13

Reliable Long-Term Navigation in Indoor Environments

Mattias Wahde, David Sandberg and Krister Wolff
Department of Applied Mechanics, Chalmers University of Technology, Göteborg
Sweden

1. Introduction

Long-term robustness is a crucial property of robots intended for real-world tasks such as, for example, transportation in indoor environments (e.g. warehouses, industries, hospitals, airports etc.). In order to be useful, such robots must be able to operate over long distances without much human supervision, something that sets high demands both on the actual robot (hardware) and its artificial brain[1] (software). This paper is focused on the development of artificial robotic brains for reliable long-term navigation (for use in, for example, transportation) in indoor environments.

Reliable decision-making is an essential tool for achieving long-term robustness. The ability to make correct decisions, in real-time and often based on incomplete and noisy information, is important not only for mobile robots but also for animals, including humans (McFarland, 1998; Prescott et al., 2007). One may argue that the entire sub-field of behavior-based robotics emerged, at least in part, as a result of the perceived failure of classical artificial intelligence to address real-time decision-making based on a robot's imperfect knowledge of the world. Starting with the *subsumption method* (Brooks, 1986), many different methods for decision-making, often referred to as methods for *behavior selection* or *action selection*, have been suggested in the literature on behavior-based robotics (see, for example, Bryson (2007); Pirjanian (1999) for reviews of such methods).

In actual applications, a common approach is to combine a *reactive layer* of decision-making, using mainly behavior-based concepts, with a *deliberative layer* using, for example, concepts from classical artificial intelligence, such as high-level reasoning) (Arkin, 1998). Several approaches of this kind have been suggested (see, for example, Arkin (1987) and Gat (1991)) and applied in different robots (see, for example, Sakagami et al. (2002)).

In the *utility function* (UF) method for decision-making (Wahde, 2003; 2009), which will be used here, a somewhat different approach is taken in which, for the purposes of decision-making, no distinction is made between reactive and deliberative aspects of the robotic brain. In this method, an artificial robotic brain is built from a repertoire of *brain processes* as well as a single decision-making system responsible for activating and de-activating brain processes

[1] The term *artificial (robotic) brain* is here used instead of the more common *control system* since the latter term, in the authors' view, signifies the low-level parts (such as motor control to achieve a certain speed) of robot intelligence, whereas more high-level parts, such as decision-making are better described by the term used here.

(several of which may run in parallel). Two kinds of brain processes are defined, namely *(motor) behaviors*, which make use of the robot's motors and *cognitive processes*, which do not. Both kinds of processes may include both reactive and deliberative aspects. Note also that, regardless of content, the brain processes used in the UF method are all written in a unified manner, described in Sect. 4 below.

As its name implies, the UF method, described in Sect. 3 below, is based on the concept of utility, formalized by von Neumann & Morgenstern (1944). The method is mainly intended for complex motor tasks (e.g. transportation of objects in arenas with moving obstacles) requiring both reactivity and deliberation. While the method has already been tested (both in simulation and in real robots) for applications involving navigation over distances of 10-20 m or less (Wahde, 2009), this paper will present a more challenging test of the method, involving operation over long distances. The task considered here will be a delivery task, in which a robot moves to a sequence of target points in a large arena containing both stationary and moving obstacles. The approach considered here involves long-term simulations, preceded by extensive validation of the simulator, using a real robot.

2. Robot

The differentially steered, two-wheeled robot (developed in the authors' group) used in this investigation is shown in Fig. 1. The robot has a circular shape with a radius of 0.20 m. Note that an indentation has been made for the wheels. The height (from the ground to the top of the laser range finder (LRF)) is around 0.84 m. The weight is around 14.5 kg. The robot's two actuators are Faulhaber 3863A024C DC motors, each with a Faulhaber 38/1S-14:1 planetary gearhead. Each motor is controlled by a Parallax HB-25 motor controller. The robot is equipped with two 8 Ah power sources, with voltages of 12 V and 7.2 V, respectively.

In addition to the two drive wheels, the robot has two Castor wheels, one in the back and one in the front. The front Castor wheel is equipped with a suspension system, in order to avoid situations where the drive wheels lose contact with the ground, due to bumps in the surface on which the robot moves. The wheels are perforated by 24 holes which, together with a Boe-Bot Digital Encoder Kit detector on each wheel, are used for measuring the rotation of the wheels for use in odometry. In addition, the robot is equipped with a Hokuyo URG-04LX LRF with a range of 4 m and a 240 degree sweep angle. For proximity detection, the robot has two forward-pointing Sharp GP2D12 IR sensors (oriented towards ± 30 degrees, respectively, from the robot's front direction), and one IR sensor, of the same type, pointing straight backwards. The robot is equipped with two Basic Stamp II microcontrollers (which, although slow, are sufficient for the task considered here), one that reads the wheel encoder signals, and one that (i) sends signals to the motor controllers and (ii) receives signals from the three IR sensors. The two Basic Stamps are, in turn, connected via a USB hub to a laptop that can be placed on top of the robot (under the beam that holds the LRF). The LRF is also connected to the laptop, via the same USB hub.

3. Decision-making structure

The decision-making structure is based on the UF method, the most recent version of which is described by Wahde (2009). The method is mainly intended for use in tasks such as navigation, transportation, or mapping. In this method, the artificial brain is built from (i) a set of brain processes and (ii) a decision-making system based on the concept of utility, allowing the robot

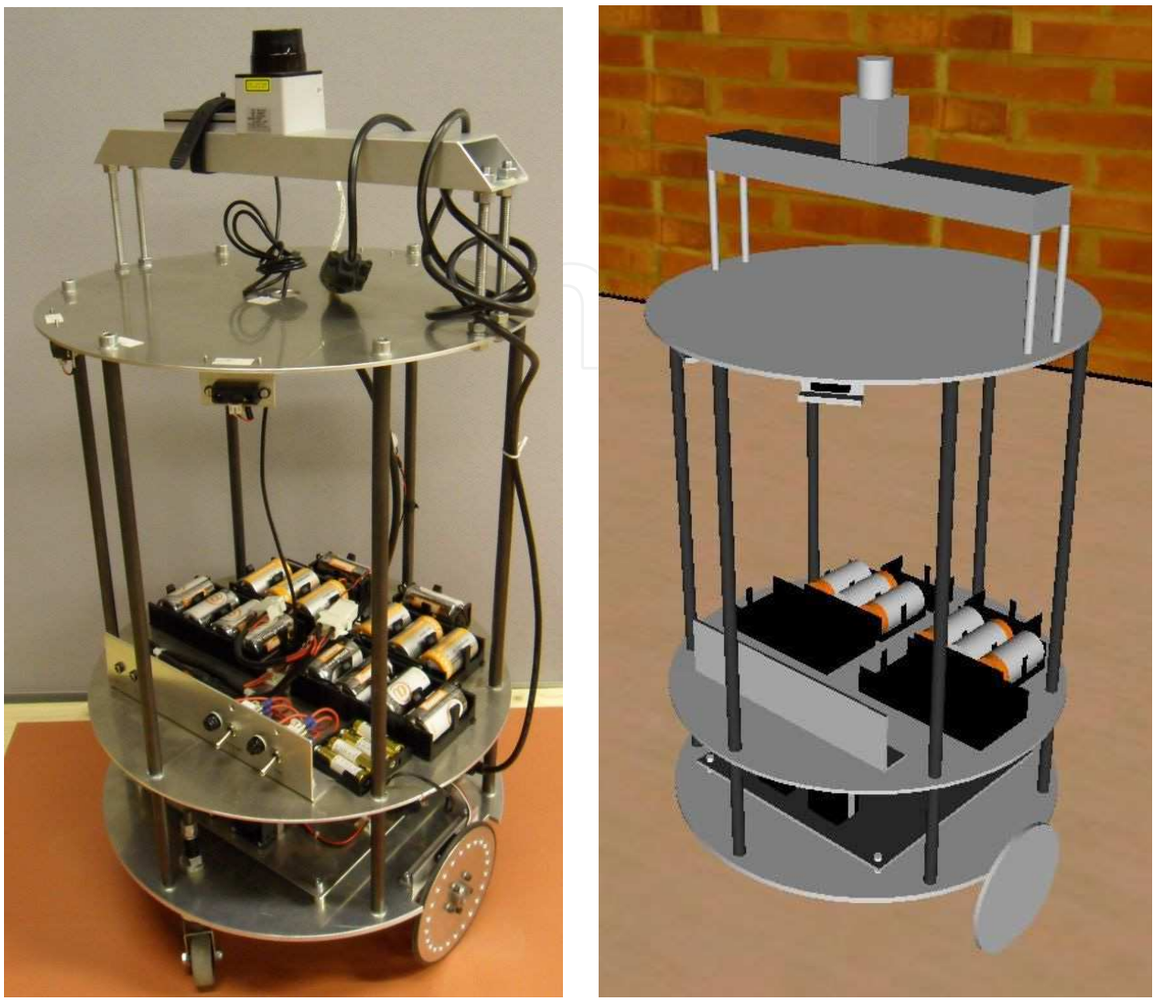


Fig. 1. Left panel: The differentially steered robot considered here. The laser range finder is mounted on top. Right panel: The simulated version of the robot.

to activate and de-activate the various brain processes. Here, only a brief description will be given. For a more detailed description, see Wahde (2009).

Brain processes are divided into several types. The two categories used here are *cognitive processes* (that do not make use of the robot's motors) and *motor behaviors* that *do* make use of the motors. In the UF method, any number of cognitive process can be active simultaneously, whereas exactly one motor behavior is active at any time, the rationale being that a robot of the kind considered here (with two wheels as its only actuators) can only carry out one motor action at any given time.

Each brain process (regardless of its type) is associated with a *utility function* whose task it is to determine the merit of running the brain process, in any given situation. The utility functions, in turn, depend on scalar *state variables*. A simple example of a state variable z may be the reading of an IR sensor mounted at the front of a robot. The value of z is an (admittedly incomplete) measure of the risks involved in moving forward. A more complex example is the *moving obstacle danger level* (see Subsect. 4.4 below), which uses consecutive LRF scans to obtain a scalar value measuring the risk of collision with moving obstacles.

State variable values are measured continuously, but asynchronously (since not all sensors are updated with the same frequency), and the most recent values available are used as inputs to the utility functions from which the utility value u_i for brain process i is obtained using the equation

$$\tau_i \dot{u}_i + u_i = \sigma_i \left(\sum_{k=1}^{m} a_{ik} z_k + b_i + \Gamma_i \right), \; i = 1, \ldots, n, \tag{1}$$

where n is the number of brain processes, τ_i are time constants determining the reaction time of the robot (typically set to around 0.1 s), m is the number of state variables (denoted z_k), and a_{ik} and b_i are tunable parameters. The squashing function $\sigma_i(x)$ is taken as $\tanh(c_i x)$ where c_i is a positive constant (typically set to 1). Thus, the squashing functions σ_i serve to keep the utility values in the range $[-1, 1]$.

Once the values of the utility functions have been computed, decision-making is simple in the UF method, and works as follows: (i) The motor behavior with largest utility (among all motor behaviors) is active and (ii) any cognitive process with positive utility is active. All other brain processes are inactive.

