A lightweight navigation system for mobile robots

M. T. Lázaro[1], G. Grisetti[1], L. Iocchi[1], J. P. Fentanes[2], and M. Hanheide[2]

[1] DIAG, Sapienza University of Rome, Italy,
{mtlazaro, grisetti, iocchi}@dis.uniroma1.it,
[2] LCAS, University of Lincoln, UK,
jpulidofentanes@lincoln.ac.uk, mhanheide@lincoln.ac.uk

Abstract. In this paper, we describe a navigation system requiring very few computational resources, but still providing performance comparable with commonly used tools in the ROS universe. This lightweight navigation system is thus suitable for robots with low computational resources and provides interfaces for both ROS and NAOqi middlewares. We have successfully evaluated the software on different robots and in different situations, including SoftBank Pepper robot for RoboCup@Home SSPL competitions and on small home-made robots for RoboCup@Home Education workshops. The developed software is well documented and easy to understand. It is released open-source and as Debian package to facilitate ease of use, in particular for the young researchers participating in robotic competitions and for educational activities.

Keywords: Navigation, Mobile Robots, Open Source, RoboCup

1 Introduction

Navigation is among the most important features of mobile robot applications and is a primary functionality in many tasks. Navigation performance of mobile robots is typically well established thanks to the availability of standard open-source software (e.g., ROS `amcl` and `move_base`) and many robotic platforms are able to properly use such off-the-shelf tools after a reasonable configuration effort. However, such solutions have some limitations: they are strongly embedded in the ROS framework, they require substantial computational power, they are sometimes not easy to tune and configure for a particular setting (robot and environment). While ROS is a de-facto standard for many robotic platforms, there are still cases in which ROS-based tools cannot be fully used. For example, SoftBank Pepper robot has its own operating system and development environment that does not support ROS on-board. Small home-made robots controlled by Arduino and Raspberry boards have limited computational power with respect to the requirements of typical ROS-based navigation modules.

In several robotic competitions, specially for young researchers, having a lightweight open-source navigation system is very important to speed-up deployment of mobile robot applications. For example, as part of the RoboCup

competitions[3], the RoboCup@Home Social Standard Platform, using SoftBank Pepper robot, introduces a new challenge in navigation because of the need of deploying a navigation system for RoboCup@Home tasks running with on-board computational resources of the robot. Similarly, in the RoboCup@Home Education initiative[4] low-cost robots with limited computational resources are used to keep the entry-level for new teams low.

Therefore, the release of a lightweight navigation system that can support both ROS and other robotic development environments, achieving the same performance of standard ROS-based tools, but requiring much less computational resources, brings many benefits in mobile robot applications specially for involving many young researchers to robotic competitions and to the use of educational robots.

In this paper, we describe the development and release of an open-source lightweight navigation system that has been integrated in and evaluated on three different robotic platforms of diverse profiles regarding their middleware support and their available computational resources: 1) ROS-based robots with good computational resources, 2) ROS-based robots with limited computational resources, 3) NAOqi-based robots (in particular, SoftBank Pepper) with limited computational resources. Notice that, while in the first case our system achieves similar performance as standard ROS-based tools (i.e., `amcl` and `move_base`), in the other two cases these tools are not suitable either because of too limited resources or non-availability of ROS middle-ware.

The proposed solution has been also been deployed in the context of the RoboCup@Home Social Standard Platform League, within the team SPQReL that is a joint research collaboration between Sapienza University of Rome, Italy and University of Lincoln, UK. Within the context of RoboCup@Home Social Standard Platform League, the navigation package described in this paper overcomes current limitations of the navigation system of the Pepper robot since because 1) it is open-source, 2) it runs on the Pepper on-board PC, 3) it provides navigation performance comparable with ROS standard tools.

2 Related work

Mobile robot navigation is one of the key competences a robot must have to be fully autonomous. For this reason, it is usual to find at least one navigation component in most robot software frameworks. Although there are multiple frameworks for mobile robot software development, in general, it is safe to say that robotics navigation specific software can be divided in three main components: localization, global path planning and local path planning or reactive navigation.

Although localization has been probably the most researched field in mobile robotics, most well known robotics frameworks like ROS or the Player project include a standard localization component based on 2D laser and an AMCL [3]

[3] www.robocup.org
[4] www.robocupathomeedu.org

localization filter. Usually this component can be replaced depending on the robot set-up, using, for example, EKF and GPS information for outdoor robot localization or the integration of vision based localization systems.

