2022-06

USING REINFORCEMENT LEARNING TO SPOOF A MONITORED KALMAN FILTER

Bonitz, Dylan A.

Monterey, CA; Naval Postgraduate School

THESIS

USING REINFORCEMENT LEARNING TO SPOOF A MONITORED KALMAN FILTER

by

Dylan A. Bonitz

June 2022

Thesis Advisor:	Mark Karpenko
Co-Advisor:	Brian M. Wade

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE USING REINFORCEMENT LEARNING TO SPOOF A MONITORED KALMAN FILTER	**5. FUNDING NUMBERS**
6. AUTHOR(S) Dylan A. Bonitz	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) DOD Space, Washington, DC, 20001	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Modern hardware systems rely on state estimators such as Kalman filters to monitor key variables for feedback and performance monitoring. The performance of the hardware system can be monitored using a chi-squared fault detection test. Previous work has shown that Kalman filters are susceptible to false data injection attacks. In a false data injection attack, intentional noise and/or bias is added to sensor measurement data to mislead a Kalman filter in a way that goes undetected by the chi-squared test. This thesis proposes a method to deceive a Kalman filter where the attack data is generated using reinforcement learning. It is shown that reinforcement learning can be used to train an agent to manipulate the output of a Kalman filter via false data injection and without being detected by the chi-squared test. This result shows that machine learning can be used to successfully perform a cyber-physical attack by an actor who does not need to have in-depth knowledge and understanding of mathematics governing the operation of the target system. This result has significant real-world impact as modern smart power grids, aircraft, car, and spacecraft control systems are all cyber-physical systems that rely on trustworthy sensor data to function safely and reliably. A machine learning derived false data injection attack against any of these systems could lead to an undetected and potentially catastrophic failure.

14. SUBJECT TERMS machine learning, reinforcement learning, Kalman filter, chi-squared, cyber physical system	**15. NUMBER OF PAGES** 105
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

USING REINFORCEMENT LEARNING TO SPOOF A MONITORED KALMAN FILTER

Dylan A. Bonitz
Lieutenant, United States Navy
BS, United States Naval Academy, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SPACE SYSTEMS OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
June 2022

Approved by: Mark Karpenko
Advisor

Brian M. Wade
Co-Advisor

James H. Newman
Chair, Space Systems Academic Group

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Modern hardware systems rely on state estimators such as Kalman filters to monitor key variables for feedback and performance monitoring. The performance of the hardware system can be monitored using a chi-squared fault detection test. Previous work has shown that Kalman filters are susceptible to false data injection attacks. In a false data injection attack, intentional noise and/or bias is added to sensor measurement data to mislead a Kalman filter in a way that goes undetected by the chi-squared test. This thesis proposes a method to deceive a Kalman filter where the attack data is generated using reinforcement learning. It is shown that reinforcement learning can be used to train an agent to manipulate the output of a Kalman filter via false data injection and without being detected by the chi-squared test. This result shows that machine learning can be used to successfully perform a cyber-physical attack by an actor who does not need to have in-depth knowledge and understanding of mathematics governing the operation of the target system. This result has significant real-world impact as modern smart power grids, aircraft, car, and spacecraft control systems are all cyber-physical systems that rely on trustworthy sensor data to function safely and reliably. A machine learning derived false data injection attack against any of these systems could lead to an undetected and potentially catastrophic failure.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Mark Karpenko and LTC Brian Wade, for their compassionate guidance through this entire process. Your time, expertise, and patience were truly above and beyond and for that, I am extremely grateful.

My sincerest thanks to my incredible wife Alexandria. Her unconditional love, support, and patience made this possible for me. I am grateful, and especially lucky to have someone like you in my corner when the going gets tough.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

This research intends to investigate the feasibility of using reinforcement learning (RL) to disrupt the operation of cyber-physical systems. Similar to the impact the internet had on the ways people transfer information, cyber-physical systems are reshaping the ways people interact with hardware. Cyber-physical systems (CPSs) integrate hardware and software components of a singular engineered system or multiple interconnected systems connected to each other or to the internet. These types of systems take in sensor data from physical systems, process the received sensor data via software, and then take an autonomous action based on the physical systems' performance [1]. Due to significant advances in processing capabilities and over-the-air data transfer speeds, CPSs have become more prolific in the last decade. They can be found in everyday life in the form of smart home systems, automated vacuums, and dynamic vehicle safety systems [2]. Connected CPSs are becoming increasingly more commonplace in large-scale infrastructure as well in the form of smart grids, cellular networks, heating ventilation and air conditioning (HVAC) control, and military systems [3]. In all cases, CPSs simplify the use or control of a single system or system of systems for a user and reduce the operator's workload by automating and simplifying as many tasks as possible.

However, automated operation of hardware via software comes at a cost: security. Cyber-physical systems are no less vulnerable to cyberattacks than cyber information systems; however, the impacts can be significantly more severe. Cyberattacks against purely cyber systems do not inflict physical damage. If a cyberattack against a website is successful, the only serious issue is that site may be down for a period of time, limiting access to critical information. However, no hardware will be physically destroyed or impaired. A successful CPS cyberattack, against a smart electrical grid for example, could cause physical equipment to overload and explode, leaving millions without power [4].

At the heart of every CPS is an information system that collects and processes all the received sensor data from various parts of a physical system. The information system in an autonomous car, for example, is constantly receiving speed data from an accelerometer, location data from the odometer and a GPS receiver, and camera data and radar data from the various outward sensing equipment. It then processes all of the data and selects an action based on its surroundings and desired end state. From this example, it is clear to see how CPSs are significantly reliant on the data from the sensors that monitor their interaction within a given environment. Control systems imbedded into cyber-physical systems rely on state estimators and fault detection algorithms for control system feedback and to establish statistical norms for ingested sensor data to help detect outliers and/or system faults.

One of the primary workhorses in the field of state estimation is the Kalman Filter, and its many derivative forms [5]. Kalman filters provide an excellent means to estimate a true system state, such as position, in a noisy environment. A Kalman filter is an iterative process that takes consecutive data inputs (measured values) and estimates the true value of the state measured when there is uncertainty or random variations in the measured values. For example, Space Situational Awareness networks rely optical imaging and radar data to track space objects. The radar data received back from the ground transmitter will be inherently noisy from environmental and atmospheric effects. In this situation, a Kalman filter can be used to estimate the true position of the spacecraft from the noisy measurements.

In CPSs where measurements are taken rapidly, the user might not want the CPS to act on every piece of information. Occasionally, a bad measurement may be taken due to an environmental anomaly that is significantly different from the norm. To combat this issue, state estimators like the Kalman filter are paired with statistical fault detection algorithms. One such fault detection method is the chi-squared test. The chi-squared test suggests a distribution of the expected observations based on the measurement degrees of freedom. If a received measurement is sufficiently far from the expected value, the system will note that measurement as potentially bad. A series of bad measurements may trigger an alarm state on a CPS and alert an end user of a potential problem. Detailed information on Kalman filters and chi-squared testing can be found in Chapter II

A. PROBLEM STATEMENT

Examining a CPS from the attacker's point of view, this research will investigate potential vulnerabilities in the relationship between the information system and physical components of a cyber-physical system. More specifically, this research will investigate the feasibility of utilizing a reinforcement learning (RL), one of the three branches of machine learning (ML), to exploit a potential weakness in Kalman filters paired with chi-squared anomaly detection. The goal of this research is to model a cyber-physical system and an imbedded reinforcement learning algorithm that can alter the intended outcome of the system without being detected. As a proof-of-principle a simple line following robot is used to represent a CPS process. The robot takes simulated positional measurements at a given rate. The goal of this experiment will be to have the reinforcement learning algorithm learn to drive the robot off its intended course by an amount defined by a malicious user by adding a fictitious signal to the measurement data. This is an example of a false data injection attack, where false data is used to spoof the sensor measurement.

B. CURRENT RESEARCH

Recent research from Zhang et al. mathematically proved that a Kalman filter paired with a chi-squared detector can be successfully mislead from its intended output [6]. In their work, corrupt sensor measurement data was used to mislead a Kalman filter. It was shown that if a malicious actor intentionally causes the Kalman filter to ingest corrupted data, the output of the Kalman filter can be directed off its trajectory by a desired amount [6]. This can be done in the presence of a chi-squared fault detector without triggering a fault if the deviations are done methodically overtime.

Hou et al. studied the effects of falsified measurement data in output tracking and control systems [7]. In their research, the authors prove that false data injected into a target tracking control system can cause the CPS to believe the target is in an untrue location. This work shows that if an adversary compromises a target acquisition system, the end user could potentially engage a false target.

Mo et al. studied the effects of false data injection into wide area wireless sensor networks [8]. This research highlights how vulnerable a large, interconnected CPS can be compromised even if only a few sensors on the network are attacked.

The aforementioned works show that it is possible to manipulate Kalman filters and chi-squared detectors with a false data injection attack. However, these approaches rely on the fact that the attacker has a deep understanding of the associated advanced mathematics. This research will investigate the feasibility of using a RL algorithm to perform the same function. Should this prove successful, a machine learning-based attack would make the barrier to entry for a malicious actor significantly easier. Highlighting the potential for a casual attacker, with limited knowledge of a target CPS and a reasonable understanding of machine learning to develop an effective exploit. Moreover, a machine learning approach can be more easily adapted to different models since you don't need to re-design the attack. Simply re-create your target and let the algorithm learn how to compromise the system.

C. SCOPE AND APPLICATION

This research investigates the potential impacts machine learning may have on state estimator spoofing. This research will be focused on injecting false positional data using reinforcement learning into the state estimator composed of a standard Kalman filter and chi-squared detector. The system of interest is a simulated line-following robot that relies on a Kalman filter for position estimation. The goal of this experiment is to 'trick' the robot into following a different path without detection. There are many other types of state estimators and fault detectors, however, this pair was chosen since it is commonly used in large-scale industrial systems like smart-grids and air traffic control [7], [9]. This research highlights a significant vulnerability in any CPS that uses Kalman filters for control system data monitoring.

D. ASSUMPTIONS

To scope the problem appropriately, it was assumed that we have already infiltrated the cyber-physical system in question. That is, the machine learning algorithm has been maliciously imbedded into the firmware of a system either via a virus attached to a firmware update or in the build phase through supply chain interdiction.

This thesis is a proof-of-concept for a simple CPS. However, these concepts can be extrapolated to effect more complex CPSs.

E. RESEARCH BENEFITS

Studying cyber security issues from the attacker's point of view can help to provide alternative perspectives for studying security from a protection-focused point of view. Understanding what an actor may be able to do can help get ahead of the problem. If a malicious actor were able to spoof the fuel remaining data on a UAV, that malicious code could potentially allow the UAV to run below its minimum fuel required to make it back to base, causing the UAV to crash or fall into adversary hands. This research will also discover if a properly tuned reinforcement learning algorithm is capable of learning to spoof a Kalman filter without being detected. The implications for successfully training a (RL) algorithm to accomplish this task means that the attacker does not necessarily have to have an in-depth knowledge of operation the target CPS or how Kalman filters operate, but just the ability to create functioning machine learning systems.

F. THESIS ORGANIZATION

Chapter I provides an overview of the research conducted, describing the relationship between Kalman filters, chi-squared testing, and cyber-physical systems. Chapter II provides supporting information on Kalman filters chi-squared fault detection and RL methods used in this research. Chapter III describes the methodology used to model a CPS, a detailed description of the RL algorithm, and covers testing and verification of the algorithm's functionality. Chapter IV presents the results of the research. Chapter V details conclusions and areas of future research and applicability.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

This chapter provides background information on the key control system and machine learning theories used in this research. Section A and B provide background on key components of Kalman filters and chi-squared testing. Section C covers machine learning principles relevant to this experiment.

A. KALMAN FILTERS

Kalman Filters are widely used in dynamic system applications. See [5] for example. One of the earliest uses of a Kalman filter was for flight guidance and navigation control of the Apollo program. Since the Apollo program, spacecraft have only grown in complexity, requiring significantly more robust command-and-control systems. Kalman filters are routinely chosen to meet the demands of modern spacecraft attitude determination and control systems since they provide reliable state estimation. The following description on how Kalman filters work is derived from [5], [10], [11].

The purpose of Kalman filter is to estimate the states of a dynamical system based on sensor measurements and past predictions of the state. A Kalman filter is needed because not all system states can be directly measured. The Kalman filter uses a model of a system to estimate the state vector, *x*. In this thesis, the line following robot comprises two state variables, the *x* position and the *y* position.

The Kalman filter operates as an iterative two-step process. In the first step, the filter approximates the true system state by propagating a noisy model

$$\dot{x}_{pred,t} = Ax_{est,t-1} + Bu + \tag{1}$$

where *A* and *B* are matrices describing the system model and *w* is Gaussian process noise that models the discrepancy between the actual system behavior and the model. The subscript "pred" refers to the *a priori* prediction of the state dynamics (before a measurement is taken) and the subscript "est" refers to the state estimate from the previous time step.

At each time step a measurement, $y = z \pm v$, is also taken, where z is the true value of the measurement and v is sensor noise. Sensor noise is assumed to be zero mean, white-Gaussian noise similar to w. At the same time, a predicted measurement can be constructed from the predicted system state

$$y_{pred} = Hx_{pred} \pm v \tag{2}$$

where H is called the measurement matrix that maps the system states to measurements.

After calculating the predicted state and predicted measurement, a predicted state error covariance matrix, can be created using equation (3).

$$\mathrm{P}_{pred,t} = AP_{est-1}A^T + Q \tag{3}$$

where $P_{pred,t}$ is the current state error covariance matrix, Q is the process noise covariance matrix, and $P_{est,t-1}$ is the state error covariance matrix from the previous step. For the initial step, a random guess is taken for $P_{est,0}$. Next the Kalman gain, K_t, is calculated using equation (4).

$$K_t = \frac{P_{pred,t}H^T}{HP_{pred,t}H^T + R} \tag{4}$$

In equation (4), R is the sensor noise covariance matrix for the measurement model, and K is the Kalman gain. The Kalman gain is used to correct the predicted states to create an estimate that also incorporates the sensor data. When the Kalman gain is large the measurement is weighted over the prediction and when the Kalman gain is small the prediction is weighted over the measurement.

The corrected or *a posteriori* estimated value of the state, $x_{est,t}$, is computed using equation (5).

$$x_{est,t} = x_{pred,t} + K(y_t - y_{pred,t}) \tag{5}$$

where y_t is the vector of true measurements taken by the sensor(s) and $y_{pred,t}$ is the predicted measurement vector at the same time-step. The difference between y_t and $y_{pred,t}$ is known as the ‘innovation’. Finally, after the estimated state is computed, an *a posteriori* state error covariance matrix is calculated using equation (6).

$$P_{est,t} = (I - K_t H) P_{pred,t} \tag{6}$$

where I is an identity matrix of size equivalent to the dimension of the system. The estimated state value $x_{est,t}$ and the estimated error covariance matrix $P_{est,t}$ are used to report the current state of the system (and its uncertainty) and also used as part of the next iteration of the Kalman filtering process.

B. CHI-SQUARED FAULT DETECTION

Chi-squared testing is a statistical test that helps to reveal outlier data within an observed data set. [12]. Such an outlier can imply that there is a problem or fault within the system. The following section is derived from [12]–[14]. The notation for chi-squared is χ^2. The chi-squared test relies on the dimensionality or degrees of freedom in the monitored data of a system, denoted by p.Degrees of freedom are the number of independent variables in the test. This research focuses on a line following robot that can move in the x and y directions. In this example, motion in each direction is a degree of freedom resulting in two degrees of freedom test. If however a fault or no fault decision is to be made by monitoring several successive observations, the degrees of freedom increase with the size of the window. The degrees of freedom determine the shape of the chi-squared distribution as shown in Figure 1.

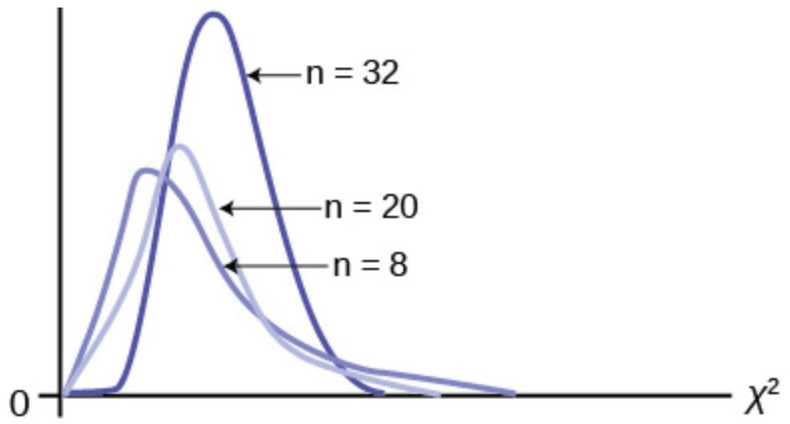


Figure 1. Chi-Squared Distribution vs. Degrees of Freedom. Source: [13].

Depicted in Figure 1, as the number of degrees of freedom in a given system increases, the chi-squared distribution begins to resemble a normal gaussian distribution.

The chi-squared test used to detect soft failures is depicted by equation (7).

$$\zeta_{\xi}(k) = \sum_{i=k-N+1}^{k} \xi^{T}(i)V^{-1}(i)\xi(i) \sim \chi^{2}(Np) \tag{7}$$

Where $\zeta_{\xi}(k)$ is a chi-square random variable with Np degree of freedom, where N is the window length, and $\xi(k)$ is a p-dimensional random gaussian white noise vector which has zero mean and covariance $V(k)$. In this experiment, the innovation of the Kalman filter is ξ and $\xi^{T}\xi$ is the innovation squared. Nominal values for the innovation should be close to zero, implying the predictions of the state made by the filter are close to the estimate, resulting in small chi-squared values. If the innovation is large then the chi-squared value will increase, potentially exceeding the threshold ε_{ξ}, triggering an alarm. Should the alarm be raised for N consecutive steps, the system will trigger a soft alarm alerting an operator or user of a potential fault in the system. A soft failure is a software detected discrepancy in a CPS. The hardware will continue to function during a soft failure, however the CPS will note that there was a fault measured.

C. REINFORCEMENT LEARNING

The purpose of this research was to investigate if a machine learning approach could be taken to deceive a Kalman filter that is monitored by a chi-squared fault detection test. To best meet the needs of this application, proximal policy optimization reinforcement learning method was chosen. The following sections provide background detail on reinforcement learning basics and proximal policy optimization methods. For additional information on machine learning methods see references [15]–[18].

1. Neural Networks

The use of neural networks is integral to modern machine learning techniques. Neural networks are comprised of an input layer, an output layer, and one or more hidden layers, depicted in Figure 2.

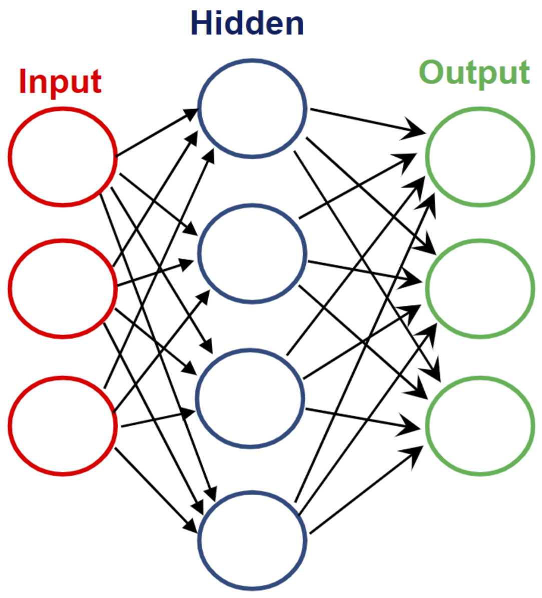


Figure 2. Neural Network with One Hidden Layer

The input layer has one neuron or node for each input variable. The output layer has one node per output variable. There can be one or multiple hidden layers with any number of nodes per layer. The composition of the hidden layers is selected by the engineer of the system. Layers are connected by edges, shown as black arrows in Figure 2.

