Robotics and Autonomous Systems 121 (2019) 103261

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot

A comparative study of bug algorithms for robot navigation

K.N. McGuire [a,*], G.C.H.E. de Croon [a,*], K. Tuyls [b]

[a] Delft University of Technology, The Netherlands
[b] University of Liverpool, United Kingdom

ARTICLE INFO

Article history:
Received 17 August 2018
Received in revised form 28 June 2019
Accepted 12 August 2019
Available online 21 August 2019

Keywords:
Bug algorithms
Robotic navigation
Comparative study
Limited sensing
Indoor navigation

ABSTRACT

This paper presents a literature survey and a comparative study of Bug Algorithms, with the goal of investigating their potential for robotic navigation. At first sight, these methods seem to provide an efficient navigation paradigm, ideal for implementations on tiny robots with limited resources. Closer inspection, however, shows that many of these Bug Algorithms assume perfect global position estimate of the robot which in GPS-denied environments implies considerable expenses of computation and memory — relying on accurate Simultaneous Localization And Mapping (SLAM) or Visual Odometry (VO) methods. We compare a selection of Bug Algorithms in a simulated robot and environment where they endure different types noise and failure-cases of their on-board sensors. From the simulation results, we conclude that the implemented Bug Algorithms' performances are sensitive to many types of sensor-noise, which was most noticeable for odometry-drift. This raises the question if Bug Algorithms are suitable for real-world, on-board, robotic navigation as is. Variations that use multiple sensors to keep track of their progress towards the goal, were more adept in completing their task in the presence of sensor-failures. This shows that Bug Algorithms must spread their risk, by relying on the readings of multiple sensors, to be suitable for real-world deployment.

1. Introduction

Robotic indoor navigation of robots has been a sought-after topic for the last few decades within the robotic community. An important stimulus for this interest is its potential for a wide range of scenarios, e.g. search-and-rescue, greenhouse observation, industrial inspection. Indoor navigation also comes with a wide range of issues, such as the absence of a reliable GPS-signal and wall interference in long-range communication. An indoor robot should preferably be autonomous and able to navigate based on its on-board sensors and computational capacity.

There has been tremendous progress in autonomous robotic navigation, up to a point that some researchers believe this to be an already solved problem. With the emerging autonomous cars, simultaneous localization and mapping (SLAM) has reached high maturity in development (see [1] for a review). SLAM is a notoriously complex and expensive algorithm, consuming much of the robot's on-board processing power. To strive towards computationally efficient methods is advantageous for any robot, but it becomes vital when the application requires the use of tiny, light-weight robots. For instance, small Micro Aerial Vehicles (MAVs), in the order of 50 g and 15 cm diameter, could be ideal for exploring small and confined spaces. However, their on-board computational resources are so limited that they cannot make use of the current SLAM methods.

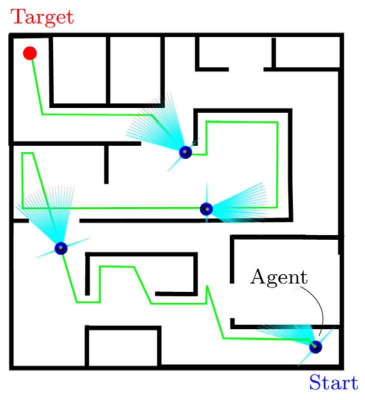


Fig. 1. An example of an agent performing a Bug Algorithm-like behavior, while navigation in an indoor environment. From a starting position (bottom-right), it moves towards the target (top-left), where it tries to move towards the target whenever it can and follows the obstacles' boundary when it hits an obstacle. Its trajectory is given in green.

* Corresponding authors.
E-mail addresses: k.n.mcguire@tudelft.nl (K.N. McGuire), g.c.h.e.decroon@tudelft.nl (G.C.H.E. de Croon).

https://doi.org/10.1016/j.robot.2019.103261
0921-8890/© 2019 Elsevier B.V. All rights reserved.

Given these strict computational requirement for tiny robotic platforms, an important question is raised: does the actual simple principle of navigation, *going from point A to point B*, need the computational and memory requirements for constructing and maintaining high-resolution metric maps? Should the complexity of the strategy not be proportional to the simplicity of the task?

There are several light-weight alternatives to SLAM to consider, such as Topological SLAM (see [2] for a review). Biologically inspired techniques like the Snapshot Model [3] and the Average Landmark Vector [4] can also be considered. These efficient methods, however, still have the tendency to scale up the memory requirements, when navigating in more complex and large environments.

In this article, we will look at a navigation method of a different kind: *Bug Algorithms*. Although the name suggests a biological origin, it is a path-planning technique that evolved from maze-solving algorithms. The main principle of Bug Algorithms is that they do not know the obstacles in their environment and only know their target's relative position. They will locally react only upon contact with obstacles and walls, in a way that lets the agents progress towards their goal, by following the obstacle's boundaries ("wall-following"), as illustrated in Fig. 1. The nature of Bug Algorithms is ideal for indoor navigation on tiny, resource-limited, robotic systems, as their potential memory and processing requirements are low, therefore expected to take up little space on the on-board computer.

In this paper we will delve into Bug Algorithms in more detail, by providing an overview of the techniques existing today. Although there have been two comparative studies on Bug Algorithms before [5,6], our distinctive contribution is that we will evaluate how suitable Bug Algorithms are in for becoming a new navigational standard within robotics. Here we will investigate the assumptions for real-world scenarios. An important conclusion of our study is that Bug Algorithms tend to heavily rely on a perfect position estimation, which cannot be taken for granted in a GPS-deprived indoor environment. Global positioning systems could be set up beforehand, such as a motion capture or Ultra-Wide-Band (UWB) localization system (like in [7]). However, in cases like search-and-rescue scenarios, it is undesirable to have humans prepare the robot's environment. The robots would need to rely on their estimated position, obtained by their own, noisy, on-board sensors. With ground-bound robots, wheel slippage [8] can cause an increasing error between the real and estimated position. The same goes for visual odometry [9], used by MAVs or hovercraft-like vehicles, where the error of the noisy velocity estimate will get accumulated over time. This is especially the case in a texture-poor environment.

We will compare a representative subset of Bug Algorithms in the ARGoS simulator, which is capable of modeling realistic physical interactions with objects in the environment. Although we will not implement as many Bug Algorithms as [6] did, we will test them in more realistic real-world conditions, containing elements such as odometry-drift or recognition-failures. We investigate their behaviors on hundreds of procedurally generated indoor environments, to compare their performance statistically. Here it is shown that the increased measurement noise on the on-board sensors causes a dramatic drop in overall performances of the Bug Algorithms. We will discuss how this affects the potential of Bug Algorithms in robotic navigation and what type of assumptions we can make about the environment, which can point us to the variations that are the most suitable.

An overview of Bug Algorithms is given in Section 2, starting from their "maze-solver" origins, to the fundamental contact-based Bug Algorithms, to the more recent extended range-based versions and hybrid solutions. This is followed by a sum-up of the methods already used in robotic navigation in Section 3. Subsequently, we perform a quantitative comparison of the Bug Algorithms performances, of which the setup is explained in Section 4. The experiments themselves are discussed in Section 5, and involve various degrees of sensor-noise and -failures. The findings of this paper will be discussed in Section 6, from which we will present our conclusions in Section 7.

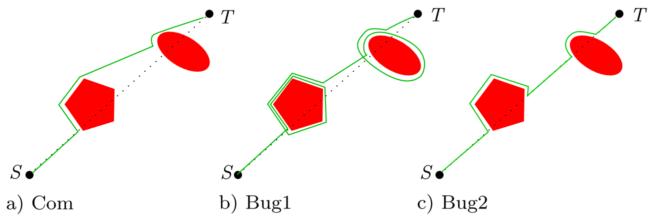


Fig. 2. The behavior of simple Bug Algorithms: (a) Com, (b) Bug1 and (c) Bug2. The S and T depicts the start and target position respectively.

2. Theory and variants of Bug Algorithms

The late 80 s is when the term *Bug Algorithms* (BAs) first came into existence, evolving from the existing path planning algorithms like Dijkstra [10] and A* [11]. However, the latter methods need to know their environment in advance, which includes start and goal positions, all obstacles and their position along the way. Maze-solving algorithms first explored the navigational problem without knowledge about the environment for enclosed spaces with walls and only one entrance and exit. If the target is reachable through a series of interconnected walls, a *wall-follower* would guarantee a quicker solution than a random walker [12]. As long as its left or right side is constantly in contact with a wall while moving through the maze, it will always reach the exit. However, if the environment is not an interconnected maze and contains disjoint obstacles between the start- and end-location, the wall-follower might get stuck in an endless loop. For typical indoor, non-maze, environments with disjoint obstacles, Bug Algorithms will be more suitable for the task.

2.1. Contact Bug Algorithms

Lumelsky and Stepanov [13] were the pioneers in developing bug algorithms. At first, they described a very simplistic BA, called the "common sense algorithm" which can be abbreviated as *Com*. Its behavior is illustrated in Fig. 2(a). The position where a BA hits the obstacle for the first time is called a *hit-point*, and it has a *leave-point* as soon as the direction to the target is free. Intuitively, Com could solve many situations; however, Lumelsky and Stepanov [13] pointed out that there are scenarios in which it cannot reach the goal. This happens when introduced to a more complex environment as, for instance, the one illustrated in Fig. 3(a).

In the same paper of Lumelsky and Stepanov [13], the *Bug1* algorithm was introduced, following a different strategy to overcome the issues that Com is facing. Every obstacle Bug1 comes across, it first has to "explore" the obstacle by following its entire border, while simultaneously keeping track of which position is the closest to the target, as shown in the simple environment in Fig. 2(b). After it encounters its original hit-point, Bug1 will continue and move towards the position closest to the target, from which it will leave the obstacle. Bug1 is able to handle environments where Com failed (as seen in Fig. 3(b)); however, it is a less intuitive approach. As it needs to know the entire border of the obstacle, this will naturally create unnecessary

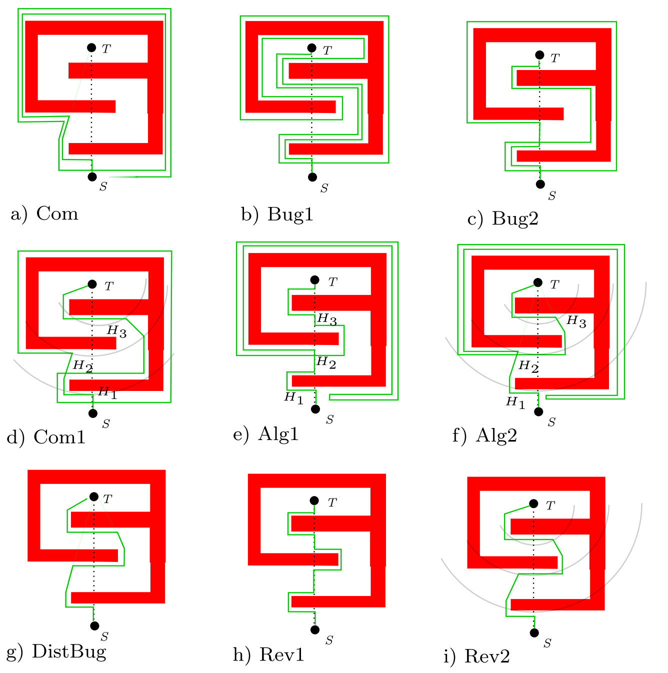


Fig. 3. Generated paths by the Bug Algorithms (a) Com, (b) Bug1, (c) Bug2, (d) Com1, (e) Alg1, (f) Alg2, (g) DistBug, (h) Rev1 and (i) Rev2 in a more challenging environment. The *S* and *T* depicts the start and target position respectively and H_i means the ith hit-point. x is the current position of the agent and CW and CCW stand for Clock Wise and Counter Clock Wise respectively.

long paths. Lumelsky and Stepanov [14] therefore proposed an alternative: *Bug2*. Between the beginning and end position, an imaginary line is drawn, called the *M-line*. In the simple scenario of Fig. 2(c), this means that the bug will follow the obstacles border until it hits the same M-line at the other side. As long as that point is closer towards the target than the hit-point's position, it will depart from the obstacle, as illustrated by Fig. 3(c).

Sankaranarayanan and Vidyasagar [15] were able to reduce the path length even further by adding memory. They extended the Bug2 algorithm with the following principle: to change its wall-following direction if it comes across a previously visited hit-point along the border of the obstacle. It has been dubbed as *Alg1*. It is true that in some situations a shorter path will be generated, however in others it will increase the path length, as can be seen in Fig. 3(e). Sankaranarayanan and Vidyasagar [15] also suggested an extended version of Com, *Com1*,[1] which remembers the previous hit-point's distance to the target. Com1 will utilize this as an extra argument to initiate the departure from the obstacle boundary, as seen in Fig. 3(d). Based on Com1, *Alg2* was created in the same paper of Sankaranarayanan and Vidyasagar [15] as well, where it, similar to Alg1, reverses the wall-following direction if it encounters a previous saved hit-point. Alg2 therefore needs to keep track of all previous hit-points on its way for the reverse local direction condition, as this will occasionally occur (Fig. 3(f)). Kamon and Rivlin [16] created a BA quite similar to Alg2, *DistBug*.[2] The only difference is that it will not remember the positions of all the previous hit-points, but solely the last one, therefore making it more memory efficient. Another intriguing aspect of DistBug, is that the local

[1] This is also being referred to as *Class1* in the studies of Noborio et al. [5] and Ng and Bräunl [6].