The parameters Γ_i (referred to as *gamma parameters*), which are normally equal to zero, allow direct activation or de-activation of a brain process. Ideally, the state variables z_k should provide the robot will the information needed to make an informed decision regarding which brain processes to use in any situation encountered. However, in practice, the parameters Γ_i are needed in certain situations. Consider, for example, a simple case in which the utility function for an obstacle avoidance behavior depends on a single state variable z (multiplied by a positive constant a) that measures obstacle proximity (using, for example, IR or sonar sensors). Now, if the obstacle avoidance behavior is activated, the robot's first action is commonly to turn away from the obstacle. When this happens, the value z will then drop, so that u also drops, albeit with a slight delay depending on the value of the time constant in Eq. (1). At this point, the obstacle avoidance behavior is likely to be de-activated again, before actually having properly avoided the obstacle. The gamma parameters, which can be set directly by the brain processes, have been introduced to prevent such problems. Thus, for example, when the obstacle avoidance behavior is activated, it can set its *own* gamma parameter to a positive value, thus normally avoiding de-activation when the state variable z drops as described above. Whenever the obstacle avoidance behavior has run its course, it can set the gamma parameter to a large negative value, thus effectively de-activating itself. Note that the decision-making system is active continuously, so that, in the example above, obstacle avoidance *can* be de-activated (even after raising its own gamma parameter) should another brain process reach an even higher utility value. Once a gamma parameter has been set to any value (positive or negative) other than zero, its magnitude falls of exponentially with time, with a time constant (τ_i^{Γ}) specific to the brain process at hand. This time constant typically takes a larger value than the time constant in the utility function.

As mentioned above, the UF method allows several (cognitive) brain processes to run in parallel with the (single) active motor behavior. Thus, for example, while moving (using a motor behavior), a robot using the UF method would simultaneously be able to run two cognitive processes, one for generating odometric pose estimates from encoder readings and one for processing laser scans in order to recalibrate its odometric readings, if needed.

As is evident from Eq. (1), for a given set of state variables, the actual decisions taken by the robot will depend on the parameters τ_i, a_{ij}, b_i, and c_i as well as the values of the gamma parameters (if used). Thus, in order to make a robot carry out a specific task correctly, the

user must set these parameters to appropriate values. Note that, for many cognitive processes (e.g. odometry, which normally should be running continuously), all that is needed is for the utility values to be positive at all times, which can easily be arranged by setting all the a-parameters for that process to zero, and the b-parameter to any positive value.
In many cases, the remaining parameters (especially those pertaining to the motor behaviors) can be set by hand using trial-and-error. In more complex situations, one can use a stochastic optimization method such as, for example, a genetic algorithm or particle swarm optimization to find appropriate parameters values. In those situations, one must first define a suitable objective function, for example the distance travelled in a given amount of time, subject to the condition that collisions should be avoided. Optimization runs of this kind are normally carried out in the simulator (described in Subsect. 5.1 below) rather than a real robot, since the number of evaluations needed during optimization can become quite large.

4. Brain processes

Just as the decision-making system, brain processes used in the UF method also have a unified structure; the main part of each brain process is a finite-state machine (FSM) consisting of *states* (in which various computations are carried out and, in the case of motor behaviors, actions are taken) and *conditional transitions* between states. In any brain process, the FSM executes the *current state* until a transition condition forces it to jump to another state, which is then executed etc. An illustration of a brain process FSM is shown in Fig. 8 below.
For the task considered here, namely reliable long-term navigation, six brain processes have been used, namely *Grid navigation* (denoted B_1), *Odometry* (B_2), *Odometric calibration* (B_3), *Moving obstacle detection* (B_4), *Moving obstacle avoidance* (B_5), and *Long-term memory* (B_6). The six brain processes will be described next.

4.1 Grid navigation

The navigation method used here is contained in a brain process called *Grid navigation*. As the name implies, the navigation method (which is intended for indoor, planar environments) relies on a grid. Dividing a given arena into a set of convex cells is a common problem in robotics, and it is typically approached using *Voronoi diagrams* (or their dual, *Delaunay triangulation*) (Okabe et al., 2000) or *Meadow maps* (Singh & Agarwal, 2010). However, these tessellations tend to generate jagged paths, with unnecessary turns that increase the length of the robot's path (a problem that can be overcome using *path relaxation*; see Thorpe (1984)). Nevertheless, in this paper, a different method for automatic grid generation will be used, in which the map of the arena is first contracted to generate a margin for the robot's movement. Next, the arena is divided into convex cells in a preprocessing step similar to that described by Singh & Wagh (1987). Note, however, that the method described in this paper also can handle non-rectangular arenas (and obstacles). Finally, the cells are joined in order to generate larger convex cells in which the robot can move freely. Once such a grid is available, a path is generated using Dijkstra's algorithm (see Sect. 4.1.2 below), optimizing the placement of waypoints (on the edges of the cells) in order to minimize the path length. The method for grid generation, which is summarized in Table 1, will now be described.

4.1.1 Grid generation method

It is assumed that a map is available, in the form of a set of closed *map curves* consisting, in turn, of a sequence of connected (i.e. sharing a common point) *map curve segments*, each

Step 1 Generate the contracted map.
Step 2 Generate the preliminary map curve intersection grid.
Step 3 Process non-axis parallel map curve segments.
Step 4 Generate the map curve intersection grid by removing cells outside the grid.
Step 5 Generate the convex navigation grid by joining cells.

Table 1. The algorithm for generating the convex navigation grid. See the main text for a detailed description of the various steps.

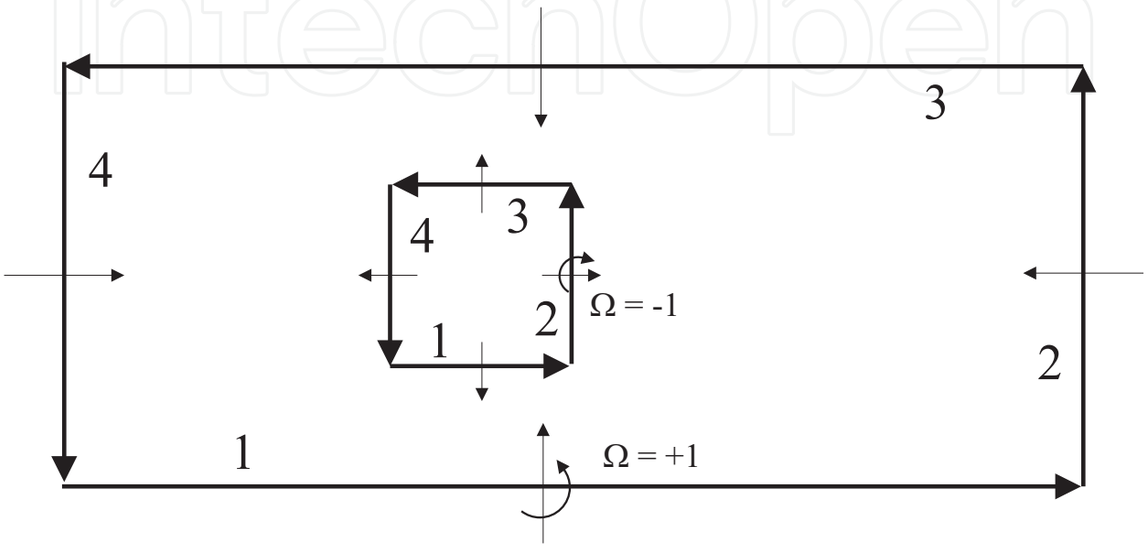


Fig. 2. The definition of in-directions in a rectangular arena with a square obstacle. In this simple arena, there are two map curves, each with four enumerated map curve segments. The thick arrow on each map curve segment points from the starting point to the end point of the segment. The thin arrows point towards the in-direction. The corresponding value of Ω (see Eq. (2)) is equal to $+1$ for the outer map curve (wall) and -1 for the inner map curve (obstacle). Note that the sign of Ω depends on the (arbitrary) choice of enumeration (clockwise or counterclockwise) for the map curve segments.

defined by a starting point $\mathbf{P}_a = (x_a, y_a)$ and an end point $\mathbf{P}_b = (x_b, y_b)$. Thus, each map curve is a simple (i.e. without self-intersections) polygon. In addition, for each map curve segment, an *in-direction* is defined. The in-direction Ω is given by the sign (either -1 or 1) of the z-component of the cross product between the vector $\mathbf{p} = \mathbf{P}_b - \mathbf{P}_a$ and a unit vector $\mathbf{d}$ (orthogonal to $\mathbf{p}$) pointing towards the side of the map curve segment which is accessible to the robot. Thus,

$$\Omega = \mathrm{sgn}\left((\mathbf{p} \times \mathbf{d}) \cdot \hat{z}\right). \tag{2}$$

The in-direction concept is illustrated in Fig. 2. Note that, for any given map curve, all segments will have the same in-direction.

The method divides the accessible area of a robot's environment into a set of convex grid cells. Due to the convexity of the grid cells the robot can move freely within any grid cell, a property that facilitates path optimization, as described below. The grid generation method operates as follows: First (Step 1 in the algorithm described in Table 1), since the robot has a certain size, the actual map is shrunk to generated a *contracted map*. The margin μ (a tunable parameter) used when generating the contracted map should be such that, if the robot is positioned on a

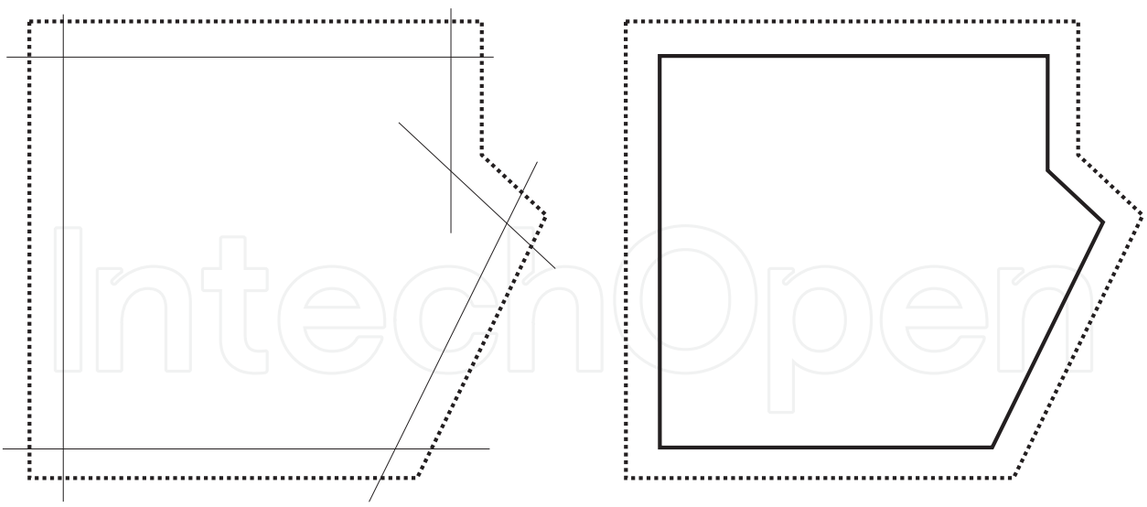


Fig. 3. The map contraction procedure. The left panel shows the original map (dotted line), along with the line segments obtained using the procedure described in the main text. In the right panel, the intersection points between those lines have been determined, resulting in the contracted map (solid line).

line in the contracted map, it should not touch any wall or other fixed obstacle. The procedure is illustrated in Fig. 3. The algorithm for generating the contracted map is straightforward: Consider a given line segment $\mathbf{p} = (p_x, p_y) = \mathbf{P}_b - \mathbf{P}_b = (x_b - x_a, y_b - y_a)$. Let $\mathbf{c}$ be a vector of length μ orthogonal to $\mathbf{p}$ and pointing towards the in-direction. Thus

$$\mathbf{c} = \mu\Omega\frac{(-p_y, p_x)}{\sqrt{p_x^2 + p_y^2}}. \tag{3}$$

It is now possible to obtain two points on the contracted map curve segment as

$$\mathbf{Q}_a = \mathbf{P}_a + \mathbf{c} \tag{4}$$

and

$$\mathbf{Q}_b = \mathbf{P}_b + \mathbf{c}. \tag{5}$$

Note, however, that these two points are normally not the end points of the contracted map curve segment. In order to determine the end points, one must first carry out the process described above for all map curve segments in a given map curve. Once that process has been completed, the end points of the contracted map curve segments can be determined as the intersection points between the lines passing through the point pairs generated during the contraction process just described. If these lines are extended to infinity, any given line may intersect many other lines. In that case, the points chosen as end points of the contracted map curve segments are taken as those intersection points that are nearest to the points $\mathbf{Q}_a$ and $\mathbf{Q}_b$, respectively. The process is illustrated in Fig. 3, where a map consisting of a single map curve, with six map curve segments, is contracted.