The integration of both global and local path planners however has had more variety of options. For example, the Player project uses a standard wavefront propagation planner [8] for generating global plans that drive the robot towards its goal, while, to deal with unmapped obstacles, it provides three different methods that drive the robot away from obstacles and towards the goal using virtual force fields [10] or decision trees [7].

More recently, the robot operating system (ROS) has proposed the use of a very flexible navigation stack [6] that uses a global dynamic window approach [2] that not only plans a global plan to the goal and then follows it blindly but also re-plans the path if the robot deviates from it to avoid unmapped obstacles. In addition to this, ROS uses a roll-out trajectory planner [5] to send velocity commands to the robot base that follows the planned trajectory as close as possible given the robot constrains and unmapped obstacles. Alternatively to this planner, ROS also provides a dynamic window planner [4] that is more efficient that the trajectory roll-out as it samples a smaller velocity space filtering out unachievable trajectories using the robot's acceleration limits.

Other authors propose having a planning library [9] that can easily be integrated in any system and provides multiple tools to create the best possible planner, these proposals can create highly customizable navigation systems and are not restricted to any specific software framework.

Our proposal is to provide an open source approach to navigation that can be used both as an additional navigation system for ROS or as an open source alternative for NAOqi-based robots that is lightweight and can be used directly in robots with limited computational power.

The proposed robot navigation suite comprises a variant of Monte Carlo Localization, a Dijkstra-based global path planner, and a policy-based dynamic obstacle avoidance, all described in detail in the following. The suite is completed by visualization and remote control tools.

3 Localization filter

The localization module implements an efficient version of the Monte Carlo Localization. The idea is to track the robot position with a particle filter. The state of the filter comprises the 2D robot pose and orientation $\mathbf{x}_t = (x_t; y_t; \theta_t)^T$ and the belief space is represented as a set of pose samples $\mathbf{x}^{(i)}$. The denser the samples in a region of the environment, the more likely is that the robot will be in that region. The state transition is governed by odometry measurements $\mathbf{u}_t = (\Delta x_t; \Delta y_t; \Delta\theta_t)^T$, that express the relative movement of the robot between subsequent time steps.

Each time a new odometry measurement is received, we generate a new set of samples according to the following equation:

$$\mathbf{x}_{t|t-1}^{i} = \mathbf{x}_{t-1}^{(i)} \oplus (\mathbf{u}_{t-1} + \mathbf{n}_{t-1}^{(i)}) \tag{1}$$

$$\mathbf{n}_{t-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_{t-1}). \tag{2}$$

Here $\mathbf{n}_{t-1}$ is a sample drawn from a zero mean Gaussian distribution with covariance $\mathbf{\Sigma}_{t-1}$ representing the additive noise affecting the odometry. The covariance of this distribution is adapted based on the magnitude of the odometry motion. If the robot does not move, then $\mathbf{\Sigma}_{t-1} = \mathbf{0}$ and no sampling is performed. If one of the samples falls in the invalid space of the environment (e.g. unknown or inside a wall), it is replaced by a new sample drawn at a random valid location. This substantially enhances the performance during global localization.

When a sensor measurement $\mathbf{z}_t$ becomes available, we refine our predicted belief $\{\mathbf{x}_{t|t-1}^{(i)}\}$ through conditioning. To this end we assign each predicted particle a weight $w^{(i)}$, proportional to the likelihood of the measurement. The likelihood $l(\mathbf{z}_t, \mathbf{x}) \in \Re$ is a function expressing how well the current measurement $\mathbf{z}_t$ approximates a predicted measurement obtained from the known map if the robot was at location $\mathbf{x}$. Ideally, if the predicted measurement and the actual one are the same, the likelihood is maximal. Once the likelihood is computed for each predicted sample, we generate the posterior distribution by replicating or suppressing samples depending on their weight. More formally, we draw a set of new indices from the weight distributions, as follows:

$$i_t \sim w^{(i)} \tag{3}$$

Eq. 3, means that the index i_t is obtained by sampling from the piecewise constant distribution of the weights. The index i has probability $w_t^{(i)}$ to be selected. The fact that an index is selected from the sampling means that the corresponding particle will appear in the update distribution. Samples having high weight can be drawn more than once. The above procedure is called resampling and has the effect of turning a weighted distribution in an unweighted one by replicating likely samples and suppressing unlikely ones.