Every edge carries a weight, *w,* which can be thought of as the strength of the connection of a particular synapse. A node with a strong weight leading to it will be more dominant than a node with a small weight. Every node other than the input nodes takes the output of the previous node, multiplies it by the edge weight *w*, adds a nodal bias, *b*, then performs an activation on that value. This is shown in Figure 3

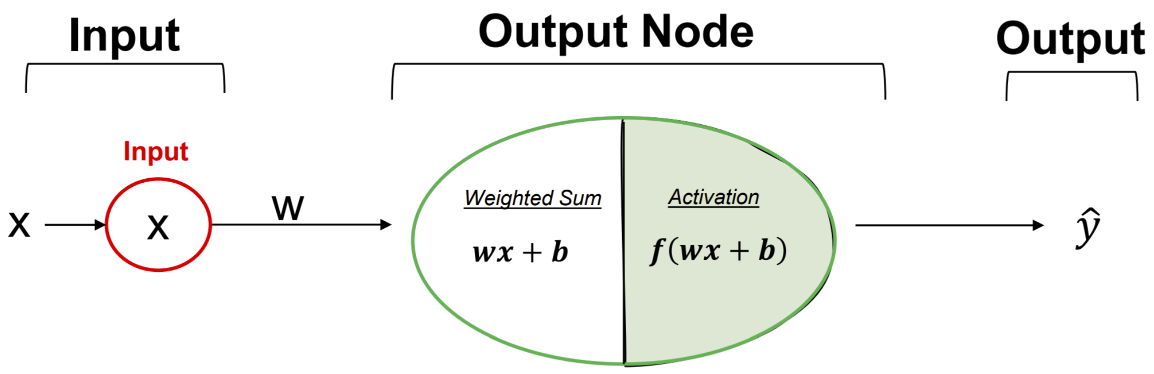
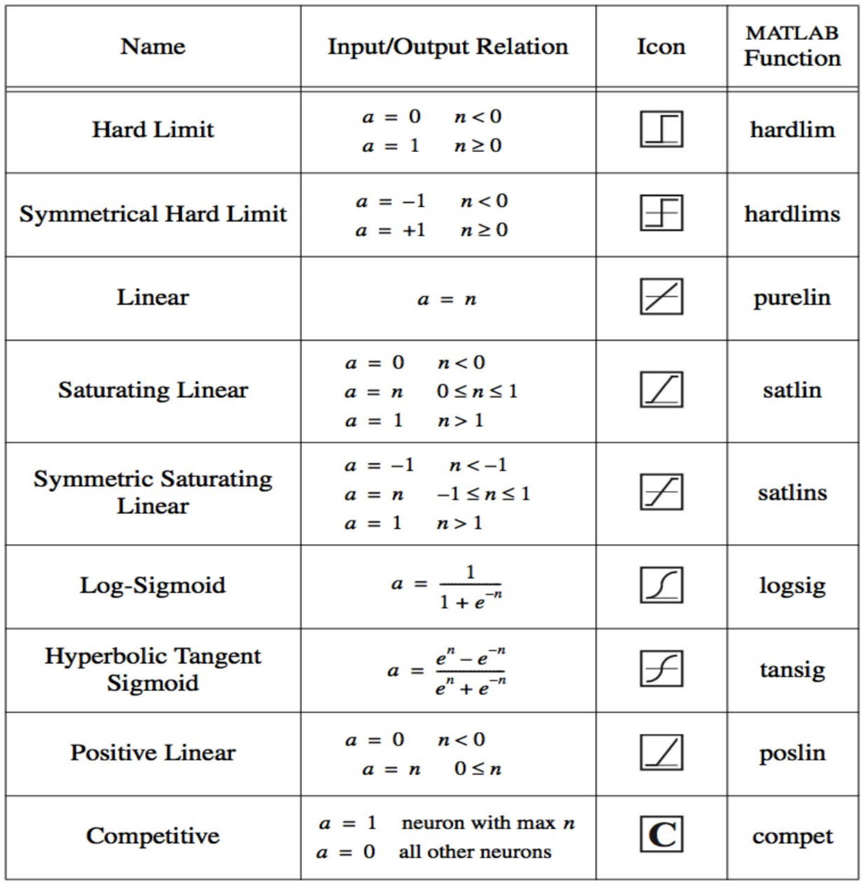


Figure 3. Single Node Actions

The activation function maps the weighted sum of the input into a new range. Hidden layer nodes produce an output, *a*, that becomes the input to the next hidden layer, or the input to the final output layer. The final output layer result is the estimation, $\hat{y}$, which represents the estimated output of the network based on the input and the current weights and biases. Figure 4 depicts the most commonly used activation functions. For additional information on activation functions and their applications, see reference [19].

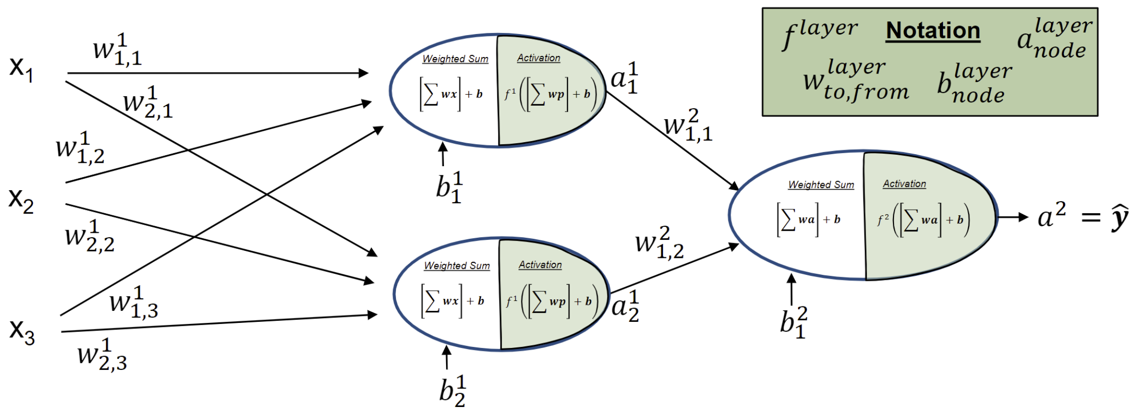


Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$		poslin
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons	$\boxed{C}$	compet

Figure 4. Common Activation Functions. Source: [20].

a. Forward Propagation

Forward propagation is the process of taking input variables through a neural network and producing an estimation, $\hat{y}$. Figure 5 details the processes of a feed forward neural network.



3 inputs, 1 2-node hidden layer, and 1 output layer.

Figure 5. Fully Connected Feed Forward Neural Network

The output of the hidden layer is computed by equation (8).

$$f^1\left(\begin{bmatrix} w^1_{1,1} & w^1_{1,2} & w^1_{1,3} \\ w^1_{2,1} & w^1_{2,2} & w^1_{2,3} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b^1_1 \\ b^1_2 \end{bmatrix}\right) = \begin{bmatrix} a^1_1 \\ a^1_2 \end{bmatrix} \tag{8}$$

After the intermediate outputs, a, are computed, the estimation of this simple network is computed by equation (9).

$$f^2\left(\begin{bmatrix} w^2_{1,1} & w^2_{1,2} \end{bmatrix}\begin{bmatrix} a^1_1 \\ a^1_2 \end{bmatrix} + \begin{bmatrix} b^2_1 \end{bmatrix}\right) = a^2 == \hat{y} \tag{9}$$

After the estimate, $\hat{y}$ is calculated, it is compared to the desired true value, y, to calculate an estimation error. The error is used to determine how to update the weights and bias of the neural network through back propagation.

b. Back Propagation

Back propagation is the process of updating the weights and bias of the neural network based on the error between the desired estimated output. The most common error metric is the sum squared error, depicted in equation (10).

$$SS_{error} = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{10}$$

For additional error cost functions applicable to various machine learning methods, see reference [19]. The weights and biases are adjusted throughout the neural network to minimize the value of the cost function. This process helps to minimize the influence of nodes that have a negative impact on the overall estimation and increase the influence of nodes with a positive impact on the network. The rate at which nodes are updated, known as the learning rate, is a network hyperparameter that is controlled by the designer. A learning rate close to 1 will cause the network to adjust weights and biases quickly, which may lead to erratic training. A very small learn rate will adjust the weights and biases more gradually, causing a more predictable training cycle. For additional information on back propagation, see reference [19].

2. Reinforcement learning basics

Reinforcement learning involves an agent interacting with an environment to influence the environment state, then receiving feedback, known as the reward, to learn how to properly interact with the given environment. The term "agent" refers to the components that make up the reinforcement learning algorithm, to include the sensors that provide information about the environment, and the neural networks and algorithms that evaluate the actions chosen by the agent. Figure 6 depicts how an agent interacts with an environment.

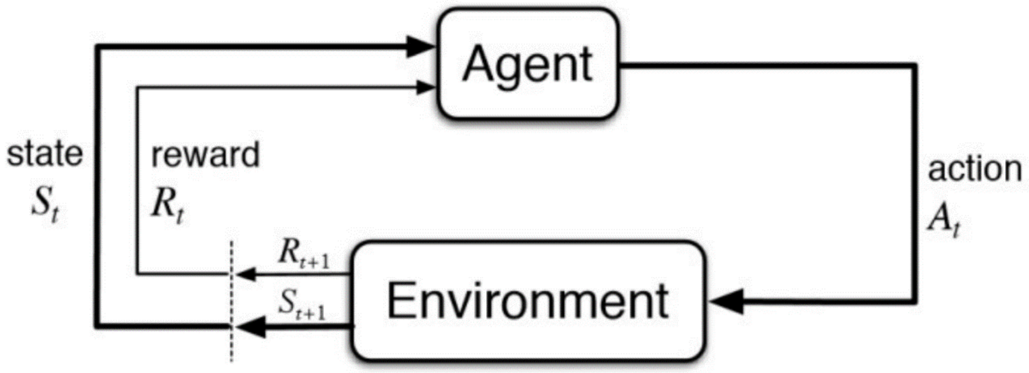


Figure 6. Basic RL Agent-Environment Interaction. Source: [16, p. 48].

In relation to this thesis, the agent action represents a change to the x or y measurement. The environment then provides a reward and a new state, via the Kalman filter, based on the previous state action pair. The state, S_t, is a vector of n variables that represent the agent's view of the environment at every timestep. The reward, R_t, is a single, real number that the agent receives as feedback from the action, A_t, taken in relation to the state. This process is how reinforcement learning algorithms learn to interact with an environment. The goal of the algorithm designer is to optimize the agent's understanding of the environment by shaping the reward function, so the agent learns how to accomplish a desired task through training simulations, by maximizing the reward. There are multiple methods of reinforcement learning, each with its own advantages and disadvantages, depending on the specific task and desired outcome. For this research, Proximal Policy Optimization was chosen since it performs better than alternatives in noisy stochastic environments [17].

3. Policy Gradient and Actor Critic Methods

The overall goal of reinforcement learning is to find the optimal policy, represented by π, to maximize the long-term reward of an individual training cycle known as an episode. The policy is the mapping of observed states to actions. The policy retains the information about a specific state and the optimal actions to take from the current state forward to maximize the long-term reward. To prevent a reinforcement learning algorithm from favoring near term rewards vice long term rewards, a discount factor, γ, is applied to the reward received at every timestep. The reward over time is depicted by equation (11),

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{11}$$

Where **G** is the cumulative long-term reward, **R** is the current step reward value, and γ is the discount factor. A small discount factor makes the algorithm more farsighted while a value close to 1 will make the algorithm greedy, favoring short term rewards. Maximizing the cumulative long-term reward is a key component of this research.

The key concept of policy gradient reinforcement learning is to increase the probabilities of actions that lead to a better long-term reward and decrease the probabilities of choosing actions that lead to lower rewards or simulation termination, until the algorithm reaches specified stopping criteria. Policy gradient methods compute an estimation of the policy gradient and plug it into a stochastic gradient ascent algorithm. The most common form of the gradient estimator is,

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta (a_t | s_t) A^{\pi_\theta}(s_t, a_t) \right] \tag{12}$$

where π_θ is the policy with hyperparameters θ, $J(\pi_\theta)$ is the expected finite-horizon undiscounted return of the policy. The advantage function of the current policy is represented by A^{π_θ}, a_t is the action at time t, and s_t is the state. After every episode, the policy is updated by equation (13).

$$\theta_{k+1} = \theta_k + a \nabla_\theta J(\pi_{\theta_k}) \tag{13}$$

Since policy gradient methods only update the policy directly, they do not learn an associated value for state action pairs over the course of the episode. Therefore, the agent does not process the potential value of being in a specific state, taking an action, and what effect that may have on the overall long-term reward. It only learns that it did well after it completes an entire episode and updates the policy based in its performance. Actor-critic methods take advantage of this short coming in policy gradient methods and combine these two methods in one agent.

Actor-critic methods combine policy-based functions, the actor, with value-based functions, the critic, to increase learning speed. The actor sets the policy for the agent. It uses policy gradient methods to derive best-action probabilities for a given state. The critic measure how good a chose action was. It then assigns a value to the previous state based on the action chosen and the projected long-term reward the agent will receive having taken that action. The relationship between the actor, critic, and the environment is depicted in Figure 7.

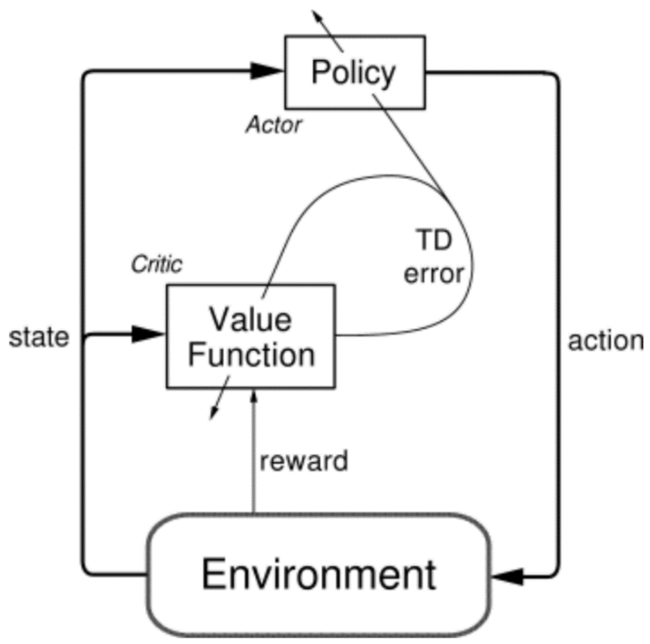


Figure 7. Actor-Critic and Environment Relationship. Source: [16].

The actor network, which builds the policy for the agent, choses an action. The agent then interacts with the environment based on that action and produces a new state and a reward for taking that action. The critic uses a temporal difference algorithm to learn the state value function for the actor's current policy[16, pp. 395–396]. Temporal difference of bootstrapping is an estimate of the value of a state based on the current estimate of the value of the next state. In short, temporal difference learning allows the critic to update the state action pair values while the algorithm is learning vice waiting for the completion of an episode to update the understand of each state action pair chosen. A positive temporal difference error tells the agent that the action resulted in a reward that was better than expected and updates its understanding of that state action pair and the policy.

The critic network update is represented by equation (14),

$$w_{t+1} = w_t + \beta \delta_t \nabla q_\pi (S_t, A_t) \tag{14}$$

where w are the weights and biases of the neural network, δ is the temporal difference, q_π is the value of each state action pair, and S and A are the state and action at each timestep respectively. The actor network update is represented by equation (15),

$$\theta_{t+1} = \theta_t + \alpha^\theta \delta_t \nabla \ln(\pi(a \mid S_t, \theta)) \tag{15}$$

4. Proximal Policy Optimization

The following section is derived from references [17], [21]. As mentioned in section 2, the advantage function in an actor critic can unintentionally cause destructively large policy updates that inhibit learning speed and progress. To help foster stable policy updates throughout training, proximal policy optimization was created. Proximal policy methods limit the policy gradient step to prevent large swings in policy updates. Mathematically, proximal policy optimization is represented by equation (16)

$$L^{CLIP}(\theta) = \hat{\mathrm{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \mathrm{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\right] \tag{16}$$

where $r_t(\theta)$ is the probability ratio of the current policy over the old policy depicted by equation (17).

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta\mathrm{old}}(a_t \mid s_t)} \tag{17}$$

The first term inside of the minimization function in equation (16) is the constructive policy iteration, which without the second term would lead to an excessively large policy update. The second term inside of the clip function in equation (16) penalizes changes to the policy that move $r_t(\theta)$ outside of the interval defined by $[1-\varepsilon, 1+\varepsilon]$. Due to the presence of the min operation, the objective behaves differently when the advantage, $\hat{A}_t$, is positive or negative. Figure 8 depicts a positive advantage, where the selected action by the agent resulted in a better-than-expected effect on the outcome.

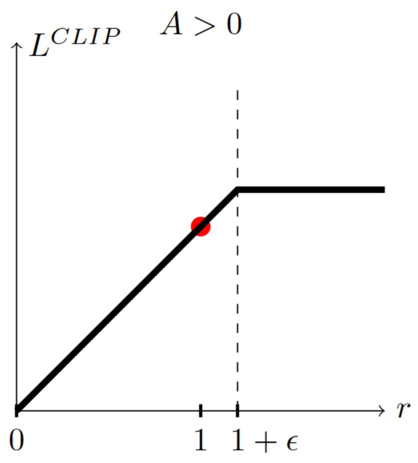


Figure 8. Single Timestep Plot with a Positive Advantage. Source: [17].

Figure 8 shows that the loss function flattens out when *r* gets too large, or if an action is significantly more likely under the current policy than it was under the previous policy. The goal is to not overdo the action update, so the objective is clipped to prevent this. The same concepts apply when the advantage is negative, as shown by Figure 9.

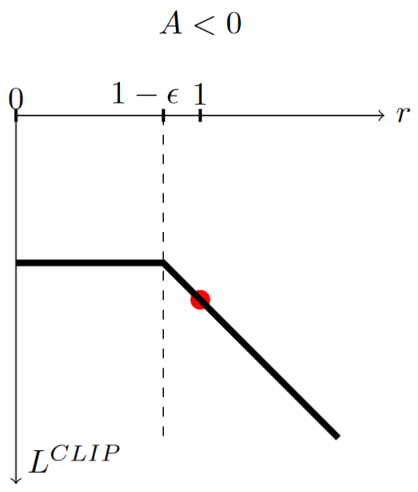


Figure 9. Single Timestep Plot with a Negative Advantage Source: [17].

The clipping approach not only helps to reduce large policy changes, but it also helps to undo policy mistakes efficiently since the new policy will constantly be compared to the old policy. The result is the final loss function for proximal policy represented by equation (18).

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathrm{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t)\right] \tag{18}$$

Where c_1 and c_2 are learning hyperparameters, $L_t^{VF}(\theta)$ is the mean squared error of the value function responsible for updating the baseline critic network and $S\left[\pi_\theta\right](s_t)$, is an entropy term used to encourage the agent to explore the action-space early in the training process and slowly fade away as training persists and the objective term $L_t^{CLIP}(\theta)$ starts dominating.

D. SUMMARY

This chapter provided an overview of the major used in this thesis. Kalman filters are recursive adaptive filters that model a state-space based on a prediction and a measurement update. Chi-squared tests, check data for goodness-of-fit, ensuring the quality of data in noisy monitored systems and providing an alarm for out of norm behavior. Reinforcement learning, specifically Proximal Policy Optimization, is a method to train an agent in the presence of a noisy dataset. Together these ingredients are used to achieve the goal of this thesis, which is to use a reinforcement learning agent to deceive a chi-squared monitored Kalman filter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. METHODOLOGY

This Chapter details the design, verification, and initial testing of the reinforcement learning agent and the unique environment of this experiment. First, a model line following robot was designed and functionally tested to operate within the reinforcement learning environment. Next, a reinforcement learning neural network was developed tested against in the environment to verify that the agent could choose actions to spoof the Kalman filter. Finally, a reward function capable of providing the agent the requisite feedback was developed and experiment shifted from algorithm testing to training.

