ABSTRACT

Title of Thesis: SAMPLING BASED MOTION PLANNING FOR MINIMIZING POSITION UNCERTAINTY WITH STEWART PLATFORMS

Ryan Ernandis
Master of Science, 2020

Thesis Directed by: Dr. Michael Otte
Department of Aerospace Engineering

The work described in this dissertation provides a unique approach to error based motion planning. Originally designed specifically for use on a parallel robot, these methods can be extended to a more general case of any well-defined robotic platforms. Requirements for application of these methods are a known method of kinematics for defining the system as well as a means of calculating noise based on the system. Two methods of error tracking and two motion planning algorithms are tested here as approaches to this problem.

Shown within are the results of the motion planning methods used. One combination of motion planning algorithm and error tracking works best as a general solution to this problem and is designed to work on a parallel robot; specifically, a Stewart platform. The motivation for use of a Stewart platform comes from research done at NASA Langley Research Center in the field of In-Space Assembly.

SAMPLING BASED MOTION PLANNING FOR MINIMIZING POSITION UNCERTAINTY WITH STEWART PLATFORMS

by

Ryan Ernandis

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master's of Science
2020

Advisory Committee:
Dr. Michael Otte, Chair/Advisor
Dr. David Akin
Dr. Huan Xu

Acknowledgments

First of all, I would like to thank my advisor, Dr. Michael Otte, for all his help in guiding and pushing me to this goal. I would also like to greatly thank my mentor and manager at NASA Langley Research Center, Dr. William Doggett, for providing funding and experience in pursuing this work. I'd like to thank Dr. Erik Komendera for pursuing internships for my senior capstone team in the NASA Big Idea Challenge of 2017 and giving me so much valuable information in the pursuit of knowledge of a Stewart platform, this pushed me down the path that I am on today! I'd also like to thank my thesis commitee, Dr. Huan Xu and Dr. David Akin, for agreeing to review and participate in my committee.

I'd also like to greatly thank my family who has been supportive of all my endeavors and pushed me to follow my dreams. I'd like to thank all my friends who've been there for me throughout graduate school as well as all of my fellow interns at NASA LaRC for being so amazing and helpful as I've pursued this degree. Lastly, I would like to thank MATLAB documentation and Stack Overflow for being there in my darkest hours to provide the guiding light to solve all my bugs.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

EKF	Extended Kalman Filter
LaRC	Langley Research Center
lmc	Local Minimum Cost
NASA	National Aeronautics and Space Administration
RRT	Rapidly-exploring Random Tree
RRT*	Rapidly-exploring Random Tree Star
RRT#	Rapidly-exploring Random Tree Sharp
SP	Stewart Platform

Chapter 1: Introduction

1.1 Motivation

Precision motion planning is important for enabling robots to autonomously perform work that requires low position error. Applications of these robots include construction for above and below ground, underwater, and in space. Also, precision motion planning is beneficial to self-driving cars and general autonomous work of robotic systems. In this dissertation, precision motion planning focused on tracking accumulated position uncertainty is applied to a parallel robot known as a Stewart platform.

Parallel robots present interesting challenges for motion planning because of the complexity associated with their forward kinematics. For reference, a serial manipulator or robot is able to have well-defined forward kinematics equations so that a given set of joint inputs results in a single known position of the end effector. A parallel robot's forward kinematics require much more work and effort to derive and in some cases can be impossible. For instance, a Stewart platform's forward kinematics solutions can result in up to 40 positions for a single set of joint inputs. Instead, inverse kinematics are well known and easily derived for parallel robots, unlike serial robots. Therefore, developing a method for motion planning on a

parallel robot provided a unique challenge.

Additional motivation behind this work centers on the research in In-Space Assembly being performed at NASA Langley Research Center (LaRC). This work aims to further the use of Stewart platforms as viable robotic end effectors and precision manipulators for in-space use as well as improve the knowledge and application of motion planning algorithms for parallel robots.

The higher level functions and approach discussed here are applicable to motion planning of any system such as a serial manipulator or vehicle. The low level, problem specific functions implemented can easily be interchanged for a different system.

1.2 Background and Overview

The field of motion planning is based on solving a very simple problem: given a starting state, move to a goal state while avoiding obstacles within a given environment. It's originally formulated as *The Piano Mover's Problem*, in which the idea is to move a piano into a home without damaging it or the home. This original problem is well defined and, as noted in [2], is known to be P-Space Hard when defined in three-dimensional space.

The general motion planning problem came to be clearly defined in the 1970s [3]. Then, in the 1980s, "perfect, combinatorial solutions, which are ideal in some settings, but not practical in most" were developed [3]. In the 1990s, sampling-based methods were developed offering a more practical method for solving these problems

by trading the algorithmic properties of completeness for a slightly less satisfying notion of probabilistic completeness [3].

The combinatorial motion planning methods developed in the 1980s, offer complete solutions for motion planning for a very specific problem: given a robot modeled as a point that can only translate [3] in two-dimensional space and where no sampling error or approximations are used. This combinatorial method plans in the configuration space, $\mathcal{C}$, of the robot. $\mathcal{C}$ is the set of all possible states of a robot in a given environment, or workspace, $\mathcal{W}$. $\mathcal{W}$ is the projection of the configuration space onto the subspace of $\mathcal{C}$ that is defined by (only) position. Combinatorial motion planning offers a map of the entire $\mathcal{C}$ that can show every possible position of the robot since it is easy to compute the topological space in which the robot can move. The main issue with this method is that when this space is extended into higher dimensions, this method breaks down into extremely complicated functions in order to model $\mathcal{C}$. For further information on this, consult [3].

The topology of $\mathcal{C}$ can vary greatly depending on both the robot being used and $\mathcal{W}$ because it must represent both spaces. Since $\mathcal{C}$ represents all transformations a robot can have, it can also be used to map out the obstacle spaces or regions where the robot cannot move to. This can create a very interesting and unique space that can be very hard to visualize once above three dimensions.

Sampling-based motion planning methods, developed in the 1990s, are far more common in higher dimensional spaces as they do not require the entire $\mathcal{C}$ to be known at the start of the algorithm. Instead, the algorithm plans by randomly checking states in $\mathcal{C}$ in order to map out regions in which it can or cannot move to. Specifically,

planning in $\mathcal{C}$ involves planning around two types of obstacle regions. The first type of region is where the robot would intersect obstacles in $\mathcal{W}$. The second region, and the more important reason for planning in $\mathcal{C}$, contains unsafe states for the robot to be in. These unsafe states include velocity limits, self intersections, physical limits, or any other constraint that may be enforced on the robot.

When planning in $\mathcal{C}$ the main concern is whether or not a sampled state is within an obstacle region. In a simple sampling-based motion planning algorithm, a starting state and a goal region are pulled from a given $\mathcal{W}$. The algorithm randomly explores $\mathcal{C}$, incrementally moving a set distance throughout the space to random states with the hope of eventually randomly finding a state within the goal region. With each movement it adds a new possible state to what is called a tree or graph structure. This tree is rooted at the starting state and is called such because, when graphed in a 2D environment, the paths resemble a tree. Subsection 1.3.1 goes into more detail of the motion planning algorithms used within this work and why they were chosen.

When using a Stewart platform, $\mathcal{C}$ can be represented using one of three methods: First, a six-dimensional space where each axis represents an actuator's position or length. This space is unique in that each point in this space can represent multiple configurations of position and rotation for the moving plate of the Stewart platform. Solving which configuration is correct for the current time involves having a method of forward kinematics that can produce a single solution. Another solution is a method of forward kinematics that produces multiple solutions combined with the history of states that the Stewart platform has been in so that the correct one can

be estimated.

The second method is to represent $\mathcal{C}$ using a six-dimensional space where each axis represents the position or rotation of the moving plate of the Stewart platform. This space is unique in that each point in this space corresponds to a single set of joint inputs which can easily be found using inverse kinematics. For Stewart platforms, and in general parallel robots, inverse kinematics are easily calculated based on some physical properties of the robot. This can be implemented with little computational effort, thus allowing the algorithm to easily calculate each state it is moving to.

The last method of representing $\mathcal{C}$ is a combination of the two prior methods. A twelve-dimensional space where each axis is either represented by the actuators' positions or the moving plate's position and rotation. This space offers a considerable challenge in mapping but would allow for the easiest implementation of motion planning. The entirety of the Stewart platform would be known and therefore no additional calculations would be needed to calculate either the joint inputs or current state of the robot. The space containing all valid states the Stewart platform can actually reach forms a six-dimensional manifold that is embedded in this twelve-dimensional space.

For ease of use and computational effort, the second method was chosen for representing $\mathcal{C}$, even though this method offers some unique challenges for motion planning. First, the motion must be planned forward, this would be easy with implementing forward kinematics; however this is not easy with Stewart platforms. Second, since this work is meant for a real world system, velocity limitations on the

actuators must be adhered to in order to be implemented properly. This results in having to check the actuators for their velocity, which results in having to calculate the actuator velocity at every single step of the motion planning to make sure they are not exceeded, on top of checking the actuator lengths to make sure they are not exceeded.

Additional challenges of implementing the motion planning algorithm to track error for the Stewart platform is that the error is weighted differently depending on the function being used. Since the rotation area is so small in comparison to the position area, a weighting factor must be placed on the rotation area so that it could have a comparable effect on the error in the system. This weighting value can be changed in order to focus more on the position or orientation but is necessary so that the algorithm does not produce a highly precise position path and an extremely imprecise rotation path. Also, plotting the data of the algorithm in a meaningful way produces some complexities, mainly in trying to show a six degree of freedom configuration space.

For this work, the decision was made to plot the system in two separate three dimensional graphs, the first of which is the position of the moving plate of the Stewart platform in the workspace. The second is the orientation of the moving plate in the workspace. This tended to produce interesting graphs as generally the algorithms were able to only go to one position but could go to the same orientation at multiple positions.

Since this work is focused on tracking and minimizing the error in the state of the Stewart platform at the end of its motion path, an Extended Kalman Filter

(EKF) was implemented within subroutines of the motion planning algorithms. An EKF was chosen as it allowed for the application of the kinematics to drive the Stewart platform as well as tracking of the error of the Stewart platform within a single framework. Subsection 1.3.2 explains the background and creation of the EKF used within this work as well as other implementations of the EKF on a Stewart platform.

The Stewart platform was first developed in 1965 by D. Stewart as a robotic platform for simulating aircraft as shown in [4]. D. Stewart's design of the Stewart platform focused on providing six-degrees of motion to a platform fixed to the ground using three actuators. When the Stewart platform started to be adapted to other applications, new configurations were designed. Some of these different configurations are discussed in [5]. The design that represents this work's Stewart platform is the 6-UPS (universal prismatic-spherical) Stewart platform. This design is based on six prismatic actuators attached to spherical joints on both the moving platform and the base platform. Within this work, the moving platform is considered the "top plate" and the base platform is considered the "base plate."

Since its development, the Stewart platform has been adapted into an extremely useful robotic platform outside of use in simulations. Specifically in this work, due to its capability for high strength and precise movements, the Stewart platform was chosen as a manipulator for use in In-Space Assembly. Work in the field of In-Space Assembly is focused on developing robotic methods for assembling satellites, space stations, and other structures in space and on other planets. More details on the Stewart platform and its use in In-Space Assembly are discussed in

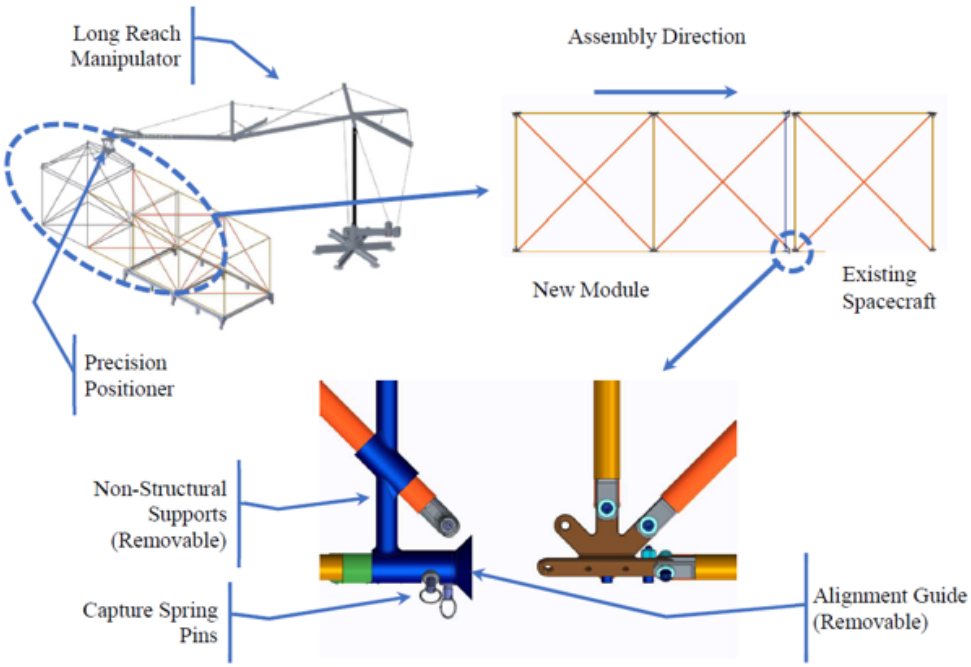

Figure 1.1: LSMS with a Stewart Platform as shown in [1]

subsection 1.3.2.

This work aims to aid in the Robotic Assembly of Modular Space Exploration Systems (RAMSES) project as described in [1]. The RAMSES project was devised to find new approaches to In-Space Assembly by creating smart structures assembled, verified, and characterized on the ground prior to launch. These structures would then be assembled in space with the use of a combination of long-reach manipulators and precision pointers. The Stewart platform is being used as a precision pointer which will grab the structures, and attach them to each other. The structures are built to include capture mechanisms and some alignment features, shown at the bottom of figure 1.1 as shown in [1], leaving the Stewart platform to only apply the force needed to trip the capture mechanisms as the structures attach to each other.

The use of Stewart platforms for In-Space Assembly has been previously re-

searched before within [6]. The Stewart platform described in [6] is used in a similar fashion as described in [1] as a precision pointer, picking and placing objects in desired locations while being connected to the end of a long reach manipulator. Additionally, tests were performed in [6] where a Stewart platform was controlled along a prescribed linear path in order to guide objects into place.

1.3 Related Work

The contributions of this dissertation to the advancement of sampling based motion planning applied for error reduction in parallel robots come from the lack of work within this specific area. Described in the following sections are the related literature discussing motion planning algorithms used in this dissertation, in section 1.3.1. Also described is the related literature discussing the application motion planning algorithms as well as the application of EKF for error tracking on Stewart platforms in section 1.3.2.

1.3.1 Motion Planning Algorithms

The first sampling based motion planning algorithms were only concerned with finding a valid path between start and goal, and thus were known as feasible path planning algorithms. An early feasible path planning algorithm that is still widely used today is known as Rapidly-exploring Random Trees, RRT [7]. The RRT algorithm used is described as being "specifically designed to handle nonholonomic constraints and high degrees of freedom" in [7] and shown in algorithm 1.

The RRT algorithm, follows the principals of sampling based motion planning. Given a starting node, the algorithm searches through $\mathcal{C}$ while avoiding obstacles in order to find a node within the goal region. The algorithm continues to run until either a node within the goal region is connected to the graph, or the maximum number of nodes in the graph is reached. The RRT algorithm is included within this work as a stepping stone for algorithmic development and to verify that a viable path to the goal can be found for the problems described within. This viability is needed initially to ensure that the two main algorithms used, RRT* and RRT#, can perform as intended. These algorithms are described in detail below.

Both RRT* and RRT# build off of RRT by not only finding a viable path to the goal, but continually optimizing this path until they meet a certain stopping criteria. This is due to both algorithms having properties of asymptotic optimality for motion planning problems where the optimal solution meets certain robustness criteria. In the case of this work, the stopping criteria is run time. Both of these algorithms were designed to optimize based on a cost function and they both are guaranteed to work for any metric set as the cost function. Also, they will work for any cost function in which the triangle inequality holds.

The cost function is most commonly used to optimize distance, either distance to the goal state or distance from the starting state. Within this work, the cost function is the cumulative error at each state, the calculation of which is discussed later in section 2.3. In both algorithms, the cost of each state is saved and when a new state is found, the state is checked for whether it is a more viable state for the path based on this cost. Both algorithms handle this viability differently.

The RRT* algorithm, described in [8] is used as an initial anytime solution algorithm. RRT* is a variant of both RRG (Rapidly-exploring Random Graph) and RRT that incrementally builds a tree, providing anytime solutions, and provably converges to an optimal solution with minimal computational and memory requirements. The RRT* algorithm modifies RRG to remove "redundant edges, i.e. edges that are not part of a shortest path from the root of the tree to a vertex" [8]. RRT* creates a rewiring of the RRT search tree that ensures the minimum-cost path is used to reach a vertex.

RRT# is the main algorithm being tested along with RRT*. RRT# guarantees asymptotic optimality and ensures that the constructed spanning tree rooted at the initial state contains lowest-cost path information for vertices which have the potential to be part of the optimal solution as described in [9]. Similar to RRT*, RRT# creates a rewiring of the RRT search tree; however, unlike RRT*, it propagates new cost short-cut information from the neighborhood in which it was discovered to any nodes in the graph that can benefit from that information. This can lead to slightly longer run times but also more optimal results.

1.3.2 Motion Planning with Stewart Platforms

Path planning in Stewart platforms has been researched before as is shown in [10], though the planning involved was to establish a path from a given start position to a given end position without encountering any singularities in the Stewart platform. Singularities are positions in the workspace where the Stewart platform

is uncontrollable and it loses some degree of constraint. Within this dissertation, the Stewart platform's environment is constrained which reduces the number of singularities encountered; however they are not completely removed.

The main focus of this dissertation was error reduction, not singularity avoidance. In [11] trajectory planning for a straight line path of a Stewart platform was researched. Additionally the Stewart platform was used to apply a force to make a connection for a given space-rated connector; however there is not a focus on the error seen by the Stewart platform and the trajectory planner does not allow it to move in a non-linear direction. The work being discussed in this dissertation aims to allow the Stewart platform to move as necessary, not just in a straight line, while also minimizing error.

Since motion planning algorithms are being used in combination with a Stewart platform to provide a path of motion, some form of kinematics needs to be implemented to produce the required motion for the Stewart platform. Generally in motion planning, forward kinematics are used to define the motion of the robot. In the case of Stewart platforms, and other parallel platforms in general, clear solutions of forward kinematics tend to be quite difficult to derive. A tractable solution to forward kinematics on parallel platforms is differential kinematics, which is performed in this work. In order to integrate the differential kinematics into the motion planning algorithm, as well as have a method for tracking error in the system, an Extended Kalman Filter (EKF) is used. A detailed explanation of differential kinematics is explained in section 2.3.3. While direct methods for forward kinematics exists they are computationally complex, as seen in [12], [13], and [14], and this

method was not pursued.

The application of the Kalman filter used within this work started with the explanations given in [15] and [16] in which the application and implementation of Kalman filters and EKF were explained in great detail. In general, a Kalman filter can be applied to any linear system in which some form of state estimation is used as well as some form of observation can be made, an EKF is applied when the system is non-linear. For the EKF used in this work, the state estimation was made based on the use of differential kinematics combined with noise. Additionally the method of observation used within this work was the application of differential kinematics without noise, giving a true state in the simulation. The notation used within the EKF comes from a variety of sources including, [15], [16], and [17].

The application of an EKF on a Stewart platform has been used before, as shown in [18]. In this application, the noises for the EKF were based on data transmission deviation between the host computer and the robot as well as measurement errors. The EKF was based on a direct kinematic method for real-time movement of the Stewart platform.

1.4 Nomenclature and Problem Formulation

This section will discuss the nomenclature and variables being used throughout this dissertation in 1.4.1. Then, the formal definition of the problem being attempted in this work is presented in section 1.4.3. Tables 1.1 and 1.2 offer a basic description of all the variables used here and are presented as a means of easy reference if needed.

1.4.1 Nomenclature

All of the variables used within this work are introduced in tables 1.1 and 1.2. These tables are presented as a reference for the variables in the work, all of which will be explained in further detail as they are introduced throughout this dissertation. For additional reference in later equations, a vector surrounded by brackets, such as $[\vec{c}\,]$, represents the skew-symmetric matrix of the vector as shown in equation 1.1.

$$[\vec{c}\,] = \begin{bmatrix} 0 & -c_3 & c_2 \\ c_3 & 0 & -c_1 \\ c_2 & c_1 & 0 \end{bmatrix} \tag{1.1}$$

1.4.2 Assumptions

The Stewart platform represented within this work comes from the design of a Stewart platform built at NASA Langley Research Center (LaRC). Additionally, when describing the workspace of the Stewart platform for motion planning purposes, the assumption was made that it would not be using any obstacles; however, functionality was built into the algorithm to account for obstacles in the future.

1.4.3 Problem Formulation

The problem of using sampling based motion planning for minimizing position uncertainty with Stewart platforms is defined as follows. Given a configuration

Variable	**Description**	**Variable**	**Description**
$\mathcal{T}$	Tree	x	A Node
$\mathcal{G}$	Graph Structure	x_{start}	Starting Node
$\mathcal{V}$	Set of Vertices	x_{rand}	Random Node
$\mathcal{E}$	Set of Edges	x_{near}	Near Node
$\mathcal{E}'$	Placeholder Edge Set	x_{min}	Minimum Node
E	Error of Robot	$x_{nearest}$	Nearest Node
$\mathcal{P}$	Path of the Robot	x_{new}	New Node
$\mathcal{C}$	Configuration Space	$\mathcal{X}_{goal}$	Goal Region
Δ	Maximum Distance to Extend	$\mathcal{X}_{near}$	Set of Nearest Nodes
q	Priority Queue	s	Successor Node
d	Number of Dimensions	c_{min}	Minimum Cost
γ_{RRT}	Ball Constant	u	Control Input
k	Indexing Variable	K	Maximum Indexing Variable

Table 1.1: Motion Planning Variables

Variable	Description	Variable	Description
$\vec{x}_{start}$	Starting State Vector	$\mathbf{P}$	Error Covariance Matrix
$\vec{x}_{goal}$	Goal State Vector	$\mathbf{Q}$	Process Noise Covariance Matrix
x_{true}	Set of Vertices	$\mathbf{R}$	Observation Noise Covariance Matrix
$\overline{x}_{true}$	Array of True States	$\mathbf{J}$	Kinematics Jacobian
$\overline{x}$	Array of States	$\mathbf{H}$	Observation Jacobian
c_{tol}	Tolerance for Goal Region	$\mathbf{S}$	Innovation Covariance Matrix
$\vec{z}$	Observed State	$\mathbf{K}$	Kalman Gain
$\vec{y}$	Innovation Residual	$\vec{v}$	Observation Noise
d	Distance to Goal	$\mathbf{D}$	Actuator Input in 2D Case
w	Weighting Variable	t	Incremental move Distance
$\{B\}$	Base Frame of the Robot	$\mathbf{R}_{z'y'x'}$	Rotation Matrix
$\{T\}$	Top Frame of the Robot	${}^{G}_{B}\mathbf{T}$	Transformation Matrix from $\{B\}$ to $\{G\}$
$\{G\}$	Base Frame of the Robot	${}^{G}\vec{p}_{diff}$	Difference in Position in $\{G\}$
${}^{B}\vec{q}_i$	Vector Pointing to Joint i in $\{B\}$	${}^{G}\vec{p}_{\Delta}$	Small Amount of Position Change in $\{G\}$
${}^{G}\hat{n}_i$	Vector Pointing along Actuator i in $\{G\}$	${}^{G}\dot{\vec{p}}$	Velocity of Position Change in $\{G\}$
${}^{B}\vec{q}_i$	Vector Pointing to joint i in $\{B\}$	${}^{G}\theta$	Amount of Rotation in $\{G\}$
L_i	Vector form of Actuator i of the Stewart Platform	${}^{G}\dot{\phi}$	Velocity of Rotation Change
$\dot{\vec{L}}$	Velocity of Actuators	${}^{G}\vec{\omega}$	Axis Angle Velocity in $\{G\}$
$\vec{V}$	Total Velocity Vector in $\{G\}$	${}^{G}\vec{\omega}_0$	Combined Axis-angle Representation of Rotation in $\{G\}$
Δt	Time Step	${}^{G}\vec{\omega}_e$	Axis of Rotation in $\{G\}$

Table 1.2: Problem Specific Variables

space, $\mathcal{C}$, a starting state, x_{start}, a goal region, $\mathcal{X}_{goal}$, find a valid path, $\mathcal{P}$, such that the state error along the path, E, is minimized. The path is valid if and only if it is fully contained within the free space of $\mathcal{C}$ and therefore it is not intersecting the obstacle space.

Chapter 2: Algorithms

This chapter will explain the algorithms that were developed and implemented in this work. Section 2.2 will explain the motion planning algorithms implemented in this work, these algorithms will be shown in an unaltered form. Following this, in section 2.3, the specific subroutines created for this work as well as the main contributions of this dissertation will be introduced. Additionally, any changes to the motion planning algorithm's implementation of the subroutines are discussed in 2.3.

2.1 Overview of Algorithmic Contributions

As described in section 1.3, sampling based motion planning algorithms work based on the optimization of a cost function. Often, this cost function is based on distance to the goal or distance from the start; however within this work the motion planning algorithms listed in section 2.2 are using a metric based on accuracy. This accuracy is defined by the cumulative error at a given state. A unique feature of using this metric is that the most optimal path can result in being a longer path than may be found using distance based metrics.

Another interesting problem that can occur by using this metric is that mul-

tiple paths can result in the same ending cost value. Currently this is not being addressed by this algorithm as the main purpose was to develop a working solution; however this can become more rare with a more contoured topological space.

Since this algorithm would be tracking error, a method for calculating that error was needed. That is where the Extended Kalman Filter (EKF) was introduced. The EKF is used as the steering function plug-in to the motion planning algorithms, it is shown on section 2.3.1. The specific variable of interest is the error covariance matrix, $\mathbf{P}$, which can be used to track the error at each step of the motion planning algorithm; however the motion planning algorithms are often designed to find the optimal solution with respect to a particular objective or cost value. This is how the error functions came to be used.

The first error function tested is the Trace function which is simply the sum of the diagonal terms of $\mathbf{P}$. The second error function tested is the Volumetric Ellipsoid function. This function calculates the volume of the ellipsoid of error at each node of the motion planning algorithm. Both of these functions are explained in more detail in section 2.3.4.

In order to apply the Stewart platform to the motion planning function, a kinematics function representing the Stewart platform must be implemented within the EKF. Normally in the EKF this would be the forward kinematics function, though as previously stated, the Stewart platform's forward kinematics are extremely computation expensive and not well known. Therefore, a Differential kinematics algorithm was written in order to simulate forward kinematics using only inverse kinematics and some estimation on the next step based on velocities of the system. A full

description of this function is shown below in section 2.3.3.

In order to learn more about implementing the EKF within the motion planning algorithms listed below, a simple 2D toy problem was created to test the functions as well as learn more about what simple changes could do to the functions. The full description of the problem is in section 3.2. The kinematics functions for this problem are shown in section 2.3.2.

Lastly, the implementation of this problem along with the Stewart platform problem shows that with a change in the kinematics function as well as adjustments to some variables, the algorithms listed below can work for a range of robots. A full list of the algorithms and a simple description of these algorithms is shown in table 2.1.

2.2 Existing Motion Planning Algorithms

The algorithms described in this section are the motion planning algorithms used within this work and can be applied in general to other problems, not solely those described in this dissertation. Algorithms 1,2 and, 3 are based on their original formulations and any variations from the originally published algorithms will be noted in section 2.3.

2.2.1 Existing RRT Algorithm

Algorithm 1 shows the RRT algorithm being used, a slight variation of the original RRT algorithm described in [7]. RRT is used as a means of verifying a

Algorithm	Description	Section
Existing Algorithms		
RRT	Initial sampling based motion planning algorithm used to ensure a viable path exists in the configuration space	2.2.1
RRT*	Sampling based motion planning algorithm used to optimize a viable path	2.2.2
RRT#	Sampling based motion planning algorithm used to optimize a viable path, improves upon RRT*	2.2.3
Extend Procedure	RRT# procedure used to create new nodes and update graph	2.2.3.1
Replan Procedure	RRT# Procedure used to propagate effects of newly added nodes in graph	2.2.3.2
Auxiliary Procedures	Auxiliary procedures used in RRT#	2.2.3.3
Modified/Created Algorithms		
Steering Function	General Steering Function used by all motion planning algorithms	2.3.1
2D Kinematics	Kinematics for 2D Toy Problem	2.3.2
SP Kinematics	Kinematics used for Stewart Platform Problem	2.3.3

Table 2.1: Algorithms

solution exists in the configuration space, $\mathcal{C}$, for the problem it is applied to. RRT will randomly branch out from a starting region in order to find a node within the goal region.

The algorithm initiates the tree structure with the starting position (line 1). Lines 2-8 show the main loop of the algorithm. First, the random state function is called returning a random node, x_{rand}, from $\mathcal{C}$. x_{rand} is also checked for being reachable by the robot based on user created check functions (line 2). Next, the nearest neighbor function is called returning the nearest node, x_{near}, from the tree

that is closest in distance to x_{rand} (line 3). Using x_{near}, the steering function creates a new state, x_{new}, some distance, Δ, toward x_{rand} and also returns the input, u, to get from x_{near} to x_{new} (line 5). The steering function takes the place of what would normally be the input and new state functions in RRT. x_{new} is then added to the set of vertices in the tree and an edge is created using the input from the steering function (lines 6,7).