Note that, in its simplest form, the map contraction method requires that no contracted map *curve* should intersect any of the other contracted map curves. The cases in which this happens

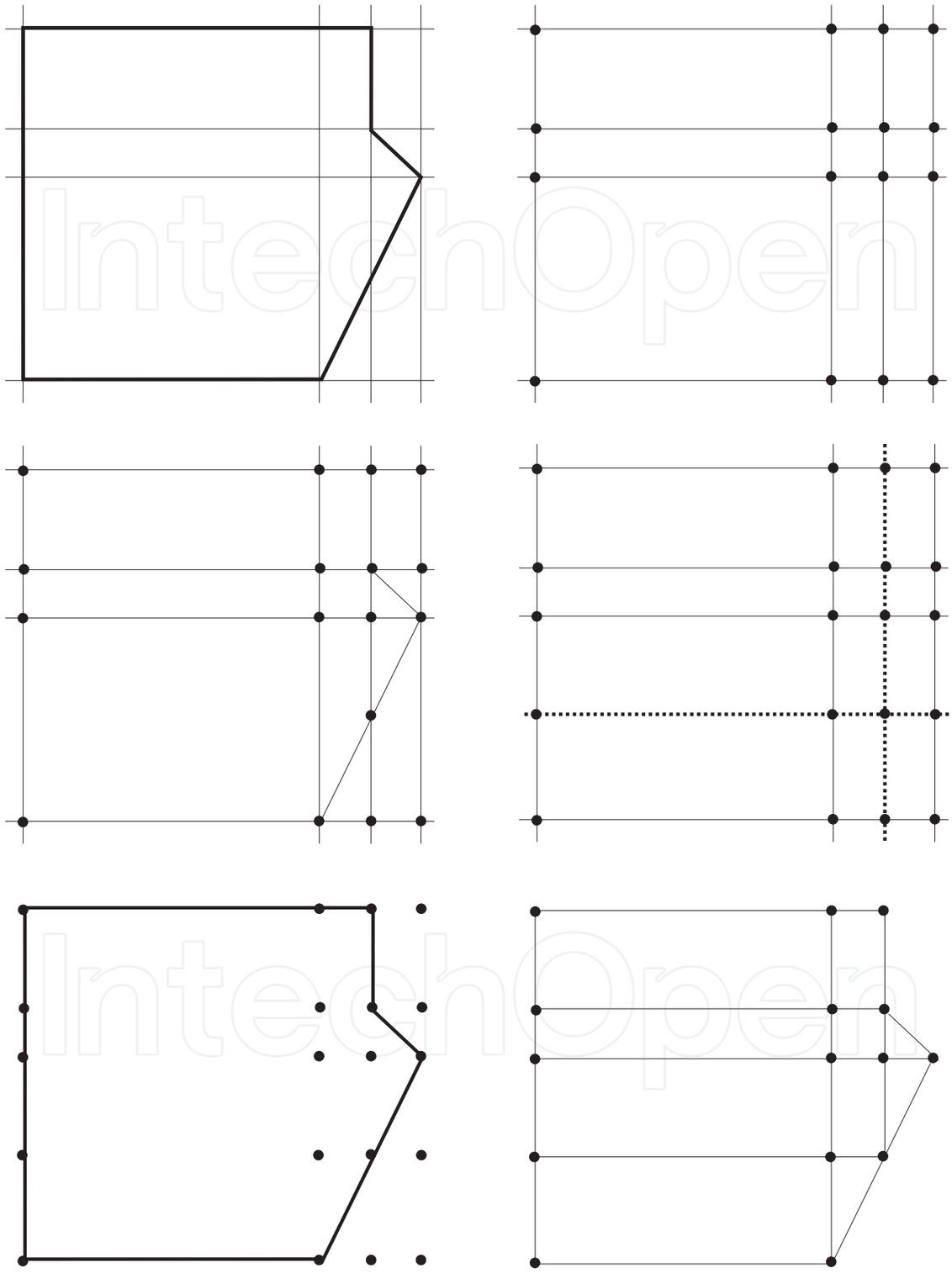


Fig. 4. The grid generation algorithm, applied to the contracted map from Fig. 3. See the main text for a detailed description of the various steps.

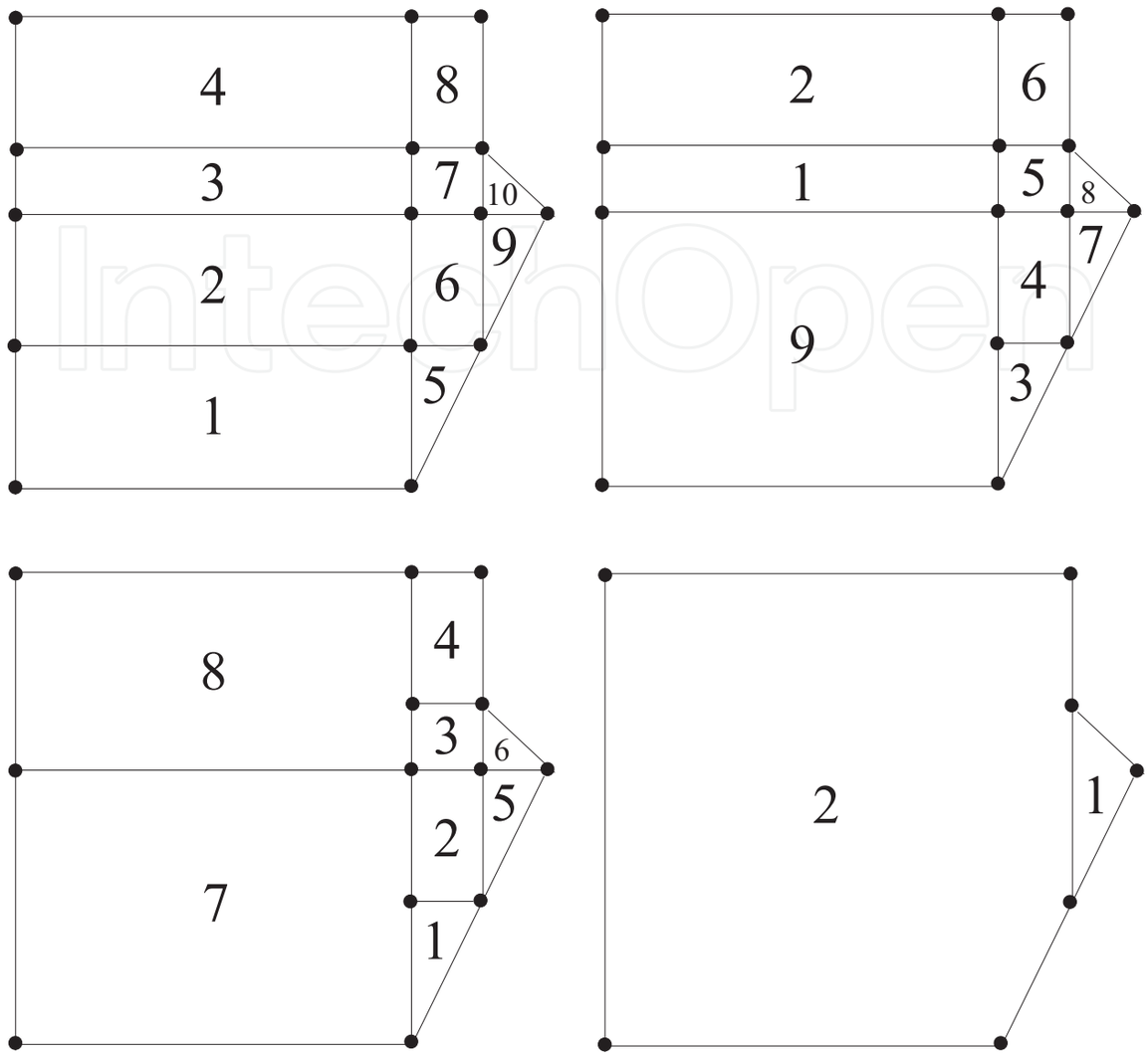


Fig. 5. The grid simplification algorithm, applied to the grid shown in Fig. 4. During grid simplification, polygonal cells are combined to form larger (but still convex) cells. Note that, in each step, the enumeration changes after the removal of the two cells that are joined. The upper left panel shows the initial grid, whereas the upper right panel shows the grid after the first step. In the bottom row, the left panel shows the grid after the second step, and the right panel shows the final result, after several steps.

can be handled, but the procedure for doing so will not be described here. Instead, we shall limit ourselves to considering maps in which the contracted map curves do not intersect.
Once the contracted map has been found, the algorithm (Step 2) proceeds to generate a temporary grid (referred to as the *preliminary map curve intersection grid*), consisting of the intersections between all axis-parallel (horizontal or vertical) map curve segments expanded to infinite length, as illustrated in the first row of Fig. 4. Next (Step 3), all non-axis parallel map curve segments are processed to check for intersections between those lines and the sides of the cells in the preliminary map curve intersection grid. For any intersection found, the

corresponding point is added to the grid, as illustrated in the middle row (left panel) of Fig. 4. Lines are then drawn horizontally and vertically from the newly added points, and the points corresponding to any intersections between those two lines and the cells of the preliminary map curve intersection grid are added as well; see the right panel in the middle row of Fig. 4. Some of the cells will be located outside the (contracted) map, as shown in the left panel in the bottom row of Fig. 4. Those cells are removed (Step 4), resulting in the *map curve intersection grid*, shown in the final panel of Fig. 4.

Note that the cells in the map curve intersection grid are convex by construction. In principle, one could just remove the inaccessible cells and use the remaining cells during navigation. However, in a realistic arena (more complex than the simple arena used for illustrating the procedure) the map curve intersection grid typically contains very many cells, thus slowing down the path generation. Furthermore, with a large number of cells, the paths generated will typically have an unwanted zig-zag nature. Thus, in Step 5 of the algorithm, cells in the map curve intersection grid are joined to form larger cells, with the condition that two cells are only joined if the result is again a convex polygon, something that can easily be checked since, in a convex polygon, all cross products between consecutive sides (i.e. sides that share a common point) have the same sign. In the current version of the algorithm, no attempt has been made to generate a minimal (in number) set of convex polygons. Instead, the cells are simply processed in order, removing joined cells and adding newly formed cells, until there are no cells left such that their combination would be convex. The first steps of this process, as well as the final result, are shown in Fig. 5. The result is referred to as the *convex navigation grid*.

4.1.2 Path planning

The path planning procedure uses the convex navigation grid (which can be generated once and for all for any given (fixed) arena). Given a starting point (i.e. the robot's estimate of its current location) and a desired goal point, a path is generated using Dijkstra's algorithm (Dijkstra, 1959), augmented with a process for selecting navigation waypoints on the edges of the grid cells through which the path passes. As in the standard Dijkstra algorithm, cells are considered in an expanding pattern from the starting cell (i.e. the cell in which the robot is currently located). For any considered cell c, all neighbors a_i of c (i.e. those cells that, partially or fully, share a common side with c) are considered. For each a_i, a randomly generated waypoint q_{c,a_i} is selected (and stored) along the common side shared by the two cells (see Fig. 6). The distance from a_i to the starting point can then be computed and stored.

