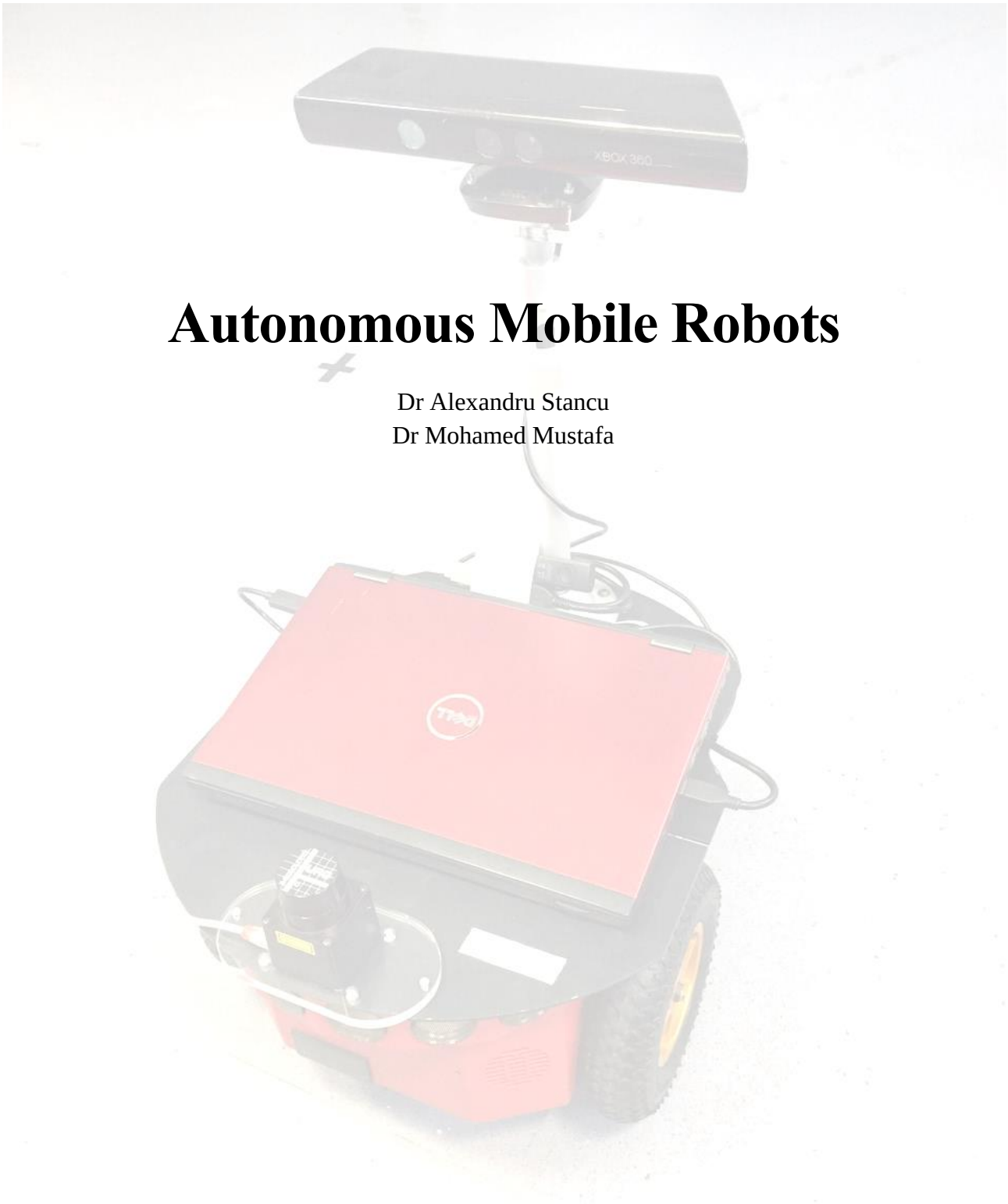


Autonomous Mobile Robots

Dr Alexandru Stancu
Dr Mohamed Mustafa

School of Electrical and Electronic Engineering
The University of Manchester

First Edition
2017

Contents

List of notation

$\emptyset$	empty set
$\mathbb{N}$	set of all positive integers
$\mathbb{Z}$	set of all integers
$\mathbb{R}$	set of all real numbers
x	scalar
$\mathbf{x}$	column vector
$\mathbf{X}$	matrix
$\nabla_x f(x)$	Jacobian Matrix $\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$, where $f(x) \in \mathbb{R}^m and\ \mathbf{x} \in \mathbb{R}^n$
$\{W\}$	world reference frame
$\{R\}$	robot reference frame
$\{C\}$	sensor reference frame
n_l	number of landmarks
n_f	number of time steps
$\mathbf{m}_i$	position of the $i-th$ landmark
M	set of all landmarks $\{m_1, m_2, \dots, m_{nl}\}$
s_k	robot pose at time step k
S_k	set of all robot poses $\{s_1, s_2, \dots, s_k\}$ up to timestep k
$\mathbf{z}_{i,p}$	observation of the $i-th$ landmark at timestep k
$Z_{i,k}$	set of all observations of the $i-th$ landmark $\{z_{i,1}, z_{i,2}, \dots, z_{i,k}\}$ up to time step k
Z_k	set of all observations of all landmarks $\{Z_{1,k}, Z_{2,k}, \dots, Z_{nl,k}\}$ up to time step k

$\mathbf{u}_k$	robot control input at time step k
U_k	set of all control inputs $\{u1;\ u2;:\ ;\ uk\}$ up to time step k
$\mathbf{h}(\mathrm{s}_k - 1, \mathbf{u}_k)$	robot motion model
$\mathbf{q}_k$	additive noise to the robot motion model
$\mathrm{g}(\mathbf{m}_i, \mathrm{s}_k)$	robot observation model
$\mathbf{r}_{i,k}$	additive nose to the robot observation model
$\Pr(\mathrm{A})$	probability of event A to occur
$p_x(\mathrm{x})$	probability density function of random vector X

Chapter 1. Introduction to Mobile Robotics

1.1 Overview

Mobile robots are constantly evolving, mainly from the beginning of the 2000s, in military domains (airborne drones), and even in medical and agricultural fields. They are in particularly high demand for performing tasks considered to be painful or dangerous to humans. This is the case for instance in mine-clearing operations, the search for black boxes of damaged aircraft on the ocean bed, planetary exploration and nuclear decommissioning. Artificial satellites, launchers (such as Ariane V), driverless subways and elevators are examples of mobile robots. Airliners, trains and cars evolve in a continuous fashion towards more and more autonomous systems and will very probably become mobile robots in the following decades.

Mobile robotics is the discipline which looks at the design of mobile robots. It is based on other disciplines such as automatic control, signal processing, mechanics, computing, electronics, etc. A mobile robot can be defined as a mechanical system capable of moving in its environment in an autonomous manner. For that purpose, it must be equipped with:

- Sensors that will help it gain knowledge of its surroundings;
- Actuators which will allow it to move;
- An intelligence (or algorithm, or control), which will allow it to determine, based on the knowledge obtained by fusing the data gathered by the sensors, the decisions in order to perform a given task.

Taking into account the level of robot knowledge (the robot understanding of its surroundings), in this unit, we addressed several probabilistic methods to control the robot at several levels of autonomy (see Figure 1.1). For instance, local knowledge of its surroundings (distance to obstacles) will lead to short term navigation, also called reacting navigation (navigation with obstacle avoidance) while global knowledge of its surroundings will lead to localisation and long term navigation (path planning). However, in many applications where the robot performs tasks in unknown environments, a more difficult problem, SLAM (Simultaneous Localisation and Mapping) must be addressed. SLAM uses local information (local surrounding observation) to simultaneously build a global map and localise the robot with the ability to recognise past surrounding observations (data association).

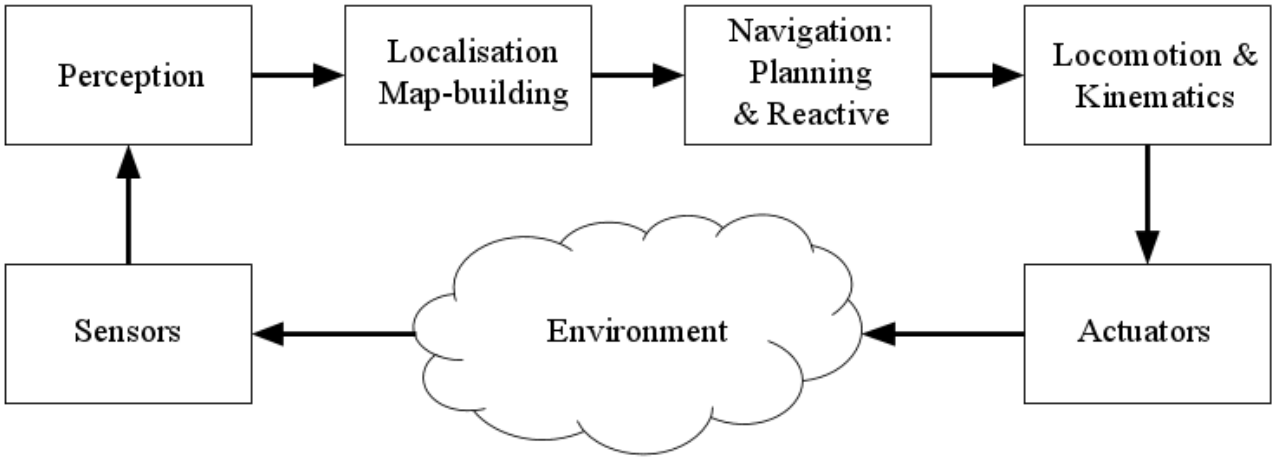


Figure 1.1 Control loop architecture for mobile robots

1.2 Classification of Mobile Robots

Mobile robots, as the name implies, have the capability to move around in their environment, not fixed to one physical location. This is in contrast with industrial robots which are usually configured as a jointed arm and an end effector attached to a fixed surface.

Several types of mobile robotics have been developed depending on the nature of application, environment and technology used. Three major classifications are identified below:

Environment of Operation:

- Terrestrial or ground-contact robots are usually referred to as Unmanned Ground Vehicles (UGVs). They are commonly wheeled or tracked. Variations include legged robots with two or more legs (humanoid, or resembling animals or insects).
- Aerial robots, also known as airborne robots are usually referred to as Unmanned Aerial Vehicles (UAVs).
- Underwater/Aquatic robots are usually called autonomous underwater vehicles (AUVs).
- Polar robots, designed to navigate icy, crevasse filled environments.

Locomotion Device:

- Legged robot: human-like legs, i.e., humanoid and android, or animal-like legs.
- Wheeled robot.
- Tracked robot.

Autonomy:

- Semi-autonomous robots perform desired tasks in unstructured environments without continuous human guidance.
- Tele-operated robots require continuous human intervention to complete a task.

Different robots are autonomous in different ways. A high degree of autonomy is particularly desirable in applications such as space exploration, cleaning floors, mowing lawns, waste water treatment, disaster response, and nuclear decommissioning.

1.3 Wheeled Mobile Robots (WMRs)

The most popular method of robot mobility has by far been provided by wheels. Wheels are used to propel many different sized robots and robotic platforms for different uses. Locomotion by wheel movement on a robot makes it a Wheeled Mobile Robot (WMR).

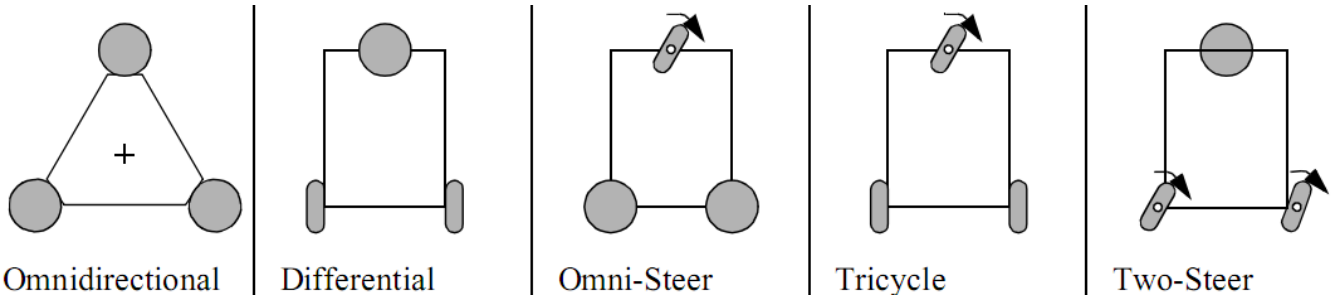


Figure 1.2: The five basic types of three-wheel configurations. The spherical wheels are castor wheels or Swedish and an arrow indicates a steering wheel [1].

Three-wheel configurations are quite common in WMRs. Five possible wheel configurations are shown in

Figure ***1.2***, these are omnidirectional, differential drive, omni-steer, tricycle and two-steer. The simplest and most common configuration for indoor robots is differential drive and will be the focus of study for this unit.

Chapter 2. Probabilistic Methods

2.1 Introduction to Probabilities

2.1.1 Definitions

2.1.1.1 Preliminaries

In probability theory, for any given experiment, an outcome is the result of that experiment, and the sample space, denoted by S, is the set of all possible outcomes. An event, denoted by A, is a set of outcomes that is also a subset of the sample space, i.e., $A \subset S$. A partition P of S is a set of nonempty disjoint subsets of S whose union is S, such that:

$$P = \left\{ A_i \subset S \;\middle|\; \begin{array}{c} \forall i,j \in \{1,\dots,n_p\},\ A_i \cap A_j = \emptyset, \\ \bigcup_{i=1}^{n_p} A_i = S \end{array} \right\} \tag{2.1}$$

The quantity $Pr\ (A)$ is the probability of event A to occur, and it satisfies the following three axioms of probability:

$$Pr\ (A) \geq 0, \quad \forall A \subset S, \tag{2.2}$$

$$Pr\ (S) = 1, \tag{2.3}$$

$$\bigcup_{i=1}^{n_p} \Pr(A_i) = \sum_{i=1}^{n_p} \Pr(A_i), \quad \forall A_i \in P. \tag{2.4}$$

From these axioms, the following consequences hold true:

$$Pr\ (\emptyset) = 0, \tag{2.5}$$

$$Pr\ (A) = 1 - \Pr(\neg A), \tag{2.6}$$

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B), \tag{2.7}$$

$$\Pr(A) \leq \Pr(B), \forall A \subset B. \tag{2.8}$$

2.1.1.2 Conditional Probability

Conditional probability is the likelihood of one event occurring given the occurrence of another event, and it is defined for any two events A, B as follows:

$$Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}, \quad (2.9)$$

where $\Pr(B) > 0$

2.1.1.3 Mutual Exclusive Events

Two events A, B are said to be mutually exclusive if A and B are disjoint sets, i.e., $A \cap B = \emptyset$. In terms of probability, mutually exclusive events correspond to the following:

$$\Pr(A \cap B) = 0, \quad (2.10)$$

$$\Pr(A \cup B) = \Pr(A) + \Pr(B), \quad (2.11)$$

$$\Pr(A|B) = 0, \quad (2.12)$$

$$\Pr(A|\neg B) = \frac{\Pr(A)}{1-\Pr(B)}. \quad (2.13)$$

2.1.1.4 Independent Events

Events A, B are *independent* if the occurrence of event A does not affect the occurrence of event B. This definition translates to probabilities as follows:

$$\Pr(A \cap B) = \Pr(A)\Pr(B), \quad (2.14)$$

$$Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A)\Pr(B), \quad (2.15)$$

$$\Pr(A|B) = \Pr(A). \quad (2.16)$$

Keep in mind that independence does not mean mutual exclusion and vice versa. In fact, for any nonempty mutual exclusive events are not independent, and vice versa.

2.1.1.5 Chain Rule for Probabilities

Conditional probabilities can be used to derive the chain rule for any two events $A, B \subset S$ as follows:

$$\Pr(A \cap B) = \Pr(A|B)\Pr(B). \quad (2.17)$$

In general, the chain rule for k events in S is:

$$\Pr\left(\bigcap_{i=1}^{k} A_i\right) = \prod_{i=1}^{k} Pr\left(A_i \middle| \bigcap_{j=1}^{i-1} A_j\right). \quad (2.18)$$

Note that A_i here is not in the partition P, otherwise the result is 0.

2.1.1.6 Law of Total Probability

The *law of total probability* computes the probability of event B given the set of k joint probabilities $Pr(B \cap A_i)$, where $A_i \in P$, and it is defined as:

$$\Pr(B) = \sum_{i=1}^{n_p} Pr(B \cap A_i), \tag{2.19}$$

$$\Pr(B) = \sum_{i=1}^{n_p} Pr(B \mid A_i) Pr(A_i). \tag{2.20}$$

2.1.1.7 Bayes' Theorem

An important consequence of law of total probability and conditional probability is *Bayes' theorem*, and it states the following:

$$Pr(A|B) = \frac{Pr(B\,|A) Pr(A)}{Pr(B)}, \tag{2.21}$$

where $Pr\,(B) > 0$. In general, if $A_i \in P$, then, Bayes' theorem corresponds to the following:

$$Pr(A_j|B) = \frac{Pr(B\,|A_j) Pr(A_j)}{\sum_{i=1}^{n_p} Pr(B \mid A_i) Pr(A_i)}. \tag{2.22}$$

2.1.2 Discrete Random Variables and Probability Distribution

A *random variable* is a function that associates each outcome in the sample space with a real number. Let X denote a random variable, then, by definition, $X : S \to \mathbb{R}$. A *discrete* random variable can take a countable number of distinct values, and it is generally associated with a *probability mass function* (PMF), denoted by $p_X\,(x)$, that return the probability of the random variable being exactly equal to some value $x \in \mathbb{R}$, such that:

$$p_X(x) = \Pr(X = x)\,, \tag{2.23}$$

$$\sum_{x=-\infty}^{\infty} p_X(x) = 1. \tag{2.24}$$

The *expected value*, or the *mean*, of a random variable is denoted by $E[X]$, and it is defined for a discrete random variable as follows:

$$E[X] = \mu_X = \sum_{i=1}^{\infty} p_X(x_i) x_i. \tag{2.25}$$

The *variance* of a random variable is defined as:

$$Var(X) = \sigma_X^2 = E[(X - \mu_X)^2] \tag{2.26}$$

$$= E[X^2] - \mu_X^2. \tag{2.27}$$

For discrete random variable, the variance is computed as follows:

$$Var(X) = \sum_{i=1}^{\infty} p_X(x_i)\,(x_i - \mu_X)^2$$

$$= \sum_{i=1}^{\infty} p_X(x_i)\, {x_i}^2 - \mu_X^2. \tag{2.28}$$

2.1.3 Continuous Random Variables and Probability Distribution

A *continuous* random variable takes an infinite number of possible values, and it is usually associated with a *probability density function* (PDF) that is denoted by $p_X\,(x)$, where $p_X(x) \leq\ 0; \forall x\ \in\ \mathbb{R}$. The probability of an event with continuous random variables is computed using the *cumulative distribution function* (CDF) which is defined as follows:

$$\Pr(X \leq x) = F_X(x) = \int_{-\infty}^{x} p_X(u)\, du \tag{2.29}$$

Note that $\lim_{x \to -\infty} F_X(x) = 0$ and $\lim_{x \to \infty} F_X(x) = 1$.

The expected value and the variance of a continuous random variable is defined in (2.30) and (2.31), respectively.

$$E[X] = \mu_X = \int_{-\infty}^{\infty} p_X(x)\, x\, dx, \tag{2.30}$$

$$Var(X) = \sigma_X^2 = \int_{-\infty}^{\infty} p_X(x)\,(x - \mu_X)^2 dx =$$

$$= \int_{-\infty}^{\infty} p_X(x)\, x^2 dx - \mu_X^2. \tag{2.31}$$

Moreover, the *support* of a continuous random variable is the smallest closed set at which the probability density function is not zero.

Conditional Distribution

Consider two continuous random variables X, Y with PDFs $p_X(x)$, $p_Y(y)$, respectively. Then, the *conditional* probability density function of X given $Y = y$ is:

$$p_X(x|Y=y) = p_{X|Y}(x|y) = \frac{p_{X,Y}(x,y)}{p_Y(y)}, \tag{2.32}$$

where $p_{X,Y}(x,y)$ is the density of the *joint probability distribution* of X and Y.

Chain Rule and Bayes' Theorem for Distributions

Let $p_{X|Y}(x|y)$ be the conditional distribution of X given $Y = y$, and $p_Y(y)$ is the PDF of Y, then, the *chain rule* describes the joint distribution of X and Y as:

$$p_{X,Y}(x,y) = p_{X|Y}(x|y)p_Y(y). \tag{2.33}$$

Note that this rule is applied to PDFs, and it is similar to the chain rule for probabilities in section 2.1.1.5. Moreover, Bayes' theorem for PDFs is defined as:

$$p_{X|Y}(x|y) = \frac{p_{X|Y}(x|y)\, p_X(x)}{p_Y(y)}. \tag{2.34}$$

Independent Random Variables

Two random variables X, Y are said to be independent if and only if:

$$p_{X,Y}(x,y) = p_X(x)p_Y(y), \tag{2.35}$$

where $p_X(x)$, $p_Y(y)$, are the PDFs of X, Y , respectively; and $p_{X,Y}(x,y)$ is the joint density of X and Y.

Marginal Distribution

The *marginal distribution* of the random variable X is defined as:

$$p_X(x) = \int_y p_{X,Y}(x,y)\ dy = \int_y p_{X|Y}(x|y)\, p_Y(y)\, dy. \tag{2.36}$$

This is equivalent to law of total probability defined in section 2.1.1, however, here it is applied to PDFs.

2.2 Gaussian Distributions

2.2.1 Introduction

A random variable X is *normally distributed*, or *Gaussian*, if its probability density function is defined as:

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} = \left(-\frac{(x-\mu_X)^2}{2\sigma^2}\right), \tag{2.37}$$

where, μ_X, σ^2 are the *mean* and *variance*, respectively; they are the distribution parameters. The notation $X \sim \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_X)$ means that the random variable X is Gaussian.

Multivariate Gaussian is a generalisation of univariate Gaussian by considering the random vector in $\mathbb{R}^n$, $X = [X_1\ X_2\ ...\ X_n]^T$,where every linear combination of its components is univariate Gaussian. In that case, the probability density function of X is defined as follows:

$$p_X(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}_X|}} exp\left(-\frac{1}{2}\ (\mathbf{x}-\boldsymbol{\mu}_X)^T \Sigma_X^{-1} (\mathbf{x}-\boldsymbol{\mu}_X)\right), \tag{2.38}$$

where, $\boldsymbol{\mu}_X \in \mathbb{R}^n$ is the *mean* vector, and $\boldsymbol{\Sigma}_X \in \mathbb{R}^{n\times n}$ is a symmetric positive definite *covariance* matrix. If $\boldsymbol{\Sigma}_X$ is singular, then, the distribution is *degenerate*. In terms of notation, $X \sim \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_X)$ is said to be a Gaussian random vector, or normally distributed random vector.

2.2.2 Properties

Gaussian distributions have a lot of useful properties and they are presented in this section, however, the derivations are omitted although they are easily obtained by applying the definitions in Chapter 2.1.3 with the Gaussian PDF.

Affine Transformation

Consider $X \sim \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_X)$ in $\mathbb{R}^n$, and let $Y = \mathbf{A}X + \boldsymbol{b}$ be the affine transformation, where $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{b} \in \mathbb{R}^m$. Then, the random vector $Y \sim \mathcal{N}(\boldsymbol{\mu}_Y, \boldsymbol{\Sigma}_Y)$ such that:

$$\begin{aligned} \boldsymbol{\mu}_Y &= \mathbf{A}\boldsymbol{\mu}_X + \mathbf{b}, \\ \boldsymbol{\Sigma}_Y &= \mathbf{A}\boldsymbol{\Sigma}_X\mathbf{A}^T. \end{aligned} \tag{2.39}$$

Sum of Independent Gaussian Distributions

Let $X \sim \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_X)$ and $Y \sim \mathcal{N}(\boldsymbol{\mu}_Y, \boldsymbol{\Sigma}_Y)$ be independent random vectors, then, the sum $\mathrm{Z} = \mathrm{X} + \mathrm{Y}$ is Gaussian such that $Z \sim \mathcal{N}(\boldsymbol{\mu}_Z, \boldsymbol{\Sigma}_Z)$ where:

$$\boldsymbol{\mu}_Z = \boldsymbol{\mu}_X + \boldsymbol{\mu}_Y,$$
$$\boldsymbol{\Sigma}_Z = \boldsymbol{\Sigma}_X + \boldsymbol{\Sigma}_Y \,. \tag{2.40}$$

Marginal Distribution

Let $X \sim \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_X)$ be a random vector in $\mathbb{R}^n$, that can be partitioned as:

$$X = \begin{bmatrix} U \\ V \end{bmatrix}, \quad \boldsymbol{\mu}_X = \begin{bmatrix} \boldsymbol{\mu}_U \\ \boldsymbol{\mu}_V \end{bmatrix}, \quad \boldsymbol{\Sigma}_X = \begin{bmatrix} \boldsymbol{\Sigma}_U & \boldsymbol{\Sigma}_{UV} \\ \boldsymbol{\Sigma}_{UV}^T & \boldsymbol{\Sigma}_V \end{bmatrix}. \tag{2.41}$$

Then, the *marginal distribution* of the random vectors U and V are Gaussians such that:

$$U \sim \mathcal{N}(\boldsymbol{\mu}_U, \boldsymbol{\Sigma}_U),$$
$$V \sim \mathcal{N}(\boldsymbol{\mu}_V, \boldsymbol{\Sigma}_V) \,. \tag{2.42}$$

Conditional Distribution

Following the partitioning defined in (2.41), the *conditional distribution* of $U|V \sim \mathcal{N}(\boldsymbol{\mu}_{U|V}, \boldsymbol{\Sigma}_{U|V})$, where:

$$\boldsymbol{\mu}_{U|V} = \boldsymbol{\mu}_U + \boldsymbol{\Sigma}_{UV}\boldsymbol{\Sigma}_V^{-1}(V - \boldsymbol{\mu}_V),$$
$$\boldsymbol{\Sigma}_{U|V} = \boldsymbol{\Sigma}_U - \boldsymbol{\Sigma}_{UV}\boldsymbol{\Sigma}_V^{-1}\boldsymbol{\Sigma}_{UV}^T \,. \tag{2.43}$$

Note that $\boldsymbol{\Sigma}_{U|V}$ is *Schur complement* of $\boldsymbol{\Sigma}_V$ in $\boldsymbol{\Sigma}_X$.

Chain Rule for Gaussian Distributions

Consider the conditional distribution $U|V \sim \mathcal{N}(\mathbf{A}V + \mathbf{b}, \boldsymbol{\Sigma}_{U|V})$, and the random vector $V \sim \mathcal{N}(\boldsymbol{\mu}_V, \boldsymbol{\Sigma}_V)$; then, the joint distribution of $X = [U^T \; V^T]^T$ is also Gaussian such that $X \sim \mathcal{N}(\boldsymbol{\mu}_X, \boldsymbol{\Sigma}_X)$, where:

$$\boldsymbol{\mu}_X = \begin{bmatrix} \mathbf{A}\boldsymbol{\mu}_V + \mathbf{b} \\ \boldsymbol{\mu}_V \end{bmatrix},$$
$$\boldsymbol{\Sigma}_X = \begin{bmatrix} \mathbf{A}\boldsymbol{\Sigma}_V\mathbf{A}^{\mathrm{T}} + \boldsymbol{\Sigma}_{U|V} & \mathbf{A}\boldsymbol{\Sigma}_V \\ (\mathbf{A}\boldsymbol{\Sigma}_{\mathrm{V}})^T & \boldsymbol{\Sigma}_V \end{bmatrix}. \tag{2.44}$$

2.3 Bayes Filter

Bayes filter is a probabilistic approach to recursively estimate some unknown PDFs as new information, such as measurements, becomes available. Consider, for example, a discrete dynamical system that is characterised by two probability distributions: (i) *state transition*, or *motion model* probability $p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k)$, and (ii) *measurement*, or *observation model* probability $p(\mathbf{z}_k|\mathbf{x}_k)$, where $\mathbf{x}$ is the *state* of the system, $\mathbf{u}$ is control input, $\mathbf{z}$ is the measurement, and k is the time step.

