Umeå University
Department of Computing Science
Master’s Thesis Project D, 20 credits

ARIA and Matlab Integration With Applications

Jonas Borgström c00jbm@cs.umu.se

3rd April 2005

Supervisor:
Thomas Hellström

Abstract

The course *Intelligent Robotics* at Umeå University let, like many other similar courses, their students explore different algorithms and architectures using real mobile robots. There is unfortunately no standardized way to interface with robots. Instead each manufacturer provide software developers kits (SDKs) for their own products. This is especially troublesome for robotics courses where students should be able to concentrate on the actual subject and not be forced to spend a lot of time learning to use complex SDKs and even new programming languages. The course at the Umeå University uses robots from two different manufacturers. Khepera robots are controlled from Matlab and AmigoBot robots from C++. Most students are already familiar with the Matlab programming language and feel a bit intimidated by the AmigoBot's powerful but complex C++ SDK named ARIA.

This master's thesis project is divided into two parts. The first part describes the design and implementation of an adapter layer between the ARIA library and Matlab. This adapter layer allows ARIA-based robots to be controlled directly from within Matlab without using the C++ SDK directly. In the second part of this project a full scale SLAM-application is created that explores some important topics in the robotics field, such as map making and continuous localization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The course *Intelligent Robotics* at Umeå University let, like many other similar courses, their students explore different algorithms and architectures using real robots. Unfortunately there is no standardized way to interface with robots. Instead each manufacturer provide SDKs[1] for their own products. This is especially troublesome for robotics courses where students should be able to concentrate on the actual subject and not be forced to spend a lot of time learning how to use different SDKs and even new programming languages.

The course uses two different types of robots: AmigoBots from ActivMedia and Khepera robots from K-Team. Khepera robots are controlled from Matlab and AmigoBot robots from C++ or Java. Most students are already familiar with the Matlab programming language and the dynamic nature makes it very suitable for the "trial and error" way of working that is common when experimenting with robots and algorithms. AmigoBots use an advanced C++ SDK called ARIA. This SDK is very powerful and well designed but has a prominently steeper learning curve compared to the rather simplistic Khepera SDK.

1.1 Goal

The goal of this master's thesis project can be divided into two parts:

The first part is to create an easy to use adapter layer between Matlab and ARIA-based robot. This layer should provide students and other developers with an easy to use and stimulating developing environment in which a few lines of Matlab code should be enough to create simple robotics programs.

In the second part, a full-scaled SLAM[2] application is developed. The purpose is to explore important topics within the robotics field such as map making and continuous localization. This application also serves as a test-framework for the adapter layer developed in part one. Experiments with this application illustrate

[1]Software Development Kit.

[2]Simultaneous Localization And Mapping.

how odometry errors affect map making and how efficient continuous localization is when dealing with this problem.

1.2 Method

In order to determine the best design and implementation for the adapter layer, the ARIA and Matlab documentation and API was studied in great detail. Effort was also made to make sure the resulting API was simple and self-explanatory enough for new users without extensive training. Furthermore it should have enough similarities with the existing C++ API for existing ARIA developers to feel comfortable and at home.

Building a SLAM-application required a lot of information searching. Searching for SLAM-related articles using search terms such as “localization” and “mapping” on CiteSeer[3] and Google were the primary sources of information. Besides searching for and reading articles a lot of time was spent on the actual implementation and experiments. The algorithms used and the application itself required a lot of tweaking and testing in order to produce good results.

1.3 Report Outline

The rest of this thesis paper follows the following disposition:

Chapter 1 This chapter.

Chapter 2 Contains a detailed study of the ARIA and Matlab internals in order to determine the best design of the adapter layer.

Chapter 3 Describes the design and implementation of the adapter layer.

Chapter 4 Describes the theory behind map making and localization.

Chapter 5 Descirbes the design and implementation of a SLAM-application as well as experiment results.

Chapter 6 Summary and conclusions.

Chapter 7 Acknowledgements.

Appendix A Adapter Programming Tutorial.

Appendix B Adapter API Documentation.

Appendix C SLAM-application source code.

[3]http://citeseer.ist.psu.edu/

Chapter 2

ARIA and Matlab

This chapter introduces ARIA and Matlab, the two major components involved in the creation of the adapter layer.

2.1 ARIA

ARIA[1] is an open source object oriented interface to ActivMedia mobile robots (see section 2.2 for a complete list of compatible robots). The interface is implemented as a cross platform[2] C++ library. Besides a lot of robot related functionality the library also contains wrappers around platform specific functions such as threading and networking.

The ARIA library is primarily designed for professional developers and is meant to be used as a solid base for large-scale applications. The library architecture is very flexible so the library can be used by both multi-threaded and single-threaded applications. ARIA communicates with the robot using either a serial tether or a TCP/IP connection.

Coordinate System

While a robot navigates in the environment, ARIA always keeps track of the robot's pose using the built in shaft encoders. The robot's believed pose can always be accessed and updated using the *getpose* and *move* methods.

The robot's pose is expressed as a point on the xy-plane and an angle θ. When the robot powers up the initial pose is set to $x = 0$, $y = 0$, $\theta = 0$, and the robot is looking along the positive x-axis. If the robot rotates +90 degrees (counter clock wise) it will be looking along the positive y-axis. The x, y position is expressed in unit of millimeters and the θ angle in the interval of +-180 degrees counter clock wise.

[1]Active Robotics Interface Application.

[2]So far only supported on Linux and Microsoft Windows.

2.2 Supported Robots

ARIA supports robots of all difference sizes and shapes. The smallest supported robot is the AmigoBot, it weighs 3.6 kg and is equipped with eight sonar sensors. On the other side of the spectrum is the PowerBot, the largest supported robot which weighs 120 kg and is equipped with 28 sonar sensors, laser mapping and a gripper arm.

Most, if not all robots manufactured by ActivMedia are supported by the ARIA library, table 2.2 contains a complete list.

Table 2.1: Robots supported by ARIA [2]

AmigoBot$^{\text{TM}}$	- Classroom and team robot.
Pioneer 3-AT	- High performance all-terrain robot.
Pioneer 3-DX	- Research and educational robot.
PatrolBot$^{\text{TM}}$	- A surveillance robot.
PowerBot$^{\text{TM}}$	- High-agility, high-payload robot.
PeopleBot$^{\text{TM}}$	- Human interface robot.

2.3 The ARIA API

ARIA has a large and powerful API [1]. The API contains a set of high-level functions that allow developers to control the robot using both built-in and user defined actions[3]. ARIA allows programmers to define new actions by subclassing `ArAction` and overriding a few class methods. These actions are run in a background thread allowing the main application thread to deal with other things than the robot's basic behavior.

It is also possible to directly control the robot's movements and sensors using a set of lower-level functions. These functions command the robot to perform different tasks such as "Give me the range reading from sonar sensor X," "Move forward with velocity Y", etc. ARIA also contains an even lower-level API that allows users to communicate with and send commands directly to the robots onboard computer, this approach however requires detailed knowledge about the properitary protocol used between ARIA and the robot.

[3]Also known as behaviors or behavioral actions.

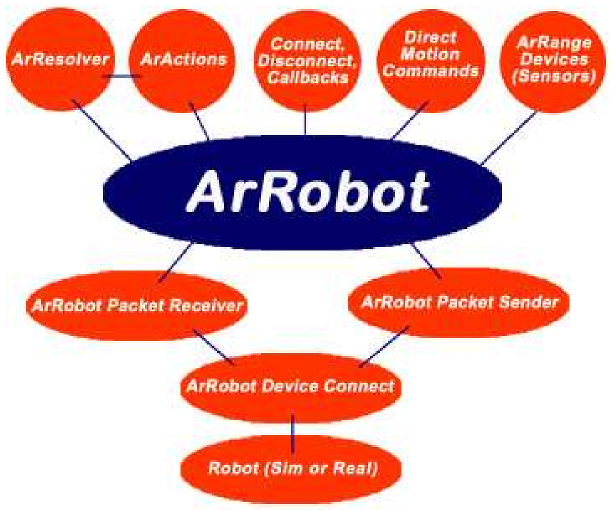


Figure 2.1: ARIA API block diagram. Figure taken from the ActivMedia Robotics Inc web page.

2.4 Matlab

Matlab is a high-level technical computing language developed by MathWorks. Their goal was to provide engineers and scientists with a more powerful and productive environment than was provided by popular programming languages like Fortran, C and C++ [11].

Matlab is not only a high-level language for technical computing. The Matlab application is more of a development environment with features like code management, interactive tools for iterative exploration, design and problem solving.

2.5 Integrating Matlab

Matlab can be integrated with other languages and applications in a at least two different ways:

- An application written in C or Fortran can use Matlab as a computation engine by using the Matlab engine library. This library allows the application to communicate with a Matlab instance via a pipe on Unix platforms and a COM-interface on Windows platforms. This allows an application to send and receive data to and from Matlab, and to process that data by calling Matlab functions.

- The other way around is also possible. Functions written in languages such as C and Fortran can also be used from within Matlab. This is achieved by using MEX-files[4]. A MEX-File consists of two distinct parts. The first part contains the routine implementing the wanted functionality. It can be a highly optimized numerical algorithm, a communication routine for some exotic hardware or something completely different.

The other part is the gateway routine. Its responsibility is to interface the computational part with Matlab. This includes verifying that the MEX-function is called with valid left- and right-hand side arguments. The gateway routine calls the computational routine after the arguments are verified.

The main reason why people sometimes use MEX-extensions instead of plain Matlab code is performance. Implementing computationally expensive algorithms in compiled languages like C/C++ or Fortran sometimes yield significant performance improvements.

MEX-extensions are also required when accessing some operating system dependent functions and use external libraries, or when communicating directly with hardware.

2.6 Matlab Limitations

The MEX API [9] and the Matlab language itself imposes a number of limitations on what an extension is able to do. The following list describes some of these limitations, their effect on this work and possible workarounds:

- *Not thread-safe*: The ARIA library can be run both single-threaded and multi-threaded, but Matlab itself is a strictly single-threaded application. This means that some of the functionality in the ARIA library will not be available from within Matlab. Fortunately most of the functionality is still available for single-threaded applications.

- *Ad-hoc OOP[5] support*: MathWorks has in recent versions of Matlab added object-oriented programming support. Basic oop-features such as function overloading, data encapsulation and inheritance are supported, but it is not as advanced as other oop-languages such as C++ or Java. Creating new classes in Matlab is both complicated and requires a lot of code. Each new class requires a directory on the filesystem that contains one Matlab file for each class method. Exporting a C++ api as *real* Matlab classes require even more elaborate work that will be described in more detail in chapter 3.

