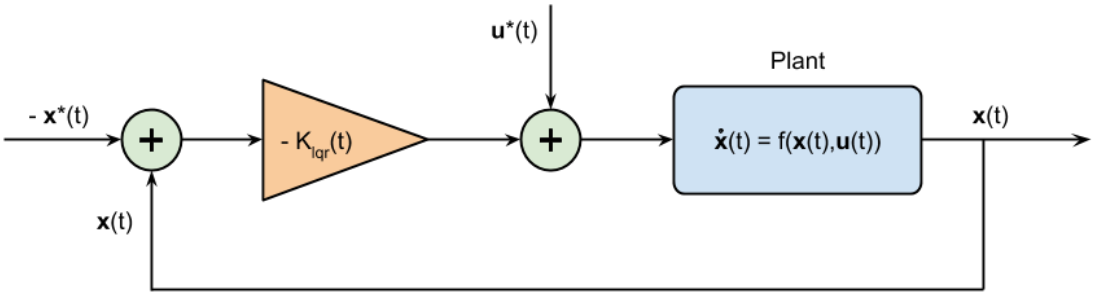

Figure 10. Diagram of the optimal feedback system using the LQR.

LQR DESIGN

The next step is to apply the theory covered in the previous section to the QBot 2 robot specifically. The statement for QBot finite-horizon LQR problem is first laid out. The LTV system is described by the matrix state equation

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} \cos\left(\frac{\pi}{2} + \omega_0 t\right) & 0 \\ \sin\left(\frac{\pi}{2} + \omega_0 t\right) & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{u}(t), \qquad \boldsymbol{x}(0) = \begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \end{bmatrix} \quad (71)$$

The A(t) and B(t) matrices, shown here explicitly, are continuous time-dependent functions. Final time is specified as $t_f = T_f \geq t_0$ and $T_f < \infty$. The three penalty weightings are set equal to the following diagonal (identity) matrices:

$$F = \begin{bmatrix} f_{11} & 0 & 0 \\ 0 & f_{22} & 0 \\ 0 & 0 & f_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (72)$$

$$Q = \begin{bmatrix} q_{11} & 0 & 0 \\ 0 & q_{22} & 0 \\ 0 & 0 & q_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (73)$$

$$R = \begin{bmatrix} r_{11} & 0 \\ 0 & r_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (74)$$

where $f_{ii} \geq 0, q_{ii} \geq 0$, and $r_{ii} > 0$. These values are just a standard initial setting and can later on be tuned to obtain the more desired response form the system. Using the general form of these parameters, we obtain the following performance index

$$J = \frac{1}{2}\boldsymbol{x}^T(T_f)F\boldsymbol{x}(T_f) + \frac{1}{2}\int_{t_0}^{T_f} \boldsymbol{x}^T(t)Q\boldsymbol{x}(t) + \boldsymbol{u}^T(t)R\boldsymbol{u}(t)dt =$$

$$= \frac{1}{2}\begin{bmatrix} x(T_f) & y(T_f) & \theta(T_f) \end{bmatrix}\begin{bmatrix} f_{11} & 0 & 0 \\ 0 & f_{22} & 0 \\ 0 & 0 & f_{33} \end{bmatrix}\begin{bmatrix} x(T_f) \\ y(T_f) \\ \theta(T_f) \end{bmatrix}$$

$$+\frac{1}{2}\int_{t_0}^{T_f}\left(\begin{bmatrix} x(t) & y(t) & \theta(t) \end{bmatrix}\begin{bmatrix} q_{11} & 0 & 0 \\ 0 & q_{22} & 0 \\ 0 & 0 & q_{33} \end{bmatrix}\begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} + \begin{bmatrix} v_C(t) & \omega(t) \end{bmatrix}\begin{bmatrix} r_{11} & 0 \\ 0 & r_{22} \end{bmatrix}\begin{bmatrix} v_C(t) \\ \omega(t) \end{bmatrix}\right)dt =$$

$$= \frac{1}{2}\left(f_{11}x(T_f)^2 + f_{22}y(T_f)^2 + f_{33}\theta(T_f)^2\right)$$

$$+\frac{1}{2}\int_{t_0}^{T_f}(q_{11}x(t)^2 + q_{22}y(t)^2 + q_{33}\theta(t)^2) + (r_{11}v_C(t)^2 + r_{22}\omega(t)^2)\,dt \qquad (75)$$

The Riccati equation for this system is then

$$\dot{K} = K\begin{bmatrix} \cos\left(\frac{\pi}{2} + \omega_0 t\right) & 0 \\ \sin\left(\frac{\pi}{2} + \omega_0 t\right) & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} \frac{1}{r_{11}} & 0 \\ 0 & \frac{1}{r_{22}} \end{bmatrix}\begin{bmatrix} \cos\left(\frac{\pi}{2} + \omega_0 t\right) & \sin\left(\frac{\pi}{2} + \omega_0 t\right) & 0 \\ 0 & 0 & 1 \end{bmatrix}K$$

$$-K\begin{bmatrix} 0 & 0 & -\cos(\omega_0 t) \\ 0 & 0 & -\sin(\omega_0 t) \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\cos(\omega_0 t) & -\sin(\omega_0 t) & 0 \end{bmatrix}K$$

$$-\begin{bmatrix} q_{11} & 0 & 0 \\ 0 & q_{22} & 0 \\ 0 & 0 & q_{33} \end{bmatrix}$$

$$K(T) = \begin{bmatrix} r_{11} & 0 \\ 0 & r_{22} \end{bmatrix} \qquad (76)$$

The solution to this system, $K(t)$, is a (symmetric) positive definite 3x3 matrix. As done in the previous section, this solution is converted to the actual gain $K_{lqr}(t)$ using the relation

$$K_{lqr}(t) = R^{-1}B(t)^T K(t) = \begin{bmatrix} \frac{1}{r_{11}} & 0 \\ 0 & \frac{1}{r_{22}} \end{bmatrix} \begin{bmatrix} \cos\left(\frac{\pi}{2} + \omega_0 t\right) & \sin\left(\frac{\pi}{2} + \omega_0 t\right) & 0 \\ 0 & 0 & 1 \end{bmatrix} K(t) \quad (77)$$

This way, the LQR gain $K_{lqr}(t)$ becomes a 2x3 matrix that will be, throughout the rest of thesis and in the MATLAB program, referred to as

$$K_{lqr}(t) = \begin{bmatrix} k_{lqr,11} & k_{lqr,12} & k_{lqr,13} \\ k_{lqr,21} & k_{lqr,22} & k_{lqr,23} \end{bmatrix} \quad (78)$$

The final optimal feedback control law for the QBot 2 is now

$$\boldsymbol{u}(t) = -K_{lqr}(t)\,[\boldsymbol{x}(t) - \boldsymbol{x}^*(t)] + \boldsymbol{u}^*$$

$$= -\begin{bmatrix} k_{lqr,11} & k_{lqr,12} & k_{lqr,13} \\ k_{lqr,21} & k_{lqr,22} & k_{lqr,23} \end{bmatrix} * \left(\begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} - \begin{bmatrix} x^*(t) \\ y(t) \\ \theta(t) \end{bmatrix} \right) + \begin{bmatrix} v_{C,0} \\ \omega_0 \end{bmatrix} \quad (79)$$

The equations developed in this section are now ready to be implemented in MATLAB and used to simulate the system's response. Before doing so, a few words need to be spend on the penalty weightings.

LQR Parameters Setting

The three penalty weighting matrices have so far been left untouched and simply set to identity matrices. This is a good starting point for the simulations, as no single system parameter is given priority over the others. However, tweaking the values of F, Q and R can have a great impact on the system's response. While tweaking of the weightings would be an obvious procedure for implementing this controller for an actual application, the simulations and experiments presented in this thesis will only be examined with the simple case where F, Q and R are set to identity matrices. The reason for this choice is that the focus of the thesis is to show how the theory of the system model and LQR is applied to the QBot robot and to verify that its response follows the desired trajectory, rather than studying the behavior of the weighting matrices to identify the best combinations of parameters.

NEURAL NETWORK PROBLEM

Artificial neural networks (ANN) are computer reasoning models based on the functioning of biological brains. They represent one of the many approaches to machine learning. As such, their main purpose is to predict the outcome of new situations based on patterns and information previously learned from a known set of training data. ANNs achieve this by uncovering the hidden relation between input and output data, and by abstracting the knowledge contained in the training data in order to be effective on never-seen-before inputs [48].

An ANN consists of a number of neurons, which are the fundamental interconnected computational building blocks. The choice of number and arrangement of neurons depends on the complexity of the tackled problem and is part of the design challenge. The structure of a single neuron is depicted in Fig. 11.

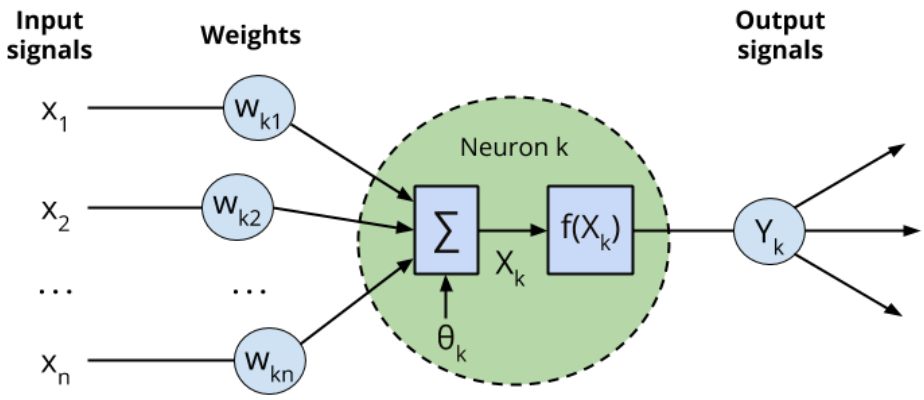

Figure 11. Single neuron structure.

As shown, the neuron has several inputs x_i, with corresponding weights w_i, and only a single output Y. First, the weighted sum of the inputs X is calculated for each neuron, according to the equation

$$X = \sum_{i=1}^{n} x_i w_i \tag{80}$$

The weighted sum is then compared to the threshold value θ and passed through an activation function. Some examples of activation functions are the step, sign, and linear functions, but the most common one is the sigmoid function [49], which is defined as

$$y_{sigmoid}(x) \coloneqq \frac{1}{1+e^{-x}} \tag{81}$$

The sigmoid function, characterized by its "S" shape as shown in Fig. 12, behaves similarly to the step functions for relatively large positive or negative input values, but for input values in the vicinity of zero it produces nonlinearly-calculated outputs between 0 and 1.

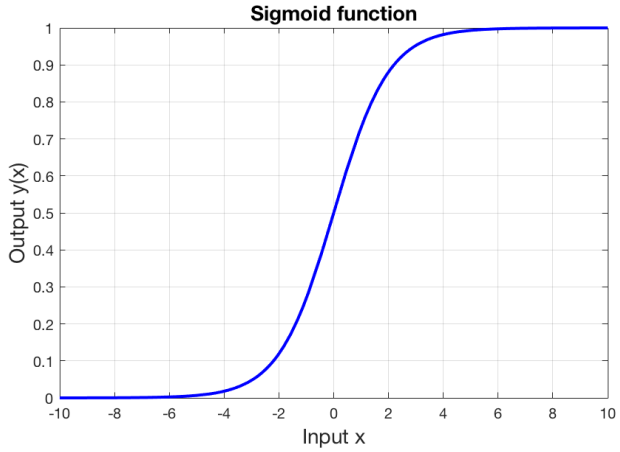

Figure 12. Sigmoid activation function plotted in MATLAB.

Using the sigmoid as the activation function, the output Y of the neuron is then computed as

$$Y = sigmoid\left[\sum_{i=1}^{n} x_i w_i - \theta\right] = \frac{1}{1+e^{-\left(\sum_{i=1}^{n} x_i w_i - \theta\right)}} \tag{82}$$

Neurons pass signals to each other through the weighted links and they are organized in layers, as shown in Fig. 13. The input and an output layer are always present, while the number of hidden layers can vary based on the problem's needs. Note that the neurons in the input layer do not actually perform any operations.

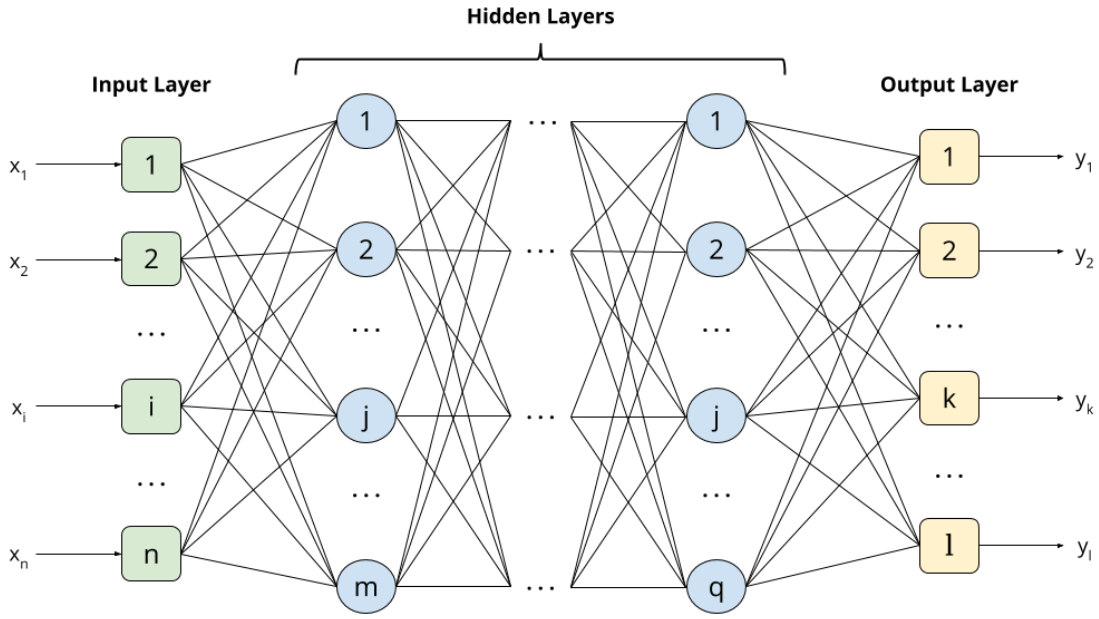

Figure 13. General neural network structure.

There are several techniques for training neural networks. One of the most common is the back-propagation training algorithm, which this thesis will mainly focus on. The name of this technique derives from its characteristic of passing the inputs signals forward through the network, and subsequently propagating the error signals backwards to adjust the weights and the thresholds. The goal of the training phase is to find the weights that minimize the error between the training data and the network's output. The backpropagation algorithm can be summarized in the following four steps [48]:

1. For each neuron, the weight and threshold values are initialized to random values chosen from within the range

$$\left(-\frac{2.4}{N_i}, +\frac{2.4}{N_i}\right) \tag{83}$$

   where N_i is the number of input signals going to the neuron i.

2. The network is activated by applying the input data from the training set and calculating the outputs of each neuron, using the sigmoid activation function. The outputs of the neurons in the first hidden layers are computed as

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) w_{ij}(p) - \theta_j(p)\right] \tag{84}$$

where p is the iteration counter, known as "epoch", and n is the number of input signals to neuron j in the hidden layer. The same operation is repeated through the other hidden layers and through the output layer. The signals leaving the neurons in the output layer represent the actual outputs of the neural network.

3. The error e_k of each neuron in the output layer is calculated as

$$e_k(p) = y_{d,k}(p) - y_k(p) \tag{85}$$

where $y_{d,k}$ is the desired output of neuron k and its value is provided by the training dataset. The error is then used to compute the error gradient δ_k

$$\delta_k(p) = y_k(p) * [1 - y_k(p)] * e_k(p) \tag{86}$$

The error gradient allows to generate the weight and threshold corrections, calculated respectively as

$$\Delta w_{jk}(p) = \alpha * y_j(p) * \delta_k(p) \tag{87}$$

$$\Delta \theta_k(p) = \alpha * (-1) * \delta_k(p) \tag{88}$$

where α is the learning rate, a design parameter that determines how large the adjustments should be at each iteration.

The new weights and thresholds, which will be used for the next iteration $p + 1$, for the output neurons can now be obtained as

$$w_{jk}(p + 1) = w_{jk}(p) + \Delta w_{jk}(p) \tag{89}$$

$$\theta_k(p + 1) = \theta_k(p) + \Delta \theta_k(p) \tag{90}$$

This operation is repeated backwards in the network through all hidden layers.

4. Repeat steps 2 and 3 by increasing the iteration counter p, until some error criterion is met. The most common error criterion is reducing the sum of squared errors, computed with the following equation, below a certain threshold.

$$e_{SSE}(p) = \sum_{k=1}^{l} [e_k(p)]^2 \tag{91}$$

where l is the number of neurons in the output layer.

Once the error criterion is met, the network is considered trained and can now be used with inputs that are not part of the training dataset. If the training data was accurate and the training was successful, the neural network should output correct values for the new inputs.

Neural Network Design

For this thesis, a number of neural networks are used for storing the LQR gain values. As a reminder, the LQR gains are found by solving the Riccati equation and they are contained in the 2x3 time-dependent matrix $K_{lqr}(t)$. When using MATLAB's built-in ode45 solver, discrete time values are used. For this reason, the stored $K_{lqr}(t)$ is not defined for any arbitrary t, but only at specific time intervals. If the stored $K_{lqr}(t)$ values were to be used for real-time applications, the sample rate of the experiment would have to match the time interval used by the ode45 perfectly in order to always have values for the LQR gains. This may not always be possible, as the real-time sample rate is usually higher.

The problem's solution chosen for this thesis is to train artificial neural networks to learn the behavior of the stored LQR gains over time. By using the back-propagation training technique, implemented through the optimized Neural Network Toolbox algorithms in MATLAB, the networks can learn the pattern of the functions that generate each LQR gain. Since the $K_{lqr}(t)$ matrix contains six elements, the choice of using a total of six neural networks is evident. Equally trivial is the selection, for each network, of one input neuron, reading in the current time, and one output neuron, producing the LQR gain associated with the input time.

The challenging part of the neural networks design then consists of choosing how many layers the network should have and how many neurons should be in each layer. For simplicity, all six neural networks will have the same structure, even though some of them would be able to perform well with a simpler structure due to the easier behavior of its assigned LQR gain over time. The solution to this challenge was obtained via trial and error, as is the norm for this type of problem given that there are no specific rules on neurons and layers selection. The experimentations involved networks with structures ranging from just 1 hidden layer with 3 neurons to 4 hidden layers with 10 neurons each.

After extensive testing, it was found that the best combination of parameters with the smallest number of layers and neurons that still guaranteed consistently accurate results was using two hidden layers with 7 neurons in each. Including the input and output neurons, the network consists of a total of 16 neurons and is shown in Fig. 14.

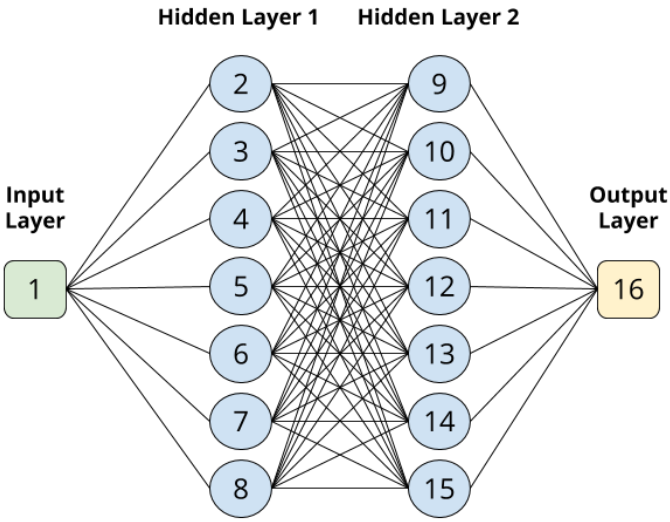

Figure 14. Representation of the neural network structure used to learn the LQR gain values.

CHAPTER 4

NUMERICAL SIMULATIONS

In this section, the results of the numerical simulations are presented and discussed. The structure and highlights of the MATLAB program developed to perform the simulations are also explained to better understand how the theory described in the previous sections can be translated into code. The numerous options toggled by the available parameters are illustrated to understand the full potential of the program, even though some settings may not be required for basic simulations.

SIMULATIONS SETUP

The entire program was developed using MATLAB R2017a with Simulink, Symbolic Toolbox, and Neural Network Toolbox used as well. The program is centered around the main script, called "QBot_2_4.m", shown in Appendix B. All dependencies of the scripts and functions are shown in Fig. 15.

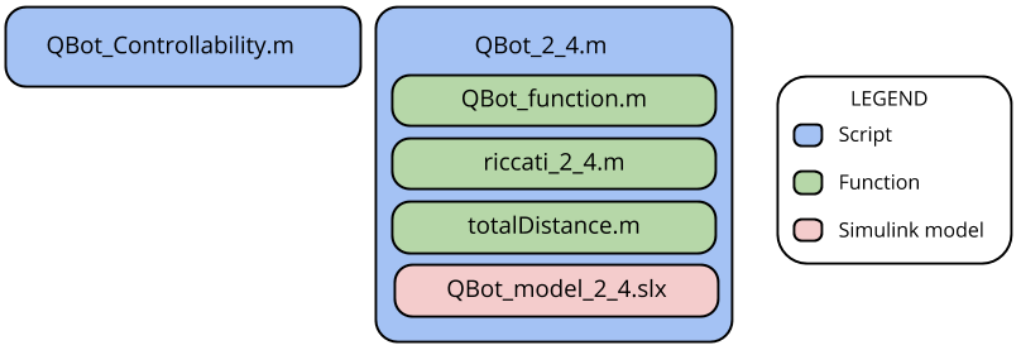

Figure 15. Dependencies of MATLAB scripts, functions and Simulink models for simulations.

Before using this script, it is necessary to run the independent script called "QBot_Controllability.m" to verify that the system is controllable. This code contains the

mathematical representation of the system, using the unicycle model, from which the $A(t)$ and $B(t)$ matrices are derived. The State Transition Matrix is obtained and used to compute the Controllability Gramian, as discussed in the previous chapter. The script then prints out whether the system is controllable or not to the command window.

For the simulations, this is the only script that needs to be run independently. All other features of the program can be called from within the main script "QBot_2_4.m". Throughout this script's execution, messages are printed to the command window giving information on the section that is currently being run.

At the very beginning of the script is section 0, called "Constant Parameters (User-Defined)". This is where the tweakable variables can be specified:

- Nominal Trajectory:
    - RADIUS: radius (in meters) of the circular trajectory center around origin.
    - PERIOD: desired time (in seconds) to complete a full rotation around the nominal path.
- Time:
    - FINAL_TIME: final simulation time in seconds (note that initial time is always zero).
    - TIME_INCREMENT: sample time in seconds.
- Initial Conditions:
    - INITIAL_X, INITIAL_Y, INITIAL_THETA: starting coordinates of the robot.
- LQR parameters:
    - F, Q, R: specifies the 3x3 matrices F and Q, and the 2x2 matrix R.
- Switches (logical):
    - useSimulink: if true, the numerical simulation will be performed using Simulink (continuous time); if false, the simulation will be done in a for loop with discrete time approximation.
        - loopAnimation: if useSimulink is false, a plot animating the robot's motion on the x-y plane will be displayed while the for loop is running.
    - useNN: if true, the six neural networks are trained to learn the LQR gains and are later used during simulation to get the real-time gains; if false, the LQR gains will be called from the matrices stored in the workspace.
    - exportToExcel: if true, at the end of the simulation, relevant data is exported to an Excel sheet.

- plotFigures: if true, at the end of the simulation, relevant figures are plotted to visually show the results.
- exportFigures: if true, along with plotFigures, the figures are saved as PNG images to the current folder.

After setting the above specified parameters, the script generates all necessary symbolic variables in section 1. It is important to note the convention used to distinguish between lowercase x and y, which represent coordinates on the x-y plane, and capitalized X and Y, which represent the vectors containing state variables and output variables respectively. For consistency, all variables of the state space representation are capitalized.

Section 2 defines the system's dynamics, again in the state space form. This includes the state equations, the inputs equations, and the output equations. Next, in section 3, the nominal trajectories are defined and used to form the $A(t)$ and $B(t)$ matrices. Section 4 is short and used to simply convert some variable names and to create the time array containing all time instants, from time 0 to FINAL_TIME with intervals TIME_INCREMENT.

Section 5 contains the core of the LQR computation. After defining some useful variables, the solution $\boldsymbol{K}(t)$ to the Riccati equation is computed. This is achieved with backwards integration of the Riccati equation using MATLAB's built-in ode45 function. This function makes use of the external function called "riccati_2_4.m", shown in Appendix C. This function takes in the A and B matrices, as well as the R and Q weightings, as parameters. It also takes the current time and the unknown X variable (symbolized by $\boldsymbol{K}(t)$ in the Riccati equation) for the ode45 evaluation. The function then returns the left-hand side of the equation, the differential term $\dot{\boldsymbol{K}}(t)$. An array containing the time instants from final time to initial time is provided to the ode45, as well as the initial condition specified in the matrix F. Because for each time interval the ode45 returns a 1x9 vector, the solution is then converted to 2x3 matrices. Next, a for loop iterating through time computes the actual gain $K_{lqr}(t)$ and the control input $\boldsymbol{u}(t)$.

In section 6, the results from the LQR computation and other variables used so far are converted to Simulink-compatible data structures. This mainly consists of transforming cells into simple matrices or vectors.

Section 7 is only active if the useNN variable is set to true. If it is, six neural networks (nn11, nn12, nn13, nn21, nn22 and nn23), one for each value of the LQR gain, are initialized

using the "fitnet" command from MATLAB's Neural Network toolbox. Each network is set up as described in the previous chapter, with one input neuron, two hidden layers with 7 neurons each, and one output neuron (indicated as [1, 7, 7, 1]), for a total of 16 neurons, shown in Fig. 4.2. The networks are then trained using the default Levenberg-Marquardt algorithm through the "train" command. After the training phase, the program tests the performance of the networks and plots the results.

Next, in section 8, the core of the simulation phase begins. If the variable useSimulink is set to true, the Simulink model called "QBot_model_2_4.slx", shown in Fig. 16, is used to simulate the system's response through the given simulation time.

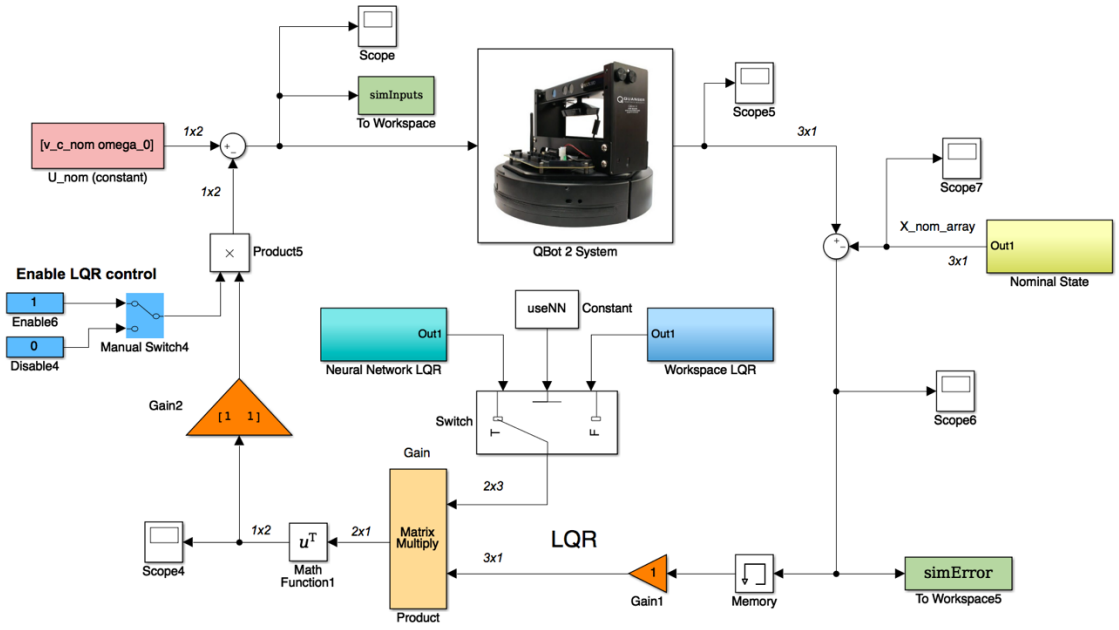

Figure 16. "QBot_model_2_4" Simulink closed-loop model.

The model, implemented as a closed-loop system, is centered around the "QBot 2 System" plant block, shown in Fig. 17. Inside this block, the inputs to the system are passed through the QBot 2 system model block to obtain the outputs. The QBot 2 model block, shown in Fig. 18, contains the function describing the QBot's motion. This function, named "QBot_function.m" and shown in Appendix D, is invoked using the Interpreted MATLAB function block. It takes in the two control inputs v_c and ω, and the angular position θ as parameters in a single vector. It then computes and returns the three velocity components $\dot{x}$,

$\dot{y}$, and $\dot{\theta}$. These three outputs signals are then integrated to calculate the updated position of the robot (x, y, and θ).

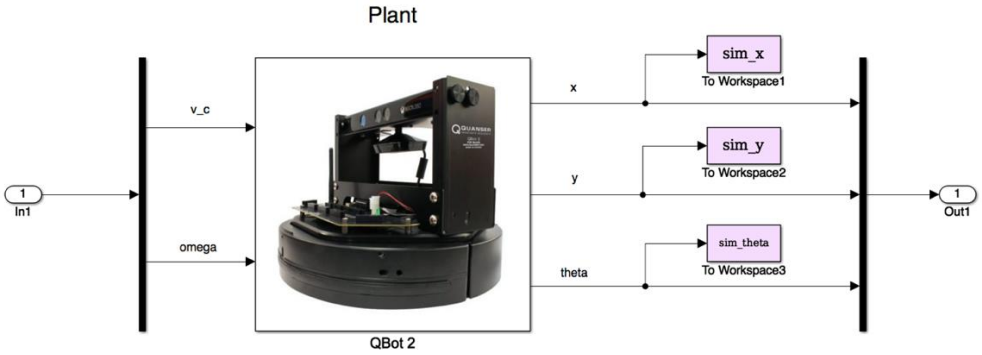

Figure 17. "QBot 2 System" block content (inside the QBot_model_2_4).

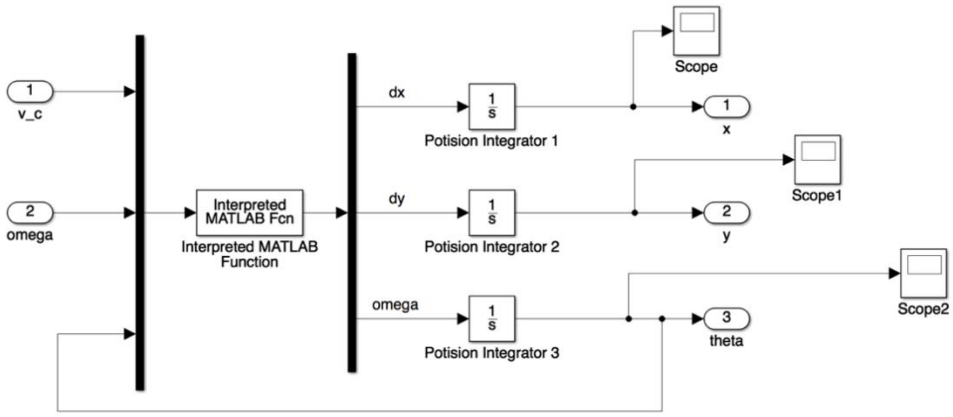

Figure 18. "QBot 2" block content (inside the QBot 2 System block).

