Nele Raya Traichel

Context Steering with Differential-Drive Robots: Reactive Navigation based on Multi-Objective Decision-Making

FAKULTÄT FÜR
INFORMATIK

Intelligent Cooperative Systems
Computational Intelligence

Context Steering with Differential-Drive Robots: Reactive Navigation based on Multi-Objective Decision-Making

Master Thesis

Nele Raya Traichel

June 28, 2022

Supervisor: Prof. Sanaz Mostaghim

Advisor: Sebastian Mai

Nele Raya Traichel: *Context Steering with Differential-Drive Robots: Reactive Navigation based on Multi-Objective Decision-Making*
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2022.

Abstract

Although coordinated motion is a basic swarm behavior in Swarm Intelligence, it still poses a major challenge in Swarm Robotics, mostly due to limitations in perception, actuation (kinematics) and computation. Steering algorithms are commonly used, making short-term choices on the next movement, considering only the local environment. However, prior work with a swarm of Turtlebot3 Burger robots (Driving Swarm) has shown, the original steering algorithm can lead to deadlocks. As an extension of the context steering approach, which again is an advancement of the classic steering algorithms, the context steering using multi-objective optimization and decision-making has proven to reduce the risk of deadlocks.
With the aim of eventually improving the behavior of the Driving Swarm, this work adapts the multi-objective context steering approach to the differential-drive kinematics and sensors of the Turtlebot3 Burger robot. The resulting implementation is tested in a simulation with one robot that has to navigate from its start position to a known goal position, in scenarios where classical steering fails due to a deadlock. The performance limits are examined in two main simulation experiments, each with five different multi-objective decision-making methods in five scenarios of increasing complexity, using two sets of parameters. It is demonstrated which features in the environment provoke collisions or deadlocks, and where the strengths and weaknesses of the decision-makers and various parameter settings lie. Moreover, advantages and disadvantages of the algorithm are discussed, providing an orientation for a later implementation with a swarm.

Contents

List of Figures

List of Tables

1 Introduction

Navigating towards a goal while coordinating all movements and avoiding collisions with obstacles or other robots is a basic swarm behavior (so-called coordinated motion [43]) in Swarm Intelligence, but still a major challenge in Swarm Robotics. Being restricted to local information gathering and on-board processing requires an inexpensive algorithm. At the same time, it should be flexible, robust and scalable and create an emergent swarm behavior, where flexibility refers to the ability to perform well regardless of the environment.

Researchers creating intelligent agents in games face the same problem of motion control. Hence, this field of research provides a solid base of algorithms for agents perceiving and searching in its local environment only. The steering behaviors introduced by Reynolds [39, 40] based on attraction and repulsion forces provide a portfolio of simple behaviors. Each of them returns a velocity vector with the preferred direction and speed (magnitude). The vectors are aggregated to determine the final movement direction. Through the aggregation of multiple simple behaviors, a complex behavior emerges. Prioritization and/or weighting methods can be used to modify the agent's behavior in desired ways.
Due to its simple but effective concept, the described steering algorithm was implemented in prior works for the task of coordinated motion with a swarm of Turtlebot3 platforms [32], a robot with non-holonomic differential-drive kinematics [41]. However, the results of the experiments conducted in reality and in simulation have shown the limits of steering. Especially in environments with a large obstacle, like a wall, the robot can get stuck behind it (stopping or oscillating) with the goal being directly on the other side.
Context steering in games [18, 19] aims to improve the movements of agents to avoid deadlocks and make them behave more natural. For this, the aggregation process is split, and more information is incorporated into the decision-making process. According to Fray [19], the reason why an option is excluded or preferred (the *context*) is just as important as the ultimate decision itself. That is

why, a so-called context map is created. For each possible movement direction, it indicates the willingness of moving towards that direction (interest map), or avoiding it (danger map).
As an extension of context steering, Dockhorn et al. [13] highlight the multi-objective nature of the aggregation and decision-making process. After all, interesting and dangerous items can be located in the same direction, leading to conflicting objectives, which makes the decision on a direction of movement a multi-objective problem. This approach has proven to reduce the risk of deadlocks and to overcome weaknesses of classical steering approaches in situations with small circular obstacles and walls using different decision-making methods.

Now, with regard to the Turtlebot3 Burger robots, the central research question is whether **multi-objective context steering applied to differential-drive robots might succeed in scenarios where classic steering ends up in a deadlock**. In search of an answer, this thesis introduces an implementation of context steering with multi-objective decision-making for this robot platform, and evaluates its performance with five different decision-making methods in five simulated scenarios of increasing complexity. In these scenarios, classic steering cannot fulfill the given task, to navigate the robot from its starting to a target position without hitting obstacles. The work focuses on robotics, considering the robot-specific restrictions in terms of its perception and actuation, as well as the fact that the system is non-deterministic. This is why the algorithm is designed and tested with one robot only, providing a baseline for a later implementation with a swarm.

Further research questions that arise along with the main question stated above and will be addressed in this thesis are:

- **Q1**: Can the robot reach the target while navigating around small circular obstacles, a wide wall, and an L-shaped obstacle?
- **Q2**: Can the robot fulfill the given navigation task in a realistic environment?
- **Q3**: With which decision-maker is the robot safest, fastest, and most flexible?
- **Q4**: What is the effect of the decision-making parameters danger weight, interest constraint, and danger constraint on the risk-aversion of the robot?

The next chapter 2 describes work related to navigation with agents and navigation in robotics. Moreover, it provides basic principles on multi-objective decision-making and kinematics of differential-drive robots. In chapter 3, first an overview of the algorithm structure is given, followed by a detailed explanation of the individual components. The design of the conducted simulation experiments, its setup, evaluation metrics and chosen parameters are described in chapter 4. The experiment results are presented and evaluated in chapter 5. The last chapter 6 states conclusions that can be drawn from the findings of the experiments with regard to the research questions, and challenges left to be addressed in future work.

2 Related Work

Since the problem of navigation has many fields of application, it is widely researched. In the sections of this chapter, first an overview on navigation and movement behaviors of intelligent agents in games is given (see section 2.1.1). In the subsequent section 2.2, an insight into navigation with mobile and especially differential-drive robots is given. Section 2.3 compares the characteristic features of the related work with the algorithm introduced in this thesis.

2.1 Navigation of Agents

The desire to create an intelligent game agent behaving naturally and unobtrusively with low computational effort led to an evolution of steering algorithms. In the following sections the original steering behaviors (see section 2.1.2), its advancement, context steering (see section 2.1.3), and the most recent publication using context steering with multi-objective optimization and decision-making 2.1.4 are outlined, preceded by an introduction into autonomous agents (see section 2.1.1).

2.1.1 Autonomous Agent

Developed in the research field of Artificial Intelligence (AI), an autonomous or intelligent agent is a software program that uses sensors and effectors to interact with its environment in pursuit of a goal, independent of a user's direct instructions. Based on the perceived data from the environment, limited to the capabilities of the sensors, the agent selects an action [45].
The scope and abilities of an agent vary with its application domain, ranging from a simple stimulus-response-agent to complex agents using AI techniques such as learning or inter-agent communication [9]. Reactive agents have no memory, they consider only the current state of the environment or an event,

and respond directly to the perceptions with a suitable action. In contrast, proactive agents do not wait for an event in the environment, but plan ahead by choosing a sequence of actions, anticipating future states and events [35]. Moreover, an intelligent agent can be embodied in a physical system, such as a robot or autonomous vehicle. A robot, thus, can be depicted as a special type of agent, equipped with hardware sensors and effectors allowing manipulation in the real world [9, 28].

In terms of navigation and motion, an action corresponds to a movement. One of the simplest agent concepts is the Braitenberg vehicle [10], whose motor speed is directly controlled by the magnitude of the sensor values. Depending on the connection of sensors and motors, it can show behaviors such as fear and aggression [45].

2.1.2 Steering Behaviors

Inspired by the behaviors presented by Braitenberg [10], Reynolds et al. [40] introduced a set of simple motion behaviors (so-called steering behaviors) for autonomous agents in computer games. Assuming a simple vehicle model with a mass, a position, an orientation, and a velocity, each movement can be described by a velocity vector with its polar components: orientation (direction) and magnitude (speed). The velocity can be altered by applying so-called steering forces.
Each motion of an agent is divided into three level: Based on the current environmental state or an event, the agent sets a goal it desires to achieve with its behavior (action selection level). Next, the agents subdivides its goal into a sequence of simpler subgoals (steering level), each described by a steering behavior, such as seek, flee, arrival, obstacle avoidance, etc. Every steering behavior provides a steering force, expressing the direction and magnitude that this behavior wishes the vehicle to move in order to fulfill its defined subgoal. Depending on the locomotion mode, the agent implements the defined motion direction and speed (locomotion level).
Due to the vector representation, several steering behaviors can be combined to create complex motion patterns. This is done by summing the vectors, optionally scaling them by adding weighting factors (see Figure 2.1). To counteract the risk of forces canceling each other out, Reynolds et al. [40] suggest establishing a prioritization order of behaviors (e.g. first avoid obstacles, then

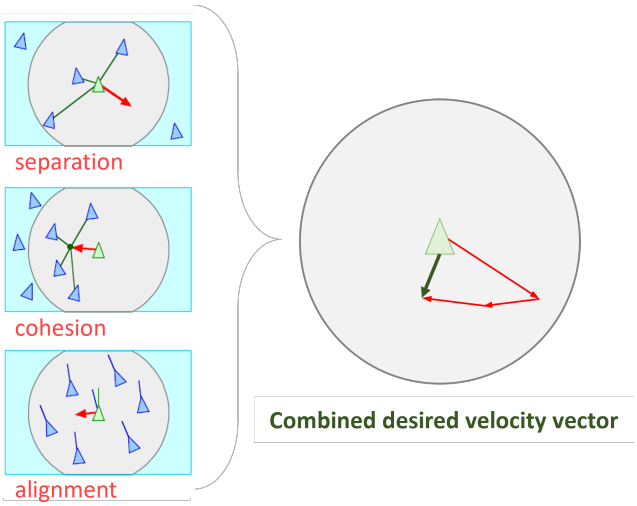


Figure 2.1: Example of combining several steering behaviors proposed by Reynolds et al. [40]. The summation of the behaviors' vectors returns the combined desired velocity vector, which is then applied to the agent with its current velocity and kinematic constraints.

follow the swarm), optionally enhanced by a probability for the use of the first or second priority behavior.

2.1.3 Context Steering

Due to the shortcomings, such as deadlocks, when using steering behaviors [40], Fray [19] introduces a concept considering the reasons of a steering behavior for preferring a certain movement direction – to enable a *context*-aware aggregation process. Instead of a behavior returning its one desired velocity vector, for *several* sensor directions $\vec{v}_s$, it assesses how much it wishes the agent to 1) move there (interest) and 2) keep away from (danger). These scalar interest and danger values are each stored in so-called context maps, an array in which each field corresponds to a direction $\vec{v}_s$ of the associated sensor s_i. At each iteration step, every behavior provides two context maps: an interest map and a danger map (see Figure 2.2).
The context value of a sensor direction $z(\vec{v}_s)$ depends on the distance $\|\vec{v}_o\|$ to

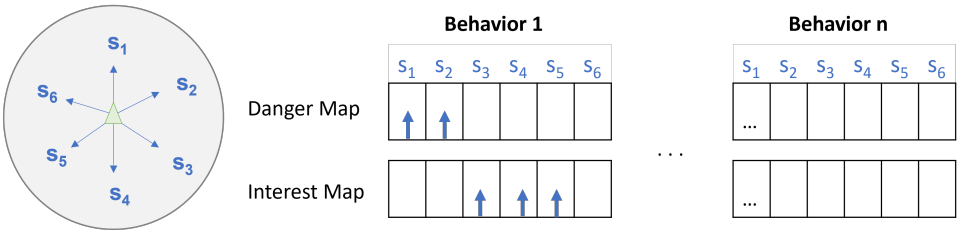


Figure 2.2: The structure of a context map. With the sensors s_1 to s_6, the agent senses its surroundings containing interesting and dangerous objects. For each direction of the sensors, every behavior rates the associated danger and interest.

the perceived object o, and on the angle ω between $\vec{v}_s$ and $\vec{v}_o$ (the direction to this object):

$$z(\vec{v}_s) = f(\omega) \cdot g(\|\vec{v}_o\|) \quad \in [0,1],$$
$$\text{with } \omega = cos^{-1} \frac{\langle \vec{v}_s, \vec{v}_o \rangle}{\|\vec{v}_s\| \|\vec{v}_o\|} \tag{2.1}$$

The sensors have a perception angle of $[\phi_{min}, \phi_{max}]$ and a perception range of $[\xi_{min}, \xi_{max}]$. Thus, the angle z_s is only calculated, if o is located in this area. The function f maps ω from $[\phi_{min}, \phi_{max}]$ to [0,1]. Similarly, g maps the distances $\|\vec{v}_o\|$ from $[\xi_{min}, \xi_{max}]$ to $[0,1]$. The mapping type of the functions f and g are again task-specific, e.g. (inverse) linear, squared or square-root [13].

As a first step of the aggregation process, the gathered information from the context maps are combined. Therefore, context maps with the same view (interest, or danger) are combined by comparing the context values of each direction across all interest or danger maps, and taking the maximum in each case. As a result, only one danger and one interest map remain. This is due to fact, that a closer object is more relevant at the current situation. In the example shown in Figure 2.3, the agent flees from walls and seeks a target. Since the flee behavior is only sensitive to dangerous and the seek behavior to interesting objects, the combined context map contains the danger map from the flee behavior and the seek behavior's interest map.
The last step of the aggregation process, i.e. selecting a movement direction and a speed, is task-specific. Fray [19] proposes the usage of masks, where the directions with the lowest danger value are selected and all other options with higher danger values are neglected. From this selection, the slot with the highest interest value is chosen. This returns the highest rated safe direction

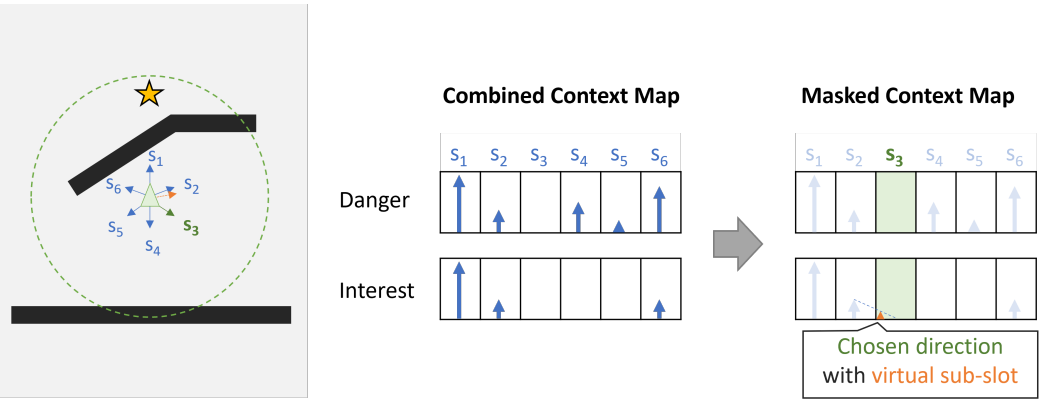


Figure 2.3: Creation of a context map in an exemplary situation, with an agent (green triangle) placed between two walls (black) and a target (yellow star). The agent has six steering directions (arrows) and a perception range (green circle).

available. The speed is proportional to the degree of interest. In the example introduced above (see Figure 2.3), the slot with the lowest danger value is s_3. As it is the only remaining direction after masking, it is also the chosen one even though its interest value is zero.
To allow the agent to have a continuous movement space while maintaining high algorithm performance (small number of context map slots), Fray [19] proposes virtual sub-slots. These are created by interpolating the values in the chosen slot and in the adjacent slots (see Figure 2.3).

2.1.4 Multi-Objective Context Steering

To provide the basic principles of multi-objective (MO) optimization and decision-making, they are introduced below, followed by an outline of the MO context steering approach.

Basics of Multi-Objective Optimization and Decision-Making

Most real-world problems have conflicting criteria, e.g. a customer wants to buy a new e-bike which should be cheap, lightweight and have a high battery capacity. The optimal solution would be best in all criteria at the same time. However, the criteria are opposed, as in the example, an e-bike with a large battery is heavy, and a lightweight bicycle is expensive. The options available

for selection (so-called individuals) fulfill only one criterion best and the others significantly worse, or they fulfill all criteria moderately well.
The degree to which a possible solution $\vec{x}$ fulfills a criterion i (an objective) can be described as an objective function $f_i(\vec{x})$. The aim of a multi-objective problem (MOP) is to find the individual $\vec{x}^*$ in the search space S which optimizes (minimizes) all conflicting objective functions simultaneously [36]:

$$\underset{\vec{x} \in S}{\textbf{argmin}}\ \vec{f}(\vec{x}) \tag{2.2}$$

The search space S is referred to as the decision space. As explained in the example above, there is no *single* optimal solution $\vec{x}^*$. Therefore, a method called Pareto-dominance is used to identify all solutions that are better regardless of user preferences. A solution x dominates another solution y (in a minimization MOP), if [36]:

$$\mathbf{x} \prec \mathbf{y} \ \Leftrightarrow\ f_i(x) \leq f_i(y)\ \wedge\ \exists j : f_j(x) < f_j(y), \qquad \forall i \in \{1, \ldots, m\} \tag{2.3}$$

The solutions that are not dominated by any other individual from the set of solutions, are called non-dominated set. In the objective space $O = \{\vec{f}(\vec{x}) \in \mathbb{R}^m \mid \vec{x} \in S\}$, the non-dominated set is called Pareto front [36].

Subsequently, from the set of non-dominated individuals, one has to be selected. Since all Pareto-optimal solutions are indifferent to each other, the user has to specify preferences [36]. This is done by using multi-objective decision-making methods, either a priori, a posteriori or interactive ones. The most common a priori method is to determine and optimize (minimize) the weighted sum of the objectives [36]:

$$f'(\vec{x}) = \sum_{i=1}^{m} w_i f_i(x),\ \text{with}\ \sum_{i=1}^{m} w_i = 1 \tag{2.4}$$

The disadvantage of this method is that weight vectors $\vec{w}$ have to be defined in advance (a priori) and solutions in concave parts of the Pareto front cannot be found [36]. In contrast, the ε-constraint method can also find solutions in the concave parts. For a two-dimensional MOP – as used in this thesis – one objective function $f_i(\vec{x})$ is optimized (minimized), while an upper bound ε is defined for the other function [13]:

$$f'(\vec{x}) = f_i(\vec{x}),\ \text{with}\ f_j(\vec{x}) \leq \varepsilon_j,\ i \neq j \tag{2.5}$$

The hybrid method combines the weighted sum and constraint method. It delimits the two-dimensional objective space according to the upper bound(s) ε_j but optimizes the weighted sum of all objective functions [13]:

$$f'(\vec{x}) = \sum_{i=1}^{m} w_i f_i(x), \text{ with } \sum_{i=1}^{m} w_i = 1, \ f_j(\vec{x}) \leq \varepsilon_j \tag{2.6}$$

When using a posteriori methods, such as evolutionary algorithms and Particle Swarm Optimization (PSO), many alternative solutions are generated, and the user can make a decision thereafter. As a composition of a priori and a posteriori method, interactive methods are guided during the search process by the user via partial preferences to gain a biased Pareto front, from which a solution is selected as per the user's exact preferences [36].

Applying Multi-Objective Optimization in Context Steering

As an extension of context steering [19] (see section 2.1.3), instead of masking, Dockhorn et al. [13] establish a multi-objective decision-making process. Because interesting and dangerous items can be located in the same direction, leading to conflicting objectives, the decision on a direction of movement becomes a MOP (see section 2.1.4).
Creating the context maps and combining multiple danger maps or multiple interest maps into one context map is identical to the original context steering concept (see section 2.1.3). However, interest and danger are seen as objective functions of the sensor directions $f_i = -z_i(\vec{v_s})$ and $f_d = z_d(\vec{v_s})$, which are to be minimized (interest is naturally maximized, thus, the negated interest is minimized):

$$\underset{\vec{v_s} \in S}{\textbf{argmin}} \ -z_i(\vec{v_s}), z_d(\vec{v_s}) \tag{2.7}$$

As a first step of MOO, all individuals $\vec{v_s} \in S$ are plotted in the objective space to determine the non-dominated set (see section 2.1.4). In the example introduced in section 2.1.4, the directions of the sensors s_4, s_5, and s_6 are dominated (see Figure 2.4); only s_1, s_2, and s_3 remain for the decision-making. Depending on the method and its parameters (danger weight w_d, interest weight w_i, danger constraint ε), different directions are selected. Throughout the whole navigation process, this affects the behavior of the agent.

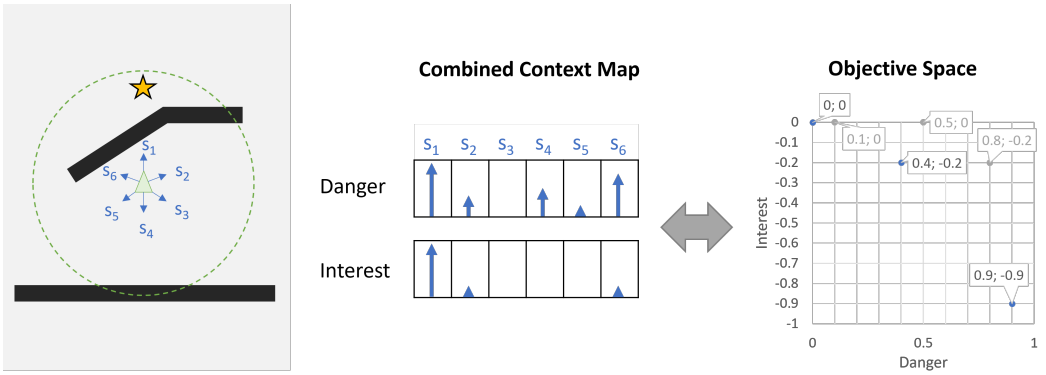


Figure 2.4: Multi-objective context mapping with a flee and a seek behavior. The agent flees from walls (black) and seeks a target (yellow star) within its perception range (green dashed circle). The values of the combined context map are plotted in the objective space to determine the Pareto-optimal individuals (blue) as per equation 2.3 and to neglect all dominated individuals (gray).

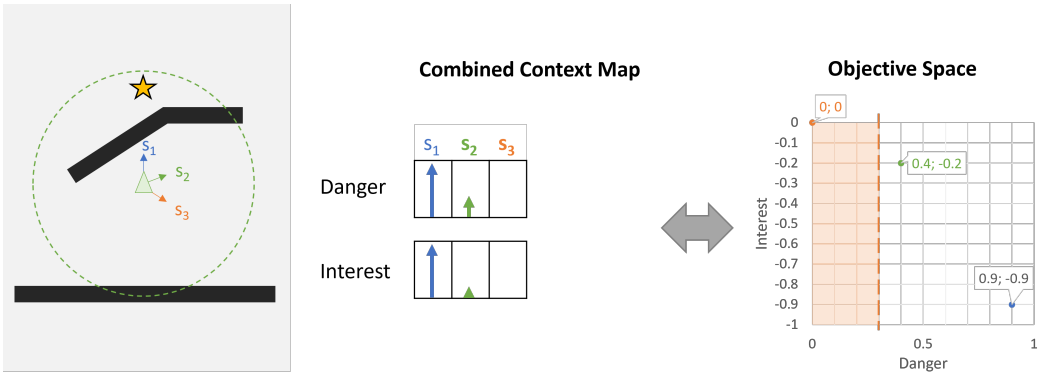


Figure 2.5: The agent chooses a different direction depending on the decision-making method. Applying the method using weighted sum as per equation 2.4 with $w_d = 0.6$, s_2 is the selected direction (green). With a constraint method as per equation 2.5 and $\varepsilon = 0.3$, s_3 is chosen (orange).

As proposed by Fray [19], Dockhorn et al. [13] additionally apply and recommend the usage of Gaussian blurring as well as history blending before MOO, and context interpolation (virtual sub-slots) after MOO. These so-called post-processing methods improve the behavior because they make the agent jitter less.

2.2 Navigation of the Driving Swarm

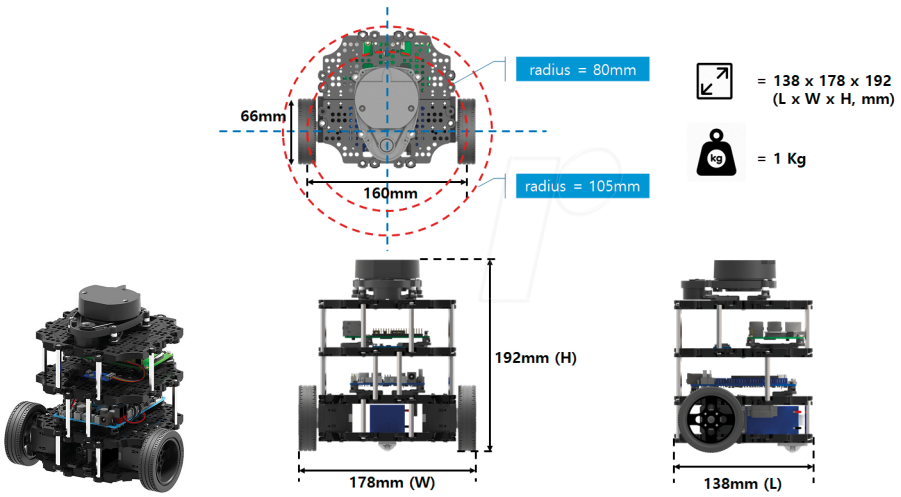
The Driving Swarm [32] is a collection of mobile differential-drive robots, namely Turtlebot3 Burger platforms [41]. Hence, in section 2.2.1 the technical features of the robot platform are presented. As it has a differential drive, subsequently the respective kinematic model is stated (see section 2.2.2). Lastly, section 2.2.3 elaborates the general requirements to navigate with a mobile robot and provides an outline of the Dynamic Window Approach.

2.2.1 Turtlebot3 Burger Platform

Intended for the application fields of teaching, research and prototyping, the Turtlebot3 Burger is programmable since it is based on the Robot Operating System (ROS). Its hard- and firmware (OpenCR board) are open source as well. With a height of about $20\,cm$ (see Figure 2.6) and a weight of $1\,kg$, it is significantly smaller than its predecessors. Due to its 360 degrees laser-based distance sensor (LDS-01) and two Dynamixel servo motors (XL430-W250-T) the robot can autonomously navigate using Simultaneous localization and mapping (see 2.2.3) [53].

