BEN-GURION UNIVERSITY OF THE NEGEV

THE FACULTY OF ENGINEERING SCIENCE

DEPARTMENT OF MECHANICAL ENGINEERING

A Novel Simple Two-Robot Precise Self-Localization Method

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE M.Sc. DEGREE

By: Dana Erez

September 2019

BEN-GURION UNIVERSITY OF THE NEGEV

THE FACULTY OF ENGINEERING SCIENCE

DEPARTMENT OF MECHANICAL ENGINEERING

A Novel Simple Two-Robot Precise Self-Localization Method

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE M.Sc. DEGREE

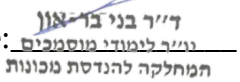
By: Dana Erez

Supervised by: Dr. David Zarrouk and Dr. Shai Arogeti

Author: Dana Erez  Date: 22/12/2019

Supervisor: David Zarrouk  Date: 22/12/2019

Supervisor: Shai Arogrti  Date: 22/12/2019

Chairman of graduate studies committee:  Date: 23/12/19

September 2019

Abstract

This Thesis presents a novel two-robot collaboration method for precise 2D self-localization using relatively simple sensors. The main advantage of this method lies in its ability to precisely measure the orientations of the robots, therefore reducing cumulative errors. Each robot is fitted with a rotating turret carrying a camera to track the moving robot and calculate the relative distance and position, and an encoder to provide the orientation of the turret. At each step, a single robot advances while the other remains stationary and measures the position of the moving robot (continuously or at the end of the step), using the angular orientation of the turret and the distance measured using the camera. The orientation of the moving robot is obtained by turning its own turret towards the static robot and measuring its turret orientation. By fusing the data from the two robots, the precise location and orientation of the moving robot are obtained.

We also present an analytical model of the position of the robots as a function of the sensor data and then proceed to present a statistical estimate using Monte Carlo simulations of the location of the robots while assuming that the sensor data includes random errors. Additionally, real-world experiments are presented and compared to simulation results.

For the two-robot system to advance autonomously, a path planning algorithm and a closed-loop controller are presented in this Thesis, given the measurements are the distance and orientation of the moving robot with respect to the stationary robot and the control inputs are the linear and angular velocities of the moving robot. The path planning algorithm include choosing a target point for the moving robot each step and finding the optimal path while avoiding obstacles in the environment such as walls, objects or the stationary robot. The closed-loop control method assumes target points in the explored environment and trajectories between them were chosen, meaning each step is a path following problem. Due to the polar characteristics of the measurements, the controller is designed in polar coordinates.

This research was recently accepted for publication in IEEE Access, the Multidisciplinary Open Access Journal of the Institute of Electrical and Electronics Engineers. Additionally, we presented this research in the 35th Israeli Conference on Mechanical Engineering (ICME 2018) and in the 6th Israeli Conference on Robotics (ICR 2019).

Key Words – **Localization, Multi-robot Systems, SLAM, Cooperative Localization, Monte Carlo Simulation, Path Planning, Path Following, Control under Constraints**.

Acknowledgments

First, I would like to express my gratitude to my advisors Dr. David Zarrouk and Dr. Shai Arogeti for guidance and encouragement during my research and thesis writing. I would like to thank them for all the support, patient advice and the trust they placed in me. I would also like to thank Prof. Per-Olof Gutman for his help and guidance during the last semester.

A special thanks goes to my friends and colleagues, Lee-Hee Drory, Nir Meiri, Dan Shachaf, Dana Shimoni and Benny Kosarnovsky for the constant support and advice on a daily basis. I would also like to thank my friends and fellow masters' students for their support, the shared lunches and laughter during our time in the university and outside of it.

Finally, I would like to thank my partner Ophir Weinreb and my family for loving and supporting me and for providing me with continuous encouragement throughout my years of study and throughout the research process and the thesis writing. A special thanks goes to my mother, Francine Ex, for devoting many hours proofreading my papers the last year.

This research was supported in part by the Pearlstone Center for Aeronautical Studies and by the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Initiative and by the Marcus Endowment Fund all at Ben-Gurion University of the Negev.

Table of Contents

Nomenclature

Symbol	Units/Dimensions	Description
A_i^j	3×3	Transformation matrix from *i*-coordinate system to *j*-coordinate system
$A, \tilde{A}$	2×2	State matrix
a, b	m, m	Length of semi axes of ellipse obstacle
B	2×1	Control matrix
b_x	-	*x* coordinate of the center of the ball in pixels, with respect to the center of the frame
(C_x,C_y)	m, m	Center of ellipse obstacle
D_x, D_y	m, m	Distance between coordinate systems in *x* and *y* directions respectively
d	m	Length of desired path
d_c	m	Current distance between the robot's current and initial positions.
$[E]_i$	3×1	Location error at step *i*
$\bar{e}$	2×1	Error state vector
e_r , e_φ	m, deg	Robot’s position and heading errors respectively
F	-	General operator
f	-	Measured location function
g	-	General function
$H(s)$	-	Transfer function
i	-	Current step number
J	-	Cost function
$[J]_i$	3×3*i*	Jacobian matrix at step *i*
K	1×2	Gain matrix
k_1, k_2	-	Control gains
L	m	Length of turret
l	m	Length of robot
M_p	%	Maximum (present) overshoot of dynamic response
N	-	Number of random error values
N_L	-	Size of the turret in image in pixels
N_x, N_y	-	Number of pixels in image in the horizontal and vertical directions respectively
n	-	Number of steps

P, Q	2×2	Positive definite matrices
p, q	m, m	Dimensions of rectangular obstacle
P_e	-	Traveling vehicle's target point
P_r	-	Stationary vehicle's position
P_0	-	Traveling vehicle's initial position
$\mathbb{R}^n$	-	Real coordinate space of n dimensions
(R_x,R_y)	m, m	Center of rectangular obstacle
$r_d = f(\alpha)$	-	Robot's desired trajectory function
(r_e,α_e)	m, deg	Traveling robot's target point
r_i	m	Measured distance between robots at step i
r_{max}	m	Distance sensor's maximal range
r_0	m	Initial distance between traveling and stationary vehicles
s	-	Laplace domain's variable
t_f	sec	Final time
t_s	sec	Settling time of dynamic response
$u(t)$	-	Control signal
$V(x)$	-	Lyapunov function
$\bar{V}$	2×1	Eigenvector
v	m/sec	Vehicles' linear velocity
v_{max}	m/sec	Maximal linear velocity
$X_i=(x_i,y_i,\theta_i)$	m, m, deg	Location and orientation of the robot at step i.
x_e	-	Equilibrium point
(x_f,y_f)	m, m	Moving robot's final position
x_{lb}, x_{ub}	m, m	Upper and lower boundaries in x direction
(x^m,y^m)	m	Robot's measured location
(x^r,y^r)	m	Robot's real location
(x_s,y_s)	m, m	Stationary robot's position
(x_0,y_0)	m, m	Moving robot's initial position
y_{lb}, y_{ub}	m, m	Upper and lower boundaries in y direction
α_{ij}	deg	Measured bearing angle of vehicle j at step i
α_{image}	deg	Angular position of the tennis ball in the image
α_{turret}	deg	Orientation of the turret

α_L	deg	View angle of the height of the turret
(α, r, φ)	deg, m, deg	Robot's position and orientation in polar coordinate system
β	deg	Angle between main axis of the ellipse and the x positive axis
γ_x	deg	Camera's field of view in the horizontal direction
γ_y	deg	Camera's field of view in the vertical direction
Δ_{res}	m	Distance sensor's resolution
$[\Delta]_i$	$3i \times 1$	Error vector at step i
Δr_i	m	Distance measurement error at step i
$\Delta\alpha_{ij}$	deg	Bearing measurement error of vehicle j at step i
$\Delta\theta$	deg	Robot's Orientation error
δ	-	Allowed range around final value response
$\delta\theta$	deg	Small angle
ε	m	Minimal allowed distance between the moving vehicle and an obstacle
λ	-	Eigenvalue
μ	-	Mean value
ρ	deg	Orientation of ellipse obstacle
σ	m	Total standard deviation
σ_d	%	Distance sensor's resolution
σ_x, σ_y	m, m	Standard deviation in the 'x' and 'y' directions respectively
σ_α	deg	Bearing sensor's resolution
σ_θ	deg	Robot's orientation standard deviation
σ_1, σ_2	m, m	Standard deviation in the direction of the semi-major and semi-minor axes of the ellipse respectively
σ^2	-	Variance
φ	deg	Robot's heading angle
φ_d	deg	Robot's desired heading
χ	deg	Angle between initial position and desired path
ψ_i	deg	Relative angle between coordinate systems at step i
ω	rad/sec	Vehicle's angular velocity
ω_{max}	rad/sec	Maximal angular velocity
ω_n	rad/sec	Natural frequency
ζ	-	Damping coefficient

List of Figures

List of Tables

1 Introduction

Many robotic applications such as search and rescue, surveillance, planetary exploration and others require Simultaneous Localization and Mapping (SLAM) of unknown unstructured locations. SLAM is known as a 'chicken and egg problem' meaning, how can a robot build or update a map of an unknown environment, while simultaneously keeping track of its own location within it?

This problem is well-researched due to its great potential in solving a wide range of robotic applications. Nowadays, as technology advances, Global Positioning System (GPS) is the simplest and most accurate localization technique. SLAM techniques become more crucial where GPS and other localization techniques are unavailable such as indoors, inside caves or in tunnels.

Many solutions for self-localization rely on measuring the relative position of the robot with respect to known features in space, also known as landmarks. However, the complexity grows in cases where there is no prior knowledge of the explored area. In 1994, Kurazume et al. first suggested *cooperative positioning* for multi-robot systems as a solution to the SLAM problem [1]. By advancing the robots in alternating steps, such that at each point in time some robots remain stationary and the others travel to new positions, the stationary robots whose absolute locations are known serve as landmarks for the traveling robots. Therefore, this method is especially useful while exploring an uncharted environment where there are no known landmarks.

The cooperative positioning method has been further developed by other groups [2]-[6] to suggest the use of different kinds of sensors to determine relative positioning with different advancing algorithms. The advantage of this method is that a unified map of the robots' trajectories is created using all available relative measurements. However, to implement this method, a centralized communication system is required. Centralized approaches, though theoretically effective, require ideal communication and high computational cost, thus making them vulnerable to single-point failures especially as the number of robots increases.

In 1997, A different method for multi-robot SLAM was suggested by Roy and Dudek known as the *rendezvous case* [7], further developed in [8]. In this method, each robot explores a different part of the environment and creates its own map. When two robots meet (this could be a random event or could be arranged by the two robots [8]), the robots measure their relative distance and bearings; this information can be used to compute the coordinate transformation

required to merge both maps. Due to noise in these measurements, the estimated transformation may be inaccurate, reducing the quality of the merged map. If landmarks are available in the explored environment, the most probable transformation between two maps can be identified as the one that produces the maximum number of landmark correspondences [9]. Available landmarks also allow successful localization of a multi robot system even when the initial positions of the robots are unknown [9],[10]. Other solutions for map merging have been offered such as using particle filters [10] or occupancy grid maps [11].

Though map merging increases complexity, this method has an obvious advantage, especially while exploring large areas, of enhancing efficiency, i.e. reducing exploration time. The exploration time could be further reduced by wisely choosing different paths for different robots. While in most methods the robots are guided to points in the explored environment which have minimum travel cost out of all unexplored points, [12] suggests an approach that takes into account not only the travel cost but also the utility of unexplored points, where the utility of a target location depends on the probability that this location is visible from a target location assigned to another robot.

The main challenge in using relative measurements is determining the absolute locations of the robots, since the locations are obtained with regard to a local coordinate system. Some solutions address this issue by combining both external measurements such as GPS [13] or an affixed IR range detector [14], which return inaccurate yet absolute locations and relative measurements between the robots to enhance accuracy and obtain the orientation of the robots as well. The practicality of these methods is limited since they require either GPS reception which is not available in many cases such as indoor or underground areas or prior placing of sensing tools. Similarly, many solutions use filtering techniques, most commonly the Extended Kalman Filter (EKF) [15]-[18], where the robots' locations are predicted by odometry data (such as linear and angular velocities) and corrected by relative measurements between neighboring robots. Recently, the use of Ultra-Wideband (UWB) range-sensors has become popular for relative distance measurements in multi-robot systems, because they make it possible to perform the localization process in a fully decentralized manner [19]-[21].

While the relative locations of a multi-robot system can be calculated by using any of the aforementioned methods, obtaining the accurate relative orientation of the robots is much more challenging. Besides visual methods [22]-[25], many attempts to find the orientation of the robots have been made using range-only measurements [6],[19]-[21],[26] and angle-only measurements [17] or a combination of both [16],[27]-[29]. The accuracy of the orientation

remains however very challenging at long distances. While some work has focused on evaluating the uncertainty of the estimated locations [30], the uncertainty in the orientation of the robots has not been appropriately examined.

Our goal in this work is to provide a simple low-cost high accuracy localization and orientation method for a multi-robot system, suitable for indoor areas where GPS signals are unavailable, and visibility is relatively low. This could be useful for underground, under water and planetary explorations or search and rescue missions in cases such as natural disasters or collapsed structures.

Figure 1.1: Examples for dark areas without GPS reception.

We would also like our system to be able to advance autonomously, i.e. plan its path and motion, in such uncharted constrained environments which lacks GPS reception. Motion planning is a fundamental problem in the field of robotics and an essential step towards complete autonomic mobile robots. Motion planning consists of computing a series of actions that drives the robot from its initial position to a desired final position, while considering its surrounding (avoiding obstacles) and its own motion limitations (kinematic/dynamic constraints or in short, differential constraints) [31]. The basic problem where the robot has no motion limitations and only an obstacle-free solution is required is a well understood problem and solutions were offered for various scenarios [32]. Since robots usually do have strict motion limitations, the previously mentioned solutions cannot actually be executed by real robots.

There are two main approaches for motion planning under differential constraints [32]: The first is a decoupling approach in which first a collision-free path is found and then the path is smoothened so that the motion constraints are fulfilled. The second is a direct approach in which the differentially constrained motion planning problem is solved all in once. While the

first approach is easier to compute, a solution is not guaranteed and even if found, may be extremely inefficient.

A direct approach on the other hand, which also includes optimizing an objective function, guaranties finding the optimal solution. Most solutions which guaranty optimization are model based methods such as the very well researched Model Predictive Control (MPC) [33]-[36] (or NMPC for nonlinear systems [37],[38]), the not as common Interpolating Control (IC) [39], and the most commonly used today sampling-based planning [32],[40]-[42], which is based on a graph search of all possible trajectories. For our work, an optimal control tool for nonlinear systems under differential constraints is used [43].

We consider two robots each of which is equipped with one camera and one rotation/bearing sensor mounted on a rotating turret. The outline of the Thesis is as follows: theoretical background is presented in Section 2, the robotic system and the localization algorithm are described in Section 3 and the error evaluation using an analytical exact method and first order approximation method is presented in Section 4. The two methods are used to statistically evaluate the location and orientation errors using Monte Carlo simulations in Section 5 and real-world experiments are described in Section 6.

Section 7 presents a path planning algorithm (Section 7.1) and a closed-loop controller for following the desired path (Section 7.2); due to the polar nature of the measurements, the controller is described in polar coordinates which was proven to be as efficient as Cartesian coordinates, and can represent non-linear distributions in the Cartesian space as linear distributions in the polar space [44]-[48]. Section 7.3 presents the fusion and implementation of both algorithms. Finally, conclusions are discussed in Section 8.

2 Theoretical Background

This section presents relevant theoretical background for this paper.

2.1 Transformation Matrix

One of the challenges when using a multi-robot system to solve a SLAM problem, is that the measurements from each robot are obtained in its local coordinate system. One of the requirements for executing a SLAM algorithm is that all information is obtained in a fixed global coordinate system. To do so, transformation matrices are used.

The main idea of this method is based on the fact that every vector in one coordinate system can be represented in a second coordinate system, by multiplying the vector with the desired coordinate system's basis vectors. In the two-dimensional case, this is a linear operation of the following form:

$$R_0^1 = \begin{bmatrix} X_0^1 & Y_0^1 \end{bmatrix} = \begin{bmatrix} x_1 \cdot x_0 & x_1 \cdot y_0 \\ y_1 \cdot x_0 & y_1 \cdot y_0 \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}, \qquad (1)$$

while R_0^1 is the *rotation matrix*, converting the vector $\begin{pmatrix} x_1 & y_1 \end{pmatrix}^T$ to the 0-coordinate system, and α is the rotation angle between the two coordinate systems.

In our case, the goal in each step is to covert the robot's position to the global coordinate system. Therefore, the difference between the robot's local coordinate systems and the global one, is not only in orientation but also the distance traveled, say D_x in the x axis direction and D_y in the y axis direction. The overall *transformation matrix* is:

$$A_0^1 = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & D_x \\ \sin(\alpha) & \cos(\alpha) & D_y \\ 0 & 0 & 1 \end{bmatrix}. \qquad (2)$$

This method is the basis of the localization method presented in this paper (3.4 and 3.5). For each step, the moving robot's position is converted to the global coordinate system; the n step's transformation matrix is received as following:

$$A_0^n = A_0^{n-1} \cdot A_{n-1}^n, \qquad (3)$$

while A_{n-1}^n is the last step's transformation matrix from the moving robot's coordinate system (n) to the stationary robot's coordinate system (n-1), and A_0^{n-1} is the overall transformation matrix

calculated at the previous step, converting from (*n-1*) coordinate system to the global coordinate system (0).

2.2 Jacobian Matrix

Suppose $F:\mathbb{R}^n \to \mathbb{R}^m$ is an operator which receives as an input a vector $q \in \mathbb{R}^n$ and produces as an output the vector $F(q) \in \mathbb{R}^m$. Then the Jacobian of *F* is a *m* by *n* matrix that is defined as follows [49]:

$$J = \begin{bmatrix} \frac{\partial F}{\partial q_1} & \cdots & \frac{\partial F}{\partial q_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \cdots & \frac{\partial f_1}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial q_1} & \cdots & \frac{\partial f_m}{\partial q_n} \end{bmatrix}, \tag{4}$$

while $f_1 \ldots f_m$ are the functions that define operator *F* and $q_1 \ldots q_n$ are the variables of these functions. In our case, the functions f determine the robot's location and orientation in space in the global coordinate system (x, y, θ), and the variables are the robots' measurements (see Section 3.4).

If *F* is differentiable at point *q*, then the Jacobian matrix defines a linear map $\mathbb{R}^n \to \mathbb{R}^m$ which is the best linear approximation of function *F* near point *q*; meaning, the Jacobian matrix can be used to approximate the value of function *F* at point *q*, without actually calculating $F(q)$. An explanation of this process in our system is presented in Section 4.2.

2.3 Probability Theory

Probability is a very broad branch in mathematics. In this section, specific concepts, regarding this paper, within probability theory will be explained.

Probability distribution is a mathematical function that provides the probability of occurrence of different possible outcomes of a certain experiment [50]. The distribution of numerical data can be accurately represented by a *histogram*. To build a histogram, the entire range of values must be divided into 'bins' i.e., a series of intervals, and then count how many values fall into each bin.

If the Probability distribution function is continuous, by sampling experiment results for many random values, a *probability density function* (*PDF*) can be obtained. These functions have two main characteristics. The first is *mean value* (also known as expected value or average), represented in Figure 2.1 as μ. The mean value is simply calculated by summing all values and dividing the sum by the number of values.

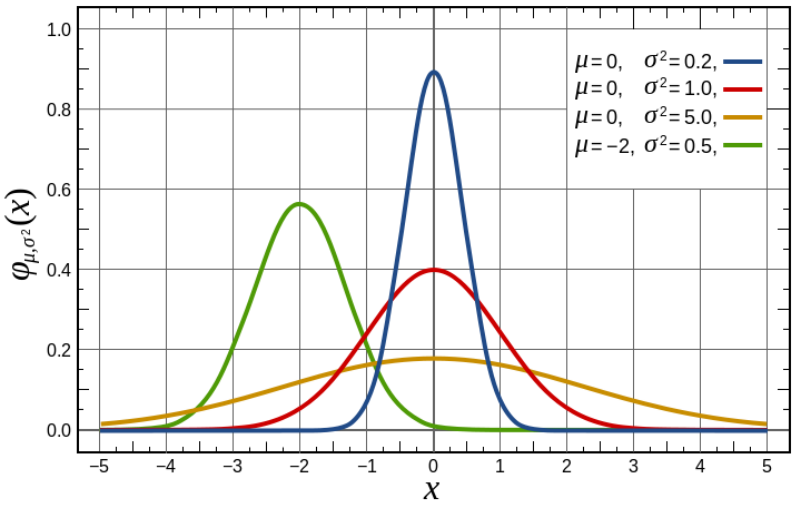


Figure 2.1: Probability density function. The red curve is a normal distribution[1].

The second characteristic is *variance*, represented in Figure 2.1 as σ^2. Variance is the expectation of the squared deviation of a random variable from its mean. The square root of the variance is called *standard deviation* - σ. Standard deviation indicates how far a set of random values are spread out from the mean value (amount of dispersion). Standard deviation is calculated as following:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}\ , \qquad (5)$$

as x_i is the measured value in all data points and N are the number of data points.

Normal distribution (also known as Gaussian distribution) is a very common continuous probability distribution. Normal distributions have a mean value of 0 and a standard deviation of 1. This distribution is often used to represent real-valued random variables whose distributions are not known.

[1] https://en.wikipedia.org/wiki/Normal_distribution

2.4 State-Space Representation

State-space representation is a mathematical method, used in the field of control-engineering, representing physical systems as a set of inputs, outputs and 'state variables', related by a first order matrix differential equation. In the case of linear, time-invariant and finite-dimensional dynamical systems (*LTI systems)*, the system can be represented by a 'state equation' and a 'measurement equation' respectively:

$$\begin{aligned} \dot{\bar{x}}(t) &= A\bar{x}(t) + B\bar{u}(t) \\ \bar{y}(t) &= C\bar{x}(t) + D\bar{u}(t) \end{aligned}, \tag{6}$$

with the description and dimensions of all symbols described in Table 2.1. Note that in the discussed case these matrices are constant, but they could be time dependent in the case of a continuous time-variant system. In many cases there is no direct connection between the input and output vectors, hence D=0.

Stability and natural response characteristics of a continuous LTI system are determined by the eigenvalues of matrix A.

Table 2.1: State space representation vectors and matrices.

Symbol	Dimensions	Description
$\bar{x}$	$n\times1$	State vector
$\bar{y}$	$m\times1$	Output/measurement vector
$\bar{u}$	$p\times1$	Input/control vector
$\bar{r}$	$p\times1$	Reference vector
A	$n\times n$	State/system matrix
B	$n\times p$	Input matrix
C	$m\times n$	Output matrix
D	$m\times p$	Feedforward matrix
K	$p\times n$	Gain matrix

2.4.1 Full State Feedback

Eq. (6) is also called the *open loop system*, which acts completely on the basis of the input. A *closed loop system* refers to a common control technique that relays on receiving feedback on the system's behavior by feeding the output back, i.e. closing the system, and altering the input accordingly. In state space representation, a *full state feedback* is utilized by the following input:

$$\bar{u} = \bar{r} - K\bar{x} \ , \tag{7}$$

where $\bar{r}$ is the reference vector (or the input of the closed-loop system) and K is a gain matrix of constant values (dimensions are described in Table 2.1). Placing the control law in Eq. (6) results in the following dynamic equation:

$$\dot{\bar{x}}(t) = (A - BK)\bar{x}(t) + B\bar{r} \,. \tag{8}$$

Therefore, the stability and natural response characteristics of the closed-loop system are determined by the eigenvalues of matrix $(A - BK)$.

2.5 Lyapunov Function

In the theory of ordinary differential equations (ODEs), Lyapunov functions are scalar functions that may be used to prove the stability of an equilibrium of an ODE [51]. Lyapunov's stability theory determines that if the solutions to a differential equation that start out near an equilibrium point x_e stay near x_e forever, then x_e is *Lyapunov stable* [52]. According Lyapunov's second method of stability, if a system $\dot{\bar{x}} = g(\bar{x})$ has a point of equilibrium at x_e=0, the point is Lyapunov stable if there exists a Lyapunov function $V(x): \mathbb{R}^n \to \mathbb{R}$ that fulfills the following:

- $V(x) = 0$ if and only if $x = 0$.
- $V(x) > 0$ if and only if $x \neq 0$.
- $\dot{V}(x) \leq 0$ for all $x \neq 0$ (or $\dot{V}(x) < 0$ for all $x \neq 0$ for asymptotic stability).

2.6 Second Order System's Response

A second order linear system is a common description of many dynamic processes. In the general form where $y(t)$ is the system's output and $x(t)$ is the system's input, in time domain the system is presented by the following differential equation:

$$\frac{1}{\omega^2}\ddot{y}(t) + \frac{2\zeta}{\omega}\dot{y}(t) + y(t) = x(t), \tag{9}$$

and in Laplace domain, the system is presented by the following transfer function:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{{\omega_n}^2}{s^2 + 2\zeta\omega_n s + {\omega_n}^2} \,, \tag{10}$$

where ζ is the system's damping ratio and ω_n is the system's natural frequency. The denominator of the transfer function is also called the *characteristic equation* of the system since it determines the system's response's behavior.

The system's time response to a unit step input i.e.:

$$x(t)=\begin{cases}1, & if\ t>0\\ 0, & if\ t\le 0\end{cases}, \tag{11}$$

depends on the placement of the system's poles (the characteristic equation's roots). If $0<\zeta<1$, the system is considered underdamped and the system contains a pair of complex poles:

$$s_{1,2}=-\zeta\omega_n \pm j\omega_n\sqrt{1-\zeta^2}\,, \tag{12}$$

The response of an underdamped second order system to a step unit input is presented in Figure 2.2. The transit response is characterized by the following:

- Delay time (t_d) – The time required for the response to reach half its final value
- Rise time (t_r) – The time required for the response to rise from 10% to 90% or from 0% to 100% of its final value.
- Peak time (t_p) – The time required for the response reaches its first peak.
- Maximal (present) overshoot (M_p) – The maximum peak's value with respect to the final value.
- Settling time (t_s) – The time required for the response to reach and stay within a range around the final value (usually 2% or 5%).

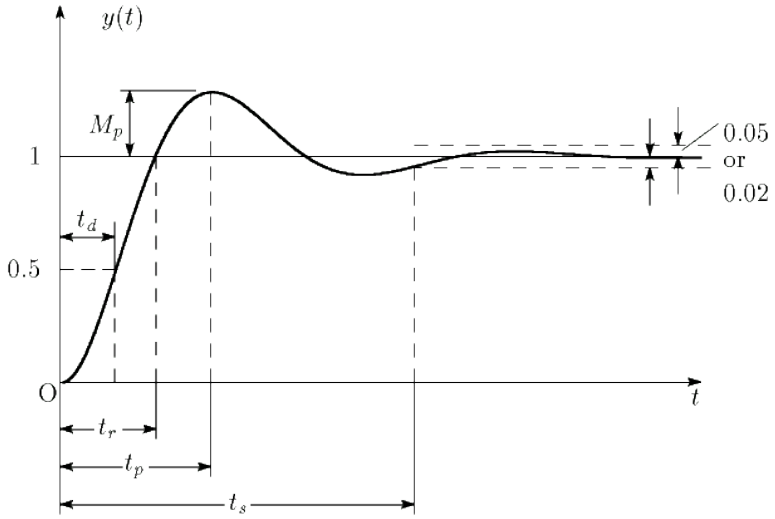


Figure 2.2: The response of an underdamped second order system to a step unit input (normalized by final value)[2].

[2] http://shiwasu.ee.ous.ac.jp/matweb_cs/help/english_sole_t_help.htm

These characteristics can be approximated by appropriate formulas. Presenting only the formulas that will be used on this paper (7.2.2), the maximal overshoot can be approximated by:

$$M_p = \exp\left(-\frac{\pi\zeta}{\sqrt{1-\zeta^2}}\right), \tag{13}$$

and the response's settling time can be approximated by:

$$t_s(\delta\%) = \frac{-\ln(\delta)}{\zeta\omega_n}, \tag{14}$$

where δ determines the allowed range around the final value.

2.7 Optimization and Cost Function

In the field of mathematics, the process of optimization is the selection of the best element, with regards to some defined criteria, from some set of available alternatives [53]. An optimization problem usually consists of maximizing or minimizing an objective real function *g* by systematically choosing input values within an allowed set and calculating the value of the function *g* [54]. When the goal is minimization, the objective function is called a loss function or a *cost function*. The cost function maps an event or values of one or more variables onto a real number, representing the “cost” associated with the event.

3 Localization Method

In this section, the assumptions and limitations of the work are defined (3.1). Then, we present our robotic setup (3.2) and two localization methods. The first method is based on a two-point measurement approach to calculate the orientation (3.3), whereas the second method, which is our newly developed method, fuses the distance and relative orientation to yield more accurate results (3.4 and 3.5).

3.1 Assumptions and Limitations

This research focuses on the problem of self-localization in areas without access to GPS signals. The solution proposed includes the use of a two-robot system advancing in alternating steps, also known as *cooperative positioning*. The research is conducted under the following assumptions and limitations:

- The research is limited to the use of a two-robot system.
- The initial position of at least one of the robots must be known.
- The system performs 2D localization, therefore the environment explored must be flat.
- At each step, one robot remains stationary, as the other robot moves in the environment. Therefore, the advance of the system is not continuous.
- The measurements are not necessarily continuous; to implement the localization method (Section 3.4 and 3.5), the relative position between the robots is measured only when the moving robot has stopped. To implement the control loop presented in Section 7.2, continues measurements are required.
- The sensors should be able to determine distance and orientation between the robots.
- The robots always stay in each other's range of 'sight' (depends on the range of the distance sensor and the camera's field of view).
- The sensors' measurement errors are assumed to be normally distributed. This is a very common assumption, broadly used in cases of implementing observing methods in order to estimate the true position of the robot, such as Kalman Filter [15]-[18].

