SLAM for Dummies

A Tutorial Approach to Simultaneous Localization and Mapping

By the ‘dummies’

Søren Riisgaard and Morten Rufus Blas

1. Table of contents

2. Introduction

The goal of this document is to give a tutorial introduction to the field of SLAM (Simultaneous Localization And Mapping) for mobile robots. There are numerous papers on the subject but for someone new in the field it will require many hours of research to understand many of the intricacies involved in implementing SLAM. The hope is thus to present the subject in a clear and concise manner while keeping the prerequisites required to understand the document to a minimum. It should actually be possible to sit down and implement basic SLAM after having read this paper.

SLAM can be implemented in many ways. First of all there is a huge amount of different hardware that can be used. Secondly SLAM is more like a concept than a single algorithm. There are many steps involved in SLAM and these different steps can be implemented using a number of different algorithms. In most cases we explain a single approach to these different steps but hint at other possible ways to do them for the purpose of further reading.

The motivation behind writing this paper is primarily to help ourselves understand SLAM better. One will always get a better knowledge of a subject by teaching it. Second of all most of the existing SLAM papers are very theoretic and primarily focus on innovations in small areas of SLAM, which of course is their purpose. The purpose of this paper is to be very practical and focus on a simple, basic SLAM algorithm that can be used as a starting point to get to know SLAM better. For people with some background knowledge in SLAM we here present a complete solution for SLAM using EKF (Extended Kalman Filter). By complete we do not mean perfect. What we mean is that we cover all the basic steps required to get an implementation up and running. It must also be noted that SLAM as such has not been completely solved and there is still considerable research going on in the field.

To make it easy to get started all code is provided, so it is basically just a matter of downloading it, compiling it, plugging in the hardware (SICK laser scanner, ER1 robot) and executing the application; Plug-and-Play. We have used Microsoft Visual

C# and the code will compile in the .Net Framework v. 1.1. Most of the code is very straightforward and can be read almost as pseudo-code, so porting to other languages or platforms should be easy.

3. About SLAM

The term SLAM is as stated an acronym for Simultaneous Localization And Mapping. It was originally developed by Hugh Durrant-Whyte and John J. Leonard [7] based on earlier work by Smith, Self and Cheeseman [6]. Durrant-Whyte and Leonard originally termed it SMAL but it was later changed to give a better impact. SLAM is concerned with the problem of building a map of an unknown environment by a mobile robot while at the same time navigating the environment using the map.

SLAM consists of multiple parts; Landmark extraction, data association, state estimation, state update and landmark update. There are many ways to solve each of the smaller parts. We will be showing examples for each part. This also means that some of the parts can be replaced by a new way of doing this. As an example we will solve the landmark extraction problem in two different ways and comment on the methods. The idea is that you can use our implementation and extend it by using your own novel approach to these algorithms. We have decided to focus on a mobile robot in an indoor environment. You may choose to change some of these algorithms so that it can be for example used in a different environment.

SLAM is applicable for both 2D and 3D motion. We will only be considering 2D motion.

It is helpful if the reader is already familiar with the general concepts of SLAM on an introductory level, e.g. through a university level course on the subject. There are lots of great introductions to this field of research including [6][4]. Also it is helpful to know a little about the Extended Kalman Filter (EKF); sources of introduction are [3][5]. Background information is always helpful as it will allow you to more easily understand this tutorial but it is not strictly required to comprehend all of it.

4. The Hardware

The hardware of the robot is quite important. To do SLAM there is the need for a mobile robot and a range measurement device. The mobile robots we consider are wheeled indoor robots. This documents focus is mainly on software implementation of SLAM and does not explore robots with complicated motion models (models of how the robot moves) such as humanoid robots, autonomous underwater vehicles, autonomous planes, robots with weird wheel configurations etc.
We here present some basic measurement devices commonly used for SLAM on mobile robots.

The robot

Important parameters to consider are ease of use, odometry performance and price. The odometry performance measures how well the robot can estimate its own position, just from the rotation of the wheels. The robot should not have an error of more than 2 cm per meter moved and 2° per 45° degrees turned. Typical robot drivers allow the robot to report its (x,y) position in some Cartesian coordinate system and also to report the robots current bearing/heading.

There is the choice to build the robot from scratch. This can be very time consuming, but also a learning experience. It is also possible to buy robots ready to use, like Real World Interface or the Evolution Robotics ER1 robot [10]. The RW1 is not sold anymore, though, but it is usually available in many computer science labs around the world. The RW1 robot has notoriously bad odometry, though. This adds to the problem of estimating the current position and makes SLAM considerably harder. ER1 is the one we are using. It is small and very cheap. It can be bought for only 200USD for academic use and 300USD for private use. It comes with a camera and a robot control system. We have provided very basic drivers in the appendix and on the website.

The range measurement device

The range measurement device used is usually a laser scanner nowadays. They are very precise, efficient and the output does not require much computation to process. On the downside they are also very expensive. A SICK scanner costs about 5000USD. Problems with laser scanners are looking at certain surfaces including glass, where they can give very bad readings (data output). Also laser scanners cannot be used underwater since the water disrupts the light and the range is drastically reduced.

Second there is the option of sonar. Sonar was used intensively some years ago. They are very cheap compared to laser scanners. Their measurements are not very good compared to laser scanners and they often give bad readings. Where laser scanners have a single straight line of measurement emitted from the scanner with a width of as little as 0.25 degrees a sonar can easily have beams up to 30 degrees in width. Underwater, though, they are the best choice and resemble the way dolphins navigate. The type used is often a Polaroid sonar. It was originally developed to measure the distance when taking pictures in Polaroid cameras. Sonar has been successfully used in [7].

The third option is to use vision. Traditionally it has been very computationally intensive to use vision and also error prone due to changes in light. Given a room without light a vision system will most certainly not work. In the recent years, though, there have been some interesting advances within this field. Often the systems use a stereo or triclops system to measure the distance. Using vision resembles the way humans look at the world and thus may be more intuitively appealing than laser or sonar. Also there is a lot more information in a picture compared to laser and sonar scans. This used to be the bottleneck, since all this data needed to be processed, but with advances in algorithms and computation power this is becoming less of a problem. Vision based range measurement has been successfully used in [8].

We have chosen to use a laser range finder from SICK [9]. It is very widely used, it is not dangerous to the eye and has nice properties for use in SLAM. The measurement error is +- 50mm, which seems like very much, but in practice the error was much

smaller. The newest laser scanners from SICK have measurement errors down to +- 5 mm.

5. The SLAM Process

The SLAM process consists of a number of steps. The goal of the process is to use the environment to update the position of the robot. Since the odometry of the robot (which gives the robots position) is often erroneous we cannot rely directly on the odometry. We can use laser scans of the environment to correct the position of the robot. This is accomplished by extracting features from the environment and re-observing when the robot moves around. An EKF (Extended Kalman Filter) is the heart of the SLAM process. It is responsible for updating where the robot thinks it is based on these features. These features are commonly called landmarks and will be explained along with the EKF in the next couple of chapters. The EKF keeps track of an estimate of the uncertainty in the robots position and also the uncertainty in these landmarks it has seen in the environment.

An outline of the SLAM process is given below.

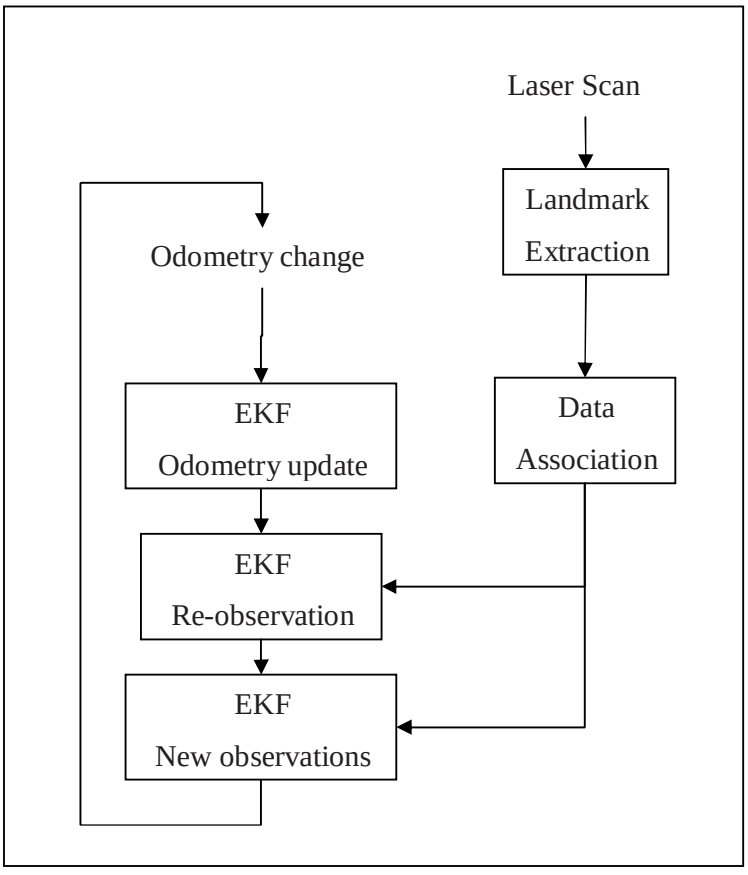


Figure 1 Overview of the SLAM process

When the odometry changes because the robot moves the uncertainty pertaining to the robots new position is updated in the EKF using Odometry update. Landmarks are then extracted from the environment from the robots new position. The robot then attempts to associate these landmarks to observations of landmarks it previously has seen. Re-observed landmarks are then used to update the robots position in the EKF. Landmarks which have not previously been seen are added to the EKF as new observations so they can be re-observed later. All these steps will be explained in the next chapters in a very practical fashion relative to how our ER1 robot was implemented. It should be noted that at any point in these steps the EKF will have an estimate of the robots current position.

The diagrams below will try to explain this process in more detail:

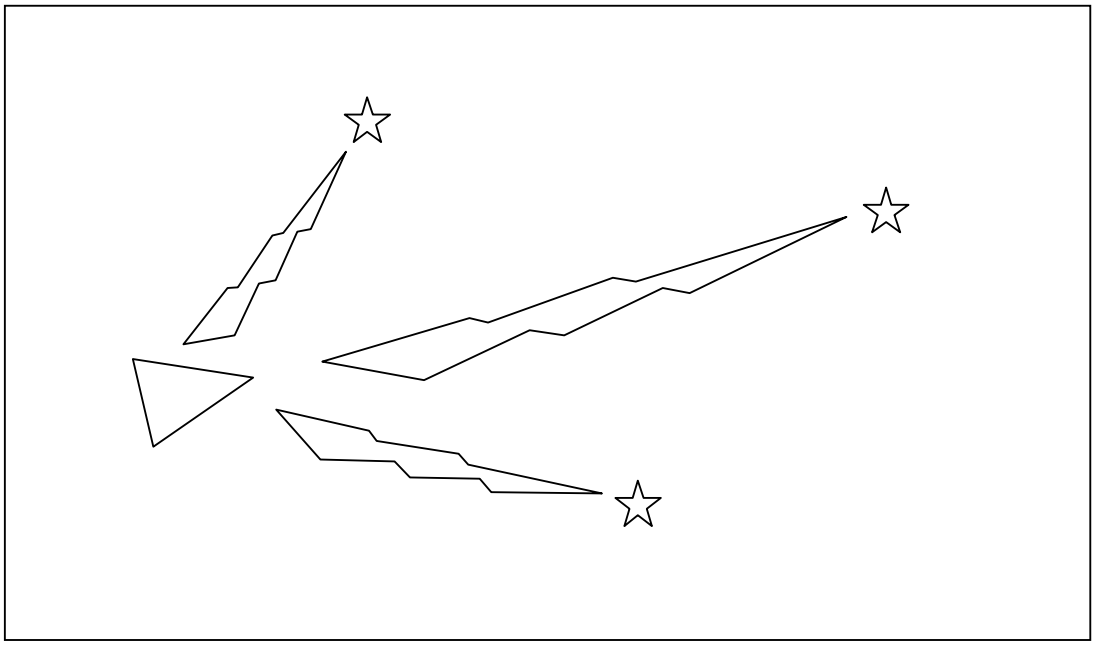


Figure 2 The robot is represented by the triangle. The stars represent landmarks. The robot initially measures using its sensors the location of the landmarks (sensor measurements illustrated with lightning).

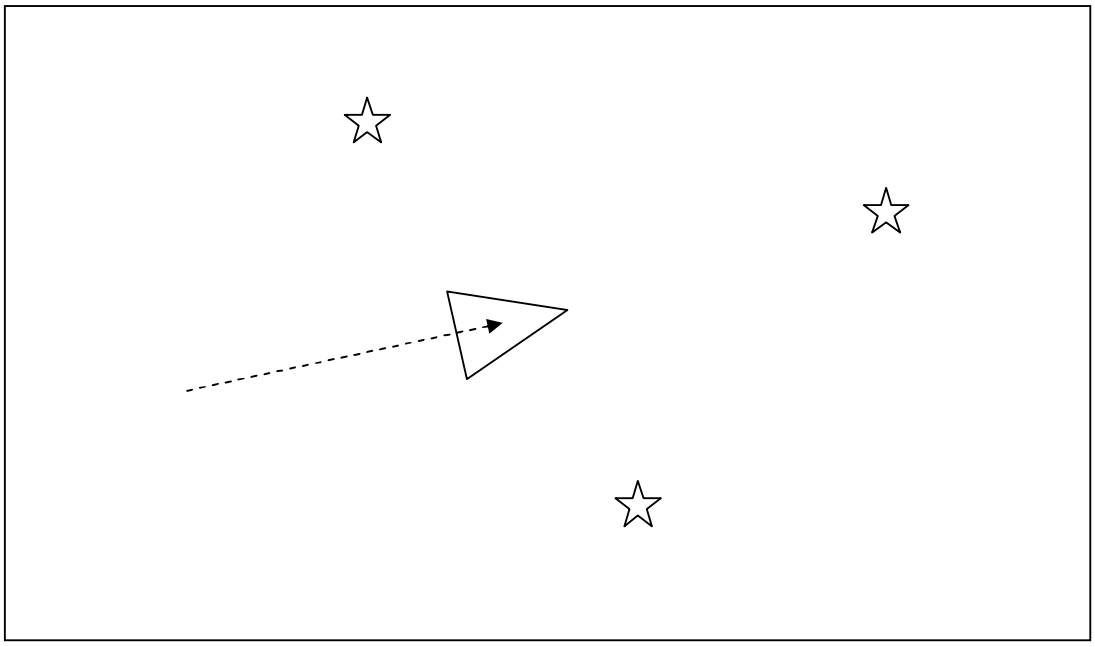


Figure 3 The robot moves so it now thinks it is here. The distance moved is given by the robots odometry.

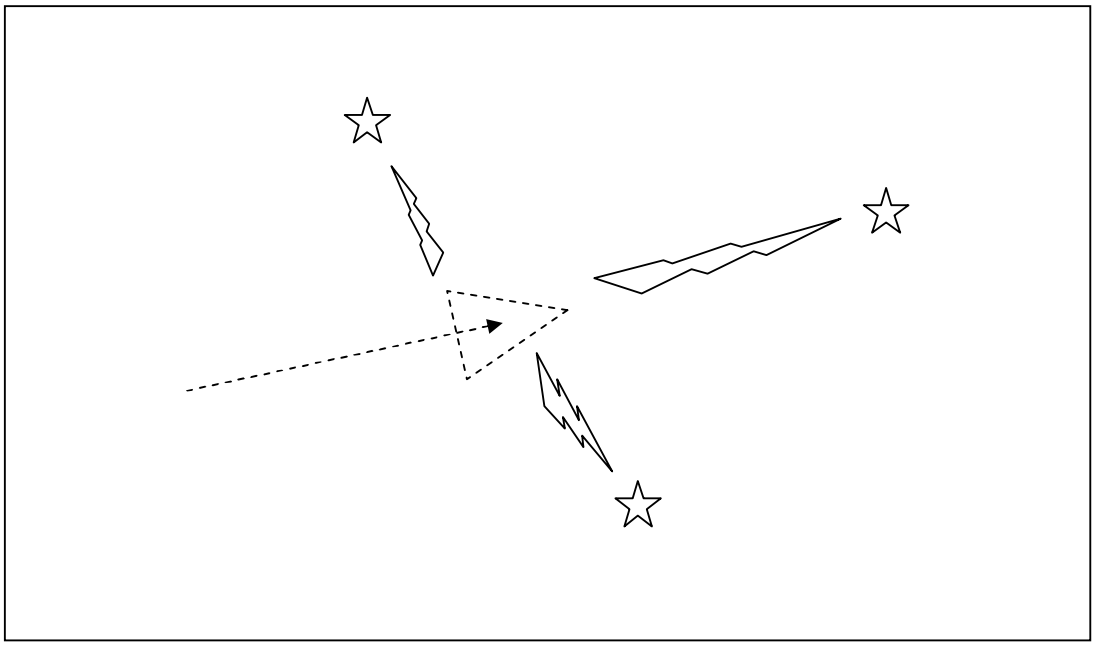


Figure 4 The robot once again measures the location of the landmarks using its sensors but finds out they don't match with where the robot thinks they should be (given the robots location). Thus the robot is not where it thinks it is.