However, the path planning algorithm should also be able to handle situations in which a moving obstacle (which, of course, would not be included in the convex navigation grid) is found. If, for a line segment l connecting c to a_i, an intersection is found between l and a circle (with an added margin) around a detected moving obstacle (see below), a large penalty is added in the computation of the distance from a_i to the starting point, thus effectively rendering that path useless. Note that the robot, of course, has no global knowledge of the positions of moving obstacles: Only those moving obstacles that it can actually *detect* are included in the analysis. Note also that the path planning method will ignore any detected moving obstacle at a distance of at least Δ (a tunable parameter) from the robot. This is so, since there is no point to account for a faraway moving obstacle (which may not even be headed in the robot's direction) while planning the path.

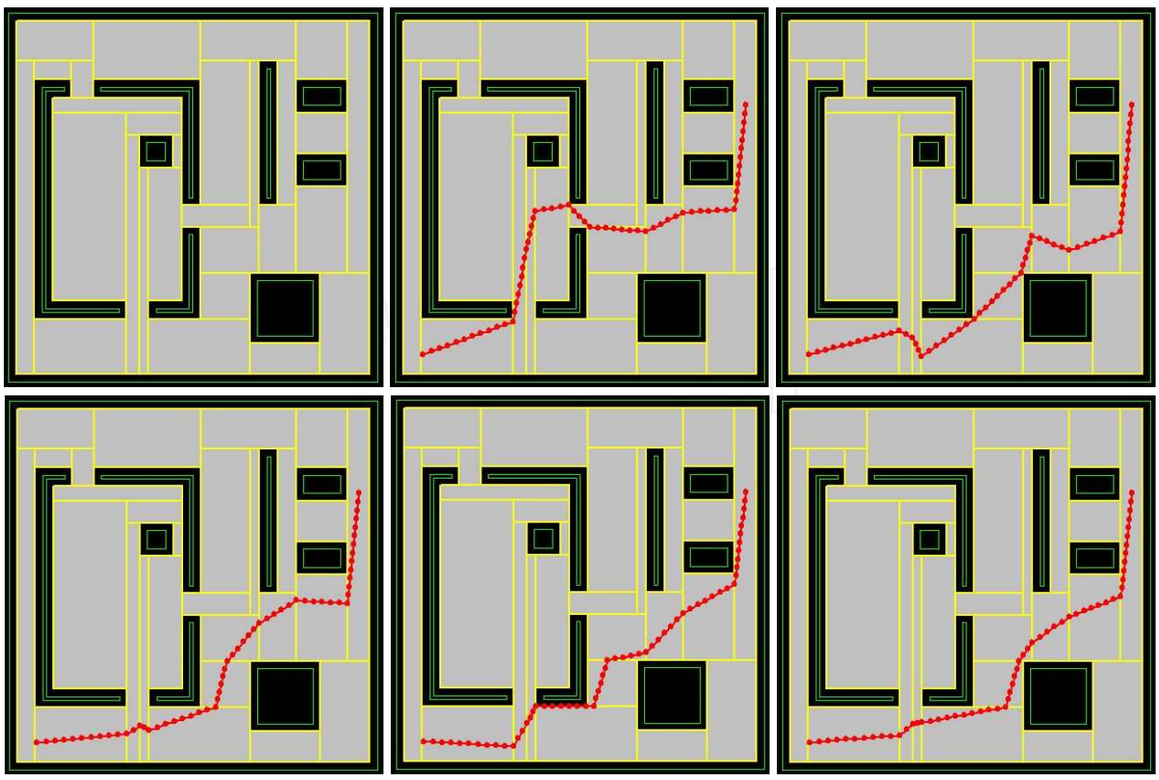


Fig. 6. Path planning: The upper left panel shows the convex navigation grid for the arena described in Sect. 6 and the remaining panels show the paths considered during path planning. In this case, the robot was located in the lower left corner of the arena, and its navigation target was located near the upper right corner. The first path considered is shown in the upper middle panel. After some time, the optimization procedure found a shorter path (upper right panel), along a different route. In the remaining steps (lower panels) the path planning algorithm further optimized the path (length minimization) by adjusting the placement of the waypoints (on the edges of the convex grid cells). Note that, in this figure, the actual navigation path is shown, i.e. not only the waypoints on the cell edges, but also all the intermediate (interpolated) waypoints.

The process is carried out for all grid cells, since the geometrically shortest path may involve a potential collision with a moving obstacle, as described above, in which case another path must be found, as shown in Fig. 7. A navigation path candidate is obtained once all grid cells have been considered.

However, as noted above, the exact locations of the waypoints q_{c,a_i} (on the edges of the navigation grid cells) are randomly chosen. Thus, an optimization procedure is applied, during which the entire path generation process is iterated n times, using different randomly selected waypoints along the cell sides (see also Fig. 6), and the shortest path thus found is taken as the final path. Once this path has been obtained, the actual *navigation path* is formed by adding interpolated waypoints between the waypoints obtained during path generation, so that the distance between consecutive waypoints is, at most, d_{wp} (a user-specified parameter).

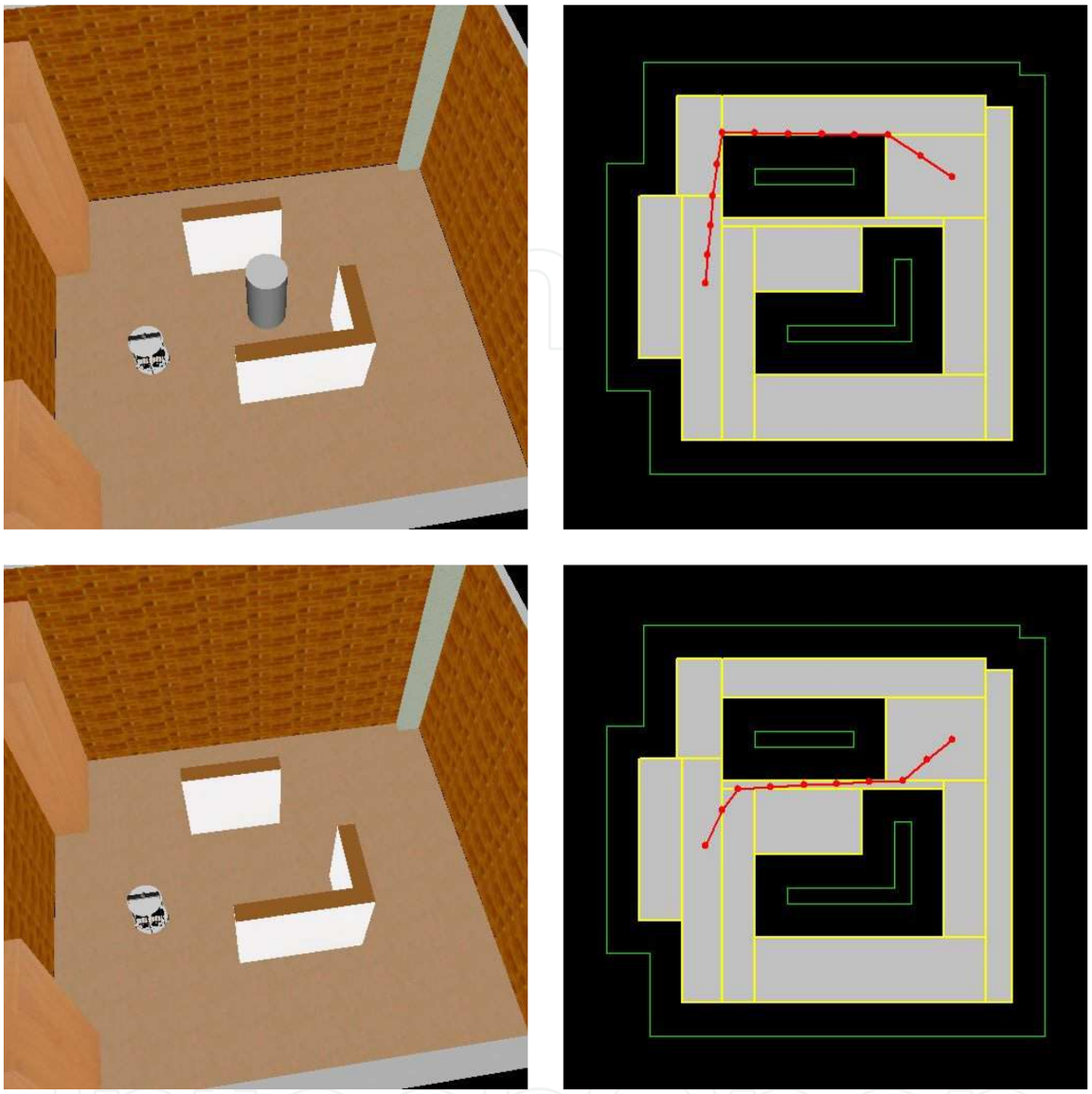


Fig. 7. Path planning with (upper panels) and without (lower panels) moving obstacles present. The moving obstacle is represented as a gray cylinder.

4.1.3 Navigation

The navigation motor behavior, illustrated in Fig. 8, consists of an FSM with 9 states. In State 1, the robot stops (path planning takes place at standstill), and jumps to State 2 in which the robot determines whether or not (based on its odometric position estimate) it is actually located in the convex navigation grid. This check is necessary, since the robot may have left the grid (slightly) as a result of carrying out moving obstacle avoidance; see below. If the robot finds that it is located *outside* the grid, it jumps to State 3 in which it checks, using the LRF[2], whether a certain sector (angular width γ_c, radius r_c) in front of the robot is clear, i.e. does

[2] It is assumed that the height of the moving obstacles is such that they are detectable by the (two-dimensional) LRF.

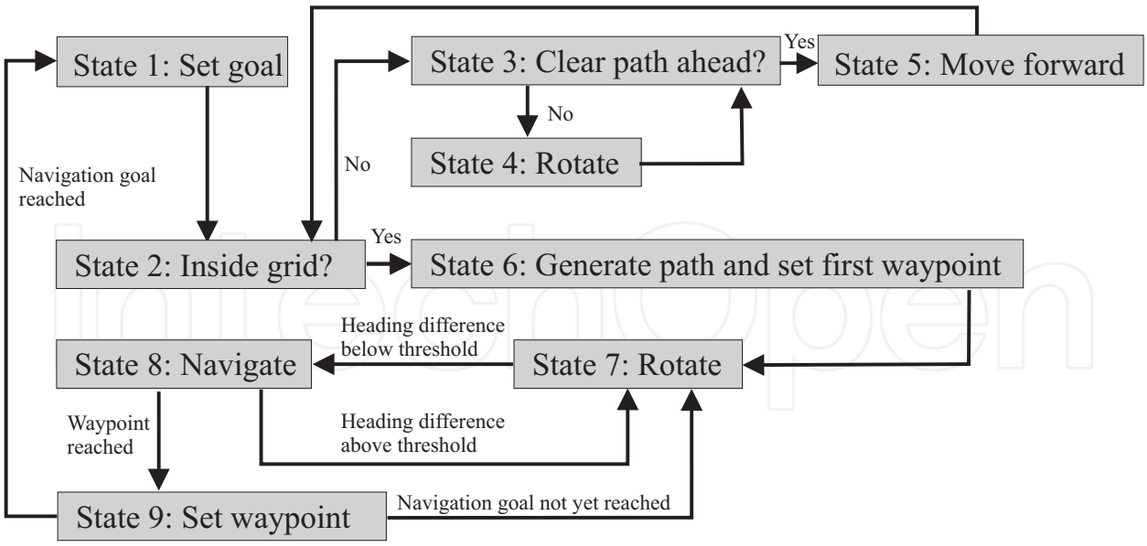


Fig. 8. The FSM implementing grid navigation.

not contain any obstacle. If this is so, the robot jumps to State 5 of the FSM, in which it sets the motor signals to equal, positive values, making the robot move forward, essentially in a straight line[3]. It then jumps back to State 2, again checking if it has reached the grid etc. If instead the robot finds that the path ahead is not clear, it jumps to State 4, in which it sets the motor signals to equal values but with opposite signs, so that it begins rotating without moving its center of mass. It then jumps back to State 3, to check whether the path ahead is clear.

