Environment Mapping using the Lego Mindstorms NXT and leJOS NXJ

Gerardo Oliveira[1], Ricardo Silva[1], Tiago Lira[1], Luis Paulo Reis[1,2]

[1] FEUP – Faculdade de Engenharia da Universidade do Porto, Portugal
[2] LIACC – Laboratório de Inteligência Artificial e Ciência de Computadores da Universidade do Porto, Portugal
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
{ei04106, ei03087, ei04085, lpreis}@fe.up.pt

Abstract. This paper presents a project of simultaneous localization and mapping (SLAM) of an indoor environment focusing on the use of autonomous mobile robotics. The developed work was implemented using the Lego Mindstorms NXT platform and leJOS NXJ as the programming language. The NXJ consists of an open source project which uses a tiny Java Virtual Machine (JVM) and provides a very powerful API as well the necessary tools to upload code to the NXT brick. In addition, the leJOS NXJ enables efficient programming and easiness of communication due to its support of Bluetooth technology. Thus, exploiting the mobile robotics platform flexibility plus the programming language potential, our goals were successfully achieved by implementing a simple subsumption architecture and using a trigonometry based approach, in order to obtain a mapped representation of the robot's environment.

Keywords: Lego Mindstorms, NXT, leJOS, NXJ, Java, Robotics, Mapping, Subsumption Architecture, Behavior-Based, Client-Server.

1 Introduction

One of the major problems in the field of robotics is known as SLAM (Simultaneous Localization and Mapping) [3], consisting of a technique used by robots and autonomous vehicles to build up a map within an unknown environment while at the same time keeping track of their current position. This is not as straightforward as it might sound due to inherent uncertainties in discerning the robot's relative movement from its various sensors. Thus, the elaboration of a project in this area has proved to be highly motivating for us since “solving” a SLAM problem is considered a notable achievement of the robotics research.

The robotic platform chosen for this project is the Lego Mindstorms NXT [4], a major improvement over their previous platform, the RCX [5]. It contains a more powerful processor, more memory, and employs enhanced sensor and actuator suites, therefore being considered the easiest way to enter in the robotics world.

For programming the NXT robot we decided to use leJOS [1, 6, 16, 17, 18], a firmware replacement for Lego Mindstorms programmable bricks, considered by many as the best platform in the moment to use software engineering ideas. The NXT brick is supported by leJOS NXJ [7], a project that includes a Java Virtual Machine (JVM) [8], so allows Lego Mindstorms robots to be programmed in the Java programming language. The leJOS platform proved to be the best choice for this project since it provides extensive class libraries that support various interesting higher-level functions such as navigation and behavior-based robotics, therefore simplifying the task of robotic piloting through the implementation of a subsumption architecture [9]. Also the API [10] support for the Bluetooth technology was essential in this work since we also implemented a client-server architecture, using the NXT robot as server and sending information to a client application via Bluetooth. Therefore, the whole logic is hosted and executed by the robot, which exchanges information with the client application, running on a computer. The data sent by the NXT is then processed and used in the environment map building.

In order to present the work developed for this project, this paper is organized as follows: Initially, in Sections 2 and 3 the hardware and software platforms are detailed, which is the Lego Mindstorms NXT and the leJOS NXJ, correspondingly. In Section 4 we give an overview of the architectures behind the developed robotic agent, namely the subsumption and client-server architectures. Section 5 details the environment mapping procedures that were used. Finally, in Sections 6 and 7, we give details about the tests we ran and their results, stating the conclusions we achieved and thereby concluding the article.

2 Lego Mindstorms NXT

Lego Mindstorms NXT is a programmable robotics kit released by Lego in late July 2006. The NXT is the brain of a Mindstorms robot. It's an intelligent, computer controlled Lego brick that lets a Mindstorms robot come alive and perform different operations.

Fig. 1. The Lego Mindstorms NXT. In the center, the NXT brick *1*. Above, the three servo motors *6*. Below, the four sensors: touch *2*, sound *3*, light *4* and ultrasonic *5*.

The NXT has three output ports for attaching motors – Ports A, B and C and four input ports for attaching sensors – Ports 1, 2, 3 and 4. It's possible to connect a USB cable to the USB port and download programs from a computer to the NXT (or upload data from the robot to the computer). As an alternative the wireless Bluetooth connection can be used for uploading and downloading.

