2019-12-13

Autonomous Path Planning Under Cyber Attacks

Christopher Michael Letherbarrow
Worcester Polytechnic Institute

Minh Hoang Le
Worcester Polytechnic Institute

Autonomous Path Planning Under Cyber Attacks

A Major Qualifying Project Report Submitted to the Faculty of Worcester Polytechnic Institute

By:

Minh Le

Christopher Letherbarrow

December 13, 2019

Advised by:

Professor Andrew Clark

Abstract

Autonomous path planning is an important application in the field of robotics and control systems engineering, especially with the rapid development and commercialization of self-driving cars and unmanned autonomous vehicles. Some of the biggest concerns in these cyber-physical systems are safety and security. Many different attacks can be and have been conducted on either the physical or cyber layer of these systems, and these attacks can cause devastating consequences. In this project, the team investigates techniques for planning and following a trajectory in the presence of an adversary who spoofs one or more sensors on a wheeled robot platform. The robot platform that the team uses is a Turtlebot 3 Burger controlled by an LQG controller. False data injection (FDI) attacks were performed on one or more sensors of the Turtlebot, and their effects on the robot's ability to follow a predetermined trajectory were analyzed. A mitigation method based on the barrier function, using sensors that are known to be secured and limiting the set of feasible control input so that the system never reaches the unsafe region, was also implemented and its effectiveness was tested. The team runs simulations and graphs the actual trajectory versus the reference trajectory and the mean square error between them to verify that the mitigation method allows the robot to maintain its ability to follow the reference trajectory. The team was successful in implementing an LQG controller in Python with a low value of dt and very accurate reference tracking, as well as proving that the mitigation method is effective against different false data injection attacks.

Acknowledgments

We would like to acknowledge the many individuals that have contributed to the success of this project. First of all, we would like to thank our project advisor, Professor Andrew Clark. Without his guidance and feedback, this project would have been nowhere near as successful as it is. We would also like to thank Hongchao Zhang and Zhouchi Li, who are always there for us when we need help. Finally, we want to thank the staff and faculty at Worcester Polytechnic Institute, who have assisted and educated us throughout our years here and have set us up for more years to come.

Table of Contents

Abstract ii

Acknowledgments iii

Table of Contents iv

List of Figures vi

1. Introduction 1

2. Background 2

    2.1 Wheeled Mobile Robot (WMR) Platform 2

    2.2. Existing Work: Model of Robot Environment 2

        2.2.1. States 3

        2.2.2. Feedback Linearization 4

    2.3 Existing Work: PID Controller with Feedback Linearization 5

    2.4 Existing Work: LQR/LQG Controller 6

    2.5 Existing Work: False Data Injection Attack and Mitigation 7

3. Methodology 11

    3.1 Choosing a New Platform 11

    3.2 Porting to Python: basic movement, ROS, organization of nodes 12

    3.3 Porting to Python: PID Controller 13

    3.4 Porting to Python: LQG controller 14

    3.5 Attack Simulation and Mitigation Methods 15

4. Testing and Results 19

    4.1 PID Controller 19

    4.2 LQG Controller 20

    4.3 Attack Simulation 22

    4.4 Mitigation Methods 24

5. Conclusion 30

5.1. Summary 30

5.2. Future Works & Improvements 30

6. References 31

7. Appendix 33

Appendix A: How to Use the Python Implementation 33

Appendix B: PID Controller Implementation in Python Code 36

Appendix C: LQG Controller Implementation in Python Code 43

Appendix D: Mitigation Method Simulation Code 50

Appendix E: Mitigation Method Implementation Code 53

Appendix F: Video Demos 61

List of Figures

Figure 1: Turtlebot 3 Burger 2
Figure 2: Model of WMR 3
Figure 3: The reference trajectory, as visualized by orange tape in the Secure Cyber Physical Systems Lab 5
Figure 4: Linearized system model block diagram 7
Figure 5: Node graph of the Python implementation of LQG controller 12
Figure 6: Block diagram of Python implementation 13
Figure 7: Simulation results of MATLAB version (left) and Python version (right) 16
Figure 8: Map generated by the Turtlebot's LIDAR 17
Figure 9: Full node graph of the Python implementation of the mitigation method 18
Figure 10: Actual trajectory vs Reference trajectory graph and Mean Square Error graph of PID controller 19
Figure 11: Three error zones, as seen on trajectory graph and mean square error graph 20
Figure 12: Trajectory and mean square error graph with T = 150 21
Figure 13: Trajectory and mean square error graph with T = 300 22
Figure 14: LQG reference tracking with attack on y-position IMU sensor 23
Figure 15: LQG reference tracking with attack on y-position IMU sensor every 20 time steps .. 24
Figure 16: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 150, dt = 0.02, random Gaussian noise) 25
Figure 17: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 100, dt = 0.01, random Gaussian noise) 26
Figure 18: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 150, dt = 0.02, constant attack) 27
Figure 19: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 300, dt = 0.02, random Gaussian noise) 28
Figure 20: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 50, dt = 0.02, random Gaussian noise).. 28
Figure 21: From left to right: LQG controller with no attack, LQG with mitigation method against random Gaussian attack, LQG with mitigation method against constant attack 29

1. Introduction

Within the field of robotics, autonomous operation is a key element in almost any application, especially with the development and commercialization of self-driving cars and the increased usage of modern unmanned ground vehicle (UGV) in military operations, construction, and space exploration [1][2]. However, many different attacks against these systems, especially sensor spoofing in the physical layer, have been documented [3]. These attacks raise a concern regarding the security and safety of these vehicles, since with a delicate control system a small piece of false data can cause dangerous instability issues. An example of this has been demonstrated by Charlie Miller and Chris Valasek, who were able to hack into an in-motion Jeep Cherokee and send false data to gain full control of the vehicle [4].

The project used a wheeled robot platform and its desired operation is to autonomously travel along a predetermined trajectory, controlled by an LQG controller. Additionally, the team implemented a mitigation method against a simulated spoofing attempt on one or more of the robot's sensors. The goal of this project was therefore divided into two specific objectives.

The first objective was to develop a real-time state feedback controller that can accurately model the real-world operation of the robot and correct its movement in order to travel along the given trajectory accurately. The team first began the process by taking pre-existing controllers and porting them over to work within the Robot Operating System (ROS) and using the Python language.

The second objective involved an element of cybersecurity on the robot platform. Within this simulation, the team considered an attacker that attempts to inject inaccurate data to one or more sensors on the robot platform. A mitigation strategy was then developed using data from the secured sensors and a barrier function.

The rest of this report is organized as follows: Chapter 2 describes the background to the project and the existing implementations that the work was based upon. Chapter 3 includes the team's methodology to implement this described controller, as well as the mitigation method on the simulated attack on the robot's sensors. Chapter 4 details the testing done on the robot in order to demonstrate the controller's effectiveness, the degradation of performance under a simulated attack, and the mitigation method to reduce the attack's effectiveness in disrupting the robot's operation, as well as the result of each test.

2. Background

2.1 Wheeled Mobile Robot (WMR) Platform

The WMR platform that was used during this project is the Turtlebot3 Burger [5]. The Turtlebot platform is a pre-designed two-wheeled robot that can be used for various research, educational and prototyping applications, and includes several position and movement sensors, as well as a 360-degree LIDAR for mapping and range-finding. The Turtlebot runs on a Raspberry Pi and an ARM-based Open Control Unit, both of which have open-source hardware and software that can be modified under an open-source license. This open-source nature and amount of pre-existing software for the Turtlebot makes it a favorable platform for the project.

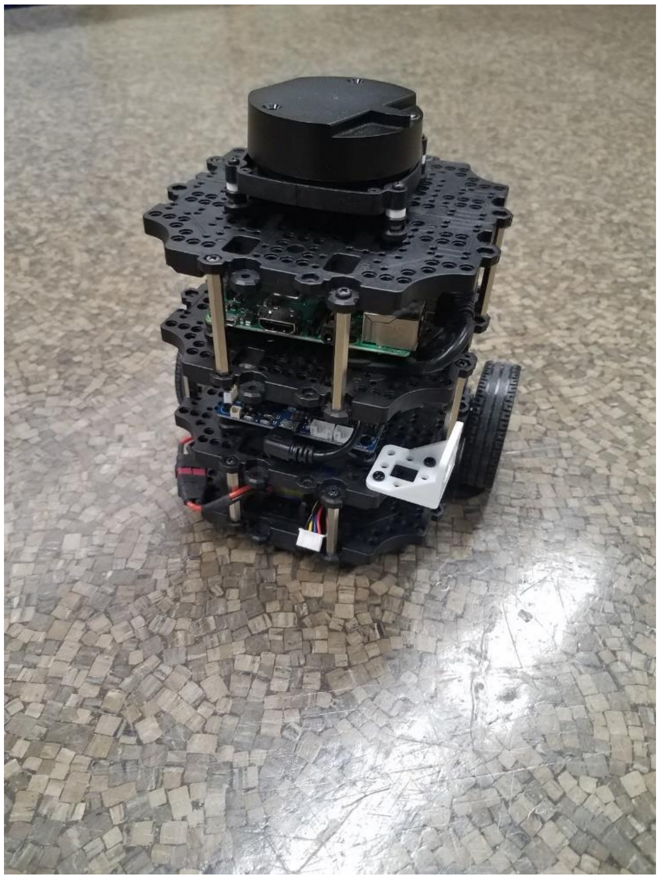


Figure 1: Turtlebot 3 Burger

2.2. Existing Work: Model of Robot Environment

In order to begin to determine a feedback controller for this robot, a description of the robot's environment and motion must first be developed. Once this description is developed, the robot’s system is then linearized using a feedback linearization process so a feedback controller can be used on the robot.

2.2.1. States

The robot’s state can be described by its current position x, y, and its orientation θ within the operating environment. Additionally, there are two control parameters: the robot's linear velocity, v, and its angular velocity, ω.

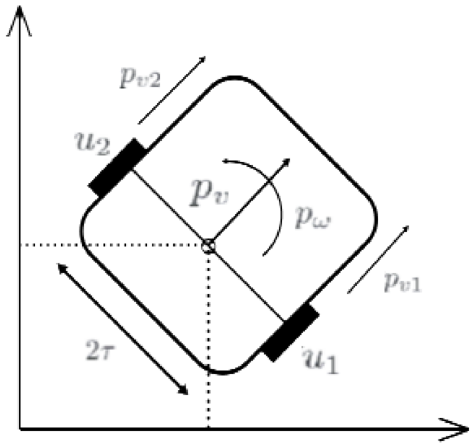


Figure 2: Model of WMR

This state vector, q, describes the robot’s current state at any given time:

$$q = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{1}$$

With this state vector, the state-space representation of this physical system can be developed. The state-space representation describes the robot’s position and orientation as a set of input, output, and state variables. The state of the robot can be described by:

$$x(t) = Ax(t) + \boldsymbol{B}u(t) + w(t) \tag{2}$$

with $x(t)$ being the state vector, $u(t)$ being the input vector, A being the state matrix, $\boldsymbol{B}$ being the input matrix. The output of the system, $y(t)$, can then be described by:

$$y(t) = \boldsymbol{C}x(t) + v(t) \tag{3}$$

with $\boldsymbol{C}$ being the output matrix and $v(t)$ being the measurement noise. This measurement noise is assumed to be insignificant.

2.2.2. Feedback Linearization

This state-space representation as developed is non-linear, that is, the change of the output is not directly proportional to the change of the input. In order to develop a feedback controller for this robot, this representation must be linearized, or transformed from a non-linear representation to a linear one using a change of variables and some control input. The feedback linearization procedure is described in detail in [6]. The output vector is defined as $\eta = (x, y)$. Differentiation with respect to time yields

$$\eta = \begin{matrix} x \\ y \end{matrix} = \begin{matrix} cos\theta & 0 \\ sin\theta & 0 \end{matrix} \binom{v}{\omega} \tag{4}$$

showing that only v affects η. Next, an integrator denoted by ξ is added so that

$$v = \xi, \qquad \xi = a \qquad \rightarrow \quad \eta = \xi \begin{matrix} \cos\theta \\ \sin\theta \end{matrix} \tag{5}$$

By further differentiating η, the following equations describing the dynamic compensator with control signals u_1, u_2 was obtained:

$$\begin{gathered} \xi = u_1\ cos\theta + u_2\ sin\theta \\ v = \xi \\ \omega = \frac{u_1\ cos\theta - u_2\ sin\theta}{\xi} \end{gathered} \tag{6}$$

which fully linearizes the system. Note that singularity occurs at $v = 0$. To avoid this singularity, the reference trajectory is assumed to be continuous and persistent, meaning at no point on the trajectory should $v = 0$. The fully linearized system can be written as:

$$\begin{matrix} x \\ y \\ x \\ y \end{matrix} = A \bullet \begin{matrix} x \\ y \\ x \\ y \end{matrix} + B \bullet \begin{matrix} u_1 \\ u_2 \end{matrix}, \tag{7}$$

$$\text{Where} \quad A = \begin{matrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}, \qquad B = \begin{matrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

In the team's implementation, the robot is following a figure-8 trajectory $(x_d(t), y_d(t))$ defined by the following equations:

$$x_d(t) = 0.8 \sin \frac{t}{10}, \qquad y_d(t) = 0.8 \sin \frac{t}{20} \tag{8}$$

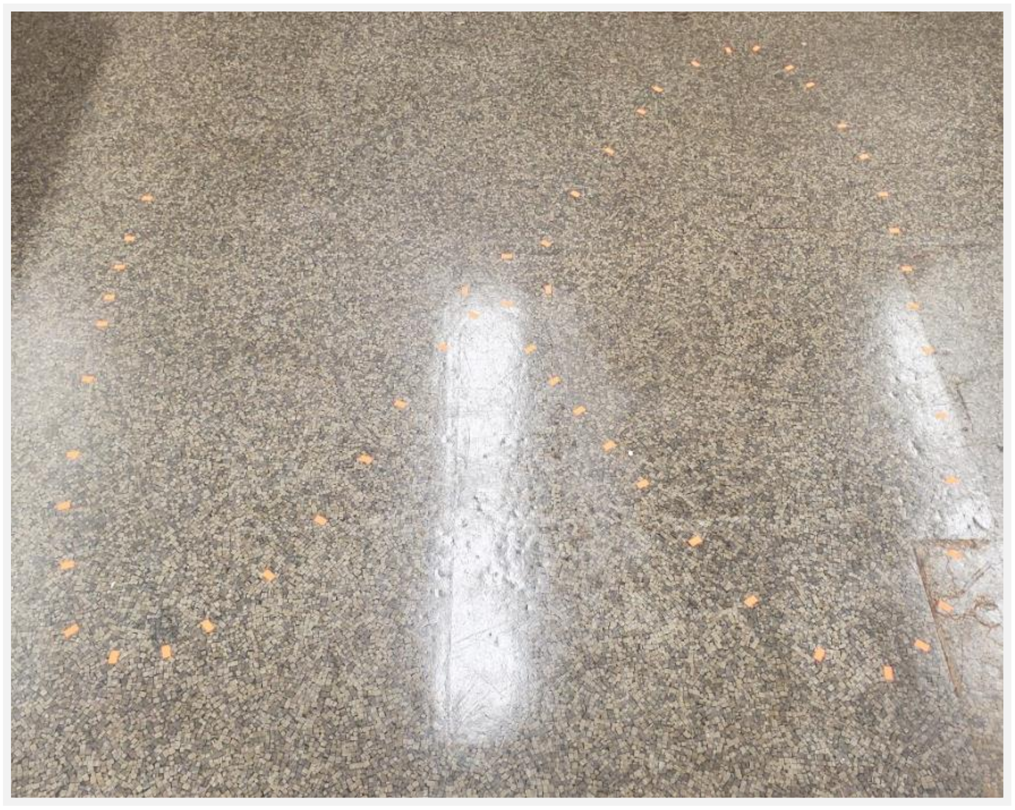


Figure 3: The reference trajectory, as visualized by orange tape in the Secure Cyber Physical Systems Lab

2.3 Existing Work: PID Controller with Feedback Linearization

A Proportional - Integral - Derivative (PID) Controller is one of the most basic and widely used feedback control loop mechanisms, due to its robustness and simplicity. A PID controller calculates the error value e(t) as the difference between a process variable - in this case, the position of the robot - and a setpoint - in this case, the reference trajectory. The PID controller then applies a correction to that error based on the predetermined proportional, integral, and derivative terms.

PID controllers have been used widely in different reference tracking applications [7]. For this PID controller implementation of a reference tracking robot platform, the team is using a similar control scheme to the one proposed by Oriolo, De Luca, and Vendittelli in [6], which employs dynamic feedback linearization for better posture stabilization and reference tracking performance.

Assume that the reference trajectory $(x_d(t), y_d(t))$ (7) is continuous and persistent, the error $e_1(t) = x_d(t) - x(t)$ and $e_2(t) = y_d(t) - y(t)$ can be calculated with $(x(t), y(t))$ as the robot's physical position. Using the feedback linearization process mentioned in section 2.2, a PD controller can be designed for the robot system [6]

$$\begin{aligned} u_1 &= x_d + k_{p1}(x_d - x) + k_{d1}(x_d - x) \\ u_2 &= y_d + k_{p2}(y_d - y) + k_{d2}(y_d - y) \end{aligned} \tag{9}$$

with proportional gains $k_{p1} = k_{p2} = 0.8$ and derivative gains $k_{p1} = k_{p2} = 1$. Control signals u_1, u_2 will then be fed to the compensator to obtain the real control inputs to the robot.

2.4 Existing Work: LQR/LQG Controller

The PID controller is one of the most popular controllers in classical control, which works best with a linear time-invariant (LTI) system with a single input and a single output. However, most WMR platforms, and specifically the Turtlebot3 Burger platform, are non-linear, dynamic systems with multiple inputs and outputs. The system has been transformed to be controllable with classical control methods using the dynamic feedback linearization method mentioned in section 2.3. However, some of the controller's design parameters must be modified, including control parameters and feedback gains, in order to get a satisfactory performance, and in many cases, this can prove to be difficult. A different approach to this reference tracking is optimal control, which minimizes the cost while optimizes the control performance.

Due to the nature of having to find the optimal control parameters and feedback gains for the system, the classic linear-quadratic regulator (LQR) controller is a good solution to this problem since its algorithm provides optimization to design a system that minimizes a predetermined cost function. Following is a summary of the system model that the team is working with, developed by Luyao Niu, Zhouchi Li and Hongchao Zhang. The system state can be described by equation (7). The quadratic cost function in continuous time is described as:

$$\begin{gathered} J = \int_0^T \left(x^T(t) - r(t) \;\; Q\big(x(t) - r(t)\big) + u^T(t)Ru(t) \right) dt \\ \text{Where} \quad Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad R = \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} \end{gathered} \tag{10}$$

Solving the two equations, the optimal control input was obtained:

$$u_k = u_d(t) - \left(R + B^T P_k B\right)^{-1} * B^T * (P_k * A * x_k + s_k) \tag{11}$$

Since the system model includes processing noise and the state of the system is not directly observed, the output and process noise are passed through a Kalman filter to receive the predicted state x, then using this as well as the reference trajectory to compute the optimal control input, making this a Linear Quadratic Gaussian (LQG) controller. Figure 4 shows the block diagram for the complete system model.