3.2 Robotic Setup

Consider a robot fitted with a rotating turret which carries a camera (see Figure 3.1). The orientation of the turret relative to the heading of the robot is measured with a bearing sensor (such as an encoder). The camera is used to detect the target and to aim the turret towards it. The distance is measured using the camera[3] whereas the bearing sensor measures its angular coordinate. The polar coordinates can then be transformed into the real Cartesian location coordinates (x^r, y^r) using:

$$\begin{aligned} x^r &= r\cos(\alpha) \\ y^r &= r\sin(\alpha) \end{aligned}, \tag{15}$$

where r is the distance of the target and α is the orientation of the turret. Practically speaking, each of the sensor measurements contains a small error. We denote by Δr and $\Delta\alpha$, respectively the distance and orientation errors. Then the coordinates (x^m, y^m) based on the sensor measurement become:

$$\begin{aligned} x^m &= (r+\Delta r)\cos(\alpha+\Delta\alpha) \\ y^m &= (r+\Delta r)\sin(\alpha+\Delta\alpha) \end{aligned}. \tag{16}$$

The distance error range is often (according to many laser sensor catalogs and visual based sensing) proportional to the measured distance, whereas the angular error is dependent on the resolution of the camera and encoder and is constant for a long range of distances (as long as the target is detected by multiple camera pixels). Assuming small measurement errors Δr, $\Delta\alpha$ and using a first order Taylor series approximation:

$$\begin{aligned} \cos(\alpha+\Delta\alpha) &= \cos(\alpha)-\sin(\alpha)\Delta\alpha \\ \sin(\alpha+\Delta\alpha) &= \sin(\alpha)+\cos(\alpha)\Delta\alpha \end{aligned}, \tag{17}$$

neglecting the product of Δr times $\Delta\alpha$, Eq. (16) becomes:

$$\begin{aligned} x^m &\approx (r+\Delta r)\cos(\alpha)-r\sin(\alpha)\Delta\alpha \\ y^m &\approx (r+\Delta r)\sin(\alpha)+r\cos(\alpha)\Delta\alpha \end{aligned}. \tag{18}$$

[3] The distance measurement could equally be achieved by using a camera or a laser range sensor. The term 'distance measurement' or 'distance sensor' refers to either kind of measurement method.

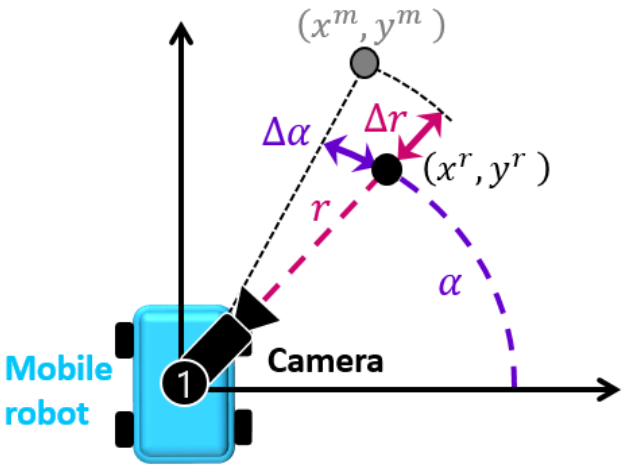


Figure 3.1: The mobile robot and its sensors. Each robot incorporates a camera fitted on a rotating turret and a bearing sensor.

3.3 Two Point Measurement Approach

A straightforward approach is to estimate the orientation of the robot by measuring the position of two specific points on its side. Assuming that the measured distance and relative orientation of two points 1 and 2 are respectively r_1, α_1, r_2 and α_2 (see Figure 3.2), the measured position of the center of the robot (x,y) and its orientation θ can be calculated as follows:

$$\begin{aligned} x &= \frac{r_1 \cos\alpha_1 + r_2 \cos\alpha_2}{2} \\ y &= \frac{r_1 \sin\alpha_1 + r_2 \sin\alpha_2}{2} \end{aligned}, \tag{19}$$

and

$$\theta = -\operatorname{atan}\left(\frac{r_2 \cos\alpha_2 - r_1 \cos\alpha_1}{r_2 \sin\alpha_2 - r_1 \sin\alpha_1}\right). \tag{20}$$

This method results in a relatively large error in the robot's orientation if the errors Δr_1,Δr_2 become significantly large relative to the distance l between the two measured points. Omitting the angle measurement errors, the maximal orientation error of the robot:

$$\Delta\theta \approx \frac{|\Delta r_1| + |\Delta r_2|}{l \cos(\alpha - \theta)}. \tag{21}$$

For example, assume a robot with a length of l=0.5 m is measured from a distance of r=10 m by a distance measurement with a resolution of 0.2%; hence Δr=2 cm. Given α=45° and θ=30°, the orientation error according to Eq. (21) is $\Delta\theta \approx$ 4.8°. Note that this orientation error for each single step is very large especially since the error is cumulative.

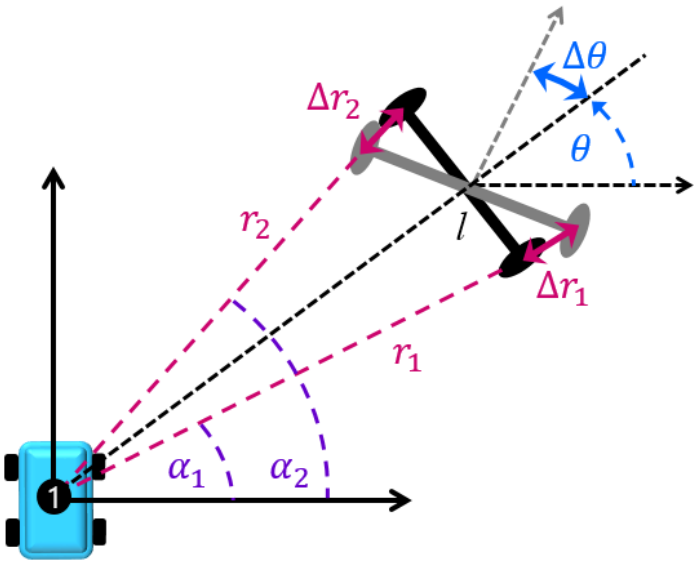


Figure 3.2: Distance measurement errors when estimating the orientation of a vehicle. In black: real position of vehicle, in grey: estimated position.

3.4 Relative Orientation Method (Suggested Method)

Our method is based on the approach of two vehicles which advance in alternating steps. At any given time, one vehicle whose position $x_{i,s}$, $y_{i,s}$ and orientation $\theta_{i,s}$ are known remains stationary, while the other vehicle advances. The index i indicates the step number and *'s'* stands for stationary. At the end of each step, the distance and bearing of the two vehicles are measured (r_i, α_{is}, α_{it}); These measurements are used to estimate the traveling vehicle's position $x_{i,t}$, $y_{i,t}$ and orientation $\theta_{i,t}$ (where *'t'* stands for traveling).

The traveling vehicle's location and orientation at each step is determined with respect to the observing vehicle's position. The general form of the Cartesian location and orientation of the traveling vehicle at step i is:

$$X_{i,t} = X_{i,s} + F\left(\theta_{i,s}, r_i, \alpha_{is}, \alpha_{it}\right), \tag{22}$$

where the vector X_i includes both the position and orientation of the vehicle: $X_i = \begin{bmatrix} x_i & y_i & \theta_i \end{bmatrix}^T$.

For example, in step 1, assume that vehicle 1 is stationary and its position $x_{1,1}$, $y_{1,1}$ and orientation $\theta_{1,1}$ are known and vehicle 2 traveled to a new position. The measured distance between the vehicles is r_1 and the measured bearing angles are α_{11} and α_{12}, where the first index refers to the step number and the second index refers to the measuring vehicle (see Figure 3.3, left). Therefore, the Cartesian position and orientation of vehicle 2 with respect to vehicle 1 is:

$$
\begin{aligned}
x_{1,2} &= r_1 \cos \alpha_{11} \\
y_{1,2} &= r_1 \sin \alpha_{11} \\
\theta_{1,2} &= \alpha_{11} + 180° - \alpha_{12}
\end{aligned}
\quad . \tag{23}
$$

By setting the initial position and orientation of vehicle 1 as the origin of the global coordinate system, meaning $x_{1,1}=0$, $y_{1,1}=0$ and $\theta_{1,1}=0$, Eq. (23) represents the global position of vehicle 2 at the end of the first step.

Note that the orientation θ is determined solely by bearing measurements and is hardly influenced at all by the distance measurement, unlike in the two-point approach (3.3), where the orientation accuracy is decreased by the distance. This is one of the key advantages of our method since distance errors tend to increase together with the distance while angle measurements remain almost unchanged.

In step 2, vehicle 1 travels to its next target point while vehicle 2 is stationary and its position is known ($x_{2,2}=x_{1,2}$, $y_{2,2}=y_{1,2}$ and $\theta_{2,2}=\theta_{1,2}$). At the end of the step, the distance and angle measurements are r_2, α_{22} and α_{21} (see Figure 3.3, center). It should be noted that the measurements are obtained with respect to vehicle's 2 current position and its local coordinate system. In order to obtain the position of vehicle 1 in the global coordinate system, a transformation is needed.

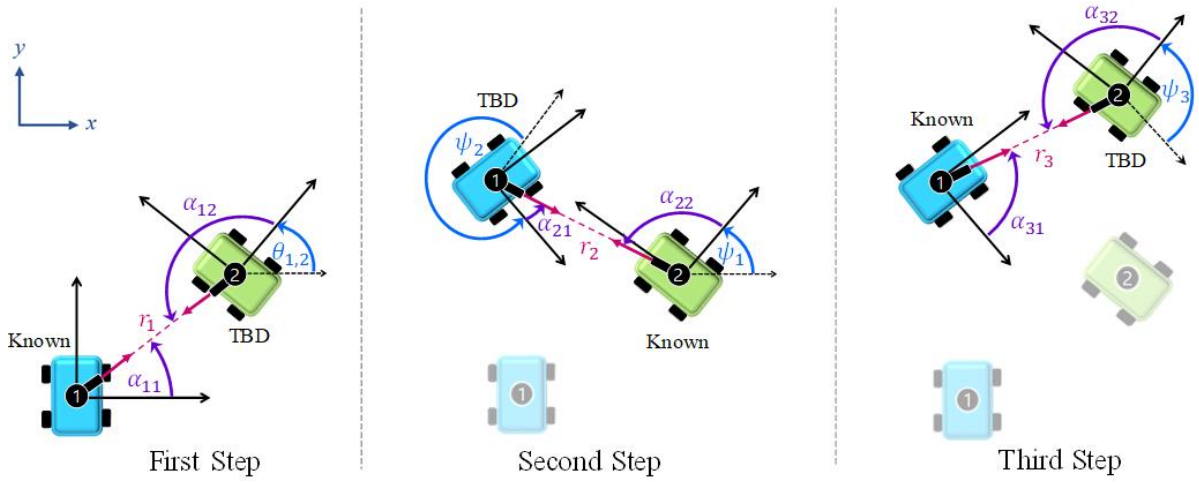


Figure 3.3: The first three steps and their measurements. At each step, one robot is static and tracks the motion of the advancing robot.

3.5 Multistep Representation using Homogeneous Coordinates

The local transformation matrix (2.1) at step *n* from the traveling vehicle's coordinate system (*n*) to the stationary vehicle's coordinate system (*n-1*) is:

$$A_{n-1}^{n}=\begin{bmatrix}\cos\psi_n & -\sin\psi_n & r_n\cos\alpha_{ns}\\ \sin\psi_n & \cos\psi_n & r_n\sin\alpha_{ns}\\ 0 & 0 & 1\end{bmatrix}, \tag{24}$$

where ψ_n is the relative angle between the two coordinate systems, hence $\psi_n=\alpha_{ns}+180°-\alpha_{nt}$ (see Figure 3.3). Obtaining the position of the traveling vehicle in the global coordinate system (0), can be achieved recursively as follows:

$$A_0^n = A_0^{n-1}\cdot A_{n-1}^n, \tag{25}$$

where A_0^{n-1} is the overall transformation matrix obtained in the last step (*n*-1), and A_{n-1}^n is the *n* step's local transformation matrix as shown in Eq. (24).

Since the transformation matrix is composed of a rotation matrix and a shifting vector, the first two expressions of the third column of matrix A_0^n are the Cartesian location of the traveling vehicle in the global coordinate system at step *n*, and the angle of the rotation matrix is the vehicle's orientation in the global coordinate system.

For example, the transformation matrix of the first step is:

$$A_0^1=\begin{bmatrix}\cos\psi_1 & -\sin\psi_1 & r_1\cos\alpha_{11}\\ \sin\psi_1 & \cos\psi_1 & r_1\sin\alpha_{11}\\ 0 & 0 & 1\end{bmatrix}. \tag{26}$$

The transformation matrix from the first to the second step:

$$A_1^2=\begin{bmatrix}\cos\psi_2 & -\sin\psi_2 & r_2\cos\alpha_{22}\\ \sin\psi_2 & \cos\psi_2 & r_2\sin\alpha_{22}\\ 0 & 0 & 1\end{bmatrix}. \tag{27}$$

Therefore, the overall transformation matrix for the second step is:

$$A_0^2=A_0^1\cdot A_1^2=\begin{bmatrix}\cos(\psi_1+\psi_2) & -\sin(\psi_1+\psi_2) & r_1\cos\alpha_{11}+r_2\cos(\psi_1+\alpha_{22})\\ \sin(\psi_1+\psi_2) & \cos(\psi_1+\psi_2) & r_1\sin\alpha_{11}+r_2\sin(\psi_1+\alpha_{22})\\ 0 & 0 & 1\end{bmatrix}. \tag{28}$$

Hence, vehicle's 1 location in the global coordinate system at the end of the second step (see Figure 3.3, center) is:

$$
\begin{aligned}
x_{2,1} &= r_1 \cos\alpha_{11} + r_2 \cos\left(\psi_1 + \alpha_{22}\right) \\
y_{2,1} &= r_1 \sin\alpha_{11} + r_2 \sin\left(\psi_1 + \alpha_{22}\right) \quad . \\
\theta_{2,1} &= \psi_1 + \psi_2 = \alpha_{11} - \alpha_{12} + \alpha_{22} - \alpha_{21}
\end{aligned} \tag{29}
$$

The general form of the location of the traveling vehicle at step *n*:

$$
\begin{aligned}
x_{n,t} &= \sum_{i=1}^{n} r_i \cos\left(\psi_{i-1} + \alpha_{is}\right) \\
y_{n,t} &= \sum_{i=1}^{n} r_i \sin\left(\psi_{i-1} + \alpha_{is}\right), \\
\theta_{n,t} &= \sum_{i=1}^{n} \psi_i
\end{aligned} \tag{30}
$$

where:

$$
\psi_i = \alpha_{is} + 180° - \alpha_{it} \, . \tag{31}
$$

4 Error Evaluation

Since all sensor measurements contain precision errors, this section presents a statistical analysis to evaluate the influence of the cumulative errors on the overall location of the robot after a large number of steps.

4.1 Exact Method

The measured location of the traveling vehicle at step n is expressed as a function of measured distances $r_1,r_2,\ldots,r_n$ and angles $\alpha_{11},\alpha_{12},\ldots,\alpha_{n1},\alpha_{n2}$ in the global coordinate system $f(r_1,\ldots,r_n,\alpha_{11},\ldots,\alpha_{n2})$; thus the real location including distance and bearing measurement errors, $\Delta r_1,\Delta r_2,\ldots,\Delta r_n$ and $\Delta\alpha_{11},\Delta\alpha_{12},\ldots,\Delta\alpha_{n1},\Delta\alpha_{n2}$ respectively, is $f(r_1+\Delta r_1,\ldots,r_n+\Delta r_n,\alpha_{11}+\Delta\alpha_{11},\ldots,\alpha_{n2}+\Delta\alpha_{n2})$.

For example, if during the first step (Eq. (23)) the distance and bearing were measured with an error of Δr_1, $\Delta\alpha_{11}$ and $\Delta\alpha_{12}$ respectively, the measured location of vehicle 2:

$$\begin{aligned} x_{1,2}^m &= \left(r_1+\Delta r_1\right)\cos\left(\alpha_{11}+\Delta\alpha_{11}\right) \\ y_{1,2}^m &= \left(r_1+\Delta r_1\right)\sin\left(\alpha_{11}+\Delta\alpha_{11}\right) \\ \theta_{1,2}^m &= \alpha_{11}+\Delta\alpha_{11}+180°-\left(\alpha_{12}+\Delta\alpha_{12}\right) \end{aligned} \quad . \tag{32}$$

This method uses the presented localization method directly (see Section 3.4 and 3.5), and hence requires multiple matrix multiplications and a large number of trigonometric calculations which result in high numerical complexity.

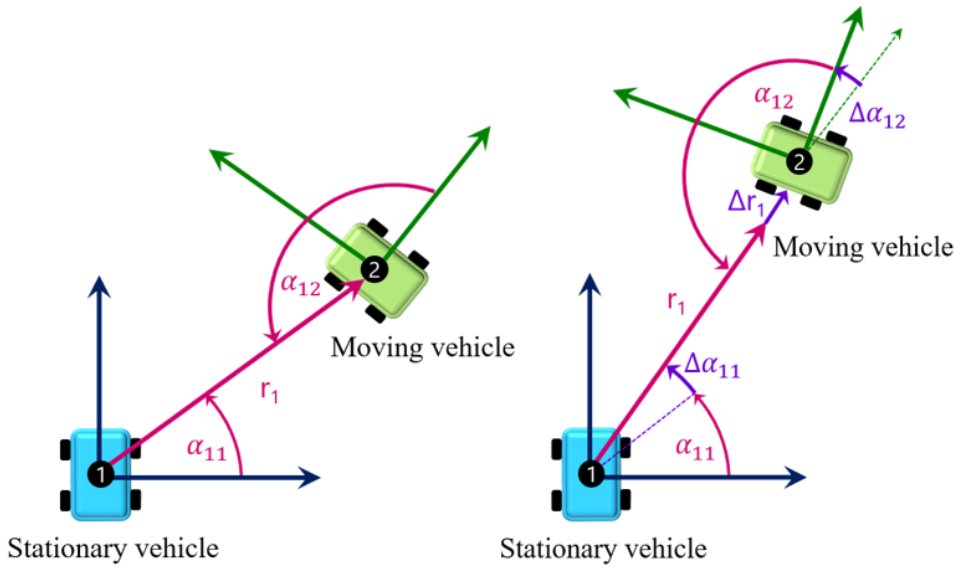


Figure 4.1: Real location (left), Measured location with sensor errors (right).

4.2 First Order Approximated Method

An approximated, yet computationally simpler method to evaluate the measured locations of the vehicles uses:

$$\begin{bmatrix} x^m \\ y^m \\ \theta^m \end{bmatrix} \approx \begin{bmatrix} x^r \\ y^r \\ \theta^r \end{bmatrix} + [J][\Delta], \tag{33}$$

where the index *'r'* refers to the real location, [Δ] is the measurement errors vector and [*J*] stands for the Jacobian matrix (2.2):

$$J_{ij} = \frac{\partial f_i}{\partial q_j}, \tag{34}$$

where f_i are the functions of Cartesian location and orientation and q_j are the variables of these functions, hence $r_1, \alpha_{11}, \alpha_{12}, \ldots, r_j, \alpha_{j1}, \alpha_{j2}$. For example, the estimated position of vehicle 2 after the first step is:

$$\begin{bmatrix} x_{1,2}^m \\ y_{1,2}^m \\ \theta_{1,2}^m \end{bmatrix} \approx \begin{bmatrix} r_1 \cos\alpha_{11} \\ r_1 \sin\alpha_{11} \\ \alpha_{11} + 180° - \alpha_{12} \end{bmatrix} + \begin{bmatrix} \cos\alpha_{11} & -r_1 \sin\alpha_{11} & 0 \\ \sin\alpha_{11} & r_1 \cos\alpha_{11} & 0 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} \Delta r_1 \\ \Delta\alpha_{11} \\ \Delta\alpha_{12} \end{bmatrix}. \tag{35}$$

At the next step, the location is determined by 6 measurements; thus, the Jacobian becomes a 3×6 matrix and the measurement error is a 6×1 vector. At step *n*, the Jacobian is a 3×3*n* matrix and the measurement error is a 3*n*×1 vector. A general form of the location error for step *n* is:

$$[E]_n = [J]_n[\Delta]_n = [E]_{n-1} + \begin{bmatrix} -r_n \sin(\theta_{n-1} + \alpha_{n,s}) \cdot \varepsilon + \Delta r_n \cos(\theta_{n-1} + \alpha_{n,s}) \\ r_n \cos(\theta_{n-1} + \alpha_{n,s}) \cdot \varepsilon + \Delta r_n \sin(\theta_{n-1} + \alpha_{n,s}) \\ \Delta\alpha_{n,s} - \Delta\alpha_{n,t} \end{bmatrix}, \tag{36}$$

where:

$$\varepsilon = \sum_{i=1}^{n} \Delta\alpha_{i,s} - \sum_{j=1}^{n-1} \Delta\alpha_{j,t}. \tag{37}$$

5 Monte Carlo Simulation

A Monte Carlo Simulation (MCS) was used in order to simulate a real-life scenario where the input of the sensors contains statistical errors. A natural random statistical error with a given standard deviation was inserted to the "measured values" and the statistical distribution of the position of the vehicles was calculated (using 10,000 simulations for each step), by both exact and approximated methods, as presented in Figure 5.1. This section first presents a comparison between the first order approximation to the exact method (5.1), the statistical distribution along the path (5.2), the influence of the sensor error on the accuracy of the measured location (5.3) and finally a comparison between different paths and advancing (parallel, alternating and following) methods (5.4).

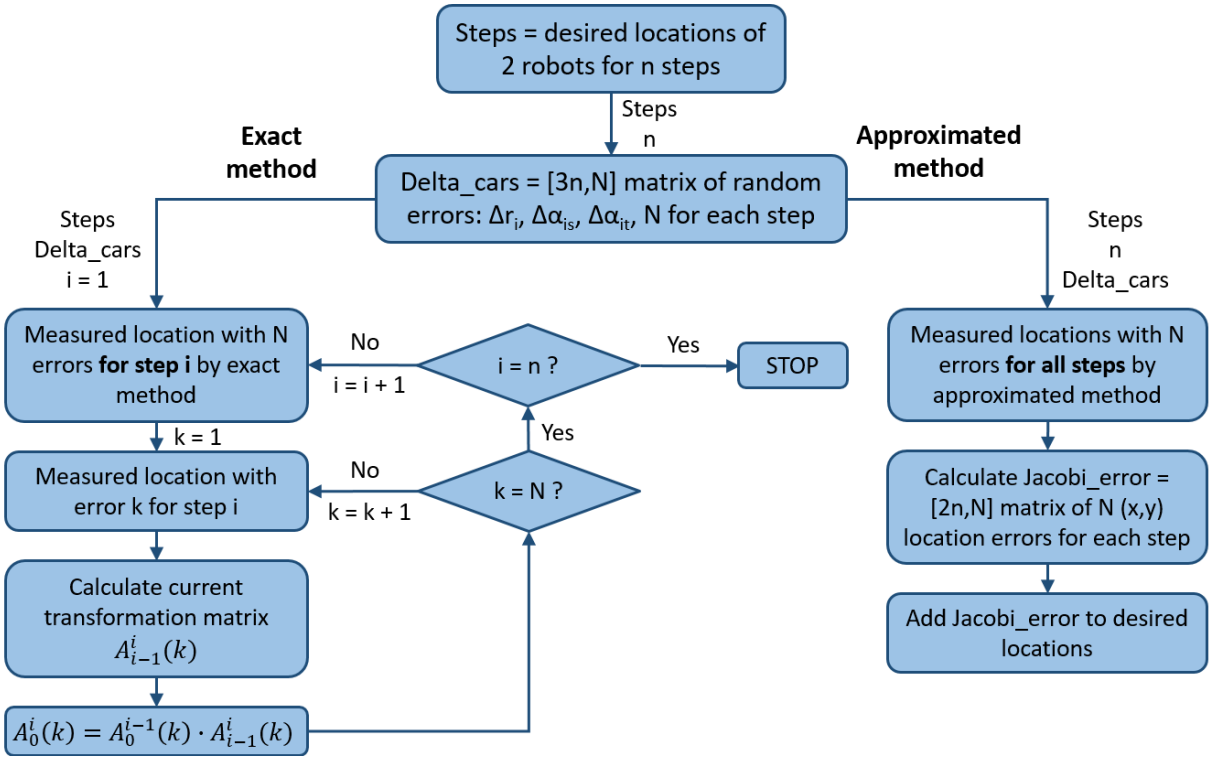


Figure 5.1: Flow chart of the Monte Carlo simulation, calculating locations with errors by two methods for a path of n steps, using a set of N random errors.

5.1 Comparing the First Order Approximated Method to the Exact Method

The MCS was first performed throughout a simple path composed of two straight lines as seen in Figure 5.2. At each step, the traveling vehicle advances by 8 m and the final distance from the stationary vehicle is 10 m; i.e. the system overall advances 200 m throughout 25 steps.

The distance and angle measurement errors were simulated as normally distributed (2.3) sets of N samples each (for each step), with zero mean. The standard deviation of the distance measurement error was set to $\sigma_d \cdot r_i$, where σ_d is the distance sensor's resolution and r_i is the current step's measured relative distance. The standard deviation of the angle measurement error was set to σ_α, the bearing sensor's resolution.

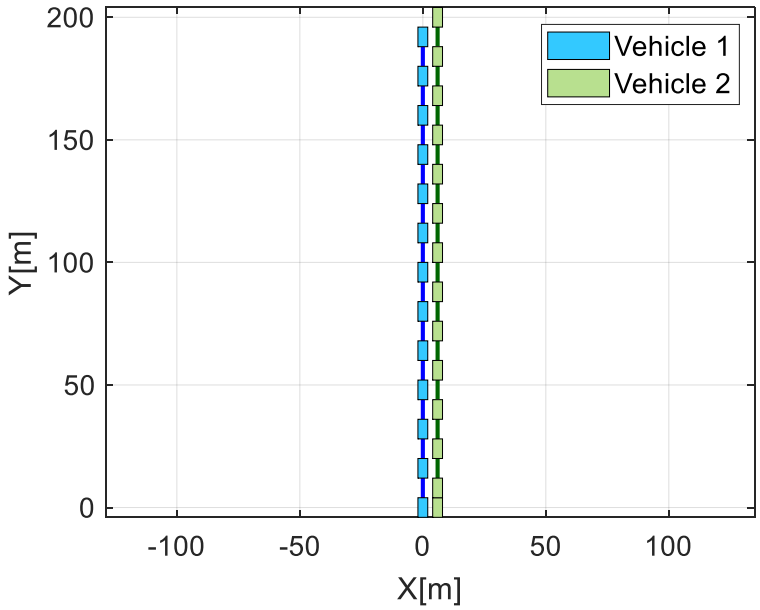


Figure 5.2: A 25 step straight line path. This simple path was chosen for our numerical MCS.

The MCS comparison was performed using the exact method (4.1, [B]7) and the approximated method (4.2, [B]5). In both cases, N=10,000; i.e., each step of the path was evaluated 10,000 times, for a set of 10,000 samples of random measurement errors ([B]4), resulting in 10,000 possible locations for each step. Figure 5.3 presents the relative difference between the final locations calculated by both methods, relative to the total traveled distance. Figure 5.3 (top) shows that for $\sigma_\alpha \leq 0.5°$ (a reasonable assumption for a standard bearing sensor), the difference between the exact and approximated methods is less than 0.1% of the traveled distance (200 meters in 25 steps). The error increases to 1.5% for $\sigma_\alpha = 2°$. Figure 5.3 (bottom) which presents the difference between the two methods as a function of distance standard deviation σ_d shows that the error is dominated by the angle error.

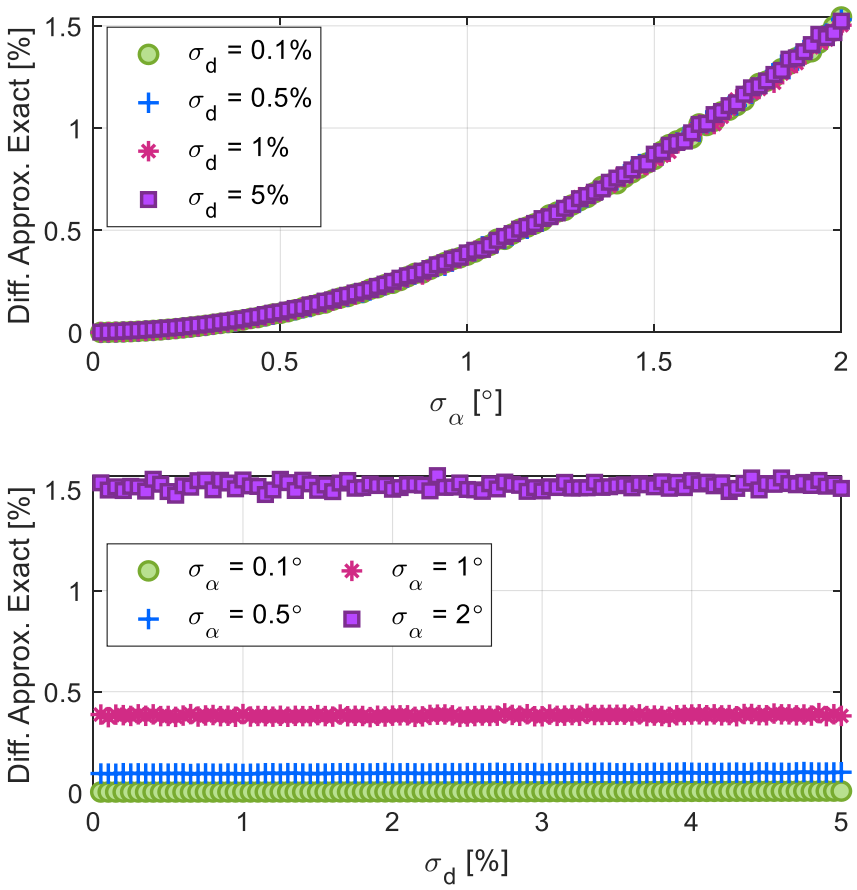


Figure 5.3: Relative difference between the last step's locations, calculated using the exact and approximated methods, relative to total distance traveled (200 m), as a function of bearing sensor's resolution (top), and as a function of range sensor's resolution (bottom). Each point is the average of 10,000 simulations.

As seen in Figure 5.1, the exact method uses transformation matrices, therefore the same matrix multiplication (with different random errors) has to be computed N times for each step, overall $N \cdot n$ matrix multiplications for n steps ([B]1, [B]7). In the approximated method on the other hand, due to the general form of the location errors (Eq. (36)-(37)), all N possible locations for each of the n steps are calculated directly using matrix addition ([B]1, [B]5). As a result, in terms of computation time, the approximated method was found to be nearly 200 times faster than the exact method computation time. Hence, the approximated method was used in the following MCS.

5.2 Statistical Distribution

This section presents a statistical analysis of the MCS location errors using σ_d=2% and σ_α=0.5°. The distribution of possible locations for each step is presented as a two-dimensional histogram (see Figure 5.4 and Appendix [B]8). The size and shape of the distribution can be described by three standard deviation values (2.3). The first is the total standard deviation according to the distance between the centroid and the different simulation results:

$$\sigma = \left(\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \mu_x)^2 + (y_i - \mu_y)^2 \right)^{0.5}, \tag{38}$$

where (μ_x,μ_y) are the coordinates of the approximated method's centroid and (x_i,y_i) are the coordinates of all possible locations, $i=1,\ldots,N$.