A. EXPERIMENT DESIGN OVERVIEW

The goal of this research was to demonstrate the potential impact of a discrete cyberattack against a Kalman filter. For simplicity, this experiment was designed to simulate a line-following robot using MATLAB. The experiment allows a user to choose a desired angle for a robot to drive along for a desired number of steps. At every step, an integrator is used to compute the true path of the robot. The robot can take a step in both the *x* and *y* direction. A constant control input of 1 unit of velocity in each axis per step used. Figure 10 shows a simple example of the robot's path.

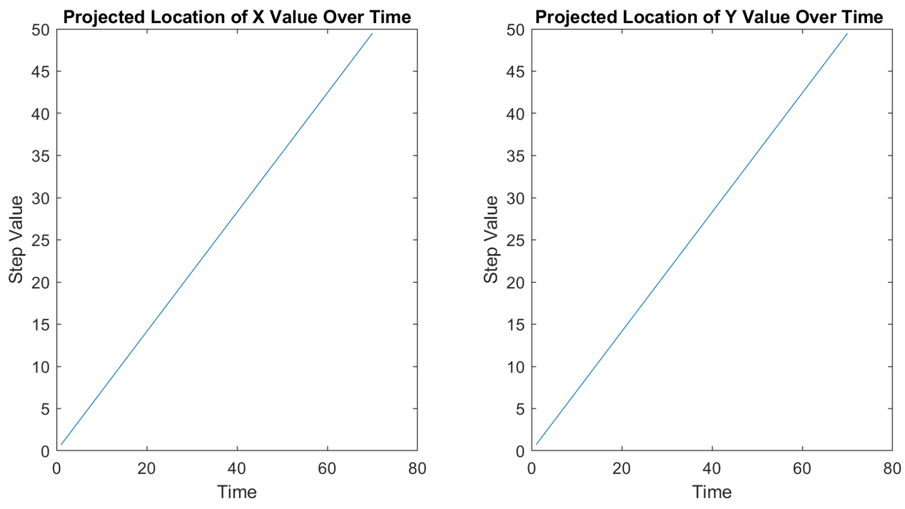


Figure 10. Projected Path over Time, Angle: 45° Number of Steps: 70

Figure ten depicts the projected path of the robot following a 45° angle for 70 steps. Next, at each projected step, a measurement of the robot's location is taken. Randomly generated noise is added to the measurement to simulate environmental factors that would affect any real-world position sensor. This measurement is what is going to be exploited by the main attack algorithm. Figure 11 depicts the noisy measurement overlaid on the projected path.

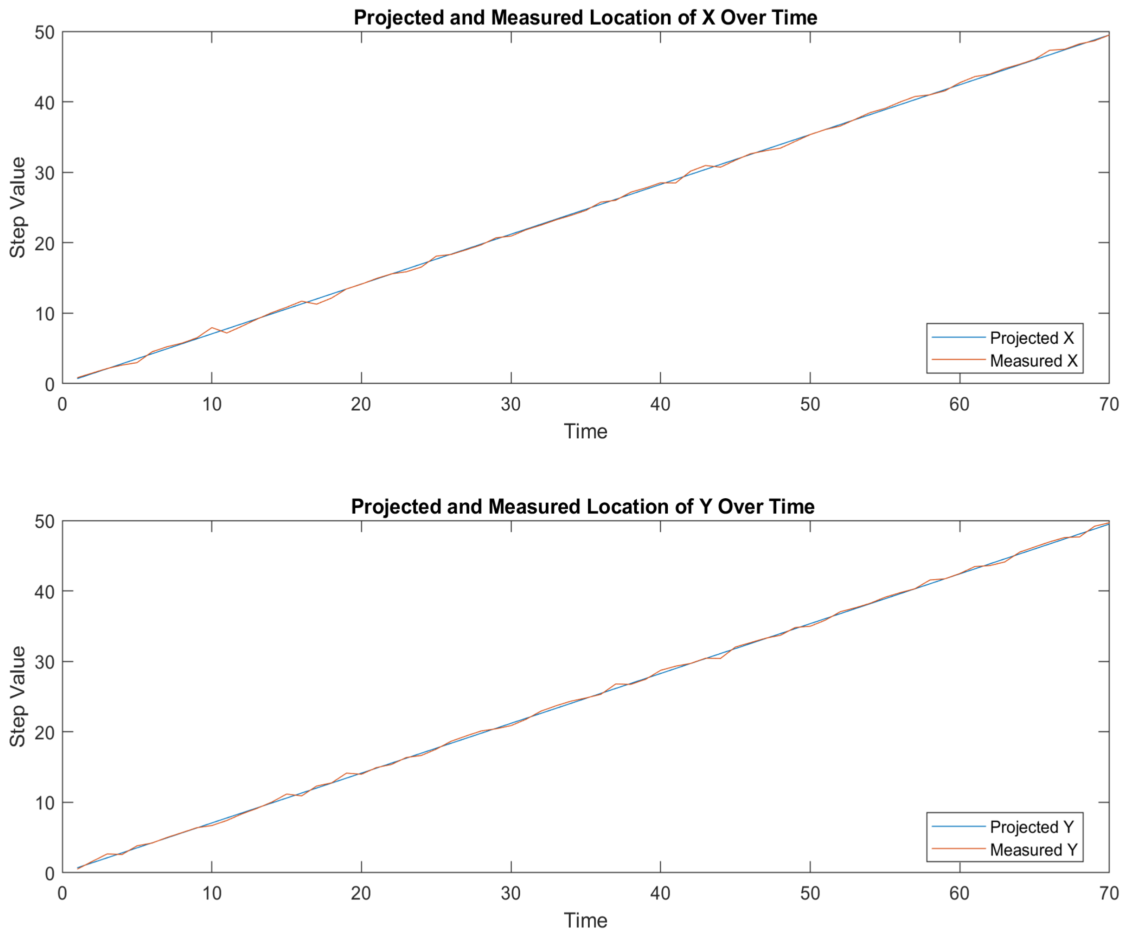
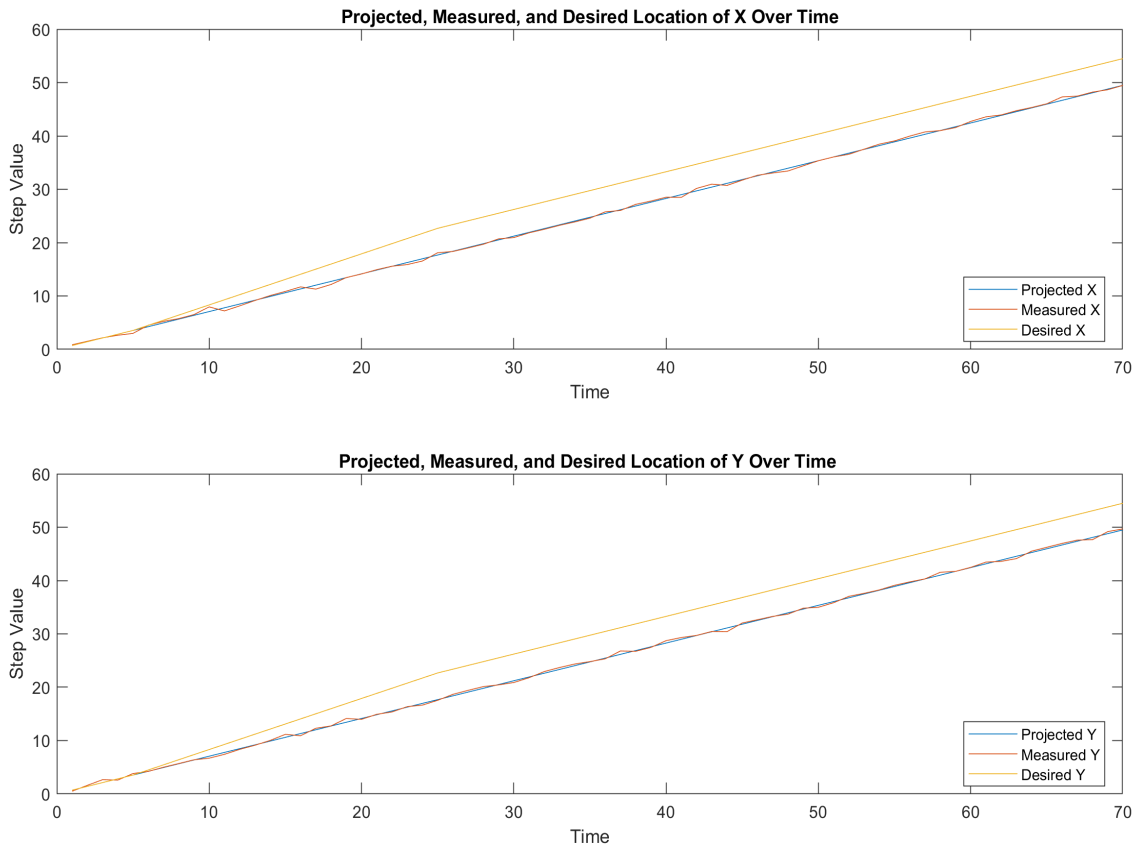


Figure 11. True Measurement Overlaid on the True Projected Path

While the true measurement is generated for each step based off the projected path, a desired attack path is also generated. The desired path is the attackers intended path for the robot. This experiment does not change the physical location of the robot. The output of the Kalman filter is the robot's perceived location. Thus, the robot is being tricked into thinking it is somewhere else. The attacker can predefine how quickly, how many times, how far off, and for how long they would like the spoofed path to deviate from the actual path. Figure 12 shows an example spoofed path, desired by the attacker.



Desired spoofed path deviation is both 5 units in *x* and *y*. The deviation starts at step 5 and reaches five units of deviation 25 steps into the simulation and maintains 5 units of deviation for the rest of the simulation.

Figure 12. Spoofed Path Overlay

In Figure 12, the desired spoofed path was set to start its deviation 5 steps into the simulation and reach a final deviation of 5 units in both *x* and *y* by step 25. Then it will maintain the 5-unit deviation for the remainder of the simulation. The attacker has the ability to make more or less sophisticated deviations and can choose to alter only *x* or *y*. All of these curves are generated at the start of each simulation in the system restart function which is discussed in detail in section B.1. of this chapter. Lastly, the goal of the algorithm is to learn to inject the optimal amount of noise/bias to the measurement at each time step to make the Kalman filter generate an estimated position path that mimics the desired spoofed path. The following sections will discuss in detail the components that make up the proximal policy reinforcement learning algorithm that will be used to deceive the Kalman filter undetected.

B. ALGORITHM DESIGN

The experiment is composed of three main components. The first components is the main script which is where a user defines the parameters of the training or testing senario. It also constructs the components of the reinforcement learning agent. The second is the reset function. The reset function initializes the training episode, builds the paths of the senario, and generates the first time-step state vector. The final component is the Kalman filter step function. The step function steps the Kalman filter one timestep and provides the agent a reward for its action in the environment. Figure 13 is a graphical representation of the relationship between the experiment scripts.

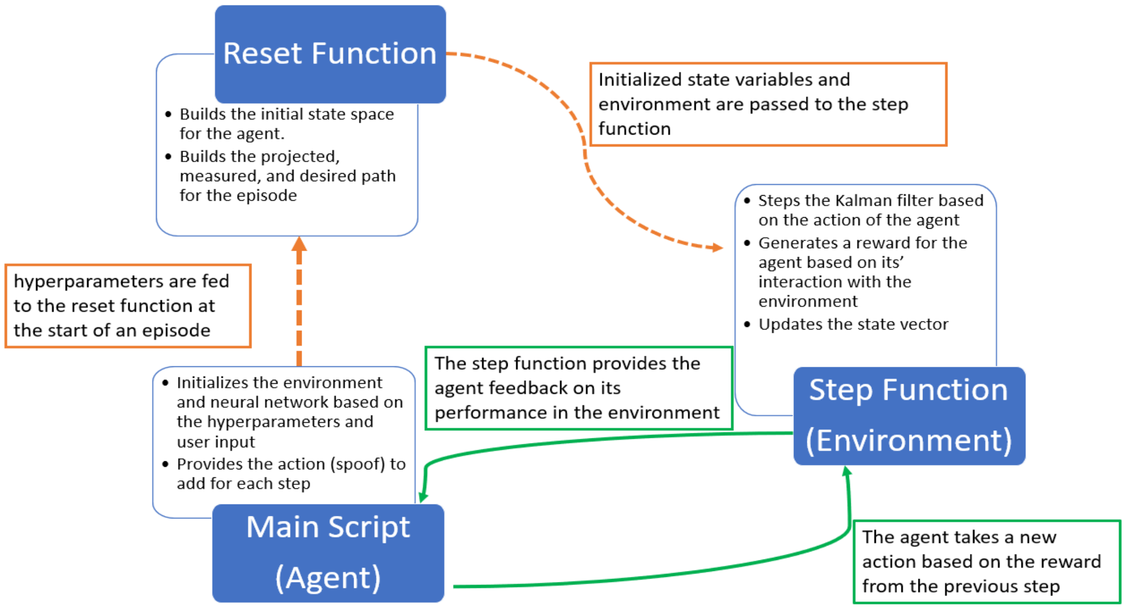

Reset Function
Builds the initial state space for the agent.
Builds the projected, measured, and desired path for the episode
Initialized state variables and environment are passed to the step function
Steps the Kalman filter based on the action of the agent
Generates a reward for the agent based on its' interaction with the environment
Updates the state vector
hyperparameters are fed to the reset function at the start of an episode
Initializes the environment and neural network based on the hyperparameters and user input
Provides the action (spoof) to add for each step
The step function provides the agent feedback on its performance in the environment
Step Function (Environment)
Main Script (Agent)
The agent takes a new action based on the reward from the previous step

Figure 13. Algorithm Flow Chart

1. Main Script

The main MATLAB script for the experiment is given Appendix A, which builds the neural networks and trains or tests the agent. Appendix A is broken into four sections. Section 1 initializes the hyperparameters of the networks and the user inputs. Section II builds the action space for the actor to choose from and constructs the reinforcement learning environment. Section III builds the actor, critic, and agent. Lastly, section IV trains or tests a new or previously saved agent.

a. Simulation Inputs, Action Space, and Reward Function Options

Table 1 depicts the simulation inputs. These are tunable inputs that an attacker can adjust to change the desired path of the robot.

Table 1. State Variables

Variable Name	**Description**	**Range of Values**
Time end	Sets the total number of steps per episode	70-200 total steps were tested
Speed	Sets a constant velocity for the training senario	Speed was kept constant at 1 for all testing
Initial angle (initial ang)	Sets the angle for the true path (x true) to be drawn from	Can accept 0–90 degrees. For training, initial angle was kept constant at 45 degrees
Distances (Dists)	Sets the desired deviation from the true path in both the *x* and *y* direction. Multiple deviations can be implemented	A 2 *x* n matrix where the first row is the starting value of the deviation and the second row is the desired end state. Ex. [0, 10; 10, 10;]
Time targets	Specify the time when the user wants the deviation to start and stop. Multiple time vectors can be created.	A 2 *x* n matrix where the first row is the start time and the second is the start time. Ex. [5, 10; 10, time end;]
Action range	Specifies the step size between -1 and 1 that builds the action space for the agent. Must be an odd number larger than three.	An action range of 5 produces the action space: [-1, -0.5, 0, 0.5, 1]
Action space reduction factor	Scales the action range	Values between .1 and 1.

Variable Name	**Description**	**Range of Values**
Noise reduction factor	Scales the noise taken by the simulated position sensor in the Kalman filter.	Values between 0 (no Noise) and 1 (full noise)

The values depicted in this table were for the final successful training session of the agent. Each episode was 100 seconds long. There was a single deviation of 10 units in both the *x* and *y* axis. The action space was reduced by 75%. The simulation had full noise.

In Table 1, "Time end" is the desired runtime of an individual simulation. One step is equivalent to 1 second. A senario with "time end" set to 100 will run for 100 seconds. The "speed" variable is a constant control input of 1 unit of motion in both the *x* and *y* direction per step. The "initial angle" draws the true path of the robot. For this experiment, the true path of the is a straight line based along the initial angle. All of the desired deviations, "Dists" are created based off the true path line. "Time targets" allows the user to specify when, and for how long, a deviation can occur. Table 2 depicts the action space, which is created based on the action range and action space reduction factor.

Table 2. Action Space

[-0.75; -0.75]	[-0.75; -0.375]	[-0.75; 0]	[-0.75; 0.375]	[-0.75; 0.75]
[-0.375; -0.75]	[-0.375;-0.375]	[-0.375; 0]	[-0.375; 0.375]	[-0.375; 0.75]
[0; -0.75]	[0; -0.375]	[0; 0]	[0; 0.375]	[0; 0.75]
[0.375; -0.75]	[0.375;-0.375]	[0.375; 0]	[0.375; 0.375]	[0.375; 0.75]
[0.75; -0.75]	[0.75;-0.375]	[0.75; 0]	[0.75; 0.375]	[0.75; 0.75]

Action range of 5. Action space reduction factor of 0.75. Total number of actions to choose from: 25.

For speed and simplicity, an "action range" of 5 discrete actions in each axis was chosen. This gives the agent a total of 25 different options to choose from at each step. An "action space reduction factor" of 0.75 was chosen based on trial and error. This reduction factor scales the available action-spaced based on the attacker's needs. In simulations with no reduction factor, the agent tended to alarm more often than simulations with some reduction. Reduction factors smaller than 0.75 made it difficult for the agent to reach the desired deviation in the 100-step simulation.

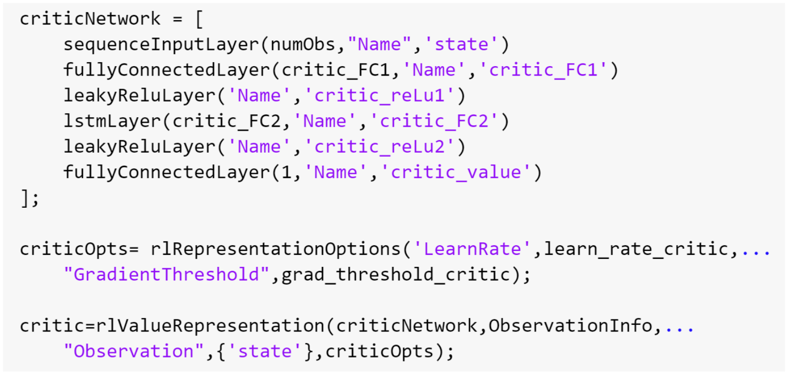
b. Critic

The critic, is the value network. The critic evaluates how good a chosen action was based on the projected long-term reward of being in a state and taking an action. Since the data is noisy, being in the same state as a pervious episode and taking the same action, may result in a lower-than-expected reward. To help the agent understand this phenomenon, a long-short-term-memory layer (LSTM) was added to the critic neural network. More information on LSTM layers can be found in reference[18] The Critic uses the hyperparameters in Table 3 to construct the network in Figure 14.

Table 3. Critic Options

Variable	**Value**
Critic FC1[a]	50
Critic FC2	20
Learn rate critic	0.0001
Gradient threshold critic	1

a. Fully Connected (FC).



```
criticNetwork = [
    sequenceInputLayer(numObs,"Name",'state')
    fullyConnectedLayer(critic_FC1,'Name','critic_FC1')
    leakyReluLayer('Name','critic_reLu1')
    lstmLayer(critic_FC2,'Name','critic_FC2')
    leakyReluLayer('Name','critic_reLu2')
    fullyConnectedLayer(1,'Name','critic_value')
];

criticOpts= rlRepresentationOptions('LearnRate',learn_rate_critic,...
    "GradientThreshold",grad_threshold_critic);

critic=rlValueRepresentation(criticNetwork,ObservationInfo,...
    "Observation",{'state'},criticOpts);
```

Figure 14. Critic Network

c. Actor

The actor is the policy-based network. The actor network chooses an action from the action space based on the value of that action determined by the critic. The actor uses the hyperparameters in Table 4 to construct the network in Figure 15.