In our system, the sensor measurements $\mathbf{z}_t$ are the laser endpoints. We use a fast but robust procedure to calculate the likelihood. First, given a robot pose hypothesis $\mathbf{x}_t^{(i)}$ and the current laser measurement, we compute the position of the laser endpoints in the map as follows:

$$\hat{z}_k^{(i)} = \mathbf{x}_{t|t-1}^{(i)} \oplus z_k \tag{4}$$

where the index k represents one specific beam within the laser measurement $\mathbf{z}_t$. Subsequently, for each endpoint we compute minimal distance between the endpoint and the closest obstacle in the map. This operation can be performed in $O(1)$, by using a precalculated grid that stores for each cell the minimal distance to the obstacles: *the distance map*. To lessen the effects of dynamic unpredicted obstacles, we clamp the reported distance to a maximum value. Let

$d_k^{(i)}$ be the distance of the k^{th} beam from the closest obstacle w.r.t. particle i. If a measurement is perfectly explained, the distances will be zero. The final likelihood of a particle computed as

$$w^{(i)} = \exp(-\sum \sigma d^{(i)}), \tag{5}$$

where σ is a scaling factor to account for different sensor accuracies. Before the resampling step the weights are normalized so that their sum is 1.

4 Path planning and obstacle avoidance

The planner module implements a fast global path planner that adapts the computed path to dynamic changes in the environment. The system takes advantage of this efficient computation to consider planning only at a global level, in contrast to other planners that manage both local and global maps, which prevents from having a unified view of the environment.

This is facilitated by an efficient implementation and management of the distance map, which, as mentioned in previous Section 3, stores in each pixel the distance to the closest occupied cell. Figure 1(middle) shows an example of distance map associated to a portion of map shown if Fig. 1(left). Once the distance map from the static map is obtained, is it possible to add the information from the dynamic obstacles in the same structure which are managed as explained in the next Section 4.1.

Using this final distance map that includes both static and dynamic information, our system approaches planning by computing the minimal paths to the goal on a 2D grid using the Dijkstra algorithm. Instead of reporting just a single path, we compute a policy such that each cell of the grid points to the closest cell to reach the goal. We assume that the robot can travel from a cell to its eight neighbors, and that the cost of the transition decreases linearly with the distance from the obstacles. Other parameters such as a safety distance from obstacles or the robot radius influences the final cost of each cell, which is saturated to a maximum value. Figure 1(right) shows an example of such grid, called *cost map*. Computing the cost on a grid 1000×1000 takes 20 ms on a Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz.

The final plan provided is executed by using the motion controller by [1] as explained in Section 4.2.

4.1 Management of dynamic obstacles

The distance map is recomputed at each new measurement by incorporating the detected obstacles. This operation can be performed efficiently in an additive fashion by adding only the obstacles to the distance map representing the static scene. In order to track dynamic obstacles we keep a list of unexplained laser endpoints (or grid cells), that are those endpoints that fall far from an occupied cell in the static map.

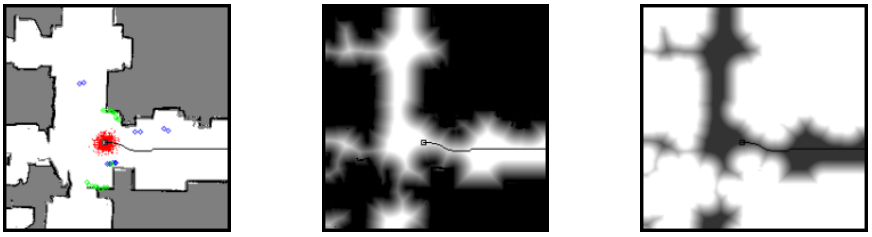

Fig. 1. Left: Portion of an input map. **Middle:** Distance map. **Right:** Cost map.

Two different policies are considered to manage the dynamic obstacles. The first policy is to suppress old obstacles when a new measurement confirms that they have been removed. Intuitively, this situation occurs when the new measurement "goes through" an existing obstacle. More concretely, obstacle points are transformed into the robot's reference frame. Those points that are in the field of view of the robot are *projected* into a circular array of K bins that will store the closest old and new obstacle falling in each bin together with their distance w.r.t. the robot. Then, given an old $\mathbf{p}_k^o$ and new $\mathbf{p}_k^n$ obstacle points falling in the same bin k whose distances to the robot frame are d_k^o and d_k^n respectively, $\mathbf{p}_k^o$ is removed if the new point appears behind the old one or if it is in a Euclidean distance lower than a threshold ϵ as

$$d_k^n > d_k^o \qquad , \qquad |d_k^n - d_k^o| < \epsilon \tag{6}$$

This procedure is illustrated in Fig. 2.