The quantities $\mathbf{x}_k$, $\mathbf{z}_k$, and $\mathbf{u}_k$ can be treated as either discrete or continuous random vectors. This filter, also known as *Recursive Bayesian estimator*, is used to estimate the *belief* of the system, denoted by $bel\,(\mathbf{x}_k)$, which it is defined as the probability distribution of the system state $\mathbf{x}_k$ conditioned on all available data: control inputs $U_k = \{\mathbf{u}_1, \dots, \mathbf{u}_k\}$ and measurements $Z_k = \{\mathbf{z}_1, \dots, \mathbf{z}_k\}$, such that:

$$bel(\mathbf{x}_k) = p(\mathbf{x}_k|U_k, Z_k) \tag{2.45}$$

This belief can be simplified to include the state transition distribution and measurement distribution. To do so use Bayes' Theorem defined in (2.34) to express the belief as:

$$\begin{aligned} bel(\mathbf{x}_k) &= \frac{p(\mathbf{z}_k|\boldsymbol{x}_k, U_{k,}Z_{k-1})\; p(\mathbf{x}_k|U_k, Z_{k-1})}{p(\mathbf{z}_k|Z_{k-1}, U_k)} \\ &= \eta\; p(\mathbf{z}_k|\boldsymbol{x}_k, U_{k,}Z_{k-1})\; p(\mathbf{x}_k|U_k, Z_{k-1}), \end{aligned} \tag{2.46}$$

where η is a normalisation constant that is independent of $\boldsymbol{x}_k$. From the measurement probability, the current measurement $\boldsymbol{z}_k$ is influenced only by the current state $\boldsymbol{x}_k$, thus:

$$p(\mathbf{z}_k|\mathbf{x}_k) = p(\mathbf{z}_k|\boldsymbol{x}_k, U_{k,}Z_{k-1}), \tag{2.47}$$

$$bel(\mathbf{x}_k) = \eta\; p(\mathbf{z}_k|\boldsymbol{x}_k)\; p(\mathbf{x}_k|U_k, Z_{k-1}) \tag{2.48}$$

Now, define $\overline{bel}\,(\boldsymbol{x}_k) = p\,(\boldsymbol{x}_k\,|U_k, Z_{k-1})$, and use the marginal distribution in (2.36) to expand $\overline{bel}\,(\boldsymbol{x}_k)$ as:

$$\begin{aligned} \overline{bel}(\mathbf{x}_k) &= p(\mathbf{x}_k|U_k, Z_{k-1}) \\ &= \int p(\mathbf{x}_k|\mathbf{x}_{k-1}, U_{k,}Z_{k-1})\; p(\mathbf{x}_{k-1}|U_{k,}Z_{k-1}) d\mathbf{x}_{k-1}. \end{aligned} \tag{2.49}$$

Note that the state transition probability follows *Markov assumption* where the current state $\boldsymbol{x}_k$ depends only on the immediate previous state and the current control input. Moreover, the dynamical system is *causal*, and current state does not depend on future inputs, thus:

$$\begin{aligned} \overline{bel}(\mathbf{x}_k) &= \int p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k)\; p(\mathbf{x}_{k-1}|U_{k-1,}Z_{k-1}) d\mathbf{x}_{k-1} \\ &= \int p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k)\; bel(\mathbf{x}_{k-1}) d\mathbf{x}_{k-1}\;. \end{aligned} \tag{2.50}$$

Finally, by putting everything together, (2.45) becomes:

$$bel(\mathbf{x}_k) = \eta\; p(\mathbf{z}_k|\mathbf{x}_k) \int p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k)\; bel(\mathbf{x}_{k-1}) d\mathbf{x}_{k-1}\;. \tag{2.51}$$

This is the recursive nature of the Bayes filter, that given a prior probability distribution of the system state, the posterior distribution is estimated using the state transition model and measurement model. In general, this process assumes the system executes a control action $\boldsymbol{u}_k$ first, and then, takes a measurement $\boldsymbol{z}_k$. Therefore, the Bayes filter has two steps: (i) *prediction* or *control update* to estimate $\overline{bel}\,(\boldsymbol{x}_k)$, and (ii) *correction* or *measurement update* to estimated $bel\,(\boldsymbol{x}_k)$.

Generally, during the control update step, the belief become less certain, i.e., increase the variance, due to the integration action. On the other hand, in the measurement update step, the belief certainty increases, i.e., the variance decreases, because of the conditional distribution.

2.3.1 Kalman Filter

Kalman filter is a special case of Bayes filter that works with linear Gaussian dynamical systems. It uses the properties of Gaussian distributions to produce a closed-form estimate of the system belief. The Kalman filter is a *parametric* approach where the belief is a PDF described by its parameters, such as the mean and covariance.

The state is a continuous random vector $\mathbf{x}_k \in \mathbb{R}^{n_s}$ such that $\mathbf{x}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. The state transition model and the measurement model are linear in $\boldsymbol{x}$ as defined in (2.52) and (2.53), respectively, where $\mathbf{u}_k \in \mathbb{R}^{n_i}$ is the control input vector, and $\mathbf{z}_k \in \mathbb{R}^{n_o}$ is the measurement vector.

$$\mathbf{x}_k = \mathbf{A}\mathbf{x}_{k-1} + \mathbf{B}\mathbf{u}_k + \mathbf{q}_k, \tag{2.52}$$

$$\mathbf{z}_k = \mathbf{C}\mathbf{x}_k + \mathbf{r}_k. \tag{2.53}$$

Note that $\mathbf{A} \in \mathbb{R}^{n_s \times n_s}$, $\mathbf{B} \in \mathbb{R}^{n_s \times n_i}$, and $\mathbf{C} \in \mathbb{R}^{n_o \times n_s}$. Moreover, $\mathbf{q}_k \in \mathbb{R}^{n_s}$ and $\mathbf{r}_k \in \mathbb{R}^{n_o}$ represent the noise in the system and measurement models. They are assumed to be zero mean, Gaussian white noise, i.e., $\mathbf{q}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k)$, and $\boldsymbol{r}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k)$, where $\mathbf{Q}_k$ and $\mathbf{R}_k$ are covariance matrices. In terms of probability distributions, the prior system belief, i.e., $bel(\boldsymbol{x}_{k-1})$, is defined in (2.54), the state transition model is in (2.55), and the measurement model is in (2.56).

$$\mathbf{x}_{k-1} | U_{k-1,} Z_{k-1} \sim \mathcal{N}(\boldsymbol{\mu}_{k-1}, \boldsymbol{\Sigma}_{k-1}), \tag{2.54}$$

$$\mathbf{x}_k | \mathbf{x}_{k-1}, \mathbf{u}_k \sim \mathcal{N}\left(\mathbf{A}\boldsymbol{\mu}_{k-1} + \mathbf{B}\mathbf{u}_{k,}\, \mathbf{Q}_k\right), \tag{2.55}$$

$$\mathbf{z}_k | \mathbf{x}_k \sim \mathcal{N}(\mathbf{C}\mathbf{x}_k, \mathbf{R}_k). \tag{2.56}$$

Similar to Bayes filter, Kalman filter follows the prediction and correction steps to estimate the belief by incorporating the control signal $\mathbf{u}_k$ and the measurement $\mathbf{z}_k$. However, since the system is linear, and all distributions are assumed to be Gaussian distributions, the properties in section 2.2 are used to derive estimation equations as shown next.

Prediction Step In this step, only the control input $\mathbf{u}_k$ is incorporated to estimate $\overline{bel}(\mathbf{x}_k)$. First, the chain rule in (2.44) is used to find the joint distribution of $\mathbf{x}_\mathrm{k} \mid U_k, Z_{k-1}$ and $\mathbf{x}_{\mathrm{k}-1} \mid U_{k-1}, Z_{k-1}$, from the prior belief in (2.54) and the state transition probability in (2.55) as follows:

$$\begin{bmatrix} \mathbf{x}_\mathrm{k} \mid U_k, Z_{k-1} \\ \mathbf{x}_{\mathrm{k}-1} \mid U_{k-1}, Z_{k-1} \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}_{t_1}, \boldsymbol{\Sigma}_{t_1}), \tag{2.57}$$

$$\boldsymbol{\mu}_{t_1} = \begin{bmatrix} \mathbf{A}\boldsymbol{\mu}_{\mathrm{k}-1} + \mathbf{B}\mathbf{u}_k \\ \boldsymbol{\mu}_{\mathrm{k}-1} \end{bmatrix} \tag{2.58}$$

$$\boldsymbol{\Sigma}_{t_1} = \begin{bmatrix} \mathbf{A}\boldsymbol{\Sigma}_{\mathrm{k}-1}\mathbf{A}^{\mathrm{T}} + \mathbf{Q}_k & \mathbf{A}\boldsymbol{\Sigma}_{\mathrm{k}-1} \\ (\mathbf{A}\boldsymbol{\Sigma}_{\mathrm{k}-1})^{\mathrm{T}} & \boldsymbol{\Sigma}_{\mathrm{k}-1} \end{bmatrix} \tag{2.59}$$

Then, define $\hat{\mathbf{x}}_k = (\mathbf{x}_k | U_k, Z_{k-1})$, and marginalise it using (2.41) such that:

$$\hat{\mathbf{x}}_k \sim \mathcal{N}(\hat{\boldsymbol{\mu}}_k, \hat{\boldsymbol{\Sigma}}_k), \tag{2.60}$$

$$\hat{\boldsymbol{\mu}}_k = \mathbf{A}\boldsymbol{\mu}_{\mathrm{k}-1} + \mathbf{B}\mathbf{u}_k, \tag{2.61}$$

$$\hat{\boldsymbol{\Sigma}}_k = \mathbf{A}\boldsymbol{\Sigma}_{\mathrm{k}-1}\mathbf{A}^{\mathrm{T}} + \mathbf{Q}_k. \tag{2.62}$$

Note that $\overline{bel}\,(\mathbf{x}_k) = p\,(\hat{\mathbf{x}}_k)$.

Correction Step The measurement $\mathbf{z}_k$ is incorporated by computing the joint distribution of $\mathbf{z}_k$ and $\hat{\boldsymbol{x}}_k$ using the chain rule, i.e., $p\,(\mathbf{z}_k, \hat{\boldsymbol{x}}_k) = p\,(\mathbf{z}_k\,|\hat{\boldsymbol{x}}_k)\,p\,(\hat{\boldsymbol{x}}_k)$, and then, rearranging the variables as follows:

$$\begin{bmatrix} \hat{\boldsymbol{x}}_k \\ \mathbf{z}_k \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}_{t_2}, \boldsymbol{\Sigma}_{t_2}), \tag{2.63}$$

$$\boldsymbol{\mu}_{t_2} = \begin{bmatrix} \hat{\boldsymbol{\mu}}_{\mathrm{k}} \\ \mathbf{C}\hat{\boldsymbol{\mu}}_{\mathrm{k}} \end{bmatrix}, \tag{2.64}$$

$$\boldsymbol{\Sigma}_{t_2} = \begin{bmatrix} \hat{\boldsymbol{\Sigma}}_{\mathrm{k}} & \hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}} \\ \mathbf{C}\hat{\boldsymbol{\Sigma}}_{\mathrm{k}} & \mathbf{C}\hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}} + \mathbf{R}_k \end{bmatrix}. \tag{2.65}$$

Finally, $p(\hat{\mathbf{x}}_k\,|\mathbf{z}_k)$ is computed using the conditional distribution in (2.43) as follows:

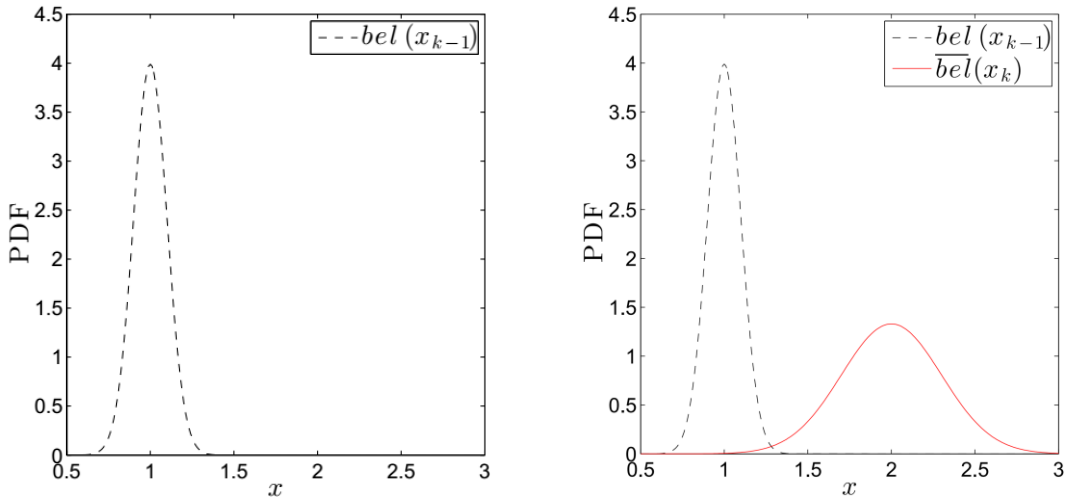
$$\hat{\mathbf{x}}_k|\mathbf{z}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{2.66}$$

$$\boldsymbol{\mu}_k = \hat{\boldsymbol{\mu}}_{\mathrm{k}} + \hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}} + \mathbf{R}_k\right)^{-1}(\mathbf{z}_k - \mathbf{C}\hat{\boldsymbol{\mu}}_{\mathrm{k}}), \tag{2.67}$$

$$\boldsymbol{\Sigma}_k = \hat{\boldsymbol{\Sigma}}_{\mathrm{k}} - \hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}} + \mathbf{R}_k\right)^{-1}\mathbf{C}\hat{\boldsymbol{\Sigma}}_{\mathrm{k}}. \tag{2.68}$$

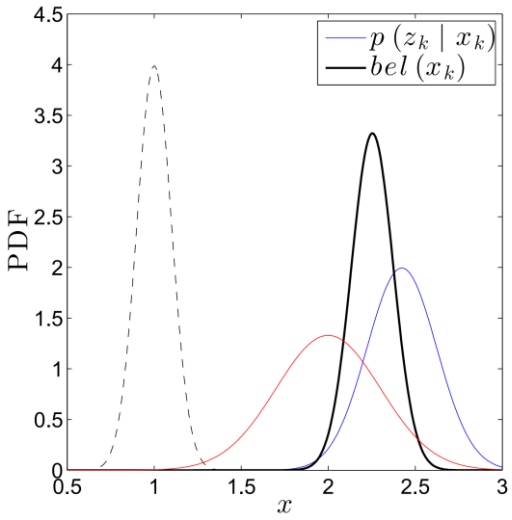
Note that $(\hat{\boldsymbol{x}}_k|\mathbf{z}_k) = (\mathbf{x}\,|U_k, Z_k)$, therefore, $bel\,(\mathbf{x}_{\mathrm{k}}) \sim \mathcal{N}\,(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. In Kalman filter convention, the quantity $\boldsymbol{K}_k = \hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\hat{\boldsymbol{\Sigma}}_{\mathrm{k}}\mathbf{C}^{\mathrm{T}} + \mathbf{R}_k\right)^{-1}$ is known as the *Kalman gain*.

Figure 2.1 illustrates the two steps of Kalman filter in $\mathbb{R}$. Note that the belief variance $\boldsymbol{\Sigma}_k$, also known as the *uncertainty*, is less than the variance after state transition $\hat{\boldsymbol{\Sigma}}_{\mathrm{k}}$ and measurement variance $\mathbf{R}_k$. The state transition and measurement models can be thought of as having two sources of information about the system state, and Kalman filter fuses these two estimates and use weighted average to compute the best estimate with variance smaller than both.



(a)

(b)



(c)

Figure 2.1 Kalman filter example in $\mathbb{R}$. (a) The initial system belief, (b) after integrating state transition model, the belief variance increases, and (c) by incorporating the measurement model, the resulting belief $bel\,(x_k)$ has variance less than $\overline{bel}\,(x_k)$ and $p\,(z_k\,|x_k)$

Keep in mind that not all dynamical systems are linear, nor the associated noises are Gaussians. In the case of the former, *Extended Kalman filter* (EKF) is a popular approach to be applied in case of nonlinear systems. If the noise is non Gaussian as well as the system is non linear, a nonparametric approach known as *particle filter*, or *Sequential Monte Carlo* (SMC), is used to estimate the belief by sampling the state space then assigning different weights to each sample.

Chapter 3. Perception. Proprioceptive and exteroceptive Sensors

3.1 Introduction

Autonomous mobile robots need to be able to acquire knowledge about their state and their environment. This is done by taking measurements using various sensors, whether proprioceptive or exteroceptive. The definition of perception, however, extends to the extraction of useful information from those measurements.

3.2 Rotary Encoders

Rotary optical encoders, are proprioceptive sensors, used to determine the angular position of the shaft they are attached to. Encoders in mobile robots are considered proprioceptive sensors because they only acquire information about the robot itself, not the structure of the environment. Essentially, it is a mechanical light chopper that produces a certain number of pulses for each shaft revolution as shown in Figure 3.1 (a). When an encoder is attached to the axle of each wheel in a differential-drive robot, it is possible to convert the number of pulses into useful information, such as the distance travelled by each wheel. By measuring the wheel diameter, the distance travelled by each wheel is calculated as follows:

$$distance = \left(\frac{counts}{CPR}\right)(\pi d) \tag{3.1}$$

where *d* is the wheel diameter, *CPR* is the count of pulses per revolution or the encoder's resolution, and *count* is the current number of pulses. Figure 3.1 (b) shows a common mechanism of connecting the encoder to track the movement of the wheel.

(a) (b)

Figure 3.1 Optical encoder. (a) Operation of optical encoder; (b) optical encoder connected to a motor and wheel © Pololu Corp.

3.3 IMUs

Another type of proprioceptive sensing device is the *inertial measurement unit* (IMU). An IMU is an electronic sensor that maintains a 6-degree-of-freedom (DOF) estimate of the relative pose of a mobile vehicle, this is, position in *x, y* and *z*, and orientation roll, pitch and yaw. Such task is achieved using a set of gyroscopes and accelerometers. IMUs are a common navigational component of aircrafts and ships, particularly for their capability to determine changes in the yaw, pitch and roll.

Figure 3.2 describes the general computation strategy followed by an IMU. In a general sense, the IMU makes use of three orthogonal accelerometers and gyroscopes. The data from the gyroscopes allows estimating the vehicle orientation. Simultaneously, the accelerometers serve to estimate the instantaneous vehicle acceleration. This data is in turn transformed using the information of the vehicle orientation relative to gravity such that the gravity vector can be estimated and extracted from the measurement. In this manner, the resulting acceleration is integrated once to obtain the vehicle velocity, and twice to obtain the position.

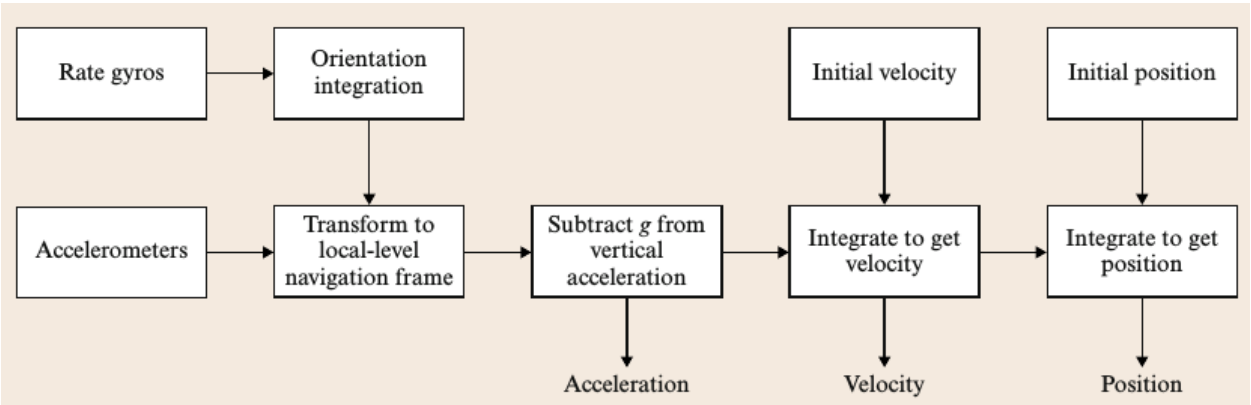


Figure 3.2 Schematic diagram of the functioning of an IMU.

Unfortunately IMUs are highly sensitive to measurement errors due to the usage of gyroscopes and accelerometers. For instance, errors in the gyroscopes readings and/or accelerometers may result in an incorrect cancellation of the gravity vector. Then, due to the double integration of the acceleration data, any residual gravity vector will result in a quadratic error in position. This problem tends to occur after a prolonged used of the mechanical components that form the IMU.

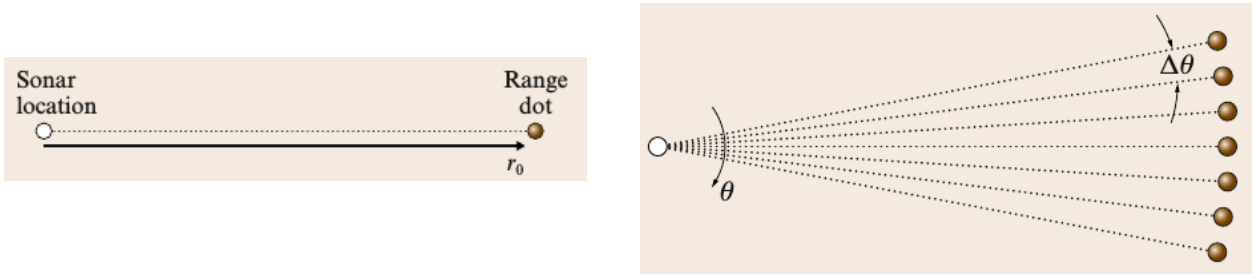
3.4 Sonars

Sonar is an exteroceptive sensor widely used in robotics. This device makes use of acoustic pulses and their echoes to measure the range distance to a certain object. Due to the fact that the speed of sound is usually known, there exists a directly proportional relationship between the speed of sound and the time of travel of an acoustic pulse.

Active sonar consists of a sound transmitter and receiver usually placed in the same position. The sonar emits an acoustic pulse, frequently called "ping", and then waits to listen for reflections (echo). Such acoustic pulse is usually generated electronically by means of a signal generator, power amplifier and electroacoustic transducer. In order to measure the distance to an object, the echo travel time t_o, also known as time-of-flight, is measured and converted into a range using the speed of sound c_s ($c_s = 343\ \mathrm{m/s}$ at standard temperature and pressure). In this manner,

$$r_o = \frac{c_s\, t_o}{2} \tag{3.2}$$

Being r_o the range in meters. The factor of 1/2 converts the round-trip distance to a range measurement. A graphical representation of the functioning principle of an active sonar is shown in Figure 3.3. Frequently sonars are equipped with an array of transmitter/receivers that allow determining several range distances simultaneously. Other types of sonars are equipped with a tilting/rotatory device that allows performing a sweep of range measurements under a certain resolution, as shown in Figure 3.3 (b).

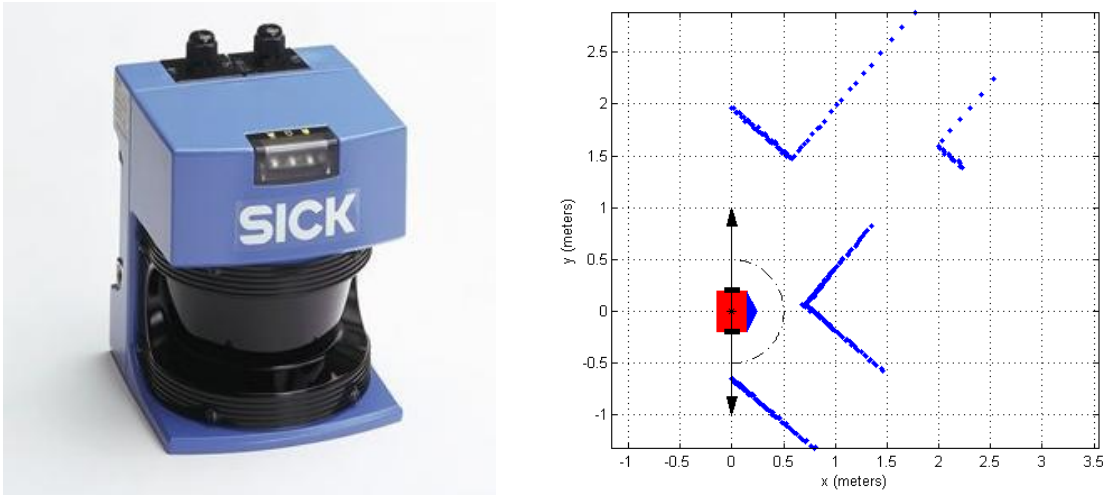


a) b)

Figure 3.3 Sonar functioning principle: a) a representation of the sonar location and a range dot, b) a sonar and an array of range measurements with a resolution of $\Delta\theta$.

3.5 Laser rangefinder

Another example of exteroceptive sensors is the laser rangefinder. It uses the principle of time-of-flight of electromagnetic wave to measure the distance between the sensor and obstacles in the environment. By rotating the sensor horizontally, it is possible to detect all obstacles around the robot within the range of the sensor, as shown in Figure 3.4 (b). These measurements can be used to build maps of the environment, which are essential for localisation and navigation.



(a) (b)

Figure 3.4 Laser rangefinder. (a) Industrial 180º laser rangefinder from Sick Inc., Germany; (b) Typical range of 2D laser rangefinder sensor with blue dots representing detected points in space.

In the robotic industry, there are many other types of sensors, each is used to measure some particular quantity or extract specific information either about the robot or its environment. It is important to note that all sensors are imperfect and their measurements are associated with some uncertainty, therefore, care must be taken when the extracted information is used for cognition and decision-making. In practice, all exteroceptive sensors onboard robots have finite range, i.e., they cannot measure the entire environment at once. Some sensors operate under line-of-sight principle, which

means they can perceive only partial information about the environment during every one cycle. Therefore, the mobile robot needs to be able to assimilate and integrate all measurements acquired over time to build a fairly complete image of the environment.

Chapter 4. Navigation

4.1 Differential Drive Robots

Also known as differential wheeled robots, these are mobile robots whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels, thereby requiring no additional steering motion.

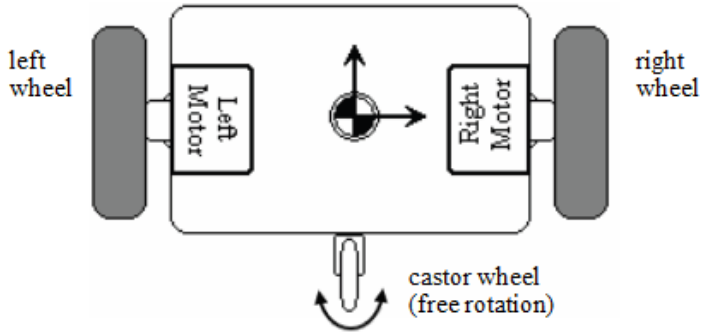


Figure 4.1 Wheel configuration of a three-wheeled differentially steered robot.

Stability of the robot on the ground is ensured by addition of one or more castor wheels, as in Figure 4.1, or ensuring that the centre of mass of the robot body is lower than the centre point of the line joining the two driving wheels. The latter design is however undesirable because the robot tends to oscillate and usually requires large wheel separation distances.

The wheels are separately powered to rotate at independent angular speeds ω_1 and ω_2. If both of the wheels are driven in the same direction and angular speed, the robot goes in a straight line, as shown in Figure 4.2. If both wheels are turned with equal speed in opposite directions the robot rotates about the central point of the axis joining the two wheels. Otherwise, the centre of rotation may fall anywhere on the line defined by the two contact points of the wheels depending on the speed and direction of rotation (Figure 4.2). While the robot is traveling in a straight line, the centre of rotation is at an infinite distance from the robot. Since the direction of the robot is dependent on the rate and direction of rotation of the two driven wheels, these two quantities should be sensed and controlled precisely.

4.1.1 Differential Drive Kinematics

The most common family of examples in mobile robotics arises from wheels that are required to roll in the direction they are pointing, without sliding sideways. This imposes a velocity constraint on rolling vehicles and the differential drive robot is not an exception. As a result, there are usually less

action variables than degrees of freedom or generalized coordinates. Such systems are therefore called *underactuated*. This leads to a formal concept known as *nonholonomic constraints*.

Mobile robot kinematics represent the relationship between the robot motor speeds (inputs) and the robot state. Mobile robot state may contain various parameters but the most important is called *pose* or *posture*, consisting the robot position and orientation with respect to a frame of reference (world frame). In order to derive the kinematics of a differential drive robot, it is instructive to first consider a simpler system; the unicycle.

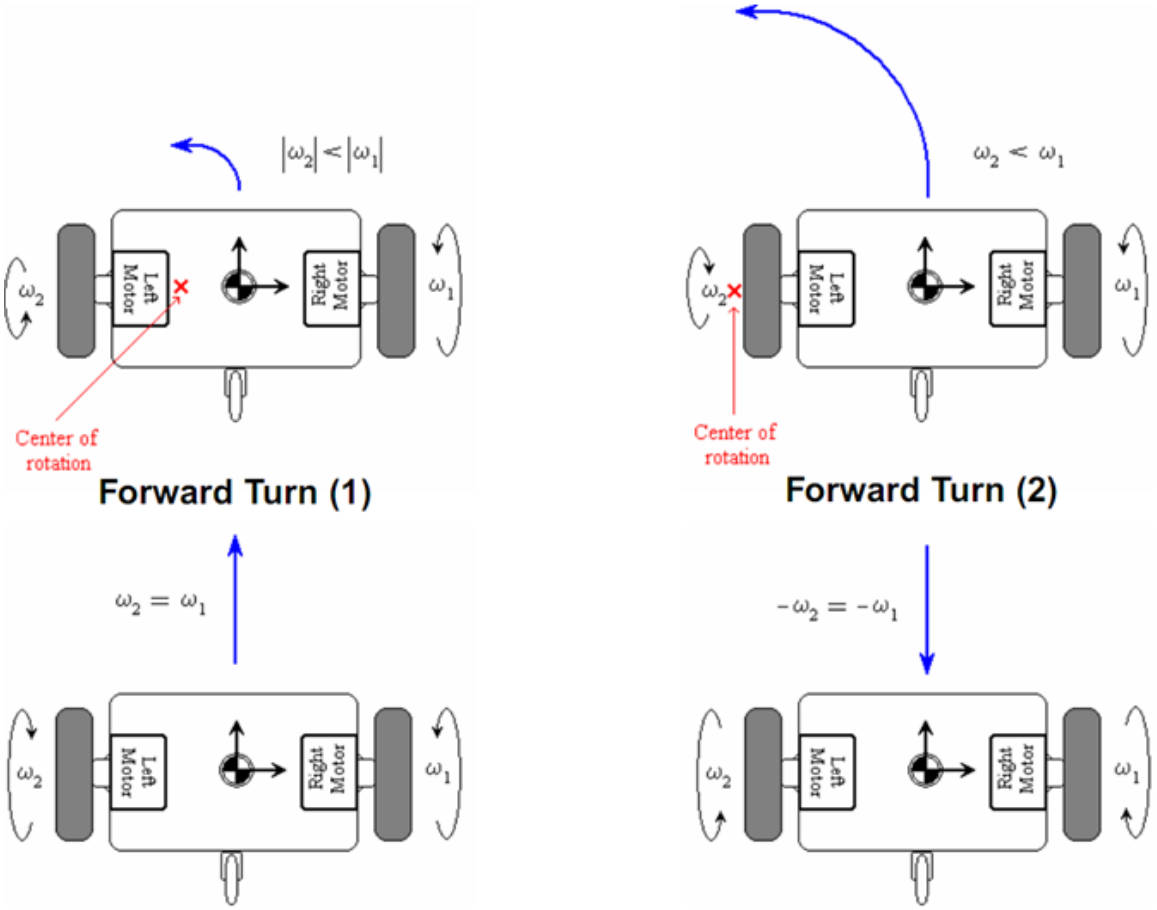


Figure 4.2 Differential Drive Robot direction determined by wheel speeds

4.1.2 Kinematics of a Unicycle

Consider the simple model of a unicycle, which is shown in Figure 4.3 in 2-dimensional space.