- *No destructor support*: The oop-system used by Matlab lacks the concept of destructors used by most other oop-languages. A destructor is a special class method that is invoked just before a class instance is garbage collected. Destructors are traditionally used by class instances to return resources back

[4]Matlab Extension components.
[5]Object-Oriented Programming.

to the operating system. This missing feature might not seem that important given that Matlab's garbage collection will take care of values no longer used. Unfortunately this is only true as long as no MEX-extensions are used. Some MEX-extensions may allocate memory that explicitly has to be freed in order to avoid memory leaks. This limitation makes it hard to export C++ class instances into Matlab. Chapter 3 presents a (partial) solution to this problem.

- *Limited type-system*: Matlab uses a somewhat limited type-system. It is for example impossible to add new classes at run-time. The only way to add a new class to the system is to physically create a new directory named *"@"+class_name*. The parent directory of this newly created directory also has to exist in the Matlab path.

- *One way communication between a MEX-routine and the Matlab core*: The MEX-API in combination with the single-threaded nature of Matlab makes it impossible for an extension to notify the main Matlab process of events, unless Matlab explicitly asks (polls) for them. With this limitation it is not possible to (from within Matlab) register callback-functions for ARIA to call when special events like lost contact with the robot occurs.

- *Only one computation routine in each MEX-file*: MEX-extensions are implemented as shared libraries in UNIX and as dynamically loaded libraries in Windows. When it comes to locating a MEX-extension Matlab uses the same naming convention as it does with *normal* functions written in Matlab, MEX-extensions must have the same filename as the name of the computation routine they contain[6].

  This naming convention is very easy to use but has at least one major drawback. It will require a lot of redundant code to create Matlab extensions for a C++ library with tens or hundreds of classes and methods. This is because a MEX-extension file has to be created for each of the class methods in the C++ library. Another problem when keeping related class methods in different shared libraries is that it becomes hard to share data between related functions that due to this limitation is located in different shared library extensions. Section 3 will describe a technique making it possible to limit the number of MEX-extensions required to export an entire C++ API to just one.

[6]MEX-extensions have a platform specific file type, ".dll" on Windows and ".mexglx" on Linux.

Chapter 3

Adapter Layer

This chapter describes the design and implementation of the adapter layer between ARIA and Matlab. The core of this layer is a single MEX-extension. The Matlab MEX API [11] is originally designed to make it possible to create and use highly optimized C/C++ or Fortran versions of individual algorithms and not to expose an entire C++ API. Fortunately that does not make it impossible. This chapter will describe how to embed multiple functions in a single MEX-extension.

3.1 MEX-Extensions

Another way to look at MEX-extensions is that a MEX-extension is basically nothing more than a way to pass a Matlab matrix to a C/C++ or Fortran function in a shared library, and a way for that function to send another matrix back to Matlab.

Matlab contains some additional functionality that allows users to invoke these extensions just like any other Matlab function-call. The parameters used in this function-call will be passed to the extensions gateway routine. The gateway can also return a matrix back to Matlab that will be treated as the functions return value.

3.2 C++ Classes In MEX-Extensions

Given that the OOP-system in Matlab already requires the user to create a separate Matlab file for each class method in every class, we might as well try to do something clever in them to avoid having to create a separate MEX-extension for each class method as well.

The only thing stopping us from adding multiple *computational routines* to a single MEX-extension is that the *gateway routine* needs some way to identify the computational routine requested. A simple solution is to prepend a routine-identifier before the actual routine parameters when invoking the MEX-function. This trick will of course make the Matlab source code required to invoke this MEX-function look a bit strange. Fortunately that source code will only live in the mandatory class-method files and will never be seen by the user.

For example the *robot* class constructor (file *@robot/robot.m*) only needs to contain the following Matlab code:

```
function o = robot(address)
self.cnx = aria_adapter(1, address);
o = class(self, 'robot');
return
```

The constructor call invokes the *aria_adapter* MEX-extension and requests to invoke the internal routine number one, the `ArRobot` constructor wrapper function. The `ArRobot` instance identifier returned by the wrapper is stored in the instance variable *self.cnx*. This instance identifier will later be used when invoking other class methods.

Even less Matlab code is required to invoke a class method. The Matlab part of the *getpose* class method (file *@robot/getpose.m*) looks like this:

```
function pose = getpose(self)
pose = aria_adapter(4, self.cnx);
return
```

The routine number four in *aria_adapter* is invoked with the `ArRobot` instance identifier as the only parameter. The wrapper function invokes the `ArRobot::getPose()` C++ method and converts the pose (position and heading) into a Matlab matrix and returns it back to Matlab.

3.3 The aria_adapter Extension

The C++ part of the adapter layer is a bit more complicated than the Matlab part. When any of the wrapped class methods are invoked, the *aria_adapter* extension is loaded into memory by Matlab and its gateway routine is invoked. Every MEX-extension has a gateway routine, it is a C function named *mexFunction* that serves as the extension entry point and is invoked by the Matlab binary every time the extension is invoked.

The main responsibility for the *aria_adapter* gateway routine is to verify that the requested ARIA function exist, and that it is called with the correct number of left and right hand side parameters, and to hand over the control to the associated wrapper function. In order to make it easy to add new functions to the adapter layer and to avoid code redundancy, the gateway routine itself does not know anything about any specific wrapped ARIA method. Instead the gateway routine only contains generic functionality to perform those tasks. The actual implementation of the gateway routine in *aria_adapter* looks like this (a few lines have been removed to improve readability and to save space):

```
void
mexFunction(int nlhs, mxArray *plhs[],
            int nrhs, const mxArray *prhs[])
{
    if (nrhs < 1)
        mexErrMsgTxt("At least one rhs argument is required.");

    if (mxGetM(prhs[0]) != 1 || mxGetN(prhs[0]) != 1 ||
        !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0])) {
        mexErrMsgTxt("The first rhs argument has to be a scalar double.");
    }
    arg1 = mxGetPr(prhs[0]);
    fid = (int)arg1[0];

    // Make sure its a valid function and the correct number
    // of arguments
    for (i = 0; i < NUMFUNCTIONS; i++) {
        if(functionmap[i].id == fid) {
            if (functionmap[i].nlhs != nlhs)
                mexErrMsgTxt("Incorrect number of lhs arguments");
            if (functionmap[i].nrhs != (nrhs - 1))
                mexErrMsgTxt("Incorrect number of rhs arguments");
            break;
        }
    }
    if (i == NUMFUNCTIONS)
        mexErrMsgTxt("Unknown function id");
    // Call the selected function wrapper
    functionmap[i].func(nlhs, plhs, nrhs, prhs);
}
```

The function table contains one entry for each embedded function wrapper. Besides the function id and function name every function entry also contains the correct number of left-hand side (return values) and right-hand side arguments (parameters) and a pointer to the wrapper function.

Using this information the gateway routine can take care of things such as verifying that the function id is correct and that the correct number of left and right hand side arguments are used. If the concept of one MEX-extension for each routine was used, this code would have to be duplicated in every extension.

```
functiontable[] = {
  {1,  "robot_connect",            1, 1, robot_connect},
  {2,  "robot_disconnect",         0, 1, robot_disconnect},
  {3,  "robot_setvel",             0, 2, robot_setvel},
  {4,  "robot_getpose",            1, 1, robot_getpose},
  {5,  "robot_readsonar",          1, 1, robot_readsonar},
  {6,  "robot_isleftmotorstalled", 1, 1, robot_isleftmotorstalled},
  ...
};
```

Most wrapper functions are relatively straight-forward like the *robot_getpose* function above, where the wrapper function converts the input parameters (prhs) from Matlab matrices into the format expected by the C++ method. Afterwards the ARIA C++ method is invoked and value returned by the method is afterwards converted back into a Matlab matrix.

```
static void
robot_getpose(int nlhs, mxArray *plhs[], int nrhs,
              const mxArray *prhs[])
{
    int rid = get_robot_id(prhs);
    ArRobot *robot = cnxTable[rid].robot;
    ArPose pose = robot->getPose();
    if (nlhs == 1) {
        plhs[0] = mxCreateDoubleMatrix(1, 3, mxREAL);
        *((double *)mxGetPr(plhs[0]))     = pose.getX();
        *(((double *)mxGetPr(plhs[0]))+1) = pose.getY();
        *(((double *)mxGetPr(plhs[0]))+2) = pose.getTh();
    }
}
```

3.4 Extending the Adapter Layer

The design with a central function-table makes it very easy to add support for new functions. The work required can be summarized into three steps:

1. Add an entry to the function-table. This table entry should contain a unique function identifier, the expected number of input and output parameters and a pointer to the C++ wrapper function. The entry can look something like this:

   ```
   {42,  "robot_mynewfunction", 1, 3, robot_mynewfunction},
   ```

2. Write a C++ wrapper function. This wrapper function is responsible for the translation of input and output parameters from and to Matlab matrices before and after the function invocation. Most wrapper functions are fairly similar so the implementation can probably be based on an existing wrapper function with a similar signature.

3. Add a new Matlab-file to the *@robot* directory. These files are short Matlab files that call the *aria_adapter* MEX-extension.

   ```
   function ret = mynewfunction(self, arg1, arg2)
   ret = aria_adapter(42, self.cnx, arg1, arg2);
   return
   ```

As long as the new function has a signature similar to some other existing function, all three steps can usually be completed simply by copying the required information from a similar existing function and modify it slightly.

3.5 Using the Adapter Layer

This section briefly describes how to use the adapter layer. See appendix A and B for more verbose instructions.