2.2.2 Kinematic Model of a Differential-Drive Robot

A differential drive has two driven wheels on one axis, and one or two passive supporting wheels, rotating freely. As the active wheels are driven separately with v_r (right) and v_l (left), the resulting external movement of the robot describes either a straight line (forward or backward), a circle (whole or a segment of it, i.e. an *arc*), or a point (turn on the spot, see Figure 2.7). This applies only under the restriction of not decoupling rotation and translation. Therefore, the external movement is represented by the linear (translational) velocity $v = \frac{v_r+v_l}{2}$ and the angular (rotational) velocity $w = \frac{v_r-v_l}{2d}$. While



(a) (b)

Figure 2.6: The Turtlebot3 Burger robot platform and its dimensions. [1, 6]

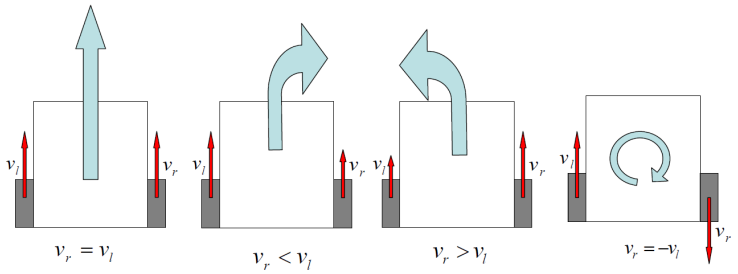
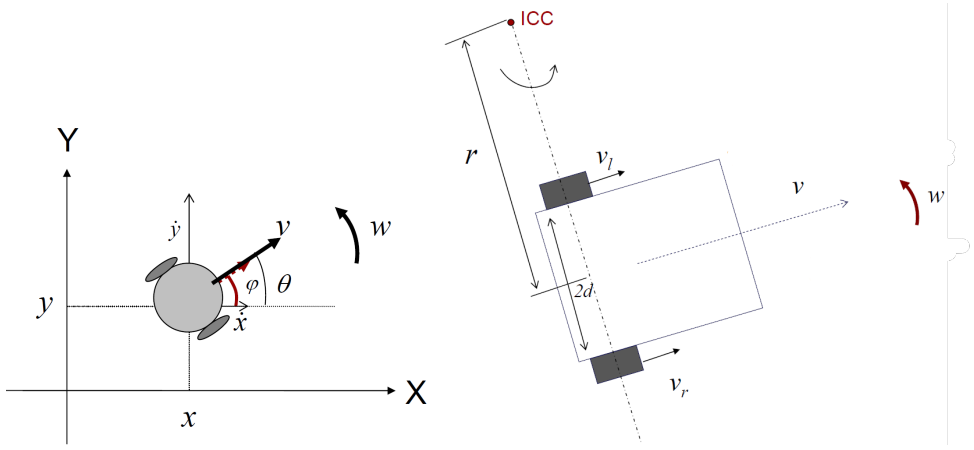


Figure 2.7: The internal (red) and external (blue) movements of a differential-drive robot. [37]



(a) (b)

Figure 2.8: A differential-drive robot and its quantities in the position and velocity space. [37]

moving, the robot rotates about a point located along the axis of the driven wheels – the so-called Instantaneous Center of Curvature (ICC, see Figure 2.8b) [15].
To further describe the kinematics, let there be a world coordinate frame W in which a robot moves. This robot has a local coordinate frame L at the midpoint between the two driven wheels (see intersection point of the two blue dashed lines in Figure 2.6). The robot's state q_t at time t is therefore described by its pose (see Figure 2.8a), which is a vector of the x and y coordinates of L in W (the position $\widehat{p}_t$ of the robot in W), and the angle between the respective x-axes of L and W (the orientation $\widehat{\theta}_t$ of the robot in W):

$$\widehat{q}_t = \begin{pmatrix} \widehat{x}_t \\ \widehat{y}_t \\ \widehat{\theta}_t \end{pmatrix} \tag{2.8}$$

Trajectories are a time sequence of the robot's current and future state $a = \{q_{t+t_j} | \ \forall j \in \{1, 2, \ldots, \Psi\}\}$. Since, in this work, the robot's egocentric coordinate frame L is used, the current state of the robot is always at the origin aligned with the x-axis $q_t = (x_t, y_t, \theta_t)^T = (0, 0, 0)^T$. Thus, the trajectories start at the origin of L and the ICC is placed at $ICC = (0, r)^T$ with

$r = \frac{v}{\omega}\ \forall \omega \neq 0$. Moving with a velocity of v and ω, the robot's state in L after τ is (forward kinematics) [15]:

$$q_{t+\tau} = \begin{pmatrix} x_{t+\tau} \\ y_{t+\tau} \\ \theta_{t+\tau} \end{pmatrix} = \begin{cases} \begin{pmatrix} \cos(\omega\tau) & -\sin(\omega\tau) & 0 \\ \sin(\omega\tau) & \cos(\omega\tau) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ -r \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ r \\ \omega\tau \end{pmatrix} & , \text{if } \omega \neq 0 \\ \begin{pmatrix} v\tau \\ 0 \\ 0 \end{pmatrix} & , \text{else} \end{cases} \tag{2.9}$$

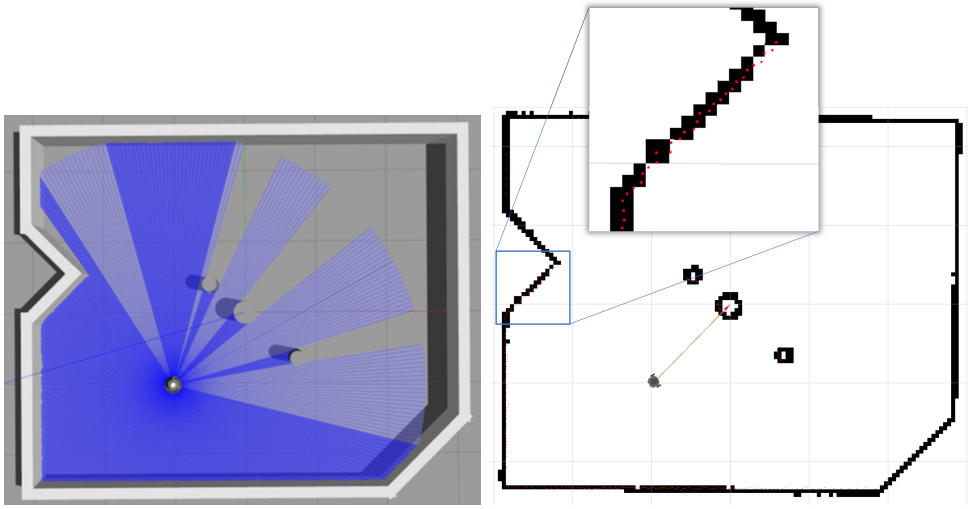
With an angular velocity being negative or positive $\omega \neq 0$, the robot describes an arc (or turns on the spot). At each future state, the orientation is in the domain $\theta \in [-\pi, \pi]$. Without rotation $\omega = 0$, the robot moves along the x-axis, maintaining the orientation $\theta_t = \theta_{t+\tau}$.

2.2.3 Mobile Robot Navigation

Navigation is the ability (of a robot) to move from the current location to a defined target location in a known or unknown environment while *avoiding collisions* with obstacles [42, 53]. Therefore, a navigation algorithm requires at least the robot's current position and the goal location in the same coordinate system, optionally also a map (map-based vs. mapless-based algorithms) [42]. Path planning, on the other hand, seeks to find the *optimal* route in a map for the robot to navigate to the target (neglecting the temporal component) [42]. Depending on the algorithm, a navigation system involves sensors perceiving the environment, the creation and interpretation of a map, (self-) localization and path planning in this map [42, 53].
These elements of navigation are described below, followed by an explanation of the functionality of the Dynamic Window Approach (DWA), a local planner taking the kinematics and dynamics of a differential-drive robot into account.

Localization and Mapping

Simultaneous localization and mapping (SLAM) is a proven method to generate a map [48]. The robot moves through the environment, either autonomously or controlled by a human, and creates a 2D or 3D representation



(a) (b)

Figure 2.9: An environment with a robot and the corresponding map. (a) The robot perceives its surroundings with a laser scanner. (b) By means of the scan points, a map is recorded using SLAM and within this map the robot can localize itself using AMCL.

of the surroundings (walls and additional objects) perceived by the available sensors (see Figure 2.9). At the same time, the robot estimates its own current pose (position and orientation) in the map recorded until then. The basic pose estimation is done with dead reckoning, measuring the rotation of the wheels to calculate the displacement from a known starting position. To enhance accuracy, additionally inertial information from the inertial measurement units (IMU) can be incorporated [53]. Probabilistic localization methods such as the Adaptive Monte Carlo Localization (AMCL) use distance sensors or cameras with particle filters to further increase the accuracy, by matching the recorded map with the currently perceived environment [53].

Path Planning and Obstacle Avoidance

While global path planning considers the entire map, local approaches apply only for the local area around the robot [53] (the definitions of navigation and local path planning are not clearly delineated). Because path planners, such as A* or simulated annealing, produce an optimized path but accept potential uncertainties in the environment, it cannot react to unforeseen obstacles [46].

For dynamic and unknown environments, local (reactive) navigation methods, like the Bug algorithm or the Dynamic Window Approach (DWA, see next section), are better suited, avoiding obstacles in the vicinity. However, they can be inefficient (not optimal), particularly in complex environments [38, 46, 54]. In robotics, global and local algorithms are often combined as hierarchical planners [49]. Steering, as defined in section 2.1, belongs to the reactive navigation algorithms, since it considers only the local environment and the agent has no or very little memory.

The Dynamic Window Approach

The Dynamic Window Approach (DWA) introduced by Fox et al. [17] is a local path planner for obstacle avoidance that considers the kinematics and dynamics of a differential-drive robot (see section 2.2.2). The aim is to find a feasible velocity command, controlling the robot's linear and angular velocity (v,ω), that brings the robot quickly to the target while avoiding collisions with obstacles in the vicinity. Hence, the search for (v,ω) is carried out directly in the two-dimensional velocity space V_s, containing all possible velocities as per the robot's hardware limitations (translational velocity limits v_{min} and v_{max}, rotational velocity limits ω_{min} and ω_{max}) [17, 46, 53]:

$$V_s = \{(v,\omega) | v_{min} \leq v \leq v_{max} \wedge \omega_{min} \leq \omega \leq \omega_{max}\} \tag{2.10}$$

The search space is restricted to those (admissible) velocities that allow the robot to stop before colliding with an obstacle (as opposed to the collision area, see Figure 2.10) [17, 46]:

$$V_a = \{(v,\omega) | v \leq \sqrt{2dist(v,\omega)a_v} \wedge \omega \leq \sqrt{2dist(v,\omega)a_\omega}\} \tag{2.11}$$

Next, based on the robot's dynamics (maximal translational acceleration a_v, maximal rotational acceleration a_ω, and the current dynamic state (especially the actual velocity (v_c, ω_c)), the so-called dynamic window V_d of feasible velocities (v', ω') in the next iteration step $t + \tau$ is determined [17, 21, 46]:

$$V_d = \{(v,\omega) | v \in [v_c - a_v\tau, v_c + a_v\tau] \wedge \omega' \in [\omega_c - a_\omega\tau, \omega_c + a_\omega\tau]\} \tag{2.12}$$

The remaining search space V_r is the intersection of the restricted areas described above $V_r = V_s \cap V_d \cap V_a$. Within V_r, a number of velocities is selected

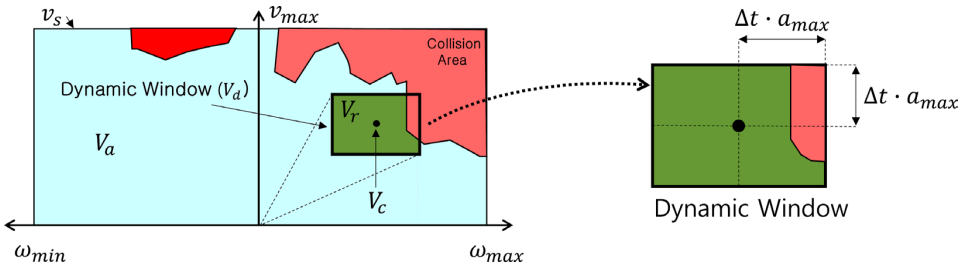


Figure 2.10: Restricted areas in the search space of the DWA. The velocity space V_s, with the admissible velocities V_a, the current velocity V_c, the dynamic window V_d, and the resulting search space V_r [53].

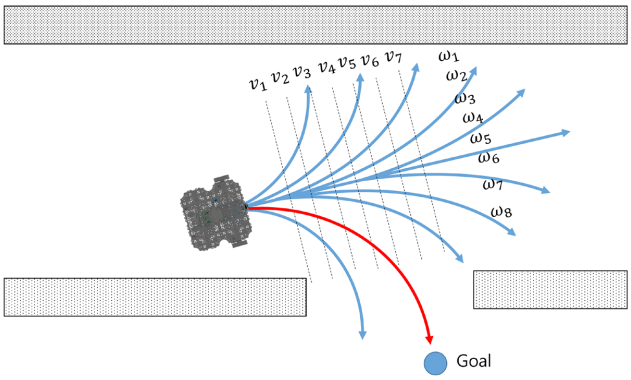


Figure 2.11: Arc-shaped trajectories uniquely defined by the linear and angular velocity (v,ω) [53]

and their associated trajectories are simulated in the time span τ (see Figure 2.11) [17]. The trajectories are evaluated based on an objective function $G(v, \omega)$ (represented as a weighted sum with the weights α, β, γ, and a scale σ) originally involving the alignment of the robot with the target direction $heading(v, \omega)$, the distance to the closest obstacle $dist(v, \omega)$, and the speed $velocity(v, \omega)$, i.e. the translational velocity component [17]:

$$G(v, \omega) = \sigma \left(\alpha \cdot heading(v, \omega) + \beta \cdot dist(v, \omega) + \gamma \cdot velocity(v, \omega)\right) \quad (2.13)$$

Depending on the implementation, more criteria can be incorporated, e.g. when using DWA in a hierarchical planner, the distance from the global path computed prior to the execution of DWA may be relevant [21, 53]. In the original concept [17], $G(v, \omega)$ is maximized. However, there are implementations with adapted criteria, *minimizing* the objective function since it is depicted as a cost function [53]. The trajectory with the highest objective value/the lowest total costs is chosen and its associated velocity pair (v, ω) is given as a command to execute [17, 21].

2.3 Distinction of this Thesis from the Related Work

The algorithm introduced in this thesis contains elements of both the MOO context steering [13] (see section 2.1.4) and the DWA [17] (see section 2.2.3). The kinematics and dynamics of the differential-drive Turtlebot3 Burger robot are respected by using a concept similar to the dynamic window. Both operate in a restricted space of translational and rotational velocity (v, ω), searching for a velocity command, executable in the next iteration step, bringing the robot quickly to the target while avoiding obstacles. However, the search space (decision space) is restricted only to the velocities that can be realized according to the dynamics; unsafe trajectories are not excluded here, as this is part of the evaluation. Like in the DWA, for the evaluation, the trajectories of the selected velocities (so-called samples) are simulated for a defined time span. However, instead of one objective function, the concepts of context steering using multi-objective optimization and decision-making [13] are applied. First, the interest and danger maps are created, followed by the determination of the non-dominated set and the application of a multi-objective decision-making

method. In doing so, the slots of the context maps do not correspond to sensor directions but to the velocity/trajectory samples.
The navigation algorithm presented here operates in the local environment. Except for the goal position, only locally perceived data is available. Unlike hierarchical planners, it does not receive any pre-optimized global plan.

3 Methodology

After the basic principles being defined and this thesis being integrated in existing related works, the necessary foundations are laid for the introduction of the methodology. In section 3.1 the requirements and resulting design decisions are explained. This is followed by a detailed explanation of the algorithm (see section 3.2) and a description of the tools used for the implementation (see section 3.3).

3.1 Requirements

Various requirements are placed on the algorithm that affect its design. On the one hand, it should induce a desired behavior of the robot. On the other hand, the application of the context steering approach to a robot with differential drive requires certain adaptations. This will be elaborated in the following sections.

3.1.1 Desired Behavior

The navigation algorithm should reliably maneuver the Turtlebot3 Burger robot from its start position to the target position without hitting obstacles in the static, planar environment (the world itself is flat but 3-dimensional, however, it is perceived as 2-dimensional at the level of the laser scanner). While navigating, the robot preferably moves in a fast and smooth manner independent of the environment, i.e. the algorithm should be flexible to adapt to different surroundings.
In terms of steering behaviors (see section 2.1.2), this involves the seek and the flee behavior. The former aims to move the robot toward the goal, while the latter aims to keep distance from obstacles [19, 40].

3.1.2 Characteristics of the Robot

Due to the robot's available sensors and actuators, the decision domain and the definition of the context values (the objective functions) are modified compared to the original context steering approach [19]. Even though the experiments are conducted in simulation, there are certain real-time conditions to be fulfilled when using a robot:

Real-Time Capability

In order for the robot to enable smooth driving behavior, decision-making must be performed simultaneously with movement. Hence, the algorithm should meet (soft) real-time conditions. A common frequency for the actuation command is $10\,Hz$. One run of the algorithm should happen within this time span. Since this cannot be guaranteed, the impact of short outages or delays is mitigated by evaluating a predicted robot state that is further in the future than the state at the time of the next decision-making (iteration). In other words, the trajectories are simulated for a longer time span τ_d and τ_i than one iteration step Ω_m takes: $\tau_d, \tau_i > \Omega_m$ (see section 3.3.1).

Actuation and Decision Domain

In context steering (see section 2.1.3), a (holonomic) agent decides on the direction of motion, because it can directly execute this movement. From the magnitude of interest for this direction, the speed is derived. The kinematics of a differential-drive robot, however, do not allow instantaneous movements in any direction within the given time step. It moves forward and rotates at the same time (see section 2.2.2). Hence, each possible movement is an arc-shaped trajectory a (except for the one pointing straight forward). A trajectory results from moving with a certain translational and rotational velocity $\vec{v} = (v, \omega)^T$ for a defined period of time τ. This means: the path, the arrival time, and the final direction the robot is facing at the destination point (the goal) are different between holonomic agents and differential-drive robots. Applying the same decision domain is unsuitable (see Figure 3.1).
This is why, at each time step t, the algorithm decides on the velocity $\vec{v}$, that the differential-drive robot executes until the next time step $t + 1$. This way, after the decision is made, no further mapping of holonomic to non-holonomic velocity is required.

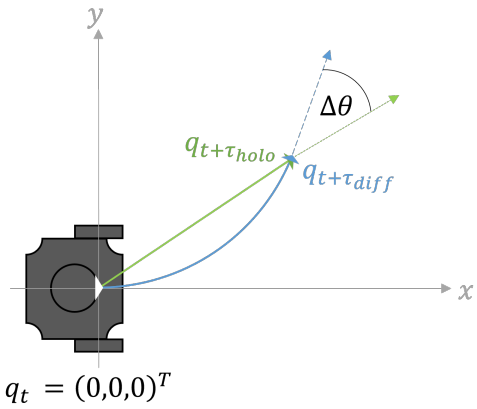
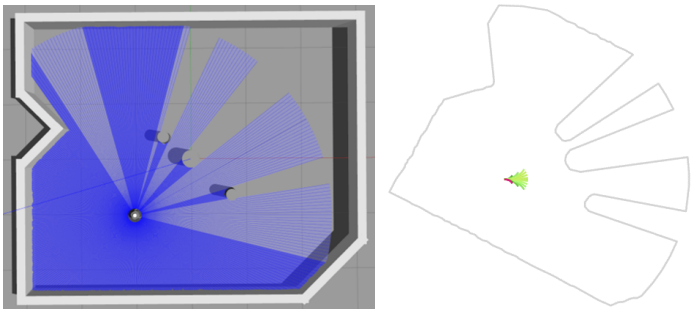


Figure 3.1: Difference between trajectories of holonomic (green) and differential-drive (blue) robots.

Perception and Context Mapping

For an agent with a flee and a seek behavior, the calculation of the context values considers the distance and direction to surrounding objects in relation to the agent's position and a set of potential movement directions (see section 2.1.3). The information that are retrieved from that are: 'When the agent heads in this direction, to what extent does it lead to a goal and/or to an obstacle? How close are they currently?' Therefore, the agent must know its own position as well as the positions of the closest objects (in the perception range).
This approach does not entirely apply to differential-drive robots. Since the trajectories are circular, at each time step t the robot faces another direction (see Figure 3.1). The question arises, which of them should be chosen to represent an entire trajectory. Furthermore, the exact positions of the environmental obstacles are not known to the robot. It cannot perceive the obstacles around it as individual entities. Instead, it perceives its environment through a laser scanner, which creates a point cloud indicating the scanned surface of surrounding objects (see section 2.2.3).
In order to use this information to create a context map for differential-drive robots, a change of perspective is required: Instead of choosing a direction that could represent the arc-shaped trajectory, the trajectory itself is understood as a geometric figure. From the scanner's point cloud, another geometric figure (the polygon P_t) is created (see Figure 3.2). This enables the calculation of the distance to the closest obstacle and to the target. The question now being



(a) (b)

Figure 3.2: The creation of the polygon P_t, modeling the local environment perceived by the robot's laser scanner.

asked is: 'How close does the robot get to obstacles moving along this path?' Yet, this does not consider the direction, which can be of major importance in certain situations, e.g. the robot is near an obstacle located behind or beside it, but all trajectories lead away from the obstacle. Therefore, as a mitigation to this problem, only the rear part of each trajectory is considered for the calculation of the distance to obstacles.

3.2 Structure of the Algorithm

The algorithm is structured into three main steps (see Figure 3.3). At each time step t, the search space V_t (referred to as decision space in the scope of MOO) is created by sampling velocities $\vec{v} = (v, \omega)^T$, that are feasible according to the robot's dynamics and its current velocity $\vec{v}_c$ (see section 3.2.1).
Next, the context values (referred to as objective values in the scope of MOO) are computed with the objective functions $z_d(\vec{v}) \in [0, 1]$ for mapping danger and $z_i(\vec{v}) \in [0, 1]$ mapping interest, $\forall \vec{v} \in V_t$ (see section 3.2.2). Since the seek behavior is only sensitive to interest objects (the goal) and the flee behavior only to danger objects (walls and other surrounding objects), the context maps are each implicitly combined by taking the seek behavior's interest map and the flee behavior's danger map. Moreover, the flee behavior always considers the closest obstacle and there is only one goal for the seek behavior to desire, which is why from the beginning there is one context value for the danger/interest map to store in each slot (as opposed to selecting the maximum to combine multiple danger/interest maps, see section 2.1.3).

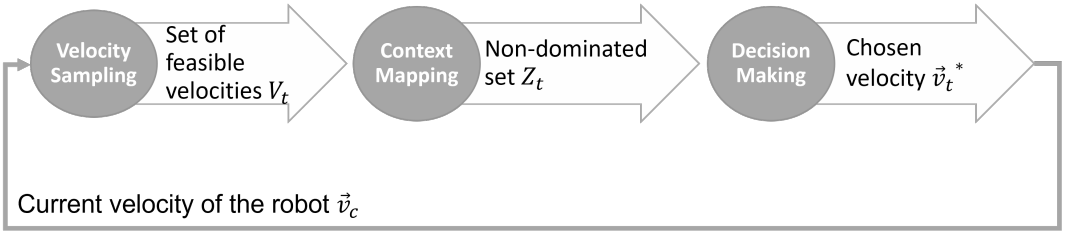


Figure 3.3: General structure of the algorithm. The circles represent the processes, the arrows the respective out- and input.

Based on these values, the non-dominated set Z_t is determined, whereof one individual $\vec{v}^*$ is selected depending on the specified decision-maker (see section 3.2.3). In the implementation presented in this thesis (see section 3.3.1), the output $\vec{v}^*$ is the new input $\vec{v}_c$ for the next iteration; the initial velocity is $\vec{v}_0 = (0, 0)$.

The following sections explain the different steps in detail.

3.2.1 Velocity Sampling

During the velocity sampling, the feasible velocity space V_t is populated with velocity samples $\vec{v}$ (referred to as individuals in the scope of MOO):

$$\vec{v} = \begin{pmatrix} v_k \\ \omega_l \end{pmatrix}, k \in \{1, 2, \ldots, n_v\} \quad l \in \{1, 2, \ldots, n_\omega\} \tag{3.1}$$

Each $\vec{v}$ executed for a time span τ results in a trajectory a, being shaped like an arc or a line pointing straight ahead (see section 2.2.2). Put in illustrative terms, the rotational component determines how much the arc is bent. The higher the rotational velocity, the stronger the arc is bent (see Figure 3.5). From the robot's point of view, positive values result in an arc bent to the left and negative values in an arc bent right (or a left/right turn on the spot). An angular velocity of zero results in a straight line. To allow a decision between directions, at least three ω-values are reasonable (positive, negative and zero). The translational velocity determines the length of the trajectory. The faster the robot is going to move, the longer the trajectory. Positive values cause forward motion, negative values cause backward motion, and at zero the robot stays in place. It is reasonable to use more than one v-value for sampling

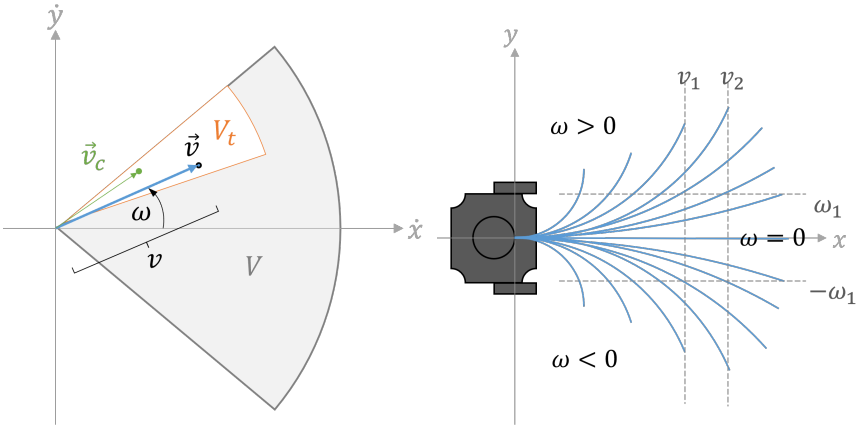


Figure 3.4: Sampling within the feasible velocity space V_t (orange).

in order to decide not only about the orientation but also about the forward speed.
For the velocity sampling, first the feasible space of velocities V_t is determined, containing all velocities reachable until the next time step:

$$V_t = \{\vec{v} \mid \omega \in [\omega_{t_{min}}, \omega_{t_{max}}], v \in [v_{t_{min}}, v_{t_{max}}]\} \tag{3.2}$$

In the velocity space $(\dot{x}, \dot{y})$, the rotational velocity ω and translational velocity v are the polar coordinates. Without constraints, this causes V_t to be shaped like a circle with its center in the origin. However, due to the robot's hardware, there are limits to its speed and acceleration, which causes V_t to be truncated (see Figure 3.4a). The minimum and maximum rotational velocity the robot allows $\omega \in [\omega_{min}, \omega_{max}]$ result in two circle segments, axis-symmetrical to the y-axis (the robot's orientation is along the x-axis). The minimum and maximum translational velocity as per the robot's hardware $v \in [v_{min}, v_{max}]$ causes the circle segment to be truncated in the x-axis. If both limits are greater than zero, this results in a ring segment. A speed limit bounds V_t statically, an acceleration limit on the other hand dynamically. Therefore, the bounds of the velocity space $[\omega_{t_{min}}, \omega_{t_{max}}]$ and $[v_{t_{min}}, v_{t_{max}}]$ have to be determined every time the actual velocity of the robot $\vec{v}_c = (v_c, \omega_c)^T$ changes (the following equations show only the rotational velocity bounds for the time span τ_s, but the translational velocity bounds are calculated in the same way):

$$\begin{bmatrix} \omega'_{min} \\ \omega'_{max} \end{bmatrix} = \begin{bmatrix} \omega_c \\ \omega_c \end{bmatrix} + \begin{bmatrix} -\ddot{\theta}_{max} \cdot \tau_s \\ \ddot{\theta}_{max} \cdot \tau_s \end{bmatrix} \tag{3.3}$$

$$\omega_{t_{min}} = \begin{cases} \omega_{min} & \text{, if } \omega'_{min} \leq \omega_{min} \\ \omega'_{min} & \text{, else} \end{cases} \tag{3.4}$$

$$\omega_{t_{max}} = \begin{cases} \omega_{max} & \text{, if } \omega'_{max} \geq \omega_{max} \\ \omega'_{max} & \text{, else} \end{cases} \tag{3.5}$$

After V_t has been determined, the samples are selected from it. Consequently, the number of samples always remains the same. This is to ensure multiple options are always available so that subsequently an informed decision can be made for one of these options. If the velocity space were to be trimmed after sampling, it may happen that only one sample remains without ever having explicitly decided on it.
The samples are evenly distributed over the feasible intervals for the defined number of samples n_ω and n_v (see Figure 3.4a). To ensure, straight movements are also included, $\omega = 0$ is added. Lastly, the selected velocities are combined, resulting in an overall number of samples $|V_t| = n_\omega(n_v + 1)$.