These steps repeat until one of two ending conditions are met, either the maximum number of iterations is met (i.e. K) or a node within the goal region, $\mathcal{X}_{goal}$, is found. The goal region is defined as a N-ball region around a specific goal state with a radius equal to the tolerance set by the user for the algorithm. For this work, the number of iterations is set as the maximum number of nodes in the tree structure and the tolerance is changed depending on the problem being run.

An example run of the RRT algorithm is shown in figure 2.1. This figure shows the entire search tree on the left side and the corresponding path of lowest error on the right side. The search tree is small as the algorithm only ran until it added a node in the goal region to the graph structure.

Starting with the RRT algorithm is necessary to ensure that a given problem has a possible solution, otherwise the RRT* and RRT# functions would not work. Once a possible solution is found the RRT algorithm is no longer needed as, with probability one, RRT will not return the most optimal solution. This is stated in Theorem 33 of [8] and proven in Appendix B of the same paper.

Algorithm 1: RRT

$\mathcal{T}$.init(x_{start});
for $k = 1...K$ **do**
    $x_{rand} \leftarrow$ RandomState();
    $x_{near} \leftarrow$ NearestNeighbor($x_{rand}, \mathcal{T}$);
    $(x_{new}, u) \leftarrow$ Steer($x_{near}, x_{rand}, \Delta$);
    $\mathcal{T}$.add_vertex(x_{new});
    $\mathcal{T}$.add_edge(x_{near}, x_{new}, u);
return $\mathcal{T}$;

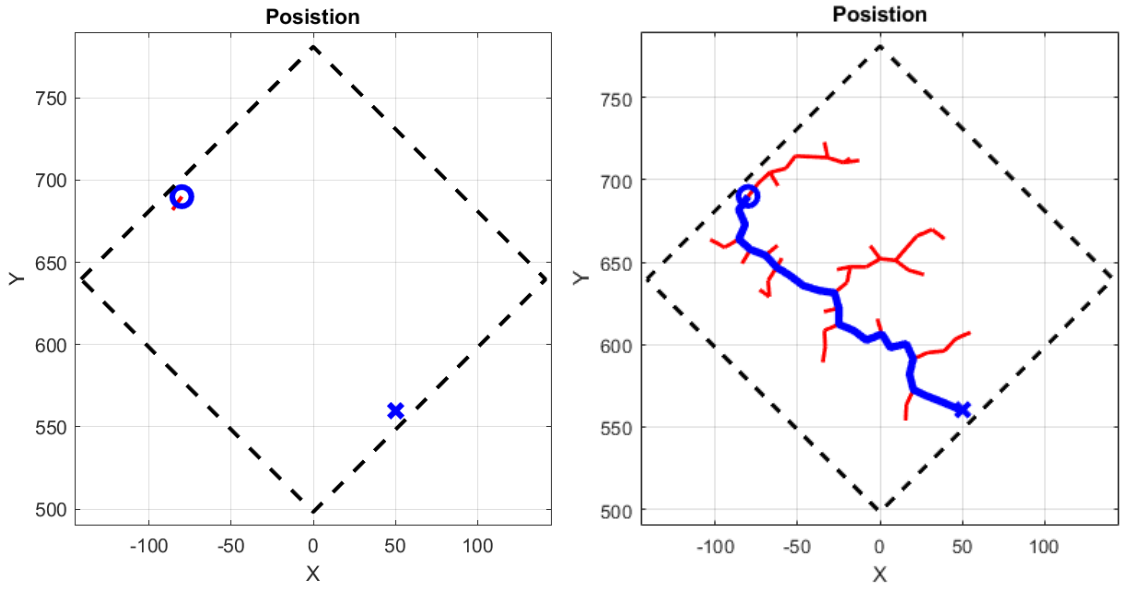

Figure 2.1: Example RRT Tree Progression

2.2.2 Existing RRT* Algorithm

The RRT* algorithm builds off of RRT by continuing to optimize the path found at the end of RRT. RRT* continues to optimize this path by continually searching for new nodes and adding these new nodes if they reduce the overall error of the path. RRT* will run this optimization until the run time is reached.

Algorithm 2 shows the RRT* algorithm that is being implemented. The algorithm is initiated with the vertex, $\mathcal{V}$, and edge, $\mathcal{E}$, sets that will store the components

of the graph structure, $\mathcal{G}$ (Line 1). In RRT* a graph structure is used rather than a tree structure as in RRT. $\mathcal{E}$ is initiated to zero, or empty and $\mathcal{V}$ is initiated with the starting position, x_{start}. In this algorithm lines 2-5 are similar to lines 2-5 in algorithm 1 with the only difference being the nearest neighbor function returning $x_{nearest}$ rather than x_{near}.

Starting at line 6 is the beginning of the improvements of RRT* on RRT. First, the algorithm checks if $x_{nearest}$ and x_{new} are collision free then it locates all the neighboring nodes within a specified distance of x_{new} (line 7). This distance is the minimum between Δ, the distance moved during the steering function to create x_{new}, and a ratio based on the number of nodes in the graph raised to the inverse of the number of dimensions and multiplied by a ball constant γ. This factor shrinks as the number of nodes increases. Then x_{new} is added into $\mathcal{V}$ (line 8). Additionally, to set the evaluation conditions used later, $x_{nearest}$ is saved as the minimum node, x_{min} and the cost of $x_{nearest}$ plus the cost of the trajectory from $x_{nearest}$ to x_{new} is saved as the minimum cost, c_{min} (line 9). Using the initial evaluation conditions, each x_{near} within the set of $\mathcal{X}_{near}$ is checked to see if the trajectory from it to x_{new} is collision free and the current cost of x_{near} plus the cost of the path from x_{near} to x_{new} is less than c_{min}. If this is true, x_{near} becomes the new x_{min} and the new c_{min} is set as this lower cost (lines 10-12).

Once all the nodes within $\mathcal{X}_{near}$ have been checked, the edge from x_{min} to x_{new} is added to $\mathcal{E}$ (line 13). Then, RRT* cycles through all of the nodes in $\mathcal{X}_{near}$ to check if x_{new} would fit as a better parent state than the current parent. This is done by calculating the new cost based on x_{new} and comparing it to the current cost at each

x_{near} (line 15). If x_{new} is a more fitting parent, then the edge dictating the path from the parent to a x_{near} is updated to reflect this (line 16).

An example run of the RRT* algorithm is shown in figure 2.2. This figure shows the progression of the RRT* algorithm from early path creation to a more filled-in configuration space.

Algorithm 2: RRT*

1 $\mathcal{V} \leftarrow \{x_{start}\}; \mathcal{E} \leftarrow 0;$
2 **for** i = *1,...,n* **do**
3 $\quad x_{rand} \leftarrow \mathsf{RandomState}();$
4 $\quad x_{nearest} \leftarrow \mathsf{NearestNeighbor}(\mathcal{G} = (\mathcal{V}, \mathcal{E}), x_{rand});$
5 $\quad x_{new} \leftarrow \mathsf{Steer}(x_{nearest}, x_{rand});$
6 $\quad$ **if** $\mathsf{CollisionFree}(x_{nearest}, x_{new})$ **then**
7 $\quad\quad \mathcal{X}_{near} \leftarrow \mathsf{Near}(\mathcal{G} = (\mathcal{V}, \mathcal{E}), x_{new}, \min\{\gamma_{RRT} * (\log(\mathrm{card}(\mathcal{V}))/\mathrm{card}(\mathcal{V}))^{\frac{1}{d}}, \Delta\});$
8 $\quad\quad \mathcal{V} \leftarrow \mathcal{V} \cup \{x_{new}\};$
9 $\quad\quad x_{min} \leftarrow x_{nearest}; c_{min} \leftarrow \mathsf{Cost}(x_{nearest}) + c(\mathsf{Line}(x_{nearest}, x_{new}));$
10 $\quad\quad$ **foreach** $x_{near} \in \mathcal{X}_{near}$ **do**
11 $\quad\quad\quad$ **if** $(\mathsf{CollisionFree}(x_{near}, x_{new})$ **and** $\mathsf{Cost}(x_{near}) + c(\mathsf{Line}(x_{near}, x_{new}))) < c_{min}$ **then**
12 $\quad\quad\quad\quad x_{min} \leftarrow x_{near}; c_{min} \leftarrow \mathsf{Cost}(x_{near}) + c(\mathsf{Line}(x_{near}, x_{new}));$
13 $\quad\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{(x_{min}, x_{new})\};$
14 $\quad\quad$ **foreach** $x_{near} \in \mathcal{X}_{near}$ **do**
15 $\quad\quad\quad$ **if** $(\mathsf{CollisionFree}(x_{new}, x_{nearest})$ **and** $\mathsf{Cost}(x_{new}) + c(\mathsf{Line}(x_{new}, x_{nearest}))) < \mathsf{Cost}(x_{near})$ **then**
16 $\quad\quad\quad\quad x_{parent} \leftarrow \mathsf{Parent}(x_{near});$
$\quad\quad\quad\quad\quad \mathcal{E} \leftarrow (\mathcal{E}\{(x_{parent}, x_{near})\}) \cup \{(x_{new}, x_{near})\}$
17 **return** $\mathcal{G} = (\mathcal{V}, \mathcal{E});$

2.2.3 Existing RRT# Algorithm

Algorithm 3 shows the body of the RRT# algorithm being implemented, this algorithm has not been changed from the original published within [9]. The entire

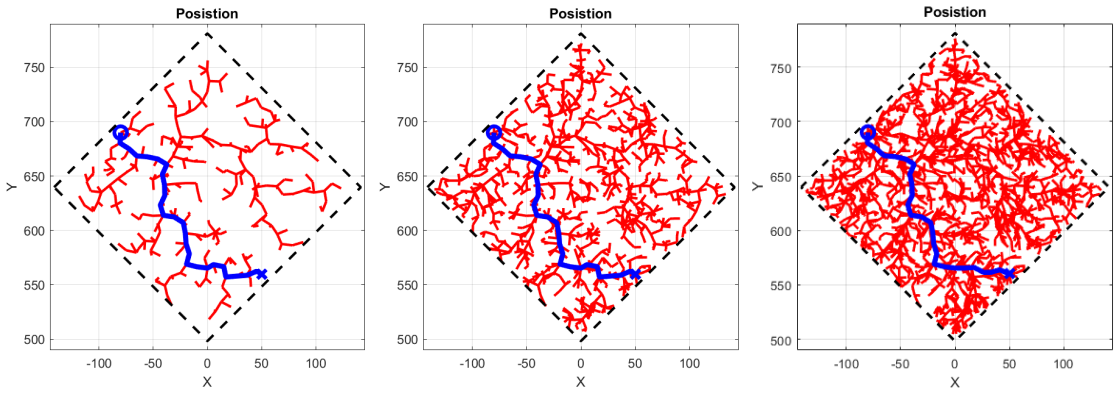

Figure 2.2: Example RRT* Tree Progression

RRT# algorithm and its additional algorithms/functions are shown between algorithm 3-6. A priority queue is used within the algorithm to keep track of vertices. RRT# performs the same operations as RRT* however with slight differences in the way error information is updated within the graph, as explained in section 1.3.1.

The body of the RRT# algorithm begins in the same fashion as RRT*, first the graph, $\mathcal{G}$, is created from the initiated sets of vertices, $\mathcal{V}$, and edges, $\mathcal{E}$ (lines 2-3). Beginning on line 4, for a set iteration amount defined by K the following three steps are done: First, a random node, x_{rand}, is found (line 5). Second, the extend function is called, adding a new node and rewiring the graph structure (line 6), which will be explained in detail in section 2.2.3.1. Third, the Replan function is called, propagating any effects of topological changes caused by adding a new node to the graph structure (line 7). This function is also explained in detail in section 2.2.3.2. Following these steps, $\mathcal{E}$ and $\mathcal{V}$ are updated and another set of edges, $\mathcal{E}'$, herein referred to as edge prime, is created (line 8). Next, for each node within $\mathcal{V}$, the edge prime set, $\mathcal{E}'$, collects the edge between each node and its parent (lines 9-10). Lastly, $\mathcal{G}$ is returned containing $\mathcal{V}$ and $\mathcal{E}'$ (line 11).

An example run of the RRT# algorithm is shown in figure 2.3. As in the RRT* figure, this figure shows the progression of the path created by RRT# as the run time of the algorithm passes.

Algorithm 3: Body of RRT#

```
RRT#(x_start, X_goal, X)
V ← {x_start}; E ← 0;
G(V, E);
for k = 1...K do
    x_rand ← RandomState();
    G ← Extend(G, x_rand);
    Replan(G, X_goal);
(V, E) ← G;  E' ← 0;
foreach x ∈ V do
    E' ← E' ∪ {(Parent(x), x)};
return  G = (V, E');
```

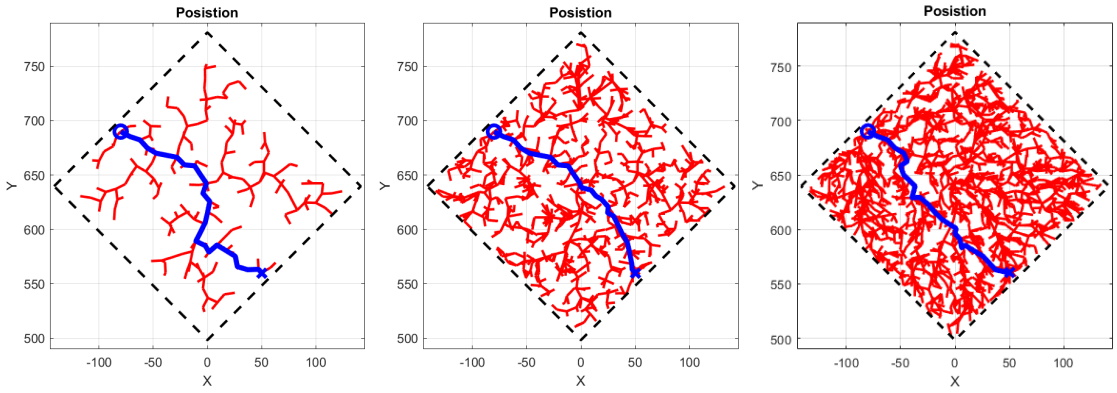

Figure 2.3: Example RRT# Tree Progression

2.2.3.1 RRT# Extend Procedure

The Extend procedure shown in algorithm 4 is used to first extend the graph towards a random node, creating a new node, then update the entire graph structure based on the new node. The extend procedure is called with inputs of $\mathcal{G}$ and x_{rand}.

The extend procedure initiates by creating a copy of the graph structure in a local set of vertices, $\mathcal{V}$, and edges, $\mathcal{E}$, and creating a secondary set of edges, $\mathcal{E}'$ (line 2). Then using the near function, the nearest node, $x_{nearest}$, is found (line 3). Using $x_{nearest}$ and x_{rand} the steering function finds a new node, x_{new} (line 4). Given this new node, as long as the path from x_{new} to $x_{nearest}$ is collision free the following begins: First, each of the nodes are sent through the Initialize function (line 6), explained in section 2.2.3.3. Following this, the near function returns all of the nodes in the graph within a certain distance of x_{new} to the set, $\mathcal{X}_{near}$ (line 7). Then, for each x_{near} in $\mathcal{X}_{near}$, as long as the path between x_{near} and x_{new} is free, the local minimum cost (lmc) of x_{new} is checked to see if x_{near} would make a better parent to x_{new} (lines 8-10). The lmc of a node is the best or lowest cost associated with a neighboring node of the node. Normally the lmc is associated with the parent node. If x_{near} is a better parent, the lmc of x_{new} is set to the cost of x_{near} plus the cost of the path from x_{near} to x_{new} and x_{near} is set as the parent of x_{new} (lines 11-12). After checking each x_{near}, the edge both to and from x_{new} is added to $\mathcal{E}'$ (line 13). Next, x_{new} is added to $\mathcal{V}$ and $\mathcal{E}$ is updated to include all of the edges added from $\mathcal{E}'$ (lines 14-15). Then the update Queue function is called, which is explained in section 2.2.3.3 (line 16). Finally, a new graph structure, $\mathcal{G}'$, is returned to the main body of RRT# (line 17).

Algorithm 4: Extend Function

```
1  Extend(G, x)
2  (V, E) ← G; E' ← 0;
3  x_nearest ← Near(G, x);
4  x_new ← Steer(x_nearest, x);
5  if CollisionFree(x_nearest, x_new) then
6      Initialize(x_new, x_nearest);
7      X_near ← Near(G, x_new, |V|);
8      foreach x_near ∈ X_near do
9          if CollisionFree(x_near, x_new) then
10             if lmc(x_new) > Cost(x_near) + c(x_near, x_new) then
11                 lmc(x_new) = Cost(x_near) + c(x_near, x_new);
12                 Parent(x_new) = x_near;
13             E' ← E' ∪ {(x_near, x_new), (x_new, x_near)};
14     V ← V ∪ {x_new};
15     E ← E ∪ E';
16     UpdateQueue(x_new);
17 return G' = (V, E);
```

2.2.3.2 RRT# Replan Procedure

Algorithm 5 shows the Replan procedure used within the main body of RRT#. The Replan procedure is used to propagate the effects of the topological changes to the graph as new vertices are added. Note that although this procedure is called Replan, this function is not doing re-planning in the sense of re-planning in response to changes in the environment. The Replan procedure is re-planning in the sense that there are nodes being added to the graph and the graph must be re-planned to propagate these changes. This naming convention holds over from the original publication of the RRT# algorithm in [9].

The procedure starts by going through the queue and returning the vertex of minimum key value, x, while it precedes the key value of the lowest cost vertex,

v^*_{goal}, in the goal set (line 2). Within this while loop, it takes this minimum key and sets the corresponding node's cost value to its lmc value then deletes the node from the queue (lines 3-5). Then, for each node that is a successor, s, to a node, x, in $\mathcal{G}$ the procedure updates the minimum cost of the s. The minimum cost is based on the cost of x plus the cost of the path from x to s. Also, the parent of s to x is updated as long as$s$ has a higher minimum cost than this new cost plus edge cost (lines 6-9). Finally, using this new cost and parent, s is sent to the Update Queue function (line 10).

Algorithm 5: Replan Function

```
1  Replan(G, X_goal)
2  while q.findmin() ≺ Key(v*_goal) do
3      x = q.findmin() ;
4      Cost(x) = lmc(x);
5      q.delete(x);
6      foreach s ∈ succ(G, x) do
7          if lmc(s) > Cost(x) + c(x, s) then
8              lmc(s) = Cost(x) + c(x, s);
9              Parent(s) = x;
10             UpdateQueue(s);
```

2.2.3.3 RRT# Auxiliary Procedures

The Initialize procedure takes the first input and sets its cost to infinity (line 2). Then, if the second input has not been created, it sets the first input's lmc to infinity and its parent node to zero (lines 4,5). Otherwise, it sets the first input's lmc to the cost of the first input plus the cost of the edge between the first and second input, as well as sets the parent of the first input as the second input (lines

7,8). These changes are saved within the node itself and each node is returned after being called.

The Update Queue function takes in a node and updates its place within the queue depending on a few factors. If the cost of the node is not equal to its lmc and the node is in the queue, then the node is updated with a new Key (lines 10,11). Otherwise, if the the cost of the node is not equal to its lmc and the node is not in the queue, it is inserted into the queue with a set key value (lines 12-14). Lastly, if the cost and lmc of the node are equal and it is in the queue, the node is deleted from the queue (lines 15-17).

For the Key function, this simply returns a vector of costs where the first element is a heuristic value of the optimal cost from the node to the goal and the second is the lmc of the node. The heuristic value used in this paper is a choice of the minimum of either the current node cost or the lmc of the node.

2.3 Problem Specific Subroutines

All functions and procedures listed within this section were created or used specifically for the two problems tested within this dissertation. The first problem is a 2D Toy problem. This problem is specified as two actuators attached to each other at a 90° angle, and then the entire system is rotated 45° from the horizontal. This problem will be expanded upon in section 3.2. The second problem is the main focus of this work, the Stewart platform problem which will be expanded upon in section 3.3. Both problems are designed to minimize error in their movements from

Algorithm 6: Auxiliary Functions

```
1  Initialize(x, x') Cost(x) ← ∞;
2  if x' = 0 then
3  |   lmc(x) ← ∞;
4  |   Parent(x) ← 0;
5  else
6  |   lmc(x) = Cost(x') + c(x', x);
7  |   Parent(x) = x';
8  UpdateQueue(x) if Cost(x) ≠ lmc(x) and x ∈ q then
9  |   q.update(x, Key(x));
10 else
11 |   if Cost(x) ≠ lmc(x) and x ∉ q then
12 |   |   q.insert(x, Key(x));
13 |   else
14 |   |   if Cost(x) = lmc(x) and x ∈ q then
15 |   |   |   q.delete(x);
16 Key(x)  return k = [h(x), lmc(x)];
```

a given starting position to a given goal position.

For both of these problems, two different error functions were created. These error functions are used as the cost functions of the motion planning algorithms. Using these error functions still keeps the motion planning algorithm within a Euclidean space as the error at each node is found in a way that still obeys the triangle inequality and it never decreases. The error is continually increasing as each path between nodes in the motion planning algorithm adds error to the system when using either error function. This method of error propagation and tracking is what is novel and of interest for this work.

2.3.1 Steering Function

The steering function used in each of the aforementioned motion planning algorithms previously discussed is shown in algorithm 7. The steering function follows the format of an Extended Kalman Filter (EKF) in order to use the error covariance matrix, $\mathbf{P}$, as a method of tracking error in the system as well as calculate the movements of the robot throughout the environment. It is initialized with a starting node, $\vec{x}_{start}$, and a random state, $\vec{x}_{goal}$, that it is steering towards. Additionally, a tolerance, c_{tol}, weighting value, w, initial $\mathbf{P}$, and process noise covariance matrix, $\mathbf{Q}$, are initialized outside the algorithm and fed into it. The tolerance is used to define the region around $\vec{x}_{goal}$ in which the algorithm assumes it is at $\vec{x}_{goal}$. w is available, if necessary, as a means of weighting certain components of the state during goal distance checking. The number of iterations is set prior to the algorithm being started as a way of changing how much the robot moves during each loop of the function.

The steering function begins by initiating the arrays of the true state, $\overline{x}_{true}$, and the predicted state, $\overline{x}$, with $\vec{x}_{start}$ (line 2). Additionally, the iterative distance, t, is set by dividing the total distance to extend, Δ, by the maximum number of iterations allowed, K (line 3). Next, the algorithm loops for a set number of iterations, K (line 4). Within the loop, the algorithm follows the basic set up of an EKF with a few additions. An EKF occurs in three basic steps, the prediction step, the observation step, and the update step. In this algorithm the prediction step occurs in lines 6-11, the observation step occurs in line 12, and the update steps

occur in lines 13-19.

Starting with the prediction step, the algorithm makes three predictions. First is the prediction of the true state, $\vec{x}_{true}$ (line 5). This step is needed as the model is not running in real time and therefore cannot take any inputs from outside sensors, if it were connected to a real time system or simulation, this step would be not be necessary. The next prediction is that of the state of the robot, $\vec{x}$ (line 6), also known as the a priori state estimate. This comes from the kinematics function, Kinematics(), which uses a problem specific kinematics function to find where the robot will be based on where it was at the last time step, $\vec{x}_{k-1}$, where it is headed, $\vec{x}_{goal}$, and how far it should move, t. Then, $\mathbf{P}$ is found based on the Jacobian of the robot's kinematics, $\mathbf{J}$, and $\mathbf{Q}$ (line 7).

The next step is the observation step. Within this step the current true state, $\vec{x}_{true}$, is taken as a measurement and with it random observation noise, $\vec{v}$, is added based on a Gaussian distribution of the observation noise covariance matrix, $\mathbf{R}$, in order to find the observed location of the robot, $\vec{z}$ (line 8). The update step then begins by finding the pre-fit residual, $\vec{y}$, given $\vec{z}$ and $\vec{x}_{true}$ (line 9). Also the innovation covariance matrix, $\mathbf{S}$, is found based on the measurement Jacobian matrix, $\mathbf{H}$, as well as $\mathbf{P}$, and $\mathbf{R}$ (line 10). Next, the Kalman gain, $\mathbf{K}$, is found given $\mathbf{P}, \mathbf{H},$ and $\vec{y}$ (line 11). $\mathbf{K}$ is then used to update the predicted state, $\vec{x}_k$, based on the amount of pre-fit residual found earlier as well as updating $\mathbf{P}$ (lines 12,13). The updated state and error covariance matrix are known as the a posteriori estimates.

At the end of the algorithm, the distance to the goal position, d, is calculated based on the current position and a weighting factor, w, if it is used. d is compared

to the set tolerance level for closeness, c_{tol}, to the goal and if it falls within that tolerance, the algorithm is exited so that it does not overshoot the desired goal position (lines 14-16).

Algorithm 7: Steering Function

1 **Steer**$(\vec{x}_{start}, \vec{x}_{goal}, c_{tol}, w, \Delta, \mathbf{P}, \mathbf{Q})$;
2 $\vec{x}_0 = \vec{x}_{start},\ \vec{x}_{true,0} = \vec{x}_{start}$; % Initialize State Vectors
3 $t = \Delta / K$; % Calculate the Distance to Move Each Step
4 **for** $k = 1...K$ **do**
5 $\quad \vec{x}_{true,k} = \mathsf{Kinematics}(\vec{x}_{true,k-1}, \vec{x}_{goal}, t)$; % Find True State for Observation
6 $\quad \vec{x}_k = \mathsf{Kinematics}(\vec{x}_{k-1}, \vec{x}_{goal}, t)$; % Find A Priori State Estimate
7 $\quad \mathbf{P} = \mathbf{J} * \mathbf{P} * \mathbf{J} + \mathbf{Q}$; % Find A Priori Error Covariance Estimate
8 $\quad \vec{z}_k = \vec{x}_{true,k} + \vec{v}$; % Make an Observation
9 $\quad \vec{y} = \vec{z}_k + \vec{x}_{true,k}$; % Find Innovation
10 $\quad \mathbf{S} = \mathbf{H} * \mathbf{P} * \mathbf{H}^T + \mathbf{R}$; % Find Innovation Covariance
11 $\quad \mathbf{K} = (\mathbf{P} * \mathbf{H}^T)/\mathbf{S}$; % Find Kalman Gain
12 $\quad \vec{x}_k = \vec{x}_k + \mathbf{K} * \vec{y}$; % Find A Posteriori State Estimate
13 $\quad \mathbf{P} = (\mathsf{I} - \mathbf{K} * \mathbf{H}) * \mathbf{P}$; % Find A Posteriori Error Covariance Estimate
14 $\quad d = \mathsf{GoalDist}(\vec{x}_k, \vec{x}_{goal}, w)$; % Calculate Current Distance to Goal State
15 $\quad$ **if** $d < c_{tol}$ **then**
16 $\quad\quad$ **exit**
17 $\quad$ % Check if Current State is Within Goal Region
18 **return** $\vec{x}, \mathbf{P}$

2.3.2 2D Toy Kinematics

The kinematics function used in the 2D Toy problem is shown in algorithm 8. This algorithm plugs into the steering function to allow it to be used by the motion planning algorithms. The algorithm starts by initializing the Jacobian, $\mathbf{J}$, previously found by a hand derivation (line 2). Then, the distance between the start and goal nodes is normalized using equation A.1 to give a unit distance movement. That unit distance is multiplied by the velocity change in position, t, and divided

by the Jacobian, creating the actuator input variable **D** (line 3). Lastly, the new position is found by multiplying **J** by **D** and then adding the initial starting node (line 4).

Algorithm 8: 2D Kinematics Function

1 **Kinematics**$(\vec{x}_{start}, \vec{x}_{goal}, \Delta)$;
2 $\mathbf{J} = \begin{bmatrix} \cos(45) & \sin(45) \\ \sin(45) & \cos(45) \end{bmatrix}$
3 $\mathbf{D} = \mathbf{J}^{-1} * \Delta * \mathsf{Normalize}(\vec{x}_{goal} - \vec{x}_{start})$;
4 **return** $\vec{x} = \mathbf{J} * \mathbf{D} + \vec{x}_{start}$;

2.3.3 Stewart Platform Kinematics

The Stewart platform problem requires a more complex kinematics function than the 2D Toy problem due to the Stewart platform being a parallel manipulator. Since the 2D Toy problem is simple and its forward kinematics are simple, the Jacobian can easily be found and used in the kinematics equation. Parallel manipulators tend to have complicated derivations of forward kinematics that are very computationally intensive. In order to bypass using such an intensive function, differential kinematics are employed here as a means of making forward movements of the Stewart platform. Differential kinematics uses inverse kinematics and a desired motion to calculate velocities for how the actuators will move and how the plates will move.