If, in State 2, the robot finds that it is located *inside* the grid, it jumps to State 6, in which it determines the starting point (from the current odometric estimate) and the navigation goal (provided by the user), then generates an optimized path using the procedure described in Subsect. 4.1.2 and, finally, sets the current waypoint as the second waypoint in the path (the first point being its current location). It then jumps to State 7, in which it checks whether (the absolute value of) the difference[4] in estimated heading (using odometry) and desired heading (obtained using the direction to the current waypoint) exceeds a threshold δ. If so, it begins a pure rotation (as in State 4, see above). Once the difference in heading drops below the threshold, the robot jumps to State 8, in which the left and right motor signals are set as

$$v_{\mathrm{L}} = V_{\mathrm{nav}} - \Delta V \tag{6}$$

and

$$v_{\mathrm{R}} = V_{\mathrm{nav}} + \Delta V, \tag{7}$$

respectively, where

$$\Delta V = K_p V_{\mathrm{nav}} \Delta\varphi, \tag{8}$$

where $\Delta\varphi$ is the difference in heading angle, and K_p is a constant. While in State 8, the robot continuously checks the difference between estimated and desired heading. If it should

[3] Even though the motor settings are equal, the actual path may differ slightly from a straight line, due to actuator noise and other similar sources of error.

[4] The difference $\Delta\phi$ between two heading angles φ_1 and φ_2 is defined as $\varphi_1 - \varphi_2 \bmod 2\pi$.

exceed the threshold δ, the robot returns to State 7. If instead the robot reaches within a distance d_{p} (around 0.1 - 0.2 m) of the current waypoint, the robot jumps to state 9 where it checks whether it has reached the navigation goal (i.e. the final waypoint). If this is the case, the robot picks the next navigation goal from the list, and returns to State 1.
If not, the robot selects the next waypoint in the current path. Now, the simplest way to do so would be to just increase the index of the waypoint by one. Indeed, this is normally what happens. However, in certain situations, it may be favorable to skip one waypoint, namely in cases where the path changes direction abruptly. In such cases, passing all waypoints will lead to an unnecessarily jerky motion. Thus, in State 9, before selecting the next waypoint, the robot considers the angle between the vectors $\mathbf{v}_1 = \mathbf{p}_{i+1} - \mathbf{p}_i$ and $\mathbf{v}_2 = \mathbf{p}_{i+2} - \mathbf{p}_{i+1}$, where $\mathbf{p}_i$ is the waypoint that the robot just has reached, and $\mathbf{p}_{i+1}$ and $\mathbf{p}_{i+2}$ are the two next waypoints (this procedure is of course skipped if $\mathbf{p}_i$ or $\mathbf{p}_{i+1}$ is the final waypoint along the path). If this angle exceeds a certain threshold *and* the path ahead of the robot is clear (measured as in State 3, but possibly with different thresholds defining the sector in front of the robot), the robot skips waypoint $i + 1$ and instead targets waypoint $i + 2$.

4.2 Odometry

The *Odometry* brain process is straightforward: Given the estimated wheel rotations since the last update (based on the pulse counts from the wheel encoders), the robot uses the simple kinematic model of a differentially steered two-wheeled robot (with two additional supports for balance) to obtain an estimate of its velocity (v_x, v_y). Using standard forward kinematics, the robot also obtains estimates of its position (x, y) and angle of heading φ.

4.3 Odometric calibration

The purpose of the *odometric calibration* brain process is to correct the inevitable errors in the pose obtained from the *odometry* brain process. Odometric calibration relies on scan matching and builds upon an earlier version (called *laser localization*) described by Sandberg et al. (2009). However, whereas the earlier version required the robot to stop before correcting the odometric errors, the new version operates continuously and in concert with other brain processes. Thus, using the brain process taxonomy described in Sect. 3, the new version is a cognitive process rather than a motor behavior.
The FSM of the odometric calibration brain process operates as follows: In State 1, the robot checks whether the (modulus of the) difference in motor signals (between the left and right motor) exceeds a given threshold $T_{\Delta v}$. If it does, the robot remains in State 1, the rationale being that scan matching is less likely to give a reliable result if the robot is currently executing a sharp turn rather than moving forward (or backward). This condition is less of a restriction than one might think: Since the grid navigation method attempts to find the shortest path from the starting point to the goal point, the resulting path typically consists of long straight lines, with occasional turns where necessary.
When the wheel speed difference drops below the threshold, the robot jumps to State 2 of the FSM, where it stores the most recent laser scan and then carries out a laser scan match. The procedure for the scan match has been described in full detail by Sandberg et al. (2009), and will not be given here. Suffice it to say that the scan matching procedure compares the stored laser scan to a virtual scan taken in the map, by directly computing (essentially) the root mean square error between the laser ray readings. Thus, unlike many other methods, this method does *not* rely on the (brittle and error-prone) identification of specific landmarks. Furthermore, unlike the version presented by Sandberg et al. (2009), the current version is able to filter out

moving obstacles. It does so by noting that there is an upper limit on the pose errors, since the odometric calibration is executed at least a few times per second. Thus, laser rays for which the discrepancy between the expected and actual reading is too large to be caused by pose errors are simply ignored in the scan matching procedure; see also Subsect. 5.3 below.
If the scan match error is below a threshold T_ϵ, the robot concludes that its pose is correct and jumps back to State 1. However, if the error exceeds the threshold, a search procedure is initiated in State 3 of the FSM. Here, the robot carries out a search in a region of size $L_x \times L_y \times L_\varphi$ in pose space, around the estimated pose (obtained from odometry) recorded when the stored laser scan was taken.
Since the search procedure takes some time (though less than one second) to carry out, once it has been completed the robot must correct for the movement that has occurred since the stored laser scan was taken. This correction is carried out using odometry, which is sufficiently accurate to provide a good estimate of the *change* in the robot's pose during the search procedure (even though the *absolute* pose may be incorrect, which is the reason for running odometric calibration in the first place!)

4.4 Moving obstacle detection

As mentioned above, in the case considered here, the only long-range sensor is an LRF mounted on top of the robot. Thus, a method for detecting moving obstacles, i.e. any obstacle that is not part of the map, will have to rely on differences between the expected and actual LRF readings at the robot's current pose. Needless to say, this requires, in turn, that the robot's pose estimate, generated by the *Odometry* and *Odometric calibration* brain processes, is rather accurate. Assuming that this is the case, the FSM implementing the *Moving obstacle detection* (MOD) brain process works as follows: The FSM cycles between Steps 1 and 2, where Step 2 is a simple waiting state (the duration of which is typically set to around 0.02 to 0.10 s). In State 1, the robot compares the current LRF readings with the expected readings (given its pose in the map). Essentially, those readings for which the difference between the actual and expected readings exceed a certain threshold T_{mo} are considered to represent moving obstacles. Thus, even in cases where the robot's pose is slightly incorrect (due to odometric drift), it will still be able reliably to detect moving obstacles, provided that T_{mo} is set to a sufficiently large value. On the other hand, the threshold cannot be set *too* large, since the robot would not be able to detect, for example, moving obstacles just in front of a wall. In the investigation considered here, the value of T_{mo} was equal to 0.30 m.
Normally, when a moving obstacle is present within the range of the LRF, a set of consecutive LRF rays will be found to impinge on the obstacle. Given the first and last of those rays, estimates can be obtained for the position and radius of the obstacle. However, in order to minimize the risk of spurious detections, a list of moving obstacle *candidates* is maintained, such that only those candidates that fulfill a number of conditions are eventually placed in the actual list of moving obstacles. To be specific, detected obstacle candidates are removed if (i) they are located beyond a certain distance $d_{\text{mo}}^{\max}$ or (ii) the inferred radius is smaller than a threshold $r_{\text{mo}}^{\min}$. An obstacle candidate that survives these two checks is artificially increased in size by a given margin (typically 50%). On the other hand, for a detected obstacle candidate such that the inferred radius (after adding the margin) exceeds a threshold $r_{\text{mo}}^{\max}$ (typically as a result of pose errors), the actual radius is set equal to $r_{\text{mo}}^{\max}$.
Despite the precautions listed above, the robot will, from time to time, make spurious detections. Hence, a probability measure p_i for each moving obstacle i is introduced as well. A newly detected moving obstacle candidate is given probability $p_i = p_0$. If the

obstacle candidate is found again (in the next time step), its probability is modified as $p_i \leftarrow \max(\alpha p_i, 1)$, where $\alpha > 1$ is a constant. If, on the other hand, the previously detected obstacle candidate is not found, the probability is modified as $p_i \leftarrow \beta p_i$, where $\beta < 1$ is a constant. Whenever the probability p_i drops below a threshold $p_{\min}$, the corresponding obstacle is removed from the list of candidates. Finally, the remaining candidates are added to the list of actual moving obstacles, when they have been detected (i.e. with probability above the minimum threshold) for at least ΔT_{mo} s (a user-specified parameter, typically set to around 0.5 s).

In order to avoid multiple detections of the same moving obstacle (remembering that the MOD method runs many times per second), detected moving obstacle candidates that are within a distance $\delta_{\text{mo}}^{\max}$ of a previously detected obstacle candidate are identified with that obstacle candidate. Obstacle candidates that do not fulfill this condition are added as separate obstacle candidates. In addition to generating a list of moving obstacles, the brain process also determines which of the moving obstacles is closest to the robot.

Furthermore, a scalar variable, referred to as the *moving obstacle danger level* (denoted μ) is computed (also in State 1 of the FSM). This variable is used as a state variable for the decision-making system, and influences the activation (or de-activation) of the moving obstacle avoidance motor behavior, which will be described below. Needless to say, a scalar variable for assessing the threat posed by a moving obstacle can be generated in a variety of ways, without any limit on the potential complexity of the computations involved. One may also, of course, consider using additional long-range sensors (e.g. cameras) in this context. Here, however, a rather simple definition has been used, in which the robot only considers moving obstacles in the frontal[5] half-plane (relative direction from $-\pi/2$ to $\pi/2$). The robot first computes the change in distance to the nearest obstacle between two consecutive executions of State 1. If the distance is increasing, the current danger level m is set to zero. If not, the current danger level is determined (details will not be given here) based on the distance and angle to the moving obstacle. Next, the actual danger level μ is obtained as $\mu \leftarrow (1-\alpha)\mu + \alpha m$, where α is a constant in the range $]0, 1]$.

4.5 Moving obstacle avoidance

The *Moving obstacle avoidance* (MOA) motor behavior is responsible for handling emergency collision avoidance[6]. Once the MOA motor behavior has been activated[7], it devotes all its computational capacity to avoiding the nearest moving obstacle; it makes no attempt to assess the danger posed by the obstacle, a task that is instead handled by the decision-making system, using the moving obstacle danger level computed in the MOD brain process. Just as in the case of the MOD brain process, MOA can also be implemented in complex and elaborate ways.