3.2.2 Context Mapping

The process of context mapping includes creating the interest and danger map, plotting their values in the objective space $O_t = \{\vec{z}(\vec{v}) \in \mathbb{R}^2 \mid \vec{v} \in V_t\}$, and subsequently determining its Pareto front $Z_t = \{\vec{v}_i \in V_t | \nexists \vec{v}_j : \vec{v}_j \neq \vec{v}_i, \vec{v}_j \succ \vec{v}_i\}$ (see Algorithm 1). The following explains in detail the steps involved in creating the danger and interest map:

Modelling the Trajectories

To calculate the context values, the trajectories resulting from $\vec{v} = (v, \omega)^T$ are evaluated. For this purpose, they need to be modeled.
As introduced in section 2.2.2, a trajectory is a time sequence of the robot's state $q_t = (x_t, y_t, \theta_t)^T$, where x_t and y_t refer to the position p_t, and θ_t denotes the orientation of the robot at time t in the robot's local coordinate frame L. Since L is an egocentric coordinate frame, placed in the midpoint between the wheels and used as the base frame for all spatial calculations, the current state is always $q_t = (0, 0, 0)^T$.

Algorithm 1: ContextMapping

Data: V_t, P_t, g_t, τ_d, τ_i, Ψ, μ, κ, ξ_{min}, ξ_{max}, λ

Result: Z_t

forall *velocity samples* $\vec{v} \in V_t$ **do**
  $q_{t+\tau_d} \leftarrow FutureState(\vec{v}, \tau_d)$;
  $a \leftarrow TrajectoryCoordinates(q_{t+\tau_d}, \Psi)$;
  if $a \cap P_t \neq \emptyset$ **then**
    $t_j \leftarrow FirstIntersection(a, P_t)$; // $t_j \in [0, \Psi]$
    // normalize, and map distance to similarity metric
    $z_d(\vec{v}) \leftarrow 1 - t_j \div \Psi$; // $z_d(\vec{v}) \in [0, 1]$
    // normalize to defined upper bound
    $z_d(\vec{v}) \leftarrow \kappa + (1 - \kappa) \cdot z_d(\vec{v})$; // $z_d(\vec{v}) \in [\kappa, 1]$
  else
    $d_a \leftarrow ShortestDistance(a, P_t, \mu)$; // $d_a \in [\xi_{min}, \xi_{max}]$
    if $\mathcal{M}_d$ *is abs* **then**
      $\xi \leftarrow \xi_{max}$
    else
      $\xi \leftarrow \max_{d_a \forall a}(d_a)$
    end
    // normalize, and map distance to similarity metric
    $z_d(\vec{v}) \leftarrow 1 - \big(\exp(\lambda \cdot d_a) - 1 \div \exp(\lambda \cdot \xi) - 1\big)$; // $z_d(\vec{v}) \in [0, 1]$
    // normalize to artificial upper bound
    $z_d(\vec{v}) \leftarrow \kappa \cdot z_d(\vec{v})$; // $z_d(\vec{v}) \in [0, \kappa]$
  end
  $q_{t+\tau_i} \leftarrow FutureState(\vec{v}, \tau_i)$
  // cosine similarity of future robot and goal direction
  $c \leftarrow \big(1 + \langle \vec{a}_{t+\tau_i}, \vec{g}_t \rangle) \div \|\vec{a}_{t+\tau_i}\| \|\vec{g}_t\|\big) \div 2$; // $c \in [0, 1]$
  $\widetilde{d}_x \leftarrow \|p_{t+\tau_i} - g_t\|$;
  // normalize to minimum and maximum of set D_t and map distance to similarity metric
  $D_t = \{\widetilde{d}_x | \forall \vec{v}\}$
  $d_x \leftarrow 1 - \big(\widetilde{d}_x - \max(D_t)\big) \div \big(\max(D_t) - \min(D_t)\big)$; // $d_x \in [0, 1]$
  $z_i(\vec{v}) \leftarrow c \cdot d_x$; // $z_i \in [0, 1]$
end
// Pareto front in objective space O_t
$O_t = \{\vec{z}(\vec{v}) \in \mathbb{R}^2 \mid \vec{v} \in V_t\}$
$Z_t \leftarrow ParetoDominant(O_t)$; // $Z_t = \{\vec{v}_i \in V_t | \ \nexists \vec{v}_j : \vec{v}_j \neq \vec{v}_i, \vec{v}_j \succ \vec{v}_i\}$
return Z_t;

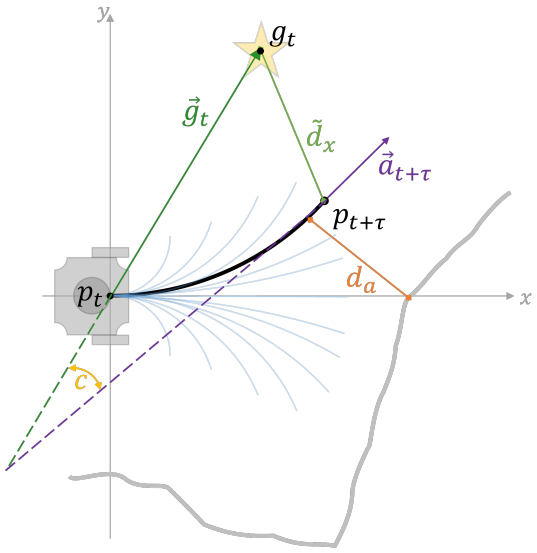
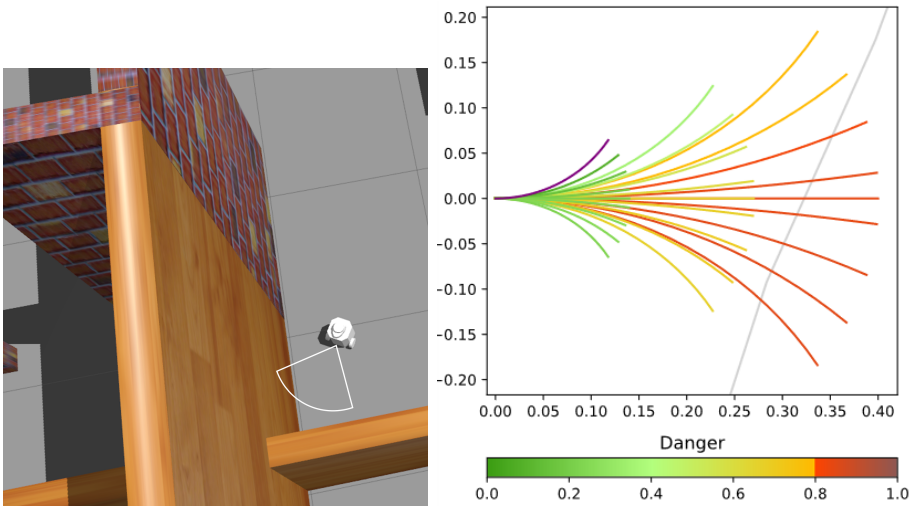


Figure 3.5: Visualization of the quantities used for context mapping.

To model a trajectory for a time span τ, the robot's future state q_{t+t_j} is determined for every sub-time step $t_j = j \cdot \frac{\tau}{\Psi}\ \forall j \in \{1, 2, \ldots, \Psi\}$ (also referred to as trajectory points), as per the forward kinematics of a mobile differential-drive robot (see section 2.2.2). The larger the number of trajectory points Ψ relative to the time span τ, the more accurately the real trajectory is modeled. Every trajectory starts at the origin. With $\omega \neq 0$, the robot rotates with a radius $r = \frac{v}{\omega}$ about a virtual point, called ICC. With $\omega = 0$, the robot moves straight ahead, thus only the x-coordinate changes $x_{t+\tau} = v\tau$. The length of each trajectory depends on v and τ, and ω determines the curvature (see Figure 3.5).

Creating the Danger Map

The danger value is represented either as the *time* at which the velocity sample results in a collision, or as the closest *distance* to the obstacles when moving along the trajectory. For these calculations, a geometric representation of the environment is required: From the laser scan data, a point cloud is created in the robot's coordinate frame (see section 2.2.3) which serves as the basis for generating a polygon P_t, representing the perceived environment (see Figure 3.2). At each time step, P_t represents the sensed environment within the perception range $[\xi_{min}, \xi_{max}]$. In some situations, e.g. in environments with several smaller obstacles, multiple polygons may be created. Due to the ego-



(a) Robot position in House1 with indicated trajectory range.

(b) Trajectories colored according to their danger values.

centric view, the polygon closest to the origin is chosen because it is the one most relevant to the robot.
The interior of the polygon depicts the area around the robot without collisions, whereas every intersection of the robot with the exterior is considered a collision (or at least a dangerous situation to be avoided). Since the Turtlebot3 Burger platform has an almost round footprint, it can be simplified into a circle with a radius of r_{robot} (see section 2.2.1). This in turn allows considering the robot as a point and artificially erode P_t by a value $\rho \geq r_{robot}$ (the idea is based on the obstacle inflation layer on top of the static map [31]).

As stated above, there are velocity samples eventually resulting in a collision, i.e. trajectories intersecting with the polygon P_t, and ones not colliding within the given time horizon. In order to be able to evaluate the danger in both cases, each has its own danger map. To combine them later again into a single danger map, they are mapped to a specific range of values (see Figure 3.6b). Since trajectories having a future collision within τ_d are more dangerous, they are mapped onto a range of $[\kappa, 1]$. The remaining non-intersecting trajectories are mapped onto a range of $[0, \kappa]$ with $\kappa \leq 1$. In the following, both danger maps are explained in detail.

Intersecting trajectories: Each intersection of a trajectory with the polygon $a \cap P_t \neq \emptyset$ denotes a predicted future collision at time $t + t_j$. The smaller t_j, the sooner the collision will follow, and the greater the danger. For each

sample, where there is at least one intersection, $t_j \leq \tau_d, j \in \{1, 2, \ldots, \Psi\}$ is determined and normalized with Ψ to the interval $[0, 1]$. Only the point where the trajectory *first* intersects with the polygon is relevant. The established danger value is also referred to as *time-based* danger.

$$z_d(\vec{v}) = \begin{cases} \kappa \cdot (1 - \frac{t_j}{\Psi}) & \text{, if } a \cap P_t \neq \emptyset \\ \kappa + (1 - \kappa) \cdot \left(1 - \frac{\exp(\lambda \cdot d_a) - 1}{\exp(\lambda \cdot \xi) - 1}\right) & \text{, else} \end{cases} \tag{3.6}$$

Non-intersecting trajectories: For the remaining trajectories not intersecting with the polygon (within τ_d), the danger value is established *distance-based*. A velocity sample is more dangerous the closer the robot gets to the obstacles while traveling the corresponding trajectory. Thus, the shortest distance of each arc to the exterior of the polygon is determined. In situations where obstacles are closer to the side or back of the robot than to its front, the shortest distance occurs at the origin. Since every trajectory starts at the origin, at least some of them would receive the same danger value and distort the objective space O_t. In order to maintain an informed decision in such cases, the first μ trajectory points (robot states) are discarded.
Since shorter distance corresponds to higher danger, the distance is mapped onto danger values in the range of $[0, 1]$, using a negative exponential function. The factor $\lambda \neq 0$ defines the slope of the function graph. Negative values produce a convex and positive values a concave shape of the graph.
To obtain $z_d(\vec{v})$, all values are either normalized to the overall maximum value ξ_{max} (absolute normalization, *abs* $\mathcal{M}_d$), or the maximum value of the current set of distances $d_{a_{max}} = \max_{d_a \forall a}(d_a)$ (relative normalization, *rel* $\mathcal{M}_d$). The comparison of both variants of normalization is a subject of the experiments (see chapter 4).

Creating the Interest Map

Whereas the perception range for obstacles is limited, the detection range for the seek object is unlimited. In other words, there is only one goal whose position $\widehat{g}$ is fixed in the map. At each time step, it is transformed into the robot's coordinateframe and made available to the robot as g_t. Like the related work (see section 2.1.4), the context value for the seek behavior takes into account the orientation and distance of the robot with respect to the goal: However, since the robot seeks to move towards the goal, for both the

orientation and the distance the future state of the robot at the time $t + \tau_i$ is crucial for the evaluation of this velocity sample in terms of the interest objective:

$$z_i(\vec{v}) = c \cdot d_x, \quad \text{with } c = \frac{1}{2}\left(\frac{\langle \vec{a}_{t+\tau_i}, \vec{g}_t \rangle}{\|\vec{a}_{t+\tau_i}\| \|\vec{g}_t\|} + 1\right) \quad \text{and } \vec{a}_{t+\tau_i} = \begin{pmatrix} \cos\theta_{t+\tau_i} \\ \sin\theta_{t+\tau_i} \end{pmatrix} \tag{3.7}$$

The cosine similarity c describes the alignment of two vectors. It is adapted so that it returns a value between 0, meaning the vectors point in exactly opposite directions, and 1, meaning the vector have the exact same orientation. Here, the future direction at the end of each trajectory $\vec{a}_{t+\tau_i}$ is compared with the direction $\vec{g}_t$ currently pointing to the goal. The higher the similarity, the greater the interest. The interest value is the product of the cosine similarity and the normalized euclidean distance d_x (between the position $p_{t+\tau_i}$ of the robot at the end of each trajectory and the goal position g_t):

$$d_x = 1 - \frac{\widetilde{d}_x - \max(D_t)}{\max(D_t) - \min(D_t)}, \quad \text{with } \widetilde{d}_x = \|p_{t+\tau_i} - g_t\| \tag{3.8}$$

The consideration of both c and d_x serves primarily to diversify the context values: All trajectories with the same ω value point in the exact same direction, meaning they have identical c value. If the calculation of the interest value were based solely on c, this would leave the objective space O_t unstructured. The additional consideration of the distance $\widetilde{d}_x$, however, provides information about how fast a velocity sample takes the robot to the goal, with shorter $\widetilde{d}_x$ corresponding to higher interest. The value is normalized by the minimum and maximum value of the current set D_t to the range of $[0, 1]$. Additionally, the distance metric is transformed into a similarity metric, so that higher d_x correspond to higher interest $z_i(\vec{v})$.

Determining the Pareto-Front

With both context maps being created, the aim is to find the individual that maximizes interest and minimizes danger – a multi-objective optimization problem. Here, a minimization problem is formulated, thus $z_i(\vec{v})$ has to be negated:

$$\underset{\vec{v} \in V_t}{\textbf{argmin}} \; -z_i(\vec{v}), z_d(\vec{v}) \tag{3.9}$$

Since the two objectives are conflicting, there is no obvious best choice. However, using the Pareto dominance (see chapter 2.1.4), the objective space O_t can be restricted: options that are worse than or at most as good as others in both objectives are excluded. Thus, only the non-dominated – good – solutions remain (see Figure 3.7). This so-called Pareto font depicts the set of individuals between which the decision maker subsequently has to choose.

3.2.3 Multi-Objective Decision-Making

In this last step of the algorithm, from the set of non-dominated individuals, one has to be selected. To find a Pareto-optimal solution corresponding to a suitable motion trajectory for the next time step, five multi-objective decision-making methods are available:

- the *weighting method* calculating the weighted sum of each individual, choosing the one with the minimum value:

$$\underset{\vec{v} \in V_t}{\textbf{argmin}} \; -w_i z_i(\vec{v}) + w_d z_d(\vec{v}) \quad \text{with } w_i = 1 - w_d, \tag{3.10}$$

- the ε_d-*constraint method* applying an upper bound ε_d on danger, choosing the individual with the minimum negative interest:

$$\underset{z_d(\vec{v}) < \varepsilon_d,\ \vec{v} \in V_t}{\textbf{argmin}} \; -z_i(\vec{v}), \tag{3.11}$$

- the *random method* applying an upper bound ε_d on danger, using a uniform probability distribution to randomly select an individual based on the interest value:

$$\vec{v} \in Z_t \text{ (randomly chosen)} \quad \text{with } z_d(\vec{v}) < \varepsilon_d,\ \vec{v} \in V_t, \tag{3.12}$$

- the *hybrid method* applying an upper bound ε_d on danger and calculating the weighted sum of the remaining individual, choosing the one with the minimum value:

$$\underset{z_d(\vec{v}) < \varepsilon_d,\ \vec{v} \in V_t}{\textbf{argmin}} \; -w_i z_i(\vec{v}) + w_d z_d(\vec{v}) \quad \text{with } w_i = 1 - w_d, \tag{3.13}$$

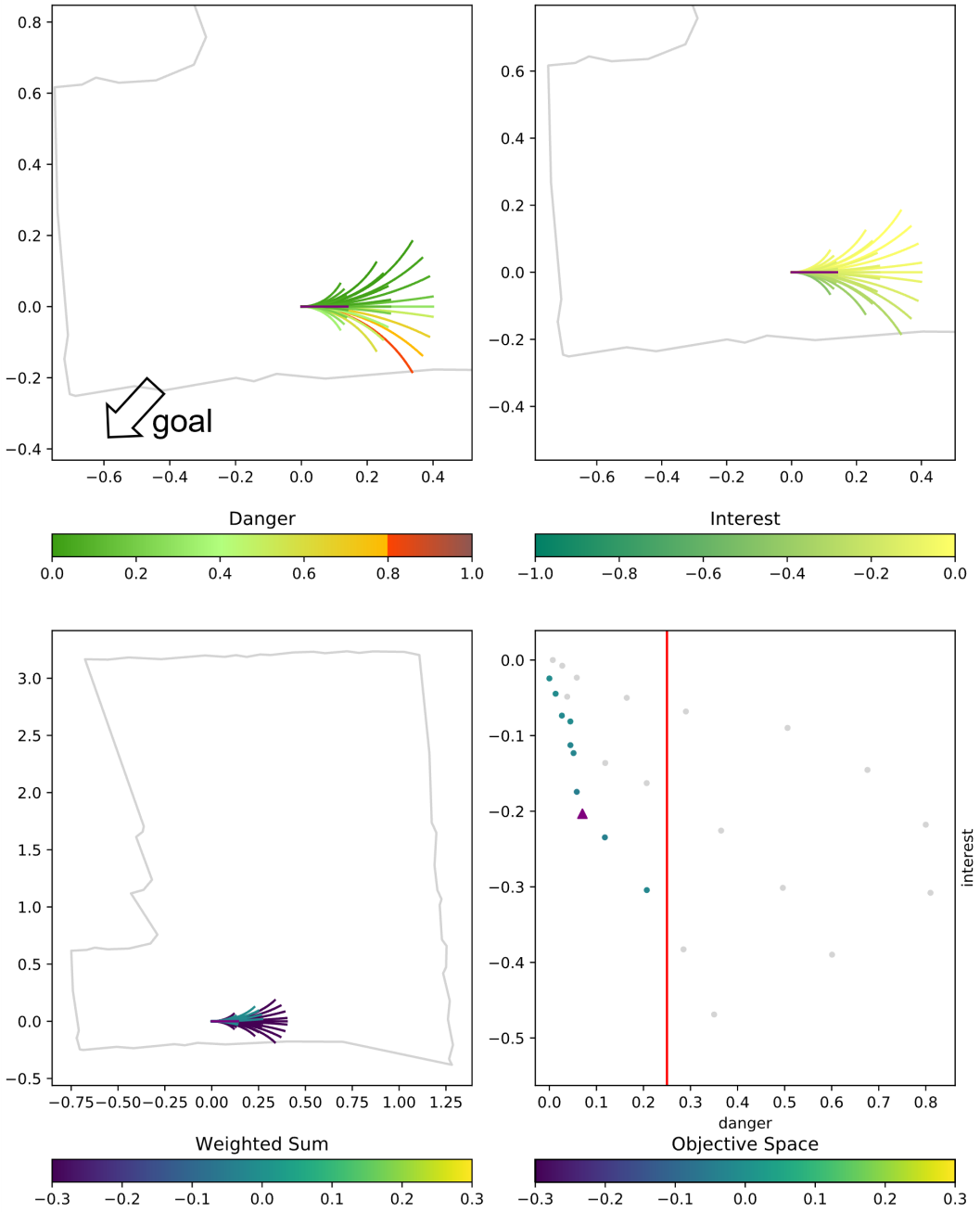


Figure 3.7: Relationship between decision space and objective space. The robot has left its starting position in the House2 scenario. The goal direction is on the lower left side. The objective space (lower right) plots individuals with their danger (upper left) and interest (upper right) values; colored ones are non-dominated, the purple triangle models the chosen one.

- the ε_i-*constraint method* applying an upper bound ε_i on interest, choosing the individual with the minimum danger:

$$\underset{z_i(\vec{v})<\varepsilon_i,\ \vec{v}\in V_t}{\textbf{argmin}} \quad z_d(\vec{v}). \tag{3.14}$$

If the objective space is unstructured, there can be multiple individuals with the same minimal value for interest, danger, or weighted sum. In this case, since all of them seem to be equally good in terms of the objectives, one individual is randomly chosen from the set (with a uniform probability distribution). In the other extreme case, none of the individuals is valid. In such situations, the robot stops $\vec{v}^* = (0, 0)^T$.

3.3 Implementation

The algorithm was implemented with the Robot Operating System 2 [12] using Python 3 language [50]. This framework, additional libraries included, and the software as well as hardware used, are explained in the following subsections.

3.3.1 Robot Operating System 2

As a set of open source software libraries, algorithms and tools, the Robot Operating System (ROS) [2] provides the basis for developing tailored robot applications [25–27]. It serves as a communication system, a framework of developer tools (e.g. visualization, data recording and replay), and an ecosystem with drivers and libraries [29]. Here, the ROS2 distribution Foxy Fitzroy is used, released in June 2020 [2].
The core of its distributed real-time architecture is a network of ROS2 [2] elements. In this so-called ROS graph, nodes communicate via topics, services, actions, or parameters with each other (see Figure 3.8). Topics are based on a publisher-subscriber model, that keeps the subscriber supplied with continuous updates in the form of messages. In contrast, services implement a request-and-response model, providing a response when explicitly requested by a client. A common service is the parameter server, enabling parameters to be stored and retrieved. Unlike ROS(1), in ROS2 [2] each node administers its own parameters. Actions unify both topics and services by providing steady feedback and allowing cancellation during execution [2].

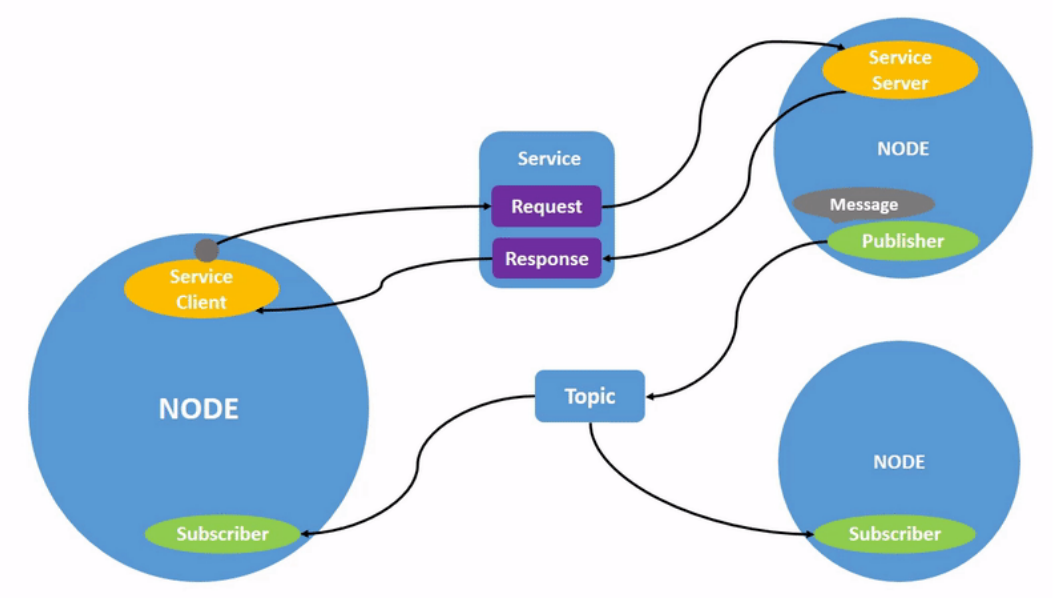


Figure 3.8: ROS graph depicting a minimal example of three nodes [2]. One node publishes to a topic, the others subscribe to it. The same node provides the server for a service which the node on the left uses as a client through an interface of service request and response.