Next, the nominal position, which is read from the workspace, is subtracted from the current position. Simultaneously, the LQR gains are read either directly from the workspace or, if the useNN variable is set to true, computed from the six neural networks. For the first case, the block called "Workspace LQR" is used and its content is shown in Fig. 19. For the second case, the block called "Neural Network LQR" is used instead and its content is shown in Fig. 20. The Interpreted MATLAB function blocks within it make use of the "sim" command to evaluate the gains through the networks.

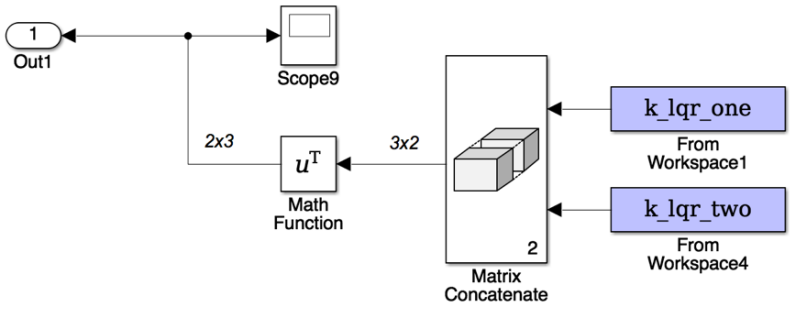

Figure 19. "Workspace LQR" block content inside the QBot 2 System block.

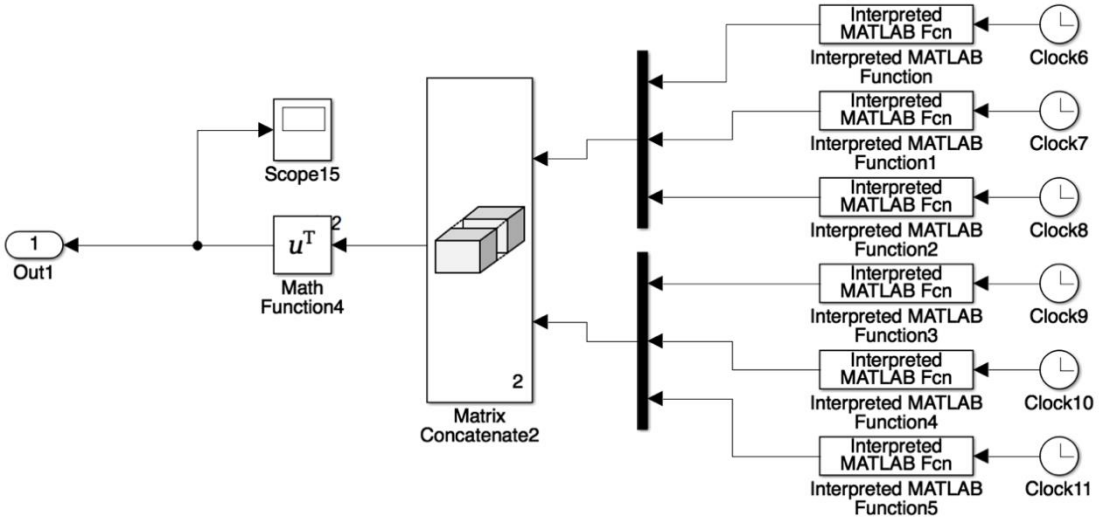

Figure 20. "Neural Network LQR" block content inside the QBot 2 System block.

The gains are then multiplied by the difference between the nominal and actual position. The result is subtracted from the constant nominal inputs $v_{c,nom}$ and ω_0. This becomes the new input to the plant, thus completing the closed-loop cycle. Throughout the Simulink model, other function blocks are placed to perform matrix operations, such as transposing and concatenating, and other blocks are used to collect data and store it to the workspace once the simulation is over.

If the variable useSimulink is set to false, the Simulink model is not run and an in-code simulation using a for loop is performed instead. Note that while Simulink is a

continuous solver, using the for-loop alternative implies switching to a discrete solving method. Its accuracy is therefor highly dependent on the chosen time step (the TIME_INCREMENT variable): the smaller the time step, the more realistic the simulation will be. Other than that, the for loop is structured the same way as the Simulink model. The only difference is that the new position is computed not by integration, but instead by multiplying the velocity times the time step (in effect, computing the differential distance travelled during the time step) and adding the result to the previous position. Also, an if-statement, triggered by the loopAnimation variable being true, can be activated to plot the new position throughout the loop as a sort of animation. At the end of the simulation, whether performed using Simulink or in-code, the total distance (in meters) travelled by the robot is computed using the function called "totalDistance", shown in Appendix H.

The last three sections do not require an in-depth explanation. Section 9 simply exports the simulation data to an appropriately-named Excel sheet, if the exportToExcel variable is set to true. Section 10 generates all the plots that present the results of the simulation. These figures include the robot's motion on the xy-plane, the LQR gains, the tracking errors and other useful information.

SIMULATION RESULTS

Running the MATLAB code and Simulink model described in the previous sections with a specified set of parameters constitutes one simulation. In this section, several simulations are presented to demonstrate that the numerically-computed response of the system is as expected and in agreement with the theoretical background. The main two varying parameters are the radius of the circular nominal trajectory r_{nom} and the tangential velocity v_C. For each chosen radius, the effect of choosing different velocities is explored. For a given nominal radius and tangential velocity, the equivalent period T_{eq}, i.e. the time it takes for the robot to complete one full circular lap, is calculated, according to the relation

$$T_{eq} = \frac{2\pi}{\omega_o} = 2\pi \frac{r_{nom}}{v_C} \tag{92}$$

For each set of chosen radius and velocity, two different initial conditions are then tested out to observe the simulated response. For this chapter, the initial conditions chosen are limited to small variations in the x-coordinate around the nominal initial conditions. Additional

simulations with different initial conditions will be presented in the Experimental Results section later on in order to be directly compared to the actual QBot's motion in the lab. A summary of all simulations presented in the next sub-sections is shown in Table 2.

Table 2. Summary of Simulations

Nominal Radius r_{nom} [m]	**Tangential Velocity v_C [m/s]**	**Equivalent Period T_{eq} [s]**	**Section**
0.5 (medium)	0.35 0.175	9 18	4.2.1 4.2.1.1 4.2.1.2
0.2 (small)	0.35	3.6	4.2.2
1.0 (large)	0.35	18	4.2.3
1.25 (extra-large)	0.35	22	4.2.4

As deducible from the table, the tangential velocity of 0.35 m/s can be regarded as the standard testing velocity. Overall, the velocity values are:

- Slow at 0.175 m/s, or 25% of the maximum linear velocity of the robot
- Fast at 0.35 m/s, or 50% of the maximum linear velocity of the robot

The selection of tangential velocities is obviously restricted by the actual robot's performance limitations. The relatively low percentages of the maximum linear velocity are due to the fact that, when the robot is following a curved path, the wheel on the outer side of the curve has to spin faster than the other wheel. Thus, the maximum linear velocity of 0.7 m/s is only achievable when travelling in a straight line, which is never the case for this thesis. Experiments with tangential velocities faster than 0.35 m/s have been shown to be inconsistent because of several reasons discussed later on. For this reason, 0.35 m/s is chosen as the upper limit for velocity during testing. For some simulations, the number of laps (and, consequently, the total run time) are reduced, solely because of error convergence being reached long before the end of the simulation.

Nominal Path Radius: 0.5 m

The analysis begins with what for this thesis is considered the standard radius for the nominal path. This is a radius of 0.5 m, representing a medium size trajectory around the origin.

TANGENTIAL VELOCITY: 0.35 M/S

The first simulations are run with the standard tangential velocity of 0.35 m/s, which is 50% of the maximum linear velocity of the QBot 2. Travelling at this velocity through the circular path of radius 0.5 m, the robot should take 9 s to complete one full lap. The simulations consist of 3 laps around the path, corresponding to a total run time of 27 s. The LQR gains and the solutions to the Riccati equations are shown in Fig. 21. These values will be the same for all subsequent runs with this radius and tangential velocity, since they are independent of initial conditions.

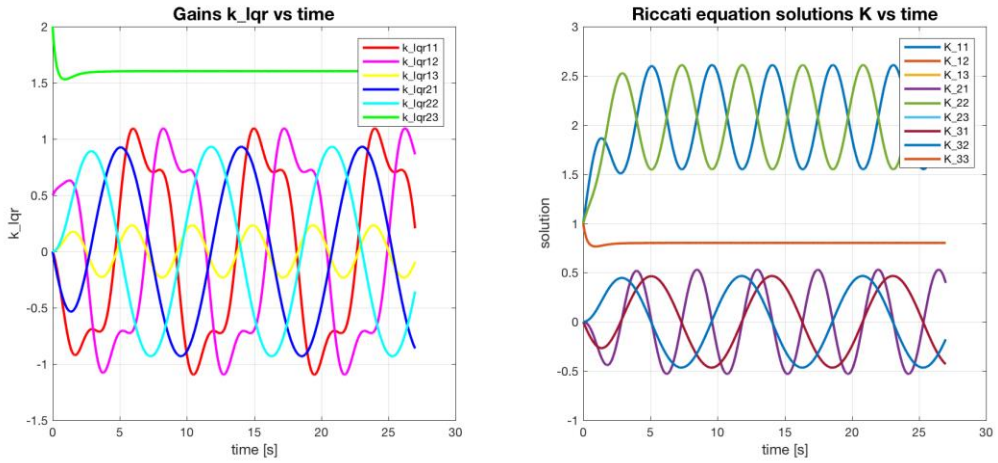

Figure 21. LQR gains and solutions to the Riccati equations for radius 0.5 m and velocity 0.35 m/s.

The LQR gains are used to train the six neural networks. Convergence is quickly reached and training is on average stopped at around the 300^{th} epoch, with final errors of less than 10^{-7}. The error during the learning process is plotted for the first neural network only, as it is similar to the other networks. It is shown below in Fig. 22. After the training phase, the networks are tested to visually verify whether the learning was successful or not. As

shown in Fig. 23, the training indeed succeeded as there is virtually no error between the values predicted by the networks and the actual LQR gains.

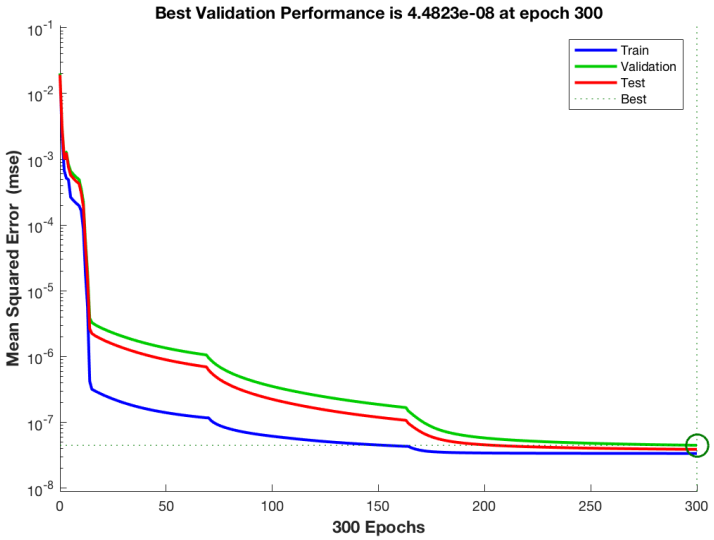

Figure 22. Error during training of neural network 1 for radius 0.5 m and velocity 0.35 m/s.

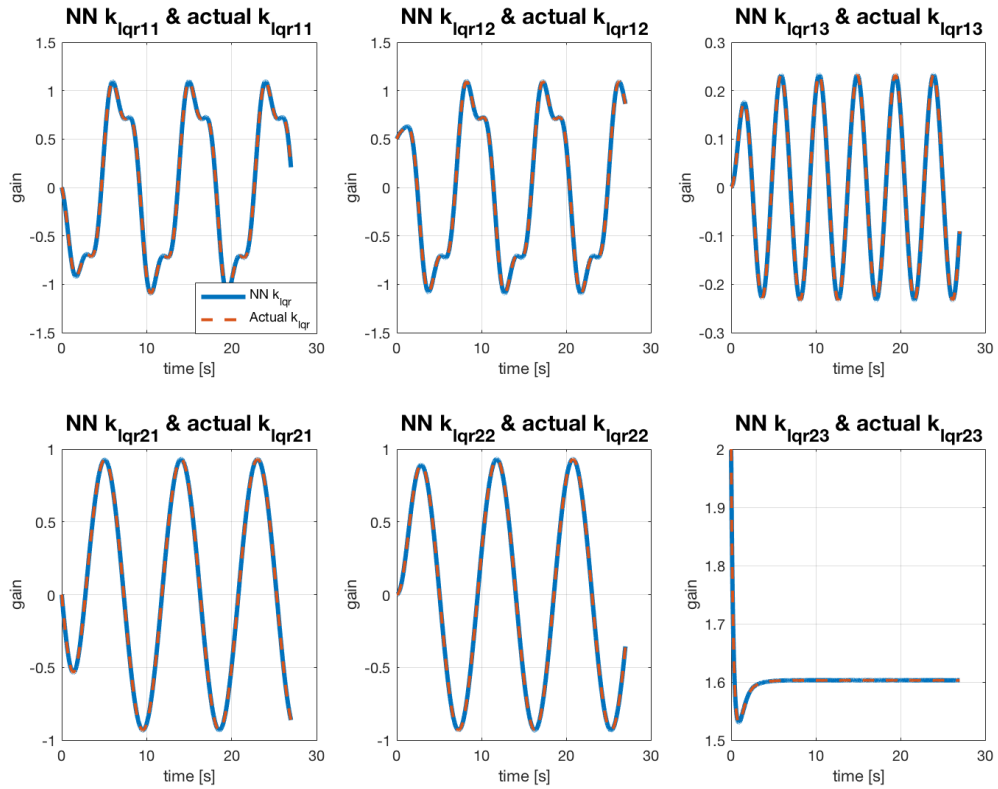

Figure 23. Neural network generated LQR gains compared to the actual LQR gains for radius 0.5 m and velocity 0.35 m/s.

With a sample time of 0.01 s, a total of 2,700 data points for each variable are calculated and recorded during the simulation. The two runs start from small deviations in the x-coordinate around the nominal initial condition $(0.5, 0, \pi/2)$: the first starts at $(0.4, 0, \pi/2)$, the second at $(0.6, 0, \pi/2)$. The simulated path of the robot and the tracking errors over time are shown in Figs. 24 and 25.

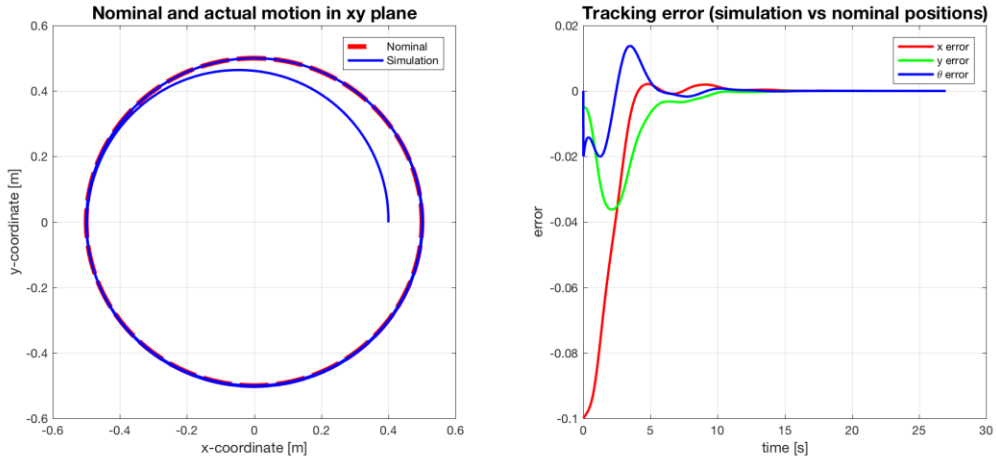

Figure 24. Simulated path and tracking error for run 1 (0.4, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

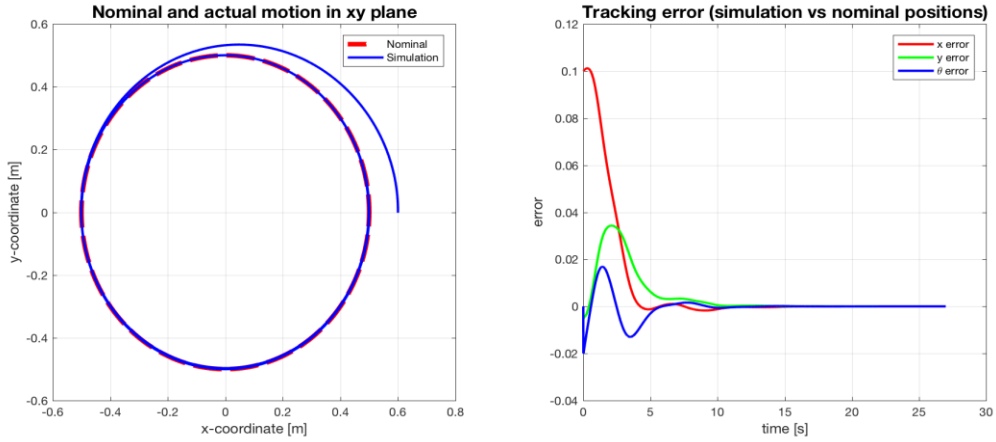

Figure 25. Simulated path and tracking error for run 2 (0.6, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

In both cases, the simulated responses represent quick and smooth adjustments of the robot's position and direction to converge into the nominal path. The tracking errors rapidly

change, with some initial overshooting, before settling at zero. In both runs, it took the robot around 10 s to remove all errors and merge into the correct path.

It is interesting to analyze the plots for the two inputs to the system, v_C and ω, shown in Figs. 26 and 27.

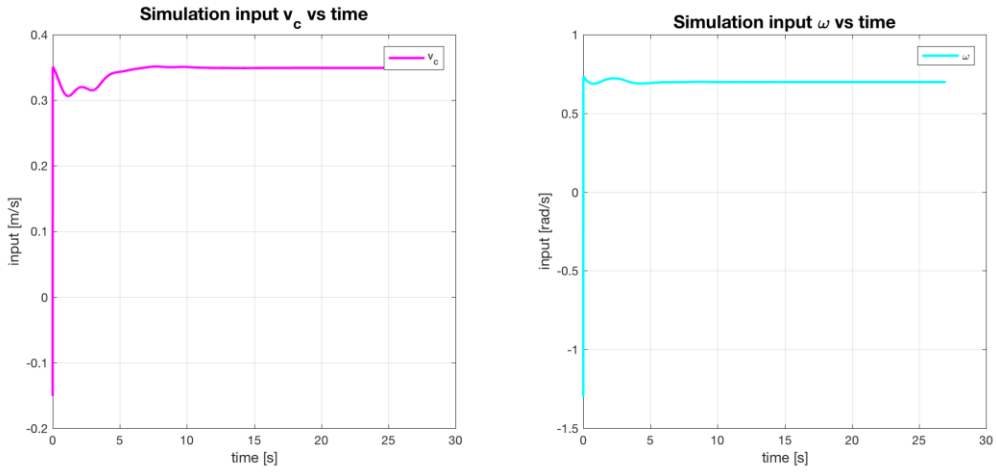

Figure 26. Simulated system inputs for run 1 (0.4, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

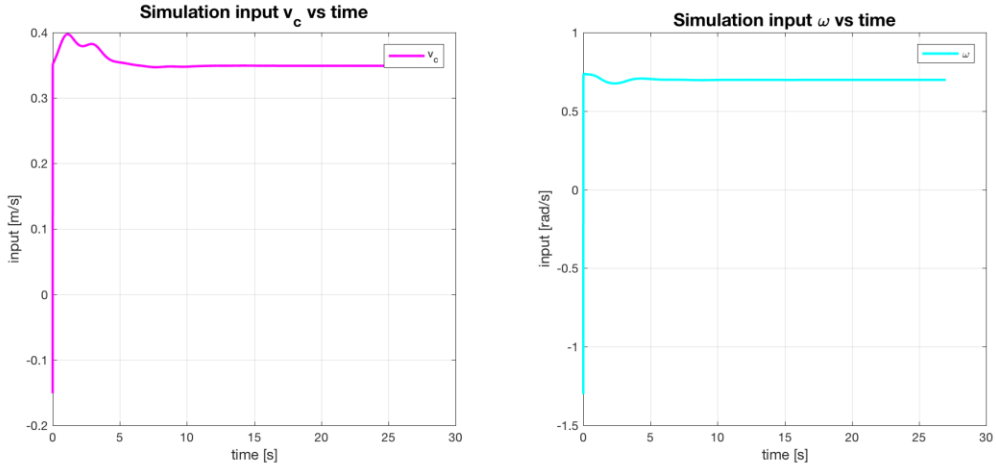

Figure 27. Simulated system inputs for run 2 (0.6, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

The two plots for the forward velocities v_C appear to be flipped vertically. This is attributed to error in the initial x-coordinate. Because the robot in run 1 is initially inside the nominal path, less distance needs to be travelled, i.e. less input effort needs to be produced, to reach the nominal path. On the other hand, in run 2, the robot begins from outside the

nominal path, thus requiring more effort in forward velocity to catch up with the nominal trajectory. Also note how the angular velocity input ω in both runs appear very similar and relatively flat. This can be explained by the fact that the robot already starts facing the correct direction and thus little adjustment is needed.

Because of the close-to-perfect replication of LQR values by the neural networks, the simulated responses using the gain values directly from the MATLAB workspace are identical to the ones presented here and thus are omitted.

TANGENTIAL VELOCITY: 0.175 M/S

Next, the tangential velocity is reduced to 0.175 m/s, which is just 25% of the maximum linear velocity of the robot. This velocity corresponds to a period of 18 s and, by still using 3 laps, the total run time is now 54 s. With the same sample time of 0.01 s, the total number of data points for each variable are now increased to 5,400.

Since the simulations were already successful with a higher velocity, it would be safe to assume that a lower velocity should work correctly as well. The reason for this analysis is in fact more focused on just comparing the results with the 0.35 m/s velocity simulations. As it turns out, the robot's trajectories, as well as the LQR gains and Riccati equation solutions, shown in Fig. 28, are not the same.

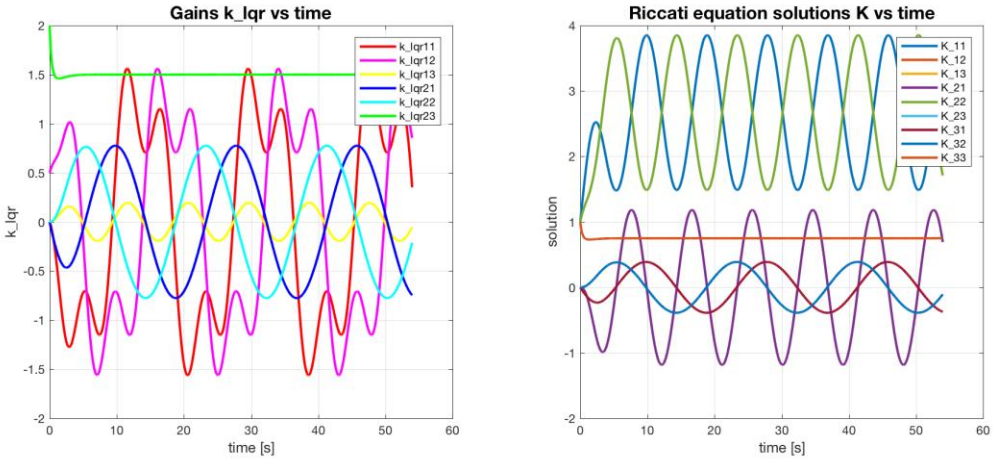

Figure 28. LQR gains and solutions to the Riccati equations for radius 0.5 m and velocity 0.175 m/s.

It is interesting to see how some of the LQR gains, in particular $k_{lqr,21}$, $k_{lqr,22}$ and $k_{lqr,23}$ remain relatively unchanged compared to the higher velocity case. Instead, the other values, $k_{lqr,11}$, $k_{lqr,12}$ and $k_{lqr,13}$, appear to have higher peaks. Since the first row of K_{lqr} affects the forward velocity, we can conclude that this parameter needs to be corrected more than in the case with faster velocity.

The neural networks again successfully learn the LQR values and their plots are omitted, as they are virtually identical to what is shown in Fig. 28. It is however interesting to analyze the simulated response of the system, which now differs from the case with a faster velocity. The simulated paths produced for both initial conditions of $(0.4, 0, \pi/2)$ and $(0.6, 0, \pi/2)$ are shown in Figs. 29 and 30.

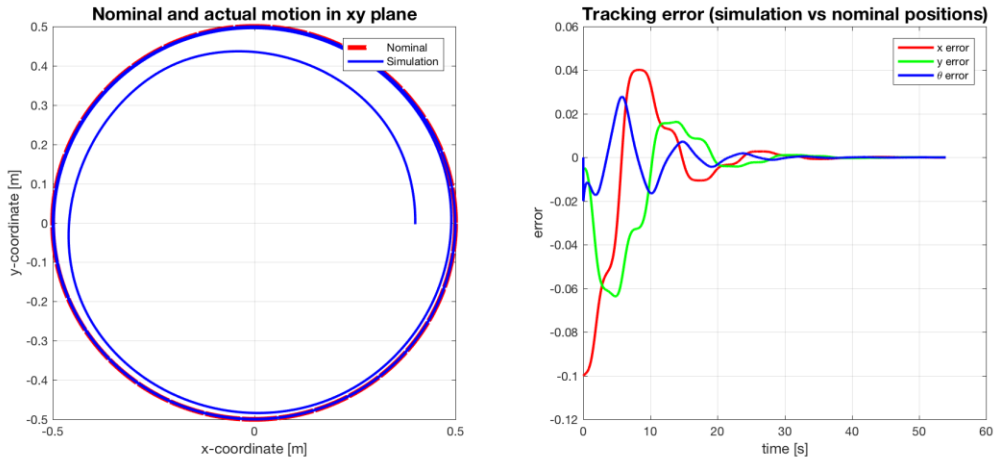

Figure 29. Simulated path and tracking error for run 1 (0.4, 0, $\pi/2$), radius 0.5 m and velocity 0.175 m/s.

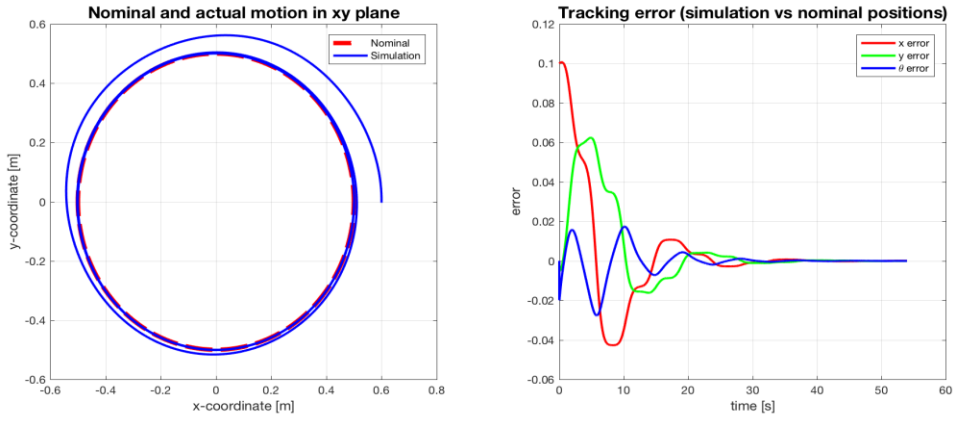

Figure 30. Simulated path and tracking error for run 2 (0.6, 0, $\pi/2$), radius 0.5 m and velocity 0.175 m/s.

While the controller again successfully and smoothly fixes the robot's trajectory to match the nominal path, it is clear that it does so slower than the previous case, taking approximately 30 seconds to eliminate all errors (compared to the previous 10 seconds). This response could of course be accelerated by varying the LQR parameters, but it is interesting to note just how much slower it is compared to when the forward velocity is 0.35 m/s.

Also consider the inputs to the system over time for both initial conditions, plotted in Figs. 31 and 32.

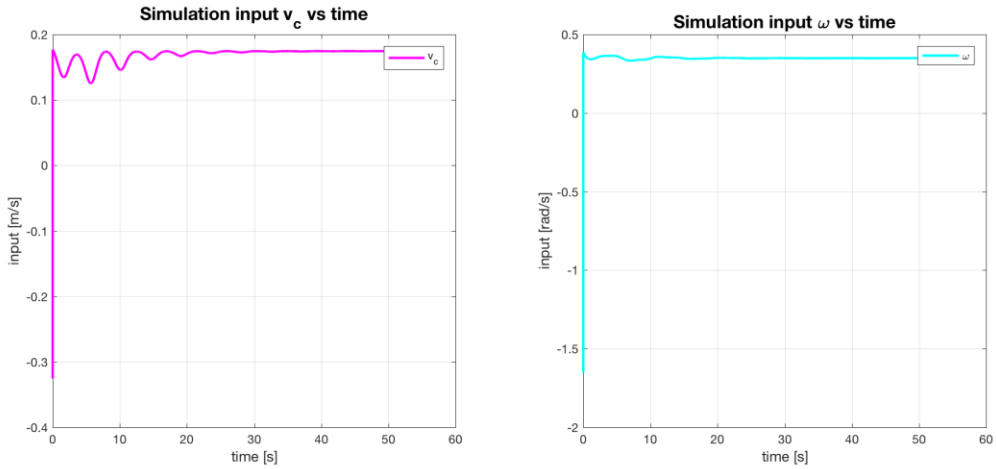

Figure 31. Simulated system inputs for run 1 (0.4, 0, $\pi/2$), radius 0.5 m and velocity 0.175 m/s.