The inverse Kinematics of the Stewart platform is summarized with equation 2.1. Given the locations of the i^{th} ball joint in the top frame, $\{T\}$, and the locations of the i^{th} ball joint in the base frame, $\{B\}$, the length of actuator i, L_i, can be

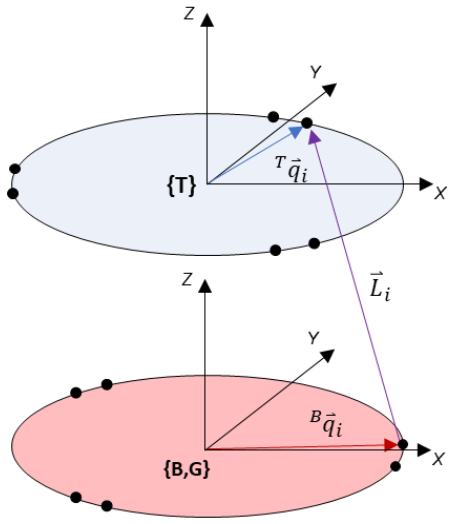

Figure 2.4: Frame Locations on Stewart Platform

found. Each ball joint location is transformed from its local frame to the global frame so that all measurements are made in a common frame as shown in figure 2.4. The location and sizes of the Stewart platform were taken from CAD files of the Stewart platform used at NASA LaRC. These locations were only approximations as manufacturing tolerances were large and defects in production of parts added to the error in exact locations of the joints in the physical system.

$$L_i = ||{}^G_T\mathbf{T}^T\vec{q_i} - {}^G_B\mathbf{T}^B\vec{q_i}|| \tag{2.1}$$

The Differential Kinematics algorithm, shown in algorithm 9, begins by creating a transformation matrix from the starting state. The transformation matrix consists of a rotation matrix in the first three rows and columns, the starting loca-

tion in the fourth column, and a row of zeros and a one in the fourth row, as shown in line 2 of the algorithm. This transformation matrix is in the ground frame, {G}, of the Stewart platform as shown in figure 2.4.

The rotation matrix used is shown in equation 2.2, where c represents the cos() function and s represents the sin() function. Next, the total difference in position is found via the difference between the starting position, $\vec{x}_{start}$, and the goal position, $\vec{x}_{goal}$ (line 3).

$$\mathbf{R}_{z'y'x'}(\alpha,\beta,\gamma) = \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix} \tag{2.2}$$

Next the total amount of rotation and the axis of that rotation from $\vec{x}_{start}$ to $\vec{x}_{goal}$ can be found by using the Rodrigues' Rotation Formula, shown in equation A.2 in appendix A. Rearranging this equation to solve for $[\vec{\omega}]\theta$ where $[\vec{\omega}]$ is the skew symmetric matrix for $\vec{\omega}$, results in the vector, $\vec{\omega}_0$ (line 4). $\vec{\omega}_0$ can then be normalized to find the axis of rotation $\vec{\omega}_e$ and the norm of $\vec{\omega}_0$ is the resulting amount of rotation, θ (lines 5,6). Then the actuator velocity, $\dot{\vec{L}}$, is set to infinity. This is done to set the while loop on line 7 to run at least one loop in order to calculate the actuator velocities of the movement.

The while loop will continue to run until the actuator velocities are set below a predetermined maximum velocity, $\vec{L}_{max}$ (line 8). This velocity is dependent on the actuators being used in the system and for this research it was set to 3.5 mm/s. The while loop begins by calculating the position change, $\vec{p}_\Delta$, based on a fraction of

the total difference in position (line 9). Next, the velocity of that position change is calculated by dividing the change by a generic time step, Δt (line 10). In this work, a time step of 1 second was used as it allowed for scaling of the movements if necessary. Both the angular velocity, $\dot{\phi}$, and the angular position, $\vec{\theta}_{\Delta}$, change are calculated in a similar manner as the position (lines 11,12). $\dot{\phi}$ must then be converted into the axis-angle velocity, $\vec{\omega}$, to get the rotation in the correct frame (line 13). Then, the rotation matrix for the angular position change, $\mathbf{R}_{\Delta}$, can be calculated using equation A.2 (line 14). Next, the total velocity vector, $\vec{V}$, for all six dimensions can be calculated with a slight change in that the angular terms are the first three elements and the position terms are the second three elements (line 15). After this, the inverse Jacobian, based on the starting position, can be calculated using the set of equations 2.3-2.6 (line 16).

Equation 2.3 calculates the inverse transpose of the Jacobian based on the starting position. ${}^{G}\vec{q}_i$ represents a transform pointing to the i^{th} base joint in the global frame, $\{G\}$, as seen in 2.5. ${}^{G}\hat{n}_i$ represents a vector starting at the i^{th} base joint of an actuator and pointing to the corresponding top joint and can be found using 2.6. Equation 2.4 shows that the Jacobian is returned as just the inverse rather than the inverse transpose.

Using InvJac and $\vec{V}$, $\dot{\vec{L}}$ can be calculated for the current movement (line 17). If $\dot{\vec{L}}$ is calculated to be above $\vec{L}_{max}$, then the loop is ran again with Δ halved (lines 18,19). Lastly, the new position, $\vec{x}$ is returned by first adding $\vec{p}_{\Delta}$ to the position terms of $\vec{x}_{start}$ (line 20). Then, the new orientation is calculated from a conversion from rotation matrix back to Euler angles based on a rotation created

by $\mathbf{R}_{\Delta}$ multiplied by $\mathbf{R}_{z'y'x'}(\vec{x}_{start})$ (line 21).

$$ {}^{s}\mathbf{J}^{-T} = \begin{bmatrix} [{}^{G}\vec{q_1}]{}^{G}\hat{n}_1 & \dots & [{}^{G}\vec{q_6}]{}^{G}\hat{n}_6 \\ {}^{G}\hat{n}_1 & \dots & {}^{G}\hat{n}_6 \end{bmatrix} \tag{2.3} $$

$$ {}^{G}\mathbf{J}^{-1} = ({}^{G}\mathbf{J}^{-T})^{T} \tag{2.4} $$

$$ {}^{G}\vec{q_i} = {}^{G}_{j_i,B}\mathbf{T} \tag{2.5} $$

$$ {}^{G}\hat{n}_i = \frac{{}^{G}_{j_i,T}\mathbf{T} - {}^{G}_{j_i,B}\mathbf{T}}{||{}^{G}_{j_i,T}\mathbf{T} - {}^{G}_{j_i,B}\mathbf{T}||} \tag{2.6} $$

2.3.4 Error Functions

The functions listed below are the error functions used in RRT to find the cost at each node. The first function is shown in equation 2.7. This is simply just using the trace of the error covariance matrix as a cost value.

$$ c = \mathsf{Trace}(\mathbf{P}) \tag{2.7} $$

The second function is shown in equation 2.8. This is the Volumetric Ellipsoid function which calculates the volume of an N-degree ellipsoid based on the eigenvalues of $\mathbf{P}$. a_i represents the semi-axis lengths of the ellipsoid and n is the number of dimensions of $\mathbf{P}$. This equation scales with the number of dimensions of a problem; however the source for this equation, reference [19], states that after the volume reaches a maximum at five dimensions after which it asymptotically approaches zero. Therefore it is still useful within the six dimensions of the Stewart platform,

but higher dimensions may prove to be unsuitable for this equation.

$$V = \prod_{i=1}^{n} a_i * \frac{2}{n} \frac{\pi^{\frac{n}{2}}}{\Gamma(\frac{n}{2})} \tag{2.8}$$

Algorithm 9: Stewart platform Differential Kinematics

Kinematics$(\vec{x}_{start}, \vec{x}_{goal}, \Delta)$

$$\mathbf{T}_{start} = \begin{bmatrix} \mathbf{R}_{z'y'x'}(\vec{x}_{start}) & \vec{x}_{start} \\ 0 \quad 0 & 0 \quad 1 \end{bmatrix}$$

${}^{G}\vec{p}_{diff} = \vec{x}_{goal} - \vec{x}_{start}$ % Total Position Difference

${}^{G}\vec{\omega}_0 = \mathsf{RodVec}(\mathbf{R}_{z'y'x'}(\vec{x}_{goal}) * \mathbf{R}_{z'y'x'}(\vec{x}_{start}))$ % Total Rotation about an Axis

${}^{G}\vec{\omega}_e = \mathsf{Normalize}({}^{G}\vec{\omega}_0)$ % Single Axis of Rotation

${}^{G}\theta = ||{}^{G}\vec{\omega}_0||$ % Amount of Rotation about Axis

$\dot{\vec{L}} = \infty$ % Initialize Actuator Velocity to be Large

while $||\dot{\vec{L}}|| > \vec{L}_{max}$ **do**

- ${}^{G}\vec{p}_{\Delta} = \Delta * \vec{p}_{diff}$ % Desired Position Change
- ${}^{G}\dot{\vec{p}} = \frac{\vec{p}_{\Delta}}{\Delta t}$ % Desired Linear Velocity
- ${}^{G}\vec{\theta}_{\Delta} = \Delta * \theta$ % Desired Angular Change
- ${}^{G}\dot{\phi} = \frac{\vec{\theta}_{\Delta}}{\Delta t}$ % Desired Angular Velocity
- ${}^{G}\vec{\omega} = \vec{\omega}_e * \dot{\phi}$ % Desired Angular Velocity in Vector Form
- ${}^{G}\mathbf{R}_{\Delta} = e^{[\vec{\omega}_e]\vec{\theta}_{\Delta}}$ % Desired Angular Change in Matrix Form
- ${}^{G}\vec{V} = \begin{bmatrix} {}^{G}\vec{\omega} \\ {}^{G}\dot{\vec{p}} - [\vec{\omega}]{}^{G}p \end{bmatrix}$ % Vector Form of Linear and Angular Velocity
- $\mathbf{J}^{-1} = \mathsf{InvJac}(\mathbf{T}_{start})$ % Inverse Jacobian for Current Movement
- $\dot{\vec{L}} = \mathbf{J}^{-1}\vec{V}$ % Actuator Velocity for Current Movement
- **if** $||\dot{\vec{L}}|| > \vec{L}_{max}$ **then**
  - $\Delta = \frac{\Delta}{2}$ % Check if Actuator Velocity over Maximum Possible

$$\vec{x} = \begin{bmatrix} \vec{p}_{\Delta} + \vec{x}_{start} \\ \mathsf{MatrixToEuler}(\mathbf{R}_{\Delta} * \mathbf{R}_{z'y'x'}(\vec{x}_{start})) \end{bmatrix}$$ % New State After Movement

return $\vec{x}, \mathbf{J} = (\mathbf{J}^{-1})^{-1}$

Chapter 3: Experiments

All experiments conducted were run using MATLAB scripts and functions. Each experiment was set up by setting the parameters shown in table 3.1.

Once all the starting parameters are set, the graph structure and a KD-Tree are initiated. The KD-Tree is used as a means of speeding up the search for nearest neighbors within the motion planning algorithms which allows the algorithm to search a smaller set of nodes rather than through all of the nodes in order to find neighboring nodes. After this, the program will run until the stopping criteria for the corresponding motion planning algorithm is met, for RRT this is when the maximum number of nodes in the graph is reached and for RRT* and RRT# this is when the run time is reached. The program follows the algorithm laid out in section 2.2 and 2.3.

3.1 Experimental Set-Up

The following subsections will go into greater detail on how the parameters are set for the motion planning algorithms. Specifically shown are the variables marked with a § in table 3.1.

Parameter	Parameter Range	Description
Algorithm	*RRT, RRT*, or RRT#*	The Algorithm to be Used for Planning
Problem	*N/A*	Either the 2D Toy Problem or Stewart platform Problem
Error Function	*Trace or Volumetric Ellipsoid*	The error function being used
c_{tol}	*Varies*[*]	Tolerance Setting
Goal Bias	0.5%	How Often to Sample the Goal Node
Max Number of Nodes	3000	Stopping criteria used by RRT
Run time	*Varies*[†]	Stopping criteria used by RRT* and RRT#
Δ	10	How far the algorithm will extend each iteration
γ_{RRT}	*Varies*[‡]	Ball constant for RRT#
x_{start}	*Varies*[§]	Starting Position
x_{goal}	*Varies*[§]	Goal Position
P	*Varies*[§]	Initial Error Covariance Matrix
Q	*Varies*[§]	Process Noise Matrix
R	*Varies*[§]	Observation Noise Matrix
Min and Max Dimensions	*Varies*[§]	Dimension Boundaries for each Degree of Freedom
w	*Varies*[§]	Weighting value, if necessary

[*] *c_{tol} = 0.1 for 2D, c_{tol} = 1.0 for SP*

[†] *2D run time was 10 min, SP run time was 30 min*

[‡] *γ_{RRT} = 200 for 2D, γ_{RRT} = 25 for SP*

[§] *Explained in Detail Below*

Table 3.1: Initial Parameters

3.1.1 Goal Tolerance

For both problems, the *algorithm*, *problem*, and *error function* are user inputs and are mainly used for comparison. The tolerance setting, c_{tol}, used for the 2D Toy problem is 0.1 mm and for the Stewart platform problem is 1 mm. Both of these values were found through trial and error. When testing smaller values for c_{tol}, the motion planning algorithm would tend to oscillate around the goal position as it had an extremely hard time getting within the tolerance values required. The *goal bias* parameter is a simple heuristic that has the effect of causing the the motion planning tree to grow toward the goal position. It is set so that there is a 0.5% chance that a randomly sampled point will be the goal position thus always pushing the algorithm to search and extend towards the goal position.

3.1.2 Stopping Criteria

When setting the stopping criteria of the motion planning algorithms there are two variables that can be adjusted: *maximum number of nodes* and *run time*. The *maximum number of nodes* is only used by the RRT algorithm as a worst case stopping criteria. The RRT algorithm will stop before reaching the maximum number of nodes if the goal is found, otherwise it will continue to run until the maximum number of nodes is reached. The maximum number of nodes was set to 3000 for both cases.

For the 2D Toy problem this equates to having the entire configuration space being broken down into 3000 N-dimensional spheres with an approximate radius

of 1.3 mm; however only approximately 46% of the configuration space is actually reachable by the robot, limiting the space in which a node can be pulled from. This translates to actually having the usable configuration space broken down into spheres of radius approximately 0.9 mm.

For the Stewart platform problem this equates to having the entire configuration space being broken down into 3000 N-dimensional spheres with an approximate radius of 2 mm, the exact percentage of the six dimensional usable configuration space for the Stewart platform was not calculated; however when comparing the euclidean workspace volumes, the usable space is approximately 66% of the total space.

3.1.3 Parameter Selection

For both tests, the *run time* can be set based on user preference but should take into account Δ. If Δ is small, e.g. 1 mm, then it can take the algorithm a long time to extend towards the goal and not leave a lot of time for optimization of the path. Conversely if Δ is large, e.g. 100 mm, then the algorithm could reach the goal quickly; however its search radius for neighboring nodes would be extremely large and could take a long time to search through all of them. Therefore, a Δ of 10 was used for both problems. This allowed for the algorithms to find an initial path to the goal in a relatively short amount of time while keeping the search neighborhood size manageable. Additionally, the ball constant value, γ_{RRT}, value plays a critical role in keeping the neighborhood size manageable.

Within both RRT* and RRT#, γ_{RRT} is used as a constant multiplier to a shrinking radius value, based on the number of nodes in the graph, used to set the neighborhood search radius. This value shrinks as the number of nodes in the graph increases. This is done so that as the configuration space becomes more densely populated with nodes, the amount of nodes in the neighborhood search radius does not increase drastically. Since a Δ value of 10 is used for both problems, the point at which the radius provided by γ_{RRT} is equal to Δ was calculated and is shown in figure 3.1 for the 2D problem and 3.2 for the Stewart platform problem.

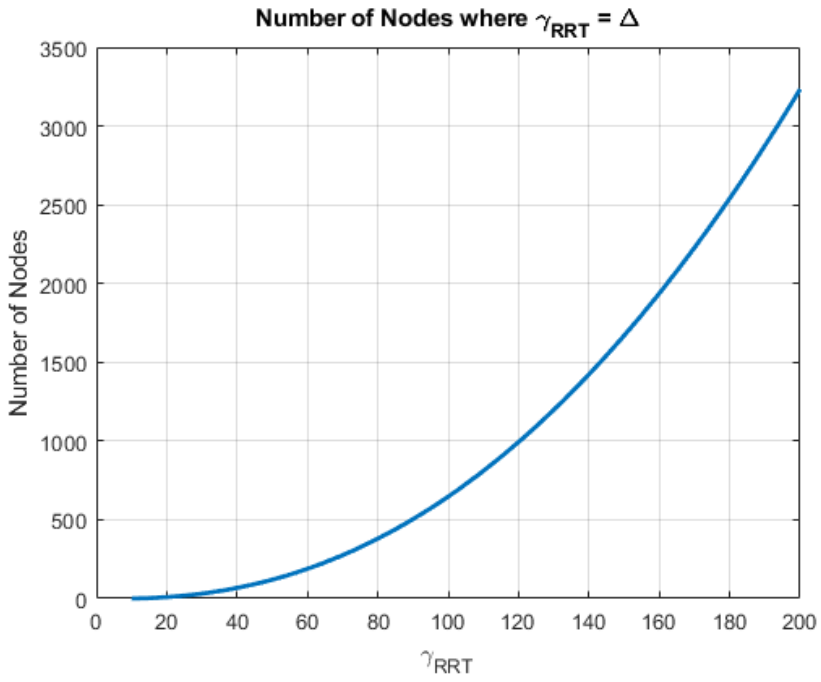

Figure 3.1: Graph Size at which γ_{RRT} Radius = Δ for 2D Problem

In the case of the 2D problem, γ_{RRT} was set to 200 as it is able to reach the point at which the neighbor will start shrinking in about 3 minutes and the tests were run for 10 minutes. This gives the algorithm just over two-thirds of the run time to optimize with a shrinking radius. In the case of the Stewart platform problem, γ_{RRT}

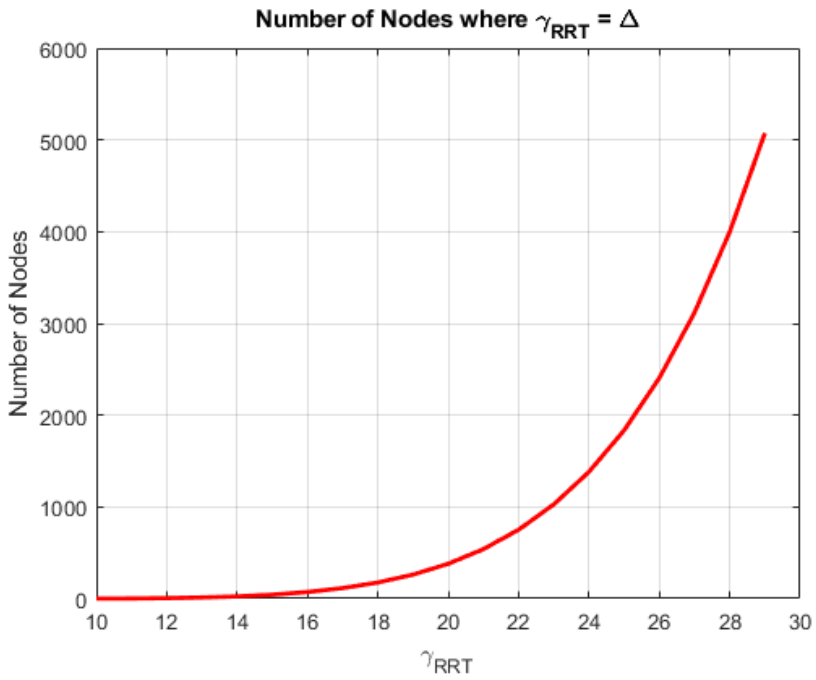

Figure 3.2: Graph Size at which γ_{RRT} Radius = Δ for SP Problem

was set to 25 as in testing the algorithm could reach the corresponding ∼1800 nodes in ∼15 minutes and the tests were run for 30 minutes, again giving the algorithm two-thirds of the run time to optimize with a shrinking radius. Additionally, these run times allow for the algorithm to run long enough to obtain the most optimal path. In testing, the optimization tends to have the most optimal path it will achieve within half the run-time given.

For both problems, the actuators used were modeled after 200 mm stroke length P-16P linear actuators made by Actuonix [20]. Using the information provided by the manufacturer, such as a repeatability of 0.8 mm and mechanical backlash of 0.3 mm, it was found that a Gaussian distribution with $\mu = 0$ and $\sigma = 0.2904$ mm would provide similar results as shown in figure 3.3. Additionally, the actuators used are driven by stepper motors and have 1024 steps. Therefore the step size can

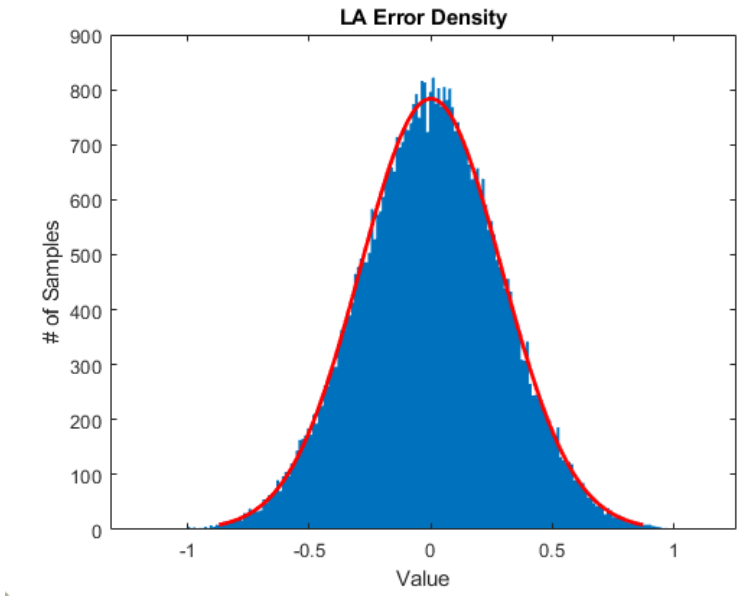

Figure 3.3: Linear Actuator Sampling Distribution

be found from dividing the stroke length by the number of steps and is calculated to be 0.1953 mm.

For each test of the two experiments, the start and goal positions were randomly found. These positions were kept constant when varying other values, such as $\mathbf{P}$, $\mathbf{Q}$, and $\mathbf{R}$, so that they can be compared using as similar set-ups as possible. The *min and max dimensions* of each problem were found prior to the problems being tested and stayed constant throughout all testing.

In the case of the 2D Toy problem, the boundary dimensions were set at the limits meaning the full movement of the actuator could be tested. This boundary limit includes large areas that would be incapable for the system to reach but it allowed for all the possible locations to be tested. For the Stewart platform case, the boundaries were set based on the limits of the entire physical system, these limits were found after doing some initial testing on the physical Stewart platform

system located at NASA LaRC. The system was driven to various positions to see how it would react and it resulted in occasionally breaking an actuator or popping a ball joint out of socket. Therefore, the limits were implemented so that these events would be less likely to occur. Additionally, within these limits the Stewart platform is limited by the maximum stroke length the actuators are set to achieve. This results in a similar situation as the 2D Toy problem where only a percentage of the configuration space is available for planning in and not the entirety of the dimensions given.

3.2 2D Toy Experiments

The 2D Toy problem was done as a way of gaining insight into the motion planning algorithms being used as well as a test bed for implementing new ideas.The setup for this problem is shown in figure 3.4. For the 2D Toy problem the Jacobian used as seen on line 2 of algorithm 8, was found based on a hand derivation from the kinematics equation. Also within algorithm 8, Δ value used is a single step size for the actuator which is different from Δ used within the EKF. Within the steering function of the 2D Toy problem, the number of iterations used was found by dividing the inputted variable Δ by the step size of the actuator and then rounding up if it was not a whole number. This was done as the kinematics only moved one step at a time. The observation Jacobian matrix, $\mathbf{H}$, used in the steering function was set to the identity matrix, $\mathbf{I}$, as it was assumed both actuators would be observable as well as their states. The observation noise matrix, $\mathbf{R}$, and the process noise covariance

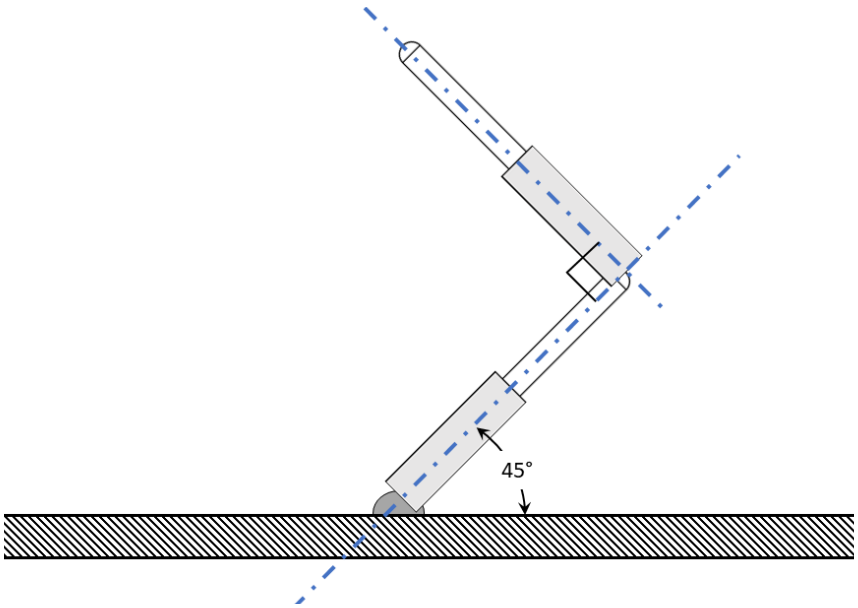

Figure 3.4: 2D Toy Problem Setup

matrix, **Q**, was found by the process shown in algorithm 10. **R** could have been found at each iteration of the steering function; however that was tested to be very time intensive and slowed the algorithm down, instead an average **R** was found and used. This process was only done once prior to the experiments being run. A two element vector, $\vec{l}$, randomly samples from the linear actuator Gaussian distribution. The variance of $\vec{l}$ is found at each loop and saved in $\vec{V}$. The covariance of $\vec{V}$ in each loop is saved in **A**. $\vec{V}$ is then multiplied by the Jacobian to find the variance in the state of each actuator represented by **B**. Then, the covariance of **B** is found and stored in **C**. Lastly, in order to get a single matrix to use for **Q**, the average of **C** was found and the single matrix for **R** is found by taking the average of **A**. The observation noise, v, comes from a random number generated from a Gaussian distribution of the observation noise matrix.

Lastly, many variations can be made during the experimentation of the 2D Toy

Algorithm 10: Method of Solving for **Q** and **R**

1 **Noise Matrix**(**J**)
2 **for** $k = 1...10000$ **do**
3 $\quad \vec{l} \sim \mathcal{N}(0, \sigma^2)$
4 $\quad \vec{V} = \mathsf{var}(\vec{l})$
5 $\quad \mathbf{A} = \mathsf{cov}(\vec{V})$
6 $\quad \mathbf{B} = \mathbf{J} * \vec{V}$
7 $\quad \mathbf{C} = \mathsf{cov}(\mathbf{B})$
8 $\mathbf{R} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{A}$
9 $\mathbf{Q} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{C}$
10 **return** $\mathbf{Q}, \mathbf{R}$

problem. A few that were done within this research include the following: variations on the amount of process noise **Q**, **R**, and starting **P** were tested. The dimensions of the 2D Toy problem are set based on the limitations of the actuators being used. These dimensions are shown in table 3.2.

Dimension	[x,y]
Minimum	[-145 mm, 490 mm]
Maximum	[145 mm, 790 mm]

Table 3.2: Dimensions of 2D Space

3.3 Stewart Platform Experiments

For the Stewart platform experiment, the Jacobian is found at each iteration in algorithm 9 and returned to the steering function. The observation noise, v, is generated in the same fashion as in the 2D Toy problem. The observation noise matrix, **R**, was found using the process laid out in algorithm 11.

This process starts by finding the inverse kinematic based actuator lengths for a given state (Line 2). Then, using algorithm 9 in combination with inverse

kinematics, a calculated actuator length and Jacobian is found at a distance $\Delta = 0.1$ away (Line 3). Then, in a loop of 5000 trials, using the normal distribution of the error as explained in section 3.2, a random sampling of actuator error is saved to a vector, $\vec{l_k}$ (Line 5). This actuator error is added to the difference in actuator lengths between the two previously calculated versions in order to find $\dot{\vec{L}}$ (Line 6). This is then multiplied by the Jacobian to give the velocity change, ΔV, for the movement (Line 7). This ΔV can then be used to find the change in position in an inverse of the method shown on line 15 of algorithm 9 (Line 8). Outside of the loop, the average ΔX is found (Line 9). Then in another loop of the same length as before, the difference in ΔX from its average is found (Line 11). This is saved as ΔX_C and is used to take the covariance of ΔX_C to find $\mathbf{R}$ (Line 12).