The second policy suppresses obstacles points after a certain time passes. This is done by assigning each obstacle point a time stamp of the moment it has been seen for the last time. This intuitive but effective policy allows to re-consider paths that could have been discarded due to temporally-static obstacles.

4.2 Motion generation

Once a final path to the goal is provided, an intermediate waypoint is computed in a short distance from the robot (e.g., 1m ahead from the current robot pose). This waypoint is used to calculate an attractive virtual force $\mathbf{F}$ that is applied to the robot to generate its movement. Then, using the motion controller described in [1] it is possible to transform this force into the desired control input $\mathbf{u} = (v \quad \omega)^T$, the linear and angular velocities of the robot using the following differential equation:

$$\dot{\mathbf{u}} = \mathbf{A}\mathbf{u} + \mathbf{B}\mathbf{F} \tag{7}$$

where

$$\mathbf{A} = -2b \begin{bmatrix} 1 & 0 \\ 0 & k_i \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & k_i h \end{bmatrix} \qquad \mathbf{F} = \begin{bmatrix} F cos\theta \\ F sin\theta \end{bmatrix} \tag{8}$$

Details on the controller parameters b (viscous friction coefficient), k_i (inertial coefficient) and h (moment arm) are explained in [1]. Notice that the

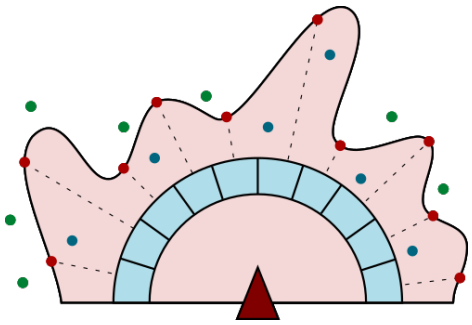

Fig. 2. Management of dynamic obstacle points. Current (red) and old (green, blue) points in the robot's field of view are projected on a circular array of fixed number of bins. Old points are removed if a new point corresponding to the same bin appears behind the old ones (blue points).

intermediate waypoint changes as the robot moves since it always refers to a fixed point w.r.t the current position of the robot, providing a smooth execution of the computed trajectory.

5 Software description

The software is released as a git repository at https://github.com/LCAS/spqrel_navigation/. Links and additional instructions are available from the repositories Wiki (https://github.com/LCAS/spqrel_navigation/wiki) and the SPQReL web site[5]. The software is released as source code to be compiled with either ROS or NAOqi, but also as Debian packages for Ubuntu 16.04LTS desktop development[6].

Integration with other components of the robotic applications is performed in different ways depending on the platform-dependent wrapper used. The ROS wrappers are compatible with `amcl` and `move_base`, thus they use the same name for ROS topics and actions allowing for an easy replacement in already existing ROS applications. The main parameters also correspond to the equivalent ones in standard ROS tools, although our navigation system has much less parameters and thus requires less configuration effort. The NAOqi wrappers use instead a different mechanism that is based on the communication through the NAOqi shared memory (ALMemory), either by means of writing and reading data directly in memory or by raising/subscribing to memory events.

Both the wrappers use a map of the environment described in the same YAML format used in standard ROS navigation applications. This map should be generated beforehand with any available tool compatible with ROS standard maps (e.g., gmapping).

[5] http://tinyurl.com/spqrel

[6] For installation of binary packages see https://github.com/LCAS/rosdistro/wiki and install `ros-kinetic-spqrel-navigation`

5.1 Visualization Tools

For ROS-based systems, RViz can be used to visualize the information relevant to the task (localization particles, path planned, etc.) For non-ROS systems, we have developed simple viewers to view the information processed by the components and to tune the parameters. In particular, we will describe below the localizer viewer and the planner viewer.

Fig. 3. Left: Localizer viewer. Red points represent the filter particles. Green points are current laser points that can be explained by the map while blue points are laser points not explained by the map (e.g., new obstacles not represented in the map). **Middle:** Planner viewer. **Right:** Distance map.

Through the localizer viewer it is possible to see the outcome of the localization system. As shown in Figure 3 (left), the viewer shows the map, the current laser scans and the particles that are currently stored in the filter. The viewer also allows for setting the initial pose of the robot in a specific pose of the environment or to call for a global localization phase if the pose is not known accurately.