[2] Here we are referring to the extended DistBug algorithm of the same paper, with the search manager and local-direction choice based on the slope of the wall.

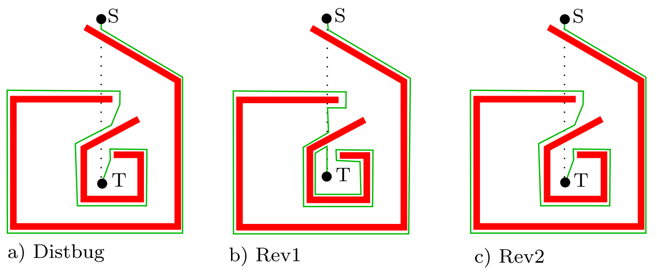


Fig. 4. An alternative complex environment to show a case that would produce a long path-length for (a) DistBug, (b) Rev1 and (c) Rev2.

wall-following direction depends on the orientation with which the BA touches the hit-point. Most times, this will naturally lead it to the target and result in a shorter path, which is noticeable in the environment illustrated in Fig. 3(g). However, there are situations where such a policy will fail, as in Fig. 4(a). At the first hit point, it would be better to follow the wall in the other direction. The same goes for *Rev1* and *Rev2*, extensions to both Alg1 and Alg2 respectively, proposed by Horiuchi and Noborio [17]. Both BAs will alternate their local direction at each (new) hit-point, which is a good strategy for environments like in Fig. 3(h) and (i). However, Fig. 4(b) and (c) show a situation where alternating the local wall-following direction is not the best policy. These examples do show that the best choice of local direction depends on the environment. It is, therefore, difficult to find a generic strategy for determining the best wall-following direction.

2.2. Bug Algorithms with a range sensor

What if the robot is able to sense obstacles already at a certain range and therefore act before touching the obstacles physically? Lumelsky and Stepanov [13] already mentioned this idea in their first paper, which has been materialized in the papers of Lumelsky and Skewis [18] and Lumelsky and Skewis [19] as *VisBug 21 & 22*. Both are based on Bug2, but are now equipped with a range sensor able to sense up to a given maximum range. The BA will still follow the M-line but they can detect "short-cuts" to the next obstacle which should reduce the total path traveled, as can be seen in Fig. 5(a).

Kamon et al. [20] introduced a successful version of the range-based Bug Algorithms, called *TangentBug*. Within the maximum range of its sensor, a local tangent graph (LTG) is constructed, as illustrated in Fig. 5(b). The LTG represents the discontinuities/borders of the detectable obstacle field around the robot. It starts out by moving towards the target while traversing the LTG edge that results in the quickest path *D* to the target (T) from its current position (x). However, if *D* of that edge increases, it will save the current range to the target as a local minimum (d_{min}) and will continue following the remaining boundary of that obstacle. If the robot senses a node on the boundary of the obstacle that is smaller than d_{min}, it trigger a leave-condition and, if possible, moves directly to the target (see Fig. 5(c)). [21] extended TangentBug to operate in 3D-environments as well (*3DBug*).

TangentBug is probably the most referred work in the field of BAs and many variants of it exists. The 360° range sensor assumption is changed to a sensor with a limited field of view with *WedgeBug* [22], for instance, to represent a stereo camera. Magid and Rivlin [23] developed a BA which will actively search for the right local wall-following direction, to prevent a long-path length. Their *CautiousBug* will not choose a direction based on the angle of attack on the hit-point, as DistBug, but will first do a spiral search along the border, with the hit-point in the center. A disadvantage of this method seems that the spiral search by itself

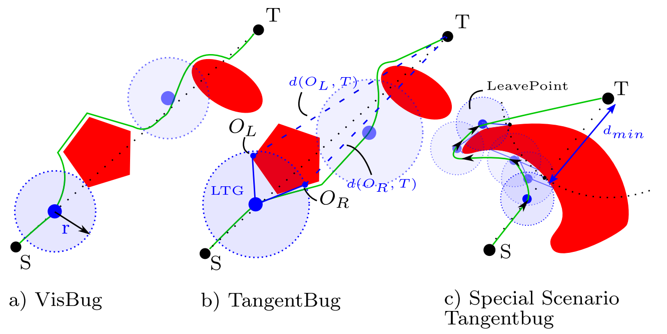


Fig. 5. The Bug Algorithms developed with obstacle detection with range sensors: (a) VisBug and (b) TangentBug. The S and T depicts the start and target position respectively. r stands for radius of the range sensor. LTG stands for local tangent graph and O_L and O_R stand for the left and right border of the detected obstacle within the range sensor respectively. $d(O_R,T)$ and $d(O_L,T)$ stand for the distance between the left and right obstacle boundary to the target, respectively. (c) A close up of a scenario in which Tangent bug is able to handle local minima.

will also produce a long path, therefore it has less of an advantage over Tangentbug. A newer variation is *InsertBug* by Xu and Tang [24], which navigates by means of way-points, placed on a safe distance from the obstacle's boundary. This could be seen as a version of TangentBug that adds a safety margin to each obstacle detected.

2.3. Special Bug Algorithms

Some BAs either take a special approach or are combined with other existing methods (*HybridBugs*). Lee et al. [25] used fuzzy logic with an adjusted Com method, a.k.a *FuzzyBug*. Assuming the BA is equipped with two single-beam sensors, pointed forward on both sides, it can detect if an obstacle is closer to its right or left. Based on a fuzzy membership function, FuzzyBug decides to follow the obstacle's boundary on its right or left, which a similar approach to DistBug's strategy. Noborio et al. [26] developed HB-I, which is another HybridBug. After each hit-point of the obstacle, HB-I moves along the border in both directions until it hits a corner. It will then select the best direction first, based on the best-first search of a decision tree. Xu [27] used a different approach with *RandomBug*. Once it detects an obstacle, it generates random points within the range of its sensor. From these points, RandomBug selects the optimal one, dependent on how close the point is to the target, and generates a motion vector towards it. This produces a path quite similar to InsertBug, but the process is highly related to rapidly-exploring random trees [28].

Taylor and LaValle [29] developed *IBug*, which is short for Intensity-bug. Its only information about its target is a wireless beacon on a specified location, of which it will navigate towards by means of the signal strength. Since they assume that IBug can make use of a "tower-orientation" sensor, the agent will move towards the beacon location. When it leaves the obstacle, IBug will temporarily save the value of the intensity (i_H) at that very moment. Here, a high intensity (signal strength) means a short distance to the target and a low intensity a large distance. While the robot follows the obstacle's boundary (always with a predefined Local direction of CW or CCW), it compares the current intensity level with i_H, as well as the current intensity and of time-steps back. If the signal strength decreases after increasing, the agent will have detected a local minima and a leave-condition is triggered, but only if the current measured intensity is larger than i_H. Although the leave condition is different, the latter comparison of intensity levels at the hit- and leave-points is quite similar in approach to Com1, with intensities substituted for distances.

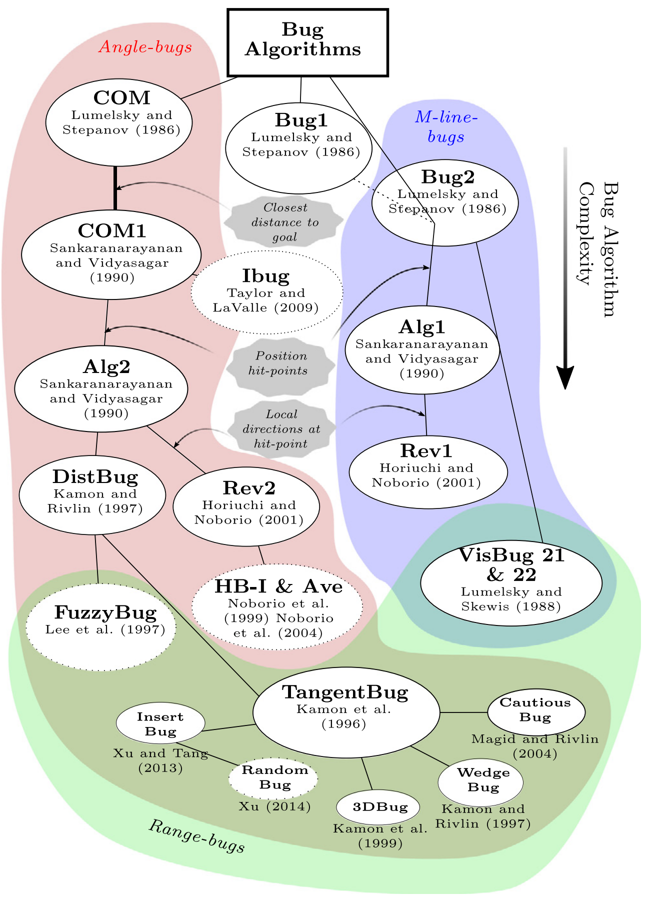

Bug Algorithms
Angle-bugs
M-line-bugs
Range-bugs
Bug Algorithm Complexity
COM Lumelsky and Stepanov (1986)
Bug1 Lumelsky and Stepanov (1986)
Bug2 Lumelsky and Stepanov (1986)
Closest distance to goal
COM1 Sankaranarayanan and Vidyasagar (1990)
Ibug Taylor and LaValle (2009)
Alg1 Sankaranarayanan and Vidyasagar (1990)
Position hit-points
Alg2 Sankaranarayanan and Vidyasagar (1990)
Local directions at hit-point
Rev1 Horiuchi and Noborio (2001)
DistBug Kamon and Rivlin (1997)
Rev2 Horiuchi and Noborio (2001)
VisBug 21 & 22 Lumelsky and Skewis (1988)
HB-I & Ave Noborio et al. (1999) Noborio et al. (2004)
FuzzyBug Lee et al. (1997)
TangentBug Kamon et al. (1996)
Cautious Bug Magid and Rivlin (2004)
Insert Bug Xu and Tang (2013)
Random Bug Xu (2014)
3DBug Kamon et al. (1999)
Wedge Bug Kamon and Rivlin (1997)

Fig. 6. An overview of all the discussed Bug Algorithms (BAs) in Section 2. These BAs are presented in a development tree of increasing complexity. It makes a distinction between Angle-Bugs, BAs that move to the target's azimuth direction, M-line-Bugs, BAs that use an M-line to navigate, and Range-Bugs, which use a range sensor to detect obstacles. The BAs noted in a dotted circle are special/ hybrid-bugs. The gray blobs indicate the type of memory and leave-condition added to the method. The latter is only shown until Rev1&2.

2.4. Overview Bug Algorithms

The BAs discussed in the previous sections are visualized in the overview of Fig. 6, where they are connected based on their development and added features. We subdivide the algorithms in a few major categories. The main division already started in the paper of Lumelsky and Stepanov [13], where they presented Com, Bug1 and Bug2. Com led to a series of Bug Algorithms that navigated in an azimuth angle towards the goal whenever it had the chance to do so. Hence, here we categorize them as *Angle-Bugs*. Lumelsky and Stepanov [13] realized that their next creation, Bug1, would create long trajectories by default. The community seems to have agreed as no extension or variation of Bug1 was developed here after, so therefore no similar Bug Algorithms have emerged since. Lumelsky and Stepanov [13]'s alternative solution, Bug2, did have more potential, so new variations of *M-line-Bugs* have been presented, leading to a separate category of BAs in the overview.

Com is arguably the simplest BA, as it uses no memory, nor determines any M-line. With Com1, Sankaranarayanan and Vidyasagar [15] added a distance-based leave condition, where it will only leave the obstacle if it reaches a position closer to the goal than before. Sankaranarayanan and Vidyasagar [15]'s Alg1 and 2 are given additional memorization tasks as increments. Not only do these BAs remember the previous minimal distance to the goal, but all the hit-points' locations in between as well to reverse their local direction. Horiuchi and Noborio [17] made Rev1&2 remember their last local wall-following direction, together with the local direction chosen at each hit-point. DistBug uses a more memory-friendly approach to determine its local wall-following direction, which is purely based on the detected slope of the approached obstacle. Eventually, the BAs started to use range-sensors at one point, creating the *Range-Bugs* category.

We can make some general observations about the overview in Fig. 6. Firstly, there are more Angle-bug-based BAs than M-line bugs. This is likely thanks to their more intuitive and less restrictive navigation strategy towards the target. Secondly, more and more features are added to the BAs as time progresses. Each new BA builds on top of an other, adding new leaving conditions and memory capabilities, therefore increasing the bug's complexity in the hope to find more efficient paths. The sole exception is the more recent Ibug, which is a recent BA variation, but is only one step away from Com1 in complexity.

3. Bug Algorithms for robotic navigation

The BAs presented in the last section are considered as a potential new robotic path planning paradigm, because of their simplicity and low memory footprint. We first will discuss how the principle of BAs translates to realistic operating conditions. Afterwards, existing BA robotic implementations will be presented and discussed on how well these studies represent real-world scenarios.