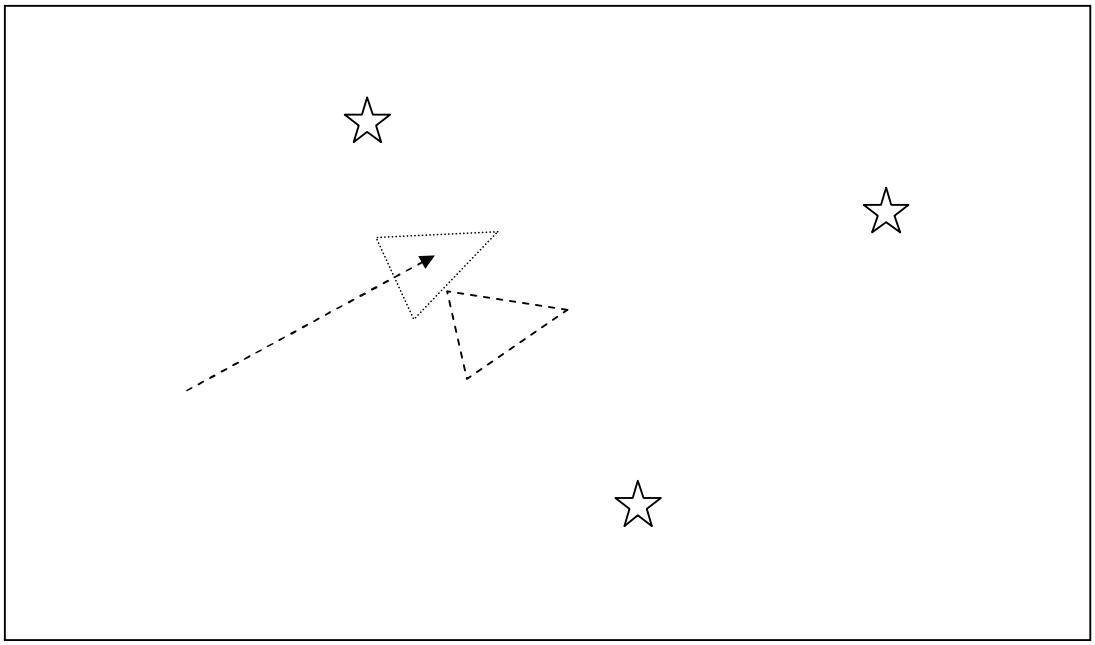


Figure 5 As the robot believes more its sensors than its odometry it now uses the information gained about where the landmarks actually are to determine where it is (the location the robot originally thought it was at is illustrated by the dashed triangle).

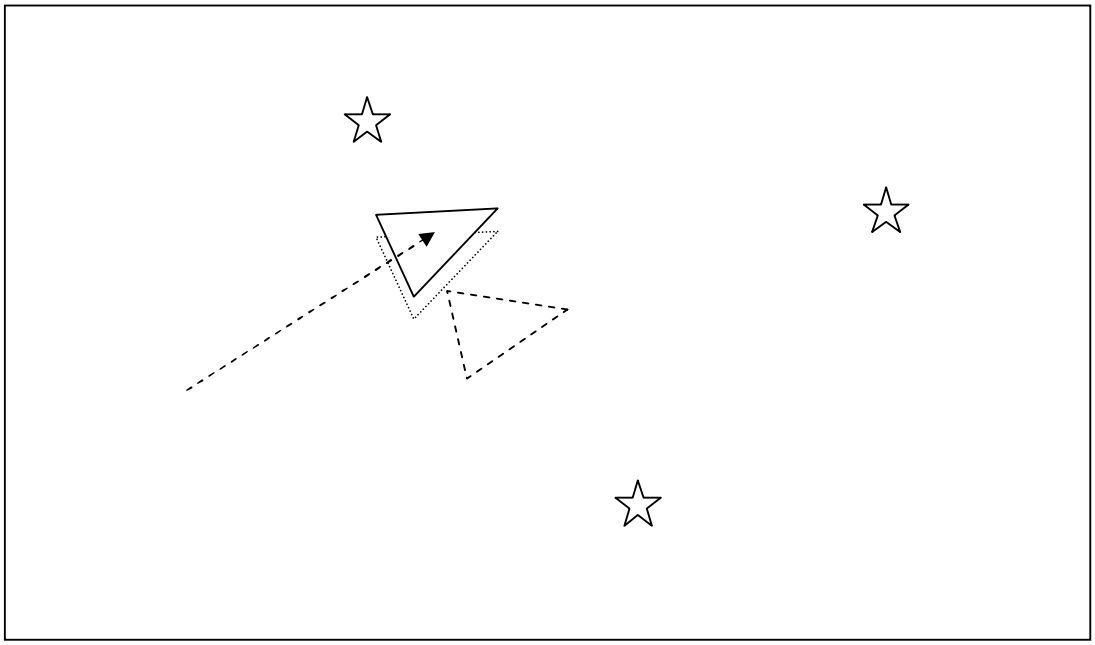


Figure 6 In actual fact the robot is here. The sensors are not perfect so the robot will not precisely know where it is. However this estimate is better than relying on odometry alone. The dotted triangle represents where it thinks it is; the dashed triangle where odometry told it it was; and the last triangle where it actually is.

6. Laser Data

The first step in the SLAM process is to obtain data about the surroundings of the robot. As we have chosen to use a laser scanner we get laser data. The SICK laser scanner we are using can output range measurements from an angle of 100° or 180°. It has a vertical resolution of 0.25°, 0.5° or 1.0°, meaning that the width of the area the laser beams measure is 0.25°, 0.5° or 1.0° wide. A typical laser scanner output will look like this:

2.98, 2.99, 3.00, 3.01, 3.00, 3.49, 3.50, ..., 2.20, 8.17, 2.21

The output from the laser scanner tells the ranges from right to left in terms of meters. If the laser scanner for some reason cannot tell the exact length for a specific angle it will return a high value, we are using 8.1 as the threshold to tell if the value is an error. Some laser scanners can be configured to have ranges longer than 8.1 meters. Lastly it should be noted that laser scanners are very fast. Using a serial port they can be queried at around 11 Hz.

The code to interface with the laser scanner can be seen in Appendix B: SICK LMS 200 interface code.

7. Odometry Data

An important aspect of SLAM is the odometry data. The goal of the odometry data is to provide an approximate position of the robot as measured by the movement of the wheels of the robot, to serve as the initial guess of where the robot might be in the EKF. Obtaining odometry data from an ER1 robot is quite easy using the built-in telnet server. One can just send a text string to the telnet server on a specific port and the server will return the answer.

The difficult part about the odometry data and the laser data is to get the timing right. The laser data at some time t will be outdated if the odometry data is retrieved later. To make sure they are valid at the same time one can extrapolate the data. It is easiest to extrapolate the odometry data since the controls are known. It can be really hard to predict how the laser scanner measurements will be. If one has control of when the measurements are returned it is easiest to ask for both the laser scanner values and the odometry data at the same time. The code to interface with the ER1 robot is shown in

Appendix C: ER1 interface code.

8. Landmarks

Landmarks are features which can easily be re-observed and distinguished from the environment. These are used by the robot to find out where it is (to localize itself). One way to imagine how this works for the robot is to picture yourself blindfolded. If you move around blindfolded in a house you may reach out and touch objects or hug walls so that you don't get lost. Characteristic things such as that felt by touching a doorframe may help you in establishing an estimate of where you are. Sonars and laser scanners are a robots feeling of touch.

Below are examples of good landmarks from different environments:

Figure 7 The statue of liberty is a good landmark as it is unique and can easily be seen from various environments or locations such as on land, from the sea, and from the air.

Figure 8 The wooden pillars at a dock may be good landmarks for an underwater vehicle.

As you can see the type of landmarks a robot uses will often depend on the environment in which the robot is operating.

Landmarks should be re-observable by allowing them for example to be viewed (detected) from different positions and thus from different angles.
Landmarks should be unique enough so that they can be easily identified from one time-step to another without mixing them up. In other words if you re-observe two landmarks at a later point in time it should be easy to determine which of the landmarks is which of the landmarks we have previously seen. If two landmarks are very close to each other this may be hard.

Landmarks you decide a robot should recognize should not be so few in the environment that the robot may have to spend extended time without enough visible landmarks as the robot may then get lost.

If you decide on something being a landmark it should be stationary. Using a person as a landmark is as such a bad idea. The reason for this criterion is fairly straightforward. If the landmark is not always in the same place how can the robot know given this landmark in which place it is.

The key points about suitable landmarks are as follows:

- Landmarks should be easily re-observable.
- Individual landmarks should be distinguishable from each other.
- Landmarks should be plentiful in the environment.
- Landmarks should be stationary.

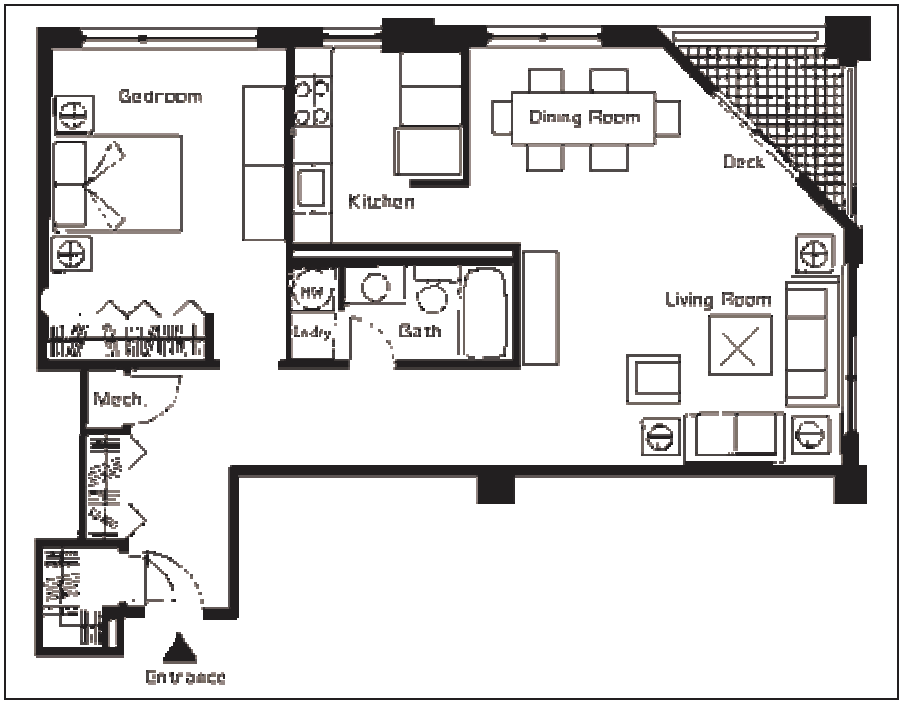


Figure 9 In an indoor environment such as that used by our robot there are many straight lines and well defined corners. These could all be used as landmarks.

9. Landmark Extraction

Once we have decided on what landmarks a robot should utilize we need to be able to somehow reliable extract them from the robots sensory inputs.

As mentioned in the introduction there are multiple ways to do landmark extraction and it depends largely on what types of landmarks are attempted extracted as well as what sensors are used.

We will present basic landmark extraction algorithms using a laser scanner. They will use two landmark extraction algorithm called Spikes and RANSAC.

Spike landmarks

The spike landmark extraction uses extrema to find landmarks. They are identified by finding values in the range of a laser scan where two values differ by more than a certain amount, e.g. 0.5 meters. This will find big changes in the laser scan from e.g. when some of the laser scanner beams reflect from a wall and some of the laser scanner beams do not hit this wall, but are reflected from some things further behind the wall.

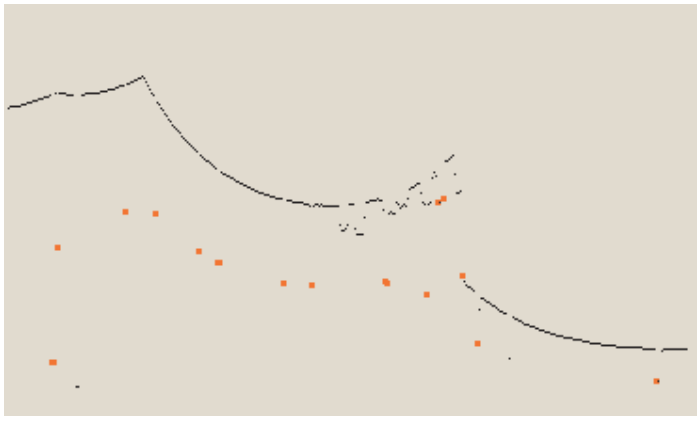


Figure 10: Spike landmarks. The red dots are table legs extracted as landmarks.

The spikes can also be found by having three values next to each other, A, B and C. Subtracting B from A and B from C and adding the two numbers will yield a value.

This method is better for finding spikes as it will find actual spikes and not just permanent changes in range.

Spike landmarks rely on the landscape changing a lot between two laser beams. This means that the algorithm will fail in smooth environments.

RANSAC

RANSAC (Random Sampling Consensus) is a method which can be used to extract lines from a laser scan. These lines can in turn be used as landmarks. In indoor environments straight lines are often observed by laser scans as these are characteristic of straight walls which usually are common.

RANSAC finds these line landmarks by randomly taking a sample of the laser readings and then using a least squares approximation to find the best fit line that runs through these readings. Once this is done RANSAC checks how many laser readings lie close to this best fit line. If the number is above some threshold we can safely assume that we have seen a line (and thus seen a wall segment). This threshold is called the consensus.

The below algorithm outlines the line landmark extraction process for a laser scanner with a 180° field of view and one range measurement per degree. The algorithm assumes that the laser data readings are converted to a Cartesian coordinate system – see Appendix A. Initially all laser readings are assumed to be unassociated to any lines. In the algorithm we only sample laser data readings from unassociated readings.

```
While
   • there are still unassociated laser readings,
   • and the number of readings is larger than the consensus,
   • and we have done less than N trials.
do
   -  Select a random laser data reading.
```

- Randomly sample S data readings within D degrees of this laser data reading (for example, choose 5 sample readings that lie within 10 degrees of the randomly selected laser data reading).

- Using these S samples and the original reading calculate a least squares best fit line.

- Determine how many laser data readings lie within X centimeters of this best fit line.

- If the number of laser data readings on the line is above some consensus C do the following:
    - calculate new least squares best fit line based on all the laser readings determined to lie on the old best fit line.

    - Add this best fit line to the lines we have extracted.

    - Remove the number of readings lying on the line from the total set of unassociated readings.

od

This algorithm can thus be tuned based on the following parameters:

N – Max number of times to attempt to find lines.

S – Number of samples to compute initial line.

D – Degrees from initial reading to sample from.

X – Max distance a reading may be from line to get associated to line.

C – Number of points that must lie on a line for it to be taken as a line.

The EKF implementation assumes that landmarks come in as a range and bearing from the robots position. One can easily translate a line into a fixed point by taking another fixed point in the world coordinates and calculating the point on the line closest to this fixed point. Using the robots position and the position of this fixed point on the line it is trivial to calculate a range and bearing from this.