Algorithm 11: Method for Calculating R for SP

1 **Noise Matrix**$(\vec{x}, \vec{x}_{goal}, \Delta)$
2 $\vec{LA}_{act} = \mathsf{IK}(\vec{x})$
3 $(\vec{LA}_{calc}, \mathbf{J}) = \mathsf{Kinematics}(\vec{x}, \vec{x}_{goal}, \Delta)$
4 **for** $k = 1...N$ **do**
5 $\quad \vec{l_k} \sim \mathcal{N}(0, \sigma^2)$
6 $\quad \dot{\vec{L}} = \vec{LA}_{calc} - \vec{LA}_{act} + \vec{l_k}$
7 $\quad \Delta V = \mathbf{J} * \dot{\vec{L}}$
8 $\quad \Delta X = \begin{bmatrix} \Delta V_{4:6} + [\Delta V_{1:3}]\vec{x}_{1:3} \\ \Delta V_{1:3} \end{bmatrix}$
9 $\overline{\Delta X} = \frac{1}{N} \sum_{i=1}^{N} \Delta X$
10 **for** $k = 1...N$ **do**
11 $\quad \Delta X_C = \Delta X - \overline{\Delta X}$
12 $\mathbf{R} = \mathsf{cov}(\Delta X_C)$

For the $\mathbf{Q}$ error used, the matrix was set up as a diagonal matrix where the first three diagonal terms are assumed to be position error of 2 mm and the second

three diagonal terms are assumed to be angular error of 2°. Since the observation Jacobian matrix is the derivative of each measurement with respect to the state and these states cannot be found easily, $\mathbf{H}$ is found numerically using a finite differences method. My assumption is that $\mathbf{H}$ follows the format shown in equation 3.1 and therefore for each derivative term, equation 3.2 can be applied. Wherein equation 3.2 adds a small h value to each state in order to calculate the change in actuator lengths through inverse kinematics because of that addition. This method is applied in each iteration of the steering function and $\mathbf{H}$ is saved and averaged with each iteration as it is likely to change based on the changing state of the system.

$$\mathbf{H} = \begin{bmatrix} \frac{\delta l_1}{\delta_x} & \frac{\delta l_1}{\delta_y} & \frac{\delta l_1}{\delta_z} & \frac{\delta l_1}{\delta_\gamma} & \frac{\delta l_1}{\delta_\beta} & \frac{\delta l_1}{\delta_\alpha} \\ & & & \vdots & & \\ \frac{\delta l_6}{\delta_x} & \frac{\delta l_6}{\delta_y} & \frac{\delta l_6}{\delta_z} & \frac{\delta l_6}{\delta_\gamma} & \frac{\delta l_6}{\delta_\beta} & \frac{\delta l_6}{\delta_\alpha} \end{bmatrix} \tag{3.1}$$

$$\frac{\delta l_i}{\delta_x} = \frac{\mathsf{IK}([x-h, y, z, \gamma, \beta, \alpha]) - \mathsf{IK}([x+h, y, z, \gamma, \beta, \alpha])}{\delta h} \tag{3.2}$$

In addition to the aforementioned variables, the weight, w, variable is used in the Stewart platform problem. The weight variable is used as means of equalizing the effect the rotation and position error has on the system. As far as variations in the experiment go, like the 2D problem there are quite a range of variables to change. Some chosen here are variations in the amount of process noise $\mathbf{Q}$, $\mathbf{R}$, and starting $\mathbf{P}$ were tested. The dimensions of the space in which the top plate of the Stewart platform are shown in table 3.3.

Dimension	[X,Y,Z,γ,β,α]
Minimum	[-100 mm, -100 mm, 327 mm,-15°,-15°,-15°]
Maximum	[100 mm, 100 mm, 527 mm, 15°, 15°, 15°]

Table 3.3: Dimensions of Stewart Platform Space

Chapter 4: Discussion of Results and Conclusions

Within this chapter the results of the experiments for both problems will be shown. Starting with the 2D Toy problem results of running RRT# and RRT* in sections 4.1 and 4.2. Following this is the results of running RRT# and RRT* for the Stewart platform problem in sections 4.3 and 4.4. Additional Stewart platform tests with varying starting positions are shown in section 4.5.

4.1 Discussion of 2D Toy Problem Results

Within this section, the results of using both the Trace function and the Volumetric Ellipsoid function are discussed along with a comparison of RRT* and RRT# for the 2D Toy problem. The discussion on both error functions highlights the differences the starting error covariance matrices have on the resulting error, both in the total error at the end of a test and the total error increase from start to finish of the test. Following the summary of results in section 4.1.1 for the Trace function are the tests results for each variable, in sections 4.1.1.1-4.1.1.3. Similarly, following the summary of results for the Volumetric Ellipsoid function in section 4.1.2 are the test results of each variable in sections 4.1.2.1-4.1.2.3.

All of the results discussed in sections 4.1.1 to 4.1.2 are using RRT# as the

motion planning algorithm of choice. The discussion of RRT* versus RRT# is in section 4.2.

4.1.1 Noise Variation Using Trace Function

Varying the initial error covariance matrix, **P**, produces the greatest effect on the ending error of the 2D Toy problem as shown in table 4.1. This is to be expected as the initial **P** represents how well known the initial state is. If the initial state is highly unknown, it is to be expected that the ending error will be highly unknown as well. In terms of error addition from the initial error, the **P** cases had the smallest average increase by a small margin. Thus showing that the initial **P** has less effect on the quality of the motion planning algorithm than the **Q** and **R** values.

Case	**Initial Error**	**Error Addition**	**Ending Error**
P	0.2, 2, 20	0.9501	8.3501
Q	2	0.9132	2.9132
R	2	1.8886	3.8886
Total	*N/A*	1.2507	5.0507

Table 4.1: Initial Error, Average Error Addition and Ending Value

The process noise covariance matrix, **Q**, and the observation noise matrix, **R**, are intertwined in their effect on the algorithm. It is known, and specifically referenced in [15], that a higher **Q** value results in the EKF filter trusting the measurement received more so than the a priori estimation it makes and therefore corrects the estimation more. Additionally, a higher **R** value results in the EKF filter trusting the measurement less and instead relying on the a priori estimate more, therefore correcting less with it.

These known properties are reflected well in the **Q** and **R** test cases. Figure 4.1 shows that the highest ending error of the **Q** cases, a value of 3.3909, was still relatively low compared to that of the **R** cases. This is believed to be caused by the fact that the measurement coming into the system is one calculated directly from the forward kinematics equations of the 2D Toy problem with some noise added to them. Since this observation is so precise, the algorithm is able to correct more for this and produce a very small amount of error addition on average, as shown in table 4.1.

Figure 4.1: Box Plot of Ending Errors using Trace Function

An interesting observation from this test is that when **R** is higher than **Q**, there is a higher ending error and higher total error change across each trial since the algorithm is relying more heavily on the estimated value being correct. This shows that the algorithm itself is not a great method of estimation; however when

combined with a relatively small initial error, decent results can still be achieved. An example of the way the error growth occurs for the case in which **R** is highest is shown in figure 4.2. Within this figure, ◯ marks the starting position, × marks the goal position, and the dashed line is the path taken from start to goal.

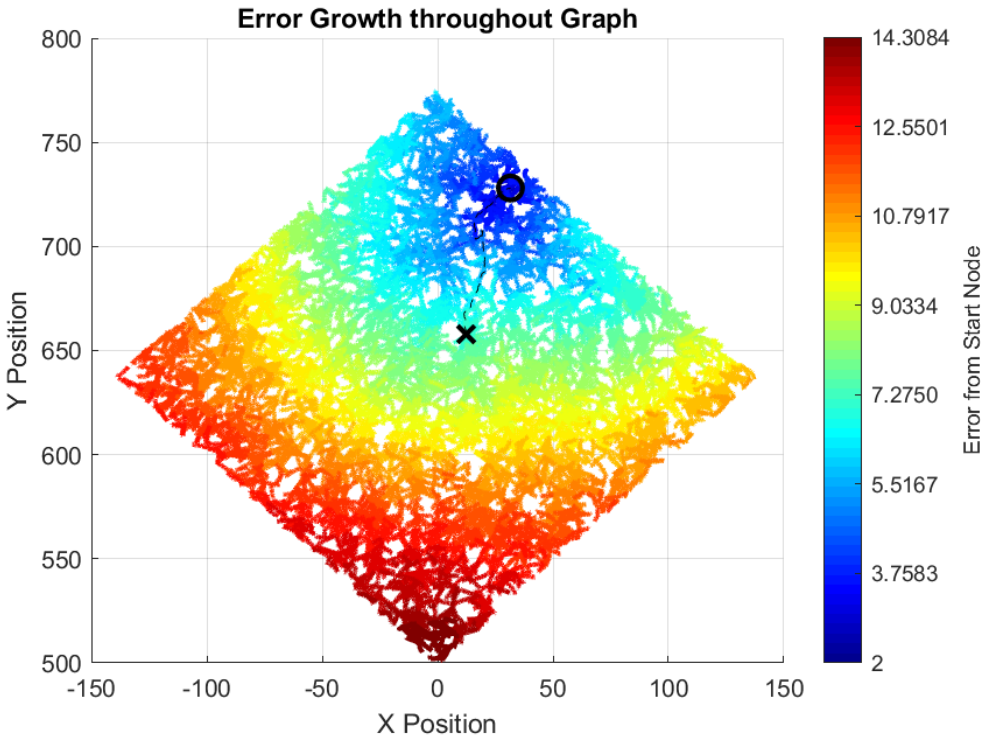

Figure 4.2: Error Growth for **R** = 10*R

In all of the graphs, the trees expanded at a fairly consistent rate. Within the first approximately 100 seconds the graphs expanded in a logarithmic fashion, after which they expanded linearly. On average, the variables ended with 7,085 nodes in the graph, with **R** tests averaging the most nodes in the graph with 7,101. Specifically, $\mathbf{R} = 10 * \mathbf{R}$ having the peak at 7,627 nodes.

4.1.1.1 Varying **P**

For the 0.1*$\mathbf{P}$ case, shown in figure 4.3, the initial error value was 0.2, the lowest value of any of the test cases. This resulted in the lowest ending error seen in any of the test cases for the 2D Toy problem. The 1*$\mathbf{P}$ case had an initial error value of 2 which matches all of the $\mathbf{R}$ and $\mathbf{Q}$ cases as they used 1*$\mathbf{P}$ also. Lastly, the 10*$\mathbf{P}$ case started with an error value of 20, the highest value and the driving factor in why the average ending of the $\mathbf{P}$ cases is so high. This is shown graphically in figure 4.1 as the box plot of the 10*$\mathbf{P}$ case is approximately triple that of the next highest plot, 10*$\mathbf{R}$.

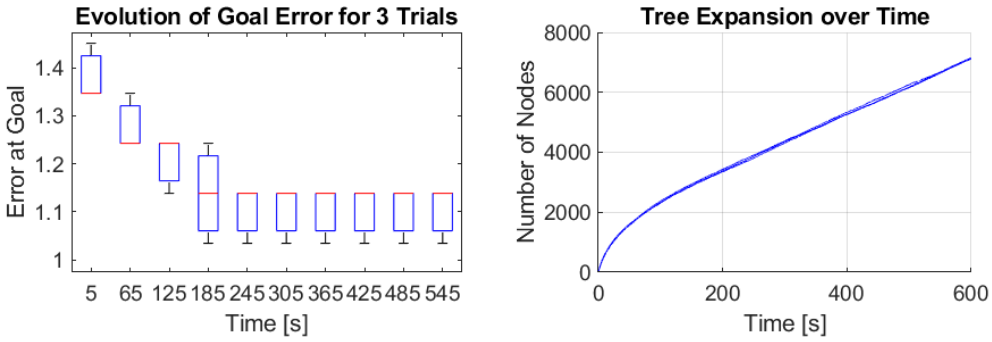

Figure 4.3: $\mathbf{P} = 0.1 * \mathbf{P}$ Test Data

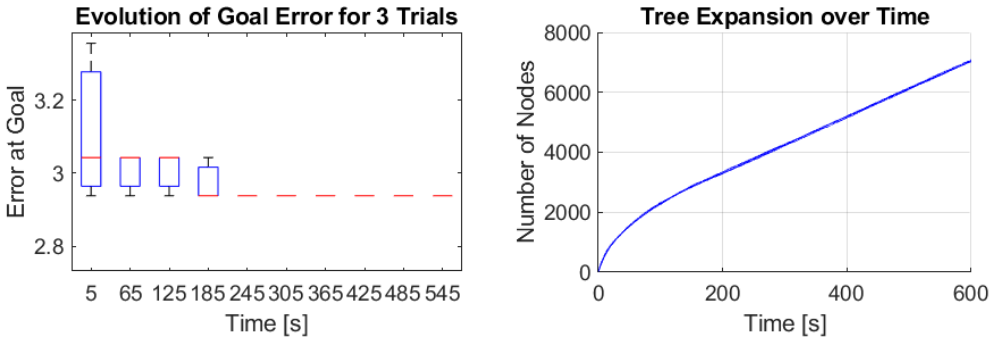

Figure 4.4: $\mathbf{P} = 1 * \mathbf{P}$ Test Data

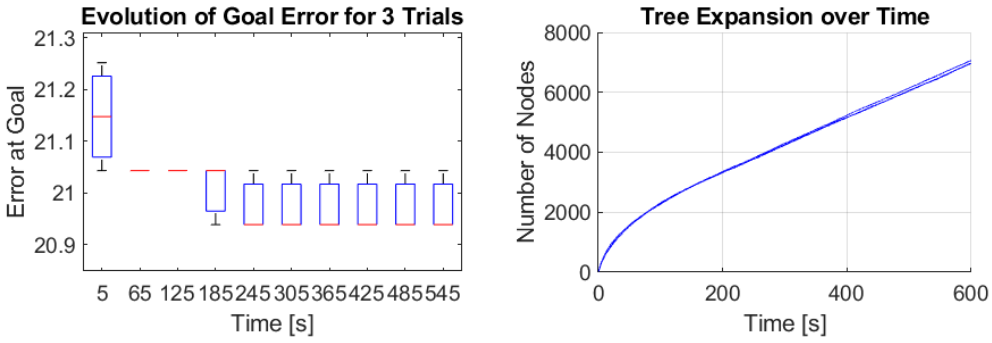

Figure 4.5: $\mathbf{P} = 10 * \mathbf{P}$ Test Data

4.1.1.2 Varying $\mathbf{Q}$

Varying $\mathbf{Q}$ with the Trace function had two interesting results. First, the average error only increased by approximately 0.5 between each step. A very linear rise compared to the $\mathbf{P}$ and $\mathbf{R}$ cases. Second, besides the $\mathbf{Q} = 0.1 * \mathbf{Q}$ case, the tests converged to the exact same error anywhere from halfway to two-thirds of the way through the test. The only other case in which this was seen during the use of the Trace function in the 2D Toy problem was during the $\mathbf{R} = 10 * \mathbf{R}$.

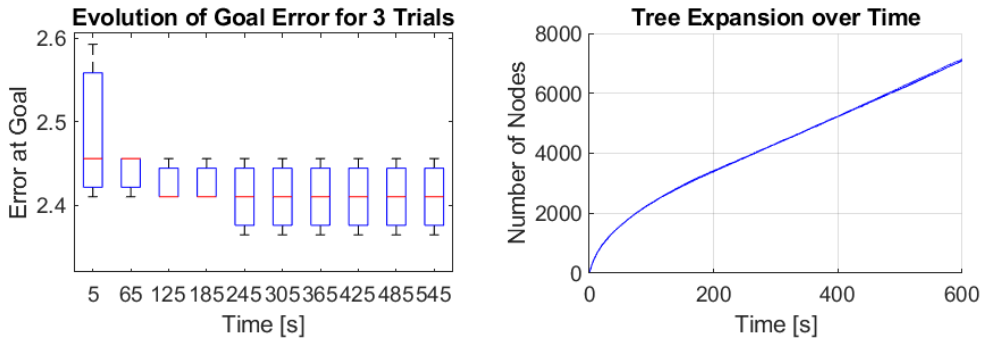

Figure 4.6: $\mathbf{Q} = 0.1 * \mathbf{Q}$ Test Data

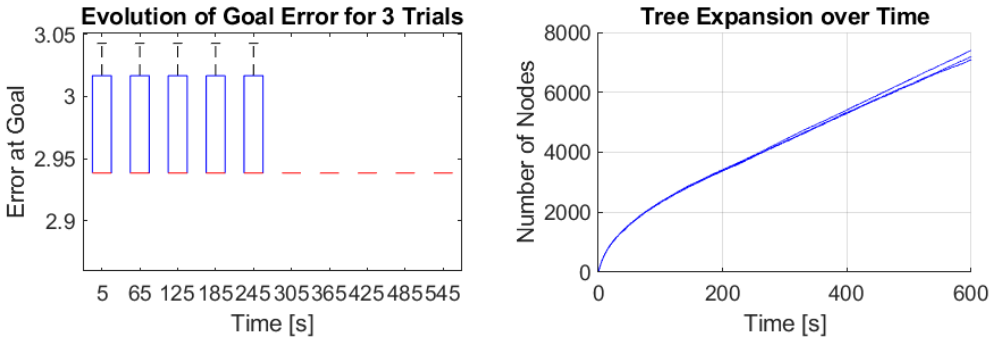

Figure 4.7: $\mathbf{Q} = 1 * \mathbf{Q}$ Test Data

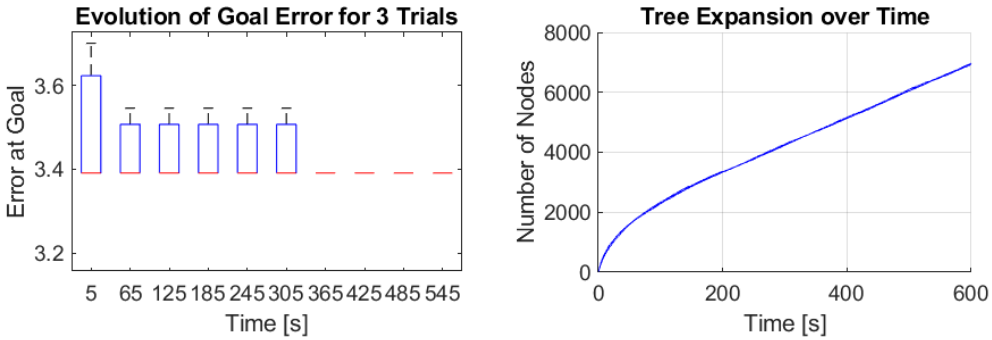

Figure 4.8: $\mathbf{Q} = 10 * \mathbf{Q}$ Test Data

4.1.1.3 Varying $\mathbf{R}$

In varying the $\mathbf{R}$ values, one effect stands out as was previously above. The average error increase of the $\mathbf{R}$ tests is almost double that of any of the other tests. This results in the ending error being so large in comparison to the results of $\mathbf{Q}$. The ending error increased by a factor of 5 between the $\mathbf{R} = 1 * \mathbf{R}$. This shows that the a priori estimate of the EKF is less reliable than the observation, even when adding more error to the observation.

Interestingly, even though $\mathbf{R}$ encounters the most error addition, it is able to include the most nodes in the graph and as the error gets worse, the number of

nodes in the graph increases. This could be caused by the error getting worse in the graph and therefore more nodes are added to the graph but are not affecting the path itself, which would take up more computation time in the algorithm.

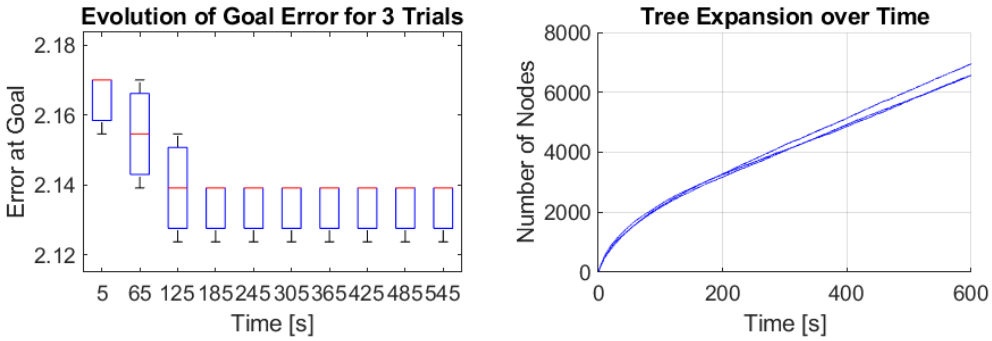

Figure 4.9: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

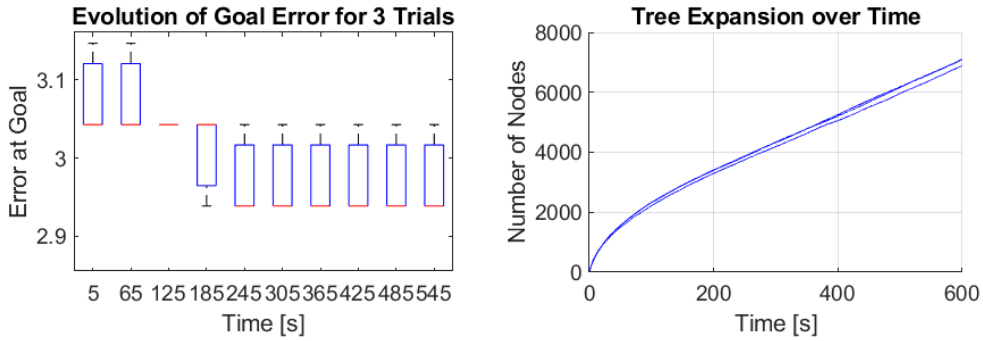

Figure 4.10: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

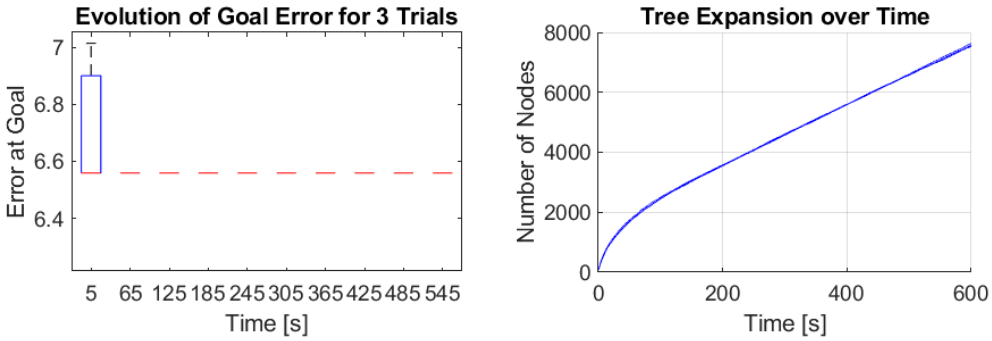

Figure 4.11: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

4.1.2 Noise Variation Using Volumetric Ellipsoid Function

The use of the Volumetric Ellipsoid function was not meant for the 2D Toy problem, rather it was meant for the Stewart platform problem. It was applied to the 2D Toy problem in order to work out any bugs or errors in its code as well as check to see if it performs similarly to the trace function.

When using the Volumetric Ellipsoid function the resulting error values are larger than that of the Trace function but follow the same patterns as that of the Trace function. The **P** values have the highest average ending error and the **R** values have the highest average increase in error; again double that of the **P** and **Q** values. This is shown in both numeric form in table 4.2 and graphic form in figure 4.12.

Case	**Initial Error**	**Error Addition**	**Ending Error**
P	0.3142, 3.1416, 31.4159	1.5652	13.1891
Q	3.1416	1.4345	4.5761
R	3.1416	2.9612	6.1028
Total	*N/A*	1.987	7.9560

Table 4.2: Average Error Addition and Ending Value

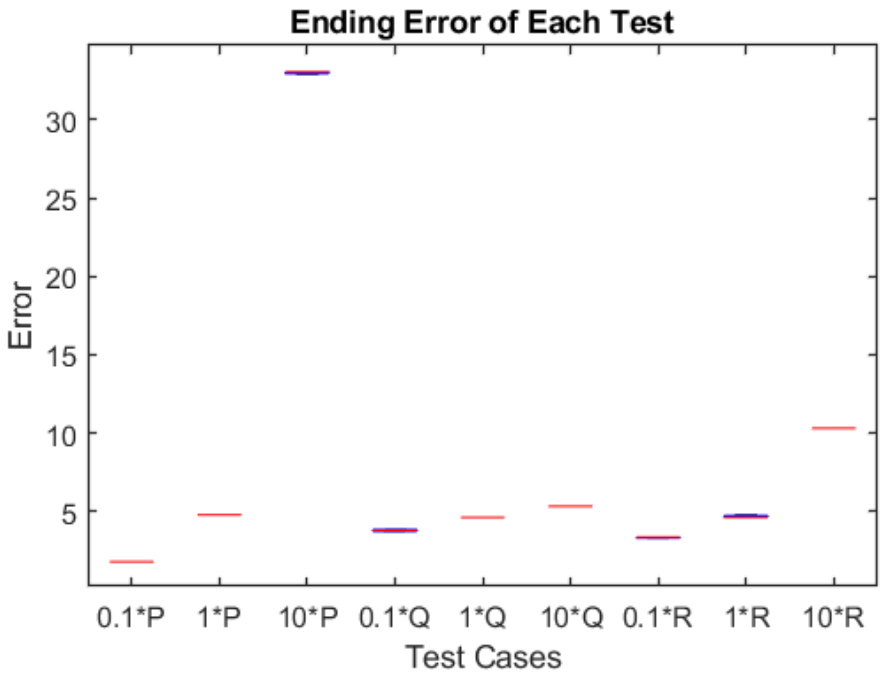

Figure 4.12: Box Plot of Ending Errors using Volumetric Ellipsoid Function

Even though the same results can be seen in the Volumetric Ellipsoid function as in the Trace function, the Volumetric Ellipsoid function was used as a way of measuring the volume of the error ellipsoid at each node in the graph. Additionally, this would be able to give an idea of the way in which the error was growing, or which axis it would grow along. Unfortunately, the error ball was quite small and unable to show up on plots in a meaningful way. However, as in the Trace function cases, an example plot of the way in which the error grows from the starting position is seen in figure 4.13. This shows the error growth pattern is similar to that of the Trace function. Within this figure, ◯ marks the starting position, × marks the goal position, and the dashed line is the path taken from start to goal.

A noticeable change when using the Volumetric Ellipsoid function is that the tests tend to converge on the same error point quickly, as shown in figures 4.14, 4.15,

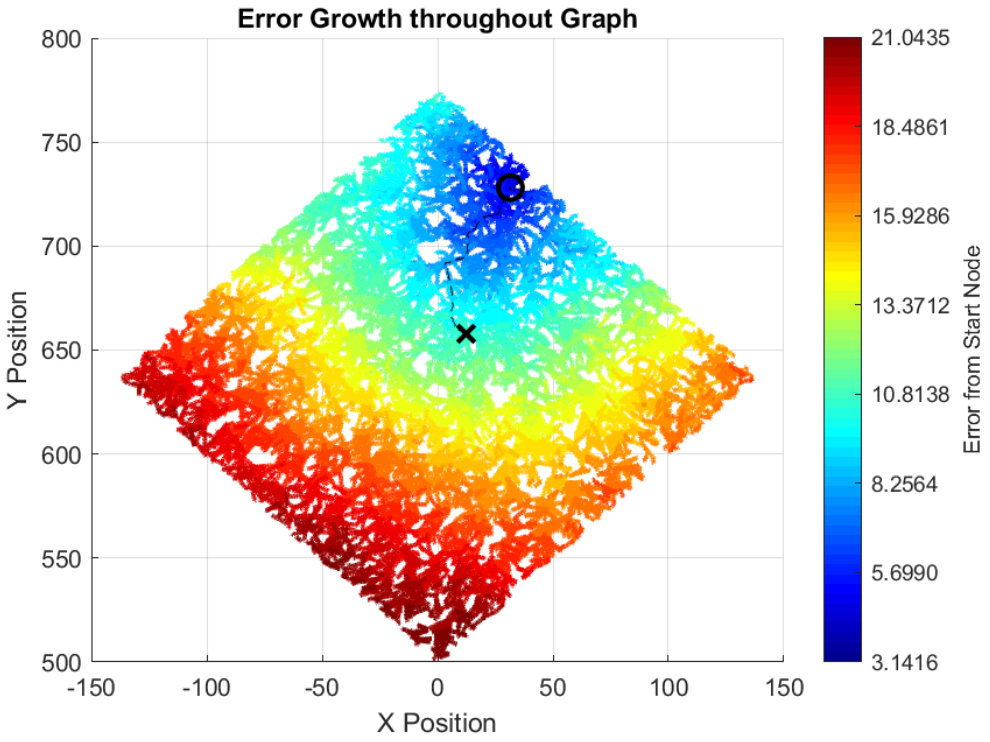

Figure 4.13: Error Growth for $\mathbf{R} = 10^{*}\mathbf{R}$

and 4.19. Another noticeable change when using the Volumetric Ellipsoid function is the average number of nodes in the graph is higher than the Trace function with 7,313, an increase of 228 nodes in the same amount of run time. The **Q** tests achieved the highest average number of nodes in the graph at 7,384 even though the highest single test value was 7,607 for the $\mathbf{R} = 10 * \mathbf{R}$ test.

4.1.2.1 Varying **P**

Varying **P** produced the same effects on the error in the graph as in the Trace function tests, though this error was noticeably higher. Specifically, the error resulted in multiples of π as the starting $\mathbf{P} = \mathbf{I}_{2x2}$, thus when entered into the Volumetric Ellipsoid equation, the volume produced is π.