Table 4. Actor Network Options

Variable	**Value**
Actor FC1[a]	150
Actor FC2	100
Actor FC3	50
Learn rate actor	0.00001
Grad threshold actor	1

a. Fully Connected (FC).

```
actorNetwork = [
    sequenceInputLayer(numObs,"Name",'state')
    fullyConnectedLayer(actor_FC1,'Name','actor_FC1')
    reluLayer('Name','actor_reLu1')
    lstmLayer(actor_FC2,'Name','actor_FC2')
    leakyReluLayer('Name','actor_reLu2')
    lstmLayer(actor_FC3,'Name','actor_FC3')
    leakyReluLayer('Name','actor_reLu3')
    fullyConnectedLayer(num_actions,'Name','actor_FC4') %one node per action
    softmaxLayer('Name','action_prob')
];

actorOpts= rlRepresentationOptions('LearnRate',learn_rate_actor,...
    "GradientThreshold",grad_threshold_actor);

actor=rlStochasticActorRepresentation(actorNetwork,ObservationInfo,...
    ActionInfo,'Observation',{'state'},actorOpts);
```

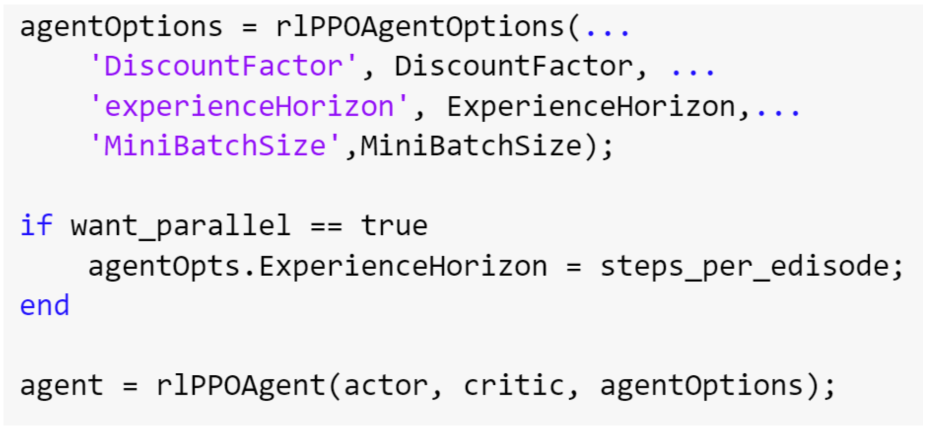
Figure 15. Actor Network

d. Agent and Training Options

The agent encompasses both the actor and the critic. The critic network maps values to individual state-action pairs, while the actor builds the policy to choose actions that optimize the cumulative long-term reward based on the values generated by the critic. The agent uses the parameters described in Table 5 to build the agent depicted in Figure 16.

Table 5. Reinforcement Learning Agent Options

Hyperparameter	**Description**	**Value**
Discount Factor	Determines if the agent favors long-term or short-term rewards.	0.99 (Far looking)
Experience Horizon	How the PPO agent looks out over a specified number of steps	150 (for a 200 second scenario)
Mini Batch Size	Determines the amount of data stored in memory per step	64



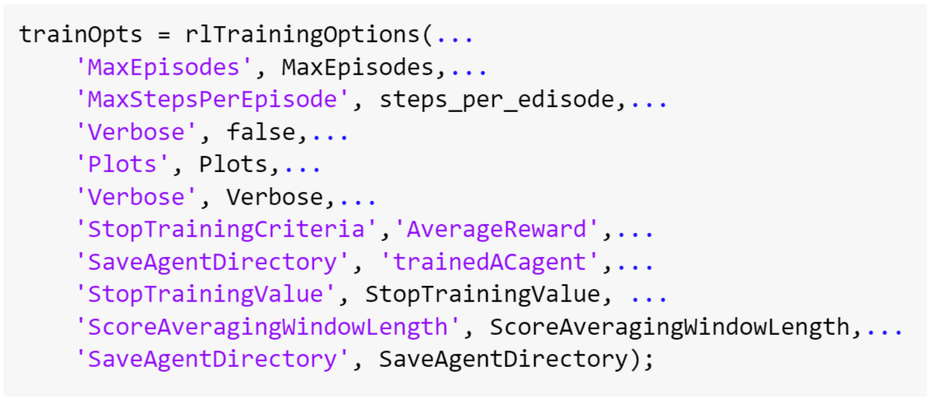
```
agentOptions = rlPPOAgentOptions(...
    'DiscountFactor', DiscountFactor, ...
    'experienceHorizon', ExperienceHorizon,...
    'MiniBatchSize',MiniBatchSize);

if want_parallel == true
    agentOpts.ExperienceHorizon = steps_per_edisode;
end

agent = rlPPOAgent(actor, critic, agentOptions);
```

Figure 16. Agent Creation

The training options set the training criteria for agent and the environment. Table 6 and Figure 17 depict the training options used. Visualization options are omitted from Table 6.

Table 6. Key Training Options

Variable	Description	Value
Max Episodes	The max number of episodes to run for if no other criteria is met	500,000
Max Steps Per Episode	Equal to the total number of steps per episode	100
Average Reward	A stop-training-criteria where once the average reward is met for the specified window length, training terminates.	See equation in Section I.f of Appendix A. 297
Score Average Window	Number of episodes steps to compute the average reward	30



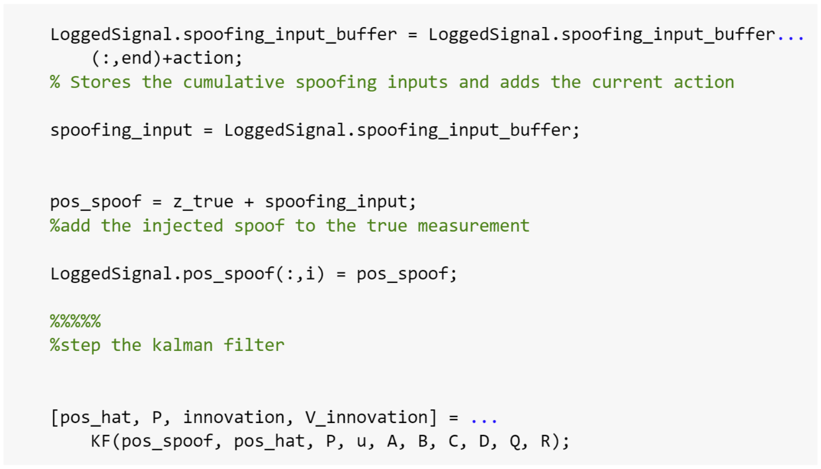
```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes', MaxEpisodes,...
    'MaxStepsPerEpisode', steps_per_edisode,...
    'Verbose', false,...
    'Plots', Plots,...
    'Verbose', Verbose,...
    'StopTrainingCriteria','AverageReward',...
    'SaveAgentDirectory', 'trainedACagent',...
    'StopTrainingValue', StopTrainingValue, ...
    'ScoreAveragingWindowLength', ScoreAveragingWindowLength,...
    'SaveAgentDirectory', SaveAgentDirectory);
```

Figure 17. Agent Training Options

The final sections of Appendix A initiate the training process, then plot relevant training data once training has terminated or completed.

2. Step Function

The second key script for the experiment is the combined Kalman filter step function and reward function, found in Appendix B. The Kalman filter step function is called by the environment of the main script for every timestep in an episode. The inputs to the Kalman filter step function are the "actions" chosen by the agent to propagate the system toward the desired path, the structure "LoggedSignal" which stores the variables that are carried forward through the system. The "out," "in," and "max-R" variables are used to build the reward function, and “Time end” specifies the duration of the training episode. The step function then performs three primary functions. First, it steps the environment one unit of time through the dynamics and Kalman filter process. During this process, the action chosen by the agent is accumulated and added to the true measurement of the system to begin the spoofing process depicted in Figure 18.



```
LoggedSignal.spoofing_input_buffer = LoggedSignal.spoofing_input_buffer...
    (:,end)+action;
% Stores the cumulative spoofing inputs and adds the current action

spoofing_input = LoggedSignal.spoofing_input_buffer;


pos_spoof = z_true + spoofing_input;
%add the injected spoof to the true measurement

LoggedSignal.pos_spoof(:,i) = pos_spoof;

%%%%%
%step the kalman filter


[pos_hat, P, innovation, V_innovation] = ...
    KF(pos_spoof, pos_hat, P, u, A, B, C, D, Q, R);
```

Figure 18. Injecting Spoof onto the True Measurement

In Figure 18, where “spoofing input buffer” is the cumulative sum of all the actions chosen by the agent for a given episode. The “spoofing input” is the current timestep’s value of “spoofing input buffer” and “pos spoof” is the summation of the spoofed data and the true measurement at the current timestep. This is recorded for later analysis.

Next, the step function performs a chi-squared test on the innovation of the Kalman filter. As discussed in Chapter II section B, the chi-squared fault detection test determines if a value is outside of normal. If the squared innovation exceeds the Chi-squared threshold, an alarm state is recorded. Alarming after a single fault is typically unreasonable since it may just be an anomaly and not a true fault. To help prevent constant alarms from being raised, systems with chi-squared fault detection systems often utilize an alarm window, where if an alarm is raised for a specified number of iterations in a row, then an actual alarm is triggered alerting the system and observers. Figure 19 and Figure 20 depict the chi-squared test and windowed alarm tracker of the step function.

```
N_window = 30;
N_chi = 2*N_window;  %the Chi^2 DOF N_chi is equal to the dimension
                     %of the innovation, in this case 2, times the
                     %length of the data window

PD = 0.997;   % Probability the system detects a bad when it occurs

l= innovation'*V_innovation^(-1)*innovation; %l is the Chi^2 value

LoggedSignal.l(1,end+1) = l;
l = LoggedSignal.l;
idx_lo = max(1,i-N_window+1);
idx_hi = i;
cl = sum(l(idx_lo:idx_hi));    %windowed sum fof Chi^2 values
```

Figure 19. Chi-squared Test on the Innovation

```
 if (cl>gammaincinv(PD,N_chi,'lower'))
     Alarm = 1; %raise an alarm
 else
     Alarm = 0;
 end
 LoggedSignal.Alarm(1,end+1) = Alarm;
l = l(1,end);

 %alarm shoudld be on for at least N_window+2 consequtive samples

 idx_lo = max(1,i-(N_window+2)+1);
 sum_Alarm = sum(LoggedSignal.Alarm(idx_lo:idx_hi))/(N_window+2);

 if (sum_Alarm>=1)
     smoothed_Alarm = 1;
 else
     smoothed_Alarm = 0;
 end

 LoggedSignal.smoothed_alarm(1,end+1) = smoothed_Alarm;
```

Figure 20. Calculating the Windowed Alarm

The goal of the agent in this experiment is to learn how to mislead the Kalman filter to allow the simulation to follow the attacker's chosen desired path, vice the true path, without triggering the windowed alarm state.

The last major component of the step function is the reward function. The reward function provides the agent feedback as to how well it is performing the specified task in the environment. Reward function shaping is an integral part of machine learning [22]. The system designer needs to find a function that allows the agent to build an understanding of how each action it can take effects the overall simulation outcome and eventually, leads the agent to learn how to take an optimal action for a given state.

Table 7 depicts the reward function options. These options are specified in the main script and are imported into the step function. The variable “Out” determines the *x*-intercepts of the reward function. For example, an “Out” of 10 makes *x*-intercepts at -10 and 10, and determines the width of the plateau. An “In” value of 1 sets the plateau of the reward function from -1 to 1. “Max reward” determines the height of the reward function and sets the *y*-value of the plot at the plateau. Figure 21 captures the component of the reward function that constructs the shape of the curve based on the user inputs defined in Table 7.

Table 7. Reward Function Options

Variable	**Value**
Out	10
In	1
Max Reward (max-R)	1

```
if x1 < -in && x1 >= -out
    reward = slope * x1 + b;
elseif (-in <= x1) && (x1 <= in)
    reward = max_R;
elseif x1 > in && x1 <= out
    reward = -slope * x1 + b;
else
    reward = -1;
    IsDone = 1;
end
```

Figure 21. Reward Function Curve Generation

A reward is generated for the *x* and *y* component of the robot’s motion separately. This helps to ensure that the agent can process a senario where there may be a different desired path in the *x* and *y* directions. The variable “x1” in Figure 21 is the difference between the current output of the Kalman filter and the desired path. The reward function, shown in Figure 21, takes the values from Table 7 and generates the reward function as shown in Figure 22.

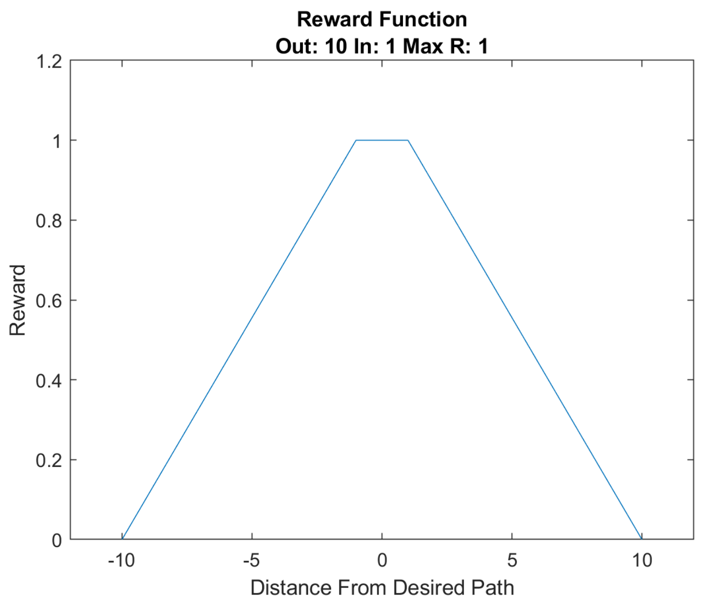


Figure 22. Final Reward Function Plot

The curve depicted in Figure 22, is used to map "x1" which is the distance the simulation is from the desired path to a reward between zero and one. Where the plateau determined by the variable "in" is the user defined tolerance for being close to the desired path. In this example, if "x1" is greater than one or less than negative one, the agent will receive a reward less than the maximum reward which is one. If "x1" is greater than ten or less than negative 10, the agent receives a negative reward, and the simulation is terminated. Since the measurement data being spoofed is inherently noisy, it is nearly impossible for the agent to perfectly match the desired path line, resulting in the agent never receiving the maximum reward. The plateau gives the agent a small band in which to take actions and still maintain the maximum reward.

3. Reset Function

The Reset function listed in Appendix C, Initializes the variables used in the step function at the beginning of each new training episode. It also builds the true, measured, and desired paths based off the parameters set by the user in the main script. The reset function is called by the training environment every time the agent successfully reaches the end of an episode, or if the episode reaches any early termination criteria. For a visual representation of the desired, true, and measured paths generated by the reset function see Figure 12.

C. TESTING

The design, build, and test of the various scripts used in this thesis were structured as follows. First, a functioning Kalman filter and chi-squared monitor were built to form a position estimator. Next, the original Kalman filter program was converted into a step function to allow it to interact with MATLAB's machine learning toolbox. Then, the step function was tested outside of the reinforcement learning environment with a known set of actions to verify that the action space available to the agent can indeed achieve the desired goal. After the action space was validated, the reward function was tuned outside of the reinforcement learning environment to ensure that the agent would receive the correct signals based on its chosen actions. Finally, after all individual components were verified to function as desired, the agent main script was setup to begin training of the agent. The following sections describe the verification testing of the individual components of the algorithm.

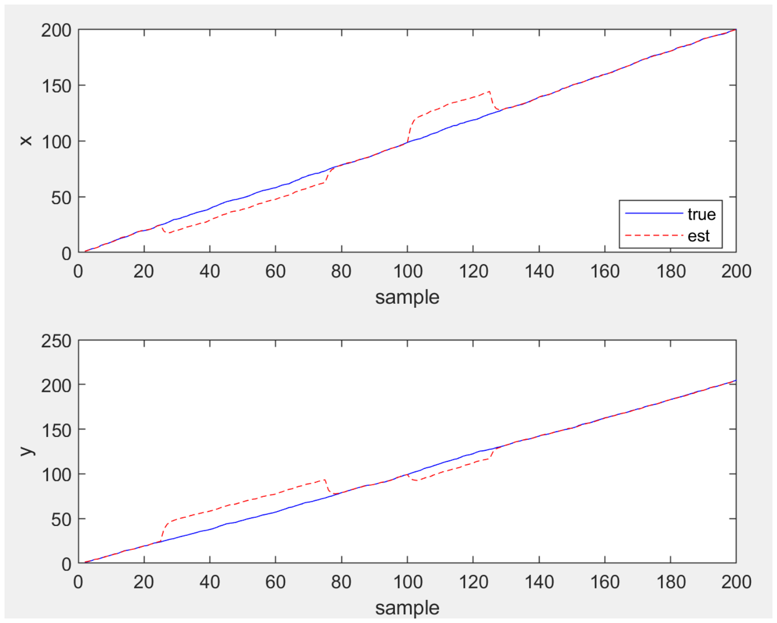
1. Spoofed Kalman Filter and Chi-Squared Verification Testing

The first part of this development was to model a simple double integrator system that incorporates a Kalman filter and performs a chi-squared fault detection test. The script also needed to test the spoofing input, which was done by adding an intentional amount of bias to the true measurement. The original test script can be found in Appendix D. Figure 23 depicts the output. This simulation ran for 200 timesteps with a constant control input of 1 unit in both the x and y directions. Figure 23 depicts the intentional spoofing inputs for the simulation.

```
spoofing_input = [0;0];
if (i>25)
    spoofing_input = [-10.5;20.5];
    %this would be selected by the AI system
end

if (i>75)
    spoofing_input = [0;0];
    %this would be selected by the AI system
end

if (i>100)
    spoofing_input = [+20.5;-10.5];
    %this would be selected by the AI system
end

if (i>125)
    spoofing_input = [0.0;0.0];
    %this would be selected by the AI system
end
```

Figure 23. Spoofing Inputs for Script Testing

The goal was to create to large deviations from the projected path by influencing the measurement with an additional signal form the spoofing input. The first term in the vector "spoofing input" is the x deviation and the second term is the y deviation. The secondary goal was to verify that the chi-squared fault detection test alarms and then recovers after the filter outputs stabilize and then to repeat this cycle for this single simulation. Figure 24 illustrates the true output against the spoofed Kalman filter output based on the spoofing inputs from Figure 23.

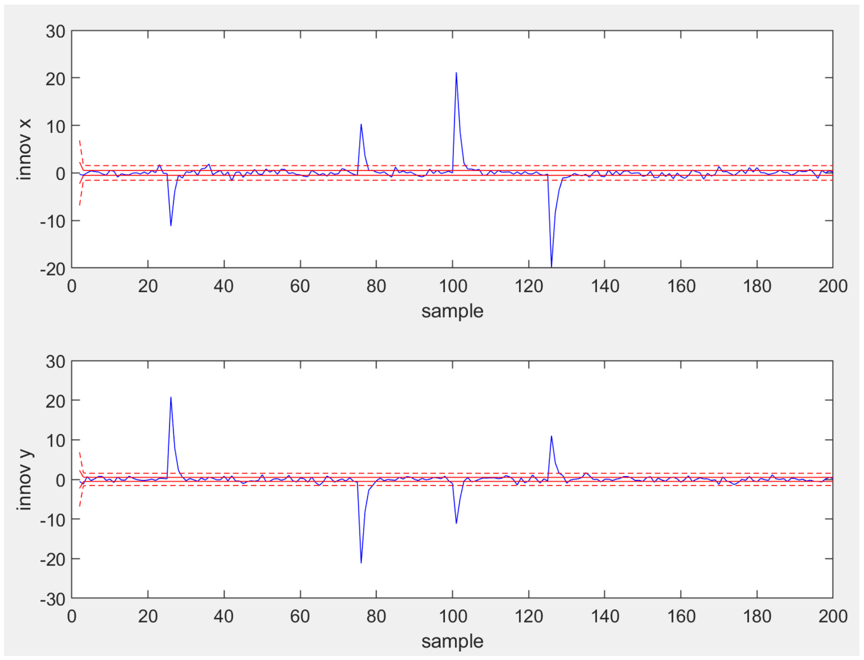


Top plot is the estimated *x* value over time. Bottom plot is the estimated *y* value over time.

Figure 24. Spoofed Kalman Filter Estimations Overlayed with the Original True Path

As shown in Figure 24, the spoofing inputs were successfully inserted into the Kalman filter output, and the estimated course deviated from the true path by the specified amounts. The Kalman filter output returned to mirroring the true path when the spoofing inputs were turned off.