3.1. Bug algorithms in real-world conditions

In the earlier literature overview in Section 2, it seems to be the case that BAs heavily rely on perfect localization. They almost all assume that the BA does not know the exact location and shape of the obstacles, however they almost all need to know the exact coordinates of their goal and their own position. The latter is used for more aspects of BAs than first meets the eye. Angle Bugs need to know the distance and azimuth angle to the target at any point. M-line-Bugs (i.e. Bug2, Alg1, Rev1) both remember the exact line (and direction) between the starting position and the goal, and recognize if they have reached it. Hit-point memorizing BAs (i.e Alg1&2) need to match their current position estimate with previously hit-point coordinates. In a typical indoor GPS-deprived environment, obtaining and maintaining a world position is a significant challenge. Real-world robots that do not have maps then will need to rely on odometry, which is prone to errors and has the tendency to drift in time from the ground truth. Some BAs (i.e. Alg1&2 and Rev1&2) have to remember the exact coordinates of where they have been, which ensures a convergence to the target. This could be done by odometry as well, or by memorizing features in the environment with scene descriptors locally with SIFT [38] or globally with Bag-of-Words [39]. As with visual odometry, the descriptor's performance highly depend on the texture of the environment. Most BAs use a Distance-to-Target (DT) measurement in their leave-condition. Next to using the drift-susceptible odometry, they could also retrieve the DT in ways such as received signal strength intensity (RSSI) of Bluetooth [40] or Ultra-Wide Band (UWB, [41]). This does of course require the placement of a wireless transmitter at the target location. Moreover, none of these sensors have a perfect ranging measurement, with errors ranging from 10 cm to 5 m.

3.2. Existing implemented Bug Algorithms for robotic navigation

This section will look at current robotic BA implementation, either in a real-world environment or a simulated scenario. An overview of these methods is presented in Table 1, which lists the platform they used and shows the sensors the robot was equipped with for local obstacle sensing and global position estimations. Several works have focused on implementation of simulated systems. The work of Ebrahimi et al. [35] developed *Uavis-Bug* for a simulated MAV for visual guided navigation. However, they combined the BA with SLAM for the obstacle detection and boundary following, from which they used a potential force field to navigate around the obstacle. Moreover, they mentioned the requirement of a motion-capture-system if their technique would be implemented in the real world. Marino et al. [37] also mentioned that they assumed the existence of an UWB-localization system. Moreover, if the agent believes it is at the right goal position but on a different floor, it will use the Dijkstra method to compute the shortest path. Even though [35] and [37] acknowledged the limited sensing, computing and energy capability of MAVs, they still combine the efficient BAs with computationally-heavy navigation techniques.

With DistBug, Kamon and Rivlin [16] showed, as one of the first, a BA implemented on an actual wheeled robot, a Nomad200. In their paper they mention that the robot, while moving to the target, only responds to local measurements by the contact sensors. However, the robot always moves towards the target after boundary following, therefore, it must also know its own and the targets position in global coordinates. Although the paper of Kamon and Rivlin [16] has not specified this, their BA would need to use a global localization system. Zhu et al. [32], Kim et al. [33] and Gulzar et al. [36] have implemented a BA on autonomous real-world wheeled robots without a tele-operator. In all cases, they were using single beam range sensors and/or a laser scanner. Again, the exact location of the robots is needed in order for the BA to move towards the target.

Mastrogiovanni et al. [31] acknowledged that a robot would not be able to know its exact position, but would need to infer it from its noisy on-board sensors. They implemented μ*Nav* on a real-world wheeled robot, AmigoBot and a hexapod, Sistino. The first platform used ultra-sonic sensors for obstacle detection and wheel-odometry for global localization. Since the wheeled robot combined its wheel-odometry with the azimuth angle toward the target, it could reach the target location from one room to another, even if the orientation was manually perturbed by the researchers. However, the operation area only spanned across 2 rooms and no notion was given of what the navigational limit was, based on accumulated errors of odometry drift. Their second robot, the hexapod, was not able to use odometry, so the azimuth angle had to be given by an external source through photo diodes.

Taylor and LaValle [34] implemented IBug on a small wheeled robot for several small-scale environments. In their previous work [29], they described the BA to be suitable for navigating towards a single wireless beacon. Nevertheless, for the test on a real robot, a Lego-Mindstorm-based platform, Taylor and LaValle [34] used an infra-red (IR) beacon instead. It proved to be challenging for their tests to use the signal strength of i.e. a WiFi beacon at a large range. The use of the IR beacon did necessitate a constant line of sight, which required the obstacles and walls to be lower than the robot itself.

If we look at the existing implementations of BAs in real or simulated robots, they all assume or need explicit global localization, either by a UWB localization system, a motion capture system or a guiding navigator, for the exception of IBug, which used a visual beacon. Mastrogiovanni et al. [31] is actually the only one that used the odometry of a (bigger) wheeled robot to

Table 1
Robotic implementations of various Bug Algorithms (BAs). These are evaluated on the type of platform used, whether the environment was real or simulated and which BA type was used. Moreover, it shows the used local sensors for obstacle detection and the used global sensor for a position estimate.

Paper	Platform	Environment	Bug algorithm	Local sensors	Global sensors
Kamon and Rivlin [16]	Wheeled robot	Real	DistBug	Range sensors	Global localization (system not given)
Laubach and Burdick [30]	Microrover	Real	RoverBug (wedgebug extended)	Stereo images	Guiding operator (First person view)
Mastrogiovanni et al. [31]	Wheeled robot Hexapod robot	Real	μNAV	Ultrasound range sensor Wheel odometry	Azimuth angle by photo diodes(only for hexapod)
Zhu et al. [32]	Wheeled robot	Real	Bug2 and a DistBug variant	Laser scanner (180 deg)	Global localization (system not given)
Kim et al. [33]	Wheeled robot	Real	Tangentbug (adjusted)	Ultrasound range sensor Wheel odometry	Global localization (system not given)
Taylor and LaValle [34]	Wheeled robot	Real	Ibug	Touch sensors	IR beacon
Ebrahimi et al. [35]	Quadcopter	Simulation	UavisBug	Camera	Motion capture system
Gulzar et al. [36]	Wheeled robot	Real	Not given	Ultrasound	Motion capture system
Marino et al. [37]	Quadcopter	Simulation	Bug2	Laser scanner (180 deg)	UWB localization

recover its own position and to update the azimuth angle towards the target; however, the real-life test was too small to draw any conclusions about the suitability of BAs for robotic navigation. In the comparative study, presented in the next sections, we will test various BAs with varying amounts of odometry drift, recognition failures and distance noise. This will show that these real-world conditions will have significant effect on the BAs' performances.

4. Experimental set-up comparative study bug algorithms

In this paper, we study whether BAs could be used for real-world robotic navigation. Most indoor environments have more complex obstacle configurations than an open environment with a few convex obstacles. There are many situations where the robots could get stuck on their way, particularly in rooms. In this section, we will present our motivation for this study and the chosen set of BAs to be evaluated. We will then provide the details of the simulation used and the procedural environment generator for typical indoor environments. Afterwards, the implementation specifics of the BAs will be presented, by explaining a wall-following paradigm, which is the foundation for all BAs to be implemented.

4.1. Motivation and choice bug algorithms

There have been previous comparative studies between existing BAs. In the paper of Noborio et al. [5], Class1, Bug2, Alg1, Alg2 and HB-I, were compared and evaluated on their generated path-length within a complex maze. They based their observations on just one indoor environment. A newer comparative study was performed by Ng and Bräunl [6], on: Bug1, Bug2, Alg1, Alg2, DistBug, TangentBug, OneBug, LeaveBug, Rev1, Rev2 and Class1. They presented the BAs with four types of environments and recorded the total path length for each run. However, the performance could not be adequately compared due to the inconsistent results. In this paper, we test a set of BAs in hundreds of procedurally generated environments, so we can statistically evaluate their performances. Moreover, set up the simulation environment to include the possibility to simulate real-world properties such as sensor noise and odometry drift.

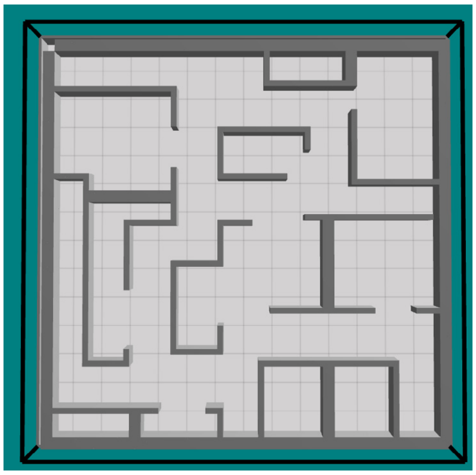
In the overview of the BA-methods existing today (Section 2.4), it can be seen that many of the methods are natural increments of one another with increasing complexity. If the fundamental BAs can be tested with these real-world conditions, we would automatically find the issues that their descendants are facing as well. Specifically, we have selected Com, Com1, Bug2 and Alg1&2. Range-bugs will not be considered as the selected BAs are the base of those more complex versions. Moreover, the selected BAs presents a mix of different types of strategies (Angle-Bugs and M-line-Bugs) and memory-use (distance and/or hit-points). We will exclude bugs that determine a local wall-following direction, as the policy for this choice is heavily influenced by the structure of the environment, as previously discussed in Section 2.1. Moreover, any special bugs will not be considered either, since they contain aspects and enhancements that no other BA-related research followed up on.

4.2. Simulation and procedurally environment generator

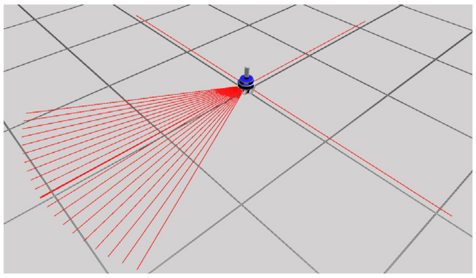
It is our ambition to test the earlier mentioned BAs in a simulator with realistic and swift physics calculations. ARGoS, a multi-physics robot simulator developed by Pinciroli et al. [42], is used for our comparative study. Its main trait is its efficiency, which enables the simulator to run many times faster than real-time, which will be essential if the BAs are evaluated in many environments. Although ARGoS does have the capability to incorporate its own, C++ based, controller for the robots, the ROS framework is used to enable Python-based controllers. The ROS messaging system is also ideal to modulate whether a new environment needs to be generated, to vary the measurement noise and select the right bug algorithm.

Since the BAs will be evaluated in many indoor environments, it would be unfeasible to design these by hand. Therefore, a procedural indoor environment generator will automatically generate a new arena for the bugs to navigate in. This process is depicted in Appendix B, where a standard indoor environment with corridors and rooms can be generated in different configurations. Rooms are especially challenging, as they can lead to agents getting stuck in loops, which will showcase the strengths and weaknesses of the evaluated BAs. The resulting environment in the ARGoS simulation is shown in Fig. 7(a).

The ARGoS FootBot is used for our experiments, which is a simulated wheeled mobile platform (see [42] for specifications). The FootBot contains many options to attach various types of sensors, however for our experiments we will only use the proximity sensors. We adapted FootBot to turn the proximity sensors into single beam range sensors with a maximum measurable distance of two meters, placed in the configuration shown in Fig. 7(b). The robot has two separate single beam range sensors located on each side and 20 range sensors pointing to the front in a wedge shape. This is to simulate a depth sensor/stereo-camera for obstacle detection with a few additional range sensors on the side. Since the robot must move towards the range-wedge configuration, its movement will be non-holonomic.



(a) Generated environment in ArGos



(b) Modified ArGos Foot-bot

Fig. 7. (a) The resulting environment from Fig. B.17(f) generated within the ARGoS simulator and (b) a modified Foot-bot simulated robot with range sensors (red lines) used for wall following. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.3. Implementation details bug algorithms

The most important element of any BA, is the ability to follow a boundary of an obstacle or wall. Based on the robot configuration in Fig. 7(b), we developed a wall following principle. Fig. 8(a–f) shows the wall following of the footbot for a right-sided local direction and Fig. 8(g) shows the state machine, which can also be found as pseudo code in Appendix C.2, Alg. C.1. If the robot moves forward and hits a wall, like in Fig. 8a, the angle of the wall can be estimated by using a RANSAC line-fit method [43] in the wedge of range sensors.[3] This is done so that the true distance $d(x, O_\perp)_R$ can be estimated from the robot to the wall.[4] If this distance becomes smaller than d_{ref}, the preferred distance from the wall, it will keep turning either CW or CCW until it is aligned with the wall. Fig. 8(b) and (c) shows this alignment for a right-side local direction. This will be the case if the measurement of the side range sensor r_s is equal $r_f \cdot \cos\beta$, where r_f is the element from the range wedge that is the closest to r_s and β is the angle between r_s and r_f.

After the robot is aligned, it will need to follow the wall, as in Fig. 8(c). Now the true distance to the wall $(d(x, O_\perp)_C)$[5] will be calculated as follows:

$$d(x, O_\perp)_C = \frac{r_s \cdot r_f \sin\beta}{\sqrt{r_s^2 + r_f^2 - 2 \cdot r_s \cdot r_f \cos\beta}} \tag{1}$$

The derivation of the latter equation can be found in Appendix C.1. The FootBot will maintain $d(x, O_\perp)_C$ to be near d_{ref}, and to keep being aligned in the process. However, since the robot's heading and the measurements of r_s and r_f are coupled, a separate control heuristic is developed to make the wall alignment possible. The details of this wall-alignment method can be found in Appendix C.2, Alg. C.2.