In the specific case of our project we simply needed two motors (connected at ports A and C), but we used all the input ports for sensors (Ultrasonic Sensor – port 1, HiTechnic [20] Compass Sensor – port 2, HiTechnic Color Sensor – port 3, Sound Sensor – port 4).

3 leJOS NXJ

The name leJOS was conceived by José Solórzano, based on the acronym for Java Operating System (JOS), the name of another operating system for the RCX, legOS, and the Spanish word "lejos".

The leJOS NXJ [7] is a project to develop a Java Virtual Machine (JVM) for Lego Mindstorms NXT, as a replacement of the original firmware of the NXT brick, thereby allowing the NXT robots to be programmed in the Java programming language, providing extensive class libraries that support various interesting higher level functions such as navigation and behavior based robotics.

Started out as a hobby open source project, by José Solórzano in late 1999, the leJOS NXJ has been evolving along the time with new features. Currently leJOS Research Team has launched the release 0.7, available for both Microsoft Windows and Linux/Mac OS X operating systems. This version includes a Windows installer to make installation a breeze for new users. Also, in order to facilitate the tasks of uploading the latest firmware into the NXT brick, uploading NXJ programs to the NXT brick and converting any Java project into NXT project, the leJOS Research Team developed a plug-in for the Eclipse IDE [11], which turn these tasks into a children's play.

From our personal experience with the leJOS NXJ platform, it proved to be an excellent choice, not only for this work, but for any kind of robotic projects using the Lego Mindstorms NXT [1], because of all the offers of the leJOS NXJ, like its preemptive threads, synchronization, multi-dimensional arrays and its well-documented robotics API, for example.

4 Architecture

In this project we made use of two software architectures: subsumption and client-server architectures, which are subsequently detailed.

Subsumption architecture is a methodology for developing artificial intelligence robots. It is heavily associated with behavior-based robotics. The term was introduced by Brooks [2] (1991), which enforces a hierarchy among behaviors so that some of them in certain conditions would inhibit other lower priority ones. A subsumption architecture is a way of decomposing complicated intelligent behavior into many

"simple" behavior modules, which are in turn organized into layers. Each layer implements a particular goal of the agent, and higher layers are increasingly more abstract. Each layer's goal subsumes that of the underlying layers. As opposed to more traditional artificial intelligence approaches, subsumption architecture uses a bottom-up design.

The choice of this architecture allowed us to address each type of action independently in different behaviors and give them a set priority. This allows for more organized and alterable code, and avoids problems in the interaction between the different behaviors.

In order to make a better research work, we approached the mapping problem in two fronts, both using a subsumption architecture. Basically one using the compass sensor and another without using it, where calculations are based on the initial position of the robot and its following movements.

For the first approach mentioned above, we used the subsumption architecture to implement a basic navigation mechanism by developing a lowest layer “obstacle avoidance”, which corresponds to the rotate (turn around its center) and retreat (drive backward) behaviors, and on top of it lays the “environment exploration” layer, implemented by a drive forward behavior. Each of these horizontal layers access all of the sensor data and generate actions for the actuators – the main caveat is that separate tasks can suppress (or overrule) inputs or inhibit outputs. This way, the lowest layers can work like fast-adapting mechanisms (e.g. reflexes), while the higher layers work to achieve the overall goal. Feedback is given mainly through the environment.

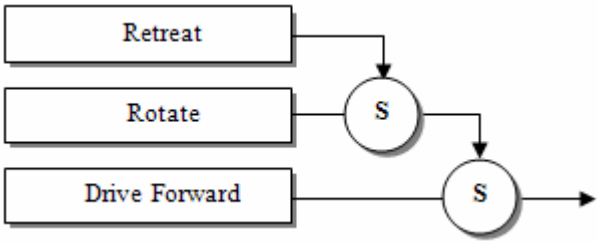

Fig. 2. Representation of the subsumption architecture implemented for the first approach. “Retreat” and “Rotate” correspond to the layer “Obstacle Avoidance” and the “Environment Exploration” is implemented by the “Drive Forward” behavior.