Before a program can communicate with a robot a TCP/IP connection has to be established between the ARIA library and the robot. When creating a new *robot* instance by calling the *robot* constructor with an ip-address as the only argument, a new connection is automatically established.

```
myrobot = robot('192.168.1.11');
```

If ARIA for some reason fail to connect to the robot, a Matlab exception will be raised. This error can be handled like this:

```
try
  myrobot = robot('192.168.1.11');
catch
  sprintf('Connection failed')
  return
end
```

The instance variable (myrobot) should be passed as the first argument to all class methods to let ARIA know which robot to send the commands to. For example to get the robot's pose the *getpose* class method is called like this:

```
pose = getpose(myrobot);
```

At the end of a program it is good practice to disconnect from the robot. This will stop the robot and disable the sonar sensors. The following line will disconnect a robot:

```
disconnect(myrobot);
```

3.6 A Sample Program

The Matlab code for a simple avoid obstacle behavior (as a Braitenberg vehicle [3]) may look like this:

```
myrobot = robot('192.168.100.11');

weights = [1 1.5 2 -2 -1.5 -1 0 0];

while ~isleftmotorstalled(myrobot) && ~isrightmotorstalled(myrobot)
  sonar = readsonar(myrobot) - 7500;
  dir = (weights * sonar') / 1000;
  setvel(myrobot, [70 70] + [-dir dir]);
end
disconnect(myrobot);
```

The Java-equivalent of the above Matlab program would look something like this:

```
public class Avoid {
    static {
        try {
            System.loadLibrary("AriaJava");
        } catch (UnsatisfiedLinkError e) {
          System.err.println("Native code library failed to load. Yada Yada" + e);
          System.exit(1);
        }
    }
    public static void main(String[] args) {
        double weights[] = {1, 1.5, 2, -2, -1.5, -1, 0, 0};
        Aria.init(0, true);

        ArRobot robot = new ArRobot("robot1", true, true, true);
        ArTcpConnection conn = new ArTcpConnection();

        conn.setPort("192.168.100.11", 8101);
        robot.setDeviceConnection(conn);
        if (!robot.blockingConnect())
        {
          System.err.println("Could not connect to robot, exiting.\n");
          System.exit(1);
        }
        robot.comInt((short)ArCommands.ENABLE, (short)1);
        robot.runAsync(true);
        while(robot.isLeftMotorStalled() != true &&
              robot.isRightMotorStalled() != true) {
            double dir = 0;
            for(int i = 0; i<8;i++) {
                dir += (robot.getSonarRange(i) - 7500) * weights[i];
            }
            dir /= 1000;
            robot.setVel2(70 - dir, 70 + dir);
        }
        robot.stopRunning(true);
        robot.disconnect();
        Aria.shutdown();
    }
}
```

3.7 Performance

The adapter layer introduces almost no overhead. The Matlab statement below is a crude benchmark that times how long it takes to call the wrapped function *sleep(0)* one million times. Running this benchmark on an AMD Duron 1.5Ghz system estimates the overhead added for one function call by the adapter layer to be less than 30μs. Most of this overhead is probably caused by Matlab's normal MEX-extension invocation overhead and not by the wrapper itself.

```
tic, for i=1:1000000, sleep(0), end, toc
```

3.8 Limitations

With the ARIA C++ library programmers can control the robot's behavior by defining a set of custom actions. A custom action is created by subclassing *ArAction* and overloading a few methods. These actions are run in a separate thread to allow the programmer to focus on other things.

Unfortunately as outlined in section 2.6 Matlab is a strictly single-threaded application and in combination with the unflexible class system makes it impossible to use or define custom actions from within Matlab. This does however not affect the possibility to control robots using the ARIA API's ordinary motion commands.

3.9 Future Work

This implementation has currently only been tested with Matlab version 7 running on Windows XP Professional. The adapter itself uses no platform dependent functionality so it should be possible to port it to other platforms supported by both ARIA and Matlab such as Linux or Mac OS X.

The adapter has so far only been tested with AmigoBot robots but should in theory work with other ARIA-based robots (see section 2.2) without requiring any modifications, except maybe wrapping a few more functions in order to support peripherals not available on the AmigoBot.

Another possible continuation of this work would be to add support for non ARIA-based robots like the Khepera robot from K-Team. The Khepera API is fairly similar to ARIA so it might be possible to develop an ARIA-compatibility layer on top of that.

Chapter 4

Map Making and Localization

This chapter covers the theory behind map making and localization.

A mobile robot exploring and mapping an unknown environment generate maps by accumulating sensor information while exploring the environment. Unfortunately readings from range sensors are only useful for the map making process if the position of the sensor at the time the reading took place is known. Shaft-encoders are the most commonly used source of odometry information. However it usually suffers from both systematic errors like inaccurate wheel measurements, and non-systematic errors like wheel slippage. These errors cause the odometry error to quickly accumulate over time. When a robot is not being localized in the environment, the quality and accuracy of the generated map will be degraded.

4.1 Sonar Sensors

Sonar sensors are one of the most commonly used sensor types for both commercial and research robots operating indoors. Sonar sensors are active sensors that emits a sound, and measures the time it takes for the sound to hit an object and bounce back. Given the *time-of-flight* and the speed of sound it is possible to calculate the distance to the object. An electical pulse causes the sonar transducer (thin metallic membrane) to generate a sound with an ultrasonic frequency. The sound wave created by the transducer is often considered to be 30° wide. At the same time as the sound wave is emitted, a timer is set. Afterwards the membrane becomes stationary and start to work like a microphone. If a strong enough *echo* is received, the timer is stopped and the measured time is considered to be the *time-of-flight*.

Sonar sensors suffers from a number of shortcomings and limitations. One problem is *specular reflection*, which is when the sound wave hits a surface at an acute angle and bounces away from the sensor instead of towards it. To make things worse, the sound can bounce off a second object and then find its way back to the sensor. In this case the *time-of-flight* will not correspond to true distance to the object. *Forshortening* is another problem that affects sonar sensors. The emitted sound wave is approximately 30° wide, this means that one side of the wave will hit a non-perpendicular surface before the other side. The reflection from that side of the wave will reach the sonar sensor before the reflection from the center of the wave. This causes the measured *time-of-flight* to be shorter, and the object to appear closer than it actually is [8].

4.2 Occupancy Grid

Occupancy grids[1] [6] are very commonly used in the field of robotics. An occupancy grid is a finite probabilistic representation of a robot's spatial knowledge. There are simple ways to update the grid with small amounts of sensor data collected from individual sensors and thereby creating an increasingly correct representation of the robot's surroundings. The final grid shows regions probably occupied, regions probably empty and unknown areas. Besides map making this representation is also used for activities such as exploration, localization, motion planning, landmark identification and obstacle avoidance [14].

An attractive feature of occupancy grids is that a single grid can incorporate information from multiple sensors of different types. A. C. Schultz and W. Adams demonstrates very accurate localization using a robot equiped with a set of 16 sonar sensors and a triangulation-based structured light range finder in [10].

4.3 Sonar Sensor Models

In order to be able to incorporate sensor information from different sensor types into a single occupancy grid, a common sensor information representation is required. A sensor model function is used to convert sensor type specific measurements to a common representation. The sensor model provides conditional probabilities for each location, given a certain sensor reading.

The AmigoBot robot used in this project is equipped with eight sonar sensors[2]. Most roboticists have converged on a model of sonar sensor uncertainty which looks like figure 4.1. The model can be divided into four different regions. The first three regions are inside the sonar sensor's field of view and the last one is outside [8].

- Region 1: Elements in this region are probably at least partially occupied.
- Region 2: Elements in this region are probably all empty.
- Region 3: The state of elements in this region is unknown because they are located behind whatever reflected the sonar pulse.
- Region 4: Nothing is known about elements outside the sonar sensors field of view.

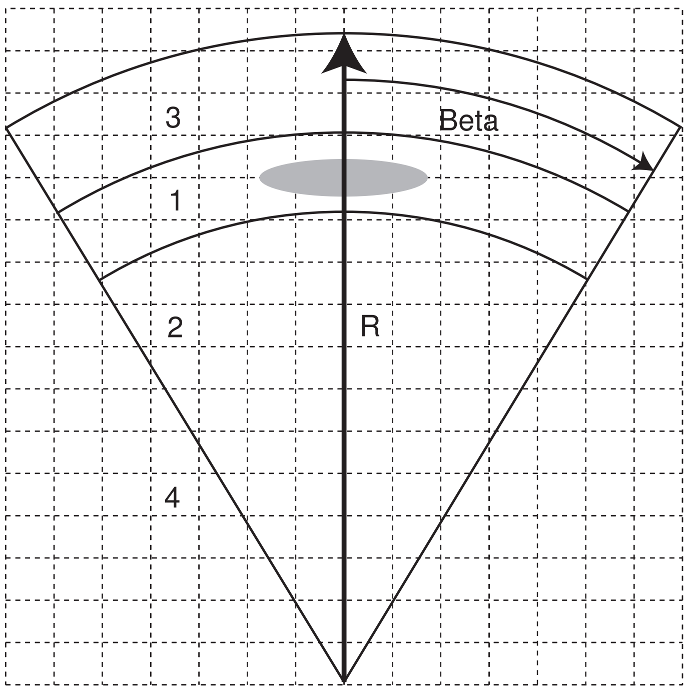


Figure 4.1: Grid elements are divided into four regions depending on their position relative to the sonar response. *R* is the sonars maximum range and β is half of the sensor cone angle width.

Given a range reading, elements in region 2 are more likely to be empty than elements in region 1. Readings are also more likely to be correct for elements close to the acoustic axis than elements at the edge of the sonar cone.

Sensor models can be generated in many different ways. Empirical models are based on interpretations of data collected from experiments. Analytical sensor models are based on an understanding of the underlying physical properties of the sensor device.

4.4 Updating occupancy grids

Many methods exist to convert sensor modes into numerical values and update occupancy grids. Bayesian sensor fusion is one of the most popular methods [7].

[1]Also known as Evidence grids, probability grids and certainty grids.
[2]Polaroid ultrasonic transducers

4.4.1 Bayesian sensor fusion

In the Bayesian approach sensor models convert sensor measurements into conditional probabilities of the form $P(s|H)$, i.e. the probability that a sensor reading s will occur, given that a certain grid element is occupied (hypothesis H).

The sonar sensor model used in this project uses the following functions to calculate the probability for each grid element located at a distance r and an angle α for the acustic axis of the sonar [8, 7]:

For elements in region 1:

$$P(\text{Occupied}) = \frac{\left(\frac{R-r}{R}\right) + \left(\frac{\beta-\alpha}{\beta}\right)}{2} \cdot \text{Max}_{\text{occupied}} \quad (4.1)$$

$$P(\text{Empty}) = 1.0 - P(\text{Occupied}) \quad (4.2)$$

For elements in region 2:

$$P(\text{Occupied}) = 1.0 - P(\text{Empty}) \quad (4.3)$$

$$P(\text{Empty}) = \frac{\left(\frac{R-r}{R}\right) + \left(\frac{\beta-\alpha}{\beta}\right)}{2} \quad (4.4)$$

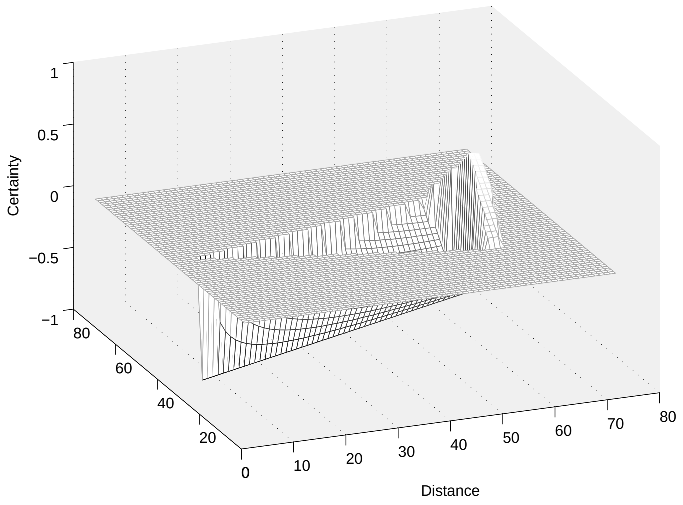


Figure 4.2: A 3D-representation of the Bayesian sonar sensor model. Notice that the certainty values are higher close to the sensor position and close to the acoustic axis.