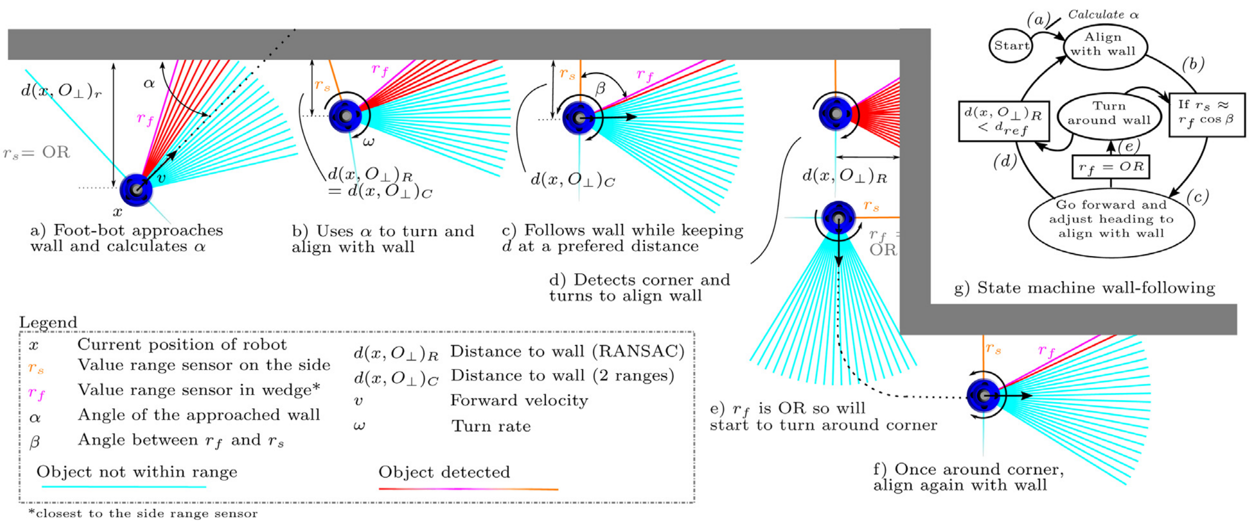
When the FootBot hits another wall during its forward motion, as in the corner in Fig. 8(d), while in its wall-following state, it will turn away from the wall until it is aligned (similar condition as with Fig. 8(b)). If during a forward motion, the front-range sensor is out of range, as in Fig. 8(e), the foot-bot will initiate a wide turn, to find the wall on the other side as in Fig. 8(f). The state machine for the wall-following behavior can be found in Fig. 8(g), of which the pseudo code can be found in Appendix C.2, Alg. C.1.

This control heuristic should result in a robust wall-following behavior, in particular for indoor environments with straight

[3] Since RANSAC uses random samples to determine the slope of the plane, some stochasticity is expected in the wall-following behavior.

[4] This only goes with the assumption that the robot will always encounter walls and no single objects, such as plants. The later will not be simulated in ARGoS; however, a classifier able to distinguish walls from small obstacles, must be added if this principle is implemented on a real robot.

[5] The R and C subscript of $d(x, O_\perp)$ enables separation of the two methods (RANSAC or only 2 ranges) of retrieving the true distance to the walls.



d(x, O⊥)r
α
rf
rs= OR
v
x
a) Foot-bot approaches wall and calculates α
rs
rf
ω
d(x, O⊥)R = d(x, O⊥)C
b) Uses α to turn and align with wall
rs
β
rf
d(x, O⊥)C
c) Follows wall while keeping d at a prefered distance
d(x, O⊥)R
d) Detects corner and turns to align wall
rs
rf = OR
e) rf is OR so will start to turn around corner
rs
rf
f) Once around corner, align again with wall
(a)
Calculate α
Start
Align with wall
(b)
d(x, O⊥)R < dref
Turn around wall
If rs ≈ rf cos β
(d)
(e)
rf = OR
(c)
Go forward and adjust heading to align with wall
g) State machine wall-following
Legend
x Current position of robot
rs Value range sensor on the side
rf Value range sensor in wedge*
α Angle of the approached wall
β Angle between rf and rs
d(x, O⊥)R Distance to wall (RANSAC)
d(x, O⊥)C Distance to wall (2 ranges)
v Forward velocity
ω Turn rate
Object not within range
Object detected
*closest to the side range sensor

Fig. 8. (a–f) Schematics to explain the wall-following paradigm for a right-sided local direction with (g) the corresponding state machine. OR stand for out of range.

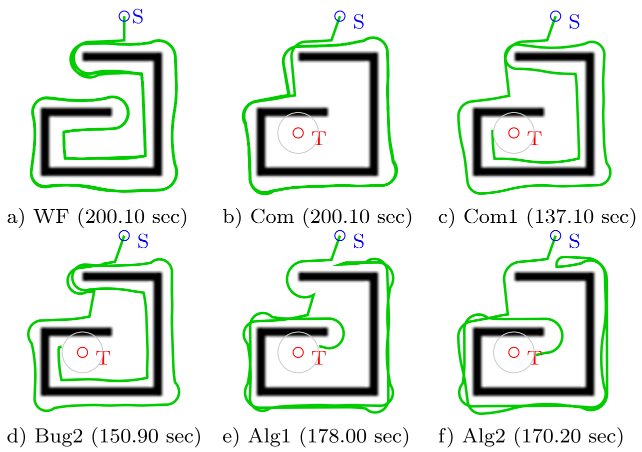


Fig. 9. The results of (a) the wall-following only (WF) and the implemented bug algorithms that use the same WF (b–e) as part of their navigation strategy. The time limit is 200 s.

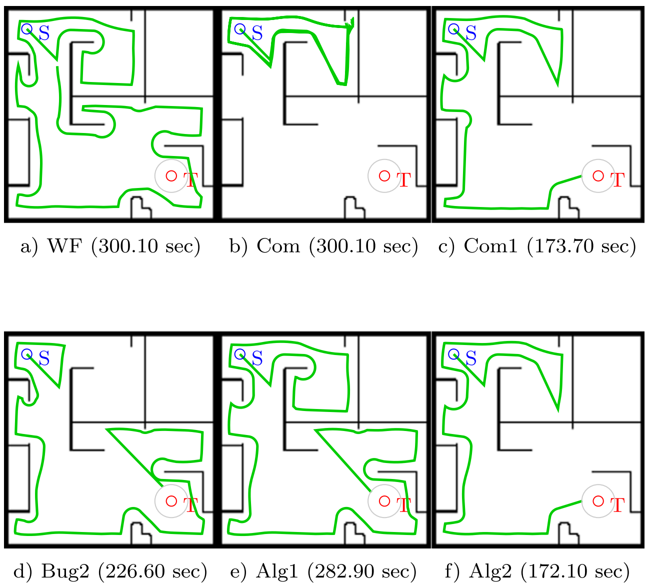


Fig. 10. Behaviors of the implemented (a) wall-follower (WF) and Bug Algorithms (BAs): (b) Com, (c) Com1, (d) Bug2, (e) Alg1 and (f) Alg2 in one generated environment. The BA starts in the top left corner at Start (S) and ends withing 1 m radius from the Target (T), with a time-limit of 300 s.

walls. The resulting wall-following behavior is shown in Fig. 9a. Here it can be seen that the wall-following produces a smooth path all along the walls of the mirrored "G". All the implemented bug algorithms, from which the pseudocode can be found in Appendix A.1, will make use of this exact same wall following behavior in their state machine.[6] The resulting trajectories of the implemented BAs in the ARGos simulated environment are shown in Fig. 9(b–f).

5. Experimental results of bug algorithms in real-world conditions

In this section, the BAs will be compared against each other on a wide range of procedurally generated environments. Moreover, we will investigate how sensitive these algorithms are to real-world conditions, subjecting them to the experimental setup explained in Section 4. The selected BAs: Com, Com1, Bug2, Alg1 and Alg2, will be evaluated first with perfect localization. After that, they will be subjected to increasing odometry drift, varying hit-point recognition failures and Distance-to-Target (DT) noise. From this, we hope to properly evaluate the BAs ability to handle real-world conditions.

5.1. Experiments with perfect localization

The implemented BAs' performances are evaluated in 200 procedurally generated environments, with a constant size of 14 by 14 m. Each BA will have one chance to navigate through the same environment with a time limit of 300 s. This should be sufficient to reach the target, while preventing the simulation to run endlessly, if one of the BAs gets stuck in a loop. Each BA's success percentage is recorded, which is the percentage of when the target is reached out of the 200 environments. Fig. 11(a) shows the percentage of BAs that made it to the goal within the required 300 s with perfect localization (indicated with $\sigma = 0.00$), where the goal is considered reached if the BA is able to get within one meter radius.

The BA's total trajectory length is recorded as well, which is normalized by dividing by the optimal path length as calculated by the A* path planning algorithm. A* will get an occupancy grid identical to the procedurally generated environment and is able to visit all the 8 neighboring cells at each step.[7] Note that the grid not available to the bug algorithms by any means. The normalization is applied in order to compare the performances adequately across the generated environments, as the optimal path will be different at each iteration. Fig. 11(b) shows a box-plot of the length of the BAs' trajectories with perfect localization. For all BAs, all path-lengths are taken into account, including the ones that did not reach the goal. Although this skews the statistics, a time limitation of 300 s will be held constant throughout the experiments to ensure consistency.

In the simulation set-up, the BA would need to leave the walls physically in order to reach the goal, which results in the wall-follower (WF) to not be able to complete the experiment at all. Com is only capable to reach the goal about 60% of the time with perfect localization. Com does not use memory and this results in it occasionally getting stuck in loops as shown in Fig. 10(b). The last four BAs, Com1, Bug2, Alg1 and 2, have a success percentage of around 90% in Fig. 11(a) with perfect localization.

In Fig. 10(d) and (e), it can be seen that Alg1 and its ancestor Bug2 need to find the M-line first before it can leave the wall. However, this restriction results in longer trajectories. Com1 and Alg2, on the other hand, will move towards the target if the chance arises, hence have more leave-opportunities along their path (Fig. 10(c) and (f)). The outcome is that in the 200 generated environments, Com1 and Alg2 have a shorter path-length than Bug2 and Alg1 in average (Fig. 11(b)).[8]

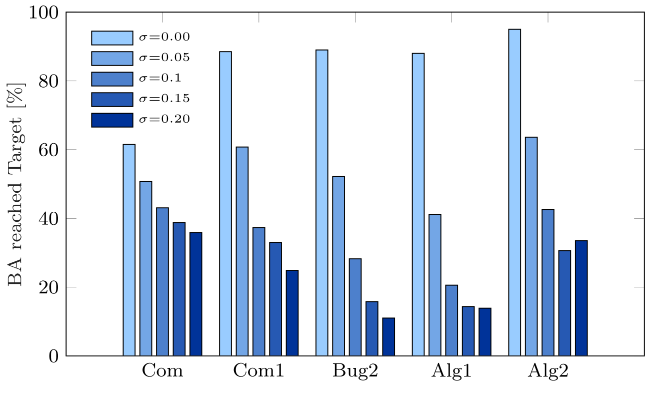
5.2. Experiments with odometry drift

In this paper, we test BAs' potential for real-world navigation purposes. Therefore, we have added more realistic elements to the simulation, based on our discussion in Section 3.1. In the

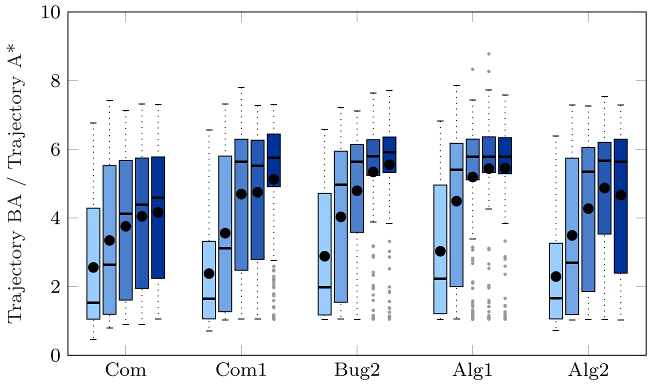
[6] The bug algorithms are implemented in Python, but to test the claim of the computational efficiency of bug algorithms, we have also implemented Com1 on a STM32F4 processor (specs: 168 MHz processor speed and 128 kB RAM memory). The average computation time is 0.04 ms.

[7] An 8-connection A* will cause the path to go through a corner of an obstacle. The grid that is available A*, will include padded wall and obstacles compared to the actual environment where the Bug Algorithms will navigate in.

[8] A bootstrapping based statistical similarity analysis of both the success rate and the trajectory length can be found in Appendix D.1.

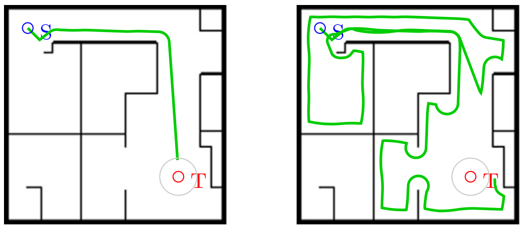


(a) Success percentage.