Since the distribution pattern of possible locations tends to yield an ellipse (see Figure 5.4), two other standard deviations were calculated according to the ellipse's axes. These values were obtained by calculating the covariance matrix of the N Cartesian locations ([B]9):

$$\mathrm{cov}\begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_N & y_N \end{bmatrix}, \tag{39}$$

resulting in a 2×2 covariance matrix, with two 2×1 eigenvectors $\{\bar{V}_1, \bar{V}_2\}$ and two corresponding eigenvalues $\{\lambda_1, \lambda_2\}$. The eigenvectors of the covariance matrix represent the direction of the ellipse's axes, and the square root of their corresponding eigenvalues represent the standard deviations in their direction. Assuming $\lambda_1 > \lambda_2$, the two standard deviations values are:

$$\sigma_1 = \lambda_1^{0.5},\ \sigma_2 = \lambda_2^{0.5}, \tag{40}$$

where σ_1 is the standard deviation in the direction of the main axis of the ellipse and σ_2 is the standard deviation in the perpendicular direction. The angle between the ellipse's main axis and the global x positive axis (see Figure 5.4 top left) is:

$$\beta = \mathrm{atan} \frac{\bar{V}_1(y)}{\bar{V}_1(x)}. \tag{41}$$

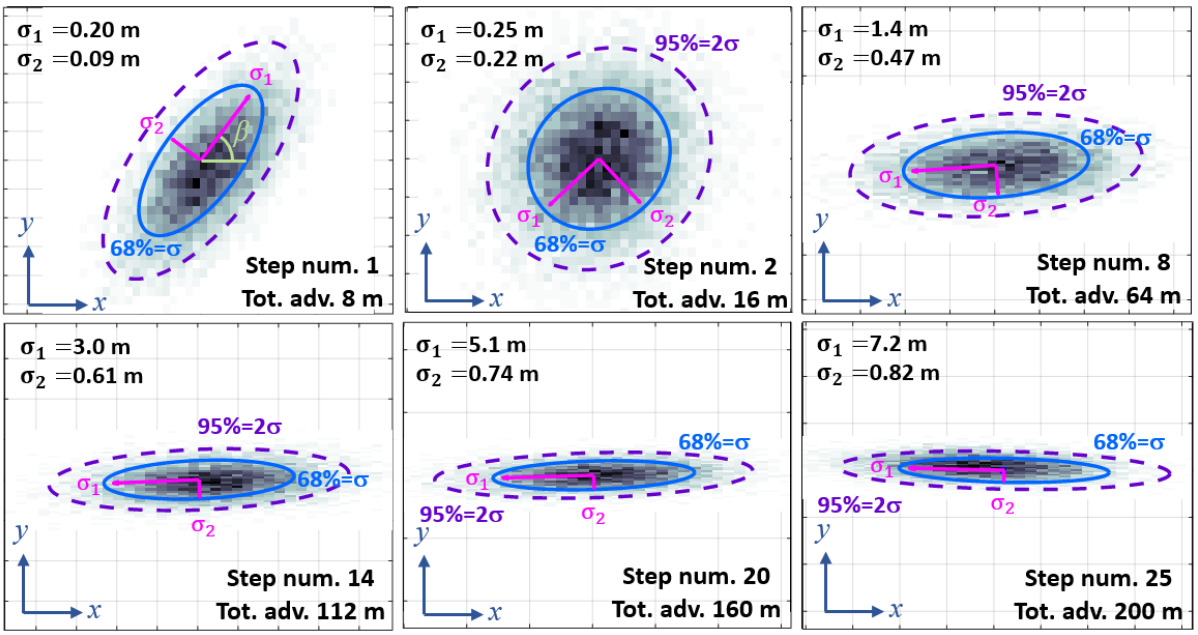


Figure 5.4: Histogram distribution of the measured locations using the MCS with 10,000 paths with σ_d =2% and σ_α=0.5° for steps 1, 2, 8, 14, 20, and 25 with the confidence distribution 68% (σ) and 95% (2σ).

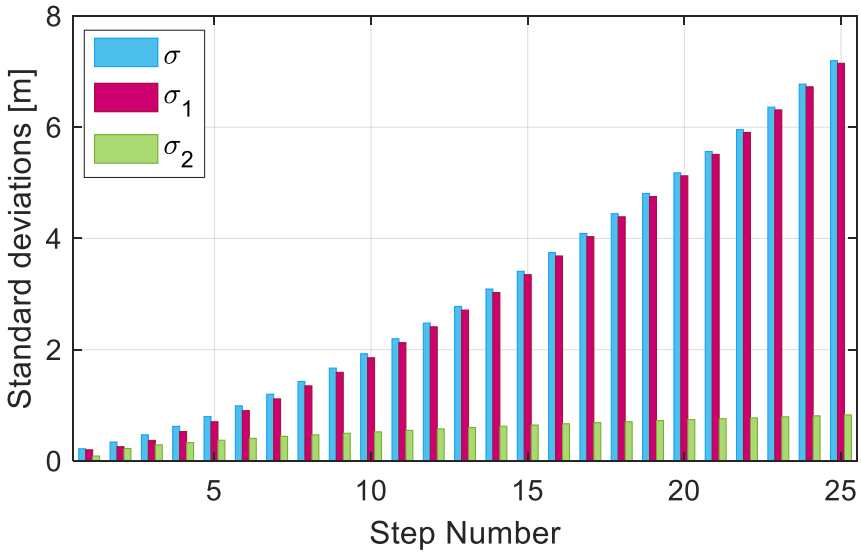


Figure 5.5: Standard deviations as a function of the number of steps for a 200 m straight path using σ_d=2% and σ_α=0.5°.

Although the error distribution of the first step seemed to be affected mostly by the distance sensor's error, the error distribution of the next step had a circular pattern. The pattern became elliptical in the next steps with σ_1 becoming larger relative to σ_2 (see Figure 5.4 and Figure 5.5). The overall standard deviation of the error σ grew almost linearly with the number of steps and distance traveled. The ratio of σ of the last step divided by the overall traveled distance is 0.036 which is in the same order of magnitude as the sensors' relative error.

Using Eq. (36), the standard deviation of the orientation of the vehicle at step *n* can be evaluated directly as follows:

$$\sigma_\theta = \sqrt{2n} \cdot \sigma_\alpha , \tag{42}$$

implying that the orientation error depends solely on the number of steps and the bearing sensor's accuracy.

Additionally, the standard deviations in the '*x*' and '*y*' axes directions can be analytically derived from Eq. (36) (see Appendices [A] and [B]6):

$$\sigma_x = \left[\sum_{i=1}^{n} r_i^2 \cos^2\left(\theta_{i-1} + \alpha_{i,s}\right)\sigma_d^2 + \sum_{j=1}^{n} \Omega(j) \left(\sum_{i=j}^{n} r_i \sin\left(\theta_{i-1} + \alpha_{i,s}\right) \right)^2 \sigma_\alpha^2 \right]^{0.5} , \tag{43}$$

$$\sigma_y = \left[\sum_{i=1}^{n} r_i^2 \sin^2\left(\theta_{i-1} + \alpha_{i,s}\right)\sigma_d^2 + \sum_{j=1}^{n} \Omega(j) \left(\sum_{i=j}^{n} r_i \cos\left(\theta_{i-1} + \alpha_{i,s}\right) \right)^2 \sigma_\alpha^2 \right]^{0.5} , \tag{44}$$

where:

$$\Omega(j) = \begin{cases} 1, & j = 1 \\ 2, & j > 1 \end{cases} . \tag{45}$$

These equations have been validated by comparing between the total standard deviation values σ calculated using the MCS (10,000 simulations) and the analytical expression (see Table 10.1).

5.3 Comparing the Influence of the Sensor Error on the Accuracy of the Measured Location

Figure 5.6 (top) presents the relative error between the measured position of the robot and the real position as a function of the standard deviation of the bearing measurement error σ_α, while Figure 5.6 (bottom) presents the same error as a function of the standard deviation of the distance measurement error σ_d. For each case, we ran 10,000 simulations, each composed of 25 steps and the total net advancement is 200 meters. The results presented in this figure show that the relative error is governed by the bearing error measurements.

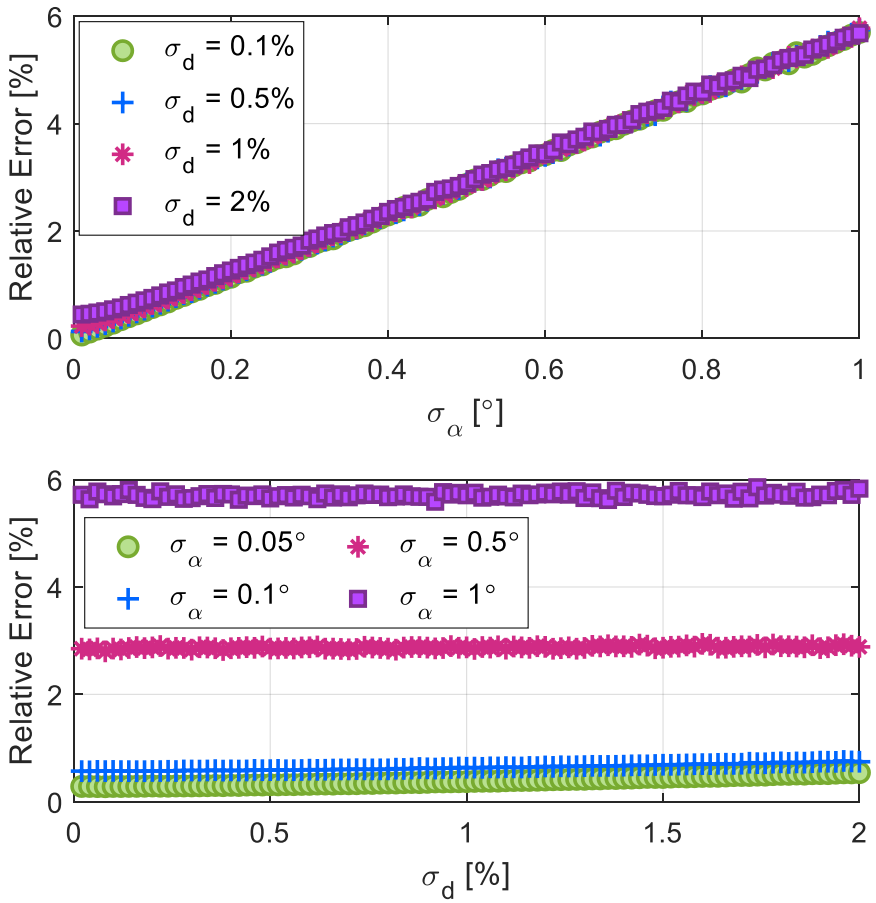


Figure 5.6: Relative error between the measured position of the robot and the real position for a travelled distance of 200 m, as a function of bearing sensor's resolution (top), and as a function of range sensor's resolution (bottom). Each point is the average of 10,000 simulations.

5.4 Path Comparison

In this section, MCS are used to statistically calculate the influence of the sensor accuracy on the location error for three different paths (straight, 'S' shape, and square) using three advancing methods (parallel, alternating and following). In total, nine scenarios were examined for four different combinations of sensor errors (see Table 5.1). The three different paths were chosen as basic segments that can be used to define more complex paths, whereas the three advancing methods map the most basic methods of forwards advancing of two vehicles.

Table 5.1: Values of sensor variables used in the simulation

Sensor variables	STD #1	STD #2	STD #3	STD #4
σ_d=error/distance	1%	5%	2%	5%
σ_α [°]	0.1	0.1	0.5	1

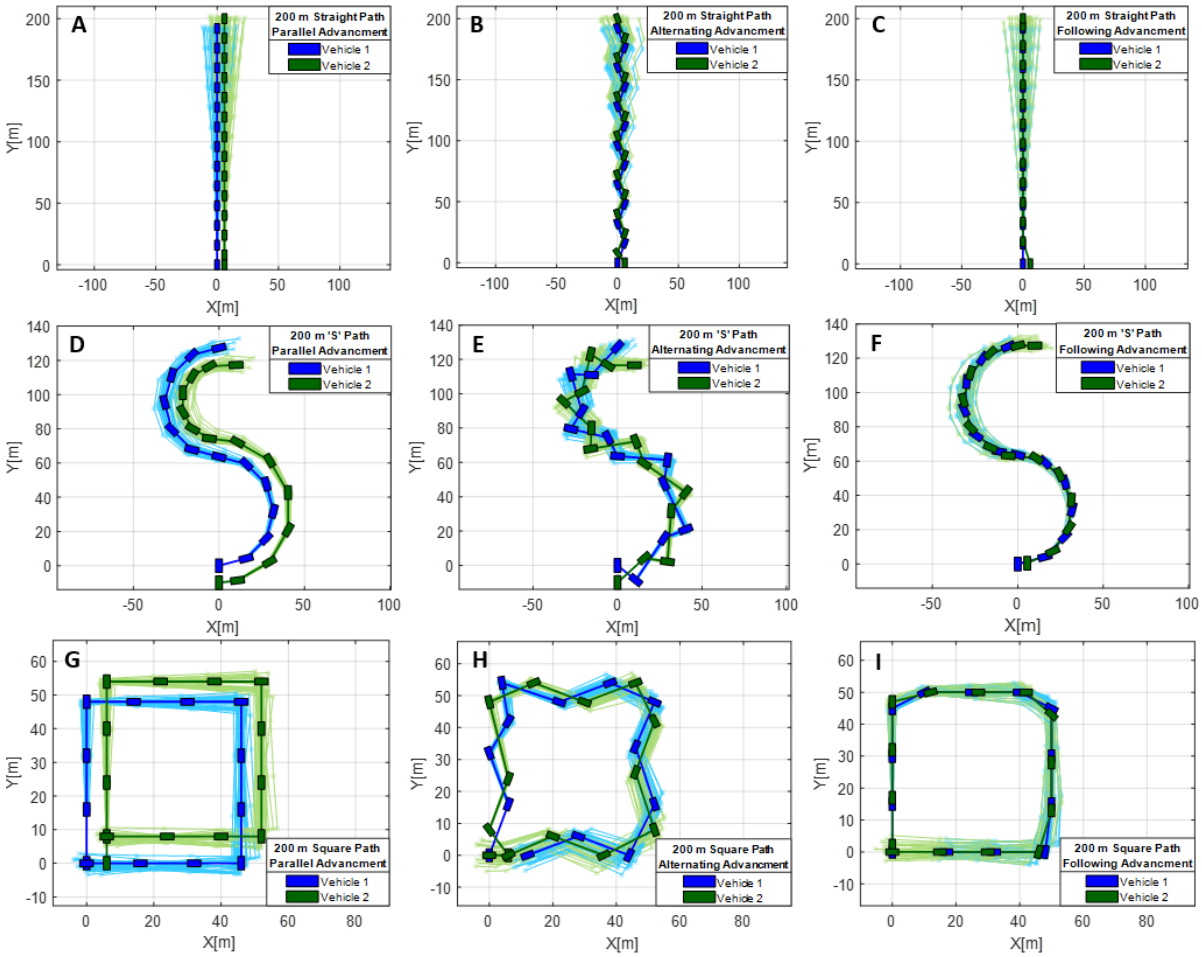


Figure 5.7: Nine scenarios of three different paths and three different advancing methods. From top to bottom: straight path, 'S' path and square path. From left to right: parallel advancing, alternating advancing and following advancing. Lighter colors present 30 optional locations due to random sensors errors with σ_d=2% and σ_α=0.5°.

Table 5.2: 200 meters straight path standard deviation values for different sensors resolution (using 10,000 simulations)

Sensor variables	Total standard deviation (σ)			Advancing direction (σ_y)			Perpendicular direction (σ_x)		
	Para.	Alter.	Follow.	Para.	Alter.	Follow.	Para.	Alter.	Follow.
σ_d=1%	1.51 m	1.49 m	1.51 m	0.404 m	0.403 m	0.631 m	1.46 m	1.43 m	1.37 m
σ_α=0.1°	[0.75%]	[0.72%]	[0.75%]	[0.20%]	[0.20%]	[0.31%]	[0.72%]	[0.72%]	[0.69%]
σ_d=5%	2.86 m	2.66 m	3.43 m	1.97 m	1.99 m	3.14 m	2.08 m	1.77 m	1.38 m
σ_α=0.1°	[1.4%]	[1.3%]	[1.7%]	[1.0%]	[1.0%]	[1.6%]	[1.0%]	[0.88%]	[0.69%]
σ_d=2%	7.23 m	7.16 m	6.97 m	0.836 m	0.840 m	1.26 m	7.18 m	7.11 m	6.86 m
σ_α=0.5°	[3.6%]	[3.6%]	[3.5%]	[0.42%]	[0.42%]	[0.63%]	[3.6%]	[3.6%]	[3.4%]
σ_d=5%	14.3 m	14.4 m	14.1 m	2.06 m	2.08 m	3.17 m	14.2 m	14.2 m	13.8 m
σ_α=1°	[7.2%]	[7.2%]	[7.1%]	[1.0%]	[1.0%]	[1.6%]	[7.2%]	[7.1%]	[6.9%]

Table 5.3: 200 meters 'S' path standard deviation values for different sensors resolution (using 10,000 simulations)

Sensor variables	Total standard deviation (σ)			Advancing direction (σ_x)			Perpendicular direction (σ_y)		
	Para.	Alter.	Follow.	Para.	Alter.	Follow.	Para.	Alter.	Follow.
σ_d=1%	1.07 m	1.14 m	1.20 m	0.926 m	0.960 m	1.02 m	0.542 m	0.609 m	0.629 m
σ_α=0.1°	[0.54%]	[0.57%]	[0.60%]	[0.46%]	[0.48%]	[0.51%]	[0.27%]	[0.30%]	[0.31%]
σ_d=5%	3.35 m	3.89 m	4.02 m	2.43 m	2.80 m	2.91 m	2.31 m	2.70 m	2.78 m
σ_α=0.1°	[1.7%]	[1.9%]	[2.0%]	[1.2%]	[1.4%]	[1.4%]	[1.2%]	[1.3%]	[1.4%]
σ_d=2%	4.46 m	4.48 m	4.86 m	4.12 m	4.11 m	4.49 m	1.71 m	1.79 m	1.87 m
σ_α=0.5°	[2.2%]	[2.2%]	[2.4%]	[2.1%]	[2.1%]	[2.2%]	[0.85%]	[0.90%]	[0.94%]
σ_d=5%	9.17 m	9.23 m	10.0 m	8.40 m	8.36 m	9.12 m	3.68 m	3.92 m	4.12 m
σ_α=1°	[4.6%]	[4.6%]	[5.0%]	[4.2%]	[4.2%]	[4.6%]	[1.8%]	[2.0%]	[2.1%]

Table 5.4: 200 meters square path standard deviation values for different sensors resolution (using 10,000 simulations)

Sensor variables	Total standard deviation (σ)			Advancing direction (σ_x)			Perpendicular direction (σ_y)		
	Para.	Alter.	Follow.	Para.	Alter.	Follow.	Para.	Alter.	Follow.
σ_d=1%	0.877 m	0.973 m	0.831 m	0.607 m	0.734 m	0.587 m	0.634 m	0.638 m	0.583 m
σ_α=0.1°	[0.44%]	[0.49%]	[0.42%]	[0.30%]	[0.37%]	[0.29%]	[0.32%]	[0.32%]	[0.29%]
σ_d=5%	3.70 m	3.96 m	3.03 m	2.54 m	3.02 m	2.14 m	2.69 m	2.56 m	2.15 m
σ_α=0.1°	[1.8%]	[2.0%]	[1.5%]	[1.3%]	[1.5%]	[1.1%]	[1.3%]	[1.3%]	[1.1%]
σ_d=2%	2.83 m	3.24 m	3.15 m	1.99 m	2.41 m	2.25 m	2.01 m	2.19 m	2.20 m
σ_α=0.5°	[1.4%]	[1.6%]	[1.6%]	[0.99%]	[1.2%]	[1.1%]	[1.0%]	[1.1%]	[1.1%]
σ_d=5%	6.07 m	6.92 m	6.48 m	4.27 m	5.14 m	4.59 m	4.32 m	4.63 m	4.57 m
σ_α=1°	[3.0%]	[3.5%]	[3.2%]	[2.1%]	[2.6%]	[2.3%]	[2.2%]	[2.3%]	[2.3%]

Figure 5.7 A-C, presents the path distribution of 30 simulations in a straight path using three different advancing methods: A) parallel, B) alternating and C) following. Figure 5.7 D-F, and G-I present the same different advancing methods for 'S' shape and square paths respectively. We used σ_d=2% and σ_α =0.5°.

The resulting total standard deviation, and its components along the direction of motion and in the vertical direction are summarized in Table 5.2, Table 5.3 and Table 5.4. Note that the standard deviation values were calculated twice; once using MCS (10,000 simulations) and then

using the analytical expressions developed in the Appendix. The relative difference between both methods is always smaller than 1% (see Appendix [A]).

For the straight path, (Table 5.2), the overall standard deviation is the largest compared to the other paths and is nearly unaffected by the advancing method. However, for the other two paths ('S' shape (Table 5.3) and square (Table 5.4)), the parallel advancing method mostly generated smaller location errors, where the square path resulted with the smallest errors.

For the straight path, the standard deviation in the vertical direction is substantially larger than the standard deviation in the direction of motion. The square path on the other hand, resulted in nearly equal standard deviations both in the parallel and perpendicular direction, due to equal advancement in both directions.

Overall, the three different advancing methods (parallel, alternating and following) do not result in significant differences in the standard deviation values within a specific path. The size and the distribution pattern of the errors are influenced mainly by the overall advancing direction of the system and almost unaffected by the relative position of the vehicles within each step.

6 Experiments

This section presents a real-world experimental system that was used to validate our algorithm (6.1), experimental results (6.2) and comparison to the previously presented Monte Carlo simulation (0).

6.1 Experimental System

To validate our algorithm and simulations, we built a two-robot experimental system fitted with rotating turrets and cameras (see Figure 6.1). Each turret is equipped with a smartphone's video camera (1080×1920 pixels at 30fps). A green 6.2 cm tennis ball was placed at the top of the turret for visual identification. The turret is connected to a servo motor controlled by an Arduino microcontroller programmed to continuously rotate the turret by steps of 45 degrees, from zero till 180 and returning (see Figure 6.2). At each stop, the turret pauses for one second. Given the camera's field of view in the horizontal direction γ_x is 40 degrees, the total field of view of each robot is 220 degrees.

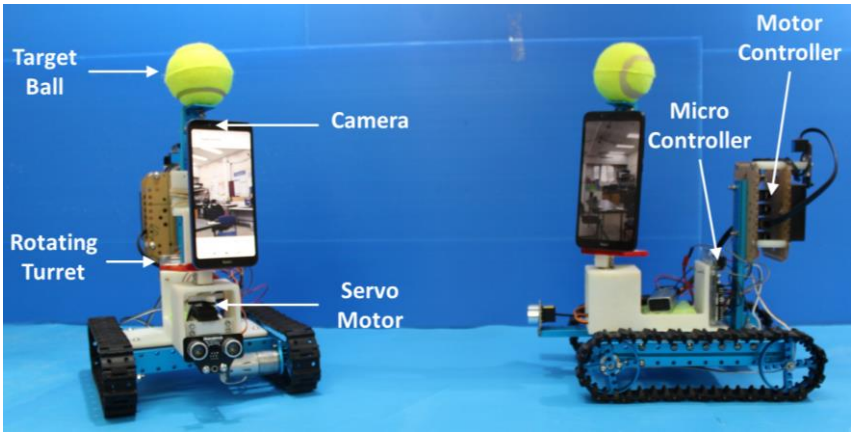


Figure 6.1: The robotic system used in the experiments.

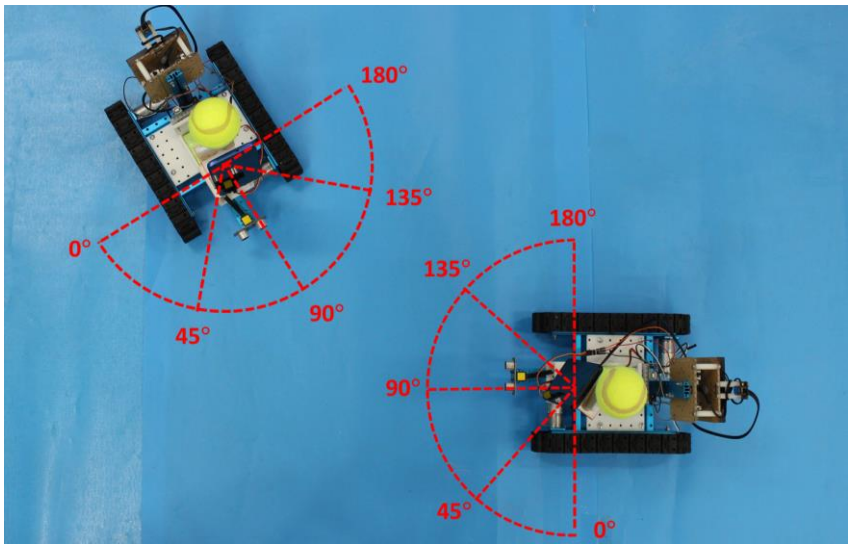


Figure 6.2: Top view of the robotics system. The turrets rotate in steps of 45 degrees.

We ran four different experiments that were each repeated five times. In each experiment, the robots (controlled by a human operator) advance in alternating steps while the turret rotates as the camera continuously records video. The localization of the robots is performed off-line at the end of the experiment (see Appendix [C]). Each step is represented by two images (1080×1920), one from each camera and the orientation of the turrets α_{turret}.

The bearing angle α of each robot is the sum of the orientation of the turret α_{turret} plus the angular position of the tennis ball in the picture α_{image}:

$$\alpha = \alpha_{turret} + \alpha_{image} . \tag{46}$$

The angular position in the image is calculated using:

$$\alpha_{image} = \text{atan}\left(\frac{b_x}{N_x / 2} \tan\left(\frac{\gamma_x}{2} \right) \right), \tag{47}$$

where b_x is the x coordinate of the center of the ball in pixels, with respect to the center of the frame (see Figure 6.3, top right) and N_x is the overall size of the image in pixels in the x direction.

The distance r between the robots (see Figure 6.3, bottom right) is calculated using:

$$r = \frac{L}{\text{atan}\left(\alpha_L\right)}, \tag{48}$$

where L is the length of the turret (21 cm) and α_L is the view angle of the height of the turret in the frame, calculated from the image:

$$\alpha_L = \frac{N_L}{N_y} \cdot \gamma_y, \tag{49}$$

where N_L is the size of the turret in pixels and N_y is the overall size of the image in pixels in the y direction. The camera's field of view γ_y in the vertical direction is 70 degrees. The distance r at each step is calculated from the average of both images (one from each robot).

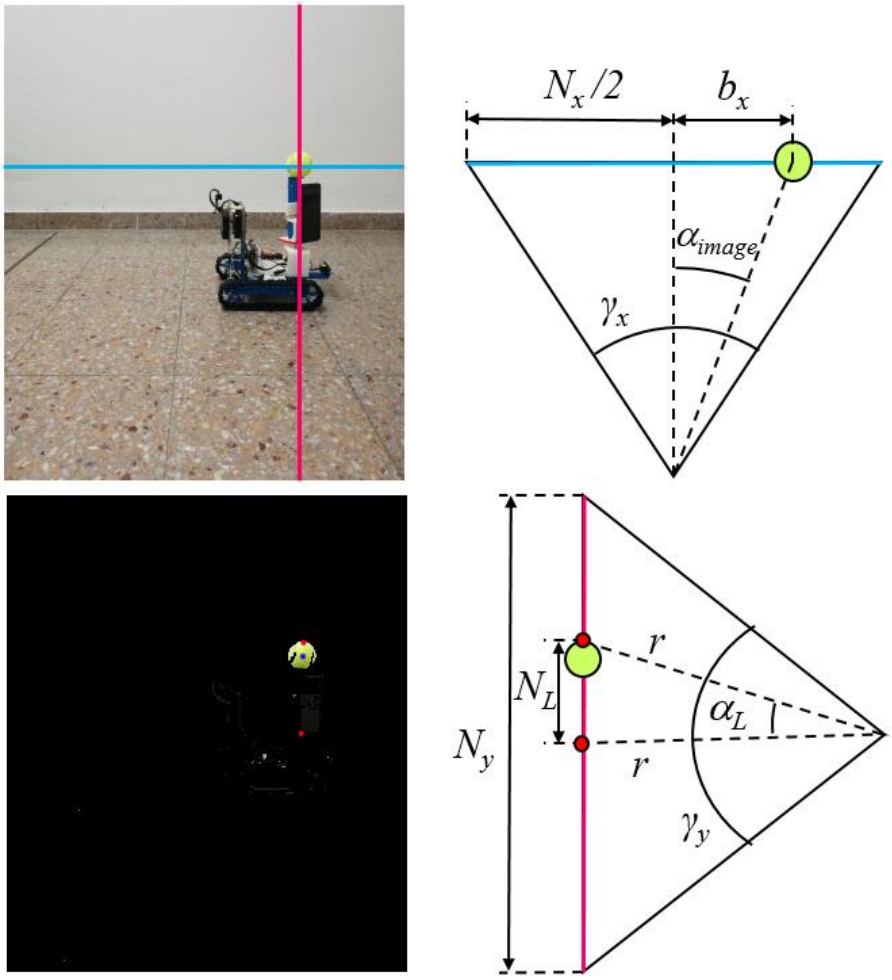


Figure 6.3: Schematic diagram of distance (bottom right) and bearing (top right) calculation from frame. Top left: original frame, bottom left: frame after image filtering, center of ball and top and bottom of turret detected.

6.2 Results

The results of the experiments are presented in Figure 6.4, Table 6.1 and Table 6.2. Both straight parallel and square parallel experiments show high repeatability with a relatively small average error (1.1% and 0.14% respectively). The total standard deviation is also relatively small with respectively 1% and 0.63%. The straight following and 'S' parallel experiments results are not as highly repeatable, indicating a possible drift in the servo motor. We believe that the most significant error in our experimental system is a systematic error in the bearing measurements which could be addressed by replacing the servo motor with a higher accuracy device. As previously discussed, the bearing sensor's error has a high impact on the localization error.