To enable the data exchange, a DDS middleware (Fast RTPS, Cyclone DDS, or RTI Connext) is used with a ROS middleware interface (RMW) on top of it [12]. Through the ROS2 Client Library (rcl) [2] and a wrapper either for Python [50] (rclPy) or C++ [47] language (rclCPP), the user can access the ROS2 features [29].

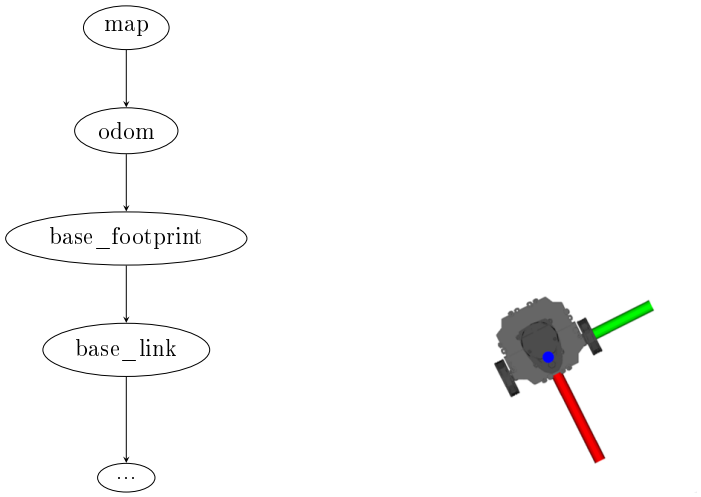
Nodes and Topics

Since each node is supposed to have a single, module purpose [2], the process and logical structure explained in the previous section 3.2 are also reflected in the ROS graph (see Figure 3.3). There are three nodes: Based on the current velocity of the robot, the 'velocity_sampling_node' publishes the set of feasible velocities at $\Omega_s = 10\,Hz$ so that the 'context_mapping_node' can evaluate them. The 'context_mapping_node' updates at a rate of $\Omega_m = 5\,Hz$ and eventually publishes a sub-set of these samples, namely the non-dominated ones including their context values. Every time the 'context_mapping_node' publishes a new message to the topic 'non_dominated_individuals', the 'de-

cision_making_node' is triggered to select one of the velocities based on their objective values with the defined decision-making method. The 'decision_making_node' publishes the velocity at $\Omega_d = 10\,Hz$ to the topic 'cmd_vel', regardless of whether a new decision has been made. The reason for this is the perspective that the software will be used on real robots. They have a control unit stopping the robot if the speed commands are not published regularly.
The update rates were chosen so that the nodes work as independently as possible with the latest information. However, due to the subdivision of the software into nodes, latency can occur so that the information on which a decision is made no longer corresponds to reality. The current velocity $\vec{v}_c$ is assumed to correspond to the velocity command chosen and published in the previous iteration $\vec{v}^*_{t-1}$. Hence, $\vec{v}_c$ is read from the latest message on the topic 'cmd_vel'. This may not always correspond to reality, especially after a collision, however in such cases the experiment run is stopped.

Localization and Coordinate Frames

Within the ecosystem of ROS2 [2], Navigation2 (Nav2) [31] provides a complete navigation system, configurable due to the modular structure. The required inputs to Navigation2 are the transformations of the different coordinate frames (tf2), a map, and sensor data. Its output are valid velocity commands for the motors of the robot [30].
The algorithm presented here substitutes the major part of Nav2 [31], such as global or local planning, as it serves as a navigation program itself. It utilizes only cartographer [52] from the SLAM-Toolbox (to generate a static map from the environment), the Map Server (to load and administer the map) and AMCL [31] (to localize the robot on the map). The Turtlebot3 Burger [41] broadcasts a tree of coordinate frames (see Figure 3.9a). For the implementation of the algorithm, as a base frame, the 'base_footprint' frame is used (see section 4.2). In this frame, the robot's own position is at the origin at all times, which corresponds to the midpoint between the wheels (see Figure 3.9b). This simplifies the calculation of distances to surrounding objects, but makes it necessary to transform the goal position from the 'map' frame.



(a) Published coordinate frames and transformations without the frames of the robots components.

(b) Location of the coordinate frame 'base_footprint' (red: x-axis, green: y-axis, blue: z-axis).

Figure 3.9: Coordinate frames.

Simulation

Another set of software libraries as part of the ROS2 [2] ecosystem is Gazebo. It includes the Gazebo multi-robot simulator, an open source robotics simulator incorporating high fidelity physics, rendering, and sensor models [24]. The implementation of this algorithm depends on Gazebo Classic version 11.2.0 [3].

Data Visualization

As a graphical interface to ROS2 [2], RViz2 [20] visualizes data from active ROS topics, utilizing different plugins [5]. Moreover, it allows interactions by means of tools like '2D Pose Estimate' or '2D Nav Goal'. For this implementation RViz2 was used to show the static map, the robot model, relevant coordinate frames, as well as self-defined markers, paths and arrow, for example indicating the goal position or the trajectories. The usage was primarily for examining transformations, positions, and orientations in the map to debug the application.

Like RViz2 [20], RQt [4] is a graphical user interface for ROS2 [2] tools available as CLT. For example, RQt can be used to view and analyze message content and metadata. Sending messages or making service calls are also possible.

3.3.2 Python Packages

Apart from the standard modules that are delivered with a Python [50] release, certain functionalities in the algorithm require additional Python packages. These include in particular NumPy, SciPy, and Shapely. NumPy enables numerical calculations, which can improve performance immensely, especially by vectorizing operations [11]. It is often used in combination with SciPy, which provides algorithms for scientific computing [51]. To compute distances and intersections of the trajectories and the polygon, the Shapely package is used. It allows manipulating and analyzing planar geometric objects [44].

3.3.3 Hardware

All tests and experiments are performed with a notebook PC model MSI GS65 Stealth 8RE-079, running with an Intel processor Core i7-8750H (2,2 GHz).

4 Experiments

In this chapter, an overview on the conducted experiments is given. Therefore, in section 4.1 the purpose and the chosen design of the experiments are stated. This is followed by a presentation of the metrics used for the evaluation in chapter 5. Section 4.3 specifies the parameters and their effect in the system, and reasons the chosen values. The last section of this chapter explains, how the experiments were launched, and how the data was acquired and processed to provide the results for subsequent evaluation.

4.1 Experiment Design

In the following, the structure of the experiments, conducted to evaluate the introduced algorithm, is presented. Subsequently, the navigation task, the chosen environments, and the scenario setups are explained (see section 4.1.2). In addition, for each scenario, a path is described that has proven successful in preliminary experiments to take the robot to the goal (see section 4.1.3). This facilitates the comparison of actual and expected routes when evaluating the results (see chapter 5).

4.1.1 Structure

To evaluate the implemented algorithm, presented in the previous chapter, two main experiments were conducted:

1. a base experiment with a *rigid* distance-based danger map, and
2. a comparison experiment with an *adaptive* distance-based danger map

With a rigid map, the distance-based danger values are normalized to the maximum perception range η_{max} (absolute normalization), whereas in the adaptive approach they are normalized to the current maximum distance $d_{a_{max}}$ (relative

normalization, see section 3.2.2). In both experiments, a simulated Turtlebot3 Burger platform performs a navigation task in different environments. Preliminary experiments have shown that the parameters are sensitive to the environment. To learn more about their effect, another set of ε-constraint and weight values is incorporated as an extension of each main experiment.

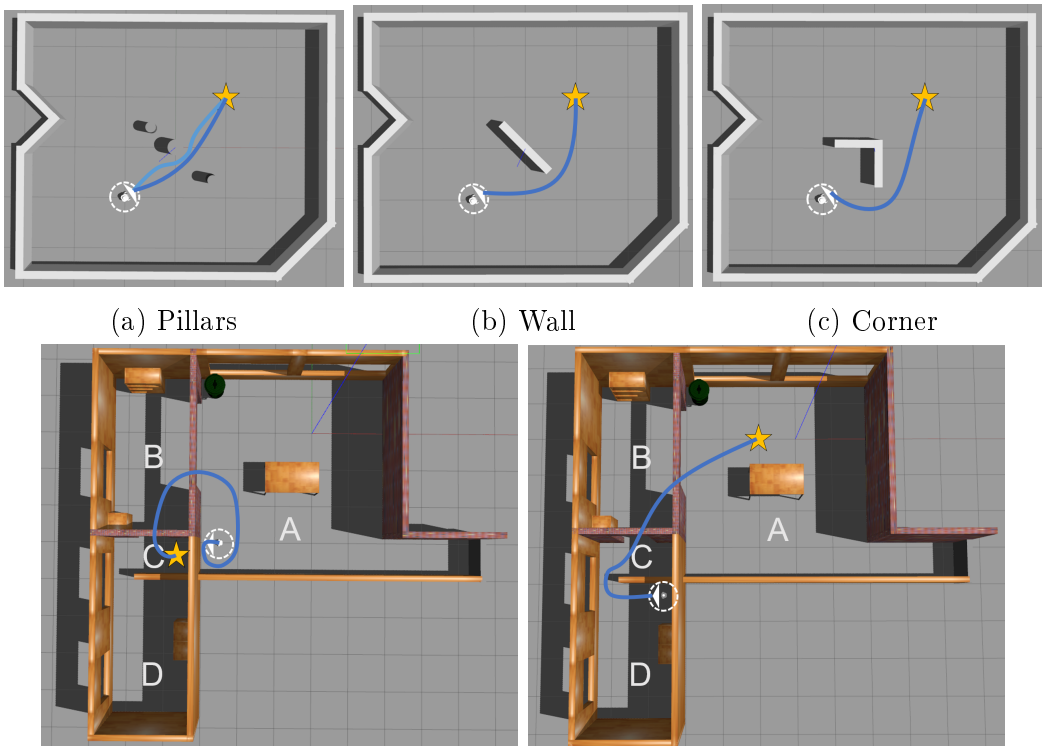
4.1.2 Scenarios

The central question of this work is, whether MO context steering applied to differential-drive robots might succeed in scenarios where steering ends up in a deadlock. Therefore, only environments in which the steering algorithm would fail are considered, i.e. the start and goal position are separated by an obstacle placed on the direct line connecting these two positions. In classical steering [39, 40], the force that pulls the robot toward the target and the force that pushes the robot back from the obstacle cancel each other out, which causes the robot to either stop or wiggle back and forth.

The task in the experiment is to move from the start position to the goal position without colliding with obstacles in the environment, which are located exactly between these two positions (the walls of the environment are naturally obstacles as well). To enable a comparison between the different decision makers in terms of their performance, five scenarios of increasing complexity are used (see Figure 4.1). Three of them appear similarly: a room with obstacles in its center. The obstacles are either three pillars, a straight wall or an L-shaped wall. The reason for the room's shape is that recognizable shapes and asymmetry facilitate localization. The other two scenarios share the same environment: an apartment that has several rooms, narrow passages and other obstacles. Two different configurations of starting and goal positions offer different levels of difficulty. Due to the system being non-deterministic, the experiments were conducted with every decision-maker in all scenarios for 11 runs each. A run is terminated when the robot reaches the goal area, collides with the environment, or after $t = 180\,s$ (maximum).

4.1.3 Expected Paths

In each scenario, certain paths have emerged where the robot successfully reaches the goal while keeping distance from obstacles without taking detours. These qualitative observations were made in preliminary experiments. The



(a) Pillars (b) Wall (c) Corner

(d) House1 (e) House2

Figure 4.1: Simulated environments used in the experiments. In each scenario, the white circle shows the robot at its starting position, the white star indicates the goal position (see Table 4.1).

Table 4.1: Scenarios with the initial state of the robot $\widehat{q_o}$ (position and orientation) and the goal position $\widehat{g}$ in the world coordinate frame.

Scenario	$\widehat{q_0}$	$\widehat{g}$
pillars, wall, corner	(-1.0, -1.0, 0.5)	(1.0, 1.0)
house1	(-2.5, -3.0, 3.14)	(-3.6, -3.3)
house2	(-3.6, -4.3, 3.14)	(-1.0, 0.0)

description of these paths serve for better comparison to the actual paths, described in chapter 5.
In the Pillars scenario (see Figure 4.1a), the robot is expected to describe a large arc curved to the right, from the start to the goal location. With a more risk-affine behavior, the path is straight at both ends, and bent only in the middle part. In general, the robot is expected to pass between the center and right outer pillar. In the Wall setting (see Figure 4.1b), the path should be shaped similarly to the large arc in the Pillars scenario, but with a stronger curvature to the left or right. To achieve this, the robot has to be a bit more risk averse. Otherwise, the pattern is the same as for the Pillars scenario: the robot travels straight toward the wall and then takes a wide turn around the wall. The same applies to the Corner scenario (see Figure 4.1c), where a strong left or right bend at the beginning would be ideal, and after passing the L-shaped obstacle, an almost straight line. However, as the results will show, the robot cannot escape the corner and moves directly towards the inner tip.
In the House1 scenario (see Figure 4.1d), for an ideal path, the robot would turn right at its starting position and travel parallel to the left wall of room A. However, the robot is expected to describe a spiral out of the corner, where the starting position is located. The remaining excepted path looks like a large inverted 'U' around the wall between rooms A, B and C. From the starting position, as the robot is attracted by the target behind the wall, the robot turns to its left towards the bottom wall of room A. To avoid turning towards the corner (which will lead to a collision or deadlock), all trajectories directing to the left and bottom wall have to be excluded. This leads to a spiral-shaped path. To achieve this behavior, the robot has to be very risk-averse, because otherwise it will be forced into the corner towards the target location.
The expected path in the House2 environment (see Figure 4.1e) describes a horizontal 'U' around the wall between room C and D. In order to prevent

the robot from moving into the right upper corner of C (which points to the target), the robot has to turn to its left facing the passage between room B and C. Therefore, the robot has to be more risk-averse, however, to cross the small passage next to the wall between room C and D, the robot must not be too risk-averse. Being located in the passage between room B and C (oriented towards room B), an almost straight path to the target is possible, with small curves avoiding the wall between room A and B.

4.2 Evaluation Metrics

The performance of each decision-maker is evaluated based on how well it meets the requirements of the desired behavior (see section 3.1.1). The **effectiveness** tells, whether the decision-making methods can steer the robot from its start position to the goal position without collisions. It is quantified by the success rate, i.e. number of runs where the goal was reached in relation to all runs. The goal is considered to be reached when the robot position is within the defined radius around the target (goal tolerance, see section 4.3. The decision-maker is considered to be effective if the robot reaches the target at least in 9 out of the total 11 runs. On top of that, the **flexibility** indicates in how many worlds the robot was effective.
In order to evaluate **safety**, the runs are differentiated according to whether they failed or were successful. For the former, it is relevant how many of the failed runs ended in a collision. For the latter, safety is measured by how many of the successful runs recorded a dangerous distance to an obstacle (meaning $d \leq 0.2m$ from the midpoint between the wheels to obstacles). To allow more fine-grained evaluation, in addition, the minimum distance over the entire run is considered.
It must be noted that the observed distance is the one measured by the laser scanner, which does not consider the robot dimensions. As the robot itself has a maximum extent of $0.1\,m$ (see section 4.3.1), at a distance considered to be dangerous there is approximately $0.01\,m$ left, depending on the orientation to the wall. Furthermore, a collision is a distance-based estimation: when reaching a distance $d \leq 0.1\,m$ (the robot radius), it is considered a collision.

The desired behavior is preferably fast, direct and smooth (see section 3.1.1). Thus, combined under the term **efficiency**, the successful runs are evaluated based on:

- the average traveled time (elapsed time until the goal was reached),
- the average traveled distance (cumulated euclidean distance between the robot positions until the goal was reached), and
- the shape of the paths as well as the chosen speed (translational velocity) during a run.

Analyzing all recorded paths – including the failed ones – can reveal additional information about the behavior, e.g. if the robot actually completed the task but just missed the target area.
Lastly, to make the algorithm transparent and comprehensible for verification, the decision space and its corresponding objective space are observed, highlighting all admissible individuals and the chosen one.

4.3 Parameter Settings

The parameters are chosen based on preliminary experiments, such that the decision-makers achieve satisfying results in as many scenarios as possible with the same parameter set (as opposed to choosing scenario-specific parameters). For the base experiment, no extreme values were used, but rather values that seem natural and allow the robot to be neither risk-affine nor risk-averse. In the following, first, the parameters that are kept unchanged throughout the experiments are introduced (see section 4.3.1). Then, the varied parameters are specified (see section 4.3.2).

4.3.1 Constants

For all nodes, the 'base_footprint' is considered the base coordinate frame to which all geometric calculations refer.

In this paragraph, the parameters of the velocity_sampling_node are explained: The time interval of the update rate of the context_mapping_node Ω_m (see section 3.3.1) predefines the temporal limits in which the velocity can be changed. Hence, the look-ahead time for sampling $\tau_s = \Omega_m = 0.2$.
The number of samples was chosen so that, on the one hand, different directions are available for selection (depending on the number of ω, see section 3.2.1). On the other hand, the algorithm should be able to choose between different speeds. For example, an expectable behavior would be slow

Table 4.2: Parameters that are constant throughout all experiments.

Parameter	**Value**	**Unit**
coordinate frame	'base_footprint'	-
τ_s	0.2	s
n_ω	8	-
n_v	3	-
v_{min}	0.07	m/s
v_{max}	0.2	m/s
$\ddot{v}_{max}$	2.5	m/s^2
ω_{min}	-1.0	rad/s
ω_{max}	1.0	rad/s
$\ddot{\omega}_{max}$	3.2	rad/s^2
ξ_{max}	3.5	m
ρ	0.1	m
Ψ	20	-
μ	5	-
κ	0.8	-
dangerous distance	0.2	m
goal tolerance	0.15	m
robot radius	0.1	m

motion in narrow corridors and fast motion in empty areas.
In this work, only positive translational velocities are considered, which means the robot cannot stop or move backwards. The values for minimum and maximum velocity and acceleration were chosen within the technical physical limits of the Turtlebot3 Burger platform[1] (see section 2.2.1), as is the perception radius ξ_{max} which corresponds to the technical laser range.

The polygon P_t is eroded by a value of $\rho = 0.1$, corresponding to the robot's radius. More precisely, since only forward movements are allowed, the value corresponds to the largest extent from the origin of the base frame (the midpoint between the wheels) to the outer corner of each wheel (see section 3.3.1). This is the minimum distance that the robot must keep in order to be able to escape from the situation by turning. With this value, the robot sometimes

[1] https://emanual.robotis.com/docs/en/platform/turtlebot3/features/

moves very close to obstacles. However, a safety buffer is not added, as this can lead to situations near walls that the robot cannot escape. The risk of dangerous distances is accepted and reduced by making the robot more risk-averse through adapting other parameters.

The value of μ defines how much (of the beginning) of the trajectory is ignored when calculating the shortest distance to the polygon. It ensures a structured objectives space, even when the closest obstacle is located behind or beside the robot. Without a truncation, all trajectories would have the same danger value because the current state of the robot (= the first point of all trajectories) is the one closest to the polygon. Ignoring the first μ sub-time steps of the trajectory diversifies the starting point and leads to different distances. However, it also needs to be ensured that the robot is not moving blindly by ignoring too much of the trajectory. Here, $\mu = 5$; with $\Psi = 20$ and $\tau_d = 2.5\,s$ this corresponds to cutting the first $0.625\,s$ of the trajectory. The remaining trajectory is kept to its end.

As described in section 3.2.2, the range of the danger value is based on

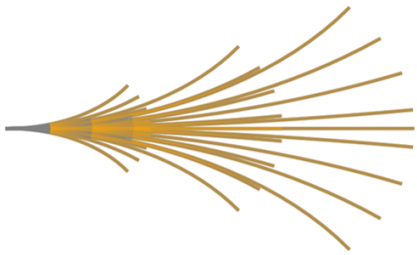


Figure 4.2: Truncating the trajectories for the distance-based danger mapping. The yellow part of the trajectory is kept for the calculation of the shortest distance to the polygon P_t.

whether there is an intersection with the polygon. The longest trajectory ends about $0.4\,m$ in front of the robot, which equals an intuitively estimated danger of 80%. Therefore, the upper bound for trajectories without intersection and at the same time lower bound for trajectories with intersection is $\kappa = 0.8$.

4.3.2 Variables

Since most parameters are strongly interdependent, when selecting the values for w_d, ε_d, and ε_i the following must be considered:

- the environment,
- the separation of value ranges of the danger map κ,
- the length of the trajectories determined by τ_d and τ_i, and
- the mapping function of the context values, including the function itself, its parameters and the normalization (absolute or relative).

Therefore, the values were determined in preliminary experiments (see Table 4.3). The time span τ determines the length of the trajectory. The larger its value, the longer the trajectory. This is why, it contributes to how risk-averse the robot behaves, especially in relation to κ. Preliminary experiments have revealed different values for τ_d and τ_i achieve better results. Additionally, for absolute normalization the values have to be higher (the trajectories longer) than using relative normalization.

Table 4.3: Overview of varied parameters with their values used in the experiment part I and II. The weight for the interest objective results from $w_i = 1 - w_d$.

		context mapping				decision-making		
		τ_d	τ_i	$\mathcal{M}_d$	λ	ε_d	ε_i	w_d
I. a	**Base Experiment**	2.5	1.5	*abs*	-2.0	0.4	-0.4	0.8
I. b	**Altered Parameters**	2.5	1.5	*abs*	-2.0	0.8	-0.02	0.92
II. a	**Comparison Experiment**	2.0	1.0	*rel*	3.0	0.25	-0.4	0.55
II. b	**Altered Parameters**	2.0	1.0	*rel*	3.0	0.05	-0.01	0.82

The normalization method $\mathcal{M}_d$ is varied only for the distance-based danger map, but always remains the same for the time-based danger map and the interest map: relative to the minimum and maximum of the current set of velocity samples. This is due to the fact that the *absolute* goal distance is not relevant when creating the interest map. After all, the interest value would increase, the further the robot is from the target. In the objective space, the range of values would change significantly over runtime. This would complicate the selection of the constraint value as well as the weight.
In contrast, for the danger objective, absolute normalization is required to ensure a minimum distance to obstacles. When normalizing to a fixed lower and upper bound, each danger value corresponds to one distance value and

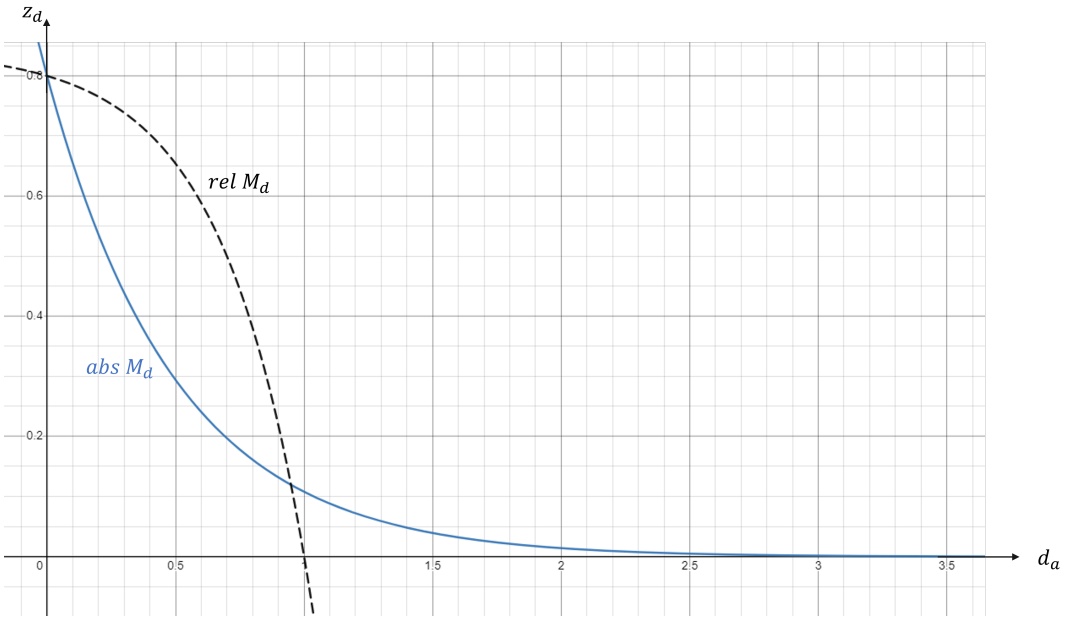


Figure 4.3: Graphs of distance-based danger mapping (objective function). The blue solid graph $h_1(x)$ describes the absolute normalization used in experiment part I. The black dashed graph shows the relative normalization used in experiment part II at $d_{a_{max}} = 1$.

vice versa – over the entire runtime. This unambiguous linkage facilitates parameterization. Using relative normalization, the correspondence changes with the distance of the robot to the obstacle(s).
When selecting the nonlinear mapping function, particularly the slope must be taken into account, which is majorly influenced by $\lambda \neq 0$ (see section 3.2.2). Negative values lead to a convex shape, and positive values to a concave shape of the graph. Where the function graph is flat, there is only a small difference between the danger values, which is why the danger objective loses importance in this range. In other words, if the danger values are too similar, only the interest values are decisive. However, this relation can be exploited: the interest value can be more decisive for longer distances and the danger value for shorter distances - without explicitly using a dynamic weighting (see Figure 4.3), as explained below.