The variables in the formulas have the following meaning: R represents the maximum range the sonar sensor can detect, β is half of the sensor cone angle width.

The effect of the $(\frac{R-r}{R})$ and the $(\frac{\beta-\alpha}{\beta})$ parts of the formulas is that elements close to the acoustic axis and close to the origin of the sonar pulse get higher probabilities than elements farther away.

These probabilities are projected onto an occupancy grid at the position and direction of the sensor, and are merged with the grid's existing spatial knowledge using Bayesian sensor fusion.

The sensor model provides conditional probabilities of the form $P(s|H)$ but the form $P(H|s)$, the probability that a grid cell is occupied given sensor reading s, is of more interest when updating an occupancy grid. Baye's rule states that:

$$P(H|s) = \frac{P(s|H)P(H)}{P(s|H)P(H) + P(s|\neg H)P(\neg H)} \tag{4.5}$$

With our sensor model notation it becomes:

$$P(\text{Occupied}|s) = \frac{P(s|\text{Occupied})P(\text{Occupied})}{P(s|\text{Occupied})P(\text{Occupied}) + P(s|\text{Empty})P(\text{Empty})} \tag{4.6}$$

Where $P(\text{Occupied})$ and $P(\text{Empty})$ probabilities represent the initial unconditional belief about the state of a grid element. $P(\text{Occupied}) = P(\text{Empty}) = 0.5$ is often used for empty occupancy grids, but could be set to some other values if the information is available for the environment being mapped.

Using equation 4.5 it is possible to populate an occupancy grid with probability values from a single sensor reading. With multiple sensor readings the equation becomes:

$$P(H|s_1, s_2, \ldots s_n) = \frac{P(s_1, s_2, \ldots s_n|H)P(H)}{P(s_1, s_2, \ldots s_n|H)P(H) + P(s_1, s_2, \ldots s_n|\neg H)P(\neg H)} \tag{4.7}$$

Equation 4.7 has one software implementation problem. It requires the program to keep track of all sensor readings while updating an element to be able to update it in the future. This is problematic because it is not known how many times each grid element will be updated. Fortunately, clever use of the relation $P(H|s)P(s) = P(s|H)P(H)$, a recursive version of equation 4.7 can be created:

$$P(H|s_n) = \frac{P(s_n|H)P(H|s_{n-1})}{P(s_n|H)P(H|s_{n-1}) + P(s_n|\neg H)P(\neg H|s_{n-1})} \tag{4.8}$$

With this equation, only the previous grid value $P(H|s_{n-1})$ needs to be saved in order to be able to update the element in the future.

4.5 Localization

For a mobile robot to be able to carry out most tasks, it needs to be able to figure out its position in the environment. Localization is the process of updating the pose[3]

[3]Position and heading (x, y, θ)

using sensor readings. J. Gutmann et al. divided the localization techniques into three different categories [5]:

- *Behavior-based*: behavior-based localization relies on the robot's actions in the environment in order to navigate. For example a *Follow-Right-Wall* behavior can be used to navigate in a maze. Simply inverting that behavior would allow the robot to find the way back to the start position.
- *Landmark-based (feature-based)*: Landmark-based localization works by identifying landmarks in the environment. Impressive geometrical localization can be achieved using landmark-based localization, the best known example is probably the GPS satellite system. One major drawback with landmark-based localization is the requirement of an adequate set of landmarks in order to work. Landmarks have to be known in advance (like the GPS satellites) or located while mapping the environment. The overall effectiveness of this method depends on the effectiveness of the landmark extraction method used.
- *Dense sensor matching (iconic)*: Unlike the landmark-based approach, dense sensor matching does not require the object recognition problem to be solved. Dense sensor matching uses whatever sensor information is available and attempt to match that information against the environments surface map, the occupancy grid.

There has been a lot of progress in the field of localization during the last few years. A lot of different techniques exist. The remaining sections of this chapter will cover the techniques used by the application described in chapter 5.

4.5.1 Odometry and Shaft Encoders

Most robots are at least equipped with some sort of shaft encoders. A shaft encoder is a proprioceptive sensor that measure the number of turns or fractions of turns a motor makes. Given that the gearing and wheel size is known it is possible to calculate the robots new position and heading using the shaft encoders.

Shaft encoders and odometry are however not very exact techniques. Errors can be divided into two categories, systematic and non-systematic errors. Systematic errors can come from inexact wheel size measurements or imperfections in the robot design such as placement or size of wheels. Non-systematic errors come from uneven floors, fast turning or different wheel slippage depending on surface type. This makes shaft encoders unfit as the primary localization system for most non-trivial robot systems.

4.6 Continuous Localization

Many localization methods try to model the odometry error or to re-localize only after an unacceptable odometry error has been detected. The *Continuous localiza-*

tion (CL) (see figure 4.3) technique proposed by Schultz et al. [10, 4, 14] takes advantage of the fact that positioning errors from shaft encoders usually accumulate over time. CL tries to perform small positioning corrections often instead of large corrections far apart in time.

Continuous localization is an exteroceptive method that builds temporary occupancy grids called short-term maps that contain recent sensor information. Given that these maps are constructed during a relatively short period of time, it is assumed that they contain relatively small amounts of odometry errors.

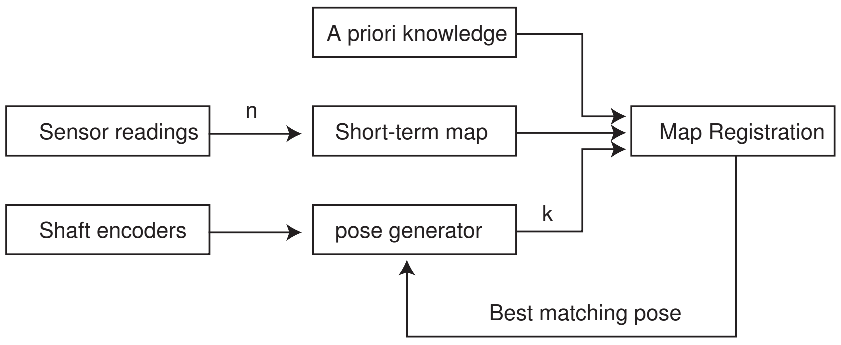


Figure 4.3: Continuous localization process

During each localization, *n* sensor sweeps are fused into each short term map. The variable *n* is a run-time parameter that depends on the environment and the equipment being used.

When a short-term map contains *enough* information, an operation called *map registration* takes place. The short-term map is matched against the robot's a priori knowledge about the environment, the long-term map. The overall effectiveness of continuous localization depends heavily on the completeness and correctness of the robot's a priori knowledge. Schultz et al. shows that the odometry error can be reduced to a constant error if the registration process has access to complete and correct spatial information [10]. Continuous localization is still effective even without any initial spatial information. An ordinary occupancy grid simultaneously updated using map making can also be used. This approach will of course not be as effective as with complete and correct spatial information, the odometry error, will slowly accumulate in the long-term map but not at all in the same rate as without using any localization.

The actual registration process is more complicated than it might appear. The fact that the short-term map is only a partial map makes it very likely that it will fit well in more than one place on the long-term map. In order to limit the risk of false matches the search is limited to *k* probable poses. The function *pose gen-*

erator generates a set of k possible poses given the robot's believed pose and the navigation performed during the lifespan of the short-term map.

For each of these k poses the match is performed by overlaying the short-term map centered over the pose k onto the global map. The match is then scored by calculating the sum of the difference between every element common to both maps using for example the following formula:

$$\sum |\mathrm{global}[i][j] - \mathrm{shortterm}[i][j]|$$

The robot's position is updated by selecting the best matching pose from the k poses generated by the *pose generator* function. If the CL process is combined with map making the short-term map is also merged with the long-term map at pose k.

Chapter 5

Full Scale SLAM-Application

This chapter summarizes the work done on the second part of this thesis project, and discusses obtained results and future work.

5.1 SLAM

In the second part of this master's thesis project a SLAM[1] application is developed. This application also serves as a test application for the Matlab adapter layer developed during the first part of this thesis project.

The SLAM problem tries to answer the following question: *"Is it possible for an autonomous vehicle starting at an unknown location in an unknown environment to incrementally build a map of the environment while simultaneously using this map to compute the vehicle's location?"* This is a very central problem in the robotics community. Solving it would allow autonomous vehicles placed in an unknown environment to build a map using only relative sensor observations, and then to use this map to navigate around the environment to perform different tasks. The main advantage of SLAM, is that no previous knowledge (apriori information) about the environment is required, everything will be accumulated at run time.

The SLAM-application created as a part of this thesis project combines techniques discussed in chapter 4 such as sensor models, occupancy grids, Bayesian sensor fusion and continuous localization. It is well known that shaft encoders are not very exact, so using the robot's built in shaft encoders as the only source of odometry information is not good enough to produce maps of reasonable quality. Continuous localization will be used to compensate for this problem. A number of tests will be performed, in order to determine how well the continuous localization technique used in this application, is able to compensate for the odometry error produced by the robot's internal shaft encoders.

[1]Simultaneous Localization And Mapping.

5.2 Design

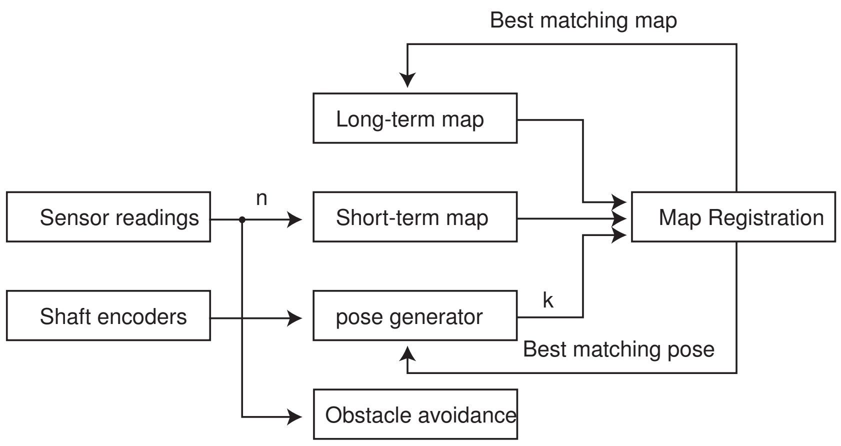


Figure 5.1: Block diagram illustrating how map making and continuous localization can be combined in a SLAM-application.