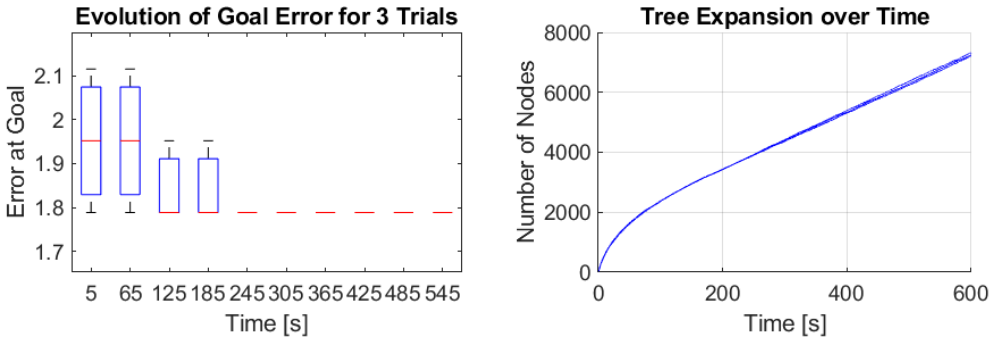

Figure 4.14: $\mathbf{P} = 0.1 * \mathbf{P}$ Test Data

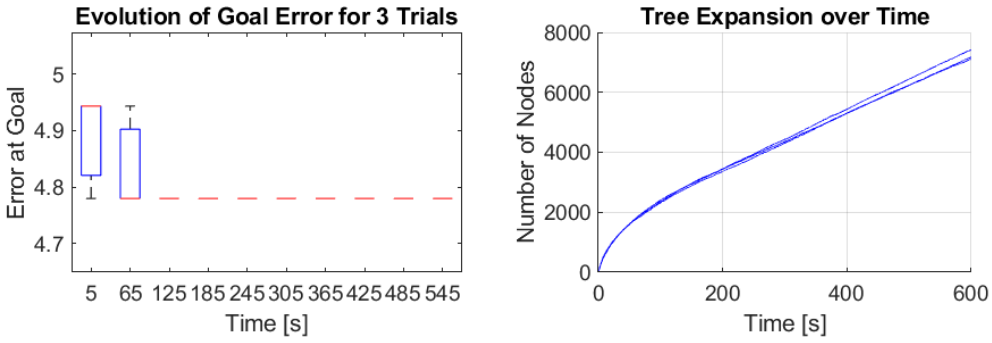

Figure 4.15: $\mathbf{P} = 1 * \mathbf{P}$ Test Data

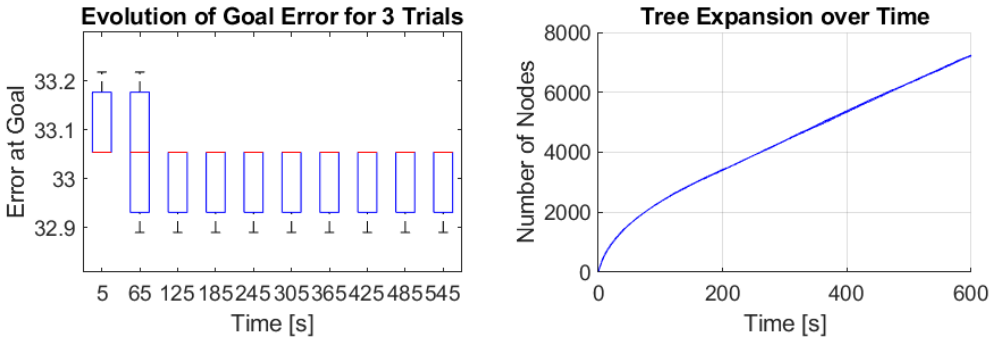

Figure 4.16: $\mathbf{P} = 10 * \mathbf{P}$ Test Data

4.1.2.2 Varying **Q**

The **Q** tests achieved the lowest ending error and error increase while also achieving the highest average number of nodes in the graph. These two attributes

are intertwined as figures 4.18 and 4.19 show. All three trials reached the same ending error and never improved, resulting in the test being able to add more nodes to the graph without spending computation time adding the nodes to the path. Additionally, figure 4.17 shows the graph was able to get a tight grouping of ending errors and still maintain a high number of nodes in the graph.

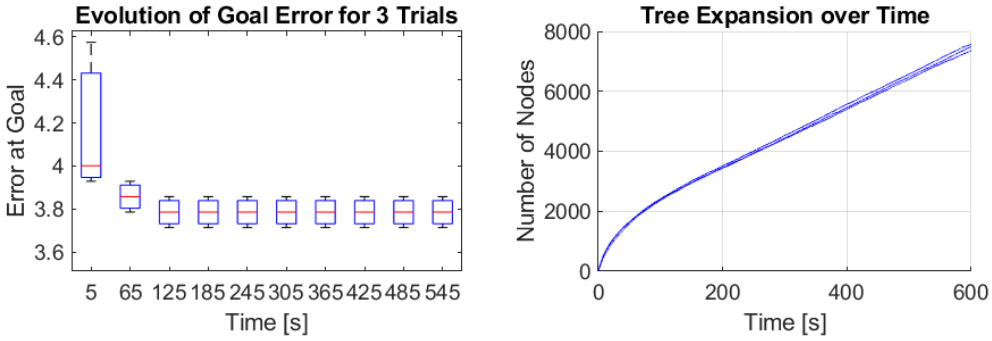

Figure 4.17: $\mathbf{Q} = 0.1 * \mathbf{Q}$ Test Data

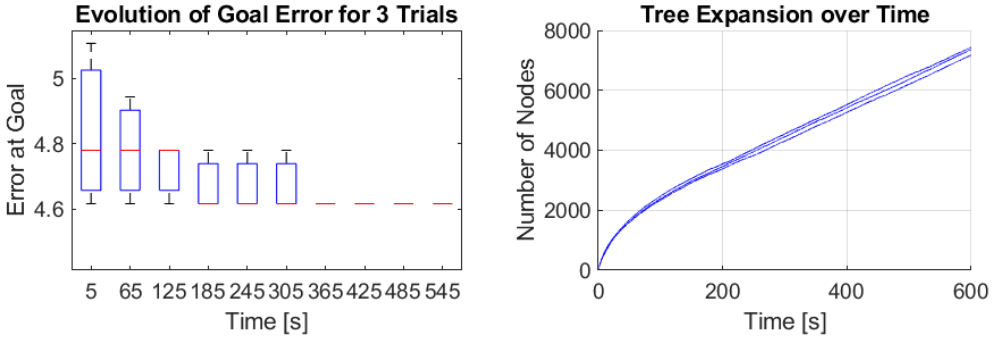

Figure 4.18: $\mathbf{Q} = 1 * \mathbf{Q}$ Test Data

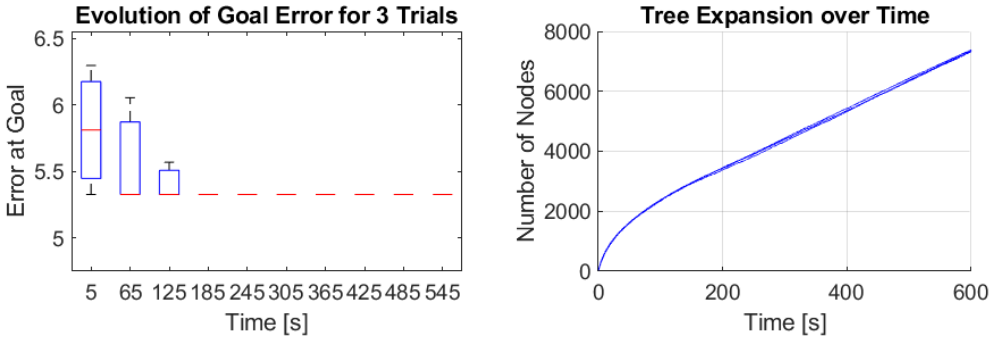

Figure 4.19: $\mathbf{Q} = 10 * \mathbf{Q}$ Test Data

4.1.2.3 Varying $\mathbf{R}$

$\mathbf{R}$ tests again produced the highest error increase from the initial given error as well as the second highest ending error, only being eclipsed by the already highly set $\mathbf{P}$ error. Similarly to the Trace function case, the error increased non-linearly as the $\mathbf{R}$ increased. Also similarly to the Trace function case, the $\mathbf{R} = 10 * \mathbf{R}$ tests found was the only $\mathbf{R}$ test in which each trial came to the exact same ending error value.

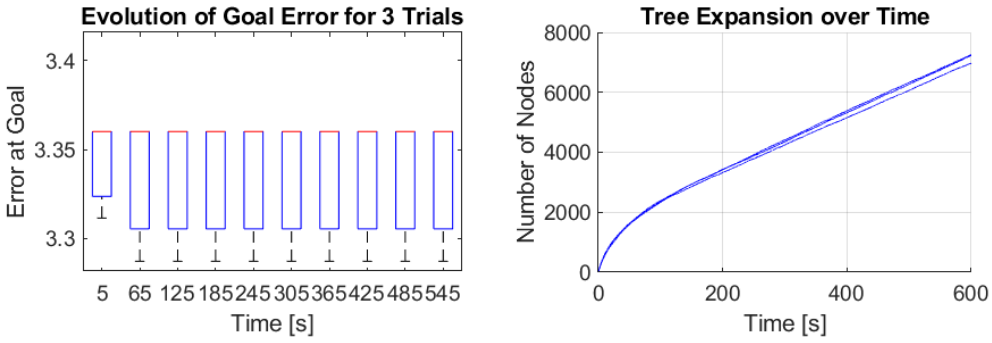

Figure 4.20: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

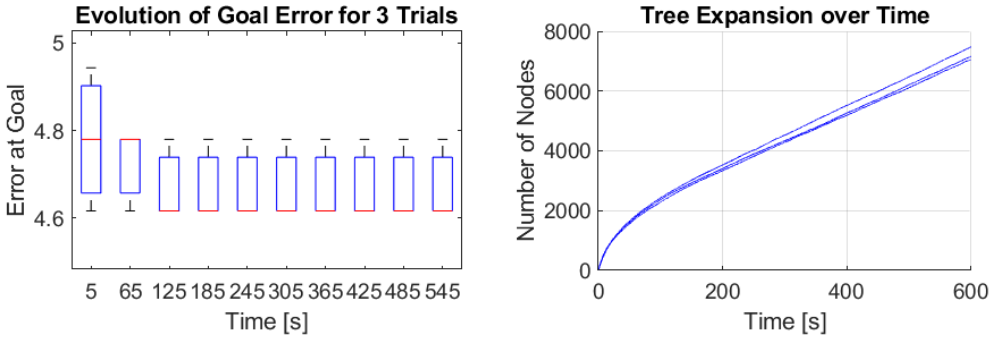

Figure 4.21: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

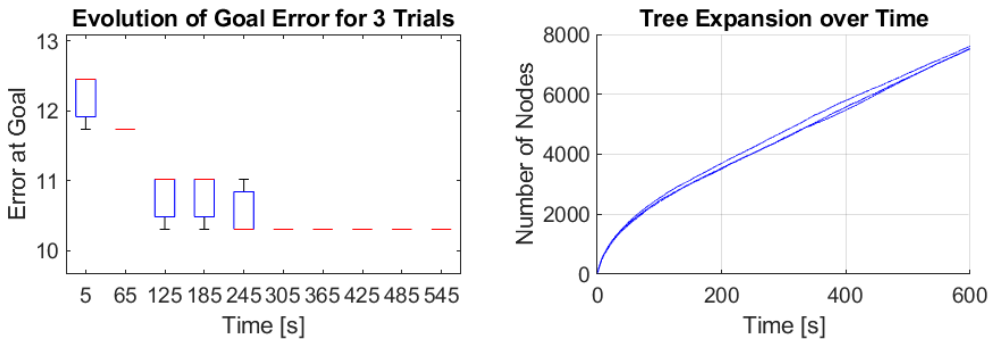

Figure 4.22: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

4.2 RRT* and RRT# Comparison for 2D Toy Problem

This section will be describing the comparison between the RRT* and RRT# motion planning algorithms. For comparison, RRT* was run with the same parameters as RRT#, including the same start and goal position, and run for the same amount of time. Also, RRT* was run for all of the **R** cases for both error functions, thus allowing a decent comparison for a variable.

One major difference across all of the tests is that RRT* was able to add more nodes to the graph. The specific differences are noted in the following subsections as they will be compared to the corresponding RRT# data. Additionally, RRT* was able to calculate its first solution more quickly than RRT#, which will be expanded upon in the following sections. The summary of the ending errors of all the test results for the RRT* tests are shown in figure 4.23.

This figure shows the end results for each of the test cases for both the Trace function, noted with a TR in the figure, and the Volumetric Ellipsoid function, noted with a VE in the figure. Both groupings of results match the same pattern

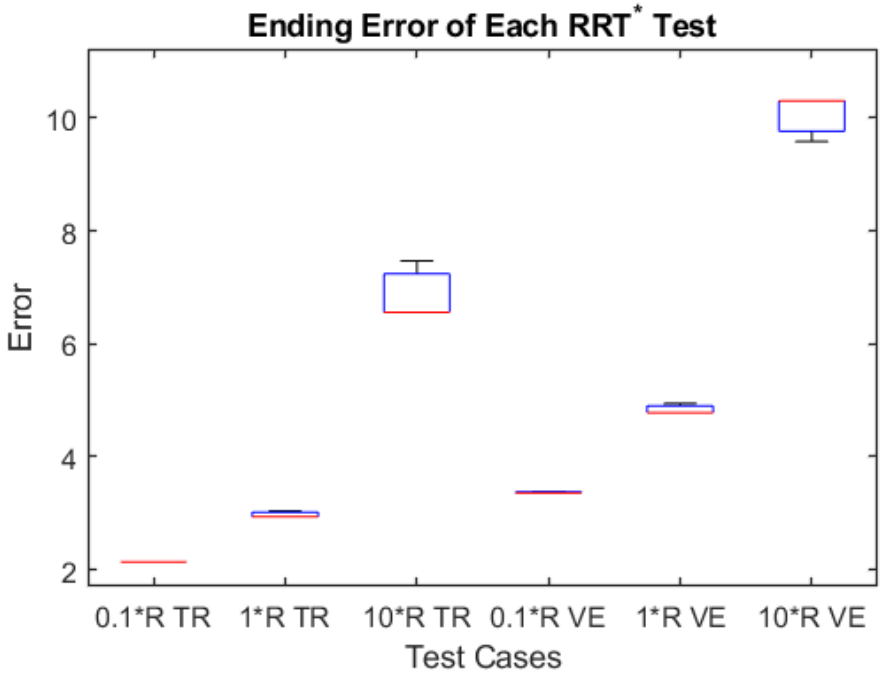

Figure 4.23: Ending Error of RRT* Tests

of a non-linear rise with the increasing multiplier to **R**. Also, this graph shows that as the multiplier increased, the range of error values also increased, leading to the assumption that with higher input error there will be a wider range of ending error making it harder to predict.

4.2.1 Algorithm Comparison Using Trace Function

RRT* runs in a much quicker method than that of RRT#, with less computation time dedicated to propagating effects of new additions to the path, it allows the algorithm to add more nodes to the graph. In these tests, RRT* obtained an average of 12,534 nodes in the graph, an increase of 76.51% from RRT#. When running the trace error function, RRT* was able to obtain its first solution in an average time of 0.43 seconds. RRT# was only able to reach its first solution in an

average of 0.74 seconds, thus giving RRT* an improvement of 41.59% in reaching the first solution.

Comparison of the ending error of each test with the corresponding RRT# test is shown in table 4.3. This shows that RRT# performs slightly better or equal to RRT* for all of the cases, but only by a slim margin of 2.5% in total.

Case	Average RRT* Error	Average RRT# Error
$\mathbf{R} = 0.1 * \mathbf{R}$	2.1391	2.1339
$\mathbf{R} = 1 * \mathbf{R}$	2.9733	2.9733
$\mathbf{R} = 10 * \mathbf{R}$	6.8826	6.5587
Total	3.9917	3.8886

Table 4.3: Average Ending Error

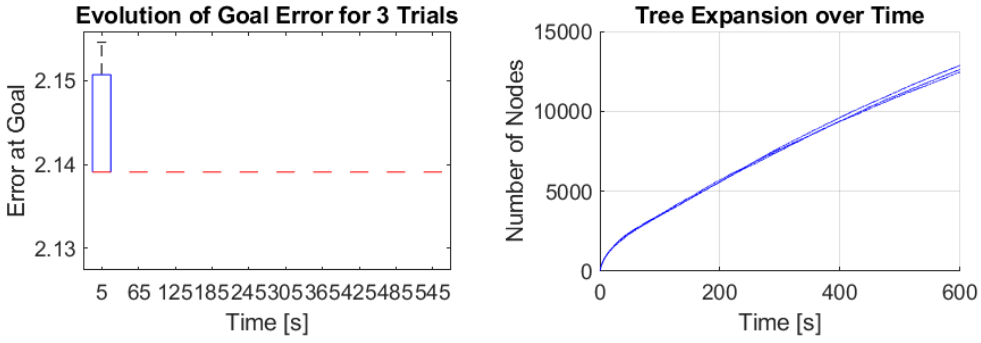

Figure 4.24: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

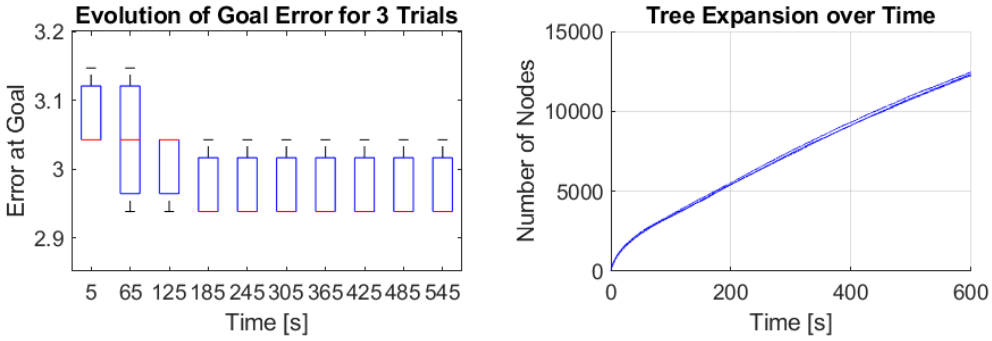

Figure 4.25: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

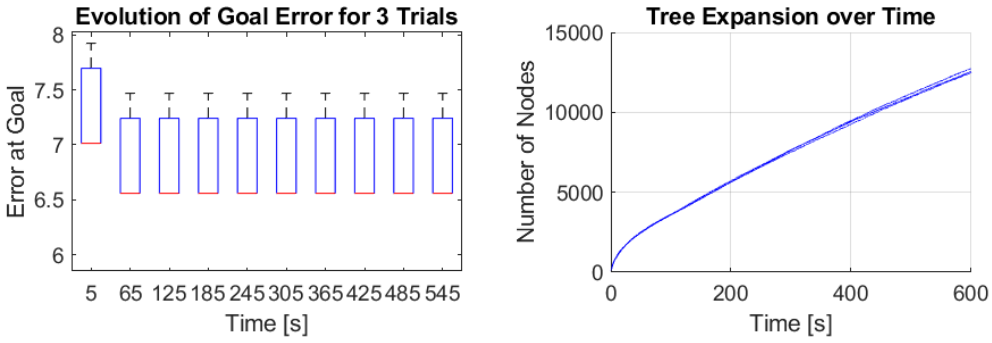

Figure 4.26: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

4.2.2 Algorithm Comparison Using Volumetric Ellipsoid Function

For comparing the Volumetric Ellipsoid function, again RRT* proves faster than RRT# in reaching its first solution in an average time of 0.45 seconds. RRT# was only able to reach its first solution in an average of 0.51 seconds; a slightly worse improvement of only 10.92% when compared to the Trace function. Also, RRT* again is able to grow to a larger size than RRT# with an average of 12,673 nodes in the graph, a greater expansion of 73.27% compared to RRT#'s 7,314 nodes on average.

In comparison of ending error, table 4.4 shows that RRT* performed a very miniscule 0.23% better than RRT# in total average only because RRT# performed so poorly in the $\mathbf{R} = 10 * \mathbf{R}$ case. However, figure 4.29 shows that some trials of the error were able to perform better than the average, reaching a lower value of 9.5863 which is a much better 6.95% improvement of the RRT# average.

Case	Average RRT* Error	Average RRT# Error
$\mathbf{R} = 0.1 * \mathbf{R}$	3.3682	3.3358
$\mathbf{R} = 1 * \mathbf{R}$	4.8342	4.6704
$\mathbf{R} = 10 * \mathbf{R}$	10.0636	10.3023
Total	6.0887	6.1028

Table 4.4: Average Ending Error

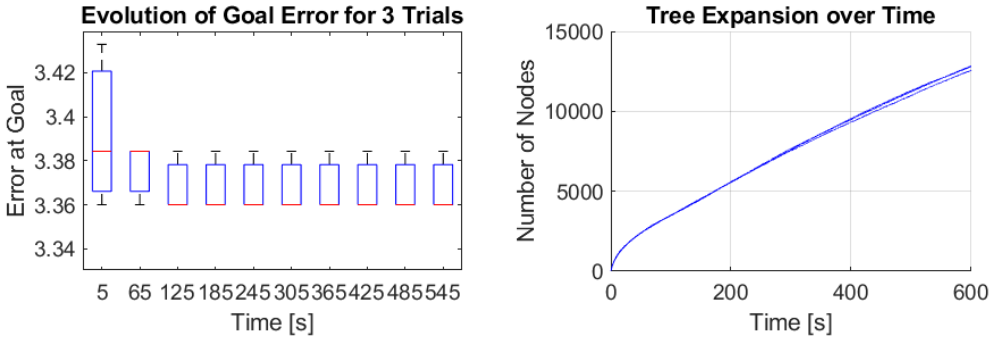

Figure 4.27: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

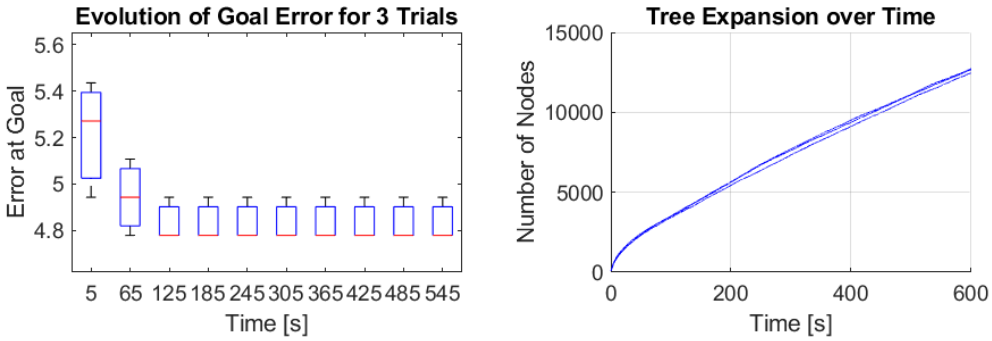

Figure 4.28: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

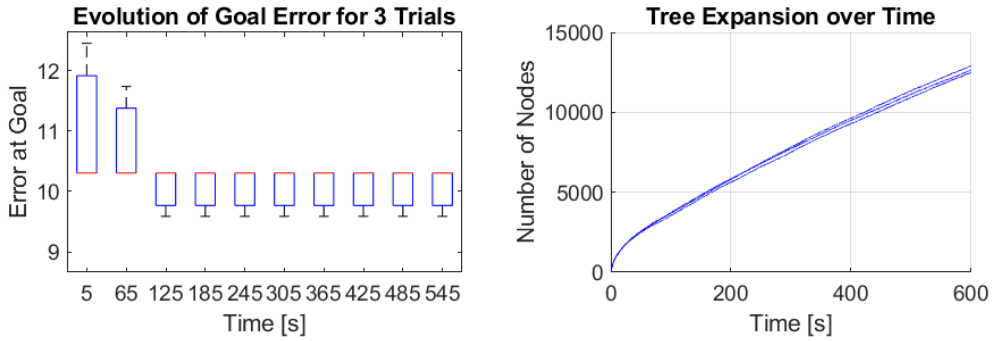

Figure 4.29: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

In summary of the 2D Toy problem comparison of RRT* and RRT#, RRT* runs more quickly than RRT# and is able to reach an initial solution faster than RRT#; however, when given a longer amount of time to run, RRT# is able to reach a better more consistent solution based on the testing done in this work.

4.3 Discussion of Stewart Platform Problem Results

Within this section, the results of using both the Trace function and the Volumetric Ellipsoid function are discussed along with a comparison of RRT* and RRT# for the Stewart platform problem. These results highlight the difference in the application of the motion planning algorithms as well as the error functions on the Stewart platform compared to the 2D Toy problem.

Following the summary of results in section 4.3.1 for the Trace function are the tests results for each variable, in sections 4.3.1.1-4.3.1.3. Similarly, following the summary of results in section 4.3.2 for the Volumetric Ellipsoid function are the test results of each variable, in sections 4.1.2.1-4.1.2.3. Lastly, a comparison of RRT* and RRT# when used on the Stewart platform is shown in section 4.4.

4.3.1 Noise Variation Using Trace Function

Table 4.5 shows that the test cases in which the process noise covariance matrix, **Q**, is varied corresponds to the highest average ending error as well as the highest change in error from the starting value. The observation noise covariance matrix, **R**, corresponds to the lowest average ending error and total error addition,

a change in the results shown in the 2D Toy problem. Also, the error covariance matrix, **P**, responded approximately as expected when it was varied, this will be expanded upon below.

Case	Initial Error	Error Addition	Ending Error
P	0.6, 6, 60	40.6265	62.8265
Q	6	166.7978	173.7978
R	6	37.6207	43.6207
Total	*N/A*	81.6817	93.0817

Table 4.5: Average Error Addition and Ending Error

Since **Q** error is highest and since using a higher **Q** value means the algorithm trusts the measurement more, either the measurement being taken is more inaccurate than the estimate or the algorithm is very susceptible to process noise; skewing results more. This point is strengthened when looking at the data of the **R** value test cases shown in figure 4.30. As the **R** value is increased, the average ending error actually goes down rather than increases as in other tests. This shows that when the algorithm relies on its own estimate more than observations being taken it has better performance.

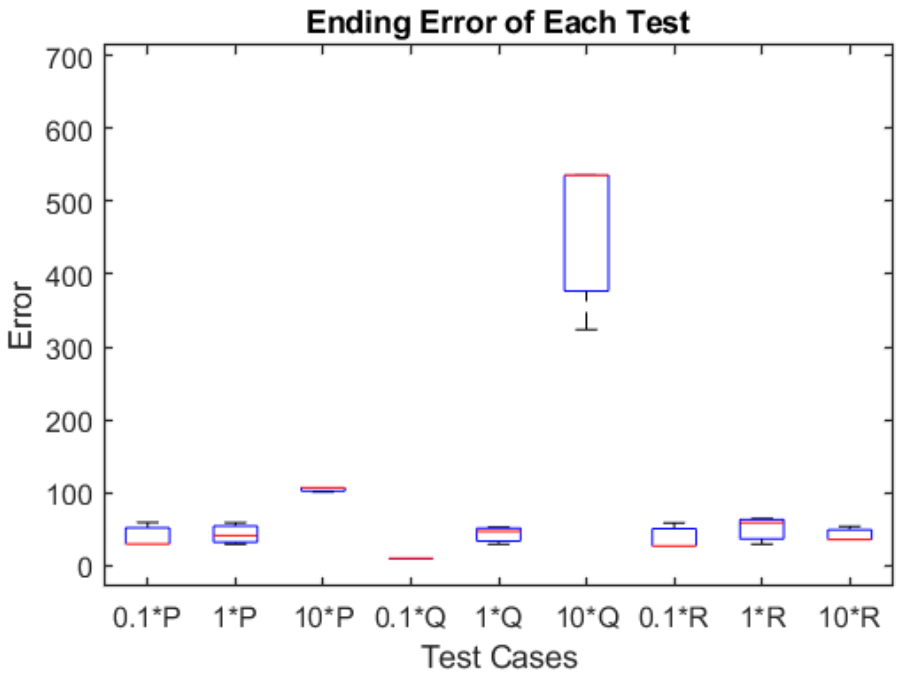

Figure 4.30: Box Plot of Ending Errors using Trace Function

An example plot of the path is shown in figure 4.31 where the path taken is represented in two different spaces. The left side is the position space in units of millimeters and the right side is the rotation space in units of degrees. The light blue spheres shown in the orientation space are the representation of the error ellipsoids associated with each movement. These ellipsoids grow with the error at each stage but are relatively small. Within both of these figures, ◯ marks the starting position, × marks the goal position, and the blue line is the path taken from start to goal.

In all of the Trace cases, the trees expanded in a somewhat non-linear fashion, more noticeably so in the $\mathbf{R} = 10 * \mathbf{R}$ and $\mathbf{Q} = 10 * \mathbf{Q}$ cases. Theses two cases were also the only two that averaged significantly less nodes in the graph than the other cases with 3,063 and 3,402, respectively. These values are roughly half the average of all the other cases which comes to 6,552 nodes in the graph.

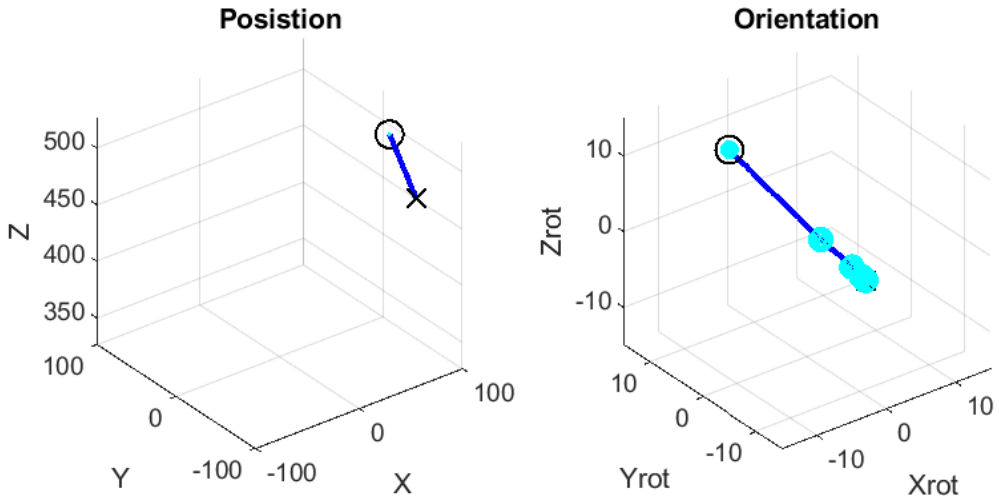

Figure 4.31: Example Trajectory Using Trace Function

4.3.1.1 Varying $\mathbf{P}$

Varying $\mathbf{P}$ produced some interesting results, with the $\mathbf{P} = 0.1 * \mathbf{P}$ and $\mathbf{P} = 1 * \mathbf{P}$ cases producing almost the same ending error at 39.915 and 43.3491, respectively. This was an increase of only 8.6% between the two cases whereas the increase from $\mathbf{P} = 1 * \mathbf{P}$ to $\mathbf{P} = 10 * \mathbf{P}$ was 142.7%. This was a massive jump, but can be attributed to the starting error being 60 for the $\mathbf{P} = 10 * \mathbf{P}$ case.