(b) Normalized trajectory length

Fig. 11. The (a) percentage of the Bug Algorithms Com, Com1, Bug2, Alg1 and Alg2, which made it to the goal of increasing velocity measurement noise (σ) which causes odometry drift, and (b) the trajectory length normalized by the optimal path calculated by A*.


a) σ=0.05 (300.10 sec) b) σ=0.10 (59.70 sec)

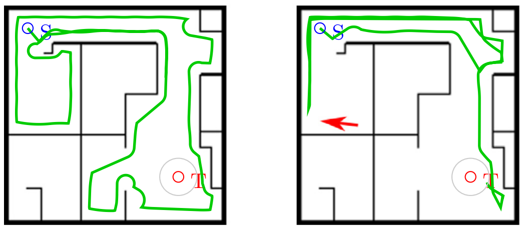

c) σ=0.15 (70.50 sec) d) σ=0.20 (300.10 sec)

Fig. 12. Example of Alg2 in environment # 123 of the experimental testing, with increasing noise variance of σ = (a) 0.05, (b) 0.10, (c) 0.15 and (d) 0.20. The BA starts in the top left corner at Start (S) and ends withing 1 m radius from the Target (T), with a time-limit of 300 s. In (d) Alg2 suddenly turns 180 degrees on the left side of the environment, without having seen that hit-point before.

absence of an exact global position, BAs will need to rely on odometry. Therefore, this section will investigate the effects of odometry drift. We assume that the BA will know its own and the target's position at the start of the experiments, but it has to keep them up-to-date with its own, noisy, velocity measurements. For these experiments, we assume that the position estimate is acquired by the latter assumption, namely:

$$\tilde{\mathbf{x}}_t = \tilde{\mathbf{x}}_{t-1} + \dot{\tilde{\mathbf{x}}}_{t-1}, \tag{2}$$

where $\tilde{\mathbf{x}}_\mathbf{t}$ is the x- and y-position estimate at a given time. $\dot{\tilde{\mathbf{x}}}_{t-1}$ is assumed to be $\mathcal{N}(\mathbf{u}_{t-1}, \sigma)$, where $\mathbf{u}$ is the actual velocity, from which the outcome on the system consists of noise with a standard deviation of σ.

Fig. 11 shows the impact on the performances of the BAs when exposed to odometry drift due to noisy velocity estimates, with a σ of 0.05, 0.10, 0.15 and 0.2. In Fig. 11(a) indicates a significant drop in all the BAs' success percentage with an increasing σ. In Fig. 11(b) we see that it has a large effect on the trajectory length overall, although there is a less significant degeneration of the Angle-Bugs' performance (Com, Com1 and Alg2). Bug2's and Alg2's performances took the deepest dive with a relatively small increment of the odometry drift, whereas Com's performance only gradually decreased. As Com does not save any position or distance-to-goal at hit-points, only its bearing estimate towards the goal is affected by faulty velocity estimates, resulting in the simplest BA outperforming the rest with σ > 0.05. Alg2 already lost its advantage to recognize previously visited places, as its success-rate is similar, if not lower, than Com1 at a σ of 0.2.

However, both Alg1 and Alg2 show signs of stagnation from σ = 0.15 and on, as their performances does not seem to decrease any further and even seem to improve slightly. At that point, it could be that it would accidentally recognize a previously hit-point at a location where it has not been before due to the odometry drift. Although seemingly unwanted, this randomness could have helped the BA to get out of difficult situations, as in Fig. 12 with Alg2.[9]

5.3. Experiments with false positive and false negative recognition rate

BAs can also recognize previous hit-points based on scene-recognition. In this paper, we will not use the techniques and descriptors discussed in Section 3.1, but will simulate their performance through false-negative (FN) and false-positive (FP) recall rates. With an increasing probability (p) of a uniform distribution, the chances of a previously visited hit-point being falsely recognized at a different location (FP) or not being recognized at the right position (FN) will increase.

Of the implemented BAs, only Alg1 and Alg2 specifically use previously visited locations to change their local wall-following direction from right- to left-sided. In Fig. 13, they are being evaluated with an increasing p(FP) in Fig. 13(a&b) or p(FN) in Fig. 13(c&d) over 100 generated environments. At a p(FP) = 0.005, there is a chance of FP occurring 1–2 times (0.5%) during the runtime of 300 s and at p(FP) = 0.025 a chance of 7–8 times (2.5%). At p(FN) = 0.2, every time the BA encounters a previous hit-point, there is a 20% chance that it will not recognize it and at p(FN) = 1.0, the hit-point will never be recalled. Fig. 13(a) and (b) shows that increasing the p(FP) has more effect on the performance of Alg1 than Alg2. This can be due to Alg2's fewer leave-restrictions, which makes it less sensitive to more frequent occurrences of FP. Fig. 13(c) and (d) show that both Alg1 as Alg2 seemed to be hardly effected by an increasing p(FN). The only trend that could

[9] Statistical correlation analysis of the effect of the increasing odometry noise both the success rate and trajectory length can be found in Appendix D.2.

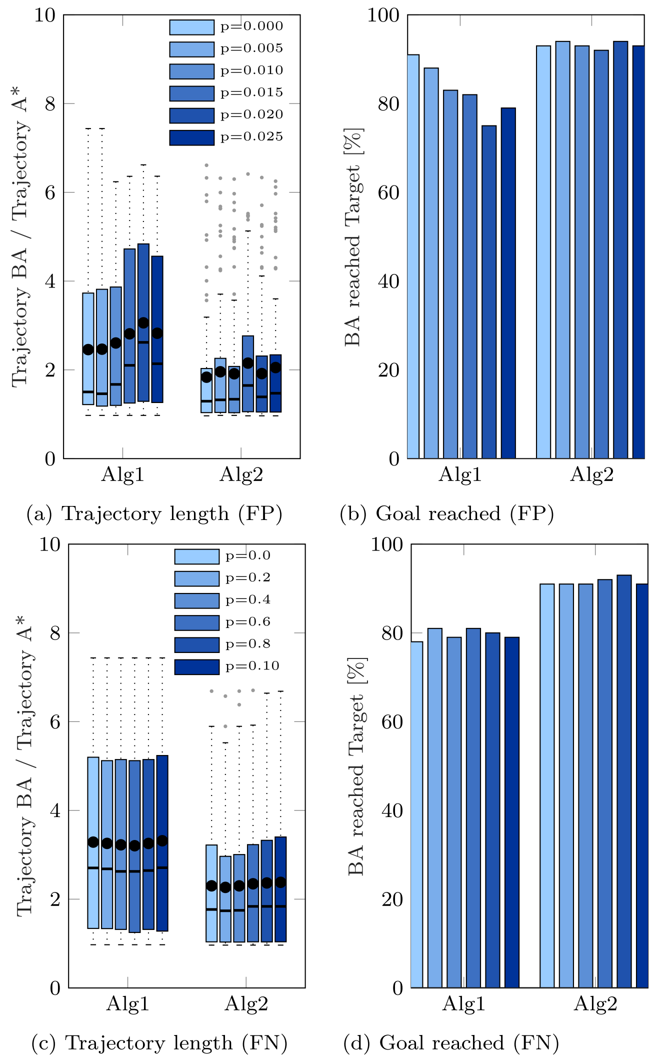


Fig. 13. The (a&c) measured trajectory length and (b&d) percentage of Alg1 and Alg2 reaching the goal, with a varying chance (p) of a false-positive (FP - a&b) or a false-negative (FN - c&d) of a previous recognized point to occur.

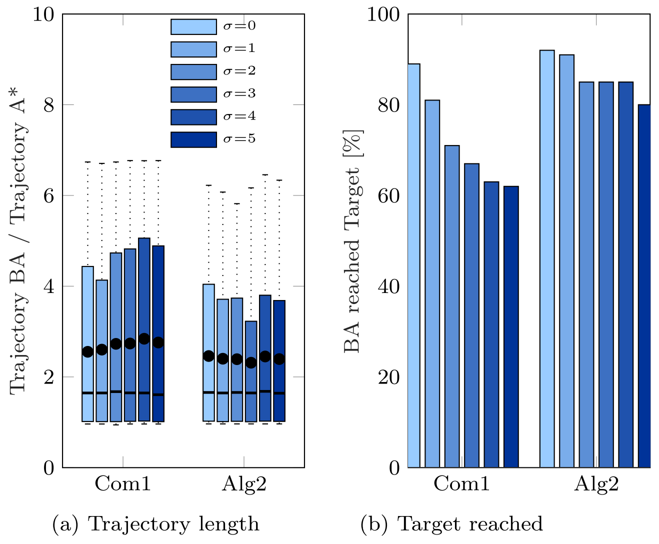


Fig. 14. The performance of Alg2 and IBug with varying distance measurement noise (σ) in meters, in the (a) normalized trajectory length and percentage (b) of bugs who made it to the goal.

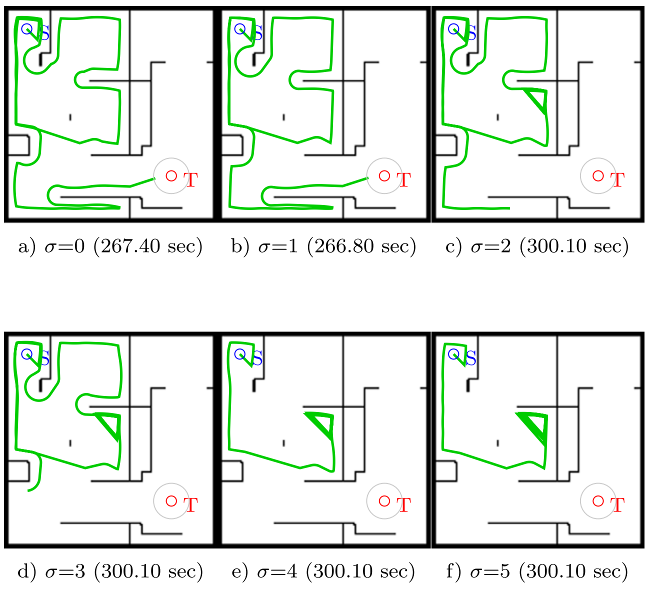


Fig. 15. An example environment with the trajectories of Com1 with increasing distance measurement noise variance (σ) in meters of (a) 0, (b) 1, (c) 2, (d) 3, (e) 4 and (f) 5 m. The BA starts in the top left corner at Start (S) and ends within 1 m radius from the Target (T), with a time-limit of 300 s.

be noticed is for Alg2 as the variance of the trajectory length slowly creeps up with an increasing p(FN) in Fig. 13(c). With p(FN) $= 1.0$, Alg1 and Alg2 are not able to recall previous hit-points, which makes them identical to their ancestors Bug2 and Com1. However, no significant difference can be noticed in the success rate of Fig. 13(d) with p(FN) $= 0.0$ and 1.0 for both Alg1 and Alg2.[10]

5.4. Experiments with distance measurement noise

BAs could also use a Distance-to-Target (DT) measurement, so here we assume that the agents are carrying a sensor able to determine this. Com1 and Alg2 both save previous DT measurements to prevent getting stuck in a loop in some situations. In Fig. 14, we are showing the (a) trajectory length and (b) success rate of the BAs subjected to increasing DT noise, while keeping both the velocity measurement noise (odometry drift) and the FP & FN rate at 0.0. The noisy DT measurements ($\tilde{d}(x,T)_t$) at time t are modeled by $\tilde{d}(x,T)_t = \mathcal{N}(d(x,T)_t, \sigma)$, where $d(x,T)_t$ is a scalar that stands for the true DT at time t and σ is the standard deviation of the noise. The degrading performance in both trajectory length and success percentage for increasing σ is more noticeable for Com1 than for Alg2. Com1's only mechanism to get out of a potential loop is to compare its current DT with a saved one to decide when to leave the wall. Once it is gradually losing this capability with the noisier DT measurements, its behavior will become more and more similar to Com's, as observed in Fig. 15. Moreover, Com1's success percentages in Fig. 15(b) drops to around 60 percent at a $\sigma = 6$ m, which is equivalent to Com's score in Fig. 11(b) with perfect localization. Alg2 is less affected by the increasing DT noise, which is likely because it can rely on memorized position as an additional leave condition.[11]

[10] Statistical correlation analyses of the effect of the increasing recognition failure rate on both the success rate and trajectory length can be found in Appendix D.3.

[11] Statistical correlation analysis of the effect of the increasing DT measurement noise on both the success rate and trajectory length can be found in Appendix D.4.

6. Discussion

This section will discuss both the experimental set-up and results and will accommodate the observations made in Section 3. The modeled real-world conditions will be discussed first, including the implementation details of the simulation and the chosen noise-models and Bug Algorithms (BAs). Here we will give some suggestions for future development in this topic. Afterwards, we will discuss the results from our experiments, from which we will determine which aspects of BAs are suitable for real-world scenarios.

6.1. Modeling real-world conditions

BAs are potentially an ideal indoor navigation paradigm for tiny robotic platforms with limited resources. Although the results show that the paths generated are sub-optimal compared to path-planning algorithms as A*, no map is needed for navigation. Nevertheless, we established that the BAs, presented in Section 2, tend to over-rely on a perfect localization, which cannot be guaranteed for all indoor environments. If no global localization system is available, the BA needs to rely on its noisy on-board sensors to know where it is and integrate this knowledge with the target's position. In Section 3, we reflected on several issues that a robotic implementation of a BA will come across. This includes: an increasing odometry drift, a mismatch between its measured position and the ground truth; recognition failures, i.e. when it fails to recognize a previous location or falsely detects one; and noisy distance to target measurements, which could interfere with the suitable leaving-condition. There are other types of sensor-noise and failures to consider, such as the noise in the laser-range sensors or (stereo-)cameras for (local) boundary/wall following. However, in this paper, we focused on the global position estimation instead, as this is an issue that all bug algorithms must deal with.

For the experiments themselves in Section 5 we used simple noise models, i.e. using a Gaussian probability distribution for the odometry drift and noisy range measurements, and a pseudo-random number generator for FNs and FPs occurrences. Future work could look at more realistic noise characteristics. For ground-bound robots, for instance, wheel slippage is determined by the materials used and the friction with the floor. If visual-odometry is used, Gaussian noise could very well be applied, however the texture of the environment is crucial to the variance. The FP & FN occurrences are also very much determined by the features of the environment, as aliasing could occur at areas that are very similar. There is no equal probability of these failures to happen throughout the trajectory of the bug. Moreover, distance measurements by radio beacons not only suffer from regular noise around the mean, but have to endure a whole range of disturbances. This includes uneven directional propagation noise, the reflection off the walls and interference of other signals. For the experiments in this paper, we wanted to have more control over the noisy measurements to find a clear correlation between the noise severity and the performance of the BAs, so we restricted ourselves to use the basic versions of the noise models.