For the second approach, we applied a subsumption architecture also based on the sensor read values, but this time the behaviors are more independent, given that even a behavior with a lower priority will end its turn before another takes place. This happens because the conditions that decide the change of behavior are calculated in the end of that behavior. All the measures and calculations are merged in order to map the environment with all the data collected by that time.

This approach has two main behaviors, the Measure behavior and the Navigate behavior. When on the Measure behavior, the robot rotates 360 degrees and at the

same time it measure the distances given by the sonar sensor on the four quadrants. When on the Navigate behavior, the robot drive forward a previous defined distance (for our tests we used distance = 30 units, where each unit correspond to a unit given by the sonar sensor). This way, when we join these two behaviors we get a step-by-step very reasonable map.

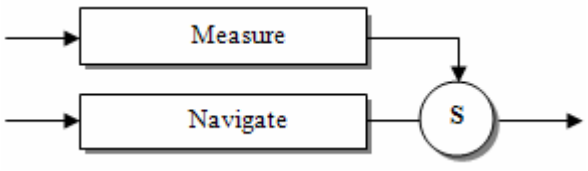

Fig. 3. Representation of the sumbsumption architecture implemented for the second approach. "Measure" and "Navigate" complement each other even if "Measure" has priority over "Navigate".

This kind of architecture (subsumption) could be implemented by the packages provided by the leJOS NXJ, which has an interface (Behavior) and a class (Arbitrator) to control the whole process:

lejos.subsumption.Behavior

An interface that represents an object embodying a specific behavior belonging to a robot. Each behavior must define the following three methods:

– boolean takeControl()

Returns a boolean value to indicate if this behavior should become active. For example, if the ultrasonic sensor indicates the robot is too close to an object, this method should return true.

– void action()

The code in this method initiates an action when the behavior becomes active. For example, if takeControl() detects the robot is too close to an object, the action() code could make the robot rotate and turn away from the object.

– void suppress()

The code in the suppress() method should immediately terminate the code running in the action() method. The suppress() method can also be used to update any data before this behavior completes.

lejos.subsumption.Arbitrator

The arbitrator regulates which behaviors should be activated, once all the behaviors are defined.

The client-server architecture model distinguishes client systems from server systems, which communicate over a computer network. A client-server application is a distributed system comprising both client and server software.

Thereby, given the fact that the robot doesn't have a great capacity for data processing, not all the calculations were made by the server (NXT brick). In our approaches we needed to send some sensorial data and processed data for a computer client application to better analyze and expose that information. The nature of that data focus on the orientation angle of the robot (obtained with the compass sensor), the distances to the obstacles (acquired by the ultrasonic sensor) and the values of a matrix representing what it "sees", that the robot keeps updating. This process of data exchange between different applications was implemented using the leJOS NXJ, via the Bluetooth wireless protocol.

Fig. 4. The client-server architecture used for communication, by means of the Bluetooth wireless protocol, between the computer application (client) and the NXT brick (server).

While the client server offered a wide range of possibilities, as remote controlled commands for example, we chose to keep the robot as autonomous as possible. Therefore this architecture was used only as a means of reporting observational data to the computer for visualization.

5 Environment Mapping

Robotic mapping [12] has been extensively studied since the 1980s. Mapping can be classified as either metric or topological. A metric approach is one that determines the geometric properties of the environment, while a topological approach is one that determines the relationships of locations of interest in the environment. Mapping can also be classified as either world-centric or robot-centric. World-centric mapping represents the map relative to some fixed coordinate system, while robot-centric mapping represents the map relative to the robot. Robotic mapping continues to be an active research area.

The environment mapping with the Lego Mindstorms NXT discussed in this paper uses a metric, world-centric approach, in order to use the simplest possible mapping algorithm. As described earlier in this document, the robot moves forward from a known starting point until it detects an obstacle with its ultrasonic sensor which is simply avoided by rotating over itself or by driving backward if too close to the object. While doing this exploration of the robot's environment, the NXT brick uses a

80x80 matrix as the representation of the world. A matrix with this dimension covers about 4x4 meters of real terrain, allowing for a more detailed representation of small spaces, while simultaneously maintaining a reasonable size that the limited memory of the robot can support. At the same time, while doing its internal mapping, the NXT transmits its traveled distance (determined with the help of the motors tachometers) and orientation data (obtained with the compass sensor) to the client application running on a computer.