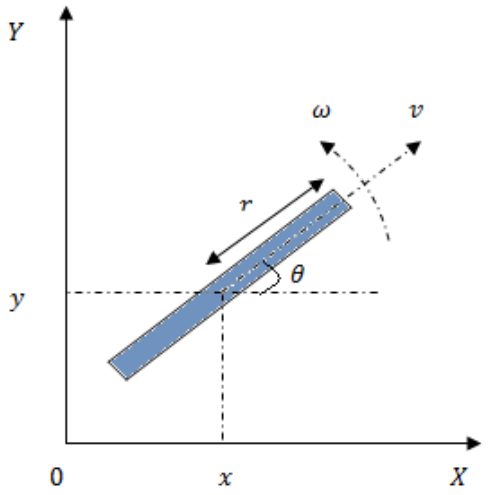


Figure 4.3 Unicycle seen from above.

Ignoring balancing concerns, there are two action variables, i.e., direct inputs to the system in the XY plane. The first one is the forward/linear velocity: $v = \omega_u r$ where ω_u is the wheel angular velocity, r is wheel radius whereas the second one is the steering velocity denoted by ω.

Hence, the set of equations:

$$\dot{x} = v \cdot cos\theta$$
$$\dot{y} = v \cdot sin\theta \quad (4.1)$$
$$\dot{\theta} = \omega$$

For (4.1) to be considered as kinematics equation of the unicycle, it must be shown to satisfy the nonholonomic constraint. The non-slip condition is derived from the fact that the velocity component in the direction perpendicular to the forward velocity is always zero, such that

$$\dot{x}sin\theta - \dot{y}cos\theta = 0 \quad (4.2)$$

Since (4.2), is satisfied by (4.1), then (4.1), fully represents the kinematics of the unicycle.

4.1.3 Kinematics of a differential drive

Consider a differential drive robot in the XY plane, with ω_l and ω_r as the angular velocities of the two driving wheels in Figure 4.4 and Figure 4.5. The wheels have a radius r with a distance of separation l. Assuming a perfectly symmetric body frame, point C - the centre of the axle between the wheels, may be considered as the robot's centre of mass and centre of rotation when the wheels are rotating with equal and opposite velocities.

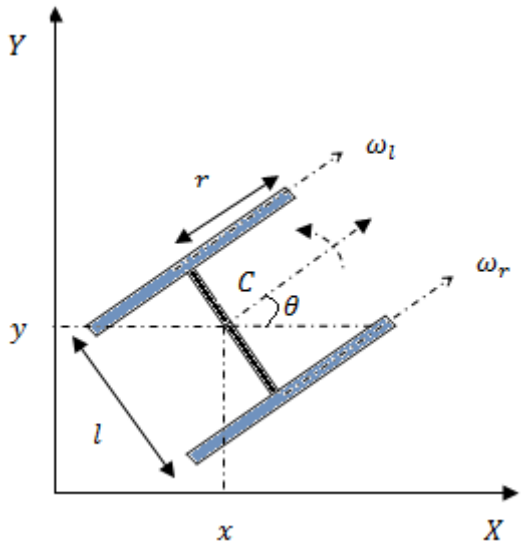


Figure 4.4 Parameters of a generic differential drive robot.

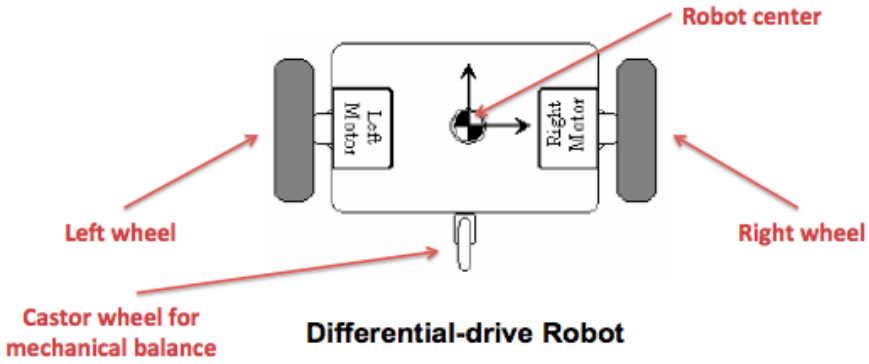


Figure 4.5 Differential-drive robot configuration

The resultant forward velocity through C may be reasoned as an average of the two forward wheel velocities given by $r(\frac{\omega_r+\omega_l}{2})$. The steering velocity may also be reasoned as proportional to the difference between wheel velocities but inversely proportional to distance between the wheels, i.e., $r(\frac{\omega_r-\omega_l}{l})$.

Thus, just like the unicycle, the configuration transition equations may be given as

$$\begin{aligned} \dot{x} &= r(\frac{\omega_r+\omega_l}{2})cos\theta \\ \dot{y} &= r(\frac{\omega_r+\omega_l}{2})sin\theta \\ \dot{\theta} &= r(\frac{\omega_r-\omega_l}{l}) \end{aligned} \tag{4.3}$$

Comparing (4.3) and (4.2) yields the transformation

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = \begin{bmatrix} r/2 & r/2 \\ r/l & -r/l \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix} \quad (4.4)$$

4.2 Motion Control of Wheeled Mobile Robots (WMR)

4.2.1 Introduction

The motion control for a mobile robot deals with the task of finding the control inputs that need to be applied to the robot such that a predefined goal can be reached in a finite amount of time.

Control of WMRs has been studied from several points of view, but essentially falls into one of the following three categories: point stabilisation, trajectory tracking, and path following.

Point Stabilisation

The objective here is to drive the robot to a desired fixed state, say a fixed position and orientation. Point stabilisation presents a true challenge to control system designers when the vehicle has nonholonomic constraints, since that goal cannot be achieved with smooth time-invariant state-feedback control laws.

Trajectory tracking

Also known as state tracking, the objective is driving the robot into following a time-parameterised state trajectory. Usually the reference trajectory is obtained by using a virtual reference robot in which case the kinematic constraints are implicitly considered by the reference trajectory. It turns out the trajectory tracking problem for WMRs is easier to solve than the stabilisation problem. In fact, the trajectory tracking problem for fully actuated systems is well understood and satisfactory solutions can be found in advanced nonlinear control textbooks. However, in case of underactuated systems, the problem is still a very active area of research.

Path Following

Here the vehicle is required to converge to and follow a path, without any temporal specifications. The underlying assumption in path following control is that the vehicle's forward speed tracks a desired speed profile, while the controller acts on the vehicle orientation to drive it to the path. Typically, smoother convergence to a path is achieved (when compared to the behaviour obtained with trajectory tracking control laws) and the control signals are less likely pushed to saturation.

4.2.2 Point stabilization

The goal can be defined in the simplest way as a set of coordinates in a multidimensional space. For instance, if the robot moving in a two dimensional space the goal is:

$\boldsymbol{X}_g = \begin{pmatrix} x_g \\ y_g \end{pmatrix}$, or $\boldsymbol{X}_g = \begin{pmatrix} x_g \\ y_g \\ \theta_g \end{pmatrix}$ if the heading needs to be controlled.

In order to build the controller we first need to define the robot parameters (inputs, outputs) and the goal. For a non-holonomic robot moving in a 2D environment the state can be represented by the position and heading $\boldsymbol{\rho}_r = \begin{pmatrix} x_r \\ y_r \\ \theta_r \end{pmatrix}$. The inputs are the linear and angular velocity of the robot $\boldsymbol{V}_r = \begin{pmatrix} V_r \\ \omega_r \end{pmatrix}$. The linear velocity of the robot, V_r, is always oriented in the direction of x axis of the robot reference frame because of the non-holonomic constraint (see Figure 4.6).

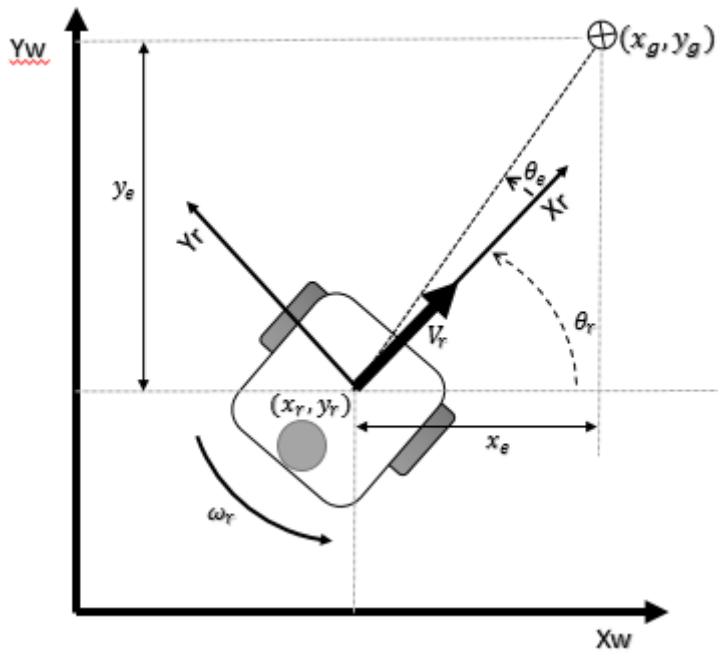


Figure 4.6

For the sake of simplicity, in this course, the goal is defined only by a 2D set of coordinates $\boldsymbol{X}_g = \begin{pmatrix} x_g \\ y_g \end{pmatrix}$.

Since the orientation of the robot at the goal is not considered for this course.

The next step for developing the control law is to compute the errors by using the goal coordinates and robot position as in the following.

$$e_x = x_g - x_r \tag{4.5}$$

$$e_y = y_g - y_r \tag{4.6}$$

$$e_\theta = atan2(e_y, e_x) - \theta_r \tag{4.7}$$

The robot position is assumed to be known and can be computed using various localisation techniques as it will be presented presented in chapter 6. Equations (4.5)-(4.7) can be represented in vector format as follows:

$$\boldsymbol{e} = \begin{pmatrix} e_x \\ e_y \\ e_\theta \end{pmatrix} \tag{4.8}$$

The general form of the control law can be written as:

$$\begin{pmatrix} V_r \\ \omega_r \end{pmatrix} = \boldsymbol{K} \begin{pmatrix} e_x \\ e_y \\ e_\theta \end{pmatrix}, \tag{4.9}$$

where

$$\boldsymbol{K} = \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \end{pmatrix}, \text{ is the control matrix.} \tag{4.10}$$

For simplicity the six controller gain parameters can be reduced to only two by defining the distance error:

$$e_d = \sqrt{e_x^2 + e_y^2} \tag{4.11}$$

The control law can now be written as:

$$V_r = K_d e_d \tag{4.12}$$

$$\omega_r = K_\theta e_\theta \tag{4.13}$$

A block diagram of the control loop is illustrated in Figure 4.7.

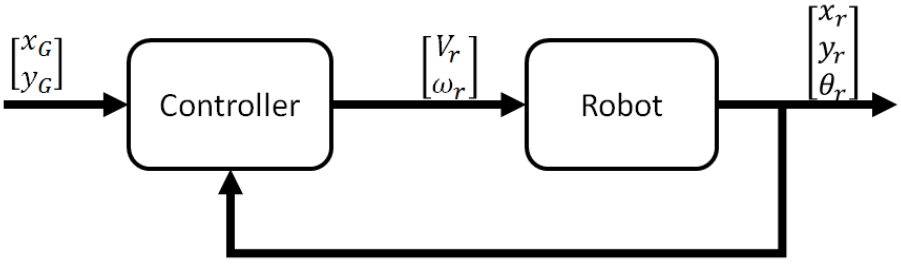


Figure 4.7 Point stabilisation control loop

Remarks:

1. The angle error e_θ has to be converted to a value in the interval $(-\pi, \pi)$ or $(0,2\pi)$ depending on the convention. In Matlab this can be done using '*wrapTo2Pi'.*

Example:

$$e_{\theta wrap} = wrapTo2PI(e_\theta) \text{ if } e_{\theta wrap} \in (0,2\pi);$$

$$e_{\theta wrap} = wrapToPI(e_\theta) \text{ if } e_{\theta wrap} \in (-\pi, \pi);$$

2. The robot will never reach the goal because error becomes zero (see Equation 4.5 and 4.6) and the control signal (linear velocity) becomes zero. To deal with this problem the goal is assumed to be reached if e_d is smaller than a threshold.

4.3 Reactive navigation (short term navigation)

Local obstacle avoidance is the process of modifying the robot trajectory by using the information provided by the sensors. The resultant motion depends on different factors, such as instantaneous sensor readings, goal position and relative location of the robot in the environment. In this section different algorithms that solve the obstacle avoidance problem will be presented. Such algorithms depend on the existence of a global map and the knowledge of the robot position in the environment.

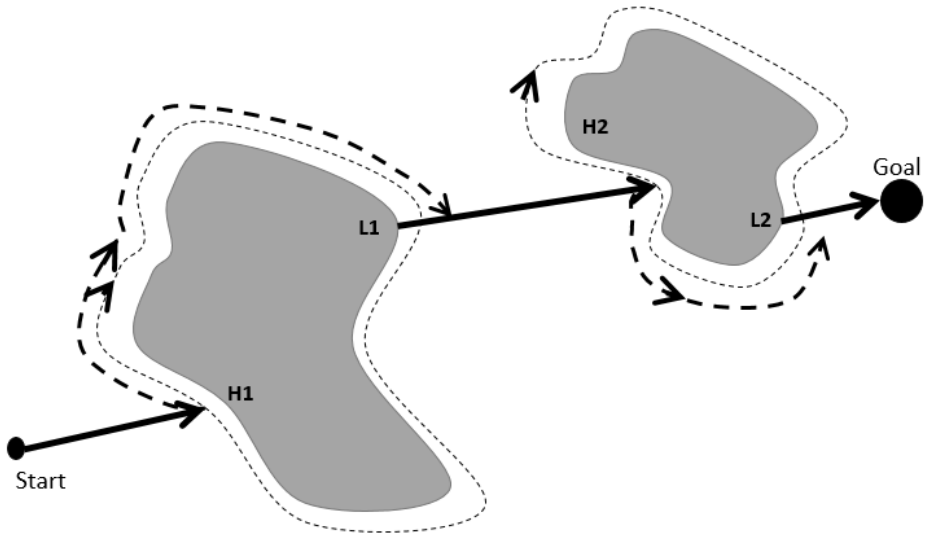
4.3.1 Bug algorithm

The simplest algorithm used to avoid obstacles is the Bug algorithm, this algorithm consists on making the robot to follow the contours of the obstacles until it reaches a point where it can continue its trajectory towards the goal (circumnavigating).

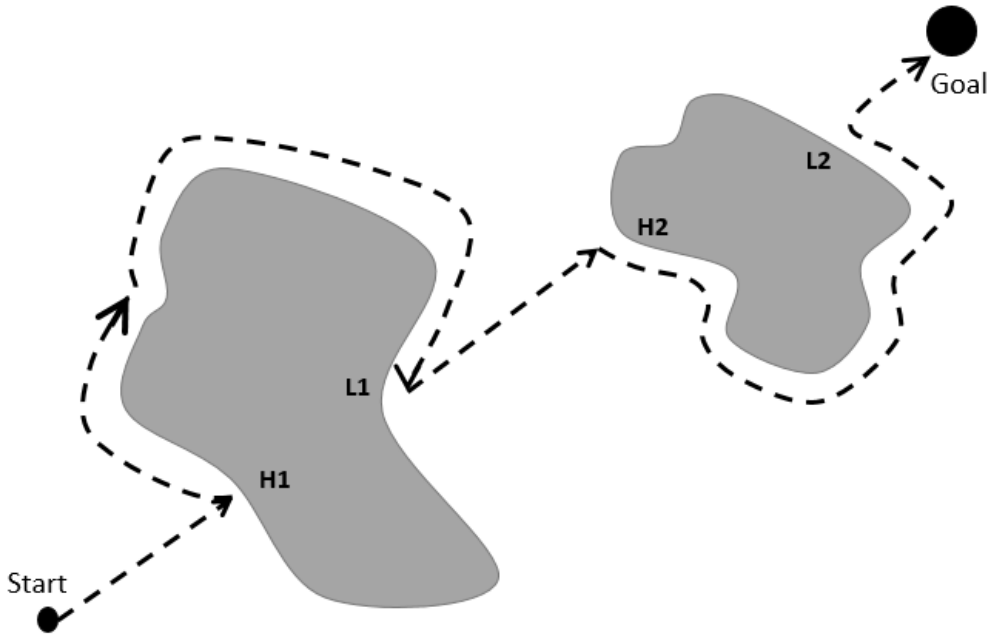
- **Bug 1** This algorithm consists on the robot to fully circumnavigate the complete obstacle one time, then choosing and departing from the point with the shortest distance towards the goal.
- **Bug 2** It consists on circumnavigating the obstacle and departing immediately after finding a point that allows it to move towards the goal.

As it can be seen from these two algorithms, the **Bug 1** is inefficient but it can guarantee the robot to reach any goal. On the other hand the **Bug 2** algorithm shortens the robot traveling but it can be inefficient (nonoptimal) in some situations.

Both algorithms are illustrated in Figure 4.8.



(a)



(b)

Figure 4.8 (a) Bug 1 algorithm with H1 and H2 as hit points and L1 and L2 as leave points. (b) Bug 2 algorithm with H1 and H2 as hit points and L1 and L2 as leave points.

4.3.2 Vector Field Histogram (VFH)

The Bug algorithm relies on the sensors to obtain information and perform obstacle avoidance; if these sensors do not provide enough information, it would be difficult to implement such an algorithm.

The VFH solves that problem by generating a probabilistic map of the environment around the robot. This map is generated using the information provided by the sensors in order to make an occupancy

grid (please refer to chapter 5). The algorithm sorts the measurements given by the sensors into a histogram as depicted in Figure 4.9 in order to calculate the probability that an obstacle is in the direction of the robot. Having obtained the histogram, the steering direction is then calculated. This is done by identifying the openings large enough for the vehicle to pass and then applying a cost function to the candidate openings and selecting the one with the lowest cost.

The cost function depends on the target orientation, wheel orientation and previous direction of the mobile robot.

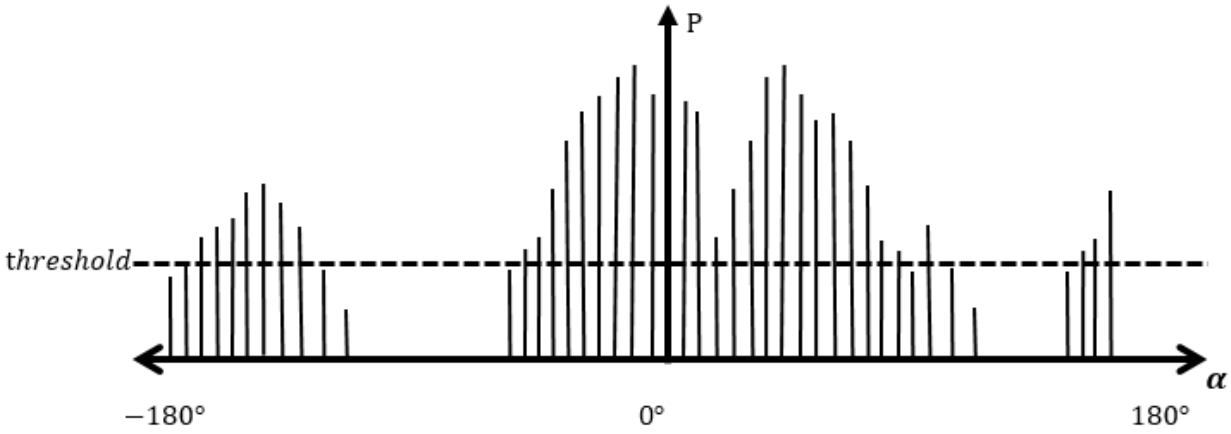


Figure 4.9. Polar Vector Field Histogram, α is the angle where the obstacle was found and P represents the probability of an obstacle in that direction, always taking into account the occupancy grids cell values [1].

Some improvements to this algorithm can be done considering the kinematic limitations of the robot resulting in masked histograms to block certain trajectories that cannot be done by the robot.

4.4 Path Planning for Mobile Robots

Path planning is the process of finding a sequence of actions to drive the mobile robot from an initial position to some desired goal position. Generally, the environment of the robot has a complex structure, for example: an autonomous vehicle driving in an urban area may have an environment that consists of roads, trees, buildings and other objects. For the path planning problem, it is common practice to simplify the environment into obstacles and free space. After that, the environment needs to be stored in a simple, abstract format to be used by path planning algorithms. In terms of the environment representation, the robotics research community developed several strategies to approach the path planning problem; these strategies include:

- Road map path planning methods such as: *visibility graph* and *Voronoi diagram*.
- Potential field path planning.
- Cell decomposition path planning.

Road map methods assume the obstacles in the environment to be in a continuous geometric description such as circles and polygons. These methods can provide optimal solutions in sparse, simple maps; however, they tend to be slow and inefficient when the environment is dense and dynamic.

In potential field method, a mathematical function is introduced to represent a field across the environment of the robot. This field directs the robot to the goal position from the current position. The method treats the robot as a point under the influence of an artificial potential field. The goal position acts as an attractive force on the robot and the obstacles act as repulsive forces. By computing the resultant force acting on this robot, the path of the robot can be obtained. This path planning method is easy to implement, and it copes well with changes in the environment; however, it does not guarantee optimality in terms of travelled distance and it might cause the robot to fall into a local minimum.

The cell decomposition method approaches the path planning problem by discretizing the environment into cells; each cell can either be an obstacle or free space. Then, a search algorithm is employed to determine the shortest path though these cells to go from the start position to the goal position. This strategy tends to work well in dense environments and some of its algorithms can handle changes in the environment efficiently.

This chapter focuses on the cell decomposition path planning. Section 4.4.1 introduces the graph representation of the environment; section 4.4.2 explains the basic concept of graph search algorithms, and section 4.4.3 focuses on a simple informed search algorithm that is used to find an optimal path for the planning problem.

4.4.1 Graph Representation of Mobile Robot Environments

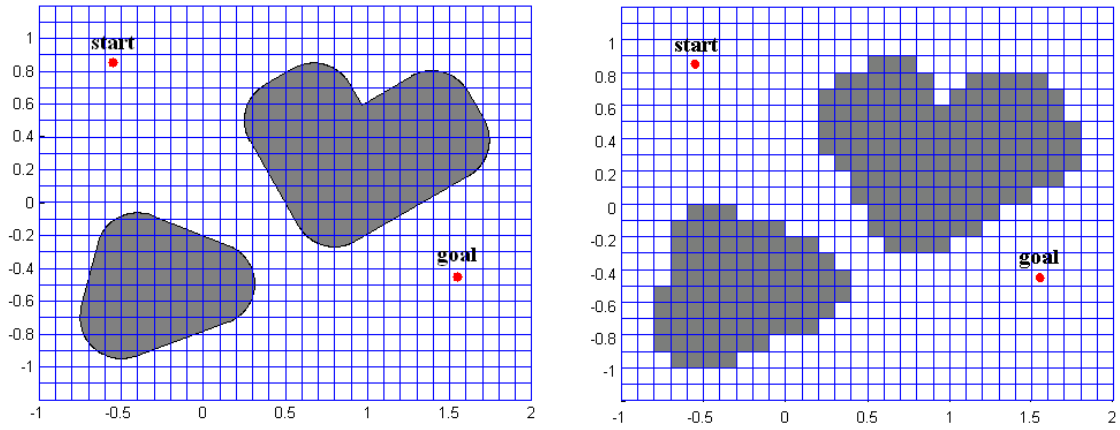
The environment of the robot is a continuous structure that is perceived by the robot sensors. Storing and processing this complex environment in a simple format can be quite challenging. One way to simplify this problem is by discretising the map using a grid. The grid discretises the world of the robot into fixed-cells that are adjacent to each other. Figure 4.10 shows common cell connection

geometries in 2D grid. With this discretisation, the robot motion is quantified by the distance to each adjacent cell. In 6-connected grid, the transition distance between any adjacent cells is the same. On the other hand, in 8-connected grid, the transition distance between diagonal adjacent cells is larger than the vertical or horizontal adjacent cells by a factor of $\sqrt{2}$.

(a) (b)

Figure 4.10 Common cell connection geometries in 2D grid based discretisation: (a) 8-connected grid; (b) 6-connected grid.

Grid-based discretisation results in an approximate map of the environment. If any part of the obstacle is inside a cell, then that cell is occupied; otherwise, the cell is free space. Figure 4.11 illustrates the discretisation of a 2D environment using 8-connect grid. Certainly, if the cell size decreases, the map becomes more accurate; nevertheless, the memory needed to store the map will increase quadratically. Therefore, it is important to select a cell size for a map that does not require large memory for storage, but captures most of the environment details.



(a) (b)

Figure 4.11 Discretisation of 2D environment using 8-connected grid: (a) Original map; (b) Discretised map.

Most grid-based path planning strategies use search algorithms to find the shortest path. Before the search algorithm is applied, the discretised map needs to be represented in an even more abstract

format; this format is known as a *graph*. From mathematics and computer science literature, a graph is a data structure that consists of two finite sets: a set of *nodes* and a set of *edges*. Each node is a storage block that contains some attributes, whereas edges are just links between different nodes. Edges in the graph may contain weights that represent transition costs between nodes. The graph in this case is called *weighted graph*. If the edges have directions between the nodes, the graph is called *directed graph*. An example of a weighted directed graph is a map of a country where each city represents a node in the graph, and the roads connecting these cities represent the edges of the graph. The length of each road is the weight associated to each edge in the graph. Figure 4.12 shows a weighted, directed graph with four nodes.

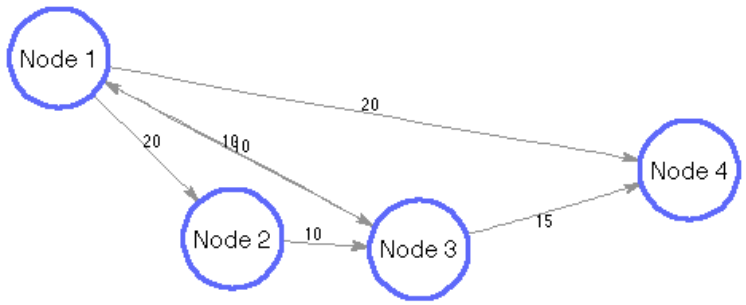


Figure 4.12 An example of weighted, directed graph.

Using this tool, it is possible to transform the grid based map into a weighted, directed graph, with the following notation: $G = (S, E)$, where S is the set of nodes that represent the possible robot locations or obstacle locations, and E is the set of edges that represent the transition cost between these locations. For mobile robots, the transition cost can be distance, time, fuel consumption, or a combination of these quantities. For simplicity, it is assumed, that transition costs represent distances only. The notation $c(s_1, s_2)$ represents the transition cost from node s_1 to node s_2.

When an 8-connected grid is used to discretize the map, the transition costs between adjacent cells is illustrated in Figure 4.11(a). The adjacent cells are called *neighbors*. If one of the neighbors is an obstacle, then the transition cost from, and to, this cell is infinity: $c(s_1, s_2) = c(s_2, s_1) = \infty$.

In a directed graph, each node has a set of predecessors and a set of successors. The predecessors of node s, denoted by $pred(s)$, are all the neighbor nodes that s may have come from. Whereas the successors of node s, denoted by $succ(s)$, are all the neighbor nodes that s may advance to. For example, in Figure 4.12, $pred(s_3) = \{s_1, s_2\}$, and $succ(s_3) = \{s_1, s_4\}$.

Each node in the graph is a data structure that can store basic attributes such as: the position of the node in the map, and the sets of predecessors and successors. The node can also store other attributes that are used by the search algorithm; these attributes may include: the distance from the start node and the estimated distance to the goal node (see sections 4.4.2 and 4.4.3.1 for more details).