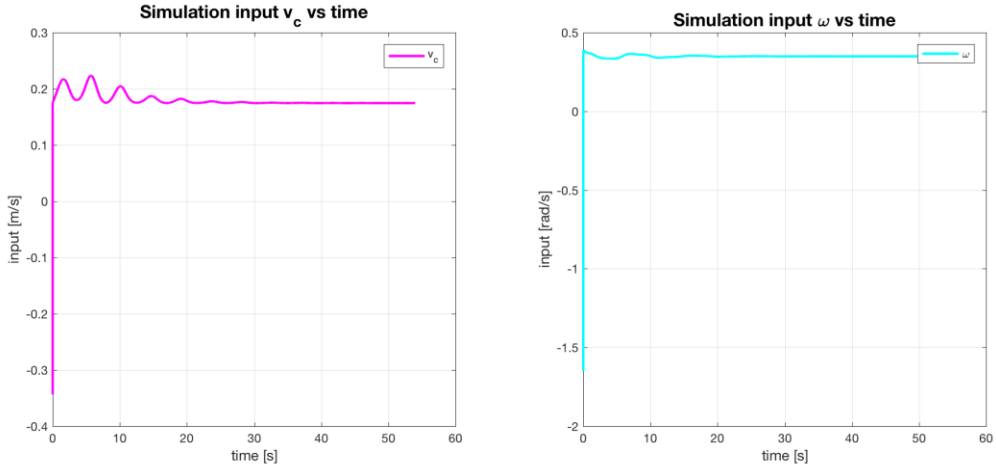

Figure 32. Simulated system inputs for run 2 (0.6, 0, $\pi/2$), radius 0.5 m and velocity 0.175 m/s.

As for the previous case, the two plots appear to be flipped vertically. It is worth mentioning, however, that the plots for the forward velocity contain significantly more oscillations that previously observed. These oscillations are due to the increased time it takes for the robot to converge onto the nominal path. The plots for the angular velocity, on the other hand, appear to be relatively flat, as noted earlier.

Nominal Path Radius: 0.2 m

The analysis continues by considering a small nominal path. The chosen radius is 0.2 m. This corresponds to a diameter of 0.4 m, which, for comparison, is just slightly larger than the robot's diameter of 0.35 m. The analysis now focuses on the standard testing velocity of 0.35 m/s. Travelling at this velocity through the circular path of radius 0.2 m, the robot takes just 3.6 second to complete a full lap and 21.6 s to complete the 6 laps that constitute one run. With the same sample time of 0.01 s, a total of 2,160 data points for each variable are generated. The LQR gains and the solutions to the Riccati equations are shown in Fig. 33. Since the neural networks are, once again, able to fully replicate the gains, their plots are omitted and only the results produced using the networks are presented.

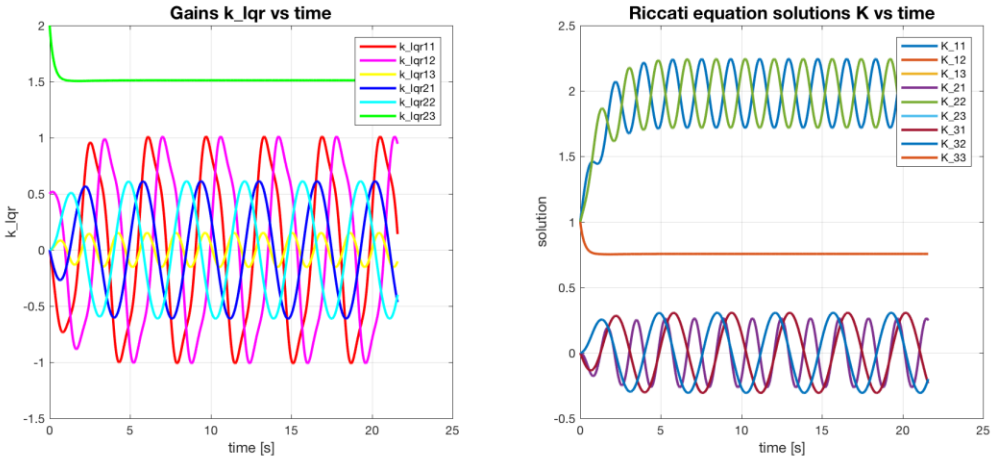

Figure 33. LQR gains and solutions to the Riccati equations for radius 0.2 m and velocity 0.35 m/s.

The system's response is tested with small deviations in the x-coordinate around the nominal initial condition $(0.2, 0, \pi/2)$, namely $(0.16, 0, \pi/2)$ and $(0.24, 0, \pi/2)$. The simulated paths and the tracking errors over time are shown in Figs. 34 and 35.

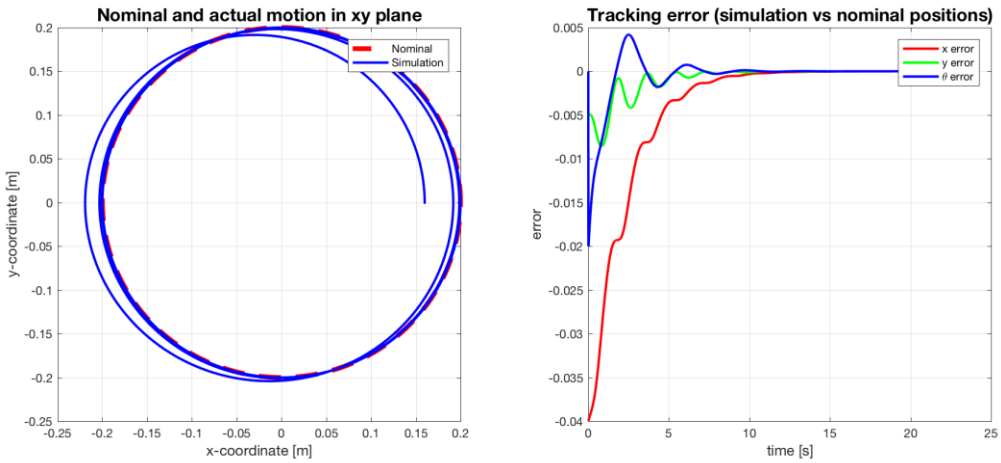

Figure 34. Simulated path and tracking error for run 1 (0.16, 0, $\pi/2$), radius 0.2 m and velocity 0.35 m/s.

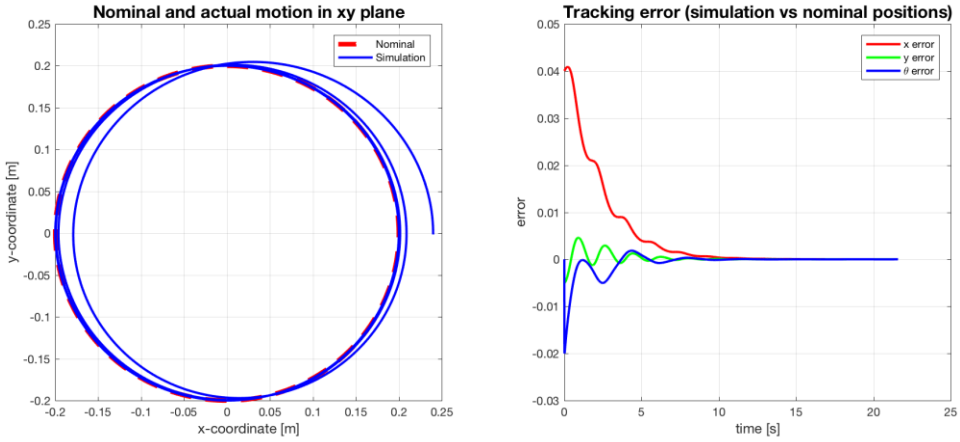

Figure 35. Simulated path and tracking error for run 2 (0.24, 0, $\pi/2$), radius 0.2 m and velocity 0.35 m/s.

In both cases, the simulated responses are successful, as the robot merges onto the nominal path by reducing the errors in approximately 10 s. It is worth noting, however, that in both cases the system noticeably overshoots and undershoots before stabilizing on the trajectory. This is attributed to the high tangential velocity relative to the small diameter of the path. Results from repeating the simulation with half of the tangential velocity, 0.175 m/s, have in fact shown that no overshooting or undershooting occurs in that case, making the system's response very similar to the 0.5 m experiment.

The plots for the system inputs for both runs are shown in Figs. 36 and 37. The forward velocity plots appear, once again, vertically flipped, for the same reasons mentioned

previously. This input becomes constant after just a few oscillations. The angular velocity plots, on the other hand, are very similar and appear to be almost entirely flat.

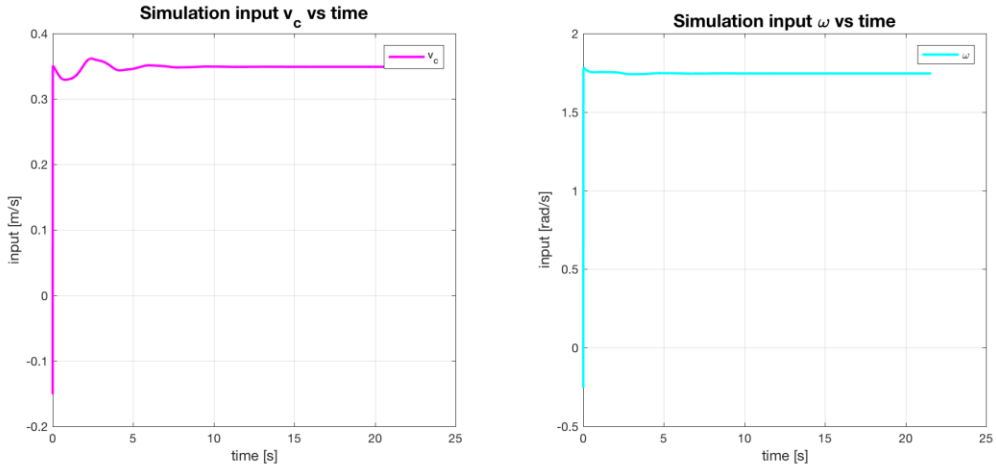

Figure 36. Simulated system inputs for run 1 (0.16, 0, $\pi/2$), radius 0.2 m and velocity 0.35 m/s.

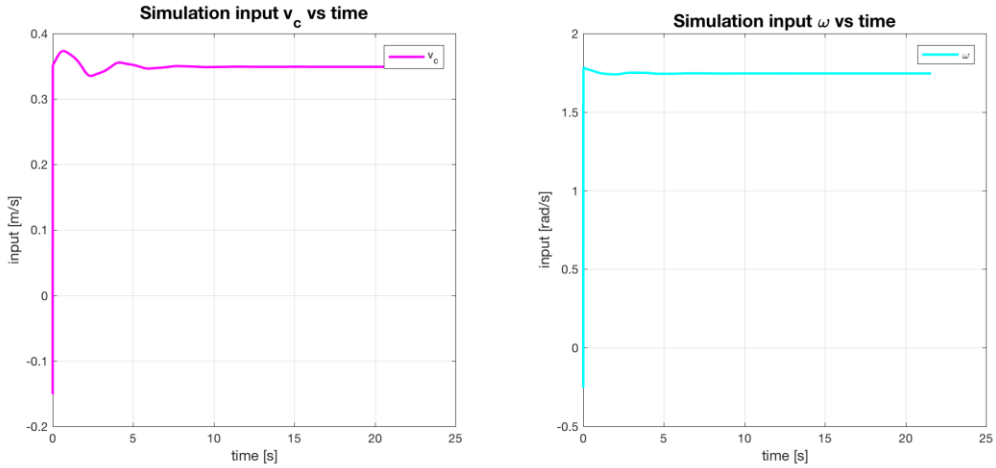

Figure 37. Simulated system inputs for run 2 (0.24, 0, $\pi/2$), radius 0.2 m and velocity 0.35 m/s.

Nominal Path Radius: 1.0 m

The simulation analysis continues with what can be considered a quite large radius for the nominal path, corresponding to 1.0 m. The analysis is still focused on the standard testing velocity of 0.35 m/s. Travelling at this velocity through the circular path of radius 1.0 m, the robot takes 18 second to complete one lap and 54 s to complete the 3 laps that

constitute one run. With the sample time of 0.01 s, a total of 5,400 data points for each variable are generated. The LQR gains and the solutions to the Riccati equations are shown in Fig. 38. Once again, the neural networks are able to fully replicate the gains and for this reason their plots are omitted.

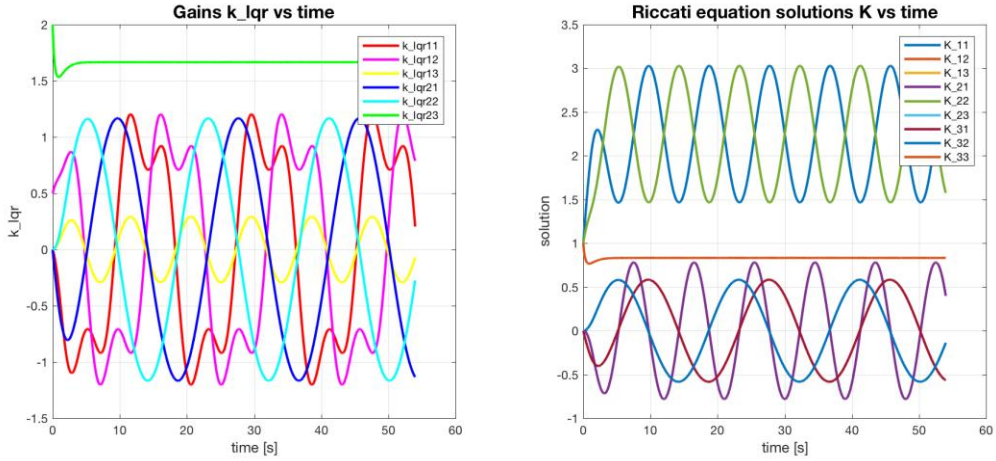

Figure 38. LQR gains and solutions to the Riccati equations for radius 1.0 m and velocity 0.35 m/s.

The system's response is tested with initial conditions $(0.8, 0, \pi/2)$ and $(1.2, 0, \pi/2)$, small variations in the x-coordinate around the nominal starting condition $(1, 0, \pi/2)$. The simulated paths and the corresponding tracking errors are shown in Figs. 39 and 40.

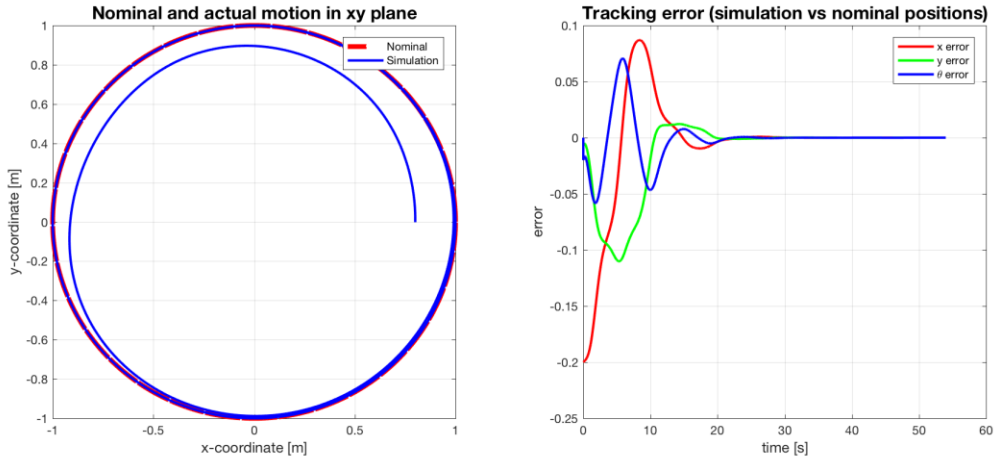

Figure 39. simulated path and tracking error for run 1 (0.8, 0, $\pi/2$), radius 1 m and velocity 0.35 m/s.

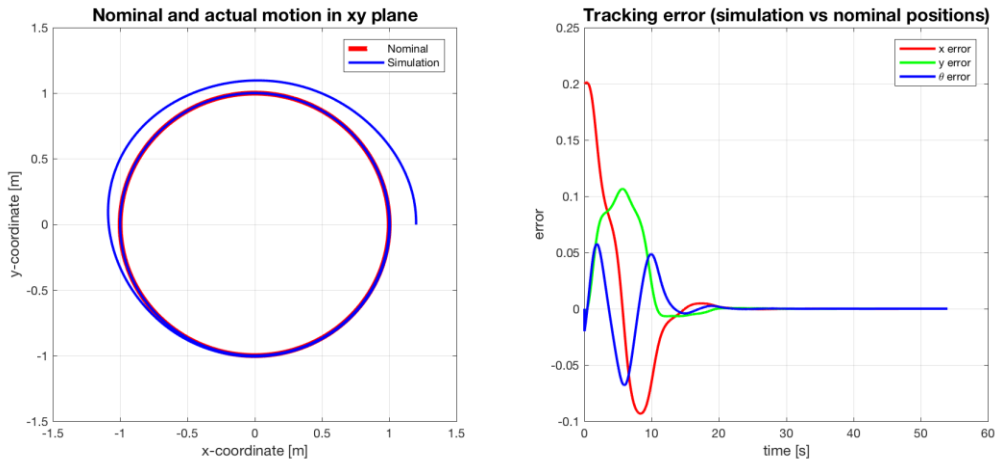

Figure 40. Simulated path and tracking error for run 2 (1.2, 0, $\pi/2$), radius 1 m and velocity 0.35 m/s.

As the plots show, the controller successfully and smoothly steers the system into the nominal trajectory, taking approximately 20 seconds to do so. This result is very similar to the simulation with the 0.5 m radius. The only difference seems to be that, in this case, the robot "coasts" next to the desired path for several seconds before merging into the trajectory, which results in the higher time it took for the errors to become zero.

As with the previous simulations, the plots for the inputs to the system are also generated, as shown in Figs. 41 and 42.

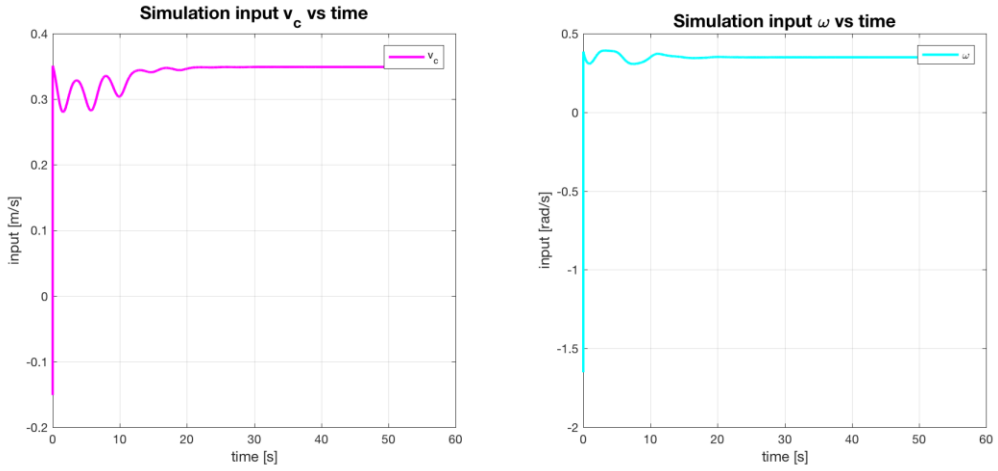

Figure 41. Simulated system inputs for run 1 (0.8, 0, $\pi/2$), radius 1 m and velocity 0.35 m/s.

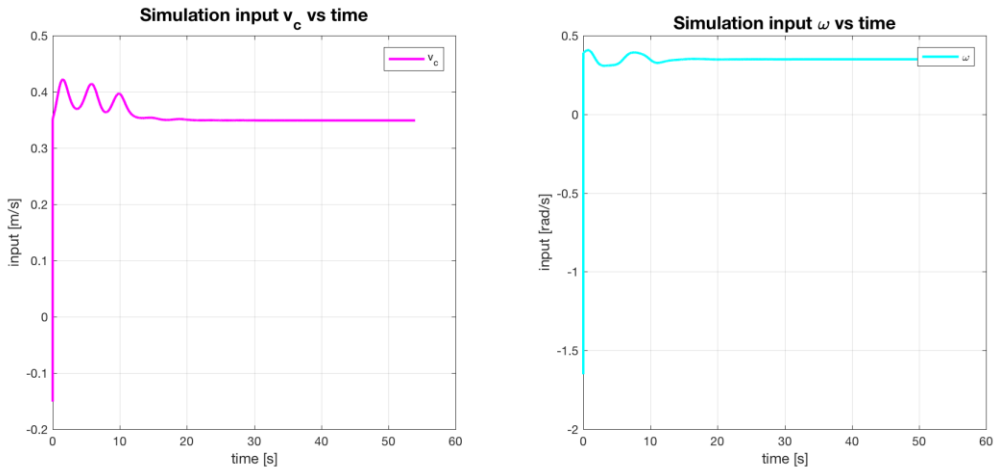

Figure 42. Simulated system inputs for run 2 (1.2, 0, $\pi/2$), radius 1 m and velocity 0.35 m/s.

The two plots appear, once again, to be flipped vertically. While the values are different, the input curves seem similar to the case with radius 0.5 m and tangential velocity 0.175 m/s. This similarity arises because this simulation is essentially a scaled-up version of the other one (twice the radius and twice the velocity). The relatively large amount of oscillations derives from the longer time it takes for the robot to converge onto the nominal path. Note how the angular input curve stabilizes at just over 10 seconds, while the input curve takes closer to 20 seconds to become constant.

Nominal Path Radius: 1.25 m

Finally, the analysis turns to the extreme scenario in which a very large radius for the nominal trajectory is chosen. This corresponds to a radius of 1.25 m, which represents the maximum circular path that fits within the work area in the actual lab. Using the standard testing velocity of 0.35 m/s, the robot takes 22s to complete a full lap and 66s to complete the 3 laps that constitute a simulation run. With the sample time of 0.01 s, a total of 6,600 data points are generated. The LQR gains and the solution to the Riccati equations for this scenario are shown in Fig. 43.

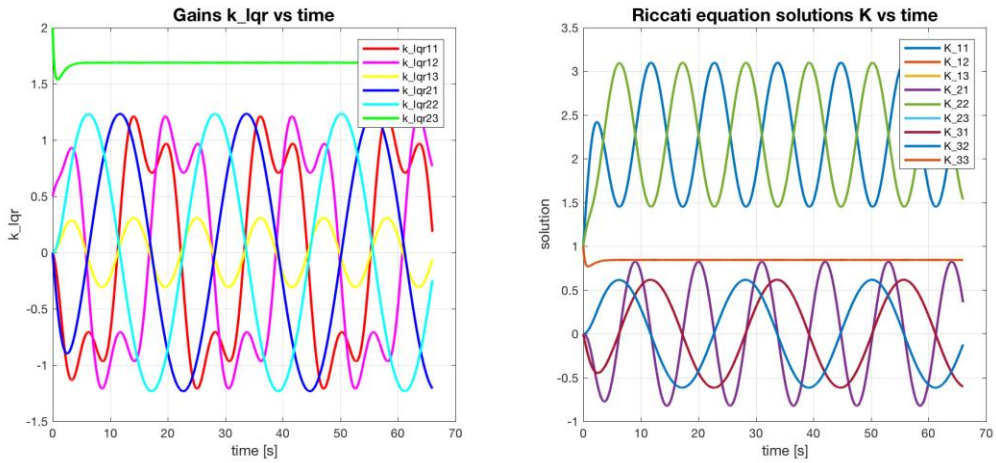

Figure 43. LQR gains and solutions to the Riccati equations for radius 1.25 m and velocity 0.35 m/s.

The overall trend of the curves in the above plots are the same as discussed previously, with the difference that the values are now higher as a result of the larger diameter of the nominal path. Again, the LQR values are perfectly learned and replicated by the neural network, thus there is no need to show the plots. The simulated path produced for the initial condition $(1, 0, \pi/2)$ is shown in Fig. 44.

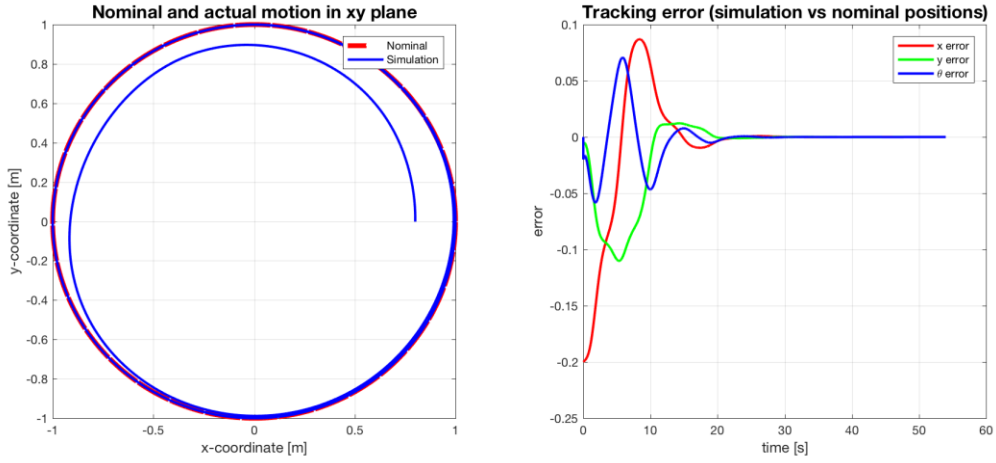

Figure 44. Simulated path and tracking error for run 1 (1, 0, $\pi/2$), radius 1.25 m and velocity 0.35 m/s.

These plots show that, even with the largest diameter path, the controller should be able to successfully steers the system onto its nominal path. The tracking error plot shows

that convergence happens in around 20 seconds. As with the previous cases, the path traced out by the robot is very smooth. The input plots are shown next in Fig. 45.

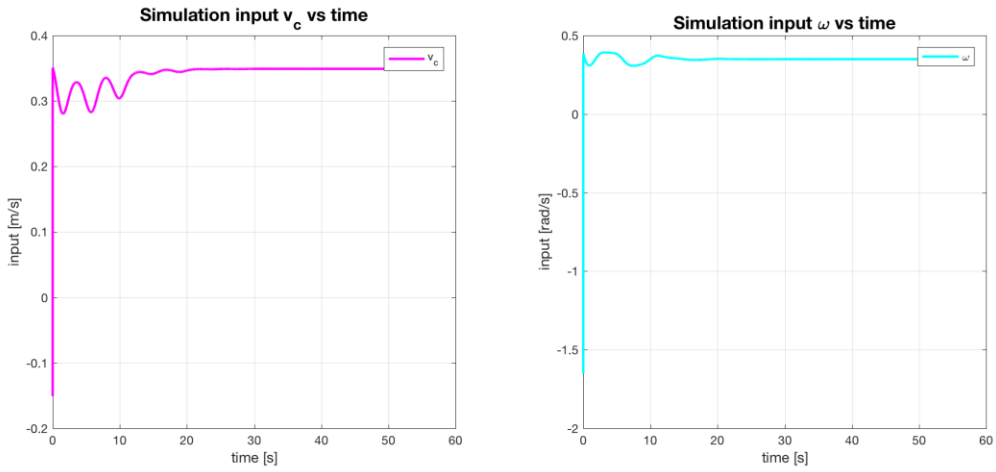

Figure 45. Simulated system inputs for run 1 (1, 0, $\pi/2$), radius 1.25 m and velocity 0.35 m/s.

Compared to the previous cases, these plots show a noticeable amount of oscillations, especially in the forward velocity plot. Overall, they appear similar to the case where the nominal path is 1 m in radius, with the main difference being the actual numerical values.

LIMITATIONS

The results presented so far are only a small sample selection of the successful outcomes. These cases, which will be again analyzed when testing on the actual QBot robot, are considered successful as the tracking error is quickly and smoothly removed. The positive outcome is also determined by the fact that the robot's motion is curvilinear and with no sudden changes in direction.

Because the LQR controller is based on a locally-linearized model of the robot, it is important to understand that it cannot be effective in any arbitrary scenario. In fact, as the tracking error becomes too large (i.e. the robot is very far from the nominal state), the system can become unstable and its motion erratic. Several examples of such behavior, triggered by letting the robot start from a state that is distant from the nominal starting state, are shown in Fig. 46.

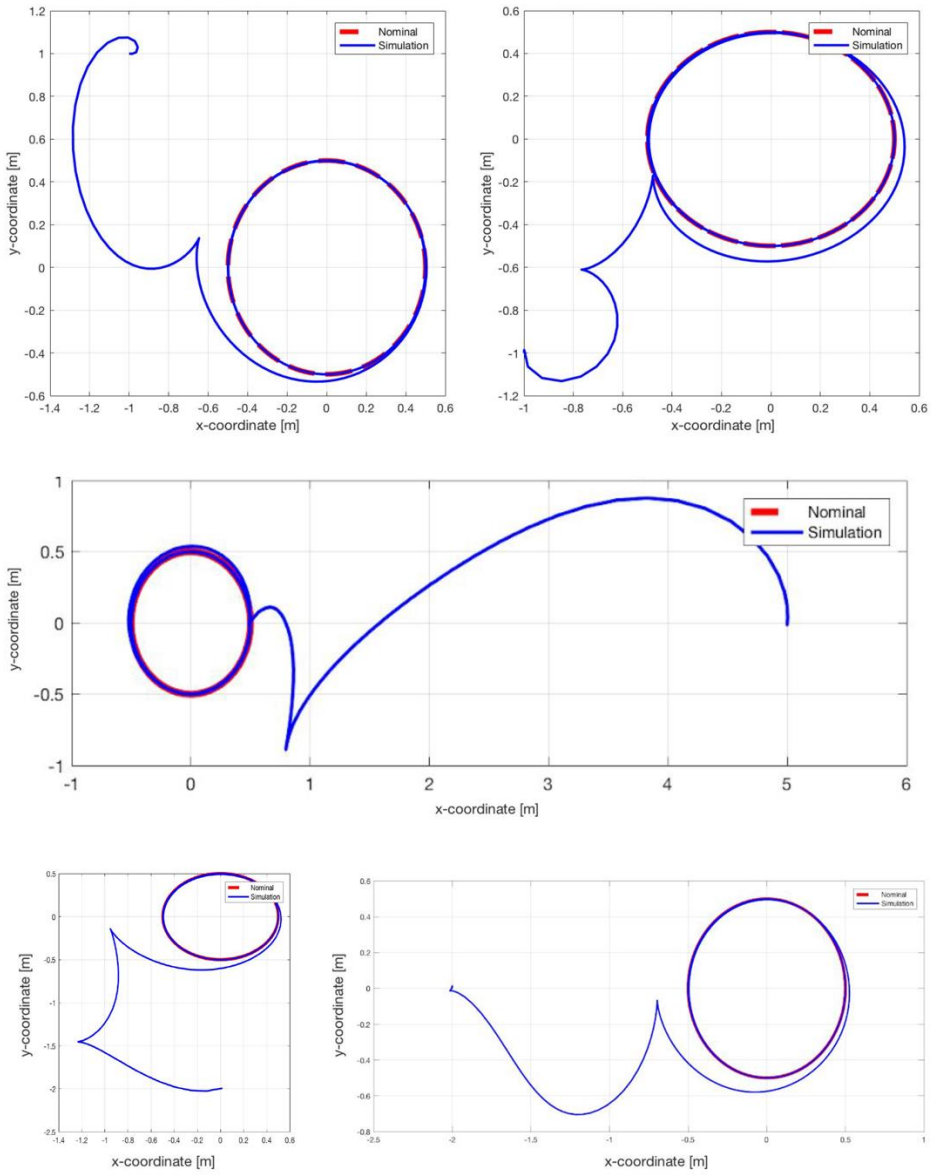

Figure 46. Examples of failed simulations (radius 0.5 m and velocity 0.35 m/s).