Therefore, our map is built on the robot in runtime, while exploring the world and simultaneously on the client program, since it allows for a much richer map representation than the NXT display.

For the measuring of its own position, the robot registers the value of the tachometers and the direction angle from the compass. In each movement the compass value is frequently updated in an attempt to reduce odometry errors. As the tachometers are much more precise than the compass, this allows the robot to have a reasonable precision in estimating its current location.

Our approach to environment mapping was to use the NXT sensorial data to calculate the absolute position of obstacles. The robot has knowledge of its starting positions and then uses simple trigonometry to calculate its absolute positions and orientations.

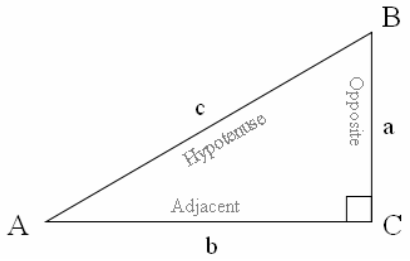

Fig. 5. Representation of the principle used for determining the absolute position of obstacles. In this right triangle: sin A = a/c; cos A = b/c.

With a bit of trigonometry, the absolute position of the detected obstacles can be calculated by using the sine and the cosine trigonometric functions, therefore resulting on the following expressions to determine the *x* (1) e *y* (2) coordinates of a certain obstacle, given the *xi* and *yi* coordinates from the previous robot position as well as the traveled distance (*distance*) and orientation (α) of the robot.

$$x = xi + distance \times \cos \alpha \, . \quad (\mathbf{1})$$

$$y = yi + distance \times \sin \alpha \, . \quad (\mathbf{2})$$

Thus, the program builds an environment map by calculating the absolute position of obstacles using trigonometry, robot ultrasonic sensor readings and previous calculations of absolute robot positions and orientations.

These calculations allow for sweeping an area, as the robot rotates, in each iteration marking the walls and free spaces in that direction.

Still, the distance sensor and the compass are relatively imprecise, and their results are often mistaken. To make the best of these readings, we used a system of probabilistic mapping based on the Bayes method. In blunt the calculated value is the probability of having a free space in a determined point of the matrix. A low value means high probability of there being an obstacle, while a high value represents a elevated probability of having walkable terrain. Each time a position is tested, the old value is combined with a new one to make an average probability (2). This new value conveys the probability of having a free space or an obstacle, and is relative to the distance. If a position has no valid value in it, it means that position has never been tested before, so we can't use the previous value to calculate the new one. In that case we use a formula where the probability of being occupied is assumed previously (1). We logically defined that the probability of being occupied equals the probability of not being occupied.

$$P(H|s) = (P(s|H) \times P(H)) / (P(s|H) \times P(H) + P(s|\sim H) \times P(\sim H)) . \quad (3)$$

$$P(H|s_n) = (P(s_n|H) \times P(H|s_{n-1})) / (P(s_n|H) \times P(H|s_{n-1}) + P(s_n|\sim H) \times P(\sim H|s_{n-1})) . \quad (4)$$

These formulas gave satisfactory results and were therefore chosen to be the final method. In practice it allows us to take into account diverse factors like distance, repetition of results, and softening of reading errors, resulting in a well balanced and flexible representation of the map.

Additionally, while the robot is designed for static environments, probabilistic mapping allows it to dynamically react to changes in the environment, like adding obstacles or opening new paths.

All these calculations are adapted to be processed on the very limited resources of the robot, which then communicates the final result to the client computer via Bluetooth, for real-time visualization of the mapping process.

6 Results

Despite its limited sensors, the robotic agent is able to create a sufficiently realistic map of its environment.

Fig. 6. Example of an environment map representation. In the left side, the real environment and in the right side the map obtained by the first approach.

To exemplify the mapping capabilities of our agent, we have some results of the MapViewer implemented on the client side, displayed in figure 6.