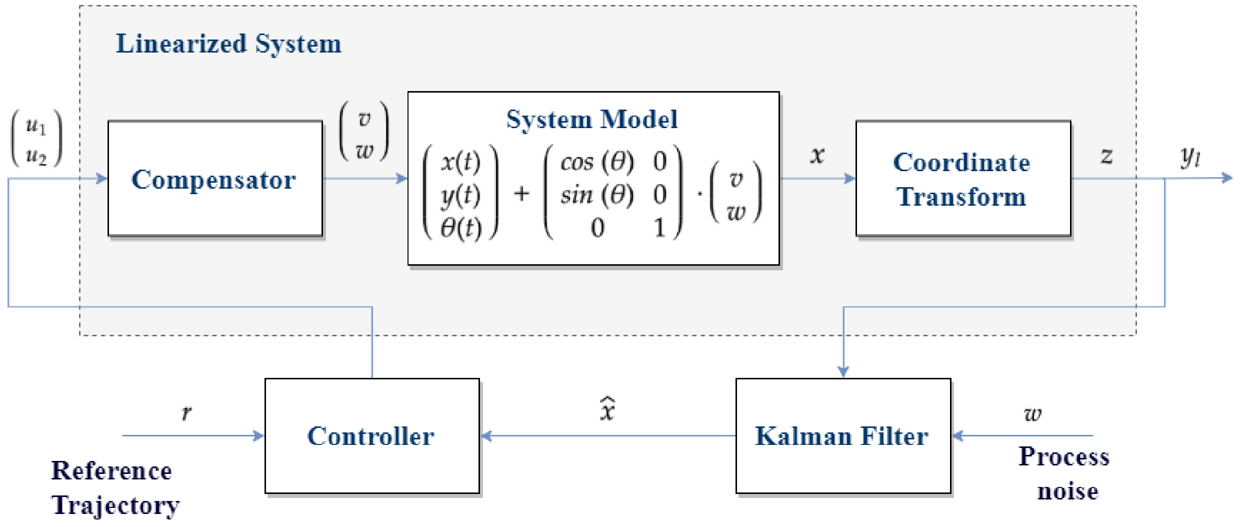

Linearized System
$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$
Compensator
$\begin{pmatrix} v \\ w \end{pmatrix}$
System Model
$\begin{pmatrix} x(t) \\ y(t) \\ \theta(t) \end{pmatrix} + \begin{pmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v \\ w \end{pmatrix}$
x
Coordinate Transform
z
y_l
r
Reference Trajectory
Controller
$\hat{x}$
Kalman Filter
w
Process noise

Figure 4: Linearized system model block diagram

2.5 Existing Work: False Data Injection Attack and Mitigation

Once the controller has been developed in order to properly have the robot travel the reference trajectory, it will then be subjected to a simulated data injection attack, which comprises one or more of the robot's sensors. During this attack, there is a number of sensors that are considered secure, and the information provided by these sensors is considered to be accurate. The other sensors on the robot, however, are considered to be compromisable due to a simulated malicious attacker, and the information received from them cannot be considered as accurate.

The attacks investigated in this project will add random Gaussian noise and constant value to the actual positional data output by the sensor. Therefore, during these attack, the controller as implemented would have an increasing amount of error and could not be used to properly follow the given trajectory during operation. The adversary is also assumed to have knowledge of the control policy and is intelligent enough to generate an attacking strategy $\boldsymbol{\tau}$ that can alter the trajectory of the robot.

False data injection attacks like this against cyber-physical systems (CPSs), especially on networked systems, have been widely studied in modern control theory. These systems are different from classical control systems in the fact that many operations and communications occur on a shared wired or wireless network [8], making them a lot more open to the cyber world, and in turn, a lot more vulnerable to attacks on the data transmission or communication layer [9]. Research has been conducted on how to detect these kinds of attack [10], and there exists many pieces of research on ways to mitigate their effects, for example, a Polynomial-based Compromise-Resilient En-route Filtering scheme (PCREF) [11] or a filtering-and-learning system to filter out the malicious sensor through an optimal filter [12].

Another existing solution for mitigating the effect of an adversary involves the use of two separate groups of sensors on the robot: the group of secured sensors, and the group of all the sensors on the robot, which includes both secured and compromised sensors. This mitigation method employs a barrier function approach mentioned in [13], which employs the framework for stochastic safety verification using barrier certificates presented in [14]. The approach is to select a control policy that minimizes the deviation of the robot's trajectory from the reference trajectory while ensuring that when the vulnerable sensor is compromised, the system remains inside the safe region, with or without the presence of an adversary. The system model being attacked by an adversary can be described by the following equations:

$$\begin{aligned} x(t) &= Ax(t) + Bu(t) + w(t) \\ y(t) &= Cx(t) + v(t) + a \end{aligned} \tag{12}$$

where w and v are processing noise and measurement noise, respectively. Variable a denotes the signal injected by the adversary. The optimal control input u can be found by solving the following problem:

$$\begin{aligned} &\underset{\mathrm{u(t)}}{\text{minimize}}\ E \int_0^T \left(x(t) - r(t)\right)^T Q\left(x(t) - r(t)\right) + u(t)^T Ru(t) \quad dt\Big] \\ &\mathrm{s.t} \qquad \|u(t) - u_\alpha(t)\|_2 \leq \gamma\ \forall t \in [0, T] \end{aligned} \tag{13}$$

which is equivalent to solving the following stochastic HJB equation:

$$
\begin{aligned}
0 = \min_{u \in U_\gamma(t)} \Big\{ \big(x(t) - r(t)\big)^T Q \big(x(t) - r(t)\big) \\
+ u(t)^T R u(t) + V_t(t,x) \\
+ V_x(t,x)\big(Ax(t) + Bu(t)\big) + \frac{1}{2} tr(V_{xx}(t,x)\Sigma_w
\end{aligned}
\tag{14}
$$

This equation is computationally very difficult to solve [13], so the relaxation of assuming the value function V is equal to the value function of the unconstrained problem was adopted, which will allow the removal of the constraint. Solving the minimizer of the unconstrained equation is equivalent to solving the following quadratically constrained quadratic program (QCQP):

$$
\begin{aligned}
&\underset{\mathrm{u}}{\text{minimize}} && u^T R u + x(t)^T P(t) B u + s(t)^T B u \\
&\text{s.t} && (u - u_\alpha)^T (u - u_\alpha) \le \gamma^2
\end{aligned}
\tag{15}
$$

in which u is the optimal control input, u_α is the control input generated by the secured sensors, and γ as the key design parameter. A high value of means that there is a wider range for control input u, which in turn will give a better control performance but also means that the attack can bias the system to an unsafe region. The team then used Proposition 1, first mentioned in [13], to calculate the value of and verify the safety criterion using sum-of-squares (SOS) optimization. The maximum value of that still ensures safety can be found using Algorithm 1, also proposed in [13]. This operation can be calculated using the SOS Toolbox in MATLAB.

Proposition 1: Suppose that there exist polynomials $\lambda_U(\bar{x}), \lambda_D(\bar{x}, \hat{u})$ and $D(\bar{x})$ such that the following hold:

$$D(\bar{x}) + \epsilon \ge 0 \tag{16}$$

$$D(\bar{x}) - 1\, \lambda_U^T(\bar{x}) g_U(\bar{x}) \ge 0 \tag{17}$$

$$D(\bar{x}) \ge 0 \tag{18}$$

$$
\begin{aligned}
-\frac{\delta D}{\delta \bar{x}}(f(\bar{x}) + \bar{B}\hat{u}) - \lambda_D^T(\bar{x}, \hat{u}) g_D^\gamma(\hat{u}) \\
-\frac{\delta D}{\delta \bar{t}} - \frac{1}{2} tr\left(\Lambda^T \frac{\delta^2}{\delta \bar{x}^2} \Lambda\right) \ge 0
\end{aligned}
\tag{19}
$$

$$\lambda_U(\bar{x}) \ge 0, \quad \lambda_D(\bar{x}, \hat{u}) \ge 0 \tag{20}$$

Then $Pr(\cup_{t \in [0,T]}\{x(t) \in U\}) \le \epsilon$

$$g_D^\gamma(\hat{u}) = \gamma^2 - \sum_{i=1}^{m} \hat{u}_i^2$$

Algorithm 1: Algorithm for computing the maximum parameter γ that ensures safety

```
1:  procedure SAFETY_SOS
2:    $\underline{\gamma} \leftarrow 0, \overline{\gamma} \leftarrow \gamma_{max}$
3:    while $\left|\underline{\gamma} - \overline{\gamma}\right| > \rho$ do
4:      $\gamma \leftarrow (\underline{\gamma} + \overline{\gamma})/2$
5:      $q$ ← SOS.Feasible(Eq. (16), Eq. (17), Eq. (18), Eq. (19), Eq. (20))
6:      if $q == 0$ then
7:        $\overline{\gamma} \leftarrow \gamma$
8:      else
9:        $\underline{\gamma} \leftarrow \gamma$
10:     end if
11:   end while
12:   return $\underline{\gamma}$
13: end procedure
```

3. Methodology

To gain an understanding of the existing work done on the robot platform, the existing MATLAB code was run on the Turtlebot3 platform. The current MATLAB implementation has a PID controller implementation and an LQG controller implementation to travel a “figure 8” shape that is described by a parametric function $f(t)$. Both implementations have an initial state that the robot is assumed to be in for the first iteration of the controller, and then updates the state using the robot’s position and trajectory to calculate the next desired point that the robot should move to. The robot’s required speed to get to that next point is then calculated and that information is sent to the robot to move to that point. Additionally, within these MATLAB implementations, the relation between the planned trajectory and the actual trajectory traveled is graphed in order to gain a better understanding of the accuracy of the state controller.

3.1 Choosing a New Platform

Using MATLAB made developing the controller easier because of MATLAB’s mathematical capabilities. The input, output, and states of the system can be described by a state-space equation which requires a lot of matrix manipulation and calculation, something that MATLAB excelled at. However, the ROS library for MATLAB is not very well-developed, making the communication between the robot and the host computer slow, inefficient, and not very modular. In contrast, Python has a much more developed ROS library, and a less robust mathematical capability than MATLAB. For the purpose of analyzing how data injection attacks would affect the reference tracking capability of the wheeled robot platform, performance is very important; which is why a port to Python was necessary.

The scope of this project was to take the existing MATLAB implementations and port them to a language that would be more efficient in running the controller. The current limitations of the MATLAB implementation result in an application that has a relatively slow runtime and is more complex to debug, and there was a desire to port this into a language that would be more efficiently compiled and easier to debug and test. On the Turtlebot3, there were two potential options for this porting process, C++ being run directly on the OpenCU, or Python being run on the Turtlebot3’s Raspberry Pi. Python was the solution that was ultimately chosen due to previous development experience in Python on the Turtlebot3 as well as the large number of pre-existing libraries that are developed for the Turtlebot3 platform in Python.

3.2 Porting to Python: basic movement, ROS, organization of nodes

Additionally, the Python implementation was done alongside the middle-ware platform Robot Operating System (ROS). ROS allows for code to run on the Turtlebot3 in independent instances called Nodes and allows for the transfer of data between nodes along Topics. This arrangement of code allows for independent functions to be developed and then applied in a variety of applications without having to redevelop additional code for each new application of the robot platform. Nodes are able to communicate with each other along topics, which are data streams over which a specific predetermined type of information can be shared amongst all nodes. This topic system relies on “publishers”, which send information out along a topic, and “subscribers”, which receives information along a topic. This system of inter-node communication allows for calculations to be done in one node, and the results of those calculations or functions to be sent to other nodes for additional functions.

Within the team’s ROS design, a node that represents the LQG or PID Controller was initialized, which was subscribed to the current state of the robot. The robot’s state was given by its position, as well as its linear and angular velocities. These values were constantly updated by the sensors on the robot, which then are input to the controller node. The controller node then takes in this information, which was then used in order to determine the set of velocity and direction commands that the robot will take in order to move to the next setpoint in the desired trajectory. This process was then repeated until the robot has traveled along the entire desired trajectory.

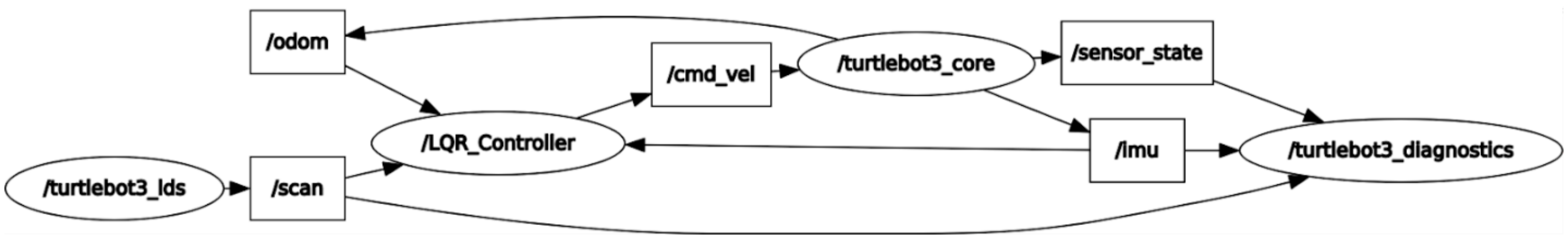


Figure 5: Node graph of the Python implementation of LQG controller

The whole process was very straightforward since ROS has a well-developed library for Python. A class called lqr_controller() was defined and initialized with all the necessary subscribers and publishers to different topics as well as all the variables and callback functions.

An object of the class was then initialized in the program's main function and the team can start sending and receiving messages to and from the topics.

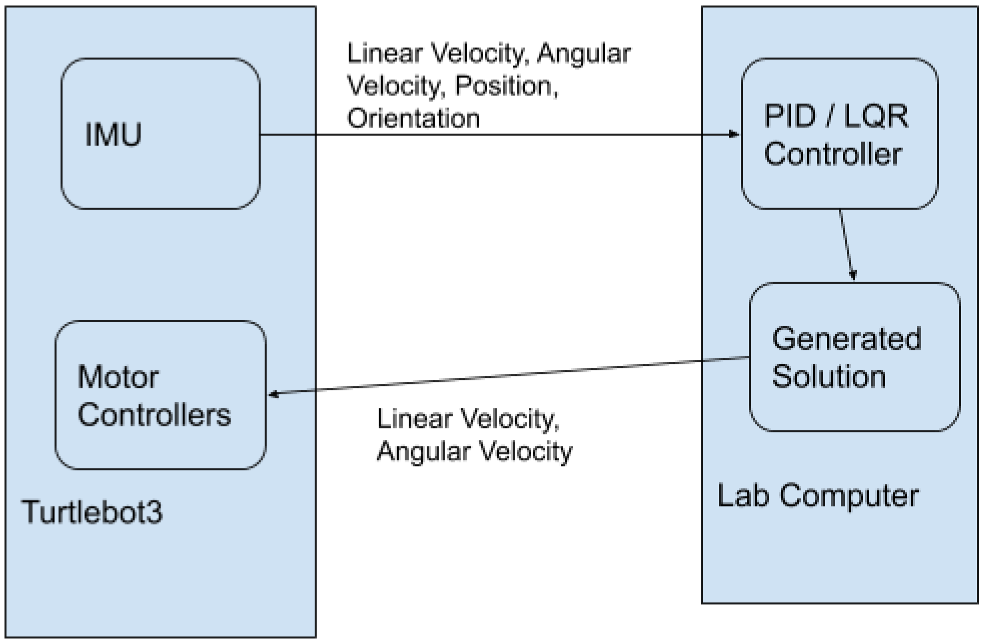

IMU
Linear Velocity, Angular Velocity, Position, Orientation
PID / LQR Controller
Generated Solution
Motor Controllers
Linear Velocity, Angular Velocity
Turtlebot3
Lab Computer

Figure 6: Block diagram of Python implementation

3.3 Porting to Python: PID Controller

The team began the porting process from MATLAB to Python with the PID controller, in order to investigate the difficulty of porting as well as any potential problems that the team might encounter before moving on to porting the LQG controller which is more complicated but also has better performance.

First, the team needed to find a way to port all the complex matrix mathematics of the controller, which are easily handled by MATLAB since MATLAB has a robust library of matrix manipulation functions. The team discovered that the numpy library in Python can provide a similar mathematical platform to MATLAB. The matrices from MATLAB could be represented as numpy arrays for manipulation and calculation. There are a few syntactical differences between using matrices in MATLAB and numpy arrays in Python, for example: matrices in MATLAB start at index 1 and numpy arrays start at index 0; matrix division is sometimes recommended in MATLAB instead of matrix inversion for speed, but this cannot be done in Python since in many cases it will result in division by zero errors. These syntactical differences were the main obstacles in the team's port to Python, some of which took the team a few days to find and debug. Another thing the team had to be concerned about was to confirm that the results of all the calculations are correct. This was achieved by printing out each of the results in the Python implementation and compare them to their counterparts in the MATLAB implementation.

The second important thing to implement is a way to keep a constant time between each time step. For each timestep length dt, the Turtlebot will carry out its movement according to the velocity message that ROS received. This is done in MATLAB by using tic and toc to start and stop the timer respectively. At first, the team simply stopped the code from running for dt seconds by using sleep(), which works, but was not ideal because this method did not keep consistent time and it is bad practice to simply stop the code from running. The team eventually used the function time.time() in Python's time library to start and stop a built-in timer. This provided an accurate time measurement as well as not requiring the code to stop running.

Finally, the team needed to find a way to graph the actual trajectory versus the reference trajectory and the mean square error in order to visualize the performance of the controller with different parameters. This was achieved with the matplotlib library, a plotting library created for Python. The matplotlib library was simple to use and very modular, allowing the team to modify all the elements in the graphs and the team was able to plot both the actual trajectory versus reference trajectory graph, and the mean square error graphs accurately.

3.4 Porting to Python: LQG controller

The port of the LQG controller from the MATLAB implementation to the Python implementation turned out to be very similar to that of the PID controller. The team was able to use the same techniques and libraries that were used in the port of the PID controller: numpy library was used to handle matrix manipulation and calculation, time library was used to keep constant time between each timestep, and matplotlib was used to plot the actual trajectory versus reference trajectory graph and mean square error.

There was a big error that required the team some time to fix. Inside the LQG loop, there was a wrong calculation of Theta which make all the other results diverge into infinity and make the system not BIBO stable. Because Python handles matrix calculations differently from MATLAB, the team had to make some changes to split a long calculation into smaller calculations and test the result every time step in order to make sure that the outputs are bounded correctly, and the system is stable. After this change, the robot was able to carry out the reference tracking with very good performance.

Another small bug that needed to be fixed was regarding the getKey() function. This function was used in order to stop the robot from running when the key “e” was pressed on the keyboard, which was important for the debugging process but irrelevant in the final outcome of the reference tracking. The function works by waiting 0.1 second for user input each time it is called. This function must then be called every time step in order for the user input to be read properly and stop the robot from running. However, for a very small value of dt (< 0.1 second), this 0.1 second delay yielded a very big decrease in performance, slowing down the robot significantly. After the controller was successfully tested and verified to be working properly, this function was removed to avoid any negative effect on performance.

3.5 Attack Simulation and Mitigation Methods

Initially, the team chose the y-position IMU sensor as the vulnerable sensor and the x-position IMU sensor and the LIDAR as the secured sensors. However, LIDAR sensors are very vulnerable in reality; also, the team was not able to produce reliable y-position data out of the LIDAR. Therefore, the team ended up using the LIDAR as the unsecured sensor.

For the mitigation method the team used a barrier function approach mentioned in section 2.5. The control input $\boldsymbol{u}$ can be found by solving the QCQP (16) The value of was calculated using MATLAB’s SOS Toolbox, and the team found that the optimum value is $\gamma = 9.626$.

In MATLAB, the QCQP was solved by setting up the objective equation with the quadobj() function and setting up the constraint equation with the quadconstr() function, and inputting the two equations into the fmincon() function to get the minimizer output. In Python, the team was able to use the cvxpy library, developed by the Convex Optimization Group at Stanford University, California. This library allows for simple solving of convex optimization problems using built-in functions. The library chooses the appropriate solver depends on the type of problem and then rewriting the problem in the selected solver standard form [15]. In this implementation, the team used the splitting conic solver (SCS) to solve the convex QCQP. There was a slight difficulty with setting up the problem: since the Variable object created by the cvxpy library is not a numpy array and does not have a matrix size, there were some issues with numpy trying to multiply matrices with incompatible sizes. This was solved by using the overloaded multiplication operator that was defined inside the library to multiply a Variable object with a normal numpy matrix. The QCQP solver was then tested in a trajectory simulation and the output was compared with the MATLAB version of the simulation, and yielded similar results, as shown in the figure below.