Using simple trigonometry one can easily calculate this point. Here illustrated using the origin as a fixed point:

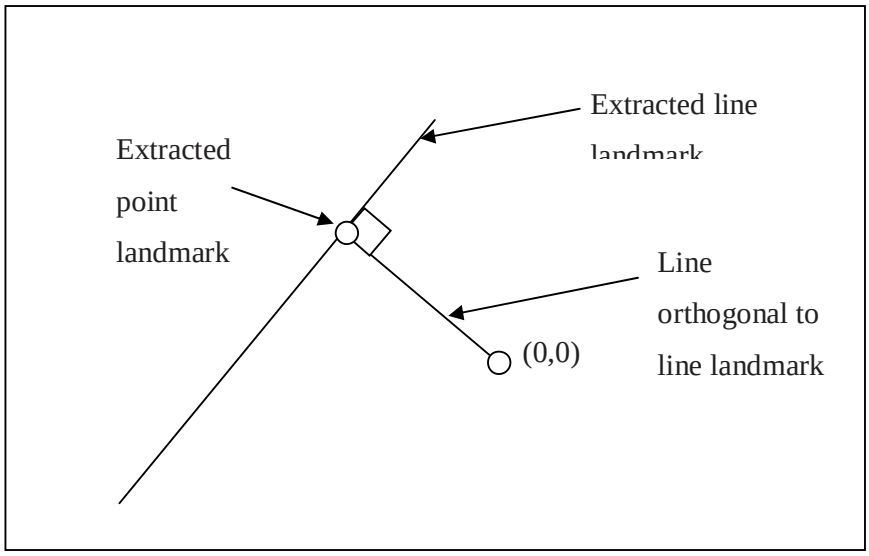


Figure 11 Illustration of how to get an extracted line landmark as a point.

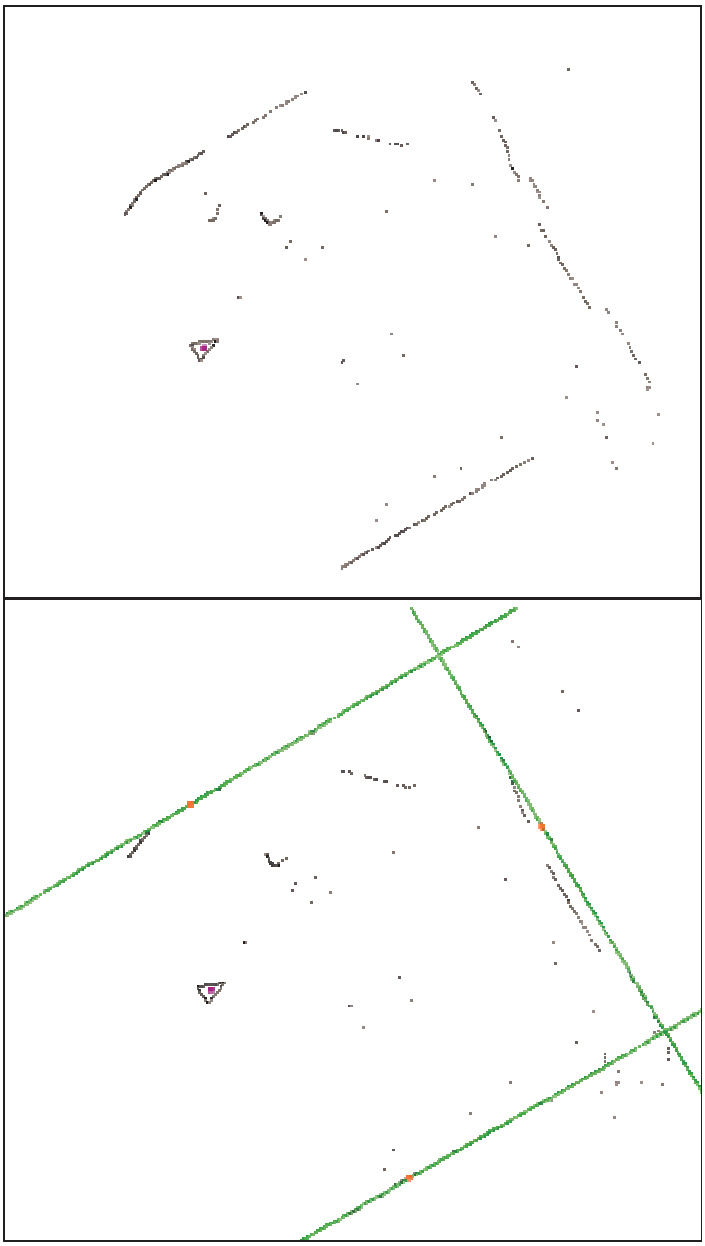


Figure 12 The RANSAC algorithm finds the main lines in a laser scan. The green lines are the best fit lines representing the landmarks. The red dots represent the landmarks approximated to points. By changing the RANSAC parameters you could also extract the small wall segments. They are not considered very reliable landmarks so were not used. Lastly just above the robot is a person. RANSAC is robust against people in the laser scan.

Another possibility is to expand the EKF implementation so it could handle lines instead of just points. This however is complicated so is not dealt with in this tutorial.

Multiple strategies

We have presented two different approaches to landmark extraction. Both extract different types of landmarks and are suitable for indoor environments. Spikes however is fairly simple and is not robust against environments with people. The reason for this is that Spikes picks up people as spikes as they theoretically are good landmarks (they stand out from the environment).
As RANSAC uses line extraction it will not pick up people as landmarks as they do not individually have the characteristic shape of a line.
A third method which we will not explore is called scan-matching where you attempt match two successive laser scans. We name it here for people interested in other approaches.

Code for landmark extraction algorithms can be found in Appendix D: Landmark extraction code.

10. Data Association

The problem of data association is that of matching observed landmarks from different (laser) scans with each other. We have also referred to this as re-observing landmarks.

To illustrate what is meant by this we will give an example:

For us humans we may consider a chair a landmark. Let us say we are in a room and see a specific chair. Now we leave the room and then at some later point subsequently return to the room. If we then see a chair in the room and say that it is the same chair we previously saw then we have associated this chair to the old chair. This may seem simple but data association is hard to do well. Say the room had two chairs that looked practically identical. When we subsequently return to the room we might not be able to distinguish accurately which of the chairs were which of the chairs we originally saw (as they all look the same). Our best bet is to say that the one to the left must be the one we previously saw to the left, and the one to the right must be the one we previously saw on the right.

In practice the following problems can arise in data association:

- You might not re-observe landmarks every time step.
- You might observe something as being a landmark but fail to ever see it again.
- You might wrongly associate a landmark to a previously seen landmark.

As stated in the landmarks chapter it should be easy to re-observe landmarks. As such the first two cases above are not acceptable for a landmark. In other words they are bad landmarks. Even if you have a very good landmark extraction algorithm you may run into these so it is best to define a suitable data-association policy to minimize this.

The last problem where you wrongly associate a landmark can be devastating as it means the robot will think it is somewhere different from where it actually is.

We will now define a data-association policy that deals with these issues. We assume that a database is set up to store landmarks we have previously seen. The database is usually initially empty. The first rule we set up is that we don't actually consider a landmark worthwhile to be used in SLAM unless we have seen it N times. This eliminates the cases where we extract a bad landmark. The below-mentioned validation gate is explained further down in the text.

1. When you get a new laser scan use landmark extraction to extract all visible landmarks.

2. Associate each extracted landmark to the closest landmark we have seen more than N times in the database.

3. Pass each of these pairs of associations (extracted landmark, landmark in database) through a validation gate.

    a. If the pair passes the validation gate it must be the same landmark we have re-observed so increment the number of times we have seen it in the database.

    b. If the pair fails the validation gate add this landmark as a new landmark in the database and set the number of times we have seen it to 1.

This technique is called the nearest-neighbor approach as you associate a landmark with the nearest landmark in the database.
The simplest way to calculate the nearest landmark is to calculate the Euclidean distance.[1] Other methods include calculating the Mahalanobis distance which is better but more complicated. This was not used in our approach as RANSAC landmarks usually are far apart which makes using the Euclidean distance suitable.

The validation gate uses the fact that our EKF implementation gives a bound on the uncertainty of an observation of a landmark. Thus we can determine if an observed landmark is a landmark in the database by checking if the landmark lies within the area of uncertainty. This area can actually be drawn graphically and is known as an error ellipse.
By setting a constant λ an observed landmark is associated to a landmark if the following formula holds:

$$\nu_i^T \mathbf{S}_i^{-1} \nu_i \leq \lambda.$$

Where ν_i is the innovation and S_i is the innovation covariance defined in the EKF chapter

11. The EKF

The Extended Kalman Filter is used to estimate the state (position) of the robot from odometry data and landmark observations. The EKF is usually described in terms of state estimation alone (the robot is given a perfect map). That is, it does not have the map update which is needed when using EKF for SLAM. In SLAM vs. a state estimation EKF especially the matrices are changed and can be hard to figure out how to implement, since it is almost never mentioned anywhere. We will go through each of these. Most of the EKF is standard, as a normal EKF, once the matrices are set up, it is basically just a set of equations.

Overview of the process

As soon as the landmark extraction and the data association is in place the SLAM process can be considered as three steps:

1. Update the current state estimate using the odometry data
2. Update the estimated state from re-observing landmarks.
3. Add new landmarks to the current state.

The first step is very easy. It is just an addition of the controls of the robot to the old state estimate. E.g. the robot is at point (x, y) with rotation theta and the controls are (dx, dy) and change in rotation is dtheta. The result of the first step is the new state of the robot (x+dx, y+dy) with rotation theta+dtheta.

In the second step the re-observed landmarks are considered. Using the estimate of the current position it is possible to estimate where the landmark should be. There is usually some difference, this is called the innovation. So the innovation is basically the difference between the estimated robot position and the actual robot position, based on what the robot is able to see. In the second step the uncertainty of each observed landmark is also updated to reflect recent changes. An example could be if

the uncertainty of the current landmark position is very little. Re-observing a landmark from this position with low uncertainty will increase the landmark certainty, i.e. the variance of the landmark with respect to the current position of the robot.

In the third step new landmarks are added to the state, the robot map of the world. This is done using information about the current position and adding information about the relation between the new landmark and the old landmarks.

The matrices

It should be noted that there is a lot of different notions for the same variables in the different papers. We use some fairly common notions.

The system state: X

x_r
y_r
$theta_r$
x_1
y_1
...
...
x_n
y_n

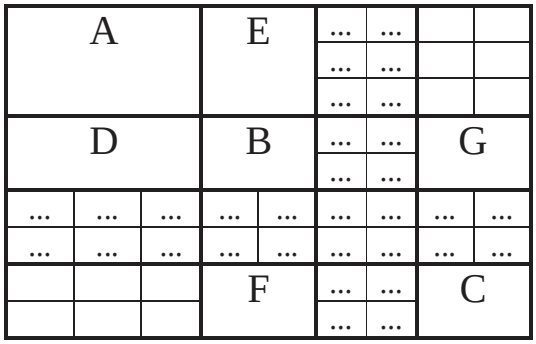
X is probably one of the most important matrices in the system along with the covariance matrix. It contains the position of the robot, x, y and theta. Furthermore it contains the x and y position of each landmark. The matrix can be seen to the right. It is important to have the matrix as a vertical matrix to make sure that all the equations will work. The size of X is 1 column wide and 3+2*n rows high, where n is the number of landmarks. Usually the values saved will be in either meters or millimeters for the ranges. Which one is used does not matter, it is just important, of course, to use the same notion everywhere. The bearing is saved in either degrees or radians. Again it is a question of using the same notion everywhere.

The covariance matrix: P

Quick math recap:

The covariance of two variates provides a measure of how strongly correlated these two variables are. Correlation is a concept used to measure the degree of linear dependence between variables.

The covariance matrix P is a very central matrix in the system. It contains the covariance on the robot position, the covariance on the landmarks, the covariance between robot position and landmarks and finally it contains the covariance between the landmarks. The figure on the right shows the content of the covariance matrix P. The first cell, A contains the covariance on the robot position. It is a 3 by 3 matrix (x, y and theta). B is the covariance on the first landmark. It is a 2 by 2 matrix, since the landmark does not have an orientation, theta. This continues down to C, which is the covariance for the last landmark. The cell D contains the covariance between the robot state and the first landmark. The cell E contains the covariance between the first landmark and the robot state. E can be deduced from D by transposing the sub-matrix D. F contains the covariance between the last landmark and the first landmark, while G contains the covariance between the first landmark and the last landmark, which again can be deduced by transposing F. So even though the covariance matrix may seem complicated it is actually built up very systematically. Initially as the robot has not seen any landmarks the covariance matrix P only includes the matrix A. The covariance matrix must be initialized using some default values for the diagonal. This reflects uncertainty in the initial position. Depending on the actual implementation there will often be a singular error if the initial uncertainty is not included in some of the calculations, so it is a good idea to include some initial error even though there is reason to believe that the initial robot position is exact.



The Kalman gain: K

The Kalman gain K is computed to find out how much we will trust the observed landmarks and as such how much we want to gain from the new knowledge they provide. If we can see that the robot should be moved 10 cm to the right, according to the landmarks we use the Kalman Gain to find out how much we actually correct the position, this may only be 5 cm because we do not trust the landmarks completely, but rather find a compromise between the odometry and the landmark correction. This is done using the uncertainty of the observed landmarks along with a measure of the quality of the range measurement device and the odometry performance of the robot. If the range measurement device is really bad compared to the odometry performance of the robot, we of course do not trust it very much, so the Kalman gain will be low. On the contrary, if the range measurement device is very good compared to the odometry performance of the robot the Kalman gain will be high. The matrix can be seen to the right. The first row shows how much should be gained from the innovation for the first row of the state X. The first column in the first row describes how much should be gained from the innovation in terms of range, the second column in the first row describes how much should be gained from the innovation in terms of the bearing. Again both are for the first row in the state, which is the x value of the robot position. The matrix continues like down through the robot position; the first three rows, and the landmarks; each two new rows. The size of the matrix is 2 columns and 3+2*n rows, where n is the number of landmarks.

x_r	x_b
y_r	y_b
t_r	t_b
$x_{1,r}$	$x_{1,b}$
$y_{1,r}$	$y_{1,b}$
...	...
...	...
$x_{n,r}$	$x_{n,b}$
$y_{n,r}$	$y_{n,b}$

The Jacobian of the measurement model: H

The Jacobian of the measurement model is closely related to the measurement model, of course, so let's go through the measurement model first. The measurement model defines how to compute an expected range and bearing of the measurements (observed landmark positions). It is done using the following formula, which is denoted h:

$$\begin{bmatrix} \text{range} \\ \text{bearing} \end{bmatrix} = \begin{bmatrix} \sqrt{(\lambda_x - x)^2 + (\lambda_y - y)^2} + v_r \\ \tan^{-1}\left(\frac{\lambda_y - y}{\lambda_x - x}\right) - \theta + v_\theta \end{bmatrix}$$

Where lambda x is the x position of the landmark, x is the current estimated robot x position, lambda y is the y position of the landmark and y is the current estimated robot y position. Theta is the robot rotation. This will give us the predicted measurement of the range and bearing to the landmark. The Jacobian of this matrix with respect to x, y and theta, H, is:

$$\begin{bmatrix} \frac{x - \lambda_x}{r} & \frac{y - \lambda_y}{r} & 0 \\ \frac{\lambda_y - y}{r^2} & \frac{\lambda_x - x}{r^2} & -1 \end{bmatrix}$$

H shows us how much the range and bearing changes as x, y and theta changes. The first element in the first row is the change in range with respect to the change in the x axis. The second element is with respect to the change in the y axis. The last element is with respect to the change in theta, the robot rotation. Of course this value is zero as the range does not change as the robot rotates. The second row gives the same information, except that this is the change in bearing for the landmark. This is the contents of the usual H for regular EKF state estimation. When doing SLAM we need some additional values for the landmarks:

X_r	Y_r	T_r	X_1	Y_1	X_2	Y_2	X_3	Y_3
A	B	C	0	0	-A	-B	0	0
D	E	F	0	0	-D	-E	0	0

When using the matrix H e.g. for landmark number two we will be using the matrix above. The upper row is for information purposes only; it is not part of the matrix.