This application combines techniques such as sensor models, occupancy grids and continuous localization into a SLAM application. All of these techniques each require a set of properly configured parameters in order to work efficiently and produce good results (see section 5.8 for more information about these parameters). In order to make it possible to find a good set of parameter settings for this particular application and equipment the actual application is split into two parts.

The first part contains the data collection and navigation logic. The first part navigates around in the environment while simultaneously collecting sensor readings and robot poses (using the adapter's *getpose* and *readsonar* functions). The robot's current pose (position and heading) and readings from the robot's sonar sensors are recorded every 1/4 second and stored on the hard-drive. The second part of the application takes care of the actual map making and localization. The data collected in part one is used to build a map (called long-term map) of the environment, this map is simultaneously used by the continuous localization process to compute and correct the robot's real position.

The main benefit of separating data collection and localization, is that it both speeds up and makes it easier to identify good parameter settings. Without this division, the time consuming effort of letting the robot physically explore the environment has to be repeated for each set of parameter settings to evaluate. Besides the speed benefit, it also make the evaluation more exact, because the exact same set of sensor readings and poses can be used to evaluate many different parameter settings. Nothing in the actual design makes it impossible to fuse the two parts together into a single application when good parameter settings have been identified.

The block diagram in figure 5.1 illustrates how the application's different logical components are interconnected, and how the short-term map is fused with the long-term map after each map registration phase. The best matching pose is also used by pose generator to update the robot's position. The process will be described in more detail in section 5.6.

5.3 Equipment

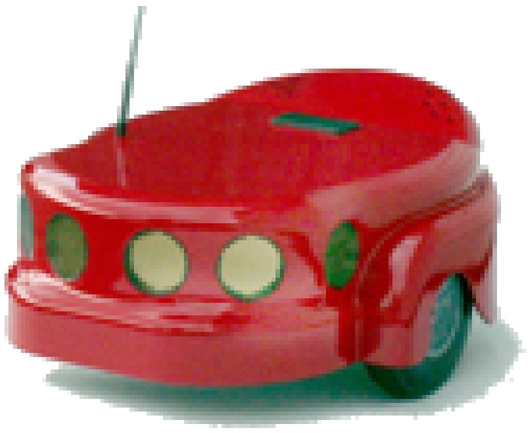


Figure 5.2: The AmigoBot robot.

An AmigoBot robot (figure 5.2) from ActivMedia Robotics Inc. was used to test this application. AmigoBot is an affordable classroom robot with good mobility in indoor environment. The robot has the following technical specification:

- Six front and two rear rangefinding sonars.
- Two 500-tick shaft encoders.
- Two wheels with individual motors and a rear caster.
- IEEE 802.11b wireless ethernet communication.

The application itself is implemented as a Matlab program, and tested on a PC running Microsoft Windows XP Professional, Matlab 7 and ARIA 2.1. The adapter layer developed as a part of this thesis project was used to communicate with the robot. Appendix C contains the full Matlab source code for both parts of the application.

5.4 Room Configurations

This application was tested in two different room configurations (see figure 5.3). Both rooms are open in the center with different hard-surfaced objects around the

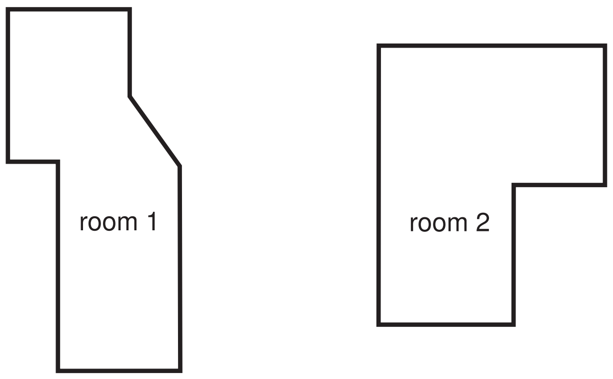


Figure 5.3: Room Configuration.

edges of the rooms creating a closed environment. The edges consists mainly of wood and metal surfaces but fortunately no glass surfaces, so the risk of specular reflection (see section 4.1) is somewhat limited. The main difference between the room configurations is that the first room consists of fairly narrow passages. The second room on the other hand contains a fairly large open area, section 5.7 explains what type of problems open areas can cause when it comes to continuous localization.

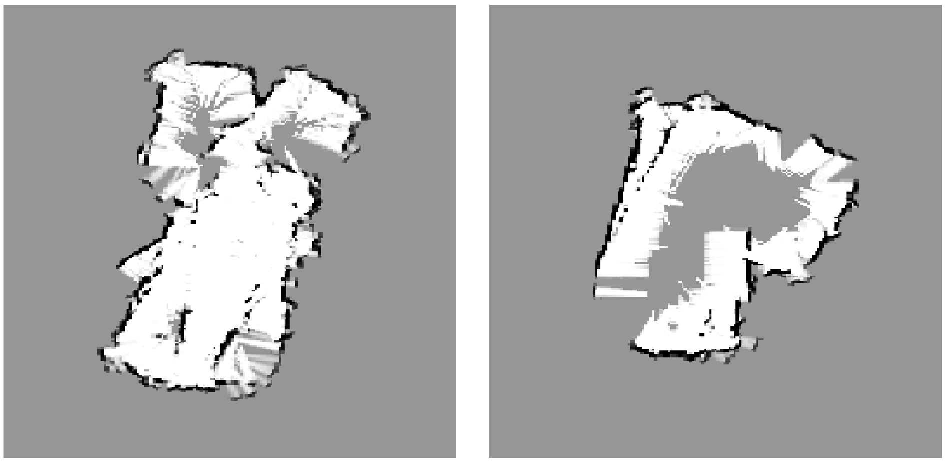
5.5 Navigation

The robot was programmed with an avoid-obstacle behavior. This behavior was implemented using one `ArActionConstantVelocity` and two `ArActionAvoidFront` ARIA actions. Simple avoid-obstacle behaviors are usually poor choices when it comes to efficient exploration of unknown environments. However this behavior fitted these rooms configuration quite well and was able to explore the entire room rather quickly without getting stuck in corners or visiting some regions a lot more frequently than others.

The application's two-part design imposes one limitation on the choice of navigation behavior. Because the navigation takes place in first part of the application but the map is generated by the second part, the behavior has to be purely reactive and can not require access to the long-term map.

5.6 SLAM And Continuous Localization

While building maps, it is very important for the robot to know its exact position in the environment. Otherwise, accumulating odometry error will soon start to distort the map being built. Figure 5.4 illustrates how generated maps for room one and two looks like when the built in shaft-encoders are used as the only source of odometry information, without any help from continuous localization.



Room 1. Room 2.

Figure 5.4: Map making without continuous localization.

The continuous localization technique described in [10] was initially intended to operate with access to complete apriori spatial knowledge about the environment. Experiments performed by Alan C. Schultz et al. showed that with access to complete spatial information about the environment, continuous localization is able to keep the odometry error at a constant level.

However the *apriori knowledge* requirement, is not compatible with the SLAM-problem. The SLAM-problem explicitly states that an autonomous vehicle *should* be able to build a map of an *unknown* environment. In order to comply with this requirement, the map registration process used in this application is altered to work like this:

The long-term map (the map being generated) is initially empty. Each localization starts with an empty short-term map. The short-term map is fused with sensor readings until it is considered mature (see the *short-term map maturity* parameter in section 5.8). The map registration process *registers* the short-term map by overlaying it on the long-term map for each of the *k* poses returned by the *pose generator*. The *map registration search space* parameter determines which poses the pose generator will return. The difference between the robot's believed pose and the best matching pose is considered to be the odometry error. After the robot's pose has been updated, the short-term map is fused with the long-term map and a new localization starts with a new and empty short-term map.

5.7 Open Areas

Large open areas are especially troublesome for continuous localization. In this context, open areas are defined as areas in the environment in which the robot's sonar sensors are unable to detect any obstacle, i.e. every obstacle in the environment is far enough away not to generate any sonar echo. At least two different open area related problems affects the effectiveness of continuous localization:

While a robot navigates through a large open area, the sonar sensors will be unable to get any range readings at all. If the area is large enough it is possible that the robot will not detect any obstacles at all during the lifespan of a single short-term map. A completely empty short-term map will fit equally well anywhere on the long-term map. Consequently, while navigating through large open areas, continuous localization will be unable to compensate for the odometry error introduced by the shaft encoders. However as long as the robot navigates through these areas in a straight line at a constant velocity the error will be quite limited and will not accumulate very fast.

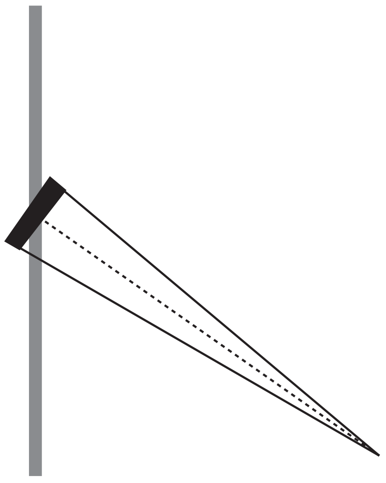


Figure 5.5: This figure illustrates how the sensor model for a single sonar reading from a wall may look like.

Completely empty short-term maps may temporarily disable continuous localization, but it is neither the only nor the worst open area related problem. A worst case scenario can look like this:

A robot navigates through an open area but one sonar sensor finally detects an approaching wall (see figure 5.5). That sensor sweep happens to be the last one for the current short-term map, so that map is registered and fused with the long-term map. During the lifespan of the next short-term map the robot gathers a number of range readings from the same wall in the newly created short-term map.

When this new short-term map is later registered, it is matched against the long-term map's current information about that area, which consists of only the information from the previous short-term map's single sensor reading. This is usually not enough information for the map registration process to perform an accurate match. A single sensor reading contains for example not enough information to determine the actual angle of the wall only the presense of an obstacle. This lack of information might result in an inaccurate registration of the short-term map and thus introducing additional odometry error into the system. This application tries to avoid this worst-case senario by making sure every short-term map contains enough information. This is done by measuring short-term map maturity by the number of actual (in range) sensor readings fused, and not by the map age in seconds. This will cause a single short-term map to be active longer if the robot navigates through an open area.

5.8 Parameters

This application contains a set of parameters that all control different aspects of the application. This section describes these parameters, their effect on the application and the values found to produce good results.

Map registration search space: The registration process can not match the short-term map with the long-term map for every possible robot pose. This would be both too computationally expensive and error prone. The *pose generator* function is used to limit the search to k probable poses. It is not easy to calculate the *k most* probable poses the robot could have, the robot could have different odometry errors in all three degrees of freedom (x, y, θ) since the last localization was made. An ideal *pose generator* function should probably consider aspects such as current position, current speed and the movements performed since the last localization when generating the list of probable poses.