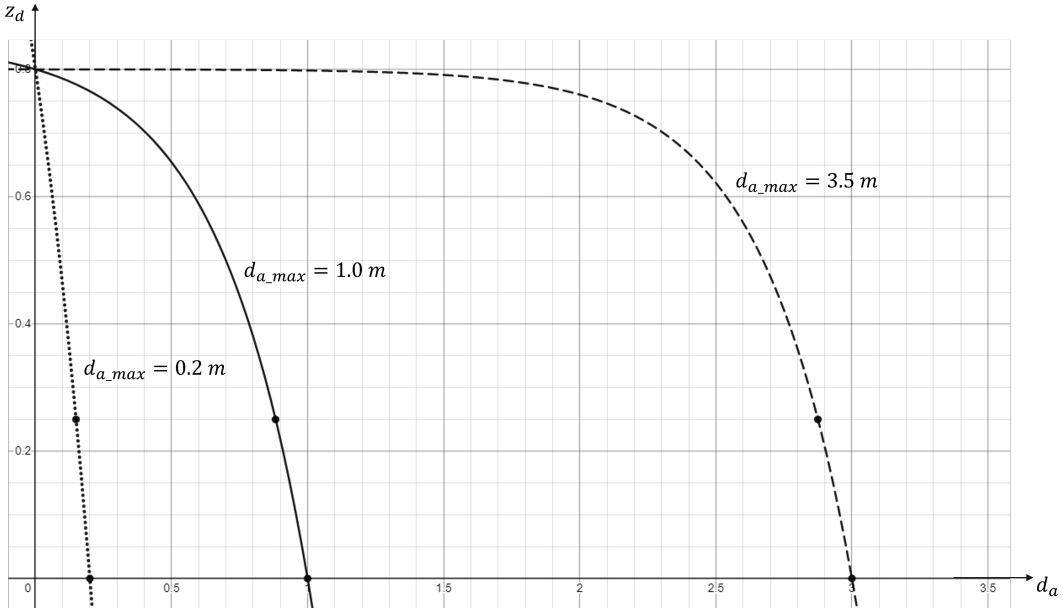


Figure 4.4: With relative mapping *rel* $\mathcal{M}_d$, the robot becomes more risk-averse with decreasing distance to an obstacle. For *rel* $\mathcal{M}_d$, with $\varepsilon_d = 0.25$ and $d_{a_{max}} = 3$ (dashed graph), the minimum allowed distance to the polygon is $d_\tau = 2.8\,m$. The permitted difference to $d_{a_{max}}$ is thus $0.2\,m$. At $d_{a_{max}} = 1$ (solid graph), the difference already reduces to $0.12\,m$; and at $d_{a_{max}} = 0.2$ (dotted graph), the value is even only $0.05\,m$. Thus, as the distance to the obstacle decreases, fewer and fewer trajectories are tolerated.

Parameterizing the Absolute Normalization of Distance-Based Danger

The robot should be more risk-averse near obstacles. Therefore, the function graph should become steeper with decreasing distance. Due to the fixed normalization with the maximum perception range, the function remains the same during the whole runtime. These are the reasons why a convex curve is chosen. The lower negative values for λ, the steeper the graph, which is desired. However, for low λ-values, the robot will approach an obstacle very directly, and then suddenly change its course at a certain distance close to the obstacle. Since this does not comply with the desired behavior, for *abs* $\mathcal{M}_d$, $\lambda = -2.0$ is chosen as a compromise.

Parameterizing the Relative Normalization of Distance-Based Danger

Still, the robot should be more risk-averse near obstacles. However, using *rel* $\mathcal{M}_d$, the slope of the function graph changes with the distance between robot and obstacle, because all distance values are normalized to $d_{a_{max}} = \underset{d_a \forall a}{max}(d_a)$ (the value of the trajectory with the currently greatest distance to the obstacle). This means, the intersection point with the x-axis is always $d_{a_{max}}$, shifting as soon as the robot moves, changing the slope of the function graph. Therefore, λ indicates which distances *relative to* $d_{a_{max}}$ are still tolerated. In extreme cases, only the velocity with $d_{a_{max}}$ is allowed. With $\varepsilon_d = \kappa$ the robot only stops, when all trajectories intersect the polygon (if the robot is able to move even then, a relative mapping must also be introduced for the time-based danger map; considering the scope of this work, this was not implemented and remains open for future work, see chapter 6).
The aim when selecting the λ value is to provide a ε_i value that corresponds as accurately as possible to the expected behavior (e.g. 0.1 is very risk-avers, 0.6 is neither risk-avers nor risk-affine etc.). Since all distance values d_a are very similar ($\Delta d_a \approx 0.3$), the function graph should be steep close to $d_{a_{max}}$ to achieve diversified danger values for the current set of individuals. With a convex function, the ε_i-value would have to be very small (< 0.1) to achieve a behavior which is neither risk-avers nor risk-affine. Because this does not correspond to the expected behavior, a concave function graph is chosen ($\lambda > 0$).
The closer the obstacle (i.e. the lower $d_{a_{max}}$), the steeper the curve should be: first, to take danger stronger into account and second, to allow only larger distances among the current set, i.e. to reduce the tolerance mentioned above when using a danger-constrained method (see Figure 4.4). Stronger changes of the steepness can be achieved with a low value of λ. High λ result in a permanently steep function graph. Hence, $\lambda = 3.0$ is chosen as a compromise.

4.4 Execution of the Experiments

The experimental process adheres to the architecture and procedure described by Mai et al. [32]. Nevertheless, application specific configurations and supplements were made in regard to launching the experiments as well as analyzing the acquired data; both described in the following two sections.

4.4.1 Launch Process

Launching an application with ROS2 entails hierarchically organized launch files, starting and stopping the necessary software components. In addition, configuration files are read out in the launch files and passed on as parameters to the nodes. When all nodes have been started, the experiment process begins, which is divided into several stages (initialization, execution and shutdown) and supervised by a superordinate command node. Via a global topic '/command', this node transmits a call to action to the robots – (get) ready, go, stop – based on the current stage. As soon as a robot has completed the action, it signals this to the command node by publishing 'ready' or 'running' respectively to its 'status' topic.
This procedure ensures that the robots always start and stop under the same conditions, while the conditions themselves can be specified based on the individual application. Moreover, the start and stop are marked in the recorded data set, facilitating subsequent data analysis.
For the experiments conducted in this work, the robot is 'ready' as soon as all subscriptions, publications, and parameters are set up, the transformation from map to robot frame is available, and the update timer is initiated. The experiments are terminated 90 seconds after command node sends the 'go' signal.

4.4.2 Data Acquisition and Processing

All data were acquired using rosbag2 [7], a CLT from the ROS2 ecosystem for recording data published on user-specific topics during the application's execution. The messages are stored with the timestamp of their publication in a serialized format in a database (here: SQLite [22]).
In order to obtain the metrics defined in section 4.2, Jupyter notebooks [16] were used, utilizing rclpy [8], Pandas [33], NumPy [11], and own modules. First, the data is retrieved from the database and deserialized. Next, for each decision step (time step t) the messages of the observed topics (see Table 4.4) are aggregated into one entry. However, each message has a special format, where the required information is wrapped in a data structure. Hence, the messages are interpreted based on a configuration file.
This procedure described above is performed for each experiment run. Subsequently, all tables are joined into one large Pandas data frame, providing the

data basis for examining and plotting utilizing the Python packages Matplotlib [23] and Seaborn [34].

Table 4.4: Topics selected for data recording. In a multi-robot scenario, common topics are used equally by all robots, while topics without a preceding slash apply to each robot in its own namespace.

common topics
/clock
/tf
topics in robot namespace
initialpose
cmd_vel
context_steering/samples
context_steering/point_cloud
context_steering/vis/params
context_steering/vis/polygon
context_steering/vis/all_and_pareto_individuals
context_steering/vis/chosen_and_admissible_individuals

5 Evaluation of the Results

In this chapter, the results of the experiments are evaluated as per the defined metrics (see section 4.2). According to the structure of the experiments (see section 4.1.1), first the results of the base experiment using a rigid distance-based danger map are analyzed as well as the effect of altered decision-making parameters (see section 5.1). Second, in section 5.2 the results of the experiment with an adaptive distance-based danger map and its parameter variation are assessed. Lastly, a summary of the most relevant findings is provided in section 5.3.
For better understanding, the observed behavior in the House1 and House2 scenarios is described by naming the rooms A, B, C, and D (see Figures 4.1d and 4.1e). In addition, the caption of the figures includes the associated part of the experiment (see Table 4.3).

5.1 Rigid Distance-Based Danger Mapping

This section is structured as per the introduced metrics, comparing the robot's actual behavior with the desired one (see section 3.1.1). First, an overview on the number of successful runs is given (see section 5.1.1). These successful runs are then analyzed in section 5.1.2 and evaluated for safety and efficiency (fast, direct, smooth paths). Afterwards, the failed runs are examined (see section 5.1.3). As the parameters have a severe effect on the performance of the robot, lastly, the results of the base experiment are compared to the ones with altered decision-making parameters (see section 5.1.4).

5.1.1 Overview on Effectiveness

In the base experiment, overall, the weighting method was the most successful one, rated by the number of runs in which the robot reached the goal (success

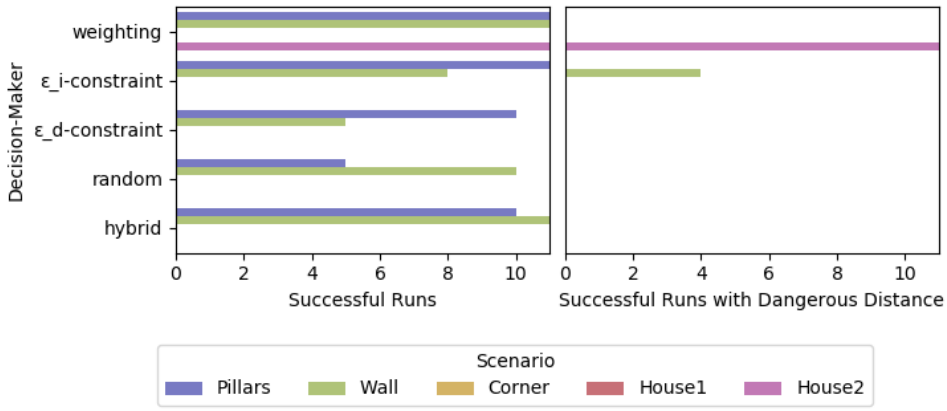


Figure 5.1: *(I. a)* Decision-makers' performance in the scenarios, assessed by the number of successful runs (left) and the number of successful runs with a dangerous distance (right)

rate, see Figure 5.1): in every run the robot succeeds in the Pillars, Wall, and House2 scenario. This makes the weighting method the most flexible one. The other decision-makers perform significantly worse. Only the ε_i-constraint method in Pillars and the hybrid method in Wall have a success rate of 100 % as well. In the other combinations of decision-making method and environment with successful runs, the robot reaches the target in at least 5 and at most 10 out of 11 runs.
The fact all decision makers succeeded several times in the Pillars and Wall world confirms they constitute the simpler environments for this navigation algorithm. In the Corner and House1 scenario, no method was able to steer the robot to the target, and in the House2 configuration, solely the weighting method succeeds.

5.1.2 Successful Runs

In the following, the successful runs are evaluated according to the defined criteria safety and efficiency (see section 4.2). Safety takes the distance to obstacles into account. A run with a distance of approximately 0.01 m between the robot's exterior and the wall (so-called dangerous situation) is considered unsafe. Efficiency evaluates the traveled time and distance, as well as the shape of the paths. Ideally, the traveled route is fast, direct (short), and smooth.

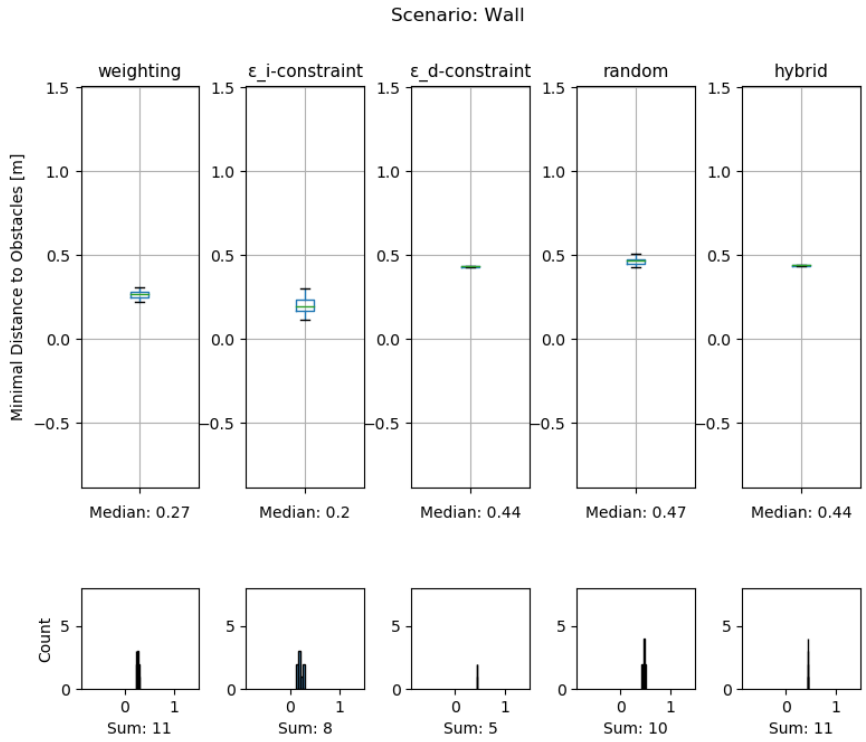


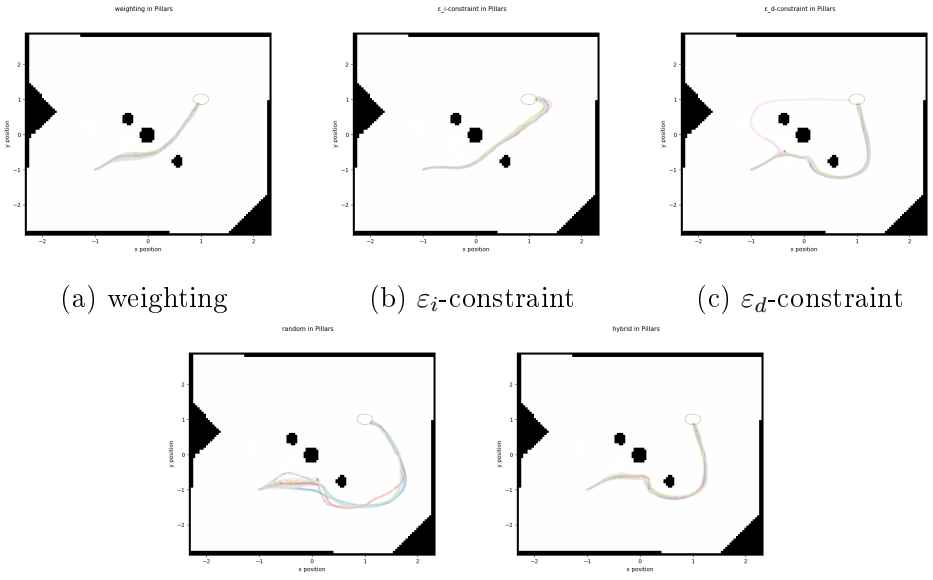
Figure 5.2: *(I. a)* Minimal distance to obstacles in the Wall scenario.

Safety

Only for the decision-making methods without a constraint on danger, dangerous situations were recorded (see Figure 5.1). In the Pillars scenario, such a situation never occurred. In all runs using the weighting method in the House2 world, the robot came extremely close to the edge of the upper wall in room D (median of $0.11\,m$, see Figure 5.8c). Using the ε_i-constraint method in the Wall scenario, in four runs the robot passes dangerously close to the wall (median of $0.2\,m$, see Figure 5.2 and 5.5b). With the danger-constrained methods, the robot kept on average a minimum distance of $0.43\,m$. In the Wall scenario, it is about twice as much as the distance of the weighting and ε_i-constraint method (see Figure 5.2).

Efficiency

Each decision maker observed individually, the path traveled is the same for each run (see Figure 5.3 and 5.5) – except for the random method, what is



(a) weighting (b) ε_i-constraint (c) ε_d-constraint

(d) random (e) hybrid

Figure 5.3: *(I. a)* Paths of the decision-makers in the Pillars scenario.

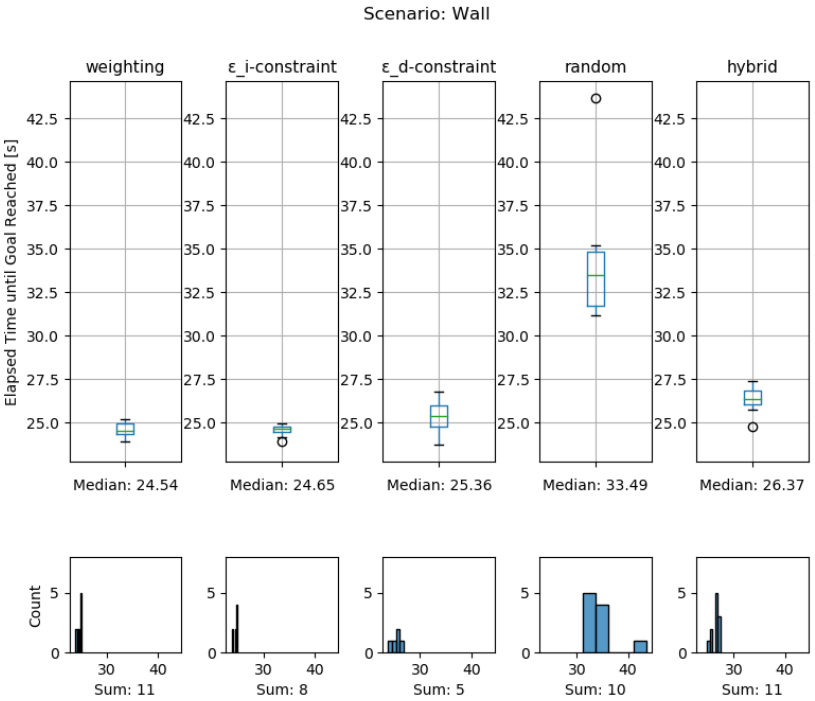
naturally due to the random choice of a speed from the non-dominated set. With the weighting method, the robot reaches the goal fastest in all scenarios where a comparison is possible, i.e. Pillars and Wall world. It is followed by the ε_i-constraint, hybrid and ε_d-constraint method in this order. With the random method, it takes the longest time in all environments (exemplary, see Figure 5.4).

Pillars Scenario: In the Pillars world, the weighting and ε_i-constraint method lead the robot between the central and right pillar. With the other methods, the robot goes around all the pillars on the right side of the room. Thus, the weighting method is more time and route efficient (see Figure 5.3a). First, the robot moves towards the central pillar because it is interested in the goal located directly on the opposite side of it. Next, the danger increases due to the approaching pillar which is why it swerves to the right side, where there is slightly more space. While maneuvering between the pillars, the danger decreases. Now the interest and danger objective are no longer in conflict, which is why the robot heads directly for the goal. The robot moves almost continuously at the maximum translational velocity (median: $0.2\,m/s$).
Compared to this path, the one created by the ε_i-constraint method, is similar. However, this method is naturally risk-averse, since it selects in the admissible

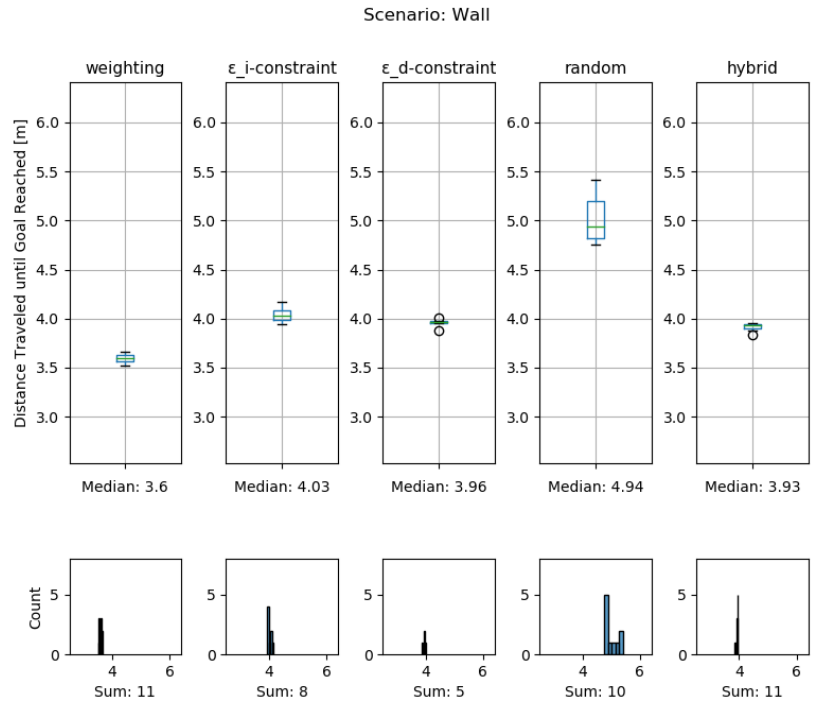
set of velocities for the one with the least danger. Therefore, from the beginning, it avoids the obstacles by moving forward to the right until it can pass straight between the middle and the rightmost pillar. However, in doing so, the robot does not move toward the target, but drives straight ahead until the danger from the right-hand wall is too high, and both the danger is the least and the interest is the greatest on the left. Thus, it turns left. The average speed is similar to the others (median: $0.15\,m/s$).
The decision-making methods constrained with a maximum danger value choose a path around all three pillars. The beginning is very similar to the weighting method, but the limitation of the danger value does not allow the robot to drive along between the pillars. The robot decelerates, occasionally even stops, and turns to the right, deciding to navigate around the right outer pillar at maximum speed (using the ε_d-constraint method, the robot once takes the left route around the pillars). Subsequently, it moves straight towards to the goal. In doing so, the route is indirect and slow.

Wall Scenario: All methods except the ε_i-constraint method steer the robot straight towards the wall (in the center of the room) until the danger prevails. Then the robot turns left or right to avoid the obstacle (with the ε_d-constraint method always left). It is noticeable, the robot passes the obstacle closer on the right than on the left; and with the weighting method it passes closer than with the danger-constrained methods.
The exception mentioned above, the ε_i-constraint method, is more time efficient. The elapsed time until the goal is reached is almost the same as with the weighting method (see Figure 5.4), and less than the remaining methods. However, it takes a longer route than the other three. It has fewer slopes, which allows the robot to travel faster on average (median: $0.17\,m/s$). Whereas with the other decision-makers, the robot travels toward the wall at maximum speed, but suddenly decelerates, and begins to travel slowly parallel to the wall. It does not increase the velocity again until it reaches the end of the wall, where the danger drops. This ultimately leads to a slower travel overall.

Another noticeable fact is that with the ε_d-constraint method, in both the Wall and Pillars world, the distance traveled is the same in all runs, but the time eventually spent by the robot varies greatly (see Figure 5.4). Examining the selected velocities reveals the robot often stops as a fallback solution, but continues shortly after. Similar can be seen with the random method, however

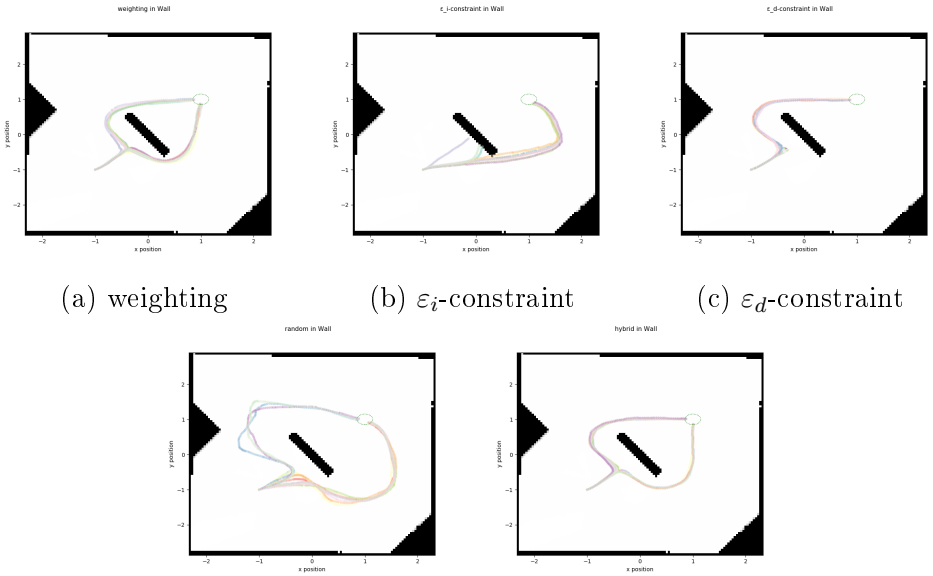


(a) Travel time



(b) Travelled distance

Figure 5.4: *(I. a)* Travel time (a) and distance (b) in the Wall scenario.



(a) weighting (b) ε_i-constraint (c) ε_d-constraint

(d) random (e) hybrid

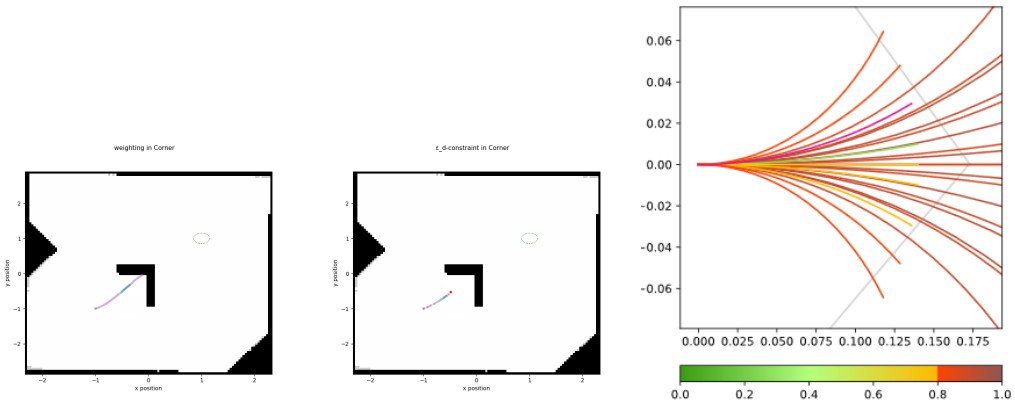
Figure 5.5: *(I. a)* Paths of the decision-makers in the Wall scenario.

it has a wide-spread travel time *and* distance. This is confirmed when looking at the paths, which are very dissimilar in shape and in speed.

5.1.3 Failed Runs

The reason the robot failed to reach the goal is either a collision or a deadlock.

Corner scenario: Regardless of the decision-maker, the robot goes straight toward the inner corner of the obstacle (see Figure 5.6). While colliding with the obstacle using the weighting and ε_i-constraint method, the robot stops in advance using the danger-constrained methods.
The special shape of the obstacle forces the robot steadily forward. This is because the distance between the trajectories to the obstacle is at its maximum straight ahead (see Figure 5.6c). Every turn to the left or right steers the robot closer to a wall, which the flee behavior tries to avoid. Moreover, the goal is straight ahead as well. Even with a different parameterization (for example with $w_d = 1$) the robot does not escape this situation, meaning the context mapping for flee behavior cannot depict the real danger for such a local environment.



(a) weighting (b) ε_d-constraint (c) danger map

Figure 5.6: *(I. a)* Paths of different decision-makers in the Corner scenario.