This means that the first 3 columns are the regular H, as for regular EKF state estimation. For each landmark we add two columns. When using the H matrix for landmark two as above we fill it out like above with X2 set to –A and –D and Y2 set to –B and -E. The columns for the rest of the landmarks are 0. are the same as the first two columns of the original H, just negated. We only use two terms, X_2 and Y_2, because the landmarks do not have any rotation.

The Jacobian of the prediction model: A

Like H, the Jacobian of the prediction model is closely related to the prediction model, of course, so let's go through the prediction model first. The prediction model defines how to compute an expected position of the robot given the old position and the control input. It is done using the following formula, which is denoted f:

$$f = \begin{bmatrix} x + \Delta t \cos\theta + q\Delta t \cos\theta \\ y + \Delta t \sin\theta + q\Delta t \sin\theta \\ \theta + \Delta\theta + q\Delta\theta \end{bmatrix}$$

Where x and y is the robot position, theta the robot rotation, Δt is the change in thrust and q is the error term. We are using the change in position directly from the odometry input from the ER1 system, so we use x, y, theta and Δx, Δy and Δtheta directly and the process noise, described later:

x + Δx + Δx * q
y + Δy + Δy * q
theta + Δtheta + Δtheta * q

Anyway we assume the linearized version when calculating the jacobian A yielding:

$$\begin{bmatrix} 1 & 0 & -\Delta t \sin\theta \\ 0 & 1 & \Delta t \cos\theta \\ 0 & 0 & 1 \end{bmatrix}$$

The calculations are the same as for the H matrix, except that we now have one more row for the robot rotation. Since it is only used for robot position prediction it will also not be extended for the rest of the landmarks. As can be seen from the first matrix, the prediction model, the term $-\Delta t * \sin\theta$ is the same as $-\Delta y$ in our case and $-\Delta t * \cos\theta$ is the same as Δx. So we can just use our control terms, yielding:

1	0	$-\Delta y$
0	1	Δx
0	0	1

The SLAM specific Jacobians: J_{xr} and J_z

When doing SLAM there are some Jacobians which are only used in SLAM. This is of course in the integration of new features, which is the only step that differs from regular state estimation using EKF. The first is Jxr. It is basically the same as the jacobian of the prediction model, except that we start out without the rotation term. It is the jacobian of the prediction of the landmarks, which does not include prediction of theta, with respect to the robot state [x, y, theta] from X:

$$J_{xr} = \begin{bmatrix} 1 & 0 & -\Delta t \sin\theta \\ 0 & 1 & \Delta t \cos\theta \end{bmatrix}$$

The jacobian J_z is also the jacobian of the prediction model for the landmarks, but this time with respect to [range, bearing]. This in turn yields:

$$J_z = \begin{bmatrix} \cos(\theta + \Delta\theta) & -\Delta t * \sin(\theta + \Delta\theta) \\ \sin(\theta + \Delta\theta) & \Delta t * \cos(\theta + \Delta\theta) \end{bmatrix}$$

The process noise: Q and W

The process is assumed to have a gaussian.noise proportional to the controls, Δx, Δy and Δt. The noise is denoted Q, which is a 3 by 3 matrix. It is usually calculated by multiplying some gaussian sample C with W and W transposed:

$c\Delta x^2$		
	$c\Delta y^2$	
		$c\Delta t^2$

$$W = \begin{bmatrix} \Delta t \cos\theta & \Delta t \sin\theta & \Delta\theta \end{bmatrix}^T$$

$$Q = WCW^T$$

C is be a representation of how exact the odometry is. The value should be set according to the robot odometry performance and is usually easiest to set by experiments and tuning the value.

In most papers the process noise is either denoted Q alone or as WQW^T. The notion C is basically never used, but is needed here to show the two approaches.

The measurement noise: R and V

The range measurement device is also assumed to have gaussian noise proportional to the range and bearing. It is calculated as VRV^T. V is just a 2 by 2 identity matrix. R is also a 2 by 2 matrix with numbers only in the diagonal. In the upper left corner we have the range, r, multiplied by some constants c and d. The constants should represent the accuracy of the measurement device. If for example the range error has 1 cm variance it should, c should be a gaussian with variance 0.01. If the bearing error is always 1 degree bd should be replaced with the number 1, presuming that degrees are used for measurements. Usually it will not make sense to make the error on the angle proportional with the size of the angle.

rc	
	bd

Step 1: Update current state using the odometry data

This step, called the prediction step we update the current state using the odometry data. That is we use the controls given to the robot to calculate an estimate of the new position of the robot. To update the current state we use the following equation:

$$\begin{bmatrix} x + \Delta t \cos\theta + q\Delta t \cos\theta \\ y + \Delta t \sin\theta + q\Delta t \sin\theta \\ \theta + \Delta\theta + q\Delta\theta \end{bmatrix}$$

In our simple odometry model we can simply just add the controls as noted in a previous chapter:

x + Δx
y + Δy
theta + Δtheta

This should be updated in the first three spaces in the state vector, X. We also need to update the A matrix, the jacobian of the prediction model, every iteration:

1	0	-Δy
0	1	Δx
0	0	1

Also Q should be updated to reflect the control terms, Δx, Δy and Δt:

$c\Delta x^2$	$c\Delta x\Delta y$	$c\Delta x\Delta t$
$cc\Delta y\Delta x$	$c\Delta y^2$	$c\Delta y\Delta t$
$c\Delta t\Delta x$	$c\Delta t\Delta y$	$c\Delta t^2$

Finally we can calculate the new covariance for the robot position. Since the covariance for the robot position is just the top left 3 by 3 matrix of P we will only update this:

$$P^{rr} = A\, P^{rr}\, A + Q$$

Where the symbol P^{rr} is the top left 3 by 3 matrix of P.

Now we have updated the robot position estimate and the covariance for this position estimate. We also need to update the robot to feature cross correlations. This is the top 3 rows of the covariance matrix:

$$P^{ri} = A\, P^{ri}$$

Step 2: Update state from re-observed landmarks

The estimate we obtained for the robot position is not completely exact due to the odometry errors from the robot. We want to compensate for these errors. This is done using landmarks. The landmarks have already been discussed including how to observe them and how to associate them to already known landmarks. Using the associated landmarks we can now calculate the displacement of the robot compared to what we think the robot position is. Using the displacement we can update the robot position. This is what we want to do in step 2. This step is run for each re-observed landmark. Landmarks that are new are not dealt with until step 3. Delaying the incorporation of new landmarks until the next step will decrease the computation cost needed for this step, since the covariance matrix, P, and the system state, X, are smaller.

We will try to predict where the landmark is using the current estimated robot position (x, y) and the saved landmark position (λ_x, λ_y). With the following formula:

$$\begin{bmatrix} \text{range} \\ \text{bearing} \end{bmatrix} = \begin{bmatrix} \sqrt{(\lambda_x - x)^2 + (\lambda_y - y)^2} + v_r \\ \tan^{-1}\left(\dfrac{\lambda_y - y}{\lambda_x - x}\right) - \theta + v_\theta \end{bmatrix}$$

We get the range and bearing to the landmark, h, also seen when calculating the jacobian H. This can be compared to the range and bearing for the landmark we get from the data association, which we will denote z. But first we need some more computations. From the previous chapters we have the jacobian H:

X_r	Y_r	T_r	X_1	Y_1	X_2	Y_2	X_3	Y_3
A	B	C	0	0	-A	-B	0	0
D	E	F	0	0	-D	-E	0	0

As previously stated the values are calculated as follows:

$$\begin{bmatrix} \frac{x-\lambda_x}{r} & \frac{y-\lambda_y}{r} & 0 \\ \frac{\lambda_y-y}{r^2} & \frac{\lambda_x-x}{r^2} & -1 \end{bmatrix}$$

Again, remember that only the first three columns and the columns valid for the current landmark should be filled out.

The error matrix R should also be updated to reflect the range and bearing in the current measurements. A good starting value for rc is the range value multiplied with 0.01, meaning there is 1 % error in the range. A good error for the bd value is 1, meaning there is 1 degree error in the measurements. This error should not be proportional with the size of the angle; this would not make sense, of course.

rc	
	bd

Now we can compute the Kalman gain. It is calculated using the following formula:

$$K = P * H^T * (H * P * H^T + V * R * V^T)^{-1}$$

The Kalman now contains a set of numbers indicating how much each of the landmark positions and the robot position should be updated according to the re-observed landmark. The term $(H * P * H^T + V * R * V^T)$ is called the innovation covariance, S, it is also used in the Data Association chapter when calculating the validation gate for the landmark.

Finally we can compute a new state vector using the Kalman gain:

$$X = X + K * (z - h)$$

This operation will update the robot position along with all the landmark positions, given the term (z-h) does not result in (0, 0). Note that (z-h) yields a result of two numbers which is the displacement in range and bearing, denoted *v*.

This process is repeated for each matched landmark.

Step 3: Add new landmarks to the current state

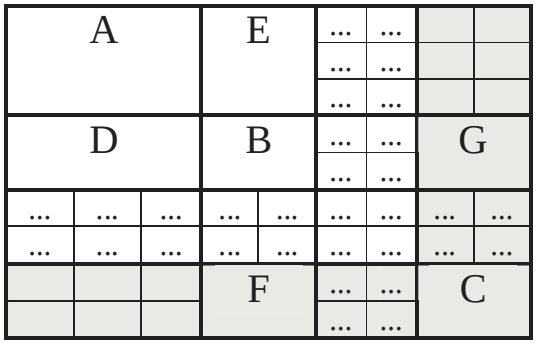
In this step we want to update the state vector X and the covariance matrix P with new landmarks. The purpose is to have more landmarks that can be matched, so the robot has more landmarks that can be matched.

First we add the new landmark to the state vector X

$$X = [X\ x_N\ y_N]^T$$

Also we need to add a new row and column to the covariance matrix, shown in the figure below as the grey area. First we add the covariance for the new landmark in the cell C, also called P^{N+1N+1}, since it is the covariance for the N+1 landmark:

$P^{N+1N+1} = J_{xr}\ P\ J_{xr}^T + J_z R J_z^T$



After this we add the robot – landmark covariance for the new landmark. This corresponds to the upper left corner of the covariance matrix. It is computed as follows:

$$P^{rN+1} = P^{rr}\ J_{xr}^T$$

The landmark – robot covariance is the transposed value of the robot – landmark covariance, corresponding to the lower right corner of the covariance matrix:

$$P^{N+1r} = (P^{rN+1})^T$$

Finally the landmark – landmark covariance needs to be added (the lowest row):

$$P^{N+1i} = J_{xr}(P^{ri})^T$$

Again the landmark – landmark covariance on the other side of the diagonal matrix is the transposed value:

$$P^{iN+1} = (P^{N+1i})^T$$

This completes the last step of the SLAM process. The robot is now ready to move again, observe landmarks, associate landmarks, update the system state using odometry, update the system state using re-observed landmarks and finally add new landmarks.

12. Final remarks

The SLAM presented here is a very basic SLAM. There is much room for improvement, and there are areas that have not even been touched. For example there is the problem of closing the loop. This problem is concerned with the robot returning to a place it has seen before. The robot should recognize this and use the new found information to update the position. Furthermore the robot should update the landmarks found before the robot returned to a known place, propagating the correction back along the path. A system such as ATLAS [2] is concerned with this.

It is also possible to combine this SLAM with an occupation grid, mapping the world in a human-readable format. Besides the obvious use for an occupation grid as a human-readable map, occupation grids can also be used for path planning. A* and D* algorithms can be built upon this. [1]

13. References:

1. Koenig, Likhachev: *Incremental A* (D*)*

2. Bosse, Newman, Leonard, Soika, Feiten, Teller: *An ATLAS framework*

3. Roy: *Foundations of state estimation* (lecture):

4. Zunino: *SLAM in realistic environments*: http://www.nada.kth.se/utbildning/forsk.utb/avhandlingar/lic/020220.pdf
5. Welch, Bishop: *An introduction to the Kalman Filter*:

6. Smith, Self, Cheesman: *Estimating uncertain spatial relationships in robotics*
7. Leonard, Durrant-Whyte: *Mobile robot localization by tracking geometric beacons*:

8. Se, Lowe, Little: Mobile Robot Localization and Mapping using Scale-Invariant Visual Landmarks: http://www.cs.ubc.ca/~se/papers/ijrr02.pdf
9. SICK, industrial sensors: http://www.sick.de
10. Evolution Robotics http://www.evolution.com

14. Appendix A: Coordinate conversion

Conversion from range and bearing to Cartesian coordinates:

$$x = range * Cos(\theta_o)$$

$$y = range * -Sin(\theta_o)$$

Converts an observation from the robots sensors to Cartesian coordinates. Where θ_o is the angle the observation is viewed at and range is the distance to the measurement. To get the coordinates with respect to a world map you must also add the robots angle θ_r relative to the world map:

$$x = range * Cos(\theta_o + \theta_r)$$

$$y = range * -Sin(\theta_o + \theta_r)$$

15. Appendix B: SICK LMS 200 interface code

LMS Interface code:

```
using System;
using SerialPorts;

namespace APULMS
{
	/// <summary>
	/// Summary description for LMS200.
	/// </summary>
	public class LMS200
	{
		private Threader th;
		public SerialPort Port;
		private WithEvents Func;
		private int PortIndex;
		private static byte[] GET_MEASUREMENTS = {0x02, 0x00, 0x02, 0x00, 0x30, 0x01, 0x31, 0x18};
		private static byte[] PCLMS_B9600 =  {0x02,0x00,0x02,0x00,0x20,0x42,0x52,0x08};
		private static byte[] PCLMS_B19200 = {0x02,0x00,0x02,0x00,0x20,0x41,0x51,0x08};
		private static byte[] PCLMS_B38400 = {0x02,0x00,0x02,0x00,0x20,0x40,0x50,0x08};

		public LMS200(Threader t, int PortIndex)
		{
			this.th = t;
			this.PortIndex = PortIndex;

			// Instantiate base class event handlers.
			this.Func = new WithEvents();
			this.Func.Error   = new StrnFunc(this.OnError);
			this.Func.RxChar  = new ByteFunc(this.OnRecvI);
			this.Func.CtsSig  = new BoolFunc(this.OnCts);
			this.Func.DsrSig  = new BoolFunc(this.OnDsr);
			this.Func.RlsdSig = new BoolFunc(this.OnRlsd);
```

```
        this.Func.RingSig = new BoolFunc(this.OnRing);

        // Instantiate the terminal port.
        this.Port = new SerialPort(this.Func);
        this.Port.Cnfg.BaudRate = SerialPorts.LineSpeed.Baud_9600;
        this.Port.Cnfg.Parity = SerialPorts.Parity.None;

        PortControl();
}

/// <summary>
/// Gives one round of measurements
/// </summary>
public void getMeasurements()
{
        SendBuf(GET_MEASUREMENTS);
}
public void setBaud(bool fast)
{
        if (fast)
        {
                SendBuf(PCLMS_B38400);
                Port.Cnfg.BaudRate = SerialPorts.LineSpeed.Baud_38400;
        }
        else
        {
                SendBuf(PCLMS_B19200);
                Port.Cnfg.BaudRate = SerialPorts.LineSpeed.Baud_19200;
        }
}

public void ClosePorts()
{
        PortControl();
}
private void PortControl()
```

```
{
	if(this.Port.IsOpen == false)
	{
		if(this.Port.Open(PortIndex) == false)
		{
			// ERROR
			return;
		}
		else
		{
			// OK
		}
	}
	else
	{
		if(this.Port.IsOpen)
		{
			this.Port.Close();
		}
		//	OK
		this.Port.Signals();
	}
	return;
}

/// <summary>
/// Handles error events.
/// </summary>
internal void OnError(string fault)
{
	//this.Status.Text = fault;
	PortControl();
}

/// <summary>
/// Immediate byte received.
```

```
        /// </summary>
        internal void OnRecvI(byte[] b)
        {
        }

        /// <summary>
        /// Set the modem state displays.
        /// </summary>
        internal void OnCts(bool cts)
        {
              System.Threading.Thread.Sleep(1);
        }

        /// <summary>
        /// Set the modem state displays.
        /// </summary>
        internal void OnDsr(bool dsr)
        {
              System.Threading.Thread.Sleep(1);
Color.Red;
        }

        /// <summary>
        /// Set the modem state displays.
        /// </summary>
        internal void OnRlsd(bool rlsd)
        {
              System.Threading.Thread.Sleep(1);
        }

        /// <summary>
        /// Set the modem state displays.
        /// </summary>
        internal void OnRing(bool ring)
        {
              System.Threading.Thread.Sleep(1);
```

```
		}

		/// <summary>
		/// Transmit a buffer.
		/// </summary>
		private uint SendBuf(byte[] b)
		{
			uint nSent=0;
			if(b.Length > 0)
			{
				nSent = this.Port.Send(b);
				if(nSent != b.Length)
				{
					// ERROR
				}
			}
			return nSent;
		}

	}
}
```