In these cases, either one or more of the three coordinates was largely varied in order to be far from the nominal initial conditions. While the robot eventually does reach the nominal condition and removes the tracking error, the way it attempts to reach to the desired path is considered incorrect. This is especially relevant because, while it may work in the simulation, it most certainly cannot work on the actual QBot. In particular, the abrupt changes in direction (at very acute angles) that happen instantaneously in the simulation are not realistically achievable by the actual robot. Actual testing in the lab reveals that these

changes in direction cause the robot to become unstable and to start either spinning around itself indefinitely or driving off the carpeted area at full speed.

The solution to this issue is to limit the choice of initial conditions to a specific range that is close enough to the nominal starting state. The boundary of the range is not clearly identifiable, as it would require running numerous simulations with increasingly diverging initial conditions until the system becomes unstable. With the simulations that were already performed, however, it is possible to get a clear picture of what starting conditions would be surely safe to start the system from. Assuming that the nominal initial condition is on the x axis ($y_0 = 0$) and that the robot is initially in the $\theta_0 = \pi/2$ orientation, the safe starting zone for the x and y coordinates is represented in Fig. 47.

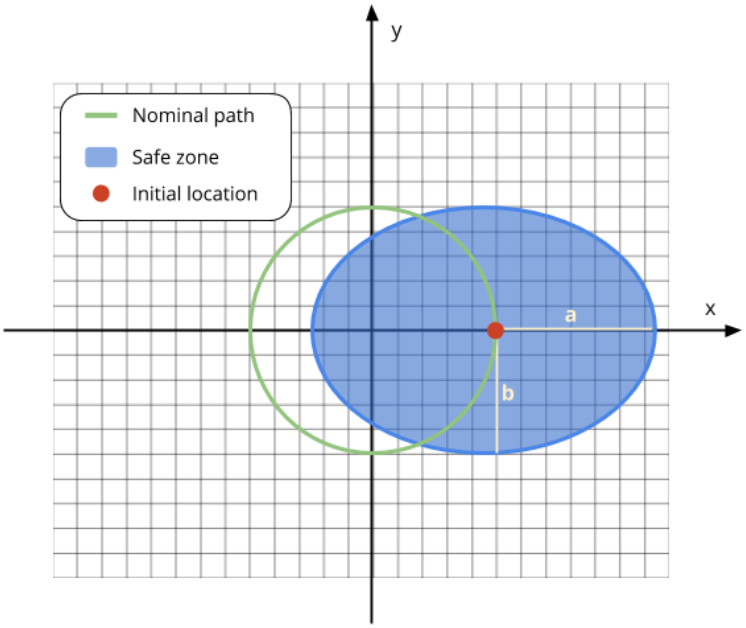

Figure 47. Safe zone for initial x and y coordinates.

As the figure shows, the boundary was found to have a close-to-elliptical shape, with $a = \frac{3}{2}r$ and $b = r$. When starting exactly on the boundary, the orientation of the robot must be nominal at $\pi/2$ in order for the robot to remain stable. The closer the initial x and y coordinates are to the nominal starting coordinates, the more the initial angular position can differ from $\pi/2$. It was also found that by fixing the initial x and y coordinates to the initial

nominal position $(r, 0)$, the robot's orientation can be varied arbitrarily between $-\pi/2$ and $\frac{3}{2}\pi$, a full 2π of possible choices, while still guaranteeing stability.

CHAPTER 5

EXPERIMENTATION

This section is concerned with the physical experiments performed using the QBot 2 and the OptiTrack system. The goal is to show that the conclusions obtained from the theory and the MATLAB-based numerical simulations hold when applied to the actual physical model. Before presenting the results, the experimental setup to run the Simulink model and to deploy the code to the robot, as well as the techniques to track its position and collect data in real time, are illustrated. Finally, the obtained results are discussed to show their validity and to highlight the shortcomings of the experiment.

EXPERIMENTAL SETUP

As mentioned in the Hardware Description section towards the beginning of this thesis, the experimentation is performed in the laboratory on a Quanser QBot 2 robot. The robot is connected to a wireless local network, to which a host computer running Windows 7 64-bit is also connected. The computer runs MATLAB version R2014a, which is recommended for best compatibility with Quanser's software. It is also directly linked to the OptiTrack Flex 3 system and runs Motive version 1.7.5 to read data from the cameras, as described in Localization Techniques section. The entire setup was configured following the instructions provided by Quanser and thus does not need to be discussed here.

Before running the experiment, the camera system needs to be calibrated through the Motive software by performing the so-called "wanding" operation and the origin-placement, using the tools shown in Fig. 48. The process consists of first moving the calibration wand around the area to sync the six cameras and increase their precision. The result of this step is represented in Fig. 49, which shows the Motive software confirming the successful outcome of the operation with an achieved camera accuracy of less than 1mm. Next, the calibration

square is placed in the center of the working area to mark the origin of the virtual coordinate frame.

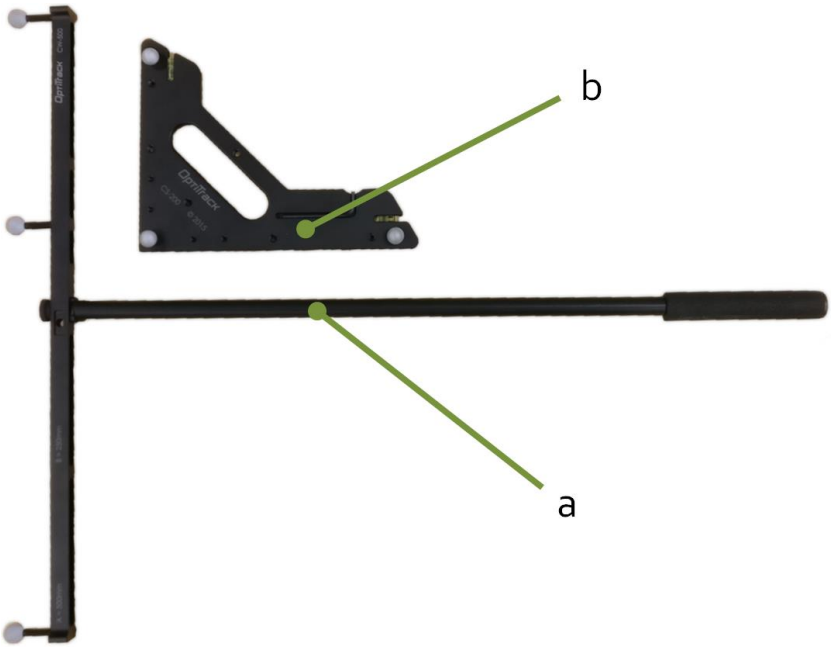

Figure 48. OptiTrack camera system calibration tools: (a) calibration wand and (b) calibration square.

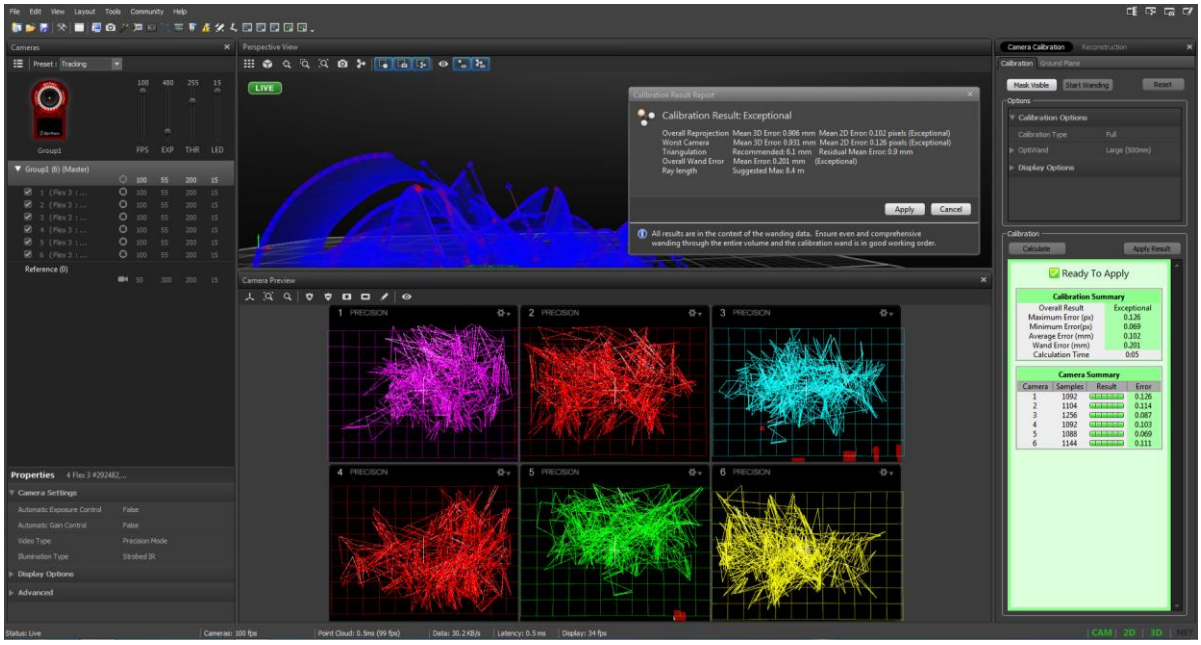

Figure 49. Motive interface after the "wanding" operation showing successful camera calibration.

Once the calibration is complete, a rigid body is created by placing the QBot with the three reflectors on the tracking area, facing the positive x-direction, and selecting the 3 markers in Motive. The calibration and the rigid body settings can then be exported for later use and Motive can be closed. The attention is now turned to MATLAB and the files shown in Fig. 50.

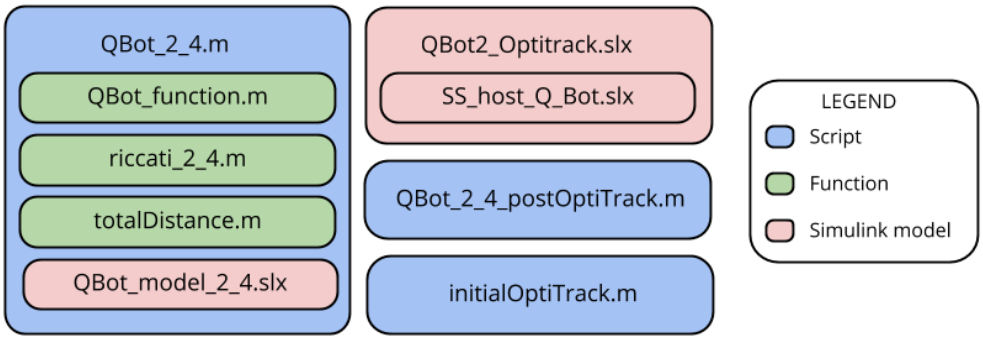

Figure 50. Dependencies of MATLAB scripts, functions and Simulink models for experiments.

The actual experiments can now be performed. The following are the steps required for one test with a given nominal path, period and initial conditions:

1. Compute LQR gains offline:

   The same script used in the previous section, called "QBot_2_4.m", is run. The parameters for the desired path, including the time settings, need to be specified at the top. The initial state of the system (x_0, y_0, θ_0) can be set to any arbitrary values at this stage, as they will be reviewed after the experiment is performed to match the actual QBot's initial position and orientation. Running the script is required to compute the LQR gains and generate the nominal states that will later be used in real-time.

2. Initialize QBot tracking:

   Open the "SS_host_Q_Bot.slx" Simulink model, shown in Fig. 51. This model reads the information from the OptiTrack camera system in real-time and converts the data into Simulink-usable global variables. First, inside the "OptiTrack Trackables" block, ensure that the "Calibration file" and the "Trackables definition file" settings are reading the correct files, previously exported from Motive. Next, build the model, connect to the target and run. If the QBot is within the trackable area, its x, y, and z coordinates, as well as its yaw, pitch and roll angles, are displayed in real-time.

   Place the QBot close to the desired initial coordinates and orientation. Because of the high sensitivity of the cameras, all values are slightly but always changing, so

achieving a specific location accurate to all decimal places is impractical and unnecessary. In the Simulink model, also note the single display that shows a "1" when the camera system is successfully tracking the robot, or a "0" otherwise. Once started, this model can be minimized and left running for the duration of several experiments.

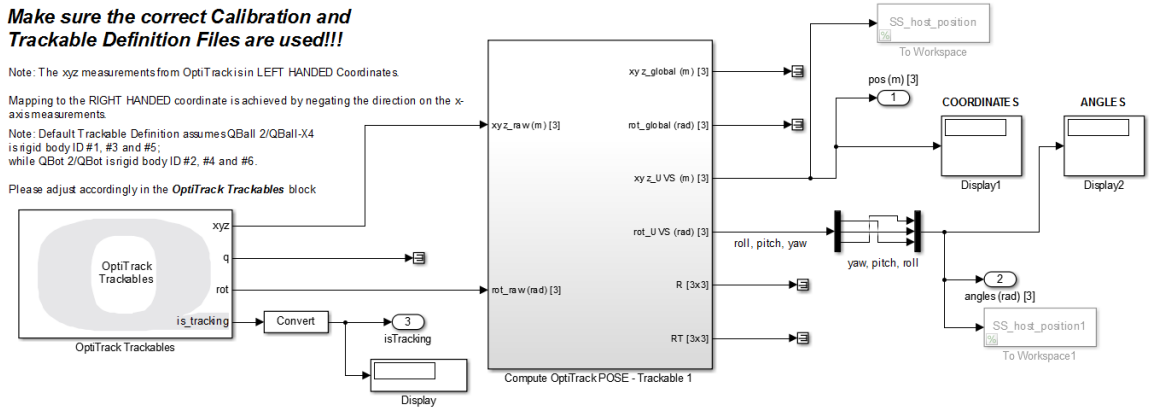

Figure 51. "SS_host_Q_Bot.xls" Simulink model.

3. Start the experiment:

   Open the Simulink model called "QBot2_Optitrack.xls", shown in Fig. 52. This model mimics the functioning of the simulation model examined in the previous chapter, with the main difference being that the outputs of the system are now the x, y and θ variables read from the OptiTrack system, instead of the output of the QBot function. The model also includes several blocks that are required for communicating with the robot and with the SS_host_Q_Bot model running in the background. The two switches on the left can trigger the activation of the input signals (top one) and the use of the LQR (bottom one) respectively. Also, several gains, used primarily for testing purposes and fine-tuning for matching of theoretical and practical, are also found throughout the model.

   The system inputs are split into two signals. The first goes into the virtual plant, shown in Fig. 53. This plant contains an equivalent version of the QBot function, converted from code into a Simulink diagram, shown in Fig. 54. The second signal, more importantly, goes into the Motor Actuation subsystem, shown in Fig. 55. This subsystem contains two blocks provided by Quanser, "QBot 2 Inverse Kinematics" and "QBot 2 Basic Motor Commands and Sensor Measurement", which convert the inputs given in unicycle model (v_C and ω) into differential-drive model inputs (ω_L and ω_R) and sends the converted signals to the actual robot. The on-board computer then takes care of converting these signals into the corresponding analog signals automatically (using PWM).

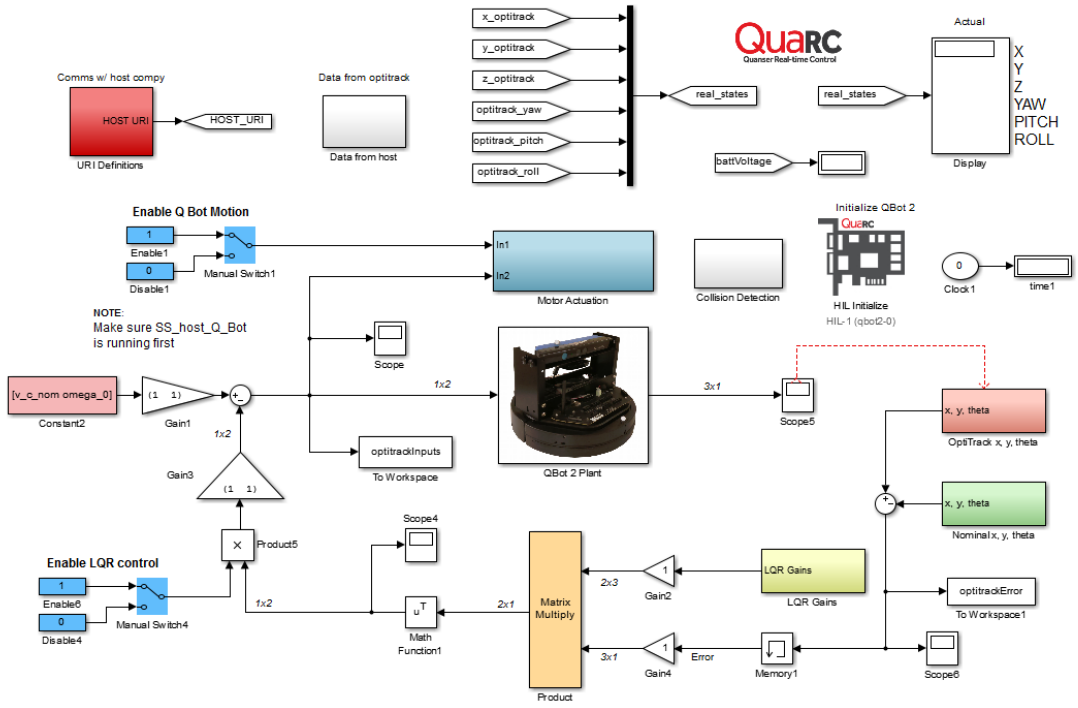

Figure 52. "QBot2_Optitrack.xls" Simulink model.

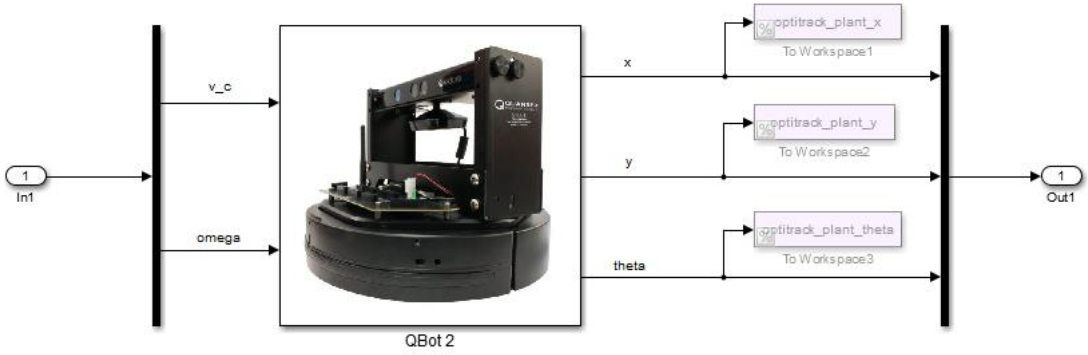

Figure 53. "QBot 2 Plant" subsystem in the "QBot2_Optitrack.xls" Simulink model.

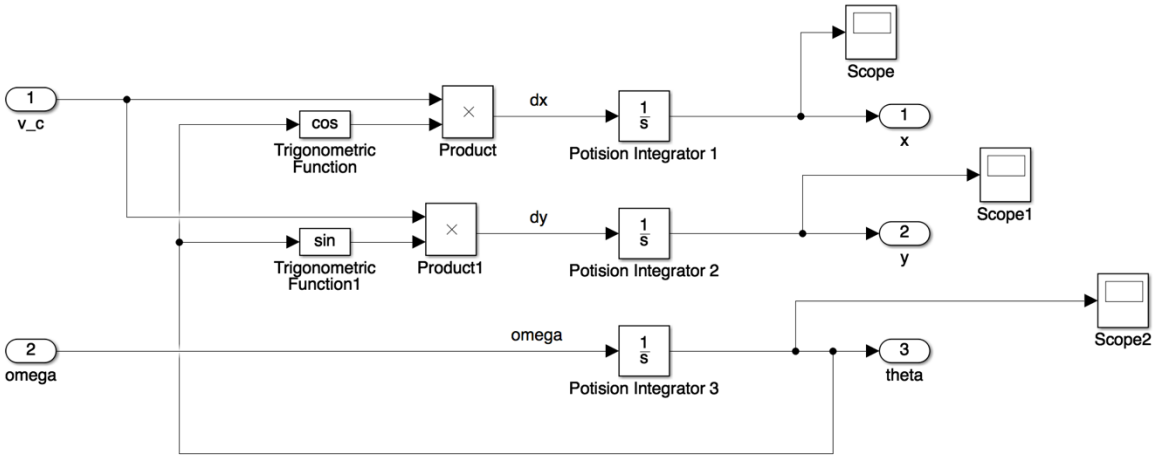

Figure 54. "QBot 2" subsystem, equivalent to the QBot function, inside the "QBot 2 Plant."

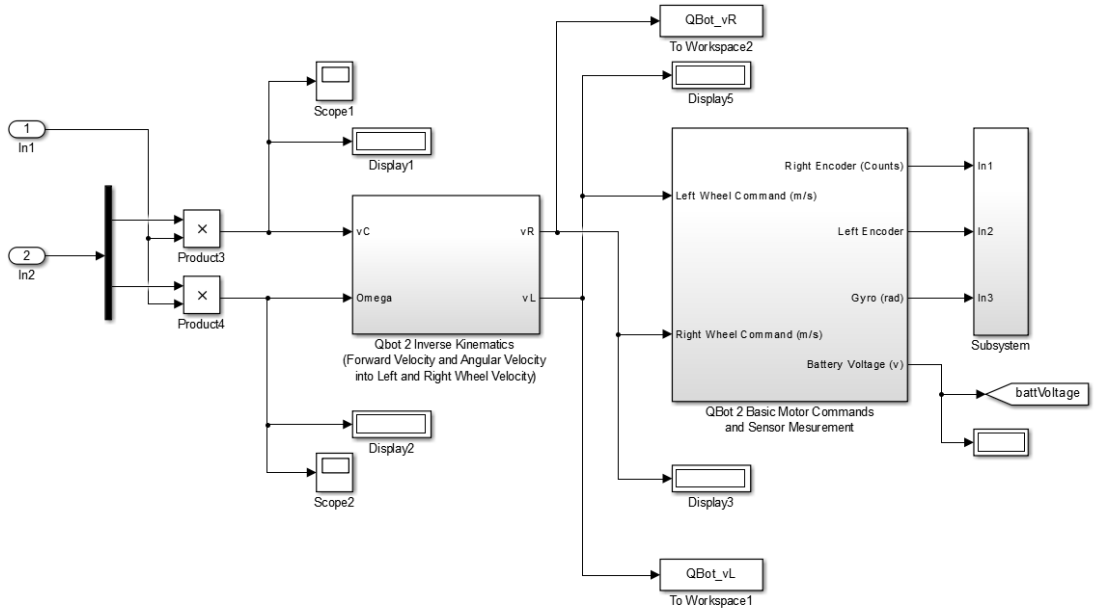

Figure 55. "Motor Actuation" subsystem in the "QBot2_Optitrack.xls" model.

The "LQR Gains" and the "Nominal x, y, theta" subsystems are identical to the ones used in the "QBot_model_2_4" model and therefor need no explanation. The "OptiTrack x, y, theta" subsystem, shown in Fig. 56, however, is new and presents an additional feature. Its main purpose is to read in the values from the tracking system.

One issue, however, arises from the fact that OptiTrack measures the angular position from $-\pi$ to $+\pi$, while the nominal position is not bounded. Therefore, if

the robot was to rotate past, for example, $+\pi$, its measured angle would switch to around $-\pi$. Since the nominal angle would instead keep increasing above $+\pi$, the tracking error for θ would suddenly jump to around -2π. To solve this issue, the subsystem "Angle Tracker", shown in Fig. 57, was created in order to keep track of the sign switching and turn the angle measurement from bounded to unbounded. This is achieved through the use of a MATLAB function, whose code is displayed in Appendix E.

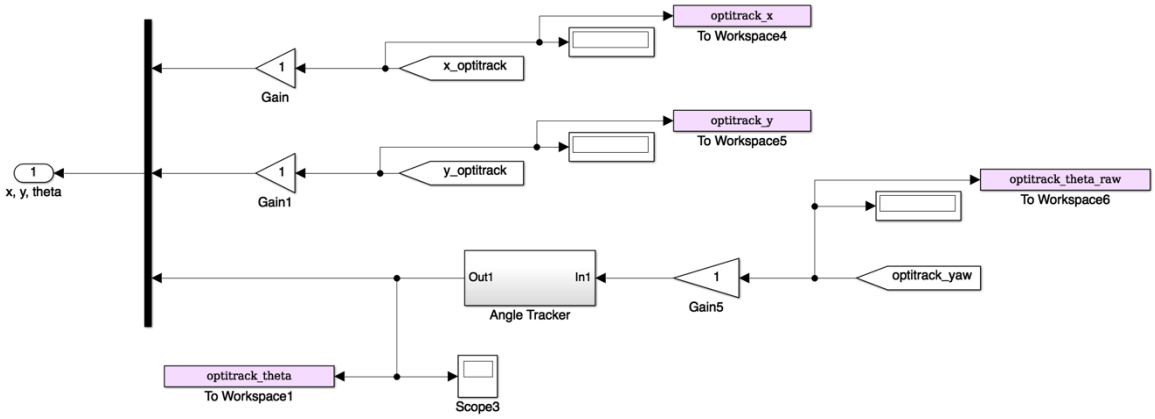

Figure 56. "OptiTrack x, y, theta" subsystem in the "QBot2_Optitrack.xls" Simulink model.

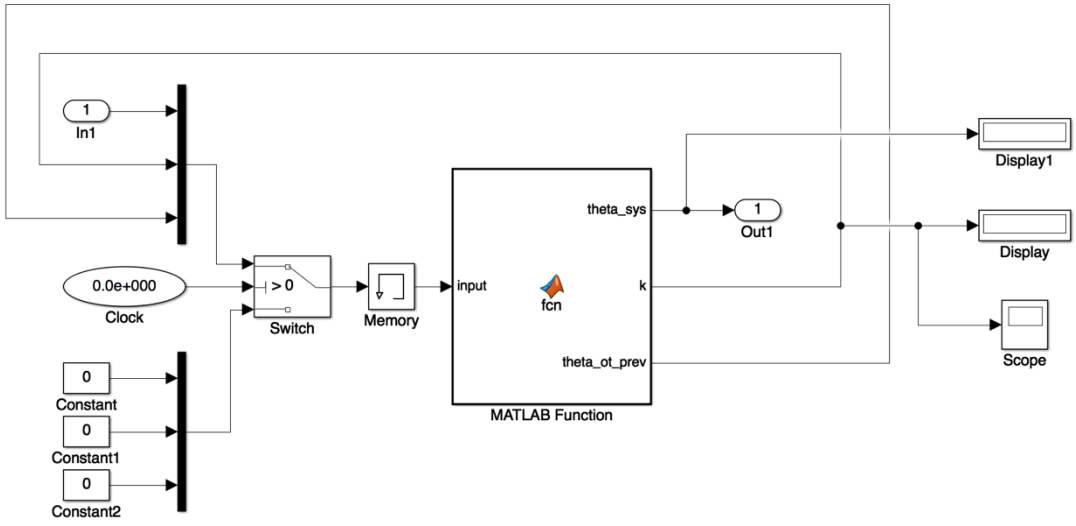

Figure 57. "Angle Tracker" subsystem inside the "OptiTrack x, y, theta" block.

This function takes in a vector containing the current angle $\theta_{ot,current}$ (measured by the OptiTrack system), the previous counter k, and the previous angle $\theta_{ot,prev}$ (also measured by the OptiTrack system). By comparing the current and the previous angle values against a set threshold, the function can determine whether the OptiTrack angle has switched sign around the $\pm 2\pi$ mark. It then keeps track of such change with the variable k, which indicates how many complete rotations the robot has completed. The unbounded angle, θ_{sys}, can then be computed with the equation

$$\theta_{sys} = \theta_{ot,current} + k(2\pi)$$

The output signal of the "OptiTrack x, y, theta" block, now containing the unbounded angle measurement, is then compared to the nominal state of the system, called by the block "Nominal x, y, theta" from the workspace, generating the tracking error. Next, the error is multiplied by the LQR gain and subtracted from the nominal input before being sent back to the plant, thus completing the closed-loop. Throughout the model, several variables are recorded and exported to the workspace for analysis.

To begin the experiment, first build the QBot2_Optitrack model and connect to the target. Wait at least 5 seconds before actually running the model to ensure that the connection between the host computer and the QBot robot has been fully established. Once the "Run" button is pressed, the robot will start moving.

4. Start comparative simulation:

   Once the QBot2_OptiTrack model has finished running, the relevant data acquired during the experiment is available in the MATLAB workspace. For now, the most useful information comes from the position and orientation values recorded over time. These are store as "optitrack_x", "optitrack_y" and "optitrack_theta". Ignore the first few values in each array, as they are merely initial placeholder (erroneous) values and not actual readings from the tracking system. Observe the values up to about the sixth time instant. A handy short script, called "initialOptiTrack.m" and shown in Appendix F, can be run to glance at these values and quickly identify the first good recorded measurements. These values can then be set as the initial conditions it in the "QBot_2_4.m" script. This way, by running the script again, the simulation will execute with exactly the same initial state as the one of actual robot before the start of the experiment.

5. Compare experiment and simulation:

   Once the QBot_2_4 script has finished running, the workspace now contains both simulation and experimental data with precisely identical initial conditions. An additional script, called "QBot_2_4_postOptiTrack.m", was created to quickly generate plots to compare the simulation and the experiment. The script, shown in Appendix G, has three switches at the top, whose functions are the following:

   - plotResults: if true, the script will display several plots, comparing the path, the coordinates, the tracking error, and the inputs recorded during the simulation and during the experiment.

- exportPlots: if true, the plots generated by the plotResults switch will be saved to the current folder as PNG images.
- exportWorkspace: if true, the entire workspace is saved as a MAT file to be later retrieved and analyzed.

The titles for the PNG and MAT files are automatically generated to highlight the main experiment settings and parameters. The figures generated by this script are crucial for the analysis of the experimental results, presented in the next section.

INPUT GAINS CORRECTION

Initial experimental attempts showed a seemingly correct response of the QBot. Starting from a position with some initial error compared to the nominal position, the robot quickly corrected its state and merged into a circular trajectory. It was soon discovered that the radius of the steady-state trajectory did not actually match the desired radius. Much research and experimentation were needed to identify the source of this error. The solution was found by running the system with open-loop control for each radius of the nominal path (0.2, 0.5, 1.0 and 1.25 m). For simplicity, only one case, with radius of 0.5 m, is covered with detail in this section, but the same analysis was performed on all other cases as well.

Running the system in open-loop around a circular path of radius 0.5 m produced the response shown in Fig. 58. Clearly, the robot stabilized on a trajectory with a radius noticeably larger than the nominal path, specifically a radius of 0.535 m.

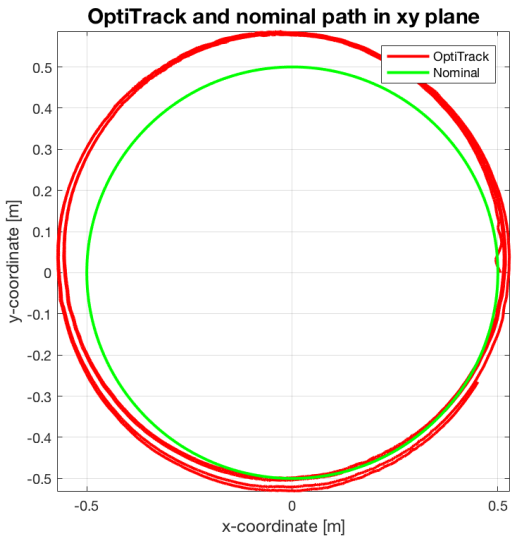

Figure 58. Open-loop response without correction.