The *pose generator* function used in this application is fairly simple but efficient. The function assumes the accumulated odometry error since the last localization is in the range of ± 1 grid elements in the xy-plane. The error on the θ axis is assumed to be in the range of $\pm 3^\circ$. With a resolution of 1 grid element and 1° this yields a total of 63 poses for the registration process to evaluate during each localization.

Short-term map maturity: The short-term map maturity determines how much information (sensor readings) to store in each short-term map. The short-term map maturity can either be measured in age, i.e. the number of seconds readings are fused into a map before it is considered to be mature, or in the actual number of (in range) sensor readings fused. Fusing 45 (in-range) sensor readings into each short-term map was found to be a good setting for this application.

Grid Resolution: The short-term map and the long-term map both uses the same grid resolution. This resolution determines how fine-grained the information stored in the maps will be. The higher resolution used the more computationally expensive the application will become. A grid resolution of 35mm x 35mm was used in these experiments.

Sensor Sweep frequency and vehicle velocity: The sensor sweep frequency and the vehicle velocity determines the density of the gathered information. During these experiments the robot traveled with an average velocity of 80 mm/s and collected four full sensor sweeps every second.

5.9 Results

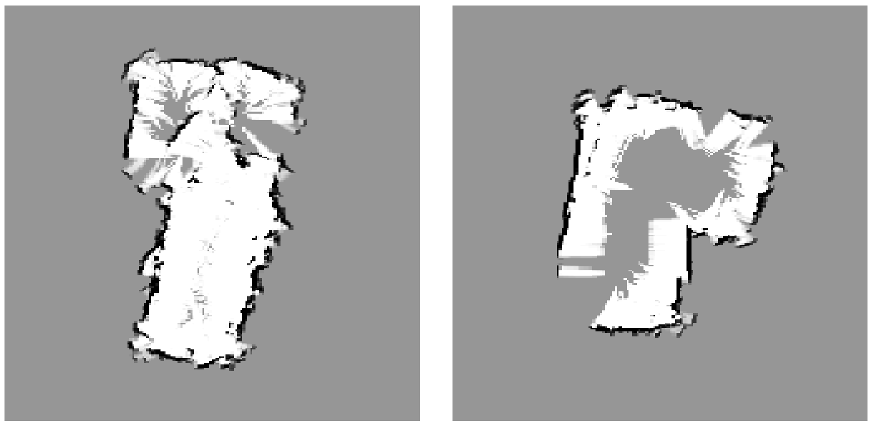
The continuous localization process relies heavily on the accuracy of the long-term map. However the quality of the long-term map simultaneously depends on the accuracy of the continuous localization process. The parameters described in section 5.8 all control the quality and accuracy of the resulting map, in one way or another. Besides these parameters, the room configuration and the robot, all affect the end result. These complex dependencies makes it difficult, or even impossible to find the optimal parameter settings to use. Figure 5.6 shows the resulting map, using three different map registration search spaces. One conclusion that can be made from these maps is, that using a larger search space is not only more computationally expensive, but can also generate worse results. Using a too large search space can result in incorrect registration of short-term maps, which results in extra odometry error being added to the map and robot's pose. Similarly, using a too small search space will stop the continuous localization process from being able to remove the odometry error introduced by the robot's shaft encoders. With the equipment and room configurations used in this thesis a search space of ± 1 grid element and $\pm 3^\circ$ gave the best results.

5.10 Limitations

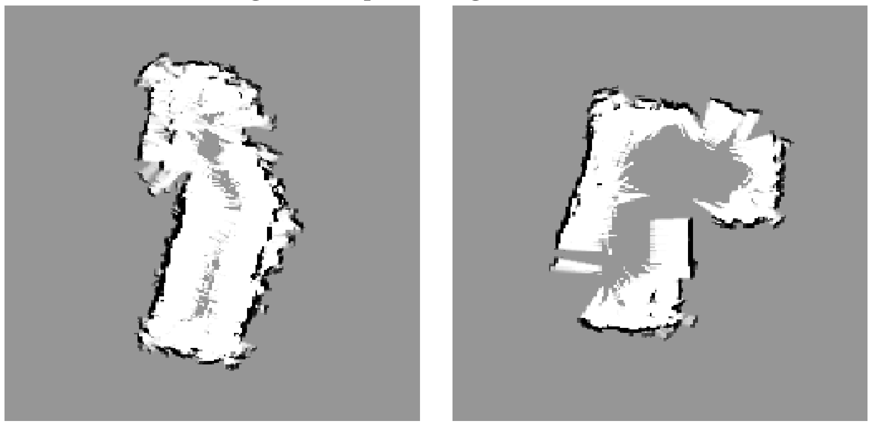
As mentioned before, continuous localization tries to correct the robot's position by matching the short-term map with the long-term map. The accuracy of this matching process is crucial to the overall effectiveness. The method works well as long as the short-term map contains sensor readings from at least one object in the environment, that is also in the long-term map. This is usually the case if the robot is equipped with enough sonar sensors and the environment does not contain large open spaces.

When a robot navigates through large open spaces, sonar sensors will be unable to get any readings because all obstacles are out of range and as a result the short-term map will become empty. The map registration process will be unable to estimate the odometry error because the short-term map will be overlayed over

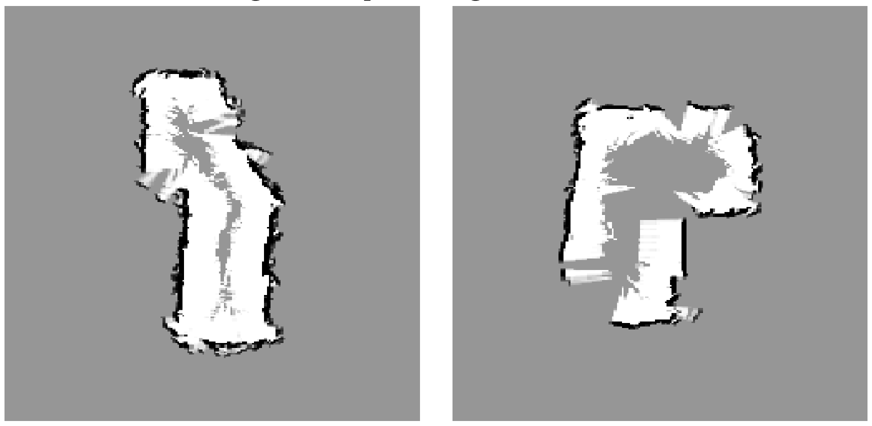
Room 1 Room 2



Using search space ±1 grid element, ±1°



Using search space ±1 grid element, ±6°



Using search space ±1 grid element, ±3°

Figure 5.6: Resulting maps using different search spaces.

an equally empty part of the long-term map and no correction can be performed. The sonar sensors on the AmigoBot robot can only detect obstacles less than 800 mm away. This means that this approach is most effective in environment where the robot most of the time will be less than 800 mm away from obstacles.

5.11 Future Work

The overall map precision and quality could probably be improved by using optimized sensor models. Joris W.M. van Dam et al. presents a way to calibrate sensor models using neural networks. This approach makes the sensor model adaptive to changes both in the environment and the characteristics of the sensors [12].

Simple exploration behaviors such as avoid-obstacle or follow-wall might work adequately in simple room configurations like the one used in these experiments. In more complex environments such as large office environments a more efficient exploration method must be used. Brian Yamauchi introduced a frontier-based approach for autonomous exploration [13]. This approach uses the concept of frontiers which is the region between unexplored areas and unoccupied areas. By navigating to a frontier more information is added to the long-term map and the frontier is consequently further pushed back. This type of frontier-based exploration is much more efficient than the avoid-obstacle behavior currently used.

Chapter 6

Summary and Conclusion

This chapter summarizes the work and the conclusions that can be drawn from the two parts this thesis project consists of.

6.1 Summary and Conclusions

This thesis presented an adapter layer between ARIA and Matlab that successfully hides some of the complexity imposed by the ARIA library itself and the C++ language. The API is modeled after ARIA's original C++ api but due to the difference in features between the C++ and Matlab languages the exposed API is not complete. Some methods could not be wrapped due to various limitations imposed mainly by Matlab. Other methods were wrapped but with slightly different signatures to make them look and feel more Matlab-like.

One conclusion that can be made is that even though ARIA is a large and complex library and C++ and Matlab are two very different programming languages, it is possible to create an at least partial adapter layer. When using programming languages with a feature set more compatible with C++ like Java or Python[1] it is possible to auto-generate most, if not all of the adapter layer using tools such as swig[2]. Unfortunately no such tool exists for Matlab so every wrapper function in this adapter layer is hand-crafted. Fortunately the design used makes it very simple and obvious where and how to make modifications in order to extend the adapter layer by wrapping new functions.

This thesis also presented a full-scale SLAM-application. This application explored important SLAM techniques such as occupancy grids, bayesian sensor fusion and continuous localization. Experiments using this applications showed how quickly odometry error accumulates and distorts the map being generated. Additional experiments demonstrated that continuous localization can be used to limit or even completely remove the odometry error introduced by shaft encoders.

[1]http://www.python.org/
[2]http://www.swig.org

One conclusion that can be made is that continuous localization can be very useful and vastly improve the quality of generated maps. The technique is effective both with and without any previous knowledge about the room configuration.

Chapter 7

Acknowledgementes

I wish to thank my supervisor Thomas Hellström at the Department of Computing Science, Umeå University for his support during this project. I would also like to thank Thomas Nyberg for proof-reading this thesis.

References

[1] ACTIVMEDIA. Aria reference manual. pdf-document, 26 Sept. 2004. `http://www.cis.ysu.edu/~john/robotics/ARIA/Aria-Reference.pdf`.

[2] ACTIVMEDIA. Robot specifications. web-site, 3 Feb. 2005. `http://www.activrobots.com/ROBOTS/specs.html` .

[3] BRAITENBERG, V. *Vehicle: Experiments in Synthetic Psychology*. Massachusetts Institute of Technology, 1984.

[4] GRAVES, K., ADAMS, W., AND SCHULTZ, A. Continuous localization in changing environments, 1997.

[5] GUTMANN, J., BURGARD, W., FOX, D., AND KONOLIGE, K. An experimental comparison of localization methods. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (1998).

[6] MORAVEC, H., AND ELFES, A. E. High resolution maps from wide angle sonar. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation* (March 1985), pp. 116–121.

[7] MORAVEC, H. P. Sensor fusion in certainty grids for mobile robots. In *Proceedings of the 1988 AI Magazine Volume 9* (1988), pp. 61–74.

[8] MURPHY, R. R. *Introduction to AI robotics*. Massachusetts Institute of Technology, Cambridge, Massachusetts, 2000.