4.4.2 Graph Search Concept

Once the graph is built, the path planning problem is transformed into a graph search problem to determine the shortest path between the start node s_{start} and the goal node s_{goal}. Almost all graph search algorithms follow the same generic algorithm with a *pseudocode* shown in Figure 4.13 Pseudocode is an informed high-level description of the operating principle of an algorithm that is easy to implement in a computer program, and **Main()** procedure is the starting point of the algorithm.

The basic idea of search algorithms is to maintain a list called *OPEN* that contains all nodes that are likely to be part of the shortest path. The *OPEN* list is initialised with only the start node s_{start}, and throughout the search more nodes are added to the list. Then, the algorithm selects a node *s* from *OPEN* list based on some criteria. This criteria is the major difference between the various search algorithms (see sections 4.3.3 for more details). If this node is the goal node s_{goal}, the algorithm terminates with success and it reconstructs the path, otherwise, all successors of *s* are added to *OPEN* list, and they label *s* as their predecessors. The process repeats until the goal node is reached, or until *OPEN* list becomes empty at which the algorithm terminates with failure, which means that solution does not exist and it is impossible to reach the goal node.

```
Main()
01      Initialize();
02      ComputeShortestPath();

Initialize()
03      Create OPEN list containing only the start node s_start;

ComputeShortestPath()
04      while (OPEN is not empty)
05              Select a node s from OPEN based on some criteria;
06              if (s = s_goal)
07                      Reconstruct the path from s_goal to s_start using predecessors;
08                      return SUCCESS;
09              else
10                      Remove node s from OPEN;
11                      for all s' ∈ successors of s
12                              Insert node s' into OPEN;
13                              Set node s as predecessor of node s';
14      return FAILURE;
```

Figure 4.13 Generic search algorithm.

The graph search algorithms are classified into two groups based on the criteria in which a candidate node is selected from *OPEN* list: uninformed search and informed search.

In uninformed search, only the basic attributes of each node, such as predecessors and successors, are used for selecting a node from *OPEN* list. Breadth-First Search (BFS) and Depth-First Search (DFS) are popular examples of uninformed search algorithms. These algorithms are easy to implement; however, they ignore the other attributes of the node, such as transition cost between nodes and the distance from the start node. Hence, they do not guarantee the optimality of the solution unless the transition cost is uniform across the graph. Moreover, these algorithms tend to be inefficient in terms of processing time and memory usage when applied to large maps.

Informed search algorithms, on the other hand, use the transition cost between nodes, the distance from the start node, an estimate to the goal node, or a combination of these attributes, to select the most promising node from *OPEN* list. These algorithms guarantee optimality in weighted graphs and they are more efficient than uninformed search algorithms in terms of processing time. Some informed search algorithms are Dijkstra's algorithm, A* algorithm, Lifelong Planning A* (LPA*) algorithm, and D* Lite algorithm. Dijkstra's algorithm is discussed in details in section 4.4.3.1.

4.4.3 Informed Search Algorithms

Informed search algorithms use node attributes other than predecessors and successors to select the most promising node from *OPEN* list. These attributes are encapsulated in a structure called key. The key is an evaluation function that guides the search to an optimal path and it makes informed search superior to uninformed search in terms of efficiency. It is important to note that there might be more than one path with the shortest distance in the same graph. Dijkstra's algorithm and A* algorithm are classical informed search algorithms that find an optimal path in a static map where the environment is assumed to remain unchanged. Lifelong Planning A* (LPA*) algorithm and D* Lite algorithm are relatively new algorithms that are optimal and efficient in planning and replanning when the environment is dynamic and the map is constantly changing.

4.4.3.1 Dijkstra's Algorithm

Dijkstra's algorithm was developed by E. Dijkstra in 1959 to find the shortest path between two fixed nodes (s_{start}, s_{goal}) in a weighted graph. The graph is assumed to be static where the transition costs between the nodes remain constant. This corresponds to a complete knowledge of the environment prior to the search process. The algorithm requires each node in the graph to have three attributes: (a) transition cost from the start node s_{start} to node s denoted by $g(s)$, (b) one predecessor of node s, and (c) the set of successors of node s.

$$node\{s\} = \begin{Bmatrix} g(s) \\ pred(s) \\ succ(s) \end{Bmatrix} \quad (4.14)$$

Because the algorithm belongs to the informed search family, it uses an evaluation function, or a key, to select a tentative node from *OPEN* list. In Dijkstra's algorithm, the evaluation function is defined as:

$$\text{key} = g(s) \quad (4.15)$$

In addition to *OPEN* list, which contains tentative nodes to be explored, Dijkstra's algorithm maintains another list called *CLOSED* to keep track of the nodes that have been already explored. Dijkstra's algorithm is summarised in Figure 4.14. It is initialised by assigning a value of infinity to $g(s)$ of each node in the graph except the start node which gets a value of zero; that corresponds to the transition cost from start node. Then, the start node s_{start} is placed in *OPEN* list, with *CLOSED* list being empty. The search starts by selecting the node with the minimum key value, i.e., $u = \arg\min_s \big(g(s)\big)$; if *u* is the goal node s_{goal}, the search is terminated and the path is reconstructed; otherwise *u* is removed from *OPEN* list and it is added to *CLOSED* list. After that, each successor of *u* is examined: if the successor node is neither in *CLOSED* list nor in *OPEN* list, it is added to *OPEN* with *u* as a predecessor and $g(successor)$ is set to equal to $\big(g(u) + c(u, successor)\big)$. If the successor node is not in *CLOSED* list but it is in *OPEN* list, then search needs to make sure it assigns each node with the shortest path from s_{start}, therefore, the old $g(s)$ of the successor is compared to the new tentative $g(s) = \big(g(u) + c(u, successor)\big)$, if the tentative $g(s)$ is less that the old value, then the tentative value replaces the old value and *u* becomes the predecessor of this node. The search will repeat the process until the goal node is reached at which the path is reconstructed by tracing back the predecessors from goal node to start node.

Dijkstra's algorithm is similar to a wavefront that propagates in all directions from the start node until the goal node is reached as shown in Figure 4.15. The search spends a considerable amount of computation in directions other than the direction of the goal node, which might be inefficient in large maps with remote start and goal nodes.

The pseudocode uses the following function to manage *OPEN* list: *OPEN*.Insert(s) inserts node s into *OPEN* list, and *OPEN*.Remove(s) removes node s from *OPEN* list.

```
Main()
01      Initialize();
02      ComputeShortestPath();

Initialize()
03      for all s ∈ S
04              g(s) = ∞;
05              pred(s) = ∅;
06      g(s_start) = 0;
07      pred(s_start) = 0;
08      OPEN = ∅;
09      CLOSED = ∅;
10      OPEN.Insert(s_start);

ComputeShortestPath()
11      while (OPEN ≠ ∅)
12              u = arg min_{s∈OPEN} (CalculateKey(s))
13              if (u = s_goal)
14                      ReconstructPath();
15                      return SUCCESS;
16              OPEN.Remove(u);
17              CLOSED.Insert(u);
18              for all s' ∈ succ(s)
19                      if (s' ∉ CLOSED)
20                              g_tentative(s') = g(u) + c(u, s')
21                              if (s' ∉ OPEN)
22                                      OPEN.Insert(s');
23                                      g(s') = g_tentative(s');
24                                      pred(s') = u;
25                              else
26                                      if (g_tentative(s') < g(s'))
27                                              g(s') = g_tentative(s');
28                                              pred(s') = u;
29      return FAILURE;

CalculateKey(s)
30      return g(s);

ReconstructPath()
31      path = [s_goal];
32      u = pred(s_goal);
33      while (u ≠ 0)
34              path = [u, path];
35              u = pred(u);
36      return path;
```

Figure 4.14 Dijkstra's Algorithm

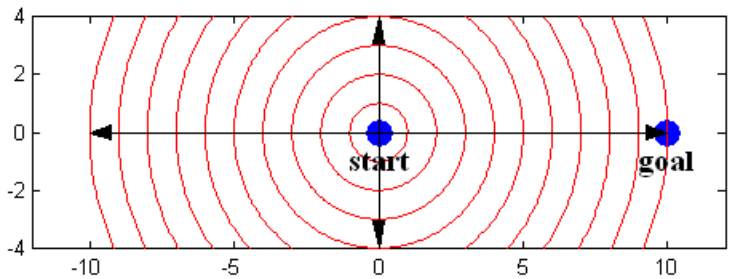


Figure 4.15 Dijkstra's search is similar to a wavefront in all directions.

Chapter 5. Mapping

5.1 Introduction

One of the vital, yet challenging, tasks for mobile robots is mapping an environment with unknown structure, where the robot needs to navigate and build a properly represented map. Acquiring an accurate map of the environment is an important element in mobile robot navigation and motion planning where the robot traverses the environment reliably and efficiently.

Before considering the different approaches to robotic mapping, it is essential to define the robot internal representation of the map. One popular representation in the robotics literature is *metric maps* where the geometric properties of the environment are captured with reference to some fixed frame of reference called *world frame* and denoted by $\{W\}$. These properties are usually represented by geometric objects such as: points, lines, planes, circles, cylinders, etc., or they can simply represent regions in the environment with obstacle, and other regions with empty space. Linear, planar and circular geometric shapes are ubiquitous in man-made environments, and sensors that return dense point cloud, such as laser scanners, stereo cameras, and structured-light scanner, can exploit the environment structure to build a compact representation of the map by detecting simple geometric shapes.

Note that the robot has to move while building the map, and since the measurements are usually relative to the robot frame $\{R\}$, transformation to the world frame $\{W\}$ requires an accurate knowledge of the robot pose; this process is known as the *localisation* problem and it is discussed in details in chapter 6.0. In this chapter it is assumed that the robot has perfect knowledge of its pose, i.e., localisation problem is assumed solved. Also, the environment is assumed static where the only moving object is the mobile robot.

The probabilistic methods for robotic mapping rely on studying the propagation of probabilistic distributions of the sensor noise and the unknown landmark locations. One of these methods is *occupancy grid mapping*, which relies on Bayes' theorem to recursively estimate the map as new measurements become available. This method is presented in the section 5.2.

5.2 Occupancy Grid

In occupancy grid, the map is represented by a grid over the environment space, e.g., 2D or 3D space, and each cell of the grid corresponds to the probability of the cell being occupied. The posterior probability of each cell in the grid is updated every time a new measurement, such as distance from laser scanner, is obtained using the previous estimate in term of *logarithmic odds* for the purpose of computational efficiency. Of course, grids with high resolution result in more accurate maps, however, that comes at an expensive computational cost.

Figure 5.1 shows an example of occupancy grid in 2D. Each cell is denoted by $c_{i,j}$ where i represents the x-axis index and j represent the y-axis index. This grid can be represented by a matrix flipped vertically, where i corresponds to the column, and j corresponds to the row.

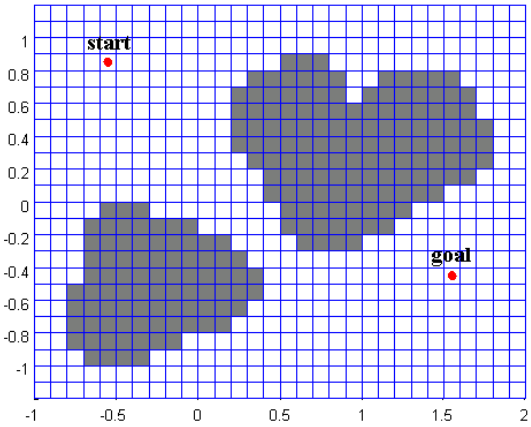


Figure 5.1: Occupancy grid in 2D with cell resolution of $0.1m \times 0.1m$. Dark regions correspond to high probability of occurrence, whereas bright regions correspond to low probability of occurrence.

The value of each cell represents the probability of occupancy $\Pr(c_{i,j} = occupied)$. For example, as shown in Figure 5.2, $\Pr(c_{1,3} = occupied) = 0.9$ means that cell $c_{1,3}$ has 90% chance to be occupied, and $\Pr(c_{2,2} = occupied) = 0.5$ means that cell $c_{2,2}$ has 50% chance to be occupied.

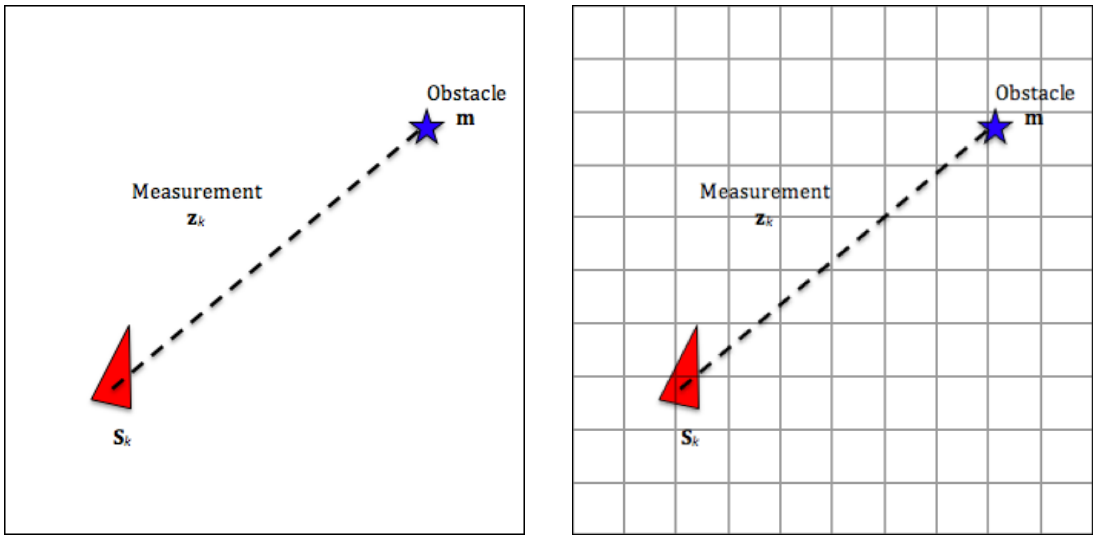
$j = 4$	0.5	0.5	0.5	0.5
$j = 3$	0.9	0.9	0.2	0.1
$j = 2$	0.8	0.5	0.2	0.1
$j = 1$	0.7	0.6	0.2	0.1
	$i = 1$	$i = 2$	$i = 3$	$i = 4$

Figure 5.2: Occupancy grid in 2D with probability of occupancy $\Pr(c_{i,j} = occupied)$.

Occupancy grid mapping requires a prior knowledge of robot pose at any time step, which can be acquired by means of encoders or IMUs. In this course, it will be assumed that the robot is equipped with a range-bearing sensor such as LIDAR or sonar.

The occupancy grid is initialized with no prior knowledge about the environment where all cells have probability of 0.5. Then, as the robot moves and observes obstacles in the environment, each corresponding cell is updated according to some strategy.

For the sake of illustration, consider a mobile robot moving in a 2D environment as shown in Figure 5.3 (a), which is equipped with a LIDAR. Also, consider an occupancy grid overlaid on the environment as shown in Figure 5.3 (b). The next two sections present two methods for updating the occupancy grid.



(a) (b)

Figure 5.3: (a) Mobile robot with LIDAR measuring an obstacle **m**. (b) The occupancy grid overlaid on the environment.

5.2.1 Binary Mapping

In binary mapping method, the occupancy grid is updated in the region starting from the robot along the measurement and finishing at the obstacle. Using the robot pose $\mathbf{s}_k$, the measurement $\mathbf{z}_k$ is projected in the world frame $\{W\}$, and the probability of the corresponding cell $c_{i,j}^{\mathbf{m}}$ is assigned a value of 1. The probability of all cells between $\mathbf{m}$ and $\mathbf{s}_k$ are assigned a value of 0. By applying this method to the example in Figure 5.4, the resulting occupancy grid map is shown in Figure 5.4.

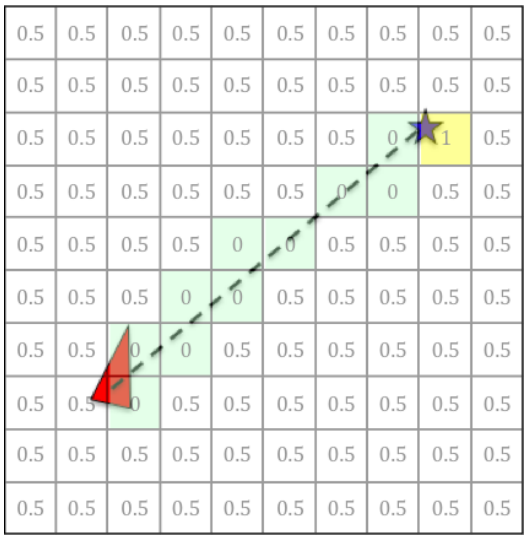


Figure 5.4: Occupancy grid after applying binary mapping method. Only cells that intersect with the measurement are updated. Green regions are empty space, whereas yellow cell is occupied.

To compute the obstacle position $\mathbf{m}$, the measurement $\mathbf{z}_k$ and the robot pose $\mathbf{s}_k$ is used as follows:

$$m_x = s_{x,k} + z_{\rho,k}\cos(s_{\theta,k} + z_{\alpha,k}), \tag{5.1}$$

$$m_y = s_{y,k} + z_{\rho,k}\sin(s_{\theta,k} + z_{\alpha,k}). \tag{5.2}$$

To update the relevant cells in the occupancy grid, it is important to convert any Cartesian coordinate, such as robot pose $\mathbf{s}_k$ and the obstacle position $\mathbf{m}$, to indices in the occupancy grid, which take the form of rows i and columns j. For example, the cell corresponding to the obstacle $\mathbf{m}$ is $c_{i,j}$ and the indices are computed as follows:

$$i = \left\lfloor \frac{m_x - ll_x}{res} \right\rfloor + 1, \tag{5.3}$$

$$j = \left\lfloor \frac{m_y - ll_y}{res} \right\rfloor + 1, \tag{5.4}$$

Where $\lfloor \dots \rfloor$ is the floor function, $\boldsymbol{ll}$ is the lower left corner in meters of the occupancy grid with respect to the world frame $\{W\}$, and res is the cell resolution in meters.

To determine the indices of the cells between the robot pose $\mathbf{s}_k$ and the obstacle position $\mathbf{m}$, an algorithm called *Bresenham Line Algorithm* is used. In the MATLAB simulator used for this course, the algorithm is implemented in the function called "*getBresenhamLine*".

5.2.2 Probabilistic Mapping

The probabilistic mapping method follows the same steps of selecting the cells to update as described in binary mapping. However, this probabilistic method uses *Bayes' theorem* to update the relevant cells by employing the sensor probabilistic model.

Assume the LIDAR has the probability of correctly detecting an obstacle denoted by $\Pr(\mathbf{z}_k|c_{i,j} = occupied)$. But the sensor can also make mistakes and it can detect obstacle even when there is no obstacle, this event has a probability denoted by $\Pr(\mathbf{z}_k|c_{i,j} \neq occupied)$. If the prior probability of the cell being occupied $\Pr(c_{i,j} = occupied)$ is known, then, Bayes' theorem can be used to compute the posterior probability that the cell is occupied conditioned on the measurements $\mathbf{z}_k$. This posterior probability is denoted by $\Pr(c_{i,j} = occupied|\mathbf{z}_k)$ and it is computed as follows:

$$\Pr(c_{i,j} = occupied|\mathbf{z}_k) = \frac{\Pr(\mathbf{z}_k|c_{i,j}=occupied)\cdot\Pr(c_{i,j}=occupied)}{\Pr(\mathbf{z}_k)}. \tag{5.5}$$

The denominator $\Pr(\mathbf{z}_k)$ can be computed using the *Law of Total Probability* as follows:

$$\begin{aligned}\Pr(\mathbf{z}_k) = \ & \Pr(\mathbf{z}_k|c_{i,j} = occupied)\cdot\Pr(c_{i,j} = occupied) \\ & + \Pr(\mathbf{z}_k|c_{i,j} \neq occupied)\cdot\Pr(c_{i,j} \neq occupied)\end{aligned} \tag{5.6}$$

Note that:

$$\Pr(c_{i,j} \neq occupied) = 1 - \Pr(c_{i,j} = occupied). \tag{5.7}$$

The same approach is used to update the empty cells between the robot $\mathbf{s}_k$ and the obstacle $\mathbf{m}$, where the probability $\Pr(c_{i,j} \neq occupied|\mathbf{z}_k)$ is computed as follows:

$$\Pr(c_{i,j} \neq occupied|\mathbf{z}_k) = \frac{\Pr(\mathbf{z}_k|c_{i,j}\neq occupied)\cdot\Pr(c_{i,j}\neq occupied)}{\Pr(\mathbf{z}_k)}. \tag{5.8}$$

For instance, consider the example shown in Figure 5.3. If $\Pr(\mathbf{z}_k|c_{i,j} = occupied) = 0.85$ and $\Pr(\mathbf{z}_k|c_{i,j} \neq occupied) = 0.22$; then, the occupancy grid will have the values shown in Figure 5.5.

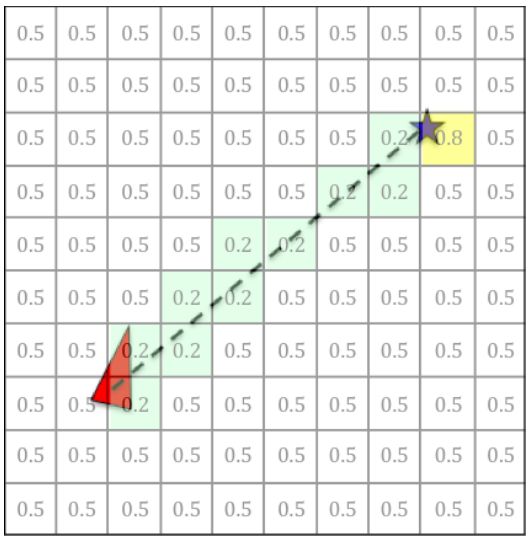


Figure 5.5: Occupancy grid after applying probabilistic mapping method. Green regions have low probability of obstacle occurrence, whereas yellow cell has high probability of obstacle occurrence.

For the sake of computational efficiency, the probabilistic occupancy grid mapping is generally addressed in terms of *probability odds*, which is defined as follows:

$$odd(c_{i,j} = occupied|\mathbf{z}_k) = \frac{\Pr(c_{i,j}=occupied|\mathbf{z}_k)}{\Pr(c_{i,j}\neq occupied|\mathbf{z}_k)}. \tag{5.9}$$

By substituting (5.5) and its complement in equation (5.8), the odds equation (5.9) becomes:

$$\begin{aligned} odd(c_{i,j} = occupied|\mathbf{z}_k) &= \frac{\Pr(\mathbf{z}_k|c_{i,j} = occupied)\cdot \Pr(c_{i,j} = occupied)}{\Pr(\mathbf{z}_k|c_{i,j} \neq occupied)\cdot \Pr(c_{i,j} \neq occupied)} \\ &= odd(\mathbf{z}_k|c_{i,j} = occupied)\cdot odd(c_{i,j} = occupied), \end{aligned} \tag{5.10}$$

Where:

$$odd(c_{i,j} = occupied) = \frac{\Pr(c_{i,j}=occupied)}{\Pr(c_{i,j}\neq occupied)} = \frac{\Pr(c_{i,j}=occupied)}{1-\Pr(c_{i,j}=occupied)}, \tag{5.11}$$

$$odd(\mathbf{z}_k|c_{i,j} = occupied) = \frac{\Pr(\mathbf{z}_k|c_{i,j}=occupied)}{\Pr(\mathbf{z}_k|c_{i,j}\neq occupied)}. \tag{5.12}$$

Now, take the logarithm of both sides of (5.10) to yield the following:

$$\begin{aligned} &\log\left(odd(c_{i,j} = occupied|\mathbf{z}_k)\right) = \log\left(odd(\mathbf{z}_k|c_{i,j} = occupied)\right) + \\ &\log\left(odd(c_{i,j} = occupied)\right). \end{aligned} \tag{5.13}$$

This last equation is used to update the occupancy grid in terms of addition instead of multiplication, which is very efficient computationally.

5.3 Practical Considerations

Generally, the robot determines its pose using measurements from noisy sensors, either proprioceptive sensors, such as rotary encoders, inertial measurement units (IMU), or exteroceptive sensors such as cameras, sonars and laser scanners. Such noise will reflect on the robot pose estimation which, in return, will reflect on the landmark position estimation.

One approach to take into account the robot pose uncertainty is to introduce a *forgetting factor* that forces the robot to forget old landmarks over time. This approach does not optimise the map estimate using all available information, however, it will produce a relatively consistent map with respect to recent measurement. In terms of occupancy grid, forgetting factor is a function of time step $0 < f(k) < \infty$ that is added, or subtracted, to the $\log\left(odd\left(c_{i,j} = occupied\right)\right)$ of each cell in the occupancy grid, such that:

$$\log\left(odd\left(c_{i,j} = occupied\right)\right) = \log\left(odd\left(c_{i,j} = occupied\right)\right) + f(k).sgn\left(\log\left(odd\left(c_{i,j} = occupied\right)\right)\right) \quad (5.14)$$

Cells that are not observed frequently will have their probability of occupancy reduced to unknown, e.g., $\Pr\left(c_{i,j} = occupied\right) = 0.5$.

The dependency between the robot pose and the landmark estimation can transform the robotic mapping problem into a robotic localisation and mapping problem. Therefore, the majority of mapping approaches consider the *Simultaneous Localisation and Mapping* (SLAM) problem, where the robot needs to use all available data from sensors to jointly estimate its pose and the position of each landmark in the environment.

Chapter 6. Mobile Robot Localisation

One of the basic functions of mobile robots is to traverse from a certain position to another one in the environment. In order to accomplish this task the mobile robot needs to know its pose in the environment at any time to determine whether it has reached its final destination; such process is called localisation. There are many techniques to approach the localisation problem and they differ in the type of sensors used and how the uncertainty is addressed. In this chapter, three popular localisation techniques are described, and their advantages and disadvantages are illustrated.

6.1 Motion-based Localisation (Dead Reckoning)

This technique uses the internal kinematics of the robot to localise it in the environment. The robot may also employ some proprioceptive sensors such as: encoders or IMUs, to provide a better estimate of the pose change over time. This method is simple to implement, and does not require sophisticated sensors. However, such technique suffers from the unbounded growth of uncertainty about the robot pose over time due to the numerical integration and accumulation of error. This problem is illustrated in the following example:

Consider a differential-drive robot that operates in a 2D environment, shown in Figure 6.1, with kinematics model given by (6.1), where V and ω are the desired linear and angular velocities of the robot, respectively.

$$\frac{d}{dt}\begin{bmatrix} s_x \\ s_y \\ s_\theta \end{bmatrix} = \begin{bmatrix} \cos(s_\theta) & 0 \\ \sin(s_\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} V \\ \omega \end{bmatrix}. \tag{6.1}$$

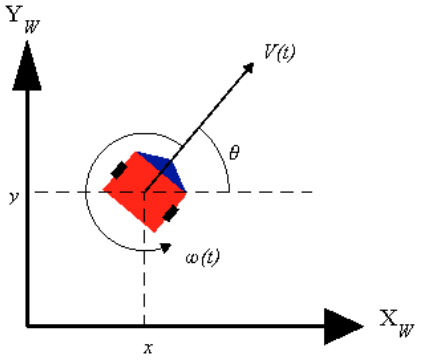


Figure 6.1: Differential-drive robot pose $\mathbf{s}_k = [s_x \quad s_y \quad s_\theta]^T$. The robot inputs are the linear and angular velocities $\mathbf{u}_k = [V \quad \omega]^T$.

If Δt is the sampling time, then it is possible to compute the incremental linear and angular displacements, Δd and $\Delta\theta$, as follows:

$$\Delta d = V.\Delta t \quad , \quad \Delta\theta = \omega.\Delta t. \tag{6.2}$$

To compute the pose of the robot at any given time step, (6.1) can be numerically integrated as shown in (6.3). This approximation follows the Markov assumption where the current robot pose depends only on the previous pose and the input velocities.

$$\begin{bmatrix} s_{x,k} \\ s_{y,k} \\ s_{\theta,k} \end{bmatrix} = \begin{bmatrix} s_{x,k-1} \\ s_{y,k-1} \\ s_{\theta,k-1} \end{bmatrix} + \begin{bmatrix} \Delta d \cos(s_{\theta,k-1}) \\ \Delta d \sin(s_{\theta,k-1}) \\ \Delta\theta \end{bmatrix}. \tag{6.3}$$

If the mobile robot is equipped with encoders attached to each wheel, where they measure the wheels displacement, Δd_r and Δd_l, during each time step, the incremental linear and angular displacements are computed as shown in (6.4), where l is the robot wheel base.

$$\begin{bmatrix} \Delta d \\ \Delta\theta \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ 1/l & -1/l \end{bmatrix} \begin{bmatrix} \Delta d_r \\ \Delta d_l \end{bmatrix}. \tag{6.4}$$

The pose estimation of a mobile robot is almost always associated with some uncertainty with respect to its state parameters. There are several factors that contribute to such uncertainty, where some are due to deterministic (systematic) errors and others are due to nondeterministic (random) errors. The deterministic errors can be identified and compensated during estimation, and they include: misalignment of wheels and unequal wheel diameter. On the other hand, nondeterministic errors, such as slipping and drifting, are random and they cause the growth of uncertainty over time. From a geometric point of view, the error in differential-drive robots is classified into three groups:

- Range error: it is associated with the computation of Δd over time.
- Turn error: it is associated with the computation of $\Delta\theta$ over time.
- Drift error: it is associated with the difference between the angular speed of the wheels and it affects the error in the angular rotation of the robot.