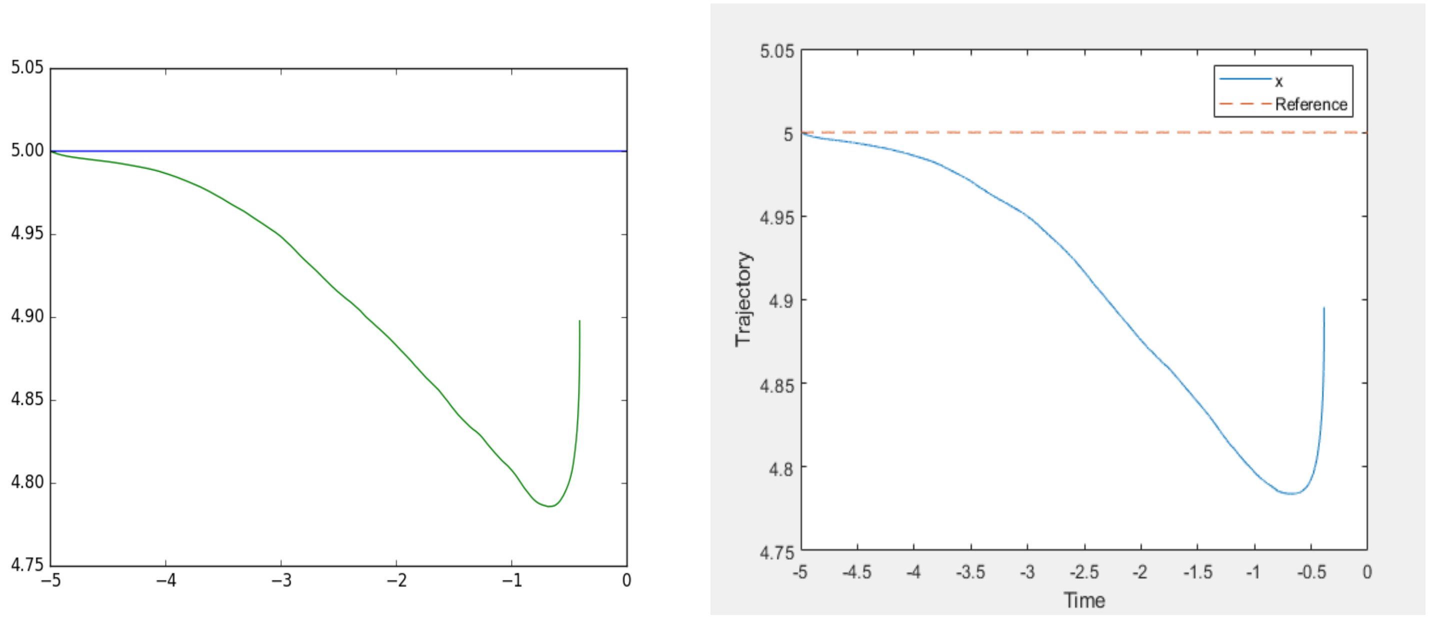


Figure 7: Simulation results of MATLAB version (left) and Python version (right)

In order to obtain a secondary set of sensors, the LIDAR on the robot was used as an unsecured sensor, while the internal IMU of the robot is used as the secured sensor. This simulated a situation in which the LIDAR on the robot is subjected to a spoofing attack. The LIDAR on the robot can be used to generate a map of the environment in which it is operating. This map was generated using the ROS GMapping package, which was an implementation of simultaneous localization and mapping (SLAM) within ROS, specifically the open-source OpenSLAM package [16]. This GMapping package takes the laser scan information from the robot's LIDAR and generates an occupancy grid of the robot, which is a 2D map of the environment in which the robot is operating [17]. The occupancy grid is then generated with each cell having a specific probability as to whether an object such as a wall or obstacle is present. This grid contains two types of grid cells, unoccupied cells which are likely to not have an object in that cell, and occupied cells which are likely to have an object in that cell. Using this GMapping package, an occupancy grid was generated of the Secure Cyber Physical Systems Lab at WPI:

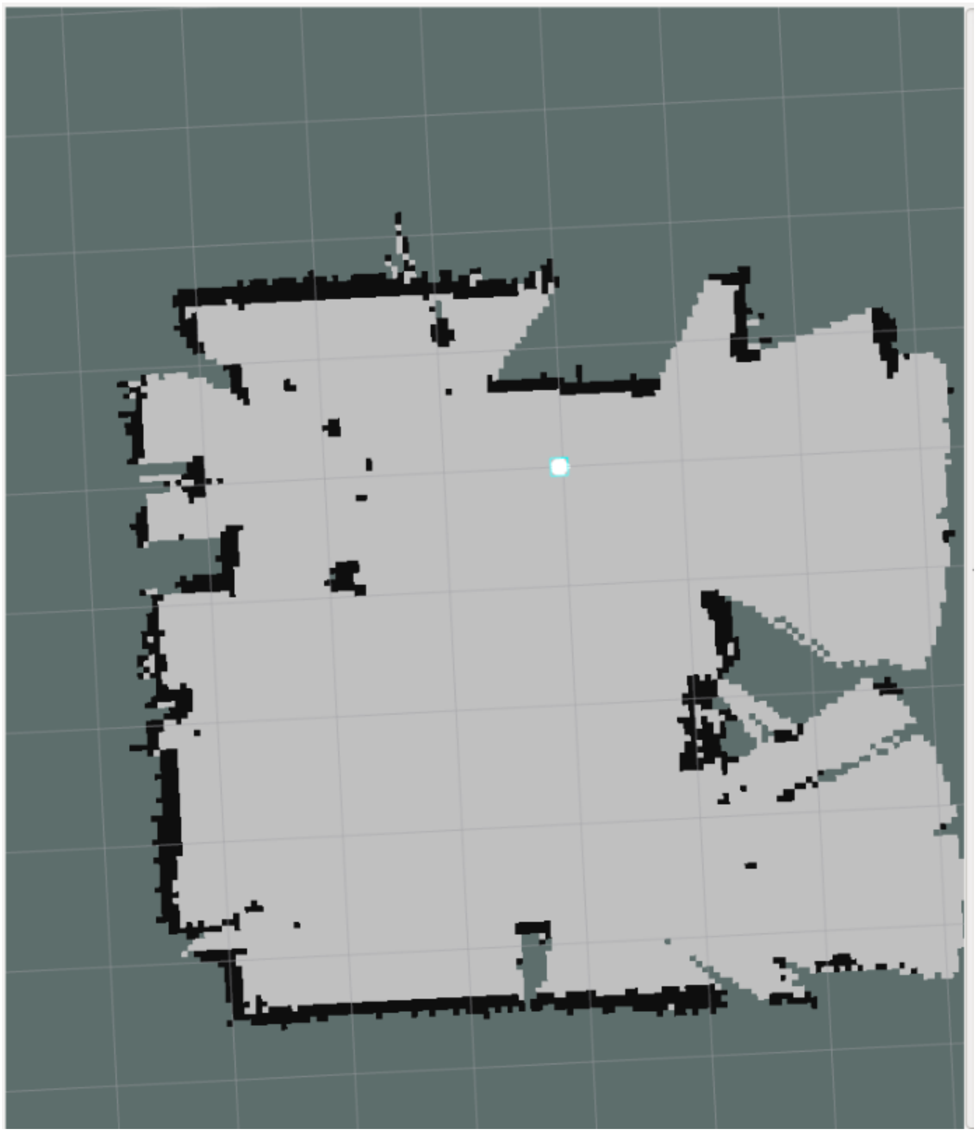


Figure 8: Map generated by the Turtlebot's LIDAR

In order to be used as a secondary sensor for the mitigation method, the GMapping package must localize the robot, or determine the robot's position within the map at any given time. The GMapping package provides a method to localize the robot via the transformation, or "/tf" library. The tf library allows for ROS to keep track of multiple reference frames at any given time, as well as the translation and rotations necessary to move between two given reference frames. The GMapping package uses the tf library for several transformations, and from these transformations, the robot's position within the map can be generated. The specific transformation of interest if that of /map -> /odom, which can generate the robot's estimated position within the map. With this transformation known, the robot's position within the map is known, and then can be used within the controller as a secondary position input. This transformation is broadcasted by the GMapping package along the /tf topic, so a specific ROS node called a transform listener is needed. This tf_listener node takes the transformation along the topic and prints out the robot's estimated position within the map, which can then be published to other nodes. In this implementation, the tf_listener node publishes this estimated position to the LQG Controller node along the /linear_trans topic, which then can be used as the unsecured input X and Y coordinates in the control loop. The ROS lqt_graph, which describes all the nodes using during a given operation, as well as the topics they are publishing and subscribed to, gives a detailed description of this setup:

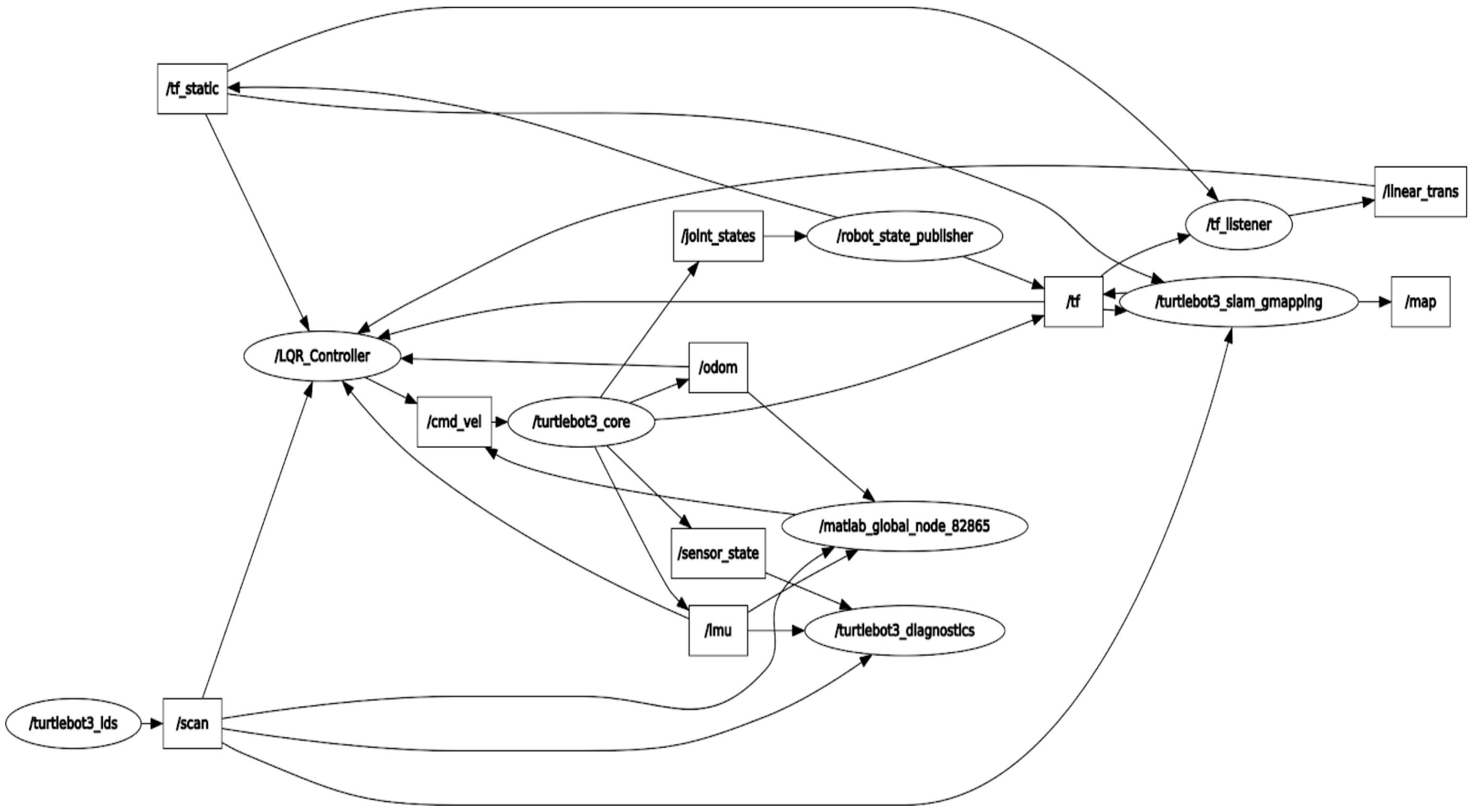


Figure 9: Full node graph of the Python implementation of the mitigation method

4. Testing and Results

4.1 PID Controller

The team's implementation of the PID controller for the autonomous path planning was not successful, even without the presence of the simulated attack. The performance of the PID controller was inaccurate and unstable; although the Turtlebot seemed to follow a rough trajectory that resembled the reference trajectory, the result was not satisfactory. The testing of the original MATLAB version also yielded the same performance issues.

When the controller and the code are initiated, the Turtlebot tries to follow a very rough trajectory that resembled the reference trajectory; however, the discrepancy between the Turtlebot's trajectory and the original trajectory, as shown by the reference trajectory versus actual trajectory graph and the mean square error versus step number graph (Figure 10), was too large for this version of controller to be used in the project to run the autonomous path planning for the robot.

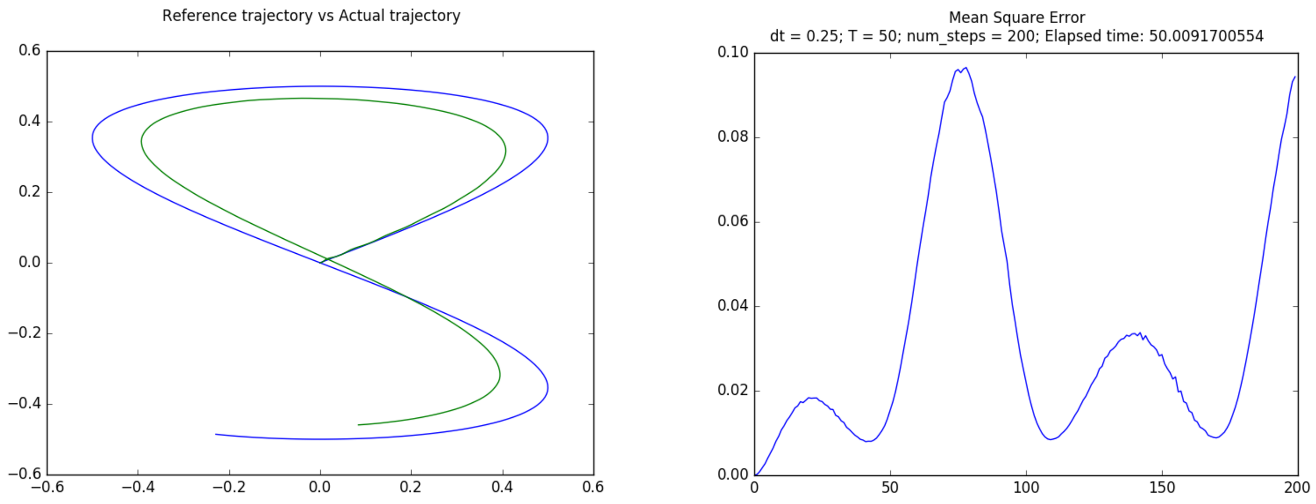


Figure 10: Actual trajectory vs Reference trajectory graph and Mean Square Error graph of PID controller

The team made a few small improvements on the performance, but the results were still not good enough for reference tracking purposes. Later on, when it was determined that the LQG controller version performed much better in every aspect, the team stopped working on the PID controller.

4.2 LQG Controller

The performance of the LQG controller was a lot more accurate and stable than that of the PID controller, which was also reflected in the original MATLAB version. When the controller and the code were initiated, the Turtlebot did a very good job with following the reference trajectory almost exactly with very low variance and error (e). The amount of time allotted for the autonomous path planning (T) and the number of steps (num_steps) the Turtlebot take can be modified to yield different results.

The team conducted more tests in order to determine the ideal values of T and num_steps, as well as dt that would optimize the Turtlebot's trajectory, minimize error, and keep runtime low enough for testing. As shown in (Figure 11), there existed three main error areas that are consistent for all T and num_steps values.

- The first error zone was at the beginning of the trajectory. The robot overshot outside of the reference trajectory then slowly return to the planned path.
- The second error zone appeared just before the midway point of the trajectory. Visually, the robot could be seen to oscillate, going back and forth constantly for a few steps, then returned to the planned path.
- The third error zone was at the end of the trajectory. The team suspected that this error is caused by the robot slowly coming to a stop slightly before the reference trajectory was supposed to end.

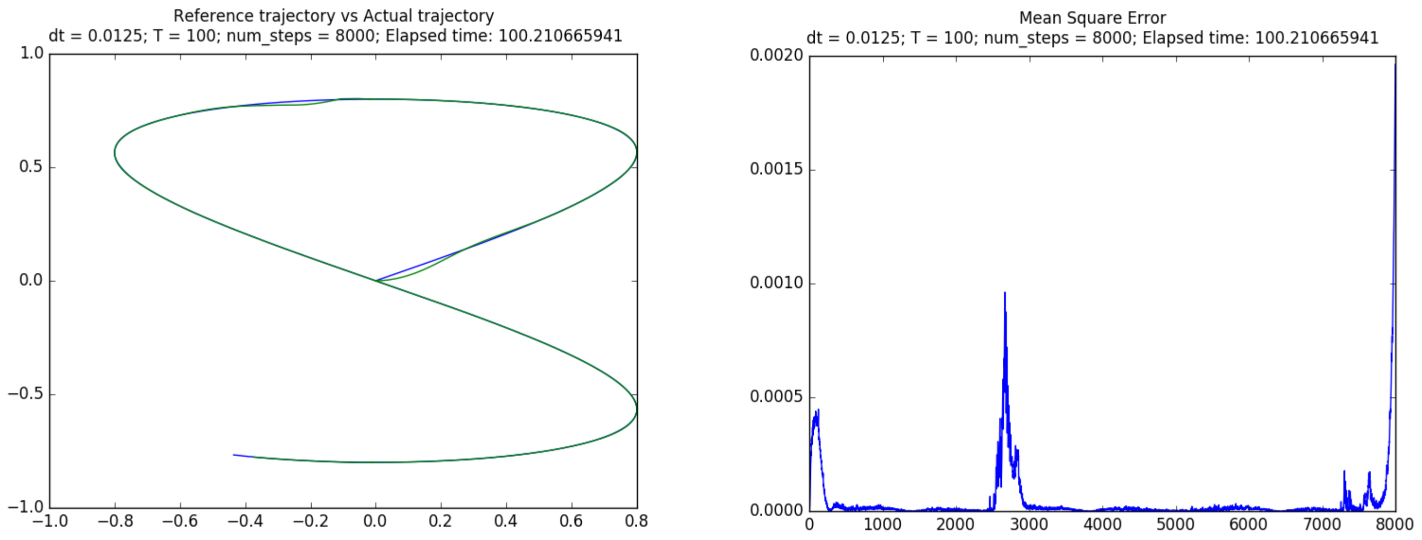


Figure 11: Three error zones, as seen on trajectory graph and mean square error graph

The team also noticed a pattern in the Turtlebot's trajectory with different values of T and num_steps. Lowering the value of T caused the robot to run off course and amplify the length of the initial overshoot before the Turtlebot come back to its planned trajectory. Any T value lower than 50 (seconds) would cause the robot to run a shrunken version of the reference trajectory with a very long overshoot, which was undesirable. The length of the initial overshoot, as well as the error between the reference trajectory and the actual trajectory, were reduced with increasing value of T; however, this will cause the robot to take a longer time to run the whole trajectory. The team noticed that for values higher than T = 150, the average error (not including the three error zones mentioned above) between the actual trajectory and the reference trajectory became very low (< 0.00002). The errors in the problem areas mentioned above also became very small, with e = 0.0002 for the initial overshoot and e = 0.0001 for the second error zone (Figure 12).