[9] PÄRT-ENANDER, E., AND SJÖBERG, A. *Användarhandledning för Matlab*. Uppsala universitet, Uppsala, Sweden, 2003.

[10] SCHULTZ, A. C., AND ADAMS, W. Continuous localization using evidence grids. pp. 2833–2839.

[11] THE MATHWORKS, I. External interfaces reference, 10 Feb. 2005. `http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiref.pdf`.

[12] VAN DAM, J. W. M., KRÖSE, B. J. A., AND GROEN, F. C. A. Adaptive sensor models. In *1996 IEEE/SICE/RSJ Intr. Conf. on Multisensor Fusion*

and Integration for Intelligent Systems, Washington D.C (Dec. 8–11, 1996), pp. 705–712.

[13] YAMAUCHI, B. A frontier-based approach for autonomous exploration. In *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation* (1997), pp. 146–151.

[14] YAMAUCHI, B., SCHULTZ, A. C., AND ADAMS, W. Mobile robot exploration and map-building with continuous localization. pp. 3715–3720.

Appendix A

Programming Tutorial

The purpose of this ARIA adapter is to provide a simple yet powerful API for Matlab users. Only a few lines of code is required to create a simple robot application.

This tutorial covers topics such as installing the software on your computer, example programs and troubleshooting. By the time you reach the end of this tutorial you will be able to install ARIA for Matlab adapter layer on your computer and write simple Matlab programs that interacts with your ARIA-based robot.

Installation

The installation is quite straight forward and consists of only two steps:

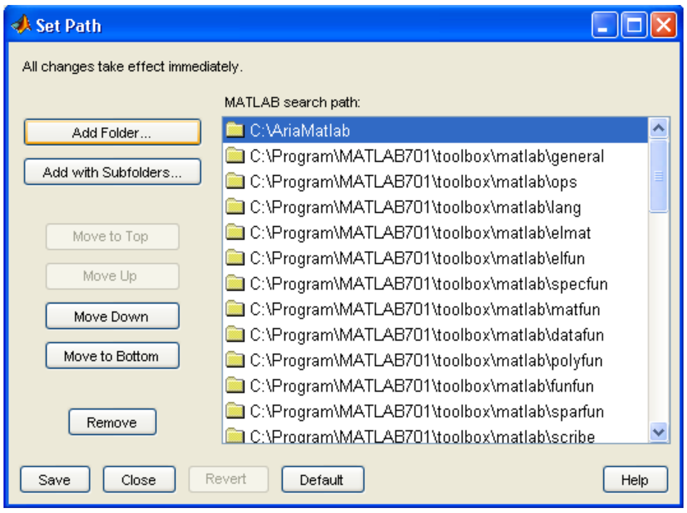
1. Download and install the ARIA distribution from the ActivMedia web-site. This distribution contains some data files required by the adapter layer. The installation program will also install the file necessary to use ARIA from C++ And Java.

2. Extract the zip-archive containing the compiled *Aria For Matlab* files to somewhere on your hard-drive, for example to `C:\AriaMatlab\`

3. In order for this extension to work Matlab must be able to find the `@robot`-class directory in the Matlab search path. A new directory can be added to the search path using the *Set Path* dialog (see fig A.1) accessible from the *File*-menu. Another option is to use the `addpath` command like this:

   ```
   addpath c:\AriaMatlab\
   ```

   Note: The addpath command only affects the currently running Matlab instance and has to be re-run every time Matlab is restarted.



All changes take effect immediately.

MATLAB search path:

```
C:\AriaMatlab
C:\Program\MATLAB701\toolbox\matlab\general
C:\Program\MATLAB701\toolbox\matlab\ops
C:\Program\MATLAB701\toolbox\matlab\lang
C:\Program\MATLAB701\toolbox\matlab\elmat
C:\Program\MATLAB701\toolbox\matlab\elfun
C:\Program\MATLAB701\toolbox\matlab\specfun
C:\Program\MATLAB701\toolbox\matlab\matfun
C:\Program\MATLAB701\toolbox\matlab\datafun
C:\Program\MATLAB701\toolbox\matlab\polyfun
C:\Program\MATLAB701\toolbox\matlab\funfun
C:\Program\MATLAB701\toolbox\matlab\sparfun
C:\Program\MATLAB701\toolbox\matlab\scribe
```

Figure A.1: The *Set Path* dialog can be used to tell Matlab where to find the *@robot* directory.

Connecting

ARIA communicates with the robots using TCP/IP connections. Each robot is identified by an ip-address. It is generally a good idea to verify that the robot's ip-address is reachable from the computer running Matlab before trying to use the adapter layer.

```
ping <your-robots-ip-address>
```

If no ping reply is received from the robot when running the above command in the command prompt please verify your network setup before continuing with this tutorial.

Before a program can communicate with a robot a TCP/IP connection has to be established between the ARIA library and the robot. When creating a new *robot* instance by calling the *robot* constructor with an ip-address as the only argument, a new connection is automatically established.

```
myrobot = robot('192.168.1.11');
```

This instance should be passed as the first argument to all class methods to let ARIA know which robot to send the commands to. At the end of a program it is good practice to disconnect from the robot. This will stop the robot and disable the sonar sensors. The following line will disconnect a robot:

```
disconnect(myrobot);
```

Coordinate System

While a robot navigates in the environment, ARIA always keeps track of the robot's pose using the robot's built in shaft encoders. The robot's believed pose can always be accessed and updated using the *getpose* and *move* methods.

The robot's pose is expressed as a point on the xy-plane and an angle θ. When the robot powers up the initial pose is set to $x = 0$, $y = 0$, $\theta = 0$, and the robot is looking along the positive x-axis. If the robot rotates +90 degrees (counter clock wise) it will be looking along the positive y-axis. The x, y position is expressed in unit of millimeters and the θ angle in the interval of +-180 degrees counter clock wise.

Examples

This section contains three simple Matlab programs that are using the ARIA adapter API. Please see appendix B for a complete API documentation.
Example 1 demonstrates how easy it is to implement a simple Braitenberg vehicle. This avoid obstacle behavior will be repeated until the robot collides with some object and at least one wheel gets stuck.

```
weights = [1 1.5 2 -2 -1.5 -1 0 0];
myrobot = robot('192.168.1.11');

while ~isleftmotorstalled(myrobot) && ~isrightmotorstalled(myrobot)
  sonar = readsonar(myrobot) - 7500;
  dir = (weights * sonar') / 1000;
  setvel(myrobot, [70 70] + [-dir dir]);
end
disconnect(myrobot);
```

Example 2 demonstrates how to use the `move` and `set(delta)heading` commands. This program will move the robot 0.5 meters forward, stop and turn 180 degrees. This behavior will be repeated 10 times before disconnecting from the robot.

```
myrobot = robot('192.168.1.11');

for i=1:10
  % Move 0.5 meters forward
  move(myrobot, 500);
  % Wait until the robot has reached the target
  while ~ismovedone(myrobot)
    sleep(100);
  end
  % Rotate 180 degrees clock-wise
  setdeltaheading(myrobot, 180);
  % Wait until the have rotated 180 degrees
  while ~isheadingdone(myrobot)
    sleep(100);
  end
end
disconnect(myrobot);
```

Example 3 demonstrates how to use the `getpose` method to fetch the robot's believed pose. This program will put the robot into *avoid-obstacle* mode and plots the path the robot travels using basic Matlab commands.

```
r = robot('192.168.1.11');
x = [];
y = [];
figure(1);
% Tell the robot to avoid obstacles
avoid(r, 80);
% Loop until we get stuck
while ~isleftmotorstalled(r) && ~isrightmotorstalled(r)
  % Save our position in two vectors
  pose = getpose(r);
  x = [x pose(1)];
  y = [y pose(2)];
  % Plot our path
  plot(x, y);
  drawnow;
  % Sleep 0.1 seconds
  sleep(100);
end
disconnect(r);
```

Troubleshooting

- *When I try to create a robot instance I get the following error message: "Undefined command/function 'robot"'.*

  Matlab is unable to find the *@robot* directory in the search path. Please make sure the directory is added using either the *Set Path* dialog or the *addpath* command.

- *Every time I try to create a robot instance I get the following error message: "connect failed".*

  ARIA failed to reach the robot. Please verify that the correct ip-address is used and that the robot responds to *ping*.

- *When I call a class method I get the following error message: "invalid robot id".*

  This happens when calling class methods *after* disconnecting from the robot. Calling class methods after disconnecting from a robot is not allowed.

- *Is there a way to make sure the robot always stops if some unexpected error happens in my program?*

  Yes, one way is to use a `try, catch` statement to make sure the disconnect() command is always executed:

  ```
  function myprogram()
  myrobot = robot('192.168.1.11')
  try
    myrealprogram(myrobot)
  catch
    lasterr
  end
  disconnect(myrobot)

  function myrealprogram(myrobot)
  <Put your real program here>
  return
  ```

Appendix B

API Documentation

The API is basically a subset of the full Aria C++ API that has been slightly modified to get a more Matlab-like look and feel. Among the changes are the use of vectors and matrices instead of helper classes like ArPose where feasible.

Robot Class Constructor

A *robot* class instance represents a connection to a robot's. A single Matlab program can connect to and control multiple robots simultaneously by creating one *robot* instance for each robot.

A robot instance is created like this:

```
myrobot = robot('192.168.1.11')
```

The string argument to the *robot* constructor is the robot's network address. It can be a hostname or an numerical ip-address. The constructor returns a new *robot* instance on success. On failure an exception is raised.