The ARGoS simulator and environment generator was very useful for this paper's experiments, as it generated new environments at a high pace and run the experiments faster than real-time. This enabled us to test the BAs on hundreds of environments, leading to more reliable results. Nevertheless, further development of these experiments must be performed in a more realistic simulation, with more types of obstacles and visual representations, to induce more challenges of a typical indoor navigation task. Moreover, as mentioned earlier, we should also include more realistic sensor models to further increase the realism of the simulation.

6.2. Bug algorithms performance in simulated real-world conditions

Generally, our experiments showed that all BAs performed worse with a higher odometry drift, noisier range measurements and increasing failure cases. The most noticeable feature, is that the BAs did not all have a similar drop in performance, which is especially noticeable with increasing odometry drift in Fig. 11. Some had a more severe response than others, namely those using memory. Com, being the simplest of all BAs, started out as the worst one of the six, to the best performing with only standard deviation of 0.1 m/s in the velocity estimation. As it only uses the odometry to get a range and bearing to the goal and nothing else, there are less "bad" decisions it could make. Since odometry is likely to be noisy on very small robots, such simplicity may be the better strategy. Nevertheless, although Com is less influenced by odometry drift, it success rate still drops to 40%, which is a low score. In general, it is ill-advised to have BAs solely rely on odometry alone.

In this paper we have not experimented with different environment sizes. However, this could also affect the performance of the BAs dealing with sensor noise. In smaller environments for instance, the odometry drift will be less severe since the optimal paths to the goal will be shorter than for larger environments. The focus of this paper was on the onboard sensors used for position estimation and to fix elements like the environment dimensions in order to properly isolate and evaluate the associating issues with real-world-like conditions. However, for future research we should also experiment with varying sizes of the experiment space.

In Sections 5.3 and 5.4, we also assumed that the BAs will also have access to measurements other than odometry. Although a decrease in performance was noticed in all the tested BAs with these specific features (Com1, Alg1 and Alg2), it became evident that Alg2 is the most resilient algorithm. With increasing FN & FP occurrences, Alg1's performance was noticeably decreasing but Alg2 was hardly affected. This indicates that the M-line-Bugs, as Alg1, seem to have a disadvantage over Angle-Bugs, as Alg1, due to their restrictive leave-condition. This is also noticeable in Section 5.2, as M-line-Bugs suffered the most from increasing odometry drift. If real-world conditions apply, BAs should rather be able to leave the wall/obstacle whenever there is the possibility to do so.

The same goes for noisy distance-to-target measurements (Section 5.4), where Com1 is performing worse than Alg2. The reasoning behind this observation is simple: Alg2 is using more mechanisms to get out of complex situations, namely remembering range measurements and locations of previous hit-points. If one of these mechanisms perform badly, then Alg2 can fall back on the other one. Now these measures are operating separately and have a different behavioral outcome; however, it could be more beneficial to a BA if they were fused together or used for cross validation and checking if the BA is stuck in a loop. Nevertheless, it is of great interest to have multiple types of measurements to rely on, either concerning position of the robot itself or the relative position of the goal.

7. Conclusion

This paper investigates the potential of Bug Algorithms as a computationally efficient method for robotic navigation. Although the general idea behind the methods seems ideal for implementation of light-weight robots, the literature survey shows that many of their variants rely on either a global localization system or perfect on-board sensors. Our simulation experiments evaluated several implemented Bug Algorithms with varying noisy measurements and failure cases, which showed a significant performance degradation of all algorithms. This indicates that Bug Algorithms cannot simply be implemented as they are on a navigating robot, which has to rely on only on-board sensors without any external help. The experimental results did, however, shed some light on how these techniques can be enhanced. Simplicity is a key element, as the most basic Bug Algorithm, Com, was also the one that was the most resilient to odometry drift. Another crucial element is a robust loop detection system, where the robot should not just rely on one but on multiple measured variables, especially in realistic, noise-inducing, environments. Considering these observations in the design of new Bug Algorithms, will make them suitable for the autonomous navigation of tiny robotic platform with limited computational resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been funded by the NWO grant of Natural Intelligence. The research has been conducted at the Delft University of Technology, Faculty of AeroSpace Engineering, Department of Control and Simulation and Liverpool University at the Faculty of Computer Science, SmartLab. I would like to thank James Butterworth for helping me with setting up the ARGoS simulation and brainstorming about Bug-Algorithms in general.

Appendix A. Specifics Bug–Algorithms

A.1. Pseudo-code bug algorithms

The pseudo-code for Com, Com1, Bug2, Alg1 and Alg2, is listed in Algorithms A.1, A.2, A.3, A.4 and A.5 respectively. T stands for target and s_{WF} is a variable that determines if the wall-following is right- ($s_{WF} = 1$) or left-sided ($s_{WF} = -1$). r_{local} stand for local sensor measurements, which can be either contact- or range-sensors. $x_g lobal$ stands for the global position estimate of the Bug Algorithm. $d(H,T)_{prev}$ stands for the previous distance of the hit-point to target and $d(x_{global},T)$ stands for the current distance from BA to target. $list_{hp}$ stand for a list of previously encountered hit-points. v is the control output for the forward velocity of the robot and c_v is the fixed velocity constant. ω is the control output for the heading of the robot in rad/s and c_ω is a fixed rate constant, to control the speed of the robot's turns. See Fig. A.16 for the visual representation of the pseudo code in state machines.

Algorithm A.1 The pseudo-code for the state-machine of Com.

```
  Init: state = "forward", $s_{WF}$
Require: $c_v$, $c_\omega$, $x_{global} r_{local}$, $list_{hp}$
  function Com
     if state is "forward"  then
        $v \leftarrow c_v$
        $\omega \leftarrow 0$
        if Obstacle is hit then
           state ← "wall_following"
     else if state is "wall_following" then
        $[v,\ \omega] \leftarrow$ Wall_Following($c_v, c_\omega, s_{WF}, r_{local}$)        ▷ See C.2
        if Way towards T is free then                            ▷ Based on $r_{local}$
           state ← "rotate_to_target"
     else if state is "rotate_to_target" then
        $v \leftarrow 0$
        $\omega \leftarrow c_\omega$
        if Heading BA same as direction T then
           state ← "forward"
     return $v, \omega$
```

Algorithm A.2 The pseudo-code for the state-machine of Com1.

```
  Init: state = "forward", $s_{WF} = 1$
Require: $c_v$, $c_\omega$, $r_{local}$
  function Com
     if state is "forward"  then
        $v \leftarrow c_v$
        $\omega \leftarrow 0$
        if Obstacle is hit then
           $d(H,T) \leftarrow d(x_{global},T)$
           state ← "wall_following"
     else if state is "wall_following" then
        $[v,\ \omega] \leftarrow$ Wall_Following($c_v, c_\omega, s_{WF}, r_{local}$)        ▷ See C.2
        if Way towards T is free and $d(x_{global},T) < d(H,T)$ then
           state ← "rotate_to_target"
     else if state is "rotate_to_target" then
        $v \leftarrow 0$
        $\omega \leftarrow c_\omega$
        if Heading BA same as direction T then
           state ← "forward"
     return $v, \omega$
```

Algorithm A.3 The pseudo-code for the state-machine of Bug2.

```
  Init: state = "forward", $s_{WF} = 1$
Require: $M-line$,$c_v$, $c_\omega$, $x_{global} r_{local}$
  function Com
     if state is "forward"  then
        $v \leftarrow c_v$
        $\omega \leftarrow 0$
        if Obstacle is hit then
           state ← "wall_following"
     else if state is "wall_following" then
        $[v,\ \omega] \leftarrow$ Wall_Following($c_v, c_\omega, s_{WF}, r_{local}$)        ▷ See C.2
        if $M-line$ is hit and BA is closer to T then
           state ← "rotate_to_target"
     else if state is "rotate_to_target" then
        $v \leftarrow 0$
        $\omega \leftarrow c_\omega$
        if Heading BA same as direction T then
           state ← "forward"
     return $v, \omega$
```

Algorithm A.4 The pseudo-code for the state-machine of Alg1.

```
  Init: $state$ = "forward", $s_{WF}$ = 1, , $list_{HP}$=[ ]
Require: $M - line$,$c_v$, $c_\omega$, $x_{global}r_{local}$
  function COM
    if $state$ is "forward"  then
      $v \leftarrow c_v$
      $\omega \leftarrow 0$
      if Obstacle is hit then
        $list_{HP} \leftarrow [list_{HP}, x_{global}]$
        $state \leftarrow$ "wall_following"
    else if $state$ is "wall_following" then
      $[v,\ \omega] \leftarrow$ Wall_Following($c_v, c_\omega, s_{WF}, r_{local}$)      ▷ See C.2
      if $x_{global}$ is in $list_{HP}$ then
        $state$ is "change_local_direction"
      if $M - line$ is hit and BA is closer to T then
        $state \leftarrow$ "rotate_to_target"
    else if $state$ is "rotate_to_target" then
      $v \leftarrow 0$
      $\omega \leftarrow c_\omega$
      if Heading BA same as direction T then
        $state \leftarrow$ "forward"
    else if $state$ is "change_local_direction" then
      $v \leftarrow 0$
      $\omega \leftarrow c_\omega$
      $s_{WF} = -1$
      if BA has rotated $18^o$ then
        $state \leftarrow$ "wall_following"
    return $v, \omega$
```

Algorithm A.5 The pseudo-code for the state-machine of Alg2.

```
  Init: $state$ = "forward", $s_{WF}$ = 1, $list_{HP}$=[ ]
Require: $c_v$, $c_\omega$, $r_{local}$
  function COM
    if $state$ is "forward"  then
      $v \leftarrow c_v$
      $\omega \leftarrow 0$
      if Obstacle is hit then
        $s_{WF} = 1$
        $d(H,T) \leftarrow d(x_{global}, T) for$
        $list_{HP} \leftarrow [list_{HP}, x_{global}]$
        $state \leftarrow$ "wall_following"
    else if $state$ is "wall_following" then
      $[v,\ \omega] \leftarrow$ Wall_Following($c_v, c_\omega, s_{WF}, r_{local}$)      ▷ See C.2
      if $x_{global}$ is in $list_{HP}$ then
        $state$ is "change_local_direction"
      if Way towards T is free and $d(x_{global}, T) < d(H,T)$ then
        $state \leftarrow$ "rotate_to_target"
    else if $state$ is "rotate_to_target" then
      $v \leftarrow 0$
      $\omega \leftarrow c_\omega$
      if Heading BA same as direction T then
        $state \leftarrow$ "forward"
    else if $state$ is "change_local_direction" then
      $v \leftarrow 0$
      $\omega \leftarrow c_\omega$
      $s_{WF} = -1$
      if BA has rotated $18^o$ then
        $state \leftarrow$ "wall_following"
    return $v, \omega$
```

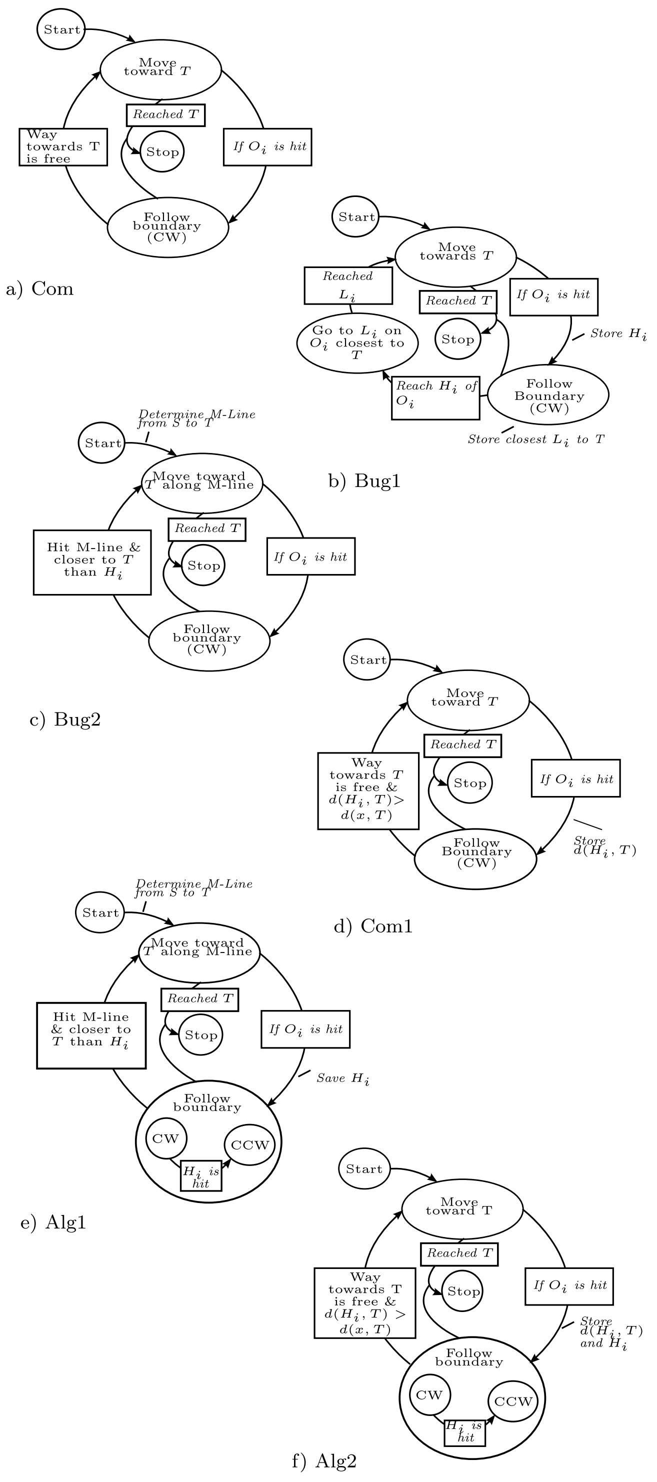


Fig. A.16. Bug algorithm state machines. The S and T represent the start and target position respectively. O_i is the ith obstacle that the bug hits and L_i and H_i is the ith leave- and hit-point, respectively.