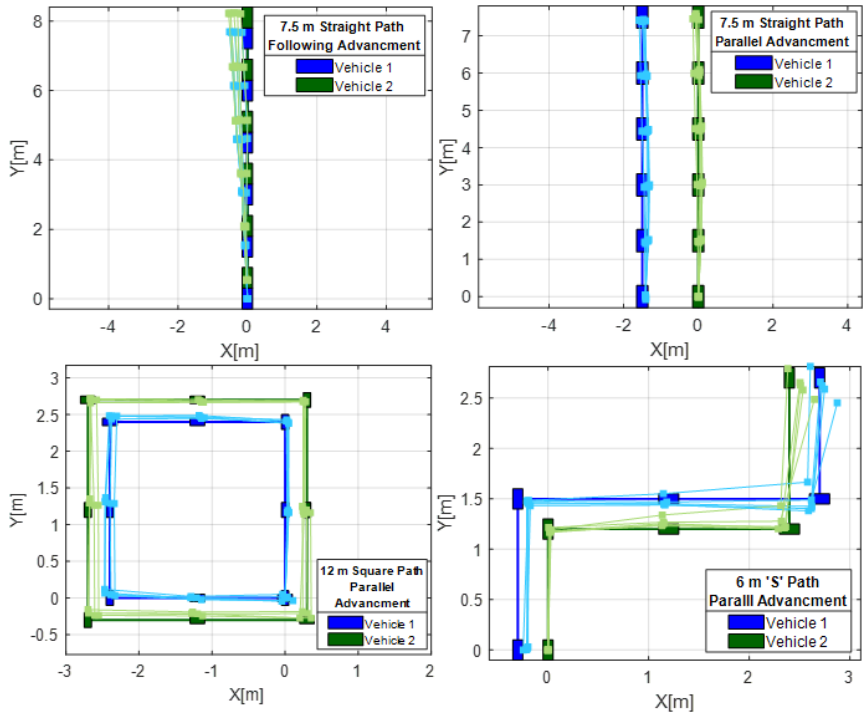


Figure 6.4: Experiment results of four different paths; top: straight path following (left) and parallel (right), bottom: square path (left) and 'S' path (right). Darker colors present real locations and lighter colors present calculated locations from five experiment results.

Table 6.1: Standard deviation values of experiments' last step results.

Paths	Total σ	Advancement dir. σ_y	Perpendicular dir. σ_x	Orientation σ_θ
Straight Follow [10 steps, 7.5 m]	0.143 m [1.9%]	0.00897 m [0.12%]	0.143 m [1.9%]	1.03°
Straight Parallel [10 steps, 7.5 m]	0.0773 m [1.0%]	0.0492 m [0.66%]	0.0597 m [0.79%]	3.00°
Square Parallel [16 steps, 12 m]	0.0751 m [0.63%]	0.0594 m [0.49%]	0.0458 m [0.38%]	1.06°
'S' Parallel [8 steps, 6 m]	0.179 m [3.0%]	0.127 m [2.1%]	0.0997 m [1.7%]	4.78°

Table 6.2: Mean error values of experiments' last step results.

Paths	Total Mean Error	Advancement dir. Mean Error	Perpendicular dir. Mean Error	Orientation Mean Error
Straight Follow [10 steps, 7.5 m]	0.395 m [5.3%]	0.152 m [2.0%]	-0.363 m [4.8%]	0.523°
Straight Parallel [10 steps, 7.5 m]	0.0797 m [1.1%]	-0.0424 m [0.56%]	-0.0278 m [0.37%]	3.57°
Square Parallel [16 steps, 12 m]	0.0174 m [0.14%]	-0.0078m [0.065%]	-0.00006m [0.0005%]	-0.011°
'S' Parallel [8 steps, 6 m]	0.207 m [3.4%]	0.0529 m [0.88%]	0.200 m [3.3%]	-3.35°

6.3 Comparison to Simulation

The system's repeatability error was evaluated by repeating the same measurement at least 10 times. The standard deviation of the calculated distances and bearing angles were σ_d=1% (with respect to the real distance) and σ_α=0.3°. The values of the standard deviation were implemented in the MCS in order to compare the simulation to the experiments. The simulation results presented in Table 6.3 show that the experimental results (Table 6.1) are of the same order as expected by the simulation. Note that since the MCS uses normally distributed random errors, the mean error for each step is zero.

For the first three experiments, straight following, straight parallel and square parallel paths, the standard deviation is very similar to the simulation, especially for the square path. In the 'S' shape path, the standard deviation of the experiment is of the same order as the simulation but is slightly more than 3 times larger. All four experiments show high compatibility in the relation between the standard deviation in the advancement and in the perpendicular directions, meaning the experimental system describes with high accuracy the distribution pattern of the location errors.

Table 6.3: Standard deviation values of simulations' last step results with σ_d=1% and σ_α=0.3°.

Paths	Total σ	Advancement dir. σ_y	Perpendicular dir. σ_x	Orientation σ_θ
Straight Following [10 steps, 7.5 m]	0.102 m [1.4%]	0.0487 m [0.65%]	0.0895 m [1.2%]	0.0234°
Straight Parallel [10 steps, 7.5 m]	0.114 m [1.5%]	0.0423 m [0.56%]	0.106 m [1.4%]	0.0234°
Square Parallel [16 steps, 12 m]	0.0778 m [0.65%]	0.0637 m [0.53%]	0.0446 m [0.37%]	0.0296°
'S' Parallel [8 steps, 6 m]	0.0523 m [0.87%]	0.0389 m [0.65%]	0.0350 m [0.58%]	0.0209°

7 Control

This section presents a path planning algorithm for the two-robot system, while considering state, input and path constraints (7.1). Then, a closed-loop path following controller is described in polar coordinates (7.2). The fusion of both algorithms is implemented in Section 7.3.

7.1 Autonomous Path Planning

In this section, the objective is to design a controller which will allow the two-robot system to advance autonomously in an uncharted constrained environment which lacks GPS reception such as a narrow underground tunnel. To do so, each step of the system is considered as an optimal control problem, where the goal is to advance to a chosen target point in minimum time, under the surrounding constraints (which will be defined shortly).

Assuming each robot is a differentially driven vehicle whose control inputs are (v,ω), defined respectively as the linear and angular velocities, the continues kinematic model of a single robot can be defined either by Cartesian or polar coordinates. After considering both representations, the Cartesian representation was chosen due to simpler representation of the constraints.

7.1.1 The Model

The continuous Cartesian kinematic model of a single robot:

$$\begin{aligned} \dot{x} &= v\sin\theta \\ \dot{y} &= v\cos\theta \\ \dot{\theta} &= \omega \end{aligned}, \tag{50}$$

where (x,y) represent the Cartesian location of the robot and θ represents the heading angle. The global coordinate system is set as the initial position of the moving robot (x_0,y_0). The heading angle θ is measured with respect to the positive y axis, clockwise (see Figure 7.1). Notice that both the heading angle θ and the angular velocity ω directions are CW.

The states of the system are:

$$\mathbf{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^T, \tag{51}$$

and the control signals are:

$$\mathbf{u} = \begin{bmatrix} v & \omega \end{bmatrix}^T. \tag{52}$$

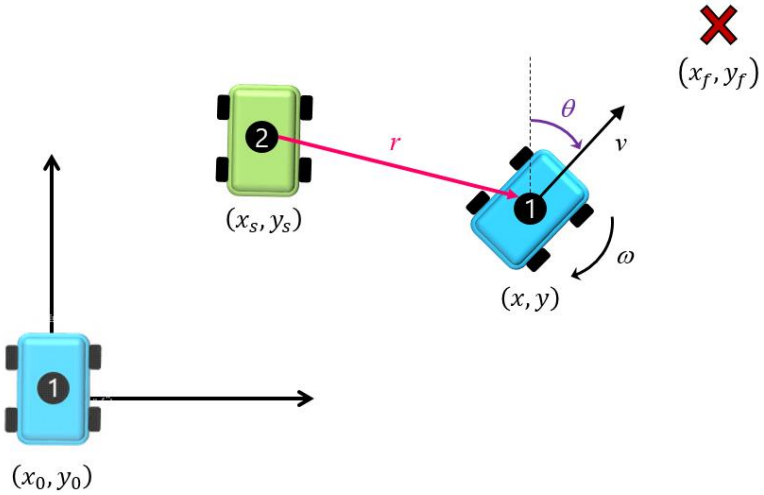


Figure 7.1: The system.

Table 7.1: List of state parameters and control inputs

State/control signal	Units	Description
x	[m]	Cartesian location with respect to initial position.
y	[m]	
θ	[rad]	Heading angle with respect to positive y axis.
v	[m/sec]	Linear velocity
ω	[rad/sec]	Angular velocity

7.1.2 Implementation

Since the system is not linear (see Eq. (50)), linear methods for control under constraints such as Model Predictive Control (MPC) or Interpolation Control (IC) are not practical. Therefore, we have decided to use FALCON.m - an optimal control tool for MATLAB, developed at the institute of *Flight System Dynamics* of *Technische Universität München* [43]. FALCON.m uses direct discretization methods in combination with gradient based numerical optimization and automatic analytic differentiation to solve mathematical optimal control problems. The numerical optimization is performed by IPOPT – Interior Point Optimizer, a software library for large scale nonlinear optimization of continuous systems [56].

As an optimization problem, the objective is solving the problem while minimizing the *cost function* (2.7). The cost function is set as the final time t_f where the robot arrives to its goal point:

$$\min J = t_f . \tag{53}$$

7.1.3 Defining Constraints

a. Movement constraints

Limiting the linear and angular velocities of the vehicle:

$$\begin{matrix} 0 \le v \le v_{\max} \\ -\omega_{\max} \le \omega \le \omega_{\max} \end{matrix}, \tag{54}$$

where the linear velocity is constrained to be positive, meaning the vehicle can only advance forwards.

b. Linear constraints due to shape of tunnel

$$\begin{matrix} x_{lb} \le x \le x_{ub} \\ y_{lb} \le y \le y_{ub} \\ -2\pi \le \theta \le 2\pi \end{matrix}, \tag{55}$$

where x_{lb}, x_{ub}, y_{lb} and y_{ub} are defined by the shape of the tunnel and the orientation of the robot is actually not constrained. If the object is advancing forwards, the orientation can be constrained as follows:

$$-\pi/2 - \delta\theta \le \theta \le \pi/2 + \delta\theta, \tag{56}$$

where $\delta\theta$ is some small angle in order to allow maneuvers (recall that θ is defined with respect to the positive y axis, see Figure 7.1).

c. Visibility by static vehicle constraint

As defined previously, the two vehicles must remain in each other's range of 'sight', therefore:

$$\varepsilon \le r \le r_{\max}, \tag{57}$$

where:

$$r = \sqrt{(x - x_s)^2 + (y - y_s)^2}, \tag{58}$$

where r is the distance between the stationary and the moving vehicle, r_{max} is the distance sensor's maximal range and ε is the minimal allowed distance between the moving vehicle and an obstacle (the stationary vehicle is also an obstacle).

d. Initial and final positions

The initial and final positions could be set either as equality or inequality constraints:

$$X_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$$
$$\begin{bmatrix} x_f & y_f & -2\pi \end{bmatrix}^T \le X_f \le \begin{bmatrix} x_f & y_f & 2\pi \end{bmatrix}^T, \tag{59}$$

meaning the final desired location is (x_f,y_f) whereas the orientation of the robot at the final point time is not constrained.

e. Avoid obstacles

The final constrains are defined by the obstacles in the environment.

1. Ellipse obstacle

The constraint due to an ellipse shaped obstacle with a center of (C_x,C_y), semi axes of a and b and orientation of ρ is defined as follows:

$$\frac{\left((x-C_x)\cos\rho+(y-C_y)\sin\rho\right)^2}{(a+\varepsilon)^2}+\frac{\left((x-C_x)\sin\rho-(y-C_y)\cos\rho\right)^2}{(b+\varepsilon)^2}\ge 1, \tag{60}$$

where the addition of ε is set so the robot will not come closer than ε to the obstacle.

2. Rectangular obstacle

The constraint due to a rectangular obstacle with a center of (R_x,R_y) and the dimensions of $2p\times 2q$ is defined as follows:

$$\left|\frac{(x-R_x)}{p+\varepsilon}+\frac{(y-R_y)}{q+\varepsilon}\right|+\left|\frac{(x-R_x)}{p+\varepsilon}-\frac{(y-R_y)}{q+\varepsilon}\right|\ge 2, \tag{61}$$

where the addition of ε is set so the robot will not come closer than ε to the obstacle. As will be demonstrated ahead, the rectangular obstacle is useful for dealing with corners in the explored tunnel.

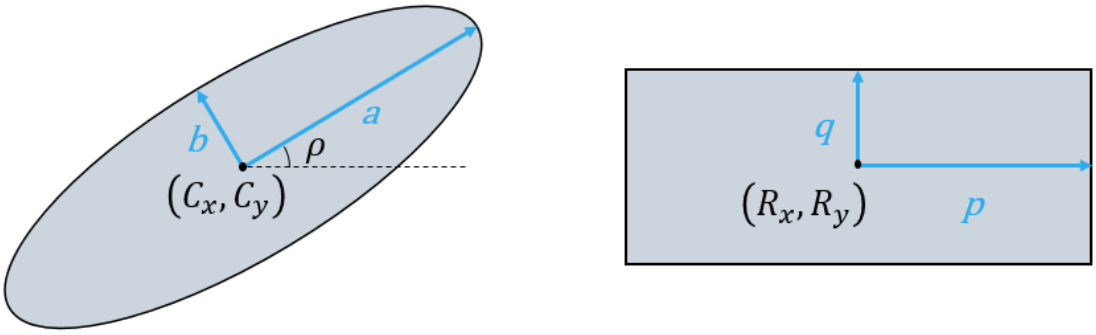


Figure 7.2: Ellipse (left) and rectangular (right) obstacles.

7.1.4 Simulation Results

This section presents the results of implementing the described control algorithm. First, single step scenarios were simulated (1) and then, a multi-step simulation is presented (2). In all figures, the two robots are marked by blue and green rectangles, black lines and ellipses represent borders and obstacles respectfully and the dashed black line represents the visibility constraint from the stationary robot.

1. Single Step Scenarios

This section presents the results of four different single step scenarios, validating the algorithm for different cases of borders and obstacles.

Table 7.2: Constant values for all single step simulations.

(x_0,y_0)	(x_s,y_s)	v_{max}	ω_{max}	r_{max}	ε
(0 [m], 0 [m])	(8 [m], 6 [m])	5 [m/sec]	π/8 [rad/sec]	10 [m]	0.5 [m]

a. Straight tunnel no obstacle

Table 7.3: Constant values for simulation (a).

(x_f,y_f)	x_{lb}	x_{ub}	y_{lb}	y_{ub}
(8 [m], 16 [m])	-2 [m]	10 [m]	0 [m]	inf

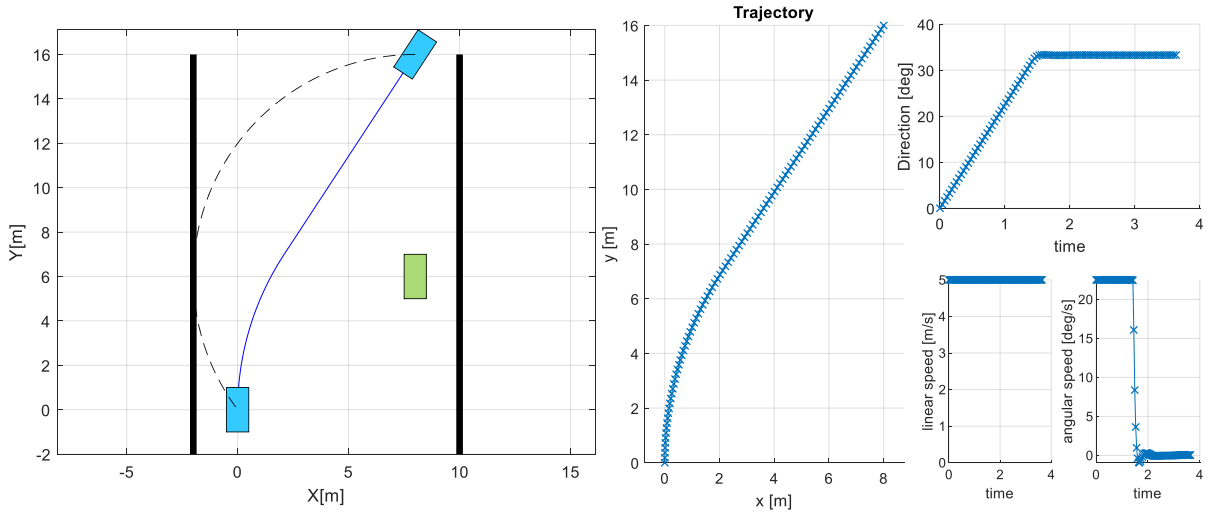


Figure 7.3: Simulation's (a) trajectory, constraints (left), state and control input values over time (right).

b. Straight tunnel with one ellipse obstacle

Same constant values as simulation (a) (Table 7.3), additional ellipse obstacle (Eq. (60)).

Table 7.4: Constant obstacle values for simulation (b).

(C_x,C_y)	a	b	φ
(4 [m], 8 [m])	3 [m]	1 [m]	$\pi/6$ [rad]

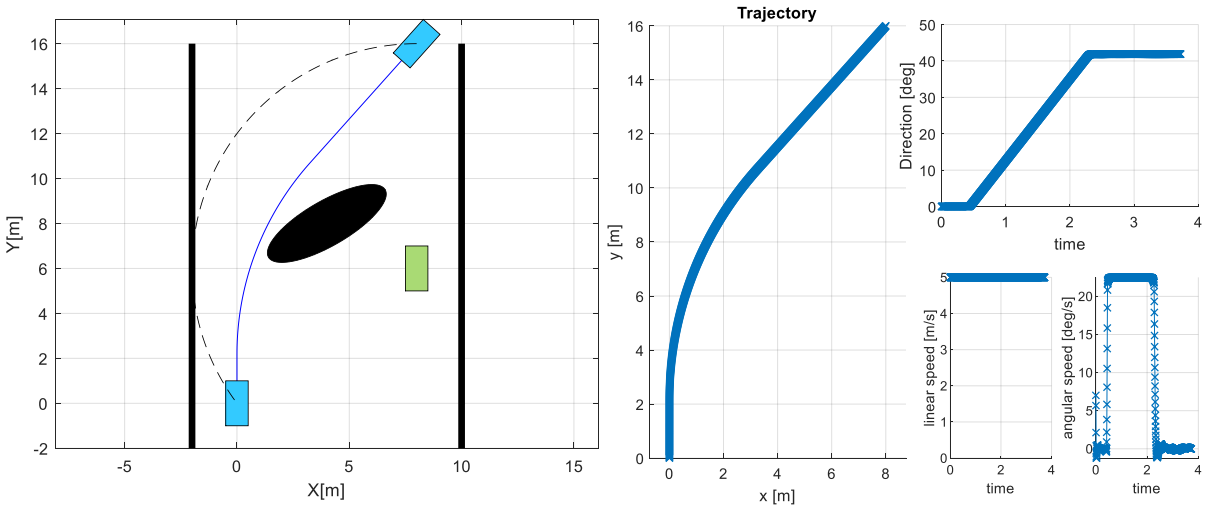


Figure 7.4: Simulation's (b) trajectory, constraints (left), state and control input values over time (right).

c. Corner no obstacles

In this case, since the borders of the tunnel are not linear, they cannot be defined in the form of Eq. (55)). Alternatively, a rectangular obstacle (square in this specific case) is used to define the corner constraint (Eq. (61)).

Table 7.5: Constant values for simulation (c).

(x_f,y_f)	x_{lb}	x_{ub}	y_{lb}	y_{ub}	(R_x,R_y)	p	q
(16 [m], 12 [m])	-2 [m]	inf	0 [m]	20 [m]	(15 [m], 5 [m])	5 [m]	5 [m]

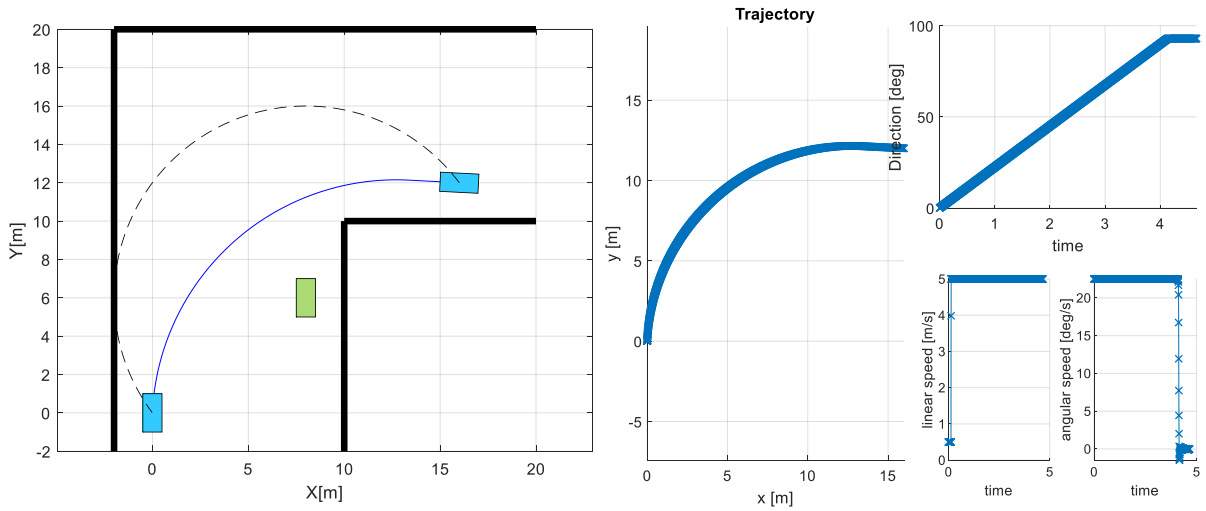


Figure 7.5: Simulation's (c) trajectory, constraints (left), state and control input values over time (right).

d. Corner and one ellipse obstacle

Same constant values as simulation (c) (Table 7.5), additional ellipse obstacle (Eq. (60)).

Table 7.6: Constant obstacle values for simulation (d).

(C_x,C_y)	a	b	φ
(5 [m], 8 [m])	3 [m]	1 [m]	$\pi/6$ [rad]

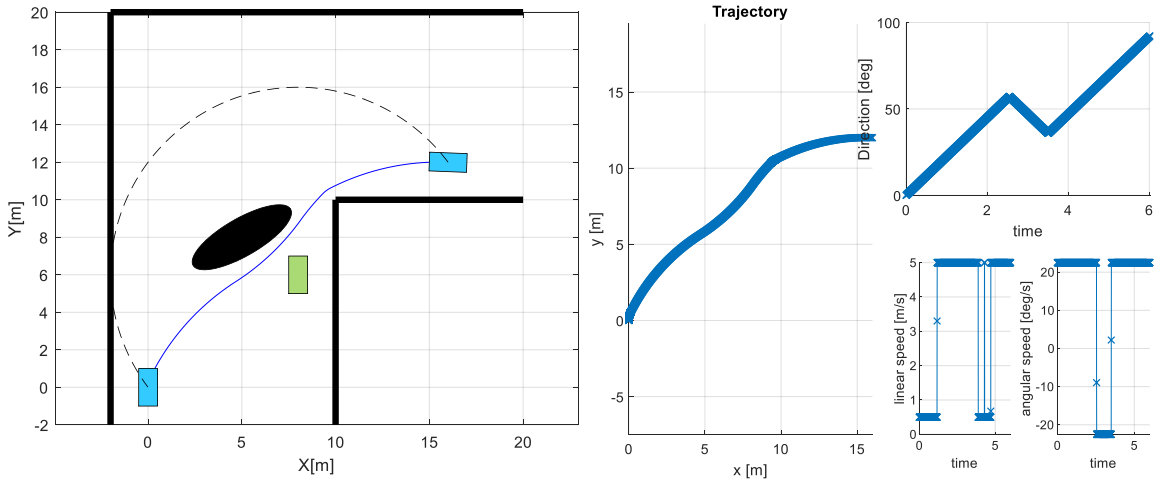


Figure 7.6: Simulation's (d) trajectory, constraints (left), state and control input values over time (right).

2. Multi-Step Simulation

Consider a long straight tunnel with many obstacles. The objective is that the robots autonomously advance in the tunnel in alternating steps while remaining in each other's range of sight and avoiding obstacles. Each step is an optimization problem solved as previously presented.

The considered tunnel is 30 meters long and 9 meters wide. The distance sensor's maximal range is 5 meters (r_{max}=5 m); hence a minimum of 6 steps are needed to cross the tunnel. The obstacles size, location and orientation are random within defined boundaries.

The flow chart of the process is described in Figure 7.7, the MATLAB codes are presented in Appendix [E] and Figure 7.8 presents the results of four successful simulations. Next, further details of each step in the process is presented.

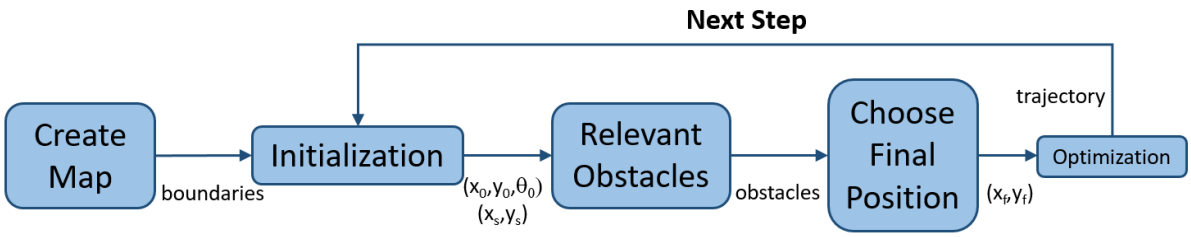


Figure 7.7: Multi-step algorithm flow chart.

a. Create Map – setting the boundaries of the tunnel.
b. Initialization – setting the initial positions of the robots for current step.
c. Relevant Obstacles – first, we tried running the optimization with multiple (5) obstacles and it crashed due to data overflow. Therefore, we decided to use for each step only the nearby obstacles, a reasonable simplification since not all obstacles are visible to the robots at all time. In practice, at each step a random obstacle is defined within the scope of the current step (distance of r_{max} from stationary vehicle); The optimization is performed under the previous and the new obstacles constraints.
d. Choose Final Position – The desired position for the traveling robot was determined as the point of most advancement in the y direction while maintaining the following conditions:
    - Distance from stationary robot (r) not larger than distance sensor's range, hence $r \leq r_{max}$.

- Not closer than ε from tunnel borders and obstacles. The condition of distance from obstacle is verified by Eq. (60).

e. Optimization – finding the optimal trajectory for the current step from (x_0,y_0) to (x_f,y_f) while avoiding obstacles using the FALCON.m tool.

Table 7.7: Constant values for all multi-step simulations.

x_{lb}	x_{ub}	y_{lb}	y_{ub}	r_{max}	ε	v_{max}	ω_{max}
-2 [m]	7 [m]	0 [m]	30 [m]	5 [m]	1 [m]	5 [m/sec]	π/8 [rad/sec]

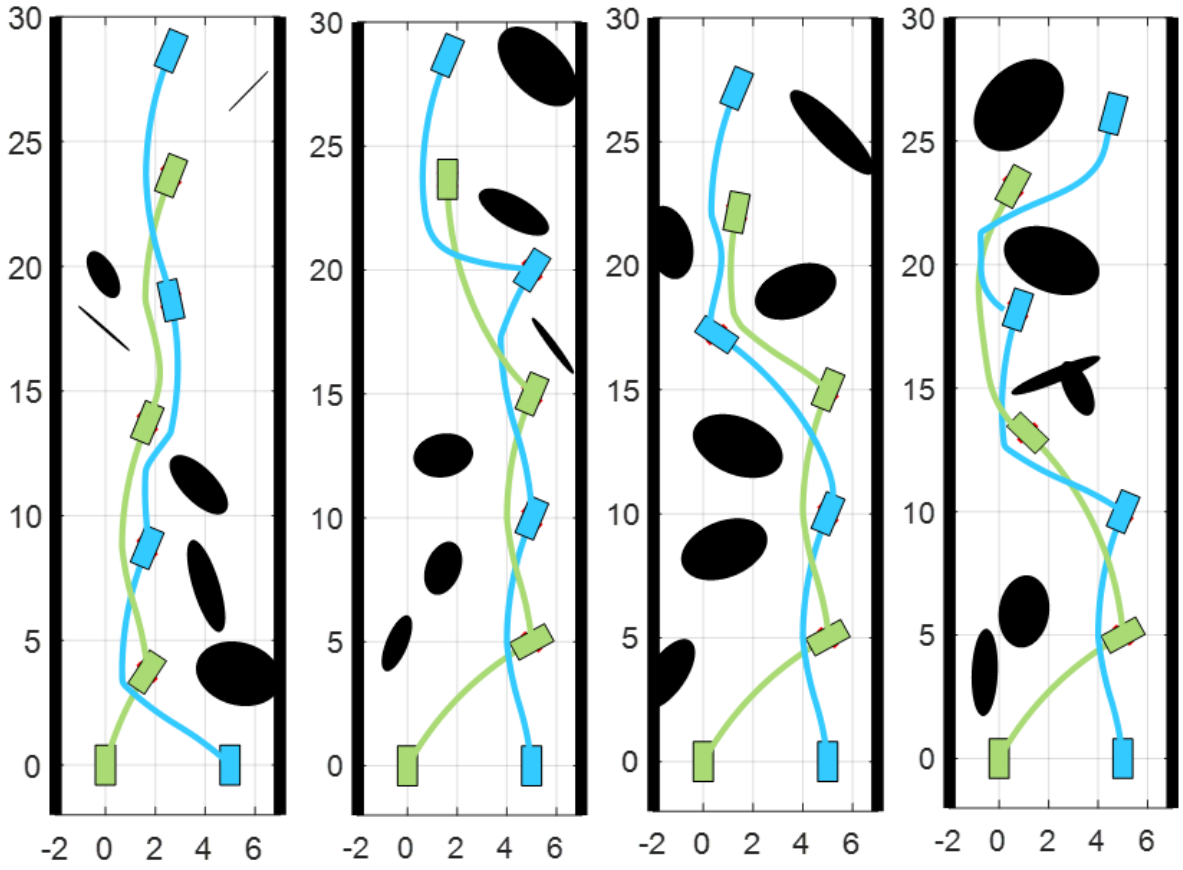


Figure 7.8: Four multi-step simulations' results with the same conditions and random obstacles.

7.2 Path Following with Polar Coordinates

This section presents a closed loop controller based on polar coordinates, designed to control the vehicles' movement along a desired path within each step. Polar coordinates were chosen due to the polar nature of the system's measurements – distance and bearing angle. The controller is designed for a differentially driven vehicle whose control inputs are (v,ω), defined respectively as the linear and angular velocities (see Figure 7.9).