It was found that the error was caused by imprecise nominal inputs, specifically the nominal forward velocity $v_{C,nom}$. To solve the issue, empirically-determined correction gains are placed after the nominal inputs in the Simulink model "QBot2_Optitrack.slx", as was shown on the left of Fig. 51. For the 0.5 m radius, the best gain values were found to be (0.83, 1). Running the QBot with open-loop control using these correction gains produced the response shown in Fig. 59. The robot now drives in a circular path with the expected radius.

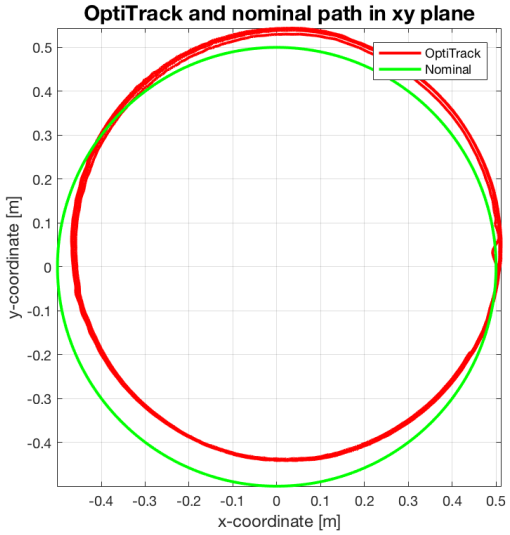

Figure 59. Open-loop response with correction gains.

These open-loop tests were repeated for the other radii used for the experiments. Each time, the correction gains were adjusted until the robot produced a path with the desired radius. The best values identified for each radius are shown in table 3.

Table 3. Nominal Input Correction Gains and Corresponding Radii

Radius [m]	Gains $v_{C,nom}$	Gains ω_0
0.2	0.675	1
0.5	0.83	
1.0	0.875	
1.25	0.90	

This operation of correction-gains tuning can be considered as a calibration of the system to sync the software with the hardware. A slight discrepancy between the theoretical model and the physical system is always present, and this is the solution identified in this

case specifically to overcome this obstacle. As the next sections will show, this approach does indeed solve the issue and ensures that the robot performs as expected.

EXPERIMENTAL RESULTS

The results from the experiments are now presented in a similar fashion to the simulations. All results are generated using the MATLAB scripts and Simulink models described in the previous sections. For the simulations, only small variations in the x-coordinate were considered for the initial conditions. Now, more scenarios are evaluated in addition to the simple case, to challenge the controller and test its performance in harder cases. A summary of the experiments is reported in Table 4.

Table 4. Summary of Experiments

Nominal Radius r_{nom} [m]	Tangential Velocity v_C [m/s]	Equivalent Period T_{eq} [s]	Section
0.5 (medium)	0.35	9	5.3.1
0.2 (small)	0.35	3.6	5.3.2
1.0 (large)	0.35	18	5.3.3
1.25 (extra-large)	0.35	22	5.3.4

Unlike for the simulations, only the cases with the higher tangential velocity are reported here. Actual testing with both low velocity (0.175 m/s) and high velocity (0.35 m/s) has shown that the robot performs well in both cases. For this reason, the easier case with the low velocity is omitted.

The plots for the LQR gains and the solutions to the Riccati equations over time are now omitted, as they are the same as the ones produced for the simulations. Note that the LQR gains are from now on read directly from the workspace, as the performance of the lab computers does not allow for real-time usage of the trained neural networks. Plots showing the robot's path, the tracking errors and the inputs are still used to analyze the performance. Side-by-side comparisons with the simulations are included to directly show the similarities.

Nominal Path Radius: 0.5 m

The analysis begins with the standard nominal path radius of 0.5 m (the medium-sized trajectory). Traveling at the tangential velocity of 0.35 m/s, it takes the robot 9 s to

complete one full circle. For the experiments, the number of laps that the robot travels is varied between 4 and 6, corresponding to total run times of 36 s and 54 s respectively.

First, the simple cases, already seen in the simulations section, are considered. Here, the robot starts from small variations in the x-coordinate around the nominal initial condition $(0.5, 0, \pi/2)$: the first run starts at approximately $(0.4, 0, \pi/2)$ and the second at approximately $(0.6, 0, \pi/2)$. The responses of the system are shown in Figs. 60 and 61. The figures contain the actual path of the QBot robot as tracked by the OptiTrack system on the left and the simulated path on the right.

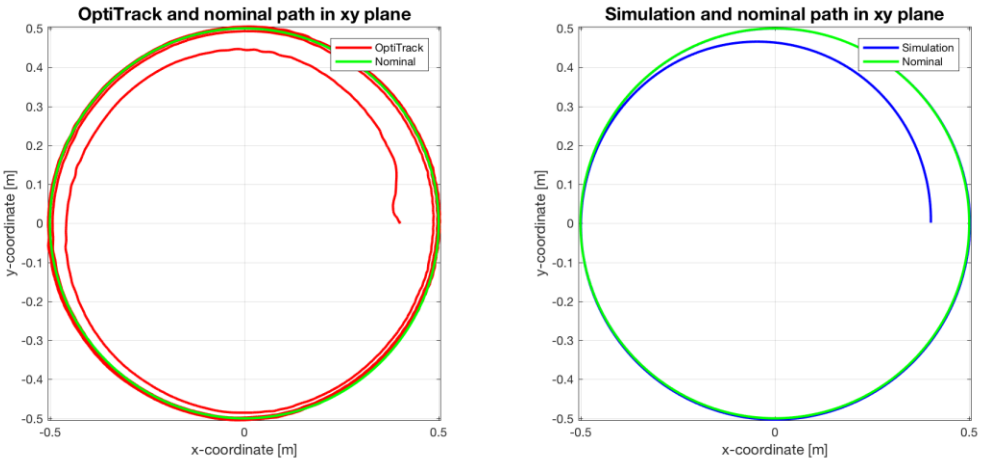

Figure 60. Actual and simulated path for run 1 (0.4, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

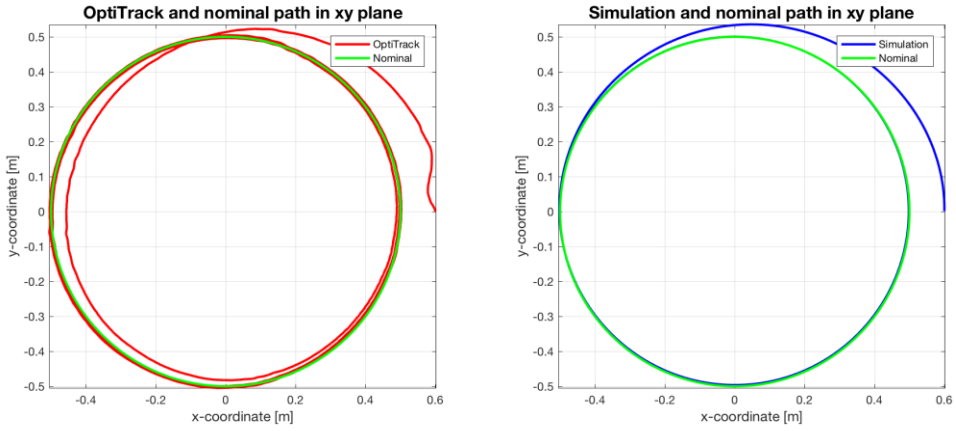

Figure 61. Actual and simulated path for run 2 (0.6, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

While overall the robot successfully merges onto the nominal trajectories, it is immediately noticeable that there is some discrepancy between the real and the simulated paths. This difference is mainly attributed to two reasons. First, when each experiment begins and the clock starts ticking, the robot experiences a delay before it actually starts moving. This means that the robot "wakes up" at a time that is really $t > t_0$ instead of $t = 0$. At that point, the nominal position is not the initial nominal position any more. Therefore, the robot finds itself immediately behind compared to where it should have started. On the other hand, the robot in the simulation always starts at exactly $t_0 = 0$, without any delays interfering with its response. The second issue is partially a result of the first one. When the robot first receives the command to turn its wheels, it always does so very energetically. This initial jolt in the motion is the reason why, in the paths traced by the actual robot, there is a slight oscillation right at the beginning. This phenomenon only happens at the very beginning of its motion and might be caused by the robot's need to compensate for the nominal position being ahead of the robot (the first issue). The initial jolts can be observed in the system inputs plots shown in Figs. 62 and 63.

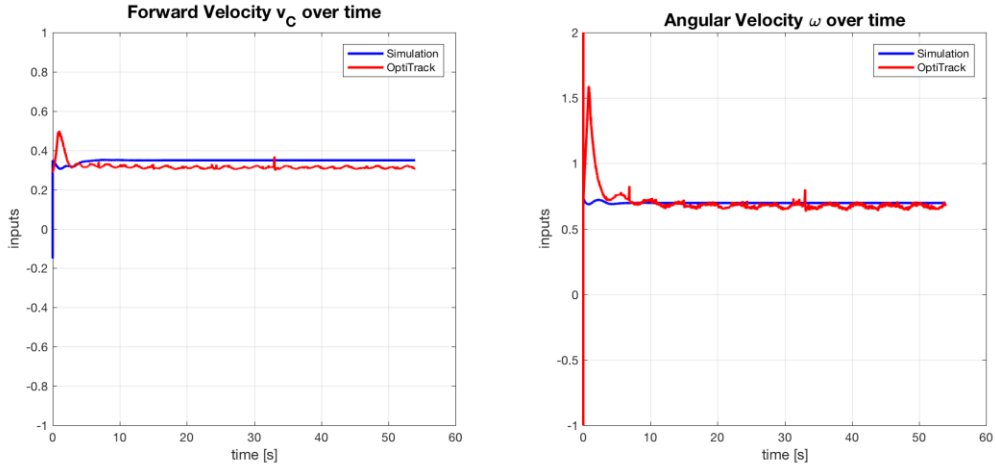

Figure 62. Actual and simulated system inputs for run 1 (0.4, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

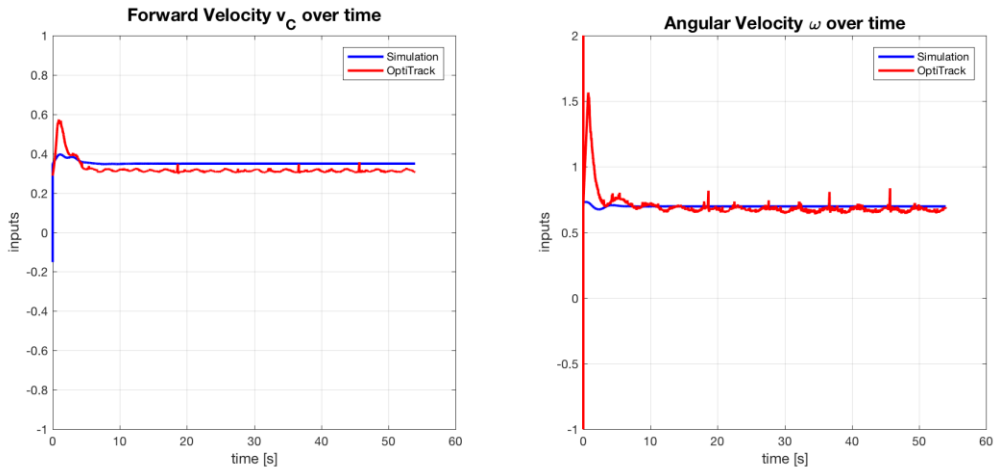

Figure 63. Actual and simulated system inputs for run 2 (0.6, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

As is evident from these plots, the initial peaks in inputs appear only in the actual (red) curves and are absent in the simulated (blue) curves. Another interesting observation regarding all four plots is the constant oscillations in the actual inputs, even though the simulation inputs appear very smooth. Looking closer at the red curves, it becomes clear that many of the oscillations' peaks happen periodically, commonly separated by 9 s. These peaks, and the oscillations in general, are caused by irregularities in the ground that the robot is travelling on and cause the spikes every time its wheels drive over them. More details and other explanations will be provided later on about this issue, as it is a common problem that is present in all experiments.

The next observation is concerned with the steady-state value reached by the input curves. In the simulations, both forward velocity and angular velocity reach constant values once the tracking errors reach zero. In the red curves, however, not only are there continuous oscillations, but the number they oscillate around is clearly not the same value as in the simulations. In all cases, in fact, the actual value is less than the simulated value. This issue is particularly evident with the forward velocity inputs. The difference in final input values between the actual system and the simulation is due to the overall discrepancy between theoretical and practical. As with the corrective input gains, the real system seems to need less input than what is calculated in theory to maintain its correct position and orientation around the nominal trajectory.

Next, the tracking errors are analyzed. Their plots, both for the actual system and for the simulation, are shown in Figs. 64 and 65.

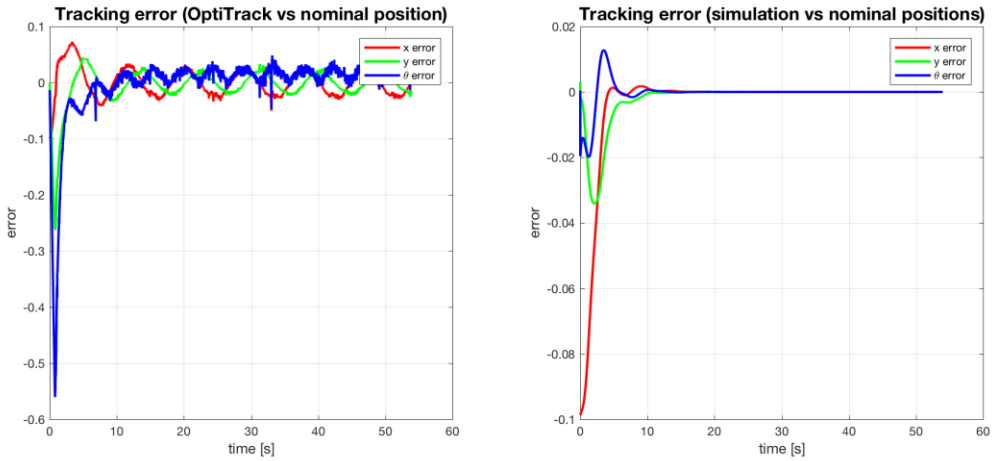

Figure 64. Actual and simulated tracking error for run 1 (0.4, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

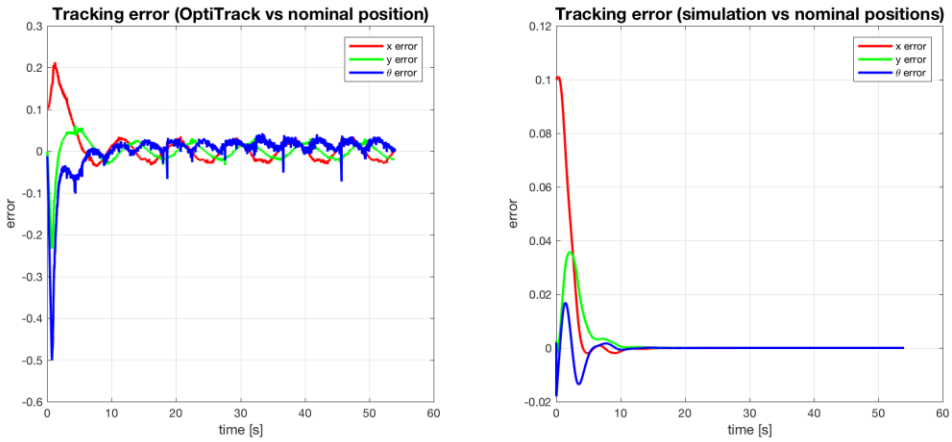

Figure 65. Actual and simulated tracking error for run 2 (0.6, 0, $\pi/2$), radius 0.5 m and velocity 0.35 m/s.

For these figures, it appears, at first glance, that the actual and the simulated plots have significant differences. When putting aside the initial jumps in the actual plots, the differences are actually not so large. The actual plots contain many oscillations, which appear to be, once again, separated by about 9 s. Just as for the input plots, these oscillations have been found to be caused mainly by irregularities on the ground that the robot is travelling over. The most important aspect to look at, however, is the fact that all errors in both the

simulated and actual plots converge to zero (or oscillate around it), meaning that the controller has successfully steered the system onto the nominal path.

Next, three experiments with substantially different initial conditions are now considered. For the first time in this thesis, the robot will now start from an initial position and orientation that is not simply a small variation with respect to the nominal starting state. The new initial conditions for runs 3, 4 and 5 are approximately $(0, 0, 0)$, $(0.5, 0, 0)$ and $(-0.5, 0, -\pi/2)$. These are just some examples of what the robot can accomplish, and other different starting conditions will be examined with the other radii for the nominal path. The paths produced by the robot, both in reality and in the simulations, for these runs are shown in Figs. 66, 67 and 68.

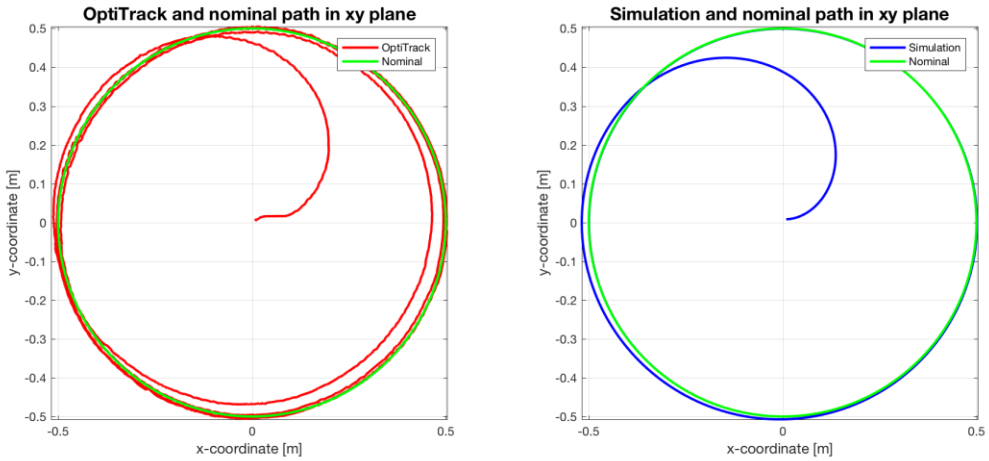

Figure 66. Actual and simulated path for run 3 (0, 0, 0), radius 0.5 m and velocity 0.35 m/s.

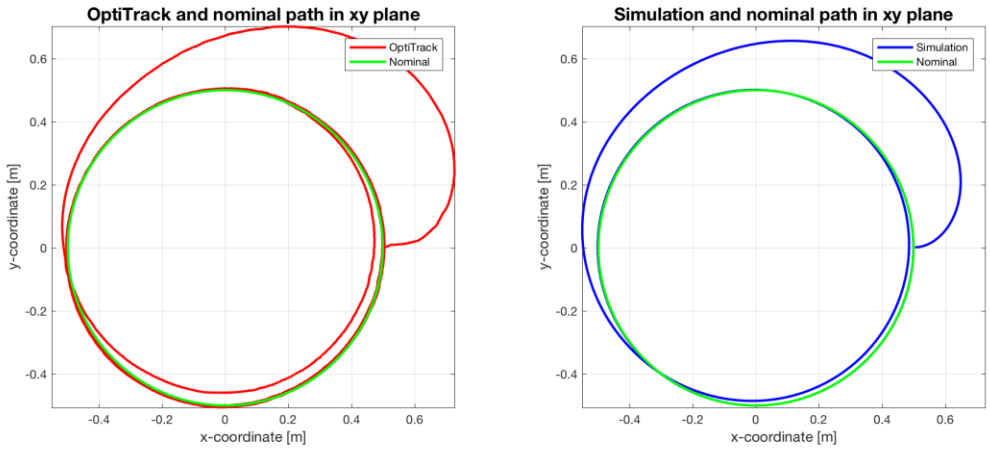

Figure 67. Actual and simulated path for run 4 (0.5, 0, 0), radius 0.5 m and velocity 0.35 m/s.

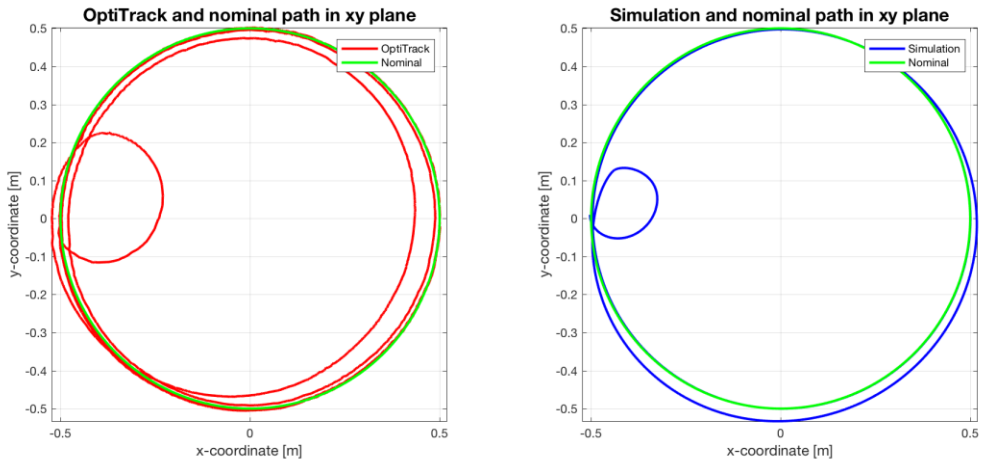

Figure 68. Actual and simulated path for run 5 (-0.5, 0, $-\pi/2$), radius 0.5 m and velocity 0.35 m/s.

Analyzing these plots, it is clear that the actual and the simulated response of the systems are in agreement. For runs 2 and 3 in particular, the paths are extremely similar. For run 4, the size of the initial curve is quite larger in the actual plot when compared to the simulated path, but the overall shape is very similar. As usual, the plots for the system inputs are also produced and shown here in Figs. 69, 70 and 71.

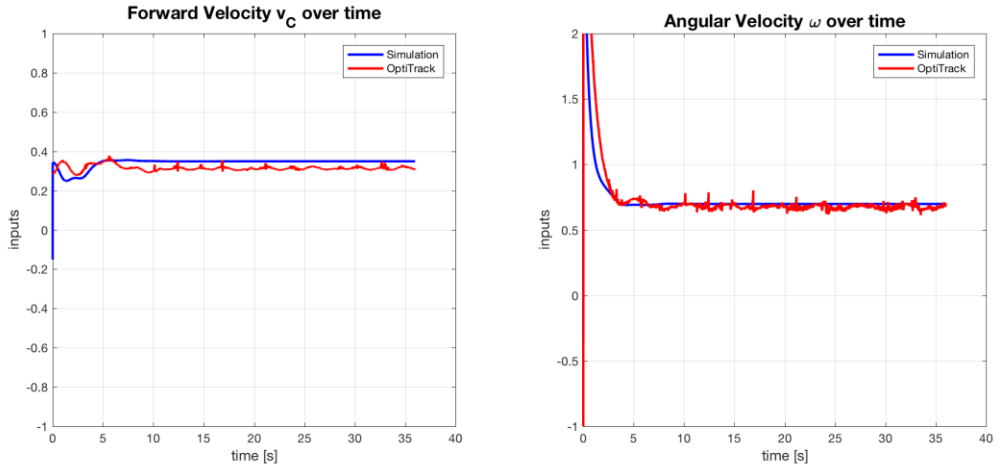

Figure 69. Actual and simulated system inputs for run 3 (0, 0, 0), radius 0.5 m and velocity 0.35 m/s.

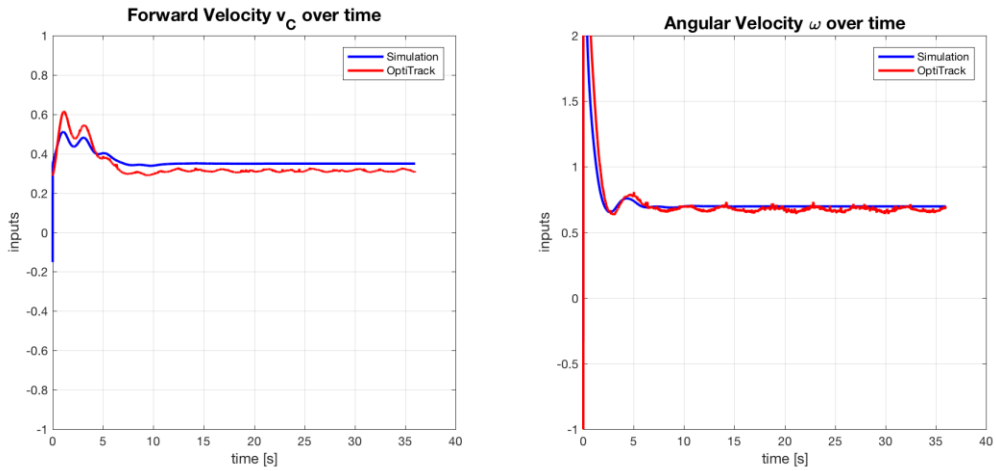

Figure 70. Actual and simulated system inputs for run 4 (0.5, 0, 0), radius 0.5 m and velocity 0.35 m/s.

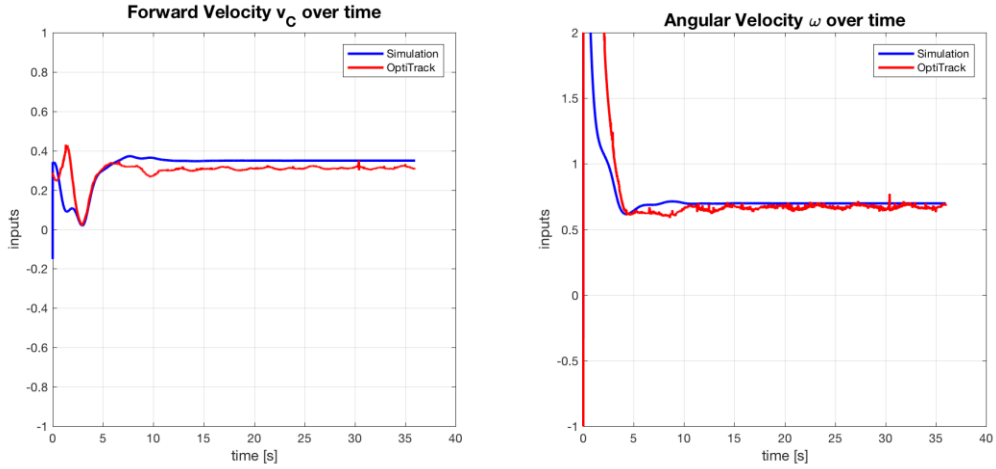

Figure 71. Actual and simulated system inputs for run 5 $(-0.5, 0, -\pi/2)$, radius 0.5 m and velocity 0.35 m/s.

The actual curves appear to be quite in agreement with the simulation curves, except for the previously discussed difference in the final steady-state value caused by the adjustment gains. Note that the runtime was reduced from 54 s to 36 s, simply because it was noted that convergence was reached much sooner (within approximately 10 s) and that running the experiment for more laps did not produce any changes after the steady-state was reached. To show that these inputs indeed steer the system to the nominal path successfully, the tracking error plots for runs 3, 4 and 5 are shown in Figs. 72, 73 and 74.

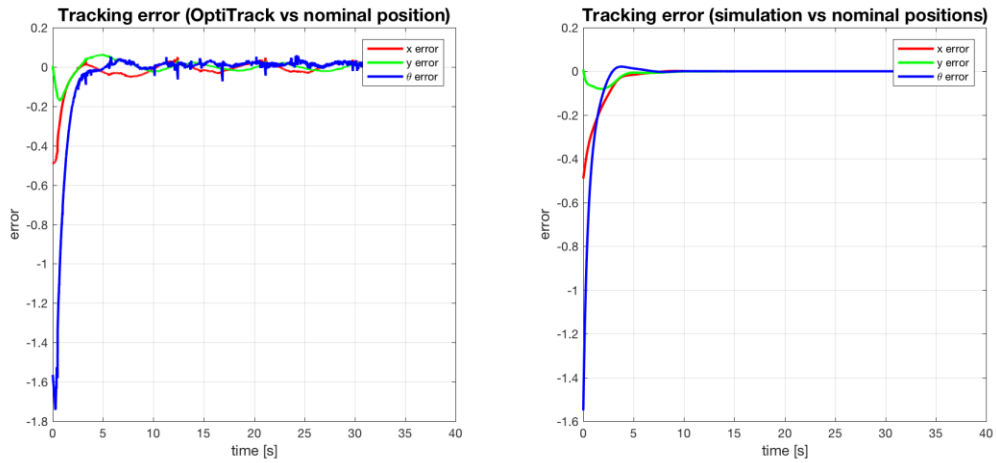

Figure 72. Actual and simulated tracking error for run 3 (0, 0, 0), radius 0.5 m and velocity 0.35 m/s.

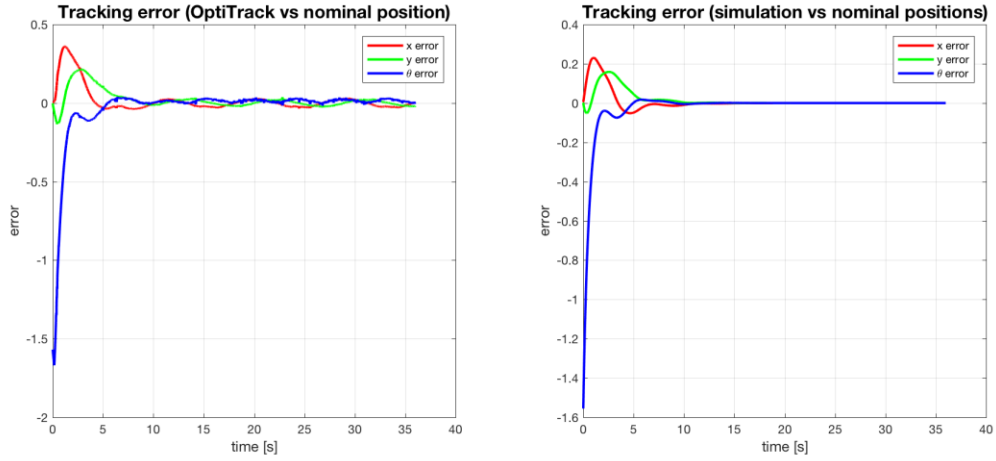

Figure 73. Actual and simulated tracking error for run 4 (0.5, 0, 0), radius 0.5 m and velocity 0.35 m/s.