For the first approach, the compass sensor is easily affected by magnetic fields of nearby objects, and this can lower the quality of the mapping in some places. Another fact that limits this approach is the complexity of coordinate all the movements and all the angles that the robot (compass sensor) can read, since the robot, in this approach can walk to every side in 360 degrees. If the robot gets a wrong read of the compass, it will mark the corresponding position with a wrong value too and walk to a certain position thinking it is another.

Each of the small squares in the visualization window represent a position of the matrix, corresponding to roughly 4.3 centimeters of real terrain in its side. The red points correspond to the trajectory of the robot. The rest represent the probability of obstacles at each position. Orange means high probability of free space, decreasing into shades of yellow. White means a 50% probability of free space, while lower probabilities (probable walls), are represented in shades of gray. Increasingly darker grays, represent greater certainty in the position of the walls.

In general, orange and black spaces will dominate the areas that the robot has seen repeatedly or at a close distance, indicating the parts of the map where risk of error is lowest.

However, our second approach, while being less flexible, works without using the compass, and therefore has no problem in the aforementioned situation. This way we get none of the problems that the compass brings, but we also don't get the advantages, so if the robot rotates too much it will not know and all the following values and map representation will not take in account that error. However, if the robot keeps its navigation almost without errors, a nice and pretty realistic map will be the result of our second approach.

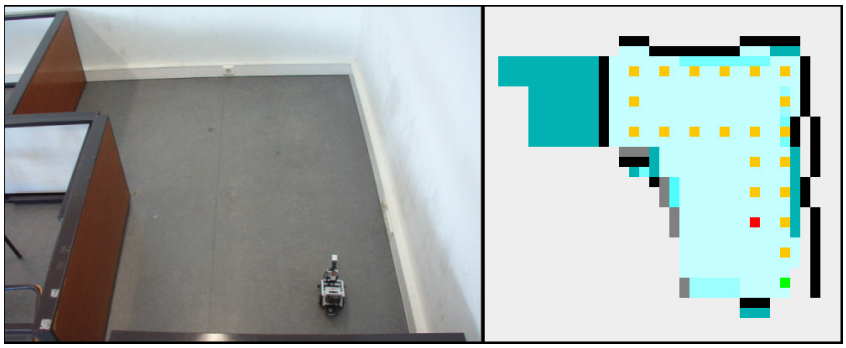

Fig. 7. Example of an environment map representation. In the left side, the real environment and in the right side the map obtained by the second approach.

In this approach, we used different measures and colors to specify the probabilities. Here, each square represent 10 units, where each unit correspond to a unit measured by the sonar sensor. The green spot is the initial position and the red one is the actual position. The brighter zones correspond to locals where the probability of being

empty is high (white and brighter blues), while the darker zones correspond to the occupied places (dark cyan to black).

Fig. 8. Example of an environment map representation. In the left side, the map obtained by the first approach, in the middle, the real environment and in the right side the map obtained by the second approach.

While we are satisfied with the results, we realize that they could be vastly improved by using more powerful sonar and compass sensors. The fact of having only one sonar vastly reduces the capacity of analyzing correctly the directions that should be taken when facing an obstacle and the real distances to them or to walls. Another difficulty was the compass errors. Even though the compass often gives good results, a bad read causes a wrong interpretation and correspondingly a false representation of the map.

7 Conclusions

Environment mapping for real robots is a complicated problem, as the system resources are limited and the operation of physical parts (motors and sensors) is highly susceptible to errors and imprecision. The inherent uncertainties in discerning the robot's relative movement from its various sensors turn mapping into a very complex and challenging task. As a matter of fact, a robot can be compared to an insect, for which the environment is not interpreted as a map, and they survive only with a triggered response (e.g. reflexes). Therefore, it was our goal to provide the robot with intelligence in order to combine the information from the past and the present, thus creating a good mapping algorithm, and fortunately we achieved that.

One factor that has contributed to our success was the flexibility of the Lego Mindstorms NXT platform associated with the potential of the leJOS API, which allowed us to program the NXT in the Java programming language, thereby giving us much more control over the robot’s sensors and actuators, manifesting itself by a more powerful programming than the one given by the original NXT-G [13] software that Lego provides.