Robot Class Methods

The OOP-system in Matlab uses a for Java and C++ programmers awkward syntax. The following Java/C++ lines

```
instance.method(arg1, arg2, ...);
```

looks like this in Matlab:

```
method(instance, arg1, arg2, ...);
```

The methods listed in the rest of this section are *robot* class-methods:

name:	**disconnect**
arguments:	Nothing
returns:	Nothing
description:	Disconnects from the robot associated with the *robot* instance. This function returns resources associated with this instance back to the operating system. The instance should not be used after this.

name:	**setvel**
arguments:	$[x, y, \theta]$ (1x3 matrix)
returns:	Nothing
description:	Sets the velocity for the wheels independently.

name:	**getpose**
arguments:	Nothing
returns:	$[x, y, \theta]$ (1x3 matrix)
description:	Returns the robot's current pose (position and heading).

name:	**readsonar**
arguments:	Nothing
returns:	1xn matrix (mm)
description:	Returns range readings from all of the robot's *n* sonar sensors. The readings are returned as a 1xn matrix. Missing values are indicated with the value 6842.

name:	**isleftmotorstalled**
arguments:	Nothing
returns:	1 if stalled else 0
description:	Returns the state of the left motor. A return value of 1 indicates that the left motor is unable to rotate the left wheel, this is usually indicates that the robot is stuck due to a collision.

name:	**isrightmotorstalled**
arguments:	Nothing
returns:	1 if stalled else 0
description:	Returns the state of the right motor. A return value of 1 indicates that the right motor is unable to rotate the right wheel, this is usually indicates that the robot is stuck due to a collision.

name:	**lock**
arguments:	Nothing
returns:	Nothing
description:	Locks the robot instance. As long as the robot instance is locked, the ARIA background thread will not update values such as *pose* or sonar readings. This function can be used to collect readings from the robot in a consistent manner.

name:	**unlock**
arguments:	Nothing
returns:	Nothing
description:	Unlocks the robot instance. See the *lock* method.

name:	**moveto**
arguments:	$[x, y, \theta]$ (1x3 matrix)
returns:	Nothing
description:	Updates the robot's idea of its position. Calling this method will not physically move the robot only update the robot's believed position.

name:	**cleardirectmotion**
arguments:	Nothing
returns:	Nothing
description:	A direct motion command (*setvel*, *move*, *setheading* ...) overrides ARIA actions. This method clears whatever direct motion command that has been gives so actions work again.

name:	**stop**
arguments:	Nothing
returns:	Nothing
description:	Stops the robot by setting the wheel and rotational velocity to 0.

name:	**move**
arguments:	distance (mm)
returns:	Nothing
description:	Moves the robot *distance* mm forward or backward. The *ismovedone* method can be used to determine when the move is completed.

name:	**ismovedone**
arguments:	Nothing
returns:	1 if no *move* command is running else 0
description:	Determines if any *move* command is running.

name:	**setheading**
arguments:	heading in degrees
returns:	Nothing
description:	Rotates the robot to the given heading angle. The *isheadingdone* method can be used to determine when the rotation is completed.

name:	**setdeltaheading**
arguments:	delta heading in degrees (relative to the robot's current heading)
returns:	Nothing
description:	Rotates the robot to the given heading angle. The *isheadingdone* method can be used to determine when the rotation is completed.
name:	**isheadingdone**
arguments:	Nothing
returns:	1 if no *set(delta)heading* command is running else 0
description:	Determines if any *set(delta)heading* command is running.
name:	**setrotvel**
arguments:	velocity (deg/s)
returns:	Nothing
description:	Tells the robot to rotate at a certain speed.
name:	**absolutemaxtransvel**
arguments:	velocity (mm/s)
returns:	Nothing
description:	Sets the robot's absolute maximum translational velocity. This serves as an upper limit for the *rotvelmax* method.
name:	**absolutemaxrotvel**
arguments:	velocity (deg/s)
returns:	Nothing
description:	Sets the robot's absolute maximum rotational velocity. This servers as an upper limit for the *transvelmax* method.
name:	**rotvelmax**
arguments:	velocity (deg/s)
returns:	Nothing
description:	Sets the maximum rotational velocity. This method controls how fast motion commands such as *setheading* and *setdeltaheading* will rotate.
name:	**transvelmax**
arguments:	velocity (mm/s)
returns:	Nothing
description:	Sets the maximum translational velocity. This method controls how fast the robot will move while performing *move* commands.
name:	**avoid**
arguments:	velocity (mm/s)
returns:	Nothing
description:	Starts a built in avoid obstacle behavior. This behavior is implemented using one `ArActionConstantVelocity` and two `ArActionAvoidFront` ARIA actions.

Utility Functions

name:	**sleep**
arguments:	time (μs)
returns:	Nothing
description:	Sleep *time* microseconds using the *ArUtils::sleep()* function.

Appendix C

SLAM Source Code

part1.m

```
function save_readings()
r = robot('192.168.100.11');
try
  setabsolutemaxrotvel(r, 40);
  avoid(r, 80);
  save_map(r, 30 .* 16);
catch
  lasterr
  setvel(r, [0, 0]);
  disconnect(r);
  return;
end
stop(r);
disconnect(r);

function save_map(r, num_readings)
for i = 1:num_readings;
  sleep(250);
  lock(r);
  saved_pose(i,:) = getpose(r);
  saved_readings(i,:) = readsonar(r);
  unlock(r);
end
save state12 saved*;
```

part2.m

```
SCALE = 35;
world_size = 300;
world = zeros(world_size, world_size);
world(:,:)= 0.5; % 0.5 = unknown
overlap = 0;
degrees = [0 -1 1 -2 2 -3 3];

% Load sensor readings and robot poses from disk
load state10 saved*;
readings = saved_readings;
pose = saved_pose;

off = 1;
while off < length(readings) - overlap
  stm_size = 0;
  num = 0;
  % Determine how many sensor sweeps required to find
  % at least 45 in-range readings
  while num < 45 && off+stm_size < length(readings)
    num = num + sum(readings(off+stm_size,:) < 6000);
    stm_size = stm_size + 1;
  end
  rpos = [];
  rstm = [];
  rposes = [];
  rscore = [];
  rdegrees = [];
  % Generate LENGTH(DEGREES) different versions of the stm. All rotated
  % differently to allow us to correct odometry errors on the theta axis
  tic;
  for k = 1:length(degrees)
    % Start with an empty short time map
    stm = zeros(world_size, world_size);
    % Everything is unknown
    stm(:,:) = 0.5;
    for j = 1:stm_size
      pos = ceil((pose(off+j,:) ./ SCALE) + (world_size ./ 2) - 40);
      % Generate a sensor model
      model = sonar_model(readings(off+j,:), pose(off+j,3)+degrees(k));
      % Add it to the short term map
      stm = bayes(stm, model, pos);
    end
    % We are only interested in a smaller square of the stm with the first
    % robot pose in the center.
    margin = 15;
    pos = ceil((pose(off+1,:) ./ SCALE) + (world_size ./ 2) - 40 - margin);
    stm = stm(pos(1)+1:pos(1)+length(model)+2*margin, ...
              pos(2)+1:pos(2)+length(model)+2*margin);

    % The POSSIBLE_POSES() function returns a matrix of possible robot
    % poses (relative to the current position).
    % We use the CALC_SCORE utility function to find out in which of these
    % poses the stm fits best.
    poses = possible_poses(pose(off+1,:));
    for l=1:size(poses,1)
      pos2 = pos(1:2) + poses(l,:);
      rscore = [rscore calc_score(world, stm, pos2)];
      rpos = [rpos; pos2];
      rstm(:,:,length(rscore)) = stm;
      rposes = [rposes ; poses(l,:)];
      rdegrees = [rdegrees degrees(k)];
    end
  end
  toc;
  % The MIN() functions returns the most probable location.
  [score, idx] = min(rscore);
  stm = rstm(:,:,idx);
  pos = rpos(idx,:);

  % Register the short time map in the global map
  world = bayes(world, stm, pos);

  left = off+stm_size-overlap:length(readings);
  a = -rdegrees(idx) / 180 * pi; T = [cos(a) -sin(a); sin(a)  cos(a)];
  pos = [pose(left,1)-pose(off+1,1) pose(left,2)-pose(off+1,2)];
  pos = pos * T;
  pose(left,1:2) = [pos(:,1)+pose(off+1,1) pos(:,2)+pose(off+1,2)];
  pose(left,1) = pose(left,1) + (SCALE .* rposes(idx,1));
  pose(left,2) = pose(left,2) + (SCALE .* rposes(idx,2));
  pose(left,3) = pose(left,3) + rdegrees(idx);

  % Display the short-time-map and the long-term map in two different figures
  figure(1);
  colormap(gray);
  imagesc(stm, [0 1]);
  figure(2);
  colormap(gray);
  imagesc(world, [0 1]);
  drawnow;
  off = off + stm_size - overlap;
end
% Display a clearer map using some crude image manipulation
figure(3);
colormap(gray);
world(find(world < 0.7)) = 0;
world(find(world >= 0.7)) = 1;
imagesc(world, [0 1]);

function poses = possible_poses(pose)
%poses = [0 0]; return;
poses = [0 0; -1 -1; 1 1; 1 -1; -1 1;
         0 -1; 0 1; -1 0; 1 0];
return

function score = calc_score(world, model, offset)
world = world(offset(1)+1:offset(1)+length(model), ...
              offset(2)+1:offset(2)+length(model));
if length(find((world ~= 0.5) .* (model ~= 0.5))) == 0
  score = 0;
  return
end
score = sum(sum(abs(world - model)));

function world = bayes(world, model, offset)
prev = world(offset(1)+1:offset(1)+length(model),
             offset(2)+1:offset(2)+length(model));
idx = find(~isnan(model));
new = prev;
new(idx) = (model(idx) .* prev(idx)) ./ (model(idx) .* ...
            prev(idx) + (1 - model(idx)) .* (1 - prev(idx)));
world(offset(1)+1:offset(1)+length(model),
      offset(2)+1:offset(2)+length(model)) = new;
```

```
function model = sonar_model(ranges, theta)
B = 7;
R = 26;
SCALE = 35;
TOLERANCE = 1;
MAXOCCUPIED = 0.98;
model_size = 59;
model = zeros(model_size, model_size);
model(:,:)= NaN;
a = -theta/180*pi; T = [cos(a) -sin(a) ; sin(a) cos(a)];
sonars = [73 105 90 ;
  130 78 41 ;
  154 30 15 ;
  154 -30 -15 ;
  130 -78 -41 ;
  73 -105 -90 ;
  -146 -60 -145 ;
  -146 60 145 ;];
% Remove out of range sensors
tmp = []; tmp2 = [];
for si = 1:length(sonars)
  if ranges(si) < 1000
    tmp = [tmp; sonars(si,:)];
    tmp2 = [tmp2; ranges(si)];
  end
end
if size(tmp, 1) == 0
  return
end
sonars = tmp;
ranges = tmp2 ./ SCALE;
sonars(:,1:2) = sonars(:,1:2)*T ./ SCALE;
sonars(:,3) = sonars(:,3) + theta;

for yi = 1:model_size
  for xi = 1:model_size
    for si = 1:size(sonars, 1)
      x = xi - model_size / 2;
      y = yi - model_size / 2;
      % optimized version of norm()
      r = ((x - sonars(si,1))^2 + (y - sonars(si,2))^2)^0.5;
      b = abs(atan2(y - sonars(si,2), x - ...
          sonars(si,1)) / pi * 180 - sonars(si,3));
      while b > 180
        b = abs(b - 360);
      end
      if b > B || r > ranges(si) + TOLERANCE/2
        continue
      end
      if abs(r - ranges(si)) <= TOLERANCE
        psn = ((((R - r) / R) + ((B-b) / B)) / 2) * MAXOCCUPIED;
        psn = 0.5 * psn + 0.5;
      else
        psn = (((R - r) / R) + ((B-b) / B)) / 2;
        psn = 1 - (0.5 * psn + 0.5);
      end
      model(xi, yi) = psn;
    end
  end
end
```