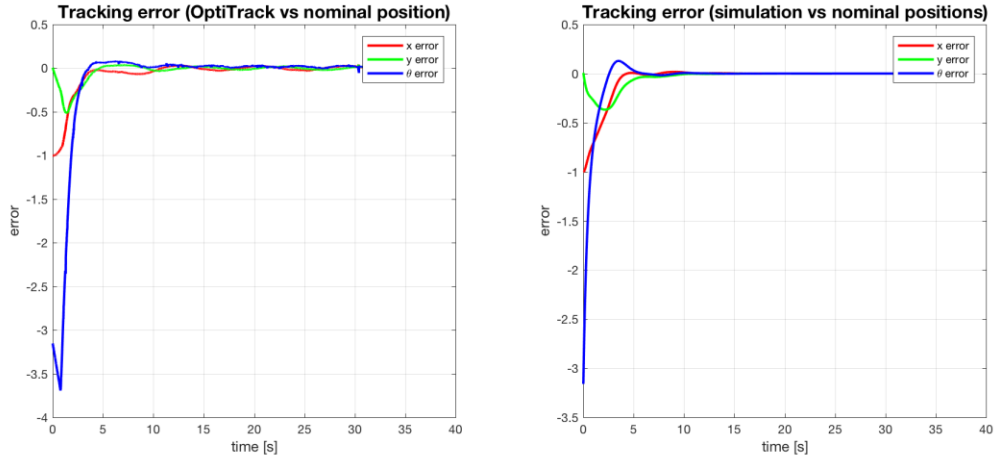

Figure 74. Actual and simulated tracking error for run 5 (−0.5, 0, −π/2), radius 0.5 m and velocity 0.35 m/s.

The tracking error plots show that all errors converge to zero in less than 10 s, despite the presence of oscillations all throughout the actual error plots. The shapes of the curves appear to be very similar, sometimes with occasional delays in the actual values compared to the simulation. Overall, the experiments for the 0.5 m radius have shown to be successful, despite the small differences between the actual and the simulated responses.

Nominal Path Radius: 0.2 m

Next, the smaller radius of 0.2 m is selected. The tangential velocity is maintained at 0.35 m/s, which corresponds to a period of just 3.6 s. Because of the short duration of the lap time and because of the delays that the QBot experiences before beginning its motion in the lab, testing with these parameters was particularly challenging. For this reason, the analysis focuses on just one case (previously seen in the simulations section), in which the robot begins from $(0.24, 0, \pi/2)$. The actual response of the system, as recorded by the OptiTrack camera system, is shown in Fig. 75 and compared to the simulated response.

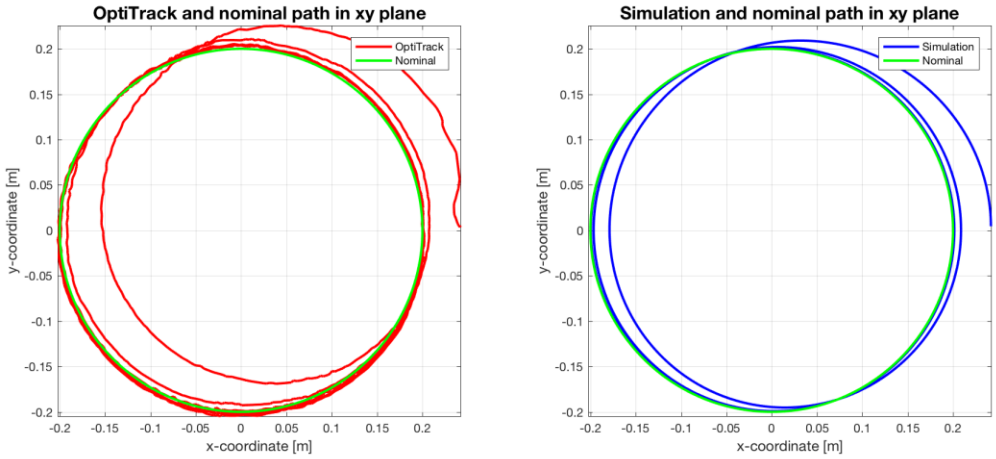

Figure 75. Actual and simulated path for run 1 $(0.24, 0, \pi/2)$, radius 0.2 m and velocity 0.35 m/s.

When compared to the 0.5 m experiments, it is immediately noticeable that actual position plot appears very jagged. The motion of the QBot in the lab is actually no different from the previous case, but its path appears more irregular simply because the scale of the plot is smaller. The same irregularities would become visible in the 0.5 m case if the plot was enlarged.

Another noticeable feature of the plots is their similarity. In both cases, the robot basically completes one and a half full circular paths that are off with respect to the nominal path, before merging smoothly onto the correct trajectory. This characteristic is evident in the tracking error plot, shown in Fig. 76.

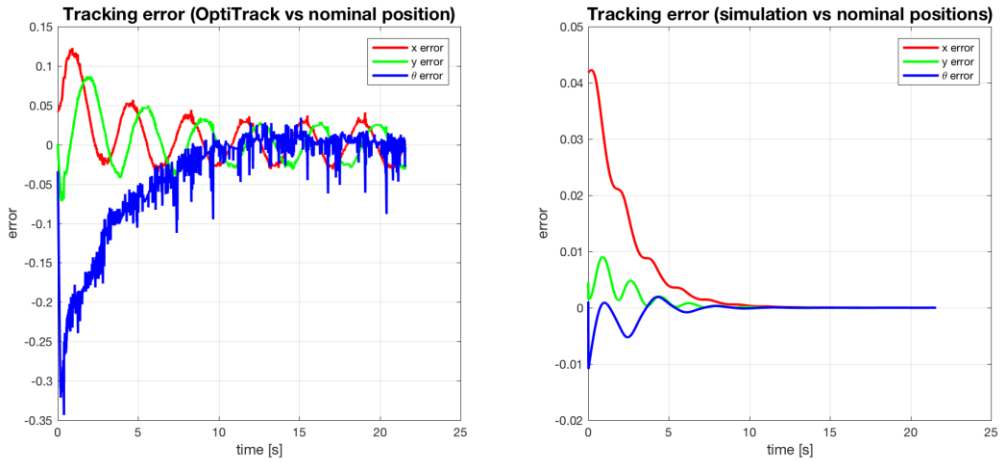

Figure 76. Actual and simulated tracking error for run 1 $(0.24, 0, \pi/2)$, radius 0.2 m and velocity 0.35 m/s.

From these plots, it appears that the robot indeed takes approximately 5 s to drastically reduce the tracking errors, followed by another 5 s to completely bring the errors to zero. Again, the large oscillations in the actual tracking error plot, especially in the θ error curve, are caused primarily by irregularities in the ground. The LQR compensates for these errors with the inputs shown in Fig. 77.

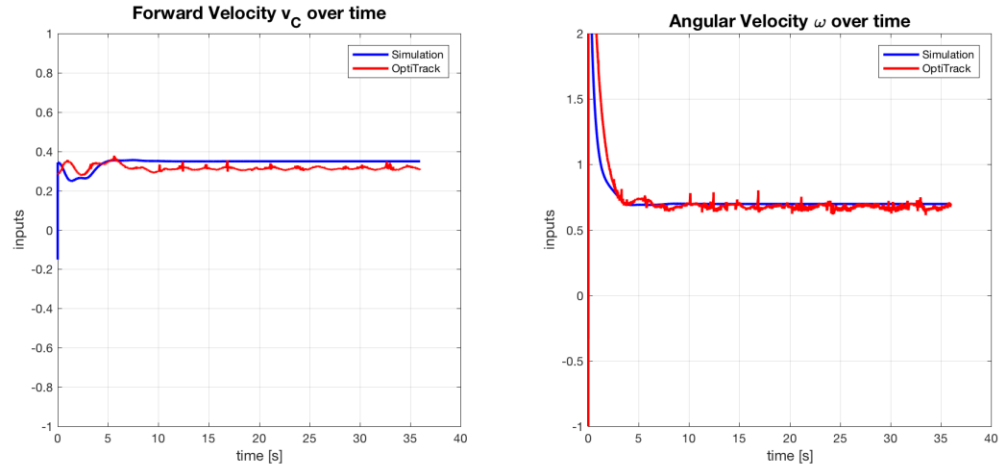

Figure 77. Actual and simulated system inputs for run 1 $(0.24, 0, \pi/2)$, radius 0.2 m and velocity 0.35 m/s.

Overall, the simulated and the actual input plots are very similar. The difference in final steady-state value is again present and due to the same reasons already explained. The oscillations in the actual input curves coincide with the irregularities in the ground and the peaks in input are the consequence to the peaks in tracking errors.

Nominal Path Radius: 1.0 m

The next experimental runs are performed on the large nominal radius of 1.0 m. The velocity is kept the same, at 0.35 m/s. At this speed, the robot takes 18 s to complete one full lap and 72 s to complete the 4 laps that constitute one run. The first two runs focus on the usual initial conditions that are close to the nominal starting state, as seen in the simulations section. The paths produced by the QBot as tracked by the OptiTrack system are shown in Figs. 78 and 79. The simulated plots are also shown on the right for comparison.

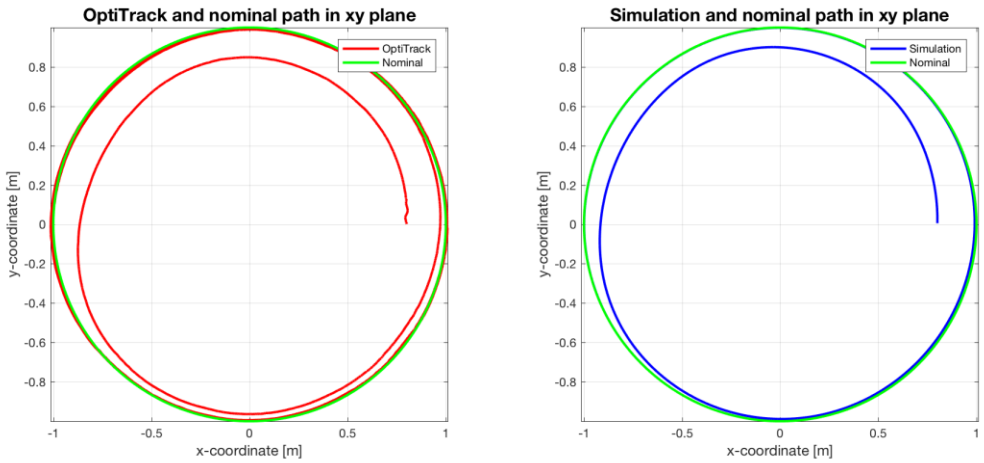

Figure 78. Actual and simulated path for run 1 $(0.8, 0, \pi/2)$, radius 1 m and velocity 0.35 m/s.

These plots show very interesting characteristics. First of all, the actual paths produced by the QBot are in general very similar to the simulated ones. In both cases, the robot smoothly adjusts its trajectory to merge onto the nominal trajectory. The initial oscillations in the actual plots are still present, as seen and discussed previously. For the first run, there is a noticeable different in that the simulated robot merges onto the nominal path in less than one lap, while the actual robot takes a little over one lap. In fact, the actual robot seems to initially follow the same path as the simulated robot, but then begins to coast along the nominal path for quite a while. This difference can be due to the fact that the tracking

error at that point is small enough that, together with the irregularities in the ground, the controller is not producing a great enough correcting input and instead lets the robot coast using the nominal inputs. This appears evident in the input plots for the first run, as shown in Fig. 80.

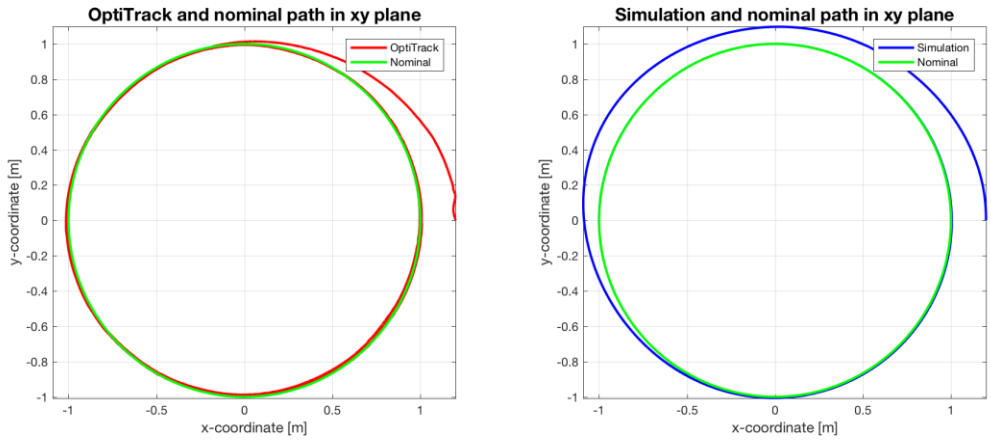

Figure 79. Actual and simulated path for run 2 $(1.2, 0, \pi/2)$, radius 1 m and velocity 0.35 m/s.

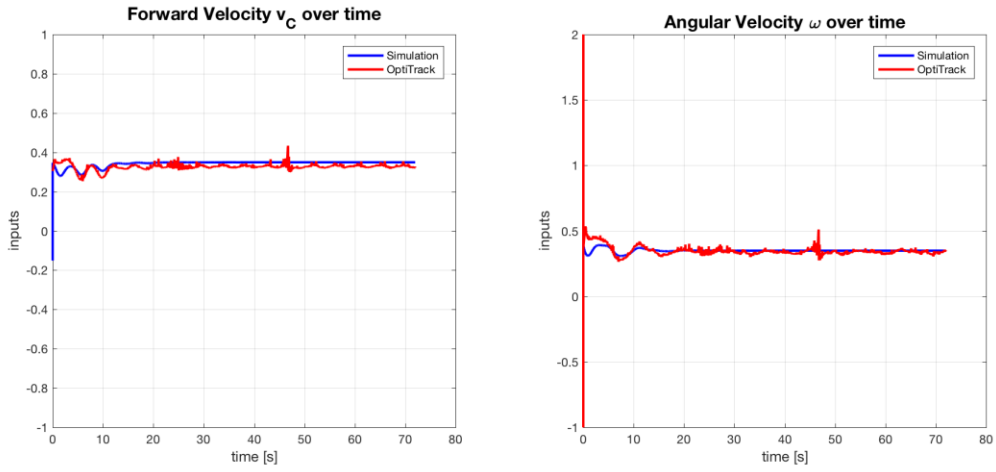

Figure 80. Actual and simulated system inputs for run 1 $(0.8, 0, \pi/2)$, radius 1 m and velocity 0.35 m/s.

From this plot, it is clear that the inputs reach their final constant values at just over 10 s, when the robot still has not completed a full lap and still has actually not reached the nominal path. The oscillations around the 18 s mark coincide with the moment the robot stops coasting along the desired trajectory and actually merges onto it. Note the peak in

actual inputs at around 46 s. This was cause by the tracking system temporarily losing full sight of all reflective markers of the robot. This is an example of another possible source of error for the experiments and will later on be discussed more in detail. Overall, the actual and the simulated curves appear very similar, except for the usual difference in the steady-state value.

For the second run, the opposite scenario is unveiled. The actual QBot, this time, merges onto the nominal path in just one quarter of a full lap, with a very smooth and quick curve in its path. The simulated path, on the other hand, shows the robot merging after three quarters of the first lap. This behavior is evident in the system inputs plots, shown in Fig. 81 as well.

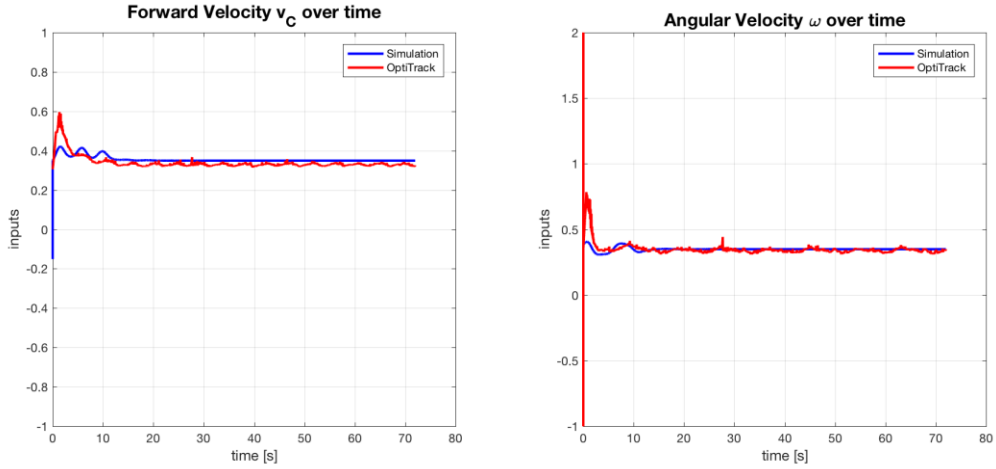

Figure 81. Actual and simulated system inputs for run 2 $(1.2, 0, \pi/2)$, radius 1 m and velocity 0.35 m/s.

In the red curves, peaks in both forward and angular velocity curves are immediately evident, happening before reaching 5 s of run time. While these peaks are present in the simulated curves as well, they are noticeable larger in the actual input curves. These inputs are clearly what causes the quick initial response of the QBot and the merging onto the nominal trajectory in less than 5 s. The rest of the simulated and actual curves are quite in agreement, with the usual differences already described. To better understand the responses of the system and the input plots for both runs, the tracking error plots shown in Figs. 82 and 83 can also be analyzed.

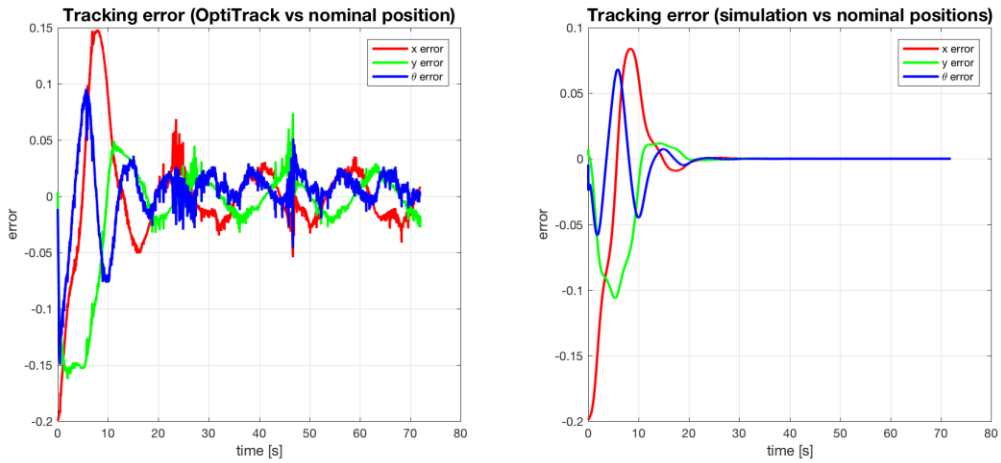

Figure 82. Actual and simulated tracking error for run 1 $(0.8, 0, \pi/2)$, radius 1 m and velocity 0.35 m/s.

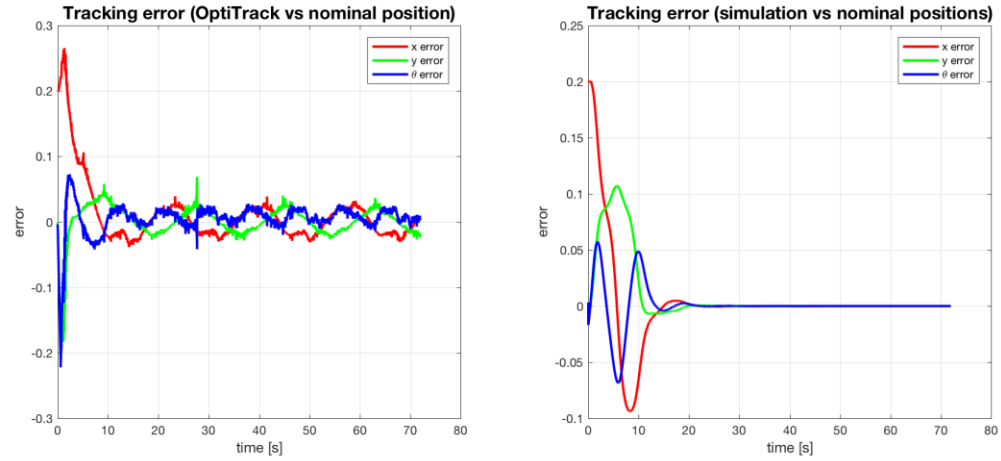

Figure 83. Actual and simulated tracking error for run 2 $(1.2, 0, \pi/2)$, radius 1 m and velocity 0.35 m/s.

For run 1, it is immediately noticeable how the peak in actual tracking errors for all three curves are much higher and slightly delayed when compared to the simulated errors. This is caused primarily by the issue involving the delay in starting the QBot's motion, as discussed previously. The actual robot takes about 25 s, compared to just 20 s of the simulated robot, to remove all errors. For run 2, a similar scenario seems to happen right at the beginning, but the robot is able to quickly correct for the growing errors and ends up removing the errors faster than the simulated response (in about 10 s compared to 18 s).

For the third run, a different arbitrary initial condition is chosen, as was done for the 0.5 m case. Here, the initial condition $(0.5, -0.35, 0.5)$ is used. The path produced by the QBot, compared to the simulated response from the same initial state, is shown in Fig. 84.

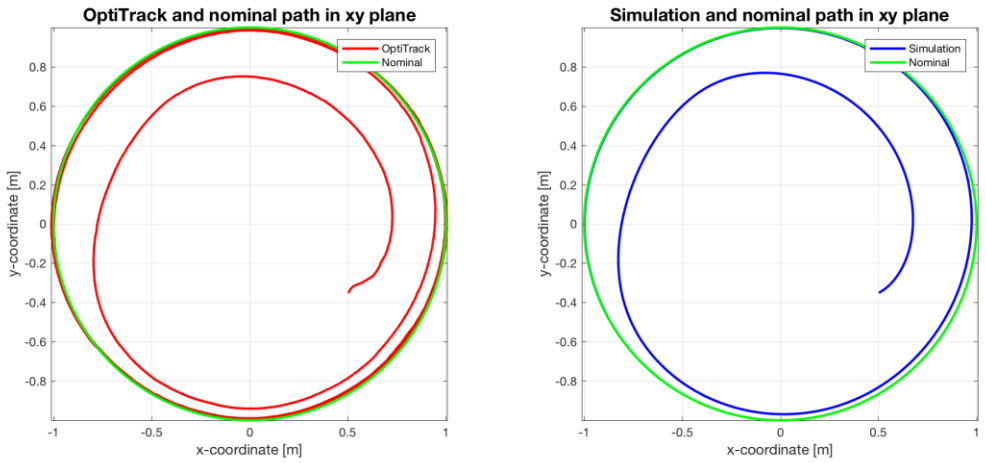

Figure 84. Actual and simulated path for run 3 $(0.5, -0.35, 0.5)$, radius 1 m and velocity 0.35 m/s.

The response produced by the robot is similar to what was shown for run 1. The actual and the simulated paths are very similar, with the main differences being (1) the usual oscillations right at the beginning of the actual robot's path and (2) the longer coasting along the nominal trajectory that the actual robot experiences. The tracking errors and the system inputs are shown in Figs. 85 and 86 respectively.

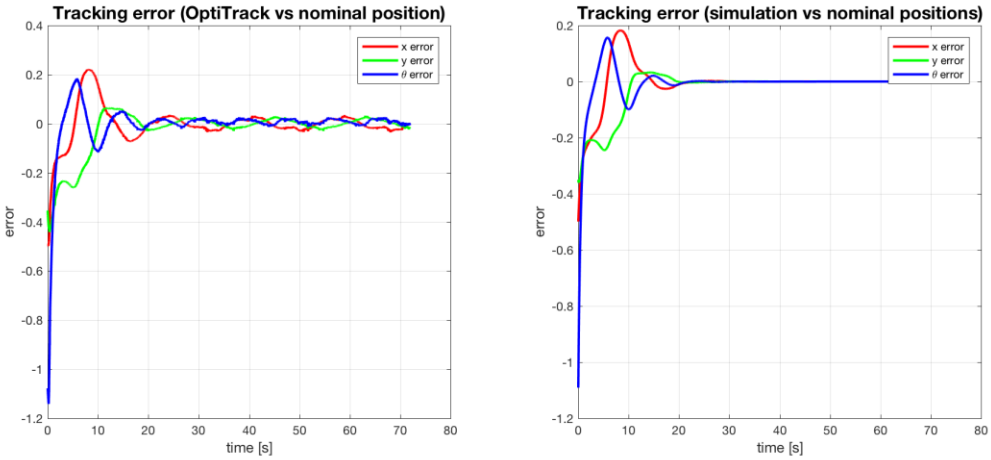

Figure 85. Actual and simulated tracking error for run 3 $(0.5, -0.35, 0.5)$, radius 1 m and velocity 0.35 m/s.

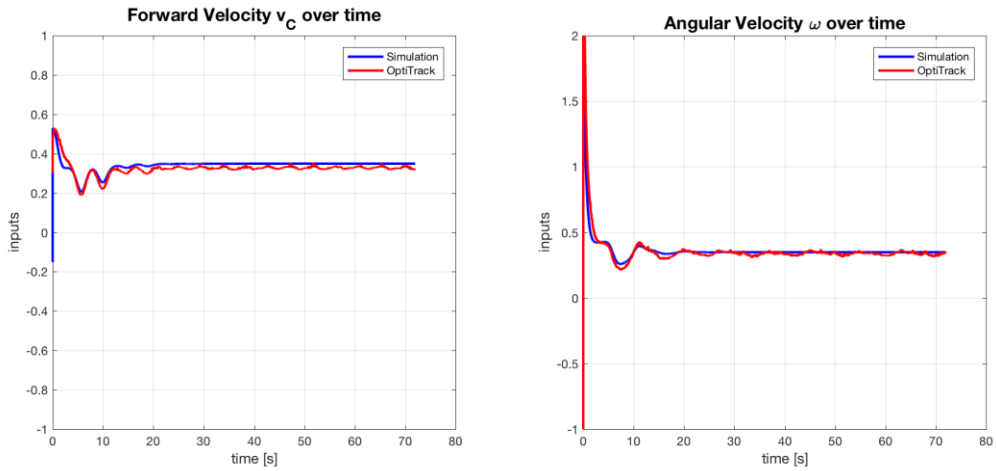

Figure 86. Actual and simulated system inputs for run 3 $(0.5, -0.35, 0.5)$, radius 1 m and velocity 0.35 m/s.

The simulated and actual tracking errors appear very similar, with slightly larger values for the actual errors. Error convergence is achieved in about 20 s in the simulation and in a slightly longer amount of time for the actual robot. This corresponds to the additional coasting of the actual robot's path. The simulated and actual system inputs also appear very close to each other. In both cases, the oscillations in the actual curves are as expected (due to ground irregularities) and the steady-state value is, once again, slightly lower for the actual inputs. Overall, the QBot performed very closely to the simulations, removing all errors with a smooth motion and merging onto the desired trajectory.

Nominal Path Radius: 1.25 m

Finally, the extreme case with a very large radius, 1.25 m, is analyzed. Because the size of the carpeted area that the lab is equipped with is barely larger than this radius, this experiment was particularly challenging. For this reason, only the case where the robot starts from within the nominal trajectory is considered (starting from outside the nominal path would mean starting from outside the carpeted area, where the OptiTrack camera system fails to accurately measure the robot's position and orientation). The starting state is $(1, 0, \pi/2)$ and the forward velocity is kept at 0.35 m/s. At this speed, the robot takes 22 s for one full lap and 88 s to complete the 4 laps that constitute one run. The actual and the simulated paths are shown in Fig. 87.

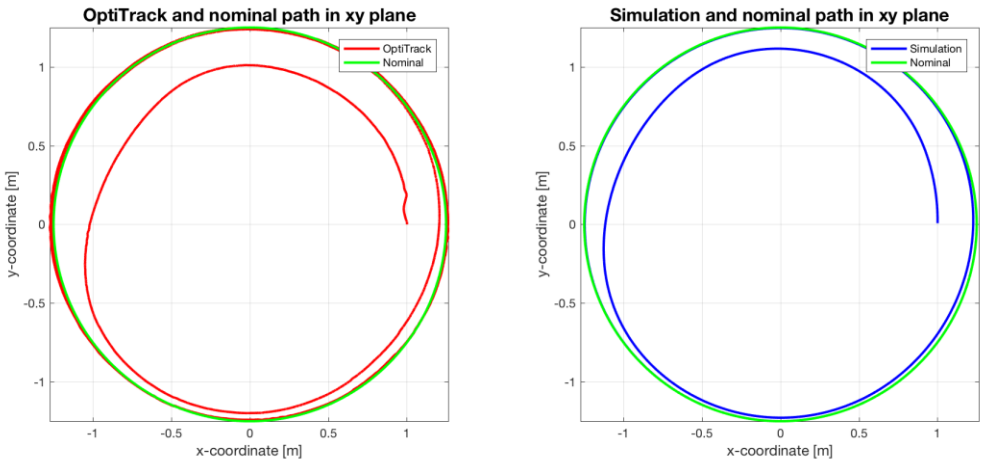

Figure 87. Actual and simulated path for run 1 $(1, 0, \pi/2)$, radius 1.25 m and velocity 0.35 m/s.

The robot once again successfully fixes its position and orientation, and merges onto the nominal trajectory with a smooth motion. Comparing the two plots, it is immediately evident that the simulated response is faster and more curvilinear. The actual QBot, on the other hand, experiences the usual oscillation right at the beginning of its motion. The actual robot also seems to initially follow the same path as the simulated one, but then, towards the top, veers off further away from the nominal trajectory. It then coasts along it for a while and merges onto it after completing one full circle. By looking closer at the simulated plot, it is interesting to note that the path actually undergoes a similar motion, getting very close to the nominal path at the bottom, but then steering more towards the inside before completing the full lap. In summary, the overall shape of the motion of the actual robot seems to be very similar to the simulation, but simply with more pronounced tracking errors. This can be easily visualized in the error plots shown in Fig. 88.

Note the different scales of the y-axis in the two plots. Clearly, the actual robot experiences greater errors than in the simulation. In both cases, the controller takes about 35 s to completely remove the error. Again, in the experiment the error is not actually constant at zero but oscillates around it because of the ground imperfections and other factors already discussed. The controller achieves this by generating the system inputs shown in Fig. 89.

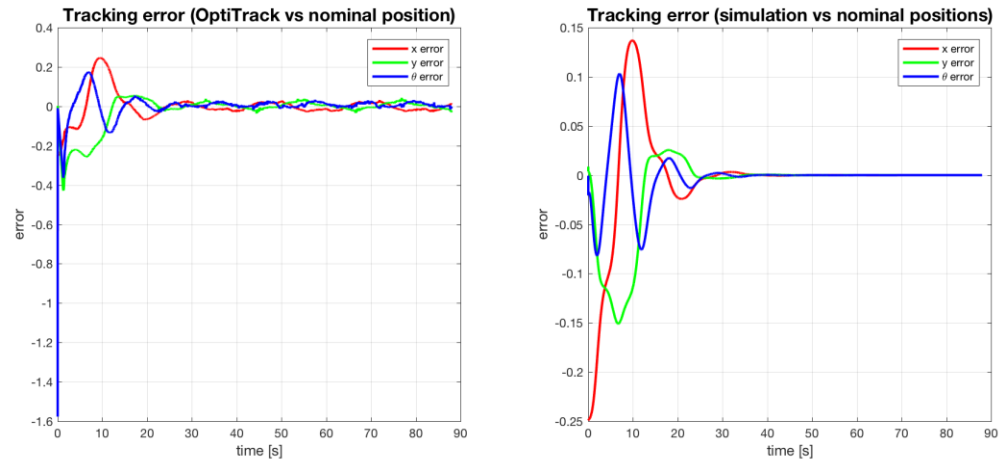

Figure 88. Actual and simulated tracking error for run 1 $(1, 0, \pi/2)$, radius 1.25 m and velocity 0.35 m/s.