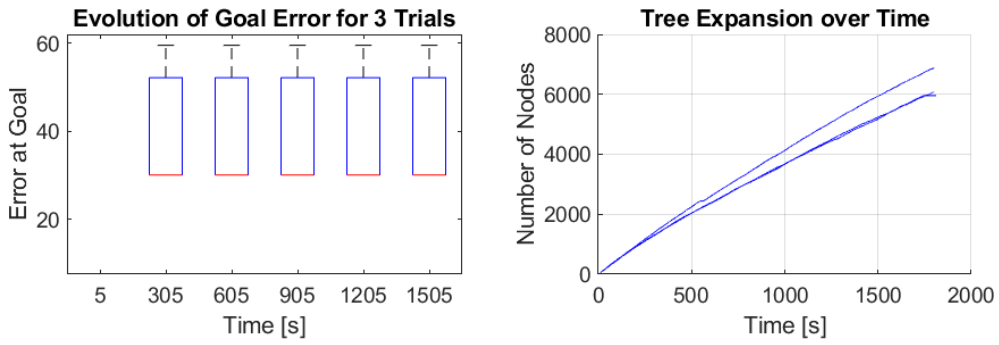

Figure 4.32: $\mathbf{P} = 0.1 * \mathbf{P}$ Test Data

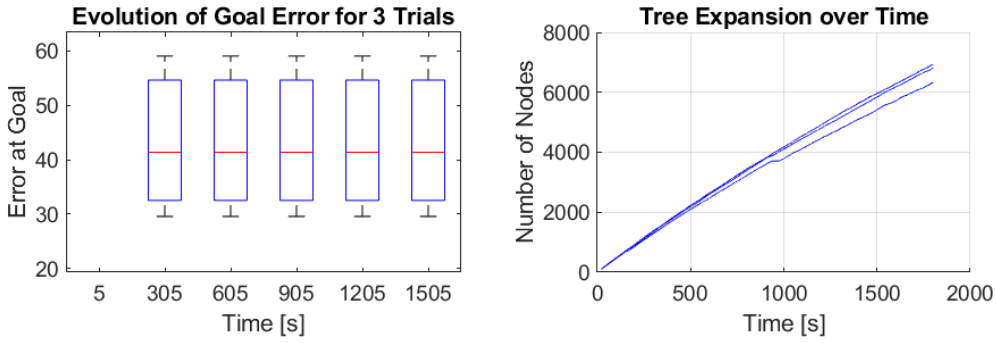

Figure 4.33: $\mathbf{P} = 1 * \mathbf{P}$ Test Data

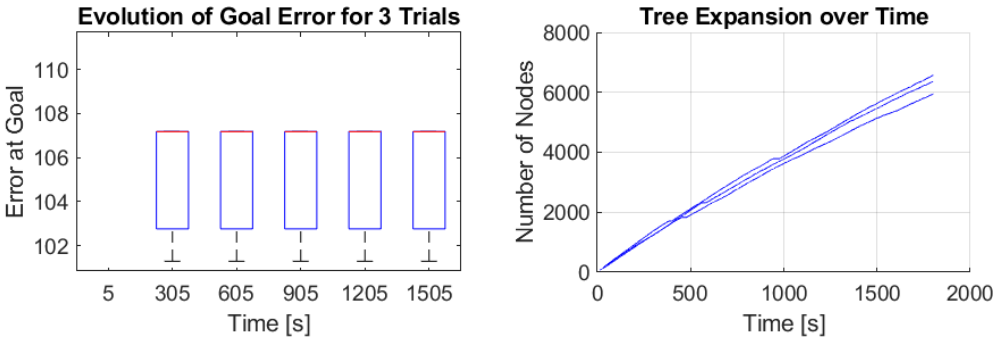

Figure 4.34: $\mathbf{P} = 10 * \mathbf{P}$ Test Data

4.3.1.2 Varying $\mathbf{Q}$

One of the most surprising results was the ending error increase of $\mathbf{Q}$ across the trials. From an increase of 333.7% between $\mathbf{Q} = 0.1 * \mathbf{Q}$ to $\mathbf{Q} = 1 * \mathbf{Q}$ to an astonishing increase of 972.9% between $\mathbf{Q} = 1 * \mathbf{Q}$ to $\mathbf{Q} = 10 * \mathbf{Q}$. This is quite a deviation from the results seen in the 2D problem but can possibly be explained by the functions being used. Since a larger $\mathbf{Q}$ than $\mathbf{R}$ corresponds to a higher correction of the estimate, the steering function is correcting the state estimate more than it needs to. On top of this is that since the $\mathbf{Q}$ is quite large in the $\mathbf{Q} = 10 * \mathbf{Q}$ case, much more noise is being added into the system resulting in a much higher error.

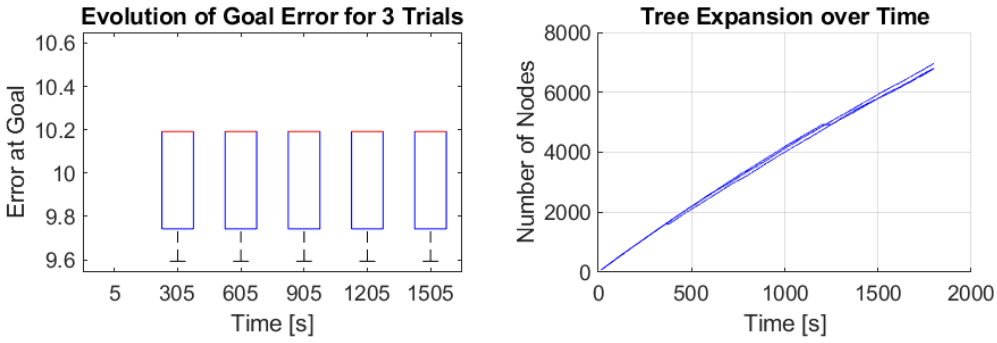

Figure 4.35: $\mathbf{Q} = 0.1 * \mathbf{Q}$ Test Data

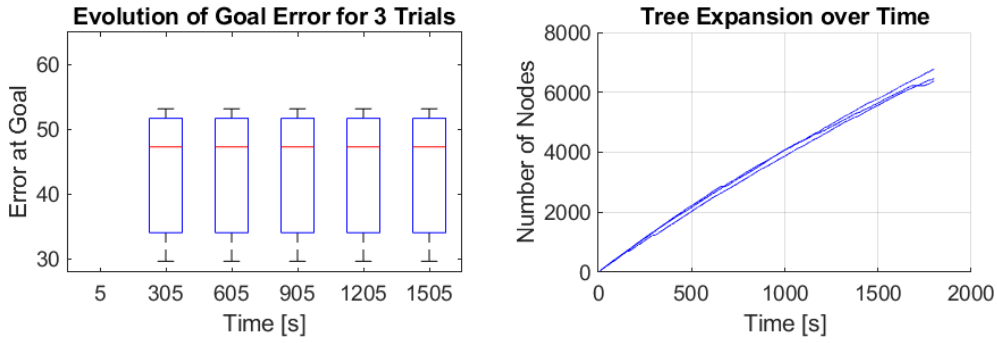

Figure 4.36: $\mathbf{Q} = 1 * \mathbf{Q}$ Test Data

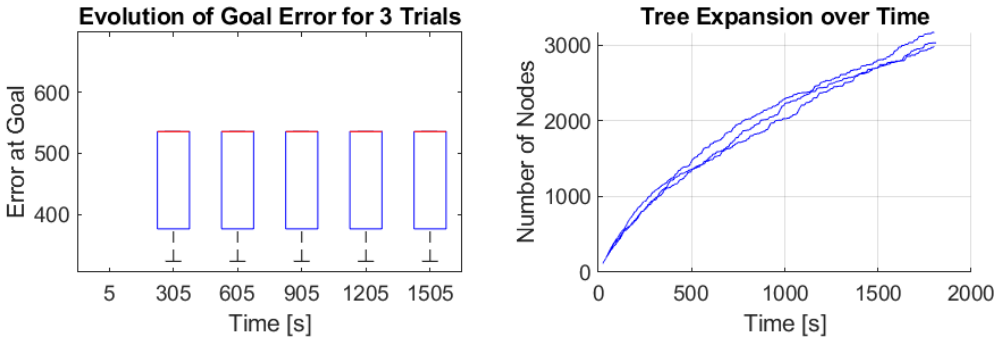

Figure 4.37: $\mathbf{Q} = 10 * \mathbf{Q}$ Test Data

4.3.1.3 Varying $\mathbf{R}$

Another surprising result is that the $\mathbf{R}$ tests actually decreased in ending error as $\mathbf{R}$ increased in size. Specifically, $\mathbf{R} = 1 * \mathbf{R}$ to $\mathbf{R} = 10 * \mathbf{R}$ resulted in a 18.11%

decrease in error. A reason for this decrease could be the same effect that caused the massive spike in the **Q** yet with opposite results. With an increase in **R** the steering function corrects the a priori estimate less based on the observation being taken, thus relying more on the initial estimate.

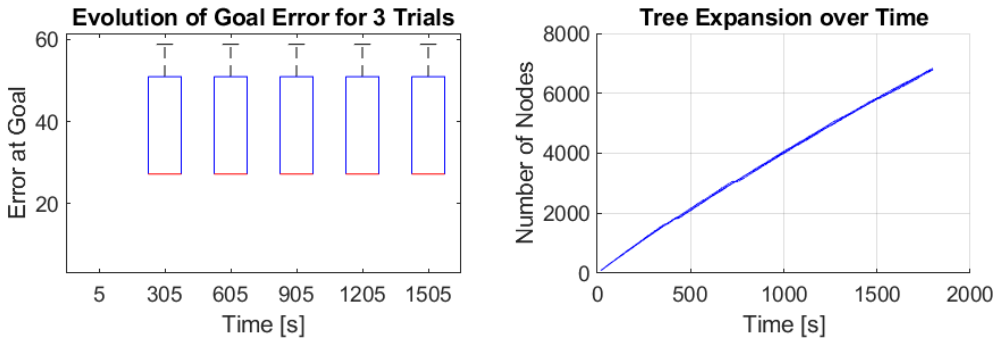

Figure 4.38: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

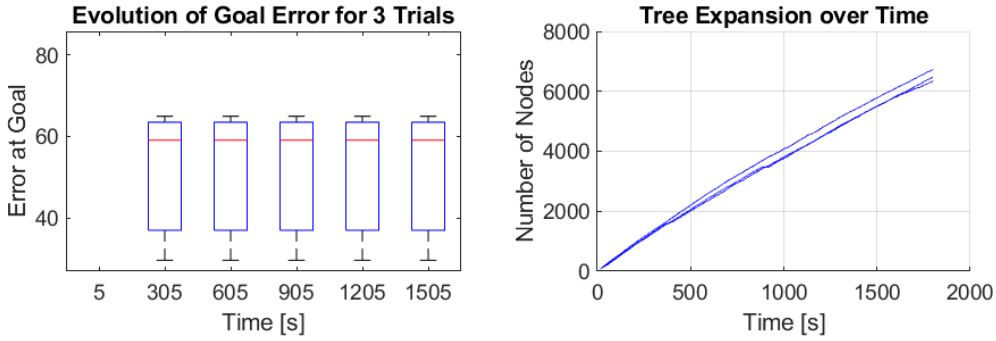

Figure 4.39: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

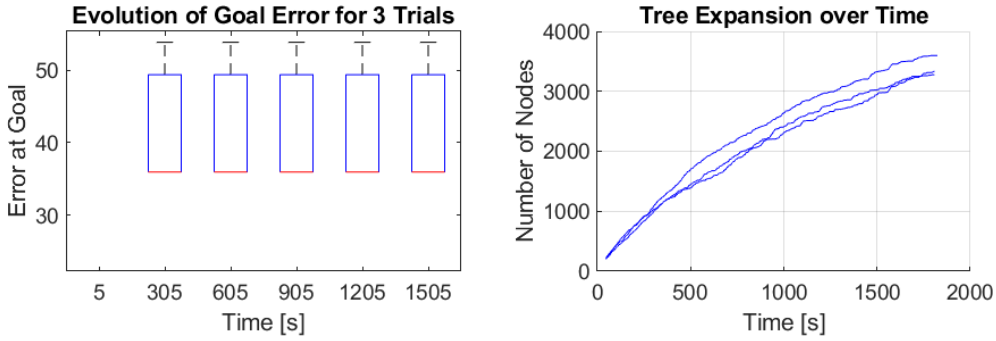

Figure 4.40: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

4.3.2 Noise Variation Using Volumetric Ellipsoid Function

Using the Volumetric Ellipsoid function with a six dimensional problem causes extremely small error, on the order of 10^{-7} of total error increase over the entire length of the path. Node to node error increase is minimal to the point that MATLAB rounding error in the solution of some functions plays a part. Specifically, in the case of the steering function when calculating $\mathbf{P}$ and the innovation covariance, $\mathbf{S}$. When initially ran, these matrices would come out to be non-symmetric which is not what is supposed to occur. When investigated further, the difference in symmetry was on the order of 10^{-16}. This was attributed to MATLAB rounding error in the calculation of the matrices. In order to fix this problem, a simple solution was used as shown in equation 4.1.

$$\mathbf{P} = \frac{\mathbf{P} + \mathbf{P}'}{2} \tag{4.1}$$

This turned out to not be a good method of tracking error for the Stewart platform as the little amount of error deviation was not useful in producing decent results. As can be seen in table 4.6 where the error addition is extremely small resulting in very little change from the initial error.

Case	**Initial Error**	**Error Addition**	**Ending Error**
P	$5.167 * 10^{-3}$, 5.1677, 5167	$6.8889 * 10^{-8}$	$1.7243 * 10^{3}$
Q	5.1677	$9.6 * 10^{-7}$	5.1677
R	5.1677	$7.9222 * 10^{-7}$	5.1677
Total	*N/A*	$6.0704 * 10^{-7}$	578.2102

Table 4.6: Average Error Addition and Ending Error

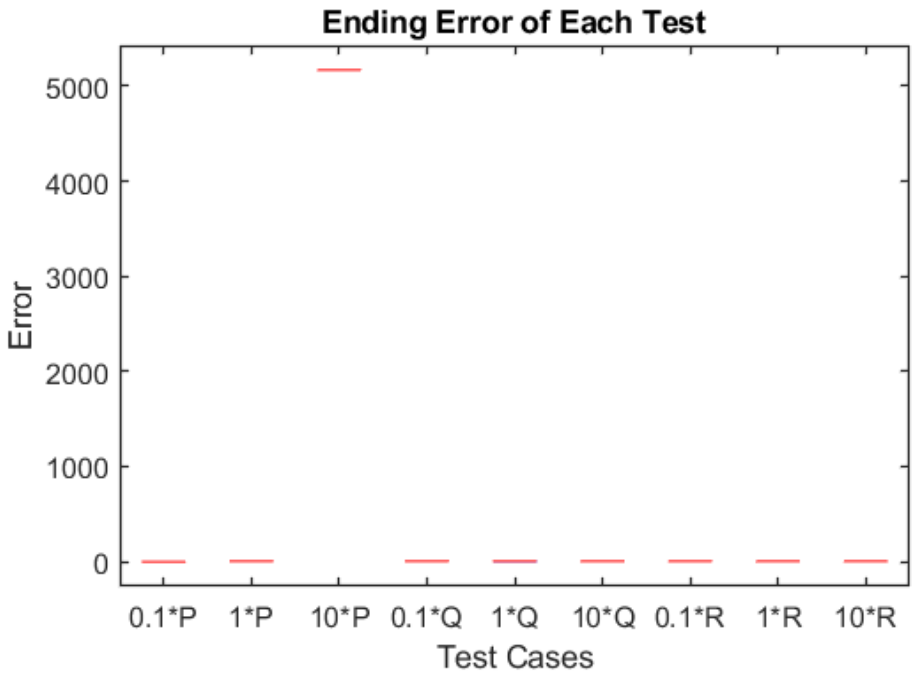

Figure 4.41: Box Plot of Ending Errors using Ellipsoid Function

4.3.2.1 Varying **P**

Using the Volumetric Ellipsoid function caused the error of the $\mathbf{P} = 10 * \mathbf{P}$ case to dwarf that of the rest of the test cases. This error averaged at 5,168 and is entirely due to the way the Volumetric Ellipsoid function handles **P** when calculating error. By taking the eigenvalues of a $10^{*}\mathbf{I}_{6x6}$ matrix and taking the product of all six eigenvalues, it multiplies the rest of the equation by 1,000. Besides this high ending error value, the **P** test cases resulted in the lowest error increase from the initial error.

Figure 4.42: $\mathbf{P} = 0.1 * \mathbf{P}$ Test Data

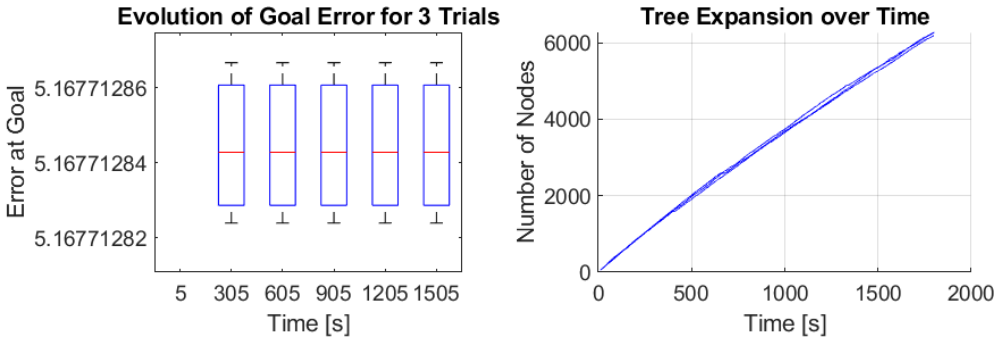

Figure 4.43: $\mathbf{P} = 1 * \mathbf{P}$ Test Data

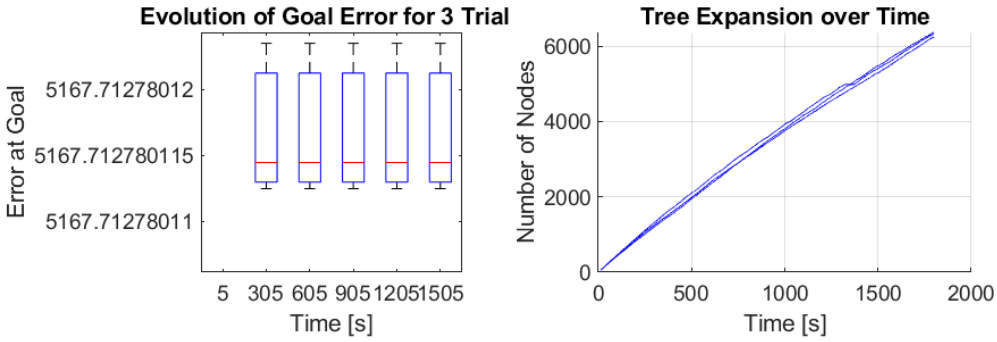

Figure 4.44: $\mathbf{P} = 10 * \mathbf{P}$ Test Data

4.3.2.2 Varying **Q**

Unlike the Trace function results for **Q** testing, the Volumetric Ellipsoid function did not produce a high change in error across the range of multipliers because

the error values were so small the change was not that noticeable. However the difference in change between the $\mathbf{Q} = 0.1 * \mathbf{Q}$ to $\mathbf{Q} = 1 * \mathbf{Q}$ and $\mathbf{Q} = 1 * \mathbf{Q}$ to $\mathbf{Q} = 10 * \mathbf{Q}$ was a 59.7% increase. The similarity of this test with the 2D Toy problem test was that the $\mathbf{Q} = 10 * \mathbf{Q}$ resulted in the lowest average tree expansion of only 2,944 nodes.

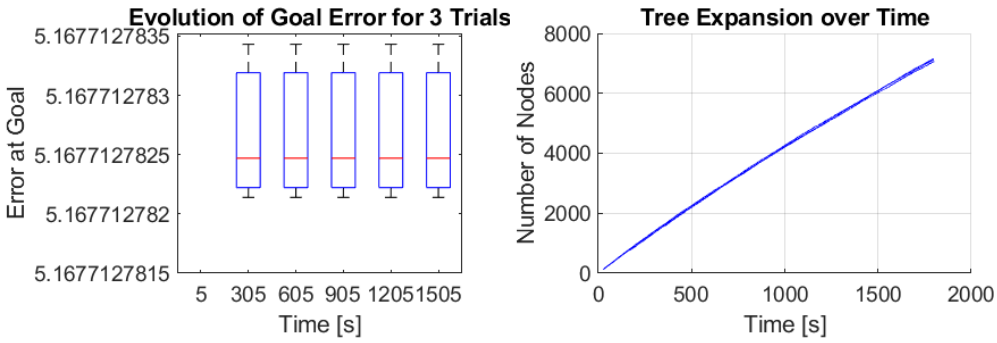

Figure 4.45: $\mathbf{Q} = 0.1 * \mathbf{Q}$ Test Data

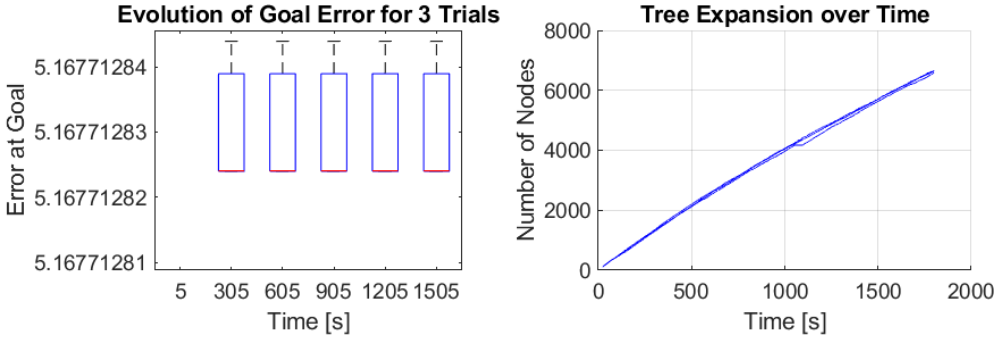

Figure 4.46: $\mathbf{Q} = 1 * \mathbf{Q}$ Test Data

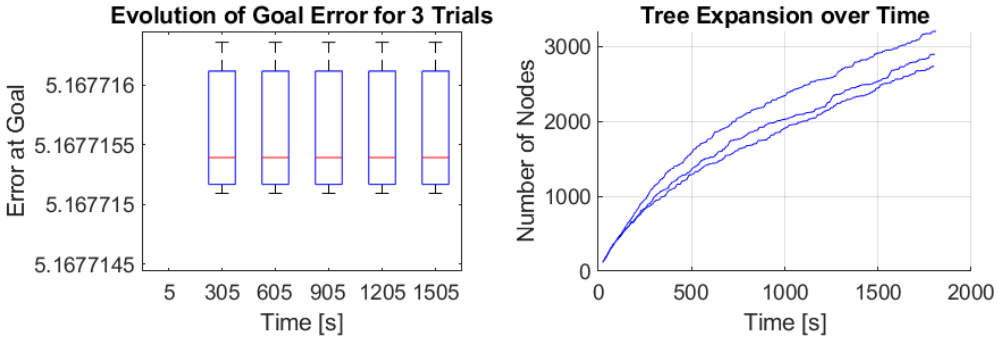

Figure 4.47: $\mathbf{Q} = 10 * \mathbf{Q}$ Test Data

4.3.2.3 Varying $\mathbf{R}$

The $\mathbf{R}$ tests for the Volumetric Ellipsoid function produced very similar results as the $\mathbf{Q}$ tests. The $\mathbf{R}$ tests had an increase in ending error with each successive raise in the multiplier applied to $\mathbf{R}$, unlike the 2D Toy problem tests. Similarly to the Trace function tests, the $\mathbf{R} = 10 * \mathbf{R}$ test also expanded less than other tests, to only an average of 3,351 nodes. The total average of all tests outside of this test and the $\mathbf{Q} = 10 * \mathbf{Q}$ test was 6,639 nodes, meaning only half as many nodes were visited compared to other testing.

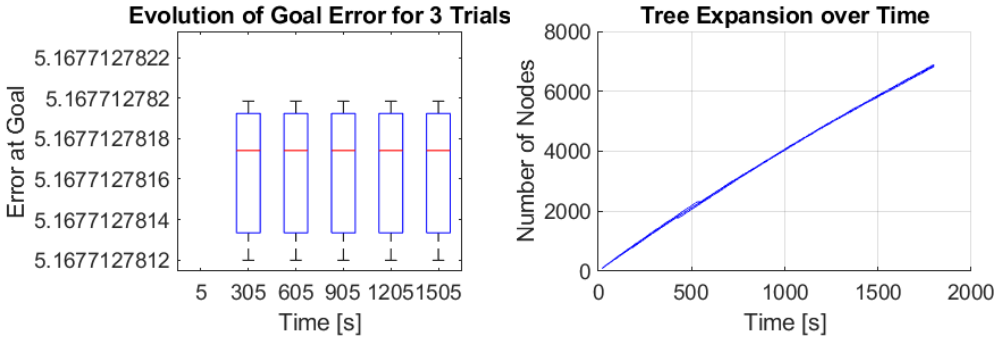

Figure 4.48: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

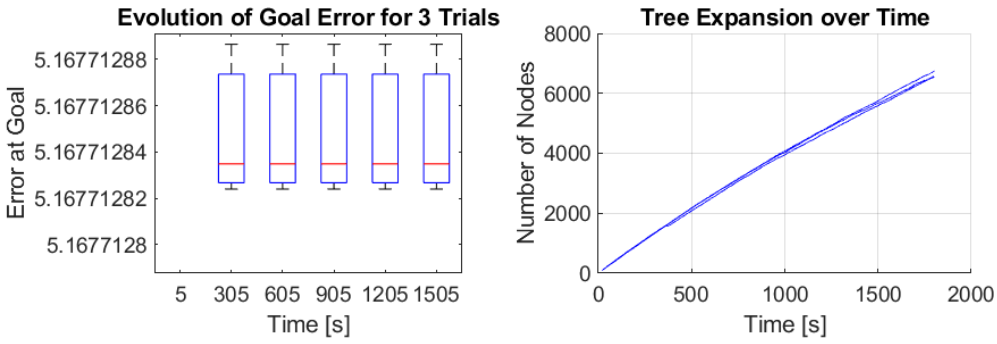

Figure 4.49: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

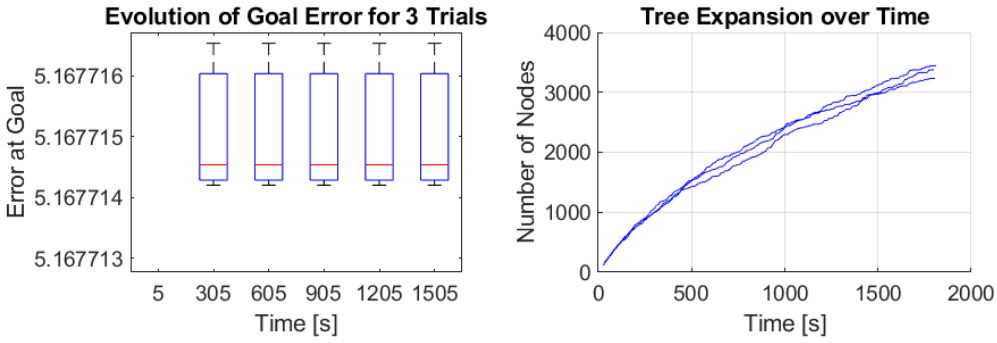

Figure 4.50: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

4.4 RRT* and RRT# Comparison for Stewart Platform Problem

Within this section is the discussion of the comparisons of the RRT* and RRT# motion planning algorithms using both error functions. The method for testing this comparison is equivalent to the method used for the 2D Toy problem. The same starting position, goal position, parameters, and run time are used for the RRT* algorithm. Also like the 2D Toy problem, the algorithm was over the variations of the $\mathbf{R}$ case.

Also similar to the 2D Toy problem was the fact that RRT* was able to add more nodes to the graph in the same run time. With a total average of 8,415 nodes compared to only 5,863 nodes for RRT#. The summarized results of the tests are shown in figure 4.51. This figure shows similar results as experienced in the RRT# tests where a very tiny error deviation occurred when using the Volumetric Ellipsoid function. Each error function will be individually analyzed in the sections below.