Figures 25 and 26 show that the chi-squared test performed on the innovation of the Kalman filter, performed as expected.



1 sigma is the solid red line. 3 sigma is the dotted red line. The innovation over time is the blue line.

Figure 25. Innovation with 1 Sigma and 3 Sigma Bounds

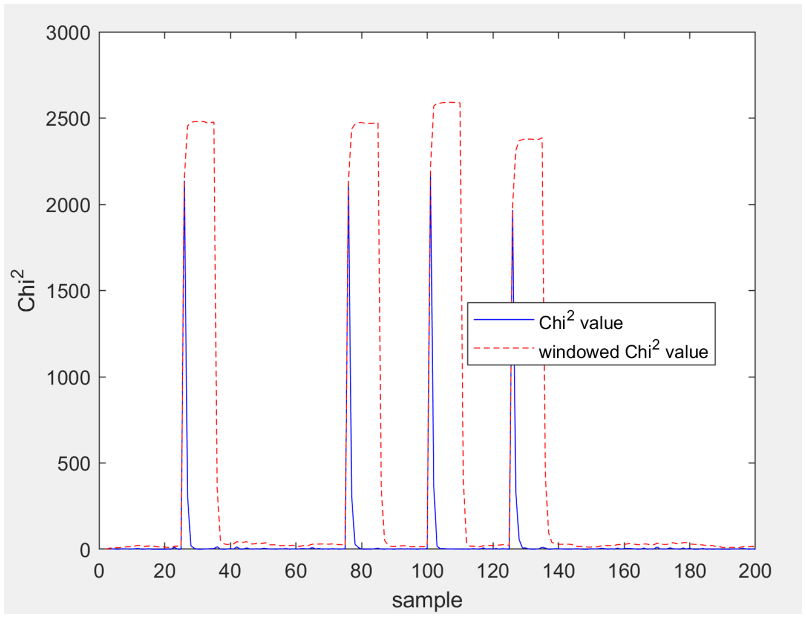


Figure 26. Chi-squared and Chi-squared Windowed Value Plot

In Figure 25, the innovation for each the *x* and *y* components stay within the three sigma standard deviation bounds used as the alarm metric for the chi-squared test until the spoofing input is introduced. At the same time, the corresponding chi-squared value spikes and begins triggering individual step alarms shown in Figure 26.

2. Step Function Verification

With the functionality of the Kalman filter, chi-squared test, and spoofing effects proven, it was necessary to build a step function to interface with MATLAB's RL toolbox. The data was carried over from step to step and iterated for the maximum number of steps in a desired simulation. To verify functionality, prior to introducing the step function to the reinforcement learning environment, the reward function was separated from the Kalman filter spoofing and chi-squared testing of the function.

The agent as discussed in Section A and shown in Table 2, will only have a limited action-space to work with. To verify that the action-space available to the agent will be sufficient for it to reach the desired deviation within the simulation time, an action-space was prescribed to the step function in a limited scenario shown in Figure 27.

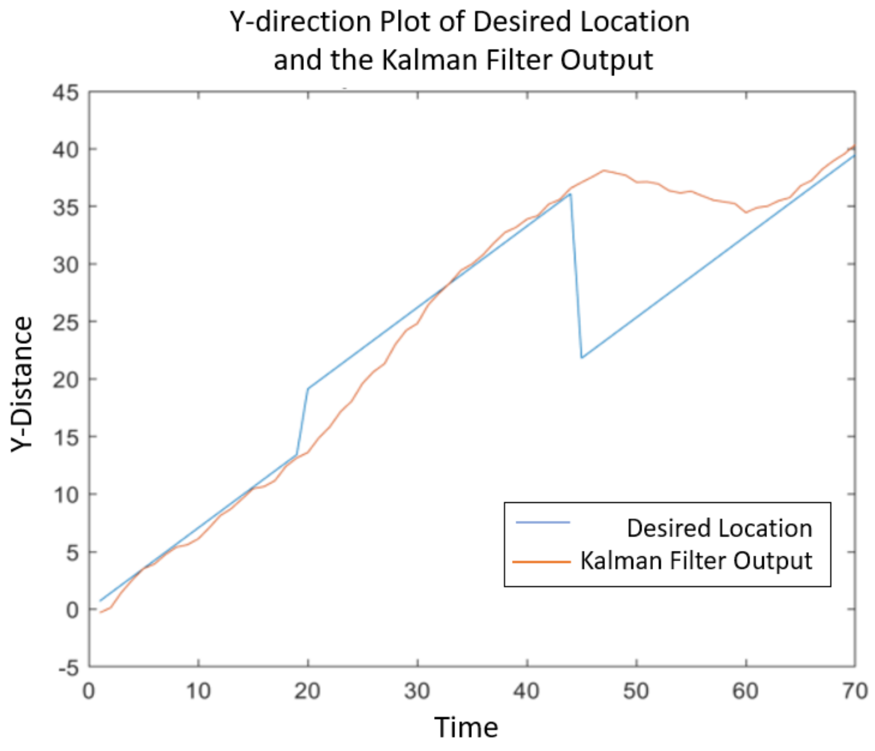
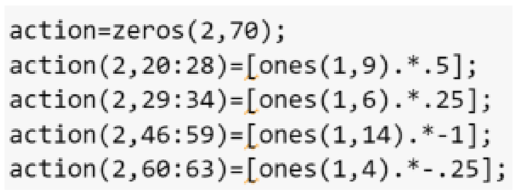


Figure 27. Action-space Testing

In this 70 timestep run, the deviation in the *x* direction was kept constant and only the desired and estimated path of the *y* direction was manipulated. The desired path in Figure 27 was created using a deflection of positive five units from step 20 to step 45, then a sharp negative step of five units off of the intended true path for the remainder of the scenario. The goal was to test that an action-space using the agent's available action sequence could be used to generate a post filter estimated position line that could reach and follow the desired deviations in the desired path before the scenario ended. The following action-sequence was fed to the Kalman filter step function shown in Figure 28 to achieve the successful plot in Figure 27.



```
action=zeros(2,70);
action(2,20:28)=[ones(1,9).*.5];
action(2,29:34)=[ones(1,6).*.25];
action(2,46:59)=[ones(1,14).*-1];
action(2,60:63)=[ones(1,4).*-.25];
```

Figure 28. Sequence for Action-Space Test.

Now that the action space available to the agent has been proven to reach the desired deflection, the chi-squared fault detection test was verified next. The first test was to prove that the alarm state would trigger for innovations that exceeded the threshold of the 2 degrees of freedom chi-squared curve and that the alarm window would activate after a specified number of consecutive alarm states. Figure 29 illustrates a successful instance where the spoofed Kalman filter caused the alarm to raise and the alarm window to activate.

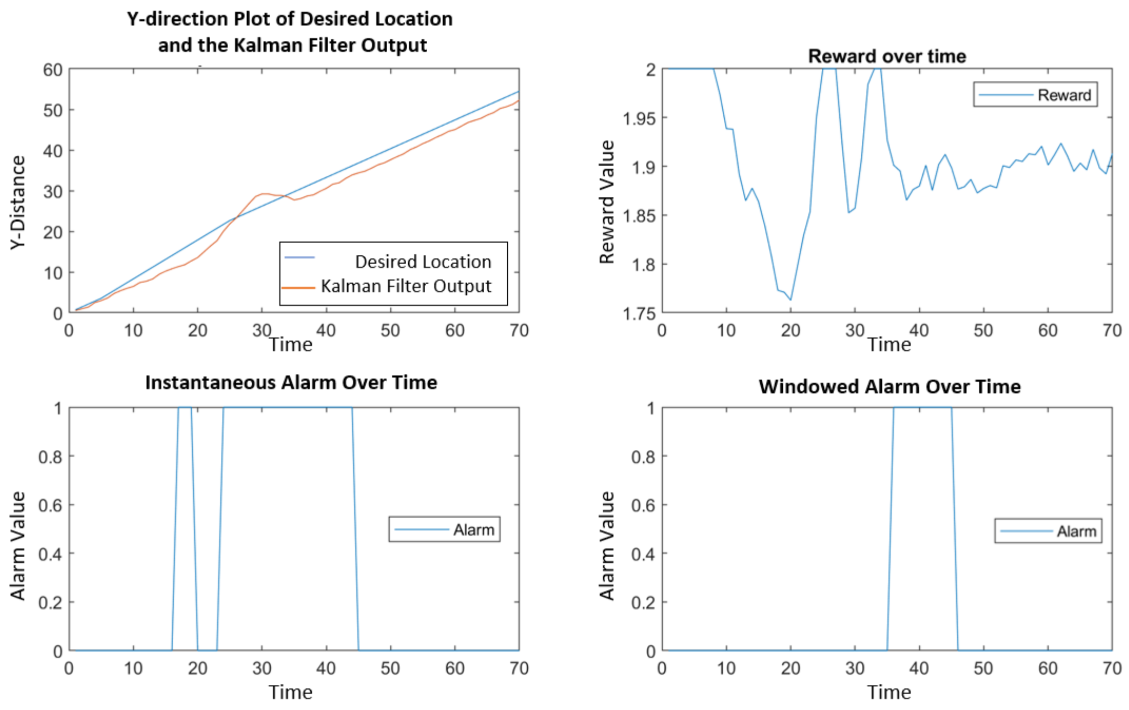


Figure 29. Verification of the Chi-Squared Alarm Logic

In Figure 29, the action-space was a series of random large action taken from step 20 to 35 in the *y* direction only. The windowed alarm threshold was set to 13. The windowed alarm is how many consecutive step alarms the system allows before the real alarm activates. In a real-world system, this is when an operator would be notified of a potential fault in a sensor. The alarm state is continuously active from step 22 to 45. The alarm window was successfully triggered at time step 35 and remained on until the Kalman filter stabilized and the innovation returned to a nominal value. Figure 29 also shows that the windowed alarm did not trigger from steps 16 to 20 when the windowed alarm threshold was not met and restarted counting consecutive alarm states at step 22.

With the functionality of the chi-squared test verified, a test was also conducted to ensure that it is possible for the agent to make it through an episode of training without triggering a windowed alarm. Figure 30 depicts the result of this test.

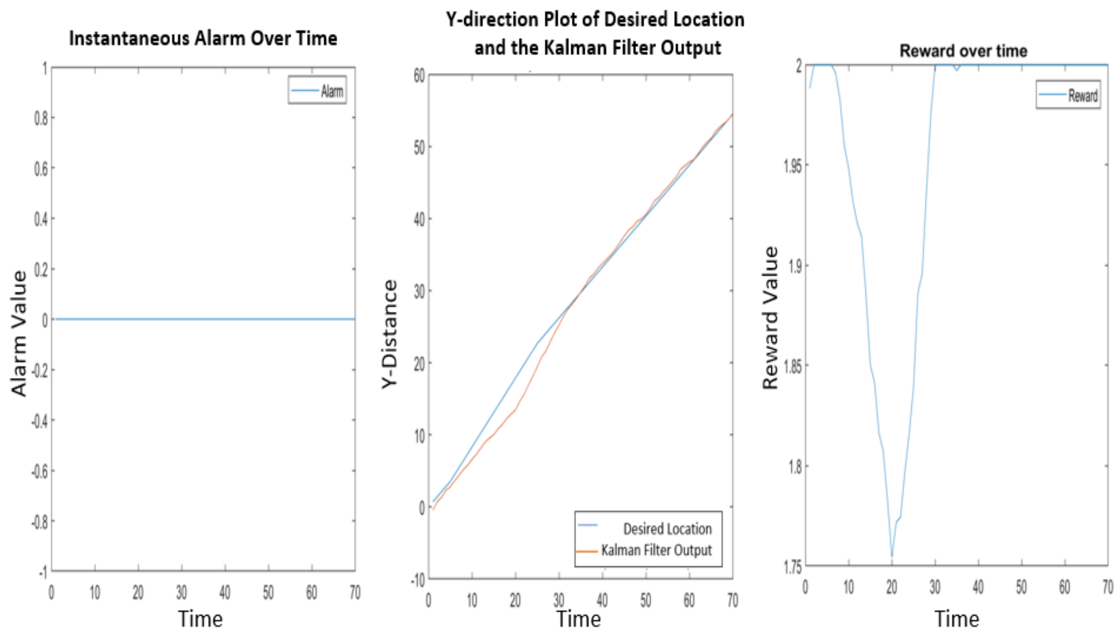


Figure 30. Action-space Test: No Alarm Triggered

In Figure 30, the prescribed action-sequence was able to slowly drive the Kalman filter estimations to stay within a 1 unit bound of the desired path while never triggering a single alarm state. This proves that the agent has the ability to never trigger a windowed alarm. The test also shows that it is feasible to learn to spoof the Kalman filter without triggering any individual alarm states.

3. Reward Function Testing

The final testing performed on the step function prior to implementation in the reinforcement learning environment was crafting a reward function that provides feedback to the agent to learn its intended task. The first part of the reward function computes the distance between the current Kalman filter output “pos hat” and the attacker’s “desired location” in both the *x* and *y* directions individually. The agent then computes the slope of the plateau function based on the user defined inputs depicted in Figure 31.

```
x1 = pos_hat(1,1)-desired_location(1,1);
y1 = pos_hat(2,1)-desired_location(2,1);

b = ((max_R)/(out - in))*in + max_R;
slope = (b - max_R)/in;
```

Figure 31. Kalman Filter Output and Desired Location Comparison

Initially, the comparison between the current filter output and the desired location were combined in a single summed squared distance. However, the result is a scalar, so the agent had no sense of which way it needed to adjust the spoofing signal to reach the desired location. The slope of the plateau is helps to shape the reward for the agent to learn exactly what it is suppose to. The slope gives the agent a better reward as it gets closer to the top of the plateau where it receives the maximum reward as long as it stays within the “In” window. This comparison is depicted in Figure 32.

```
function [reward,IsDone] = reward_calc(x1,b,slope,max_R,in,out)
    IsDone = 0;
    if x1 < -in && x1 >= -out
        reward = slope * x1 + b;
    elseif (-in <= x1) && (x1 <= in)
        reward = max_R;
    elseif x1 > in && x1 <= out
        reward = -slope * x1 + b;
    else
        reward = -1;
        IsDone = 1;
    end
end
```

Figure 32. Reward Function Plateau Generation

If the comparison is outside the bounds of the plateau but inside the bounds of the values of “Out,” the agent will receive a reward for how well it is moving in both the *x* and *y* directions. If the agent does not stay within the curve and exceeds the value of out at any point, the agent is given a negative reward and the simulation terminates early. Strong negative rewards coupled with termination criteria help the agent to understand the bounds of the environment clearly, and also to help it to understand that there are bad actions to choose and to avoid them.

The agent also needed to build an understanding of the instantaneous alarm state and windowed alarm and how to avoid triggering the windowed alarm. Depicted in Figure 33 is the code that provides additional feedback to the agent on the alarm state and windowed alarm that is included in the reward function.

```
if smoothed_Alarm >= 1
    reward_alarm_on = -50;
else
    reward_alarm_on = 0;
end

reward_alarm = 1 - sum_Alarm;

reward = reward_x + reward_y + reward_alarm + reward_alarm_on;

if (IsDone_y + IsDone_x) > 0 || smoothed_Alarm >= 1
    IsDone = 1;
else
    IsDone = 0;
end
```

Figure 33. Reward Calculation and Alarm State

In Figure 33, the “smoothed alarm” and “sum Alarm” values are imported from the step function and are used to provide the agent feedback on the alarm state. If “smoothed Alarm” is triggered, its value will be one, which will give the agent a strong negative reward for tripping the full windowed alarm. To help the agent understand that it is getting closer or farther away from the windowed alarm state, the agent gets a reward for how close it is to “sum alarm,” which is the calculation for smoothed alarm mapped between zero and one. If the agent hasn’t triggered many alarms, “sum alarm” will be small and the agent will receive an additional reward point. As “sum alarm” gets larger the reward from “reward alarm” will get increasingly smaller until the “smoothed alarm” is triggered, and the episode is terminated.

D. SUMMARY

The Above sections describe the methods utilized to develop and verify the environment for the agent and reward function shaping prior to training. The Kalman filter responds to the spoofing input action-space and the chi-squared test functions properly when performed on the innovation of the filter. The reward function is properly shaped to give the agent the constructive feedback on its actions in the environment and gives the agent a range of values that achieve the maximum reward to compensate for the inherently noisy measurement data. In the next chapter, the results of reinforcement learning are presented.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. REINFORCEMENT LEARNING RESULTS

This chapter presents the results of the reinforcement learning experiment. For simplicity, the training scenario was limited to directing the output of the Kalman filter ten units off of the true path in both the *x* and *y* directions to show a proof of concept. After successfully training the agent to deceive the Kalman filter and chi-squared test, a new deviation was given to the agent to test its understanding of the objective, and if it could potentially follow a new desired path without any additional training.

A. FIRST TRAINING EVOLUTION

After successful reward shaping iterations to provide the agent proper feedback on its progress in the environment, the final hyperparameters chosen for the scenario can be found in Appendix A, Section I. For the desired ten-unit deviation, the outside bound of the plateau function was set to 10, the inside bound was set to 1, and the maximum reward was set to 1. This curve, depicted in Figure 22 gives the agent the maximum reward for every step that the agent chooses an action that moves the Kalman filter output within ± 1 of the desired path. The agent can receive a maximum reward per step of three. Where one full point is awarded for following the *x* deviation, another point for the *y* deviation and another point for not triggering alarms. Figure 34 depicts the initial training progress of the reinforcement learning agent. As training progressed, Figure 34 shows that the agent began to receive a greater reward per episode.