Thread for retrieving the data:

```
using System;
using SerialPorts;
using System.Threading;

namespace APULMS
{
	/// <summary>
	/// Summary description for Threader.
	/// </summary>
	public class Threader
	{
		public byte[] buffer;
		public int bufferSize;
		public int bufferWritePointer;
		public int bufferReadPointer;
		public SerialPort p;
		public bool cont = true;

		public Threader(int BufferSize)
		{
			this.bufferSize = BufferSize;
			buffer = new byte[this.bufferSize];
			ResetBuffer();
		}

		public Threader()
		{
			this.bufferSize = 20000;
			buffer = new byte[this.bufferSize];
			ResetBuffer();
		}

		public void ResetBuffer()
		{
```

```
	bufferWritePointer = 0;
	bufferReadPointer = 0;
	for (int i = 0; i < this.bufferSize; i++)
		buffer[i] = 0x00;
}
private int pwrap(int p)
{
	while (p >= bufferSize)
		p -= bufferSize;
	return (p);
}

public void getData()
{
	byte[] b;
	uint nBytes;
	int count = 0;

	while (true)
	{
		// Get number of bytes
		nBytes = this.p.Recv(out b);
		if(nBytes > 0)
		{
			int i = 0;
			for (; i < nBytes && i < b.Length; i++)
				buffer[pwrap(bufferWritePointer+i)] = b[i];
			bufferWritePointer = pwrap(bufferWritePointer + i);
			// restart no data counter
			count = 0;
		}
		else
		{
			// no data, count up
			count++;
		}
```

```
                        if (count > 100)
                        {
                              // Wait until told to resume
                              Monitor.Wait(this);
                              // Reset counter after wait
                              count = 0;
                        }
                  }
            }
      }
}
```

16. Appendix C: ER1 interface code

ER1 interface code, ApplicationSettings.cs:

```
//------------------------------------------------------------------------------
// <autogenerated>
//     This code was generated_u98 ?y a tool.
//     Runtime Version: 1.0.3705.0
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------------

namespace ER1 {
    using System;
    using System.Data;
    using System.Xml;
    using System.Runtime.Serialization;


    [Serializable()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Diagnostics.DebuggerStepThrough()]
    [System.ComponentModel.ToolboxItem(true)]
    public class ApplicationSettings : DataSet_u123 ?

        private ER1DataTable tableER1;

        public ApplicationSettings()_u123 ?
            this.InitClass();
            System.ComponentModel.CollectionChangeEventHandler schemaChangedHandler = new
System.ComponentModel.CollectionChangeEventHandler(this.SchemaChanged);
            this.Tables.CollectionChanged += schemaChangedHandler;
            this.Relations.CollectionChanged += schemaChangedHandler;
```

```
        }

        protected ApplicationSettings(SerializationInfo info, StreamingContext context) {
            string strSchema = ((string)(info.GetValue("XmlSchema", typeof(string))));
            if ((strSchema != null)) {
                DataSet ds = new DataSet();
                ds.ReadXmlSchema(new XmlTextReader(new System.IO.StringReader(strSchema)));
                if ((ds.Tables["ER1"] != null)) {
                    this.Tables.Add(new ER1DataTable(ds.Tables["ER1"]));
                }
                this.DataSetName = ds.DataSetName;
                this.Prefix = ds.Prefix;
                this.Namespace = ds.Namespace;
                this.Locale = ds.Locale;
                this.CaseSensitive = ds.CaseSensitive;
                this.EnforceConstraints = ds.EnforceConstraints;
                this.Merge(ds, false, System.Data.MissingSchemaAction.Add);
                this.InitVars();
            }
            else {
                this.InitClass();
            }
            this.GetSerializationData(info, context);
            System.ComponentModel.CollectionChangeEventHandler schemaChangedHandler = new
System.ComponentModel.CollectionChangeEventHandler(this.SchemaChanged);
            this.Tables.CollectionChanged += schemaChangedHandler;
            this.Relations.CollectionChanged += schemaChangedHandler;
        }

        [System.ComponentModel.Browsable(false)]

[System.ComponentModel.DesignerSerializationVisibilityAttribute(System.ComponentModel.DesignerSerializationVisibilit
y.Content)]
        public ER1DataTable ER1 {
            get {
                return this.tableER1;
```

```
    }
}

public override DataSet Clone() {
    ApplicationSettings cln = ((ApplicationSettings)(base.Clone()));
    cln.InitVars();
    return cln;
}

protected override bool ShouldSerializeTables() {
    return false;
}

protected override bool ShouldSerializeRelations() {
    return false;
}

protected override void ReadXmlSerializable(XmlReader reader) {
    this.Reset();
    DataSet ds = new DataSet();
    ds.ReadXml(reader);
    if ((ds.Tables["ER1"] != null)) {
        this.Tables.Add(new ER1DataTable(ds.Tables["ER1"]));
    }
    this.DataSetName = ds.DataSetName;
    this.Prefix = ds.Prefix;
    this.Namespace = ds.Namespace;
    this.Locale = ds.Locale;
    this.CaseSensitive = ds.CaseSensitive;
    this.EnforceConstraints = ds.EnforceConstraints;
    this.Merge(ds, false, System.Data.MissingSchemaAction.Add);
    this.InitVars();
}

protected override System.Xml.Schema.XmlSchema GetSchemaSerializable() {
    System.IO.MemoryStream stream = new System.IO.MemoryStream();
```

```
    this.WriteXmlSchema(new XmlTextWriter(stream, null));
    stream.Position = 0;
    return System.Xml.Schema.XmlSchema.Read(new XmlTextReader(stream), null);
}

internal void InitVars() {
    this.tableER1 = ((ER1DataTable)(this.Tables["ER1"]));
    if ((this.tableER1 != null)) {
        this.tableER1.InitVars();
    }
}

private void InitClass() {
    this.DataSetName = "ApplicationSettings";
    this.Prefix = "";
    this.Namespace = "http://tempuri.org/ApplicationSettings.xsd";
    this.Locale = new System.Globalization.CultureInfo("en-US");
    this.CaseSensitive = false;
    this.EnforceConstraints = true;
    this.tableER1 = new ER1DataTable();
    this.Tables.Add(this.tableER1);
}

private bool ShouldSerializeER1() {
    return false;
}

private void SchemaChanged(object sender, System.ComponentModel.CollectionChangeEventArgs e) {
    if ((e.Action == System.ComponentModel.CollectionChangeAction.Remove)) {
        this.InitVars();
    }
}

public delegate void ER1RowChangeEventHandler(object sender, ER1RowChangeEvent e);

[System.Diagnostics.DebuggerStepThrough()]
```

```
public class ER1DataTable : DataTable, System.Collections.IEnumerable {

    private DataColumn columnIP;

    private DataColumn columnPassword;

    private DataColumn columnPort;

    internal ER1DataTable() :
            base("ER1") {
        this.InitClass();
    }

    internal ER1DataTable(DataTable table) :
            base(table.TableName) {
        if ((table.CaseSensitive != table.DataSet.CaseSensitive)) {
            this.CaseSensitive = table.CaseSensitive;
        }
        if ((table.Locale.ToString() != table.DataSet.Locale.ToString())) {
            this.Locale = table.Locale;
        }
        if ((table.Namespace != table.DataSet.Namespace)) {
            this.Namespace = table.Namespace;
        }
        this.Prefix = table.Prefix;
        this.MinimumCapacity = table.MinimumCapacity;
        this.DisplayExpression = table.DisplayExpression;
    }

    [System.ComponentModel.Browsable(false)]
    public int Count {
        get {
            return this.Rows.Count;
        }
    }
```

```
internal DataColumn IPColumn {
    get {
        return this.columnIP;
    }
}

internal DataColumn PasswordColumn {
    get {
        return this.columnPassword;
    }
}

internal DataColumn PortColumn {
    get {
        return this.columnPort;
    }
}

public ER1Row this[int index] {
    get {
        return ((ER1Row)(this.Rows[index]));
    }
}

public event ER1RowChangeEventHandler ER1RowChanged;

public event ER1RowChangeEventHandler ER1RowChanging;

public event ER1RowChangeEventHandler ER1RowDeleted;

public event ER1RowChangeEventHandler ER1RowDeleting;

public void AddER1Row(ER1Row row) {
    this.Rows.Add(row);
}
```

```
        public ER1Row AddER1Row(string IP, string Password, int Port) {
            ER1Row rowER1Row = ((ER1Row)(this.NewRow()));
            rowER1Row.ItemArray = new object[] {
                    IP,
                    Password,
                    Port};
            this.Rows.Add(rowER1Row);
            return rowER1Row;
        }

        public System.Collections.IEnumerator GetEnumerator() {
            return this.Rows.GetEnumerator();
        }

        public override DataTable Clone() {
            ER1DataTable cln = ((ER1DataTable)(base.Clone()));
            cln.InitVars();
            return cln;
        }

        protected override DataTable CreateInstance() {
            return new ER1DataTable();
        }

        internal void InitVars() {
            this.columnIP = this.Columns["IP"];
            this.columnPassword = this.Columns["Password"];
            this.columnPort = this.Columns["Port"];
        }

        private void InitClass() {
            this.columnIP = new DataColumn("IP", typeof(string), null, System.Data.MappingType.Element);
            this.Columns.Add(this.columnIP);
            this.columnPassword = new DataColumn("Password", typeof(string), null,
System.Data.MappingType.Element);
            this.Columns.Add(this.columnPassword);
```

```
        this.columnPort = new DataColumn("Port", typeof(int), null, System.Data.MappingType.Element);
        this.Columns.Add(this.columnPort);
    }

    public ER1Row NewER1Row() {
        return ((ER1Row)(this.NewRow()));
    }

    protected override DataRow NewRowFromBuilder(DataRowBuilder builder) {
        return new ER1Row(builder);
    }

    protected override System.Type GetRowType() {
        return typeof(ER1Row);
    }

    protected override void OnRowChanged(DataRowChangeEventArgs e) {
        base.OnRowChanged(e);
        if ((this.ER1RowChanged != null)) {
            this.ER1RowChanged(this, new ER1RowChangeEvent(((ER1Row)(e.Row)), e.Action));
        }
    }

    protected override void OnRowChanging(DataRowChangeEventArgs e) {
        base.OnRowChanging(e);
        if ((this.ER1RowChanging != null)) {
            this.ER1RowChanging(this, new ER1RowChangeEvent(((ER1Row)(e.Row)), e.Action));
        }
    }

    protected override void OnRowDeleted(DataRowChangeEventArgs e) {
        base.OnRowDeleted(e);
        if ((this.ER1RowDeleted != null)) {
            this.ER1RowDeleted(this, new ER1RowChangeEvent(((ER1Row)(e.Row)), e.Action));
        }
    }
```

```
    protected override void OnRowDeleting(DataRowChangeEventArgs e) {
        base.OnRowDeleting(e);
        if ((this.ER1RowDeleting != null)) {
            this.ER1RowDeleting(this, new ER1RowChangeEvent(((ER1Row)(e.Row)), e.Action));
        }
    }

    public void RemoveER1Row(ER1Row row) {
        this.Rows.Remove(row);
    }
}

[System.Diagnostics.DebuggerStepThrough()]
public class ER1Row : DataRow {

    private ER1DataTable tableER1;

    internal ER1Row(DataRowBuilder rb) :
            base(rb) {
        this.tableER1 = ((ER1DataTable)(this.Table));
    }

    public string IP {
        get {
            try {
                return ((string)(this[this.tableER1.IPColumn]));
            }
            catch (InvalidCastException e) {
                 throw new StrongTypingException("Cannot get value because it is DBNull.", e);
            }
        }
        set {
            this[this.tableER1.IPColumn] = value;
        }
    }
```

```
public string Password {
    get {
        try {
            return ((string)(this[this.tableER1.PasswordColumn]));
        }
        catch (InvalidCastException e) {
            throw new StrongTypingException("Cannot get value because it is DBNull.", e);
        }
    }
    set {
        this[this.tableER1.PasswordColumn] = value;
    }
}

public int Port {
    get {
        try {
            return ((int)(this[this.tableER1.PortColumn]));
        }
        catch (InvalidCastException e) {
            throw new StrongTypingException("Cannot get value because it is DBNull.", e);
        }
    }
    set {
        this[this.tableER1.PortColumn] = value;
    }
}

public bool IsIPNull() {
    return this.IsNull(this.tableER1.IPColumn);
}

public void SetIPNull() {
    this[this.tableER1.IPColumn] = System.Convert.DBNull;
}
```

```
    public bool IsPasswordNull() {
        return this.IsNull(this.tableER1.PasswordColumn);
    }

    public void SetPasswordNull() {
        this[this.tableER1.PasswordColumn] = System.Convert.DBNull;
    }

    public bool IsPortNull() {
        return this.IsNull(this.tableER1.PortColumn);
    }

    public void SetPortNull() {
        this[this.tableER1.PortColumn] = System.Convert.DBNull;
    }
}

[System.Diagnostics.DebuggerStepThrough()]
public class ER1RowChangeEvent : EventArgs {

    private ER1Row eventRow;

    private DataRowAction eventAction;

    public ER1RowChangeEvent(ER1Row row, DataRowAction action) {
        this.eventRow = row;
        this.eventAction = action;
    }

    public ER1Row Row {
        get {
            return this.eventRow;
        }
    }
```

```
            public DataRowAction Action {
                get {
                    return this.eventAction;
                }
            }
        }
    }
}
```

Robot.cs:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Configuration;
using System.Threading;

using ProtoSystems.Networking;

namespace ER1
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>
	public class Robot
	{
		public ProtoSystems.Networking.ScriptingTelnet telnet=new ScriptingTelnet();
		public bool connected=false;
		public Settings settings = new Settings();
		public object syncVar = new object();
		public Mutex lck = new Mutex();
		ER1.Robot.Listener listeningThread;


		public Robot()
		{
			if (this.telnet.Connected==true)
			{
				this.telnet.Disconnect();
			}
			this.telnet.Address = "localhost";
			this.telnet.Port = 9000;
			this.telnet.Timeout = 0;
			try
```

```
        {
                if (this.telnet.Connect()==false)
                {
                        Console.Write("ER1: Failed to Connect (check Robot Control Center is running)\n");
                }
                else
                {
                        this.telnet.SendMessage("login hest");
                        System.Threading.Thread.Sleep(500);
                        this.telnet.ReceivedData();
                }
                //start listening thread
                listeningThread = new Listener(telnet,syncVar);
                Thread tr = new Thread(new ThreadStart(listeningThread.Listening));
                tr.Start();

        }
        catch(Exception ex)
        {
        }
}

public string Command(string cmd)
{
        string reply;

        lock(syncVar)
        {
                Monitor.Pulse(syncVar);
                this.telnet.SendMessage(cmd);
                //Console.Write("Sender sent!\n");
                try
                {
                        Monitor.Wait(syncVar);
                        //Console.Write("Sender no longer waiting\n");
                        reply = listeningThread.receivedData;
```