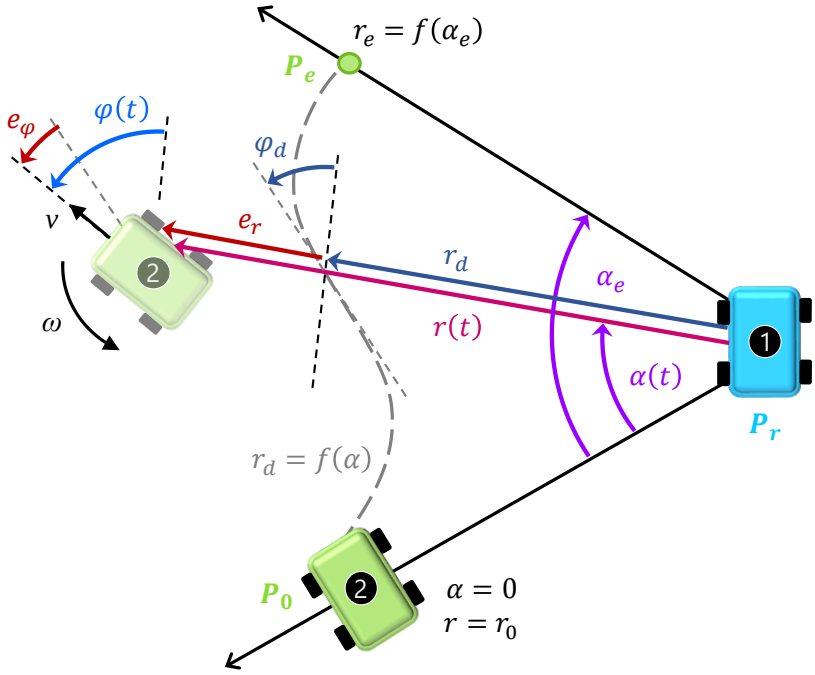


Figure 7.9: Differentially driven vehicle model in polar coordinates. P_r – stationary vehicle and polar coordinate system origin, P_0 – traveling vehicle's initial position, P_e – traveling vehicle's target point.

The kinematic model of the vehicle, using polar coordinates is:

$$\begin{aligned} \dot{\alpha} &= \frac{v}{r}\cos\varphi \\ \dot{r} &= v\sin\varphi \\ \dot{\varphi} &= \omega + \frac{v}{r}\cos\varphi \end{aligned} \quad , \tag{62}$$

where the state of the vehicle is defined by (α, r, φ). The variables α and r define the vehicle's location whereas φ defines its heading, measured from the perpendicular to the radius r (see Figure 7.9). The origin of the polar coordinate system is set to the position of the stationary vehicle P_r, and the initial position of the traveling vehicle P_0 is set to $\alpha=0$, $r=r_0$. The traveling vehicle's target point P_e is located at distance r_e from the origin and at angle α_e from its initial

position. The desired trajectory of the traveling vehicle (dashed grey line in Figure 7.9) is $r_d = f(\alpha)$.

The position error (in the direction of r) of the traveling vehicle with respect to the desired trajectory is:

$$e_r(t) = r(t) - r_d(\alpha) = r(t) - f(\alpha). \tag{63}$$

The time derivative of the position error is:

$$\dot{e}_r(t) = \dot{r} - f'(\alpha)\dot{\alpha} = \left(\sin\varphi - f'(\alpha)\frac{\cos\varphi}{r}\right)v \quad . \tag{64}$$

We define the heading error of the traveling vehicle with respect to the desired trajectory as:

$$e_\varphi(t) = \sin\varphi - f'(\alpha)\frac{\cos\varphi}{r}. \tag{65}$$

where $|r| > \varepsilon$ and $\varepsilon > 0$. This definition represents the heading error because if the vehicle is on the desired path, the desired advancement direction is:

$$f'(\alpha) = \frac{df(\alpha)}{d\alpha} = r\frac{\sin\varphi_d}{\cos\varphi_d}. \tag{66}$$

In other words, if $e_\varphi = 0$ then $\varphi(t) = \varphi_d$, the vehicle is in the desired direction. Deriving Eq. (65) by time yields:

$$\begin{aligned}
\dot{e}_\varphi &= \cos\varphi\dot{\varphi} - f''(\alpha)\dot{\alpha}\frac{\cos\varphi}{r} + f'(\alpha)\frac{\sin\varphi\dot{\varphi}r + \cos\varphi\dot{r}}{r^2} \\
&= -f''(\alpha)\frac{\cos\varphi}{r}\dot{\alpha} + f'(\alpha)\frac{\cos\varphi}{r^2}\dot{r} + \left(f'(\alpha)\frac{\sin\varphi}{r} + \cos\varphi\right)\dot{\varphi} \\
&= -f''(\alpha)\frac{\cos^2\varphi}{r^2}v + f'(\alpha)\frac{\cos\varphi\sin\varphi}{r^2}v + \left(f'(\alpha)\frac{\sin\varphi}{r} + \cos\varphi\right)\left(\omega + \frac{v}{r}\cos\varphi\right) \\
&= -f''(\alpha)\frac{\cos^2\varphi}{r^2}v + f'(\alpha)\frac{\cos\varphi\sin\varphi}{r^2}v + \left(f'(\alpha)\frac{\sin\varphi\cos\varphi}{r^2} + \frac{\cos^2\varphi}{r}\right)v + \left(f'(\alpha)\frac{\sin\varphi}{r} + \cos\varphi\right)\omega \\
&= \left(\frac{\cos^2\varphi}{r} + 2f'(\alpha)\frac{\cos\varphi\sin\varphi}{r^2} - f''(\alpha)\frac{\cos^2\varphi}{r^2}\right)v + \left(f'(\alpha)\frac{\sin\varphi}{r} + \cos\varphi\right)\omega
\end{aligned} \quad . \tag{67}$$

Similar to the design method presented in [55], by choosing the input of the angular velocity as:

$$\omega = \frac{r}{f'(\alpha)\sin\varphi + r\cos\varphi}\left(f''(\alpha)\frac{\cos^2\varphi}{r^2} - 2f'(\alpha)\frac{\cos\varphi\sin\varphi}{r^2} - \frac{\cos^2\varphi}{r} + u\right)v \quad , \tag{68}$$

with the signal $u(t)$ to be designed, the following **second order chain model** is achieved:

$$\begin{aligned}
\dot{e}_r &= e_\varphi v \\
\dot{e}_\varphi &= uv
\end{aligned} \quad . \tag{69}$$

The proper choice of $u(t)$ should compel the errors to converge to zero. Using a standard full state feedback method (2.4.1):

$$u(t) = -k_1 e_r - k_2 e_\varphi , \tag{70}$$

leads to the following closed loop system in a state space representation (2.4):

$$\begin{aligned} \dot{\bar{e}} &= \tilde{A}\bar{e}v \\ \bar{e} = \begin{bmatrix} e_r \\ e_\varphi \end{bmatrix} \quad , \quad \tilde{A} &= \begin{bmatrix} 0 & 1 \\ -k_1 & -k_2 \end{bmatrix}. \end{aligned} \tag{71}$$

The constants k_1,k_2 should be chosen so that the matrix $\tilde{A}$ is stable, under the assumption that $v>0$. Choosing a larger k_1 will cause faster convergence of the location but slower convergence of the heading angle, and vice versa. A stable matrix $\tilde{A}$ and $v>0$ can be shown to ensure stability of the closed loop by using the following Lyapunov function (2.5):

$$V = \bar{e}^T P \bar{e} \quad , \quad P > 0 , \tag{72}$$

which is a positive definite matrix. Its derivative is always negative:

$$\begin{aligned} \dot{V} &= \dot{\bar{e}}^T P\bar{e} + \bar{e}^T P\dot{\bar{e}} = v\bar{e}^T \tilde{A}^T P\bar{e} + \bar{e}^T P\tilde{A}\bar{e}v \\ &= \bar{e}^T (\tilde{A}^T P + P\tilde{A})\bar{e}v = -Qv < 0 \end{aligned} . \tag{73}$$

Alternatively, the system can also be expressed using the path variable *'p'* instead of the time *'t'*:

$$\begin{bmatrix} \dot{e}_r \\ \dot{e}_\varphi \end{bmatrix} = \begin{bmatrix} de_r/dt \\ de_\varphi/dt \end{bmatrix} = \left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} e_r \\ e_\varphi \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \right) \frac{dp}{dt} . \tag{74}$$

By multiplying both sides of the equation by *dt/dp*, we obtain:

$$\begin{aligned} \bar{e}\,' &= (A - BK)\bar{e} \\ A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \; B &= \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \; K = \begin{bmatrix} k_1 & k_2 \end{bmatrix}, \; u = -K\bar{e} \end{aligned} , \tag{75}$$

and the state variable derivatives are by the path variable *'p'*.

The standard full state feedback structure of the dynamic system in state space representation allows for the use of standard methods to determine the control values [k_1,k_2], such as pole placement or linear-quadratic regulator (LQR). It should be noted that performance is obtained relative to the path variable *'p'* instead of the time *'t'*.

After planning appropriate control variables k_1 and k_2, the angular velocity that should be applied:

$$\omega = \frac{r}{f'(\alpha)\sin\varphi + r\cos\varphi}\left(f''(\alpha)\frac{\cos^2\varphi}{r^2} - 2f'(\alpha)\frac{\cos\varphi\sin\varphi}{r^2} - \frac{\cos^2\varphi}{r} - k_1 e_1 - k_2 e_2 \right) v \,. \tag{76}$$

It could be seen that the input signal reaches singularity when:

$$f'(\alpha)\sin\varphi + r\cos\varphi = 0 \,. \tag{77}$$

To understand the physical meaning of this phrase, it could be written as follows:

$$f'(\alpha)\sin\varphi + r\cos\varphi = r\frac{\sin\varphi_r}{\cos\varphi_r}\sin\varphi + r\cos\varphi = r\frac{\cos(\varphi_r - \varphi)}{\cos\varphi_r} = 0 \,, \tag{78}$$

hence, this singularity accrues when the angle between the heading of the vehicle and the desired path is $\pm\pi/2$. In this scenario, the direction of the required angular velocity is undefined since the convergence towards the desired path can be reached by turning left or right equally.

7.2.1 Straight Line Path Following and Convergence

At this point, the assumption is that at each step, the vehicle is given a target point. If there are no obstacles in the explored area, an optional solution is to advance in a straight line to the target point; this section presents the implementation of the path following algorithm presented above while the desired path is a straight line.

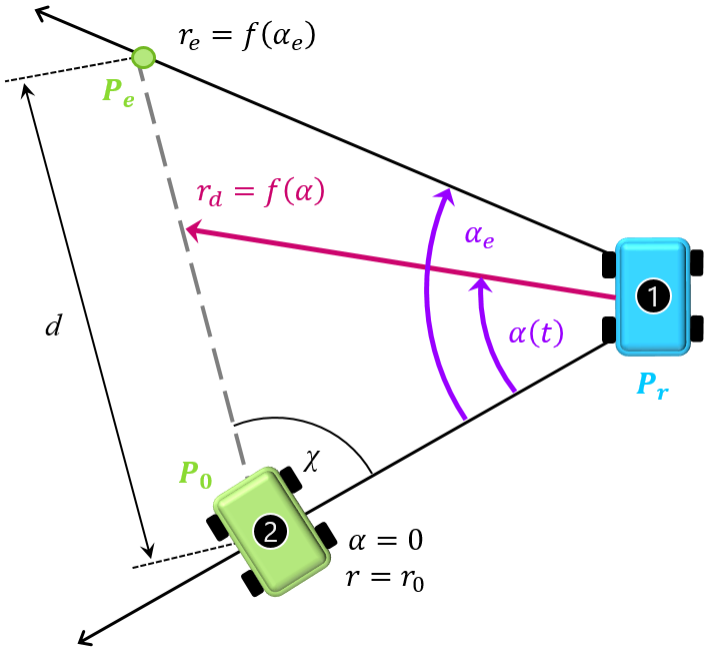


Figure 7.10: Straight Line advancement strategy. P_r is the stationary observing vehicle's position, P_0 is the moving vehicle's initial position and P_e is the moving vehicle's target point.

The measurements of the moving vehicle are obtained with respect to point P_r, according to polar coordinates; therefore, the desired path will be represented according to a polar frame

which origin is at P_r and the angle $\alpha = 0$ represents the moving vehicle's initial position, as seen in Figure 7.10. The given data of the problem are r_0, r_e, α_e, where r_0 describes the initial position and r_e, α_e describe the final desired position. The desired path is a straight line, but it needs to be described in polar coordinates i.e. $r_d = f(\alpha)$.

From the given data, the length of the desired path *d* can be calculated using the law of cosines:

$$d = \sqrt{r_0^2 + r_e^2 - 2 r_0 r_e \cos \alpha_e} \,. \tag{79}$$

The angle χ (see Figure 7.10) can be calculated using the law of sines:

$$\frac{r_e}{\sin \chi} = \frac{d}{\sin \alpha_e} \quad \rightarrow \quad \chi = \arcsin \frac{r_e \sin \alpha_e}{d} \,. \tag{80}$$

The law of sines can be used again to calculate the desired path:

$$\frac{r_d}{\sin \delta} = \frac{r_0}{\sin(180 - \alpha - \chi)} \quad \rightarrow \quad r_d = f(\alpha) = \frac{r_0 \sin \chi}{\sin(\alpha + \chi)} = r_0 \sin \chi \csc(\alpha + \chi) \,. \tag{81}$$

In order to implement the previously presented controller, the first and second derivatives of $f(\theta)$ are needed:

$$\begin{aligned} f'(\alpha) &= -r_0 \sin \chi \csc(\alpha + \chi) \cot(\alpha + \chi) \\ f''(\alpha) &= r_0 \sin \chi \csc(\alpha + \chi) \left(\cot^2(\alpha + \chi) + \csc^2(\alpha + \chi) \right) \end{aligned} \,. \tag{82}$$

7.2.2 Simulation

Our control method was simulated using MATLAB Simulink program with a control rate of 10^4 loops per meter, see Appendix [D]. The values of the variables used in the simulation are presented in Table 7.8. If the maximum desired overshoot is 10% (2.6), then the damping coefficient is:

$$M_p = \exp\left(-\frac{\pi \zeta}{\sqrt{1 - \zeta^2}} \right) = 0.1 \;\rightarrow\; \zeta \approx 0.6 \,. \tag{83}$$

Demanding convergence after half of the path with a tolerance of 5%:

$$P_s(5\%) = \frac{-\ln(0.05)}{\zeta \omega_n} = \frac{1}{2} d \;\rightarrow\; \omega_n = \frac{10}{d} \,. \tag{84}$$

The characteristic equation (2.6) of the system (Eq. (75)) is:

$$\Delta(s) = s^2 + k_2 s + k_1 . \tag{85}$$

By comparing to the standard characteristic equation (Eq. (10)), the control values should be:

$$k_1 = \frac{100}{d^2} \ , \ \ k_2 = \frac{12}{d} \ . \tag{86}$$

The simulation results are presented in Figure 7.11. Figure 7.11 (left) shows that in the first case, the robot accurately followed the trajectory with a very small error which was 4 orders smaller than the length of the trajectory. In the second case (Figure 7.11, right), where the robot was initially positioned with an incorrect heading of nearly 90 degrees, the solution converged to the desired trajectory (5% angular error), within half of the trajectory and the overshoot was smaller than 10% as desired.

Table 7.8: Simulation Variables

Variables	(x_r,y_r)	(x_0,y_0)	(r_e,α_e)	v
Values	(0,0)	(-15,0)	(21.2,45°)	1 [m/sec]

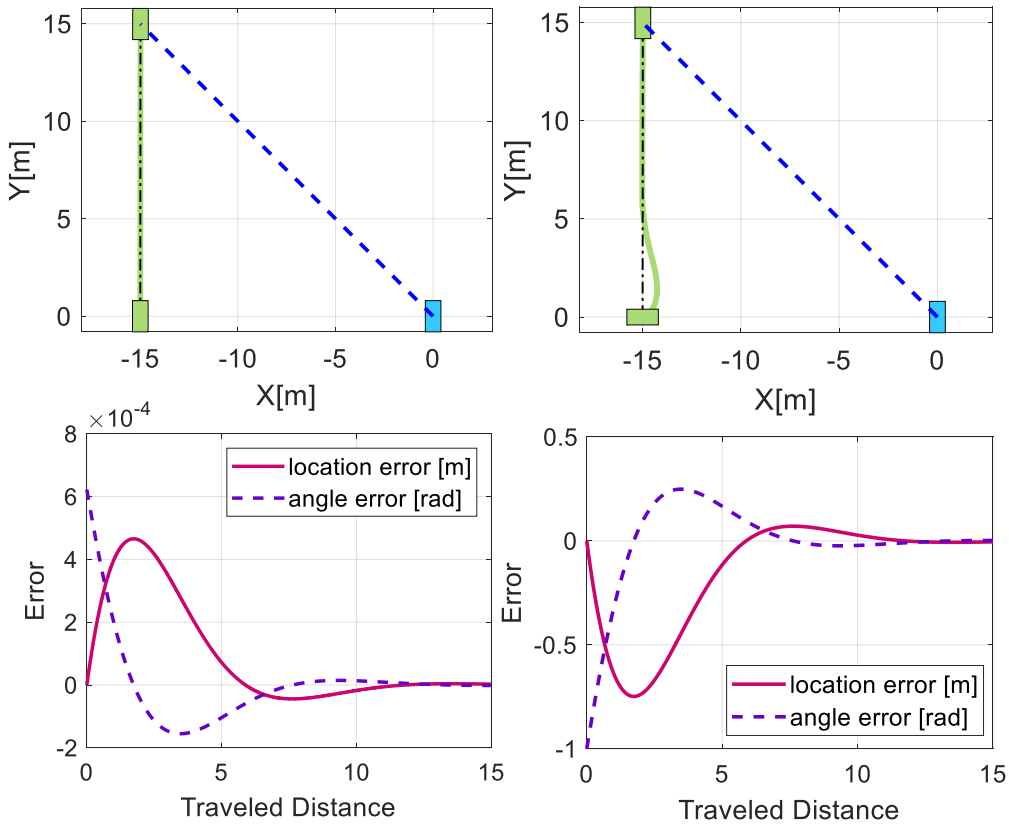


Figure 7.11: Simulation results. Top: the traveling robot's path (green solid line), while the observing robot measures relative location and orientation (blue dashed line). Bottom: corresponding location and angle errors relative to the desired path. Left: initial orientation of 0°, right: initial orientation of -90°+ε.

7.3 Path Planning and Following

This section presents the fusion and implementation of the two presented algorithms. In Section 7.1, we found the optimal trajectory for each step in a constrained environment. The outputs of the algorithm are a set of (x_d,y_d,θ_d) points representing the robot's optimal trajectory and the required open-loop control inputs (v_d,ω_d). Given continues measurements of the distance and bearing with respect to the stationary robot, the presented closed-loop controller (7.2) could be used to follow the optimal path, giving the system the ability to self-adjust depending on the outputs and measurements (rather than implementing the path planning algorithm directly as an open-loop controller). It should be mentioned, that this implementation requires that the robots stay in each other's range of 'sight' and any obstacle between them must be adequately low.

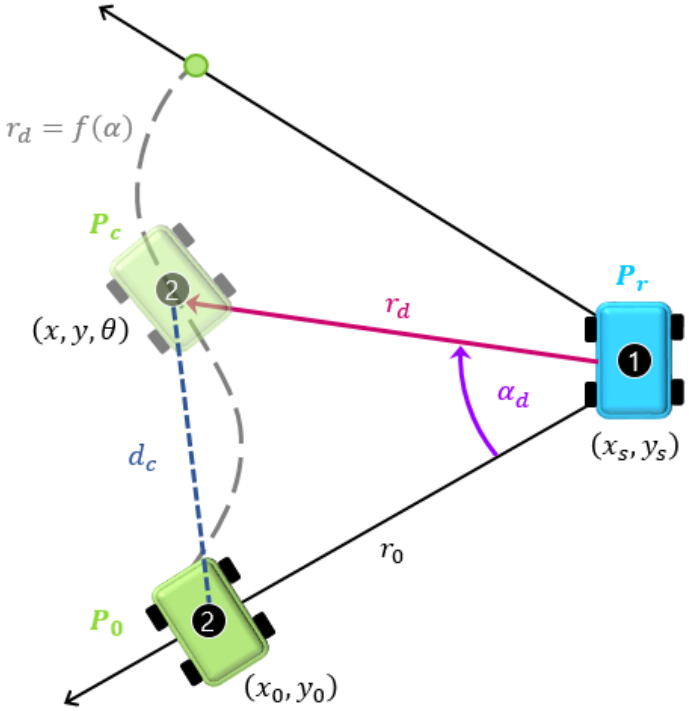


Figure 7.12: Path following in polar coordinates. P_r is the stationary observing vehicle's position, P_0 is the moving vehicle's initial position and P_c is the moving vehicle's current position.

In order to implement the closed-loop controller presented in Section 7.2, the desired path must be represented as $r_d = f(\alpha)$, while the origin of the polar coordinate system is set to the position of the stationary vehicle (x_s,y_s). Representing the optimal trajectory (x_d,y_d) in polar coordinates (r_d,α_d):

$$r_d = \sqrt{\left(x_d - x_s\right)^2 + \left(y_d - y_s\right)^2}\,, \tag{87}$$

and the angle is derived from the law of cosines:

$$\alpha_d = \text{acos}\left(\frac{r_d^{\,2} + r_0^{\,2} - d_c^{\,2}}{2 r_d r_0}\right). \tag{88}$$

In Eq. (88), r_0 is the initial distance of the moving vehicle:

$$r_0 = \sqrt{\left(x_0 - x_s\right)^2 + \left(y_0 - y_s\right)^2}, \tag{89}$$

and d_c is the current distance of the moving vehicle from its initial position:

$$d_c = \sqrt{\left(x_d - x_0\right)^2 + \left(y_d - y_0\right)^2}. \tag{90}$$

Notice that we use only the location (x_d,y_d) and not the orientation θ_d obtained by the optimal path algorithm.

In order to implement the closed-loop controller, the desired trajectory must be described as a continues function rather than a set of points. We used a 4-th order polynomial curve fitting:

$$r_d\left(\alpha\right) = p_1\alpha^4 + p_2\alpha^3 + p_3\alpha^2 + p_4\alpha + p_5, \tag{91}$$

making the first and second derivatives:

$$r_d\,'\left(\alpha\right) = 4p_1\alpha^3 + 3p_2\alpha^2 + 2p_3\alpha + p_4, \tag{92}$$

$$r_d\,''\left(\alpha\right) = 12p_1\alpha^2 + 6p_2\alpha + 2p_3. \tag{93}$$

7.3.1 Simulation

We used the closed-loop polar coordinate controller on the optimal trajectories obtained from the four single step scenarios presented in Section 7.1.4 (1). The linear velocity was set to be the constant maximal velocity v=5 m/sec. We used the control values designed in Section 7.2.2 (Eq. (86)), which should result in maximum overshoot of 10% and convergence after half of the path with a tolerance of 5%.

The following figures present the actual trajectory (solid green line) as well as the desired optimal trajectory (black dotted line). Additionally, location and angle errors relative to the desired path are presented, as well as control input values.

a. Straight tunnel no obstacle

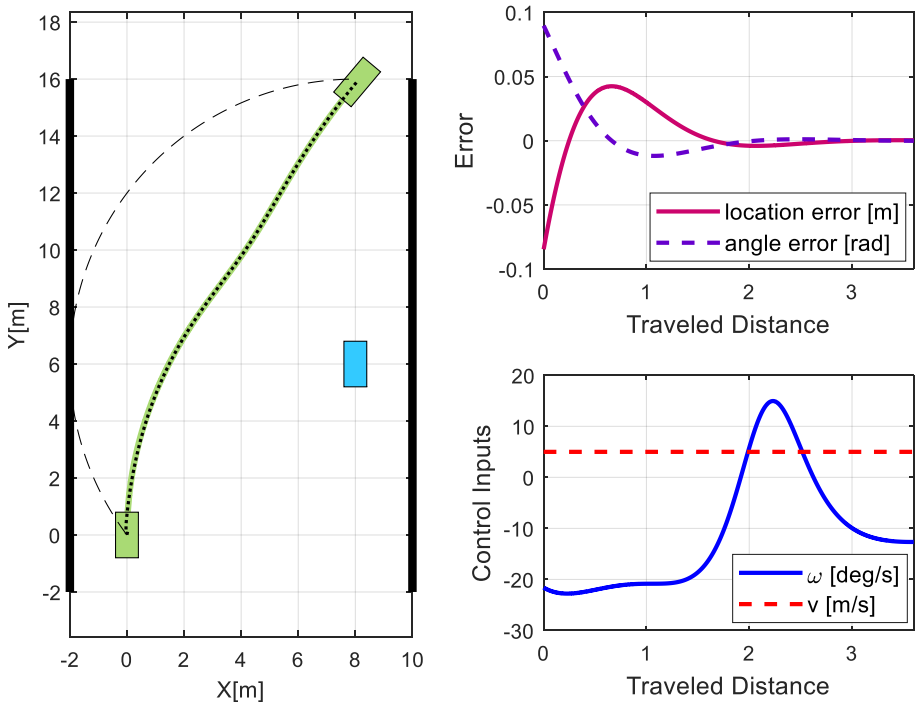


Figure 7.13: Simulation's (a) trajectory and constraints (left), location and angle errors relative to the desired path (top right), and control input values (bottom right).

b. Straight tunnel with one ellipse obstacle

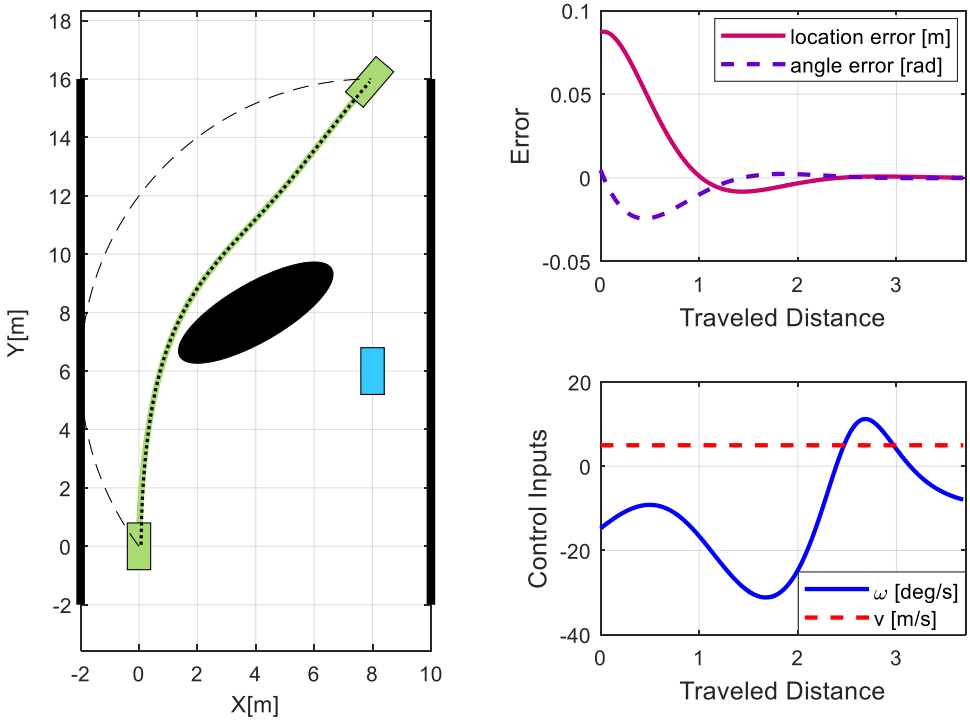


Figure 7.14: Simulation's (b) trajectory and constraints (left), location and angle errors relative to the desired path (top right), and control input values (bottom right).

c. Corner no obstacles

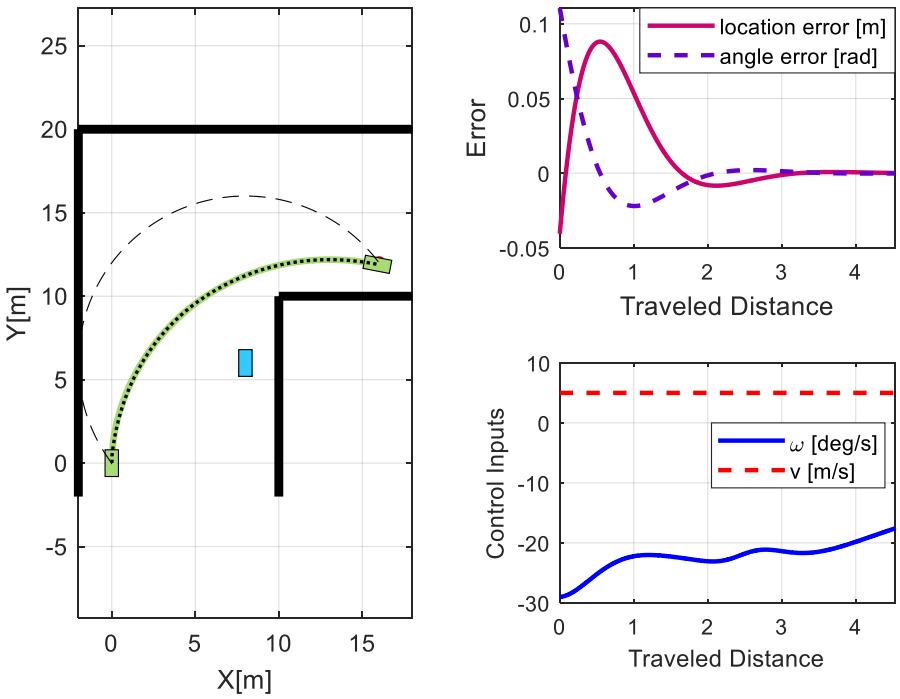


Figure 7.15: Simulation's (c) trajectory and constraints (left), location and angle errors relative to the desired path (top right), and control input values (bottom right).

d. Corner and one ellipse obstacle

In the last case the optimal trajectory was not as well fitted by a 4-th order polynomial function, thus a 6-th order polynomial was used and the derivatives were changed respectively.