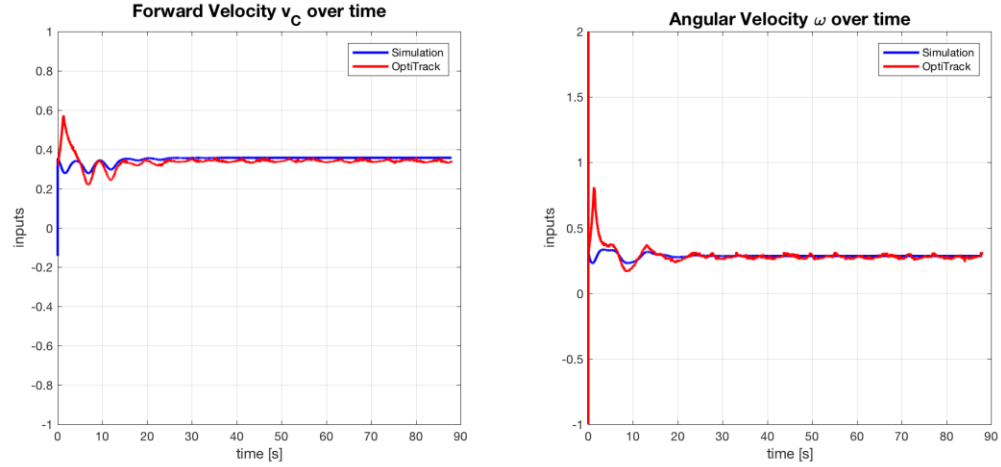

Figure 89. Actual and simulated system inputs for run 1 $(1, 0, \pi/2)$, radius 1.25 m and velocity 0.35 m/s.

The only major difference between actual and simulated results in the two plots is the initial peak in actual system inputs, caused by the oscillations at the start of the QBot's motion. The other difference is, once again, the slight discrepancy between the steady-state values. For both the actual robot's inputs, the final value is lower than in the simulation because of the difference between theoretical and practical already discussed. Comparing the results, the actual and the simulated curves are overall quite in agreement.

DISCUSSION OF RESULTS

The results presented in this chapter were obtained by running the MATLAB code and the Simulink models that control the QBot 2 using the LQR and measure its location using the OptiTrack camera system. The experimental data was examined in contrast with the nominal path and then compared with the simulated system response. The initial state of the simulation was matched exactly with the starting conditions of the actual robot in the lab, in order to replicate the same scenario with precision.

What emerged from the analysis is that the results are overall successful. The system was correctly set up and the communication between the host computer and the robot's hardware was established. With the LQR controller active, the QBot was able to navigate over the carpeted area and smoothly merge onto the nominal path. That is, as long as the initial position and orientation were not too far from the nominal starting state. This is, as discussed in the previous chapter, one of the limitations of a controller that is based on a locally linearized model. In the experiments, the QBot tended to reach the nominal path in approximately the same time as one period, i.e. just as it completes its first lap. The paths that the robot traced out in the lab were very close to what the simulations predicted, indicating that the theoretical basis produced the expected results when applied to the real model. In some cases, the difference between the actual and the simulated routes appeared to be larger, but the overall shape of the paths maintained a good similarity.

The merging of the robot onto the nominal path coincided with the elimination of the tracking error, i.e. the difference between the actual and the nominal position and orientation. In the simulations, once the errors in the x, y and θ coordinates reached zero, the plotted error curves became perfectly flat. This is the ideal scenario, where, with no outside disturbances, the errors become constant once they all become zero. In the experiments, the tracking errors followed very similar patterns to what the simulations predicted during the first time period. The actual values occasionally differed more noticeably, but the shape of the plotted curves remained alike. However, once the QBot stabilized on the nominal trajectory, the tracking errors did not become constant as in the simulations. This is because of the presence of a variety of external disturbances that prevented the errors from staying fixed at zero. The main reason for this phenomenon is the presence of irregularities on the ground. The carpeted area in the lab is made of several square puzzle-like pieces, each with

slightly different thicknesses. When the robot travels from one piece onto another, it is effectively going up or down a step, which causes it to rock. Because the IR-reflective markers are mounted on the top of the robot, the swiveling detected by the OptiTrack cameras is amplified. The periodicity of the oscillations and jumps in the error tracking plots coincide precisely with the moment the robot is moving across two carpet pieces. Another important factor that causes the error to suddenly vary is the fact that, in unpredictable occurrences, the OptiTrack camera system could momentarily lose tracking of the reflective markers. This was primarily due to interference from outside natural light peeking through the window covers and, more rarely, to the fact that, at very particular locations and orientation, some of the markers may have been hidden to a large enough number of cameras because of features of the robot blocking the view. Vibrations of the cameras, caused by the air-conditioning system or by foot-steps on the above floor, may also have caused minor fluctuations in the error. The important aspect to remember is that all tracking error oscillations fluctuate around zero, indicating that in absence of external disturbances, the error should have indeed become constant.

To achieve the suppression of the three tracking errors, the controller sends two input signals, one for the forward velocity and one for the angular velocity, to the plant. The input to the system depends on the error at the current time t, so the plots for the simulated and the actual inputs are not precisely the same. First, the initial part of the input plots are discussed. As the robot first begins its motion, before reaching the nominal trajectory, the actual and the simulated inputs appear overall very similar. Occasionally, the range of values might be different or the peaks shifted, but the overall shape of the curves is comparable. The differences are primarily caused by the issue with the actual robot having a delay from the moment the experiment is started (i.e. the clock starts ticking) to the moment the robot actually begins its motion. This delay was reduced thanks to the precautions described at the beginning of this chapter, but it was never possible to completely remove it. To achieve good results, the only solution was to repeat the experiment several times until a run with a short delay was performed. Nonetheless, this delay is the reason for the peaks in the actual inputs and for the oscillations at the beginning of the robot's motion. Next, the second part of the input plots are analyzed. This part is concerned with when the robot actually reaches the nominal trajectory and consequently removes all tracking errors. In the simulations, this part

of the plot is constant. Once the robot is on the correct trajectory, in absence of external disturbances, all that is needed to keep the robot on its path are the nominal inputs. In the experiments, however, two issues caused slightly different results. The first issue is the same discussed for the differences between the simulated and the actual paths: disturbances such as ground irregularities cause continuous and often periodic errors that the controller needs to compensate for. The second issue is concerned with the final values that the actual forward velocity inputs reach. These values are not equal to the nominal values but are always slightly or sometimes noticeably lower than them. This can be explained with the same reason that lead to the introduction of the corrective gains. Tests on the open-loop system revealed that the actual forward velocity input required to make the QBot travel around a circle with the nominal path was always lower than what was calculated numerically. For this reason, the final value around which the actual forward velocity oscillates is always lower than the nominal one. Despite the small differences between the simulated and the actual inputs, the response of the system can be considered successful, as the tracking error is brought down to zero and the discrepancy between the simulated and the actual paths are negligible.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The goal of this research was to synthesize optimal controllers to control differential-drive robots and to experimentally validate the theory on a Quanser QBot 2. We followed the LQR design approach using the linearized model of the QBot robot around a time-varying trajectory. We addressed the controllability analysis of the model and utilized a neural network approach to implement the time-varying optimal state feedback controller. The numerical and experimental results were in agreement and illustrated the power of the approach.

MAIN CHALLENGES AND LIMITATIONS

The challenges encountered during the development of this thesis were numerous. Most have been solved, after extensive research, and trial and error. A smaller number of issues, not solvable using the techniques used in this thesis, have been examined and identified as system limitations.

One of the earliest difficulties encountered was turning the theoretical knowledge about the Linear Quadratic Regulator into actual code and Simulink model, in a fully developed program that could both simulate the robot's response using numerical computations, as well as interfacing with Quanser's software to run the code on the actual QBot 2. The latter was particularly challenging, as the documentation and resources for Quanser's Simulink blocks was fairly limited. The rarity of this device was also noticeable in the lack of online discussions or tutorials about its usage and applications. This issue was solved by combining the pieces of available documentation with trial and error to eventually make the MATLAB code and Simulink models run successfully.

Another great challenge emerged during the experimental phase. Despite successful setup and calibrations of the OptiTrack camera system, the system would frequently lose

tracking of the QBot robot during experiments. This was found to be caused mainly by interferences in the infrared light spectrum (used by the tracking cameras). Because the windows in the Robotics and Controls Laboratory cannot be fully shielded, during sunny days some of the outside light would penetrate into the room and interfere with the correct identification of the reflective markers mounted on the QBot robots. This issue was solved by limiting light infiltration as much as possible and, mainly, by performing the experiments during evening or night time (or on particularly cloudy days). Loss of tracking was also occasionally observed when the robot was facing certain directions. The cause of this issue was that one or more of the reflective markers would be covered by parts of the robot itself, resulting in some of the ceiling-mounted cameras not detecting all the markers. This phenomenon, while usually lasting less than a second, was enough to cause jerky motion of the robot. The issue was fixed by relocating the markers in places where they would always be visible to the cameras.

Similarly, it was observed that the OptiTrack system would experience interruptions in the tracking when the robot reached certain locations. The issue would arise, in particular, when the robot was travelling close to the edges of the carpeted area of the floor. To fix this problem, an entire removal and relocation of some of the ceiling cameras would have been necessary. Because of the complexity of this operation, this issue is simply considered a limitation of the current setup. To overcome it, it was necessary to avoid having the robot drive too close to the edges throughout all experiments, thus limiting the radius of the nominal trajectory or avoiding starting locations too far from the center.

Another major issue was encountered during the experiments. After the LQR has guided the QBot onto its nominal trajectory, effectively reducing the tracking error to zero, the robot's inputs should, in theory, stabilize around some constant value. In practice, however, the inputs to the robot still contained irregularities, which, after further analysis, were found to be cyclical. Comparing these observations with the actual motion of the QBot, it was concluded that the oscillations coincided with terrain irregularities that the robot was cyclically driving over throughout each circular lap. Specifically, differences in thickness between the pieces composing the carpeted floor acted like steps that the robot had to climb over or descend down, causing abrupt changes in velocity and direction. Solving this issue completely would have required using a flatter type of floor, with a material that would still

guarantee absence of reflections for the infrared cameras. Again, this issue was taken as a limitation of the current setup and taken into consideration when analyzing the experimental results.

A noticeable issue often appeared right before starting the experiment. After building and running the Simulink model "QBot2_Optitrack.slx", the computer clock starts ticking right away. For the QBot to actually start receiving the signals however, it can sometimes take up to 3 seconds. By the time the robot begins its motion, the nominal location and orientation, which start "moving" as soon as the clock start ticking, have already shifted to a state different from the initial condition. The robot then finds itself being behind and its efforts to catch up to the nominal state cause it to have a response that differs largely from the simulated system response. This issue was only partially resolved. It was found that waiting at least 5-6 seconds before running the model after connecting to the target, as suggested by the manufacturer, indeed helps reduce the chances that the robot takes more than 2 seconds to start moving. A full solution to the problem was not identified. Thus, it was taken as a limitation and many of the experiments had to be run several times until the robot finally responded in a relatively short amount of time (usually, less than 1 second was acceptable).

Another challenging task was encountered when testing out new initial conditions. It was known from the beginning that the controller would not be able to be effective in any arbitrary position in space, due to the fact that it is based on a linearized model of the system. However, identifying the limits within which the robot would correctly work could not be determined within the theoretical derivations. This had to be done manually by repeating both simulations and experiments to understand in what position and orientation in space the robot would become unstable. Identifying an approximate area that would guarantee the successful outcome of the experiment was the solution to this issue.

The last major challenge was identifying and solving the discrepancy between the theoretical and the practical realms. When the actual QBot was initially tested with the LQR, the response seemed stable and correct. Upon closer examination of the path, as measured by the OptiTrack system, it was found that the robot always stabilized around a circular path with a radius larger than the desired one. Extensive experimentation was required to understand that the values of nominal inputs (for forward and angular velocities) that work

for the simulations are actually too high in reality to produce the same outcome with the actual robot. The solution to this issue was the introduction of the corrective input gains, which reduce the nominal forward velocity input to the value that allows the QBot to perfectly drive on the nominal trajectory. The values of the corrective gains depend on the radius of the nominal path. They were determined experimentally by repeatedly running the experiment with open-loop control and varying the gains until the robot could drive in a circle with the same size as the nominal path.

FUTURE WORK

The theory and the MATLAB program developed for this thesis represent a solid foundation for applying time-varying LQR controllers to differential drive robots. Since only one path shape was considered, applying the controller to a greater variety of trajectory shapes is the next natural step to extend the system's capabilities. If the map of the environment is already known, a path planning algorithm could be implemented in order to determine the best trajectory to reach a specific goal. The algorithm could break the trajectory up into a number of simpler curves, around which the system can be linearized, and the program can then be modified so that the robot can travel along each segment continuously. These operations could be performed offline before deploying the code on the robot, or in real-time to constantly adjust throughout the robot's voyage.

A better implementation of this concept can be elaborated by making use of the information from available sensors. The sensory data could be integrated with the control of the robot by using simultaneous localization and mapping (SLAM) methods. For instance, the live stream of images captured by the RGB camera (from the Kinect sensor mounted on the QBot 2) can be used to detect objects, map the environment and aid in the navigation. Computer vision techniques, such as filtering in combination with Canny edge detection [50] or Hough transform for region detection [51], can be used to identify relevant shapes or to read QR codes. The depth camera of the Kinect sensor can also be used to measure the distance of objects and to obtain a 3D map of the surrounding environment. This way, the map of the environment does not need to be known beforehand but can instead be constructed as the robot moves through space.

If more autonomy is desired, the capabilities of the robot can be extended by using more advanced artificial intelligence techniques. If sufficient amounts of data about how the robot should behave are available, a convolutional neural network can be trained to learn the data and subsequently mimic the behavior, even in not-previously-seen scenarios. This can be useful, for example, if the robot is required to drive along a "road", with clear markings delimiting the lane and with traffic signs providing information on how to safely navigate. For this purpose, the neural network can be trained with videos showing correct ways to navigate the environment, paired with the corresponding inputs to the system. If no training data is available, reinforcement learning techniques could be implemented in order for the robot to learn from its own mistakes. Altogether, the information provided by the numerous sensors mounted on ground vehicles such as the Quanser QBot 2 can be used together with the time-varying LQR controller developed in this thesis to make the robot more intelligent and autonomous.

REFERENCES

[1] Siciliano, B., L. Sciavicco, L. Villani, and G. Oriolo. "Mobile Robots." In *Robotics: Modelling, Planning and Control*, 469-521. London: Springer, 2009.

[2] Bonkenburg, T. "Robotics in Logistics - A DPDHL Perspective on Implications and Use Cases for the Logistics Industry." White Paper, DHL, Troisdorf, Germany, March 2016. http://www.dhl.com/content/dam/downloads/g0/about_us/logistics_insights/dhl_trendreport_robotics.pdf.

[3] Hennessey, M. P. "OTTO 1500 Self-Driving Vehicle (SDV) for Heavy-Load Material Transport in Warehouses, Distribution Centers and Factories." Wikimedia Commons. Last updated August 9, 2016. https://commons.wikimedia.org/wiki/File:OTTO_1500_.jpg.

[4] Scolaro, Christina M. "This Weed-Killing AI Robot Can Tell Crops Apart." *CNBC*, June 4, 2018. https://www.cnbc.com/2018/06/04/weed-killing-ai-robot.html.

[5] Dickerson, Kelly. "NASA's Robot Army of 'Swarmies' Could Explore Other Planets." *Space*, August 25, 2014. https://www.space.com/26935-nasa-robot-swarmies-army-space-exploration.html.

[6] Reuters. "Roomba Vacuum Maker iRobot Has a New Strategy Turning Heads." *Fortune*, July 24, 2017. http://fortune.com/2017/07/24/roomba-vacuum-irobot-smart-home/.

[7] Robinson, Melia. "Tiny Self-Driving Robots Have Started Delivering Food On-Demand in Silicon Valley." *Business Insider*, April 30, 2018. https://www.businessinsider.com/doordash-delivery-robots-starship-technologies-2017-3.

[8] Heater, Brian. "Taking a Ride in MIT's Self-Driving Wheelchair." *TechCrunch*, June 24, 2017. https://techcrunch.com/2017/06/24/taking-a-ride-in-mits-self-driving-wheelchair/.

[9] Kastrenakes, Jacob. "This Small Remote-Controlled Robot Can Tow up to 9,000 Pounds." *The Verge*, March 29, 2018. https://www.theverge.com/circuitbreaker/2018/3/29/17176378/rvr-remote-control-moving-robot-trailer-valet.

[10] Ridden, Paul. "Misty Robotics Rolls Out Accessible, Affordable Personal Robot." *New Atlas*, May 2, 2018. https://newatlas.com/misty-robotics-personal-robot/54463/.

[11] Tsuchiya, K., T. Urakubo, and K. Tsujita. "A Motion Control of a Two-Wheeled Mobile Robot." In *1999 IEEE International Conference on Systems, Man, and Cybernetics*, 690-96. Piscataway, NJ: IEEE, 1999.

[12] Pappas, G. J., and K. J. Kyriakopoulos. "Modeling and Feedback Control of Nonholonomic Mobile Vehicles." In *Proceedings of the 31st IEEE Conference on Decision and Control*, 2680-85. Piscataway, NJ: IEEE, 1992.

[13] Zohaib, Muhammad, Mustafa Pasha, Hafza Bushra, and Jamshed Iqbal. "Addressing Collision Avoidance and Nonholonomic Constraints of a Wheeled Robot: Modeling and Simulation." In *2014 International Conference on Robotics and Emerging Allied Technologies in Engineering (iCREATE)*, 1-6. Piscataway, NJ: IEEE, 2014.

[14] Torres, Edison Orlando Cobos, S. Konduri, and Prabhakar Reddy Pagilla. "Study of Wheel Slip and Traction Forces in Differential Drive Robots and Slip Avoidance Control Strategy." *2014 American Control Conference* 139 (January 2017): 014505-1-6.

[15] Tinh, Nguyen Van, Nguyen T. Lin, Pham T. Cat, Pham Minh Tuan, Mai N. Anh, and Nguyen P. T. Anh. "Modeling and Feedback Linearization Control of a Nonholonomic Wheeled Mobile Robot with Longitudinal, Lateral Slips." Paper presented at the 2016 IEEE International Conference on Automation Science and Engineering (CASE), Fort Worth, TX, August 2016.

[16] Mendes, Ellon P., and Adelardo A. D. Medeiros. "Identification of Quasi-linear Dynamic Model with Dead Zone for Mobile Robot with Differential Drive." In *Proceedings of the 2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*, 132-37. Piscataway, NJ: IEEE, 2010.

[17] Dave, Piyush N., and J. B. Patil. "Modeling and Control of Nonlinear Unmanned Ground All Terrain Vehicle." Paper presented at the 2015 International Conference on Trends in Automation, Communications and Computing Technology (I-TACT-15), Bangalore, India, December 2015.

[18] Mohamed, Haytham, Yehia Z. Elhalwagy, Amr Wassal, and Nevin Darwish. "Modeling and Simulation of Four-Wheel Steering Unmanned Ground Vehicles Using a PID Controller." Paper presented at the 2014 International Conference on Engineering and Technology (ICET), Cairo, Egypt, April 2014.

[19] Rashid, Razif, Irraivan Elamvazuthi, Mumtah Begam, and M. Arrofiq. "Differential Drive Wheeled Mobile Robot (WMR) Control Using Fuzzy Logic Techniques." In *2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*, 51-55. Piscataway, NJ: IEEE, 2010.

[20] Jin, Xiaolong, Zhibao Su, Xijun Zhao, Jianwei Gong, and Yan Jiang. "Design of a Fuzzy-PID Longitudinal Controller for Autonomous Ground Vehicle." In *Proceedings of 2011 IEEE International Conference on Vehicular Electronics and Safety*, 269-73. Piscataway, NJ: IEEE, 2011.

[21] Berkemeier, Matthew, Sostenes Perez, and David Bevly. "On the Suitability of Nonlinear Model Predictive Control for Unmanned Ground Vehicles." Paper presented at the 2014 American Control Conference, Portland, OR, June 2014.

[22] Yakub, Fitri, and Yasuchika Mori. "Model Predictive Control Based on Kautz Function for Autonomous Ground Vehicle Path Following Control Application." In *2014*

Proceedings of the SICE Annual Conference (SICE), 1035-40. Piscataway, NJ: IEEE, 2014.

[23] Kogan, Michael, Peter T. Jardine, and Sidney N. Givigi. "Architecture for Testing Learning-Based Autonomous Vehicle Control Design." Paper presented at the 2018 Annual IEEE International Systems Conference (SysCon), Vancouver, BC, Canada, April 2018.

[24] Hernandez, Wilmar, and Norberto Cañas. "Non-Linear Control of an Autonomous Ground Vehicle." In *37th Annual Conference of the IEEE Industrial Electronics Society*, 2678-83. Piscataway, NJ: IEEE, 2011.

[25] Medina-Garciagiedo, Veronica, and Alexander Leonessa. "Tracking Control of a Nonholonomic Ground Vehicle." In *Proceedings of the 2011 American Control Conference*, 1710-13. Piscataway, NJ: IEEE, 2011.

[26] Donaire, Alejandro, Jose Guadalupe Romero, Tristan Perez, and Romero Ortega. "Smooth Stabilisation of Nonholonomic Robots Subject to Disturbances." In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 4385-90. Piscataway, NJ: IEEE, 2015.

[27] Hemmat, Mohammad Ali Askari, Zhingxiang Liu, and Youmin Zhang. "Real-Time Path Planning and Following for Nonholonomic Unmanned Ground Vehicles." In *2017 International Conference on Advanced Mechatronic Systems (ICAMechS)*, 202-7. Piscataway, NJ: IEEE, 2017.

[28] Mohammed, Ibrahim, Berat A. Erol, Ikram Hussain Mohammed, Patrick J. Benavidez, and Mo Jamshidi. "Improved Route Optimization for Autonomous Ground Vehicle Navigation." In *2017 12th System of Systems Engineering Conference (SoSE)*, 1-6. Piscataway, NJ: IEEE, 2017.

[29] Lakhdari, Abdelhalim, and Nouara Achour. "Probabilistic Roadmaps and Hierarchical Genetic Algorithms for Optimal Motion Planning." In *2014 Science and Information Conference*, 221-25. Piscataway, NJ: IEEE, 2014.

[30] Kaygisiz, B. H., A. M. Erkmen, and I. Erkmen. "GPS/INS Enhancement Using Neural Networks for Autonomous Ground Vehicle Applications." *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)* 4 (December 2003): 3763-68.

[31] Medina, Sergio, Paola Lancheros, Laura Sanabria, Nelson Velasco, and Leonardo Solaque. "Localization and Mapping Approximation for Autonomous Ground Platforms, Implementing SLAM Algorithms." In *2014 III International Congress of Engineering Mechatronics and Automation (CIIMA)*, 1-5. Piscataway, NJ: IEEE, 2014.

[32] Tai, Lei, Giuseppe Paolo, and Ming Liu. "Virtual-to-Real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation." In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 31-36. Piscataway, NJ: IEEE, 2017.

[33] Chiu, Fu-Hsin Steven, and Min-Fan Ricky Lee. "Intelligent Multi-Behavior Control and Coordination for the Autonomous Differential Drive Mobile Robot." In *2013*

International Conference on Fuzzy Theory and Its Applications (iFUZZY), 31-36. Piscataway, NJ: IEEE, 2013.

[34] Kobuki. “iClebo Kobuki.” Yujin Robot. Accessed October 15, 2017. http://kobuki.yujinrobot.com/.

[35] Yujin Robot. “Yujin Robot.” Yujin Robot. Accessed October 15, 2017. http://en.yujinrobot.com/.

[36] Quanser. “QBot 2 for QUARC.” Quanser. Accessed August 26, 2017. https://www.quanser.com/products/qbot-2-quarc/.

[37] Quanser. "System Specifications and Model Parameters." In *QBOT 2 - User Manual*, 6. Markham, ON, Canada: Quanser, 2015.

[38] Monforte, J. C. “Introducing Constraints.” In *Geometric, Control and Numerical Aspects of Nonholonomic Systems*, 43-44. New York: Springer, 2002.

[39] LaValle, Steven M. “A Differential Drive.” In *Planning Algorithms*, 726-29. New York: Cambridge University Press, 2006.

[40] Tzafestas, Spyros G. “Unicycle.” In *Introduction to Mobile Robot Control*, 41-43. London: Elsevier, 2014.

[41] Ogata, Katsuhiko. “Controllability.” In *Modern Control Engineering*, 675-77. Upper Saddle River, NJ: Prentice Hall, 1997.

[42] Lu, Ping. “Solutions of Linear Systems.” In *AE696 Lecture Notes - Optimal Control*, 6-7. Self published, 2018.

[43] Chen, Chi-Tsong. "Solution of Linear Time-Varying (LTV) Equations." In *Linear System Theory and Design*, 135. Oxford: Oxford University Press, 2012.

[44] Chen, Chi-Tsong. “LTV State-Space Equations.” In *Linear System Theory and Design*, 213. Oxford: Oxford University Press, 2012.

[45] OptiTrack. “Flex 3.” Leyard. Accessed November 12, 2017. http://optitrack.com/products/flex-3/.

[46] Lu, Ping. “Linear Quadratic Regulator Problem.” In *AE696 Lecture Notes - Optimal Control*, 146-51. Self published, 2018.

[47] Murray, R. “Lecture 2 - LQR Control.” In *Control and Dynamical Systems (CDS 110b) Lecture Notes*, 1-2. Pasadena: California Instute of Technology, 2006.

[48] Negnevitsky, Michael. “Artificial Neural Networks.” In *Artificial Intelligence - A Guide to Intelligent Systems*, 165-85. Harlow, England: Pearson Education, 2005.

[49] Tarokh, Mahmoud. “Artificial Neural Networks.” In *CS 657 - Intelligent Systems and Control*, 33. Self published, 2018.

[50] Efford, Nick. “The Canny Edge Detector.” In *Digital Image Processing*, 171-75. Harlow, England: Addison-Wesley, 2000.

[51] Tarokh, Mahmoud. “Hough Transform for Region Detection.” In *CS 559 Lecture Notes - Computer Vision*, 12-16. Self published, 2017.