```
                  return reply;
            }
            catch
            {
                  return "";
            }
      }
}

private string CleanDisplay(string input)
{
      input.Replace("|","");
      input = input.Replace("_(0x _(B","|");
      input = input.Replace("_(0 x_(B","|");
      input = input.Replace("_)0_=_>","");
      input = input.Replace("_[0m_>","");
      input = input.Replace("_7_[7m","[");
      input = input.Replace("_[0m*_8_[7m","]");
      input = input.Replace("_[0m","");
      return input;
}

public class Listener
{
      public ProtoSystems.Networking.ScriptingTelnet telnet;
      public String receivedData;
      public object syncVar;
      public string temp;

      public Listener(ProtoSystems.Networking.ScriptingTelnet telnet, object syncVar)
      {
            this.telnet = telnet;
            this.syncVar = syncVar;
      }

      public void Listening()
```

```
{
      while(true)
      {
            Listen();
            System.Threading.Thread.Sleep(10);
      }
}

public void Listen()
{
      lock(syncVar)
      {
            if (this.telnet.Connected==true)
            {
                  if (this.telnet.ReceivedData()>0)
                  {
                        temp = CleanDisplay(this.telnet.GetReceivedData());
                        if(temp.Length>0)
                        {
                              //Console.Write("Listener got:" + temp +"\n");
                              receivedData = temp;
                              Monitor.Pulse(syncVar);
                              Monitor.Wait(syncVar);
                        }
                  }
            }
      }
}

private string CleanDisplay(string input)
{
      input.Replace("|","");
      input = input.Replace("_(0x _(B","|");
      input = input.Replace("_(0 x_(B","|");
      input = input.Replace("_)0_=_>","");
      input = input.Replace("_[0m_>","");
```

```
                        input = input.Replace("_7_[7m","[");
                        input = input.Replace("_[0m*_8_[7m","]");
                        input = input.Replace("_[0m","");
                        return input;
                    }
                }
            }
        }
```

ScriptingTelnet.cs:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.IO;
using System.Threading ;

namespace ProtoSystems.Networking
{
	/// <summary>
	/// Summary description for clsScriptingTelnet.
	/// </summary>
	public class ScriptingTelnet
	{
		private IPEndPoint iep ;
		private AsyncCallback callbackProc ;
		private string address ;
		private int port ;
		private int timeout;
		private bool connected=false;
		private Socket s ;
		Byte[] m_byBuff = new Byte[32767];
		private string strWorkingData = "";                    // Holds everything received from the server
since our last processing
		private string strFullLog = "";

		public ScriptingTelnet()
		{
		}
		public ScriptingTelnet(string Address, int Port, int CommandTimeout)
		{
			address = Address;
			port = Port;
			timeout = CommandTimeout;
```

```
}
public int Port
{
	get{return this.port;}
	set{this.port = value;}
}
public string Address
{
	get{return this.address;}
	set{this.address = value;}
}
public int Timeout
{
	get{return this.timeout;}
	set{this.timeout = value;}
}
public bool Connected
{
	get{return this.connected;}
}

private void OnRecievedData( IAsyncResult ar )
{
	// Get The connection socket from the callback
	Socket sock = (Socket)ar.AsyncState;

	// Get The data , if any
	int nBytesRec = sock.EndReceive( ar );

	if( nBytesRec > 0 )
	{
		// Decode the received data
		string sRecieved = CleanDisplay(Encoding.ASCII.GetString( m_byBuff, 0, nBytesRec ));

		// Write out the data
		if (sRecieved.IndexOf("[c") != -1) Negotiate(1);
```

```
            if (sRecieved.IndexOf("[6n") != -1) Negotiate(2);

            Console.WriteLine(sRecieved);
            strWorkingData += sRecieved;
            strFullLog += sRecieved;

            // Launch another callback to listen for data
            AsyncCallback recieveData = new AsyncCallback(OnRecievedData);
            sock.BeginReceive( m_byBuff, 0, m_byBuff.Length, SocketFlags.None, recieveData , sock );

      }
      else
      {
            // If no data was recieved then the connection is probably dead
            Console.WriteLine( "Disconnected", sock.RemoteEndPoint );
            sock.Shutdown( SocketShutdown.Both );
            sock.Close();
            //Application.Exit();
      }
}

private void DoSend(string strText)
{
      try
      {

            Byte[] smk = new Byte[strText.Length];
            for ( int i=0; i < strText.Length ; i++)
            {
                  Byte ss = Convert.ToByte(strText[i]);
                  smk[i] = ss ;
            }

            //s.Send(smk,0 , smk.Length , SocketFlags.None);
            //s.Send(strText.ToCharArray(),strText.Length,SocketFlags.None);
            IAsyncResult ar2 = s.BeginSend(smk , 0 , smk.Length , SocketFlags.None , callbackProc , s );
```

```
            s.EndSend(ar2);

      }
      catch(Exception ers)
      {
            //MessageBox.Show("ERROR IN RESPOND OPTIONS");
      }
}

private void Negotiate(int WhichPart)
{
      StringBuilder x;
      string neg;
      if (WhichPart == 1)
      {
            x = new StringBuilder();
            x.Append ((char)27);
            x.Append ((char)91);
            x.Append ((char)63);
            x.Append ((char)49);
            x.Append ((char)59);
            x.Append ((char)50);
            x.Append ((char)99);
            neg =  x.ToString();
      }
      else
      {

            x = new StringBuilder();
            x.Append ((char)27);
            x.Append ((char)91);
            x.Append ((char)50);
            x.Append ((char)52);
            x.Append ((char)59);
            x.Append ((char)56);
            x.Append ((char)48);
```

```
            x.Append ((char)82);
            neg = x.ToString();
      }
      SendMessage(neg,true);
}

private string CleanDisplay(string input)
{

      input = input.Replace("_(0x _(B","|");
      input = input.Replace("_(0 x_(B","|");
      input = input.Replace("_)0_=_>","");
      input = input.Replace("_[0m_>","");
      input = input.Replace("_7_[7m","[");
      input = input.Replace("_[0m*_8_[7m","]");
      input = input.Replace("_[0m","");
      return input;
}



/// <summary>
/// Connects to the telnet server.
/// </summary>
/// <returns>True upon connection, False if connection fails</returns>
public bool Connect()
{
      IPHostEntry IPHost = Dns.Resolve(address);
      string []aliases = IPHost.Aliases;
      IPAddress[] addr = IPHost.AddressList;

      try
      {
            // Try a blocking connection to the server
```

```
                        s                       = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
                        iep                     = new IPEndPoint(addr[0],port);
                        s.Connect(iep) ;

                        // If the connect worked, setup a callback to start listening for incoming data
                        AsyncCallback recieveData = new AsyncCallback( OnRecievedData );
                        s.BeginReceive( m_byBuff, 0, m_byBuff.Length, SocketFlags.None, recieveData , s );

                        // All is good
                        this.connected=true;
                        return true;
                  }
                  catch(Exception eeeee )
                  {
                        // Something failed
                        return false;
                  }

            }


            public void Disconnect()
            {
                  if (s.Connected) s.Close();
            }

            /// <summary>
            /// Waits for a specific string to be found in the stream from the server
            /// </summary>
            /// <param name="DataToWaitFor">The string to wait for</param>
            /// <returns>Always returns 0 once the string has been found</returns>
            public int WaitFor(string DataToWaitFor)
            {
                  // Get the starting time
                  long lngStart = DateTime.Now.AddSeconds(this.timeout).Ticks;
```

```
	long lngCurTime = 0;

	while (strWorkingData.ToLower().IndexOf(DataToWaitFor.ToLower()) == -1)
	{
		// Timeout logic
		lngCurTime = DateTime.Now.Ticks;
		if (lngCurTime > lngStart)
		{
			throw new Exception("Timed Out waiting for : " + DataToWaitFor);
		}
		Thread.Sleep(1);
	}
	strWorkingData = "";
	return 0;
}


/// <summary>
/// Waits for one of several possible strings to be found in the stream from the server
/// </summary>
/// <param name="DataToWaitFor">A delimited list of strings to wait for</param>
/// <param name="BreakCharacters">The character to break the delimited string with</param>
/// <returns>The index (zero based) of the value in the delimited list which was matched</returns>
public int WaitFor(string DataToWaitFor,string BreakCharacter)
{
	// Get the starting time
	long lngStart = DateTime.Now.AddSeconds(this.timeout).Ticks;
	long lngCurTime = 0;

	string[] Breaks = DataToWaitFor.Split(BreakCharacter.ToCharArray());
	int intReturn = -1;

	while (intReturn == -1)
	{
		// Timeout logic
		lngCurTime = DateTime.Now.Ticks;
```

```
                        if (lngCurTime > lngStart)
                        {
                              throw new Exception("Timed Out waiting for : " + DataToWaitFor);
                        }

                        Thread.Sleep(1);
                        for (int i = 0 ; i < Breaks.Length ; i++)
                        {
                              if (strWorkingData.ToLower().IndexOf(Breaks[i].ToLower()) != -1)
                              {
                                    intReturn = i ;
                              }
                        }
                  }
                  return intReturn;

            }


            /// <summary>
            /// Sends a message to the server
            /// </summary>
            /// <param name="Message">The message to send to the server</param>
            /// <param name="SuppressCarriageReturn">True if you do not want to end the message with a carriage
return</param>
            public void SendMessage(string Message, bool SuppressCarriageReturn)
            {
                  strFullLog += "\r\nSENDING DATA ====> " + Message.ToUpper() + "\r\n";
                  Console.WriteLine("SENDING DATA ====> " + Message.ToUpper());

                  if (! SuppressCarriageReturn)
                  {
                        DoSend(Message + "\r");
                  }
                  else
                  {
```

```
            DoSend(Message);
      }
}


/// <summary>
/// Sends a message to the server, automatically appending a carriage return to it
/// </summary>
/// <param name="Message">The message to send to the server</param>
public void SendMessage(string Message)
{
      strFullLog += "\r\nSENDING DATA ====> " + Message.ToUpper() + "\r\n";
      Console.WriteLine("SENDING DATA ====> " + Message.ToUpper());

      DoSend(Message + "\r\n");
}


/// <summary>
/// Waits for a specific string to be found in the stream from the server.
/// Once that string is found, sends a message to the server
/// </summary>
/// <param name="WaitFor">The string to be found in the server stream</param>
/// <param name="Message">The message to send to the server</param>
/// <returns>Returns true once the string has been found, and the message has been sent</returns>
public bool WaitAndSend(string WaitFor,string Message)
{
      this.WaitFor(WaitFor);
      SendMessage(Message);
      return true;
}


/// <summary>
/// Sends a message to the server, and waits until the designated
/// response is received
```

```
/// </summary>
/// <param name="Message">The message to send to the server</param>
/// <param name="WaitFor">The response to wait for</param>
/// <returns>True if the process was successful</returns>
public int SendAndWait(string Message, string WaitFor)
{
      SendMessage(Message);
      this.WaitFor(WaitFor);
      return 0;
}

public int SendAndWait(string Message, string WaitFor, string BreakCharacter)
{
      SendMessage(Message);
      int t = this.WaitFor(WaitFor,BreakCharacter);
      return t;
}


/// <summary>
/// A full log of session activity
/// </summary>
public string SessionLog
{
      get
      {
            return strFullLog;
      }
}


/// <summary>
/// Clears all data in the session log
/// </summary>
public void ClearSessionLog()
{
```

```
            strFullLog = "";
        }


        /// <summary>
        /// Searches for two strings in the session log, and if both are found, returns
        /// all the data between them.
        /// </summary>
        /// <param name="StartingString">The first string to find</param>
        /// <param name="EndingString">The second string to find</param>
        /// <param name="ReturnIfNotFound">The string to be returned if a match is not found</param>
        /// <returns>All the data between the end of the starting string and the beginning of the end
string</returns>
        public string FindStringBetween(string StartingString, string EndingString, string ReturnIfNotFound)
        {
            int intStart;
            int intEnd;

            intStart = strFullLog.ToLower().IndexOf(StartingString.ToLower());
            if (intStart == -1)
            {
                return ReturnIfNotFound;
            }
            intStart += StartingString.Length;

            intEnd = strFullLog.ToLower().IndexOf(EndingString.ToLower(),intStart);

            if (intEnd == -1)
            {
                // The string was not found
                return ReturnIfNotFound;
            }

            // The string was found, let's clean it up and return it
            return strFullLog.Substring(intStart, intEnd-intStart).Trim();
        }
```

```
            public int ReceivedData()
            {
                  return this.strWorkingData.Length;
            }
            public string GetReceivedData()
            {
                  string data=this.strWorkingData;
                  this.strWorkingData="";
                  return data;
            }

      }
}
```

Settings.cs:

```
using System;

namespace ER1
{
      /// <summary>
      /// Summary description for Settings.
      /// </summary>
      public class Settings
      {
            ApplicationSettings settingsDataSet=new ApplicationSettings();
            string fileName=System.IO.Path.GetFullPath(".") + @"\AppSettings.xml";
            private string ip="";
            private string password="";
            private int port;
            public Settings()
            {
                  try
                  {
                        settingsDataSet.ReadXml(fileName);
                  }
```

```
        catch
        {
                settingsDataSet.ER1.AddER1Row("localhost","",9000);
        }
        ip = ((ApplicationSettings.ER1Row)settingsDataSet.ER1.Rows[0]).IP;
        port = ((ApplicationSettings.ER1Row)settingsDataSet.ER1.Rows[0]).Port;
        password = ((ApplicationSettings.ER1Row)settingsDataSet.ER1.Rows[0]).Password;
}
public string Password
{
        get{return password;}
        set
        {
                password=value;
                ((ApplicationSettings.ER1Row)settingsDataSet.ER1.Rows[0]).Password=password;
        }
}
public string IP
{
        get{return ip;}
        set
        {
                ip=value;
                ((ApplicationSettings.ER1Row)settingsDataSet.ER1.Rows[0]).IP=ip;
        }
}
public int Port
{
        get{return port;}
        set{
                port=value;
                ((ApplicationSettings.ER1Row)settingsDataSet.ER1.Rows[0]).Port=port;
        }
}
public void Save()
{
```

```
                this.settingsDataSet.WriteXml(fileName);
            }
        }
}
```