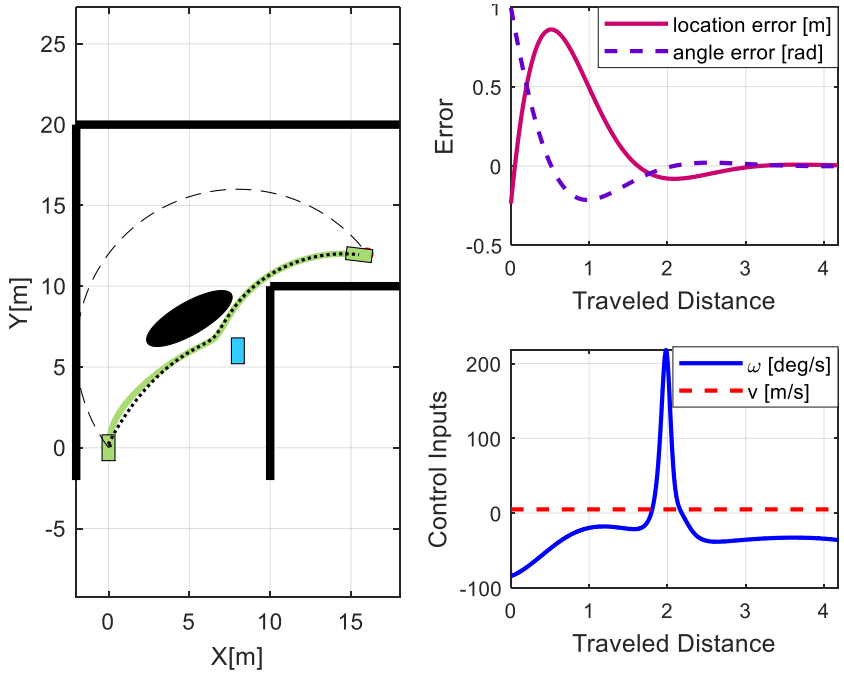


Figure 7.16: Simulation's (d) trajectory and constraints (left), location and angle errors relative to the desired path (top right), and control input values (bottom right).

As seen in Figure 7.13, Figure 7.14 and Figure 7.15, the robot successfully followed the desired path with very low location and angle errors for simulations (a)-(c). The top right figures show that the solution converged to the desired trajectory (5% error), with a settling distance that is under half of the trajectory and the overshoot was smaller than 10% as desired. Simulation (d) was also successful, though required very high angular velocity (Figure 7.16 bottom right), since the closed-loop controller is designed for geometric following of a path and not a trajectory, i.e. velocity constraints are not considered.

Comparing the required control signals in both methods, the closed-loop resulted in smoother angular velocity as opposed to the open-loop (see Figure 7.3, Figure 7.4, Figure 7.5 and Figure 7.6 bottom right), which resulted in abruptly switching control signals between the upper and lower bounds, resembling the 'bang-bang' control method [57]. On the other hand, as opposed to the closed-loop solution, in the path planning algorithm the control inputs are constrained; Figure 7.16 bottom right shows that not constraining the angular velocity could result in high and possibly not practicable values. It should be reminded, that the linear velocity was set to a constant 5 m/sec and the angular velocity depends directly on the linear velocity (Eq. (76)). Therefore, the problem of high angular velocity values could be addressed by lowering the linear velocity appropriately once the angular velocity reaches its upper boundary.

This method has the clear advantage of a closed-loop controller as opposed to an open-loop controller, giving the system the ability to overcome disturbances. Though it should be reminded, that this method requires continues measuring throughout each step, or partial measurement combined with estimation; for example, measuring the position of the moving vehicle by the static vehicle and estimating its orientation, meaning only the static vehicle performs measurements. Implementing the open loop algorithm on its own (Section 7.1), requires that measurements will be obtained only at the end of each step.

8 Conclusions

In this paper we presented a simple, low cost method for precise multi-robot self-localization that relies on distance and bearing measurements. The system can be deployed in indoor areas where GPS signals are unavailable, and visibility is relatively low. The key advantage of this method is that it reduces the errors resulting from the inaccuracies of evaluating the orientation of the robots. We developed an analytical solution for the position of the robots and a numerical simulation to account for the statistical sensors' errors. We show that the total relative error (cumulative error divided by travelled distance) is on the same order of magnitude as the sensors' relative errors (error divided by distance), and that the angular error has a larger impact on the location errors than the distance error, thus making it important to use a relatively accurate bearing sensor.

Given that the sensor measurement contains statistical errors, we ran a Monte Carlo Simulation (MCS) and determined the spatial distribution of the measured/estimated location of the robot with the given sensors' random errors (10,000 simulations for each case). To reduce the MCS computation time, we developed an approximated error evaluation method based on first order linear approximation. This method was 200 times faster than the direct method.

We then used the MCS to compare between different paths and advancing methods. We found that the chosen path governed the size of the location error, whereas the different advancing methods had little influence on the total error. For a given equal number of steps and total travelled distance, the smallest error is in the square path, followed by the 'S' shaped path and the largest error is with the straight path. Overall, using our localization algorithm, it is best to increase the size of the steps and decrease their number in order to reduce the bearing errors and increase the accuracy of the localization.

Next we present a two-robot system used to further validate our algorithm by real-world experiments. We performed experiments in four different paths, calculated the standard deviation and mean error values and compared the results to the Monte Carlo simulation. The results show that the method is very accurate with errors of about 1-3% of the total distance traveled.

Finally, we developed a path planning algorithm and a closed-loop path following controller, allowing the two robots to autonomously advance in an uncharted constrained area. In the path planning algorithm cartesian coordinates are used due to simpler representation of the path and obstacle constraints whereas in the path following algorithm polar coordinates are

used due to the polar nature of the system's measurements. The path planning algorithm finds the optimal (shortest) trajectory at each step while avoiding obstacles and remaining visible to the stationary robot. Following the optimal path is obtained by using a closed-loop controller; we found that our path following algorithm quickly converges to the desired path even when the initial error is large.

Besides its advantages, the method presented in this research does require a line of sight between the two cooperating robots and that one of the robots must remain fully static during each step. Another limitation is that the method is currently limited to 2D localization. However, we expect that it can be generalized to 3D problems by adding another relative bearing measurement between the robots. Additionally, the presented algorithm can be further developed to a multi-robot system (three or more robots), enabling to reduce the cumulative error by a proper estimation algorithm.

As to the control algorithm, we believe the presented path planning algorithm can be further developed to fit more complex constraints. For example, more complex shaped obstacles can be approximated as a combination of the presented ellipse and rectangle obstacles, or curvy tunnel boundaries could be defined by ellipse obstacles, similar to the shown case of a rectangle obstacle used to define a straight corner. Additionally, the visibility by the static vehicle constraint could be formulated so that obstacles are also taken into consideration, ensuring the two robots always remain in each other's range of sight, even in the presents of high obstacles.

9 References

[1] R. Kurazume, S. Nagata and S. Hirose, "Cooperative positioning with multiple robots", *IEEE International Conference on Robotics and Automation*, pp. 1250–1257, 1994.

[2] R. Kurazume, S. Nagata and S. Hirose, "Study on cooperative positioning system", *IEEE International Conference on Robotics and Automation*, pp. 1421-1426, 1996.

[3] I. M. Rekleitis, G. Dudek and E. E. Milios, "Multi-robot exploration of an unknown environment, efficiently reducing the odometry error*", International Joint Conference in Artificial Intelligent (IJCAI)*, Vol. 2, pp. 1340–1345. 1997.

[4] I. M. Rekleitis, G. Dudek and E. E. Milios, "Experiments in free-space triangulation using cooperative localization", *IEEE/RSJ International Conference Intelligent Robotic Systems*, 2003.

[5] A. J. Davison and N. Kita, "Active visual localization for cooperating inspection robots", *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1709–1715, 2000.

[6] M. Moors, F. Schneider and D. Wildermuth, "Relative position estimation in a group of robots", *International Conference on Climbing and Walking Robots*, 2003.

[7] G. Dudek and N. Roy, "Multi-robot rendezvous in unknown environments, or, what to do when you're lost at the zoo," *In Proceedings of the AAAI National Conference Workshop on Online Search*, pp. 22–29, 1997.

[8] Roy N. and Dudek G., "Collaborative robot exploration and rendezvous: Algorithms, performance bounds and observations", *Autonomous Robots*, vol. 11, pp 117–136, 2001.

[9] Zhou X. S. and Roumeliotis S. I., "Multi-robot SLAM with unknown initial correspondence: The robot rendezvous case", *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.

[10] Howard A., "Multi-robot simultaneous localization and mapping using particle filters", *SAGE journals*, vol. 25, pp. 1243-1256, 2006.

[11] Birk A. and Carpin S., "Merging occupancy grid maps from multiple robots", *IEEE*, vol. 94, pp. 1384 – 1397, 2006.

[12] Burgard W., Moors M., Fox D., Simmons R. and Thrun S., "Collaborative multi-robot exploration", *IEEE International Conference on Robotics and Automation*, 2000.

[13] M. Todescato, A. Carron, R. Carli, A. Franchi, and L. Schenato, "Multi-robot localization via GPS and relative measurements in the presence of asynchronous and lossy communication," *IEEE European Control Conference*, pp. 2527–2532, 2016.

[14] A. T. Rashid, M. Frasca, A. A. Ali, A. Rizzo, and L. Fortuna, "Multi-robot localization and orientation estimation using robotic cluster matching algorithm," *Robotics and Autonomous Systems*, vol. 63, pp. 108–121, 2015.

[15] A. Martinelli, F. Pont and R. Siegwart, "Multi-robot localization using relative observations", *IEEE International Conference on Robotics and Automation*, pp. 2797-2802, 2005.

[16] O. De Silva, G. K. Mann, and R. G. Gosine, "Efficient distributed multi-robot localization: A target tracking inspired design," *IEEE International Conference on Robotics and Automation*, pp. 434–439. 2015.

[17] C. Lin, Z. Lin, R. Zheng, G. Yan, and G. Mao, "Distributed source localization of multi-agent systems with bearing angle measurements", *IEEE Transactions on Automatic Control*, vol. 61, no. 4, pp. 1105–1110, 2016.

[18] L. Luft, T. Schubert, S. Roumeliotis, and W. Burgard, "Recursive decentralized localization for multi-robot systems with asynchronous pairwise communication," *International Journal of Robotic Research*, 2018.

[19] J. Liu, J. Pu, L. Sun and Y. Zhang, "Multi-robot cooperative localization with range-only measurement by UWB", *IEEE Chinese Automation Conference*, 2018.

[20] C. Pierre, R. Chapuis, R. Aufrere, J. Laneurit, and C. Debain, “Range-only based cooperative localization for mobile robots,” *IEEE International Conference on Information Fusion,* pp. 1933–1939, 2018.
[21] Y. Cao, M. Li, I. Vogor, S. Wei and G. Beltrame, “Dynamic range-only localization for multi-robot systems”, *IEEE Access,* vol. 6, 2018.
[22] S. Se, D. Lowe, and J. Little, “Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks”, *International Journal of Robotic Research,* vol. 21, no. 8, pp. 735–758, 2002.
[23] A. Gil, O. Reinoso, M. Ballesta and M. Julia, “Multi-robot visual SLAM using a Rao-Blackwellized particle filter”, *Robotics and Automation Systems,* vol. 58, no. 1, pp. 68-80, 2010.
[24] J. Fuentes-Pacheco, J. Ruiz-Ascencio and J. M. Rendón-Mancha, “Visual simultaneous localization and mapping: a survey”, *Artificial Intelligent Review,* vol. 43, pp. 55-81, 2015.
[25] M. Saska et al., “System for deployment of groups of unmanned micro aerial vehicles in GPS-denied environments using onboard visual relative localization,” *Autonomous Robots,* vol. 41, no. 4, pp. 919–944, 2017.
[26] C. Alejandro and R. Nagpal, “Distributed range-based relative localization of robot swarms”, *Algorithmic Foundations of Robotics XI,* Springer International Publishing, pp. 91-107, 2015.
[27] A. Prorok, A. Bahr, and A. Martinoli, “Low-cost collaborative localization for large-scale multi-robot systems,” *IEEE International Conference on Robotics and Automation,* pp. 4236–4241, 2012.
[28] X. Zhou and S. Roumeliotis, “Determining the robot-to-robot 3D relative pose using combinations of range and bearing measurements (Part II)”, *IEEE International Conference on Robotics and Automation,* pp. 4736–4743, 2011.
[29] X. Zhou and S. Roumeliotis, “Determining 3D relative transformations for any combination of range and bearing measurements,” *IEEE Transactions on Robotics*, vol. 29, no. 2, pp. 458 – 474, 2013.
[30] S.T. Pfister, K.L. Kriechbaum, S.I. Roumeliotis and J.W. Burdick, "Weighted range sensor matching algorithms for mobile robot displacement estimation", *IEEE International Conference on Robotics and Automation*, 2002.
[31] S. M. LaValle, "Planning algorithms", *Cambridge University Press*, 2006.
[32] S. M. LaValle, "Motion planning", *IEEE Robotics Automation Magazine*, vol. 18, no. 2, pp. 108-118, 2011.
[33] C. E. Garcia, D. M. Prett, and M. Morari, "Model predictive control: Theory and practice - a survey", *Automatica*, vol. 25, no. 3, pp. 335-348, 1989.
[34] M. Morari and J.H. Lee, “Model predictive control: Past, present and future,” *Computers and Chemical Engineering,* vol. 23, no. 4-5, pp. 667-682, 1999.
[35] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert, “Constrained model predictive control: Optimality and stability,” *Automatica,* vol. 36, no. 6, pp. 789–814, 2000.
[36] F. Borrelli, A. Bemporad, and M. Morari, “Predictive control for linear and hybrid systems”, [Online], Available: http://www.mpc.berkeley.edu/mpc-course-material, 2014.
[37] E. Allgower and A. Zheng (eds), "Nonlinear model predictive control", vol. 26 of Progress in Systems and Control Theory, *Birkhauser,* 2000.
[38] D.H. Shim, H.J. Kim and S. Sastry, “Decentralized nonlinear model predictive control of multiple flying robots”, *IEEE International Conference on Decision and Control*, 2003.
[39] H. N. Nguyen, "Constrained control of uncertain, time-varying, discrete-time systems: an interpolation-based approach", Lecture Notes in Control and Information Sciences 451, *Springer*, 2014.

[40] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning", *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846-894, 2011.
[41] M. Elbanhawi and M. Simic, "Sampling-based robot motion planning: A review", *IEEE Access*, vol. 2, pp. 56-77, 2014.
[42] E. Schmerling, L. Janson and M. Pavone, "Optimal sampling-based motion planning under differential constraints: the driftless case", *IEEE International Conference on Robotics and Automation*, pp. 2368-2375, 2015.
[43] M. Rieck, M. Bittner, B. Gruter, J. Diepolder, and P. Piprek, "Falcon.m user guide" [Online], Available: www.falcon-m.com, 2019.
[44] A. D. Luca, G. Oriolo, and M. Vendittelli, "Control of wheeled mobile robots: An experimental overview," In: S. Nicosia, B. Siciliano, A. Bicchi, P. Valigi (eds) Ramsete, Lecture Notes in Control and Information Sciences, *Springer*, vol. 270, pp. 181–223, 2001.
[45] D. K. Chwa, "Sliding-mode tracking control of nonholonomic wheeled mobile robots in polar coordinates," *IEEE Transactions in Control Systems Technology*, vol. 12, no. 4, pp. 637–644, 2004.
[46] J. Djugash, S. Singh and B. P. Grocholsky, "Modeling mobile robot motion with polar representations", *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.
[47] J. Park and B. Kuipers, "A smooth control law for graceful motion of differential wheeled mobile robots in 2D environment," *IEEE International Conference on Robotics and Automation*, pp. 4896–4902, 2011.
[48] J. Cornejo, J. Magallanes, E. Denegri and R. Canahuire, "Trajectory tracking control of a differential wheeled mobile robot: A polar coordinates control and LQR comparison," *IEEE International Conference Electronics, Electrical Engineering and Computing*, no. 2, pp. 1–4, 2018.
[49] M. Hazewinkel, ed., "Jacobian", Encyclopedia of Mathematics, *Springer Science+Business Media B.V. / Kluwer Academic Publishers*, 2001.
[50] B. S. Everitt, "The Cambridge dictionary of statistics", Cambridge (3rd edition), *Cambridge University Press*, 2006.
[51] W. Hahn, "Theory and application of Liapunov's direct method", Englewood Cliffs, *NJ: Prentice-Hall*, 1963.
[52] A. M. Lyapunov, "The general problem of the stability of motion" (In Russian), *Doctoral dissertation, Univ. Kharkov*, 1892, English translations: (1) "Stability of Motion", Academic Press, New-York & London, 1966 (2) "The General Problem of the Stability of Motion", (A. T. Fuller trans.) Taylor & Francis, London 1992.
[53] G. B. Dantzig, "The nature of mathematical programming", Archived at the Wayback Machine, Mathematical Programming Glossary, *INFORMS Computing Society*, 2014.
[54] D. Z. Du, P. M. Pardalos, W. Wu, "History of optimization", In C. Floudas, P. Pardalos, (eds.), Encyclopedia of Optimization, *Boston: Springer*, pp. 1538–1542, 2008.
[55] S. A. Arogeti and N. Berman, "Path following of autonomous ground vehicles in the presence of sliding effects," *IEEE Transactions on Vehicular Technology*, vol. 61, no. 4, pp. 1481–1492, 2012.
[56] A. Wächter and L.T. Biegler, "On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming", *Mathematical Programming*, vol. 106, no. 1, pp. 25-57, 2006.
[57] R. Bellman, I. Glicksberg and O. Gross, "On the 'bang-bang' control problem," *Quarterly of Applied Mathematics*, vol. 14, pp. 11-18, 1956.

10 Appendices

[A] ANALYTICAL STANDARD DEVIATION AND COMPARISON TO SIMULATION

In this appendix we develop an analytical approximation of the standard deviation of the location error as a function of the standard deviations of the sensor accuracy. The approximated location and orientation of the vehicles is cumulative, meaning that it is affected by all the previous error measurements of the distance and bearing. The measurement error of the first step (see Eq. (35)) is:

$$E_1 = \begin{bmatrix} \cos\alpha_{11} & -r_1\sin\alpha_{11} & 0 \\ \sin\alpha_{11} & r_1\cos\alpha_{11} & 0 \\ 0 & 1 & -1 \end{bmatrix} \Delta_1 . \tag{94}$$

At the second step it becomes:

$$\begin{aligned} E_2 = E_1 &+ \begin{bmatrix} 0 & -r_2\sin\left(\theta_1+\alpha_{2,s}\right) & r_2\sin\left(\theta_1+\alpha_{2,s}\right) \\ 0 & r_2\cos\left(\theta_1+\alpha_{2,s}\right) & -r_2\cos\left(\theta_1+\alpha_{2,s}\right) \\ 0 & 0 & 0 \end{bmatrix} \Delta_1 \\ &+ \begin{bmatrix} \cos\left(\theta_1+\alpha_{2,s}\right) & -r_2\sin\left(\theta_1+\alpha_{2,s}\right) & 0 \\ \sin\left(\theta_1+\alpha_{2,s}\right) & r_2\cos\left(\theta_1+\alpha_{2,s}\right) & 0 \\ 0 & 1 & -1 \end{bmatrix} \Delta_2 \end{aligned} . \tag{95}$$

And in the third step:

$$\begin{aligned} E_3 = E_2 &+ \begin{bmatrix} 0 & -r_3\sin\left(\theta_2+\alpha_{3,s}\right) & r_3\sin\left(\theta_2+\alpha_{3,s}\right) \\ 0 & r_3\cos\left(\theta_2+\alpha_{3,s}\right) & -r_3\cos\left(\theta_2+\alpha_{3,s}\right) \\ 0 & 0 & 0 \end{bmatrix} \Delta_1 \\ &+ \begin{bmatrix} 0 & -r_3\sin\left(\theta_2+\alpha_{3,s}\right) & r_3\sin\left(\theta_2+\alpha_{3,s}\right) \\ 0 & r_3\cos\left(\theta_2+\alpha_{3,s}\right) & -r_3\cos\left(\theta_2+\alpha_{3,s}\right) \\ 0 & 0 & 0 \end{bmatrix} \Delta_2 \\ &+ \begin{bmatrix} \cos\left(\theta_2+\alpha_{3,s}\right) & -r_3\sin\left(\theta_2+\alpha_{3,s}\right) & 0 \\ \sin\left(\theta_2+\alpha_{3,s}\right) & r_3\cos\left(\theta_2+\alpha_{3,s}\right) & 0 \\ 0 & 1 & -1 \end{bmatrix} \Delta_3 \end{aligned} . \tag{96}$$

Generalizing the total error at step *n*, the error E_n in the directions *x*,*y* and the orientation θ is:

$$E_n = \begin{bmatrix} E_x \\ E_y \\ E_\theta \end{bmatrix}_n = \sum_{i=1}^{n} B_{i\_n} \Delta_i \,, \tag{97}$$

where the sensor error Δ_i at step i is:

$$\Delta_i = \begin{bmatrix} \Delta r_i & \Delta\alpha_{i,s} & \Delta\alpha_{i,t} \end{bmatrix}^T, \tag{98}$$

and the matrix B_{i_n} is calculated using:

$$B_{i\_n} = B_i + \sum_{j=i+1}^{n} C_j \,, \tag{99}$$

where:

$$B_i = \begin{bmatrix} \cos\left(\theta_{i-1} + \alpha_{i,s}\right) & -r_i \sin\left(\theta_{i-1} + \alpha_{i,s}\right) & 0 \\ \sin\left(\theta_{i-1} + \alpha_{i,s}\right) & r_i \cos\left(\theta_{i-1} + \alpha_{i,s}\right) & 0 \\ 0 & 1 & -1 \end{bmatrix}, \tag{100}$$

and:

$$C_j = \begin{bmatrix} 0 & -r_j \sin\left(\theta_{j-1} + \alpha_{j,s}\right) & r_j \sin\left(\theta_{j-1} + \alpha_{j,s}\right) \\ 0 & r_j \cos\left(\theta_{j-1} + \alpha_{j,s}\right) & -r_j \cos\left(\theta_{j-1} + \alpha_{j,s}\right) \\ 0 & 0 & 0 \end{bmatrix}. \tag{101}$$

Using Eq. (97), and under the assumption of random uncorrelated sensor measurement errors Δ_i (i.e. covariance of any two measurements is zero),

$$\mathrm{var}\left(\Delta_i\right) = \begin{bmatrix} \mathrm{var}\left(\Delta r_i\right) \\ \mathrm{var}\left(\Delta\alpha_{i,s}\right) \\ \mathrm{var}\left(\Delta\alpha_{i,t}\right) \end{bmatrix} = \begin{bmatrix} r_i^2 \sigma_d^2 \\ \sigma_\alpha^2 \\ \sigma_\alpha^2 \end{bmatrix}, \tag{102}$$

the variance of the total measurement error is:

$$\mathrm{var}\left(E_n\right) = \sum_{i=1}^{n} \mathrm{var}\left(B_{i\_n} \Delta_i\right). \tag{103}$$

The standard deviation in the x and y directions, respectively σ_x and σ_y are (Eq. (43)-(45)):

$$\sigma_x = \left[\sum_{i=1}^{n} r_i^2 \cos^2\left(\theta_{i-1} + \alpha_{i,s}\right) \sigma_d^2 + \sum_{j=1}^{n} \Omega(j) \left(\sum_{i=j}^{n} r_i \sin\left(\theta_{i-1} + \alpha_{i,s}\right) \right)^2 \sigma_\alpha^2 \right]^{0.5}, \tag{104}$$

$$\sigma_y = \left[\sum_{i=1}^{n} r_i^2 \sin^2\left(\theta_{i-1} + \alpha_{i,s}\right)\sigma_d^2 + \sum_{j=1}^{n} \Omega(j) \left(\sum_{i=j}^{n} r_i \cos\left(\theta_{i-1} + \alpha_{i,s}\right) \right)^2 \sigma_\alpha^2 \right]^{0.5}, \quad (105)$$

where:

$$\Omega(j) = \begin{cases} 1, & j = 1 \\ 2, & j > 1 \end{cases}. \quad (106)$$

And the standard deviation of the orientation σ_θ is (Eq. (42)):

$$\sigma_\theta = \sqrt{2n} \cdot \sigma_\alpha \ . \quad (107)$$

The validation of Eq. (104)-(106) is presented in Table 10.1 where a comparison is made between the total standard deviation values σ calculated using the MCS (10,000 simulations) and the analytical expression. The results show that the largest relative difference between the two methods is 0.79%.

Table 10.1: Total standard deviation values using 10,000 simulations, analytic calculation and relative difference, 200 meters 'S' path.

Sensor variables	Parallel			Alternating			Following		
	Simulation	Analytic	Relative diff.	Simulation	Analytic	Relative diff.	Simulation	Analytic	Relative diff.
σ_d=1% σ_α=0.1°	1.073 m	1.075 m	0.23%	1.137 m	1.134 m	0.27%	1.198 m	1.204 m	0.55%
σ_d=5% σ_α=0.1°	3.353 m	3.372 m	0.58%	3.887 m	3.857 m	0.79%	4.023 m	4.010 m	0.32%
σ_d=2% σ_α=0.5°	4.465 m	4.468 m	0.080%	4.484 m	4.499 m	0.32%	4.864 m	4.843 m	0.45%
σ_d=5% σ_α=1°	9.173 m	9.148 m	0.27%	9.233 m	9.276 m	0.46%	10.01 m	9.965 m	0.43%

[B] MATLAB CODES FOR MONTE CARLO SIMULATION

MATLAB codes used to obtain the results presented in Section 5.

1. Main

```
function slam
% Calculates errors between desired location and real location (with Jacobi
errors) for N possible random errors
% Car2 takes first step and then alternate
% Car1 initial position [0,0,0]

global N; % how many dots, number of errors for each step
N=10000;
global NUM_OF_DIFF3;
NUM_OF_DIFF3=1;
global NUM_OF_DIFF2;
NUM_OF_DIFF2=1;
global delta_range delta_angle
delta_range = 0.02;
delta_angle = 0.5*pi/180;
global steps;
colormap bone

% 2 straight lines 10 steps 100m
Car1 = [ 0 0 0;  0 20 0;  0 40 0;  0 60 0;  0 80 0;  0 100 0];
Car2 = [10 0 0; 10 10 0; 10 30 0; 10 50 0; 10 70 0; 10  90 0];

map(Car1,Car2)
steps = path_to_steps(Car1,Car2);

n=size(steps,2); % number of steps
sum_1=1; %number of steps
sum_2=1;
orientation=0; % the orientation angle
car11=zeros(3,N); % matrix of X Y of cars with real errors
car22=zeros(3,N);
Ae = eye(3);
Ae=repmat(Ae,1,N); % Rotation matrix for error calculation
AA=eye(3); % Rotation matrix for no error calculation
car1_Jerrors=zeros(2,N); % matrix of X Y of cars with jacobian
car2_Jerrors=zeros(2,N);
errorBar1=zeros(1,n);
sigma_1_his=zeros(1,n);
sigma_2_his=zeros(1,n);
sigma_tot=zeros(1,n);
angles=zeros(1,n);

% N random errors for each step
Deltas_cars = rand_n;
[Jacob,xi,yi] = Jacob_multipication(Deltas_cars);

% calculating the variance directly from Jacobian
stds = var_direct_calc;
sigma_x_direct = stds(1,:);
sigma_y_direct = stds(2,:);
sigma_t_direct = stds(3,:);

%---------------------INITIALIZE-----------------------------------
car = steps(1,:);
```

```
alpha_s = steps(2,:)*pi/180;
R = steps(3,:);
alpha_t = steps(4,:)*pi/180;
phi = alpha_s+pi-alpha_t;
prev_phi = zeros(1,N);

figure

for i=1:n %for each step
%-----------------------WITH ERRORS (REAL LOCATION)------------------
    r = Deltas_cars(NUM_OF_DIFF3,:); % N radius errors for current step
    a1 = Deltas_cars(NUM_OF_DIFF3+1,:); % N angle errors for current step
    a2 = Deltas_cars(NUM_OF_DIFF3+2,:); % N angle errors for current step
    j = 1;
    if car(i)==1
       sum_1=sum_1+1;
       for k=1:N
           [car11(:,k),A] =
error11(steps,Ae(:,j:j+2),a1(k),a2(k),r(k),i,prev_phi(k));
           Ae(:,j:j+2) = A; % for next step
           j=j+3;
       end
       prev_phi = car11(3,:); % for next step
    else
       sum_2=sum_2+1;
       for k=1:N
           [car22(:,k),A] =
error11(steps,Ae(:,j:j+2),a1(k),a2(k),r(k),i,prev_phi(k));
           Ae(:,j:j+2) = A; % for next step
           j=j+3;
       end
       prev_phi = car22(3,:); % for next step
    end
%----------------------ORIGIN (NO ERRORS)-----------------------------
    % The current rotation matrix
    A_curent=[cos(phi(i)) -sin(phi(i)) R(i)*cos(alpha_s(i));
              sin(phi(i))  cos(phi(i)) R(i)*sin(alpha_s(i));
              0         0              1];
    AA=AA*A_curent;
    orientation=orientation+phi(i);
    if orientation >= 2*pi
        orientation = orientation - 2*pi;
    end
    if car(i)==1
        orientation1=orientation;
        Car1(sum_1,1)=AA(1,3);
        Car1(sum_1,2)=AA(2,3);
        Car1(sum_1,3)=orientation1;
    else
        orientation2=orientation;
        Car2(sum_2,1)=AA(1,3);
        Car2(sum_2,2)=AA(2,3);
        Car2(sum_2,3)=orientation2;
    end
%----------------------JACOBI----------------------------------------
% calculating real locations by equation:
% [real location] = [desired location] + [J]*[delta]
    JacobiError = Jacob(NUM_OF_DIFF2:NUM_OF_DIFF2+1,:); % Jacobian*delta
    if car(i)==1
       car1_Jerrors(1,:) = xi(i) + JacobiError(1,:);
       car1_Jerrors(2,:) = yi(i) + JacobiError(2,:);
    else
```

```
        car2_Jerrors(1,:) = xi(i) + JacobiError(1,:);
        car2_Jerrors(2,:) = yi(i) + JacobiError(2,:);
    end
%-------------------------DIFFERENCES-------------------------
    if car(i)==1
        cc_J = [mean(car1_Jerrors(1,:)), mean(car1_Jerrors(2,:))];
        cc_R = [mean(car11(1,:)), mean(car11(2,:))];
    else
        cc_J = [mean(car2_Jerrors(1,:)), mean(car2_Jerrors(2,:))];
        cc_R = [mean(car22(1,:)), mean(car22(2,:))];
    end
    % comparing locations of Jacobi calc and direct calc
    if car(i)==1
        error_x1 = mean(abs(car1_Jerrors(1,:)-car11(1,:)));
        error_y1 = mean(abs(car1_Jerrors(2,:)-car11(2,:)));
    else
        error_x1 = mean(abs(car2_Jerrors(1,:)-car22(1,:)));
        error_y1 = mean(abs(car2_Jerrors(2,:)-car22(2,:)));
    end
    errorBar1(i) = sqrt(error_x1^2+error_y1^2); % radius error
%-------------------------------------------------------------
    % calculating distance std (Jacobi)
    if car(i)==1
        for j=1:N
            % distance squared
            disX(j) = (car1_Jerrors(1,j)-cc_J(1))^2;
            disY(j) = (car1_Jerrors(2,j)-cc_J(2))^2;
        end
    else
        for j=1:N
            disX(j) = (car2_Jerrors(1,j)-cc_J(1))^2;
            disY(j) = (car2_Jerrors(2,j)-cc_J(2))^2;
        end
    end
    sigma_x_jacobi(i) = sqrt(sum(disX(:))/(N-1));
    sigma_y_jacobi(i) = sqrt(sum(disY(:))/(N-1));
    dis = disX + disY;
    sigma_d_jacobi(i) = sqrt(sum(dis(:))/(N-1)); % variance eq.
% % %-----------------------PLOT---------------------------------
%     map of desired location of cars + histogram
    if car(i)==1
        [X,Y,C] = histogram2(car1_Jerrors(1,:),car1_Jerrors(2,:));
        if i==1 || i==2 || i==8 || i==14 || i==20 || i==25
      image('XData',X,'YData',Y,'CData',C,'CDataMapping','scaled');
            hold on
        end
    else
        [X,Y,C] = histogram2(car2_Jerrors(1,:),car2_Jerrors(2,:));
        if i==1 || i==2 || i==8 || i==14 || i==20 || i==25
      image('XData',X,'YData',Y,'CData',C,'CDataMapping','scaled');
            hold on
        end
        hold on
    end
    c = plot(cc_J(1),cc_J(2),'*m'); % plotting Jacobian centroid
    d = plot(cc_R(1),cc_R(2),'xr'); % plotting direct calc centroid
    hold on
    axis equal;
% % %-------------------------------------------------------------
%     calculating std values and plot ellipse
    if car(i)==1
```

```
        [sigma_1_his(i),sigma_2_his(i),angle] =
error_ellipse(car1_Jerrors',i);
    else
        [sigma_1_his(i),sigma_2_his(i),angle] =
error_ellipse(car2_Jerrors',i);
    end

    sigma_tot(i) = sqrt(sigma_1_his(i)^2 + sigma_2_his(i)^2);
    angles(i) = angle*180/pi; % rad to deg

    NUM_OF_DIFF3=NUM_OF_DIFF3+3;
    NUM_OF_DIFF2=NUM_OF_DIFF2+2;
end

a=plot(Car1(:,1),Car1(:,2),'s-b'); % connecting the path
b=plot(Car2(:,1),Car2(:,2),'s-g');
axis equal;
grid on;
xlabel('x[m]');
ylabel('y[m]');
legend([c d],'Approx. centroid','Exact centroid');
hold off

figure
bar([sigma_tot', sigma_1_his', sigma_2_his']);
grid on
% title('From histogram');
xlabel('Step Number');
ylabel('Standard deviations [m]');
legend('\sigma_{tot}','\sigma_1','\sigma_2');
```