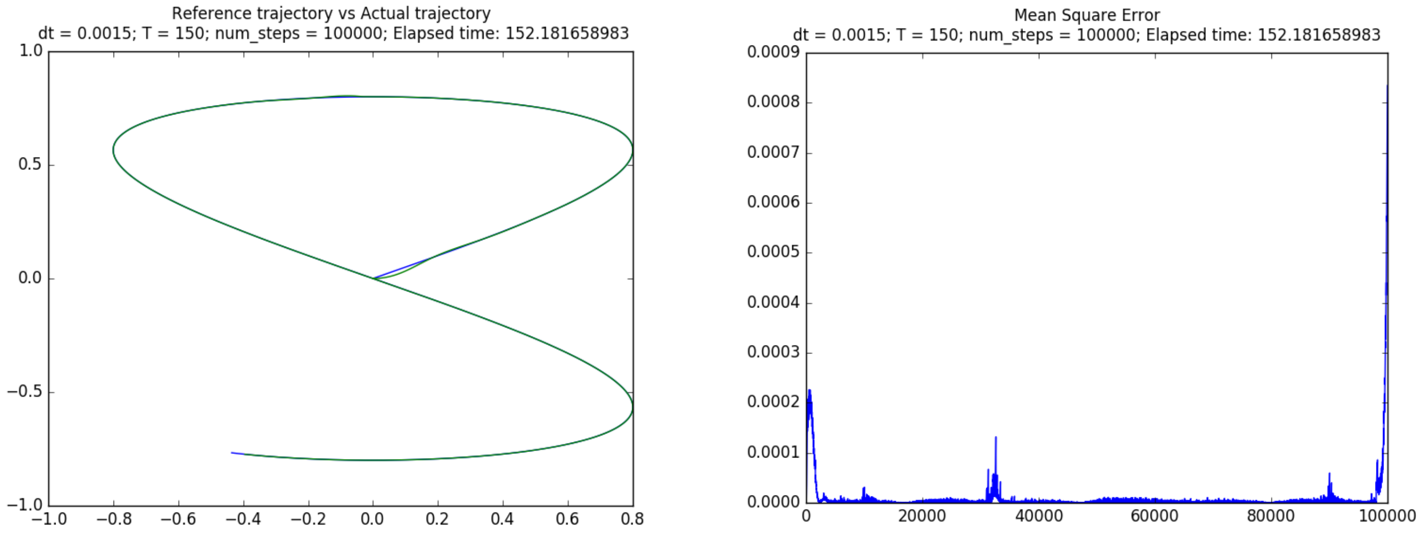


Figure 12: Trajectory and mean square error graph with T = 150

At T = 300, the error in the second zone is nearly indistinguishable from the general noise of the control system (Figure 13). Increasing T afterward yields insignificant reduction to the errors. For best performance, the team would recommend using T = 300; for general testing and attack simulation, the team would recommend using T = 150.

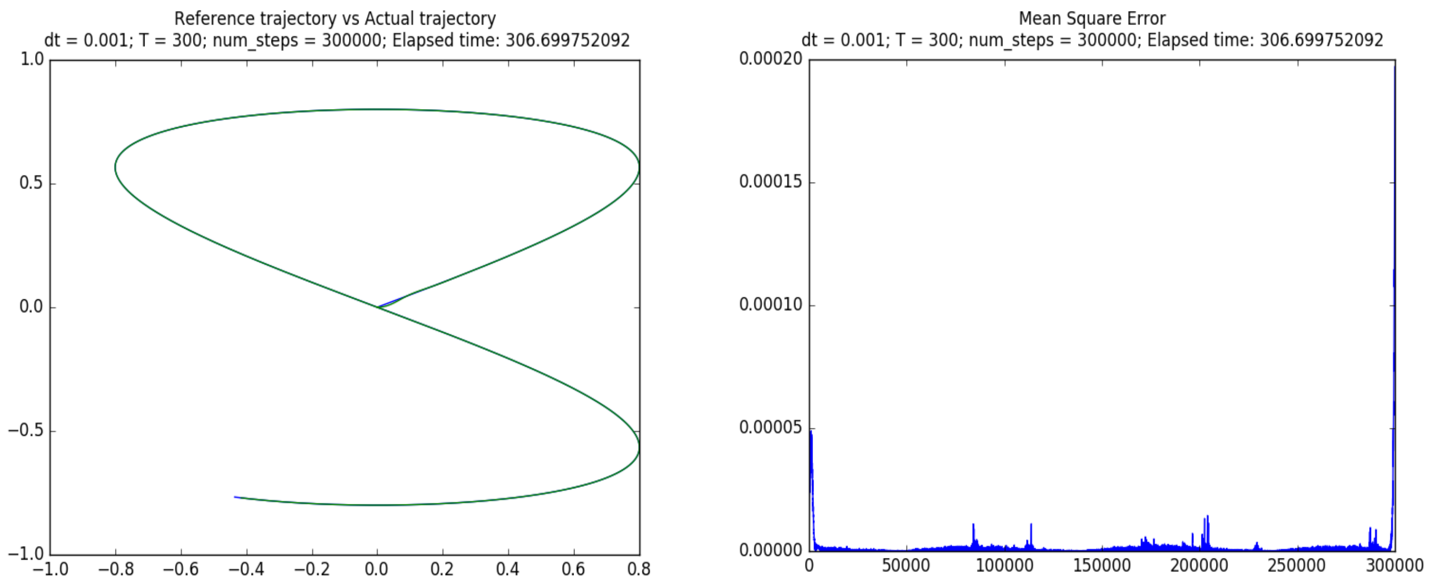


Figure 13: Trajectory and mean square error graph with T = 300

The team also experimented with different values of num_steps, which would also change the value of dt accordingly, as dt = T/num_steps. The higher the value of num_steps, the lower dt gets and the smaller the length of the error zones will become. The team has found no performance issues with raising the value of num_steps, as long as the value of dt is kept larger than the time the computer needs to perform all the calculations, about 0.0002s. If the value of dt is lower than the time the computer needs to perform the calculations in the loop, the Turtlebot will experience unstable behaviors as some of the calculations will be lost and override before getting sent from the host computer to the Turtlebot. In order to make ample room for the computer to calculate each loop and the data to be sent to the Turtlebot, the team suggests setting the num_steps value so that dt value equals to 0.001 for the best performance. Due to how accurate the Turtlebot's trajectory was compared to the reference trajectory, the team decided that this version of the LQG controller is sufficient to control the robot and move on to attack simulation on the Turtlebot's sensor.

4.3 Attack Simulation

Once the robot was able to travel the given planned trajectory with good performance, the attack simulation was then considered on the robot. For the first attack, the team simulated a simple random noise injection into the y-position sensor of the robot.

Within the simulation, an attacker was attempting to send false data to the robot's sensors which results in inaccurate tracking of the robot's position. This was modeled within the code application as a random number within the bound $[a, b]$ that was added or subtracted to the measured value of the robot's y position. This resulted in an inaccurate measurement for the current position of the robot which is then fed into the feedback of the LQG controller. Injecting false data at every time step caused the robot to behave very erratically since the false data would cascade and the LQG controller did not have enough time to stabilize the system. The resulting reference tracking attempt of the robot can be seen in Figure 14.

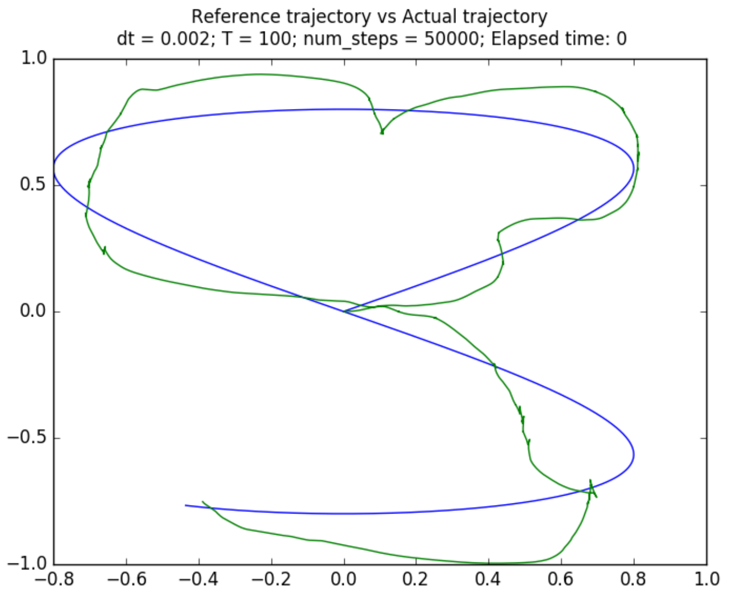


Figure 14: LQG reference tracking with attack on y-position IMU sensor

In order to test the ability of the LQG controller to self-stabilize the system in the case of a weaker attack, instead of injecting false data every time step, the attacker injected a larger false data every 20 time steps. The LQG controller did a good job of stabilizing the robot platform in this case, with a reasonably small deviation from the reference trajectory, as seen in Figure 15

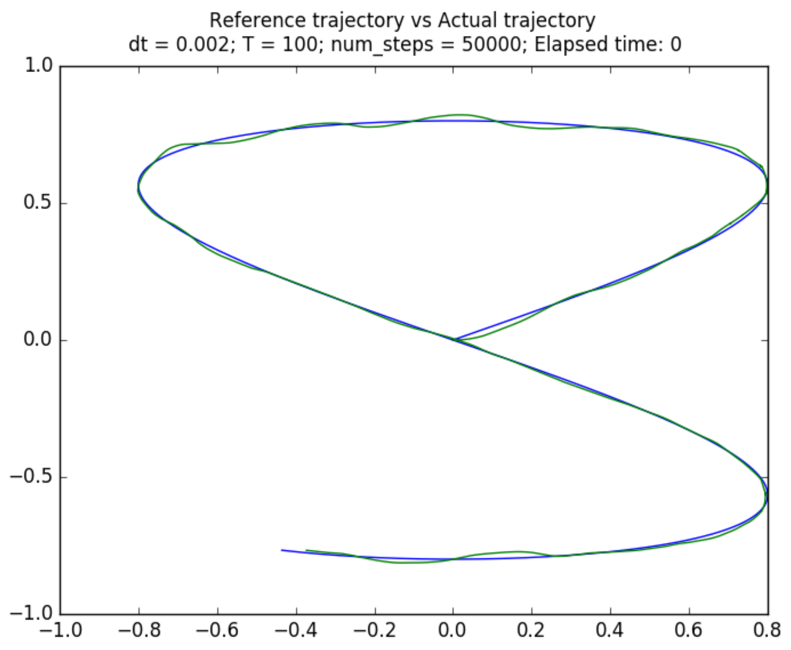


Figure 15: LQG reference tracking with attack on y-position IMU sensor every 20 time steps

Later on, the team tested another attack, this time on the y-position of the robot received by the LIDAR. The Turtlebot's LIDAR would do an initial scan of its surroundings to get an occupancy grid and determine its initial position, then compare its current position to the initial position to generate y-position measurement. Random Gaussian noise was then added to this reading and fed into the current position matrix of the robot.

4.4 Mitigation Methods

The team decided to select the LIDAR as the unsecured sensor, since in practical applications of autonomous driving and reference tracking, LIDAR is the most easily and commonly compromised sensor [18], while the IMU sensors are very secured in general. Furthermore, the y-position received from the LIDAR, due to calculation errors, transmission delay from the robot to the transform listener to the controller, and the inherent inconsistency of the occupancy grid being generated each time, is not as accurate as the position received by the IMU sensors.

With this simulated attack on the robot's sensors in place, the mitigation method was then implemented in order to properly retain control of the system as the robot completes the trajectory. As previously introduced, the mitigation method relies on two sets of sensors, the set of secured sensors, and the set of all sensors used in order to determine the robot's position. The barrier function, with the set of possible control input limited by the design parameter γ, prevents the robot from reaching the unsafe region. The team then calculated the optimal control input by solving the QCQP mentioned in section 2.5. Within the attack simulation, the secured sensor was the robot's IMU, and the unsecured sensor is the LIDAR on the robot, with the reported position of the robot from the GMapping transformation modified by random Gaussian noise. This modified position value was then fed into the mitigation method calculation as the unsecured input, with the IMU position value used as the secured input.

The results were satisfactory: the robot was able to stay very close to the reference trajectory with the mitigation method implemented, as shown in figure 16. The team was able to get dt value to as low as 0.02 with no issues with instability, essentially turning the system into a continuous-time one. This was due to the greatly improved performance of the QCQP solver in the Python implementation of the mitigation method compared to the MATLAB version.

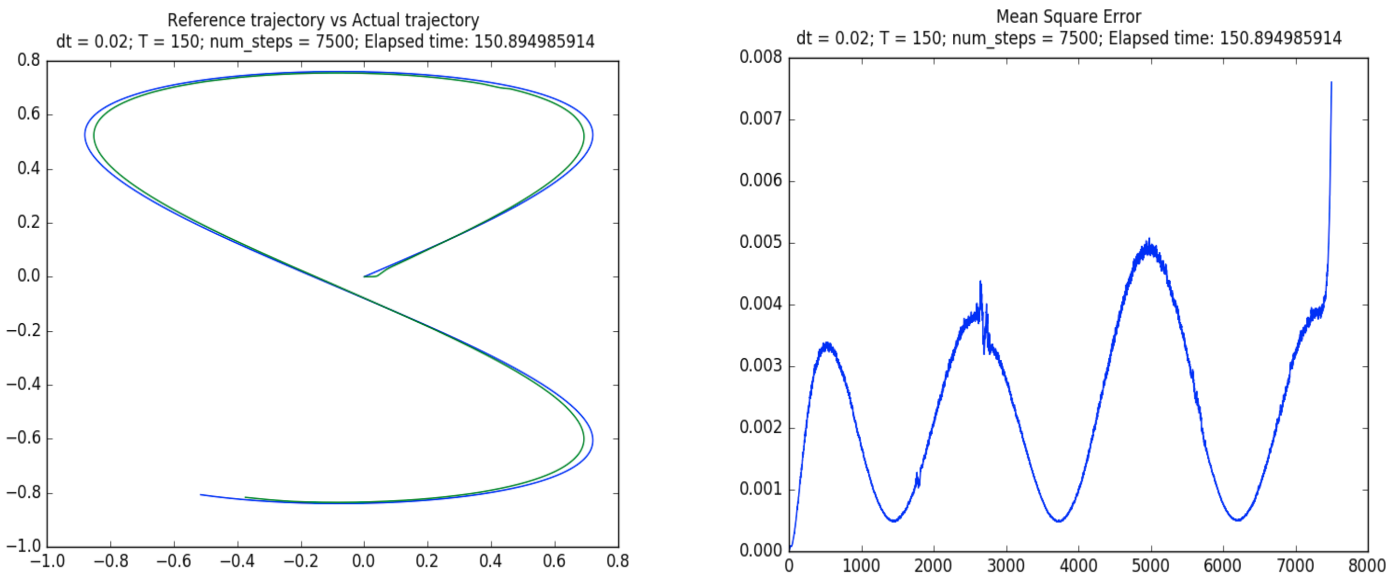


Figure 16: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 150, dt = 0.02, random Gaussian noise)

The team was also able to get dt value to be as low as 0.01, by setting T = 100 and num_steps = 10000, and receive a good result from the controller; however, while debugging, the team noticed that the elapsed time between each time step can become higher than 0.01 second due to calculation time for solving the QCQP. This can be seen in figure 17; note the elapsed runtime is 108 seconds, exceeding T value by 8 seconds, which might cause the robot's movement to be inaccurate. Since the control input calculations take around 0.01 second to complete, the team recommend that dt value should not be set to less than 0.02.

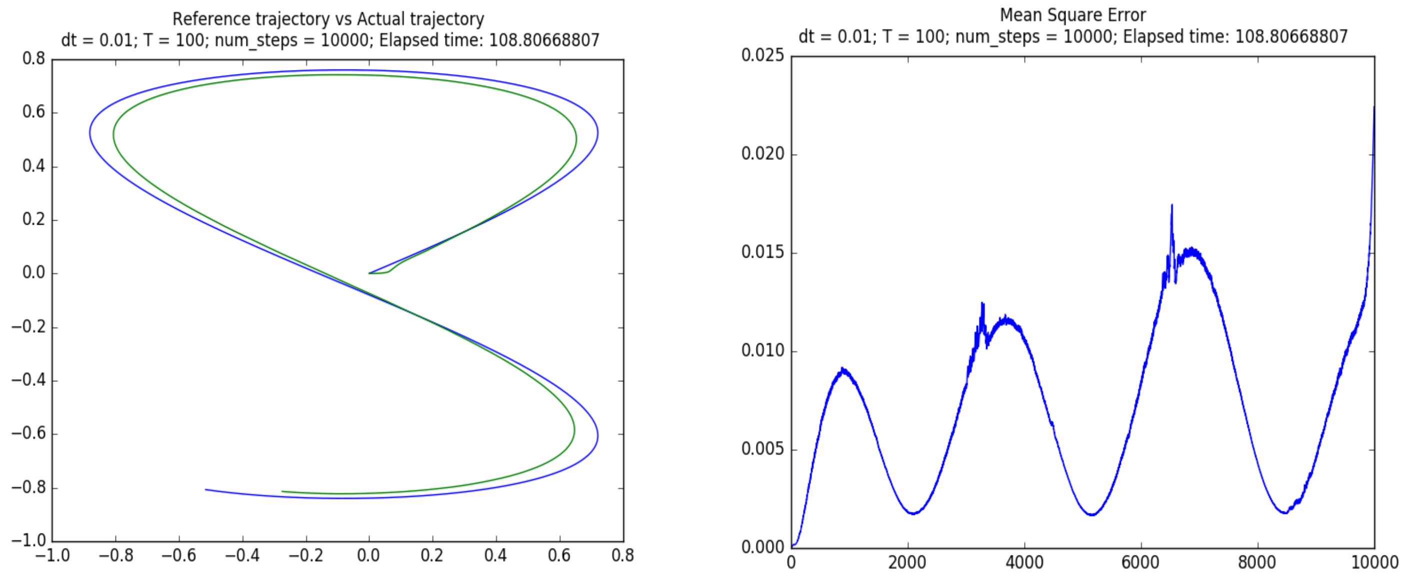
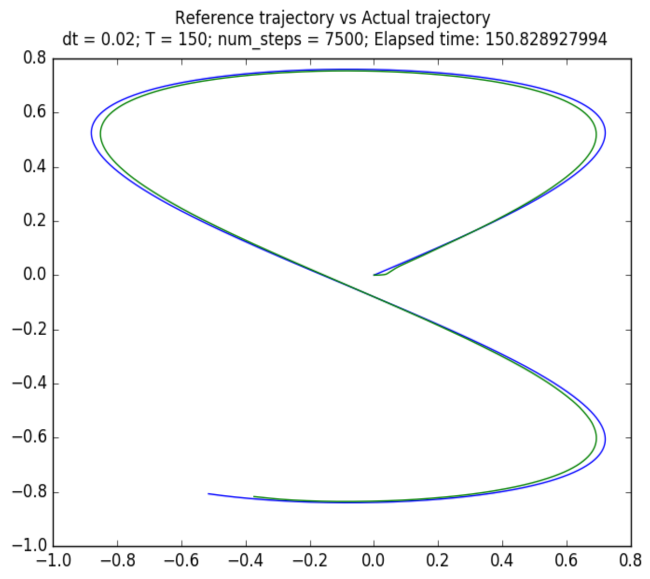


Figure 17: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 100, dt = 0.01, random Gaussian noise)

Another test of the mitigation method was done with a different attack, by adding a constant to the y position generated from the LIDAR. The results were very similar to the mitigation method against the random Gaussian noise attack, proving that the mitigation method works well regardless of the false data injection attack performed on the y position of the LIDAR.



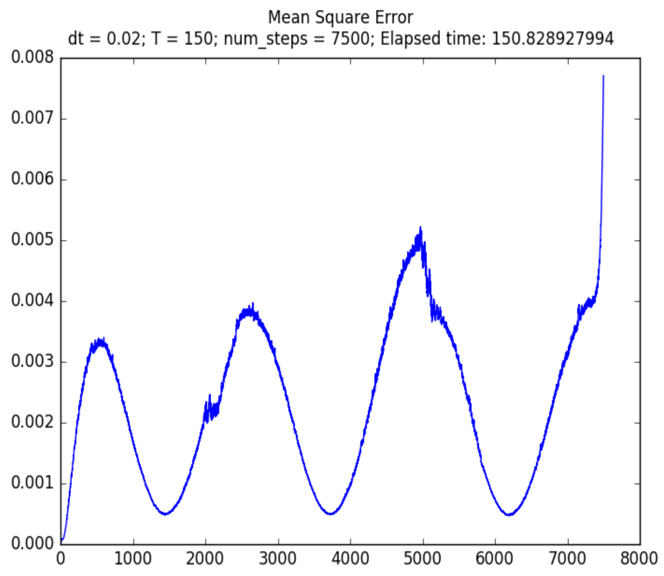


Figure 18: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 150, dt = 0.02, constant attack)

The team also experimented with different values of T, as high as T = 300 and as low as T = 50 while keeping dt at 0.02. At T = 300, the accuracy at which the Turtlebot can follow the reference trajectory under attack was greatly improved (Figure 19). However, this came at a cost of increasing the runtime to 300 seconds which is not practical for testing purposes. Meanwhile, at T = 50, the actual trajectory appeared to be shrunk when compared to the reference trajectory and some instability was observed (Figure 20).