Similarly, the planner viewer allows to visualize the state of the navigation system. As shown in Figure 3 (middle), the viewer shows the map, current pose of the robot provided by the localization system, current goal (if given) and the computed path if the goal is reachable from the current pose. From this viewer it is also possible to visualize the current distance map computed from the given map and the added dynamic obstacles (see Figure 3 (right)), cancel a goal or restore the distance map to its original state (i.e., cancel the obstacles added during the navigation task).

6 Experiments

In this section we present evaluation experiments of our navigation system. The experiments aim at verifying that our navigation system achieves similar performance of a typical ROS-based navigation system for mobile robots in office-like

environments, but with much less computational resources. The navigation software has been validated on three different robots in an office-like environment as described in the next sections.

6.1 Experimental Environment

The environment considered in the experiments reported in this paper is a typical indoor environment in our Department, representative of other similar environments, including RoboCup@Home scenarios.

The total size of the environment is about 50mx50m for which a map was acquired at a resolution of 5cm, generating a grid of 1010x1070 pixels (illustrated in Figure 3) and given in input to the three robotic platforms.

6.2 Robots

Three different kinds of robots have been used in these experiments, illustrated in Figure 4.

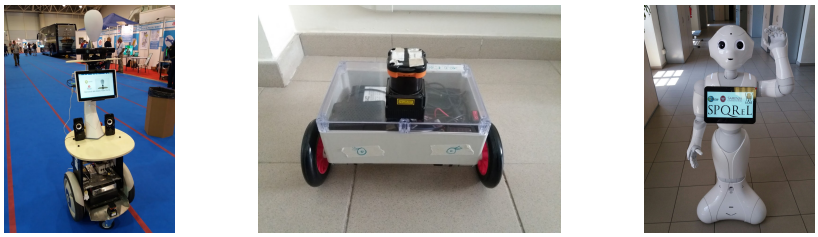

Fig. 4. Left: Robot Diago. **Middle:** Robot MARRtino. **Right:** SoftBank Robotics Pepper.

Diago[7] is a robot based on a Segway base and a home-made torso supporting many sensors and mainly used for Human-Robot Interaction (HRI) tasks. It includes a powerful laptop with Intel i7-6700 CPU @ 3.40GHz, 16GB RAM.

MARRtino[8] is a low-cost home-made robot controlled with an Arduino board and a Raspberry PI 3 Model B operated by a 1.2GHz Quad core ARMv8 CPU with 1GB RAM and running Ubuntu 16.04 and ROS Kinetic.

Pepper[9] is a robot developed by SoftBank, specially designed for HRI and social interaction. It has an omni-directional drive platform and carries an Intel Atom E3845 @ 1.91GHz Quad core as processing unit with 4GB RAM. Its development environment is NAOqi, while ROS is not supported on-board of the robot.

[7] https://sites.google.com/a/dis.uniroma1.it/diago/
[8] https://sites.google.com/dis.uniroma1.it/marrtino
[9] https://www.softbank.jp/en/robot/

All the robots are equipped with a laser range finder for localization and obstacle avoidance and a differential drive navigation mechanism. Although other sensors are available on some of these robots and Pepper is an omni-directional drive platform, these additional features are not used in the experiments reported here, in order to compare the three platforms on a common set of sensor and mobility features. It is worth to mention the challenge of approaching laser-based navigation for Pepper robot due to its limited range (about 5m) and sparsity of the provided laser data (45 points in a field of view of 240 degrees).

6.3 Experimental results

The experiments reported in this paper consist in the execution of a path in the environment with static and dynamic obstacles. The overall length of the path followed by the robots is about 30m. The robots start from a known initial pose and they come back to the same position. During this path we measured execution time and computational resources. Moreover, we qualitatively observed that the path performed by the robots and the way in which obstacles were avoided was adequate to the situation. This is demonstrated through videos of the use of the system available at the SPQReL website[10].

We have compared performance of our navigation system with respect to the ROS standard navigation modules `amcl` and `move_base`. We decided to use the default parameters for both the ROS modules and our components. This choice was motivated by our interest in evaluating these tools from a non-expert user perspective. We want to point out that the resources consumption of our software is minimal for the localization process when the robot is not moving, or for the path planning process when a goal is not active. For these reasons, measurements reported in this section consider the situation in which a goal is active and the robot is moving towards it.