The probabilistic mapping method based on the Bayes’ rule proved to be a serious improvement, allowing for flexibility in the creation of the map. The variation of probabilities softens the erratic results, progressively creating a better map.

The two approaches we used allowed us to have different processes of solving a map, and each is advantageous in a certain set of conditions. The first approach is more flexible and gives more detailed results, while the second one is more stable and less sensitive to sensor errors.

As future work, since one of the biggest troubles while trying to "solve" a SLAM problem are the inaccuracies associated with the measured distance and direction travelled, then any features being added to the map will contain corresponding errors. In this manner, if not corrected, these positional errors build cumulatively, grossly distorting the map and therefore the robot's ability to know its precise location. Thus, it would be interesting, in this project's concern to study techniques to compensate for this, such as recognizing features that it has come across previously and re-skewing recent parts of the map to make sure the two instances of that feature become one. The application in our project of other statistical techniques, such as Kalman filters [14, 19] or Monte Carlo methods [15], for instance, would definitely improve our work.

Acknowledgments

This research was supported by the Portuguese Science and Technology Foundation (FCT) under the project "ACORD – Adaptative Coordination of Robotic Teams" (PTDC/EIA/70695/2006), funded through the POCI/FEDER program.

References

1. Juan Antonio Breña Moral, "Develop leJOS Programs Step by Step", Version 0.4, July 2008
2. Brooks, R. A., "Intelligence without Reason", In Proceedings of the 12th International Joint Conference on Artificial Intelligence, pp. 569-595. Sydney, Australia, 1991
3. Cyrill Stachniss, Udo Frese, Giorgio Grisettil, "Simultaneous Localization and Mapping (SLAM)", Available HTTP: http://openslam.org/
4. The LEGO Group, Lego Mindstorms NXT, Available HTTP: http://mindstorms.lego.com/
5. The LEGO Group, Lego Mindstorms RCX, Available at: http://en.wikipedia.org/wiki/Lego_Mindstorms#RCX
6. Jose Solorzano, "leJOS: Java for LEGO Mindstorms", Available at: http://lejos.sourceforge.net/
7. Jose Solorzano and others, "leJOS NXJ: NXJ technology", Available at: http://lejos.sourceforge.net/nxj.php
8. Sun Microsystems, Inc., Java Virtual Machine (JVM), 1994-2009, Available at: http://java.sun.com/docs/books/jvms/
9. Joe Jones, Daniel Roth, "Robot Programming: A Practical Guide to Behavior-Based Robotics", McGraw-Hill/TAB Electronics, 1st edition, 2003 December 12
10. ComputerWorld, "QuickStudy: Application Programming Interface (API)", Available at: http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=43487
11. The Eclipse Foundation, Eclipse IDE, Available at: http://www.eclipse.org/
12. Andreas Nüchter, "3D Robotic Mapping: The Simultaneous Localization and Mapping Problem with Six Degrees of Freedom", Springer, 1st edition, 2009 January 22

13. The LEGO Group, NXT-G: LEGO MINDSTORMS NXT Software, Available at: http://mindstorms.lego.com/eng/Overview/NXT_Software.aspx
14. Greg Welch, Gary Bishop, The Kalman Filter: Some tutorials, references, and research, Available at: http://www.cs.unc.edu/~welch/kalman/
15. Christian P. Robert, George Casella, "Monte Carlo Statistical Methods", Springer; 2nd edition, 2005 July 26
16. Brian Bagnall, "Maximum Lego NXT: Building Robots with Java Brains", Variant Press, 2nd edition, 2007 April 30
17. Martijn Boogaarts, Jonathan A. Daudelin, Brian L. Davis, Jim Kelly, Lou Morris, Fay and Rick Rhodes, Matthias Paul Scholz, Christopher R. Smith, Rob Torok, Chris Anderson, "The LEGO MINDSTORMS NXT Idea Book: Design, Invent, and Build", No Starch Press, 2007 August 28
18. Owen Bishop, "Programming Lego Mindstorms NXT", Syngress, 2008 June 9
19. Peter Maybeck, "Stochastic Models, Estimation, and Control", Academic Pr, Volume 1, 1979 June
20. Dataport Systems, Inc, HiTechnic Products, Available HTTP: http://www.hitechnic.com/