2. Map of Robot Positions

```
function map(Car1,Car2)
%the function's input- mat for each car: x;y;teta(deg NOT rad)
%the function's output- figure of mapping with orientations

lengtha=size(Car1,1);
lengthb=size(Car2,1);
plot(Car1(:,1),Car1(:,2),'b',Car2(:,1),Car2(:,2),'g','LineWidth',1);
L = 4;
W = 2;

for i=1:lengtha %patch for the car and orientation
    origin1=[Car1(i,1);Car1(i,2);0]; %the specific x,y of the car
    coord=[Car1(i,1)-W Car1(i,1)+W Car1(i,1)+W Car1(i,1)-W;
           Car1(i,2)-L Car1(i,2)-L Car1(i,2)+L Car1(i,2)+L;
           0 0 0 0];
    %the coords of the polygon car
    vectors=coord-[origin1,origin1,origin1,origin1];
    %vectors of the polygon, from origin to the coords
    rotvectors=rotz(Car1(i,3))*vectors;
    %rotating the vectors according to the angle in deg
    newcoord=rotvectors+[origin1,origin1,origin1,origin1];
    %finding the new coords of the polygon after the rotation
    ax1=patch(newcoord(1,:),newcoord(2,:),[51 202 255]/255);
    hold on
end
for i=1:lengthb
```

```
    origin2=[Car2(i,1);Car2(i,2);0];
    coord=[Car2(i,1)-W Car2(i,1)+W Car2(i,1)+W Car2(i,1)-W;
           Car2(i,2)-L Car2(i,2)-L Car2(i,2)+L Car2(i,2)+L;
           0 0 0 0];
    vectors=coord-[origin2,origin2,origin2,origin2];
    rotvectors=rotz(Car2(i,3))*vectors;
    newcoord=rotvectors+[origin2,origin2,origin2,origin2];
    ax2=patch(newcoord(1,:),newcoord(2,:),[169 218 116]/255);
    hold on
end

axis equal %otherwise the polygons are deformed
grid on
box on
legend([ax1,ax2],'Vehicle 1','Vehicle 2','Location','northoutside');
xlabel('X[m]');
ylabel('Y[m]');
end
```

3. Path to Steps

```
function steps = path_to_steps(Car1,Car2)
% take path of 2 cars [X,Y,theta] and create steps matrix:
% [moving car index, angle from stationary car, distance, angle from
traveling car]
% angles in deg

n = size(Car1,1) + size(Car2,1) - 2; % number of steps (first 2 initial
positions)
steps = zeros(4,n);
moving_car = 2; % Car2 takes first step
sum1 = 1;
sum2 = 1;
for i = 1:n
    if moving_car == 1
        sum1 = sum1 + 1;
        steps(1,i) = 1;
        x_dis = Car1(sum1,1)-Car2(sum2,1);
        y_dis = Car1(sum1,2)-Car2(sum2,2);
        angle = atan2(y_dis,x_dis)*180/pi; % angle between cars if both
have 0 orientation
        steps(2,i) = angle - Car2(sum2,3); % angle - orientation Car2
        steps(4,i) = angle + 180 - Car1(sum1,3);
        steps(3,i) = sqrt(x_dis^2 + y_dis^2);
        moving_car = 2; % for next step
    else
        sum2 = sum2 + 1;
        steps(1,i) = 2;
        x_dis = Car2(sum2,1)-Car1(sum1,1);
        y_dis = Car2(sum2,2)-Car1(sum1,2);
        angle = atan2(y_dis,x_dis)*180/pi;
        steps(2,i) = angle - Car1(sum1,3);
        steps(4,i) = angle + 180 - Car2(sum2,3);
        steps(3,i) = sqrt(x_dis^2 + y_dis^2);
        moving_car = 1;
    end
end
end
```

4. Random Measurement Errors

```
function Deltas_cars = rand_n
% returns a matrix [3*n,N] of N random errors for each step with normal
distribution
% angles in rad

global delta_range delta_angle
global N steps
n=size(steps,2); % number of steps
R=steps(3,:); % vector of all radiuses for each step
Deltas_cars=zeros(2*n,N);
j=1;
rng('shuffle');%in order to initialize the generator and get different
random numbers
% randn returns random numbers normally with variance 1 and mean 0
for i=1:n
    Deltas_cars(j,:) = delta_range*R(i)*randn(1,N); % delta r
    Deltas_cars(j+1,:) = delta_angle*randn(1,N); % delta alpha_s
    Deltas_cars(j+2,:) = delta_angle*randn(1,N); % delta alpha_t
    j=j+3;
end
end
```

5. Location Errors via Jacobi Mathod

```
function [Jacobi,X,Y] = Jacob_multipication(Deltas_cars)
% calculates the Jacobi matrix for current steps
% + multiplication of Jacobi matrix and errors
% + parametric location of car (X,Y)
% Jacobi(2*n,N) contains multiplication of Jacobi and errors
% each 2 lines for each step - (x,y) errors

N = size(Deltas_cars,2);
global steps;
n = size(steps,2); % number of steps

Jacobi = zeros(2*n,N);
X = zeros(n,1); Y = zeros(n,1);
x = 0;           y = 0;
J_old = 0;
j = 1; k = 1;
theta_prev = 0;
D_ai = 0;
% desired positions data
alpha_s = steps(2,:)*pi/180; % to rad
R = steps(3,:);
alpha_t = steps(4,:)*pi/180;
phi = alpha_s+pi-alpha_t;

for i=1:n
    ri = R(i); % current radius
    D_ri = Deltas_cars(k,:);
    D_ai = D_ai + Deltas_cars(k+1,:);
    S = sin(theta_prev+alpha_s(i));
    C = cos(theta_prev+alpha_s(i));
    J_times_delta = [-ri*S*D_ai + D_ri*C;
                      ri*C*D_ai + D_ri*S];
    X(i) = x + ri*C; Y(i) = y + ri*S; % desired positions
    x    = X(i);     y    = Y(i);    % for next step
```

```
    Jacobi(j:j+1,:) = J_old + J_times_delta;
    % for next step
    J_old = Jacobi(j:j+1,:);
    theta_prev = theta_prev + phi(i);
    D_ai = D_ai - Deltas_cars(k+2,:);
    j=j+2;
    k=k+3;
end
end
```

6. Analytic Standard Deviation Calculation

```
function stds = var_direct_calc
% calculating the variance directly from Jacobian calc

global steps delta_range delta_angle
n = size(steps,2); % number of steps
alpha_s = steps(2,:)*pi/180;
R = steps(3,:);
alpha_t = steps(4,:)*pi/180;
phi = alpha_s+pi-alpha_t;
var_r = delta_range^2;
var_a = delta_angle^2;
std_x = zeros(1,n);
std_y = zeros(1,n);
% first step
A = [cos(alpha_s(1)) -R(1)*sin(alpha_s(1)) 0;
     sin(alpha_s(1))  R(1)*cos(alpha_s(1)) 0;
     0                1                    -1];
std_x(1) = sqrt(R(1)^2*A(1,1)^2*var_r + (A(1,2)^2 + A(1,3)^2)*var_a);
std_y(1) = sqrt(R(1)^2*A(2,1)^2*var_r + (A(2,2)^2 + A(2,3)^2)*var_a);
allA = zeros(3,3*n); % matrix of n A matrices
allA(:,1:3) = A; % allA = [A 0 0 ... 0]
prev_phi = phi(1); % for next step

for i = 2:n
    A = [cos(prev_phi+alpha_s(i)) -R(i)*sin(prev_phi+alpha_s(i)) 0;
         sin(prev_phi+alpha_s(i))  R(i)*cos(prev_phi+alpha_s(i)) 0;
         0                         1                             -1];
 B = [0 -R(i)*sin(prev_phi+alpha_s(i)) R(i)*sin(prev_phi+alpha_s(i));
     0  R(i)*cos(prev_phi+alpha_s(i)) -R(i)*cos(prev_phi+alpha_s(i));
     0  0                                                          0];
   % inserting A and B matrices to general allA matrix
     k = 1;
     for j = 1:i-1 % adding matrix B
         allA(:,k:k+2) = allA(:,k:k+2) + B;
         k = k + 3;
     end
     allA(:,3*i-2:3*i) = A;
     % calculating stds
     k = 1;
     A_rx = 0; A_ax = 0; A_ry = 0; A_ay = 0;
     for j = 1:i
         A_rx = A_rx + (R(j)*allA(1,k))^2;
         A_ax = A_ax + allA(1,k+1)^2 + allA(1,k+2)^2;
         A_ry = A_ry + (R(j)*allA(2,k))^2;
         A_ay = A_ay + allA(2,k+1)^2 + allA(2,k+2)^2;
         k = k + 3;
     end
     std_x(i) = sqrt(A_rx*var_r + A_ax*var_a);
```

```
     std_y(i) = sqrt(A_ry*var_r + A_ay*var_a);
     prev_phi = prev_phi + phi(i); % for next step
end
std_d = sqrt(std_x.^2 + std_y.^2);
stds = [std_x; std_y; std_d];
end
```

7. Location via Exact Method

```
function [Car,A] = error11(steps,Ae,a1,a2,r,i,prev_phi)
% calcs location  with errors by exact method
% f(R+delta_R , teta+delta_teta)
Car=[0;0;0]; % X Y theta

alpha_s = steps(2,i)*pi/180 + a1;
R = steps(3,i) + r;
alpha_t = steps(4,i)*pi/180 + a2;
phi = alpha_s+pi-alpha_t;
% The current rotation matrix
A_current = [cos(phi) -sin(phi) R*cos(alpha_s);
             sin(phi)  cos(phi) R*sin(alpha_s);
             0         0        1];
A = Ae*A_current; % The overall rotation matrix

Car(1) = A(1,3);
Car(2) = A(2,3);
Car(3) = prev_phi + phi;

end
```

8. 2D Histogram

```
function [X,Y,C] = histogram2(x,y)
% plot the distribution of random errors for each step;
% locations of dots (x,y)
% X,Y - axis, C - distribution matrix
% Copyright (c) by R. Moddemeijer, Date: 2001/02/05 09:54:29
colormap bone

minx = min(x); maxx = max(x);
deltax = (maxx-minx)/(length(x)-1);
ncellx = 2*ceil(length(x)^(1/3));
miny = min(y); maxy = max(y);
deltay = (maxy-miny)/(length(y)-1);
ncelly = ncellx;

lowerx = minx - deltax/2;
upperx = maxx + deltax/2;
lowery = miny - deltay/2;
uppery = maxy + deltay/2;

result(1:ncellx,1:ncelly)=0;

xx=round( (x-lowerx)/(upperx-lowerx)*ncellx + 1/2 );
yy=round( (y-lowery)/(uppery-lowery)*ncelly + 1/2 );

for n=1:length(xx)
  indexx=xx(n);
```

```
  indexy=yy(n);
  if indexx >= 1 && indexx <= ncellx && indexy >= 1 && indexy <= ncelly
    result(indexx,indexy)=result(indexx,indexy)+1;
  end
end

L =  max(max(result));
C = (L - result)./L; % reverse the matrix and normalize to fit colormap
(values between 0&1)
X = [minx maxx]; % x axis
Y = [miny maxy]; % y axis

end
```

9. Error Ellipse

```
function [sigma_x,sigma_y,angle] = error_ellipse(data,i)
% data = JacobiError matrix [J]*[delta], size: [N,2]
% plots the ellipse's main axes and returns stds and rotation angle
% Copyright Vincent Spruyt
% http://www.visiondummy.com/2014/04/draw-error-ellipse-representing-
covariance-matrix/

% Calculate the eigenvectors and eigenvalues
covariance = cov(data);
[eigenvec, eigenval ] = eig(covariance);
% Get the index of the largest eigenvector
[largest_eigenvec_ind_c, ~] = find(eigenval == max(max(eigenval)));
largest_eigenvec = eigenvec(:, largest_eigenvec_ind_c);
% Get the largest eigenvalue
largest_eigenval = max(max(eigenval));
% Get the smallest eigenvector and eigenvalue
if(largest_eigenvec_ind_c == 1)
    smallest_eigenval = max(eigenval(:,2));
    smallest_eigenvec = eigenvec(:,2);
else
    smallest_eigenval = max(eigenval(:,1));
    smallest_eigenvec = eigenvec(1,:);
end

% Calculate the angle between the x-axis and the largest eigenvector
angle = atan2(largest_eigenvec(2), largest_eigenvec(1));

% This angle is between -pi and pi.
% Let's shift it such that the angle is between 0 and 2pi
if(angle < 0)
    angle = angle + 2*pi;
end
% To keep angles between -pi/2 and pi/2
if angle > pi/2 && angle < pi % second Quadrant
    angle = angle - pi;
end
if angle > pi && angle < 3*pi/2 % third Quadrant
   angle = angle - pi;
end
if angle > 3*pi/2 && angle < 2*pi % forth Quadrant
   angle = angle - 2*pi;
end

% Get the coordinates of the data mean
```

```
avg = mean(data);
X0=avg(1);
Y0=avg(2);

%% drawing error ellipses
if i == 1 || i == 2 || i == 8 || i == 14 || i == 20 || i == 25
    % Get the 68%~sigma% confidence interval error ellipse
    chisquare_val = 2.408; % for 70%
    theta_grid = linspace(0,2*pi);
    a=sqrt(chisquare_val*largest_eigenval);
    b=sqrt(chisquare_val*smallest_eigenval);
    % the ellipse in x and y coordinates
    ellipse_x_r  = a*cos( theta_grid );
    ellipse_y_r  = b*sin( theta_grid );
    % Define a rotation matrix
    R = [ cos(angle) sin(angle); -sin(angle) cos(angle) ];
    %let's rotate the ellipse to some angle phi
    r_ellipse = [ellipse_x_r;ellipse_y_r]' * R;
    % plotting confidence ellipses
    plot(r_ellipse(:,1) + X0,r_ellipse(:,2) + Y0,'Color',[0 0.4
1],'LineWidth',3) % blue
    hold on;

    %% Plot the eigenvectors
    quiver(X0, Y0, largest_eigenvec(1) *sqrt(chisquare_val*
largest_eigenval),  largest_eigenvec(2)*sqrt(chisquare_val*
largest_eigenval), '-m', 'LineWidth',3);
    quiver(X0, Y0,
smallest_eigenvec(1)*sqrt(chisquare_val*smallest_eigenval),
smallest_eigenvec(2)*sqrt(chisquare_val*smallest_eigenval), '-m',
'LineWidth',3);

    % Get the 95%~2*sigma% confidence interval error ellipse
    chisquare_val = 5.991;
    theta_grid = linspace(0,2*pi);
    a=sqrt(chisquare_val*largest_eigenval);
    b=sqrt(chisquare_val*smallest_eigenval);
    % the ellipse in x and y coordinates
    ellipse_x_r  = a*cos( theta_grid );
    ellipse_y_r  = b*sin( theta_grid );
    % Define a rotation matrix
    R = [ cos(angle) sin(angle); -sin(angle) cos(angle) ];
    % let's rotate the ellipse to some angle phi
    r_ellipse = [ellipse_x_r;ellipse_y_r]' * R;
    % plotting confidence ellipses
    plot(r_ellipse(:,1) + X0,r_ellipse(:,2) + Y0,'--','Color',[0.4 0
0.8],'LineWidth',3) % purple
end
sigma_x = sqrt(largest_eigenval);
sigma_y = sqrt(smallest_eigenval);
end
```

[C] MATLAB CODES FOR EXPERIMENT RESULTS ANALYSIS

MATLAB codes used to obtain the results presented in Section 0. All presented codes refer to the straight parallel path, similar codes were used for the other three paths.

1. Create Excel File

```
% create excel file
path = 'straight parallel';
exp = ' exp5';
title = {'x center','y center','ML'};
sheet = 1;

car = 'L';
fileName = strcat(path,exp,car,'.xlsx');
xlswrite(fileName,title,sheet,'A1');
for i = 0:10
    img = strcat(car,num2str(i),'.jpg');
    toExcel(fileName,img,i)
end

car = 'R';
fileName = strcat(path,exp,car,'.xlsx');
xlswrite(fileName,title,sheet,'A1');
for i = 0:10
    img = strcat(car,num2str(i),'.jpg');
    toExcel(fileName,img,i)
end
```

2. Write Image Data to Excel

```
function toExcel(fileName,img,i)
% write to excel coordinates of center of ball (in pixels) X
% and length of height (in pixels) ML

X = findCenter(img,i); % [x,y] 2 values
ML = findDis(img); % 1 value
line = num2str(i+2);
range = strcat('A',line);
sheet = 1;
xlswrite(fileName,[X ML],sheet,range);

end
```

3. Find Center

```
function X = findCenter(img,i)
% find x center of tennis ball in image I
% NL = number of pixels of diameter

RGB = imread(img);

huelow = 45; % hue value of ball color
huehigh = 100;
satlow = 25;
```

```
sathigh = 200;
if mod(i,2) == 0 % even
    radiusRange = [75 100];
else             % uneven
    radiusRange = [15 25];
end

%% filter
% filter by hue (color of ball)
I = rgb2hsv(RGB); % hsv
% Create mask based on hue value
BW = ((I(:,:,1) >= huelow/256 & I(:,:,1) <= huehigh/256) & ...
      (I(:,:,2) >= satlow/256 & I(:,:,2) <= sathigh/256));
% Initialize output masked image based on input image
maskedRGBImage = RGB;
% Set background pixels where BW is false to zero
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;

%% find center
sens = 0.9;
radii = [];
while isempty(radii) % no center found
    [centers,radii] =
imfindcircles(maskedRGBImage,radiusRange,'Sensitivity',sens);
    sens = sens + 0.01;
end
figure; imshow(maskedRGBImage); title(img); hold on
scatter(centers(1,1),centers(1,2),'*r')
X = centers(1,:);

end
```

4. Plot Results and Calculate Standard Deviation and Mean Error Values

```
% 5 experiments
clear
CarR = zeros(6,3,5);
CarL = zeros(6,3,5);

CarRreal = [0 0 0; 0 1.5 0; 0 3 0; 0 4.5 0; 0 6 0; 0 7.5 0];
CarLreal = [-1.5 0 0; -1.5 1.5 0; -1.5 3 0; -1.5 4.5 0; -1.5 6 0; -1.5 7.5
0];

title = {'7.5 m Straight Path','Parallel Advancment'};
map(CarLreal,CarRreal,title); hold on; axis equal;

for i = 1:5
    [CarR(:,:,i),CarL(:,:,i)] = singleExp(i+1); % 2 to 6
    scatter(CarL(:,1,i),CarL(:,2,i),50,[51 202 255]/255,'s','filled'); %
blue
    plot(CarL(:,1,i),CarL(:,2,i),'Color',[51 202 255]/255);
    scatter(CarR(:,1,i),CarR(:,2,i),50,[169 218 116]/255,'s','filled'); %
green
    plot(CarR(:,1,i),CarR(:,2,i),'Color',[169 218 116]/255);
    drawnow
end

%% calc errors
% mean error, mean of both cars last step
disX_R = mean(CarR(end,1,:)-CarRreal(end,1));
```

```
disY_R = mean(CarR(end,2,:)-CarRreal(end,2));
disR = sqrt(disX_R^2 + disY_R^2);
disX_L = mean(CarL(end,1,:)-CarLreal(end,1));
disY_L = mean(CarL(end,2,:)-CarLreal(end,2));
disL = sqrt(disX_L^2 + disY_L^2);

disX = (disX_R + disX_L)/2;
disY = (disY_R + disY_L)/2;
dis  = (disR   + disL)/2;

% std, mean of both cars last step
sigmaX_R = std(CarR(end,1,:));
sigmaY_R = std(CarR(end,2,:));
sigmaX_L = std(CarL(end,1,:));
sigmaY_L = std(CarL(end,2,:));

sigmaX = sqrt((sigmaX_R^2 + sigmaX_L^2)/2);
sigmaY = sqrt((sigmaY_R^2 + sigmaY_L^2)/2);
sigmaD = sqrt(sigmaY^2 + sigmaY^2);

% orientation error and std
% angles [-180 180] so around 0 will be +- small numbers
CarR(:,3,:) = wrapTo180(CarR(:,3,:));
CarL(:,3,:) = wrapTo180(CarL(:,3,:));

orienR = mean(CarR(end,3,:)-CarRreal(end,3));
orienL = mean(CarL(end,3,:)-CarLreal(end,3));
orien_error = (orienR + orienL)/2;

sigma_theta_R = std(CarR(end,3,:));
sigma_theta_L = std(CarL(end,3,:));
sigma_theta = sqrt((sigma_theta_R^2 + sigma_theta_L^2)/2);

% to excel
fileName = 'straight parallel errors';
sheet = 1;
title = {'sigma_d','sigma_x','sigma_y',...
         'mean error','mean error x','mean error y',...
         'sigma_theta','mean error theta'};
xlswrite(fileName,title,        sheet,'A1');
xlswrite(fileName,sigmaD,       sheet,'A2');
xlswrite(fileName,sigmaX,       sheet,'B2');
xlswrite(fileName,sigmaY,       sheet,'C2');
xlswrite(fileName,dis,          sheet,'D2');
xlswrite(fileName,disX,         sheet,'E2');
xlswrite(fileName,disY,         sheet,'F2');
xlswrite(fileName,sigma_theta,  sheet,'G2');
xlswrite(fileName,orien_error,  sheet,'H2');

%% simulation
delta_range = 0.01;
delta_angle = 0.3*pi/180;
steps = path_to_steps(CarRreal,CarLreal); % car1 = CarR, car2 = CarL
Deltas_cars = rand_n(steps,delta_range,delta_angle);
[stds,mean_error] = jacobi_method(steps,Deltas_cars,delta_angle);
```

5. Single Experiment

```
function [CarR,CarL] = singleExp(num)

CarR = [0 0 0];
path = 'straight parallel exp';

fileName = strcat(path,num2str(num),'R.xlsx');
[rR,alphaR] = ImagePros(fileName);
fileName = strcat(path,num2str(num),'L.xlsx');
[rL,alphaL] = ImagePros(fileName);

r = (rL + rR)/2;
ThetaL = findTheta('L',0); % [deg]
ThetaR = findTheta('R',0); % [deg]
alpha_s = ThetaR + alphaR(1);
alpha_t = ThetaL + alphaL(1);
x = r(1)*cos(alpha_s*pi/180);
y = r(1)*sin(alpha_s*pi/180);
theta = alpha_s + 180 - alpha_t;
CarL = [x y theta];

moving = 'L'; % first car to move
A_last = eye(3);
theta = 0;
sumL = 1; sumR = 1;

for i = 1:10
    ThetaL = findTheta('L',i); % [deg]
    ThetaR = findTheta('R',i); % [deg]
    if moving == 'L'
        alpha_s = ThetaR + alphaR(i+1);
        alpha_t = ThetaL + alphaL(i+1);
    else
        alpha_s = ThetaL + alphaL(i+1);
        alpha_t = ThetaR + alphaR(i+1);
    end
    phi = (alpha_s + 180 - alpha_t);
A_new = [cos(phi*pi/180) -sin(phi*pi/180) r(i+1)*cos(alpha_s*pi/180);
         sin(phi*pi/180)  cos(phi*pi/180) r(i+1)*sin(alpha_s*pi/180);
         0                0               1                          ];
    A = A_last*A_new;
    x = A(1,3); y = A(2,3); theta = wrapTo360(theta + phi);
    if moving == 'L'
        sumL = sumL + 1;
        CarL(sumL,:) = [x y theta];
        moving = 'R'; % for next step
    else
        sumR = sumR + 1;
        CarR(sumR,:) = [x y theta];
        moving = 'L'; % for next step
    end
    A_last = A; % for next step
end
```

6. Calculate Distance and Bearing from Image

```
function [r,alpha] = ImagePros(fileName)
% calc distance and orientation from other robot for each step

global N M FoV_x FoV_y
FoV_x = 40*pi/180; % camera's field of view in x direction[rad]
FoV_y = 70*pi/180; % camera's field of view in y direction[rad]
L = 21/100; % length from turret (top surface) to top of ball [m]
N = 1080; % number of pixels in horizontal direction
M = 1920; % number of pixels in vertical direction

xCenter = xlsread(fileName,'A:A'); % x center of ball
x = N/2 - xCenter;
alpha = atan(x./(N/2)*tan(FoV_x/2))*180/pi; % [deg] without correction of
theta
ML = xlsread(fileName,'C:C');
alphaL = (ML/M)*FoV_y; % [rad]
r = L./tan(alphaL);

end
```

[D] SIMULINK MODEL PATH FOLLOWING ALGORITHM

Simulink model and MATLAB codes used to obtain the results presented in Section 7.2.

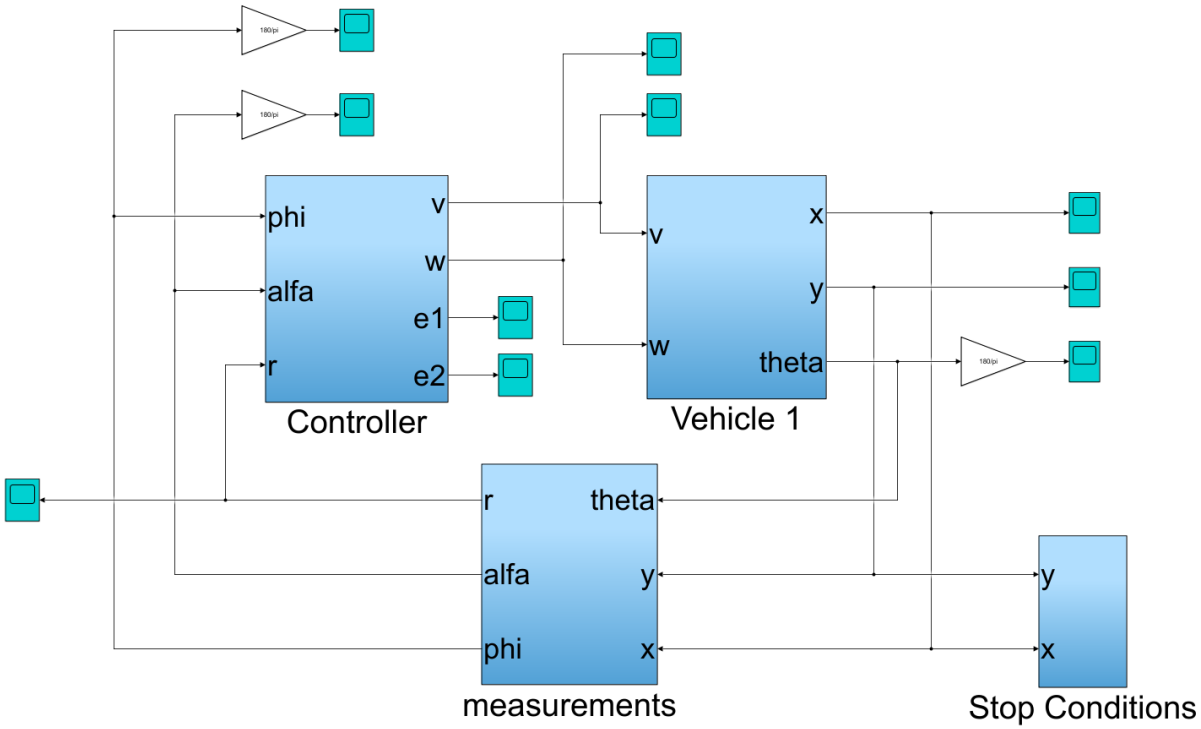


Figure 10.1: Block diagram of model.