Tables 1 and 2 summarize the computational resources consumption by the different software suites in terms of percentage of CPU and memory usage on the three platforms. As shown by the results, our implementation for the localization and path planning processes is more efficient in terms percentage of CPU usage with respect to the ROS standard modules while the differences on percentage of memory usage are anecdotal. It is worth to notice that the current implementation is single-thread for each component (one thread for the localizer and one thread for the planner), so other processes of the application can use other available cores.

We would like to point out that the values reported in Table 2 for the robot MARRtino running `move_base` are a best-case performance. We experienced several issues and warnings when running `move_base` on the Raspberry which, in general, were due to missed rates in control loops, robot pose requests or map updates. These issues make the experience of using `move_base` on the Raspberry difficult and confusing for students or other non-expert users.

[10] http://tinyurl.com/spqrel/photo-video

Table 1. Average computational resources required for the localization process on the different platforms with the ROS standard tools and our proposed software. Data obtained using the Linux command `top`.

Localization			
Robot	**Software**	**%CPU**	**%Mem**
Diago	ROS - `amcl`	5.65	0.2
	ROS - SPQReL	3.12	0.4
MARRtino	ROS - `amcl`	7.14	3.2
	ROS - SPQReL	9.06	5.6
Pepper	NAOqi - SPQReL	2.15	1.8

Table 2. Average computational resources required for the path planning process on the different platforms with the ROS standard tools and our proposed software. Data obtained using the linux command `top`.

Path planning			
Robot	**Software**	**%CPU**	**%Mem**
Diago	ROS - `move_base`	41.2	0.3
	ROS - SPQReL	25.72	0.8
MARRtino	ROS - `move_base`	103.72*	2.1
	ROS - SPQReL	98.07	11.15
Pepper	NAOqi - SPQReL	38.46	2.8

Furthermore, we have measured execution times of each cycle of our localization and path planning processes. Times are sumarized in Table 3. As shown in the table, if laser and odometry data are acquired at 10 Hz (typical case), only one core will be used at 100 %.

Table 3. Average execution times of each cycle of the localization and path planning processes on the different platforms.

	Cycle times	
Robot	**Localization**	**Path Planning**
Diago	2.55 ms	36.1 ms
MARRtino	15 ms	320 ms
Pepper	7 ms	104.25 ms

7 Conclusions

The open-source lightweight navigation system described in this paper and released as open-source and Debian package can improve and speed up development of many mobile robot applications, specially when low-cost robots or robots with limited computational resources are used. This feature is very important to spread the use of mobile robot applications and for educational purposes.

The system described in this paper can thus be very useful to young researchers that are willing to build robotic applications with limited resources. Some examples in which our software have been successfully applied is within RoboCup@Home SSPL and RoboCup@Home Education competitions in which standard navigation systems (e.g., ROS `amcl` and `move_base`) cannot run. Moreover, the open-source distribution allows the research community to improve the system (e.g., adding more features) and to increase its scope (e.g., porting it to other robotic platforms). Finally, we believe this software provides a didactic contribution when teaching subjects like autonomous robots or probabilistic robotics, since the code was actually developed in such a context.

Acknowledgements

The work is partially supported by the European Community's funded project 732773 'ILIAD', and the RoboCup Federation's Collaboration funds.

References

1. J.R. Asensio and L. Montano. A kinematic and dynamic model-based motion controller for mobile robots. In *The 15th IFAC Triennial World Congress*, Barcelona, Spain, July 21-26 2002.
2. Oliver Brock and Oussama Khatib. High-speed navigation using the global dynamic window approach. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 1, pages 341–346. IEEE, 1999.
3. Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. *AAAI/IAAI*, 1999(343-349):2–2, 1999.
4. Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.
5. Brian P Gerkey and Kurt Konolige. Planning and control in unstructured terrain. In *ICRA Workshop on Path Planning on Costmaps*, 2008.
6. Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. The office marathon: Robust navigation in an indoor office environment. In *International Conference on Robotics and Automation*, 2010.
7. Javier Minguez and Luis Montano. Nearness diagram (nd) navigation: collision avoidance in troublesome scenarios. *IEEE Transactions on Robotics and Automation*, 20(1):45–59, 2004.
8. Robin R Murphy, Ken Hughes, and Eva Noll. An explicit path planner to facilitate reactive control and terrain preferences. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 3, pages 2067–2072. IEEE, 1996.
9. Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. http://ompl.kavrakilab.org.
10. I. Ulrich and J. Borenstein. Vfh+: Reliable obstacle avoidance for fast mobile robots. In *IEEE International Conference on Robotics and Automation(ICRA '98)*, pages 1572–1577, 1998.