Due to such uncertainty, it is possible to represent the belief of the robot pose by a Gaussian distribution, where the mean vector $\boldsymbol{\mu}_k$ is the best estimate of the pose and the covariance matrix $\boldsymbol{\Sigma}_k$ is the uncertainty of the pose that encapsulates the above errors. This distribution is denoted by $\mathbf{s}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$.

In the context of probability, the robot pose at time step k, denoted by $\mathbf{s}_k$, can be described with Markov assumption as function of all previous robot pose $\mathbf{s}_{k-1}$ and the current control input $\mathbf{u}_k = [V_k \quad \omega_k]^T$. This process is called the *robot motion model* and it is defined as follows:

$$\mathbf{s}_k = \mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k) + \mathbf{q}_k, \tag{6.5}$$

where $\mathbf{q}_k$ is an additive Gaussian noise such that $\mathbf{q}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k)$, and $\mathbf{Q}_k$ is a positive semidefinite covariance matrix. This noise is directly related to the error sources described above. The function $\mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k)$ is generally nonlinear, and in the case of a differential-drive robot described in Figure 6.1, this function is defined as:

$$\mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k) = \begin{bmatrix} s_{x,k-1} + \Delta t. V_k. \cos(s_{\theta,k-1}) \\ s_{y,k-1} + \Delta t. V_k. \sin(s_{\theta,k-1}) \\ s_{\theta,k-1} + \Delta t. \omega_k \end{bmatrix}, \tag{6.6}$$

Assume that the robot pose at time step $k-1$ is given by a Gaussian distribution such that $\mathbf{s}_{k-1} \sim \mathcal{N}(\boldsymbol{\mu}_{k-1}, \boldsymbol{\Sigma}_{k-1})$. Then, the above setup can be used to estimate the robot pose at time step k by linearizing the robot motion model using first-order Taylor expansion around $\boldsymbol{\mu}_k$ as follows:

$$\boldsymbol{\mu}_k = \mathbf{h}(\boldsymbol{\mu}_{k-1}, \mathbf{u}_k), \tag{6.7}$$

$$\mathbf{H}_k = \nabla_{\mathbf{s}_k} \mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k)\big|_{\mathbf{s}_{k-1}=\boldsymbol{\mu}_{k-1}}, \tag{6.8}$$

$$\mathbf{s}_k \approx \boldsymbol{\mu}_k + \mathbf{H}_k(\mathbf{s}_{k-1} - \boldsymbol{\mu}_{k-1}). \tag{6.9}$$

Equation (6.8) represents the Jacobian matrix of $\mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k)$ with respect to each variable in $\mathbf{s}_{k-1}$, evaluated at $\mathbf{s}_{k-1} = \boldsymbol{\mu}_{k-1}$. In the case of a differential-drive robot, the Jacobian $\mathbf{H}_k$ is computed as follows:

$$\mathbf{H}_k = \begin{bmatrix} \frac{\partial h_1}{\partial s_{x,k}} & \frac{\partial h_1}{\partial s_{y,k}} & \frac{\partial h_1}{\partial s_{\theta,k}} \\ \frac{\partial h_2}{\partial s_{x,k}} & \frac{\partial h_2}{\partial s_{y,k}} & \frac{\partial h_2}{\partial s_{\theta,k}} \\ \frac{\partial h_3}{\partial s_{x,k}} & \frac{\partial h_3}{\partial s_{y,k}} & \frac{\partial h_3}{\partial s_{\theta,k}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\Delta t. V_k. \sin(\mu_{\theta,k-1}) \\ 0 & 1 & \Delta t. V_k. \cos(\mu_{\theta,k-1}) \\ 0 & 0 & 1 \end{bmatrix}. \tag{6.10}$$

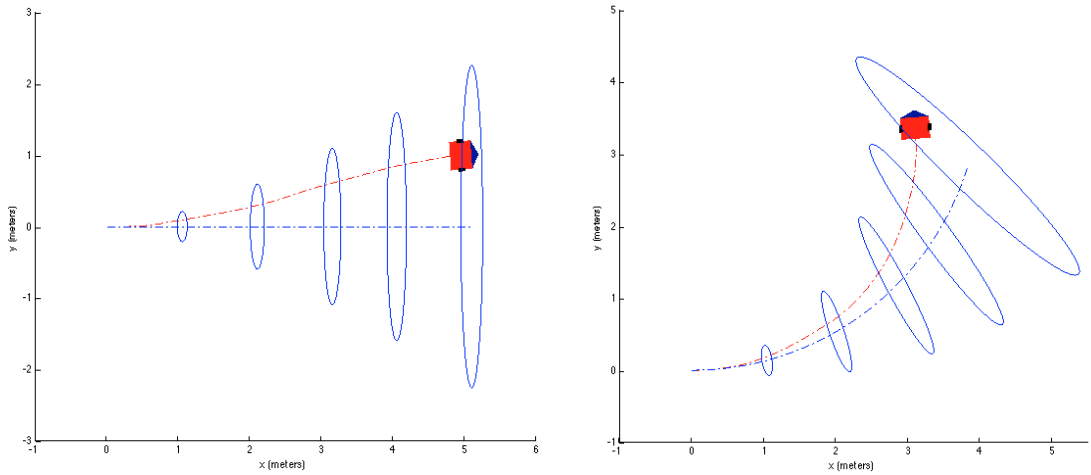
Since the robot motion model is linearised and all uncertainties are Gaussians, it is possible to compute the covariance $\boldsymbol{\Sigma}_k$ associated with the robot pose at time step k using the properties of Gaussians as follows:

$$\boldsymbol{\Sigma}_k = \mathbf{H}_k \boldsymbol{\Sigma}_{k-1} \mathbf{H}_k^{\mathrm{T}} + \mathbf{Q}_k \, . \tag{6.11}$$

Thus, the estimated pose at time step k is Gaussian such that $\mathbf{s}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, and it is computed recursively using the pose at time step $k-1$ and the input vector $\mathbf{u}_k$. The initial robot pose is assumed known such that $\boldsymbol{\mu}_k = \mathbf{0}$, and $\boldsymbol{\Sigma}_k = \mathbf{0}$.

According to (6.11), the pose uncertainty will always increase every time the robot moves due to the addition of the nondeterministic error represented by $\mathbf{Q}_k$, which is positive semi-definite. This result is illustrated in Figure 6.2 where the joint uncertainty of s_x and s_y is represented by the ellipsoid around the robot. In Figure 6.2(a), as the robot moves along the x-axis, its uncertainty along the y-

axis increases faster than the x-axis due to the drift error. Figure 6.2(b) shows that the uncertainty ellipsoid is no longer perpendicular to the motion direct as soon as the robot starts to turn.



(a) (b)

Figure 6.2: Unbounded growth of uncertainty for differential-drive robot where the blue-dashed line is the estimated pose, red-dashed line is the true pose subject to nondeterministic errors, and the blue ellipsoid represents 99% confidence of the pose in s_x and s_y. (a) Robot with straight move command only; (b) robot with straight and turn move commands.

With this type of localisation, where the robot depends solely on its proprioceptive sensors, the pose uncertainty will grow unbounded over time as the robot moves. This growth will soon make the estimated pose invalid for the robot to make proper navigational decisions, such as performing optimal path planning. One way to overcome this issue is to use a map; this approach is called map-based localisation and it is discussed in Section 6.2.

6.2 Map-based Localisation

In this approach, the robot utilises a map of the environment along with some exteroceptive sensor, such as LIDAR or camera, to localise itself within the map. This technique is superior to the dead-reckoning localisation because the uncertainty of the pose does not grow unbounded, but it is limited by the noise of the exteroceptive sensor. Figure 6.3 shows the major differences between dead-reckoning localisation and map-based localisation.

Dead-reckoning localisation:	**Map-based localisation:**
1. Kinematic/dynamic model 2. Proprioceptive sensors: - Encoders - Inertial measurement unit (IMU)	1. Kinematic/dynamic model 2. Proprioceptive sensors: - Encoders - Inertial measurement unit (IMU) 3. Exteroceptive sensors: - LIDAR - Camera 4. Map of the environment
(a)	(b)

Figure 6.3: Requirement for: (a) dead-reckoning localisation, (b) map-based localisation.

Map-based localisation requires a map that contains useful information about the environment, and it includes landmarks. These landmarks can be simple features such as: points, lines and planes, or they can be more complex features such as: textures, arbitrary shapes, etc. The type of the existing exteroceptive sensor determines the complexity of the map features. Of course, complex the feature in the map, the more processing needed prior localisation. In this course, only point features are considered, and each landmark is denoted by $\mathbf{m}_i$. The set of all landmarks in the environment is the map and it is denoted by M, such that:

$$M = \{\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_{n_l}\}. \tag{6.12}$$

Where n_l is the total number of landmarks in the environment. There are many map-based localisation techniques, however, the most popular are the probabilistic techniques, where the robot pose is represented by probability distribution over the state space, for example $\mathbf{s}_k \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. Almost all

probabilistic localisation techniques use Bayes filters, and they follow Markov assumption, where the current state of the robot, e.g. the pose, depends on the current inputs and only on the immediate previous pose. Also, the Markov assumption requires that the observation at time step k depends only on the state at that time step.

In addition to the robot motion model described in motion-based localisation, map-based localisation uses it exteroceptive sensor to detect landmarks in the environment and match them to landmarks in the map. This process corresponds to the *robot observation model*, and it is defined as follows:

$$\mathbf{z}_{i,k} = \mathbf{g}(\mathbf{m}_i, \mathbf{s}_k) + \mathbf{r}_{i,k}, \tag{6.13}$$

where $\mathbf{z}_{i,k}$ is the measurement obtained by the robot exteroceptive sensor that correspond to the i-th landmark in the map, and $\mathbf{r}_{i,k}$ is an additive Gaussian noise such that $\mathbf{r}_{i,k} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k)$ where $\mathbf{R}_k$ its positive semidefinite covariance matrix. The function $\mathbf{g}(\mathbf{m}_i, \mathbf{s}_k)$ is generally nonlinear.

In this course, the procedure for map-based localisation applied at each time step is as follows:

(i) Use the robot motion model to estimate the robot pose (1st source of information)
(ii) Match each measurement with a single landmark in the map. This step is known as data association and it is assumed solved in this course.
(iii) Use the matched measurement/landmark pair to estimate the robot pose (2nd source of information).
(iv) Combine the two sources of robot pose using weight average method, e.g., Kalman filter (KF) or Extended Kalman filter (EKF).

6.2.1 Combining two sources of information (Sonar and LIDAR)

Most of the times a mobile robot can be equipped with multiple sensors in order to improve the reliability and accuracy of the measurements. For instance, the mobile robot can use two sensors to measure distance: ultrasonic range sensor (sonar) and laser range finding sensor (LIDAR). The LIDAR sensor has far better accuracy and range compared with the sonar, but it can give flawed measurements in some situations when the sonar doesn't have any issues (ex. when measuring reflective or transparent surfaces). To avoid these problems we need to combine the measurements provided by the two sensors into one single measurement. This type of approach is called *sensor fusion* and when it is done in an adequate way it provides better accuracy for the combined measurement.

The sensor fusion can be computed in a very efficient way as long as the sensor noise is assumed to be unimodal, zero-mean with Gaussian distribution. Based on each measurement we can estimate the

robot's position such that the sonar estimation is done at step k and is denoted q_1 and the LIDAR-based estimation is done at step $k+1$ and is denoted q_2. Both estimations have a Gaussian probability distribution with associated variance σ_1^2 and σ_2^2 respectively.

The two robot position estimates are:

$$\hat{q}_1 = q_1 \; with \; variance \; \sigma_1^2 \tag{6.14}$$

$$\hat{q}_2 = q_2 \; with \; variance \; \sigma_2^2 \tag{6.15}$$

We assume there is no robot movement between step k and step $k+1$. In order to *fuse*(combine) the two probability distributions and get the best estimation for the measurement (denoted $\hat{q}$) we can use as weighted least-squares techniques. The criteria to be minimised can be written:

$$S = \sum_{i=1}^{n} w_i(\hat{q} - q_i)^2 \tag{6.16}$$

where w_i is the weight of measurement i. To find the estimation with the minimum error we set the derivative of S to zero:

$$\frac{\partial S}{\partial \hat{q}} = \frac{\partial S}{\partial \hat{q}} \sum_{i=1}^{n} w_i(\hat{q} - q_i)^2 = 2\sum_{i=1}^{n} w_i(\hat{q} - q_i) = 0 \tag{6.17}$$

$$\sum_{i=1}^{n} w_i\hat{q} - \sum_{i=1}^{n} w_i q_i = 0 \tag{6.18}$$

$$\hat{q} = \frac{\sum_{i=1}^{n} w_i q_i}{\sum_{i=1}^{n} w_i} \tag{6.19}$$

For the measurement weight w_i we can use:

$$w_i = \frac{1}{\sigma_i^2} \tag{6.20}$$

The value of the estimation $\hat{q}$ using two measurements can be written as:

$$\hat{q} = \frac{\frac{1}{\sigma_1^2} q_1 + \frac{1}{\sigma_2^2} q_2}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}} = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} q_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} q_2 \tag{6.21}$$

$$\frac{1}{\hat{\sigma}^2} = \frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2} = \frac{\sigma_1^2 + \sigma_2^2}{\sigma_1^2 \sigma_2^2} \tag{6.22}$$

$$\hat{\sigma}^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2} \tag{6.23}$$

From (6.23) it can be seen that the combined variance $\hat{\sigma}^2$ is smaller than each individual measurement variance σ_i taken independently. This means that the uncertainty of the measurement estimation has been decreased by combining the two measurements. We can notice that even poor measurements such as those coming from the sonar can only increase the accuracy of the measurement estimation.

Equation (6.21) can be rewritten as:

$$\hat{q} = q_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}(q_2 - q_1) \tag{6.24}$$

Also (6.23) can be written as:

$$\hat{\sigma}^2 = \left(1 - \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}\right)\sigma_1^2 \tag{6.25}$$

Equations (6.24) and (6.25) are related to the Kalman filter where the Kalman gain is defined as:

$$K = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} \tag{6.26}$$

6.2.2 Kalman Filter Localisation

Consider a mobile robot that is equipped with an exteroceptive sensor that measures distinguished landmarks in the environment and matched them against different landmarks in the robot internal map that represents the environment. The robot also knows its motion model. With this setup, the robot has two sources of information about its pose:

(i) The robot motion model that guesses where the robot will reside given its previous pose.

(ii) The observation model that uses exteroceptive sensor to measure the robot pose with respect to the map.

Combining these two poses can be done in a similar manner to that explained in section 6.2.1. Kalman filter and Extended Kalman filter are efficient recursive methods that solve the map-based localisation problem as described next.

Recall the mobile robot with a motion model:

$$\mathbf{s}_k = \mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k) + \mathbf{q}_k, \tag{6.27}$$

where $\mathbf{q}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k)$. Also, recall the robot observation model:

$$\mathbf{z}_{i,k} = \mathbf{g}(\mathbf{m}_i, \mathbf{s}_k) + \mathbf{r}_{i,k}, \tag{6.28}$$

where $\mathbf{r}_{i,k} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k)$. If $\mathbf{h}$ and $\mathbf{g}$ are linear functions with respect to their variables, the Kalman filter can be used directly and optimality is guaranteed. Otherwise, Extended Kalman filter is used where both functions $\mathbf{h}$ and $\mathbf{g}$ are linearised around current robot pose estimate using first-order Taylor expansion as follows:

$$\hat{\boldsymbol{\mu}}_k = \mathbf{h}(\boldsymbol{\mu}_{k-1}, \mathbf{u}_k), \tag{6.29}$$

$$\mathbf{H}_k = \nabla_{\mathbf{s}_k} \mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k)\big|_{\mathbf{s}_{k-1}=\boldsymbol{\mu}_{k-1}}, \tag{6.30}$$

$$\mathbf{s}_k \approx \hat{\boldsymbol{\mu}}_k + \mathbf{H}_k(\mathbf{s}_{k-1} - \boldsymbol{\mu}_{k-1}). \tag{6.31}$$

$$\hat{\mathbf{z}}_{i,k} = \mathbf{g}(\mathbf{m}_i, \hat{\boldsymbol{\mu}}_k), \tag{6.32}$$

$$\mathbf{G}_k = \nabla_{\mathbf{s}_k} \mathbf{g}(\mathbf{m}_i, \mathbf{s}_k)\big|_{\mathbf{s}_k=\hat{\boldsymbol{\mu}}_k}, \tag{6.33}$$

$$\mathbf{z}_{i,k} \approx \hat{\mathbf{z}}_{i,k} + \mathbf{G}_k(\mathbf{s}_k - \hat{\boldsymbol{\mu}}_k). \tag{6.34}$$

Based on Extended Kalman filter theory, the aim of the linearised model is to propagate covariance matrices based on Gaussian distributions. Thus, provided that $\mathbf{s}_{k-1} \sim \mathcal{N}(\boldsymbol{\mu}_{k-1}, \boldsymbol{\Sigma}_{k-1})$ is available, the *prediction step* of the extended Kalman filter is done in a similar manner to motion-based localisation as follows:

$$\hat{\boldsymbol{\mu}}_k = \mathbf{h}(\boldsymbol{\mu}_{k-1}, \mathbf{u}_k)\,, \tag{6.35}$$

$$\hat{\boldsymbol{\Sigma}}_k = \mathbf{H}_k \boldsymbol{\Sigma}_{k-1} \mathbf{H}_k^{\mathrm{T}} + \mathbf{Q}_k\,. \tag{6.36}$$

Then, the *correction step* of extended Kalman filter is carried out as follows:

$$\hat{\mathbf{z}}_{i,k} = \mathbf{g}(\mathbf{m}_i, \hat{\boldsymbol{\mu}}_k)\,, \tag{6.37}$$

$$\mathbf{Z}_k = \mathbf{G}_k \hat{\boldsymbol{\Sigma}}_k \mathbf{G}_k^T + \mathbf{R}_k\,, \tag{6.38}$$

$$\mathbf{K}_k = \hat{\boldsymbol{\Sigma}}_k \mathbf{G}_k^T \mathbf{Z}_k^{-1}, \tag{6.39}$$

$$\boldsymbol{\mu}_k = \hat{\boldsymbol{\mu}}_k + \mathbf{K}_k(\mathbf{z}_{i,k} - \hat{\mathbf{z}}_{i,k}), \tag{6.40}$$

$$\boldsymbol{\Sigma}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{G}_k)\hat{\boldsymbol{\Sigma}}_k. \tag{6.41}$$

Putting all of the above together, the EKF localisation procedure for known data association is summarised in Algorithm 1. The robot initial pose is required for this procedure, otherwise, it will be assumed that $\boldsymbol{\mu}_0 = \mathbf{0}$, and $\boldsymbol{\Sigma}_0 = \mathbf{0}$, i.e., the robot starts at the origin of the world frame $\{W\}$.

Algorithm 1 EKF Localisation with known data association

1: **function** EKF-Localisation (M, $\boldsymbol{\mu_{k-1}}$, $\boldsymbol{\Sigma_{k-1}}$, $\mathbf{u}_k, \mathbf{z}_{i,k}, \mathbf{Q}_k, \boldsymbol{R_k}$)

2: $\quad \widehat{\boldsymbol{\mu}}_k \leftarrow \mathbf{h}(\boldsymbol{\mu_{k-1}}, \mathbf{u}_k)$

3: $\quad \mathbf{H}_k \leftarrow \nabla_{\mathbf{s}_{k-1}} \mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k)|_{\mathbf{s}_{k-1}=\boldsymbol{\mu}_{k-1}}$

4: $\quad \widehat{\boldsymbol{\Sigma}}_k \leftarrow \mathbf{H}_k \boldsymbol{\Sigma}_{k-1} \mathbf{H}_k^T + \mathbf{Q}_k$

5: $\quad$ **if** $\mathbf{z}_{i,k}$ **corresponds to landmark** $\mathbf{m}_i \in \boldsymbol{M}$

6: $\quad\quad \hat{\mathbf{z}}_{i,k} \leftarrow \mathbf{g}(\mathbf{m}_i, \widehat{\boldsymbol{\mu}}_k)$

7: $\quad\quad \mathbf{G}_k \leftarrow \nabla_{\mathbf{s}_k} \mathbf{g}(\mathbf{m}_i, \mathbf{s}_k)|_{\mathbf{s}_k=\widehat{\boldsymbol{\mu}}_k}$

8: $\quad\quad \mathbf{Z}_k \leftarrow \mathbf{G}_k \widehat{\boldsymbol{\Sigma}}_k \mathbf{G}_k^T + \mathbf{R}_k$

9: $\quad\quad \mathbf{K}_k \leftarrow \widehat{\boldsymbol{\Sigma}}_k \mathbf{G}_k^T \mathbf{Z}_k^{-1}$

10: $\quad\quad \boldsymbol{\mu}_k \leftarrow \widehat{\boldsymbol{\mu}}_k + \mathbf{K}_k(\mathbf{z}_{i,k} - \hat{\mathbf{z}}_{i,k})$

11: $\quad\quad \boldsymbol{\Sigma}_k \leftarrow (\mathbf{I} - \mathbf{K}_k \mathbf{G}_k) \widehat{\boldsymbol{\Sigma}}_k$

12: $\quad$ **return** $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$

Kalman filter is a popular method to solve the localisation problem. It is continuous in the state space and its computational complexity is at least quadratic with respect to the state dimension, i.e. $\mathcal{O}(n^2)$, due to the matrix multiplication in (6.39). However, this method requires both the motion and the sensor models to be linear. Also, the noise associated with each model needs to be Gaussian. Kalman filter method looks at the localisation problem as a tracking problem where the initial pose is known with some uncertainty; therefore, Kalman filter does not support multimodal distribution. Due to this fact, Kalman filter localisation cannot solve the *kidnaped robot problem* when the robot collides or moves unexpectedly. Moreover, this method fails when the robot makes a mistake during the data association process, e.g. matching one detected wall with a completely different wall in the map.

6.2.3 Case Study

Consider a robot moving in 2D fully observable environment with 6 distinct landmarks as shown in Figure 6.4. The robot pose represents position and orientation of the robot such that $\mathbf{s}_k = [s_{x,k} \quad s_{y,k} \quad s_{\theta,k}]^T$, and the landmark is a stationary point feature denoted by $\mathbf{m}_i = [m_{x,i} \quad m_{y,i}]^T$.

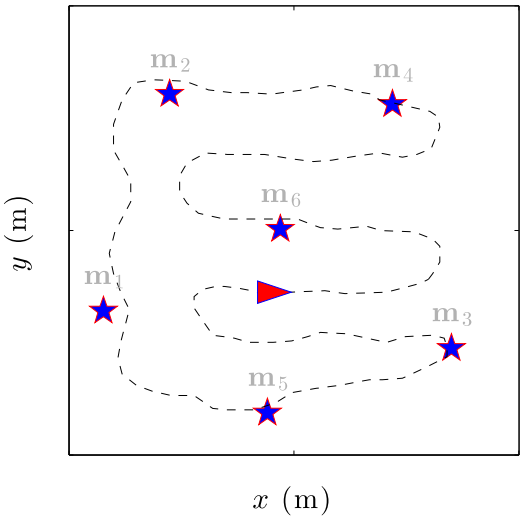


Figure 6.4: Example of mobile robot moving in 2D environment with 6 landmarks represented by the stars, whereas the dashed line represents the robot path. The red triangle represents the initial robot pose, and it points towards the direction of motion.

The robot has two control inputs $\mathbf{u}_k = [V_k \quad \omega_k]^T$, where V_k is the robot linear velocity and ω_k is the robot angular velocity. The robot motion model is defined as follows:

$$s_{x,k} = s_{x,k-1} + \Delta t.V_k.\cos s_{\theta,k-1} + q_{x,k}, \tag{6.42}$$

$$s_{y,k} = s_{y,k-1} + \Delta t.V_k.\sin s_{\theta,k-1} + q_{y,k}, \tag{6.43}$$

$$s_{\theta,k} = s_{\theta,k-1} + \Delta t.\omega_k + q_{x,k}, \tag{6.44}$$

Where Δt is the temporal length of consecutive time steps, and the vector $\boldsymbol{q}_k = [q_{x,k} \quad q_{y,k} \quad q_{\theta,k}]^T$ represent the motion noise. The robot is equipped with a 360° range-bearing exteroceptive sensor. Reference to principle of operation of the range-bearing sensor illustrated in Figure 6.5, the robot observation model is defined as follows:

$$z_{\rho,i,k} = \sqrt{(m_{x,i} - s_{x,k})^2 + (m_{y,i} - s_{y,k})^2} + r_{\rho,i,k}, \tag{6.45}$$

$$z_{\alpha,i,k} = \text{atan2}(m_{y,i} - s_{y,k}, m_{x,i} - s_{x,k}) - s_{\theta,k} + r_{\alpha,i,k} \tag{6.47}$$

Where $\mathbf{z}_{i,k} = [\rho_{i,k} \quad \alpha_{i,k}]^{\boldsymbol{T}}$is the measurement vector that consists of landmark angle and bearing in robot frame $\{R\}$, and $\mathbf{r}_{i,k} = [r_{\rho,i,k} \quad r_{\alpha,i,k}]^{\boldsymbol{T}}$ is the measurement noise vector. The noise of both motion and observation models is Gaussian.

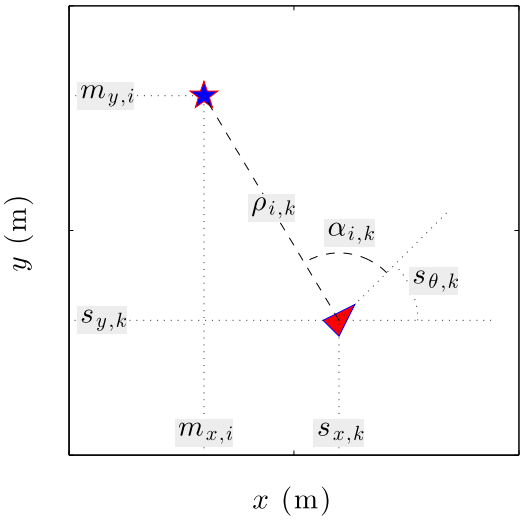


Figure 6.5: Observation model for a mobile robot in 2-D environment at time step k. The triangle is the robot pose $\mathbf{s}_k$, and the star is the landmark $\mathbf{m}_i$, and both are with respect to the world reference frame $\{W\}$, whereas the measurements $\rho_{i,k}, \alpha_{i,k}$ are in the robot frame $\{R\}$.

Since the robot motion and observation models are nonlinear, EKF is used where the Jacobian matrix $\mathbf{H}_k$ is defined in (6.10). The observation model Jacobian matrix $\mathbf{G}_k$ is computed as follows:

$$\Delta x = m_{x,i} - \hat{s}_{x,k}, \tag{6.48}$$

$$\Delta y = m_{y,i} - \hat{s}_{y,k}, \tag{6.49}$$

$$p = \Delta x^2 + \Delta y^2, \tag{6.50}$$

$$\mathbf{G_k} = \begin{bmatrix} \frac{\partial g_1}{\partial s_{x,k}} & \frac{\partial g_1}{\partial s_{y,k}} & \frac{\partial g_1}{\partial s_{\theta,k}} \\ \frac{\partial g_2}{\partial s_{x,k}} & \frac{\partial g_2}{\partial s_{y,k}} & \frac{\partial g_2}{\partial s_{\theta,k}} \end{bmatrix} = \begin{bmatrix} -\frac{\Delta x}{\sqrt{p}} & -\frac{\Delta y}{\sqrt{p}} & 0 \\ \frac{\Delta y}{p} & -\frac{\Delta x}{p} & -1 \end{bmatrix}. \tag{6.51}$$

For this example, the prediction step and correction step of EKF follow the same equations described in Section 6.2.2. Figure 6.6 shows the result of EKF localisation applied to the example illustrated in Figure 6.4.

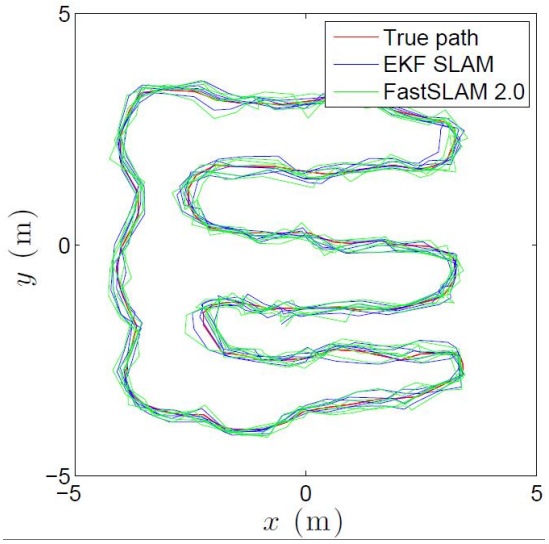


Figure 6.6: Estimated robot pose after applying EKF localisation.

6.2.4 Practical Considerations

It is important to handle angles carefully and consider the wrapping effect. For example, consider the angle x in radian; from the point of view of the robot orientation, this angle is the same as $x + 2n\pi$. Equations 6.24, 6.27, and 6.33 show that the resultant angle can exceed the $[-\pi : \pi]$ limit. This can lead to wrong estimation of the mean $\widehat{\boldsymbol{\mu}}_k$ in the correction step in EKF. To avoid this problem, it is important to wrap these angles and limit them to values between $-\pi$ and π. In Matlab, this is done using "*wrapToPi*" function.