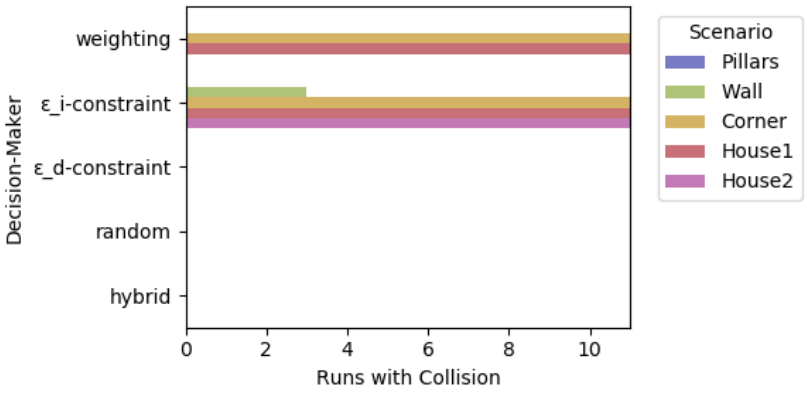
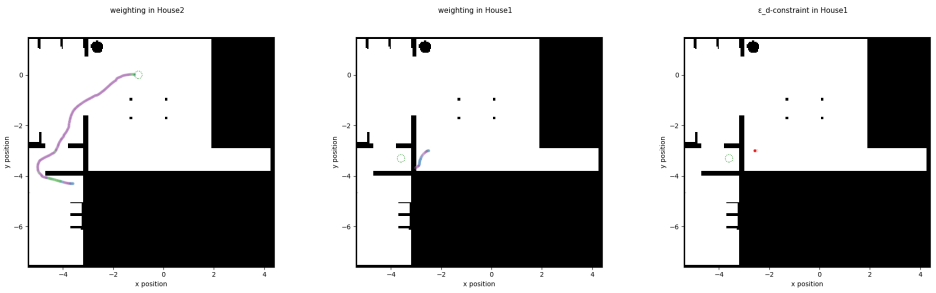


Figure 5.7: *(I. a)* The number of failed runs, ended in a collision.

Decision-Makers with collisions: In the environments where the robot did not succeed using the weighting method (Corner, House1), the runs terminated in a collision (see Figure 5.7). With the ε_i-constraint method it collides in most scenarios; in every run in the Corner, House1, and House2 world, and three times in the Wall environment. Those three runs differ from the rest of the paths in this environment (see Figure 5.5b): once it drives straight into the wall, twice it steers to the right side of the wall but turns towards its center before passing it.

Decision-Makers with deadlocks: The decision-makers with a limitation on danger, namely the ε_d-constraint, random, and hybrid method, did not once lead the robot to a collision. However, in the failed runs, the robot stops en-



(a) weighting (b) weighting (c) ε_d-constraint

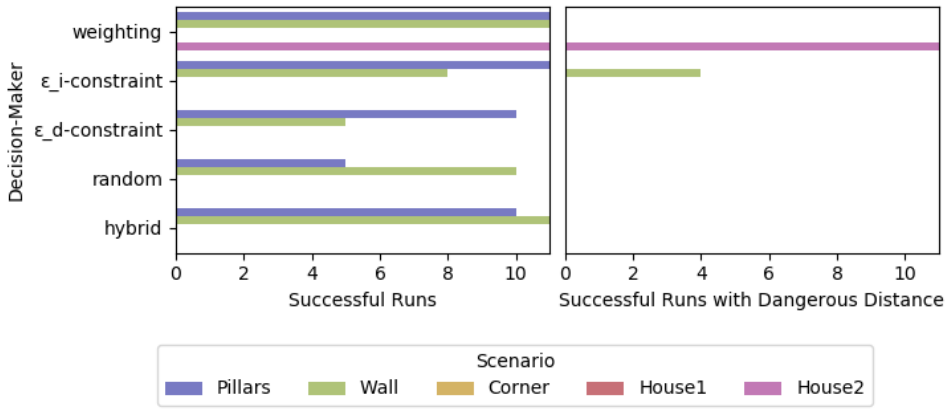
Figure 5.8: *(I. a)* Paths of weighting and ε_d-constraint method (a) in House2, and (b) to (c) in House1. Using methods without a constraint on danger, the robot can collide or move in a dangerous distance to obstacles. With a constraint, the robot can be stuck in a deadlock.

tirely at a certain distance in front of the obstacle, because all velocity samples are more dangerous than allowed by the constraint. In general, stopping due to high risk prevents the robot from colliding. However, often this keeps the robot from being able to move at all.
In the scenarios House1 and House2, the robot cannot even leave the start position (see Figure 5.8). In the failed runs in the Wall environment, the robot moves directly towards the wall but suddenly stops due to the lack of admissible options (see Figure 5.5c). Similarly, in the Pillars world, the robot is stuck in between the central and the right-outer pillar.

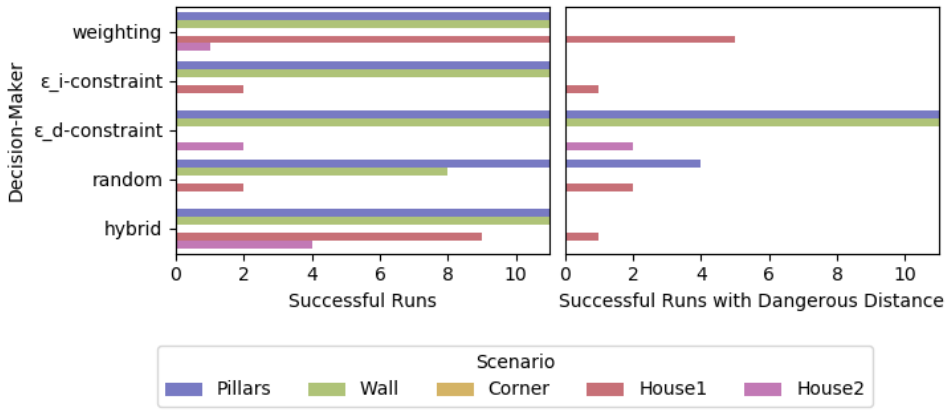
5.1.4 Altered Decision-Making Parameters

For the base experiment, the parameters were chosen such that, with the same parameter set, the decision-makers fulfill the given task *flexibly* in as many scenarios as possible (see section 4.3). However, preliminary experiments have shown that the parameters are scenario-dependent. To point out the significance of optimal parameters and simultaneously gain a first insight into their effects, the base experiment is repeated with altered decision-making parameters (w_d, ε_i and ε_d). Since a full parameter variation is beyond the scope of this work, examples are presented, in which a decision-maker can achieve the goal in scenarios where it was not successful in the base experiment.

Overall, with the altered parameters, the decision-makers achieve a higher success rate in more scenarios. At the same time, dangerous distances were



(a) Base experiment



(b) Base experiment with altered parameters

Figure 5.9: *(I. b)* Performance of the decision-makers, assessed by the number of successful runs (left) and the number of successful runs with a dangerous distance (right)

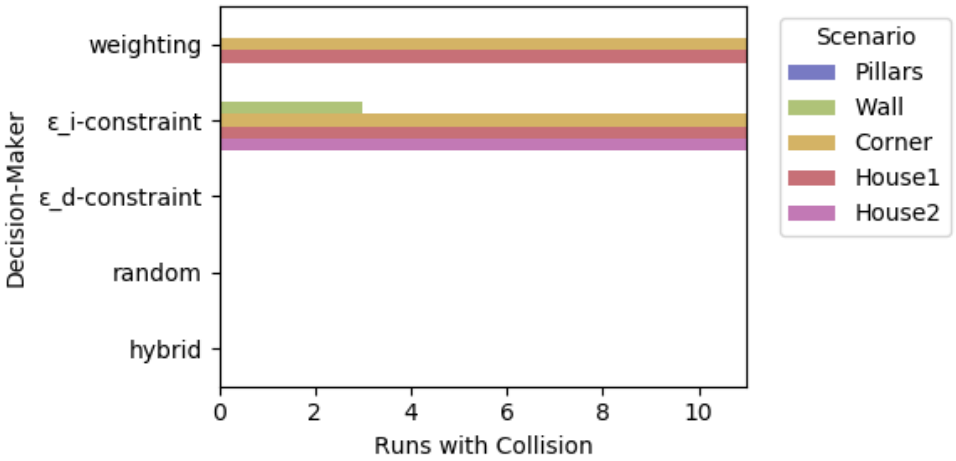
reached more often. Still, no decision-maker can navigate the robot around the corner-shaped obstacle. The number of collisions using the weighting and the ε_i-constraint method is comparably equal in both experiment parts. However, collisions are now also recorded for the ε_d-constraint method.

High Danger Weight

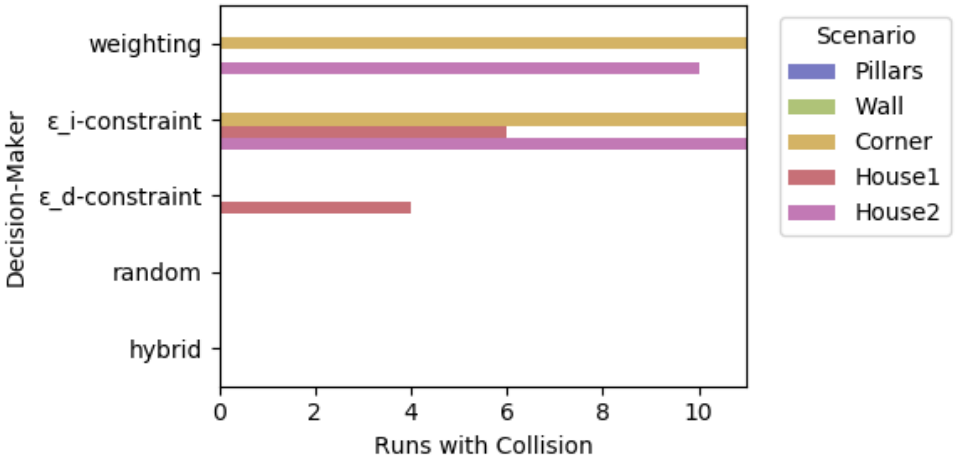
Through the preliminary experiments, it became apparent that for House1 scenario, the robot in general has to be very risk-averse. In the base experiment, the weighting method already succeeded in the Pillars, the Wall and the House2 world. With an increased value of $w_d = 0.92$, the robot now reliably reaches the goal in the Pillars, the Wall, and the House1 scenario (see Figure 5.11a and 5.12a). However, it only succeeds once in the House2 world. The robot is too risk-averse for this situation, as it avoids the left and upper wall of room D, leading the robot further away from the goal. At a certain point, it turns towards the desired goal direction, ultimately putting the robot in the same situation as in the Corner scenario, where it collides with the inner part of the triangle.
The more risk-averse behavior also affects the results in the Pillars and Wall environment. The robot no longer passes between the pillars, but travels around all three pillars (see Figure 5.11a). Similarly, in the Wall scenario, it moves around the wall with more distance.

Low Minimum Interest

With a high constraint on interest, the robot may reach the goal faster, but on the other hand, it has only a limited range of directions at its disposal. To pass a large obstacle, and ultimately reach the goal, a lower constraint on interest can be beneficial, leading to a slower but effective behavior. With lowering the value to $\varepsilon_i = -0.02$, there is almost no interest required for a velocity sample to be admissible. Since the robot always selects the one with the least danger from the set of admissible individuals, the behavior is very risk-averse.
The robot now succeeds in every run in the Pillars *and* the Wall scenario (see Figure 5.9b), with paths similarly to the ones using the weighting method explained above. According to Figure 5.9b, it even succeeds twice in the House1 environment. However, when examining the paths (see Figure 5.12b) and the goal distance over time, it becomes apparent that the localization information

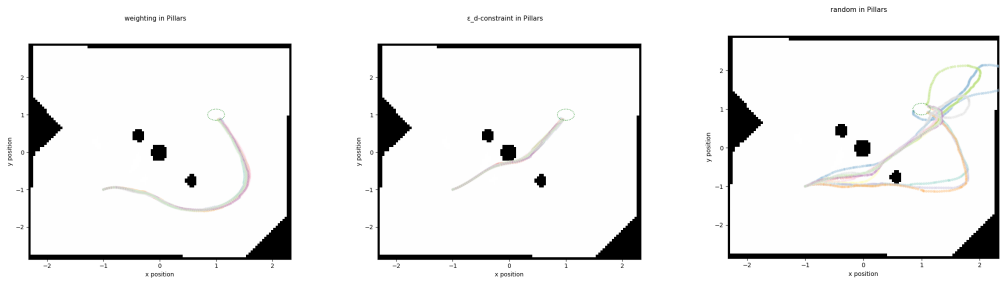


(a) Base experiment



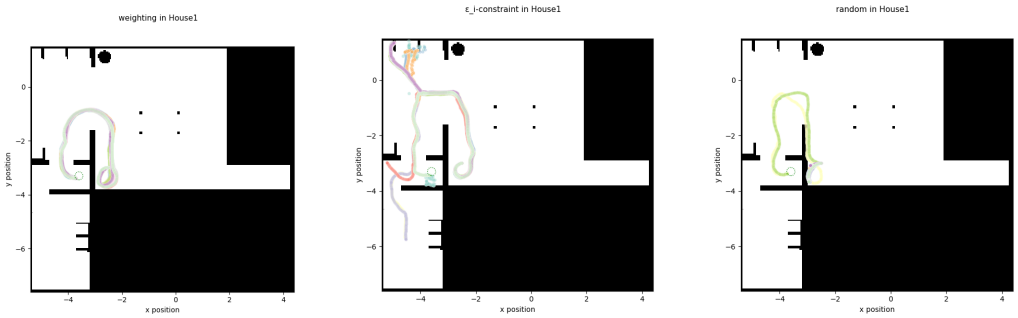
(b) Base experiment with altered parameters

Figure 5.10: *(I. b)* Number of collisions in the scenarios.



(a) weighting (b) ε_d-constraint (c) random

Figure 5.11: *(I. b)* Paths of different decision-makers in the Pillars scenario.



(a) weighting (b) ε_i-constraint (c) random

Figure 5.12: *(I. b)* Paths of different decision-makers in the House1 scenario.

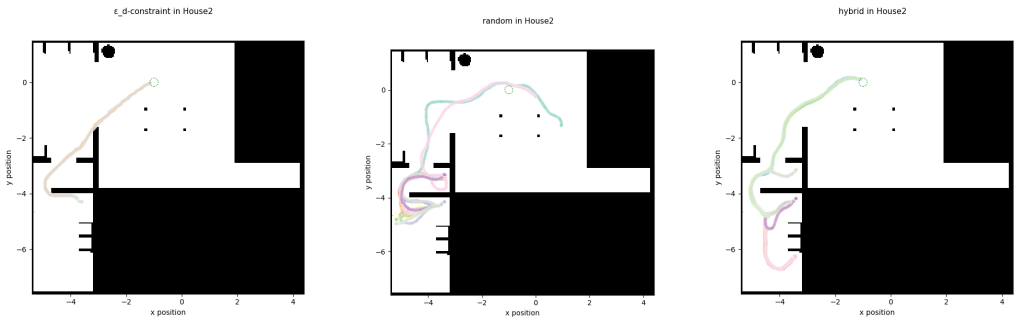
is wrong in the first run. Hence, the robot actually succeeded only once in the House1 world. However, in all runs, the robot manages to escape the corner situation in room A, follow the wall, and enter room B. In four runs, it turns towards the target, drive through the narrow passage on the bottom of room B, and reach room C where the target is located. Though, the robot turns only once towards the goal. In the other three cases, it turns away from it.
In the House2 world, the robot mostly turns away from the goal, staying in the room of the starting position. In five runs, it takes the turn and passes through the small passage. Since the next doorway is smaller than the space on the right side, it turns towards the corner of the room pointing to the goal. Here, the robot collides with the wall. This again shows: when the robot is risk-averse, it will reach for the largest space available.

High Maximum Danger

Unlike the other methods, a more risk-averse behavior (lower ε_d) does not cause the robot to reach the goal in the House1 or House2 scenario. When examining the decision space in the base experiment, it is noticeable that in the scenarios Pillars, Wall and Corner all options are available for a long time while heading for the goal and at the same time approaching the obstacle. As soon as the distance $d_{a_{max}}$ is reached, for which $z_d(\vec{v}) = \varepsilon_d$, all individuals exceed the constraint. Since there are no more allowed velocities at this distance to a wall, the robot stops. The reason for this is that the distance d_a is very similar for all trajectories (see section 4.3). Despite the mapping function diversifying the values, to maintain mobility of the robot, a scenario-sensitive ε_d value would be required, defined up to several decimal places. As this value is manually chosen, this approach is not practical. Consequently, for this part of the experiment to succeed in the House1 and House2 world, a more risk-affine (higher ε_d) behavior is chosen.
In order to exclude all trajectories with a future collision, for this experiment $\varepsilon_d = \kappa = 0.8$ is chosen. Lower values always lead to the situation explained above.

With the *ε_d-constraint method* using this value, the robot now accomplishes the given task in *all runs* in the Pillars and the Wall scenario. Twice it reaches the goal in the House2 world (see Figure 5.9b). However, with an average minimum distance of $0.1\,m$ (Wall, House2) and $0.11\,m$ (Pillars), the robot moves dangerously close to the obstacles in the environment (see Figure 5.13a and 5.11b).
Although in these three scenarios a collision did not occur in the simulation, when the same experiment is performed with a real robot, a collision is highly probable due to errors in perception and actuation. The collision rate in the House1 world confirms this: in four runs the robot collides with the wall.

The *random method* succeeds in all runs in the Pillars scenario (see Figure 5.9b), which is a major improvement compared to the performance in the base experiment, where it succeeded in five runs (see Figure 5.1). With this higher constraint value, it additionally succeeds twice in the House1 scenario. In the Wall environment, however, it performs worse, reaching the goal only eight times instead of ten.
In about a third of the runs, the robot gets dangerously close to the right outer pillar (see Figure 5.11c). In both successful runs in the House1 scenario,



(a) ε_d-constraint (b) random (c) hybrid

Figure 5.13: *(I. b)* Paths of different decision-makers in the House2 scenario.

dangerous situations occur when leaving the corner and moving parallel to the left wall of room A (see Figure 5.12c).
According to the success rate, the robot did not succeed in the House2 world. Though, the paths in Figure 5.13b reveal, the robot did actually accomplish the given task of maneuvering through the world to the goal, but it just missed the defined goal area (tolerance of $0.15\,m$). Anyway, in most cases, the robot is already stuck in either room C or D.

With the altered parameters, the hybrid method performs similarly well as the weighting method, which is much better compared to the base experiment. It succeeds in all runs in the Pillars and the Wall scenario. Additionally, in the House1 world, it succeeds nine times, but in the House2 world, four times (which is the best result over all decision-makers for this scenario with altered parameters).
During the two failed runs in the House1 scenario, the robot moves towards the lower left corner of room A and, thus, maneuvers itself into a deadlock. In the House2 scenario, it gets into a deadlock in the upper right corner of room C or D (directed towards the goal position, see Figure 5.13c).

5.2 Adaptive Distance-Based Danger Mapping

Since the methods with a constraint on ε_d using absolute normalization cannot adapt to different spaces, a relative normalization is considered. On the one hand, this variant is more complicated in terms of parameterization. The minimum distance to be maintained from obstacles does not permanently cor-

respond to *one* defined value, but is determined dynamically: the closer the robot is to the obstacle, the smaller the permissible minimum distance (see section 4.3.2). On the other hand, this allows adaptive behavior in different environments, keeping the robot able to move. The values for w_d, ε_i, and ε_d were determined in preliminary experiments based on the same criteria as in the base experiment, i.e. to be as universal as possible.

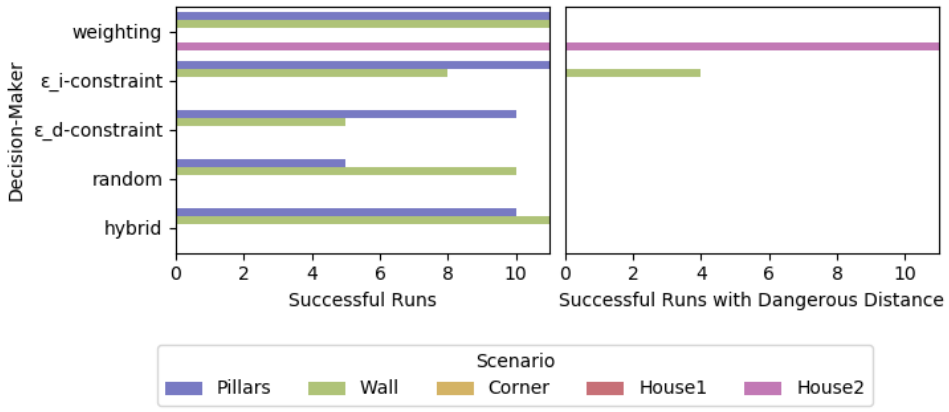
For better comparison, this section is structured in the same way as the previous one. First the effectiveness of the decision-making methods is assessed (see section 5.2.1), then the safety and efficiency of the successful runs (see section 5.2.2), and the reasons for failures (see section 5.2.3). Lastly, the effect of altered decision-making methods is evaluated (see section 5.2.4).

5.2.1 Overview on Effectiveness

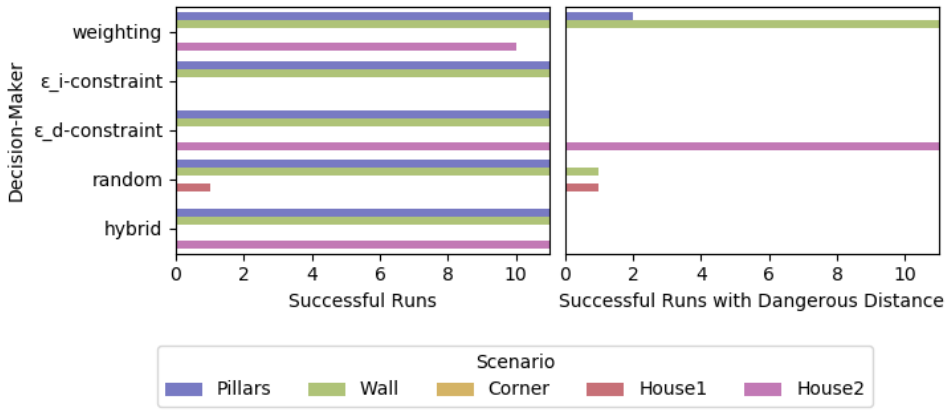
In total, the number of successful runs is higher than in the base experiment, but also the number of dangerous situations increased. In the Pillars and the Wall scenario, all decision-making methods can steer the robot to the goal in every run (see Figure 5.14b). This is a large improvement compared to the base experiment, especially for the methods with a danger constraint. Even in the House2 world, the weighting, the ε_d-constraint, and the hybrid method succeed in (almost) every run. In fact, the ε_d-constraint and hybrid method have the highest success rate. Similarly to the weighting method in the base experiment, both succeed in all runs in the Pillars, Wall, and House2 scenario. Also with relative normalization, no decision maker can navigate the robot to the goal in the corner scenario. In the House1 scenario, only the random method succeeds once.

5.2.2 Successful Runs

As in the corresponding section of the base experiment (see section 5.1.2), the successful runs are rated by the safety and efficiency of the travelled route (see section 4.2). While safety reviews the shortest distance between the robot and obstacles, efficiency assesses the traveled time and distance, as well as the shape of the paths.



(a) Base Experiment with absolute normalization.



(b) Comparison experiment with relative normalization

Figure 5.14: *(II. a)* Performance of the decision-makers, assessed by the number of successful runs (left) and the number of successful runs with a dangerous distance (right)

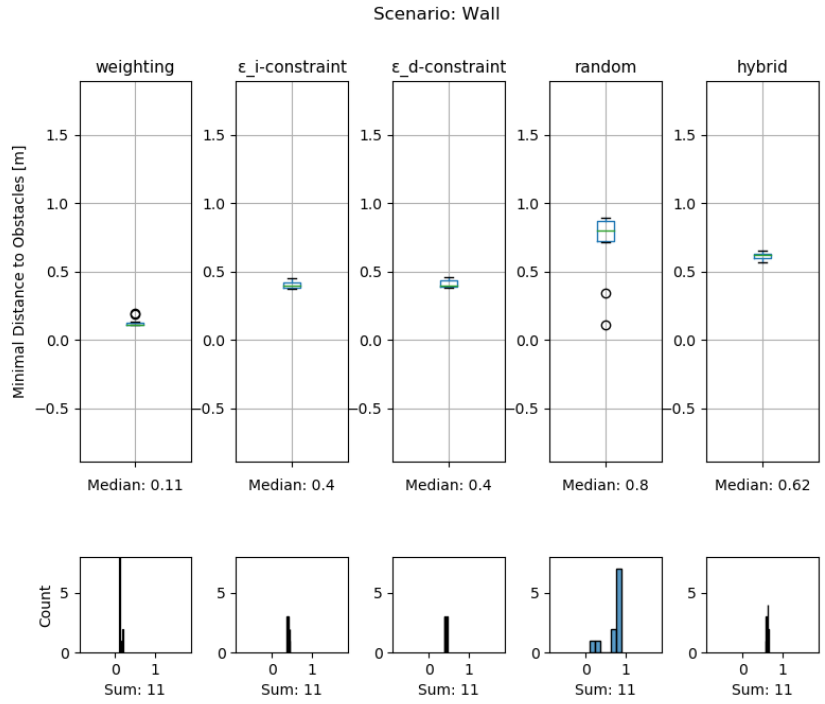
Safety

Similarly to the weighting method in the base experiment, the ε_d-constraint method always reaches a dangerous distance to the upper wall in room D of the House2 scenario (see Figure 5.15b) The weighting and the hybrid method maintain a distance more than twice as large. Also with relative normalization, the weighting method navigates the robot in dangerous situations (see Figure 5.14b); in this experiment, though, in the Pillars and the Wall scenario. This is mostly due to the comparably low danger weight $w_d = 0.55$. By behaving in a more risk-affine manner, the robot is able to accomplish the given task in three scenarios, while generally maintaining less distance from the obstacles.
For the random method, one dangerous situation was recorded in each of the Wall and House1 scenarios (see Figure 5.15a). In the Wall scenario, the median distance of the random and hybrid method is about twice as much as the distance of the other methods.
Since the hybrid method is a hybrid of weighting and constraint, it could be expected that it too would put the robot in dangerous situations. However, it always maintains a non-hazardous distance. Altogether, the hybrid and the ε_i-constraint method are the only decision-makers without a dangerous distance. Yet, the latter collides in all failed runs (see section 5.2.3). This makes the hybrid method the safest one in this experiment.