Appendix B. Procedural environment generator

The procedural environment generator specifics is shown in Fig. B.17. First, in a coarse grid world (a), two entities are initialized on the exact position of the start and target position to be in the eventual task. They will perform a simple 4-connected path generation, where they will have a certain chance of going straight (p_{str}). The chance of either going left or right is equal to $1 - p_{str}$. Each agent will leave a corridor trace, as can be seen in (b), until, in (c), the amount of corridors hit a density threshold ($t_{cor} = 0.4$), which is the number of grid-cells occupied with a corridor divided by the total number of existing grid cells in the environment. A connectivity check is performed, to check if the initial position of the robots are connected by these corridors, which will re-initiate the process in case it fails. This is to ensure

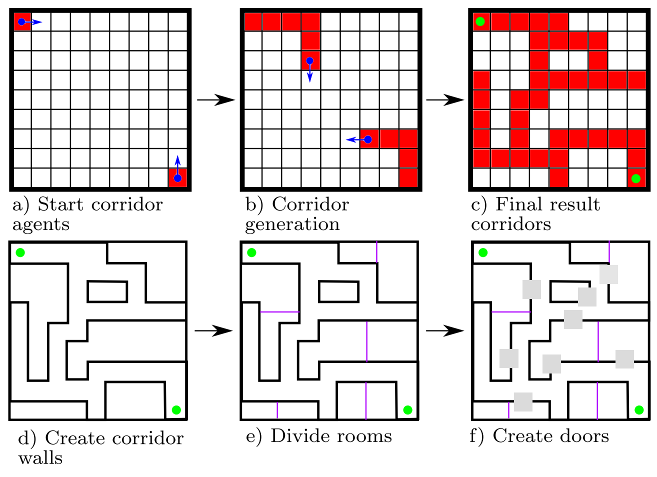


Fig. B.17. The steps of the procedural generated environment method will be explained here. The corridor-generating random agents (blue circles) start in (a) at the same positions as the start and target locations of the experiment. These will move forward in (b), while occasionally turning left and right, while leaving a corridor trace (red blocks). Once it reaches the corridor-density threshold in (c), the corridors-cells are tested for interconnectivity, such that the target position can be reached from the starting position (green circles). The corridor walls are created in (d) and then, in (e), remaining non-corridor spaces are then divided into rooms (purple stripes) and random door-openings (gray blocks) are created along the border of the corridors in (f). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

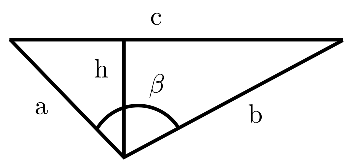


Fig. C.18. Visualization of the triangle configuration for the derivation.

that the BA is always able to reach its final destination. Next, walls will be added to these corridors (d). The remaining areas will act as rooms and are divided when they are too large in (e). Finally, in (f), random openings are added along the border of the corridors to create passages these areas.

Appendix C. Wall following

C.1. Calculation real distance from wall

In Fig. C.18, the configurations of the solved triangle is solved, where we want to calculate h (height triangle) with the triangle sides of a and b and the angle β. c is the triangle side that will be unknown, so a formula will be derived that will only use a, b and β.

The geometrical equations used to achieve the ranges are the triangle area formula:

$$A = \frac{c \cdot h}{2}, \tag{C.1}$$

the SAS triangle rule:

$$A = \frac{a \cdot b \cdot \sin\beta}{2}, \tag{C.2}$$

the cosine-rule:

$$c = \sqrt{a^2 + b^2 - 2 \cdot a \cdot b \cos\beta} \tag{C.3}$$

with A is the area of the triangle.

Substitute A in Eq. (C.2) for the right side of Eq. (C.1), and solve for h:

$$h = \frac{a \cdot b \cdot \sin\beta}{c} \tag{C.4}$$

Now substitute c in Eq. (C.4), for the right side of Eq. (C.3), which results in the following equation:

$$h = \frac{a \cdot b \sin\beta}{\sqrt{a^2 + b^2 - 2 \cdot a \cdot b \cos\beta}} \tag{C.5}$$

C.2. Pseudo code wall following

The procedure of the wall-following behavior is listed in this appendix in Algorithms C.1 and C.2. Ts_{WF} is a variable that determines if the wall-following is right- ($s_{WF} = 1$) or left-sided ($s_{WF} = -1$), $d(x, O_\perp)$ is the current distance to the robot calculated perpendicular from the wall and d_{ref} is the preferred distance from the wall in meters and t_d is the threshold to determine if the robot near d_{ref}. r_s and r_f are the side and range sensor's measurement in meters and β is the angle between them. If $s_{WF} = 1$, then r_s is the right range sensor and if $s_{WF} = -1$, then r_s is the left range sensor. v is the control output for the forward velocity of the robot and c_v is the fixed velocity constant. ω is the control output for the heading of the robot in rad/s and c_ω is a fixed rate constant, to control the speed of the robot's turns.

Algorithm C.1 The procedure of the wall-following behavior.

```
  Init: $state$ = "rotate_to_align_wall"
Require: $s_{WF}$, $d(x, O_\perp)$, $d_{ref}$, $r_s$, $r_f$, $c_v, c_\omega$, $\beta$
  $\beta$ = 60 deg
  function WALL_FOLLOWING
      if $state$ is "rotate_to_align_wall" then
          $v \leftarrow 0$
          $\omega \leftarrow -1 \cdot s_{WF} \cdot c_\omega$                ▷ Turn away from the wall
          if $r_s \approx r_f \cdot \cos(\beta)$ then
              $state$ ← "wall_following_and_aligning"
          if $r_f$ = OR then
              $state$ ← "rotate_around_corner"
      else if $s_{WF}$ is "wall_following_and_aligning" then
          $v \leftarrow c_v$
          $\omega$ ← Wall_Following_and_Aligning()                ▷ See C.2
          if $d(x, O_{min}) < d_{ref}$ then
              $state$ ← "rotate_to_align_wall"
          if $r_f$ is OR then
              $state$ ← "rotate_around_corner"
      else if $state$ is "rotate_around_corner" then
          $v \leftarrow c_v$
          $\omega \leftarrow s_{WF} \cdot v/d_{ref}$                ▷ Wide turn, radius = $d_{ref}$
          if $r_s \approx r_f \cdot \cos(\beta)$ then
              $state$ ← "wall_following_and_aligning"
          if $d(x, O_{min}) < d_{ref}$ then
              $state$ ← "rotate_to_align_wall"
    return $\omega$
```

Algorithm C.2 The procedure of keeping the heading of the FootBot aligned with the wall during wall following.

```
Require: $s_{WF}$, $d(x, O_\perp)$, $d_{ref}$, $r_s$, $r_f$, $c_w$, $\beta$
  function WALL_FOLLOWING_AND_ALIGNING
    if $|d_{ref} - d(x, O_\perp)| > -t_d$ ) then          ▷ If too far from $d_{ref}$
      if $d_{ref} - d(x, O_\perp) > t_d$ then              ▷ If too far from wall
        $\omega = s_{WF} \cdot c_\omega$                   ▷ Turn towards the wall
      else                                                  ▷ If too close to wall
        $\omega = -s_{WF} \cdot c_\omega$                  ▷ Turn from the wall
    else if $|d_{ref} - d(x, O_\perp)| < t_d$ then         ▷ If close to $d_{ref}$
      if $r_s > r_f \cdot \cos\beta$ then                  ▷ Fine tune alignment
        $\omega = s_{WF} \cdot c_\omega$                   ▷ Turn towards the wall
      else
        $\omega = -s_{WF} \cdot c_\omega$                  ▷ Turn from the wall
    else                                                    ▷ Do not adjust the turn
      $\omega = 0$
    return $\omega$
```

Appendix D. Statistical tests

D.1. Bootstrapping bug algorithms

In Fig. 11 for perfect localization ($\sigma = 0$), the resulting performance values per bug algorithm was shown. Here, both the success rate and the trajectory length are subjected to a bootstrapping test, to evaluate whether the bug algorithms belong to the same distribution (null-hypothesis). Table D.2 contains the bootstrapping tests from the data presented in Fig. 11(a) and Table D.3 for Fig. 11(b) for $\sigma = 0$.

D.2. Correlation analysis odometry noise

In order to evaluate whether an relationship exists between the increasing odometry noise and the degeneration of the performances of the bug algorithms, the data presented in Fig. 11 are subjected to regression analysis. Table D.5 contains the logistic regression analysis with a R2 value, from the trajectory length data presented in Fig. 11(a) and Table D.4 contains the logistic regression analysis with a pseudo-R2 value, from the success rate data presented in Fig. 11(b).

D.3. Correlation analysis recognition failures

In order to evaluate whether an relationship exists between the increasing failing recognition rate and the degeneration of the performances of the bug algorithms Alg1 and Alg2, the data presented in Fig. 13 are subjected to regression analysis. Table D.7 contains the logistic regression analysis with a R2 value, from the trajectory length data presented in Fig. 13(a) and Table D.6 contains the logistic regression analysis with a pseudo-R2 value, from the success rate data presented in Fig. 13(b). Table D.9 contains the logistic regression analysis with a R2 value, from the trajectory length data presented in Fig. 13(c) and Table D.8 contains the logistic regression analysis with a pseudo-R2 value, from the success rate data presented in Fig. 13(d).

D.4. Correlation analysis distance sensor noise

In order to evaluate whether an relationship exists between the increasing distance measurement noise and the degeneration of the performances of the bug algorithms Alg1 and Alg2, the data presented in Fig. 14 are subjected to regression analysis. Table D.11 contains the logistic regression analysis with a R2 value, from the trajectory length data presented in Fig. 14(a) and Table D.10 contains the logistic regression analysis with a pseudo-R2 value, from the success rate data presented in Fig. 14(b).

Table D.2
Bootstrapping results on the trajectory length of the evaluated bug algorithms with a sample size 10 000. The value “1” means that the null-hypothesis (the evaluated data comes from the same distribution) holds, while “0” means it is rejected.

	Com	Com1	Bug2	Alg1	Alg2
Com	1	1	0	0	1
Com1	1	1	0	0	1
Bug2	0	0	1	1	0
Alg1	0	0	1	1	0
Alg2	1	1	0	0	1

Table D.3
Bootstrapping results on the success rate of the evaluated bug algorithms with a sample size 10 000. The value “1” means that the null-hypothesis (the evaluated data comes from the same distribution) holds, while “0” means it is rejected.

	Com	Com1	Bug2	Alg1	Alg2
Com	1	0	0	0	0
Com1	0	1	1	1	0
Bug2	0	1	1	1	0
Alg1	0	1	1	1	0
Alg2	0	0	0	0	1

Table D.4
Linear regression evaluation of the trajectory lengths against the measurement noise, including the intercept, slope and R2 value per bug algorithm.

	Com	Com1	Bug2	Alg1	Alg2
Slope	8.081	13.642	13.561	11.857	12.523
Intercept	2.752	2.724	3.152	3.522	2.654
R2	0.076	0.189	0.217	0.173	0.161

Table D.5
Logistic regression evaluation of the success rate against the measurement noise, including the intercept, slope and (pseudo) R2 value per bug algorithm.

	Com	Com1	Bug2	Alg1	Alg2
Slope	−1.240	−3.100	−3.860	−3.480	−3.110
Intercept	0.587	0.800	0.779	0.706	0.847
R2	0.035	0.189	0.343	0.323	0.198

Table D.6
Linear regression evaluation of the trajectory lengths against the False Positive recognition rate, including the intercept, slope and R2 value per bug algorithm.

	Alg1	Alg2
Slope	0.5472	0.1722
Intercept	2.4302	1.8843
R2	0.0112	0.0020

Table D.7
Logistic regression evaluation of the success rate against the False Positive recognition rate, including the intercept, slope and (pseudo) R2 value per bug algorithm.

	Alg1	Alg2
Slope	−0.1443	−0.0014
Intercept	0.9105	0.9418
R2	0.4652	0.7773

Table D.8
Linear regression evaluation of the trajectory lengths against the False Negative recognition rate, including the intercept, slope and R2 value per bug algorithm.

	Alg1	Alg2
Slope	0.1873	1.0584
Intercept	3.2478	2.2733
R2	0.0000	0.0006

Table D.9
Logistic regression evaluation of the success rate against the False Negative recognition rate, including the intercept, slope and (pseudo) R2 value per bug algorithm.

	Alg1	Alg2
Slope	0.0577	0.1010
Intercept	0.8018	0.9192
R2	0.3668	0.7171

Table D.10
Linear regression evaluation of the trajectory lengths against the distance measurement noise, including the intercept, slope and R2 value per bug algorithm.