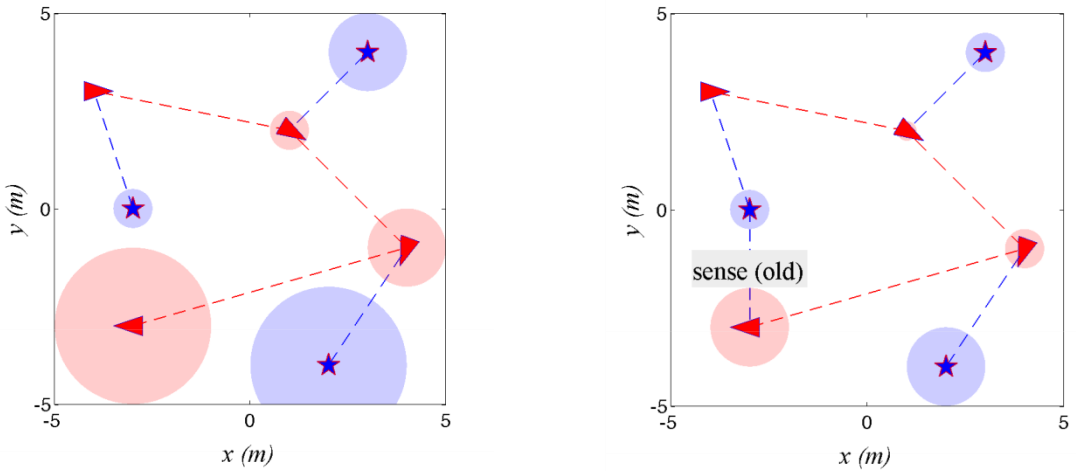
Chapter 7. Introduction to Simultaneous Localisation and Mapping (SLAM)

Consider an environment consisting of a finite number of distinct landmarks, where a mobile robot is navigating without prior knowledge of the landmark positions. Localisation is the problem of determining the robot position in the environment, whereas mapping is the problem of determining the position of the landmarks. These positions are usually considered with respect to some fixed reference frame denoted by $\{W\}$.

Generally, mobile robots are equipped with two types of on-board sensors: (i) proprioceptive sensors, such as encoders, inertial measurement units (IMU), etc., to convey the internal state of the robot, e.g., robot velocity, acceleration, etc.; and (ii) exteroceptive sensors, such as cameras, laser scanners, sonars etc., to report the state of the environment, for example, the position of the wall with respect to the robot reference frame denoted by $\{R\}$.

Under the assumption of perfect sensors, if the robot position is known exactly, the landmark positions in $\{W\}$ can be easily computed via projecting the measurements obtained using the robot on-board sensors. Likewise, if all landmark positions are known, then, localisation is straightforward. Unfortunately, all types of sensors are noisy, and they return measurements with some form of uncertainty.

SLAM is the problem of estimating the robot position and landmark locations given noisy on-board sensors. This problem is generally complex as illustrated in Figure 7.1, where the robot starts at some known position in $\{W\}$. At any time step k, when the robot moves, its position uncertainty will experience an unbounded growth due to the noise in the proprioceptive sensor. Any landmark detected at that time step will inherit the noise of the robot position, as well as the noise of the exteroceptive sensor. To limit this uncertainty propagation growth, the robot must observe an old known landmark, which will instantly provide high confidence about previous estimates.
The SLAM problem is complex and it presents many challenges. For example, there is the data association challenge where the robot needs to distinguish different landmarks and decides whether a landmark was previously observed or the dynamic environment challenge where the landmark positions are changing over time.



(a) (b)

Figure 7.1 Illustration of the SLAM problem where the robot, represented by red triangle, moves according to motion model represented by dashed red line and red region is the robot pose uncertainty. On the other hand, the landmark, represented by blue star, is observed (blue dashed line) by the robot using measurement model. Blue region represents the uncertainty of the landmark position. (a) Over time, uncertainties increase as robot observes new landmarks, and (b) as soon as an old landmark is observed, all uncertainties instantly decrease due to the correlation between the landmarks and the robot path.

Chapter 8. Recommended readings

For more details and illustrative demos please consult the Autonomous Systems Research Theme at The University of Manchester:

http://www.eee.manchester.ac.uk/our-research/research-themes/autonomoussystems/?_ga=1.186792424.1137636322.1480078878

[1]R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, 1st ed. Cambridge, Massachussets: The MIT Press, 2004.

[2] M. M. A. Mustafa, "GUARANTEED SLAM AN INTERVAL APPROACH," The University of Manchester, 2017.

[3] M. M. A. Mustafa, "Path Planning for Autonomous Mobile Robots," The University of Manchester, 2012.

[4] S. Pacheco-Gutierrez, "3D RECONSTRUCTION AND GUARANTEED PRIMITIVE SHAPE ESTIMATION USING INTERVAL ANALYSIS," The University of Manchester, 2016.

[5] Bruno Siciliano and Oussama Khatib. 2007. *Springer Handbook of Robotics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Appendix

MATLAB Mobile Robot Simulator Manual

1.0 Introduction

The MATLAB Mobile Robot Simulator is an open source program developed by the Autonomous Systems Research Theme in the University of Manchester. It aims to provide a simple and easy platform to implement and test various robotics concepts such as: motion control, path planning, localization, mapping, noise propagation, feature extraction, etc.

This platform allows building different types of 2D environments, then use it to control a single or multiple robots with various types of. This simulator also considers the noise in the sensors and the robot interaction with the environment, e.g., wheel slippage and robot drift error.

2.0 Getting Started

- Download `MobileRobotsSimulator_2017v1` folder from the University of Manchester Blackboard.
- Open `MobileRobotsSimulator_2017v1` folder. You should see a list of files and folders similar to those in Figure 2.1.

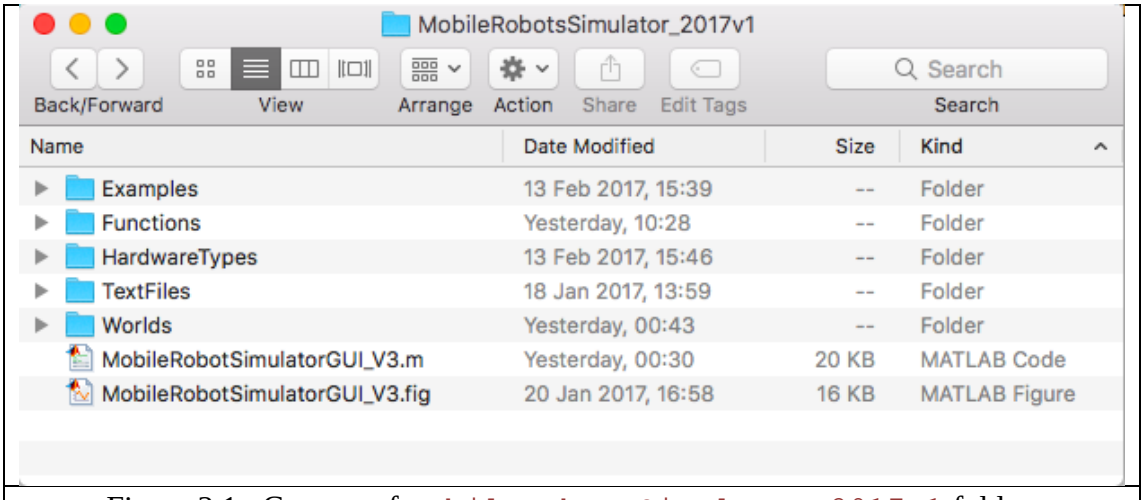


Figure 2.1: Content of `MobileRobotsSimulator_2017v1` folder.

- Open `MobileRobotSimulatorGUI_V3.m` file as shown in Figure 2.2.

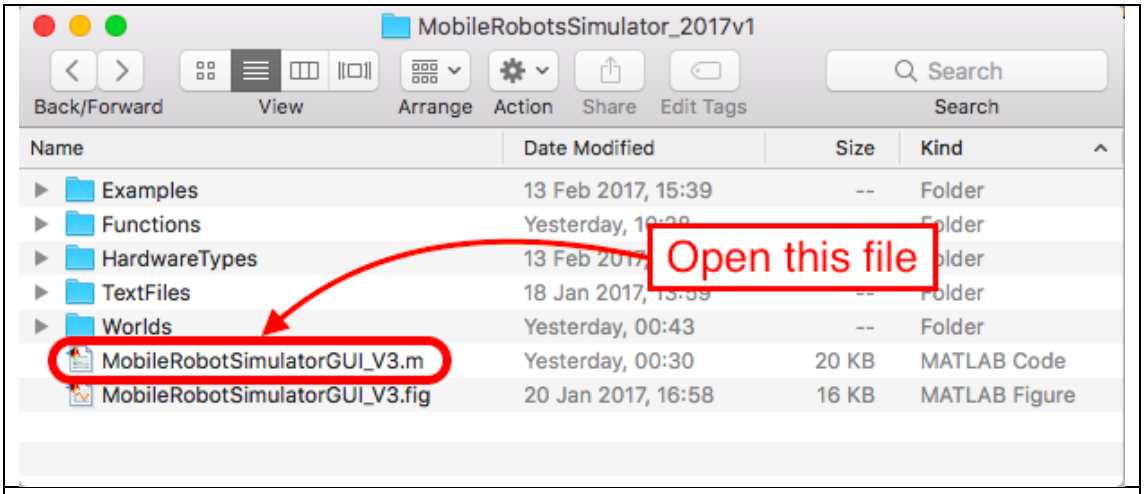


Figure 2.2: `MobileRobotsSimulatorGUI_V3.m` file inside `MobileRobotsSimulator_2017v1` Folder.

- Make sure `MobileRobotSimulatorGUI_V3.m` opens in MATLAB as shown in Figure 2.3, not any other text editor.
  Remark: it is recommended not to edit or modify the content of this file unless the user has sufficient knowledge about the structure of the program.

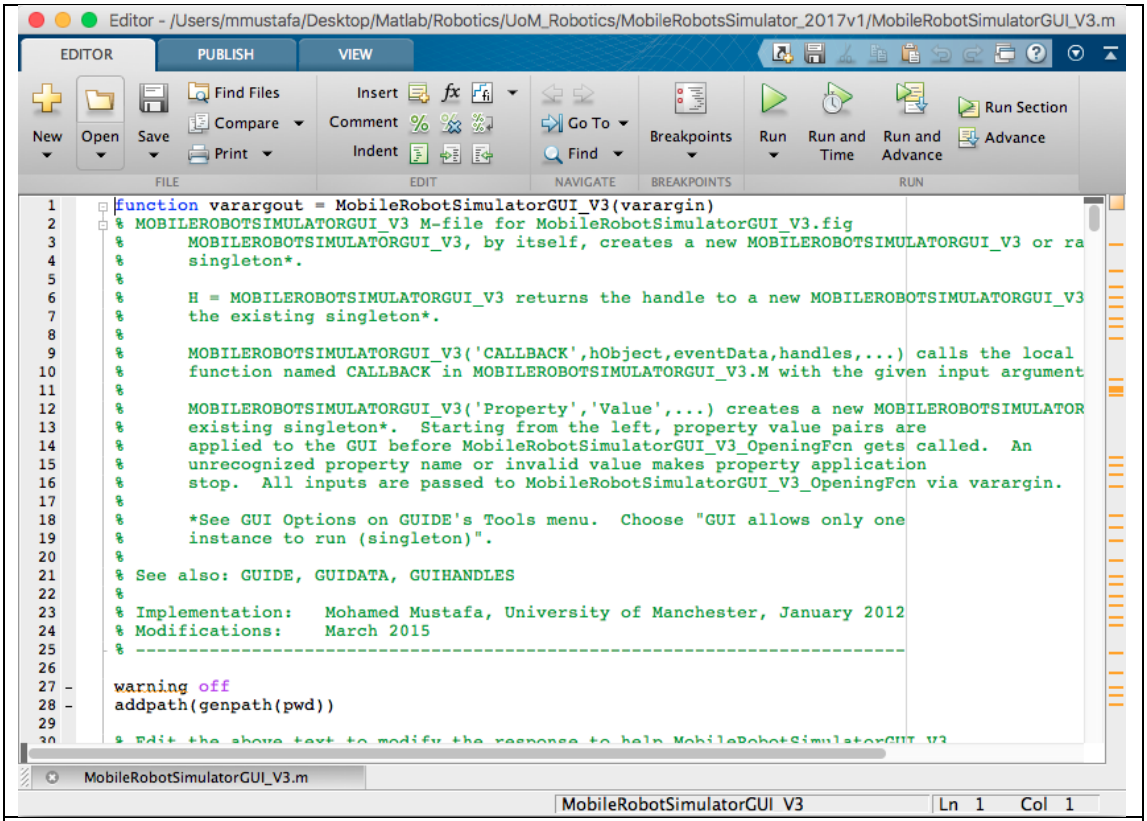


Figure 2.3: `MobileRobotsSimulatorGUI_V3.m` file opened in MATLAB.

- Run `MobileRobotsSimulatorGUI_V3.m` file by clicking on the Run button as shown in Figure 3.4.

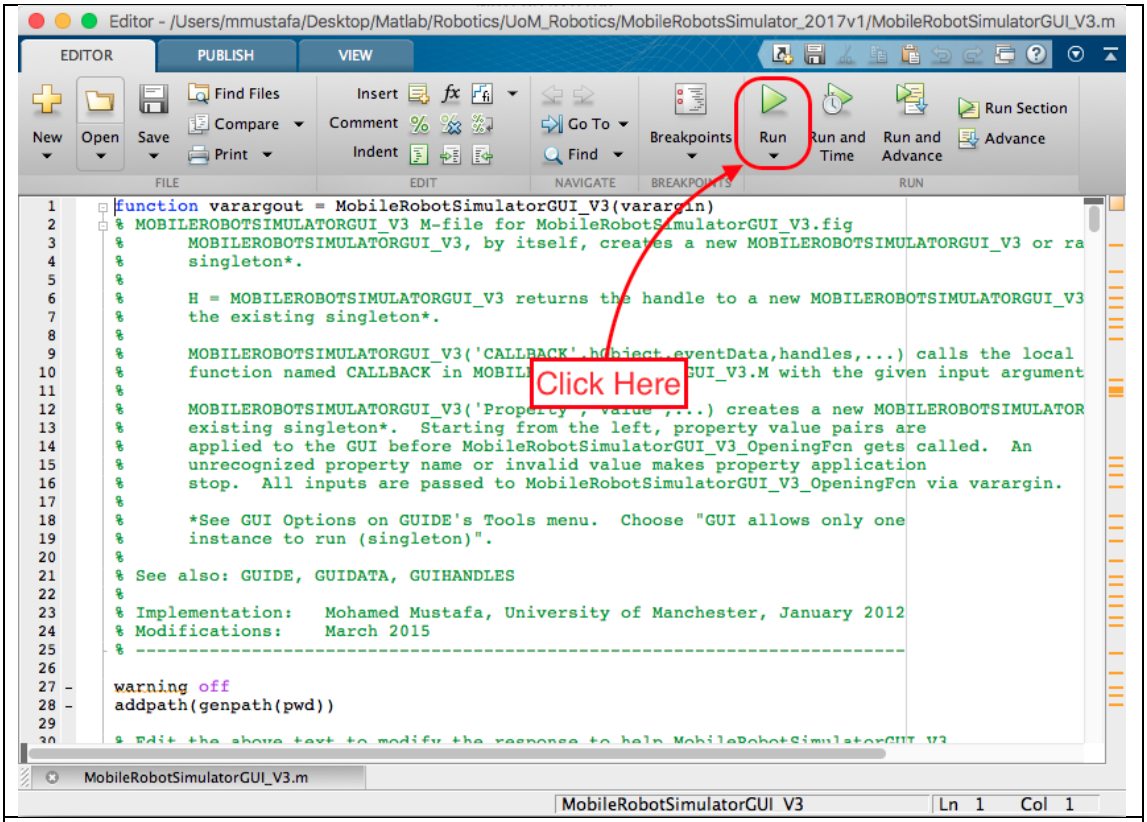


Figure 2.4: Run `MobileRobotsSimulatorGUI_V3.m` file.

- In the Dialog Box, select `Change Directory` option as shown in Figure 2.5 to add all files inside `MobileRobotsSimulator_2017v1` folder and subfolders to MATLAB path.

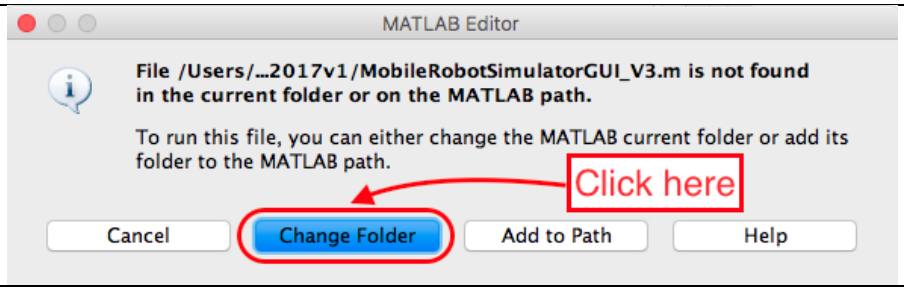


Figure 2.5: Change MATLAB path to include all files inside `MobileRobotsSimulator_2017v1` folder.

- A *graphical user interface* (GUI) will appear in a new window as shown in Figure 2.6.

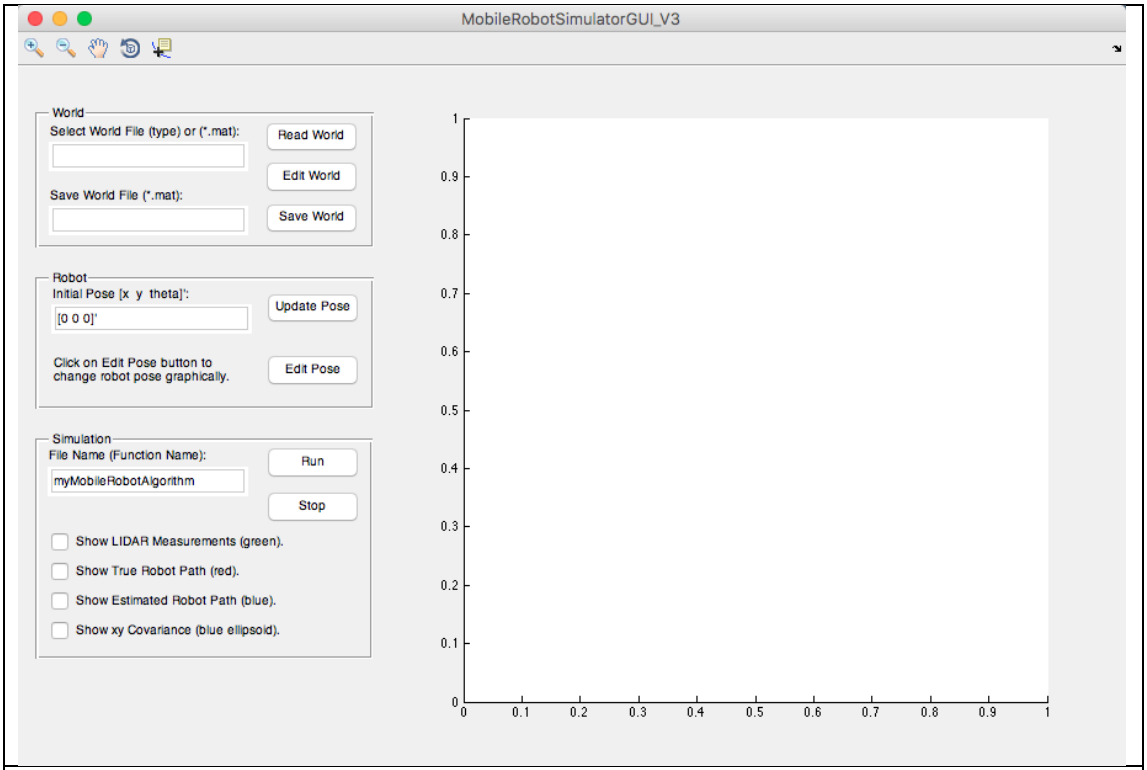


Figure 2.6: Mobile Robot Simulator GUI.

- Type `polygons` in the field highlighted in Figure 2.7. Then, click on `Read World` button. Note: There are other World types and they are explained in details in the next sections.

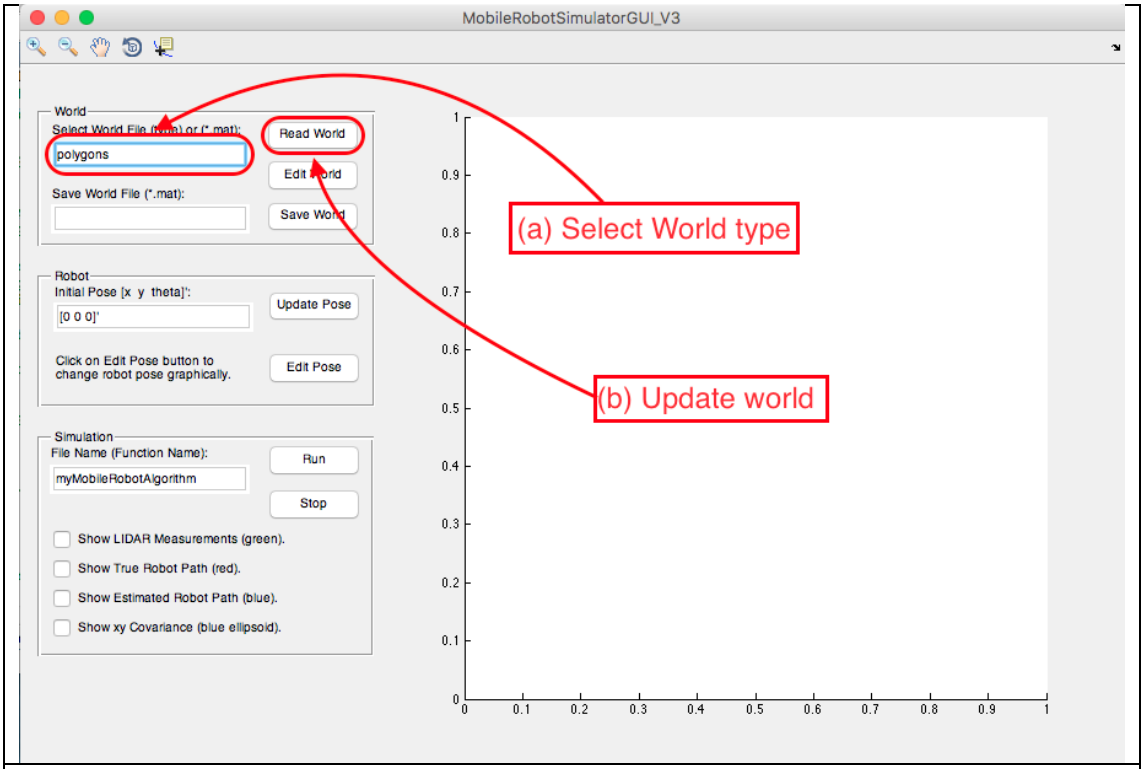


Figure 2.7: Select `polygons` World.

- The default polygons world is empty 2D space with lower-left corner at $[-5 \quad -5]^T$ and upper-right corner at $[5 \quad 5]^T$. Note the new parameters in the figure region of the GUI as shown in Figure 2.8.

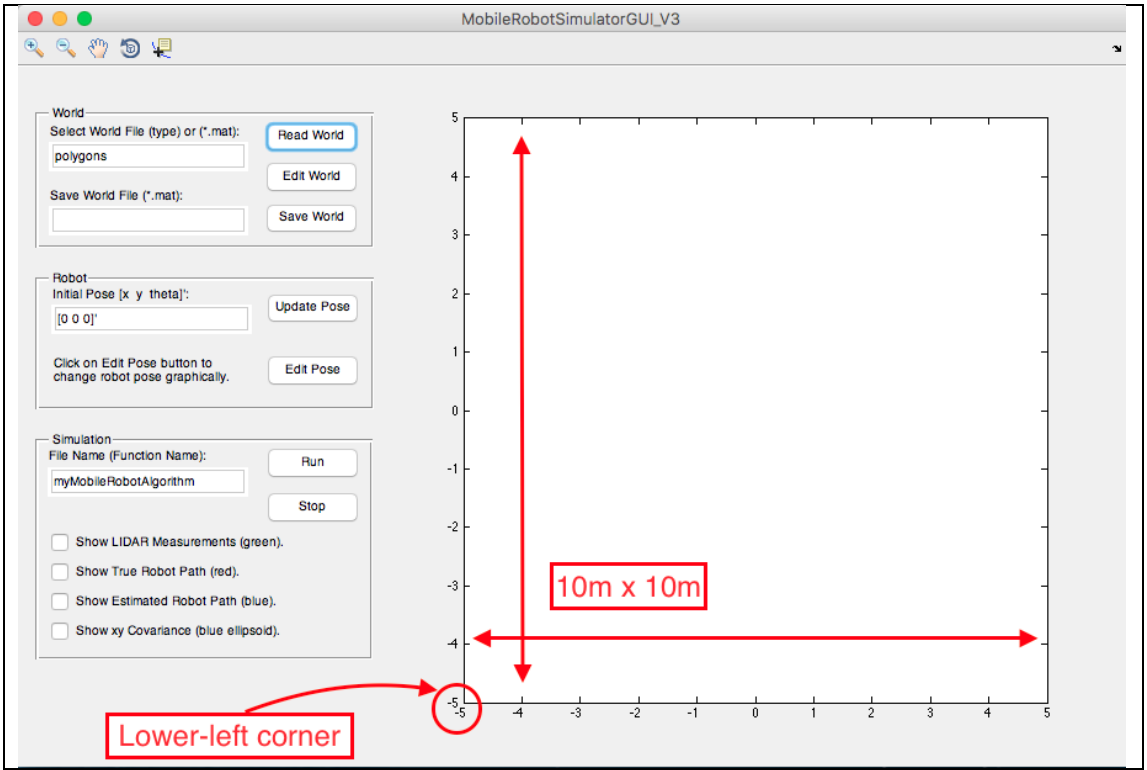


Figure 2.8: Empty `polygons` World.

- To add polygon obstacles to the world, click on `Edit World` as shown in Figure 2.9.

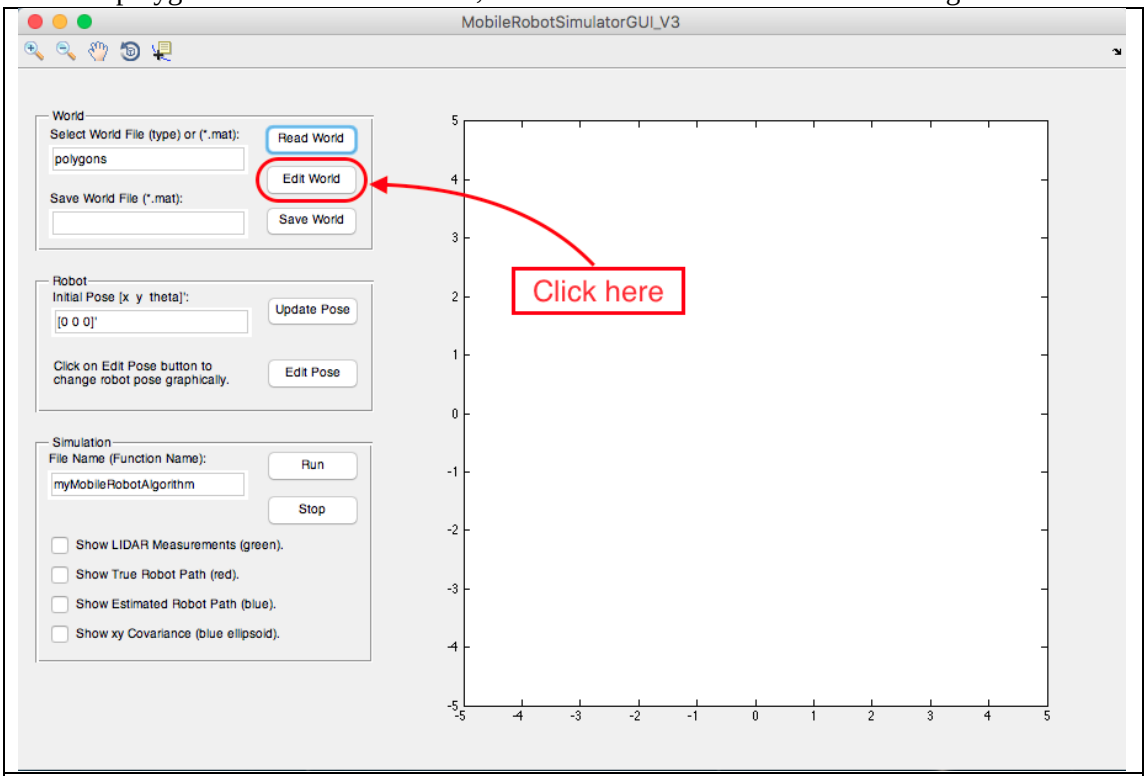


Figure 2.9: Edit `polygons` World.

- Depending on the world type, follow the instructions displayed at the top of the plot in the GUI as shown in Figure 2.10. Use the crosshair to create obstacles on the plot region inside the GUI.