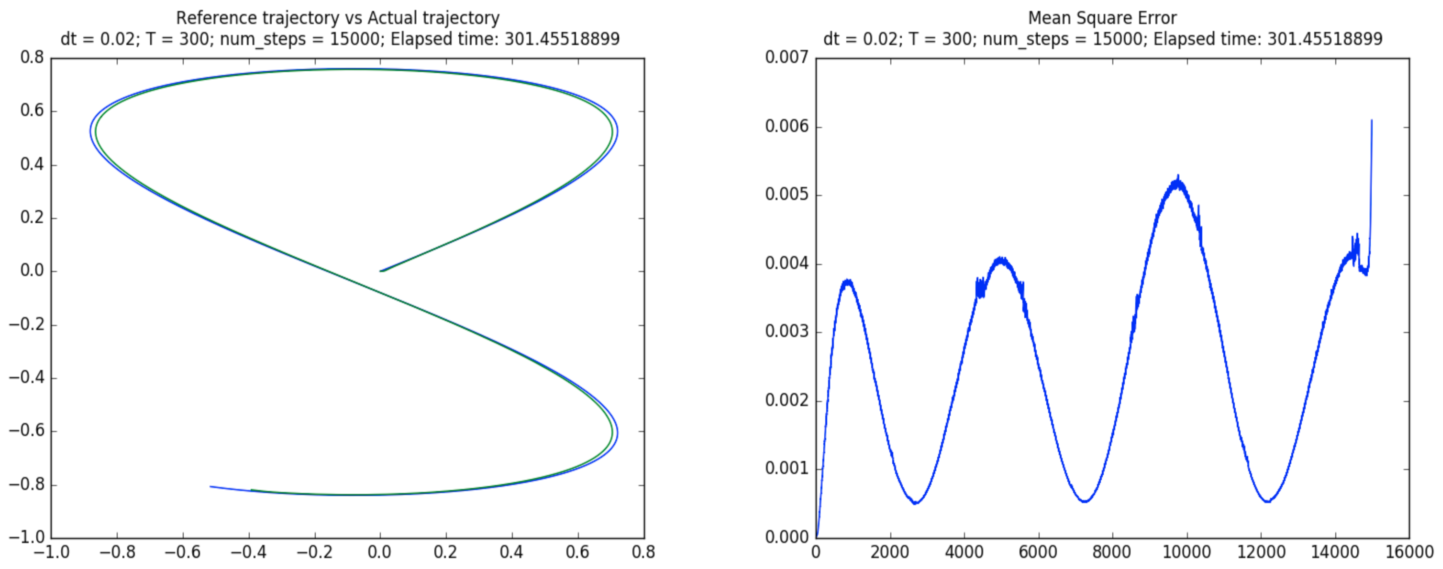


Figure 19: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 300, dt = 0.02, random Gaussian noise)

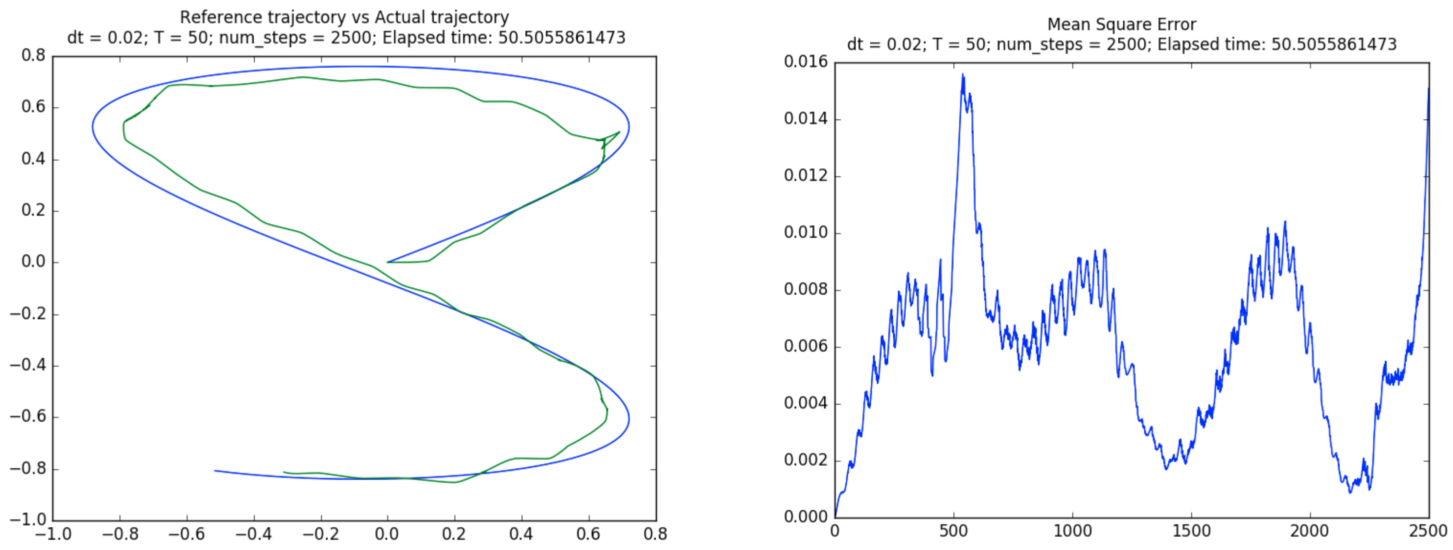


Figure 20: Reference trajectory vs Actual trajectory graph and mean square error graph of the Python implementation of the mitigation method (T = 50, dt = 0.02, random Gaussian noise)

Similar behavior was observed during the testing of the Python implementation of the LQG controller, so the effects of runtime on the robot's accuracy for reference trajectory tracking was expected for the Python implementation of the mitigation method. Therefore, for best performance, the team would recommend using T = 300; for general testing, the team would recommend using T = 150, modifying number of timesteps proportionally to keep dt value at a constant 0.02. Overall, the mitigation method does a good job of preventing the system from deviating too far from the reference trajectory.

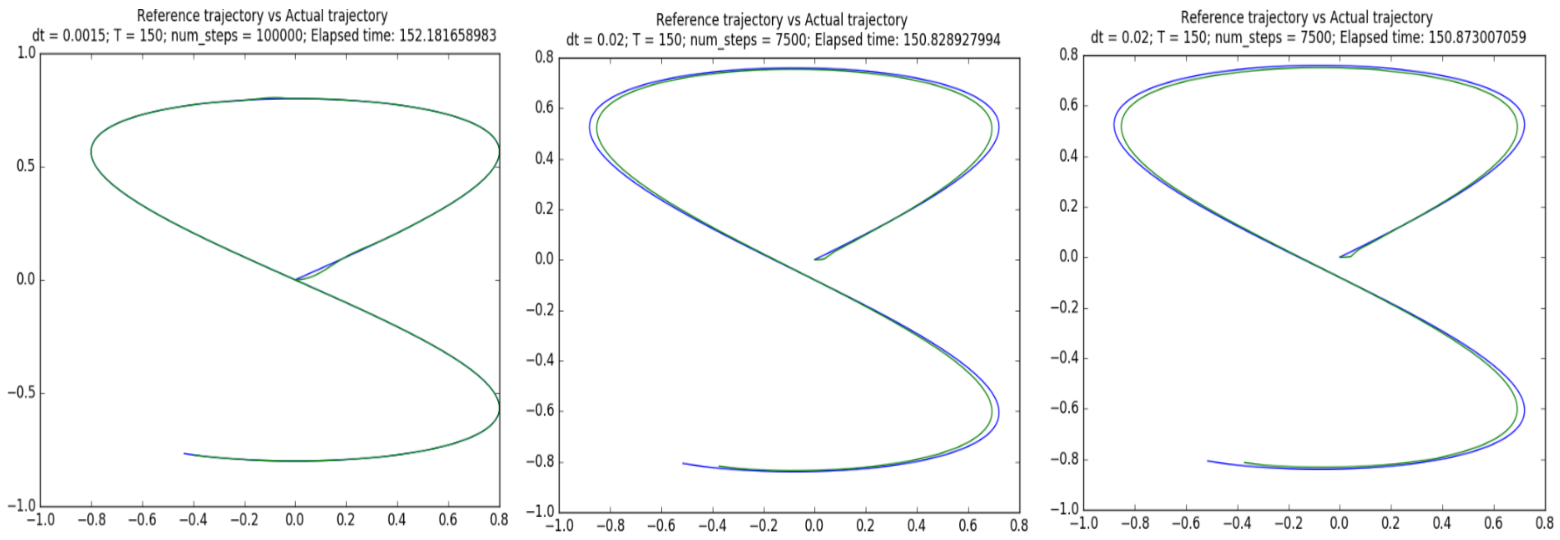


Figure 21: From left to right: LQG controller with no attack, LQG with mitigation method against random Gaussian attack, LQG with mitigation method against constant attack

5. Conclusion

5.1. Summary

The project was successful in implementing a Python version of an LQG controller for the purpose of autonomous path planning and mitigation against false data injection attacks on a compromised sensor by using other known secured sensors. The Python version was improved and had a huge performance improvement over its MATLAB counterpart, allowing dt to be as low as 0.001 in the LQG controller implementation and as low as 0.02 in the mitigation method implementation, making the system essentially continuous-time. This is due to the very well-developed ROS library that is available on Python, as well as the speed of calculations of the two mathematical libraries, cvxpy and numpy. Both factors contributed to allowing the control of the robot and the calculations of the control input to be carried out very quickly.

5.2. Future Works & Improvements

The Python implementation of the LQG controller and the mitigation method works very well; however, as of right now, the programs are required to be set up on a host computer with all the dependencies installed. Putting the code onto the Raspberry Pi 3 on the Turtlebot and install the necessary dependencies there will allow any host computer to run the reference tracking on the Turtlebot just by connecting to the Raspberry Pi. During the process of trying to port the code and running it on the Raspberry Pi, the team ran into a few difficulties. Firstly, there is no official documentation for this procedure for the Turtlebot, and most ROS applications the team researched are using a host computer to control the Turtlebot. Secondly, since the Raspberry Pi is much less powerful than a PC, it will take the complex matrix multiplication and other calculations much longer to carry out, which will impact dt and performance. However, if the code is able to be installed and run on the Turtlebot as an embedded system, it will provide additional flexibility and autonomy to the system by not requiring a host computer to which the robot is tethered to.

6. References

[1] E. Z. Macarthur, D. Macarthur, and C. Crane, “Use of cooperative unmanned air and ground vehicles for detection and disposal of mines,” *Intelligent Systems in Design and Manufacturing VI*, Sep. 2005.

[2] E. Coelingh and J. Ekmark, “Self-driving Vehicles Will Revolutionize the Transportation System,” *ATZelectronics worldwide*, vol. 14, no. 9, pp. 16–21, 2019.

[3] D. Davidson, H. Wu, R. Jellinek, T. Ristenpart, and V. Singh, “Controlling UAVs with Sensor Input Spoofing Attacks,” *Proceedings of the 10th USENIX Conference on Offensive Technologies*, pp. 221–231, 2016.

[4] A. Greenberg, “Hackers Remotely Kill a Jeep on the Highway—With Me in It,” *Wired*, Jul. 2015.

[5] Open Source Robotics Foundation.“TurtleBot,” TurtleBot. [Online]. Available: https://www.turtlebot.com/.

[6] G. Oriolo, A. D. Luca, and M. Vendittelli, “WMR control via dynamic feedback linearization: design, implementation, and experimental validation,” IEEE Transactions on Control Systems Technology, vol. 10, no. 6, pp. 835–852, 2002.

[7] Y. Li, Y.-Q. Jiang, L.-F. Wang, J. Cao, and G.-C. Zhang, “Intelligent PID guidance control for AUV path tracking,” *Journal of Central South University*, vol. 22, no. 9, pp. 3440–3449, 2015.

[8] X. Ge, F. Yang, and Q.-L. Han, “Distributed networked control systems: A brief overview,” Information Sciences, vol. 380, pp. 117–131, 2017.

[9] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, “Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving,” *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS 19*, 2019.

[10] Y. Guan and X. Ge, “Distributed Attack Detection and Secure Estimation of Networked Cyber-Physical Systems Against False Data Injection Attacks and Jamming Attacks,” *IEEE Transactions on Signal and Information Processing over Networks*, vol. 4, no. 1, pp. 48–59, 2018.

[11] X. Yang, J. Lin, P. Moulema, W. Yu, X. Fu, and W. Zhao, “A Novel En-route Filtering Scheme against False Data Injection Attacks in Cyber-Physical Networked Systems,” *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 2012.

[12] A. Chattopadhyay and U. Mitra, “Security against false data injection attack in cyber-physical systems,” *IEEE Transactions on Control of Network Systems*, pp. 1–1, 2019.

[13] L. Niu, Z. Li, and A. Clark, “LQG Reference Tracking with Safety and Reachability Guarantees under False Data Injection Attacks,” 2019 American Control Conference (ACC), 2019.

[14] S. Prajna, A. Jadbabaie, and G. Pappas, “Stochastic safety verification using barrier certificates,” *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*, 2004.

[15] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd, “A rewriting system for convex optimization problems,” *Journal of Control and Decision*, vol. 5, no. 1, pp. 42–60, Feb. 2018.

[16] G. Grisetti; C. Stachniss; W. Burgard, “GMapping”, OpenSLAM [Online]. Available: https://openslam-org.github.io/gmapping.html. [Accessed 06-Dec-2019]

[17] B. Gerkey, Open Source Robotics Foundation. “GMapping,” ros.org. [Online]. Available: http://wiki.ros.org/gmapping. [Accessed: 06-Dec-2019].

[18] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, “Adversarial Sensor Attack on LiDAR-based Perception in Autonomous Driving,” Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS 19, 2019.

7. Appendix

Appendix A: How to Use the Python Implementation

Getting Started

These instructions will get you a copy of the project running on a local machine, in which the robot attempts to autonomously drive a "figure eight" trajectory while there is a simulated attack on the robot's LIDAR sensor, injecting it with false position data. The software uses a mitigation method in order to reduce the effect of the simulated attack and allows for the robot to travel the pre-planned trajectory with little disruption to desired operation. The complete code repository can be accessed at https://github.com/hoangminh2408/PathPlanningMQP2019

Prerequisites

This project runs code on a remote computer using Ubuntu Linux 16.04 with ROS Kinetic Installed. This remote computer communicates to a Turtlebot3 Burger robot via Wi-Fi connection.

Installing

First, in order to run the simulation, ROS Kinetic must be installed. Installation instructions for ROS Kinetic can be found on the ROS Documentation Website. Once ROS is installed, create a Catkin Workspace in which the project package can be created. Once the workspace directory is created, the workspace must be created with catkin via

```
catkin_make
```

and the workspace must be sourced with

```
source devel/setup.bash
```

The entire repository can then be placed as a folder within the workspace directory.

Running the simulation

The project consists of two tests, the LQG Controller which allows for the robot to travel the preplanned trajectory, and the LQG Controller with the mitigation method under a simulated false data injection attack.

LQG Controller Test

In order to run the LQG Controller test, roscore must first be started in a terminal window on a remote computer:

```
roscore
```

Then, connect to the Turtlebot3 from the remote computer in a second terminal window and bringup the basic ROS packages on the turtlebot (must be run on the Turtlebot3 itself via SSH, not on the remote computer)

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Once roscore is up and the bringup packages have been run on the robot, navigate to the pathplanning package folder within the catkin workspace using a third terminal window. An easy way to navigate to this folder within ros is:

```
roscd pathplanningmqp
```

Once in the pathplanningmqp directory, run the test using:

```
rosrun pathplanningmqp FBL_LQD_ros.py
```

The robot must be placed in the middle of a room with moderate amount of free space to complete the trajectory. The robot will then attempt to travel the "figure eight" curve autonomously, with information about each iteration of the loop printed to the terminal. Once the robot has traveled the required amount, it will stop automatically. The program then displays graph outputs of the robot's trajectory over time compared to the actual pre-programmed trajectory, as well as other data to measure how closely the robot was able to follow the pre-programmed trajectory.

False Data Injection (FDI) Test

The FDI Test involves the LQG controller being run while a simulated attacker injects false information to the robot's Y position sensor. This requires the use of two separate sets of sensors, the Turtlebot3's LIDAR and it's internal IMU, which both are able to report the robot's position. Within this simulation, the LIDAR's Y position is given a random offset each iteration of the control loop. The mitigation method then takes the IMU and LIDAR position values, and calculates a

The first two steps to run the FDI test are the same as running the LQG test, which includes running roscore and the bringup packages, then navigating to the pathplanningmqp directory within the catkin workspace. Next, the built-in ROS GMapping package must be run in a third terminal window. This allows for the generation of a map of the robot's environment by the LIDAR:

```
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

With the gmapping running, the transform listener must be run in a seperate window, which allows for the estimated position of the robot within the map to be output to the LQG Controller. This uses the ROS /tf library:

```
rosrun pathplanningmqp transform_listener.py
```

With the GMapping and transform listener running, the FDI test can then be run:

```
rosrun pathplanningmqp FBI_LQR_FDI_simulation.py
```

The robot will then attempt to travel the same trajectory as in the LQG Test, while mitigating the simulated attack on the robot's LIDAR sensor.

Additional Notes

If the Turtlebot3 model is not set within ROS, an error may occur when trying to run either of the tests. In order to set the model, use:

```
export TURTLEBOT3_MODEL=burger
```

If the python file cannot be executed, make sure that the python file is given proper executable user permissions:

```
chmod +x FILE_NAME_HERE.py
```

Built With

- Turtlebot3 - An open-source autonomous robot platform
- Robot Operating System (ROS) - Robot Middleware Package used on Turtlebot3
- NumPy - Mathematical Library for Python in order to complete calculations using matricies
- CVXPy - Convex Optimization solver library for Python

Appendix B: PID Controller Implementation in Python Code

```
#Filename: PID_ros.py
#!/usr/bin/env python
import numpy as np
import rospy
import roslib
import tf
import math
from tf.transformations import euler_from_quaternion
import copy
import time
import sys, select, os
import matplotlib.pyplot as plt
if os.name == 'nt':
  import msvcrt
else:
  import tty, termios
from geometry_msgs.msg import PoseStamped, Twist, Pose, PoseWithCovariance
from nav_msgs.msg import Odometry
from sensor_msgs.msg import Imu, LaserScan
from tf.transformations import euler_from_quaternion
#Done as a node itself

# def controller_init(robot):

# robot.vel_msg.linear.x = Xi
# robot.vel_msg.angular.z = omega
# robot.vel_pub.publish(robot.vel_msg)
print("Initializing Controller...")
T = 100
num_steps = 600
tgetkey = 0
n = 3
m = 2
p = 3
pMinusS = np.array([2])
A = np.identity(3)
B = np.array([[1, 0],
              [1, 0],
              [0, 1]])
C = np.identity(3)
Sigma_w = np.identity(3)
Sigma_v = np.identity(3)
Q = np.array([[1, 0, 0],
              [0, 1, 0],
              [0, 0, 1]])
R = np.array([[1, 0],
              [0, 1]])
start_point = np.array([0, 0, 0])
rd_tar = 1
rd_obs = 1
target = np.array([2, 0.001, 0])
obs = np.array([-1, 1])
t = np.linspace(0, 99, num = num_steps
)

kp1 = 1
kp2 = 1
kd1 = 0.8
```

```
kd2 = 0.8

x1 = 0.5*np.sin(t/10)
x2 = 0.5*np.sin(t/20)

parametric_func = np.zeros((2,num_steps))
parametric_func[0] = x1
parametric_func[1] = x2

dt = float(T)/float(num_steps);
s = np.zeros((n, num_steps))
b = np.zeros((n,n,num_steps))

s[:,num_steps-1]=[0,0,0]
b[:,:,num_steps-1]=np.zeros(3)
degree = 3

g_D = rd_tar^2
g_U = rd_obs^2

ref_traj = parametric_func
diffrc = ref_traj[:,0]
# ref_traj[:,:] = ref_traj[:,:] - diffrc[:];
robot_pos = np.zeros((2,num_steps))

ref_length = len(ref_traj[1])
ref_traj = np.concatenate((ref_traj, np.ones((1,ref_length))))
for i in range(0,ref_length-1):
    ref_traj[2,i] = np.arctan((ref_traj[1,i+1]-ref_traj[1,i])/(ref_traj[0,i+1]-ref_traj[0,i]));
ref_traj[2,ref_length-1] = ref_traj[2,ref_length-2]

ref_traj_dot = np.zeros((3,ref_length))
for i in range (1, ref_length):
    ref_traj_dot[0,i] = ref_traj[0,i]-ref_traj[0,i-1]
    ref_traj_dot[1,i] = ref_traj[1,i]-ref_traj[1,i-1]
    ref_traj_dot[2,i] = ref_traj[2,i]-ref_traj[2,i-1]

ref_traj_db_dot = np.zeros((3,ref_length))
for i in range(0, ref_length-1):
    ref_traj_db_dot[0,i] = ref_traj_dot[0,i+1]-ref_traj_dot[0,i]
    ref_traj_db_dot[1,i] = ref_traj_dot[1,i+1]-ref_traj_dot[1,i]
    ref_traj_db_dot[2,i] = ref_traj_dot[2,i+1]-ref_traj_dot[2,i]
# add the third dimension: xdd, ydd
# rd = [x_dot; y_dot];
ref_length = len(ref_traj[2])
rd = np.zeros((2,ref_length-1))
for i in range (0, ref_length-1):
    rd[0,i] = ref_traj[0,i+1]-ref_traj[0,i]
    rd[1,i] = ref_traj[1,i+1]-ref_traj[1,i]

rdd = np.zeros((2,ref_length-2))
for i in range(0,ref_length-2):
    rdd[0,i] = rd[0,i+1]-rd[0,i]
    rdd[1,i] = rd[1,i+1]-rd[1,i]

# redefine start point and target
# start_point = ref_traj(:,1);

target = ref_traj[:,ref_length-1]

if dt <= 0:
    dt = 1e-4;

if n <= 0:
    n = 2;


if m <= 0:
    m = 2;


if p <= 0:
    p = 2;

[secureSensors] = pMinusS.shape;
if secureSensors > p:
    print('The number of secure sensors should be smaller than or equal to the total number of sensors.')