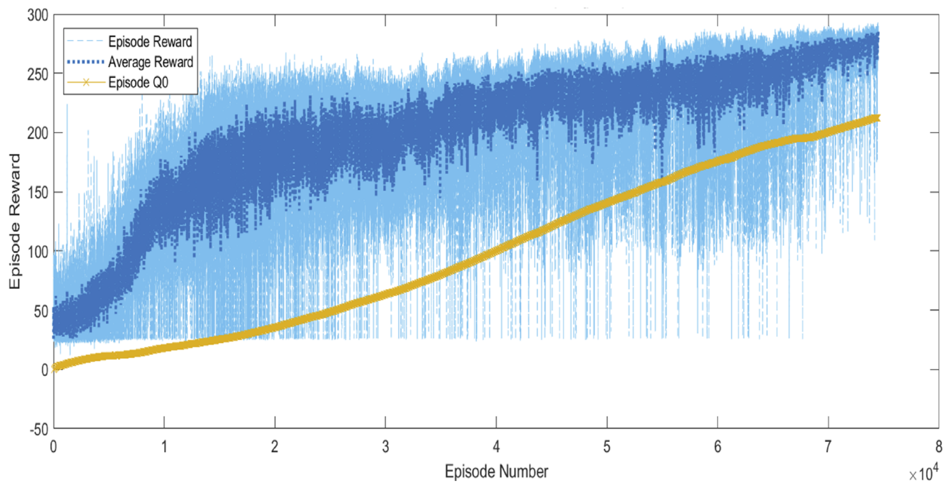


Figure 34. Initial Training Results

To facilitate initial training, the alarm window variable “N window” was set to 30, the stop-training-value was set to 90% of the maximum reward, and the score-average-window-length was set to 10. A large alarm window was used initially to help the agent explore the action-space without triggering a windowed alarm as this would terminate a scenario early and potentially inhibit learning. The maximum reward for an episode is 300. Since there is inherent random noise in the system, and on the measurements, it is nearly impossible for the agent to get a perfect score with the limited discrete action-space chosen. Therefore, setting the stop training value to 90% of the maximum reward is a reasonable goal given the constraints of the environment. The score-average-window-length is how many episodes in a row the agent needs to score above the stop training value. The Agent successfully met the stop training criteria after 74,000 episodes. The episodes with low total rewards are instances where the agent met any of the “IsDone” criteria specified in the reward function of Appendix B. The likely culprit for the noisiness of the data set is the clause in the reward function that returns a negative reward and an “IsDone” flag whenever the distance between the Kalman filter output and the desired path is greater than the bounds of the plateau function. The goal of this clause is to teach the agent to fight to stay within the bounds of the plateau and if it cannot, it receives a negative reward, and the episode terminates early. The noisiness will decrease with future training cycles. Figure 35 shows the results for the initial training cycle

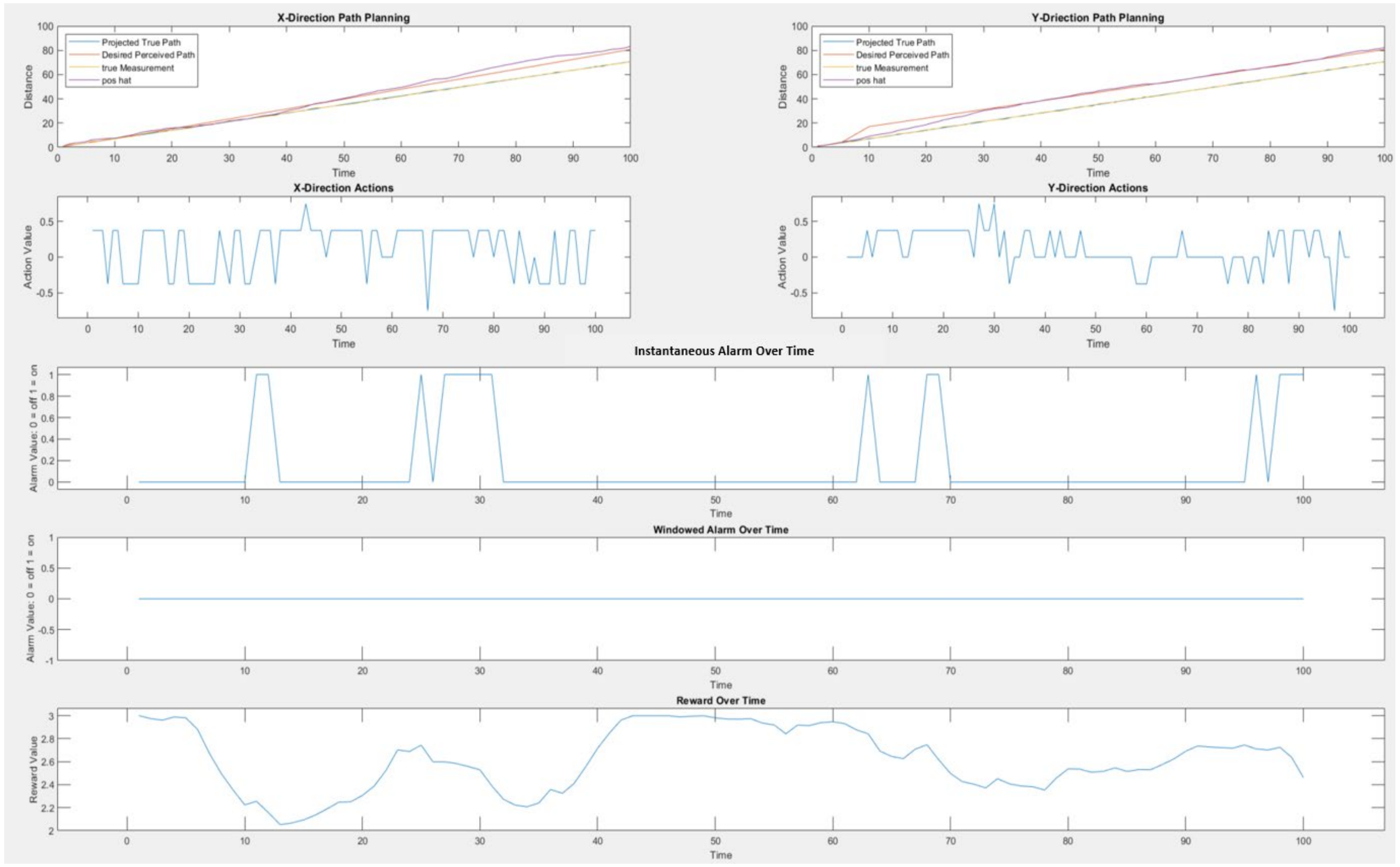


Figure 35. Initial Training Final Episode Results

In Figure 35, The *x* direction path, and actions are depicted in the upper left and the *y* direction path, and actions are on the upper left. The *y* direction desired path is a 10-unit deviation from the projected path starting at step five and maintaining the deviation from step 10 to the end of the episode. The *x* direction desired path is a gradual deviation from the true path, reaching 10 units of deviation by the end of the episode. The purple line is the output of the Kalman filter as it is misled from the blue true path by the actions chosen by the reinforcement learning agent. In Figure 34, it is evident that the agent has learned that the closer it can stay to the desired path, the more often it will receive the maximum reward per step. The agent at the end of the initial training cycle also struggled to stay within ± 1 of the desired path longer than a few steps at a time. Overall, the agent is able to choose action that allow it to stay just outside the bounds of the plateau function.

B. SECOND TRAINING EVOLUTION

Next, the same agent was loaded and trained further. This time, the stopping criteria was made more challenging. The score-average-window-length was set to 30 and the stop training value was set to 95% of the maximum reward. This means that the agent will take the average score of the past 30 episodes at a time, and training will stop once the average meets or exceeds 285 points per episode. In this iteration of training, the alarm window was still set to 30. Now that the agent has learned how to interact with the environment for the duration of an episode, the goal of the second training session was to improve the agent's ability to follow the desired path line with greater accuracy. Figure 36 shows the training progress for the second training cycle.

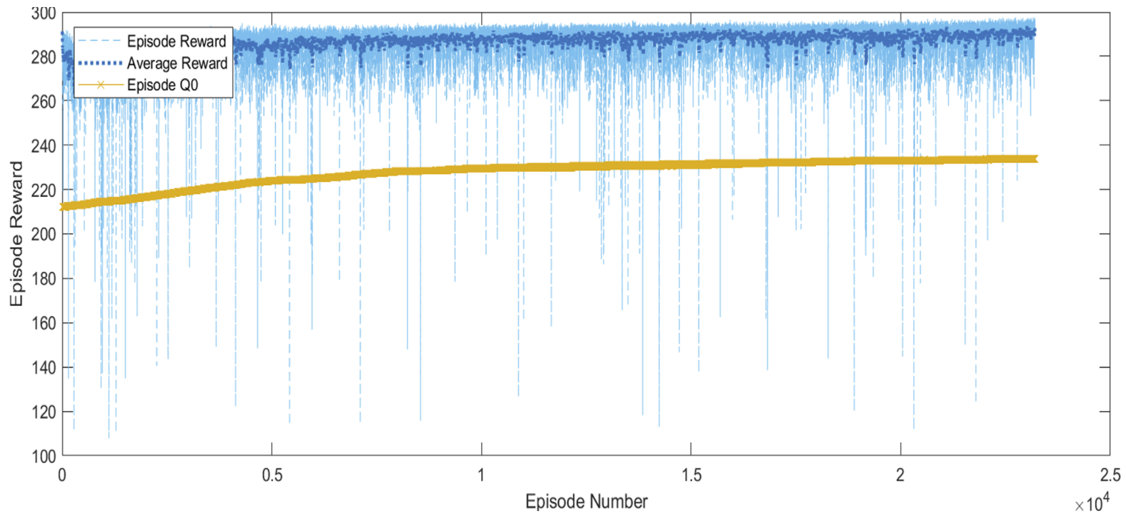


Figure 36. Second Training Cycle

Depicted in Figure 36, the agent continued to fine tune its understanding of the environment and achieved the stricter stopping criteria after 22,000 episodes. This is a significant improvement over the previous training cycles since the training stopped significantly earlier than the first cycle. The episode reward values are significantly less noisy than the first training cycle but, instances of low reward early termination episodes are still present. This is likely due to the previously mentioned issue of the width of the plateau function coupled with the inherent noisiness of the measurement data. Figure 37 shows the final episode results of the second training cycle.

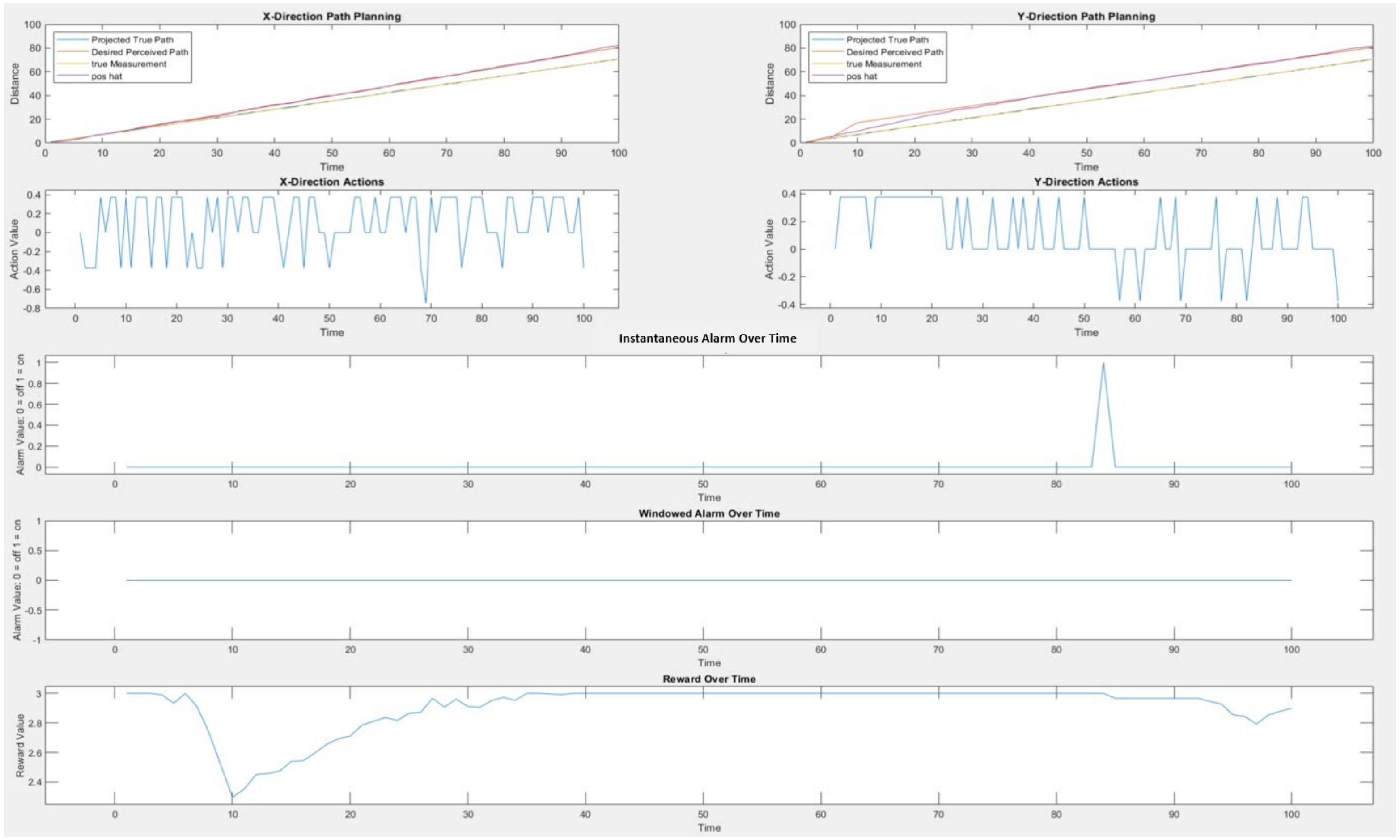


Figure 37. Second Cycle Final Episode Results

In Figure 37, the mislead Kalman filter output tracks the attacker's desired path significantly more accurately. The agent appears to have learned to favor taking the smallest action available at each time. This is most likely to avoid triggering the chi-squared alarm. The agent is able to keep both the x and y deviations within ± 1 of the desired path for a majority of the episode.

C. FINAL TRAINING EVOLUTION

Finally, the same agent was loaded and trained a third time. This time, the stop-training-value was set to 96% of the maximum reward and the alarm window was set to 10. It is evident from Figure 36 and Figure 37, that due to the decrease in step reward value as the agent activates the instantaneous alarm, the Agent has built a reasonable understanding to avoid any number alarms even when the windowed alarm value was set significantly higher. This last training cycle is to confirm the agent's understanding of triggering the windowed alarm and to ensure that the final agent can deceive the Kalman filter without triggering a practical alarm configuration. Figure 38 depicts the training history for the last training cycle.

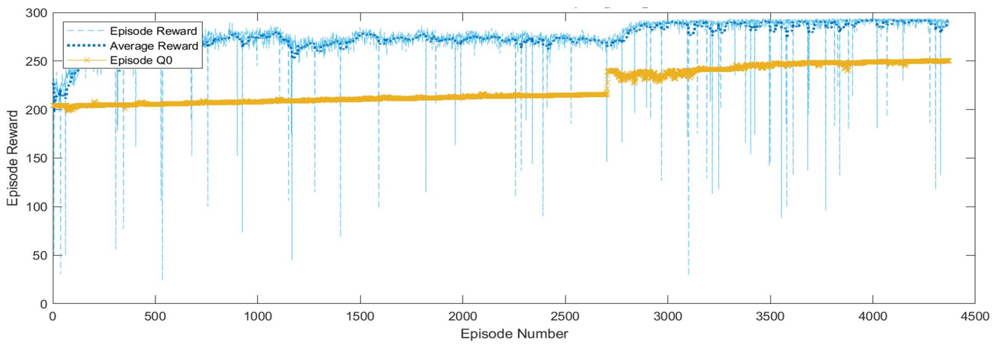


Figure 38. Final Training Cycle History

On the Final training cycle, the agent reached the average reward stopping criteria just short of 4,500 episodes. The variations in episode rewards have significantly diminished form the original training session however, there are still occasional episodes where the agent terminates early. The early terminations at this point in training are likely due to the random system noise and measurement noise impacting the agent's ability to stay withing the reward function curve. Figure 39 shows the final episode results for last training session.

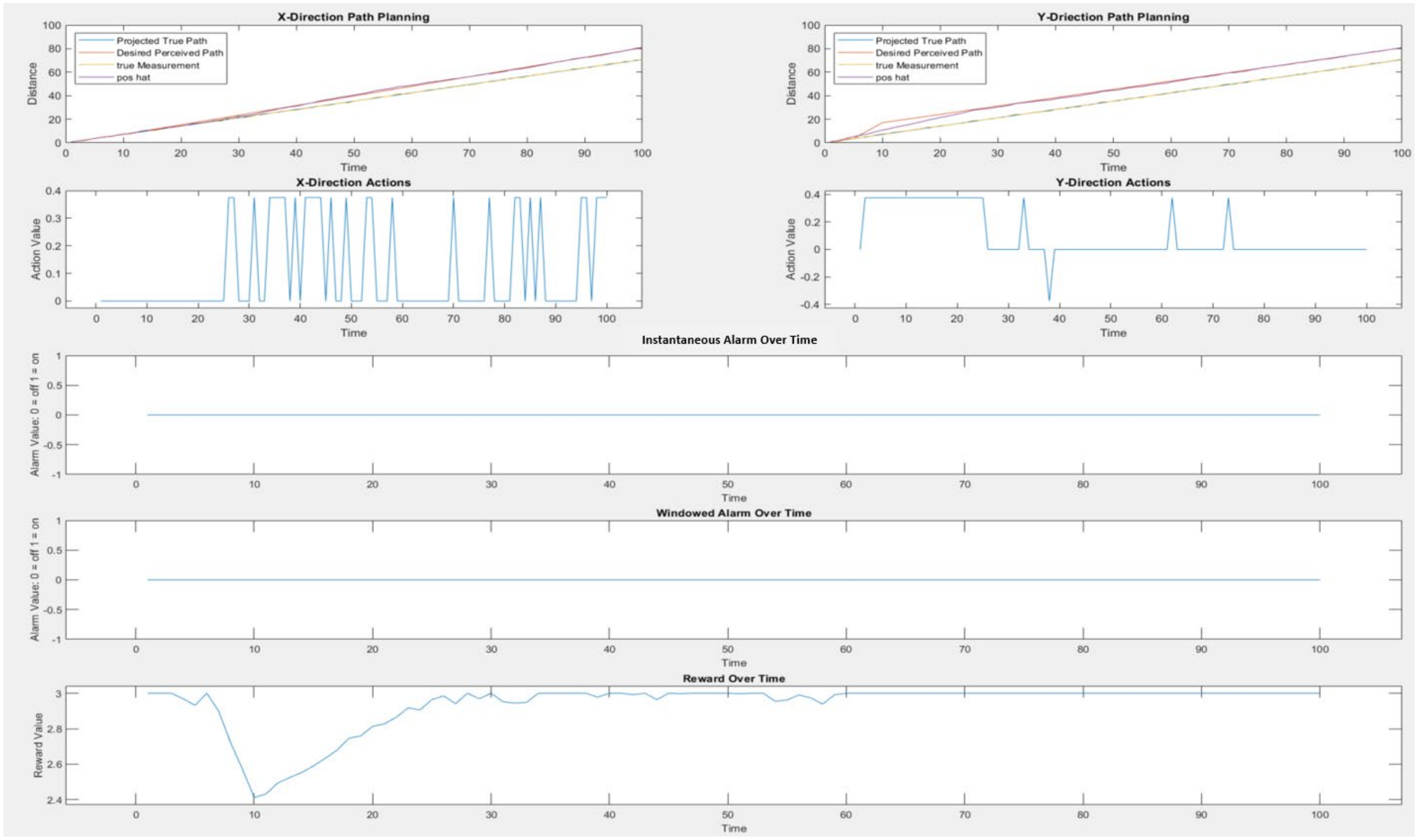


Figure 39. Final Training Cycle Last Episode Results

By the end of the final training session, the agent had optimized its ability to adjust the output of the Kalman filter to follow to desired path without being detected. Outside of the anomalies depicted in Figure 39, the agent was capable of keeping the output of the Kalman filter within ± 1 of the desired path for over half of the total steps in an episode. The anomalies are likely instances where the agent fails to stay within the bounds of the reward function. Figure 39 also highlights an interesting capability of the system. Even given the limited action-space of this experiment, and the short 100 step simulation time, the agent learned to adjust the Kalman filter output without triggering a single alarm. This shows that a more sophisticated model with a highly sensitive chi-squared test could potentially still be deceived by a simple machine learning algorithm.

The last test performed on the agent was to give it an entirely new desired path to see if, without any additional training, the agent could potentially follow a new path. Figure 40 depicts the *y* direction simulation results for the fully trained agent attempting to follow a -10 deviation from projected, desired path.

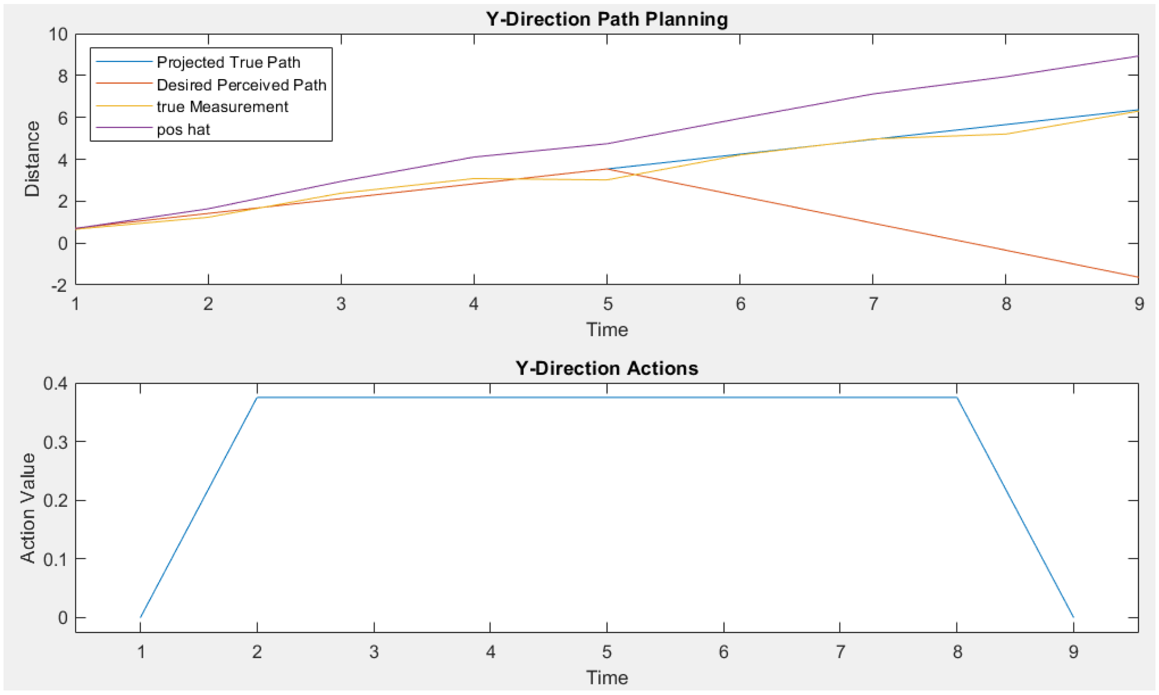


Figure 40. New Desired Path Attempt On Trained Agent

In Figure 40, it is clear that the agent is assuming it is in the scenario it has been trained for and continues to try and move the Kalman filter output +10 units vice -10 units away from the projected path. For the agent to learn any additional desired paths, a new training regimen similar to the three-step process previously performed would be needed.