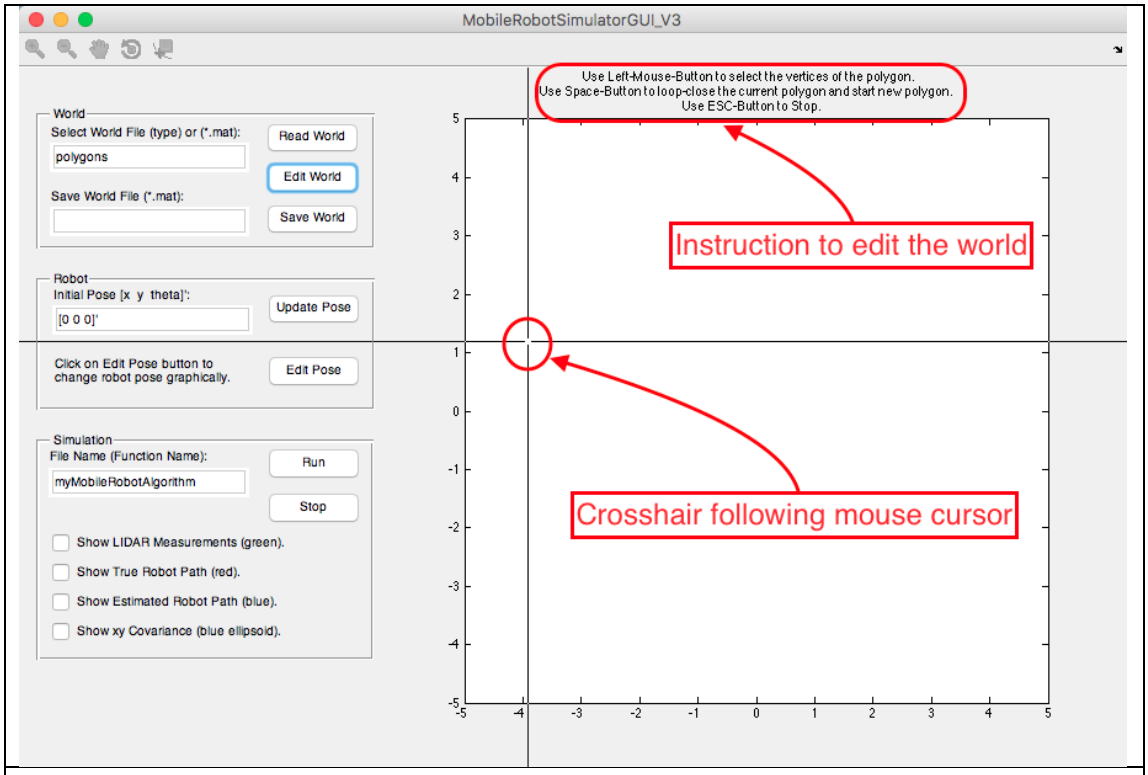


Figure 2.10: Edit `polygons` World.

- Since the selected world type is `polygons`, then, adding obstacles is done by specifying the polygon vertices. An example is shown in Figure 2.11.

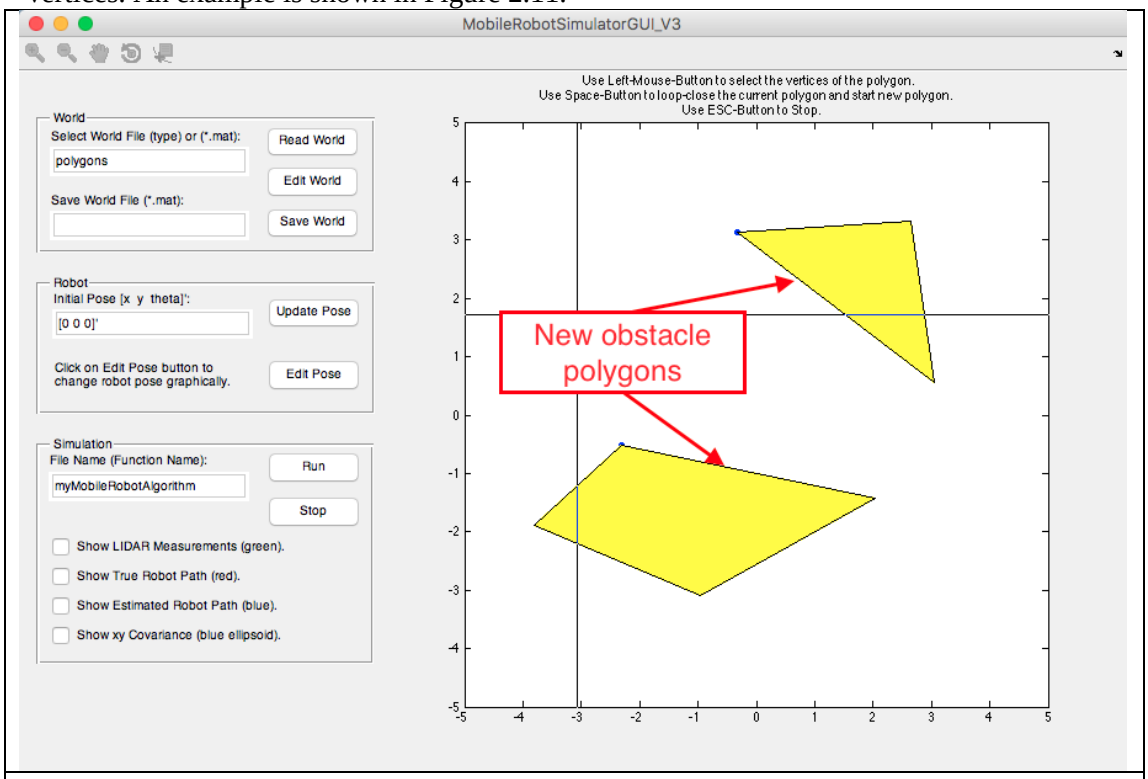


Figure 2.11: Edit `polygons` World.

- To save the current world, type the name of the file (no spaces), followed by the extension `.mat` in the field shown in Figure 2.12. This file is saved in `MobileRobotsSimulator_2017v1/Worlds` Folder.

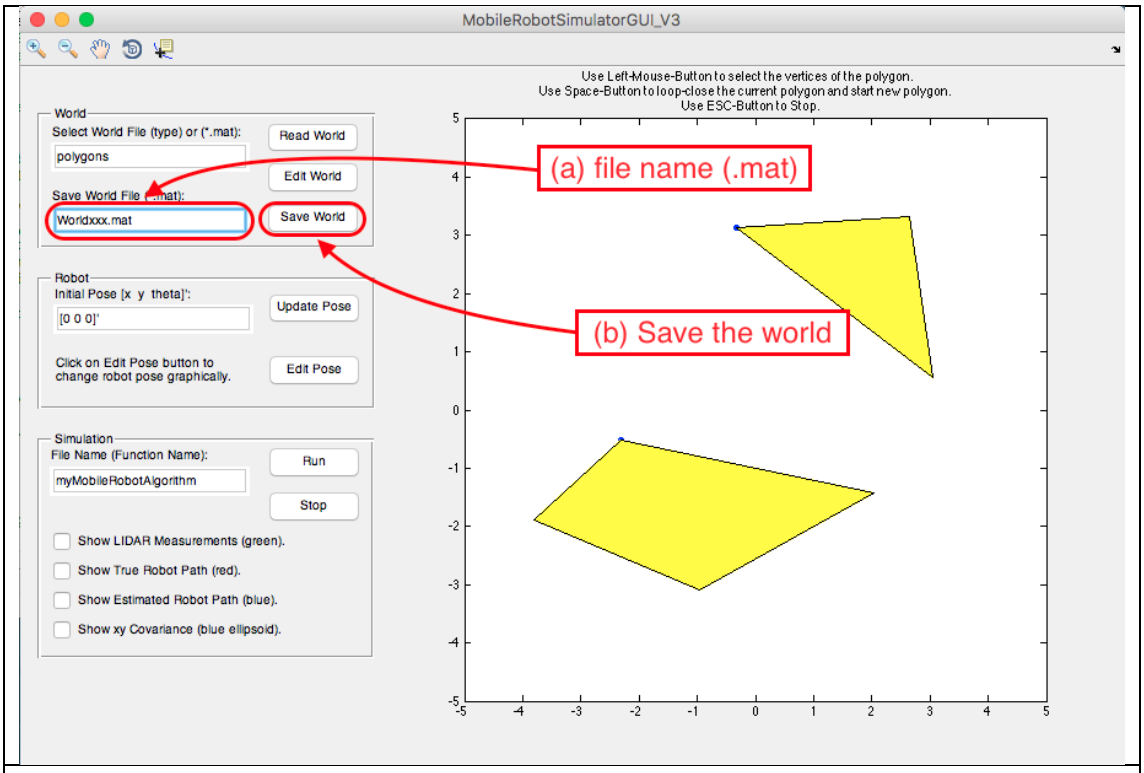


Figure 2.12: Save `polygons` World in `Worldxxx.mat` file.

- This simulator supports one robot type with the following specifications:
  - Differential-drive robot
  - Max radius = $0.5m$.
  - Wheel base = $0.4m$.
  - LiDAR that measures range and bearing over range of 360°.

- To place the robot in the current world, type the robot pose $[s_x \quad s_y \quad s_\theta]^T$ inside `Initial Pose` field as shown in Figure 2.13. The angle s_θ is in radian. Then, click on `Update Pose` button. The robot is shown an isosceles triangle pointing in the direction of motion.

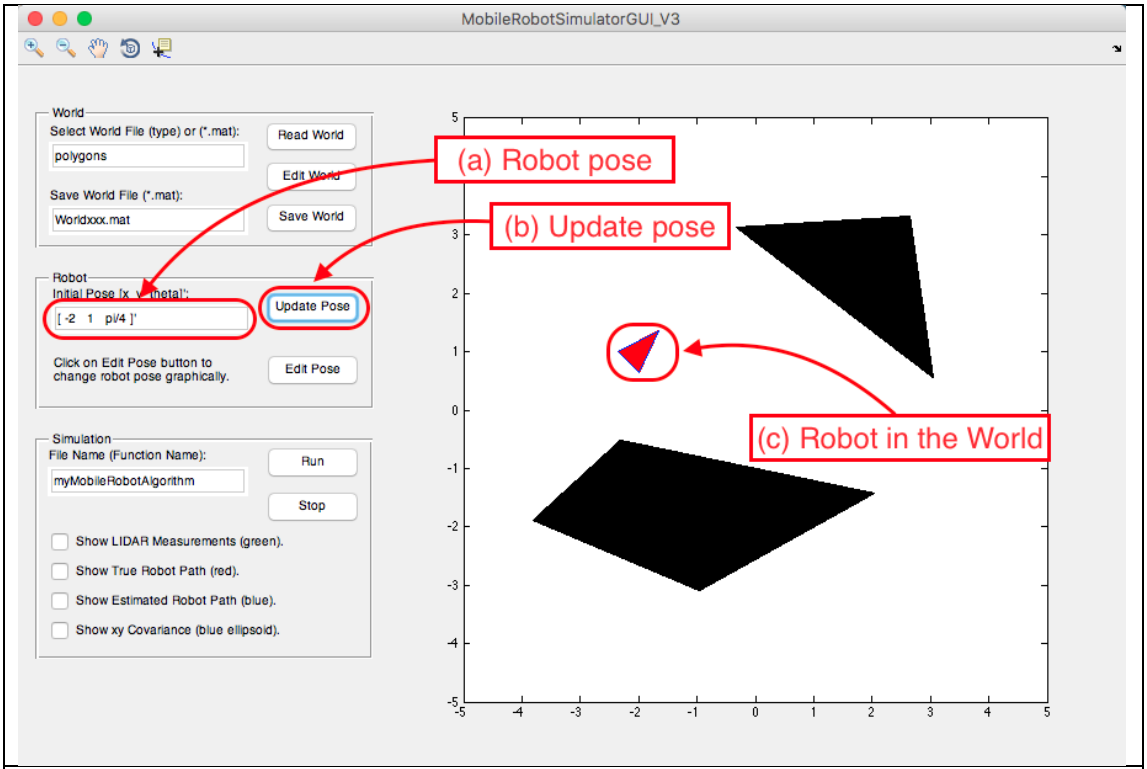


Figure 2.13: Place a robot in the world using initial pose field.

- An alternative method to change the pose is by clicking on `Edit Pose` button as shown in Figure 2.14. The instructions are as follows:
  - Use the mouse left-button to select the s_x and s_y. After the click, the robot will appear at the new location.
  - To change the orientation s_θ, left-click at any point around the robot. To finish editing, click on `Esc` button.

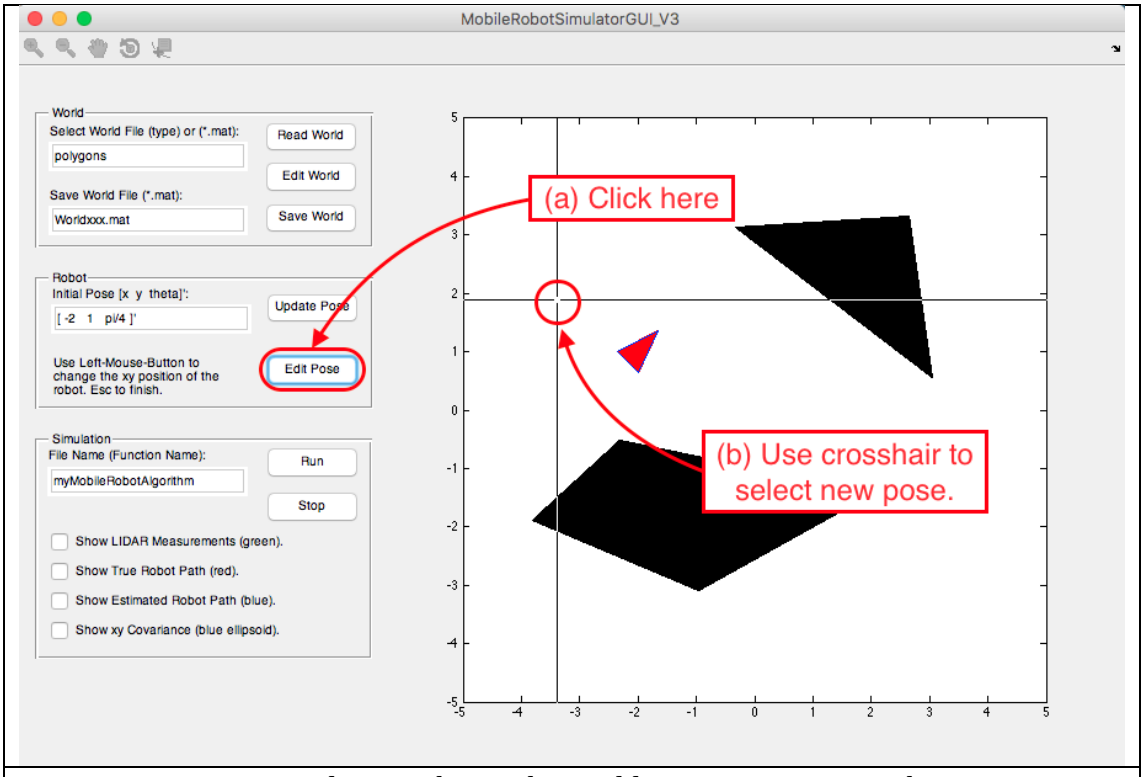


Figure 2.14: Place a robot in the world using `Edit Pose` button.

- Once the World is created and the Robot is positioned in the World, it is possible to simulate and control the robot by developing algorithms in MATLAB m-file. All programs to control the robot as inside `MobileRobotsSimulator_2017v1/Example` folder.

- The simulator default control program is called `myMobileRobotAlgorithm` as shown in Figure 2.15. The details of this program as explained in the next section.
    - Click on Run button.
    - The robot will not move (this is explained later). There are several lines appear in the top and bottom of the figure in the GUI. These lines show the time of simulation, the robot true pose, and the robot linear and angular velocities.

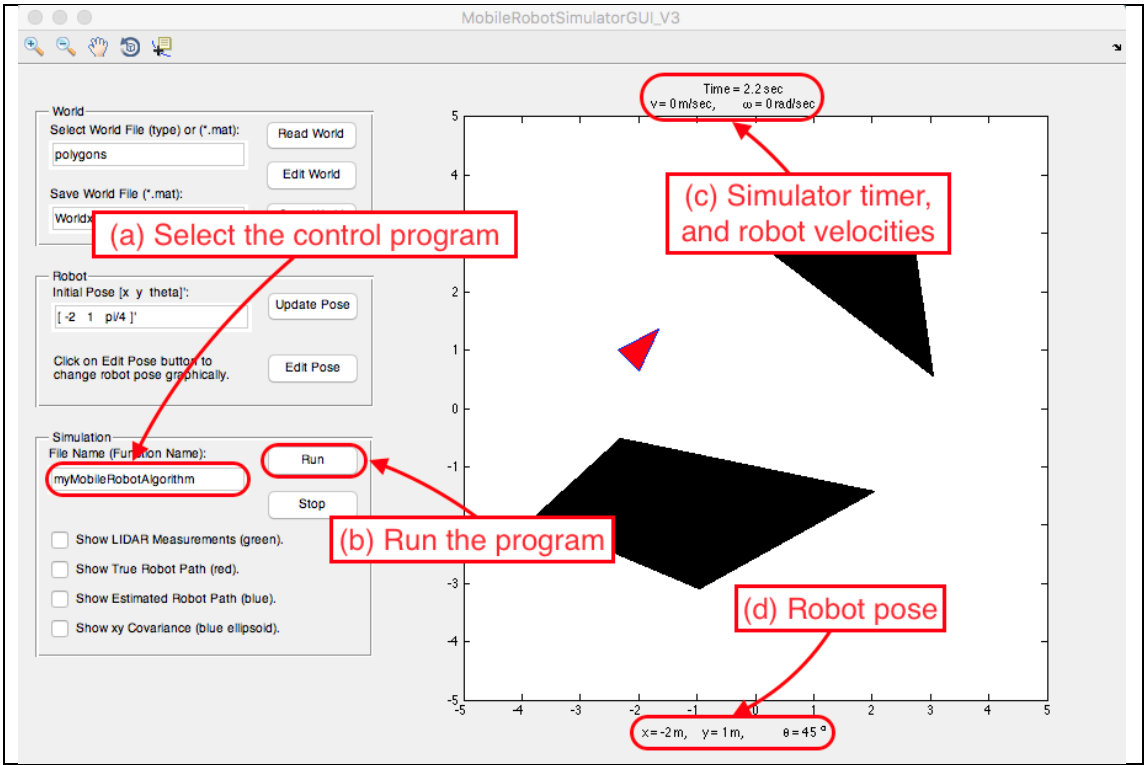


Figure 2.15: Run `myMobileRobotAlgorithm` program.

- Check `Show LiDAR Measurement (green)` box to view a subset of the measurements collected by the LiDAR as shown in Figure 2.16.

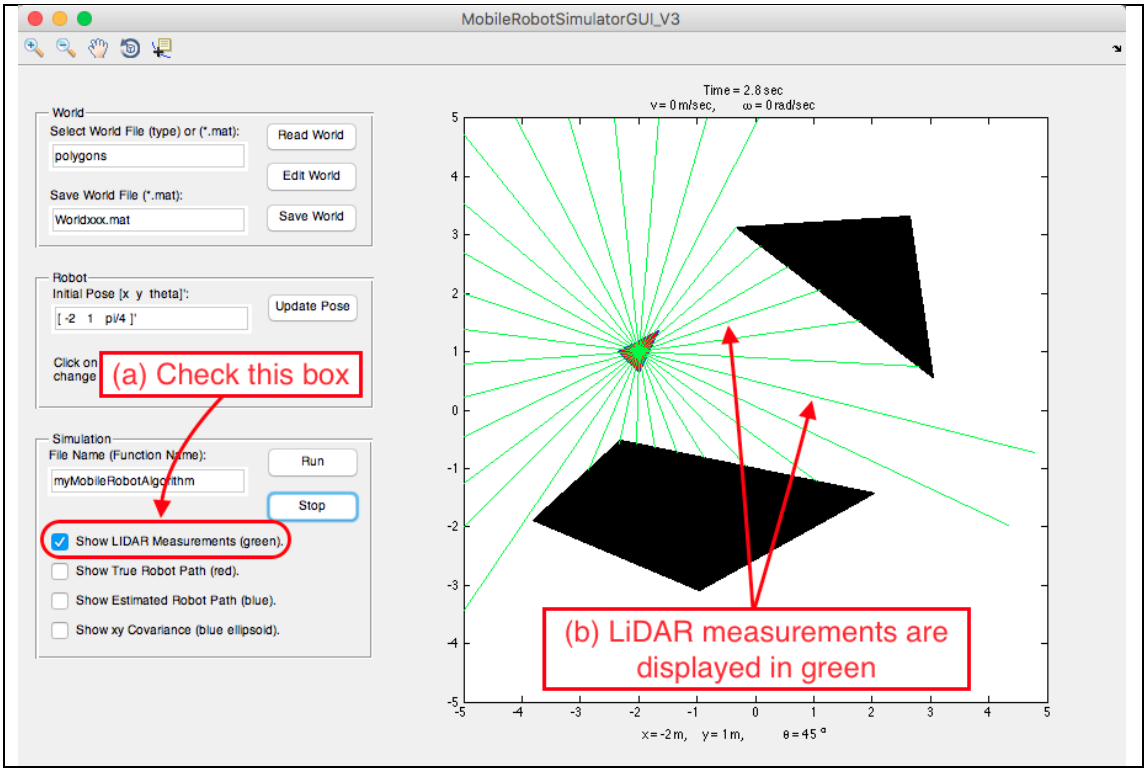


Figure 2.16: Show the LiDAR measurement in the simulator.

- To stop the simulator, click on Stop button as shown in Figure 2.17.

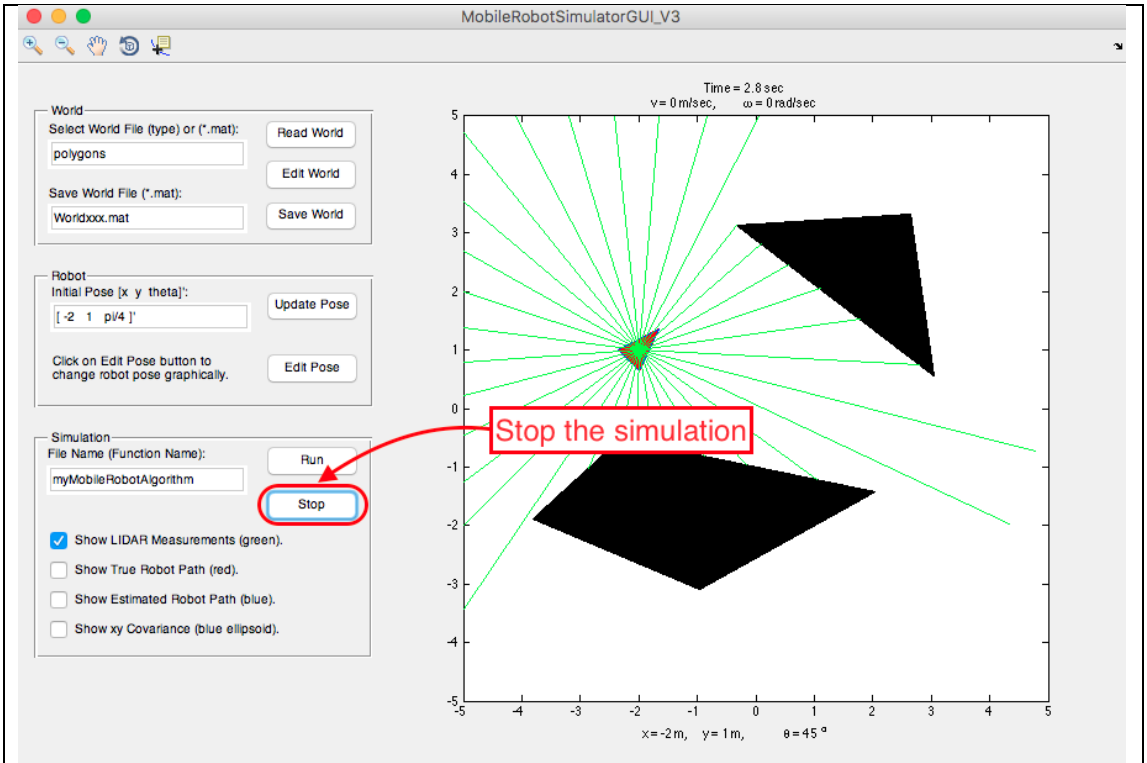


Figure 2.17: Stop the simulator.

3.0 How it works

When the simulation starts by clicking on `Run` button, the pseudo code in Algorithm 1 is executed.

Algorithm 1 MATLAB Mobile Robot Simulator

```
1:    function Run( )
2:        Initialization;
3:        while not Stop button (every 0.2 seconds)
4:            Update Sensor values;
5:            Evaluate control algorithm developed by the user. To move the robot,
              this algorithm needs to specify the robot linear and angular velocities;
6:            Apply the velocities to the robot;
7:            Plot everything;
```

Step 5 in Algorithm 1 is where the user m-file is applied, for instance `myMobileRobotAlgorithm`. Note that this function is called inside a while loop, thus, State Machines are very useful tools to develop efficient programs.

Before we start developing control programs, it is important to explore the different MATLAB structures used in this simulator. There are three basic structures:

(i) world, (ii) sensor, and (iii) control algorithm.

3.1 World Structure

3.1.1 Polygons World

The world structure is built when the world is created/read using the GUI. When `Read World` button is pressed, MATLAB simulator executes the following command:

```
W1 = worldGenerate2('polygons');
```

The return value `W1` is a structure with several fields. With `polygons` world the fields in `W1` are accessed and defined as follows:

Field	Value	Description
W1.Type	'polygons'	This field helps developing different interaction algorithms with different sensor types.
W1.LLcorner	$\begin{bmatrix}-5\\-5\end{bmatrix}$	2×1 vector representing the lower left corner of the world. x is the first element, and y is the second element.
W1.URcorner	$\begin{bmatrix}5\\5\end{bmatrix}$	2×1 vector representing the upper right corner of the world. x is the first element, and y is the second element.
W1.Polygons	[ ]	$1 \times n$ cell array where each cell corresponds to an obstacle polygon. Each polygon is represented by a 2D array that contains its vertices. The default number of polygons is zero.

To add obstacles in the form of polygons, use the procedure described in Section 2.0. It is possible to generate worlds with different dimensions. For example, consider the following command:

```
W1 = worldGenerate2('polygons',[-2 -3]',[4 5]');
```

This world has lower left corner at $[-2 \quad -3]^T$ and upper right corner at $[4 \quad 5]^T$. Note that the upper right vector must be larger than the lower left vector element-wise.

To plot this world in the current figure, use the following command:

```
worldPlot2(W1);
```

To edit this world similar to GUI as explained in Section 2.0, use the following command:

```
W1 = worldEdit2(W1);
```

To save the world in a mat-file, use the following command:

```
worldSave2(W1,'Worldxxx.mat');
```

Note that using the above command will save 'Worldxxx.mat in the current working directory of MATLAB.

There are other types of world that can be generated using `worldGenerate2` function, such as: `grid` and `simple`.

3.1.2 Grid World

In `grid` world, the environment consists of square cells of equal size that are arranged in a 2D grid. The size of the cell is determined by the resolution of the `grid` world. The size of the world is determined by the size of the lower left corner and the upper left corner. Each cell can be either occupied when it is assigned a value of 1, or unoccupied when it is assigned a value of 0.

To generate `grid` world, use the following command:

```
W1 = worldGenerate2('grid');
```

With `grid` world the fields in `W1` are accessed and defined as follows:

Field	Value	Description
`W1.Type`	`'grid'`	This field helps developing different interaction algorithms with different sensor types.
`W1.res`	0.1	Scalar representing the resolution of the grid, and it equals to the length and width of each cell in the grid.
`W1.corner`	$\begin{bmatrix} -5 \\ -5 \end{bmatrix}$	2×1 vector representing the lower left corner of the world. x is the first element, and y is the second element.
`W1.OGrid`	$\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$	100×100 array where each element represents a cell that is either empty with value of zero, or occupied with value of 1. This array is initialized with 0's.

It is possible to generate worlds with different dimensions. Consider for example the following command:

```
W1 = worldGenerate2('grid',[-2 -3]',[4 5]',1);
```

To edit the world use `worldEdit2` function and follows the instructions. To plot `W1` after adding obstacle, use `worldPlot2` function. The resulting plot is shown in Figure 3.1.

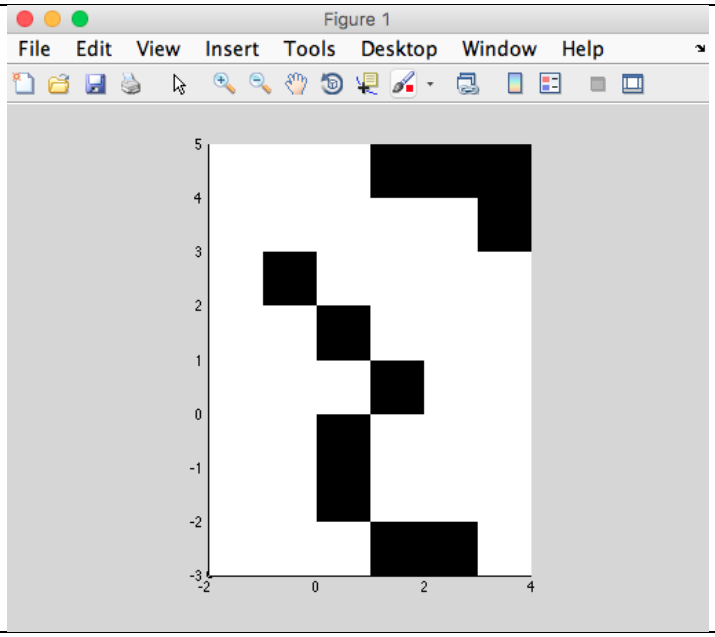


Figure 3.1: `grid` world with resolution of $1m$. Black regions are obstacles, whereas white regions are empty space.