[rowA, colA] = A.shape;
if rowA != n or colA != n:
    print('A should be an n*n matrix.')

[rowB, colB] = B.shape;
if rowB != n or colB != m:
    print('B should be an n*m matrix.')

[rowC, colC] = C.shape;
if rowC != p or colC != n:
    print('C should be an p*n matrix.')

[rowQ, colQ] = Q.shape;
if rowQ != n or colQ != n:
    print('Q should be an n*n matrix.')

[rowR, colR] = R.shape;
if rowR != m or colR != m:
    print('R should be an m*m matrix.')

#parameters initialization
C_alpha = C[pMinusS-1,:];
Sigma_v_alpha = Sigma_v[pMinusS-1, pMinusS-1];
R_inv = np.linalg.inv(R);
Sigma_v_inv = np.linalg.inv(Sigma_v);tgetkey

x_hat = np.zeros((n, num_steps));
x_alpha_hat = np.zeros((n,num_steps));
x_real = np.zeros((n,num_steps));
x0 = start_point;
x_hat[:,0] = x0;
x_alpha_hat[:,0] = x0;
x_real[:,0] = x0;

G = Sigma_w;

P = np.zeros((n,n,num_steps));

Sigma_x = np.zeros((n, n, num_steps));

u = np.zeros((m, num_steps));
y = np.zeros((p, num_steps));

x_p = np.zeros((n,1));
```

```
Sigma_x_p = np.zeros((n,n));

error = np.zeros((1,num_steps));
cost = np.zeros((1,num_steps));
start_time = 0
elapsed_time = 0

finish = False;
# set(gcf,'CurrentCharacter','@'); % set to a dummy character

first_step_angle = np.arctan((ref_traj[1,1] - ref_traj[1,0])/(ref_traj[0,1] - ref_traj[0,0]));
init_angle = 0;
theta = first_step_angle-init_angle;
state_init = [0,0,1e-4];
#Initial B
B_ind = 0;

B = np.array([[np.cos(0.0079),0],
              [np.sin(0.0079),0],
              [0,1]]);

y[:,0] = state_init;

# plot(y(1,1),y(2,1))
# hold on
for i in range (0,num_steps):
    P[:,:,i] = np.subtract(b[:,:,i],Q);

Phi = np.zeros((n,n,num_steps));
Theta = np.zeros((n,p,num_steps));
Theta[:,:,0] = Phi[:,:,0]*C.conj().transpose()*Sigma_v_inv;
for i in range (1,num_steps):
    Phi_temp = A*Phi[:,:,i-1]*A.conj().transpose()+Sigma_w;
    Theta[:,:,i] = Phi_temp*C.conj().transpose()*np.linalg.inv(C*Phi_temp*C.conj().transpose()+Sigma_v);
    Phi[:,:,i] = A*Phi_temp*A.conj().transpose() + Sigma_w -
                 Phi_temp*C.conj().transpose()*np.linalg.inv(C*Phi_temp*C.conj().transpose()+Sigma_v)*
                 C*Phi_temp.conj().transpose();


u[0,0] = ref_traj_db_dot[0,0] + kp1*(ref_traj[0,0]-y[0,0])
u[1,0] = ref_traj_db_dot[1,0] + kp2*(ref_traj[1,0]-y[1,0])


def getKey():
    if os.name == 'nt':
      return msvcrt.getch()

    tty.setraw(sys.stdin.fileno())
    rlist, _, _ = select.select([sys.stdin], [], [], tgetkey)
    if rlist:
        key = sys.stdin.read(1)
    else:
        key = ''

    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key

def plotting():
    global ref_traj, robot_pos
    fig1 = plt.figure()
    fig1.suptitle("Reference trajectory vs Actual trajectory")
    plt.ioff()
    plt.plot(ref_traj[0,:], ref_traj[1,:], label = 'Reference trajectory')
    plt.plot(robot_pos[0,:], robot_pos[1,:], label = 'Actual trajectory')

    fig2 = plt.figure()
    fig2.suptitle("Mean Square Error\n" + "dt = " + str(dt) + "; T = " + str(T) + "; num_steps = " +
                  str(num_steps) + "; Elapsed time: " + str(elapsed_time))
    error = np.zeros(num_steps)
    for i in range(0, num_steps):
        error[i] = math.pow((np.linalg.norm(x_hat[0:2,i]-ref_traj[0:2,i])),2)/2;
    plt.plot(error)

    plt.show()

class pid_controller:
    def __init__(self):
        print("Creating PID Controller Node")
        rospy.init_node('PID_Controller')
        self.vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size = 2)
        self.odom_sub = rospy.Subscriber('/odom', Odometry, callback=self.odom_callback)
        self.imu_sub = rospy.Subscriber('/imu', Imu, callback=self.imu_callback)
        self.scan_sub = rospy.Subscriber('/scan', LaserScan, callback=self.scan_callback)
        self.odom_msg= Odometry()
        self.pose_msg = Pose()
        self.vel_msg = Twist()
        self.imu_msg = Imu()
        self.scan_msg = LaserScan()
        self.odom_updated = False
        self.imu_updated = False
        self.scan_updated = False

        self.odom_pose = Pose()
        print("Robot.py Initialized")

    def odom_callback(self, msg):
        self.odom_msg = msg
        self.odom_updated = True

    def imu_callback(self, msg):
        self.imu_msg = msg
        self.imu_updated = True

    def scan_callback(self, msg):
        self.scan_msg = msg
        self.scan_updated = True


    # Odom
    def pid_loop(self, msg, i):
        global B, robot_pos
        if i == 0:
            stime1 = time.time()
            Xi = u[0,0]*np.cos(y[2,0])*dt+u[1,0]*np.sin(y[2,0])*dt
            omega = (u[1,0]*np.cos(y[2,0])-u[0,0]*np.sin(y[2,0]))/Xi
            if omega > 0.3:
                omega = 0.3
            elif omega < -0.3:
                omega = -0.3
            self.vel_msg.linear.x = Xi
            self.vel_msg.angular.z = omega
            self.vel_pub.publish(self.vel_msg)
            elapsed = time.time() - stime1
            while elapsed < dt:
                elapsed = time.time() - stime1
        else:
            stime1 = time.time()
            print("Step number: " + str(i))
            tbot_x = msg.pose.pose.position.x
            tbot_y = msg.pose.pose.position.y
            robot_pos[0,i] = tbot_x
            robot_pos[1,i] = tbot_y
            quat = (msg.pose.pose.orientation.x, msg.pose.pose.orientation.y, msg.pose.pose.orientation.z,
                    msg.pose.pose.orientation.w)
            angles = euler_from_quaternion(quat)
            y[:,i] = [tbot_x, tbot_y, angles[2]];
            x_temp = np.matmul(A,x_hat[:,i-1]) + np.matmul(B,u[:,i-1]);
            x_hat[:,i] = x_temp + np.matmul(Theta[:,:,i],(y[:,i]-np.matmul(C,x_temp)));
            B = np.array([[np.cos(y[2,i]),0],[np.sin(y[2,i]),0],[0,1]])
            u[0,i] = ref_traj_db_dot[0,i] + kp1*(ref_traj[0,i]-y[0,i]) +
                     kd1*(ref_traj_dot[0,i]-y[0,i]+y[0,i-1]);
            u[1,i] = ref_traj_db_dot[1,i] + kp2*(ref_traj[1,i]-y[1,i]) +
                     kd2*(ref_traj_dot[1,i]-y[1,i]+y[1,i-1]);
            Xi = u[0,i]*np.cos(y[2,i])*dt+u[1,i]*np.sin(y[2,i])*dt
            omega = (u[1,i]*np.cos(y[2,i])-u[0,i]*np.sin(y[2,i]))/Xi
            # if omega > 0.3:
            #  omega = 0.3
            # elif omega < -0.3:
            #  omega = -0.3
            if y[2,i] > math.pi or y[2,i] < -math.pi:
             y[2,i] = y[2,i] - 2*math.pi*np.sign(y[2,i]);
            # error[i] = math.pow(np.linalg.norm(y[0:2,i] - ref_traj[0:2,i]),2)/2
            self.vel_msg.linear.x = Xi
            self.vel_msg.angular.z = omega
            self.vel_pub.publish(self.vel_msg)
            elapsed = time.time() - stime1
            while elapsed < dt:
                elapsed = time.time() - stime1
            print("Elapsed time:" + str(elapsed))
            print("----------------------")
        # print(self.vel_msg)

if __name__ == "__main__":
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)
    try:
        Robot = pid_controller()
        # controller_init(Robot)
        # Robot.odom_callback(Robot.odom_msg)
        start_time = time.time()
        for i in range(0, num_steps):
            key = getKey()
            if key == 'e':
                Robot.vel_msg.linear.x = 0
                Robot.vel_msg.angular.z = 0
                Robot.vel_pub.publish(Robot.vel_msg)
                exit()
                break
            Robot.pid_loop(Robot.odom_msg,i)
        Robot.vel_msg.linear.x = 0
        Robot.vel_msg.angular.z = 0
        Robot.vel_pub.publish(Robot.vel_msg)
        elapsed_time = time.time() - start_time
        plotting()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

Appendix C: LQG Controller Implementation in Python Code

```
#Filename: FBI_LQR_ros.py
#!/usr/bin/env python
import rospy
import roslib
import tf
import math
from tf.transformations import euler_from_quaternion
import copy
import time
import matplotlib.pyplot as plt
import time
import sys, select, os
if os.name == 'nt':
  import msvcrt
else:
  import tty, termios
from geometry_msgs.msg import PoseStamped, Twist, Pose, PoseWithCovariance
from nav_msgs.msg import Odometry
from sensor_msgs.msg import Imu, LaserScan
from tf.transformations import euler_from_quaternion

#Initialize Controller Variables
print("Initializing Controller Variables")
print("...............................")

#Change T and num_steps to modify runtime and dt (dt = T/num_steps)
T = 150;
num_steps = 15000;
tgetkey = 0; #for getkey() function used for debugging, recommend not to use because this will increase dt
significantly

n = 3;
m = 2;
p = 3;

pMinusS = np.array([2]);
A = np.identity(3);
B = np.array([[1, 0],
              [1, 0],
              [0, 1]])
C = np.identity(3);
Sigma_w = np.array([[1e-6, 0, 0],
                    [0, 1e-6, 0],
                    [0, 0, 1e-6]]);
Sigma_v = np.array([[1e-6, 0, 0],
                    [0, 1e-6, 0],
                    [0, 0, 1e-6]]);
Q = np.array([[1, 0, 0],
              [0, 1, 0],
              [0, 0, 1]]);
R = np.array([[1, 0],
              [0, 1]]);

rd_tar = 1;
rd_obs = 1;
target = np.array([2, 0.001, 0]);
obs = np.array([-1, 1]);
```

```
#Reference trajectory
t = np.linspace(0.0, 100.0, num = num_steps);
x1 = 0.8*np.sin(t/10);
x2 = 0.8*np.sin(t/20);

parametric_func = np.zeros((2,num_steps))
parametric_func[0] = x1
parametric_func[1] = x2

dt = float(T)/float(num_steps);
s = np.zeros((2, num_steps));
stemp = np.array([[0],[0]]);
b = np.zeros((2,2,num_steps));
s[:,num_steps-1]=[0,0];
A_l = np.identity(2);
B_l = dt*np.identity(2);
Q_l = np.identity(2);
degree = 3;
B_lh = B_l.conj().transpose()
g_D = rd_tar^2;
g_U = rd_obs^2;

ref_traj = parametric_func
diffrc = ref_traj[:,0]

ref_length = len(ref_traj[1]);
ref_traj = np.concatenate((ref_traj, np.ones((1,ref_length))));

for i in range(0,ref_length-1):
    ref_traj[2,i] = np.arctan((ref_traj[1,i+1]-ref_traj[1,i])/(ref_traj[0,i+1]-ref_traj[0,i]));
ref_traj[2,ref_length-1] = ref_traj[2,ref_length-2];

ref_traj_dot = np.zeros((3,ref_length));
for i in range (1, ref_length):
    ref_traj_dot[0,i] = (ref_traj[0,i]-ref_traj[0,i-1])/dt
    ref_traj_dot[1,i] = (ref_traj[1,i]-ref_traj[1,i-1])/dt
    ref_traj_dot[2,i] = (ref_traj[2,i]-ref_traj[2,i-1])/dt

ref_traj_db_dot = np.zeros((3,ref_length))
for i in range(0, ref_length-1):
    ref_traj_db_dot[0,i] = (ref_traj_dot[0,i+1]-ref_traj_dot[0,i])/dt
    ref_traj_db_dot[1,i] = (ref_traj_dot[1,i+1]-ref_traj_dot[1,i])/dt
    ref_traj_db_dot[2,i] = (ref_traj_dot[2,i+1]-ref_traj_dot[2,i])/dt

ref_length = len(ref_traj[2]);
rd = np.zeros((2,ref_length-1));
for i in range (0, ref_length-1):
    rd[0,i] = ref_traj[0,i+1]-ref_traj[0,i];
    rd[1,i] = ref_traj[1,i+1]-ref_traj[1,i];

rdd = np.zeros((2,ref_length-2));
for i in range(0,ref_length-2):
    rdd[0,i] = rd[0,i+1]-rd[0,i];
    rdd[1,i] = rd[1,i+1]-rd[1,i];

# redefine start point and target
# start_point = ref_traj(:,1);
start_point = ref_traj[:,0];
target = ref_traj[:,ref_length-1]

if dt <= 0:
    dt = 1e-4;

if n <= 0:
    n = 2;

if m <= 0:
    m = 2;

if p <= 0:
    p = 2;

[secureSensors] = pMinusS.shape;
if secureSensors > p:
    print('The number of secure sensors should be smaller than or equal to the total number of sensors.')

[rowA, colA] = A.shape;
if rowA != n or colA != n:
    print('A should be an n*n matrix.')

[rowB, colB] = B.shape;
if rowB != n or colB != m:
    print('B should be an n*m matrix.')

[rowC, colC] = C.shape;
if rowC != p or colC != n:
    print('C should be an p*n matrix.')

[rowQ, colQ] = Q.shape;
if rowQ != n or colQ != n:
    print('Q should be an n*n matrix.')

[rowR, colR] = R.shape;
if rowR != m or colR != m:
    print('R should be an m*m matrix.')

C_alpha = C[pMinusS-1,:];
Sigma_v_alpha = Sigma_v[pMinusS-1, pMinusS-1];
R_inv = np.linalg.inv(R);
Sigma_v_inv = np.linalg.inv(Sigma_v);

x_hat = np.zeros((n, num_steps));
x_alpha_hat = np.zeros((n,num_steps));
x_real = np.zeros((n,num_steps));
x0 = start_point;
x_hat[:,0] = x0;
x_alpha_hat[:,0] = x0;
x_real[:,0] = x0;

G = Sigma_w;

P = np.zeros((n,n,num_steps));

Sigma_x = np.zeros((n, n, num_steps));

u = np.zeros((m, num_steps));
y = np.zeros((p, num_steps));

x_p = np.zeros((n,1));
Sigma_x_p = np.zeros((n,n));

error = np.zeros((1,num_steps));
```

```
cost = np.zeros((1,num_steps));
finish = False;

#Calculate s matrix
for j in range(num_steps-2, -1, -1):
    k = -(np.linalg.inv(B_l.conj().transpose()*b[:,:,j+1]*B_l+R)*B_l.conj().transpose()*b[:,:,j+1])*A_l;
    b[:,:,j] = A_l.conj().transpose()*(b[:,:,j+1]-b[:,:,j+1]*B_l*np.linalg.inv(B_l.conj().transpose()*
                b[:,:,j+1]*B_l+R)*B_l.conj().transpose()*b[:,:,j+1])*A_l+Q_l;
    ref_traj_a = np.array([[ref_traj[0,j+1]],[ref_traj[1,j+1]]])
    stemp = np.matmul((A_l.conj().transpose() + k.conj().transpose()*B_l.conj().transpose()),stemp) –
            np.matmul(Q_l,ref_traj_a);
    s[0,j] = stemp[0]
    s[1,j] = stemp[1]

first_step_angle = np.arctan((ref_traj[1,1] - ref_traj[1,0])/(ref_traj[0,1] - ref_traj[0,0]));
init_angle = 0;
theta = first_step_angle-init_angle;
state_init = [0,0,1e-4];
B_ind = 0;

B = np.array([[np.cos(0.0079),0],
              [np.sin(0.0079),0],
              [0,1]]);
y[:,0] = state_init;

P = np.zeros(b.shape)
for i in range (0,num_steps):
    P[:,:,i] = np.subtract(b[:,:,i],Q_l);

Phi = np.zeros((n,n,num_steps));
Theta = np.zeros((n,p,num_steps));
Theta[:,:,0] = Phi[:,:,0]*C.conj().transpose()*Sigma_v_inv;

#Calculate initial control input u
uu1 = np.linalg.inv(np.matmul(np.matmul(B_lh,b[:,:,0]),B_l)+R)
uu2 = np.matmul(uu1,B_lh)
uu3 = (np.matmul(np.matmul(b[:,:,0],A_l),x_hat[0:2,0])+s[:,0])
uu3 = np.reshape(uu3,(2,1))
uu4 = np.matmul(uu2,uu3)/dt
uu4 = np.reshape(uu4,(2,1))
uu5 = np.reshape(ref_traj_db_dot[0:2,0]*dt,(2,1)) - uu4
u[:,0] = np.reshape(uu5,(1,2))

start_time = 0
elapsed_time = 0

#Plot graphs
def plotting():
    global ref_traj, y, dt, T, num_steps, elapsed_time
    plt.ioff()

    #Reference trajectory vs Actual trajectory
    fig1 = plt.figure()
    fig1.suptitle("Reference trajectory vs Actual trajectory\n " + "dt = " + str(dt) + "; T = " + str(T) + ";
                  num_steps = " + str(num_steps) + "; Elapsed time: " + str(elapsed_time))
    plt.plot(ref_traj[0,:], ref_traj[1,:], label = 'Reference trajectory')
    plt.plot(y[0,:], y[1,:], label = 'Actual trajectory')

    #Mean Square Error
    fig2 = plt.figure()
    fig2.suptitle("Mean Square Error\n" + "dt = " + str(dt) + "; T = " + str(T) + "; num_steps = " +
                    str(num_steps) + "; Elapsed time: " + str(elapsed_time))
    error = np.zeros(num_steps)
    for i in range(0, num_steps):
        error[i] = math.pow((np.linalg.norm(x_hat[0:2,i]-ref_traj[0:2,i])),2)/2;
    plt.plot(error)

    #Reference x vs Actual x
    fig3 = plt.figure()
    fig3.suptitle("Reference x vs Actual x")
    plt.plot(ref_traj[0,:], label = "Reference x")
    plt.plot(y[0,:], label = "Actual x")

    #Reference y vs Actual y
    fig4 = plt.figure()
    fig4.suptitle("Reference y vs Actual y")
    plt.plot(ref_traj[1,:], label = "Reference y")
    plt.plot(y[1,:], label = "Actual y")

    plt.show()

#Use this function to stop the robot with a key press, not recommended because this will increase dt 
significantly. Set tgetkey = 0 to disable
def getKey():
    global tgetkey
    if os.name == 'nt':
      return msvcrt.getch()

    tty.setraw(sys.stdin.fileno())
    rlist, _, _ = select.select([sys.stdin], [], [], tgetkey)
    if rlist:
        key = sys.stdin.read(1)
    else:
        key = ''

    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key

#LQR controller class, initialize node, publishers, and subscribers
class lqr_controller:
    def __init__(self):
        print("Creating LQR Controller Node")
        print("............................")
        rospy.init_node('LQR_Controller')
        self.vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size = 2)
        self.odom_sub = rospy.Subscriber('/odom', Odometry, callback=self.odom_callback)
        self.imu_sub = rospy.Subscriber('/imu', Imu, callback=self.imu_callback)
        self.scan_sub = rospy.Subscriber('/scan', LaserScan, callback=self.scan_callback)
        self.odom_msg = Odometry()
        self.pose_msg = Pose()
        self.vel_msg = Twist()
        self.imu_msg = Imu()
        self.scan_msg = LaserScan()
        self.odom_updated = False
        self.imu_updated = False
        self.scan_updated = False

    def odom_callback(self, msg):
        self.odom_msg = msg
        self.odom_updated = True

    def imu_callback(self, msg):
        self.imu_msg = msg
        self.imu_updated = True

    def scan_callback(self, msg):
        self.scan_msg = msg
        self.scan_updated = True

    def lqr_loop(self, msg, i):
        global A, B, Xi, omega,n,x_hat,dt
        #Calculate initial Xi and omega
        if i == 0:
            stime1 = time.time()
            Xi = u[0,0]*np.cos(x_hat[2,0])*dt+u[1,0]*np.sin(x_hat[2,0])*dt
            omega = dt*(u[1,0]*np.cos(x_hat[2,0])-u[0,0]*np.sin(x_hat[2,0]))/Xi
            self.vel_msg.linear.x = Xi
            self.vel_msg.angular.z = omega
            self.vel_pub.publish(self.vel_msg)
            elapsed = time.time() - stime1

            #Loop until elapsed >= dt
            while elapsed < dt:
                elapsed = time.time() - stime1
        else:
            stime1 = time.time() #start stopwatch for one single step
            print("Step number " + str(i))
            x_temp = np.matmul(x_hat[:,i-1],A).reshape(-1, 1) +
                     np.matmul(B,np.array([[1.5*Xi*dt],[omega*dt]]));
            A_ext = np.array([[1, 0, -dt*Xi*np.sin(x_hat[2,i-1])],
                     [0, 1, dt*Xi*np.cos(x_hat[2,i-1])],
                     [0, 0, 1]])
            Phi_temp = np.matmul(np.matmul(A_ext,Phi[:,:,i-1]),A_ext.conj().transpose())+Sigma_w;

            #Get positions from Turtlebot
            tbot_x = msg.pose.pose.position.x
            tbot_y = msg.pose.pose.position.y
            quat = (msg.pose.pose.orientation.x, msg.pose.pose.orientation.y, msg.pose.pose.orientation.z,
                    msg.pose.pose.orientation.w)
            angles = euler_from_quaternion(quat)
            y[:,i] = [tbot_x, tbot_y, angles[2]];

            z = y[:,i].reshape(-1,1)-np.matmul(C,x_temp);
            s_temp = np.matmul(np.matmul(C,Phi_temp),C.conj().transpose())+Sigma_v;
            Theta[:,:,i] = np.matmul(np.matmul(Phi_temp,C.conj().transpose()),np.linalg.inv(s_temp));
            x_hat[:,i] = np.reshape(x_temp + np.matmul(Theta[:,:,i],z),3);
            Phi[:,:,i] = np.matmul((np.identity(n) - np.matmul(Theta[:,:,i],C)),Phi_temp)
            B = np.array([[np.cos(x_hat[2,i]),0],[np.sin(x_hat[2,i]),0],[0,1]])

            #Calculate control input u
            uu1 = np.linalg.inv(np.matmul(np.matmul(B_lh,b[:,:,i]),B_l)+R)
            uu2 = np.matmul(uu1,B_lh)
            uu3 = (np.matmul(np.matmul(b[:,:,i],A_l),x_hat[0:2,i])+s[:,i])
            uu3 = np.reshape(uu3,(2,1))
            uu4 = np.matmul(uu2,uu3)/dt
            uu4 = np.reshape(uu4,(2,1))
            uu5 = np.reshape(ref_traj_db_dot[0:2,i]*dt,(2,1)) - uu4
            u[:,i] = np.reshape(uu5,(1,2))

            #Calculate Xi and omega
            Xi = u[0,i]*np.cos(x_hat[2,i])*dt+u[1,i]*np.sin(x_hat[2,i])*dt
            if Xi != 0:
                omega = dt*(u[1,i]*np.cos(x_hat[2,i])-u[0,i]*np.sin(x_hat[2,i]))/Xi
            else:
                omega = 0
            print("Xi is: " + str(Xi))
            print("omega is: " + str(omega))
            print("IMU X: " + str(tbot_x))
            print("IMU Y: " + str(tbot_y))

            #Publish message to velocity topic
            self.vel_msg.linear.x = Xi
            self.vel_msg.angular.z = omega
            self.vel_pub.publish(self.vel_msg)
            elapsed = time.time() - stime1
            while elapsed < dt:
                elapsed = time.time() - stime1
            print("Elapsed time:" + str(elapsed))
            print("----------------------")

#Main function
if __name__ == "__main__":
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)
    try:
        start_time = time.time()
        Robot = lqr_controller()
        for i in range(0, num_steps):
            key = getKey()
            if key == 'e':
                Robot.vel_msg.linear.x = 0
                Robot.vel_msg.angular.z = 0
                Robot.vel_pub.publish(Robot.vel_msg)
                exit()
                break
            Robot.lqr_loop(Robot.odom_msg,i)
        Robot.vel_msg.linear.x = 0
        Robot.vel_msg.angular.z = 0
        Robot.vel_pub.publish(Robot.vel_msg)
        elapsed_time = time.time() - start_time
        plotting()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

Appendix D: Mitigation Method Simulation Code