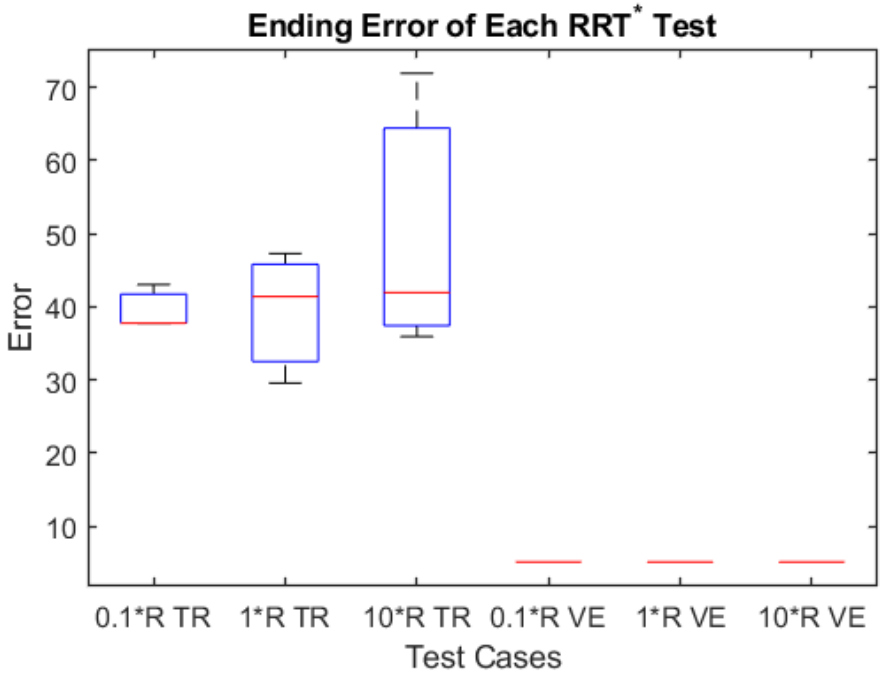

Figure 4.51: Ending Error of RRT* Tests

4.4.1 Algorithm Comparison Using Trace Function

RRT* is a quicker running algorithm than RRT#, again being able to reach a first solution faster than RRT# and obtain more nodes in the graph than RRT#. For quickness, RRT* was able to obtain the first solution in an average of 14.61 seconds whereas RRT# took 31.22 seconds, giving RRT* a 53.2% faster time to first solution. As mentioned earlier, RRT* was able to grow to a larger size than RRT#, with it being able to grow to an average of 8,475 nodes using the Trace function. An increase in size of 53.2% compared to RRT# reaching only 5,532 nodes.

Comparison of the ending error of each test with the corresponding RRT# test is shown in table 4.7. This table shows that RRT# performed worse than RRT* by only 1.54%. This was mainly due to the $\mathbf{R} = 1 * \mathbf{R}$ tests of RRT# performing

so poorly. Without these tests performing poorly, RRT# would have shown to be better as it outperformed RRT* in both the $\mathbf{R} = 0.1 * \mathbf{R}$ and $\mathbf{R} = 10 * \mathbf{R}$ tests.

Case	Average RRT* Error	Average RRT# Error
$\mathbf{R} = 0.1 * \mathbf{R}$	39.5196	37.7273
$\mathbf{R} = 1 * \mathbf{R}$	39.4137	51.2036
$\mathbf{R} = 10 * \mathbf{R}$	49.9151	41.9312
Total	42.9494	43.6207

Table 4.7: Average Ending Error

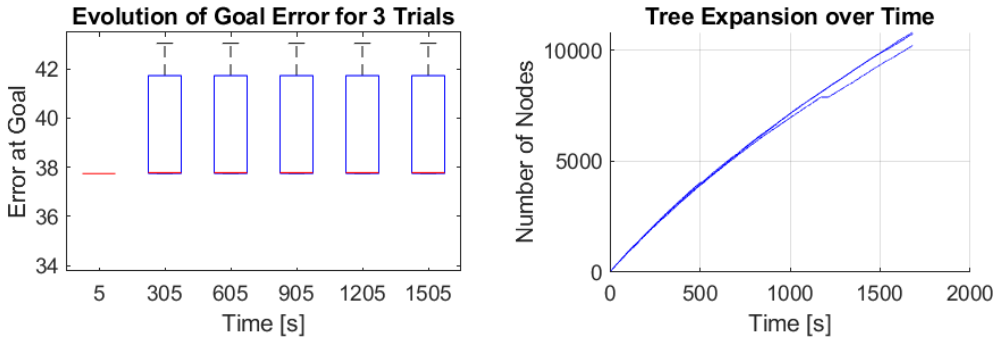

Figure 4.52: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

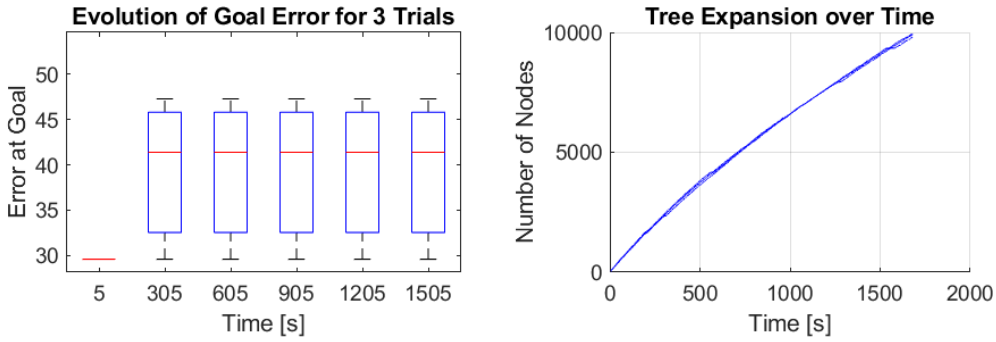

Figure 4.53: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

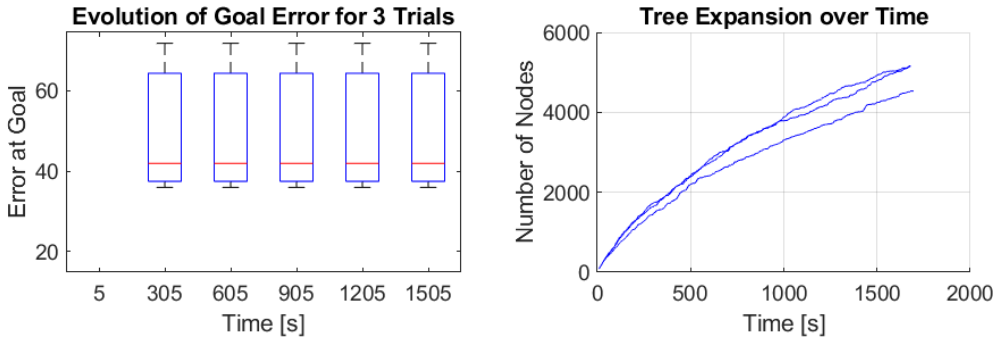

Figure 4.54: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

4.4.2 Algorithm Comparison Using Volumetric Ellipsoid Function

For comparing the Volumetric Ellipsoid function, again RRT* proves faster than RRT# reaching its first solution in an average time of 12.39 seconds whereas RRT# was only able to reach its first solution in an average time of 27.33 seconds, an improvement upon the Trace function time of 15.14%. Again RRT* was able to reach more nodes than RRT#, with an average of 8,355 nodes compared to 5,608 nodes for RRT#. Table 4.8 shows that RRT# is minimally better than RRT* with a total average ending error difference of $9 * 10^{-8}$. This shows that the Volumetric Ellipsoid function is not well suited for use with the RRT* algorithm, as was the case with the RRT# algorithm. The error itself is minuscule compared to the error values seen in the Trace function and is slightly more computationally expensive but able to reach the first solution slightly quicker than the Trace function.

Case	Average RRT* Error	Average RRT# Error
$\mathbf{R} = 0.1 * \mathbf{R}$	5.16771278	5.16771278
$\mathbf{R} = 1 * \mathbf{R}$	5.16771285	5.16771285
$\mathbf{R} = 10 * \mathbf{R}$	5.16771535	5.167715090
Total	5.16771366	5.16771357

Table 4.8: Average Ending Error

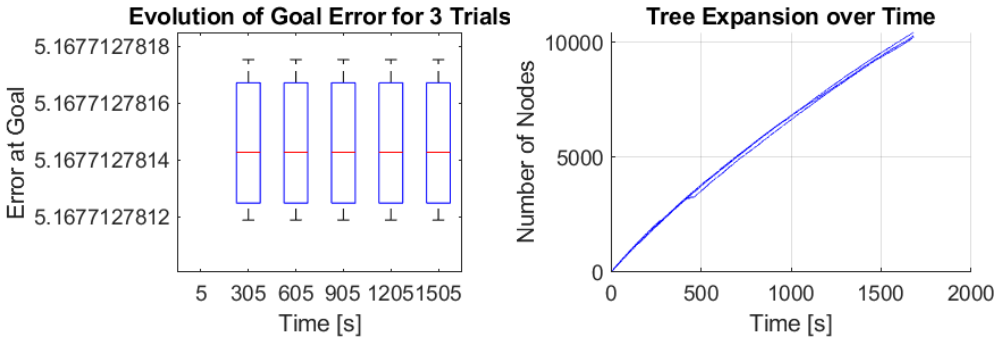

Figure 4.55: $\mathbf{R} = 0.1 * \mathbf{R}$ Test Data

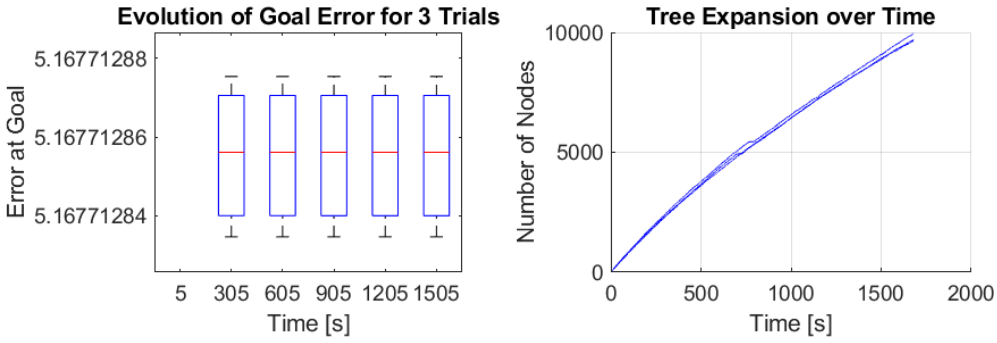

Figure 4.56: $\mathbf{R} = 1 * \mathbf{R}$ Test Data

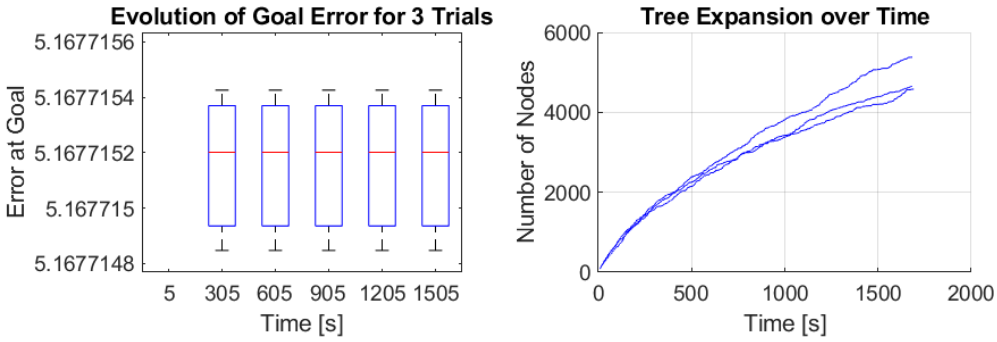

Figure 4.57: $\mathbf{R} = 10 * \mathbf{R}$ Test Data

In final comparison of the RRT* and RRT# motion planning algorithms, RRT# performs better for accuracy when used on a Stewart platform, keeping the same result as in the 2D Toy problem case.

4.5 Additional RRT# Testing on the Stewart Platform

Within this platform is additional testing done on the Stewart platform at various locations of the workspace. A total of ten different starting and goal cases were tested. Cases 1-5 were randomly selected and cases 6-10 were hand picked based on certain aspects or to create certain paths for the Stewart platform to move through. The full list is shown in table 4.9. For each of these cases, five trials were run using the Volumetric Ellipsoid function, available in section 4.5.1, and five trials were run using the Trace function, available in section 4.5.2. In the setup of these tests, the error covariance matrix, $\mathbf{P}$, was set to $\mathbf{P} = 1 * \mathbf{P}$. Similarly, the observation noise covariance matrix, $\mathbf{R}$, and process noise covariance matrix, $\mathbf{Q}$, were set to $\mathbf{R} = 1*\mathbf{R}$ and $\mathbf{Q} = 1*\mathbf{Q}$, respectively. This allowed for a simple common testing setting for all cases. Additionally, the runtime was set at 30 minutes, similar to the previously run cases. All other values were kept consistent with the previously run RRT# cases.

The positions for case 6 were chosen to test how the algorithm would handle a large movement from the bottom of its space to the top of the opposite side along with some rotation of the moving plate. This gives an idea of how well the algorithm works with large movements as this is one of the largest movements the Stewart platform could see. Case 7 was picked for a similar reason as case 6; however the start and goal sides were inverted, meaning the start is at the top of the region and the goal is at the bottom. Case 8 was chosen to see how the algorithm handles a purely vertical movement with some rotation added in. Case 9 was chosen to see

Case #	Start Position	Goal Position
1	[-78.05, 28.41, 447.63, -13.33°, 6.46°, 4.98°]	[79.43, -11.12, 434.12, -9.03°, 6.05°, 12.77°]
2	[24.8, 8.04, 473.2, -8.98°, 13.44°, -14.12°]	[13.64, -64.57, 347.92, -8.16°, -8.45°, -7.58°]
3	[66.73, -65.85, 476.99, -7.55°, -11.44°, 8.44°]	[-58.94, 80.32, 389.53, 11.78°, -14.92°, 13.23°]
4	[16.47, 11.46, 422.55, 8.69°, -14.38°, 3.85°]	[38.35, .72.13, 523.54, -10.68°, 9.95°, 4.86°]
5	[37.84, 91.48, 354.33, 0.44°, 7.16°, 2.46°]	[76.07, -29.19, 470.6, -10.72°, 13.28°, -1.37°]
6	[-100, -100, 332, 5°, 5°, 5°]	[100, 100, 487, -5°, -5°, -5°]
7	[-100, 100, 487, 5°, 5°, 5°]	[100, -100, 332, -10°, 5°, -10°]
8	[0, 0, 377, -15°, -15°, -15°]	[0, 0, 527, 0°, 0°, 0°]
9	[100, -100, 427, 15°, -15°, 15°]	[-100, 100, 427, -15°, 15°, -15°]
10	[0, -100, 427, 0°, 0°, 0°]	[0, 100, 427, 0°, 0°, 0°]

Table 4.9: Additional Test States

how the algorithm handles large changes in rotation along with straight line motion as this path is all within the same Z-axis. Case 10 was chosen to show how the algorithm handles no rotation and a pure straight line motion along the Y-axis.

4.5.1 Testing with Volumetric Ellipsoid Function

Figure 4.58 shows the range and average error accumulation of the additional Volumetric Ellipsoid based tests. The accumulated error stated fairly consistent on average between $6 * 10^{-8}$ and $10 * 10^{-}8$ with outliers being cases 4, 8, and 9. Case 8 had a smaller amount of average error than most other cases as the linear motion was purely within the Z direction and the rotation motion was about all three axes,

thus meaning there would be less error about the other two linear directions. Case 9 had the highest average error as it was a combination of long travel distance in the X and Y direction combined with the largest change in rotation of 30° about all three axes.

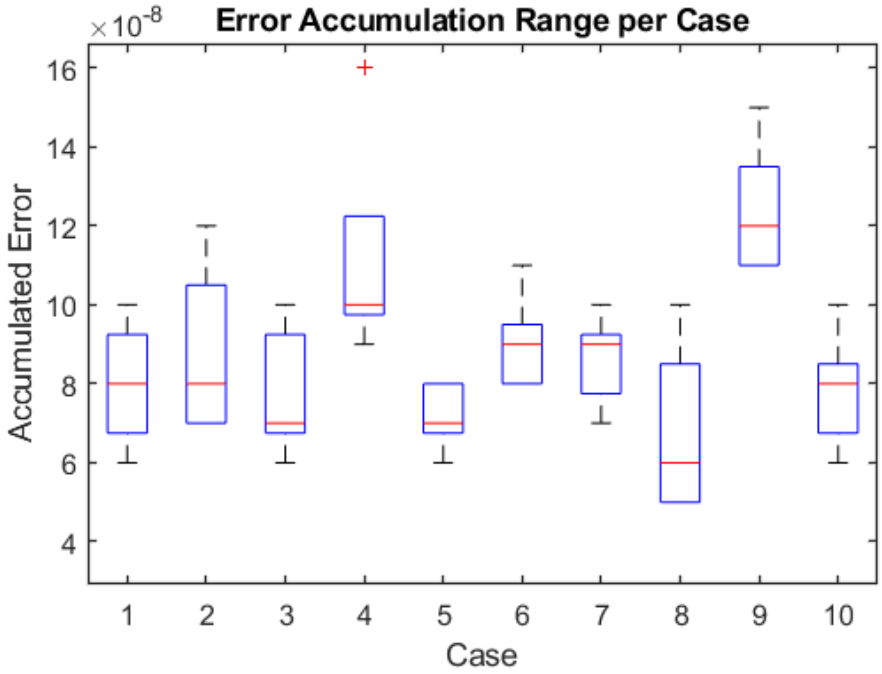

Figure 4.58: Ending Error of Additional Tests

Figures 4.59-4.68 show example paths taken within all ten cases. These paths were chosen to represent these cases because they were the best result of each of the trials ran. Within all of the Stewart platform testing of the motion planning algorithms, it was noticed that the produced error accumulation is extremely small compared to that of the trace function and it did not follow the same increase as the 2D case. Initially, this error was believed to come from the Volumetric Ellipsoid function performing poorly in higher dimensions; however with further investigation this was found not to be the case. The reasoning behind this error comes from the

adjusted error estimate within the steering function. The amount of error being added because of a measurement is most likely too small for a real life scenario. This small amount of error, especially small for the rotation elements, causes the Kalman gain to become quite small. In turn, meaning that the steering function is saying that there is very little error being added within each step of the steering function.

Figures 4.59 to 4.63 show the running of the motion planning algorithm within the five random cases. Within all of these figures, ◯ marks the starting position, × marks the goal position, and the blue line is the path taken from start to goal. An interesting observation is that in most of the cases, there is quite a deviation from what may be the intuitive path, i.e. a straight line from start to goal. Specifically within the orientation portion of these plots the graphs show quite sharp changes in orientation.

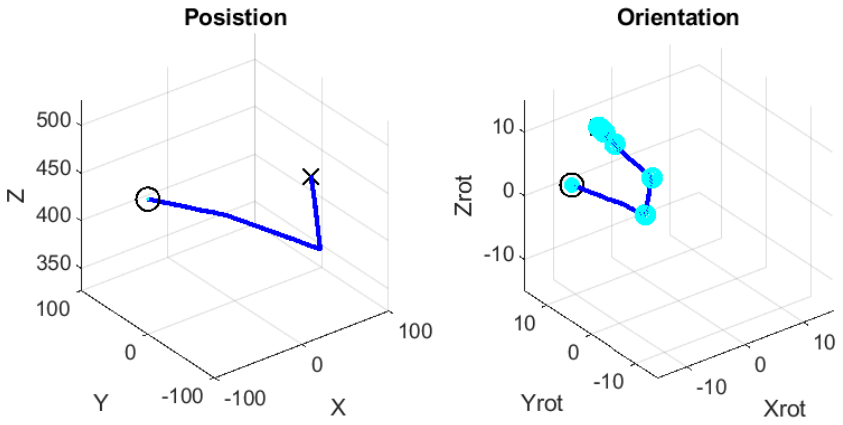

Figure 4.59: Case #1 Using Volumetric Ellipsoid Function

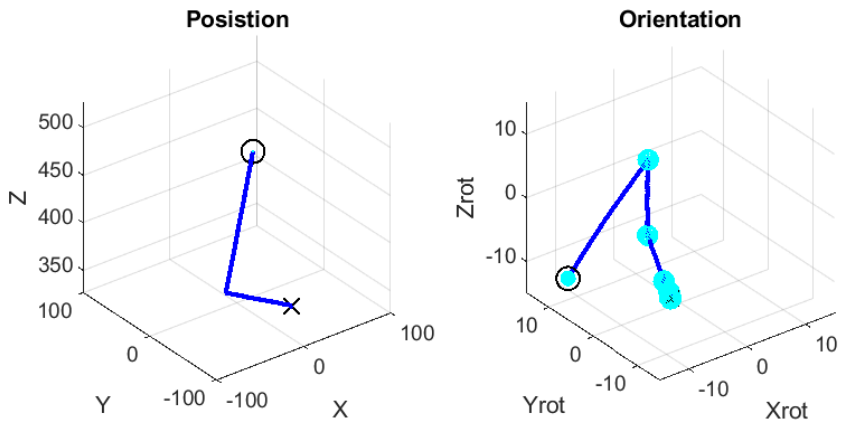

Figure 4.60: Case #2 Using Volumetric Ellipsoid Function

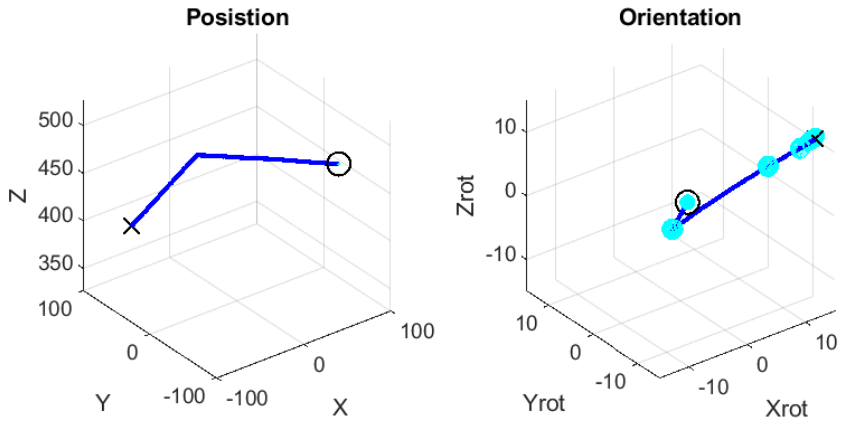

Figure 4.61: Case #3 Using Volumetric Ellipsoid Function

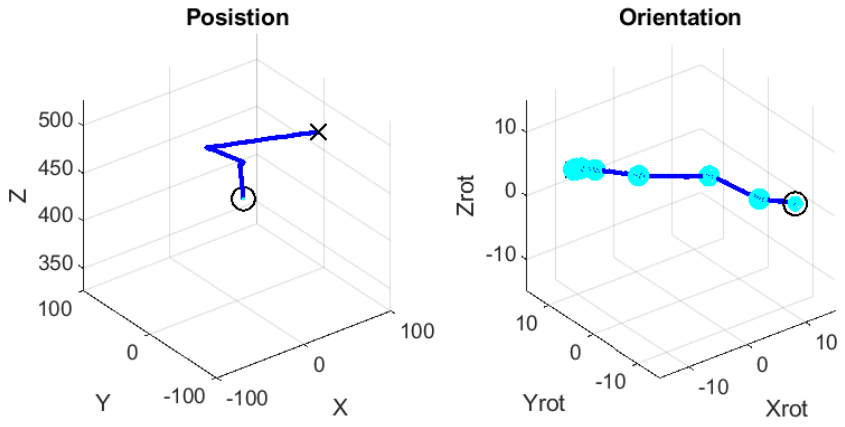

Figure 4.62: Case #4 Using Volumetric Ellipsoid Function

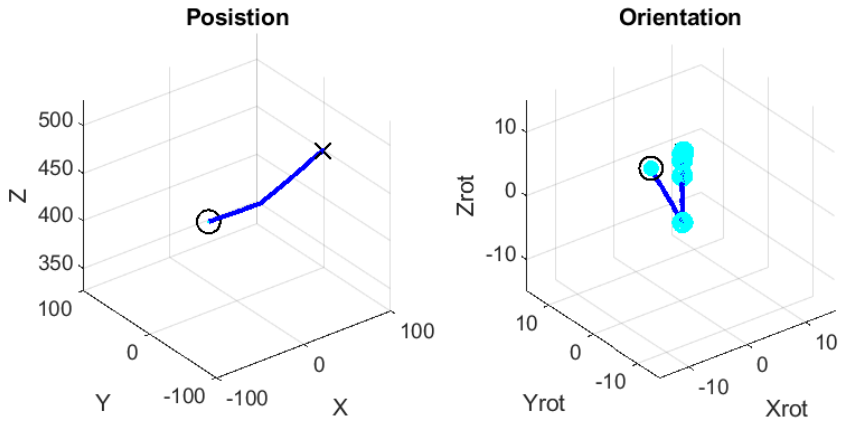

Figure 4.63: Case #5 Using Volumetric Ellipsoid Function

Within figures 4.64 to 4.68 are the best paths taken for the hand picked states. Case 8, shown in figure 4.66, in particular shows a very nice straight line path within the position space and a fairly straight path in the orientation space, showing a quite simple path to the goal. Interestingly for case 10, in figure 4.68, when the Stewart platform has no rotation component necessary for movement, it still has some movement in the rotation space due to the error provided by the steering function.

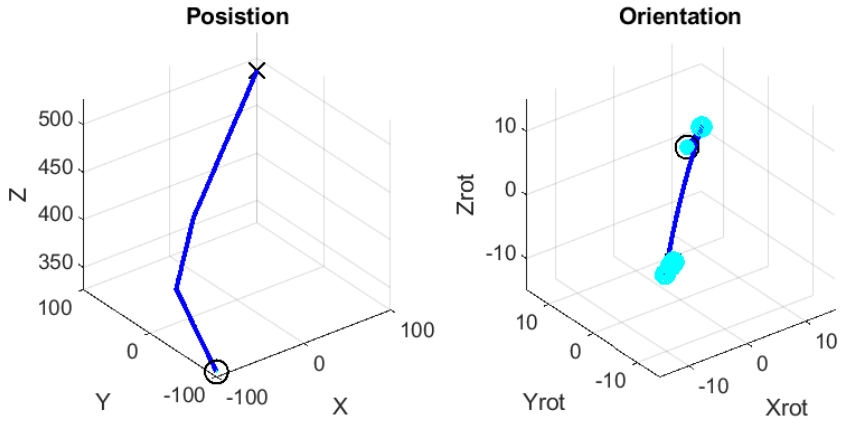

Figure 4.64: Case #6 Using Volumetric Ellipsoid Function

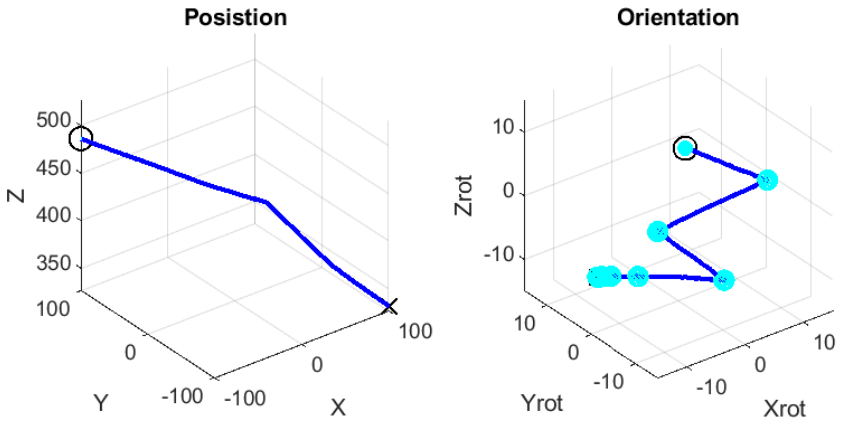

Figure 4.65: Case #7 Using Volumetric Ellipsoid Function

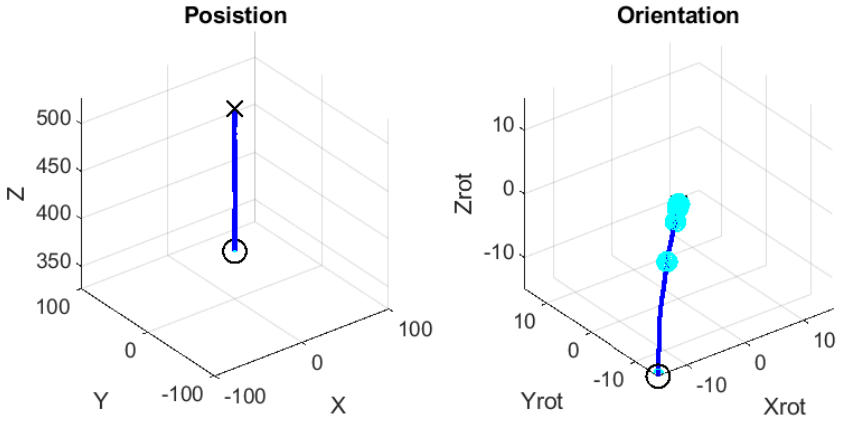

Figure 4.66: Case #8 Using Volumetric Ellipsoid Function

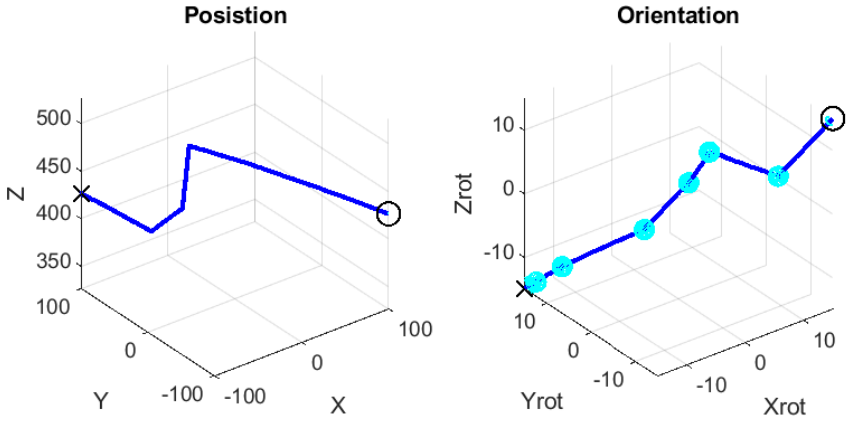

Figure 4.67: Case #9 Using Volumetric Ellipsoid Function

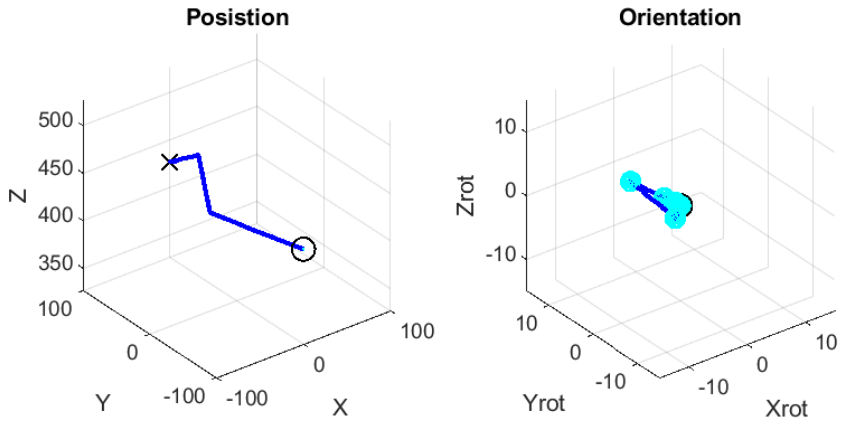

Figure 4.68: Case #10 Using Volumetric Ellipsoid Function

4.5.1.1 Position Only Volumetric Ellipsoid Function Testing

Within this subsection, a variation of the calculation of error when using the Volumetric Ellipsoid function is tested. This variation uses only the component of the error covariance matrix, **P**, that deals with error in position (the top left 3x3 matrix). This is done to test how only looking into position error affects the path calculated by the motion planning algorithm. Figure 4.69 shows the summary results of this test, both the number of trials taken and the range of error values calculated. This summary shows the range of errors experienced as well as the tracking of each trial individually.