Note that the cells in `grid` world can take values between 0 and 1. Such values represent the probability of the cell being occupied. Plotting such `grid` world will result in a grayscale figure where darker regions represent high probability of occupancy. This approach is helpful when Occupancy Grid mapping is implemented.

3.1.3 Simple World

In `simple` world, the obstacles are described in terms of point features.

To generate `simple` world, use the following command:

```
W1 = worldGenerate2('simple');
```

With `simple` world the fields in `W1` are accessed and defined as follows:

Field	Value	Description
`W1.Type`	`'simple'`	This field helps developing different interaction algorithms with different sensor types.
`W1.LLcorner`	$\begin{bmatrix}-5\\-5\end{bmatrix}$	2×1 vector representing the lower left corner of the world. x is the first element, and y is the second element.
`W1.URcorner`	$\begin{bmatrix}5\\5\end{bmatrix}$	2×1 vector representing the upper right corner of the world. x is the first element, and y is the second element.
`W1.Points`	[ ]	$2 \times n$ array where each column vector corresponds to an obstacle point. The default number of points is zero.

It is possible to generate worlds with different dimensions. Consider for example the following command:

```
W1 = worldGenerate2('simple',[-2 -3]',[4 5]');
```

To edit the world use `worldEdit2` function and follows the instructions. To plot `W1` after adding obstacle, use `worldPlot2` function. The resulting plot is shown in Figure 3.2.

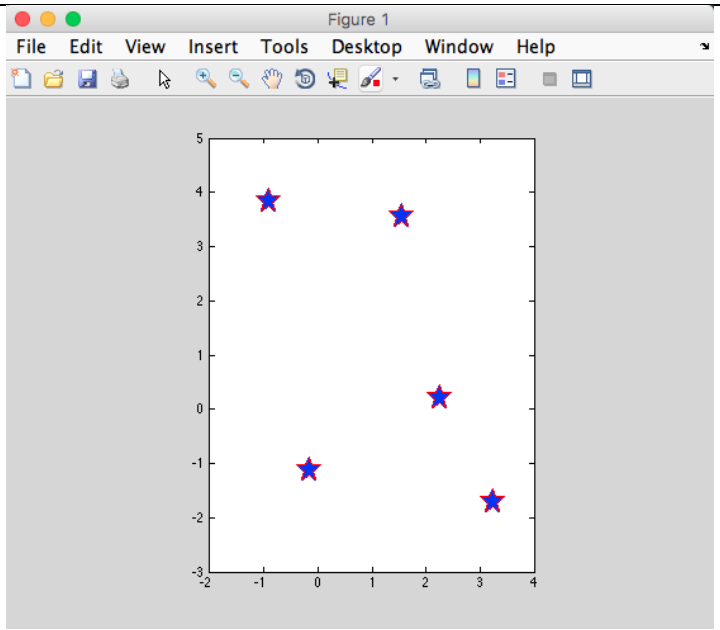


Figure 3.2: `simple` world with resolution with stars representing obstacles or landmarks.

3.1.4 Saved Worlds with `mat` extension

To load worlds that are already saved as mat-files, use the following command:

```
W1 = worldGenerate2('Worldxxx.mat');
```

All other World functions are applicable to `W1`.

3.2 Sensor Structure

The Sensor structure is built when the robot is created using the GUI. The fields inside the sensor structure depend entirely on the type of the current world. For example, if the current world has type of `grid` or `polygons`, then the sensor generated is created using the following command:

```
Sensor = sensorGenerate1('Hokuyo_URG_04LX_UG01.txt');
```

The text file `'Hokuyo_URG_04LX_UG01.txt'` is inside `MobileRobotsSimulator_2017v1/HardwareTypes` folder, and it has all specification of the LiDAR used for simulation.

The return value `Sensor` is a structure with several fields defined as follows:

Field	Value	Description
`Sensor.Type`	`'LIDAR'`	This is a string to distinguish from other types.
`Sensor.MaxRho`	7	Scalar (in meters) that represents the maximum distance the sensor can measure.
`Sensor.MaxTheta`	360	Scalar (in degrees) that represents the angular of the sensor.
`Sensor.ResTheta`	0.3519	Scalar (in degrees) that represents the angular resolution of the sensor.
`Sensor.CovRho`	1.44e-4	Scalar (in meters) that represents the variance of the measured distance.
`Sensor.CovTheta`	0	Scalar (in degrees) that represents the variance of the measured angle.
`Sensor.theta`	$[-\pi \quad \cdots \quad \pi]$	1×1024 vector in radian that represents each angle over the angular span on the sensor. The difference between consecutive angle in this vector equals to `Sensor.ResTheta`.
`Sensor.rho`	$[d_1 \quad \cdots \quad d_{1024}]$	1×1024 vector in meters that represents the distance measurements at each angle.
`Sensor.rhoCov`	$[c_1 \quad \cdots \quad c_{1024}]$	1×1024 vector in meters that represents the variance of each distance measurements. In this sensor type, this vector has the same value at all elements that equals to `Sensor.CovRho`.

3.3 Control Algorithm Structure `MyAlg`

This structure is where the user can develop control strategies for the mobile robot. This structure is developed in an m-file created by the user and saved inside `MobileRobotsSimulator_2017v1/Examples` folder. The m-file can have any name, but it must have a certain function definition as follows:

```
function MyAlg = myMobileRobotAlgorithm(MyAlg,Sensor)
% Your code
return
```

The name of m-file containing this code must have the same name, i.e., `myMobileRobotAlgorithm.m`. The inputs of this function should be exactly as shown above `(MyAlg,Sensor)`. Also, the output of the function should always be `MyAlg`.

`MyAlg` is a structure that contains several fields. Some of these fields are necessary for simulation, whereas others are necessary for specific algorithms. Note that `MyAlg` is an input and output of this file. This means that the user can save some parameters to be used later in the algorithm. Remember that this function is called inside the simulation loop as illustrated in Algorithm1. The user is free to add any new fields that the algorithm demands. This capability facilitates the usage of state machines as illustrated in the next section.

The basic and necessary fields in `MyAlg` are as follows:

Field	Value	Description
`MyAlg.time`	…	Scalar (in seconds) that represents the current time since the beginning of the simulation. Do not change this parameter.
`MyAlg.firstTime`	…	Boolean flag that is set to 1 during the first iteration of the while loop in Algorithm 1. Then, it stays 0 until for the rest of simulation. This flag is used to initialize the state machine. Do not change this parameter.
`MyAlg.v`	…	Scalar that represents the robot linear velocity in m/sec. The user can modify this parameter to command the robot to move in the next time step.
`MyAlg.w`	…	Scalar that represents the robot angular velocity in rad/sec.

		The user can modify this parameter to command the robot to move in the next time step.
`MyAlg.motionCov`	…	3 × 3 matrix that represent the motion noise with respect to the robot pose. This noise is in the form of covariance matrix. Do not change this parameter.
`MyAlg.initial_pose`	…	3 × 1 vector that represents the robot initial pose. Do not change this parameter.
`MyAlg.robot_wb`	…	Scalar (in meters) that represents the robot maximum diameter. Do not change this parameter.
`MyAlg.isDone`	…	Boolean flag that is used to alert the simulator to stop, even if the `Stop` button is not pressed. If the value is 0, the simulator continues the simulation. If the value is 1, the simulator terminates the while loop in Algorithm 1 and stops the simulation. The user can change this flag when the algorithm completes the proposed tasks.

4.0 Examples of Control Algorithms

The folder `MobileRobotsSimulator_2017v1/Examples` contains all examples presented below.

4.1 Drive Straight

- Create a new m-file and save it as `driveStraight.m` inside `MobileRobotsSimulator_2017v1/Examples` folder. Type the following program inside this m-file:

```
function MyAlg = driveStraight(MyAlg,Sensor)
if MyAlg.time<10
    MyAlg.v = 0.3;          % linear velocity (m/sec)
    disp('Driving...')
else
    MyAlg.v = 0;
    disp('Stop!')
    MyAlg.isDone = 1;       % stop simulation
end
return
```

- In the simulator GUI, select an empty polygons world, and place the robot at the origin with $\mathbf{s}_0 = [0 \quad 0 \quad 0]^T$.

- In `File Name (Function Name)` field in the GUI, type the name of the above function with the extension, e.g., `driveStraight`, as shown in Figure 4.1.

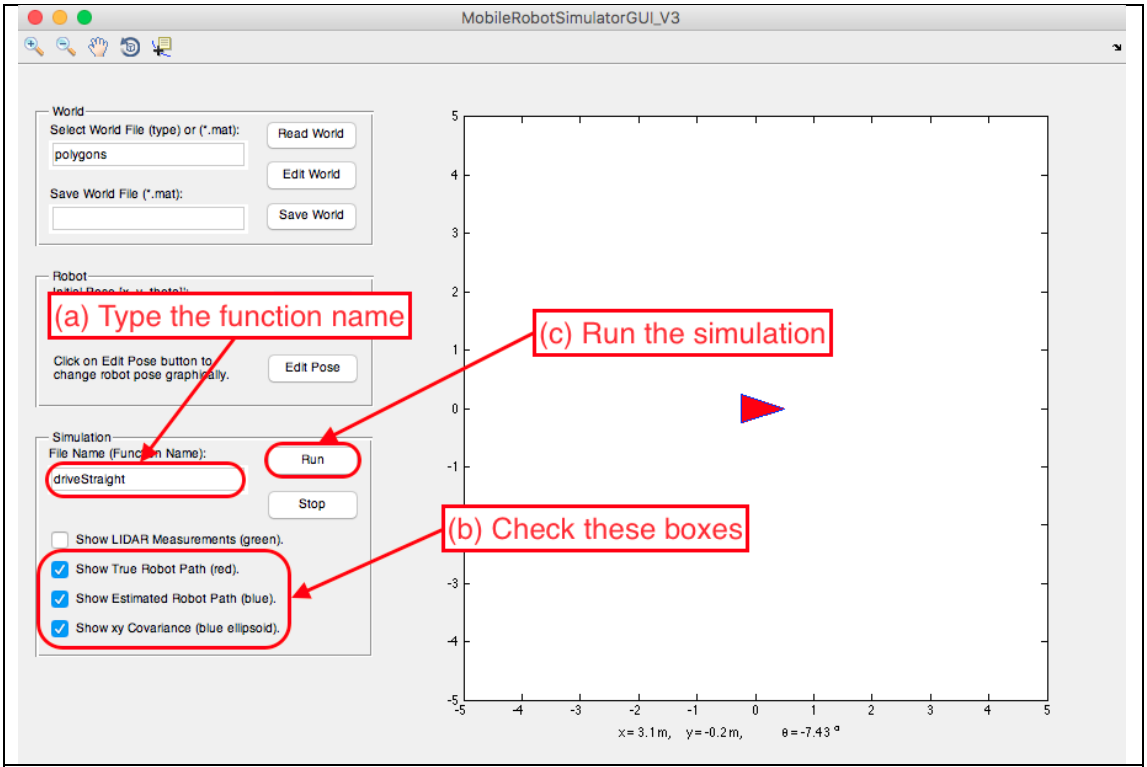


Figure 4.1: Drive straight example before simulation.

- Figure 4.2 shows the result at the end of the simulation. Note that the simulation terminates after about 10 seconds. By observing the program in `driveStraight.m` file, there is an `if-else` statement that commands the robot to move with linear velocity of 0.3 m/sec if the timer is less than 10 seconds. Otherwise, the robot stops and `MyAlg.isDone` flag is set to value of 1 to force the simulator to stop.

- This program is the simplest form of state machine, and it does not have any loops because there is an outer while loop.

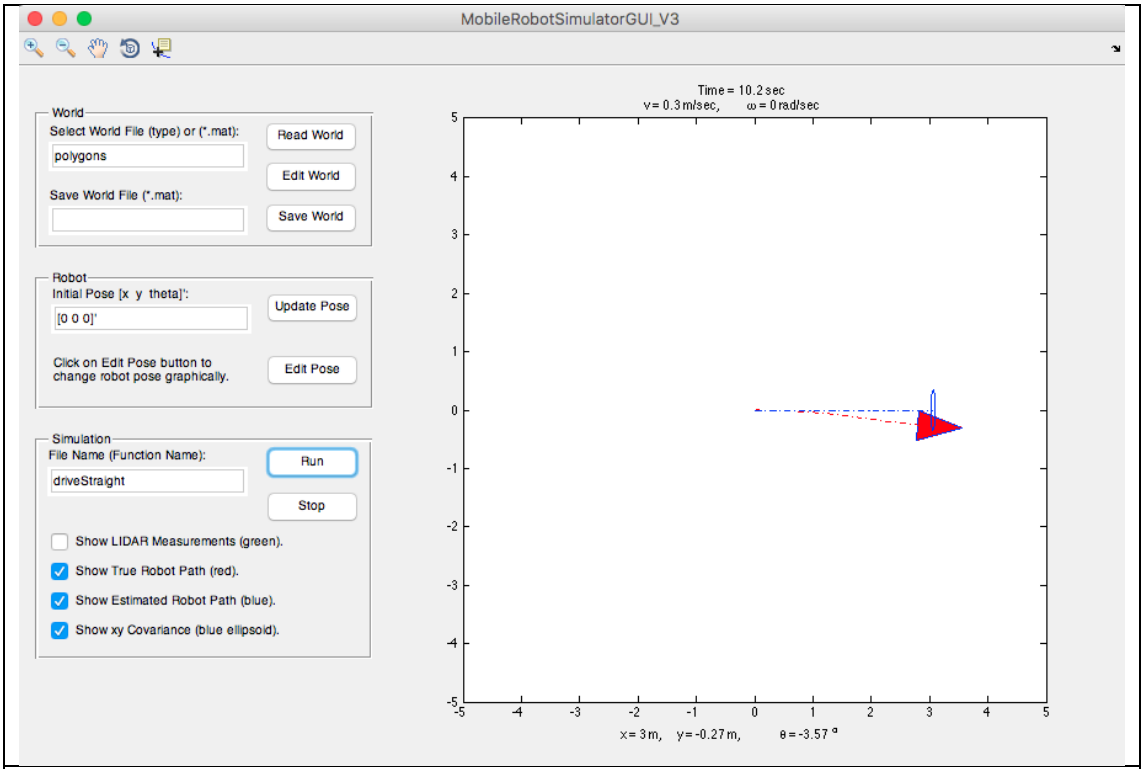


Figure 4.2: Drive straight example after simulation.

4.2 Drive in a Circle

- Create a new m-file and save it as `driveCircle.m` inside `MobileRobotsSimulator_2017v1/Examples` folder. Type the following program inside this m-file:

```
function MyAlg = driveCircle(MyAlg,Sensor)
R = 2;       % radius (meters)
v = 0.4;     % linear velocity
w = v/R;     % angular velocity
if MyAlg.time<30
    MyAlg.v = v;          % linear velocity (m/sec)
    MyAlg.w = w;          % angular velocity
    disp('Driving...')
else
    MyAlg.v = 0;
    MyAlg.w = 0;
    disp('Stop!')
    MyAlg.isDone = 1;        % stop simulation
end
return
```

- In the simulator GUI, select an empty polygons world, and place the robot at the origin with $\mathbf{s}_0 = [0 \quad 0 \quad 0]^T$.

- In `File Name (Function Name)` field in the GUI, type the name of the above function with the extension, e.g., `driveCircle`.

- Figure 4.3 shows the result at the end of the simulation. Note that there is difference between the robot estimated pose and the robot true pose. The ellipsoid illustrates the robot pose uncertainty in $[s_x \quad s_y]^T$, and it grows over time because the robot here uses its motion model only to estimate its pose.

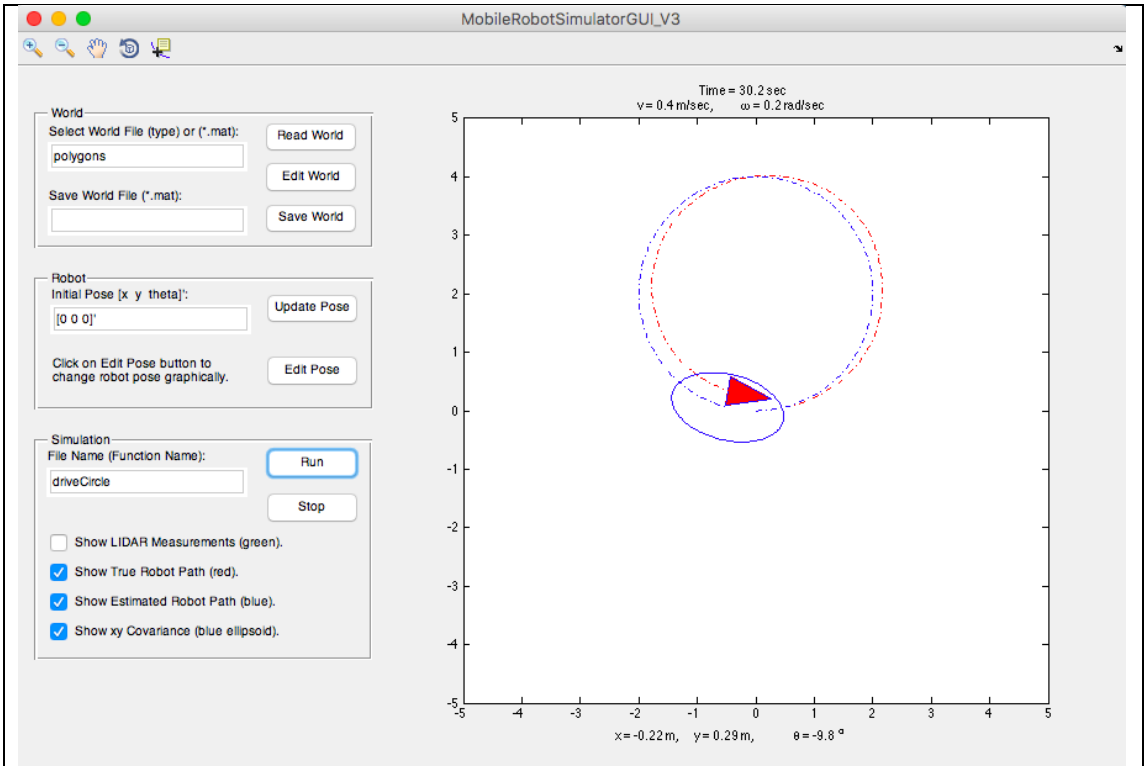


Figure 4.3: Drive in a circle example after simulation.

4.3 Drive in a Square

- Create a new m-file and save it as `driveSquare.m` inside `MobileRobotsSimulator_2017v1/Examples` folder. Type the following program inside this m-file:

```
function MyAlg = driveSquare(MyAlg,Sensor)
% Initialization
if MyAlg.firstTime
    MyAlg.time1 = MyAlg.time;    % save current time
    MyAlg.state = 1;
    MyAlg.side_counter = 0; % for stopping condition
end

% (a) Select the state
% --------------------
if MyAlg.state==1
    if MyAlg.time-MyAlg.time1>=6
        MyAlg.side_counter = MyAlg.side_counter+1;
        if MyAlg.side_counter==4
            disp('Done!')
            MyAlg.state = 3;
        else
            disp('Turn 90 degrees.')
            MyAlg.state = 2;
            MyAlg.time1 = MyAlg.time;
        end
    end
elseif MyAlg.state==2
    if MyAlg.time-MyAlg.time1>=5
        disp('Straight.')
        MyAlg.state = 1;
        MyAlg.time1 = MyAlg.time;
    end
elseif MyAlg.state==3
    MyAlg.isDone = 1;        % stop simulation
end

% (b) Apply the state
% -------------------
if MyAlg.state==1        % straight
    MyAlg.v = 0.4;
    MyAlg.w = 0;
elseif MyAlg.state==2    % turn
    MyAlg.v = 0;
    MyAlg.w = 0.3;
elseif MyAlg.state==3    % stop and terminate
    MyAlg.v = 0;
    MyAlg.w = 0;
end
return
```

- In the simulator GUI, select an empty polygons world, and place the robot at the origin with $\mathbf{s}_0 = [0 \quad 0 \quad 0]^T$.

- In `File Name (Function Name)` field in the GUI, type the name of the above function with the extension, e.g., `driveSquare`.

- Figure 4.4 shows the result at the end of the simulation. The above program shows how the state machine is used to drive the robot in a square shape. The basic idea is to have three states: (i) drive straight for 6 seconds, (ii) turn 90° CCW, and (iii) stop and terminate simulation. Transition between these states is decided using a series of `if-else` statements. Again, there is no loops in this program.

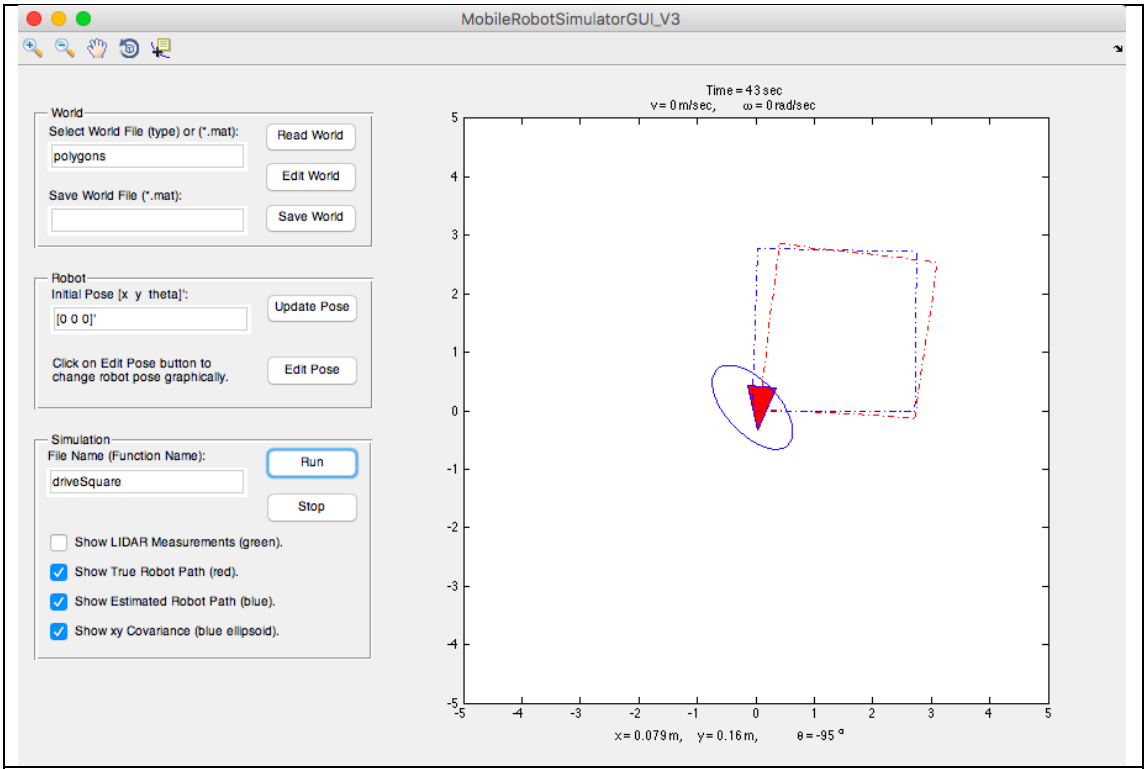


Figure 4.4: Drive in a square example after simulation.

MATLAB Mobile Robot Simulator
List of Functions

Category	Function Name	Purpose	Examples of Usage	Explanation
World Functions	worldGenerate2	Create a new world (or read an existing world) for robot to operate.	`W1 = worldGenerate2('grid');`	W1 is an empty 2D **grid** world, with lower left corner located at (-5m, -5m), upper right corner at (5m, 5m), and cell resolution of 0.1m.
			`W1 = worldGenerate2('grid',[0 0]',[10 20]');`	W1 is an empty 2D **grid** world, with lower left located at corner (0m, 0m), upper right corner at (10m, 20m), and cell resolution of 0.1m. All vectors are column vectors.
			`W1 = worldGenerate2('grid',[0 0]',[10 20]',1);`	W1 is an empty 2D **grid** world, with lower left located at corner (0m, 0m), upper right corner at (10m, 20m), and cell resolution of 1m. All vectors are column vectors.
			`W2 = worldGenerate2('polygons');`	W2 is an empty 2D **polygons** world, with lower left corner located at (-5m, -5m), upper right corner at (5m, 5m).
			`W2 = worldGenerate2('polygons',[0 0]',[10 20]');`	W2 is an empty 2D **polygons** world, with lower left corner located at (0m, 0m), upper right corner at (10m, 20m). All vectors are column vectors.
			`W3 = worldGenerate2('simple');`	W3 is an empty 2D **simple** world, with lower left corner located at (-5m, -5m), upper right corner at (5m, 5m).
			`W3 = worldGenerate2('simple',[0 0]',[10 20]');`	W3 is an empty 2D **simple** world, with lower left corner located at (0m, 0m), upper right corner at (10m, 20m). All vectors are column vectors.
			`W4 = worldGenerate2('World1.mat');`	Read and load mat-file with name 'World1.mat' to W4 data structure. The mat-file should be in the Matlab path.
	worldEdit2	Edit an existing world using mouse and keyboard.	`W1 = worldEdit2(W1);`	Edit world W1. Follow the instructions in Figure window.
	worldPlot2	Plot an existing world in the current figure.	`worldPlot2(W1);`	Plot world W1 on the current figure.
	worldSave2	Save an existing world as a mat-file	`worldSave2(W1,'World1.mat');`	Save world W1 as a mat-file in the current working directory with the name World1.mat.

Category	Function Name	Purpose	Examples of Usage	Explanation
Robot Functions	robotGenerate1	Create a robot and place it in the existing world.	`R1 = robotGenerate1('P3_DX.txt');`	Create a differential-drive robot with parameters defined in a text file named 'P3_DX.txt'. The text file needs to be in Matlab path. The robot is placed at the origin (0m, 0m) of the existing world and facing the positive direction of the world x-axis.
			`R1 = robotGenerate1('P3_DX.txt', [1 3 pi/4]');`	Create a differential-drive robot with parameters defined in a text file named 'P3_DX.txt'. The text file needs to be in Matlab path. The robot is placed at (1m, 3m) with respect to the existing world and pi/4 radians with positive direction of the world x-axis. All vectors are column vectors.
	robotPlot1	Plot the robot in the current figure.	`robotPlot1(R1);`	Plot robot R1 in the current figure. To plot the robot on the existing world, make sure to use <hold on> before calling this function, then use <hold off> after this function. The robot appearance is simple (Isosceles triangle to represent the robot current orientation).
			`robotPlot1(R1, 'Simple');`	Robot appearance is Isosceles triangle.
			`robotPlot1(R1, 'Robot1');`	Robot appearance is a differential-drive robot.
	robotUpdate1	Update the robot pose given control inputs and change in time.	`R1 = robotUpdate1(R1, [0.3, 0.1, 0.2]');`	Robot linear velocity is 0.3m/sec, robot angular velocity is 0.1rad/sec (+ CCW, - CW), and change in time is 0.2sec.
Sensor Functions	sensorGenerate1	Create a sensor and place it the current robot.	`S1 = sensorGenerate1('SimpleDepth2.txt');`	Create a simple depth sensor in the robot that can measure the distance and angle of point features in the world. The parameters of the sensor are defined in SimpleDepth2.txt file.
			`S1 = sensorGenerate1('Hokuyo_URG_04LX_UG01.txt');`	Create depth sensor in the robot that can simulates Hokuyo URG_04LX_UG01 Lidar. The parameters of the sensor are defined in Create a simple depth sensor in the robot that can measure the distance and angle of point features in the world. The parameters of the sensor are defined in Hokuyo_URG_04LX_UG01.txt file.
	sensorState3	Update the sensor state (sensor measurements) taking into account the robot pose and the world.	`S1 = sensorState3(S1, R1, W1);`	Update the range vector <rho> and the bearing vector <theta> inside S1 based on the robot pose inside R1 and the world W1.
	sensorPlotMeasurements2	Plot the sensor measurements given the robot pose in R1.	`sensorPlotMeasurements2(S1, R1);`	Plot the measurements of "DEPTH" sensor or "LIDAR" sensor in the world frame {W} using the robot pose in R1. The plot is in the current figure.
Other Functions	getBresenhamLine	Find the index of cells that intersect the line passing through two points in a grid world.	`ind = getBresenhamLine(W1, [0 1; 2 3]');`	This function works with grid worlds only in W1. <ind> is (1xn) vector with absolute indeces of cells in the grid that intersect the line segment between the point (0,1) and the point (2,3).
	reduce_angle_fullCirlce	Wrap angle in radians to [−pi pi].	`phi = reduce_angle_fullCircle(5*pi/2);`	By subtracting multiple 2π from 5π/2, the value of <phi> is π/2.