17. Appendix D: Landmark extraction code

Landmarks.cs

```
using System;

namespace APUData
{
      /// <summary>
      /// Summary description for Landmarks.
      /// </summary>
      public class Landmarks
      {
            double conv = Math.PI / 180.0; // Convert to radians
            const int MAXLANDMARKS = 3000;
            const double MAXERROR = 0.5; // if a landmark is within 20 cm of another landmark its the same landmark
            public int MINOBSERVATIONS = 15; // Number of times a landmark must be observed to be recognized as a
landmark
```

```
            const int LIFE = 40;
            const double MAX_RANGE = 1;

            const int MAXTRIALS = 1000; //RANSAC: max times to run algorithm
            const int MAXSAMPLE = 10; //RANSAC: randomly select X points
            const int MINLINEPOINTS = 30; //RANSAC: if less than 40 points left don't bother trying to find
consensus (stop algorithm)
            const double RANSAC_TOLERANCE = 0.05; //RANSAC: if point is within x distance of line its part of line
            const int RANSAC_CONSENSUS = 30; //RANSAC: at least 30 votes required to determine if a line

            double degreesPerScan = 0.5;

            public class landmark
            {
                  public double[] pos; //landmarks (x,y) position relative to map
                  public int id; //the landmarks unique ID
                  public int life; //a life counter used to determine whether to discard a landmark
                  public int totalTimesObserved; //the number of times we have seen landmark
                  public double range; //last observed range to landmark
                  public double bearing; //last observed bearing to landmark

                  //RANSAC: Now store equation of a line
                  public double a;
                  public double b;
```

```
                public double rangeError; //distance from robot position to the wall we are using as a landmark
(to calculate error)
                public double bearingError; //bearing from robot position to the wall we are using as a landmark
(to calculate error)

                public landmark()
                {
                      totalTimesObserved = 0;
                      id = -1;
                      life = LIFE;
                      pos = new double[2];
                      a = -1;
                      b = -1;
                }
                //keep track of bad landmarks?
          }

          landmark[] landmarkDB = new landmark[MAXLANDMARKS];

          int DBSize = 0;

          int[,] IDtoID = new int[MAXLANDMARKS,2];
          int EKFLandmarks = 0;
```

```
public int GetSlamID(int id)
{
	for(int i=0; i<EKFLandmarks; i++)
	{
		if(IDtoID[i, 0] == id)
			return IDtoID[i,1];
	}
	return -1;
}

public int AddSlamID(int landmarkID, int slamID)
{
	IDtoID[EKFLandmarks, 0] = landmarkID;
	IDtoID[EKFLandmarks, 1] = slamID;
	EKFLandmarks++;
	return 0;
}


public Landmarks(double degreesPerScan)
{
	this.degreesPerScan = degreesPerScan;
```

```
	for(int i=0; i<landmarkDB.Length; i++)
	{
		landmarkDB[i] = new landmark();
	}
}

public int RemoveBadLandmarks(double[] laserdata, double[] robotPosition)
{
	double maxrange = 0;
	for(int i=1; i<laserdata.Length-1; i++)
	{
		//distance further away than 8.1m we assume are failed returns
		//we get the laser data with max range
		if (laserdata[i-1] < 8.1)
			if (laserdata[i+1] < 8.1)
				if (laserdata[i] > maxrange)
					maxrange = laserdata[i];
	}

	maxrange = MAX_RANGE;
	double[] Xbounds = new double[4];
	double[] Ybounds = new double[4];
	//get bounds of rectangular box to remove bad landmarks from
```

```
                Xbounds[0] =Math.Cos((1 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[0];
                Ybounds[0] =Math.Sin((1 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[1];
                Xbounds[1] =Xbounds[0]+Math.Cos((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;
                Ybounds[1] =Ybounds[0]+Math.Sin((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;
                Xbounds[2] =Math.Cos((359 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[0];
                Ybounds[2] =Math.Sin((359 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[1];
                Xbounds[3] =Xbounds[2]+Math.Cos((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;
                Ybounds[3] =Ybounds[2]+Math.Sin((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;

                /*
                // the below code starts the box 1 meter in front of robot
                Xbounds[0] =Math.Cos((0 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[0];
                Xbounds[1] =Xbounds[0]+Math.Cos((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;
```

```
                Xbounds[0] =Xbounds[0]+Math.Cos((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180));
//make box start 1 meter ahead of robot
                Ybounds[0] =Math.Sin((0 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[1];
                Ybounds[1] =Ybounds[0]+Math.Sin((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;
                Ybounds[0] =Ybounds[0]+Math.Sin((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180));
//make box start 1 meter ahead of robot

                Xbounds[2] =Math.Cos((360 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[0];
                Xbounds[3] =Xbounds[2]+Math.Cos((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;
                Xbounds[2] =Xbounds[2]+Math.Cos((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180));
//make box start 1 meter ahead of robot
                Ybounds[2] =Math.Sin((360 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange+robotPosition[1];
                Ybounds[3] =Ybounds[2]+Math.Sin((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
maxrange;
                Ybounds[2] =Ybounds[2]+Math.Sin((180 * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180));
//make box start 1 meter ahead of robot
                */
```

```
//now check DB for landmarks that are within this box
//decrease life of all landmarks in box.  If the life reaches zero, remove landmark

double pntx, pnty;
for(int k=0; k<DBSize+1; k++)
{
	pntx = landmarkDB[k].pos[0];
	pnty = landmarkDB[k].pos[1];
	int i = 0;
	int j = 0;
	bool inRectangle;
	//if(robotPosition[2]>0 && robotPosition[2]<180)
	if(robotPosition[0]<0 || robotPosition[1]<0)
		inRectangle = false;
	else
		inRectangle = true;

	for(i=0; i < 4; i++)
	{
		if (((((Ybounds[i] <= pnty) && (pnty < Ybounds[j])) || ((Ybounds[j] <= pnty) && (pnty
< Ybounds[i]))) && (pntx < (Xbounds[j] - Xbounds[i]) * (pnty - Ybounds[i]) / (Ybounds[j] - Ybounds[i]) +
Xbounds[i])))
		{
			if(inRectangle==false)
```

```
                        inRectangle=true;
                  else
                        inRectangle=false;
            }
            j = i;
            i = i + 1;
      }
      if(inRectangle)
      {
            //in rectangle so decrease life and maybe remove
            landmarkDB[k].life--;
            if(landmarkDB[k].life<=0)
            {
                  for(int kk=k; kk<DBSize; kk++)  //remove landmark by copying down rest of DB
                  {
                        if(kk==DBSize-1)
                        {
                              landmarkDB[kk].pos[0] = landmarkDB[kk+1].pos[0];
                              landmarkDB[kk].pos[1] = landmarkDB[kk+1].pos[1];
                              landmarkDB[kk].life = landmarkDB[kk+1].life;
                              landmarkDB[kk].id = landmarkDB[kk+1].id;
                              landmarkDB[kk].totalTimesObserved =
landmarkDB[kk+1].totalTimesObserved;
                        }
```

```
                                    else
                                    {
                                          landmarkDB[kk+1].id--;
                                          landmarkDB[kk].pos[0] = landmarkDB[kk+1].pos[0];
                                          landmarkDB[kk].pos[1] = landmarkDB[kk+1].pos[1];
                                          landmarkDB[kk].life = landmarkDB[kk+1].life;
                                          landmarkDB[kk].id = landmarkDB[kk+1].id;
                                          landmarkDB[kk].totalTimesObserved =
landmarkDB[kk+1].totalTimesObserved;
                                    }
                              }
                              DBSize--;
                        }
                  }
            }
            return 0;
      }

      public landmark[] UpdateAndAddLineLandmarks(landmark[] extractedLandmarks)
      {

            //returns the found landmarks
            landmark[] tempLandmarks = new landmark[extractedLandmarks.Length];
```

```
	for(int i=0; i<extractedLandmarks.Length; i++)
	tempLandmarks[i] = UpdateLandmark(extractedLandmarks[i]);

	return tempLandmarks;
}

/*
public landmark[] UpdateAndAddLandmarks(double[] laserdata, double[] robotPosition)
{

	//have a large array to keep track of found landmarks
	landmark[] tempLandmarks = new landmark[400];
	for(int i=0; i<tempLandmarks.Length;i++)
		tempLandmarks[i]= new landmark();

	int totalFound = 0;

	double val = laserdata[0];
	for (int i = 1; i < laserdata.Length - 1; i++)
	{
		// Check for error measurement in laser data
		if (laserdata[i-1] < 8.1)
			if (laserdata[i+1] < 8.1)
				if ((laserdata[i-1] - laserdata[i]) + (laserdata[i+1] - laserdata[i]) > 0.5)
```

```
                        tempLandmarks[i] = UpdateLandmark(laserdata[i], i, robotPosition);
                  else
                        if((laserdata[i-1] - laserdata[i]) > 0.3)
                        tempLandmarks[i] = UpdateLandmark(laserdata[i], i, robotPosition);
                  else if (laserdata[i+1] < 8.1)
                        if((laserdata[i+1] - laserdata[i]) > 0.3)
                              tempLandmarks[i] = UpdateLandmark(laserdata[i], i, robotPosition);
}
//get total found landmarks so you can return array of correct dimensions
for(int i=0; i<tempLandmarks.Length;i++)
      if(((int)tempLandmarks[i].id) !=-1)
            totalFound++;

//now return found landmarks in an array of correct dimensions
landmark[] foundLandmarks = new landmark[totalFound];

//copy landmarks into array of correct dimensions
int j = 0;
for(int i=0; i<((landmark[])tempLandmarks).Length;i++)
      if(((landmark)tempLandmarks[i]).id !=-1)
      {
            foundLandmarks[j] = (landmark)tempLandmarks[i];
            j++;
      }
```

```
                return foundLandmarks;
            }
            */

            public landmark[] UpdateAndAddLandmarksUsingEKFResults(bool[] matched, int[] id, double[] ranges,
double[] bearings, double[] robotPosition)
            {
                landmark[] foundLandmarks = new landmark[matched.Length];
                for(int i=0; i< matched.Length; i++)
                {
                    foundLandmarks[i] = UpdateLandmark(matched[i], id[i], ranges[i], bearings[i],
robotPosition);
                }
                return foundLandmarks;
            }

            private landmark UpdateLandmark(bool matched, int id, double distance, double readingNo, double[]
robotPosition)
            {
                landmark lm;
                if(matched)
                {
                    //EKF matched landmark so increase times landmark has been observed
```

```
                        landmarkDB[id].totalTimesObserved++;
                        lm = landmarkDB[id];
                }
                else
                {
                        //EKF failed to match landmark so add it to DB as new landmark
                        lm = new landmark();

                        //convert landmark to map coordinate
                        lm.pos[0] =Math.Cos((readingNo * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
distance;
                        lm.pos[1] =Math.Sin((readingNo * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) *
distance;

                        lm.pos[0]+=robotPosition[0]; //add robot position
                        lm.pos[1]+=robotPosition[1]; //add robot position

                        lm.bearing = readingNo;
                        lm.range = distance;

                        id = AddToDB(lm);

                        lm.id = id;
                }
```

```
	//return landmarks
	return lm;
}

private landmark UpdateLandmark(landmark lm)
{
	//try to do data-association on landmark.
	int id = GetAssociation(lm);

	//if we failed to associate landmark, then add it to DB
	if(id==-1)
		id = AddToDB(lm);

	lm.id = id;
	//return landmarks
	return lm;
}

public int UpdateLineLandmark(landmark lm)
{
	//try to do data-association on landmark.
	int id = GetAssociation(lm);

	//if we failed to associate landmark, then add it to DB
```

```
	if(id==-1)
		id = AddToDB(lm);
	return id;
}

public landmark[] ExtractLineLandmarks(double[] laserdata, double[] robotPosition)
{
	//two arrays corresponding to found lines
	double[] la = new double[100];
	double[] lb = new double[100];
	int totalLines = 0;

	//array of laser data points corresponding to the seen lines
	int[] linepoints = new int[laserdata.Length];
	int totalLinepoints = 0;

	//have a large array to keep track of found landmarks
	landmark[] tempLandmarks = new landmark[400];
	for(int i=0; i<tempLandmarks.Length;i++)
		tempLandmarks[i]= new landmark();

	int totalFound = 0;

	double val = laserdata[0];
```

```
double lastreading = laserdata[2];
double lastlastreading = laserdata[2];

/*
//removes worst outliers (points which for sure aren't on any lines)
for (int i = 2; i < laserdata.Length - 1; i++)
{
      // Check for error measurement in laser data
      if (laserdata[i] < 8.1)
            if(Math.Abs(laserdata[i]-lastreading)+Math.Abs(lastreading-lastlastreading) < 0.2)
            {
                  linepoints[totalLinepoints] = i;
                  totalLinepoints++;
                  //tempLandmarks[i] = GetLandmark(laserdata[i], i, robotPosition);
                  lastreading = laserdata[i];
                  lastreading = laserdata[i-1];
            }
            else
            {
                  lastreading = laserdata[i];
                  lastlastreading = laserdata[i-1];
            }
}
```

```
*/
//FIXME - OR RATHER REMOVE ME SOMEHOW...
for (int i = 0; i < laserdata.Length - 1; i++)
{
                  linepoints[totalLinepoints] = i;
                  totalLinepoints++;
}

#region RANSAC
//RANSAC ALGORITHM
int noTrials = 0;
Random rnd = new Random();
while(noTrials<MAXTRIALS && totalLinepoints > MINLINEPOINTS)
{
      int[] rndSelectedPoints = new int[MAXSAMPLE];
      int temp = 0;
      bool newpoint;

      //– Randomly select a subset S1 of n data points and
      //compute the model M1
      //Initial version chooses entirely randomly.  Now choose
      //one point randomly and then sample from neighbours within some defined
      //radius
      int centerPoint = rnd.Next(MAXSAMPLE, totalLinepoints-1);
```

```
rndSelectedPoints[0] = centerPoint;

for(int i = 1; i<MAXSAMPLE; i++)
{
	newpoint = false;
	while(!newpoint)
	{
		temp = centerPoint + (rnd.Next(2)-1)*rnd.Next(0, MAXSAMPLE);
		for(int j = 0; j<i; j++)
		{
			if(rndSelectedPoints[j] == temp)
				break;  //point has already been selected
			if(j>=i-1)
				newpoint = true;  //point has not already been selected
		}
	}
	rndSelectedPoints[i] = temp;
}

//compute model M1
double a=0;
double b=0;
//y = a+ bx
```

```
                        LeastSquaresLineEstimate(laserdata, robotPosition, rndSelectedPoints, MAXSAMPLE, ref a, ref
b);

                        //– Determine the consensus set S1* of points is P
                        //compatible with M1 (within some error tolerance)
                        int[] consensusPoints = new int[laserdata.Length];
                        int totalConsensusPoints = 0;

                        int[] newLinePoints = new int[laserdata.Length];
                        int totalNewLinePoints = 0;

                        double x = 0, y =0;
                        double d = 0;
                        for(int i=0; i<totalLinepoints; i++)
                        {
                              //convert ranges and bearing to coordinates
                              x =(Math.Cos((linepoints[i] * degreesPerScan * conv) + robotPosition[2]*conv) *
laserdata[linepoints[i]])+ robotPosition[0];
                              y =(Math.Sin((linepoints[i] * degreesPerScan * conv) + robotPosition[2]*conv) *
laserdata[linepoints[i]])+ robotPosition[1];

                              //x =(Math.Cos((linepoints[i] * degreesPerScan * conv)) *
laserdata[linepoints[i]]);//+robotPosition[0];
```

```
                        //y =(Math.Sin((linepoints[i] * degreesPerScan * conv)) *
laserdata[linepoints[i]]);//+robotPosition[1];


                        d = DistanceToLine(x, y, a, b);

                        if (d<RANSAC_TOLERANCE)
                        {
                              //add points which are close to line
                              consensusPoints[totalConsensusPoints] = linepoints[i];
                              totalConsensusPoints++;
                        }
                        else
                        {
                              //add points which are not close to line
                              newLinePoints[totalNewLinePoints] = linepoints[i];
                              totalNewLinePoints++;
                        }

                  }


                  //– If #(S1*) > t, use S1* to compute (maybe using least
                  //squares) a new model M1*
```

```
                        if(totalConsensusPoints>RANSAC_CONSENSUS)
                        {
                              //Calculate updated line equation based on consensus points
                              LeastSquaresLineEstimate(laserdata, robotPosition, consensusPoints,
totalConsensusPoints, ref a, ref b);
                              //for now add points associated to line as landmarks to see results
                              for(int i=0; i<totalConsensusPoints; i++)
                              {
                                    //tempLandmarks[consensusPoints[i]] = GetLandmark(laserdata[consensusPoints[i]],