```
#Filename: lqgRT_py.py
#!/usr/bin/env python
import matlab.engine
import numpy as np
import math
import copy
import time
import matplotlib.pyplot as plt
import sys, select, os
if os.name == 'nt':
  import msvcrt
else:
  import tty, termios
import cvxpy as cvx
# T                -- the final time of the system
# dt               -- the time duration of each iteration
# num_steps        -- the number of iterations during T, 1000 is not enough
# n                -- the dimension of the system state
# m                -- the dimension of the system input
# p                -- the dimension of the system observation
# pMinusS          -- the index list of the secure sensors(row vector)
# A                -- n*n state matrix
# B                -- n*m input matrix
# C                -- p*n output matrix
# C_alpha          -- the matrix obtained by selecting the rows from C indexed in
#                     the observation matrix y that are not affected by the adversary
# Q                -- n*n cost matrix
# R                -- m*m cost matrix
# Sigma_w          -- n*n autocorrelation matrix
# Sigma_v          -- p*p autocorrelation matrix
# Sigma_v_alpha    -- the covariance matrix of v_alpha
# ref_traj         -- n*num_steps polynomial reference trajectory
# degree           -- the degree of the polyfit for the reference trajectory
eng = matlab.engine.start_matlab()
def lqgRT_v2(T, num_steps, n, m, p, pMinusS, A, B, C, Sigma_w, Sigma_v, Q, R, start_point, rd_tar, rd_obs,
             target, obs, t, parametric_func, degree):
    dt = float(T)/float(num_steps);
    g_D = math.pow(rd_tar,2)
    g_U = math.pow(rd_obs,2)

    ref_traj = parametric_func
    s_coeff = np.zeros((n,degree + 1))
    for i in range(0,n):
        s_coeff[i,:] = np.polyfit(t, np.reshape(ref_traj[i,:],num_steps), degree)
    for i in range(0,n):
        ref_traj[i,:] = np.polyval(s_coeff[i,:], t)
    if dt <= 0:
        dt = 1e-4
    if n <= 0:
        n = 2
    if m <= 0:
        m = 2
    if p <= 0:
        p = 2
    secureSensors = len(pMinusS)
    C_alpha = C[pMinusS-2,:]
    Sigma_v_alpha = Sigma_v[pMinusS-2, pMinusS-2]
    R_inv = np.linalg.inv(R)
    P = np.zeros((n,n,num_steps))

    x = np.zeros((n, num_steps))
    s = np.zeros((n, num_steps))
    x_hat = np.zeros((n, num_steps))
    x_alpha_hat = np.zeros((n,num_steps))
    x_real = np.zeros((n,num_steps))
    x0 = start_point
    x[:,0] = np.reshape(x0,2)
    x_hat[:,0] = np.reshape(x0,2)
    x_alpha_hat[:,0] = np.reshape(x0,2)
    x_real[:,0] = np.reshape(x0,2)

    G = Sigma_w;
    Sigma_v_inv = np.linalg.inv(Sigma_v)

    Phi = np.zeros((n,n));
    Theta = np.zeros((n, p));
    Phi_alpha = np.zeros((n,n))
    Theta_alpha = np.zeros((n, secureSensors))

    A_h = A.conj().transpose()
    B_h = B.conj().transpose()
    C_h = C.conj().transpose()
    C_alpha_h = C_alpha.conj().transpose()
    Phi_alpha_h = Phi_alpha.conj().transpose()
    for i in range(num_steps - 2, -1, -1):
        P[:,:,i] = P[:,:,i+1] + dt*(np.matmul(A_h,P[:,:,i+1]) + np.matmul(P[:,:,i+1],A) -
                   np.matmul(np.matmul(np.matmul(np.matmul(P[:,:,i+1],B),R_inv),B_h),P[:,:,i+1]) + Q)
        dsdt = (A_h - np.matmul(np.matmul(np.matmul(P[:,:,i],B),R_inv),B_h)).conj().transpose()
        dsdt = np.matmul(dsdt, s[:,i+1]) - np.matmul(Q,ref_traj[:,i+1])
        s[:,i] = s[:,i+1] + dsdt*dt
    BG = np.hstack((B,G))
    start_time = time.time()
    K = np.array([[22.8662692463450,22.8662692463450],[22.8662692463450,22.8662692463450]])
    # K = np.asarray(eng.KalmanOutput(matlab.double(A.tolist()), matlab.double(BG.tolist()),
          matlab.double(C.tolist()), 0, matlab.double(Q.tolist()), matlab.double(R.tolist()), 0, nargout = 1))
    Phi_alpha_prev = Phi_alpha + 0.0001
    Theta_alpha_prev = Theta_alpha + 0.0001
    Phi_alpha_bool =  abs(Phi_alpha - Phi_alpha_prev) >= 0.001
    while np.all(abs(Phi_alpha - Phi_alpha_prev) >= 0.001) or
          np.all(abs(Theta_alpha_prev - Theta_alpha_prev) >= 0.001):
        Phi_alpha_prev = Phi_alpha_prev
        Theta_alpha_prev = Theta_alpha
        dPhi_alpha_dt = np.matmul(A,Phi_alpha) + np.matmul(Phi_alpha,A_h) + Sigma_w -
                        np.matmul(np.matmul(np.matmul(np.matmul(Phi_alpha,
                        C_alpha_h),np.linalg.inv(Sigma_v_alpha)),C_alpha),Phi_alpha_h)
        Phi_alpha = Phi_alpha + dt*dPhi_alpha_dt
        Theta_alpha = np.matmul(np.matmul(Phi_alpha, C_alpha_h), np.linalg.inv(Sigma_v_alpha))

    gamma = 9.9216
    Phi = np.zeros((n,n))
    Phi_alpha = np.zeros((n,n))
    Phi_alpha_h = Phi_alpha.conj().transpose()
    for i in range(1, num_steps):
        u_alpha = np.matmul(np.matmul(np.matmul(-0.5*R_inv,B_h),P[:,:,i-1]),x_alpha_hat[:,i-1]) -
                  np.matmul(np.matmul(0.5*R_inv,B_h), s[:,i-1])
        u_alpha = np.reshape(u_alpha,(2,1))
        start_time = time.time()
        z = cvx.Variable(2)
        obj = cvx.Minimize(cvx.quad_form(z,R) + np.matmul(np.matmul(x_hat[:,i-1].T, P[:,:,i-1]),B.T)*z +
                np.matmul(s[:,i].T,B)*z)
        prob = cvx.Problem(obj,[0.5*cvx.quad_form(z,np.eye(2))+(-2*u_alpha.T*z) +
                 (np.matmul(u_alpha.T,u_alpha)) - math.pow(gamma,2) <= 0])
        prob.solve()
        u_ast = np.reshape(2*z.value,(2,1))
        print(u_ast)
        # u_ast =
eng.QCQPSolver(matlab.double(B.tolist()),matlab.double(R.tolist()),matlab.double(u_alpha.tolist()),gamma,matlab
.double(P.tolist()),matlab.double(x_hat.tolist()),matlab.double(s.tolist()),i+1,n,matlab.double(start_point.tol
ist()), nargout = 1)
        elapsed_time = time.time() - start_time
        w = np.random.normal(0,1,(n,1))
        dxdt = np.reshape(np.matmul(A, x[:,i-1]),(2,1)) + np.matmul(B,u_ast) + w
        x[:,i] = np.reshape(np.reshape(x[:,i-1],(2,1)) + dxdt*dt,2)
        v = np.random.normal(0,1,(p,1))
        v_alpha = v[pMinusS-1,:]
        attack = np.random.random((p,1))
        attack[pMinusS-1,0] = 0
        y = np.reshape(np.matmul(C,x[:,i]),(2,1)) + v + attack
        y = np.reshape(y,(2,1))
        y_alpha = np.matmul(C_alpha,x[:,i]) + v_alpha
        dPhi_alpha_dt = np.matmul(A,Phi_alpha) + np.matmul(Phi_alpha, A_h)
        dPhi_alpha_dt = dPhi_alpha_dt + Sigma_w -
                        np.matmul(np.matmul(np.matmul(Phi_alpha,C_alpha_h)*Sigma_v_alpha,C_alpha),Phi_alpha_h)
        dPhi_dt = np.matmul(A,Phi) + np.matmul(Phi, A_h) + Sigma_w -
                  np.matmul(np.matmul(np.matmul(np.matmul(Phi,C_h),Sigma_v_inv),C),Phi.conj().transpose())
        Phi_alpha = Phi_alpha + dt*dPhi_alpha_dt
        Theta_alpha = np.matmul(Phi_alpha,C_alpha_h)*Sigma_v_alpha
        Phi = Phi + dt*dPhi_alpha_dt
        Theta = np.matmul(np.matmul(Phi,C_h),Sigma_v_inv)
        Phi_alpha_h = Phi_alpha.conj().transpose()
        dxhat_dt = np.reshape(np.matmul(A,x_hat[:,i-1]),(2,1)) + np.matmul(B,u_ast) +
                   np.matmul(Theta,(y - np.reshape(np.matmul(C,x_hat[:,i-1]),(2,1))))
        x_hat[:,i] = np.reshape(np.reshape(x_hat[:,i-1],(2,1)) + dt * dxhat_dt,2)
        dxhat_alpha_dt = np.reshape(np.matmul(A,x_alpha_hat[:,i-1]),(2,1)) + np.matmul(B, u_ast) +
                         np.matmul(Theta_alpha, (y_alpha - np.matmul(C_alpha, x_alpha_hat[:,i-1])))
        x_alpha_hat[:,i] = np.reshape(np.reshape(x_alpha_hat[:,i-1],(2,1)) + dt*dxhat_alpha_dt,2)
        x_real[:,i] = np.reshape(np.reshape(x_real[:,i-1],(2,1)) + dt*(np.reshape(np.matmul(A,x_real[:,i-
1]),(2,1)) + np.matmul(B,u_ast)),2)

        print("Time elapsed: " + str(elapsed_time))
        print(i)
    plotting(ref_traj, x_real)
def plotting(ref_traj, x_real):
    plt.ioff()
    fig1 = plt.figure()
    plt.plot(ref_traj[0,:], ref_traj[1,:], label = 'Reference trajectory')
    plt.plot(x_real[0,:], x_real[1,:], label = 'Actual trajectory')
    plt.show()
if __name__ == "__main__":
    t = np.linspace(-5,0,10000);
    # t = np.reshape(t,(1,10000))
    x1 = t
    x2 = 5*np.ones(10000);
    lqgRT_v2(1, 10000, 2, 2, 2, np.array([2]), np.identity(2), np.identity(2),
             np.array([[1,1], [1, -1]]), np.identity(2), np.identity(2), np.identity(2),
             1e-3*np.identity(2), np.array([[-5], [5]]), 1, 1, np.array([[0], [5]]),
             np.array([[0], [2.5]]), t, np.vstack((x1,x2)), 5)
```

Appendix E: Mitigation Method Implementation Code

```
#Filename: FBI_LQR_FDI_simulation.py
#!/usr/bin/env python
import numpy as np
import rospy
import roslib
import tf
import math
from tf.transformations import euler_from_quaternion
import copy
import time
import matplotlib.pyplot as plt
import sys, select, os
if os.name == 'nt':
  import msvcrt
else:
  import tty, termios
from geometry_msgs.msg import PoseStamped, Twist, Pose, PoseWithCovariance
from nav_msgs.msg import Odometry
from sensor_msgs.msg import Imu, LaserScan
from tf.transformations import euler_from_quaternion
from pathplanningmqp.msg import transform
import cvxpy as cvx

#Initialize Controller Variables
print("Initializing Controller Variables")
print("...............................")