Figure 4.70 shows the best case path created and figure 4.71 shows the worst path created, based on the error in the system. In the best case scenario, the path is still not a very smooth point from start to goal but still does not double back on itself at least, unlike in the worst case scenario. In the worst case scenario, the path doubled back on itself prior to heading to the goal position, even doing this

within the orientation space. Using the position only values for calculating error did address the creation of such small error at the goal as shown in table 4.10. This shows that with using the Volumetric Ellipsoid function, where the ellipsoid is projected into a subspace only about position, the error accumulation comes to a more meaningful amount.

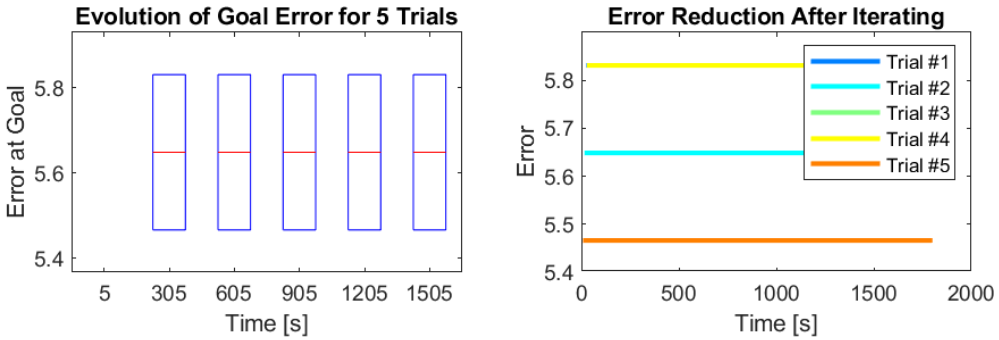

Figure 4.69: Position Only Error Tracking Summary

Trial	Initial Error	Error Addition	Ending Error
1	4.188	1.643	5.831
2	4.188	1.461	5.649
3	4.188	1.278	5.466
4	4.188	1.643	5.831
5	4.188	1.278	5.466
Total	4.188	1.461	5.649

Table 4.10: Average Error Addition and Ending Error

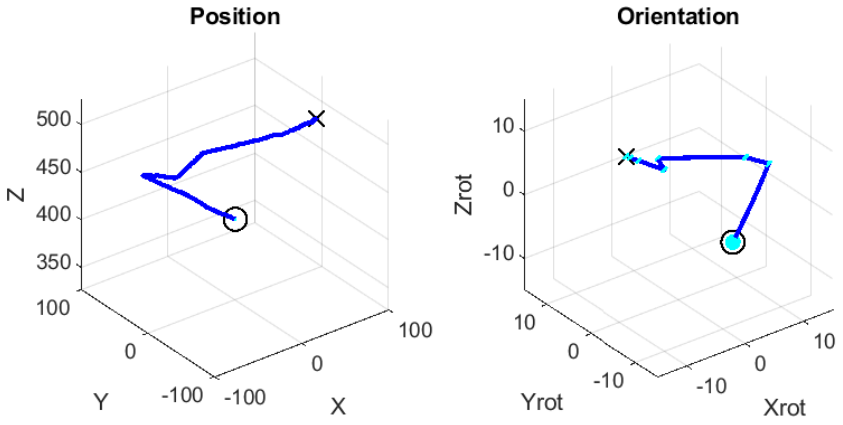

Figure 4.70: Trial #5, Lowest Error Trial

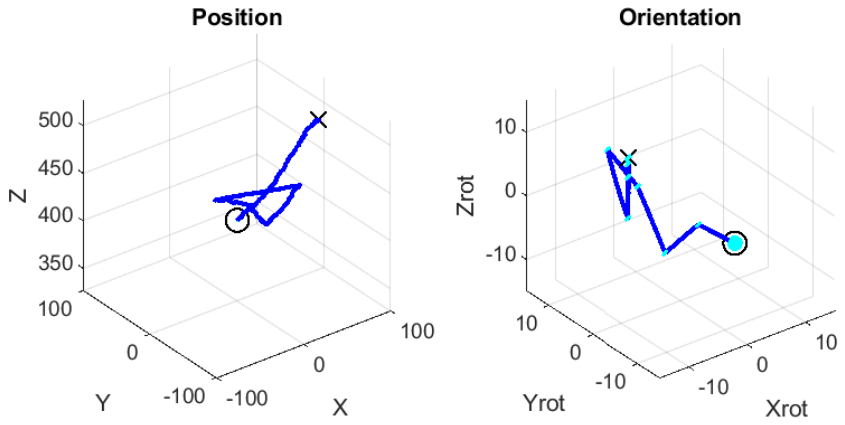

Figure 4.71: Trial #4, Highest Error Trial

4.5.2 Testing with Trace Function

Figure 4.72 shows the range and average accumulated error within the Trace function tests. Compared to the Volumetric Ellipsoid results, it can be seen that case 6 produced some of the highest error whereas this fell within the average in the Volumetric Ellipsoid function tests. Interestingly, case 9 caused approximately the same highest error but also had the highest range of all the tests, this case was also a heavy outlier in the Volumetric Ellipsoid function tests.

Figures 4.73-4.82 represent the best trials for all ten cases when using the Trace function. Within this, figures 4.73 to 4.82 show how the trace function calculates a different path to the goal for the random cases when compared to the Volumetric Ellipsoid function. Again, these cases show sharp turns within the orientation space and occasionally within the position space. Overall, this shows some of the characteristics about the motion of the Stewart platform, specifically that it can immediately switch from going in one rotation direction to the exact opposite rotation

direction. This allows the motion planning algorithm to plan for sharp movements around possible obstacles or around singularities within the workspace.

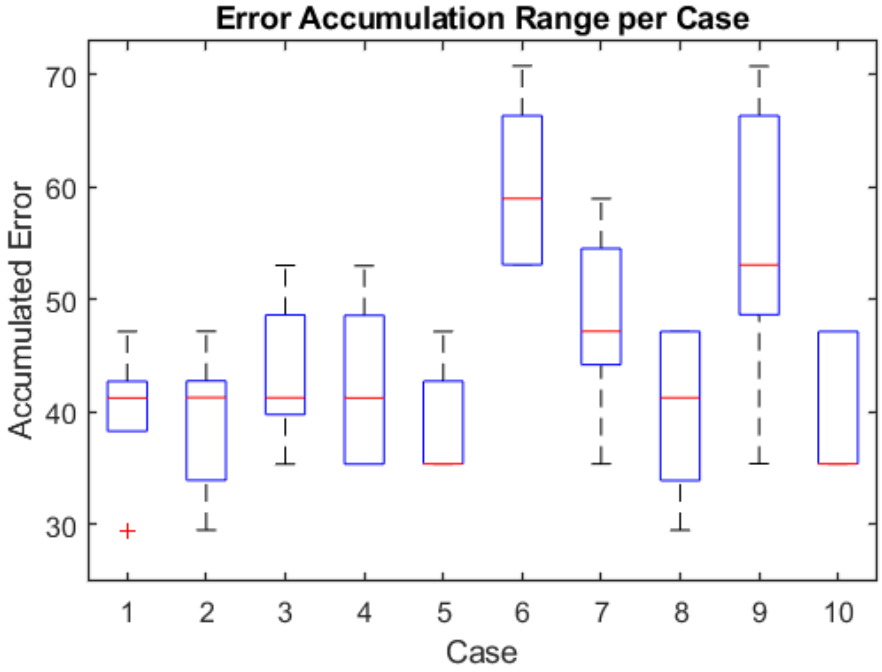

Figure 4.72: Ending Error of Additional Tests

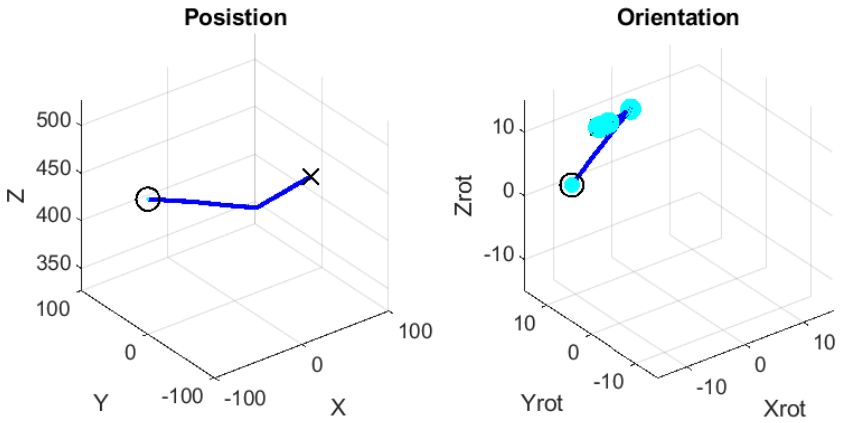

Figure 4.73: Case #1 Using Trace Function

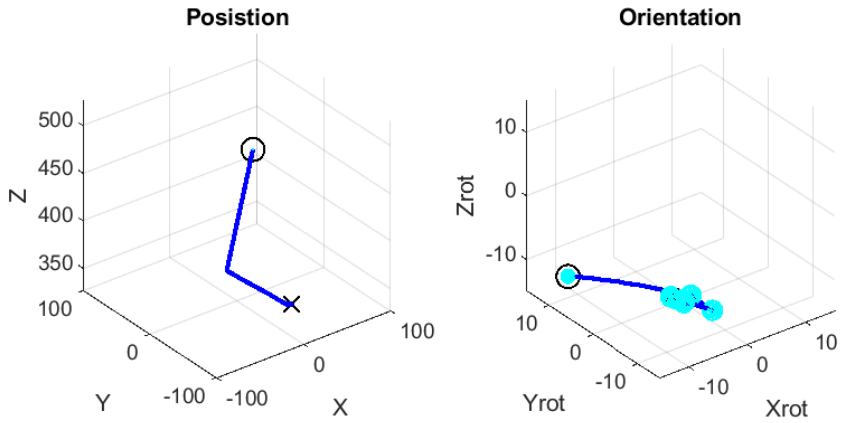

Figure 4.74: Case #2 Using Trace Function

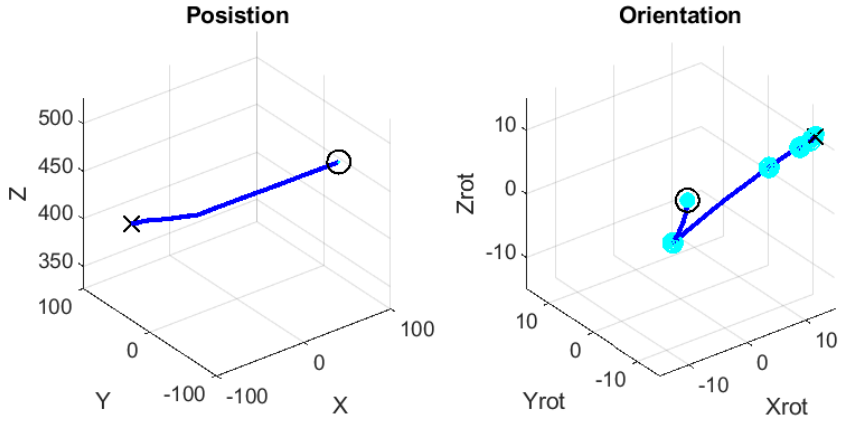

Figure 4.75: Case #3 Using Trace Function

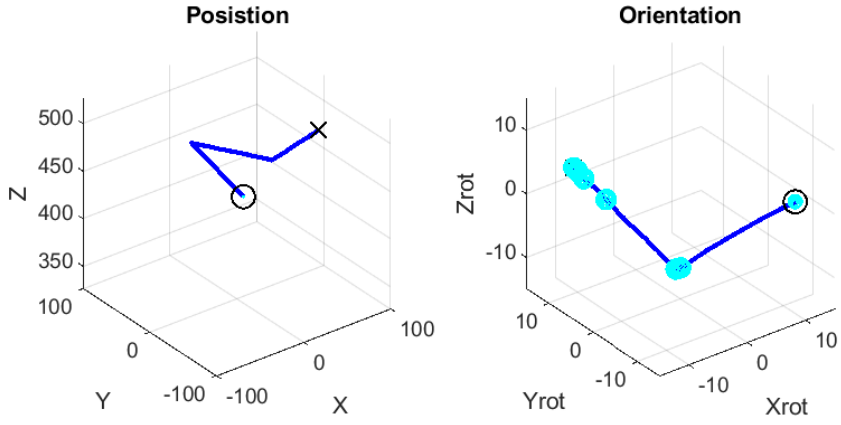

Figure 4.76: Case #4 Using Trace Function

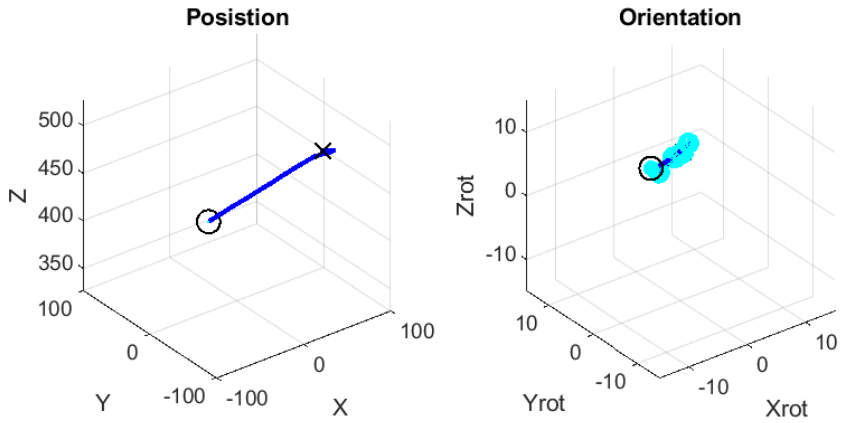

Figure 4.77: Case #5 Using Trace Function

Comparing the results of the hand picked positions with the same results in the Volumetric Ellipsoid function, there are some key differences. For case 6, as seen in figure 4.78, the position path is far more abrupt in changes than that of the Volumetric Ellipsoid function whereas the orientation path is linear. This could be due to the error in the system being more in the position side rather than in the orientation side when this path was calculated. Another key difference is in the orientation space of case 7. The Volumetric Ellipsoid function shows a far greater range of changes occurring along the path than the Trace function case in figure 4.79. Another key difference is in case 9, shown in figure 4.81, where the position and orientation paths are smoother and follow what may be considered the most optimal length path between the start and goal states. Within the Volumetric Ellipsoid function, the paths are quite a bit more jagged with a greater range of change occurring in both the position and orientation spaces. Lastly, case 10 shows that in both the Volumetric Ellipsoid and Trace function trials, there is enough error to cause movement within the orientation space even though there was to be

no movement within that space.

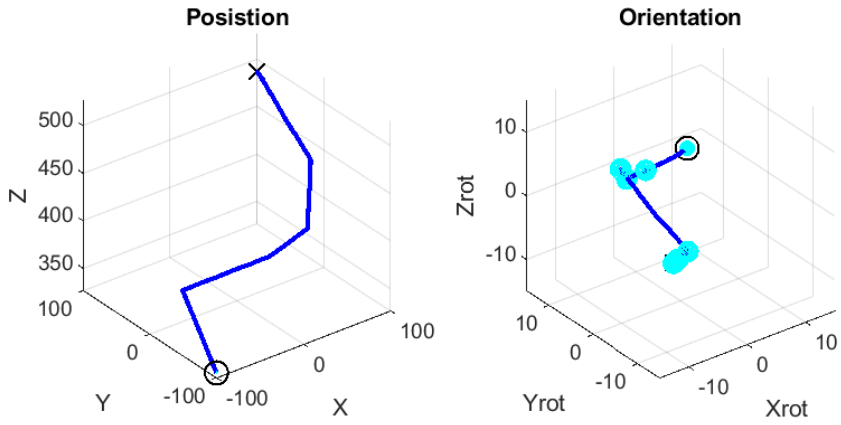

Figure 4.78: Case #6 Using Trace Function

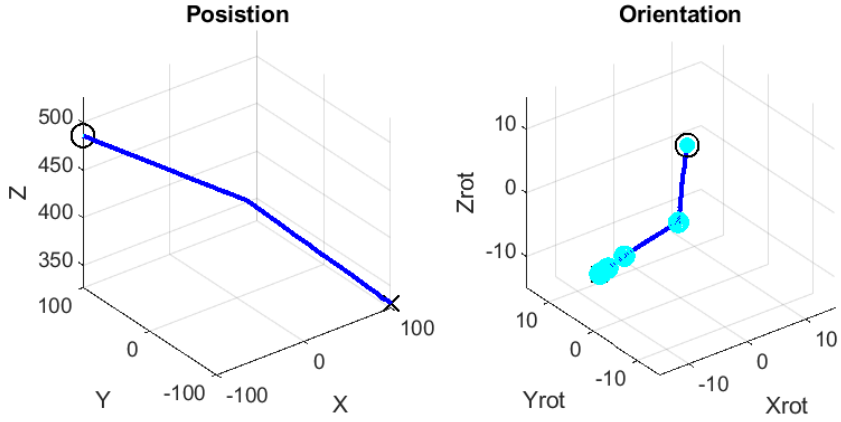

Figure 4.79: Case #7 Using Trace Function

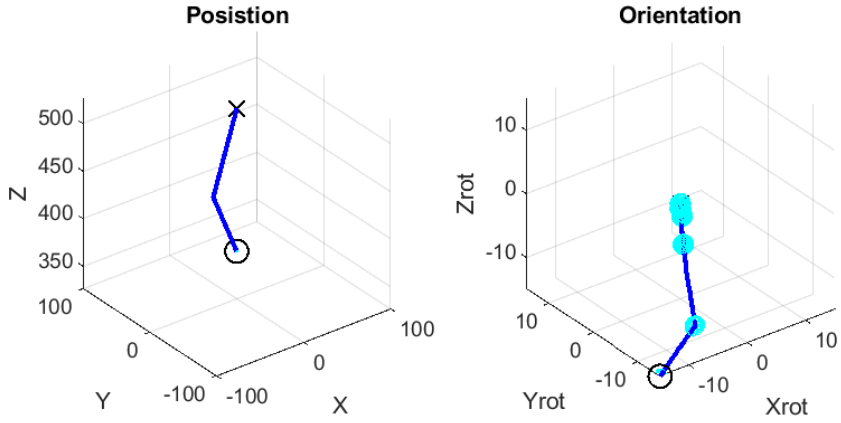

Figure 4.80: Case #8 Using Trace Function

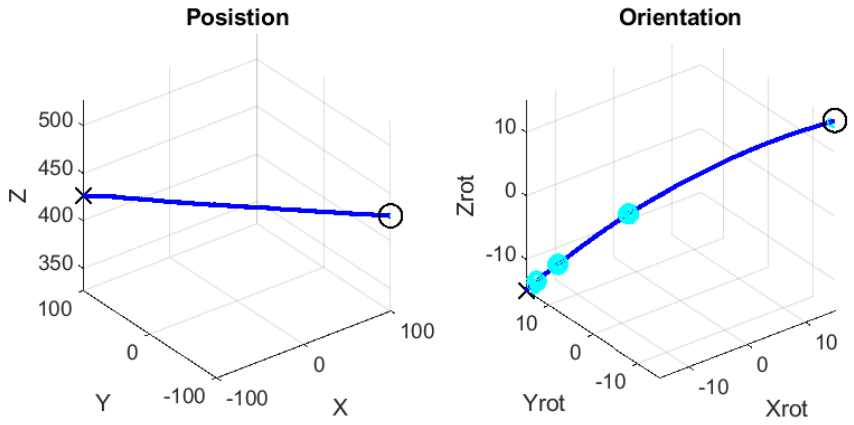

Figure 4.81: Case #9 Using Trace Function

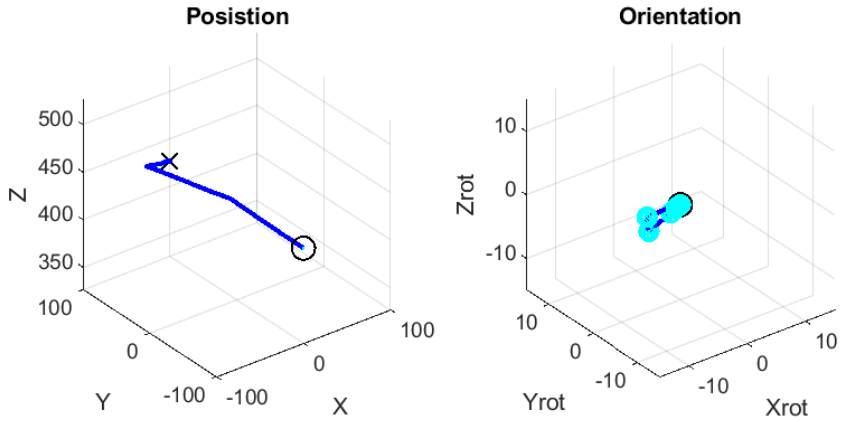

Figure 4.82: Case #10 Using Trace Function

4.6 Conclusions and Summary

4.6.1 Conclusions

In conclusion, the following takeaways should be noted: First, RRT# performs better than RRT* in most cases when given a long enough time to run. RRT* performs better if a solution is needed quickly as it is more likely to give a solution in a short amount of time. Second, the Trace function worked as a better metric in high dimensions for the cost function of the motion planning algorithms compared

to the Volumetric Ellipsoid function. It was comparable to the Volumetric Ellipsoid function in lower dimensions as it produced similar results, so it may be a better overall metric function. Third, when using the steering function for the Stewart platform, a high **R** to **Q** ratio gives some of the best results for error analysis. Combining this with a well-known starting position, e.g. a low initial **P**, gives the best chance of a low error solution. Lastly, computationally MATLAB is limited in size to the arrays that it can handle and therefore limits the capability of testing in some cases to shorter run-times. It should be avoided when longer run times and larger data handling capabilities are necessary.

4.6.2 Summary

The work described in this dissertation provides a unique approach to error based motion planning for a Stewart platform as well as a 2D Toy problem. Using RRT# motion planning algorithm combined with a Trace function for error tracking was shown to work best as a solution for tracking and planning for low cumulative error along the path found by the motion planning algorithm. and can be extended to a more general case of any well-defined robotic platform. It was also shown that RRT*, while not as efficient as RRT#, can be used if necessary for this approach.

Appendix A: Additional Equations

Within this section are all of the additional non-essential auxiliary equations used within the algorithms and subroutines. Equation A.1 is the normalize function used in algorithm 8 and 9.

$$\vec{x}_{norm} = \frac{\vec{x}}{||\vec{x}||} \tag{A.1}$$

Equation A.2 is the Rodrigues Rotation formula used during the Stewart platform kinematics. This formula was used to calculate a rotation vector $[\vec{\omega}]\theta$ where $[\vec{\omega}]$ is the skew-symmetric matrix of $\vec{\omega}$. This rotation vector represented the axis of rotation from one point to another and the amount of rotation in each of the axes.

$$e^{[\vec{\omega}]\theta} = \mathbf{R} = \mathbf{I} + \sin\theta[\vec{\omega}] + (1 - \cos\theta)[\vec{\omega}]^2 \tag{A.2}$$

Bibliography

[1] William R Doggett, John Dorsey, John Teter, David Paddock, Thomas Jones, Erik E Komendera, Lynn Bowman, Chuck Taylor, and Martin Mikulas. Persistent assets in zero-g and on planetary surfaces: Enabled by modular technology and robotic operations. In *2018 AIAA SPACE and Astronautics Forum and Exposition*, page 5305, 2018.

[2] John H Reif. Complexity of the mover's problem and generalizations. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 421–427. IEEE, 1979.

[3] S. M. LaValle. Motion planning. *IEEE Robotics Automation Magazine*, 18(1):79–89, 2011.

[4] Doug Stewart. A platform with six degrees of freedom. *Proceedings of the institution of mechanical engineers*, 180(1):371–386, 1965.

[5] Filip Szufnarowski. Stewart platform with fixed rotary actuators: a low cost design study. *Advances in medical Robotics*, 2013.

[6] Erik E Komendera and John Dorsey. Initial validation of robotic operations for in-space assembly of a large solar electric propulsion transport vehicle. In *AIAA SPACE and Astronautics Forum and Exposition*, page 5248, 2017.

[7] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[8] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[9] Oktay Arslan and Panagiotis Tsiotras. Use of relaxation methods in sampling-based algorithms for optimal motion planning. In *2013 IEEE International Conference on Robotics and Automation*, pages 2421–2428. IEEE, 2013.

[10] Bhaskar Dasgupta and TS Mruthyunjaya. Singularity-free path planning for the stewart platform manipulator. *Mechanism and Machine Theory*, 33(6):711–725, 1998.

[11] CC Nguyen, SS Antrazi, Z-L Zhou, and CE Campbell. Experimental study of motion control and trajectory planning for a stewart platform robot manipulator. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 1873–1878. IEEE, 1991.

[12] C. Mavroidis. Completely Specified Displacements of a Rigid Body and Their Application in the Direct Kinematics of In-Parallel Mechanisms. *Journal of Mechanical Design*, 121(4):485–491, 12 1999.

[13] Carlo Innocenti. Forward kinematics in polynomial form of the general stewart platform. *J. Mech. Des.*, 123(2):254–260, 2001.

[14] Domagoj Jakobović and Leo Budin. Forward kinematics of a stewart platform mechanism. *Faculty of Electrical Engineering and Computing Faculty of Electrical Engineering and Computing Unska*, 3:10000, 2002.

[15] Matthew B Rhudy, Roger A Salguero, and Keaton Holappa. A kalman filtering tutorial for undergraduate students. *International Journal of Computer Science & Engineering Survey*, 8(1):1–9, 2017.

[16] Greg Welch, Gary Bishop, et al. An introduction to the kalman filter, 1995.

[17] Ian Reid and Hilary Term. Estimation ii. *University of Oxford, Lecture Notes*, 2001.

[18] Rui Zeng, Yongjia Zhao, and Shulin Dai. A novel direct kinematic method based on extended kalman filter for stewart platform. *International Journal of Control and Automation*, 8(9):331–344, 2015.

[19] JA Wilson. Volume of n-dimensional ellipsoid. *Sciencia Acta Xaveriana*, 1(1):101–6, 2010.

[20] Actuonix. P16-p linear actuator with feedback. `https://www.actuonix.com/P16-P-Linear-Actuator-p/p16-p.htm`, 2019. Accessed on 2019-03-28.