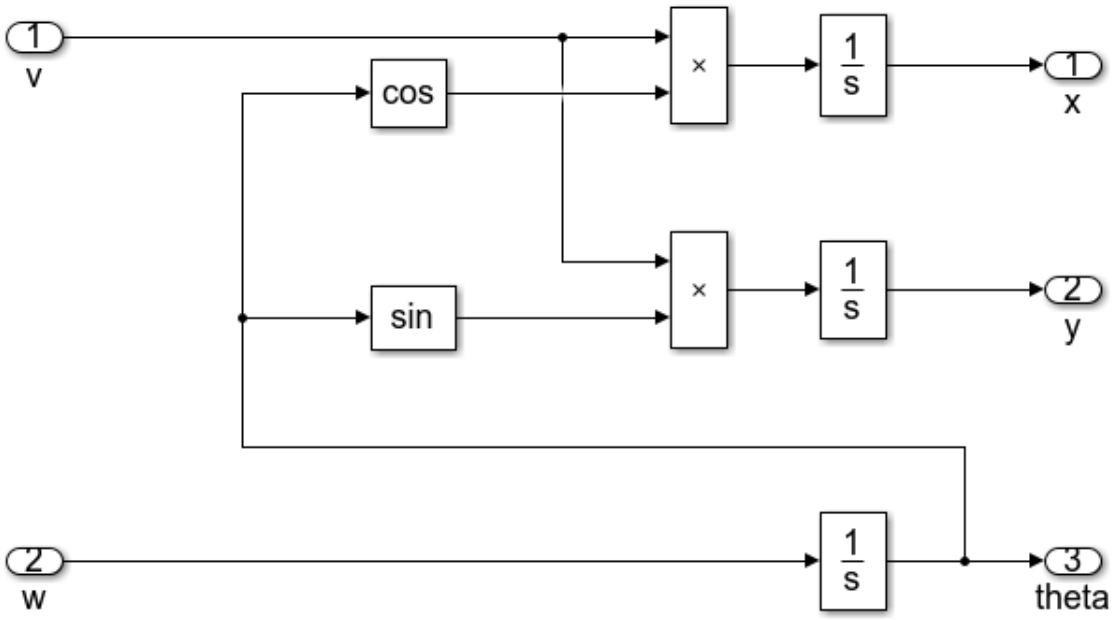


Figure 10.2: Vehicle 1 block diagram, describing the vehicle's kinematics in Cartesian coordinates.

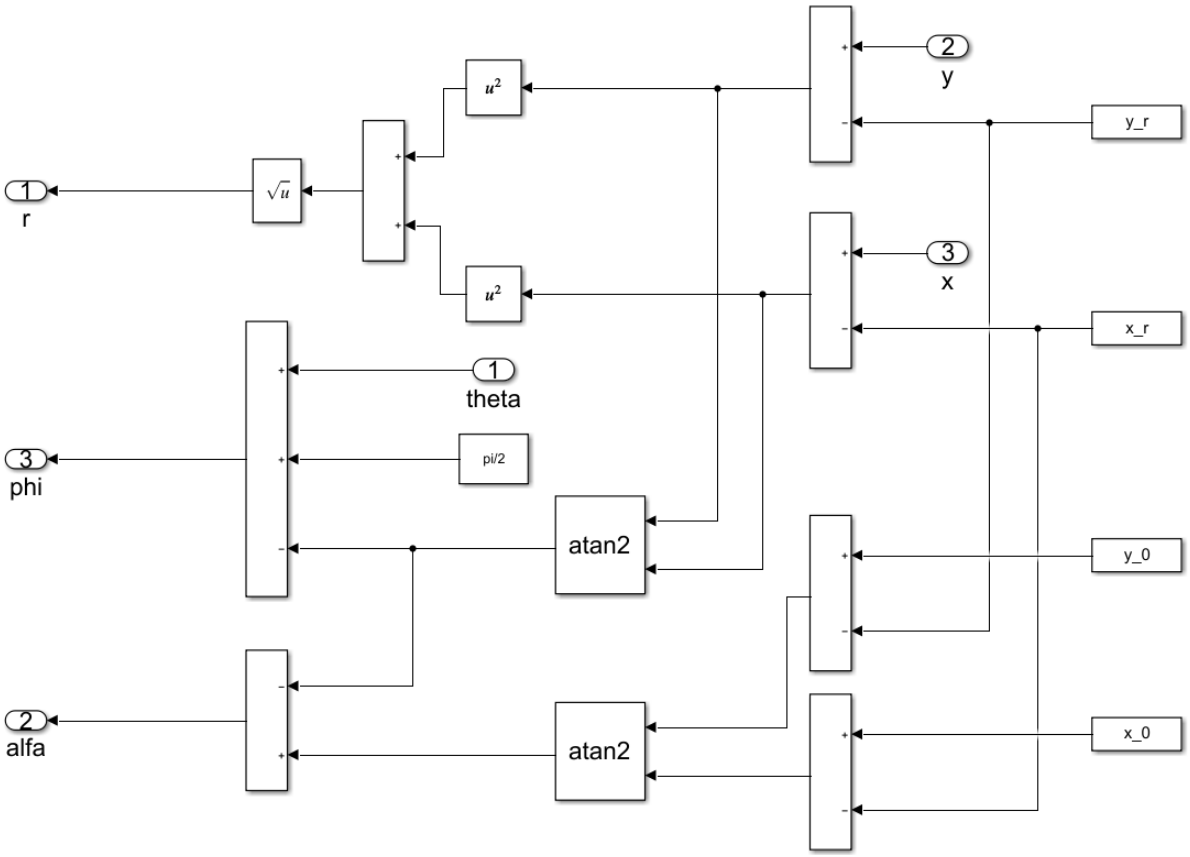


Figure 10.3: Measurements block diagram, describing the vehicle's location in polar coordinates, with respect to the stationary vehicle.

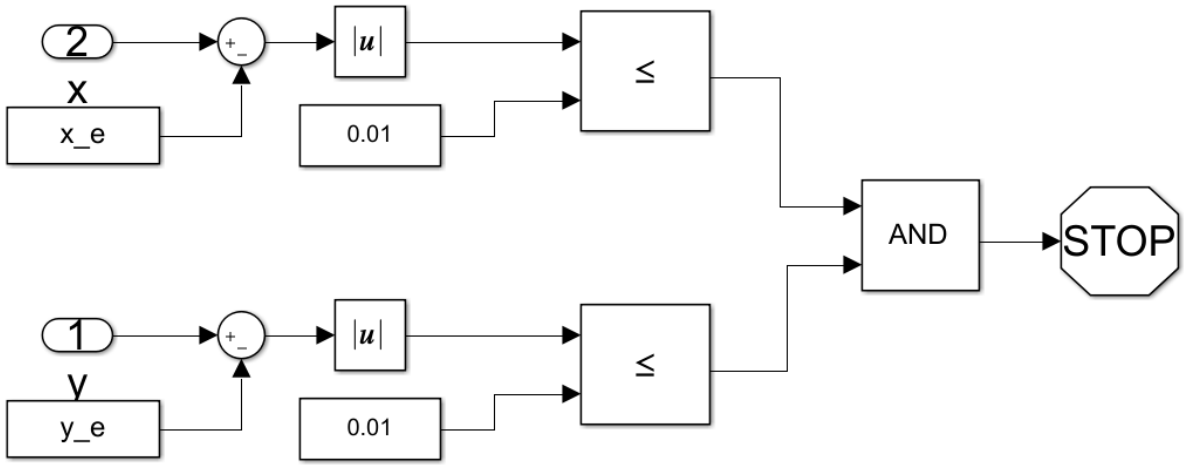


Figure 10.4: Stop Condition block diagram.

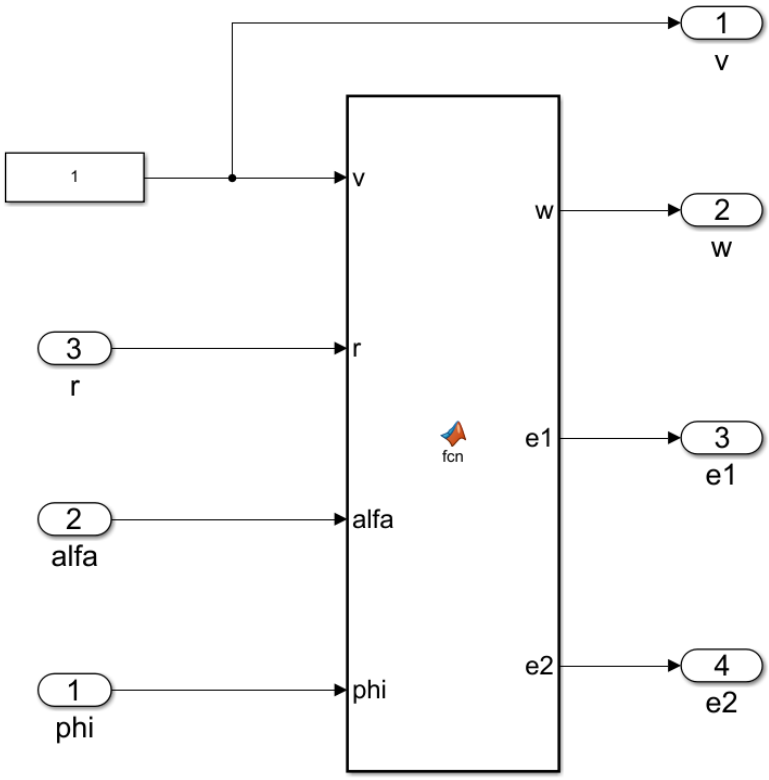


Figure 10.5: Controller block diagram, calculates the required angular velocity for path following.

1. Controller Function

```
function [w,e1,e2] = fcn(v,r,alfa,phi, x_0, y_0, x_r, y_r, x_e, y_e)

r_0 = sqrt((x_0-x_r)^2+(y_0-y_r)^2);
r_e = sqrt((x_e-x_r)^2+(y_e-y_r)^2);
L = sqrt((x_0-x_e)^2+(y_0-y_e)^2);
alfa_e = acos((r_0^2+r_e^2-L^2)/(2*r_0*r_e));
chi = asin(r_e*sin(alfa_e)/L);

r_d = r_0*sin(chi)/sin(alfa+chi);
dr_d = -r_0*sin(chi)*csc(alfa+chi)*cot(alfa+chi);
ddr_d = r_0*sin(chi)*csc(alfa+chi)*(cot(alfa+chi)^2+csc(alfa+chi)^2);

e1 = r - r_d;
e2 = sin(phi) - (dr_d*cos(phi)/r);
% pole placement
k1 = 100/(L^2);
k2 = 12/L;
u = -k1*e1 -k2*e2;

w = (r/(dr_d*sin(phi)+r*cos(phi)))*...
    ((ddr_d*cos(phi)^2/r^2)-(2*dr_d*cos(phi)*sin(phi)/r^2)-
(cos(phi)^2/r)+u)*v;

end
```

2. Path Following

```
function
path_following_no_video(x,y,theta,x_0,y_0,theta_0,x_r,y_r,r_e,alfa_e,e1,e2)

% size of vehicle
L = 0.8;
W = 0.4;

figure;
% plot stationary car
theta_r = 90; % deg
origin = [x_r;y_r;0];
coord = [x_r-L x_r+L x_r+L x_r-L;
         y_r-W y_r-W y_r+W y_r+W;
         0 0 0 0];
vectors = coord - [origin,origin,origin,origin];
rotvectors = rotz(theta_r)*vectors;
newcoord = rotvectors + [origin,origin,origin,origin];
patch(newcoord(1,:),newcoord(2,:),[51 202 255]/255);
hold on

% plot actual locations
xx = x.signals.values;
yy = y.signals.values;
plot(xx,yy,'Color',[169 218 116]/255,'LineWidth',3);

% plot desired path
alfa_e = alfa_e*pi/180;
alfa_0 = atan2((y_0-y_r),(x_0-x_r)) + 2*pi;
if alfa_0 < 0
    alfa_0 = alfa_0 + 2*pi;
end

theta_e = alfa_0 - alfa_e;
x_e = x_r + r_e*cos(theta_e);
y_e = y_r + r_e*sin(theta_e);
line([x_0 x_e],[y_0 y_e],'color','k','LineStyle','-.','LineWidth',1);
scatter(x_e,y_e,'r','filled'); % target point
grid on
xlabel('X[m]'); ylabel('Y[m]');

% plot end position (from function 'map')
theta_e_real = theta.signals.values(end);
x_e_real = xx(end);
y_e_real = yy(end);
origin = [x_e_real;y_e_real;0];
coord = [x_e_real-L x_e_real+L x_e_real+L x_e_real-L;
         y_e_real-W y_e_real-W y_e_real+W y_e_real+W;
         0 0 0 0];
vectors = coord - [origin,origin,origin,origin];
rotvectors = rotz(theta_e_real)*vectors;
newcoord = rotvectors + [origin,origin,origin,origin];
patch(newcoord(1,:),newcoord(2,:),[169 218 116]/255);

% plot initial position
origin = [x_0;y_0;0];
coord = [x_0-L x_0+L x_0+L x_0-L;
         y_0-W y_0-W y_0+W y_0+W;
         0 0 0 0];
vectors = coord - [origin,origin,origin,origin];
```

```
rotvectors = rotz(theta_0)*vectors;
newcoord = rotvectors + [origin,origin,origin,origin];
patch(newcoord(1,:),newcoord(2,:),[169 218 116]/255);

% plot observation
line([x_r x_e_real],[y_r y_e_real],'color','b','LineStyle','--
','LineWidth',2);
hold on
axis equal;
box on

figure;
E1 = e1.signals.values(:);
E2 = e2.signals.values(:);
s = e1.time;
a = plot(s,E1,'-',s,E2,'--');
a(1).Color = [204 4 109]/255; a(1).LineWidth = 2;
a(2).Color = [102 0 204]/255; a(2).LineWidth = 2;
legend('location error [m]','angle error [rad]');
xlabel('Traveled Distance');
ylabel('Error');
grid on;

end
```

[E] MATLAB CODES FOR PATH PLANNING SIMULATION

1. Main Function for Multi-Step Simulation

```
% main
% straight tunnel with 5 obstacles
close all
clear

global eps rmax xmin xmax ymin ymax
eps = 1;
rmax = 5; % max distance between cars

[xlimit,ylimit] = create_map;

% initial positions
x_0 = 0; y_0 = 0; theta_0 = 0;
x_s = 5; y_s = 0;
Car1 = [x_s y_s theta_0]; mapSingle(Car1,[ 51 202 255]/255); % blue
Car2 = [x_0 y_0 theta_0]; mapSingle(Car2,[169 218 116]/255); % green
F(1) = getframe(gcf); % for video

% new obstacle
obstacles = singleObs(x_s,y_s,x_0,y_0);
F(2) = getframe(gcf);

% plot r constraints
t = 0:0.01:2*pi;
x = x_s + rmax*cos(t);
y = y_s + rmax*sin(t);
r = plot(x,y,'--k'); hold on
F(3) = getframe(gcf);

% choose final position
[x_f,y_f] = finalPos(xlimit,obstacles,x_s,y_s);
plot(x_f,y_f,'rx','MarkerSize',8,'LineWidth',2);
F(4) = getframe(gcf);

% initialization
car = 2; % moving car
color = [169 218 116]/255; % green
sum1 = 1; sum2 = 1;
f = 4;

for i = 1:6 % 6 steps
    %% optimize trajectory
    Car =
optimize(x_s,y_s,x_0,y_0,theta_0,x_f,y_f,obstacles,xlimit,ylimit);

    %% plot path
    n = size(Car,1);
    a = 0; % stop condition
    j = 1; % 1 to n
    k = 1; % frames count
    h = animatedline('Color',color,'LineWidth',2);
    axis([xmin xmax ymin ymax])
    while a~=1
        if j == n % go through all points
           a = 1; % stop
        end
        addpoints(h,Car(j,1),Car(j,2));
```

```
        drawnow limitrate
        F(f+k) = getframe(gcf);
        k = k + 1;
        j = j + 5; % draw only every 5 points
    end
    delete(r); % delete r constraint
    Car = Car(end,:); % only last position
    mapSingle(Car,color); % plot square

    %% update locations
    rng('shuffle'); % in order to initialize the generator and get
different random numbers
    if car == 1 && i ~= 6
        sum1 = sum1 + 1;
        Car1(sum1,:) = Car;
        % for next step
        car = 2; color = [169 218 116]/255; % green
        x_s = Car1(sum1,1); y_s = Car1(sum1,2);
        x_0 = Car2(sum2,1); y_0 = Car2(sum2,2); theta_0 = Car2(sum2,3);
        % current and new obstacle
        obstacles = [obstacles(end,:); singleObs(x_s,y_s,x_0,y_0)];
        F(f+k) = getframe(gcf);
        % plot r constraints
        t = 0:0.01:2*pi;
        x = x_s + rmax*cos(t);
        y = y_s + rmax*sin(t);
        r = plot(x,y,'--k'); hold on
        F(f+k+1) = getframe(gcf);
        % choose final position
        [x_f,y_f] = finalPos(xlimit,ylimit,obstacles,x_s,y_s);
        X = plot(x_f,y_f,'rx','MarkerSize',8,'LineWidth',2); hold on
        F(f+k+2) = getframe(gcf);
    elseif car == 2 && i ~= 6
        sum2 = sum2 + 1;
        Car2(sum2,:) = Car;
        % for next step
        car = 1; color = [51 202 255]/255; % blue
        x_s = Car2(sum2,1); y_s = Car2(sum2,2);
        x_0 = Car1(sum1,1); y_0 = Car1(sum1,2); theta_0 = Car1(sum1,3);
        % current and new obstacle
        obstacles = [obstacles(end,:); singleObs(x_s,y_s,x_0,y_0)];
        F(f+k) = getframe(gcf);
        % plot r constraints
        t = 0:0.01:2*pi;
        x = x_s + rmax*cos(t);
        y = y_s + rmax*sin(t);
        r = plot(x,y,'--k'); hold on
        F(f+k+1) = getframe(gcf);
        % choose final position
        [x_f,y_f] = finalPos(xlimit,ylimit,obstacles,x_s,y_s);
        X = plot(x_f,y_f,'rx','MarkerSize',8,'LineWidth',2); hold on
        F(f+k+2) = getframe(gcf);
    end
    f = length(F);
end

delete(r); delete(X);

video = VideoWriter('Control under Constraints.avi','Uncompressed AVI');
open(video)
writeVideo(video,F)
close(video)
```

2. Create Map

```
function [xlimit,ylimit] = create_map
% tunnel limits: xlimit = [xlb, xub], ylimit = [ylb,yub]

%% straight tunnel with obstacles
global xmin xmax ymin ymax

n = 1000;
xmin = -2; xmax = 7;
xlimit = [xmin,xmax];
ymin = -2; ymax = 30;
ylimit = [ymin,ymax];
figure(1);
plot(xmin*ones(1,n),linspace(ymin,ymax,n),'-k','lineWidth',4);
hold on; axis equal; ylim([ymin,ymax]);
plot(xmax*ones(1,n),linspace(ymin,ymax,n),'-k','lineWidth',4);

end
```

3. Plot Car

```
function p = mapSingle(Car,color)
%the function's input- vector for each car: x;y;teta(deg NOT rad)
%the function's output- figure of mapping with orientations

global xmin xmax ymin ymax

L = 0.8;
W = 0.4;

origin=[Car(1);Car(2);0]; %the specific x,y of the car
coord=[Car(1)-W Car(1)+W Car(1)+W Car(1)-W;
       Car(2)-L Car(2)-L Car(2)+L Car(2)+L;
       0 0 0 0];
%the coords of the polygon car
vectors=coord-[origin,origin,origin,origin];
%vectors of the polygon, from origin to the coords
rotvectors=rotz(-Car(3)*180/pi)*vectors;
%rotating the vectors according to the angle in deg
newcoord=rotvectors+[origin,origin,origin,origin];
%finding the new coords of the polygon after the rotation
p = patch(newcoord(1,:),newcoord(2,:),color);

axis equal % otherwise the polygons are deformed
axis([xmin xmax ymin ymax])
grid on
hold on

end
```

4. Single Obstacle

```
function obstacle = singleObs(x_s,y_s,x_0,y_0)
% obstacle: [Cx,Cy,a,b,phi]
% within current step size (rmax from stationary car)

global xmin xmax rmax
rng('shuffle'); % in order to initialize the generator and get different
random numbers
minRadi = 0.5; maxRadi = 2;
c = 0; % for stop condition
tol = 5;
while c~=1 % continue until obstacle is not to close to both cars
    a  = minRadi + (maxRadi-minRadi)*rand;
    b  = minRadi + (maxRadi-minRadi)*rand;
    Cx =    xmin + (xmax-xmin)*rand;
    Cy =    y_s  +        rmax*rand;
    phi = 2*pi*rand;
    theta = 0:0.01:2*pi;
    obstacle = [Cx, Cy, a, b, phi];
    % check if obstacle is too close to cars
    if isClose2Obs(x_0,y_0,obstacle,tol) == 0 ||
isClose2Obs(x_s,y_s,obstacle,tol) == 0
        c = 1; % stop, good obstacle
    end
end
% plot obstacle
xellipse = Cx + a*cos(theta+phi);
yellipse = Cy + b*sin(theta);
patch(xellipse,yellipse,'k');
hold on

   end
```

5. Is Close to Obstacle

```
function c = isClose2Obs(x,y,obstacles,tol)
% check of point is at least eps away from all obstacles
% too close --> c = 1, not close --> c = 0
% obstacles: [Cx,Cy,a,b,phi] each 5x1 vector for 5 obstacles

numObs = size(obstacles,1);
c = 0; % point is not near obstacle
for i = 1:numObs
    Cx  = obstacles(i,1); Cy = obstacles(i,2);
    a   = obstacles(i,3); b  = obstacles(i,4);
    phi = obstacles(i,5);
    X = (x-Cx)*cos(phi) + (y-Cy)*sin(phi);
    Y = (x-Cx)*sin(phi) - (y-Cy)*cos(phi);
    % check if point is inside obstacle ellipse (+tol)
    if (X^2)/((a+tol)^2) + (Y^2)/((b+tol)^2) <= 1
       c = 1; % inside
    end
end
end
```

6. Final Position

```
function [x_f,y_f] = finalPos(xlimit,obstacles,x_s,y_s)
% choose final position for current phase by the following algorithm: y_e =
ymax while r<=rmax and not colliding with obstacles

global eps rmax

t = 0:0.01:2*pi;
% all positions that are rmax from stationary car
xrmax = x_s + rmax*cos(t);
yrmax = y_s + rmax*sin(t);
% delete positions that are outside of tunnel boundaries or in obstacles
(and don't come closer than eps)
s = length(t);
i = 1;
tol = 2;
while s ~= 0
    % outside of tunnel boundaries
    if xrmax(i) < xlimit(1)+eps || xrmax(i) > xlimit(2)-eps
        xrmax(i) = [];
        yrmax(i) = [];
    % check if point is at least tol away from all obstacles
    elseif isClose2Obs(xrmax(i),yrmax(i),obstacles,tol) == 1 % too close
        xrmax(i) = [];
        yrmax(i) = [];
    else
        i = i + 1; % if point not erased, continue to next
    end
    s = s - 1; % for stop condition
end

[y_f,ind] = max(yrmax);
x_f = xrmax(ind);

end
```

7. Optimize

```
function Car =
optimize(x_s,y_s,x_0,y_0,theta_0,x_f,y_f,obstacles,xlimit,ylimit)
% straight tunnel with 5 ellipse obstacle
% (x_s,y_s) - stationary car position
% (x_0,y_0,theta_0) - initial position moving car
% (x_f,y_f) - final desired position moving car

global eps
Vmax = 5;
Wmax = pi/8;

%% Define States Controls and Parameter
States = [...
    falcon.State('x',xlimit(1)+eps, xlimit(2)-eps, 1/abs(xlimit(2)-
xlimit(1)));...
    falcon.State('y', ylimit(1), ylimit(2), 1/abs(ylimit(2)-ylimit(1)));...
    falcon.State('theta',-pi/2-Wmax,pi/2+Wmax,1)]; % go only forwards
Controls = [...
    falcon.Control('V',     0,  Vmax, 1);...
    falcon.Control('W', -Wmax,  Wmax, 1)];
```

```
tf = falcon.Parameter('FinalTime', 20, 0, 100, 0.1);

%% Define Optimal Control Problem
problem = falcon.Problem('Car');
% Specify Discretization
tau = linspace(0,1,1001);
% Path Constraint
pathconstraints = [...
    falcon.Constraint('r_lb', -inf, 0);...
    falcon.Constraint('r_ub', -inf, 0)];
numObs = size(obstacles,1);
for i = 1:numObs
    name = strcat('obstacle',num2str(i));
pathconstraints = [pathconstraints; falcon.Constraint(name,-inf, 0)];
end

% Path constraint builder
path =
falcon.PathConstraintBuilder('CarPCon',[],States,[],[],@source_path);
path.addConstantInput('x_s', [1,1]);
path.addConstantInput('y_s', [1,1]);
path.addConstantInput('obstacles', [numObs,5]);
path.Build();

%% Add a new Phase
phase = problem.addNewPhase(@source_car, States, tau, 0, tf);
phase.addNewControlGrid(Controls, tau);
% Set Boundary Condition
phase.setInitialBoundaries([x_0; y_0; theta_0]);
phase.setFinalBoundaries([x_f; y_f; -pi*2],[x_f; y_f; 2*pi]);
% Set initial guess
initGuess = [linspace(0,x_f, length(tau));
             linspace(0,y_f, length(tau));
             linspace(0,pi/2,length(tau))];
phase.StateGrid.setValues(tau,initGuess);
% apply Path Constraint
pathc = phase.addNewPathConstraint(@CarPCon, pathconstraints, tau);
pathc.addConstants(x_s);
pathc.addConstants(y_s);
pathc.addConstants(obstacles);
% Add Cost Function
problem.addNewParameterCost(tf);

% Prepare problem for solving
problem.Bake();

%% Solve problem
solver = falcon.solver.ipopt(problem);
solver.Options.MajorIterLimit = 1000;
solver.Options.MajorFeasTol   = 1e-5;
solver.Options.MajorOptTol    = 1e-5;
solver.Solve();

%% Plot
Car = [phase.StateGrid.Values(1,:)' phase.StateGrid.Values(2,:)'
phase.StateGrid.Values(3,:)'];

end
```

8. Source Car

```
function states_dot = source_car(states, controls)

% Extract states
theta = states(3);

% Extract controls
V = controls(1);
W = controls(2);

% implement state derivatives here
x_dot      = V*sin(theta);
y_dot      = V*cos(theta);
theta_dot  = W;
states_dot = [x_dot; y_dot; theta_dot];

end
```

9. Source Path

```
function [constraints] = source_path(states, x_s, y_s, obstacles)

global eps rmax

% Extract states
x = states(1);
y = states(2);

% implement constraint values here
% r constraint
r = sqrt((x-x_s)^2+(y-y_s)^2);
r_lb = eps - r;
r_ub = r - rmax;
constraints = [r_lb; r_ub];
% ellipse obstacle
numObs = size(obstacles,1);
for i = 1:numObs
    % Extract parameters
    Cx  = obstacles(i,1);
    Cy  = obstacles(i,2);
    a   = obstacles(i,3);
    b   = obstacles(i,4);
    phi = obstacles(i,5);
    % ellipse obstacle
    X = (x-Cx)*cos(phi) + (y-Cy)*sin(phi);
    Y = (x-Cx)*sin(phi) - (y-Cy)*cos(phi);
    obstacle = 1 - (X^2)/((a+eps)^2) - (Y^2)/((b+eps)^2);
    constraints = [constraints; obstacle];
end
end
```

תקציר

תזה זו מציגה מערכת חדשנית של שני רובוטים המשתפים פעולה לקבלת מיקום עצמי דו-ממדי בדיוק גבוה, תוך שימוש בחיישנים פשוטים יחסית. היתרון המרכזי של שיטה זו מצוי ביכולת של המערכת למדוד את האוריינטציה של הרובוטים באופן מדויק ובכך להפחית את השגיאות המצטברות במדידת המיקום. כל אחד מהרובוטים מצויד בצריח מסתובב ועליו מצלמה אשר משמשת למעקב אחר הרובוט השני ומדידת המיקום והמרחק היחסיים ביניהם, ואינקודר למדידת זווית הצריח. בכל צעד, רובוט אחד מתקדם בעוד הרובוט השני נותר נייח ומודד את המיקום היחסי של הרובוט הנייד (באופן מתמשך או בסיום הצעד), בשימוש מדידת זווית הצריח והמרחק הנמדד באמצעות המצלמה. האוריינטציה של הרובוט הנייד מחושבת באמצעות סיבוב הצריח שלו לכיוון הרובוט הסטטי ומדידת זווית הצריח. באמצעות איחוד המידע הנמדד משני הרובוטים, המיקום והאוריינטציה של הרובוט הנייד מתקבלים באופן מדויק.

בנוסף, מוצג מודל אנליטי של מיקום הרובוטים כפונקציה של המידע מהחיישנים. לאחר מכן, מוצג שערוך סטטיסטי של מיקום הרובוטים באמצעות סימולציית מונטה קרלו, בהנחה כי מדידות החיישנים כוללות שגיאות רנדומליות. כמו כן, מוצג ניסוי בזמן אמת של המערכת והשוואה לתוצאות הסימולציה.

בשביל ששני הרובוטים יתקדמו בצורה אוטונומית, אלגוריתם תכנון מסלול ובקר בחוג סגור מוצגים בתזה זו, בהנחה כי המדידות הן המרחק והאוריינטציה ביחס לרובוט הנייח ואותות הבקרה הן מהירות קווית ומהירות סיבובית של הרובוט הנייד. אלגוריתם תכנון המסלול כולל בחירת נקודת יעד עבור הרובוט הנייד בכל צעד ומציאת המסלול האופטימלי תוך התחמקות ממכשולים בסביבה כמו קירות, חפצים או הרובוט הנייח. חוג הבקרה הסגור מתבסס על בחירה קודמת של נקודות יעד בסביבה הנחקרת וכן בחירה של מסלולים ביניהן כלומר, כל צעד הינו בעיית עקיבה אחר מסלול. בשל המאפיינים הפולריים של המדידות, חוג הבקרה תוכנן בקורדינטות פולריות.

מחקר זה התקבל לפרסום לאחרונה ב-IEEE Access, עיתון אינטרנטי רב-תחומי בגישה חופשית של מוסד מהנדסי חשמל ואלקטרוניקה. בנוסף, מחקר זה הוצג בכנס הישראלי ה-35 להנדסת מכונות (ICME 2018) ובכנס הישראלי ה-6 לרובוטיקה (ICR 2019).

אוניברסיטת בן גוריון בנגב

הפקולטה למדעי ההנדסה

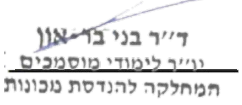
המחלקה להנדסת מכונות

מערכת מיקום-עצמי של שני רובוטים בעלת דיוק גבוה

חיבור זה מהווה חלק מהדרישות לקבלת תואר מגיסטר בהנדסה

מאת: דנה ארז

מנחים: ד"ר דוד זרוק וד"ר שי ארוגטי

מחברת: דנה ארז		תאריך: 22/12/2019
מנחה: דוד זרוק		תאריך: 22/12/2019
מנחה: שי ארוגטי		תאריך: 22/12/2019
אישור יו"ר ועדת תואר שני מחלקתית: 		תאריך: 23/12/19

ספטמבר 2019

אוניברסיטת בן גוריון בנגב

הפקולטה למדעי ההנדסה

המחלקה להנדסת מכונות

מערכת מיקום-עצמי של שני רובוטים בעלת דיוק גבוה

חיבור זה מהווה חלק מהדרישות לקבלת תואר מגיסטר בהנדסה

מאת: דנה ארז

ספטמבר 2019