#Change T and num_steps to modify runtime and dt (dt = T/num_steps)
T = 150;
num_steps = 7500;
tgetkey = 0; #for getkey() function used for debugging, recommend not to use because this will increase dt
significantly

rd_tar = 1;
rd_obs = 1;
target = np.array([2, 0.001, 0]);
obs = np.array([-1, 1]);

#Reference trajectory
t = np.linspace(1, 100.0, num = num_steps);
x1 = 0.8*np.sin(t/10);
x2 = 0.8*np.sin(t/20);
parametric_func = np.zeros((2,num_steps))
parametric_func[0] = x1
parametric_func[1] = x2

n = 4;
n_l = 4;
m = 2;
p = 5;
pMinusS = np.array([1, 2, 4, 5]);

A = np.array([[0, 0, 1, 0],
              [0, 0, 0, 1],
              [0, 0, 0, 0],
              [0 ,0 ,0 ,0]]);

B = np.array([[0, 0],
              [0, 0],
              [1, 0],
              [0, 1]])

C = np.array([[1, 0, 0, 0],
              [0, 1, 0, 0],
              [0, 1, 0, 0],
              [0, 0 ,1, 0],
              [0, 0 ,0 ,1]]);

Sigma_w = np.array([[1e-6, 0, 0, 0],
                    [0, 1e-6, 0, 0],
                    [0, 0, 1e-6, 0],
                    [0, 0, 0, 1e-6]]);

Sigma_v = np.array([[1e-6, 0, 0, 0 ,0],
                    [0, 1e-6, 0, 0, 0],
                    [0, 0, 6*1e-4, 0, 0],
                    [0, 0, 0, 1e-6, 0],
                    [0, 0, 0, 0, 1e-6]]);
Q = np.identity(4);
R = np.identity(2);

dt = float(T)/float(num_steps);
s = np.zeros((n_l, num_steps));
b = np.zeros((n_l,n_l,num_steps));

stemp = np.array([[0],[0]]);

s[:,num_steps-1]=[0,0,0,0];
A_l = np.identity(2);
B_l = dt*np.identity(2);
Q_l = np.identity(2);
degree = 3;
B_lh = B_l.conj().transpose()
g_D = rd_tar^2;
g_U = rd_obs^2;

ref_traj = parametric_func

ref_length = len(ref_traj[1]);
ref_traj = np.concatenate((ref_traj, np.ones((1,ref_length))));
for i in range(0,ref_length-1):
    ref_traj[2,i] = np.arctan((ref_traj[1,i+1]-ref_traj[1,i])/(ref_traj[0,i+1]-ref_traj[0,i]));
ref_traj[2,ref_length-1] = ref_traj[2,ref_length-2];
start_point = ref_traj[:,0]

ref_traj_dot = np.zeros((3,ref_length));
for i in range (1, ref_length):
    ref_traj_dot[0,i] = (ref_traj[0,i]-ref_traj[0,i-1])/dt
    ref_traj_dot[1,i] = (ref_traj[1,i]-ref_traj[1,i-1])/dt
    ref_traj_dot[2,i] = (ref_traj[2,i]-ref_traj[2,i-1])/dt

ref_traj_db_dot = np.zeros((3,ref_length))
for i in range(0, ref_length-1):
    ref_traj_db_dot[0,i] = (ref_traj_dot[0,i+1]-ref_traj_dot[0,i])/dt
    ref_traj_db_dot[1,i] = (ref_traj_dot[1,i+1]-ref_traj_dot[1,i])/dt
    ref_traj_db_dot[2,i] = (ref_traj_dot[2,i+1]-ref_traj_dot[2,i])/dt
```

```
rd = np.zeros((2,ref_length-1));
for i in range (0, ref_length-1):
    rd[0,i] = ref_traj[0,i+1]-ref_traj[0,i];
    rd[1,i] = ref_traj[1,i+1]-ref_traj[1,i];

rdd = np.zeros((2,ref_length-2));
for i in range(0,ref_length-2):
    rdd[0,i] = rd[0,i+1]-rd[0,i];
    rdd[1,i] = rd[1,i+1]-rd[1,i];

start_point = np.concatenate((start_point, ref_traj_dot[0:2, 0]));
target = ref_traj[:,ref_length-1]
ref_traj = np.concatenate((ref_traj[0:2,:], ref_traj_dot[0:2,:]))
s_coeff = np.zeros((n,degree + 1))

diffrc = ref_traj[:,0]
diffrc = np.reshape(diffrc,(4,1))
ref_traj[:,:] = ref_traj[:,:] - diffrc;
start_point = ref_traj[:,0]
if dt <= 0:
    dt = 1e-4;

if n <= 0:
    n = 2;

if m <= 0:
    m = 2;

if p <= 0:
    p = 2;

[secureSensors] = pMinusS.shape;
if secureSensors > p:
    print('The number of secure sensors should be smaller than or equal to the total number of sensors.')

[rowA, colA] = A.shape;
if rowA != n or colA != n:
    print('A should be an n*n matrix.')

[rowB, colB] = B.shape;
if rowB != n or colB != m:
    print('B should be an n*m matrix.')

[rowC, colC] = C.shape;
if rowC != p or colC != n:
    print('C should be an p*n matrix.')

[rowQ, colQ] = Q.shape;
if rowQ != n or colQ != n:
    print('Q should be an n*n matrix.')

[rowR, colR] = R.shape;
if rowR != m or colR != m:
    print('R should be an m*m matrix.')

C_alpha = C[pMinusS-1,:];
Sigma_v_alpha = np.array([[1e-6, 0, 0, 0],
                          [0, 1e-6, 0, 0],
                          [0, 0, 1e-6, 0],
                          [0, 0, 0, 1e-6]])
```

```
Sigma_v_alpha_inv = np.linalg.inv(Sigma_v_alpha)
R_inv = np.linalg.inv(R);
Sigma_v_inv = np.linalg.inv(Sigma_v);

x_hat = np.zeros((n, num_steps));
x_alpha_hat = np.zeros((n,num_steps));
x_real = np.zeros((n,num_steps));
x0 = ref_traj[:,0];
x_hat[:,0] = x0;
x_alpha_hat[:,0] = x0;
x_real[:,0] = x0;

G = np.identity(n);

P = np.zeros((n,n,num_steps));

Sigma_x = np.zeros((n, n, num_steps));

u = np.zeros((m, num_steps));
u_ast = np.zeros((m, num_steps))
u_diff = np.zeros((m, num_steps))

y = np.zeros((p, num_steps));
y_alpha = np.zeros((secureSensors, num_steps));
y_dist = np.zeros((1, num_steps));

x_p = np.zeros((n,1));
Sigma_x_p = np.zeros((n,n));

error = np.zeros((1,num_steps));
cost = np.zeros((1,num_steps));
finish = False;

P = np.zeros((n,n))
P_prev = P+0.001

while np.all(abs(P - P_prev)) >= 0.001:
    P_prev = P
    P = P + (2*Q + np.matmul(A.T, P) + np.matmul(P, A) -
0.5*np.matmul(np.matmul(np.matmul(np.matmul(P,B),R_inv),B.T),P)) * dt
BG = np.hstack((B,G))

#Kalman filter coefficients
K = np.array([[1.28718850581117,    -6.38557171480376e-16,       -2.40028202841119e-17,
               0.414213562373096,    1.58212337667351e-16],
              [7.26223872225245e-17, 1.28602530882083,     0.00214337551470138,
               -1.42961771886354e-15,       0.414011544555625],
              [0.414213562373095, 2.63342314537682e-16,    -5.03336638922339e-18,
               0.910179721124455,    1.37263357977565e-16],
              [-1.22893379785640e-16, 0.414011544555625, 0.000690019240926037,
               -1.37425909154522e-16, 0.910114698839084]])

Phi_alpha = np.zeros((n,n))
Theta_alpha = np.zeros((n,secureSensors))

Phi_alpha_prev = Phi_alpha + 0.001
Theta_alpha_prev = Theta_alpha + 0.001

while np.all(abs(Phi_alpha - Phi_alpha_prev)) >= 0.001:
    Phi_alpha_prev = Phi_alpha;
    Theta_alpha_prev = Theta_alpha;
    dPhi_alpha_dt = np.matmul(A,Phi_alpha) + np.matmul(Phi_alpha, A.T) + Sigma_w –
                    np.matmul(np.matmul(np.matmul(np.matmul(Phi_alpha_prev,C_alpha.T),
                    Sigma_v_alpha_inv),C_alpha),Phi_alpha.T)
    Phi_alpha = Phi_alpha + dt*dPhi_alpha_dt
    Theta_alpha = np.matmul(np.matmul(Phi_alpha,C_alpha.T),Sigma_v_alpha_inv)

P = np.zeros((n,n,num_steps));
s = np.zeros((n, num_steps));

for i in range(num_steps - 2, -1, -1):
    P[:,:,i] = P[:,:,i+1] + dt*(np.matmul(A.T,P[:,:,i+1]) + np.matmul(P[:,:,i+1],A) -
0.5*np.matmul(np.matmul(np.matmul(np.matmul(P[:,:,i+1],B),R_inv),B.T),P[:,:,i+1]) + 2*Q)
    dsdt = (A.T - 0.5*np.matmul(np.matmul(np.matmul(P[:,:,i],B),R_inv),B.T))
    dsdt = np.matmul(dsdt, s[:,i+1]) - 2*np.matmul(Q,ref_traj[:,i+1])
    s[:,i] = s[:,i+1] + dsdt*dt

Phi = np.zeros((n,n,num_steps))
Phi_alpha = np.zeros((n,n,num_steps))
Theta = np.zeros((n,p,num_steps))
Theta_alpha = np.zeros((n,secureSensors,num_steps))

#Gamma should not exceed 15
gamma = 15

for i in range(1, num_steps):
    dPhi_alpha_dt = np.matmul(A, Phi_alpha[:,:,i-1]) + np.matmul(Phi_alpha[:,:,i-1], A.T) + Sigma_w -
np.matmul(np.matmul(np.matmul(np.matmul(Phi_alpha[:,:,i-
1],C_alpha.T),Sigma_v_alpha_inv),C_alpha),Phi_alpha[:,:,i-1].T)
    Phi_alpha[:,:,i] = Phi_alpha[:,:,i-1] + dt*dPhi_alpha_dt
    Theta_alpha[:,:,i] = np.matmul(np.matmul(Phi_alpha[:,:,i],C_alpha.T),Sigma_v_alpha_inv)

    dPhi_dt = np.matmul(A,Phi[:,:,i-1]) + np.matmul(Phi[:,:,i-1], A.T) + Sigma_w -
np.matmul(np.matmul(np.matmul(np.matmul(Phi[:,:,i-1],C.T),Sigma_v_inv),C),Phi[:,:,i-1].T)
    Phi[:,:,i] = Phi[:,:,i-1] + dt*dPhi_dt
    Theta[:,:,i] = np.matmul(np.matmul(Phi[:,:,i],C.T),Sigma_v_inv)

first_step_angle = np.arctan((ref_traj[1,1] - ref_traj[1,0])/(ref_traj[0,1] - ref_traj[0,0]));
init_angle = 0;
theta = first_step_angle-init_angle;
state_init = [0,0,1e-4];
B_ind = 0;
Xi = np.zeros((1,num_steps))
omega = np.zeros((1,num_steps))
angles = 0

def plotting():
    global ref_traj, y, dt, T, num_steps, elapsed_time
    plt.ioff()

    #Reference trajectory vs Actual trajectory
    fig1 = plt.figure()
    fig1.suptitle("Reference trajectory vs Actual trajectory\n " + "dt = " + str(dt) + "; T = " + str(T) + ";
num_steps = " + str(num_steps) + "; Elapsed time: " + str(elapsed_time))
    plt.plot(ref_traj[0,:], ref_traj[1,:], label = 'Reference trajectory')
    plt.plot(y[0,:], y[1,:], label = 'Actual trajectory')

    #Mean Square Error
    fig2 = plt.figure()
    fig2.suptitle("Mean Square Error\n" + "dt = " + str(dt) + "; T = " + str(T) + "; num_steps = " +
str(num_steps) + "; Elapsed time: " + str(elapsed_time))
    error = np.zeros(num_steps)
    for i in range(0, num_steps):
        error[i] = math.pow((np.linalg.norm(x_hat[0:2,i]-ref_traj[0:2,i])),2)/2;
    plt.plot(error)

    #Reference x vs Actual x
    fig3 = plt.figure()
    fig3.suptitle("Reference x vs Actual x")
    plt.plot(ref_traj[0,:], label = "Reference x")
    plt.plot(y[0,:], label = "Actual x")

    #Reference y vs Actual y
    fig4 = plt.figure()
    fig4.suptitle("Reference y vs Actual y")
    plt.plot(ref_traj[1,:], label = "Reference y")
    plt.plot(y[1,:], label = "Actual y")

    #Reference y vs Lidar's y
    fig5 = plt.figure()
    fig5.suptitle("Reference y vs Lidar's y")
    plt.plot(ref_traj[1,:], label = "Reference y")
    plt.plot(y[2,:], label = "Lidar's y")
    plt.show()

class lqr_controller:
    def __init__(self):
        print("Creating LQR Controller Node")
        print("............................")
        rospy.init_node('LQR_Controller')
        self.listener = tf.TransformListener()

        self.vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size = 2)
        self.odom_sub = rospy.Subscriber('/odom', Odometry, callback=self.odom_callback)
        self.imu_sub = rospy.Subscriber('/imu', Imu, callback=self.imu_callback)
        self.scan_sub = rospy.Subscriber('/scan', LaserScan, callback=self.scan_callback)
        self.trans_sub = rospy.Subscriber('/linear_trans', transform, callback=self.trans_callback)
        self.odom_msg = Odometry()
        self.pose_msg = Pose()
        self.vel_msg = Twist()
        self.imu_msg = Imu()
        self.scan_msg = LaserScan()
        self.trans_msg = transform()
        self.odom_updated = False
        self.imu_updated = False
        self.scan_updated = False
        self.trans_updated = False

    def odom_callback(self, msg):
        self.odom_msg = msg
        self.odom_updated = True

    def imu_callback(self, msg):
        self.imu_msg = msg
        self.imu_updated = True

    def scan_callback(self, msg):
        self.scan_msg = msg
        self.scan_updated = True
    def trans_callback(self, msg):
        self.trans_msg = msg
        self.trans_updated = True
    def lqr_loop(self, msg, i, trans_msg):
        global A, B, Xi, omega,n,x_hat,dt,angles
        if i == 0:
            stime1 = time.time()
            tbot_x = msg.pose.pose.position.x
            tbot_y = msg.pose.pose.position.y
            quat = (msg.pose.pose.orientation.x, msg.pose.pose.orientation.y, msg.pose.pose.orientation.z,
                    msg.pose.pose.orientation.w)
            angles = euler_from_quaternion(quat)
            print(tbot_x)
            print(tbot_y)

            y_lidar = trans_msg.linear_transform #from transform listener
            print("Y LIDAR IS:" + str(y_lidar*8))
            y[:,0] = [tbot_x, tbot_y, y_lidar*8, 0, 0]
            y_alpha[:,0] = [y[0,0],y[1,0],y[3,0],y[4,0]]
            u[:,0] = -0.5*(np.matmul(np.matmul(np.matmul(R_inv,B.T),P[:,:,0]),x_hat[:,0])) -
                     0.5*(np.matmul(np.matmul(R_inv,B.T),s[:,0]))
            u_ast[:,0] = u[:,0]
            Xi[0,0] = 0.9*T*dt*(u_ast[0,0]*np.cos(angles[2]) + u_ast[1,0]*np.sin(angles[2]))
            if Xi[0,i] != 0:
                omega[0,0] = 0.9*T*dt*(u_ast[1,0]*np.cos(angles[2]) - u_ast[0,0]*np.sin(angles[2]))/Xi[0,0]
            else:
                omega[0,0] = 0
            self.vel_msg.linear.x = Xi[0,0]
            self.vel_msg.angular.z = omega[0,0]
            self.vel_pub.publish(self.vel_msg)
            elapsed = time.time() - stime1
            while elapsed < dt:
                elapsed = time.time() - stime1
        else:
            stime1 = time.time()
            print("Step number " + str(i))

            #Predicting states
            dxhat_dt = np.reshape(np.matmul(A,x_hat[:,i-1]),(4,1)) +
                       np.reshape(np.matmul(B,u_ast[:,i-1]),(4,1)) +
                       np.matmul(Theta[:,:,i-1],np.reshape(y[:,i-1],(5,1)) -
                       np.reshape(np.matmul(C,x_hat[:,i-1]),(5,1)))
            print("DXHAT_DT: " + str(dxhat_dt))
            x_hat[:,i] = np.reshape(np.reshape(x_hat[:,i-1],(4,1)) + dt * dxhat_dt, 4)
            print("X_HAT: " + str(x_hat[:,i]))
            dxhat_alpha_dt = np.reshape(np.matmul(A,x_alpha_hat[:,i-1]),(4,1)) +
                             np.reshape(np.matmul(B, u_ast[:,i-1]),(4,1)) +
                             np.matmul(Theta_alpha[:,:,i-1], (np.reshape(y_alpha[:,i-1],(4,1)) -
                             np.reshape(np.matmul(C_alpha, x_alpha_hat[:,i-1]),(4,1))))
            print("DXHAT_ALPHA_DT: " + str(dxhat_dt))
            x_alpha_hat[:,i] = np.reshape(np.reshape(x_alpha_hat[:,i-1],(4,1)) + dt*dxhat_alpha_dt,4)
            print("X_ALPHA_HAT: " + str(x_alpha_hat[:,i]))
            u[:,i] = -0.5*(np.matmul(np.matmul(np.matmul(R_inv,B.T),P[:,:,i]),x_alpha_hat[:,i])) -
                     0.5*(np.matmul(np.matmul(R_inv,B.T),s[:,i]))
            print("U: " + str(u[:,i]))

            #QCQP Solver
            z = cvx.Variable(2)
            obj = cvx.Minimize(cvx.quad_form(z,R) + np.matmul(B.T,np.reshape(np.matmul(x_hat[:,i], P[:,:,i]) +
                  s[:,i],(4,1))).T*z)
            prob = cvx.Problem(obj,[0.5*cvx.quad_form(z,np.eye(2))+(-2*u[:,i].T*z) +
                   (np.matmul(u[:,i].T,u[:,i])) - math.pow(gamma,2) <= 0])
            prob.solve(solver = 'SCS')
            u_ast[:,i] = np.reshape(2*z.value,2) #Control input u alpha
            print("Control input: " + str(u_ast[:,i]))
            Xi[0,i] = (9000/T)*dt*(u_ast[0,i]*np.cos(angles[2]) + u_ast[1,i]*np.sin(angles[2]))
            if Xi[0,i] != 0:
                omega[0,i] = (9000/T)*dt*(u_ast[1,i]*np.cos(angles[2]) - u_ast[0,i]*np.sin(angles[2]))/Xi[0,i]
            else:
                omega[0,i] = 0
            print("Xi is: " + str(Xi[0,i]))
            print("omega is: " + str(omega[0,i]))
            self.vel_msg.linear.x = Xi[0,i]
            self.vel_msg.angular.z = omega[0,i]
            self.vel_pub.publish(self.vel_msg)
            elapsed = time.time() - stime1
            while elapsed < dt:
                elapsed = time.time() - stime1
            print("Elapsed time:" + str(elapsed))

            #Get positions from Turtlebot
            tbot_x = msg.pose.pose.position.x
            tbot_y = msg.pose.pose.position.y
            quat = (msg.pose.pose.orientation.x, msg.pose.pose.orientation.y, msg.pose.pose.orientation.z,
                    msg.pose.pose.orientation.w)
            angles = euler_from_quaternion(quat)
            print("angles: " + str(angles))

            #Get Y from lidar
            y_lidar = trans_msg.linear_transform
            print("Y LIDAR IS:" + str(y_lidar*8))
            print("ACTUAL Y IS: " + str(tbot_y))
            print("----------------------")
            a = 0.01*np.random.randn()
            y[0,i] = tbot_x
            y[1,i] = tbot_y
            y[2,i] = y_lidar*8+a
            y[3,i] = (tbot_x - y[0,i-1])/dt
            y[4,i] = (tbot_y - y[1,i-1])/dt
            y_alpha[:,i] = [y[0,i],y[1,i],y[3,i],y[4,i]]

if __name__ == "__main__":
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)
    try:
        start_time = time.time()
        Robot = lqr_controller()
        for i in range(0, num_steps):
            Robot.lqr_loop(Robot.odom_msg,i,Robot.trans_msg)
        Robot.vel_msg.linear.x = 0
        Robot.vel_msg.angular.z = 0
        Robot.vel_pub.publish(Robot.vel_msg)
        elapsed_time = time.time() - start_time
        print("x_hat is: " + str(x_hat))
        print("u is: " + str(u))
        plotting()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

Appendix F: Video Demos

LQG controller:

https://www.youtube.com/watch?v=ZOC838J_mFE&feature=share

Mitigation method:

https://www.youtube.com/watch?v=7I73yQhsI0o&feature=share