Here, however, a rather simple MOA method has been used, in which the region in front of the robot is divided into four zones, as illustrated in Fig. 9. Should the MOA motor behavior be activated, the actions taken by the robot depend on the zone in which the moving obstacle

[5] The robot thus only handles frontal collisions; it is assumed that a moving obstacle, for example a person, will not actively try to collide with the robot from behind.

[6] Note that the grid navigation motor behavior also handles *some* collision avoidance, in the sense that it will not deliberately plan a path that intersects a nearby moving obstacle.

[7] Note that, whenever the MOA motor behavior is activated, the grid navigation motor behavior is turned off, and vice versa. This is so since, in the UF method, only one motor behavior is active at any given time; see Sect. 3.

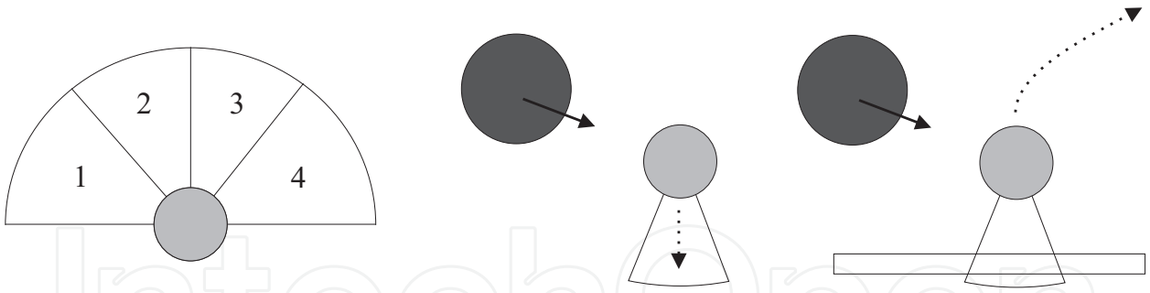


Fig. 9. Left panel: The zones used in the moving obstacle avoidance motor behavior. The robot is shown as a light gray disc. Middle and right panels: The two evasive maneuvers defined for cases in which the moving obstacle (dark gray disc) is located in Zone 1. If the robot is able to move backwards (based on the readings from the rear IR sensor) it does so (middle panel). On the other hand, if an obstacle behind the robot prevents it from moving backwards (right panel), it attempts instead to evade the obstacle by moving forward and to the right.

is located. For example, if the moving obstacle is located in Zone 1, the robot checks (using the rear IR sensor) whether it can move backwards without collisions. If so, it sets the motor signals to equal, negative values, thus (asymptotically) moving straight backwards, until it has moved a certain, user-specified distance, at which point the robot stops as the MOA motor behavior is awaiting de-activation.

If the robot is unable to move backwards (either initially or after some time), it instead attempts to circumnavigate the moving obstacle by turning right, setting the left motor signal to a large positive value, and the right motor signal to a small negative value. Once the robot has turned a certain angle (relative to the initial angle of heading), it instead sets the motor signals to equal positive values, thus (asymptotically) moving forward in a straight line, until it has moved a certain distance. At some point, the obstacle will disappear from the field of view of the LRF (either because it has passed the robot, or as a result of the robot's evasive action), at which point the obstacle danger level (μ) drops, and the MOA motor behavior is de-activated shortly thereafter, so that grid navigation can be resumed. A similar, but somewhat more elaborate, procedure is used if the obstacle is located in Zone 2. Zones 3 and 4 are handled as Zones 2 and 1, respectively, but with the motor signal signs etc. reversed.

The utility function for the MOA motor behavior uses the obstacle danger level (see Subsect. 4.4) as a state variable. Hence, when the perceived obstacle danger level rises above a threshold, which is dependent on the parameter values used in Eq. (1), the MOA motor behavior is activated. At this point, the robot attempts to avoid the moving obstacle, as described above. In cases where it does so by turning, the perceived obstacle danger level typically drops. Thus, in order to avoid immediate de-activation, the MOA motor behavior uses a gamma parameter to raise its own utility once activated (in State 1); see also Sect. 3 above.

4.6 Long-term memory

The *Long-term memory* (LTM) brain process simply stores (and makes available to the other brain processes) the resources necessary for the robot to carry out its navigation task. In the case considered here, the LTM stores the map of the arena (needed during odometric calibration) and the convex navigation grid (needed for navigation).

5. Implementation and validation

5.1 The general-purpose robotics simulator

A simulator referred to as the *General-purpose robotics simulator* (GPRSim) has been written with the main purpose of evaluating the UF decision-making method. The simulator allows a user to set up an arbitrary arena, with or without moving obstacles, and a robot, including both its physical properties (e.g. actuators, sensors, physical parameters such as mass, moment of inertia etc.) and its artificial brain (i.e. a set of brain process, as described above, and the decision-making system implementing the UF method, also described above). In order to make the simulations as realistic as possible, the simulator introduces noise at all relevant levels (e.g. in sensors and actuators). Moreover, simulated sensor readings are taken only with the frequency allowed by the physical counterpart.

5.2 Simulations vs. reality

Simulations are becoming increasingly important in many branches of science, and robotics is no exception. However, in the early days of behavior-based robotics many researchers expressed doubts as to the possibility of transferring results obtained in simulations onto real robots. In particular, Brooks took an especially dim view of the possibility of such transfer, as discussed in Brooks (1992). Similar critique of simulations has been voiced by, among others, Jakobi et al. (1995) and Miglino et al. (1996). However, their critique mainly targeted naive simulation approaches that use idealized sensors and actuators in block-world-like environments. In fact, both Jakobi et al. (1995) and Brooks (1992) also note that simulations are indeed useful, provided that great care is taken to simulate, for example, actuators and sensors in a realistic manner and to validate the simulation results using real robots. Furthermore, even though the discrepancy between simulations and reality is likely to increase as ever more complex robotic tasks are considered (as argued by Lipson (2001)), it is also likely that carrying out proper tests of such robots will, in fact, *require* simulations to be used, particularly if the simulations can be executed faster than real time. In addition, careful simulations allow researchers to test various hardware configurations before embarking on the often costly and time-consuming task of constructing the real counterpart.

In our view, simulations are an essential tool in mobile robot research, but should only be applied in cases where one can argue convincingly that (i) the various hardware components (e.g. sensors and actuators) can be simulated in a reliable way, and (ii) the actions taken do not require extreme accuracy regarding time or space. As an example regarding simulation of hardware components, consider the sensors used on our robot; in the application considered here almost all relevant sensing is carried out using the LRF, a sensor that is, in fact, very simple to simulate, in view of its high accuracy: Our LRF has a typical error of $\pm$ 1 mm out to distances of around 4 m. On the other hand, if a *camera* had been used on the real robot, it is unlikely that a reliable simulation could have been carried out. Even though GPRSim allows an almost photo-realistic representation of the environment, the readings from a simulated camera would be unlikely to capture all aspects (such as shadows, variations in light levels etc.) of vision. Regarding time and space, the implementation of the navigation method (described above) introduces, by construction, a margin between the robot and the arena objects. Thus, while the actions of the simulated robot may differ somewhat from those of a real robot (and, indeed, also differ between different runs in the simulator, depending on noise in actuators and sensors), there is little risk of *qualitative* differences in the results obtained.

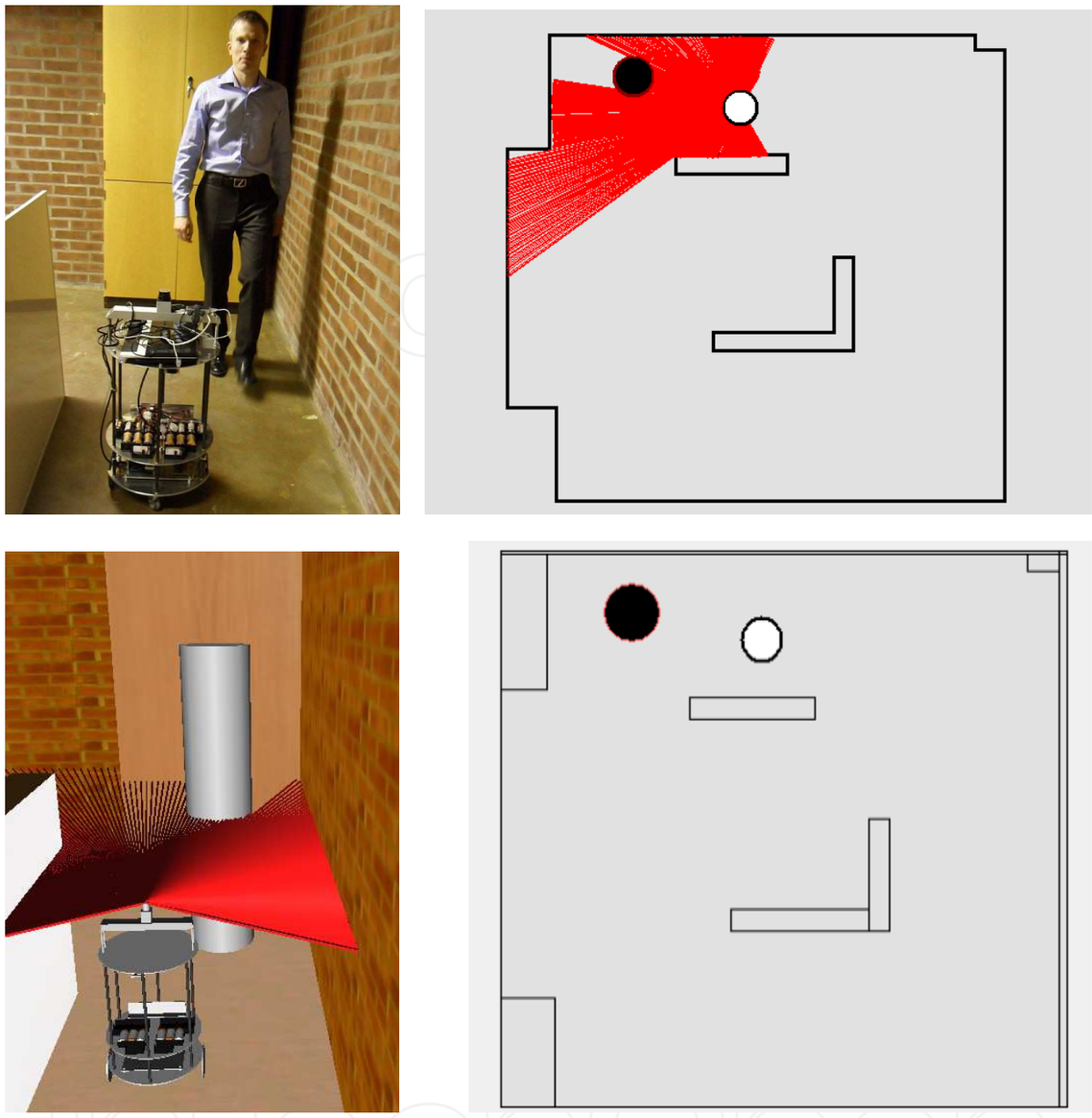


Fig. 10. Upper panels: Detection of a moving obstacle by the real robot. Left: The person approaching the robot. Right: a screenshot showing the LRF rays (for the real robot). The robot is shown as a white disc at its estimated pose, and the detected moving obstacle as a black disc. Lower panels: The same situation, shown for the simulator. Here, the laser rays (which obviously are not visible in the real robot) have been plotted in the screenshot rather than in the map view shown in the right panel.

Nevertheless, before using any simulator for realistic robotics experiments, careful validation and system identification must be carried out, a process that will now be described briefly, in the case of our simulator.