D. SUMMARY

After three incremental training evolutions, the reinforcement learning agent was able to learn how to successfully mislead an operator by adjusting the Kalman filter output in both the *x* and *y* directions without being detected by the chi-squared fault detection test. There are occasional instances where the agent terminates an episode early, which is likely due in part to the inherent noise of the system and the limited action-space available to the agent. Additional testing proved that a fully trained agent for one scenario does not inherently have the ability to track a new desired path without specified training for the additional task. This is expected, and if an attacker wanted to have multiple options, each case would need to be explicitly trained via curriculum learning.

The combined computational power required to run an agent of this complexity or greater could be extremely difficult to conceal on an intended target system. A potential solution is to run training simulations offline to build a repository of desired deviations and then save the actions chosen by the agent. The action sequence for a specified attack could then be uploaded onto the target system. These actions are the key to altering the Kalman filter. They cause the filter to deviate from its true projected course. An attacker could potentially incorporate the actions chosen by the final trained agent into a much smaller malicious package to implant on a system.

V. CONCLUSION AND FUTURE WORK

A. CONCLUSIONS

Many modern cyber-physical systems rely on Kalman filters to process sensor data [23]. From smart grids, to unmanned aerial vehicles, to spacecraft, Kalman filters are prolific in everyday life. A potential malicious actor could potentially train an algorithm to perform a specific effect on any of the aforementioned systems, with grave consequences. The results of this thesis support continued research into the potential applications of using machine learning to spoof operational cyber-physical system in the eventual pursuit to devise future countermeasures to mitigate the threat. Studying system vulnerabilities from an attackers point of view can provide insight that might be overlooked from a purely defensive perspective.

Specifically, this thesis used a reinforcement learning-based approach for learning how to mislead a Kalman filter that is monitored using a chi-squared fault detection test. The simulation utilized MATLAB's reinforcement learning toolbox to build a proof-of-concept environment to train a reinforcement learning agent to inject spoofing inputs to mislead the operator of a line following robot. The environment and reward function of the simulation were carefully crafted to provide the agent feedback on its interactions with the environment to accomplish the task. The methodology section of this document describes in detail how the unique components of the environment were verified. The research and analysis presented show that there is significant potential for using reinforcement learning to mislead complex systems that rely on Kalman filters for control system monitoring. The results show that with three training sessions totaling 37 hours of combined training time, a reinforcement learning algorithm, even one with a limited action-space, can be taught not only to deceive a Kalman filter, but also to never trigger a single alarm state. This work focused on a specialized task: training the agent to perfect one specific attack pattern. A more generalized approach may have yielded more versatile utility, however, the cost of training time would have risen exponentially.

B. RECOMMENDATIONS FOR FUTURE WORK

There are several potential future research applications. The first, with access to a supercomputer capable of significant parallel processing, investigates if a more general form of the algorithm can be trained to handle a larger range of desired attacks vice one attack per training cycle. Proving the effectiveness of a generalized form of this algorithm would help to show the overall cyber lethality of this machine learning algorithm. A generalized algorithm will take significant time to train, however, an attack with a real-world data set and access to a supercomputer could potentially train an agent to be capable of multiple desired effects on a cyber-physical system.

Another area for future work is to train an agent in a simulated environment utilizing a real-world dataset. Acquiring input data to a real-world systems' Kalman filter and the settings of its fault detection monitor, then use this research as a model to design a new reinforcement learning agent to deceive the operator of a real system in a simulated environment.

A final area for potential future work would be to train an agent to mislead a controlled real-world system, like an unmanned aerial vehicle, then devise a way to upload the agent or specific action-sequences to test the effectiveness of the malicious attack in practice. This could help to provide key insight into where a potential attacker may try to hide malicious Kalman filter spoofing code and give protection agencies insights into how to harden against these attacks.

APPENDIX A. MAIN KALMAN FILTER SPOOFING SCRIPT

Main Script for Kalman Filter Spoofing

The following sections of code build the agent and the environment for testing and evaluation of RL applications on Kalman Filter Spoofing.

Required Scripts:

Kalman_step_func.mlx : step the environment 1 step through a Kalman Filter with a Chi-squared test performed on the innovation and gives the agent a reward based on its actions

Kalman_sys_startup.mlx : Initializes the variables for the agent, environment, and step function for each episode of training.

Section I: Initialization of Parameters

Written By: LT Dylan A. Bonitz, USN,

LTC Brian M. Wade, USA

Dr. Mark Karpenko

Completed: 30 Mar 2022

This section contains the simulation inputs that the user can define, the hyperparameters for the neural networks, and the training options.

```
close all; clc; clear;
```

Section I.a: Simulation Inputs

```
rng(42); % Random Seed

time_end = 100; % max time of sim
speed = 1; % speed of vechicle
initial_ang = 45; % projected angle of travel degrees

noise_reduction_factor = 1; % Controls the strength of the noise, values
from 0-1 only
```

```
 dists = [0, 10; 10, 10;];
 % n x 2 array of n number of (x,y) shifts from projected course

 time_targets = [5, 10; 10, time_end;];
 % n x 2 array of times when the shifts where each n row is (start_time,
end_time)

 action_range = 5;
 % choose the range of available action for the agent to learn.
 % the more actions, the longer the training will take

 action_space_reduction_factor=0.75;
```

Section I.b: Reward Function Options

```
 out = 10; % The x-intercepts of the plateau reward funciton ex. out = 10
builds a
 % reward funciton with x-intercepts at -10 and 10.

 in = 1; % the range of the plateau. a value of one sets the plateau from
-1 to 1.
 % in the “in” range, the agent will receive the max_R value

 max_R = 1; % the max reward for a given timestep that defines the hight
of the plateau.
 % a value of 1 sets the plateau at 1.
```

Section I.c: Critic Options

```
 critic_FC1 = 50; % Number of nodes in the first layer
 critic_FC2 = 20; % Number of nodes in the second layer
 learn_rate_critic = 0.0001;
 grad_threshold_critic = 1; % 0.5
```

Section I.d: Actor Options

```
 actor_FC1 = 150; % 50
 actor_FC2 = 100; % 30
 actor_FC3 = 50; % 20
 learn_rate_actor = 0.00001;
 grad_threshold_actor = 1; % 0.5
```

Section I.e: Agent Options

```
DiscountFactor = 0.99;
% a high discount factor makes the agent look farther into the future

ExperienceHorizon = 150; % Looks out over all steps
MiniBatchSize = 64;
```

Section I.f: Training Options

```
MaxEpisodes = 500000;
steps_per_edisode = time_end;
doTraining = true;
Load_agent = true;
want_parallel = false;
ScoreAveragingWindowLength = 30; % Number of episodes to average together
Max_reward_stop = .99; % value to scale strop training value
StopTrainingValue = 3*(time_end*max_R)*Max_reward_stop;
% Verbose = true;
```

Section I.g: Multi-Core Training

```
%Start worker pool
if want_parallel == true
 poolobj = gcp('nocreate'); % If no pool, do not create new one.
 %delete(poolobj)
 if isempty(poolobj)
   poolsize_want = round(.9*feature('numcores'));
   if poolsize_want < feature('numcores')
     poolsize = poolsize_want;
   else
     poolsize = feature('numcores')-1;
   end
   parpool('local',poolsize);
 else
   poolsize = poolobj.NumWorkers;
 end
end
```

Section II: Action Space and Environment Setup

Section II.a: Action and Observation Spaces

```
 numObs = 8; % number of state variables
 ObservationInfo = rlNumericSpec([numObs 1]);
 ObservationInfo.Name = 'KF Info';
 ObservationInfo.Description =
['difference_x','difference_y','p_11',"p_22",...
  "innovation",'l','Alarm'];

 % This loop sets up the discrete action space. The Algoithm can choose to
a spoof
 % between negative 1 and 1.
 % 'action_range' input. An action_range of 5 means that x and Y can have
 % an input of [-1,-0.5, 0, 0.5, 1].
 % The action space reduction factor scales the action space.
 ii=0;
 action_space = cell(action_range);
 for i = -1:2/(action_range-1):1
  ii=ii+1;
  jj=0;
  for j = -1:2/(action_range-1):1
    jj=jj+1;

    action_space{ii,jj}=[i;j].*action_space_reduction_factor;
  end
 end

 ActionInfo = rlFiniteSetSpec(action_space);
 ActionInfo.Name = 'Spoofing Action';
 num_actions = length(ActionInfo.Elements);
```

Section II.b: Reinforcement Learning Environment Setup

```
 StepHandle = @(Action,LoggedSignal) Kalman_step_func...
  (Action,LoggedSignal,out,in,max_R,time_end);

 ResetHandle = @() kalman_sys_startup...
  (time_end,initial_ang,dists,time_targets,speed,noise_reduction_factor);

 env = rlFunctionEnv(ObservationInfo,ActionInfo,StepHandle,ResetHandle);
```

Section III: Actor, Critic, and Agent Setup

Section III.a: Critic Network Setup and Critic Training Options

```
criticNetwork = [
 sequenceInputLayer(numObs,”Name”,’state’)
 fullyConnectedLayer(critic_FC1,’Name’,’critic_FC1’)
 leakyReluLayer(‘Name’,’critic_reLu1’)
 lstmLayer(critic_FC2,’Name’,’critic_FC2’)
 leakyReluLayer(‘Name’,’critic_reLu2’)
 fullyConnectedLayer(1,’Name’,’critic_value’)
];

criticOpts= rlRepresentationOptions(‘LearnRate’,learn_rate_critic,...
 “GradientThreshold”,grad_threshold_critic);

critic=rlValueRepresentation(criticNetwork,ObservationInfo,...
 “Observation”,{‘state’},criticOpts);
```

Section III.b: Actor Network Setup and Actor Training Options

```
actorNetwork = [
 sequenceInputLayer(numObs,”Name”,’state’)
 fullyConnectedLayer(actor_FC1,’Name’,’actor_FC1’)
 reluLayer(‘Name’,’actor_reLu1’)
 lstmLayer(actor_FC2,’Name’,’actor_FC2’)
 leakyReluLayer(‘Name’,’actor_reLu2’)
 lstmLayer(actor_FC3,’Name’,’actor_FC3’)
 leakyReluLayer(‘Name’,’actor_reLu3’)
 fullyConnectedLayer(num_actions,’Name’,’actor_FC4’) %one node per action
 softmaxLayer(‘Name’,’action_prob’)
];

actorOpts= rlRepresentationOptions(‘LearnRate’,learn_rate_actor,...
 “GradientThreshold”,grad_threshold_actor);

actor=rlStochasticActorRepresentation(actorNetwork,ObservationInfo,...
 ActionInfo,’Observation’,{‘state’},actorOpts);
```

Section III.c: Agent Setup, Training and File Save Options

```
agentOptions = rlPPOAgentOptions(...
 ‘DiscountFactor’, DiscountFactor, ...
 ‘experienceHorizon’, ExperienceHorizon,...
 ‘MiniBatchSize’,MiniBatchSize);
```

```
if want_parallel == true
 agentOpts.ExperienceHorizon = steps_per_edisode;
end

agent = rlPPOAgent(actor, critic, agentOptions);

agent_folder = ‘saved_agents’;
image_folder = ‘images_3’;
agentName = ‘trained_PPO_agent_1’;

Verbose = false;
Plots = ‘training-progress’;

%Create agent and image folders for saved agents and images
if ~exist(agent_folder, ‘dir’)
 mkdir(agent_folder)
end

if ~exist(image_folder, ‘dir’)
 mkdir(image_folder)
end

%Setup Training for Agent
saved_agent_name = agentName;
SaveAgentDirectory = fullfile(agent_folder,saved_agent_name);

trainOpts = rlTrainingOptions(...
 ‘MaxEpisodes’, MaxEpisodes,...
 ‘MaxStepsPerEpisode’, steps_per_edisode,...
 ‘Verbose’, false,...
 ‘Plots’, Plots,...
 ‘Verbose’, Verbose,...
 ‘StopTrainingCriteria’,’AverageReward’,...
 ‘SaveAgentDirectory’, ‘trainedACagent’,...
 ‘StopTrainingValue’, StopTrainingValue, ...
 ‘ScoreAveragingWindowLength’, ScoreAveragingWindowLength,...
 ‘SaveAgentDirectory’, SaveAgentDirectory);
```

Section IV: Train, Test, and Visualize the Agent

Section IV.a: Parallel Processing

```
if want_parallel == true
 trainOpts.UseParallel = true;
```

```
  trainOpts.ParallelizationOptions.Mode = “async”;
  trainOpts.ParallelizationOptions.DataToSendFromWorkers = “experiences”;
  trainOpts.ParallelizationOptions.StepsUntilDataIsSent = ...
    agentOptions.ExperienceHorizon;
end
```

Section IV.b: Train and/or Test the Agent

This section trains a new or existing agent based on the load_agent check. In both cases, once training is complete, the agent is saved and a plot of the training history is generated and saved.

```
if Load_agent == true
 load(‘saved_agents/trained_PPO_agent_1.mat’,’agent’);
end

if doTraining == true

 trainingStats = train(agent, env, trainOpts);
 save(trainOpts.SaveAgentDirectory, ‘agent’);

 figure()
 plot(trainingStats.EpisodeIndex, trainingStats.EpisodeReward,...
   ‘--’,’Color’,[0.3010 0.7450 0.9330])
 hold on
 plot(trainingStats.EpisodeIndex, trainingStats.AverageReward,...
   ‘:’,’Color’,[0 0.4470 0.7410], ‘LineWidth’,2)
 plot(trainingStats.EpisodeIndex, trainingStats.EpisodeQ0,...
    ‘-x’,’Color’,[0.9290 0.6940 0.1250])
 hold off
 xlabel(‘Episode Number’)
 ylabel(‘Episode Reward’)
 title(strcat(‘Reward Training History for ‘, agentName, ‘ Agent’))
 legend(‘Episode Reward’, ‘Average Reward’, ‘Episode Q0’,...
   ‘location’, ‘northwest’)

 image_file = strcat(‘TrainingHistory_’, agentName, ‘.png’);
 image_save_path = fullfile(image_folder,image_file);
 set(gcf,’position’,[50,50,1200,400])
 saveas(gcf,image_save_path)
else
 load([‘agents\trained_AC_agent.mat’]);
end
```

Section IV.c: Last Episode of Training Visualization

This section is used to visualized any of the stored data in the structure logged signals. This section is primarily used to visualize the reward over time vs. the path planning plot.

```
z_true=env.LoggedSignals.z_true’;
desired_location= env.LoggedSignals.desired_location’;
projected_location=env.LoggedSignals.x_true’;
pos_hat=env.LoggedSignals.pos_hat’;
REWARD=env.LoggedSignals.reward;
sum_alarm_hist = env.LoggedSignals.smoothed_alarm;

figure()
subplot(1, 3, 1)
plot(1:1:time_end,projected_location(:, 2))
hold on
plot(1:1:time_end,desired_location(:, 2))
hold on
plot(1:1:time_end,z_true(:, 2))
hold on
plot(1:1:time_end,pos_hat(2:end, 2))
hold off
title(‘Path Planning’)
legend(‘Projected True Path’, ‘Desired Perceived Path’,...
 ‘true Measurement’,’pos hat’, ‘location’, ‘northwest’)

subplot(1,3,2)
plot(1:1:time_end,REWARD)
title(‘reward over time’)

subplot(1,3,3)
plot(1:1:time_end,sum_alarm_hist)
title(‘Alarm strength over time’)

image_file = strcat(‘Sim_outcome_’, agentName, ‘.png’);
image_save_path = fullfile(image_folder,image_file);
set(gcf,’position’,[50,50,1200,400])
saveas(gcf,image_save_path)
```

APPENDIX B. ALGORITHM STEP FUNCTION

```
function [NextObs,reward,IsDone,LoggedSignal] = Kalman_step_func...
 (action,LoggedSignal,out,in,max_R,time_end)
```

Section I: Function Constants

```
 A = eye(2);    % linear system A matrix
 B = eye(2);    % linear system B matrix
 C = eye(2);    % linear system C matrix
 D = [0,0;0,0];  % linear system D matrix
 Q = 0.1*eye(2);  % process noise covariance
 R = 0.1*eye(2);  % measurement noise covariance
 u = [1;1];    % Constant Control input
```

Section II: Unpack LoggedSignal Structure

```
 pos_hat = LoggedSignal.pos_hat(:,end);
 % Previous estimiated position

 i = length(LoggedSignal.pos_hat(1,:)); % Step Counter

 desired_location = LoggedSignal.desired_location;
 % Vecotr containing the desired deviation line

 p_11 = LoggedSignal.p_11(1,end);
 %Upper left Diagonal of the Covariance Matrix P

 p_22 = LoggedSignal.p_22(1,end);
 % Bottom Right Diagonal of the COvariance Matrix P

 P = [p_11,0;0,p_22]; % Re-constructed Covariance MAtrix

 x_true = LoggedSignal.x_true(:,i);
 z_true = LoggedSignal.z_true(:,i);
```

Section III: Step the Kalman Filter

```
    LoggedSignal.spoofing_input_buffer =
LoggedSignal.spoofing_input_buffer...
      (:,end)+action;
    % Stores the cumulative spoofing inputs and adds the current action

    spoofing_input = LoggedSignal.spoofing_input_buffer;


    pos_spoof = z_true + spoofing_input;
    %add the injected spoof to the true measurement

    LoggedSignal.pos_spoof(:,i) = pos_spoof;

    %%%%%
    %step the kalman filter


    [pos_hat, P, innovation, V_innovation] = ...
      KF(pos_spoof, pos_hat, P, u, A, B, C, D, Q, R);

    if i==1
      LoggedSignal.innovation(:,end) = innovation;

    else
      LoggedSignal.innovation(:,end+1) = innovation;

    end


    LoggedSignal.pos_hat(:,end+1) = pos_hat;


    LoggedSignal.p_11(1,end+1) = P(1,1);


    LoggedSignal.p_22(1,end+1) = P(2,2);


    p_11 = P(1,1);
    p_22 = P(2,2);
```

Section IV: Implement the Chi-squared Fault Detector

```
N_window = 30;
N_chi = 2*N_window; %the Chi^2 DOF N_chi is equal to the dimension
           %of the innovation, in this case 2, times the
           %length of the data window

PD = 0.997;  % Probability the system detects a bad when it occurs

l= innovation'*V_innovation^(-1)*innovation; %l is the Chi^2 value

LoggedSignal.l(1,end+1) = l;
l = LoggedSignal.l;
idx_lo = max(1,i-N_window+1);
idx_hi = i;
cl = sum(l(idx_lo:idx_hi));  %windowed sum fof Chi^2 values
```