APPENDIX A

CONTROLLABILITY MATLAB CODE

Script name: Qbot_Controllability.m

```
% Quanser QBot 2 CONTROLLABILITY
% Version 2.4
% Code by Vittorio Longi

clear all
clc

syms omega t t_0 t_f

%% State-space matrices

% State matrix
A = [ 0 0 -sin((pi/2)+omega*t);
      0 0 cos((pi/2)+omega*t)
      0 0           0           ];
A = simplify(A);

% Input matrix
B = [ cos((pi/2)+omega*t) 0
      sin((pi/2)+omega*t) 0
                0           1 ];
B = simplify(B);

%% Identity Property A(t) * A_int(t) = A_int(t) * A(t)

A_int = int(A, t_0, t);

if all(all((A*A_int) == 0)) && all(all((A_int*A) == 0))
    display('A matrix identity property: satisfied');
else
    display('A matrix identity property: failed');
end

%% State Transition Matrix

Phi = eye(3,3)+A_int;

%% State Transition matrix double property

% Partial derivative of Phi w.r.t. time
dPhi = diff(Phi,t);

% Property 1
if all(all((A*Phi)==dPhi))
    display('State transition matrix Property 1: satisfied')
else
        display('State transition matrix Property 1: failed')
end

% Property 2
```

```
Phi_0 = [ 1,0,-(sin(omega*t_0)-sin(omega*t_0))/omega;  0,1,(cos(omega*t_0)-
cos(omega*t_0))/omega;  0, 0, 1];

if all(all((Phi_0)==eye(3)))
    display('State transition matrix Property 2: satisfied')
else
        display('State transition matrix Property 2: failed')
end

%% Controllability

% Controllability Gramian
W_integrand = simplify(Phi*B*transpose(B)*transpose(Phi));

W_c = simplify( int(W_integrand, t_0, t_f) );

% Symmetry check
if W_c(2,1)-W_c(1,2) == 0 && W_c(3,1)-W_c(1,3) == 0
    fprintf('W_c is symmetric\n');
end

% Rank check
rank = rank(W_c)

% Controllability conclusion
if rank == 3
    fprintf('The system is controllable\n')
else
    fprintf('The system is NOT controllable\n')
end
```

APPENDIX B

MAIN MATLAB SCRIPT

Script name: QBot_2_4.m

```
% Quanser QBot 2
% LQR-based Control and Simulation
% Version 2.4
% Code by Vittorio Longi

%clear all
close all, clc, tic
display('QBot_2_4 started')

%% 0. CONSTANT PARAMETERS (USER-DEFINED)

% circular path radius      [m]
RADIUS = 0.5;
% period                    [s]
PERIOD = 9;

% total simulation time     [s]
FINAL_TIME = PERIOD*5;
% time step size            [s]
TIME_INCREMENT = 0.01;
% Standard testing: FINAL_TIME = 10, TIME_INCREMENT = 0.01

% initial x-coordinate      [m]
INITIAL_X = 0.6;
% initial y-coordinate      [m]
INITIAL_Y = 0;
% initial theta angle       [rad]
INITIAL_THETA = pi/2;

% Note how changing initial conditions does not change K_lqr
% (K_lqr is independent of initial conditions)

% LQR Design Parameters

% 1. Parameter F (penalizes final state error)
F = eye(3);

% 2. Parameter Q (penalizes state error over time)
Q = eye(3);

% 3. Parameter R (penalizes input effort over time)
R = eye(2);


% SWITCHES:
useSimulink         = true;
loopAnimation       = false; % only if useSimulink = false
useNN               = false;
exportToExcel       = false;
plotFigures         = true;
exportFigures       = false;  % only if plotFigures = true
```

```
%% 1. SYMBOLIC VARS

% Robot coordinates
syms x y theta      % current location & orientation

% Robot velocities
syms v_c        % central forward velocity
syms omega      % angular velocity

syms omega_0    % initial angular velocity
global omega_0


% State vars
syms X X1 X2 X3         % X = [X1; X2; X3] = [x; y; theta]
syms dX dX1 dX2 dX3     % dX = [dX1; dX2; dX3] = [dx; dy; dtheta]

% Output vars
syms Y Y1 Y2 Y3         % Y = [Y1; Y2; Y3] = [x; y; theta]

% Input vars
syms U U1 U2            % U = [U1; U2] = [v_c; omega]


% Other vars
syms r      % radius of circular path
syms t      % time


%% 2. SYSTEM DYNAMICS

% State eqns
dX1 = U1*cos(X3);
dX2 = U1*cos(X3);
dX3 = U2;
dX = [dX1; dX2; dX3];

% Input eqns
U1 = v_c;
U2 = omega;
U = [U1; U2];

% Output eqns
Y1 = x;
Y2 = y;
Y3 = theta;
Y = [Y1; Y2; Y3];


%% 3. NOMINAL SOLUTIONS
% For cicular path with radius r and period T
% where omega_0 = 2*pi/T (for reference, theta = omega_0*t)

X1_nom = r*cos(omega_0*t);
X2_nom = r*sin(omega_0*t);
X3_nom = (pi/2)+omega_0*t;
X_nom = [X1_nom; X2_nom; X3_nom];

dX1_nom = diff(X1_nom,t);
% manually, dX1_nom = U1 * cos(X3); or dX1_nom = U1 * cos(X3+(pi/2));
dX2_nom = diff(X2_nom,t);
% manually, dX2_nom = U1 * sin(X3); or dX2_nom = U1 * sin(X3+(pi/2));
dX3_nom = diff(X3_nom,t);
% manually, dX3_nom = U2;
dX_nom = [dX1_nom; dX2_nom; dX3_nom];

U1_nom = r*omega_0;
U2_nom = omega_0;
U_nom = [U1_nom; U2_nom];
```

```
% A := df/dx(x*, u*)
% State matrix (derived by hand)
A = [ 0 0 -U1_nom*sin((pi/2)+omega_0*t);
    0 0 U1_nom*cos((pi/2)+omega_0*t)
    0 0           0           ];
A = simplify(A);


% Input matrix (derived by hand)
B = [ cos((pi/2)+omega_0*t) 0
    sin((pi/2)+omega_0*t) 0
    0           1 ];
B = simplify(B);


%% 4. PARAMETERS
T = PERIOD; % period (to complete one circle)

omega_0 = (2*pi)/T; % symbolic to value
r = RADIUS;         % symbolic to value

tf = FINAL_TIME;  % final sim time
t_increment = TIME_INCREMENT;    % time accuracy/delay in each time frame
t_array = 0 : t_increment : tf-t_increment;   % array of every time instant


%% 5. LQR STATE FEEDBACK

k_lqr = {};
U_lqr = {};
X_nom_array = {};

U_nom = eval(U_nom);
A = eval(A);

dX = A*X_nom + B*U_nom;

display('Solving Riccati equation...')
tspan_ric = flip(t_array);  % array of every time instant from final to initial
[t_ric,K_inline] = ode45( @(t,dX)riccati_2_4(t,dX,A,B,R,Q), tspan_ric, F);
display('     Finished solving Riccati equation')

% Re-format F from 1x9 (inline) to 3x3 (square)
K = [];
for i = 1:max(size(K_inline))
    K(:,:,i) = reshape(K_inline(i,:),[3 3]);
end

display('Computing gains k_lqr...')
for i = 1:length(t_array)
    %fprintf('LQR looping at time t = %2.4f\n', t);

    t = t_array(i); % get current time from time array

    k_lqr_tmp = inv(R) * transpose(B) * K(:,:,i);
    k_lqr_tmp = eval(k_lqr_tmp);

    % add current k_lqr to array
    k_lqr{i} = k_lqr_tmp;

    % evaluate nominal state X_nom at current time
    X_nom_array{i} = eval(X_nom);

    % calculate and evaluate input U determine with LQR at current time
    U_lqr{i} = -k_lqr{i}*(X - X_nom) + U_nom;
    U_lqr{i} = eval(U_lqr{i});

end
```

```
display('     Finished computing gains k_lqr')


%% 6. CONVERT TO SIMULINK-COMPATIBLE DATA STRUCTURES

% Splitting X_nom_array ("3x1") into three time-stamped arrays ("1x2", time and x/y/theta
value)
x_nom_array = transpose(t_array);
y_nom_array = transpose(t_array);
theta_nom_array = transpose(t_array);

% Splitting k_lqr ("2x3") into two time-stamped arrays ("1x4", time and first row of k_lqr)
k_lqr_one = transpose(t_array);
k_lqr_two = transpose(t_array);


for i = 1:length(t_array)
    % X_nom_array
    X_nom_array_tmp = X_nom_array{i};

    x_nom_array(i,2) = X_nom_array_tmp(1);
    y_nom_array(i,2) = X_nom_array_tmp(2);

    %{
    % Limit angle values to [-pi, pi]
    tmp_angle = X_nom_array_tmp(3);
    theta_nom_array(i,2) = tmp_angle - 2*pi*floor( (tmp_angle)/(2*pi) );
    %}

    theta_nom_array(i,2) = X_nom_array_tmp(3); % no limiting

    % k_lqr
    k_lqr_tmp = k_lqr{i};

    k_lqr_one(i,2:4) = k_lqr_tmp(1,:);
    k_lqr_two(i,2:4) = k_lqr_tmp(2,:);

end


%% 7. TRAIN ARTIFICIAL NEURAL NETWORK WITH LQR GAIN k_lqr

if useNN

    % using MATLAB's Neural Network Toolbox
    display('Training Neural Networks to learn LQR gains...')
    hiddenLayers = [7 7]; % use [7 7] for best results

    % use standard traingd (average results) or trainlm (best results)
    global nn11; global nn12; global nn13; global nn21; global nn22; global nn23;
    nn11 = fitnet(hiddenLayers, 'trainlm'); nn21 = fitnet(hiddenLayers, 'trainlm');
    nn12 = fitnet(hiddenLayers, 'trainlm'); nn22 = fitnet(hiddenLayers, 'trainlm');
    nn13 = fitnet(hiddenLayers, 'trainlm'); nn23 = fitnet(hiddenLayers, 'trainlm');

    % train neural network w/ data
    [nn11, pr11] = train(nn11, k_lqr_one(:,1)', k_lqr_one(:,2)');
    [nn12, pr12] = train(nn12, k_lqr_one(:,1)', k_lqr_one(:,3)');
    [nn13, pr13] = train(nn13, k_lqr_one(:,1)', k_lqr_one(:,4)');
    [nn21, pr21] = train(nn21, k_lqr_two(:,1)', k_lqr_two(:,2)');
    [nn22, pr22] = train(nn22, k_lqr_two(:,1)', k_lqr_two(:,3)');
    [nn23, pr23] = train(nn23, k_lqr_two(:,1)', k_lqr_two(:,4)');
    %view(nn11)

    display('     Finished training Neural Networks')

    display('Testing trained Neural Networks...')
    for i = 1:length(t_array)
        nn11_out(i) = sim(nn11, t_array(i));
        nn12_out(i) = sim(nn12, t_array(i));
```

```
        nn13_out(i) = sim(nn13, t_array(i));
        nn21_out(i) = sim(nn21, t_array(i));
        nn22_out(i) = sim(nn22, t_array(i));
        nn23_out(i) = sim(nn23, t_array(i));
    end

    if plotFigures
        figure
        subplot(2,3,1)
        plot(t_array, nn11_out,'LineWidth',3)
        hold on
        plot(t_array, k_lqr_one(:,2),'--','LineWidth',2)
        grid on
        title('NN k_{lqr11} & actual k_{lqr11}','FontSize', 15)
        legend('NN k_{lqr}', 'Actual k_{lqr}')
        xlabel('time [s]')
        ylabel('gain')

        subplot(2,3,2)
        plot(t_array, nn12_out,'LineWidth',3)
        hold on
        plot(t_array, k_lqr_one(:,3),'--','LineWidth',2)
        title('NN k_{lqr12} & actual k_{lqr12}','FontSize', 15)
        grid on
        xlabel('time [s]')
        ylabel('gain')

        subplot(2,3,3)
        plot(t_array, nn13_out,'LineWidth',3)
        hold on
        plot(t_array, k_lqr_one(:,4),'--','LineWidth',2)
        title('NN k_{lqr13} & actual k_{lqr13}','FontSize', 15)
        grid on
        xlabel('time [s]')
        ylabel('gain')

        subplot(2,3,4)
        plot(t_array, nn21_out,'LineWidth',3)
        hold on
        plot(t_array, k_lqr_two(:,2),'--','LineWidth',2)
        title('NN k_{lqr21} & actual k_{lqr21}','FontSize', 15)
        grid on
        xlabel('time [s]')
        ylabel('gain')

        subplot(2,3,5)
        plot(t_array, nn22_out,'LineWidth',3)
        hold on
        plot(t_array, k_lqr_two(:,3),'--','LineWidth',2)
        title('NN k_{lqr22} & actual k_{lqr22}','FontSize', 15)
        grid on
        xlabel('time [s]')
        ylabel('gain')

        subplot(2,3,6)
        plot(t_array, nn23_out,'LineWidth',3)
        hold on
        plot(t_array, k_lqr_two(:,4),'--','LineWidth',2)
        title('NN k_{lqr23} & actual k_{lqr23}','FontSize', 15)
        grid on
        xlabel('time [s]')
        ylabel('gain')
    end

    display('     Finished testing Neural Networks')

end


%% 8. SIMULATION
```

```
% Initial conditions (coordinates)
x_0 = INITIAL_X;
y_0 = INITIAL_Y;
theta_0 = INITIAL_THETA;

% Reference (nominal) velocities
v_c_nom = r*omega_0;
omega_0;


if useSimulink

    display('Started simulation with Simulink')
    sim('QBot_model_2_4', t_array)

    % Re-format inputs (from 3D to 2D)
    tmpInputs = [];
    for i = 1:length(t_array)
        tmpInputs = [tmpInputs; simInputs(1,1,i), simInputs(1,2,i)];
    end
    inputs = tmpInputs;

    display('     Finished simulation with Simulink')

else


    display('Started in-code simulation')
    x = x_0; y = y_0; theta = theta_0;
    inputs = [v_c_nom, omega_0];


    for i = 1:length(t_array)-1

        % Compute velocities
        velocities = QBot_function([inputs(i,1), inputs(i,2), theta(i)]);
        dx = velocities(1); dy = velocities(2); dtheta = velocities(3);

        % Update position
        x(i+1) = x(i) + dx*TIME_INCREMENT;
        y(i+1) = y(i) + dy*TIME_INCREMENT;
        theta(i+1) = theta(i) + dtheta*TIME_INCREMENT;

        tmp = [x(i+1), y(i+1), theta(i+1)];

        % Subtract nominal position from position
        tmp = tmp - [x_nom_array(i,2), y_nom_array(i,2), theta_nom_array(i,2)];

        % Multiply result by LQR gain

        if useNN
            nn_klqr(1,1) = sim(nn11, t_array(i));
            nn_klqr(1,2) = sim(nn12, t_array(i));
            nn_klqr(1,3) = sim(nn13, t_array(i));
            nn_klqr(2,1) = sim(nn21, t_array(i));
            nn_klqr(2,2) = sim(nn22, t_array(i));
            nn_klqr(2,3) = sim(nn23, t_array(i));
            tmp = nn_klqr*transpose(tmp);
        else
            tmp = k_lqr{i}*transpose(tmp);
        end

        tmp = transpose(tmp);

        % Subtract result from nominal inputs
        tmp = [v_c_nom omega_0] - tmp;
        inputs(i+1,:) = tmp;

        if loopAnimation
```

```
            plot(x(i),y(i),'.','color','b','markers',12);
            hold on
            grid on
            pause(0.05);
        end

    end

    tout = t_array;
    display('     Finished in-code simulation')
end


% ODOMETER
odometer = totalDistance(sim_x,sim_y);
fprintf('Total distance travelled: %.2f m\n', odometer);
% calibrate with 2*pi*r compared to robot starting at (r, 0, pi/2)


%% 9. EXPORT DATA TO EXCEL SHEET
% Radius and period determine file name, initial conditions determine sheet
% name (time parameters not considered).

if exportToExcel

    filename = sprintf('Results_QBot_R%0.2f_P%0.2f.xlsx', RADIUS, PERIOD);
    sheetname = sprintf('x%0.2f_y%0.2f_th0%0.2f', INITIAL_X, INITIAL_Y, INITIAL_THETA);

    % Time
    xlswrite(filename, cellstr('time'), sheetname, 'A1');
    xlswrite(filename, transpose(t_array), sheetname, 'A2');

    % Applied inputs (v_c and omega)
    inputs = squeeze(inputs);

    xlswrite(filename, cellstr('v_c'), sheetname, 'B1');
    xlswrite(filename, transpose(inputs(1,:)), sheetname, 'B2');

    xlswrite(filename, cellstr('omega'), sheetname, 'C1');
    xlswrite(filename, transpose(inputs(2,:)), sheetname, 'C2');

    % k_lqr
    xlswrite(filename, cellstr('k_lqr_11'), sheetname, 'D1');
    xlswrite(filename, k_lqr_one(:,2), sheetname, 'D2');

    xlswrite(filename, cellstr('k_lqr_12'), sheetname, 'E1');
    xlswrite(filename, k_lqr_one(:,3), sheetname, 'E2');

    xlswrite(filename, cellstr('k_lqr_13'), sheetname, 'F1');
    xlswrite(filename, k_lqr_one(:,4), sheetname, 'F2');

    xlswrite(filename, cellstr('k_lqr_21'), sheetname, 'G1');
    xlswrite(filename, k_lqr_two(:,2), sheetname, 'G2');

    xlswrite(filename, cellstr('k_lqr_22'), sheetname, 'H1');
    xlswrite(filename, k_lqr_two(:,3), sheetname, 'H2');

    xlswrite(filename, cellstr('k_lqr_23'), sheetname, 'I1');
    xlswrite(filename, k_lqr_two(:,4), sheetname, 'I2');

    display('Exported data to Excel sheet')

end


%% 10. PLOTS

if plotFigures
    display('Plotting graphs...')
```

```
%% A. PATH and TRACKING ERROR: Nominal vs Simulation
figA = figure('pos',[10 10 1000 400]);
subplot(1,2,1)
nom = [X_nom_array{1,:}];
plot(nom(1,:), nom(2,:),'--','LineWidth',4, 'color', 'r');
hold on
plot(sim_x, sim_y, 'LineWidth',2, 'color', 'b');
title('Nominal and actual motion in xy plane','FontSize', 15)
xlabel('x-coordinate [m]','FontSize', 12)
ylabel('y-coordinate [m]','FontSize', 12)
grid on
legend('Nominal', 'Simulation')

subplot(1,2,2)
plot(tout, simError(1:end,1),'LineWidth',2, 'color', 'r');
hold on
plot(tout, simError(1:end,2),'LineWidth',2, 'color', 'g');
hold on
plot(tout, simError(1:end,3),'LineWidth',2, 'color', 'b');
hold on
title('Tracking error (simulation vs nominal positions)','FontSize', 15)
xlabel('time [s]','FontSize', 12)
ylabel('error','FontSize', 12)
grid on
legend('x error', 'y error', '\theta error')


%% B. COORDINATES: Simulation
figB = figure('pos',[10 10 1000 400]);
subplot(1,2,1)
plot(tout,sim_x,'b','LineWidth',2)
hold on
plot(tout,sim_y,'r','LineWidth',2)
title('Simulation coordinates (x,y) vs time','FontSize', 15)
xlabel('time [s]','FontSize', 12)
ylabel('position [m]','FontSize', 12)
grid on
legend('x position', 'y position')

subplot(1,2,2)
plot(tout,sim_theta,'LineWidth',2)
title('Simulation angular position \theta vs time','FontSize', 15)
xlabel('time [s]','FontSize', 12)
ylabel('theta angle [rad]','FontSize', 12)
grid on


%% C. INPUTS: Simulation
figC = figure('pos',[10 10 1000 400]);
subplot(1,2,1);
plot(tout,inputs(:,1),'LineWidth',2,'Color','m');
title('Simulation input v_c vs time','FontSize', 15)
grid on
xlabel('time [s]','FontSize', 12)
ylabel('input [m/s]','FontSize', 12)
legend('v_c')

subplot(1,2,2);
plot(tout,inputs(:,2),'LineWidth',2,'Color','c');
title('Simulation input \omega vs time','FontSize', 15)
xlabel('time [s]','FontSize', 12)
ylabel('input [rad/s]','FontSize', 12)
legend('\omega')
grid on


%% D. LQR Gains and Riccati Equation solutions
figD = figure('pos',[10 10 1000 400]);
subplot(1,2,1)
plot(tout,k_lqr_one(:,2),'r',  tout,k_lqr_one(:,3),'m',  tout,k_lqr_one(:,4),'y', ...
```

```
        tout,k_lqr_two(:,2),'b',  tout,k_lqr_two(:,3),'c',  tout,k_lqr_two(:,4),'g', ...
        'LineWidth',2);
    title('Gains k\_lqr vs time','FontSize', 15)
    xlabel('time [s]','FontSize', 12)
    ylabel('k\_lqr','FontSize', 12)
    grid on
    legend('k\_lqr11', 'k\_lqr12', 'k\_lqr13', 'k\_lqr21', 'k\_lqr22', 'k\_lqr23')

    subplot(1,2,2)
    K11 = K(1,1,:); K11 = K11(:,:)';
    plot(tout,K11,'LineWidth',2);
    hold on
    K12 = K(1,2,:); K12 = K12(:,:)';
    plot(tout,K12,'LineWidth',2);
    hold on
    K13 = K(1,3,:); K13 = K13(:,:)';
    plot(tout,K13,'LineWidth',2);
    hold on
    K21 = K(2,1,:); K21 = K21(:,:)';
    plot(tout,K21,'LineWidth',2);
    hold on
    K22 = K(2,2,:); K22 = K22(:,:)';
    plot(tout,K22,'LineWidth',2);
    hold on
    K23 = K(2,3,:); K23 = K23(:,:)';
    plot(tout,K23,'LineWidth',2);
    hold on
    K31 = K(3,1,:); K31 = K31(:,:)';
    plot(tout,K31,'LineWidth',2);
    hold on
    K32 = K(3,2,:); K32 = K32(:,:)';
    plot(tout,K32,'LineWidth',2);
    hold on
    K33 = K(3,3,:); K33 = K33(:,:)';
    plot(tout,K33,'LineWidth',2);
    hold on
    title('Riccati equation solutions K vs time','FontSize', 15)
    xlabel('time [s]','FontSize', 12)
    ylabel('solution','FontSize', 12)
    grid on
    legend('K\_11', 'K\_12', 'K\_13', 'K\_21', 'K\_22', 'K\_23', 'K\_31', 'K\_32', 'K\_33')

    display('     Finished plotting graphs')


    %% EXPORT PLOTS
    simTitle = sprintf('sim_r%.2f_p%.2f_x%.2f_y%.2f_th%.2f', ...
        RADIUS, PERIOD, INITIAL_X, INITIAL_Y, INITIAL_THETA);

    if exportFigures
        display('Exporting figures...')

        saveas(figA, sprintf('%s_path_and_error.png',simTitle));
        saveas(figB, sprintf('%s_coordinates.png',simTitle));
        saveas(figC, sprintf('%s_inputs.png',simTitle));
        saveas(figD, sprintf('%s_LQR_Riccati.png',simTitle));

        display('     Finished exporting figures')
    end

end

display('QBot_2_4 program ended')
```

APPENDIX C

RICCATI EQUATION MATLAB FUNCTION

Function name: riccati_2_4.m

```
function dXdt = riccati(t,X,A,B,R,Q)

global omega_0

A = eval(A);
B = eval(B);

X = reshape(X,size(A));  % Convert from (n^2)-by-1 to n-by-n

dXdt = -X*A - A.'*X + X*B*inv(R)*B.'*X - Q;
%dXdt = A.'*X + X*A - X*B*B.'*X + Q;

dXdt = dXdt(:); % Convert from n-by-n to (n^2)-by-1

end
```

APPENDIX D

QBOT 2 SYSTEM MODEL FUNCTION

Function name: QBot_function.m

```
function [ dX ] = Qbot_function( input_vector )
% Function takes inputs v_c, omega and theta (current)
% computes and returns outputs dx, dy, dtheta (new).

% Extract variables and parameters from the input vector
v_c = input_vector(1);
omega = input_vector(2);
theta = input_vector(3);

dx = v_c*cos(theta);
dy = v_c*sin(theta);
dtheta = omega;

% dX
dX = [dx; dy; dtheta];

end
```

APPENDIX E

ANGLE TRACKING MATLAB CODE

Code of MATLAB Function block found in:

"QBot2_Optitrack.xls" model / "OptiTrack x, y, theta" block / "Angle Tracker" block

```
function [theta_sys, k, theta_ot_prev]   = fcn(input)

theta_ot_curr = input(1);
k = input(2);
theta_ot_prev = input(3);

threshold = 3;

if theta_ot_curr < -threshold && theta_ot_prev > threshold
    k = k + 1;
elseif theta_ot_curr > threshold && theta_ot_prev < -threshold
    k = k - 1;
else
    k = k;
end

theta_ot_prev = theta_ot_curr;
theta_sys = theta_ot_curr + k*(pi*2);

end
```

APPENDIX F

INITIAL COORDINATES MATLAB CODE

Script name: initialOptiTrack.m

```
% Quanser QBot 2 INITIAL COORDINATES
% Quickly print initial conditions after running OptiTrack experiment


clc

fprintf('First OptiTrack: x = %.4f, y = %.4f, theta = %.4f\n', ...
    optitrack_x(1), optitrack_y(1), optitrack_theta(1))

fprintf('Second OptiTrack: x = %.4f, y = %.4f, theta = %.4f\n', ...
    optitrack_x(2), optitrack_y(2), optitrack_theta(2))

fprintf('Third OptiTrack: x = %.4f, y = %.4f, theta = %.4f\n', ...
    optitrack_x(3), optitrack_y(3), optitrack_theta(3))

fprintf('Fourth OptiTrack: x = %.4f, y = %.4f, theta = %.4f\n', ...
    optitrack_x(4), optitrack_y(4), optitrack_theta(4))
```

APPENDIX G

SIMULATION VS EXPERIMENT COMPARISON MATLAB CODE

Script name: QBot_2_4_postOptiTrack.m

```
% Quanser QBot 2
% Post-OptiTrack Analysis
% Version 2.4
% Code by Vittorio Longi

clc
clear v_c_sim omega_sim v_c_optitrack omega_optitrack

plotResults = true;
exportPlots = false;
exportWorkspace = false;

%% ODOMETER
odometer_optitrack = totalDistance(optitrack_x(2:end), optitrack_y(2:end));
odometer_sim = totalDistance(sim_x(2:end), sim_y(2:end));
fprintf('Simulation path length: %.2f \t\t OptiTrack path length: %.2f m\n',
odometer_optitrack,odometer_sim);


if plotResults
    display('Plotting figures')

    %% 1. PATH: OptiTrack vs Simulation
    fig1 = figure('pos',[10 10 1000 400]);
    subplot(1,2,1)
    plot(optitrack_x(2:end), optitrack_y(2:end),'LineWidth',2, 'color', 'r');
    hold on
    plot(x_nom_array(:,2),y_nom_array(:,2),'LineWidth',2, 'color', 'g');
    title('OptiTrack and nominal path in xy plane','FontSize', 15)
    xlabel('x-coordinate [m]','FontSize', 12)
    ylabel('y-coordinate [m]','FontSize', 12)
    grid on
    legend('OptiTrack', 'Nominal')
    xlim([ min([min(sim_x(:)),min(optitrack_x(2:end))]),
max([max(sim_x(:)),max(optitrack_x(2:end))]) ]);
    ylim([min([min(sim_y(:)),min(optitrack_y(2:end))]),
max([max(sim_y(:)),max(optitrack_y(2:end))]) ]);
    daspect([1 1 1])

    subplot(1,2,2)
    plot(sim_x(:),sim_y(:),'LineWidth',2, 'color', 'b');
    hold on
    plot(x_nom_array(:,2),y_nom_array(:,2),'LineWidth',2, 'color', 'g');
    title('Simulation and nominal path in xy plane','FontSize', 15)
    xlabel('x-coordinate [m]','FontSize', 12)
    ylabel('y-coordinate [m]','FontSize', 12)
    grid on
    legend('Simulation', 'Nominal')
```

```
    xlim([ min([min(sim_x(:)),min(optitrack_x(2:end))]),
max([max(sim_x(:)),max(optitrack_x(2:end))]) ]);
    ylim([min([min(sim_y(:)),min(optitrack_y(2:end))]),
max([max(sim_y(:)),max(optitrack_y(2:end))]) ]);
    daspect([1 1 1])


    %% 2. COORDINATES: Simulation vs Nominal   and   OptiTrack vs Nominal
    fig2 = figure('pos',[10 10 1000 800]);

    % x position
    subplot(3,3,1)
    plot(tout, x_nom_array(:,2),'LineWidth',2, 'color', 'g');
    grid on
    title('x position vs time','FontSize', 15)
    xlabel('time [s]','FontSize', 12)
    ylabel('x-coordinate [m]','FontSize', 12)
    legend('Nominal')

    subplot(3,3,4)
    plot(tout, sim_x,'LineWidth',2, 'color', 'b');
    grid on
    xlabel('time [s]','FontSize', 12)
    ylabel('x-coordinate [m]','FontSize', 12)
    legend('Simulation')

    subplot(3,3,7)
    plot(tout, optitrack_x(2:end),'LineWidth',2, 'color', 'r');
    grid on
    xlabel('time [s]','FontSize', 12)
    ylabel('x-coordinate [m]','FontSize', 12)
    legend('OptiTrack')


    % y position
    subplot(3,3,2)
    plot(tout, y_nom_array(:,2),'LineWidth',2, 'color', 'g');
    grid on
    xlabel('time [s]','FontSize', 12)
    ylabel('y-coordinate [m]','FontSize', 12)
    title('y position vs time','FontSize', 15)
    legend('Nominal')

    subplot(3,3,5)
    plot(tout, sim_y,'LineWidth',2, 'color', 'b');
    xlabel('time [s]','FontSize', 12)
    ylabel('y-coordinate [m]','FontSize', 12)
    grid on
    legend('Simulation')

    subplot(3,3,8)
    plot(tout, optitrack_y(2:end),'LineWidth',2, 'color', 'r');
    xlabel('time [s]','FontSize', 12)
    ylabel('y-coordinate [m]','FontSize', 12)
    grid on
    legend('OptiTrack')


    % theta position
    subplot(3,3,3)
    plot(tout, theta_nom_array(:,2),'LineWidth',2, 'color', 'g');
    grid on
    xlabel('time [s]','FontSize', 12)
    ylabel('\theta-angle [rad]','FontSize', 12)
    title('\theta position vs time','FontSize', 15)
    legend('Nominal')

    subplot(3,3,6)
    plot(tout, sim_theta,'LineWidth',2, 'color', 'b');
    xlabel('time [s]','FontSize', 12)
```

```
ylabel('\theta-angle [rad]','FontSize', 12)
grid on
legend('Simulation')

subplot(3,3,9)
plot(tout, optitrack_theta(2:end),'LineWidth',2, 'color', 'r');
grid on
xlabel('time [s]','FontSize', 12)
ylabel('\theta-angle [rad]','FontSize', 12)
legend('OptiTrack')



%% 3. TRACKING ERROR: OptiTrack vs Simulation

% Fix first and last few theta OptiTrack error (jump at the beginning and end)
optitrackError(1,3) = optitrackError(3,3);
optitrackError(2,3) = optitrackError(3,3);
optitrackError(end-1,3) = optitrackError(end-2,3);
optitrackError(end,3) = optitrackError(end-2,3);

fig3 = figure('pos',[10 10 1000 400]);
subplot(1,2,1)
plot(tout, optitrackError(2:end,1),'LineWidth',2, 'color', 'r');
hold on
plot(tout, optitrackError(2:end,2),'LineWidth',2, 'color', 'g');
hold on
plot(tout, optitrackError(2:end,3),'LineWidth',2, 'color', 'b');
hold on
title('Tracking error (OptiTrack vs nominal position)','FontSize', 15)
xlabel('time [s]','FontSize', 12)
ylabel('error','FontSize', 12)
grid on
legend('x error', 'y error', '\theta error')

subplot(1,2,2)
plot(tout, simError(1:end,1),'LineWidth',2, 'color', 'r');
hold on
plot(tout, simError(1:end,2),'LineWidth',2, 'color', 'g');
hold on
plot(tout, simError(1:end,3),'LineWidth',2, 'color', 'b');
hold on
title('Tracking error (simulation vs nominal positions)','FontSize', 15)
xlabel('time [s]','FontSize', 12)
ylabel('error','FontSize', 12)
grid on
legend('x error', 'y error', '\theta error')


%% 4. INPUTS: OptiTrack vs Simulation

for i = 1:length(tout)
    v_c_sim(i) = simInputs(1,1,i);
    omega_sim(i) = simInputs(1,2,i);
end

for i = 1:length(tout)
    v_c_optitrack(i) = optitrackInputs(1,1,i+1);
    omega_optitrack(i) = optitrackInputs(1,2,i+1);
end

fig4 = figure('pos',[10 10 1000 400]);
subplot(1,2,1)
plot(tout, v_c_sim,'LineWidth',2, 'color', 'b');
hold on
plot(tout, v_c_optitrack,'LineWidth',2, 'color', 'r');
hold on
title('Forward Velocity v_C over time','FontSize', 15)
xlabel('time [s]','FontSize', 12)
ylabel('inputs','FontSize', 12)
```

```
    grid on
    legend('Simulation', 'OptiTrack')
    ylim([-1 1])

    subplot(1,2,2)
    plot(tout, omega_sim,'LineWidth',2, 'color', 'b');
    hold on
    plot(tout, omega_optitrack,'LineWidth',2, 'color', 'r');
    hold on
    title('Angular Velocity \omega over time','FontSize', 15)
    xlabel('time [s]','FontSize', 12)
    ylabel('inputs','FontSize', 12)
    grid on
    legend('Simulation', 'OptiTrack')
    ylim([-1 2])

    display('     Finished plotting figures')

end


%% EXPERIMENT TITLE

expTitle = sprintf('r%.2f_p%.2f_x%.2f_y%.2f_th%.2f', ...
        RADIUS, PERIOD, INITIAL_X, INITIAL_Y, INITIAL_THETA);

%% EXPORT PLOTS

if exportPlots
    display('Exporting plots')
    saveas(fig1, sprintf('%s_path.png',expTitle));
    saveas(fig2, sprintf('%s_coordinates.png',expTitle));
    saveas(fig3, sprintf('%s_error.png',expTitle));
    saveas(fig4, sprintf('%s_inputs.png',expTitle));
    display('     Finished exporting plots')
end

%% EXPORT WORKSPACE

if exportWorkspace
    display('Exporting workspace')
    filename = sprintf('%s.MAT', expTitle);
    save(filename);
    display('     Finished exporting workspace')
end
```

APPENDIX H

DISTANCE TRAVELLED MATLAB CODE

Function name: totalDistance.m

```
function [ distance ] = totalDistance( x_coordinates, y_coordinates )
% Given a set of (x,y) coordinate pairs, this function returns the total
% distance traced by the coordinate pairs.

if length(x_coordinates) ~= length(y_coordinates)
    display('ERROR: Length of x and y coordinates history must be same');
end

distance = 0;

for i = 2:length(x_coordinates)

    x1 = x_coordinates(i-1);
    x2 = x_coordinates(i);
    y1 = y_coordinates(i-1);
    y2 = y_coordinates(i);

    tmpDistance = sqrt((x2-x1)^2 + (y2-y1)^2);

    distance = distance + tmpDistance;

end


end
```