	Com1	Alg2
Slope	0.0501	−0.0075
Intercept	2.5783	2.4204
R2	0.0019	0.0001

Table D.11
Logistic regression evaluation of the success rate against the distance measurement noise, including the intercept, slope and (pseudo) R2 value per bug algorithm.

	Com1	Alg2
Slope	−0.0557	−0.0225
Intercept	0.8682	0.9283
R2	0.2412	0.5583

References

[1] G. Bresson, Z. Alsayed, L. Yu, S. Glaser, Simultaneous localization and mapping: A survey of current trends in autonomous driving, IEEE Trans. Intell. Veh. 2 (3) (2017) 194–220, URL https://doi.org/10.1109/TIV.2017.2749181.

[2] J. Boal, A. Sánchez-Miralles, A. Arranz, Topological simultaneous localization and mapping: a survey, Robotica 32 (05) (2014) 803–821, URL https://doi.org/10.1017/S0263574713001070.

[3] B.A. Cartwright, T.S. Collett, Landmark learning in bees, J. Comp. Physiol. 151 (4) (1983) 521–543, URL https://doi.org/10.1007/BF00605469.

[4] D. Lambrinos, R. Möller, T. Labhart, R. Pfeifer, R. Wehner, A mobile robot employing insect strategies for navigation, Robot. Auton. Syst. 30 (1) (2000) 39–64, URL https://doi.org/10.1016/S0921-8890(99)00064-0.

[5] H. Noborio, K. Fujimura, Y. Horiuchi, A comparative study of sensor-based path-planning algorithms in an unknown maze, in: Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113), Vol. 2, 2000, pp. 909–916 vol.2, URL https://doi.org/10.1109/IROS.2000.893135.

[6] J. Ng, T. Bräunl, Performance comparison of bug navigation algorithms, J. Intell. Robot. Syst. 50 (1) (2007) 73–84, URL https://doi.org/10.1007/s10846-007-9157-6.

[7] M.W. Mueller, M. Hamer, R. D'Andrea, Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation, in: 2015 IEEE International Conference on Robotics and Automation (ICRA), 2015, pp. 1730–1736, URL https://doi.org/10.1109/ICRA.2015.7139421.

[8] J. Borenstein, L. Feng, Measurement and correction of systematic odometry errors in mobile robots, IEEE Trans. Robot. Autom. 12 (6) (1996) 869–880, URL https://doi.org/10.1109/70.544770.

[9] D. Scaramuzza, F. Fraundorfer, Visual odometry [tutorial], IEEE Robot. Autom. Mag. 18 (4) (2011) 80–92, URL https://doi.org/10.1109/MRA.2011.943233.

[10] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1) (1959) 269–271, URL https://doi.org/10.1007/BF01386390.

[11] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107, URL https://doi.org/10.1109/TSSC.1968.300136.

[12] S. Mishra, P. Bande, Maze solving algorithms for micro mouse, in: 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems, 2008, pp. 86–93, URL https://doi.org/10.1109/SITIS.2008.104.

[13] V. Lumelsky, A. Stepanov, Dynamic path planning for a mobile automaton with limited information on the environment, IEEE Trans. Automat. Control 31 (11) (1986) 1058–1063, URL https://doi.org/10.1109/TAC.1986.1104175.

[14] V.J. Lumelsky, A.A. Stepanov, Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape, Algorithmica 2 (1) (1987) 403–430, URL https://doi.org/10.1007/BF01840369.

[15] A. Sankaranarayanan, M. Vidyasagar, A new path planning algorithm for moving a point object amidst unknown obstacles in a plane, in: Proceedings., IEEE International Conference on Robotics and Automation, Vol. 3, 1990, pp. 1930–1936, URL https://doi.org/10.1109/ROBOT.1990.126290.

[16] I. Kamon, E. Rivlin, Sensory-based motion planning with global proofs, IEEE Trans. Robot. Autom. 13 (6) (1997) 814–822, URL https://doi.org/10.1109/70.650160.

[17] Y. Horiuchi, H. Noborio, Evaluation of path length made in sensor-based path-planning with the alternative following, in: Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164), Vol. 2, 2001, pp. 1728–1735, URL https://doi.org/10.1109/ROBOT.2001.932860.

[18] V. Lumelsky, T. Skewis, A paradigm for incorporating vision in the robot navigation function, in: Proceedings. 1988 IEEE International Conference on Robotics and Automation, 1988, pp. 734–739, URL https://doi.org/10.1109/ROBOT.1988.12146.

[19] V.J. Lumelsky, T. Skewis, Incorporating range sensing in the robot navigation function, IEEE Trans. Syst. Man Cybern. 20 (5) (1990) 1058–1069, URL https://doi.org/10.1109/21.59969.

[20] I. Kamon, E. Rivlin, E. Rimon, A new range-sensor based globally convergent navigation algorithm for mobile robots, in: Proceedings of IEEE International Conference on Robotics and Automation, Vol. 1, 1996, pp. 429–435, URL https://10.1109/ROBOT.1996.503814.

[21] I. Kamon, E. Rimon, E. Rivlin, Range-sensor based navigation in three dimensions, in: Proceedings 1999 IEEE International Conference on Robotics and Automation, Vol. 1, 1999, pp. 163–169, URL https://doi.org/10.1109/ROBOT.1999.769955.

[22] S.L. Laubach, J.W. Burdick, An autonomous sensor-based path-planner for planetary microrovers, in: Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C), Vol. 1, 1999, pp. 347–354, URL https://doi.org/10.1109/ROBOT.1999.770003.

[23] E. Magid, E. Rivlin, CaUtiousbug: a competitive algorithm for sensory-based robot navigation, in: Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566), Vol. 3, 2004, pp. 2757–2762, URL https://doi.org/10.1109/IROS.2004.1389826.

[24] Q.-L. Xu, G.-Y. Tang, Vectorization path planning for autonomous mobile agent in unknown environment, Neural Comput. Appl. 23 (7–8) (2013) 2129, URL https://doi.org/10.1007/s00521-012-1163-3.

[25] S. Lee, T.M. Adams, B. yeol Ryoo, A fuzzy navigation system for mobile construction robots, Autom. Constr. 6 (2) (1997) 97–107, URL https://doi.org/10.1016/S0926-5805(96)00185-9.

[26] H. Noborio, Y. Maeda, K. Urakawa, Three or more dimensional sensor-based path-planning algorithm hd-i, in: Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No.99CH36289), Vol. 3, 1999, pp. 1699–1706, URL https://doi.org/10.1109/IROS.1999.811723.

[27] Q.-l. Xu, Randombug: Novel path planning algorithm in unknown environment, Open Electr. Electron. Eng. J. 8 (2014) 252–257, URL https://doi.org/10.2174/1874129001408010252.

[28] S.M. LaValle, J. James J. Kuffner, Randomized kinodynamic planning, Int. J. Robot. Res. 20 (5) (2001) 378–400, URL https://doi.org/10.1177/02783640122067453.

[29] K. Taylor, S.M. LaValle, I-bug: An intensity-based bug algorithm, in: Proc. IEEE Int. Conf. Robotics and Automation, 2009, pp. 3981–3986, URL https://doi.org/10.1109/ROBOT.2009.5152728.

[30] S. Laubach, J. Burdick, Roverbug: Long range navigation for mars rovers, Experimental Robotics VI (2000) URL https://doi.org/10.1007/BFb0119412.

[31] F. Mastrogiovanni, A. Sgorbissa, R. Zaccaria, Robust navigation in an unknown environment with minimal sensing and representation, IEEE Trans. Syst. Man Cybern. B 39 (1) (2009) 212–229, URL https://doi.org/10.1109/TSMCB.2008.2004505.

[32] Y. Zhu, T. Zhang, J. Song, X. Li, A new bug-type navigation algorithm considering practical implementation issues for mobile robots, in: Proc. IEEE Int. Conf. Robotics and Biomimetics, 2010, pp. 531–536, URL https://doi.org/10.1109/ROBIO.2010.5723382.

[33] D.-H. Kim, K. Shin, C.-S. Han, J.Y. Lee, Sensor-based navigation of a car-like robot based on bug family algorithms, Proc. Inst. Mech. Eng. Part C 227 (6) (2013) 1224–1241, URL https://doi.org/10.1177/0954406212458202.

[34] K. Taylor, S.M. LaValle, Intensity-based navigation with global guarantees, Auton. Robots 36 (4) (2014) 349, URL http://dx.doi.org/10.1007/s10514-013-9356-x.

[35] A. Ebrahimi, F. Janabi-Sharifi, A. Ghanbari, Uavisbug: vision-based 3D motion planning and obstacle avoidance for a mini-UAV in an unknown indoor environment, Can. Aeronaut. Space J. 60 (01) (2014) 9–21, URL https://doi.org/10.5589/q14-005.

[36] M.M. Gulzar, Q. Ling, M. Yaqoob, S. Iqbal, Realization of an improved path planning strategy, in: 2015 International Conference on Control, Automation and Information Sciences (ICCAIS), 2015, pp. 384–389, URL https://doi.org/10.1109/ICCAIS.2015.7338698.

[37] R. Marino, F. Mastrogiovanni, A. Sgorbissa, R. Zaccaria, A minimalistic quadrotor navigation strategy for indoor multi-floor scenarios, in: E. Menegatti, N. Michael, K. Berns, H. Yamaguchi (Eds.), Intelligent Autonomous Systems, Vol. 13, Springer International Publishing, Cham, 2016, pp. 1561–1570, URL https://doi.org/10.1007/978-3-319-08338-4_112.

[38] T. Goedemé, M. Nuttin, T. Tuytelaars, L. Van Gool, Omnidirectional vision based topological navigation, Int. J. Comput. Vis. 74 (3) (2007) 219–236, URL https://doi.org/10.1007/s11263-006-0025-9.

[39] F. Fraundorfer, C. Engels, D. Nister, Topological mapping, localization and navigation using image collections, in: 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2007, pp. 3872–3877, URL https://doi.org/10.1109/IROS.2007.4399123.

[40] M.S. Bargh, R. de Groote, Indoor localization based on response rate of bluetooth inquiries, in: Proceedings of the First ACM International Workshop on Mobile Entity Localization and Tracking in GPS-Less Environments, in: MELT '08, ACM, New York, NY, USA, 2008, pp. 49–54, URL http://doi.acm.org/10.1145/1410012.1410024.

[41] K. Guo, Z. Qiu, W. Meng, L. Xie, R. Teo, Ultra-wideband based cooperative relative localization algorithm and experiments for multiple unmanned aerial vehicles in GPS denied environments, Int. J. Micro Air Veh. 9 (3) (2017) 169–186, URL https://doi.org/10.1177/1756829317695564.

[42] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L.M. Gambardella, M. Dorigo, Argos: A modular, parallel, multi-engine simulator for multi-robot systems, Swarm Intell. 6 (4) (2012) 271–295, URL https://doi.org/10.1007/s11721-012-0072-5.

[43] M.A. Fischler, R.C. Bolles, Random sample consensus: A paradigm for model fitting with applications to image analysis and automated Cartography, Commun. ACM 24 (6) (1981) 381–395, URL http://doi.acm.org/10.1145/358669.358692.

Kimberly McGuire is a PhD candidate at the faculty of Aerospace Engineering of the Delft University of Technology, concentrated in autonomous navigation on lightweight pocket drones at the MAVlab. She has a broad research-interest in embodied intelligence for robotics, in both autonomous navigation and cognition. In 2012 she received her B.Sc degree in Industrial Design Engineering and her M.Sc. degree in the field of Mechanical Engineering in 2014 at the Delft University of Technology, specialized in biologically inspired Robotics.

Guido de Croon: Received his M.Sc. and Ph.D. in the field of Artificial Intelligence (AI) at Maastricht University, the Netherlands. His research interest lies with computationally efficient algorithms for robot autonomy, with an emphasis on computer vision and evolutionary robotics. Since 2008 he has worked on algorithms for achieving autonomous flight with small and light-weight flying robots, such as the DelFly flapping wing MAV. In 2011-2012, he was a research fellow in the Advanced Concepts Team of the European Space Agency, where he studied topics such as optical flow based control algorithms for extraterrestrial landing scenarios. Currently, he is associate professor at TU Delft and scientific lead of the Micro Air Vehicle lab (MAV-lab) of Delft University of Technology.

Karl Tuyls (FBCS) is a research scientist at DeepMind and is also a professor of Computer Science at the University of Liverpool, UK. Previously, he held positions at the Vrije Universiteit Brussel, Hasselt University, Eindhoven University of Technology, and Maastricht University. At the University of Liverpool he was director of research of the school of Electrical Engineering & Electronics and Computer Science. He founded and led the smARTLab robotics laboratory since 2013 (http://wordpress.csc.liv.ac.uk/smartlab/). Prof. Tuyls has received several awards with his research, amongst which: the Information Technology prize 2000 in Belgium, best demo award at AAMAS'12, winner of the German Open robocup@work competitions in 2013 and 2014, world champion of the RoboCup@Work competitions in 2013 and 2014, winner of the RoCKIn@work competition in 2015. Furthermore, his research has received substantial attention from national and international press and media (http://karltuyls.net). He is a fellow of the British Computer Society (BCS), is on the editorial board of the Journal of Autonomous Agents and Multi-Agent Systems, and is editor-in-chief of the Springer briefs series on Intelligent Systems. Prof. Tuyls is also a member of the board of directors of the International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).