Section V: Impliment the Alarm-state and Windowed Alarm

```
if (cl>gammaincinv(PD,N_chi,'lower'))
  Alarm = 1; %raise an alarm
else
  Alarm = 0;
end
LoggedSignal.Alarm(1,end+1) = Alarm;
l = l(1,end);

%alarm shouldd be on for at least N_window+2 consequtive samples

idx_lo = max(1,i-(N_window+2)+1);
sum_Alarm = sum(LoggedSignal.Alarm(idx_lo:idx_hi))/(N_window+2);

if (sum_Alarm>=1)
  smoothed_Alarm = 1;
else
  smoothed_Alarm = 0;
end

LoggedSignal.smoothed_alarm(1,end+1) = smoothed_Alarm;
```

Section VI: Reward Calculation and State Vector Composition

```
desired_location = desired_location(:,i);

[reward,IsDone,x1,y1]=reward_func...
 (i,pos_hat,desired_location,out,in,max_R,time_end,...
 sum_Alarm, smoothed_Alarm);

LoggedSignal.x_difference(:,end+1) = x1;
x_difference = LoggedSignal.x_difference(:,end);

LoggedSignal.y_difference(:,end+1) = y1;
y_difference = LoggedSignal.y_difference(:,end);

LoggedSignal.State = ...
 [x_difference;y_difference;p_11;p_22;innovation;l;sum_Alarm];
State = LoggedSignal.State;
NextObs = State;

if i == 1
 LoggedSignal.reward=reward;
else
 LoggedSignal.reward(i)=reward;
end
```

Step Function Nested Functions

Kalman Filter Function

```
   function [pos_hat_new, P_new, innovation, V_innovation] =...
       KF(z, pos_hat, P, u, A, B, C, D, Q, R)

     %Predict
     pos_hat = A*pos_hat+B*u;
     P = A*P*A’+Q;

     %Update
     Vk = (C*P*C’+R);
     V_innovation = Vk;
     Kk = P*C’*Vk^(-1);
     innovation = z-C*pos_hat-D*u;
     pos_hat_new = pos_hat+Kk*innovation;
```

```
    %P_new=P-Kk*C*P;
    P_new = (eye(2)-Kk*C)*P*(eye(2)-Kk*C)’+Kk*R*Kk’;

  end
```

Reward Function

```
function [reward,IsDone,x1,y1]=reward_func...
    (pos_hat,desired_location,out,in,max_R, sum_Alarm, smoothed_Alarm)


  x1 = pos_hat(1,1)-desired_location(1,1);
  y1 = pos_hat(2,1)-desired_location(2,1);


  b = ((max_R)/(out - in))*in + max_R;
  slope = (b - max_R)/in;

  [reward_x,IsDone_x] = reward_calc(x1,b,slope,max_R,in,out);

  [reward_y,IsDone_y] = reward_calc(y1,b,slope,max_R,in,out);

  if smoothed_Alarm >= 1
    reward_alarm_on = -50;
  else
    reward_alarm_on = 0;
  end

  reward_alarm = 1 - sum_Alarm;

  reward = reward_x + reward_y + reward_alarm + reward_alarm_on;

  if (IsDone_y + IsDone_x) > 0 || smoothed_Alarm >= 1
    IsDone = 1;
  else
    IsDone = 0;
  end


  function [reward,IsDone] = reward_calc(x1,b,slope,max_R,in,out)
    IsDone = 0;
    if x1 < -in && x1 >= -out
      reward = slope * x1 + b;
    elseif (-in <= x1) && (x1 <= in)
```

```
        reward = max_R;
      elseif x1 > in && x1 <= out
        reward = -slope * x1 + b;
      else
        reward = -1;
        IsDone = 1;
      end
    end

  end

end
```

APPENDIX C. ALGORITHM RESET FUNCTION

Kalman Filter Spoof Initialization Function

This function initializes the parameters of the Kalman filter step function, and the variables that are used to create the environment for the agent

Section I: Parameter Initialization and State-space Assembly

- Initializes the parameters utilized by the step function.
- Runs Section II and builds the projected (true estimate) and Desired path (Deviation from true).
- Computes the true measurement path based of the true estimate.
- Compiles the state space that the agent observes.

```
function [InitialObservation,LoggedSignal] = kalman_sys_startup...
 (time_end,initial_ang,dists,time_targets,speed,noise_reduction_factor)


C = eye(2);   %linear system C matrix

D = [0,0;0,0]; %linear system D matrix

R=0.1*eye(2);  %measurement noise covariance

u=[1;1]; %assume a constant control input

P = 5*eye(2); %initial Kalman filter covariance

p_11 = (P(1,1)); % upper left diagonal of the covariance matrix

p_22 = (P(2,2)); % lower left diagonal of the covariance matrix

time = 1:1:time_end; % total scenario runtime in seconds



[projected_location, desired_location] = make_paths...
 (initial_ang, speed, dists, time_targets, time);
 x_true=projected_location';
 desired_location=desired_location';

 for j = 1:time_end

  z_true(:,j)  = C*x_true(:,j) + D*u + chol(R)*...
    (randn(2,1).*noise_reduction_factor); % Noisy true measurements
 end



 pos_hat = z_true(:,1)+[0.5*randn;0.5*randn]; %first estimated position
guess at time zero

 % Zeroise tracked data for the first observation at time zero.
 innovation=[0;0];
 sum_Alarm=0;


 l=0;


 x_difference = 0;
 y_difference = 0;

 LoggedSignal.pos_hat = pos_hat;
 LoggedSignal.x_true = x_true;
 LoggedSignal.z_true = z_true;
 LoggedSignal.x_difference = 0;
 LoggedSignal.y_difference = 0;
 LoggedSignal.p_11 = p_11;

 LoggedSignal.p_22 = p_22;
 LoggedSignal.innovation = innovation;
 LoggedSignal.l = l;
 LoggedSignal.Alarm = 0;
 LoggedSignal.desired_location = desired_location;
 LoggedSignal.spoofing_input_buffer=[0;0];
 LoggedSignal.smoothed_alarm = 0;
 LoggedSignal.reward = 0;

 LoggedSignal.State = [x_difference;y_difference
;p_11;p_22;innovation;l;sum_Alarm];
 %Contains the data that is presented to the agent
 InitialObservation = LoggedSignal.State; % first system observation at
time zero
 % This is the first set of data points observed by the agent
```

Section II: Projected and Desired Path Generation

The function below builds the projected and desired paths. The measured path is built off the projected path in lines 27–30.

```
  function [projected_location, desired_location] = make_paths...
      (initial_ang, speed, dists, time_targets, time)

    %make projected path
    projected_location = zeros(length(time), 2);
    for i = 1:length(time)
      offset = [0, 0];
      path = find_xy_from_angle(initial_ang, offset, i, speed);
      projected_location(i,:) = path;
    end

    %make desired path
    desired_location = zeros(size(projected_location));
    present_shift = [0, 0];
    shift = present_shift;
    num_shifts = size(time_targets, 2);
    current_shift = 1;

    for i = 1:length(time)
      if current_shift <= num_shifts
        if i >= time_targets(1,current_shift)
          x_shift = interp1(time_targets(current_shift,:),...
            [present_shift(1), dists(current_shift, 1)], i);

          y_shift = interp1(time_targets(current_shift,:),...
            [present_shift(2), dists(current_shift, 2)], i);
          shift = [x_shift, y_shift];

          if i == time_targets(current_shift, 2)
            current_shift = current_shift + 1;
            present_shift = shift;
          end
        end
      end
      path = find_xy_from_angle(initial_ang, shift, i, speed);
      desired_location(i,:) = path;
    end

  end


    function [path] = find_xy_from_angle(ang, offset, time_step, speed)
      dist = speed*time_step;
      x_pos = cosd(ang)*dist;
      y_pos = sind(ang)*dist;
      path = [x_pos, y_pos] + offset;
    end

  end
```

APPENDIX D. ORIGINAL KALMAN FILTER AND CHI-SQUARED SCRIPT

```
close all; clc; clear all;

 A = eye(2);   %linear system A matrix
 B = eye(2);   %linear system B matrix
 C = eye(2);   %linear system C matrix
 D = [0,0;0,0]; %linear system D matrix
 Q=0.1*eye(2);  %process noise covariance
 R=0.1*eye(2);  %measurement noise covariance

 %generate ‘truth’ data
 N = 200; %number of samples

 x_true(:,1) = [0;0];  %true initial condition
 u = [1;1];       %assume a constant control input
 for i=1:N
%    x_true(:,i+1) = A*x_true(:,i) + B*u + Q*randn(2,1);
%    z_true(:,i)  = C*x_true(:,i) + D*u + R*randn(2,1);

   x_true(:,i+1) = A*x_true(:,i) + B*u + chol(Q)*randn(2,1);
   % Q is random system noise

   z_true(:,i)  = C*x_true(:,i) + D*u + chol(R)*randn(2,1);
   % R is random measurement noise from envrionment
 end


 P = 5*eye(2); %initial Kalman filter covariance
 x_hat = x_true(:,1)+[0.5*randn;0.5*randn];
 for i=2:N

   spoofing_input = [0;0];
   if (i>25)
     spoofing_input = [-10.5;20.5];
     %this would be selected by the AI system
   end

   if (i>75)
     spoofing_input = [0;0];
     %this would be selected by the AI system
   end
```

```
    if (i>100)
      spoofing_input = [+20.5;-10.5];
      %this would be selected by the AI system
    end

    if (i>125)
      spoofing_input = [0.0;0.0];
      %this would be selected by the AI system
    end



    z_spoof = z_true + spoofing_input;
    %add the injected spoof to the true measurement

    %%%%%
    %step the kalman filter
    [x_hat(:,i), P, innovation(:,i), V_innovation] = KF...
      (z_spoof(:,i), x_hat(:,i-1), P, u, A, B, C, D, Q, R);


    p11(1,i) = P(1,1);
%     p12(1,i) = P(1,2);
%     p21(1,i) = P(2,1);
    p22(1,i) = P(2,2);

    v11(1,i) = V_innovation(1,1);
%     v12(1,i) = V_innovation(1,2);
%     v21(1,i) = V_innovation(2,1);
    v22(1,i) = V_innovation(2,2);

    %%%%%%%%%%
    %implement Chi^2 fault detector

    N_window = 10;
    N_chi = 2*N_window; %the Chi^2 DOF N_chi is equal to the dimension
               %of the innovation, in this case 2, times the
               %length of the data window (see “Tuning
               %Windowed Chi-Squared Detectors for Sensor
               %Attacks - and ref 24 therein


    PD = 0.997;
    %detection threshold
    l(1,i) = innovation(:,i)'*V_innovation^(-1)*innovation(:,i);
    %l is the Chi^2 value

    idx_lo = max(1,i-N_window+1);
    idx_hi = i;
    cl(1,i) = sum(l(idx_lo:idx_hi));  %windowed sum fof Chi^2 values


    %%%%%%%%
    %set the alarm state
%     if (cl(1,i)>chi2inv(PD,N_chi))
    %if (cl(1,i)>gammaincinv(PD/2,N_chi/2,'lower'))
    if (cl(1,i)>gammaincinv(PD,N_chi,'lower'))

      Alarm(1,i) = 1; %raise an alarm
    else
      Alarm(1,i) = 0;
    end

    idx_lo = max(1,i-(N_window+2)+1);
    sum_Alarm(1,i) = sum(Alarm(idx_lo:idx_hi))/(N_window+2);

    if (sum_Alarm(i)>=1)
      smoothed_Alarm(1,i) = 1;
    else
      smoothed_Alarm(1,i) = 0;
    end
  end
  l(1) = [];
  cl(1) = [];
  Alarm(1) = [];
  smoothed_Alarm(1) = [];
  v11(1) = [];
  v22(1) = [];
  p11(1) = [];
  p22(1) = [];
  innovation(:,1) = [];
  x_hat(:,1) = [];
  x_true(:,1) = [];
  x_true(:,end) = [];

  figure
  plot(2:N,l,'-b')
  hold on
```

```
plot(2:N,cl,’--r’)
legend(‘Chi^2 value’,’windowed Chi^2 value’)
xlabel(‘sample’)
ylabel(‘Chi^2’)

figure
plot(2:N,Alarm,’o’)
hold on
plot(2:N,smoothed_Alarm,’*’)
xlabel(‘sample’)
ylabel(‘Alarm’)

figure
subplot(211)
plot(2:N,innovation(1,:),’-b’)
hold on
plot(2:N,[3*sqrt(v11);-3*sqrt(v11)],’--r’)
plot(2:N,[sqrt(v11);-sqrt(v11)],’r’)
xlabel(‘sample’)
ylabel(‘innov 1’)
subplot(212)
plot(2:N,innovation(2,:),’-b’)
hold on
plot(2:N,[sqrt(v22);-sqrt(v22)],’r’)
plot(2:N,[3*sqrt(v22);-3*sqrt(v22)],’--r’)
xlabel(‘sample’)
ylabel(‘innov 2’)

figure
subplot(211)
plot(2:N,x_true(1,:) - x_hat(1,:),’-b’)
hold on
plot(2:N,[3*sqrt(p11);-3*sqrt(p11)],’--r’)
plot(2:N,[sqrt(p11);-sqrt(p11)],’r’)
xlabel(‘sample’)
ylabel(‘x error 1’)
subplot(212)
plot(2:N,x_true(2,:) - x_hat(2,:),’-b’)
hold on
plot(2:N,[sqrt(p22);-sqrt(p22)],’r’)
plot(2:N,[3*sqrt(p22);-3*sqrt(p22)],’--r’)
xlabel(‘sample’)
ylabel(‘x error 2’)

figure
```

```
 subplot(211)
 plot(2:N,x_true(1,:),'-b')
 hold on
 plot(2:N,x_hat(1,:),'--r')
 xlabel('sample')
 ylabel('x 1')
 legend('true','est')
 subplot(212)
 plot(2:N,x_true(2,:),'-b')
 hold on
 plot(2:N,x_hat(2,:),'--r')
 xlabel('sample')
 ylabel('x 2')

 figure
 histogram(cl,'Normalization','pdf')

 idx = find(Alarm==1);
 Avg_Alarm = sum(Alarm(idx))/(N-1)


function [x_hat_new, P_new, innovation, V_innovation] = KF...
 (z, x_hat, P, u, A, B, C, D, Q, R)

%Predict
x_hat=A*x_hat+B*u;
P=A*P*A'+Q;

%Update
Vk=(C*P*C'+R);
V_innovation = Vk;
Kk=P*C'*Vk^(-1);
innovation = z-C*x_hat-D*u;
x_hat_new=x_hat+Kk*innovation;
%P_new=P-Kk*C*P;
P_new=(eye(2)-Kk*C)*P*(eye(2)-Kk*C)'+Kk*R*Kk';

end
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] M. Fakhari Mehrjardi, H. Sanusi, Mohd. A. Mohd. Ali, and M. A. Taher, “PD Controller For Three-Axis Satellite Attitude Control Using Discrete Kalman Filter,” in *2014 International Conference on Computer, Communications, and Control Technology (I4CT)*, Sep. 2014, pp. 83–85. doi: 10.1109/I4CT.2014.6914151.

[2] W. M. Taha, A.-A. M. Taha, and J. Thunberg, “Cyber-Physical Systems: A Model-Based Approach.” Springer Nature Switzerland AG, Jan. 2021. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-36071-9

[3] Md. A. Rahman and H. Mohsenian-Rad, “False Data Injection Attacks Against Nonlinear State Estimation In Smart Power Grids,” in *2013 IEEE Power Energy Society General Meeting*, Jul. 2013, pp. 1–5. doi: 10.1109/PESMG.2013.6672638.

[4] D. Levshun, A. Chechulin, I. Kotenko, and Y. Chevalier, “Design and Verification Methodology for Secure and Distributed Cyber-Physical Systems,” in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, CANARY ISLANDS, Spain, Jun. 2019, pp. 1–5. doi: 10.1109/NTMS.2019.8763814.

[5] S. Bhaumik and P. Date, “The Kalman filter and the extended Kalman filter,” in *Nonlinear Estimation*, Chapman and Hall/CRC, 2019.

[6] Z. Zhang, L. Zhou, and P. Tokekar, “Strategies to Inject Spoofed Measurement Data to Mislead Kalman Filter,” *ArXiv171002442 Cs*, Sep. 2020, Accessed: Jan. 23, 2022. [Online]. Available: http://arxiv.org/abs/1710.02442

[7] F. Hou, Z. Pang, Y. Zhou, and D. Sun, “False Data Injection Attacks For A Class of Output Tracking Control Systems,” in *The 27th Chinese Control and Decision Conference (2015 CCDC)*, Qingdao, China, May 2015, pp. 3319–3323. doi: 10.1109/CCDC.2015.7162493.

[8] Y. Mo, E. Garone, A. Casavola, and B. Sinopoli, “False Data Injection Attacks Against State Estimation In Wireless Sensor Networks,” in *49th IEEE Conference on Decision and Control (CDC)*, Dec. 2010, pp. 5967–5972. doi: 10.1109/CDC.2010.5718158.

[9] K. Nakayama, N. Muralidhar, C. Jin, and R. Sharma, “Detection of False Data Injection Attacks in Cyber-Physical Systems using Dynamic Invariants,” in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, Boca Raton, FL, USA, Dec. 2019, pp. 1023–1030. doi: 10.1109/ICMLA.2019.00173.

[10] “How Kalman Filters Work, Part 1 | An Uncommon Lab.” https://www.anuncommonlab.com/articles/how-kalman-filters-work/#lkf (accessed Apr. 19, 2022).

[11] W. Franklin, “Kalman Filter Explained Simply,” *The Kalman Filter*, Dec. 31, 2020. https://thekalmanfilter.com/kalman-filter-explained-simply/ (accessed Apr. 19, 2022).

[12] @pramodAIML, “What Is Chi-Square Test & How Does It Work?,” *The Startup*, Aug. 22, 2020. https://medium.com/swlh/what-is-chi-square-test-how-does-it-work-3b7f22c03b01 (accessed Apr. 24, 2022).

[13] “Facts About the Chi-Square Distribution – Introductory Business Statistics.” https://opentextbc.ca/introbusinessstatopenstax/chapter/facts-about-the-chi-square-distribution/ (accessed Apr. 24, 2022).

[14] R. Da, “Failure Detection of Dynamical Systems with the State Chi-Square Test,” *J. Guid. Control Dyn.*, vol. 17, no. 2, pp. 271–277, 1994, doi: 10.2514/3.21193.

[15] P. Kormushev, S. Calinon, and D. G. Caldwell, “Reinforcement Learning in Robotics: Applications and Real-World Challenges [dagger],” *Robotics*, vol. 2, no. 3, pp. 122–148, 2013, doi: http://dx.doi.org/10.3390/robotics2030122.

[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, Second edition. Cambridge, Massachusetts: The MIT Press, 2018.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” *ArXiv170706347 Cs*, Aug. 2017, Accessed: Apr. 19, 2022. [Online]. Available: http://arxiv.org/abs/1707.06347

[18] M. Bernico, *Deep Learning Quick Reference: Useful Hacks for Training and Optimizing Deep Neural Networks with TensorFlow and Keras*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2018. Accessed: May 03, 2022. [Online]. Available: http://ebookcentral.proquest.com/lib/ebook-nps/detail.action?docID=5322203

[19] D. Graupe, *Principles Of Artificial Neural Networks (2nd Edition)*. Singapore, SINGAPORE: World Scientific Publishing Company, 2007. Accessed: Apr. 29, 2022. [Online]. Available: http://ebookcentral.proquest.com/lib/ebook-nps/detail.action?docID=312337

[20] “Neural Network Design.” https://hagan.okstate.edu/nnd.html (accessed Apr. 29, 2022).

[21] “Proximal Policy Optimization — Spinning Up documentation.” https://spinningup.openai.com/en/latest/algorithms/ppo.html (accessed Apr. 18, 2022).

[22] P. L. Gilabert, D. López-Bueno, T. Quynh Anh Pham, and G. Montoro, “Machine Learning for Digital Front-End: a Comprehensive Overview,” in *Machine Learning for Future Wireless Communications*, 1st ed., F. Luo, Ed. Wiley, 2020, pp. 327–381. doi: 10.1002/9781119562306.ch17.

[23] O. Inverso, A. Bemporad, and M. Tribastone, “SAT-Based Synthesis of Spoofing Attacks in Cyber-Physical Control Systems,” in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, Porto, Apr. 2018, pp. 1–9. doi: 10.1109/ICCPS.2018.00009.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California