consensusPoints[i], robotPosition);
                                    //Remove points that have now been associated to this line
                                    newLinePoints.CopyTo(linepoints, 0);
                                    totalLinepoints = totalNewLinePoints;
                              }
                              //add line to found lines
                              la[totalLines] = a;
                              lb[totalLines] = b;
                              totalLines++;

                              //restart search since we found a line
                              //noTrials = MAXTRIALS;  //when maxtrials = debugging
                              noTrials = 0;
                        }
                        else
```

```
                        //DEBUG add point that we chose as middle value
                        //tempLandmarks[centerPoint] = GetLandmark(laserdata[centerPoint], centerPoint,
robotPosition);


                  //– If #(S1*) < t, randomly select another subset S2 and
                  //repeat

                  //– If, after some predetermined number of trials there is
                  //no consensus set with t points, return with failure
                  noTrials++;
            }

            #endregion

            //for each line we found:
            //calculate the point on line closest to origin (0,0)
            //add this point as a landmark
            for(int i=0; i < totalLines; i++)
            {
                  tempLandmarks[i] = GetLineLandmark(la[i], lb[i], robotPosition);
                  //tempLandmarks[i+1] = GetLine(la[i], lb[i]);
            }
```

```
                //for debug add origin as landmark
                //tempLandmarks[totalLines+1] = GetOrigin();

                //tempLandmarks[i] = GetLandmark(laserdata[i], i, robotPosition);

                //now return found landmarks in an array of correct dimensions
                landmark[] foundLandmarks = new landmark[totalLines];

                //copy landmarks into array of correct dimensions
                for(int i=0; i<foundLandmarks.Length;i++)
                      {
                            foundLandmarks[i] = (landmark)tempLandmarks[i];
                      }

                return foundLandmarks;
          }

          private void LeastSquaresLineEstimate(double[] laserdata, double[] robotPosition, int[] SelectedPoints,
int arraySize, ref double a, ref double b)
          {
                double y; //y coordinate
                double x; //x coordinate
                double sumY=0; //sum of y coordinates
                double sumYY=0; //sum of y^2 for each coordinate
```

```
double sumX=0; //sum of x coordinates
double sumXX=0; //sum of x^2 for each coordinate
double sumYX=0; //sum of y*x for each point

//DEBUG
/*
double[] testX = {0, 1};
double[] testY = {1, 1};
for(int i = 0; i < 2; i++)
{
      //convert ranges and bearing to coordinates
      x = testX[i];
      y = testY[i];

      sumY  += y;
      sumYY += Math.Pow(y,2);
      sumX  += x;
      sumXX += Math.Pow(x,2);
      sumYX += y*x;
}

a = (sumY*sumXX-sumX*sumYX)/(testX.Length*sumXX-Math.Pow(sumX, 2));
b = (testX.Length*sumYX-sumX*sumY)/(testX.Length*sumXX-Math.Pow(sumX, 2));
*/
```

```
                for(int i = 0; i < arraySize; i++)
                {
                      //convert ranges and bearing to coordinates
                      x =(Math.Cos((SelectedPoints[i] * degreesPerScan * conv) + robotPosition[2]*conv) *
laserdata[SelectedPoints[i]])+robotPosition[0];
                      y =(Math.Sin((SelectedPoints[i] * degreesPerScan * conv) + robotPosition[2]*conv) *
laserdata[SelectedPoints[i]])+robotPosition[1];

                      //x =(Math.Cos((rndSelectedPoints[i] * degreesPerScan * conv)) *
laserdata[rndSelectedPoints[i]]);//+robotPosition[0];
                      //y =(Math.Sin((rndSelectedPoints[i] * degreesPerScan * conv)) *
laserdata[rndSelectedPoints[i]]);//+robotPosition[1];

                      sumY  += y;
                      sumYY += Math.Pow(y,2);
                      sumX  += x;
                      sumXX += Math.Pow(x,2);
                      sumYX += y*x;
                }

                b = (sumY*sumXX-sumX*sumYX)/(arraySize*sumXX-Math.Pow(sumX, 2));
                a = (arraySize*sumYX-sumX*sumY)/(arraySize*sumXX-Math.Pow(sumX, 2));
          }
```

```
private double DistanceToLine(double x,double y,double a,double b)
{
	/*
	//y = ax + b
	//0 = ax + b - y
	double d = Math.Abs((a*x - y + b)/(Math.Sqrt(Math.Pow(a,2)+ Math.Pow(b,2))));
	*/

	//our goal is to calculate point on line closest to x,y
	//then use this to calculate distance between them.

	//calculate line perpendicular to input line. a*ao = -1
	double ao = -1.0/a;
	//y = aox + bo => bo = y - aox
	double bo = y - ao*x;

	//get intersection between y = ax + b and y = aox + bo
	//so aox + bo = ax + b => aox - ax = b - bo => x = (b - bo)/(ao - a), y = ao*(b - bo)/(ao - a) +
bo


	double px = (b - bo)/(ao - a);
	double py = ((ao*(b - bo))/(ao - a)) + bo;
```

```
        return Distance(x, y, px, py);
}

public landmark[] ExtractSpikeLandmarks(double[] laserdata, double[] robotPosition)
{

        //have a large array to keep track of found landmarks
        landmark[] tempLandmarks = new landmark[400];
        for(int i=0; i<tempLandmarks.Length;i++)
                tempLandmarks[i]= new landmark();

        int totalFound = 0;

        double val = laserdata[0];
        for (int i = 1; i < laserdata.Length - 1; i++)
        {
                // Check for error measurement in laser data
                if (laserdata[i-1] < 8.1)
                        if (laserdata[i+1] < 8.1)
                                if ((laserdata[i-1] - laserdata[i]) + (laserdata[i+1] - laserdata[i]) > 0.5)
                                        tempLandmarks[i] = GetLandmark(laserdata[i], i, robotPosition);
                                else
                                        if((laserdata[i-1] - laserdata[i]) > 0.3)
```

```
                        tempLandmarks[i] = GetLandmark(laserdata[i], i, robotPosition);
                  else if (laserdata[i+1] < 8.1)
                        if((laserdata[i+1] - laserdata[i]) > 0.3)
                              tempLandmarks[i] = GetLandmark(laserdata[i], i, robotPosition);
      }
      //get total found landmarks so you can return array of correct dimensions
      for(int i=0; i<tempLandmarks.Length;i++)
            if(((int)tempLandmarks[i].id) !=-1)
                  totalFound++;

      //now return found landmarks in an array of correct dimensions
      landmark[] foundLandmarks = new landmark[totalFound];

      //copy landmarks into array of correct dimensions
      int j = 0;
      for(int i=0; i<((landmark[])tempLandmarks).Length;i++)
            if(((landmark)tempLandmarks[i]).id !=-1)
            {
                  foundLandmarks[j] = (landmark)tempLandmarks[i];
                  j++;
            }

      return foundLandmarks;
}
```

```
private landmark GetLandmark(double range, int readingNo, double[] robotPosition)
{
      landmark lm = new landmark();

      //convert landmark to map coordinate
      lm.pos[0] =Math.Cos((readingNo * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) * range;
      lm.pos[1] =Math.Sin((readingNo * degreesPerScan * conv)+(robotPosition[2]*Math.PI/180)) * range;

      lm.pos[0]+=robotPosition[0]; //add robot position
      lm.pos[1]+=robotPosition[1]; //add robot position

      lm.range = range;
      lm.bearing = readingNo;

      //associate landmark to closest landmark.
      int id = -1;
      int totalTimesObserved =0;
      GetClosestAssociation(lm, ref id, ref totalTimesObserved);

      lm.id = id;
      //return landmarks
      return lm;
}
```

```
private landmark GetLineLandmark(double a, double b, double[] robotPosition)
{
      //our goal is to calculate point on line closest to origin (0,0)


      //calculate line perpendicular to input line. a*ao = -1
      double ao = -1.0/a;

      //landmark position
      double x = b/(ao - a);
      double y = (ao*b)/(ao - a);

      double range = Math.Sqrt(Math.Pow(x-robotPosition[0],2) + Math.Pow(y-robotPosition[1],2));
      double bearing = Math.Atan((y-robotPosition[1])/(x-robotPosition[0])) - robotPosition[2];

      //now do same calculation but get point on wall closest to robot instead

      //y = aox + bo => bo = y - aox
      double bo = robotPosition[1] - ao*robotPosition[0];
      //get intersection between y = ax + b and y = aox + bo
      //so aox + bo = ax + b => aox - ax = b - bo => x = (b - bo)/(ao - a), y = ao*(b - bo)/(ao - a) +
bo
      double px = (b - bo)/(ao - a);
```

```
                double py = ((ao*(b - bo))/(ao - a)) + bo;

                double rangeError = Distance(robotPosition[0], robotPosition[1], px, py);
                double bearingError = Math.Atan((py - robotPosition[1])/(px - robotPosition[0])) -
robotPosition[2]; //do you subtract or add robot bearing? I am not sure!

                landmark lm = new landmark();

                //convert landmark to map coordinate
                lm.pos[0] =x;
                lm.pos[1] =y;

                lm.range = range;
                lm.bearing = bearing;

                lm.a = a;
                lm.b = b;

                lm.rangeError = rangeError;
                lm.bearingError = bearingError;

                //associate landmark to closest landmark.
                int id = 0;
                int totalTimesObserved = 0;
```

```
        GetClosestAssociation(lm, ref id, ref totalTimesObserved);

        lm.id = id;
        lm.totalTimesObserved = totalTimesObserved;
        //return landmarks
        return lm;
}

private landmark GetLine(double a, double b)
{

        //our goal is to calculate point on line closest to origin (0,0)


        //calculate line perpendicular to input line. a*ao = -1
        double ao = -1.0/a;

        //get intersection between y = ax + b and y = aox
        //so aox = ax + b => aox - ax = b => x = b/(ao - a), y = ao*b/(ao - a)

        double x = b/(ao - a);
        double y = (ao*b)/(ao - a);
```

```
        landmark lm = new landmark();

        //convert landmark to map coordinate
        lm.pos[0] =x;
        lm.pos[1] =y;

        lm.range = -1;
        lm.bearing = -1;
        lm.a = a;
        lm.b = b;


        //associate landmark to closest landmark.
        int id = -1;
        int totalTimesObserved = 0;
        GetClosestAssociation(lm, ref id, ref totalTimesObserved);

        lm.id = id;
        //return landmarks
        return lm;
}

private landmark GetOrigin()
{
```

```
        landmark lm = new landmark();

        //convert landmark to map coordinate
        lm.pos[0] =0;
        lm.pos[1] =0;


        lm.range = -1;
        lm.bearing = -1;


        //associate landmark to closest landmark.
        int id = -1;
        int totalTimesObserved = 0;
        GetClosestAssociation(lm, ref id, ref totalTimesObserved);

        lm.id = id;
        //return landmarks
        return lm;
}

private void GetClosestAssociation(landmark lm, ref int id, ref int totalTimesObserved)
```

```
{   //given a landmark we find the closest landmark in DB
      int closestLandmark = 0;
      double temp;
      double leastDistance = 99999; //99999m is least initial distance, its big
      for(int i=0; i<DBSize; i++)
      {
            //only associate to landmarks we have seen more than MINOBSERVATIONS times
            if(landmarkDB[i].totalTimesObserved>MINOBSERVATIONS)
            {
                  temp = Distance(lm, landmarkDB[i]);
                  if(temp<leastDistance)
                  {
                        leastDistance = temp;
                        closestLandmark = landmarkDB[i].id;
                  }
            }
      }
      if(leastDistance == 99999)
            id = -1;
      else
      {
            id = landmarkDB[closestLandmark].id;
            totalTimesObserved = landmarkDB[closestLandmark].totalTimesObserved;
      }
```

```
}


private int GetAssociation(landmark lm)
{   //this method needs to be improved so we use innovation as a validation gate
      //currently we just check if a landmark is within some predetermined distance of a landmark in DB
      for(int i=0; i<DBSize; i++)
      {
            if(Distance(lm, landmarkDB[i])<MAXERROR  && ((landmark)landmarkDB[i]).id != -1)
            {
                  ((landmark)landmarkDB[i]).life=LIFE;  //landmark seen so reset its life counter
                  ((landmark)landmarkDB[i]).totalTimesObserved++; //increase number of times we seen
landmark
                  ((landmark)landmarkDB[i]).bearing = lm.bearing; //set last bearing seen at
                  ((landmark)landmarkDB[i]).range = lm.range; //set last range seen at
                  return ((landmark)landmarkDB[i]).id;
            }
      }
      return -1;
}


public landmark[] RemoveDoubles(landmark[] extractedLandmarks)
{
```

```
int uniquelmrks = 0;
double leastDistance = 99999;
double temp;
landmark[] uniqueLandmarks = new landmark[100];

for(int i=0; i<extractedLandmarks.Length; i++)
{
      //remove landmarks that didn't get associated and also pass
      //landmarks through our temporary landmark validation gate
      if(extractedLandmarks[i].id != -1  && GetAssociation(extractedLandmarks[i]) != -1)
      {
            leastDistance = 99999;

            //remove doubles in extractedLandmarks
            //if two observations match same landmark, take closest landmark
            for(int j=0; j<extractedLandmarks.Length; j++)
            {
                  if(extractedLandmarks[i].id == extractedLandmarks[j].id)
                  {
                        if (j < i)
                              break;
                        temp = Distance(extractedLandmarks[j],
landmarkDB[extractedLandmarks[j].id]);
                        if(temp<leastDistance)
```

```
                                    {
                                          leastDistance = temp;
                                          uniqueLandmarks[uniquelmrks] = extractedLandmarks[j];
                                    }
                              }
                        }
                  }
                  if (leastDistance != 99999)
                        uniquelmrks++;
            }

            //copy landmarks over into an array of correct dimensions
            extractedLandmarks = new landmark[uniquelmrks];
            for(int i=0; i<uniquelmrks; i++)
            {
                  extractedLandmarks[i] = uniqueLandmarks[i];
            }

            return extractedLandmarks;
      }

      public int AlignLandmarkData(landmark[] extractedLandmarks,ref bool[] matched, ref int[] id, ref
double[] ranges, ref double[] bearings, ref double[,] lmrks, ref double[,] exlmrks)
      {
```

```
int uniquelmrks = 0;
double leastDistance = 99999;
double temp;
landmark[] uniqueLandmarks = new landmark[100];

for(int i=0; i<extractedLandmarks.Length; i++)
{
      if(extractedLandmarks[i].id != -1)
      {
            leastDistance = 99999;

            //remove doubles in extractedLandmarks
            //if two observations match same landmark, take closest landmark
            for(int j=0; j<extractedLandmarks.Length; j++)
            {
                  if(extractedLandmarks[i].id == extractedLandmarks[j].id)
                  {
                        if (j < i)
                              break;
                        temp = Distance(extractedLandmarks[j],
landmarkDB[extractedLandmarks[j].id]);
                        if(temp<leastDistance)
                        {
                              leastDistance = temp;
```

```
                              uniqueLandmarks[uniquelmrks] = extractedLandmarks[j];
                        }
                  }
            }
      }
      if (leastDistance != 99999)
            uniquelmrks++;
}
matched = new bool[uniquelmrks];
id = new int[uniquelmrks];
ranges = new double[uniquelmrks];
bearings = new double[uniquelmrks];
lmrks = new double[uniquelmrks,2];
exlmrks = new double[uniquelmrks,2];
for(int i=0; i<uniquelmrks; i++)
{
      matched[i] = true;
      id[i] = uniqueLandmarks[i].id;
      ranges[i] = uniqueLandmarks[i].range;
      bearings[i] = uniqueLandmarks[i].bearing;
      lmrks[i,0] = landmarkDB[uniqueLandmarks[i].id].pos[0];
      lmrks[i,1] = landmarkDB[uniqueLandmarks[i].id].pos[1];
      exlmrks[i,0] = uniqueLandmarks[i].pos[0];
      exlmrks[i,1] = uniqueLandmarks[i].pos[1];
```

```
		}
		return 0;
	}

	public int AddToDB(landmark lm)
	{
		if(DBSize+1 < landmarkDB.Length)
		{
			//for(int i=0; i<DBSize+1; i++)
			//{
			//if(((landmark)landmarkDB[i]).id != i)//if(((landmark)landmarkDB[i]).id == -1  ||
((landmark)landmarkDB[i]).life <= 0)
			//{
			((landmark)landmarkDB[DBSize]).pos[0] = lm.pos[0];  //set landmark coordinates
			((landmark)landmarkDB[DBSize]).pos[1] = lm.pos[1];  //set landmark coordinates
			((landmark)landmarkDB[DBSize]).life = LIFE; //set landmark life counter
			((landmark)landmarkDB[DBSize]).id = DBSize; //set landmark id
			((landmark)landmarkDB[DBSize]).totalTimesObserved = 1; //initialise number of times we've
seen landmark
			((landmark)landmarkDB[DBSize]).bearing = lm.bearing; //set last bearing was seen at
			((landmark)landmarkDB[DBSize]).range = lm.range; //set last range was seen at
			((landmark)landmarkDB[DBSize]).a = lm.a; //store landmarks wall equation
			((landmark)landmarkDB[DBSize]).b= lm.b; //store landmarks wall equation
			DBSize++;
```

```
            return (DBSize-1);
      }
      return -1;
}

public int GetDBSize()
{
      return DBSize;
}
public landmark[] GetDB()
{
            landmark[] temp = new landmark[DBSize];
            for(int i=0; i<DBSize; i++)
            {
                  temp[i] = landmarkDB[i];
            }
            return temp;
}

private double Distance(double x1, double y1, double x2, double y2)
{
      return Math.Sqrt(Math.Pow(x1-x2,2)+Math.Pow(y1-y2, 2));
}
```

```
        private double Distance(landmark lm1, landmark lm2)
        {
            return Math.Sqrt(Math.Pow(lm1.pos[0]-lm2.pos[0],2)+Math.Pow(lm1.pos[1]-lm2.pos[1], 2));
        }

    }
}
```