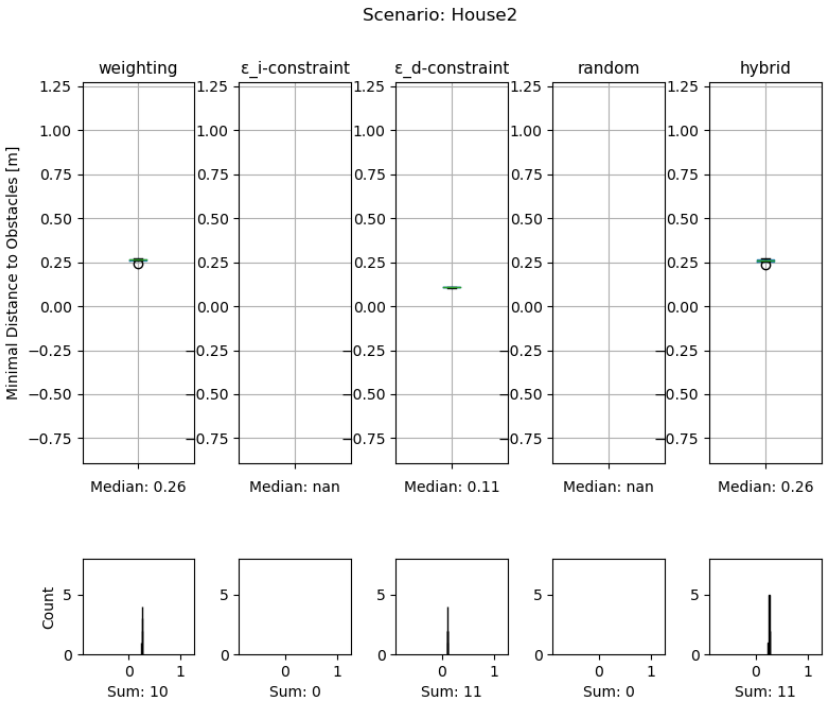
Efficiency

In this comparison experiment, like in the base experiment, each decision-maker repeatedly chooses almost the same path, except for the random method. The weighting method has the lowest travel time in the Pillars scenario, but the ε_d-constraint method in the Wall and the House2 world. Again, with the random method, the robot takes the most time in the Pillars and Wall environment.

Pillars scenario: Compared to the base experiment, the paths of each method look similar for the weighting, ε_i-constraint and random method, but differently for the ε_d-constraint and hybrid method (see Figure 5.16). More precisely, instead of navigating around all three pillars, the robot now selects the path between the central and the right outer pillar. Though, with the weighting method, the robot once chooses to travel between the central and the left outer pillar. This is of interest since this is the narrowest part of the whole

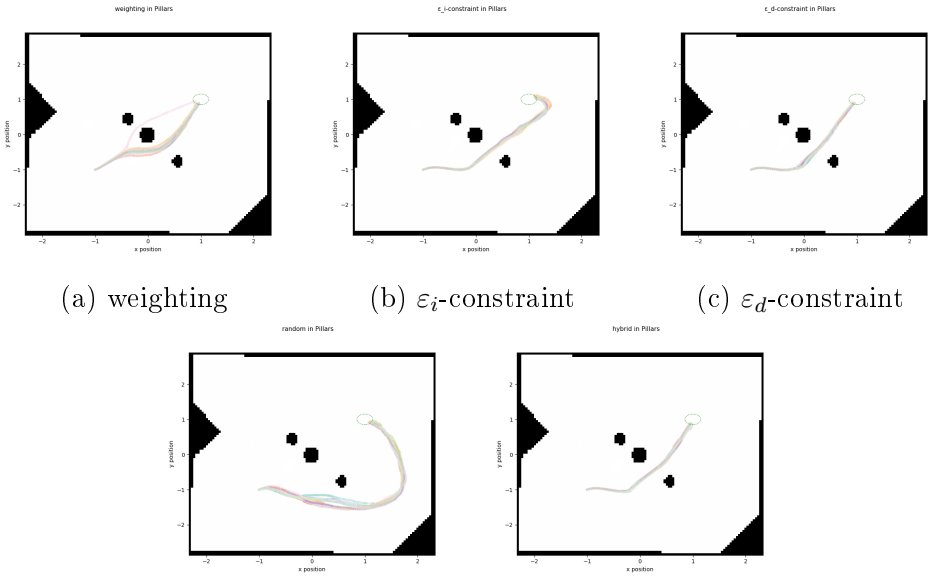


(a) Wall



(b) House2

Figure 5.15: *(II. a)* Minimal distance to obstacles during the runs.



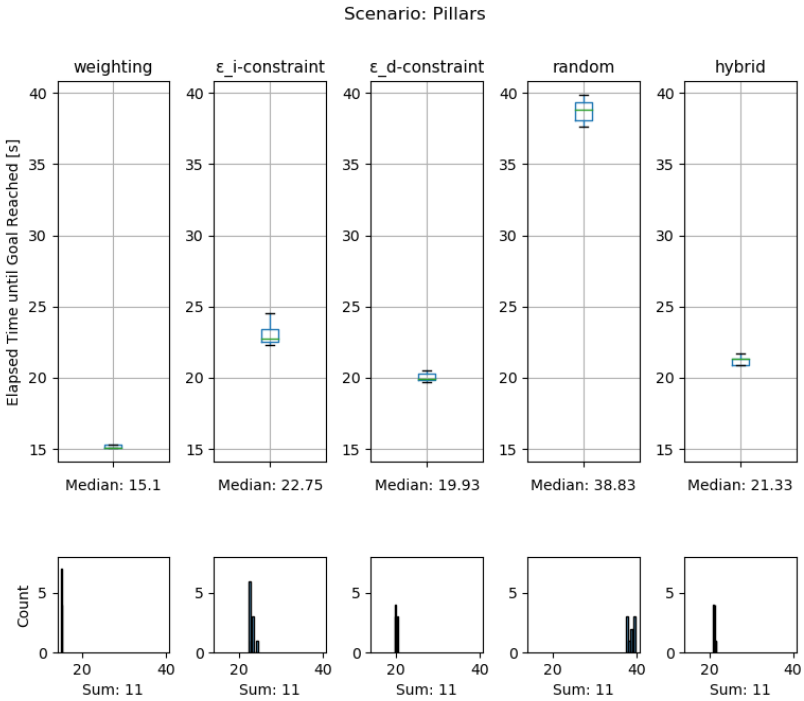
(a) weighting (b) ε_i-constraint (c) ε_d-constraint

(d) random (e) hybrid

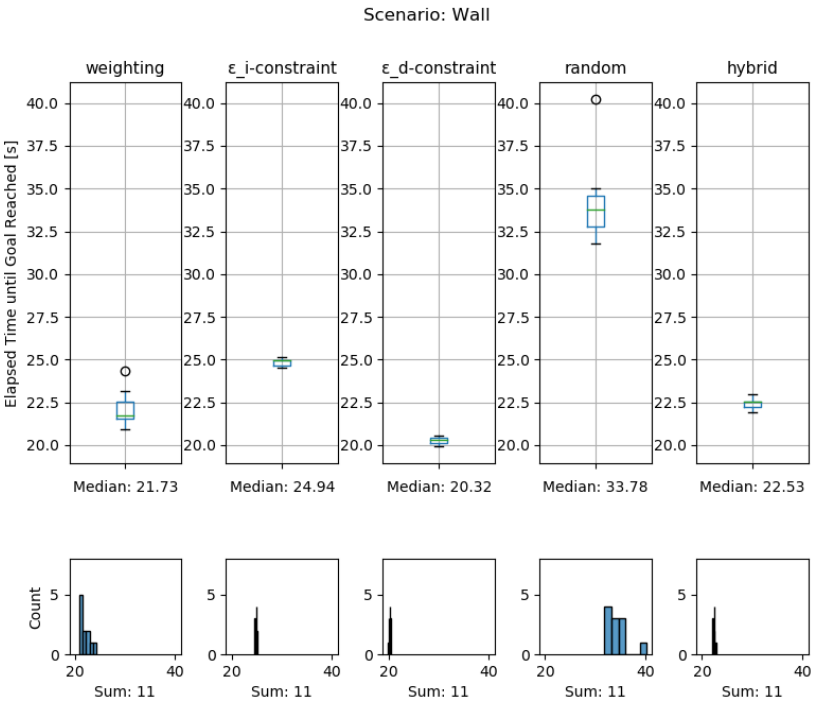
Figure 5.16: *(II. a)* Paths of the decision-makers in the Pillars scenario.

environment. On the other hand, as stated above, the danger weight is comparably low, leading to a more risk-affine behavior.
As in the base experiment, the weighting method has by far the lowest travel time (see Figure 5.17a), majorly because of the high average speed (median of $0.2\,m/s$) but also because of the direct route. Due to the more direct routes of the danger-constrained methods, they achieve a much lower travel time, e.g. the ε_d-constraint method saves up to almost $10\,s$, making it the second-fastest method. It is followed by the hybrid and ε_i-constraint method. The ε_d-constraint method also took a short route, but traveled on an average speed of merely $0.16\,m/s$, leading to a longer travel time. Even though the ε_i-constraint method has the same average speed, it is slower than the hybrid method. When examining the paths (see Figure 5.16b), it becomes apparent that this is due to the additional turn close to the goal, making the route longer than necessary.

Wall scenario: The general shape of the weighting method is similar to the paths in the base experiment. However, due to the higher risk-affinity, the robot moves closer to the obstacles before swerving to the side (see Figure 5.18). In the base experiment, the paths of the danger-constrained and weighting methods look similar. In this comparison experiment, though, the shape

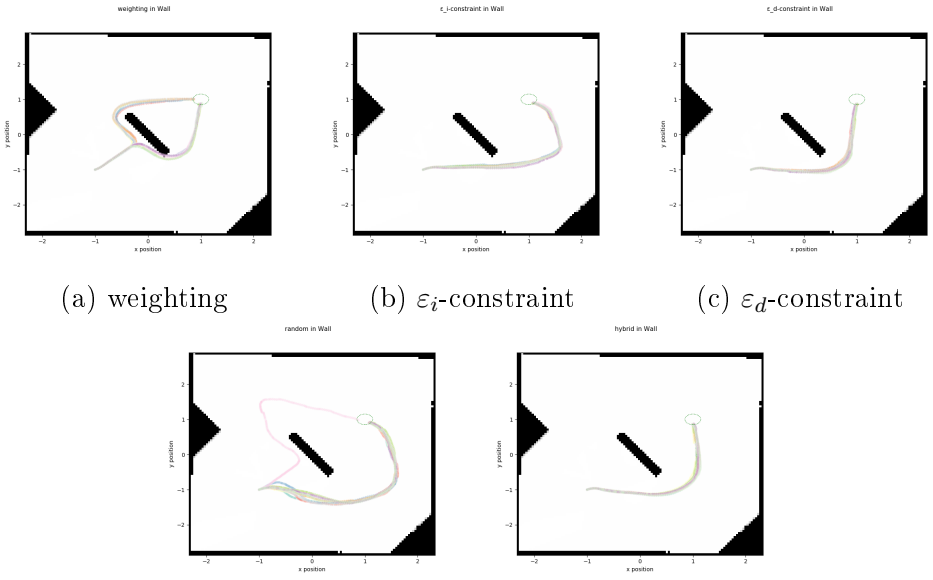


(a) Pillars



(b) Wall

Figure 5.17: *(II. a)* Travel time in the (a) Pillars and (b) Wall scenario.



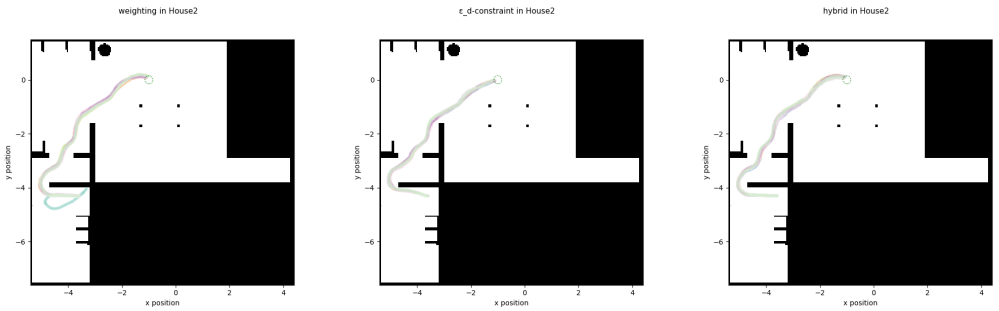
(a) weighting (b) ε_i-constraint (c) ε_d-constraint

(d) random (e) hybrid

Figure 5.18: *(II. a)* Paths of the decision-maker in the Wall scenario.

of the danger-constrained decision-makers has changed. Instead of straightly approaching the obstacle in the center, and then slowly closely driving around it, the robot now moves in a large bow right from the start position. It is rather comparable with the path created using the ε_i-constraint method. This may not significantly reduce the traveled distance, but keeping a safe distance to the obstacle allows higher speed on average. In the end, for the ε_d-constraint method, the average travel time is about $5\,s$ less compared to the base experiment. It is now the fastest decision-maker (see Figure 5.17b). Again, with the ε_i-constraint method the robot travels at the same average velocity but due to the indirect route, it takes even longer than the weighting and the hybrid method.

House1 scenario: In almost every run, the robot turns to its left towards the bottom wall of room A, where it stops due to the lack of admissible options. During the one successful run (see Figure 5.21a), it moves in a smaller turning loop, and thereby escapes the deadlock. As soon as the robot is oriented parallel to the left wall of room A, it travels along this wall until reaching the opening to room B. Then it turns in the direction of room C, adjusts its course to pass the small opening and reaches the goal. With $0.12\,m/s$ on average the robot is very slow.



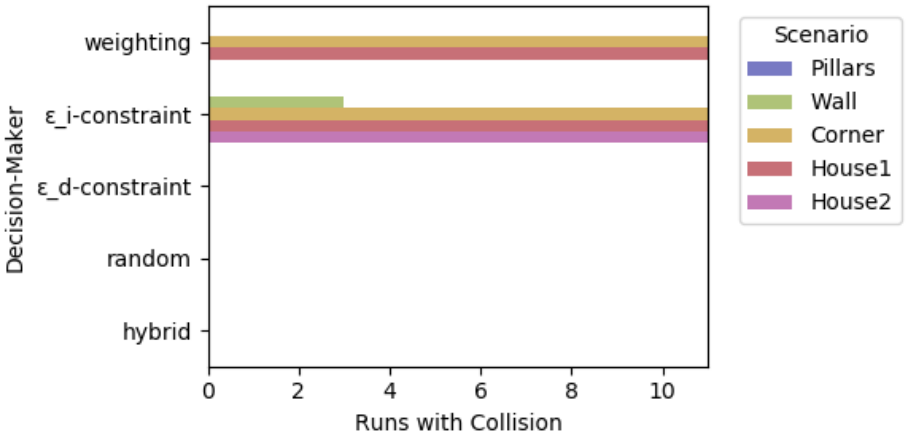
(a) weighting (b) ε_d-constraint (c) hybrid

Figure 5.19: *(II. a)* Paths of successful decision-makers in the House2 scenario.

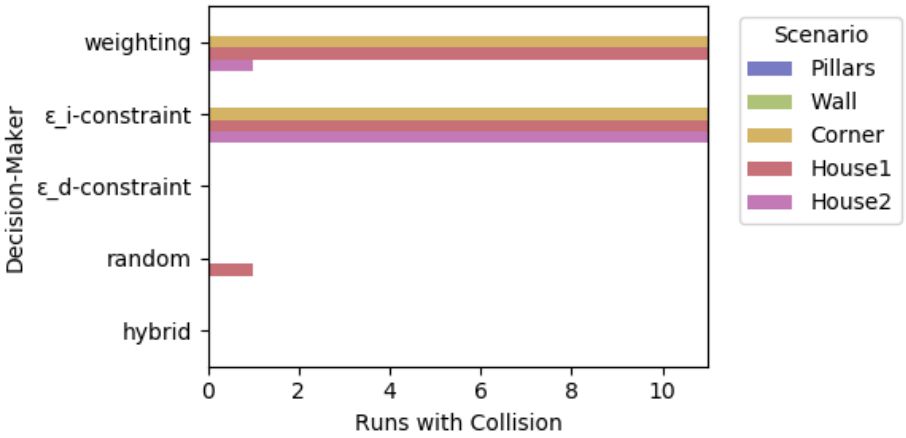
House2 scenario: The paths of the ε_d-constraint, weighting and hybrid method look very similar. Except, in one run using the weighting method, the robot turns back to the start position and collides with the right upper corner of room D. As already mentioned, the path of the former method differs only in the generally smaller distance from the walls. This makes the route shorter, but also a somewhat more dangerous. Despite the paths look identical in the base experiment using the weighting method, all methods in this comparison experiment are slower due to the lower average speed. The ε_d-constraint method has the lowest travel time again. However, this is mainly due to the higher average speed, as the traveled distance is almost the same for all the succeeded decision-makers. It is the greater average distance to obstacles of the weighting and hybrid method that makes the difference here.

5.2.3 Failed Runs

The total number of collisions decreased by one compared to the base experiment (see Figure 5.20). Like explained in the base experiment (see section 5.1.3), the methods without a constraint result in a collision when not being successful. The ones with a constraint do not collide, but get stuck in a deadlock. Though, for the random method, there is one collision in the House1 environment recorded. With the weighting method, the robot collides once in the House2 world. Using the ε_i-constraint method, it does not collide with the obstacles in the Wall scenario anymore (compared to the base experiment). But in the House2 world, the robot still collides promptly with the upper wall of room D in the same way as in the base experiment. In the same world, using

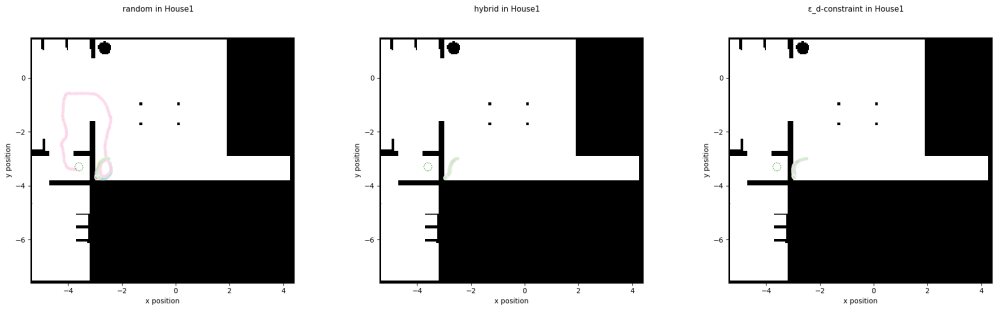


(a) Base experiment with absolute normalization



(b) Comparison experiment with relative normalization

Figure 5.20: *(II. a)* Number of collisions in the scenarios.



(a) random (b) hybrid (c) ε_d-constraint

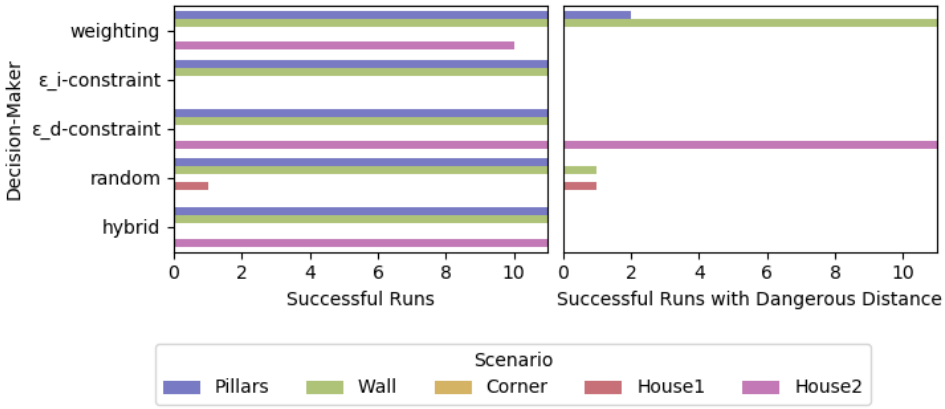
Figure 5.21: *(II. a)* Paths of (a) a successful run and (b)-(c) failed runs in the House1 scenario.

the random method, the robot first navigates away from the starting position in room D and then either turns back to the upper right corner, moves down to the bottom right corner, or reaches room C but collides with the right upper corner (pointing in the goal direction).
No decision-maker is able to navigate the robot around the triangle in the Corner scenario. This underlines once again that the algorithm itself cannot model the solution space for such situations. Also, in the House1 scenario, no decision-making method accomplishes the given task repeatedly (see Figure 5.21). Except for the one run using the random method, all decision-makers steer the robot further into the left bottom corner of room A, ultimately colliding or stopping in time (but thus not being able to move anymore).

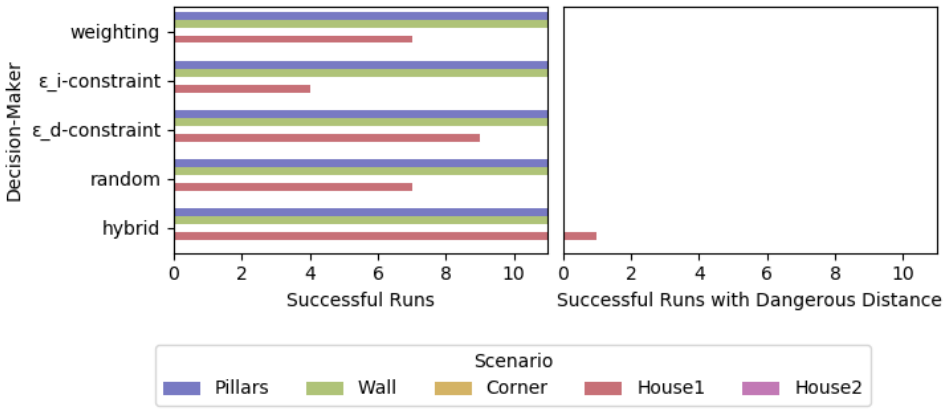
5.2.4 Altered Decision-Making Parameters

As in the Base Experiment, the performance of the robot in each scenario is highly dependent on the parameter values. In order to demonstrate their effect, other parameters were chosen, primarily with the aim of succeeding in the House1 scenario.

With a higher danger weight, lower minimum interest, and lower maximum danger than in the original comparison experiment, the robot does not succeed in the House2 scenario anymore but in House1 with each decision-making method (see Figure 5.22b). It still accomplishes the given task with every decision-maker in all runs in the Pillars and the Wall environment. Again, the Corner scenario is without any successful run.



(a) Comparison experiment



(b) Comparison experiment with altered parameters

Figure 5.22: *(II. b)* Performance of the decision-makers in each scenario.

The number of dangerous situations dropped to an overall number of one (hybrid method in House1). Like already in the other experiments, the robot collides in every failed run using the weighting and the ε_i-constraint method. With the danger-constrained methods, it does not collide once (see Figure 5.23b).
The paths in the Pillars and Wall scenario are a large curve around the obstacles in the middle of the room. With every decision-maker in the House2 world, the robot moves to the lower right end of room D and collides with or stops in one of the two corners there. Only with the ε_d-constraint method, the robot moves either there as well or to room C, stopping in its upper right corner.

High Danger Weight

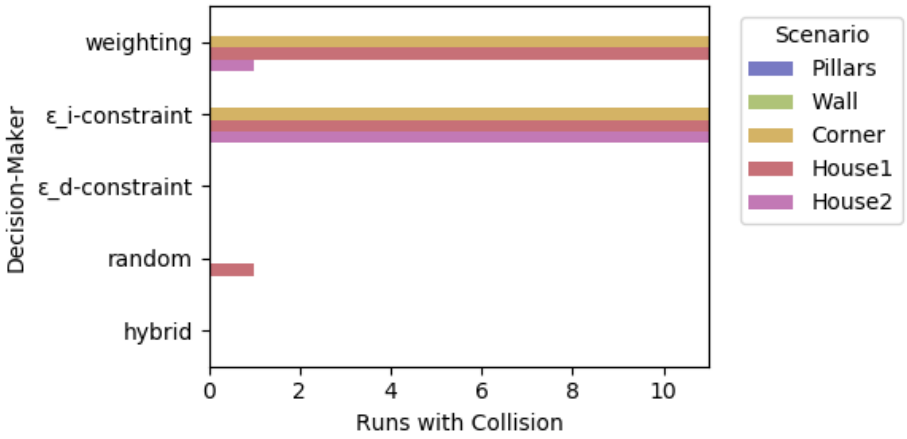
The high danger weight of $w_d = 0.82$ was chosen for the same reasons as in the base experiment with altered parameters (see section 5.1.4). In the world of House1, the robot succeeds in seven runs. However, in three out of these runs, it first moves around the entire table in room A before entering room B, and subsequently reaching the goal (see Figure 5.24a). In failed runs, the robot collides with the left bottom corner of room A.

Low Minimum Interest

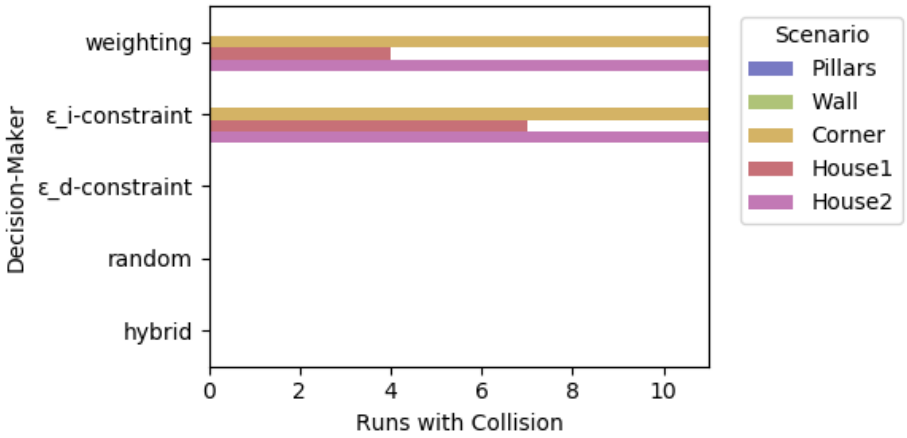
The low minimum interest of $\varepsilon_i = -0.01$ was chosen for the same reasons as in the base experiment with altered parameters (see section 5.1.4). In four runs, the robot succeeds in the House1 scenario. Once it accomplishes to move to room C, but turns away from the goal and enters room D, where it eventually collides. However, in most runs, the robot collides with the left upper corner of room B.

Low Maximum Danger

Due to the adaptive danger value, the robot does no longer stop moving at a fixed distance. Hence, for an environment with little space, such as the world of House1, the maximum allowed danger ε_d can be reduced to gain a risk-averse behavior.
With the low danger limit of $\varepsilon_d = 0.05$, the ε_d-constraint method steers the

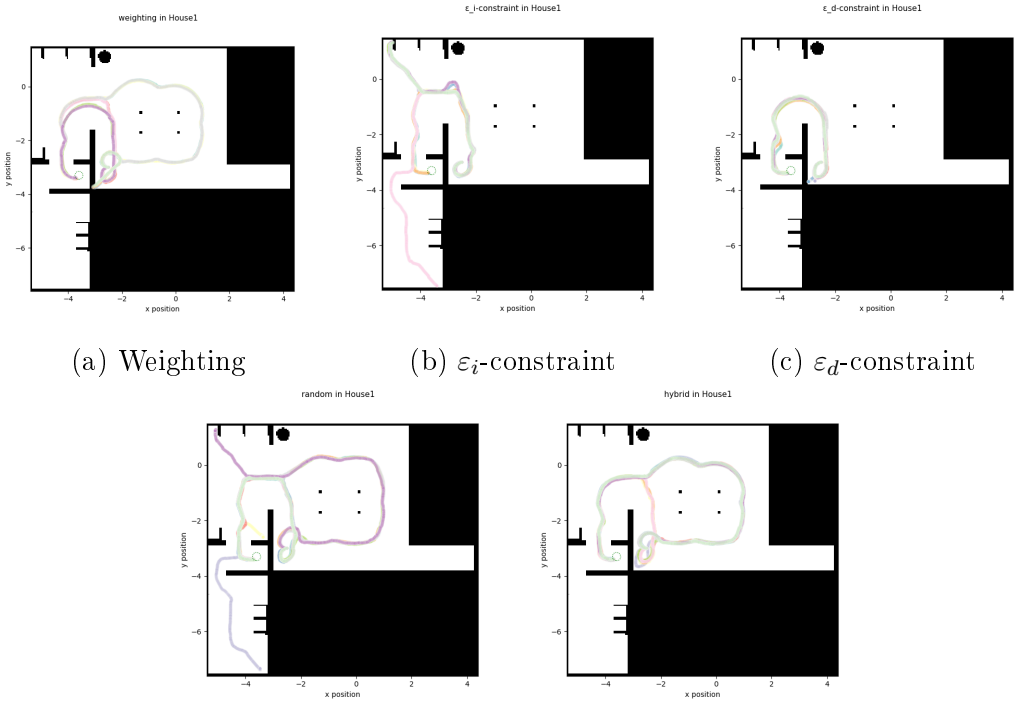


(a)



(b)

Figure 5.23: *(II. b)* Number of collisions in each scenario.



(a) Weighting (b) ε_i-constraint (c) ε_d-constraint

(d) random (e) hybrid

Figure 5.24: *(II. b)* Paths of the decision-makers in the House1 scenario.

robot in nine runs to the goal. In the other two runs, it moves to the left bottom corner of room A and stops due to the lack of admissible options. Using the random method, the robot reaches the goal seven times. However, in three out of these runs, it takes the detour around the table described above. The same applies to the execution with the hybrid method. In three runs, the robot chooses the direct path, but in most cases the navigates around the table. With the hybrid method, the robot reaches the goal in every run, making it the most successful one in this part of the experiment.

5.3 Summary of Results

The introduced algorithm was executed with five different decision-making methods in five different simulated scenarios; first, with absolute normalization (I. a, base experiment) and second, with relative normalization (II. a, comparison experiment) of the distance-based danger map. Additionally, both main experiments were repeated with altered parameters (I. b and II. b), pointing out the significance of scenario-specific optimization. The task in all scenarios is to steer the differential-drive robot from its start position to the goal position without collisions[1].

To enable a comparison between the decision-makers, the scenarios were chosen with increasing difficulty in terms of risking a collision or a deadlock. In most experiments, the robot reached the goal in every run in the Pillars and the Wall environment (in the base experiment, in at least 5 of the total 11 runs). This confirms that they are less difficult environments for this navigation algorithm. Hence, in the following, they are referred to as the basic scenarios.
In the *Pillars environment*, the chosen path depends on the degree of risk-aversion. Being highly risk-averse, the robot drives around all three pillars, choosing the route with the least danger. With a bit more risk-affinity, the robot passes between the middle and the outer right pillar. Is the behavior more risk-affine, even the path between the middle and the outer left pillar is chosen. In the *Wall scenario*, the robot selects either the left or the right way around the wall in the center. With a high risk-aversion, mostly the right side is chosen, keeping a large distance from the walls. In the *Corner scenario*, all

[1]For complete data of the experiments, see https://archive.org/details/context-steering-with-differential-drive-robots

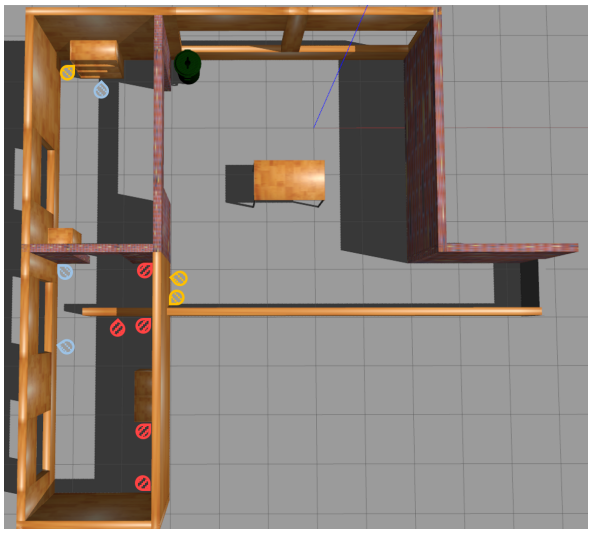


Figure 5.25: The most common collision points in House1 (yellow), in House2 (red), and in both only when using the random method (blue).

decision-makers fail, steering the robot straight into the corner. Subsequently, it either collides or stops.
The *House1 and House2 scenarios* combine the main obstacles of the three aforementioned environments: small circular, large I-shaped and L-shaped obstacles. They only differ in the configuration of start and goal position. With moderate values for w_d, ε_i, and ε_d, the robot can succeed in the House2 scenario with at least one decision-maker. In order to reach the goal in the House1 scenario, the behavior has to be very risk-averse. Then again, the robot can no longer cope with the House2 scenario.

As expected, in *failed runs*, the decision-makers without a constraint on danger led the robot into a collision. All decision-makers with a danger constraint steer the robot towards the obstacle, until it is unable to move due to the lack of admissible trajectories (deadlock). There is one exception: using the random method, the robot accomplished the task but just missed the goal area. The main reasons for a collision or deadlock are: the interest is too high outweighing the danger in this situation, the robot is so risk-averse that it escapes into a supposedly larger space while moving further away from the goal, or the robot is facing a corner. In accordance with the results in the Pillars, Wall and Corner worlds, in the House1 and House2 scenarios, the robot mostly collides with or is stuck close to a *corner*, less often a wall (see Figure 5.25).

Table 5.1: Summary of performance of the decision-makers. The decision-makers are rated as efficient (+), inefficient (–), or neutral (o) for those scenarios and parameters settings (I. a-II. b), where the decision-maker was effective (successful in at least 9 out of 11 runs). Runs with dangerous situations are excluded.

		Pillars	Wall	Corner	House1	House2
weighting	I. a	+	+			
	I. b	–	o			
	II. a	+				+
	II. b	o	o			
ε_i-constraint	I. a	o				
	I. b	–	–			
	II. a	o	+			
	II. b	–	–			
ε_d-constraint	I. a	–				
	I. b					
	II. a	+	+			
	II. b	o	o		+	
random	I. a		–			
	I. b					
	II. a	–	–			
	II. b	o/–	–			
hybrid	I. a	o	+			
	I. b	o	+			
	II. a	+	+			+
	II. b	o	o		o/–	

For the evaluation of the performance of the *decision-makers*, the effectiveness is of primary interest. As a further assessment of the successful runs, safety and efficiency are taken into account. Additionally, the paths of the failed runs are analyzed. An overview of the *merged* performance results of the decision-makers is provided in Table 5.1, where the efficiency of effective and at the same time safe behavior is rated, which is why only successful runs without a dangerous situation are counted. In the following, however, the performance criteria and the results are presented *individually*.
In the base experiment, the weighting method was effective in three scenarios (Pillars, Wall, House2). The hybrid method was effective in both basic scenarios. The remaining decision-makers were effective in only one of these two scenarios. The same pattern emerges with altered parameters: the weighting method is effective in the Pillars, Wall, and House1 worlds; just like the hybrid method (however, weighting always succeeds and hybrid only 9 times in House1). The rest is effective in at most both of the basic scenarios. Overall, the weighting method is effective in most scenarios, and thus is the most flexible decision-maker. Moreover, it is the most time-efficient one. At the same time, it is risky to use it, as all failed runs are due to a collision. Also, the number of runs with a dangerous distance to obstacles is comparably high. A safer alternative is the hybrid method, which is similarly flexible with a suitable parametrization, but stops the robot in dangerous situations.
These results are confirmed by the comparison experiment. The ε_d-constraint method is the fastest one. However, the hybrid method proves to be the most flexible and safest one, especially with a behavior which is neither risk-averse nor risk-affine (see Table 5.1). It navigates the robot to the goal in the basic scenarios and the House2 (moderate parameters) or House1 world (risk-averse parameters). Since the hybrid method has a constraint on danger but then selects the individual with the minimum weighted sum, it depends on the constraint, if the paths look similar to the weighting ones. Thus, with a high ε_d value, the travel time is marginally larger. With a high w_d and low ε_d value, the hybrid method is particularly risk-averse, which could be seen for example in the last parameter variation in House1, when the robot mostly took the route around the table. In such cases, the hybrid method is significantly slower.

Relative to the base experiment, the overall results of the comparison experiment are better in terms of the effectiveness, safety and efficiency. This shows, absolute *normalization* delimits the potential of the algorithm. A low ε_d value restricts the robot's freedom of movement to such an extent that in some sce-

narios the robot does not even start moving. Only at a value greater than or equal to κ (the limit value of the value ranges of distance- and time-based danger) the robot can navigate reasonably freely. However, it then moves dangerously close to the obstacles.
With relative normalization, due to the dynamic calculation of the danger values based on the current distance to the environment, the minimum allowed distance is adaptive, maintaining the range of motion for most situations. Moreover, without explicitly using a dynamic weighting, the interest value is more decisive for longer distances and the danger value for shorter distances. The great disadvantage of the relative normalization is its comparably unintuitive parameterization.

6 Conclusion and Future Work

As a foundation to reduce the risk of deadlocks when implementing coordinated motion with the Driving Swarm [32] in environments with large obstacles, this work adapts the multi-objective (MO) context steering approach [13] to the requirements of the differential-drive robot 'Turtlebot3 Burger' [41].
Due to the special kinematics, the decision space was modified. Instead of deciding on the direction of motion, the trajectory for the next time step is decided. Moreover, novel objective functions for interest and danger were introduced, adjusted to the robot's kinematics and perception. As it perceives its environment via a laser scanner, the danger value of a trajectory is evaluated based on the shortest distance between the trajectory and the surroundings. For trajectories that are predicted to collide with an obstacle, the danger value is assessed by the remaining time until the collision. The interest value considers the distance to the target and the alignment of the robot's future direction at the end of a trajectory with the target direction.
The performance of this approach was evaluated through simulation experiments, comparing five different multi-objective decision-making methods (MOO solvers) in five scenarios of increasing complexity using four different parameter configurations. Thereby, it was demonstrated that MO context steering applied to differential-drive robots can be successful in environments where classic steering [39, 40] fails. The following findings have been obtained:

Q1: For the navigation algorithm it is most simple to navigate around smaller round obstacles than around large wide ones. With the best parameter configuration for each decision-maker, the robot achieves the goal in 100 percent of the cases. L-shaped obstacles constitute the greatest difficulty, as the success rate of 0 percent verifies. As soon as the robot faces the inner part of the tip, it drives straight towards it and collides or gets into a deadlock.

Q2: As a composition of the elements described above, the robot can navigate in realistic environments with a success rate of up to 100 percent, depending on the decision-maker and the parameter configuration. The average across

all decision-makers (with the best parameter configuration for each) is 60 and 76 percent, for two different start and goal positions. Most collisions and deadlocks occur in one of the room corners, confirming corners are most difficult to overcome.
From the fact that all decision-makers regardless of the parametrization fail in the described corner situations, it can be concluded that the danger map itself cannot always sufficiently represent the real danger. The context mapping methodology must maintain a structured objective space in all situations, representing the objective of each behavior (e.g. seek, flee). This means, the available information of the environment need to be encoded such that each option has an (almost) unique position in the objective space. Otherwise, the robot cannot differentiate between the options and the decision is made uninformed. One strategy to detect such situations is to execute the Seek and Flee behavior one by one (e.g.$w_d = \{0, 1\}$) and check if the robot behaves as intended - if not, then the respective context mapping (objective function) needs to be adjusted.
Nevertheless, there are always more complex cases conceivable, where this reactive navigation algorithms reaches its limits and fails to lead the robot out. In such cases, a specific sequence of actions is necessary, requiring a memory about the performed actions.

Q3: The experiments were conducted using either the weighting, the ε_i-constraint, the ε_d-constraint, the random, or the hybrid decision-making method. Moreover, two normalization variants of the distance-based danger map were contrasted: the absolute normalization to the maximum possible distance (the perceptual limit) and the relative normalization to the current maximum value of the distances between trajectories and environment.
As complementary experiments with altered parameter values have shown, the scenario-specific performance strongly relies on the chosen parameter set (see also Q5). Nevertheless, certain patterns can be discerned in the results: Using the absolute normalization (rigid danger mapping), the weighting method is the most time efficient and flexible one. However, besides the ε_i-method, this is also the most dangerous one, since it lets the robot collide instead of stopping it when the danger is too high, as the danger-constrained methods do. The safest yet comparably performant method is the hybrid method.
With the relative normalization (adaptive danger mapping), the overall performance is considerably better. While the ε_d-constraint method is the fastest one, the most flexible MOO methods are the weighting and the hybrid method.

Taking additionally safety into account, the hybrid method is the best option. In every scenario, the random method has the longest travel time. With a lower ε_d, it performed quite effectively. As this method chooses randomly from the set of admissible solutions, it becomes less effective with a higher ε_d (which increases the number of admissible trajectories). As a lower baseline, it proves the other decision-making methods are making better decisions based on the available information.

Q4: The risk-aversion of the robot depends on a large set of mostly inter-dependent parameters. With a set of constant context-mapping parameter values (described below), only altering the decision-making parameters danger weight w_d, interest constraint ε_i, and danger constraint ε_d affects the distance the robot keeps to the surroundings. With a higher weight on danger, a lower minimum interest, and a lower maximum danger, the robot behaves more risk-averse, i.e. the chosen route is further away from obstacles, but also longer and slower. However, in narrow environments, with a low maximum danger and the use of rigid danger mapping, the robot did not move at all.
As stated above, the risk-aversion depends not only on the decision-making parameters, but also on: the environment, the value ranges of the danger map split by κ, the length of the trajectories determined by τ_d and τ_i, and the mapping function of the context values (the function itself, its parameters and the normalization variant). By implementing the navigation task as a 2D multi-objective problem, the decision and objective space is completely transparent, making the decision-making process easier to control (especially compared to DWA). Thereby, it was detected that all decision-making methods are sensitive to the structure and value ranges of the danger and interest values in the objective-space. On the one hand, the parameter values have to be selected carefully, on the other hand, the sensitivity can be exploited by using a non-linear mapping function. Because in both the absolute and relative mapping, the slope of the function curve determines the risk-aversion. Through this, a risk-averse behavior near obstacles and risk-taking behavior at greater distances can be produced.

The results of this work have shown that the developed algorithm is more successful than classical context steering, but deadlocks still occur, especially in corner situations. Regarding this, more findings have been obtained during the experiments, that are important for further research:
Contrary to initial assumptions, the inclusion of zero in the value range of translational velocity did not cause the robot to escape from deadlocks by

turning on the spot. As it does not bring the robot closer to the target, its interest value is low, which is why this solution is dominated when determining the Pareto front.
In the presented methodology, the distance-based danger is established on the shortest distance of the trajectory to the environment (represented by the polygon). This allows a check for collisions along the entire trajectory. However, this is done by considering the whole environment, not only the part of the local environment the trajectory is located. Additionally, it is not controlled where (at which point in the trajectory) and in which direction the shortest distance is. As a result, it is unknown whether the trajectory steers the robot towards the obstacle or away from it. Different situations can lead to the same danger value, not representing the reality. The danger objective is currently defined to keep distance from obstacles. Considering the general direction of the trajectory, may let the robot move *away* from obstacles. As a possible implementation, around each trajectory, a segment could be created in which the minimum distance is calculated. Alternatively, in addition to the method presented in this work, the distance to obstacles in the direction of the trajectory's endpoint (the orientation of the robot's final state) could be taken into account. Moreover, it could be considered at which point in the trajectory the shortest distance was measured. Then the evaluation of the danger can be made in the same way as with the time-based danger map.
Yet, overall, the algorithm introduced in this thesis already provides a good basis for a better behavior of the Driving Swarm in environments with large obstacles.

Future Work

The work has shown that the parameters are interdependent and can be optimized scenario-specifically or -universally. However, a full parameter variation could help to reveal exact dynamics in the parameter structure and show how to produce even better behavior, especially with the hybrid method. Alternatively, the parameters can be optimized using metaheuristic methods like evolutionary algorithms, as shown in [14].
In the experiments conducted in this work, the goal position was known to the robot. A task in an open scenario with unknown goal position(s) (or objects of interest, as in [13]) located within the local perception range would thus be a good addition to further evaluate the algorithm and decision-makers.

To further improve the behavior, especially to reduce oscillating in situations that are difficult to decide because of many equally good options, post-processing methods like history blending [13] can be incorporated. It creates a direction bias, by considering the previously chosen direction. For a differential-drive robot, it could be adapted by choosing a velocity close to the previous one.
This work is meant to provide a baseline for later implementation with a swarm. For example, flocking (cohesion, alignment, separation) could to be incorporated by adding a third dimension to the multi-objective problem.
With regard to the application on real robots, the following approaches are available for improving the real-time capability and the robustness: Context interpolation [13] adapted to the modified decision space could enhance the algorithm performance. To avoid that the robot accomplishes the task but collides afterwards, it is recommended to also consider the behavior after reaching the goal. In the further development of the algorithm, when running the simulation, artificial errors can be added to sensors and actuators to evaluate the robustness. However, in order to make a clear statement about the performance of the algorithm in the real world, it has to be tested with the real robots of the Driving Swarm [32].

Bibliography

[1] Turtlebot 3. URL https://robots.ieee.org/robots/turtlebot3/.

[2] Ros 2 documentation: Foxy, . URL https://docs.ros.org/en/foxy/Releases.html.

[3] Gazebo, . URL https://gazebosim.org/about.

[4] Overview and usage of rqt, . URL https://docs.ros.org/en/foxy/Concepts/About-RQt.html.

[5] Ros 2 - data display with rviz2. URL https://www.stereolabs.com/docs/ros2/rviz2/.

[6] Robotis turtlebot3 burger. URL https://cdn-reichelt.de/bilder/web/xxl_ws/X200/TURTLEBOT3_DIMENSION1.png.

[7] Recording and playing back data, 2018. URL https://docs.ros.org/en/foxy/Tutorials/Ros2bag/Recording-And-Playing-Back-Data.html.

[8] rclpy, 2019. URL https://docs.ros2.org/latest/api/rclpy/about.html.

[9] T. Bösser. Autonomous agents. In *International Encyclopedia of the Social & Behavioral Sciences*, pages 1002–1006. Elsevier, 2001. ISBN 9780080430768.

[10] Valentino Braitenberg. *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.

[11] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane,

Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[12] Vincenzo DiLuoffo, William R. Michalson, and Berk Sunar. Robot operating system 2. *International Journal of Advanced Robotic Systems*, 15 (3):172988141877001, 2018.

[13] Alexander Dockhorn, Sanaz Mostaghim, Martin Kirst, and Martin Zettwitz. Multi-objective optimization and decision-making in context steering. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021. ISBN 978-1-6654-3886-5.

[14] Alexander Dockhorn, Martin Kirst, Sanaz Mostaghim, Martin Wieczorek, and Heiner Zille. Evolutionary algorithm for parameter optimization of context steering agents. *IEEE Transactions on Games*, (v):1, 2022.

[15] Gregory Dudek and Michael Jenkin. *Computational principles of mobile robotics*. Cambridge university press, 2010.

[16] F. Loizides, B. Schmidt, Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing, editors. *Jupyter Notebooks – a publishing format for reproducible computational workflows*, 2016.

[17] Dieter. Fox, Wolfram. Burgard, and Sebastian. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[18] Andrew Fray. Steering behaviours are doing it wrong, 2013. URL `https://andrewfray.wordpress.com/2013/02/20/steering-behaviours-are-doing-it-wrong/`.

[19] Andrew Fray. Context steering: Behavior-driven steering at the macro scale. In *Game AI Pro 360*, pages 147–158. CRC Press, 2019.

[20] David Gossow, Chad Rockey, Kei Okada, Julius Kammerl, Acorn Pooley, and Rein Appeldoorn. Rviz, 2021. URL `https://github.com/ros2/rviz`.

[21] Christian Henkel, Alexander Bubeck, and Weiliang Xu. Energy efficient dynamic window approach for local path planning in mobile service robotics. *IFAC-PapersOnLine*, 49(15):32–37, 2016.

[22] Richard D. Hipp. SQLite, 2020. URL `https://www.sqlite.org/index.html`.

[23] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[24] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, pages 2149–2154. IEEE, 28 Sept.-2 Oct., 2004. ISBN 0-7803-8463-6.

[25] Anis Koubâa, editor. *Robot Operating System (ROS): The Complete Reference (Volume 1)*, volume 625 of *Springer eBook Collection Engineering*. Springer, Cham, 1st ed. 2016 edition, 2016. ISBN 978-3-319-26054-9.

[26] Anis Koubâa, editor. *Robot Operating System (ROS): The Complete Reference (Volume 2)*, volume 707 of *Springer eBook Collection Engineering*. Springer, Cham, 2017. ISBN 978-3-319-54927-9.

[27] Anis Koubâa, editor. *Robot Operating System (ROS): The Complete Reference (Volume 3)*, volume 778 of *SpringerLink Bücher*. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91590-6.

[28] Franz J. Kurfess. Artificial intelligence. In *Encyclopedia of Physical Science and Technology*, pages 609–629. Elsevier, 2003. ISBN 9780122274107.

[29] Huseyin Kutluca. Robot operating system 2 (ros 2) architecture, 2020. URL `https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776`.

[30] Steve Macenski and Ruffin White. Navigation 2, 2020. URL `https://navigation.ros.org/index.html#`.

[31] Steve Macenski, Francisco Martin, Ruffin White, and Jonatan Gines Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2718–2725. IEEE, 24.10.2020 - 24.01.2021. ISBN 978-1-7281-6212-6.

[32] Sebastian Mai, Nele Traichel, and Sanaz Mostaghim. Driving swarm: A swarm robotics framework for intelligent navigation in a self-organised world. In *Accepted at: 2022 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2022.

[33] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56, 2010.

[34] Michael Waskom, Olga Botvinnik, Drew O'Kane, Paul Hobson, Saulius Lukauskas, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi Warmenhoven, Julian de Ruiter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba, Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Alistair Miles, Yoav Ram, Tal Yarkoni, Mike Lee Williams, Constantine Evans, Clark Fitzgerald, Brian, Chris Fonnesbeck, Antony Lee, and Adel Qalieh. mwaskom/seaborn: v0.8.1 (september 2017), 2017. URL `https://doi.org/10.5281/zenodo.883859`.

[35] Sanaz Mostaghim. Intelligente systeme: Kapitel 1: Agenten systeme, 2018.

[36] Sanaz Mostaghim. Swarm intelligence: Multi-objective problems (mops), 27.11.2019.

[37] Mohamed Oubbati. Einführung in die robotik: Differentialsantrieb, 2012. URL `https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.130/Mitarbeiter/oubbati/RobotikWS1113/Folien/Differentialantrieb.pdf`.

[38] B. K. Patle, Ganesh Babu L, Anish Pandey, D.R.K. Parhi, and A. Jagadeesh. A review: On path planning strategies for navigation of mobile robot. *Defence Technology*, 15(4):582–606, 2019.

[39] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In Maureen C. Stone, editor, *Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87*, pages 25–34. ACM Press, 1987. ISBN 0897912276.

[40] Craig W. Reynolds et al. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999.

[41] Robotis. Turtlebot3. URL `https://emanual.robotis.com/docs/en/platform/turtlebot3/features/`.

[42] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. A review of mobile robots: Concepts, methods, theoretical framework, and applications. *International Journal of Advanced Robotic Systems*, 16(2): 172988141983959, 2019.

[43] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm robotic behaviors and current applications. *Frontiers in robotics and AI*, 7:36, 2020.

[44] Sean Gillies et al. Shapely: manipulation and analysis of geometric objects, 2007.

[45] Daniel Shiffman. *The nature of code: Simulating natural systems with processing*. Selbstverl., version 1.0, generated december 6, 2012 edition, 2012. ISBN 978-0985930806.

[46] Gerald Steinbauer. Mobile robots: Reactive navigation. URL `http://www.ist.tugraz.at/steinbauer_mediawiki/images/a/a2/Mr_12_reactive_navigation.pdf`.

[47] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

[48] Sebastian Thrun and John J. Leonard. Simultaneous localization and mapping. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 871–889. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-30301-5.

[49] Cristina Urdiales, Antonio Bandera, Eduardo Pérez, Alberto Poncela, and Francisco Sandoval. Hierarchical planning in a mobile robot for map learning and navigation. In Janusz Kacprzyk, Changjiu Zhou, Darío Maravall, and Da Ruan, editors, *Autonomous Robotic Systems*, volume 116 of *Studies in Fuzziness and Soft Computing*, pages 165–188. Physica-Verlag HD, 2003. ISBN 978-3-7908-2523-7.

[50] Guido van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[51] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, War-

ren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, Polat, VanderPlas, Jake, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. Scipy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17:261–272, 2020.

[52] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.

[53] Pyo YoonSeok, HanCheol Cho, RyuWoon Jung, and TaeHoon Lim. *ROS Robot Programming: From the basic concept to practical programming and robot application.* Seoul, Republic of Korea, 2017. ISBN 979-11-962307-1-5.

[54] Han-ye Zhang, Wei-ming Lin, and Ai-xia Chen. Path planning for the mobile robot: A review. *Symmetry*, 10(10):450, 2018.

Declaration of Authorship

I hereby declare that this thesis was created by me and me alone using only the stated sources and tools.

Nele Raya Traichel

Magdeburg, June 28, 2022