5.3 Validation of the simulator

Using the real robot, several experiments have been carried out in order to validate the GPRSim simulator and to set appropriate parameter values for the simulator. In fact, careful validation of GPRSim (as well as system identification of sensors and actuators) has been carried out in previous (unpublished) work, in which navigation over distances of 10-20 m was considered. The odometric calibration method has been carefully tested both in those experiments and as a separate brain process; see Sandberg et al. (2009).

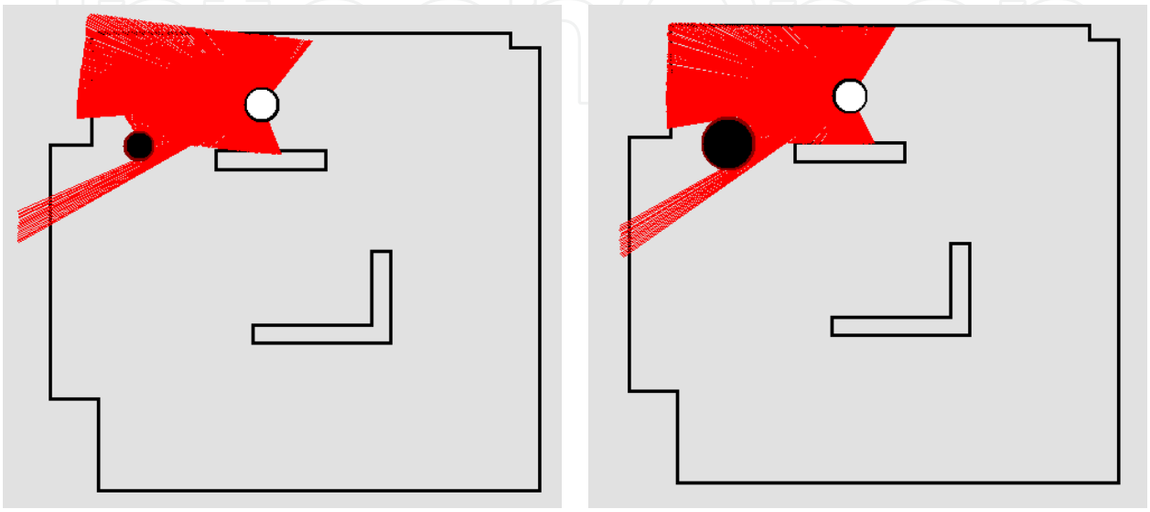


Fig. 11. An example of odometric calibration in the presence of a moving obstacle (shown as a black disc). The real robot, shown as a white disc, was used. The same laser readings (from a single laser scan) are shown in both panels. In the left panel, the origin and direction of the laser readings correspond to the robot's estimated (and incorrect) pose before odometric calibration. In the right panel, the estimated pose has been corrected.

However, in the previous work, moving obstacles were not included. Thus, here, a thorough effort has been made to ascertain the validity of the simulation results in situations where moving obstacles are present. Basically, there are two things that must be tested namely (i) moving obstacle detection and (ii) odometric calibration in the presence of moving obstacles. An example of the validation results obtained for moving obstacle detection is shown in Fig. 10. The upper panels show the results obtained in the real robot, and the lower panel show the results from GPRSim, for a given robot pose and obstacle configuration. As can be seen, the robot (shown as a white disc) detected the moving obstacle (shown as a black disc) in both cases.

Fig. 11 shows the results (obtained in the real robot) of a validation test regarding odometric calibration. In this particular test, the actual position of the robot was $(x, y) = (2.45, 4.23)$ m. The heading angle was equal to π radians (so that the robot was directed towards the negative x-direction). For the purposes of testing the odometric calibration process, an error was introduced in the *estimated* pose; the estimated position was set to $(x, y) = (2.25, 4.20)$ m, and the estimated heading to 3.0 radians. The left panel shows the robot and the moving obstacle (black disc), as well as the actual LRF readings, plotted with the origin and direction provided by the robot's *estimated* pose. As can be seen, the map is not very well aligned with the readings. The right panel shows the same laser readings, again with origin and direction given by the robot's estimated pose, but this time using the estimate obtained *after*

odometric calibration. At this point, the alignment is almost perfect, despite the presence of the moving obstacle. Repeating the experiment 10 times, the estimated position (after odometric calibration) was found to be $(x, y) = (2.40 \pm 0.02, 4.22 \pm 0.01)$ m and the estimated heading was 3.11 ± 0.02 radians, where the errors correspond to one standard deviation. The corresponding values found in the simulator were $(x, y) = (2.38 \pm 0.03, 4.23 \pm 0.01)$ m, whereas the estimated heading was found to be 3.13 ± 0.02 radians.Several experiments of this kind were carried out. In all cases, both the simulated robot and the real robot were able to correct their estimated pose using odometric calibration, with essentially identical level of precision.

6. Results

The (simulated) arena used for studying long-term robustness is shown from above in the upper left panel of Fig. 12. The size of the arena, which represents a warehouse environment, is 10×10 m. The large object near the lower right corner represents an elevator shaft (inaccessible to the robot).

Regarding the decision-making system, a single state variable was used, namely the obstacle danger level described in Subsect. 4.4 above. The four cognitive brain processes (*Odometry*, *Odometric calibration*, *Moving obstacle detection*, and *Long-term memory*) were allowed to run continuously. Hence, in their utility functions, the parameters a_{ij} were equal to zero, and b_i was positive. The main task of the decision-making system was thus to select between the two motor behaviors, namely *Grid navigation* (B_1) and *Moving obstacle avoidance* (B_5), using the obstacle danger level as the state variable (z_1). Since only the relative utility values matter (see Sect. 3), the parameters a_{1j} were set to zero, b_{1j} was set to a positive value (0.25), and the time constant τ_1 was set to 0.10 s, thus making u_1 rapidly approach a constant value of around 0.245. After extensive testing, suitable parameters for B_5 were found to be: $a_{51} = 1.00$, $b_5 = -0.45$, and $\tau_1 = 0.10$ s. The gamma parameter Γ_5 (used as described in Subsect. 4.5 above) was set to 3.0 in State 1 of B_5. The corresponding time constant (for exponential decay of the gamma parameter) was set to 1.0 s. The sigmoid parameters c_i were set to 1 for all brain processes.

Several runs were carried out with GPRSim, using the robot configuration described above. Here, the results of two runs will be presented.

6.1 Run 1: Stationary arena

As a first step, a long run (*Run 1*) was carried out, without any moving obstacle present, with the intention of testing the interplay between navigation, odometry, and odometric calibration. The robot was released in the long corridor near the left edge of the arena, and was then required to reach a set of target points in a given order. A total of 11 different target points were used. When the final point was reached, the robot's next target was set as the first target point, thus repeating the cycle. The right panel of Fig. 12 shows the locations of the 11 target points. The robot's trajectory during the first 100 meters of movement (256 s) is shown in the lower left panel of the figure. Both the actual trajectory (thick green line) and the odometric trajectory (thin red line) are shown. As can be seen, the robot successfully planned and followed the paths between consecutive target points. Over the first 100 m of the robot's

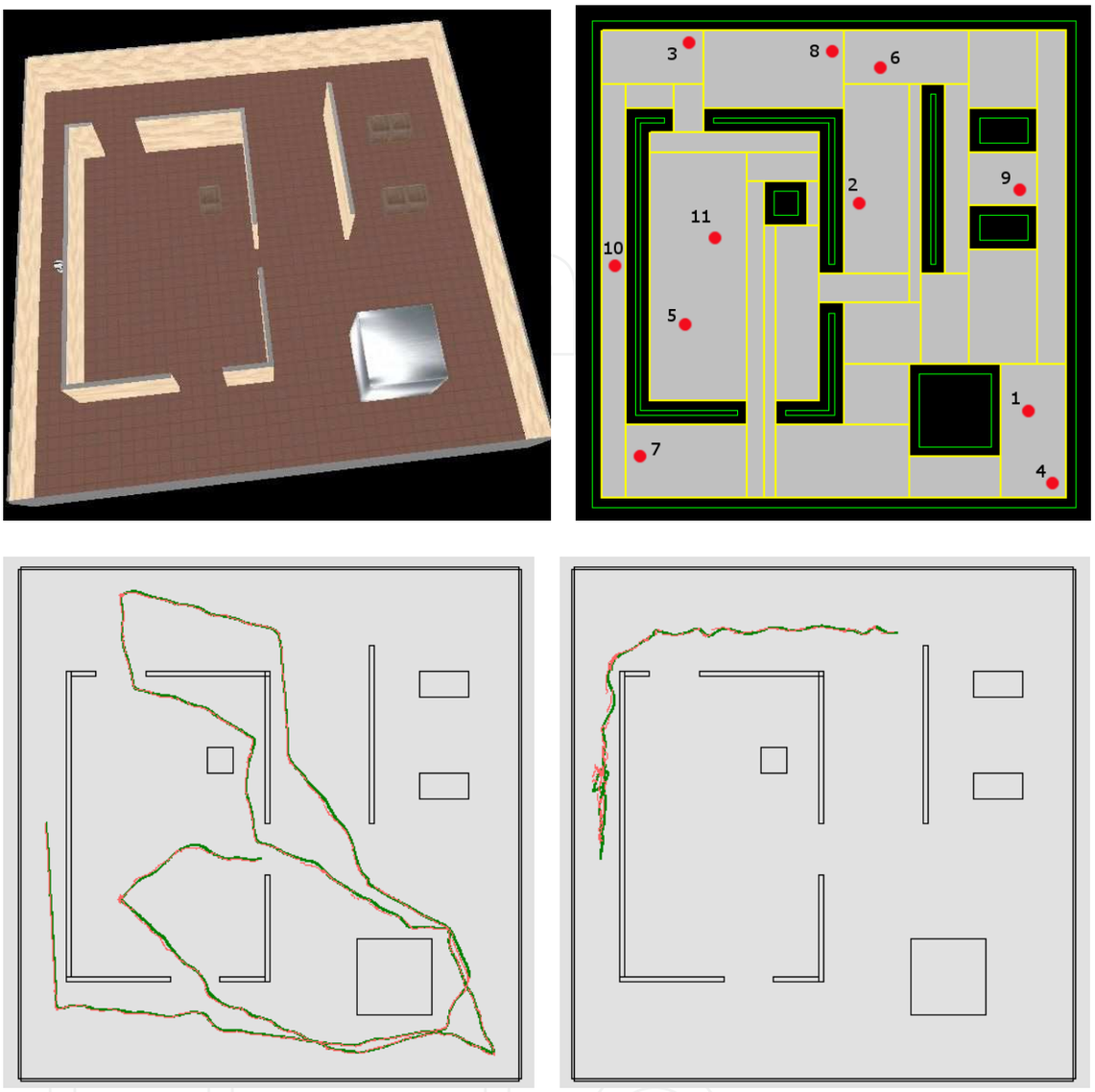


Fig. 12. Upper left panel: The arena used for the simulations. Upper right panel: The 11 target points, with enumeration. Note that, after reaching the 11th target point, the robot's next target becomes the first target point etc. The robot is initially located near target point 10, as can be seen in the upper left panel. Lower left panel: The trajectory for the first 100 m of navigation. The actual trajectory is shown as a thick green line, and the odometric trajectory as a thin red line. Lower right panel: A later part of the trajectory showing the robot approaching 200 m of distance covered. As can be seen, in the long corridor, the robot suffered a rather large pose error at one point. The error was swiftly corrected, however, so that the robot could continue its task.

movement, the average position error[8] was around 0.063 m, and the average heading error

[8] The average position error is computed by averaging the *modulus* of the position error over time. Similarly the average heading error is obtained by averaging the *modulus* of the heading error (modulo 2π) over time.

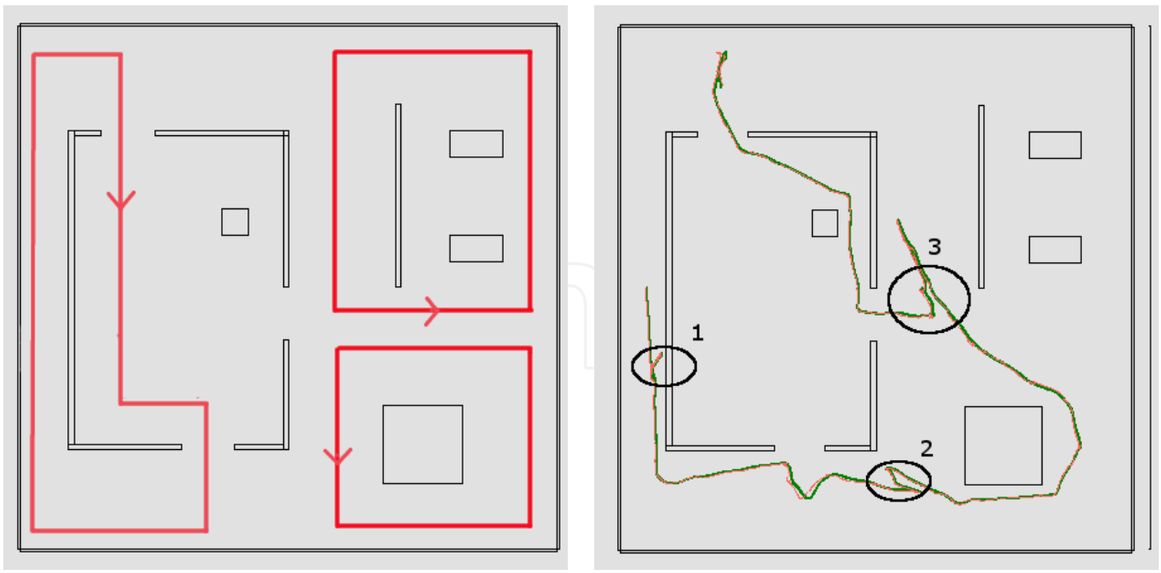


Fig. 13. Left panel: The trajectories of the moving obstacles in Run 2. Right panel: The robot's trajectory during the first 50 m of movement. The ellipses highlight the robot's three evasive maneuvers.

0.031 radians (1.8 degrees). Note, however, that most of the contribution to the error comes from a few, rather rare occasions, where the robot's pose accuracy was temporarily lost. In all cases, the robot quickly recovered (through continuous odometric calibration) to find a better pose estimate. An example of such an error correction is shown in the lower right panel of Fig. 12.

6.2 Run 2: Arena with moving obstacles

In the second run (*Run 2*) presented here, the robot moved in the same arena as in Run 1, and with the same sequence of target points. However, in Run 2, there were three moving obstacles present in the arena. The approximate trajectories of the moving obstacles are illustrated in the left panel of Fig. 13. The right panel shows the first 50 m of the robot's movement. Here, the average position error was around 0.068 m and the average heading error around 0.025 radians (1.4 degrees). The ellipses highlight the three cases where the robot was forced to make an evasive maneuver (during the first 50 m of movement) in order to avoid a collision with a moving obstacle. Note that the odometric calibration process managed to maintain a low pose error even during the evasive maneuver. The run was extended beyond 100 m, with only a small increase in pose error; after 100 m, the average position error was 0.096 m and the average heading error 0.038 radians (2.2 degrees). In this run, the first 100 m took 352 s to traverse (compared to the 256 s in Run 1), due to the evasive maneuvers. Note that, after the third evasive maneuver, the robot modified the original path towards target point 3 (cf. the lower left panel of Fig. 12), having realized that a shorter path was available after evading the moving obstacle.

7. Discussion and conclusion

Even though the results presented in Sect. 6 show that the method used here is capable of achieving robust navigation over long distances in the presence of moving obstacles, a

possible criticism of our work is that there is no guarantee that the robot will not suddenly fail, for example by generating an incorrect heading estimate, such that its position error will grow quickly. Indeed, in very long runs, exactly such problems do occur, albeit rarely. Our approach to overcoming those problems would follow the philosophy of error correction rather than error avoidance. An example of this philosophy can be found in the topic of gait generation for humanoid robots: Rather than trying to generate a humanoid robot that will never fall, it is better to make sure that the robot can get up if it *does* fall. This approach has been applied in, for example, the HRP-2 robot, see Kaneko et al. (2002), and many other humanoid robots subsequently developed.

Similarly, rather than trying to further tune the brain process parameters or the parameters of the decision-making system, we would, for full long-term reliability, advocate an approach in which the robotic brain would be supplemented by (at least) two additional brain processes. These are: (i) An *Emergency stop* motor behavior which would cause the robot to stop if it finds itself in a situation where a collision with a stationary object is imminent (moving obstacles being handled by the *Moving obstacle avoidance* behavior). Such situations would occur, for example, if the odometric calibration suffers a catastrophic failure so that the error in the robot's estimated pose grows without bound. Once the robot has stopped it should turn towards the most favorable direction for calibration (which would hardly be the direction towards the stationary object and neither any direction in which *no* part of the map would be visible). Once a suitable direction has been found, the robot should activate the second added brain process, namely *Wake-up*. This cognitive process would attempt a global search (again matching the current laser scan to virtual scans taken in the map) for the robot's current pose. With these two brain processes, which are to be added in future work, the robot would be able to correct even catastrophic errors.

As for the currently available brain processes, the *Moving obstacle avoidance* brain process could be improved to handle also situations in which *several* moving obstacles are approaching the robot simultaneously. However, to some extent, the brain process already does this since, before letting the robot move, it generally checks whether or not there is a clear path in the intended direction of movement; see also the right panel of Fig. 9. In cases where multiple moving obstacles are approaching, typically the best course of action (at least for a rather slow robot) is to do nothing.

Another obvious part of our future work is to carry out long-term field tests with the real robot. Such field tests could, for example, involve a delivery task in an office environment, a hospital, or a factory.

To conclude, it has been demonstrated that the approach of using modularized brain processes in a given, unified format coupled with decision-making based on utility functions, is capable of robustly solving robot motor tasks such as navigation and delivery, allowing the robot to operate autonomously over great distances. Even so, the robot may occasionally suffer catastrophic errors which, on the other hand, we suggest can be dealt with by the inclusion of only a few additional brain processes.

8. References

Arkin, R. C. (1987). *Towards Cosmopolitan Robots: Intelligent Navigation in Extended Man-Made Environments*, PhD thesis, University of Massachusetts.

Arkin, R. C. (1998). *Behavior-Based Robotics*, MIT Press.

Brooks, R. A. (1986). A robust layered control system for a mobile robot, *IEEE J. of Robotics and Automation* RA-2(1): 14–23.

Brooks, R. A. (1992). Artificial life and real robots, *Proceedings of the First European Conference on Artificial Life*, pp. 3–10.

Bryson, J. J. (2007). Mechanisms of action selection: Introduction to the special issue, *Adaptive Behavior* 15: 5–8.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs, *Numerische Mathematik* 1: 269–271.

Gat, E. (1991). *Reliable Goal-directed Reactive Control for Real-world Autonomous Mobile Robots*, PhD thesis, Virginia Polytechnic Institute and State University.

Jakobi, N., Husbands, P. & Harvey, I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics, *in* F. Morán, A. Moreno, J. J. Merelo & P. Chacón (eds), *Proceedings of the Third European Conference On Artifical Life, LNCS 929*, Springer, pp. 704–720.

Kaneko, K., Kanehiro, F., Kajita, S. et al. (2002). Design of prototype humanoid robotics platform for HRP, *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2002)*, pp. 2431–2436.

Lipson, H. (2001). Uncontrolled engineering: A review of nolfi and floreano's evolutionary robotics (book review), *Artificial Life* 7: 419–424.

McFarland, D. (1998). *Animal Behaviour: Psychobiology, Ethology and Evolution, 3rd Ed.*, Prentice Hall.

Miglino, O., Lund, H. H. & Nolfi, S. (1996). Evolving mobile robots in simulated and real environments, *Artificial Life* 2: 417–434.

Okabe, A., Boots, B., Sugihara, K. & Chiu, S. N. (2000). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd edn, John Wiley and Sons, Ltd.

Pirjanian, P. (1999). Behavior coordination mechanisms - state-of-the-art, *Technical report*, Institute for Robotics and Intelligent Systems, University of Southern California.

Prescott, T. J., Bryson, J. J. & Seth, A. K. (2007). Introduction. modelling natual action selection, *Philos. Trans. R. Soc. Lond. B* 362: 1521–1529.

Sakagami, Y., Watanabe, R., Aoyama, C., Matsunaga, S., Higaki, N. & Fujimura, K. (2002). The intelligent ASIMO: System overview and integration, *Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2478–2483.

Sandberg, D., Wolff, K. & Wahde, M. (2009). A robot localization method based on laser scan matching, *in* J.-H. Kim, S. S. Ge, P. Vadakkepat, N. Jesse et al. (eds), *Proocedings of FIRA2009, LNCS 5744*, Springer, pp. 179–186.

Singh, J. S. & Wagh, M. D. (1987). Robot path planning using intersecting convex shapes: Analysis and simulation, *IEEE Journal of Robotics and Automation* RA-3: 101–108.

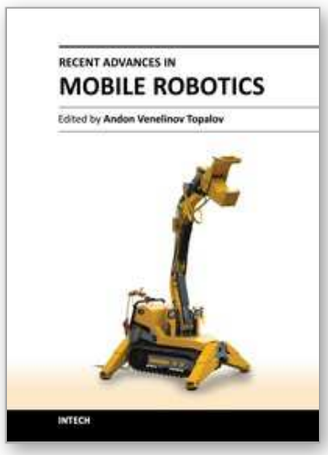
Singh, S. & Agarwal, G. (2010). Complete graph technique based optimization in meadow method of robotic path planning, *International Journal of Engineering Science and Technology* 2: 4951–4958.

Thorpe, C. E. (1984). Path relaxation: Path planning for a mobile robot, *Proceedings of AAAI-84*, pp. 318–321.

von Neumann, J. & Morgenstern, O. (1944). *Theory of Games and Economic Behavior*, Princeton University Press.

Wahde, M. (2003). A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions, *Journal of Systems and Control Engineering (IMechI)* 217: 249–258.

Wahde, M. (2009). A general-purpose method for decision-making in autonomous robots, *in* B.-C. Chien, T.-P. Hong, S.-M. Chen & M. Ali (eds), *Next-Generation Applied Intelligence, LNAI 5579*, Springer, pp. 1–10.



Recent Advances in Mobile Robotics

Edited by Dr. Andon Topalov

ISBN 978-953-307-909-7
Hard cover, 452 pages
Publisher InTech
Published online 14, December, 2011
Published in print edition December, 2011

Mobile robots are the focus of a great deal of current research in robotics. Mobile robotics is a young, multidisciplinary field involving knowledge from many areas, including electrical, electronic and mechanical engineering, computer, cognitive and social sciences. Being engaged in the design of automated systems, it lies at the intersection of artificial intelligence, computational vision, and robotics. Thanks to the numerous researchers sharing their goals, visions and results within the community, mobile robotics is becoming a very rich and stimulating area. The book Recent Advances in Mobile Robotics addresses the topic by integrating contributions from many researchers around the globe. It emphasizes the computational methods of programming mobile robots, rather than the methods of constructing the hardware. Its content reflects different complementary aspects of theory and practice, which have recently taken place. We believe that it will serve as a valuable handbook to those who work in research and development of mobile robots.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen