Self-Contained Map Based Navigation in Autonomous Robotic Units

Oddbjørn Kvalsund

December 10, 2008

Abstract

Autonomous robotic units such as reconnaissance robots are dependent on reliable and precise sources of navigation data. In some circumstances the positioning solutions widely available today, GPS and commercial IPS-solutions, are not enough to secure reliable positioning data due to their sensitivity to electromagnetic- and radio-interference. This thesis proposes a set of algorithms and techniques that can be used as a part of a standalone position-recognition system that provides another level of redundancy in such appliances.

Preface

The work presented in this thesis signifies the end result of a two year masters degree in computer science attained at Østfold University College, Norway. Year one of this programme was attained at the Department of Electrical and Electronic Engineering at Newcastle University, England. The first part of this thesis was written during the spring of 2006 and the final chapters were written during the summer and autumn months of 2008.

Autonomous robotic units and safety-oriented systems development has been the research field of professor Rune Winther for a number of years. This paper combines these two topics by describing methods for safe self-contained robotic navigation through a semi-charted two-dimentional terrain.

I wish to thank prof. Rune Winther for his knowledgeable and supportive counseling through the completion of this thesis.

Oslo, December 10th 2008

Oddbjørn Kvalsund

Contents

List of Figures

List of Algorithms

Prerequisites

Due to the time and space constraints of this thesis, every subject mentioned can not be covered in full detail. Consequently it is assumed that the reader is fluent in procedural and object-oriented computer programming, has a firm understanding of fundamental algorithms, algorithm analysis using big O notation, the most common data structures and that he or she understands basic geometric maths.

Chapter 1

Introduction

The purpose of this thesis is to find the best possible set of techniques and algorithms for enabling an autonomous robotic unit to navigate within a semi-charted two-dimensional terrain using only on-board maps, sensors and computing power. Such a system can provide yet another safety level of navigational redundancy when used in conjunction with global or indoor positioning systems. It can also be invaluable when the robot is used as a reconnaissance unit for charting a semi-charted area in greater detail.

The specific test-case unit used throughout the development of this thesis is the ActivMedia Robotics P3-AT robot quipped with sixteen range sonars, one 180 degree laser range device and an onboard computer running Microsoft Windows XP. The techniques described herein are not limited to this specific robot, but access to a semi-powerful computer and a high precision range device forms the basis of the methods described.

1.1 Problem Description

For a robot to be able to successfully navigate from point A to point B in a semi-charted area, it needs the following information:

- The robot's current position and heading.
- A map of the area of which it is to navigate, containing at least one travelable path from point A to point B.

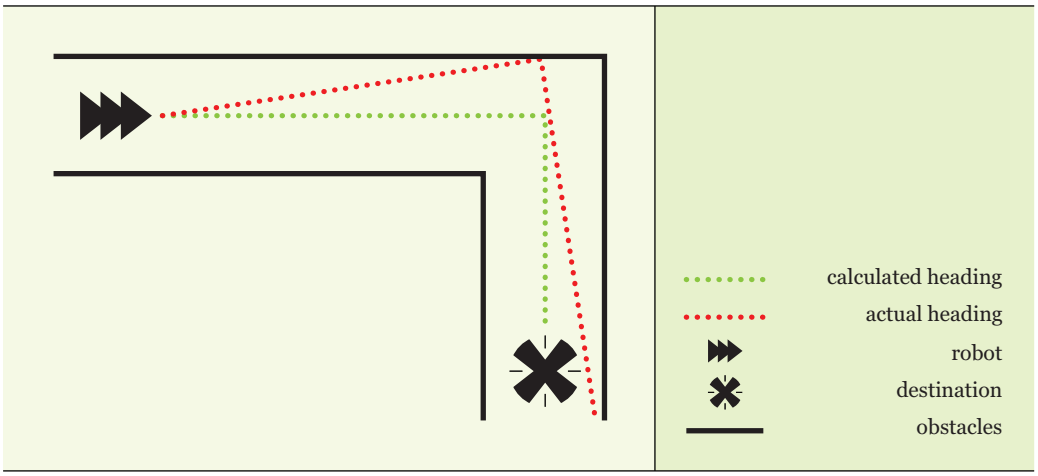


Figure 1.1: Possible consequence of skewed heading.

In addition to this information the robot needs to possess the following capabilities:

- The ability to move.
- The ability to calculate the possible routes from A to B avoiding obstacles.
- The ability to refine the map in real time if obstacles blocking its path are found.

The fundamental problem with these criterias is that they all heavily depend on the very first criterion; at every time knowing the robots exact position and heading. This information may be 100% accurate as the robot starts moving from point A, but as inaccuracies in calculated moved distance and rotated heading with respect to actual moved distance and rotated heading accumulate, the robot may have drifted enough to collide with objects outside its path, as in Figure 1.1. It is therefore crucial that the robot can resyncronize its position and heading during a move without third part intervention. The simplest method of assuring position is by using a positioning system such as IPS (Indoor Positioning System) or GPS (Global Positioning System). There are many types of IPSs available, most of which are implemented by triangulating beaming signals transmitted by fixed-position nodes

within, or at the bounds of, the covered area. GPS is similar to this, but is implemented on a much larger scale using more than two dozen satellites in orbit around the earth. Traditional GPS is position-wise accurate to 5-10 meters [Wing et al. 2005] and using improved systems such as Differential GPS (DGPS) or Wide Area Augmentation System (WAAS) the accuracy can be accurate within 1-2 meters [Lo et al. 2002]. These types of positioning system are excellent choices when the operating environment permits it, they do however have drawbacks:

- They require a certain number of fixed position nodes to be available at all times.
- They are sensitive to radio interference.
- Their accuracy is not always sufficient.

The focus of this thesis is therefore on developing a set of algorithmic and geometric techniques that allows the robot to determine its own position and heading based on laser range data. In addition to this we will look into extensions to an existing pathfinding algorithm, making it suitable for real world robotic usage.

1.2 Feature Detection

A significant section of this paper deals with feature-recognition in 2D range scans, such as the 180 degree laser scans produced by the onboard laser-unit of the P3-AT. This feature-recognition can be thought of as finding the *semantics* of the data, i.e. we attribute the single-point scan results to being part of an obstacle within the lasers scan range. Figure 1.2 shows a possible interpretation of a 2D range scan. By examining the obstacles found and matching them to a map of the relevant area, the exact position and heading of the robot can be found.

1.3 Map Matching

The purpose of doing feature detection is to enable the robot to match the refined laser rangefinder readings to a preloaded map and thereby determining

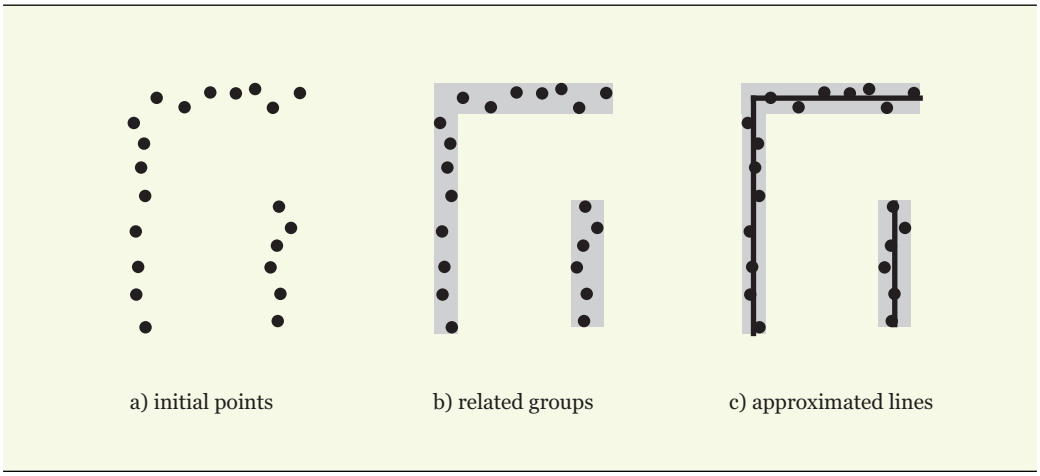


Figure 1.2: Steps involved in 2D feature recognition.

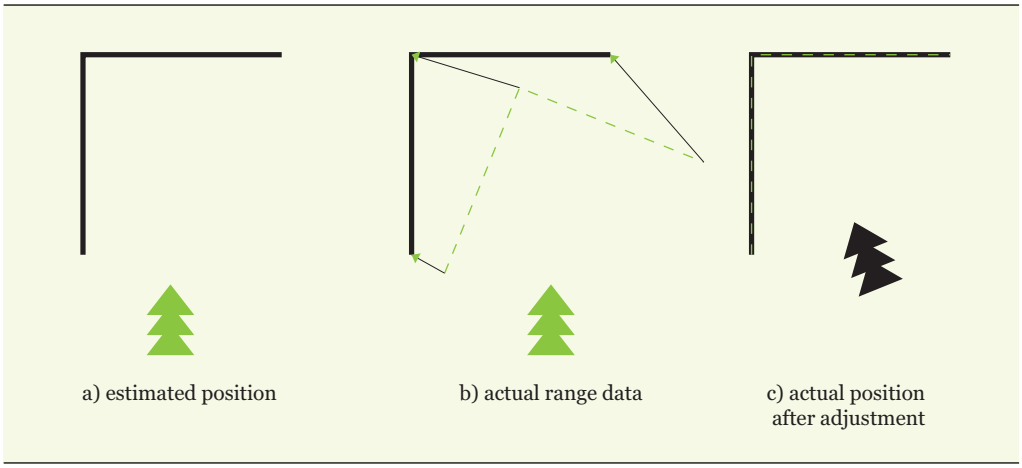


Figure 1.3: Matching rangefinder data to a map.

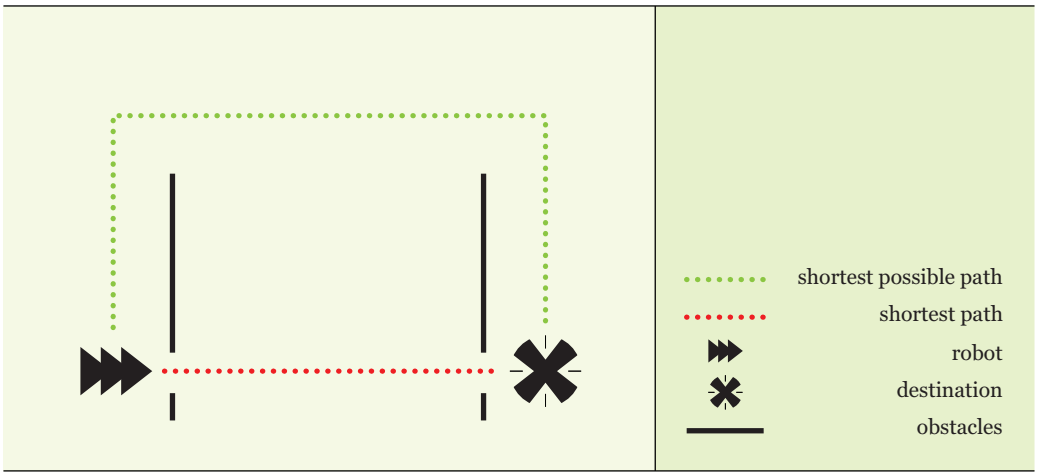


Figure 1.4: Pathfinding considering the vehicle shape and size.

its exact position and heading. The actual process of matching the range data to the map is called *map matching*. This process needs to be as exact as possible, but still robust with respect to noise and uncharted objects the robot might sense in its surroundings. See figure 1.3 for a visualisation of the problem. Section 4.4 covers the subject of map matching in detail.

1.4 Pathfinding

For the robot to be truly autonomous it needs to be able to determine the most appropriate route from point A to point B. This process is called pathfinding and has been thoroughly researched throughout the history of computer science, with Dijkstras Algorithm [Dijkstra 1959], developed by Edsger Wybe Dijkstras in 1959, probably being the most common algorithm. However, most pathfinding algorithms assume that the object that is to move from A to B is a single point in space or a tile in a square tile grid. It is not taken into consideration that the moving object may have complex geometrical shapes as in Figure 1.4. We investigate this matter in section 4.3.2 and propose a solution that allows the robot to calculate optimal routes and move at a high degree of freedom, but still maintain navigational safety.

1.5 Map Refinement

Recording discovered, uncharted objects in the map area allows the robot to avoid having to try a blocked route several times, thus making its operation more effective. This sort of dynamic charting can also be of interest to human operators as it enables the robot to start out with a low-detail map and refining it automatically, minimising human interaction. The subject of map adjustment is covered in detail in section 2.4.

1.6 Thesis Overview

Chapter two covers the background information that serves as a basis for the techniques and algorithms presented in later chapters. We look into methods of interpreting two-dimensional data, in both vector- and raster-form, so that the data can be processed in the specific context of robotic navigation. We then go into pathfinding and analyse algorithms and problems associated with determining shortest possible paths between two points.

Chapter three presents a selection of possible scenarios where the system described in this thesis may prove itself valuable.

Chapter four describes the proposed main system of this thesis and covers detailed coverage of specific issues to consider.

Chapter five summarises our findings and we propose a number of ideas for future work.

Chapter 2

Background

This chapter presents the fundamental techniques and principles that forms the basis for processing the problems addressed in this thesis.

Section 2.1 presents the general form of digital 2D maps, along with the most common terms and formulas needed to work with maps.

Section 2.2 covers the basic geometric transformations forming the basis for the algorithms in chapter four.

Section 2.3 focuses on the subject of feature recognition in vector graphics, that is the methods used to attribute higher order meaning to geometric primitives such as points, lines and polygons.

Section 2.4 presents the concept of map refinement and the considerations that need to be taken when implementing such algorithms.

Section 2.5 explores the problems encountered when faced with the task of algorithmically finding the most efficient paths between two points in planar space and the existing algorithms available for these types of problems.

Section 2.6 addresses the field of image processing, which extends the topics dealt with in section 2.3, and the additional algorithms needed to do feature recognition in raster form graphics. Image processing is not directly a part of this thesis, but we need to present a few techniques as basis for the ideas presented as possible future work in chapter five.

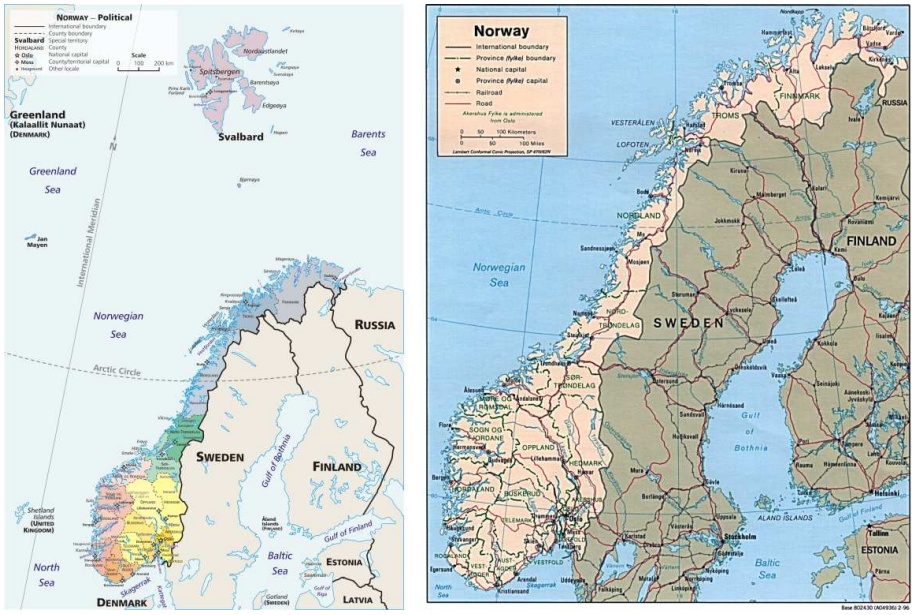


Figure 2.1: A political and a geographic map of Norway.

2.1 Digital Maps

A map is a two- or three-dimensional graphical or numerical approximation of a discrete real-world area, describing one or more of the areas features. It is only an approximation due to natures infinite complexity and resolution, but it is accurate enough for a number of applications. Maps are widely used for navigational purposes, but may also serve other uses, such as political maps that visualise or describe population statistics, or economic maps that visualise economic activity and/or natural resources in an area. Simple maps may only describe its inherent objects by geometrical boundaries, while more sophisticated maps may contain multiple layers of information. Figure 2.1 shows a simple and a more complex map of the same area.

2.1.1 Raster Maps

Generally digital map representation can be divided into two categories; *raster maps* and *vector maps*. A *raster map* is a map where the map area has been divided into discrete subunits of finite size and every subunit describes a certain

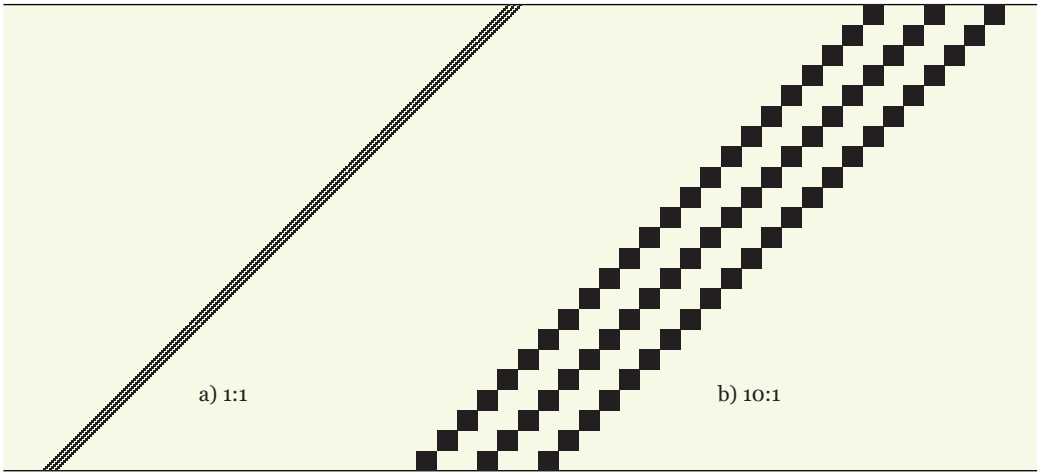


Figure 2.2: An example of the consequences of upscaling a raster image.

feature. Raster maps are also known as bitmaps. Every map that is to be presented in a graphical form is sooner or later converted to a raster map, since all visualisation units work in finite resolution, e.g. a computer screen has a given maximum pixel resolution and a printer can print a maximum number of points per square inch. Raster maps have the advantage of being easy to process and draw to various surfaces, they do however scale poorly, meaning that the perceived image quality may be reduced when the image is scaled up, as shown in Figure 2.2, they consume a lot of storage and they bear no semantics as to what features they describe.

2.1.2 Vector Maps

Vector maps are maps where features are textually and numerically described rather than drawn. Features are thus described in the form of geometrical primitives such as points, lines and polygons. Vector maps scale very well, they are compact and it is easy to embed descriptive meta-information onto every feature. On the negative side they require potentially very advanced rendering software to visualise and manipulation of complex vector graphics can be computationally expensive. Figure 2.3 shows an excerpt from a SVG-file and the resulting rendering.

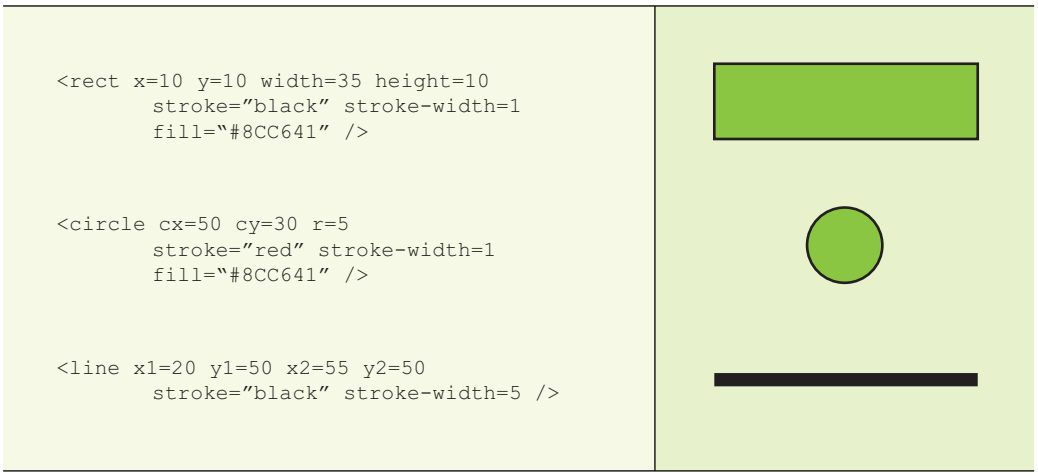


Figure 2.3: Vector graphics representation with rendered output.

2.1.3 Coordinate Systems

The purpose of a coordinate system is to serve as a reference point for positioning of the objects in the map. All positions must be seen relative to the coordinate system for which they are given. Two types of coordinate systems are used in maps; the *geographical* coordinate system and the *rectangular* (or *Cartesian*) coordinate system. Global coordinates are usually given in geographical coordinates, with the x-axis measured in 180 degrees of latitude and the y-axis measured in 360 degrees of longitude, both of which are centered around the Earth's polar axis. Smaller two-dimensional maps are usually given in rectangular coordinates. Figure 2.4 shows a typical rectangular coordinate system, with positive x-axis pointing right and positive y-axis pointing up, containing a line of length 35 and width 2 placed in coordinate $30, 48$, a circle of radius 5 placed $50, 30$ and a rectangle with width 35 and height 10 placed in $10, 10$. It is worth mentioning that while the coordinate system used by most computer graphics routines is similar to this, the y-axis is most often reversed so that the positive y-axis is downward. This can be achieved by doing a matrix multiplication with the y-value set to -1:

$$\begin{vmatrix} 1 & 0 \\ 0 & -1 \end{vmatrix}$$

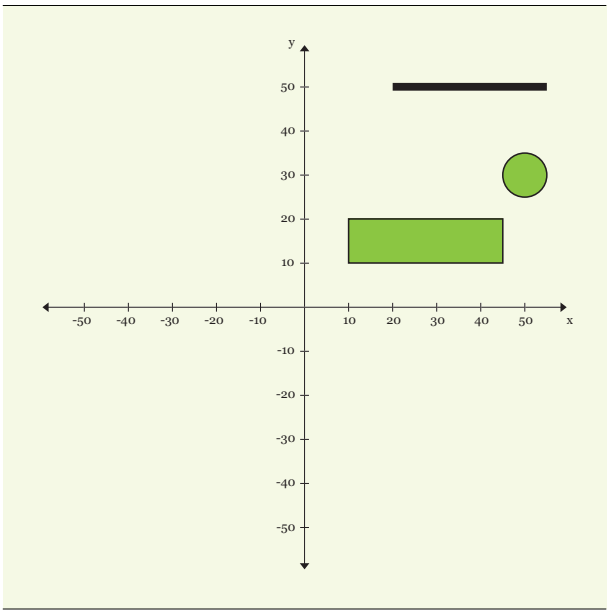


Figure 2.4: The rectangular coordinate system.

2.1.4 Scale

For a map to be useful to humans, we need to know how the map data relates to the area it describes. In graphical maps, printed or in raster form, this relation is called the scale. The scale describes the ratio to which any unit length on the map needs to be multiplied to equal the actual length the maps describes. If a map scale is expressed as $1 : 1,000,000$ this means one unit on the map is one million units in real length, e.g 1 cm on the map is 10 km in real length. In raster maps knowing the scale of the map is essential, in vector maps we are more interested in the unit of which the coordinates are given. The scale can be expressed as

$$\text{Scale} = \frac{\text{Distance}_{\text{map}}}{\text{Distance}_{\text{real}}}$$

2.1.5 Distance Measures

To determine the distance between two points on a two-dimensional map, two common methods exist, the Euclidean distance and the City Block/Man-

hattan distance.

The Euclidean distance is the direct point-to-point distance and assumes that movement at any angle is possible. It is based on the Pythagorean theorem

$$h^2 = o^2 + a^2$$

where h is the hypotenuse of the triangle and o and a are the two sides. The shortest distance between two points is thus

$$d_e = \sqrt{(o^2 + a^2)}$$

The other common distance measure is called the City Block/Manhattan distance. This method assumes that it is only possible to move horizontally and vertically and thereby simplifying the calculation, leading to a more efficient, but not optimal distance algorithm. The Manhattan distance is calculated as

$$d_m = o + a$$

2.2 Geometrical Transformations

In this section the three most basic and common geometrical transformations are presented; translation, scaling and rotation. The transformations are explained with reference to digital images, but the term "image" is here used loosely and covers both raster images and vectorized graphic data. It is important to note that these transformations not only apply to full images, but also sub-images or *regions* of the full image. We will refer to these regions as "image elements". The smallest image element is the point or pixel.

2.2.1 Translation

Two-dimensional translation is the simplest of the basic image transformations, it involves displacing an image element at (x_1, y_1) by the specified translation (β_x, β_y) as shown in Figure 2.5. Mathematically it can be described as

$$x_2 = x_1 + \beta_x$$
$$y_2 = y_1 + \beta_y$$

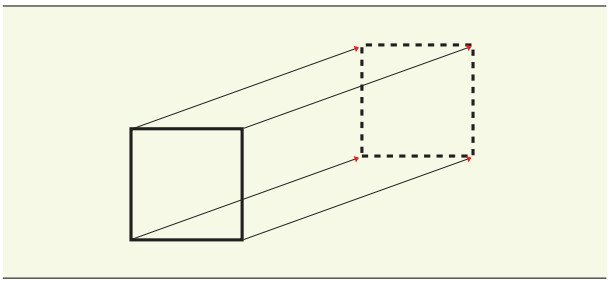


Figure 2.5: Simple two-dimensional translation.

2.2.2 Scaling

Scaling is often referred to as zooming or resizing an image element. Scaling an image element by a factor of 2^n with $n > 0$ is the simplest case, as pixels in the image simply are repeated 2^n times in both dimensions. Similarly when reducing an image size by a factor of $\frac{1}{2^n}$, only every 2^n pixel is kept and the rest are discarded. When scaling to a factor that is not a multiple of 2, it is not this simple. Imagine the case of scaling a 2×2 image element by 1.5, resulting in a 3×3 image, what values would be appropriate for the pixels with coordinates $x = 2$ or $y = 2$? The answer is that the pixel value needs to be *interpolated* between the values proceeding and succeeding the actual pixel. A simple method of interpolating, that would work in our $2 \times 2 \Rightarrow 3 \times 3$ example, is to simply take the mean value of the surrounding pixels. But for the general case of scaling to a factor β, more sophisticated interpolation techniques must be applied. The most common technique today is called *bi-cubic interpolation* and it calculates a weighted average of a 4×4 grid around the considered pixel. It can be expressed as

$$F(x', y') = \sum_{m=-1}^{2} \sum_{n=-1}^{2} F(x+m, y+n) R(m-dx) R(n-dy)$$

where the weighting function $R(x)$ is defined as

$$R(x) = \frac{1}{6}\left(P(x+2)^3 - 4P(x+1)^3 + 6P(x)^3 - 4P(x-1)^3\right)$$

and $P(x)$ is defined as

$$P(x) = \max(x, 0)$$

$F(x', y')$ is the pixel value for the resulting image in position x', y', which is related to $F(x, y)$ in the original image by $x' = x\frac{w'}{w}$ and $y' = y\frac{h'}{h}$. dx and dy are the respective decimal parts of x and y after calculating these indices; $x = x'\frac{w}{w'}$ and $y = y'\frac{h}{h'}$.

2.2.3 Rotation

Rotating an image by an angle θ around the point (x_c, y_c) will result in a image where the pixel (x', y') in the rotated image will correspond to the pixel (x, y) in the original image according to the following formula:

$$x' = (x - x_c)\cos(\theta) - (y - y_c)\sin(\theta) + x_c$$
$$y' = (x - x_c)\sin(\theta) + (y - y_c)\cos(\theta) + y_c$$

and inversely

$$x = (x' - x_c)\cos(-\theta) - (y' - y_c)\sin(-\theta) + x_c$$
$$y = (x' - x_c)\sin(-\theta) + (y' - y_c)\cos(-\theta) + y_c$$

One way of calculating the rotated image is to iterate over all (x', y'), calculate (x, y) and simply use the pixel value of (x, y). The problem in this approach is that these formulae return floating point coordinates, thus if we want to calculate what point (x, y) corresponds to (x', y') the formula may return e.g. $(1.5, 1.5)$ or $(1.666667, 1.4177)$ which does not make sense in a context of discrete integral array indexes. The pixel value of (x', y') must therefore be *interpolated* from the surrounding pixels of (x, y).

The bounding rectangle of a rotated rectangular image will also be dimensionally bigger than the original image for all angle values except multiples of 90°. The new pixels (x', y') for which there is no corresponding (x, y) will usually be set transparent, if supported by the image format, or to some predefined color e.g. black or white. See Figure 2.6 for an illustration of an image rotated around its own center.

2.3 Feature Recognition

Feature recognition is a term used in computer vision to describe the process of extracting meaningful information from raw sensor data. The specific nuance of feature recognition described in this paper is the process of extracting

Figure 2.6: Rotation by $-20°$.

polylines from laser rangefinder scans. One of the main objectives of this thesis is to describe a simple method of matching laser range scans to vector maps to be able to accurately determine the robots position at any time. These laser range data readings are basically a set of semi-accurate distances to the nearest reflective object at a given angle from the sensor (see figure 2.7 for a visualization). From these isolated points in the 2D plane we wish to find line tendencies in the range data that can be matched to 2D map data objects, such as walls.

2.3.1 Range Data Grouping

When analyzing a set of range data readings to extract recognizable features, one of the first operations to perform is to organize the readings into related groups based on reading features. This is done as part of the process to simplify the range data readings down to more manageable geometric shapes. A simple range data grouping algorithm could group readings based on Euclidian distance alone, such that reading p belongs to group G if the Euclidian distance from p to any reading in G is less than the threshold t_1. A more sophisticated algorithm could take the group's curve into consideration, so that reading p only belongs to group G if p diverges from G's curvature by less than the threshold t_2. Figure 2.8 shows the difference between a simple

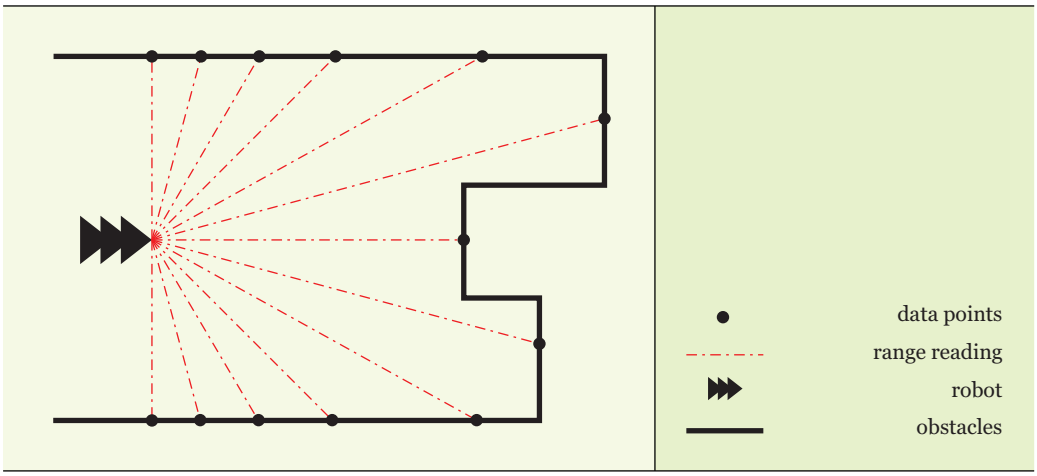


Figure 2.7: A laser rangefinder scan.

and a more advanced grouping algorithm. The algorithm used when grouping the points at the left considers only Euclidian distance between points to determine the group a point belongs to, while the more advanced algorithm, used on the points on the right hand side, considers Euclidian distance as well as curvature when determining how to group points. After completing the range data grouping, the range scan data readings are organized into a number of groups that each represent a part of a geometrical map feature or obstacle. For the remainder of this thesis I will focus on grouping range data readings into lines as the line works well for approximating other shapes and since the line is the simplest two dimensional shape.

2.3.2 Line Approximation

After grouping the range data readings we need to determine the approximation of the line that the readings represent. This means that we are trying to find the line that best fit the reading points such that the sum of the point-to-line distances is minimized.

One well known method of finding such a line is called the method of *Least-Squares Estimation*. This method determines the estimators b_1 and b_0 for the best fit line function through the reading points. The "best fit line" is defined as the line that minimizes the sum of vertical point-to-line distances,

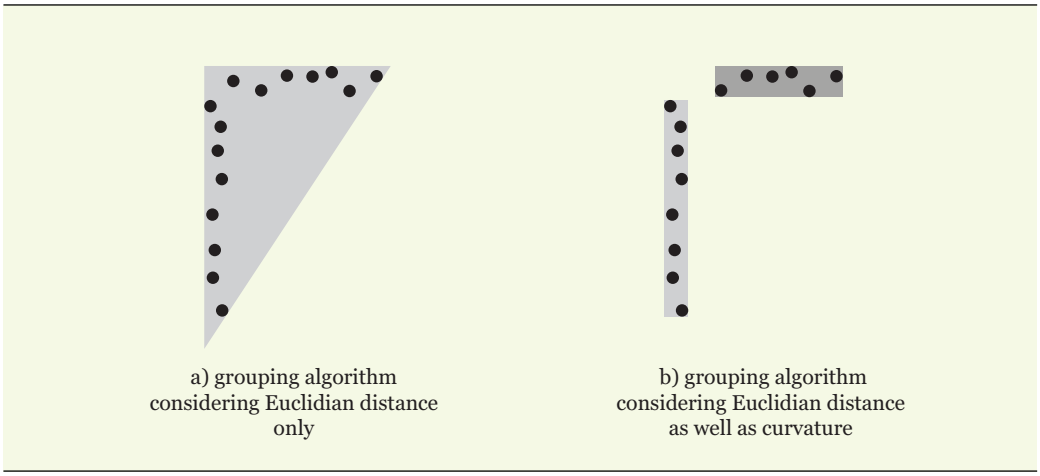


Figure 2.8: Simple vs. more advanced range data grouping.

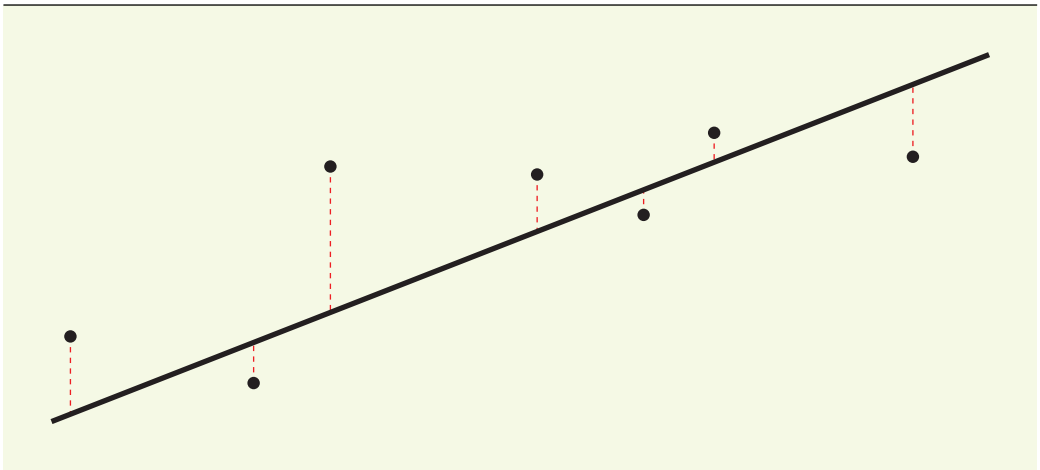


Figure 2.9: The best fit line as calculated by least squares estimation.

as illustrated in figure 2.9. The line function is defined as

$$y(x) = b_1 x + b_0$$

And to determine the estimators b_1 and b_0 the following equations are applied

$$b_1 = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$$

$$b_0 = \frac{\sum y_i - b_1 \sum x_i}{n}$$

To find the segment boundaries of the estimated line we simply calculate $(x_{min}, y(x_{min}))$ and $(x_{max}, y(x_{max}))$.

2.4 Map Refinement

When moving in a semi charted dynamic environment, a robot is bound to sense objects and obstacles that do not exist in the robot's preloaded map. These objects can be static or semi-static obstacles, such as furniture and moveable walls, or highly dynamic objects such as people, animals or other moving robots. By storing information about these discovered objects into the robots map, more efficient paths can be obtained by avoiding routes through highly dynamic areas. The following sections describe which aspects should be considered when adjusting maps in real time.

2.4.1 Object isolation and feature recognition

When a new object is discovered, its form and position need to be determined. The form may have to be simplified into a manageable complexity and the position needs to be adjusted from the original readings by the robots corrected position and heading. The subject of object isolation and feature recognition is covered in detail in chapter 4.

2.4.2 Obstacle persistency

When dealing with moving obstacles, an object can not be considered a permanent obstacle, but should rather have a timeout to ensure that the robot

will reevaluate the objects persistency when the timeout expires. The timeout mechanism can be absolute in the sense that the obstacle is never reevaluated before the timeout expires or it can be weighted so that if the robot runs out of possible routes to its target, it reevalutes the registered obstacles in the order they were discovered, thereby applying the assumption that the "oldest" obstacle is most likely to have moved. An even more sophisticted obstacle tracking algorithm may even try to determine a moving obstacles direction or pattern and use this prediction in its evalutation of the obstacle's persistency.

2.4.3 Data structure order

When processing large amounts of map data, it is often beneficial to have objects located geographically "close" to each other also relatively close to each other in the iterative order of the data structure. This can speed up the process of determining whether a node is walkable in a pathfinding scenario or make the process of asynchronously drawing a map to a screen more smooth by drawing the map area by area instead of drawing seemingly randomly ordered objects. Due to the relatively low complexity of the maps processed as part of this thesis, the subject of data structure order will not be covered in detail.

2.4.4 Efficiency of analysis and insertion

The process of analysing the readings and updating the map data structure may become a performance issue if the frequency of sensor data readings and/or data amount available at each update is larger than the available processing unit is able to process in due time. One should therefore carefully choose processing algorithms and data structures that are suitable to the scenario in question. Theoretical analysis and a selection of actual test metrics for the algorithms and data structures described in this paper are available in section 4.3.4.

2.4.5 Inaccuracy evaluation

In addition to discovering uncharted obstacles, the roaming robot may also detect errors in the map that facilitates new routes. By correcting these errors

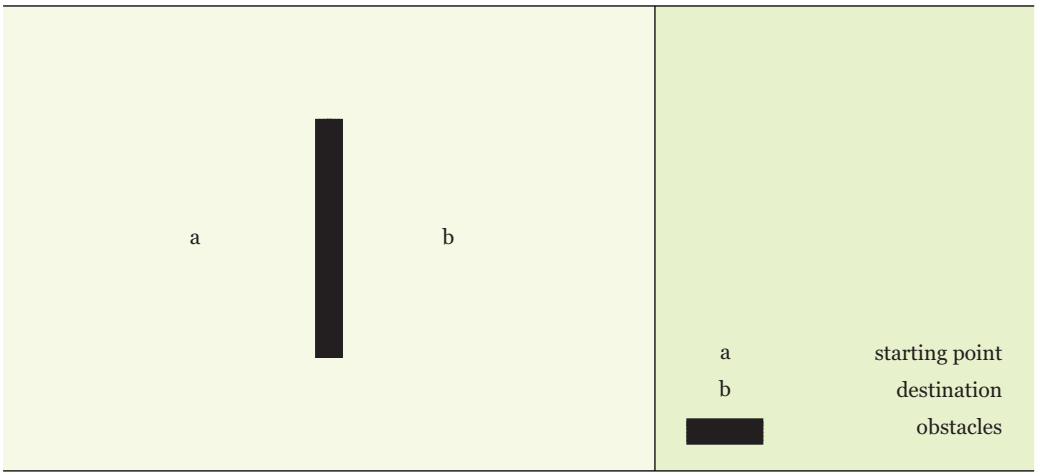


Figure 2.10: A possible pathfinding scenario.

in the map new and more efficient routes can be found and/or the robot can continue when all routes in the known map have been exhausted. Due to the complexities such functionality introduces, the topic of inaccuracy evaluation is not covered in this thesis.

2.5 Pathfinding

To computationally find the shortest possible path between two points in a partially blocked two-dimensional map in the shortest possible amount of time has been greatly researched throughout the history of computer science. The topic of pathfinding is often referred to as *the single-source shortest path problem*. Figure 2.10 illustrates the problem; we want to find the shortest possible path from A to B and we can not travel through the black rectangle in the middle of the map. However, the term "shortest" is not always 100% accurate. In many situations one is interested in finding the path with the lowest *cost*, implying that some operations or moves are more costly in terms of time/resources than others. E.g. for a primitive robot the cost of rotation might outweigh a slightly longer route if the robot has to stop, rotate and start again for every rotation. In this paper the terms cost and distance are used interchangably.

The map used in a pathfinding algorithm is usually represented by a

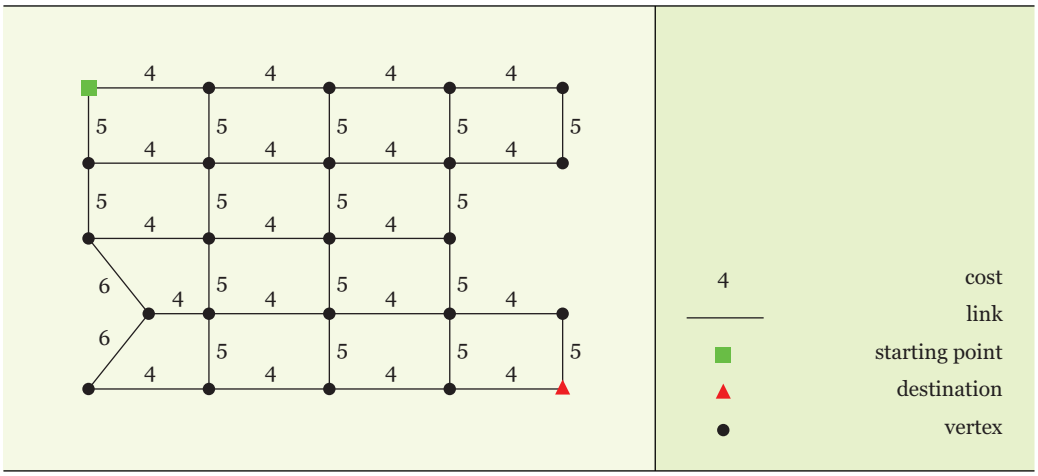


Figure 2.11: A weighted graph.

weighted graph. A weighted graph is a node-structure where every node, called a *vertex*, is connected to one or more vertices via a *link*, and the link between vertices is associated with a given cost. In a *directed weighted graph* the associated cost may differ depending on what vertex one is moving from/to or moving may be restriced to one direction only. See Figure 2.11 for an example weighted graph.

Pathfinding as discussed in this chapter is limited to discreet two-dimensional space and with a fixed progression of waypoints. Related problems, such as The Traveling Salesman, are not discussed here.

2.5.1 Dijkstras Algorithm

One of the first solutions to the pathfinding problem was Dijkstras algorithm, published by Dutch computer scientist Edsger Dijkstra in 1959. Dijkstras algorithm is exhaustive, meaning that it explores every possible path to ensure that the path found is indeed the absolutely shortest. Because of this it has a worst case complexity of $O(V^2)$, where V is the number of vertices.

The general problem solved by Dijkstras algorithm is to find the shortest path from s to t in the graph consisting of the set of vertices V. The algorithm maintains two lists, S and Q, where S is the set of vertices from V for which we know the shortest possible path and Q is remaining vertices from V. For

each vertex v in V, the algorithm stores the shortest path known from s to v, by storing a reference to its *previous*, and the total distance d between s and v. The algorithm first sets all vertices in V to have undefined previous and an infinite distance to its previous. s is set to have distance 0, S is set as an empty set and Q to contain all vertices in V. The algorithm then goes into a loop where it selects the vertex u from Q with the lowest distance and adds this to S, for the first iteration this will always be s as per the initilization. An inner loop then iterates over all vertices v connected to u to see if the distance from s to v would be lower if v had u as previous, if so: set u to be v's previous. Ultimately u will be equal to t and the shortest path from t to s can be found by backtracking through the previous-references.

Dijkstras algorithm is widely used and is especially suited for graphs where no initial information about the form of the graph is present, e.g. the destination might not be known, so the algorithm works as a combined pathfinding and search-algorithm, already knowing the shortest possible path when the vertex fulfilling the search criteria is found. For more structured graphs algorithms such as the *A-Star algorithm*, may be more appropriate.

2.5.2 The A-Star Algorithm

The A-Star algorithm was designed by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968 and is perhaps the most widely known pathfinding algorithm. In contrast to Dijkstras algorithm the A-Star algorithm does not simply iterate through every possible combination of moves through the graph, instead it employs a *heuristic*, a guess, to estimate what would be the best path from the current vertex to the destination and explores this route first, before trying the other paths if the seemingly optimal route is blocked.

Central to the A-Star algorithm is the heuristic function, $h(v)$, that estimates the best-case cost of moving from vertex v to the destination t. With a simple two-dimensional map this function may simply return the straight line distance between v and t. For every v the distance travelled from the starting point s to v, $g(v)$, is also stored. The total estimated cost, $f(v)$, for a vertex v is thus $g(v) + h(v)$.

The A-Star algorithm also maintains two lists, the open-list, O, and the closed-list, C. O contains all vertices that have not yet been visited and C the vertices which have been visited.

The algorithm starts by setting C empty and O to contain only s. It then enters an loop where for every iteration the vertex v with the lowest $f(v)$ is moved from O to C and for every adjacent vertex u to v the following is done:

1. If u is the destination, the shortest path have been found, break from loop.
2. If u is not walkable or if it is found in C, continue to processing next adjacent vertex.
3. If u is not in O, set v as previous vertex to u, calculate $h(u)$, $g(u)$ and $f(u)$ and add u to O.
4. If u is in O, check if $f(u)$ is lower with v as its previous rather than its current previous, if so: set v as u's previous, update $h(u)$, $g(u)$ and $f(u)$.
5. If O is empty and the destination has not been found, there is no walkable path, break from loop.

As with Dijkstras algorithm, the shortest path from s to t can be found by backtracking through the previous-references, starting at t.

2.5.3 Shape Compensation in Pathfinding Algorithms

The pathfinding algorithms described in this chapter use obstacle maps to determine the shortest travelable route from start to destination. When querying a map to determine whether a vertex is travelable or not it is easy to forget that the shape of the traveling object needs to be considered. The maps used in two-dimensional pathfinding are usually $N \times M$ grids, but it is not given that the traveling object is of size 1×1 and even if it was, the object could probably not move diagonally between two obstacles that are placed diagonally of each other as in Figure 2.12. To handle these types of problems for general polygon objects many advanced methods have been designed, see the work done by Eyal Flato [Flato 2000]. For simpler and less general geometrical shapes, two different approaches can be used depending on usage scenario.

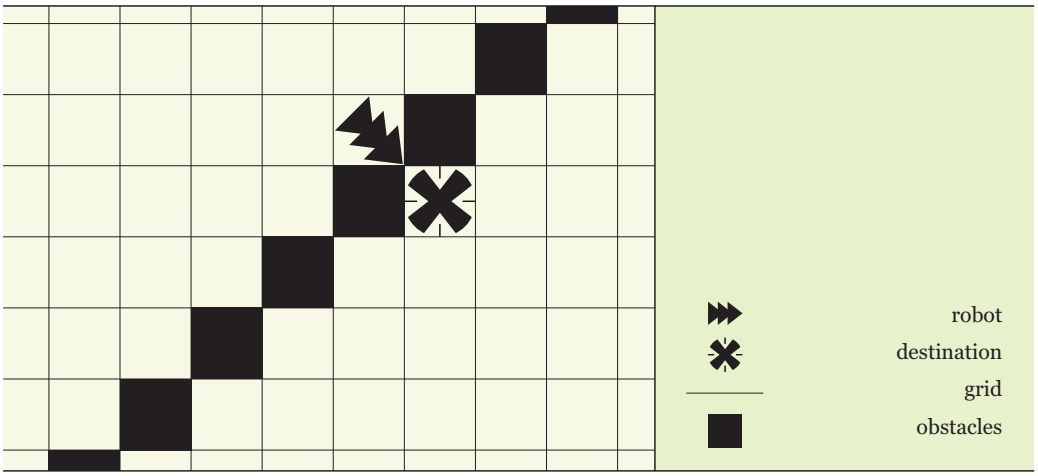


Figure 2.12: Obstacle misrepresentation in a map.

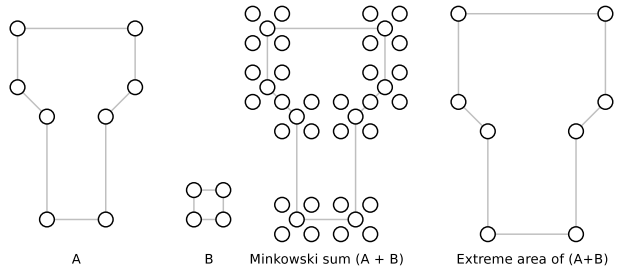


Figure 2.13: Finding the Minkowski Sum of two polygons.

Minkowski Sums

Finding the Minkowski Sum of two polygons involves summing the individual vertex coordinates into a new polygon and then finding the extreme convex area of the resulting polygon. See figure 2.13 for a visualisation.

When working on the subject of pathfinding, using Minkowski Sums is useful for finding the "travelable" area of the map, for a given rotation of the robot. By summing the vertices of the vehicle polygon to every vertex in every map polygon and finding the new extreme area of each map polygon, the untravelable areas for this particular vehicle is discovered and can be excluded from the map.

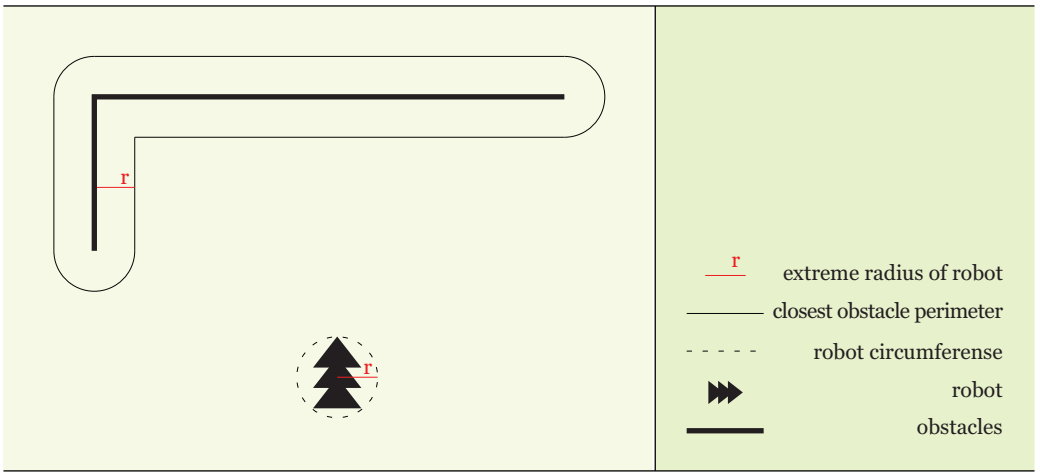


Figure 2.14: Obstacle perimeter calculated using robot radius.

Considering robot radius

The robot that forms the basis for this thesis, the ActivMedia Robotics P3-AT, has the ability to rotate around its own vertical axis. Considering this and the fact that the robot may at any point in the route need to rotate if it runs into obstacles or needs to correct its heading, we can simplify the robots shape into a single circle with radius equal to the distance from the robots center of rotation to its most distant point. When we treat the robot as a circular shape we do not need to apply any Minkowski Sums to any of the map polygons, we simply define any area within distance s of a map obstacle to be untravelable, where s is the radius of the circular robot representation. This way the robot is never allowed to travel within a distance to a map obstacle that might cause the robot to crash or to get stuck and we avoid the complexities that techniques such as the Minkowski Sum involve. Figure 2.14 shows the obstacle perimeter as calculated by this method.

2.6 Digital Image Processing

In this section we briefly describe a set of methods that can be useful as a background for further work from this thesis. Some thoughts on further work are presented in chapter 5.4.

2.6.1 Digital Images

Digital images may come from a variety of sources, but they can be broadly divided into two categories; aquired and synthesised. Aquired images come from sensors that capture the state of some real-world situation at one point in time, e.g. a digital photograph or a plot of robot laser range readings. Synthesised images are visualisations of non-visual data either made with human interaction or completely auto-generated, e.g. a digital drawing or a fractal image.

2.6.2 Digital Image Representation

Conventional digital images, as will be described in this section, are stored as a two-dimensional numerical matrix, where every vertex in the matrix describes a single point in two-dimensional space called a *pixel*. The amount of data describing each pixel depends on the image format, but at least a measure of color or light intensity is required. A very simple *binary image format* may contain only the data 1 or 0 for every pixel, which we may interpret as light or dark respectively. A greyscale image format could represent every pixel as a number between 0 and 255, giving 256 shades of grey, whilst a more complex image format may contain highly detailed color-information and levels of transparency for every pixel. One popular model of describing color is by separating the color into red, green and blue channels (RGB) and have a finite set of values for each channel. An image format capable of distinguishing between over 16 million color variations could store each channel in one eight-bit byte, giving $2^8 = 256$ variations for each channel, summing up to $256^3 = 16777216$ variations for each pixel. It is generally accepted that the human eye can not distinguish between more than 16000000 colors, so a 24-bit image can be said to be "true color", meaning that the image format is capable of containing every color the human eye can see.

To be stored as standalone files, the image data needs to be serialized into a byte stream and this byte stream is usually preceeded by a section of metadata describing the pixelformat and metrics of the image data. The pixels in the byte stream usually come row-by-row, thus a single pixel (x,y) can be accessed at offset $x + (y \times \text{row-width})$ in the byte stream. It is however, from a programmers point of view, often more convenient to be able to access each pixel as image[x][y] and most programming languages provices facilities

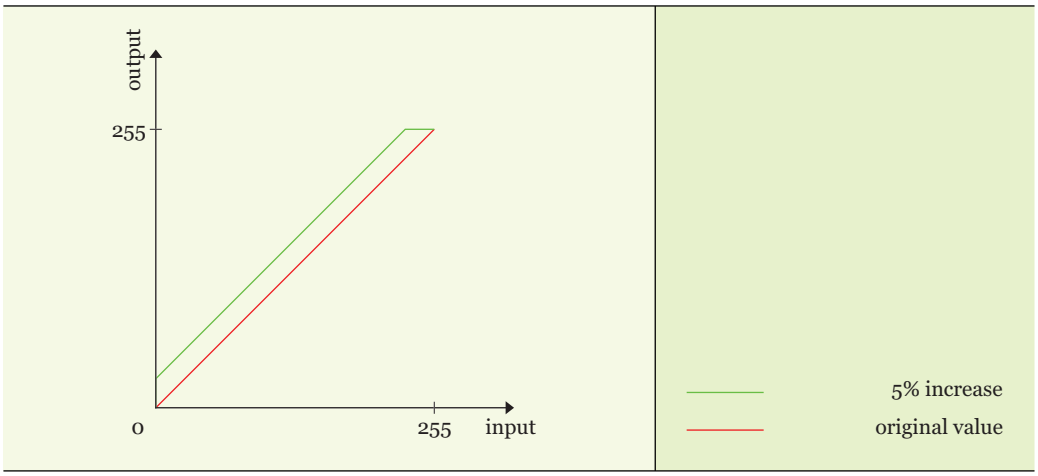


Figure 2.15: Graph showing increase in brightness.

for this.

Digital images can easily become storage intensive, as their size often surpasses 1000x1000 pixels. A 24-bit image at that size would require almost 2.9 MB of storage, causing heavy loads on transfer and processing resources. To ease this load, a number of *image compression algorithms* have been developed, with JPEG, PNG and GIF being the most common container-formats. The main difference between the image compression algorithms is whether they are non-lossy or lossy, meaning whether the algorithm produces a one-to-one copy of the original image when decompressing, or if an approximation is made.

2.6.3 Brightness and Contrast

The brightness of an image describes the luminance of the image, that is the percieved amount of light the image radiates. For a 8-bit greyscale image the brightness μ of a pixel is simply the greyscale value of the pixel, thus a pixel with value 255 is at maximum brightness or *saturation*. For a 24-bit color pixel the brightness μ is defined as $\mu = \frac{R+B+G}{3}$. In other words the mean value of the color components.

To increase the brightness of a greyscale image by e.g. 5%, $255 * 0.05$ is added to all pixel values and those values falling outside the range are set to

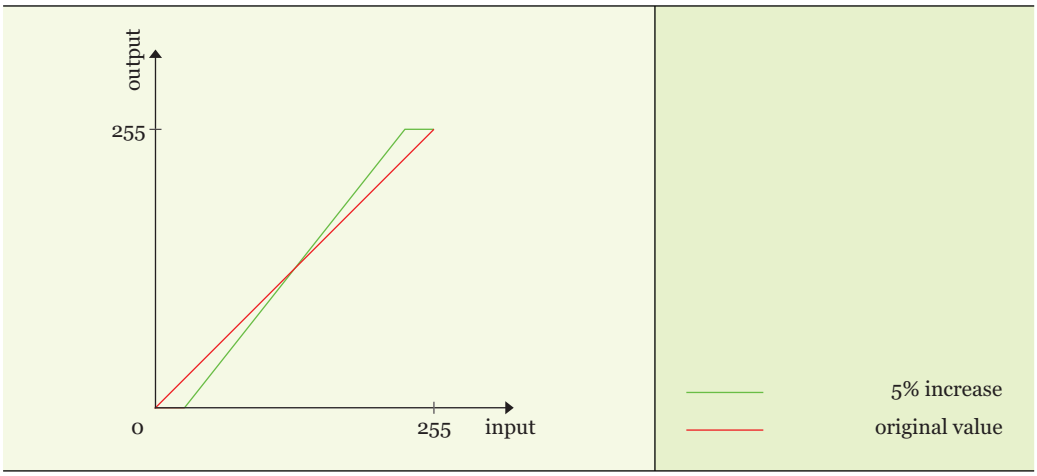


Figure 2.16: Graph showing increase in contrast.

255, as shown in Figure 2.15.

Contrast is the difference between maximum and minimum brightness in an image and is calculated using the formula:

$$\text{Contrast} = \frac{\mu\text{max} - \mu_{\text{min}}}{\mu\text{max} + \mu_{\text{min}}}$$

To increase the contrast of a greyscale image by e.g. 5%, all values below $255 * 0.05$ are set to black (0), all values over $255 - (255 * 0.05)$ are set to white (255) and all values between are scaled linearly between 0 and 255 as shown in figure 2.16. The result of increasing the contrast of a whole image can be seen in Figure 2.17.

2.6.4 Surface Smoothing

To simplify images so that they can more easily be processed, the first step is to *smoothen* the image. Smoothing removes small pixel value variations so that image sectors appear more unified and the image surface appears more smooth, simplifying complex operations such as edge detection. Surface smoothing is usually done by applying a $n \times n$ sliding window over the image, sorting the values of the window and then finding the median value and setting this as the new pixel value. This process is called convolution and the sliding

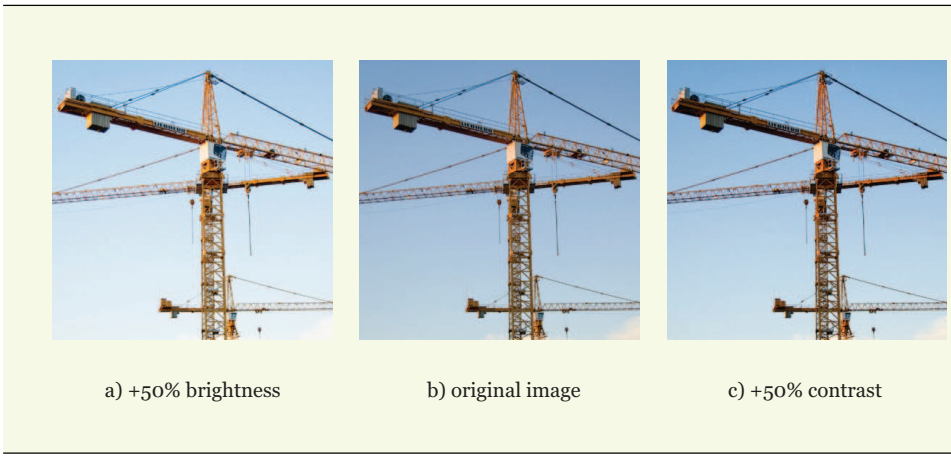


Figure 2.17: The visual effects of adjusting brightness and contrast.

window is sometimes called a *kernel* if there is given different weighting on the pixels in the window. The median is used instead of the mean so that the effect of stray noise pixels is minimised. Note that if using large kernels the complexity of the smoothing rises dramatically, a 3×3 or 5×5 window is usually sufficient.

Using a median filter such as the one in Figure 2.18 can cause edges to become blurry, but by using a specialized weighted window, such as the *Kuwahara filter*, this effect can be reduced. The Kuwahara filter divides the square window with sides of size $K = 4L + 1$, L being an integer, into four rectangles and and sets the value of the center pixel to be the mean brightness of the rectangle with the lowest *variance* (= mean squared deviation).

2.6.5 Edge Detection

Edge detection is the process of finding pixels that lie on the border between two image elements with vertical or horizontal dissimilarity higher than a threshold value K. Finding edges in an image gives us the contours in the image and thus brings us closer to computationally finding what the image presents.

When working with greyscale pictures the difference between two pixels in the image can be determined by subtracting the lower value from the higher, but how can such a difference be determined for color images with colors

```
1  void smooth(Image *img)
2  {
3    int x, y;
4
5    for(y = 0; y < img->height; y++) {
6      for(x = 0; x < img->width; x++) {
7        int kx, ky;
8        int kernel[3][3];
9
10       for(ky = -1; ky < 2; ky++) {
11         for(kx = -1; kx < 2; kx++) {
12           kernel[ky+1][kx+1] =
13             getpixelvalue(img->data, y + ky, x + kx);
14         }
15       }
16       sort(kernel, 9);
17       setpixelvalue(img->data, y, x, median(kernel));
18     }
19   }
20 }
```

Figure 2.18: C-like implementation of a surface smoothing algorithm.

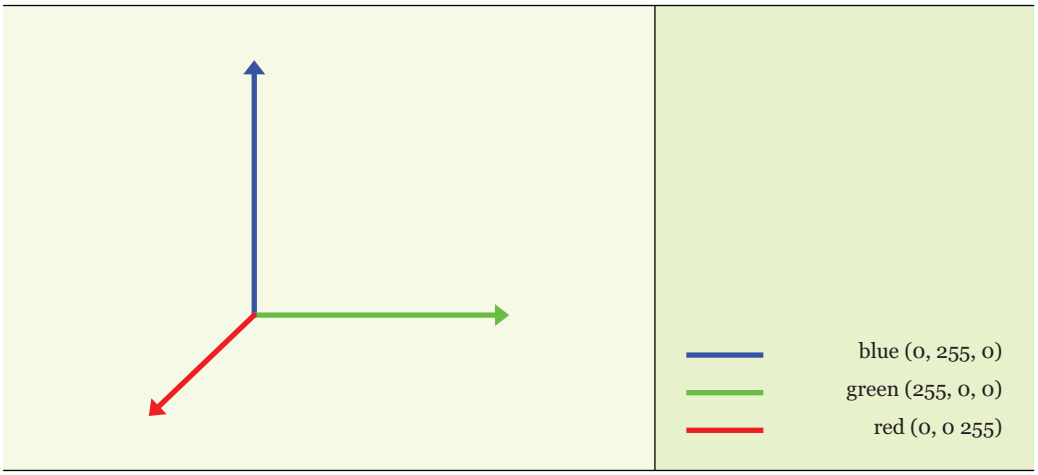
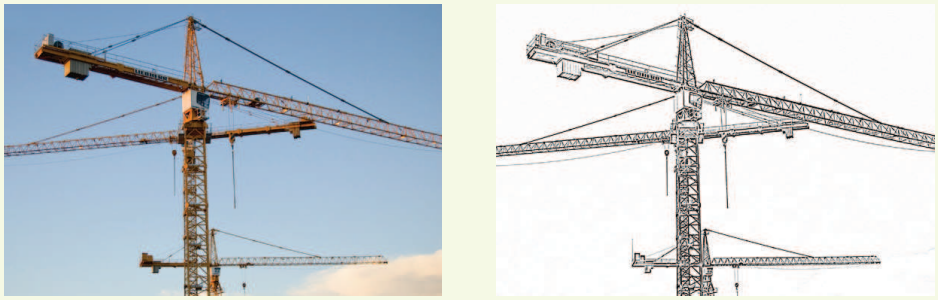


Figure 2.19: 3D representation of RGB-colors.



a) original image b) simple edge detection algorithm

Figure 2.20: A simple edge detectior applied to a 512x384 color picture.

values represented separately as red, green and blue? The solution is to represent the color elements as vectors in three-dimensional space as shown in Figure 2.19. The difference is then defined as the 3D Euclidian distance between the two pixels, defined as $\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$.

Simple Approach

A very simple approach to edge detection would be to iterate through the pixels of the image, and if the difference between the current and the pixel below or the pixel to the right is greater than K we have found an edge. Although very simple, this algorithm can be remarkably efficient on some images, see Figure 2.20 to see the results of running this algorithm on a relatively complex image and Figure 2.21 for an example implementation.

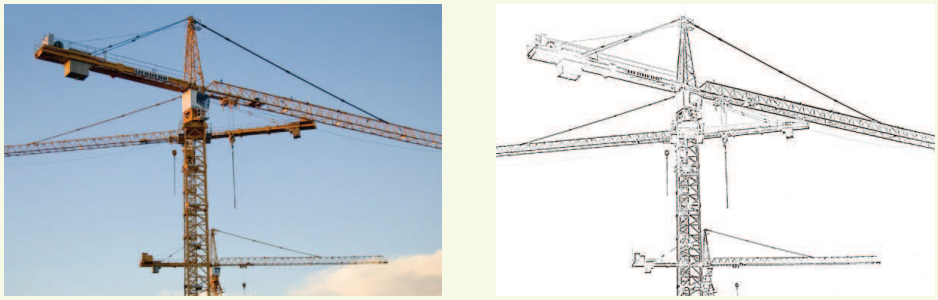
Canny Edge Detector

The Canny edge detector was developed by John F. Canny in 1986 and claims to be an "optimal" edge detector based on the following criteria (courtesy of http://en.wikipedia.org/wiki/Canny_edge_detector):

- *Good detection* - the algorithm should mark as many real edges in the image as possible.

```
1  void edge_detect(Image *img, double K)
2  {
3    unsigned int x, y;
4    unsigned int bpl = img->sizeX * 3; // Bytes per line
5
6    for(y = 1; y < img->sizeY; y++) {
7      for(x = 0; x < bpl; x += 3) {
8        unsigned char *R = (img->data + (bpl*y) + (x+0));
9        unsigned char *G = (img->data + (bpl*y) + (x+1));
10       unsigned char *B = (img->data + (bpl*y) + (x+2));
11
12       // Prefixes: c = current, b = below and r = right
13       unsigned char cr = *R, br = 0, rr = 0; // Red
14       unsigned char cg = *G, bg = 0, rg = 0; // Green
15       unsigned char cb = *B, bb = 0, rb = 0; // Blue
16
17       if(x < (bpl-1)) {
18         rr = *(img->data + (bpl * y) + (x+3) + 0);
19         rg = *(img->data + (bpl * y) + (x+3) + 1);
20         rb = *(img->data + (bpl * y) + (x+3) + 2);
21       }
22
23       if(y < (img->sizeY-1)) {
24         br = *(img->data + (bpl * (y+1)) + (x) + 0);
25         bg = *(img->data + (bpl * (y+1)) + (x) + 1);
26         bb = *(img->data + (bpl * (y+1)) + (x) + 2);
27       }
28
29       if(sqrt((cr-rr)*(cr-rr) + (cg-rg)*(cg-rg) + (cb-rb)*(cb-rb)) >= K
30          || sqrt((cr-br)*(cr-br) + (cg-bg)*(cg-bg) + (cb-bb)*(cb-bb)) >= K) {
31         *R = *G = *B = 0;
32       } else {
33         *R = *G = *B = 255;
34       }
35     }
36   }
37 }
```

Figure 2.21: C-like implementation of a simple edge detection algorithm.



a) original image b) the Canny edge detector applied

Figure 2.22: The Canny edge detector applied to a 640x480 color picture.

- *Good localisation* - edges marked should be as close as possible to the edge in the real image.

- *Minimal response* - a given edge in the image should only be marked once, and where possible, image noise should not create false edges.

The Canny edge detector goes through three stages, first it smooths the image using a gaussian mask, then it iterates through every pixel and marks the highest intensity edge and finally the algorithm traces through all the high intensity lines and marks the pixels there as edges. The algorithm takes three parameters; the size of the gaussian mask and the high and low threshold for the line detection. Pixels that fall below the low threshold as not considered important enough to be part of the line and pixels over the high threshold are considered noise.

The Canny edge detector is highly complex, but there are freely available implementations in libraries such as the *Intel Open Source Computer Vision Library*. Figure 2.22 shows an original and resulting image from applying the Canny algorithm with gaussian mask size 3×3, high threshold 0.80 and low threshold 0.30.

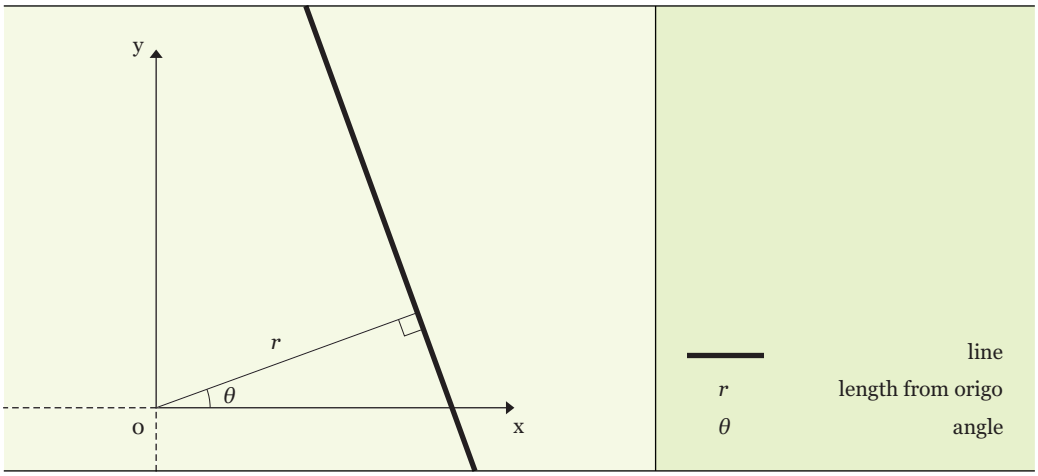


Figure 2.23: Parametric representation of a line.

Hough Transform

The Hough transform is a general method for finding line tendencies in a set of separate points. The word "tendencies" is used to indicate that the Hough transform can not 100% determine the lines in a set of two-dimensional points, but it can indicate the probability of a particular line in the specified plane. The transform is tolerant of gaps in lines and image noise, but is does not indicate the boundaries of a line, only its orientation. This limitation can be overcome by analysing each line and finding the extreme points on that that line, but this of course increases complexity.

The Hough transform represents lines in their *parametric* or *normal* form;

$$x\cos\theta + y\sin\theta = r$$

where r is the length from the *origo* ($(0,0)$) to a normal on this line and θ is the angle of this normal-line with respect to the x-axis. See Figure 2.23 for a visualisation.

For every edge point in the picture, found using an edge detection algorithm like the Canny edge detector, a number of lines at different angles are plotted through the point, and r and θ are calculated for each line. Figure 2.24 shows the lines at angles 0, 45, 90 and 135 degrees plotted through each point and Figure 2.25 shows r and θ for one of the lines. r and θ for each line are then plotted as a sinusoidal curve to a graph as $y = r\sin(x+\theta)$ and the points

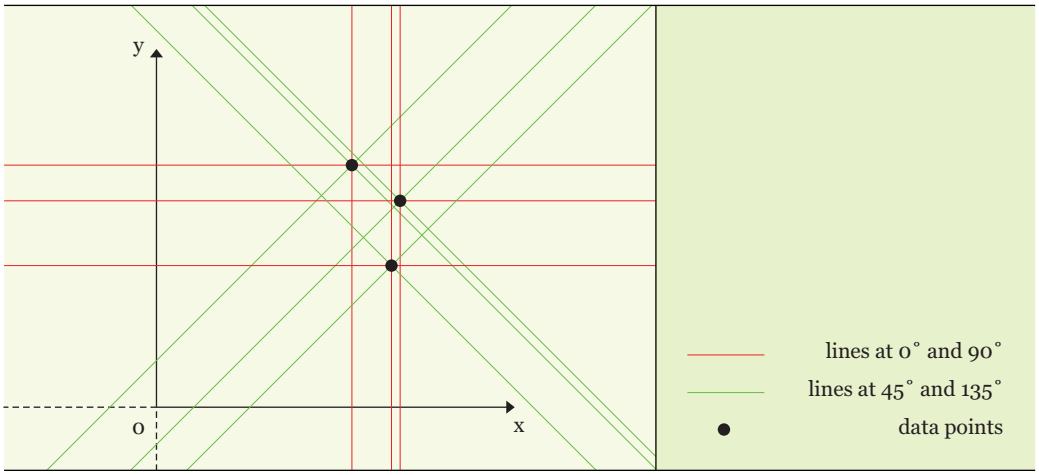


Figure 2.24: An example of Hough lines through points.

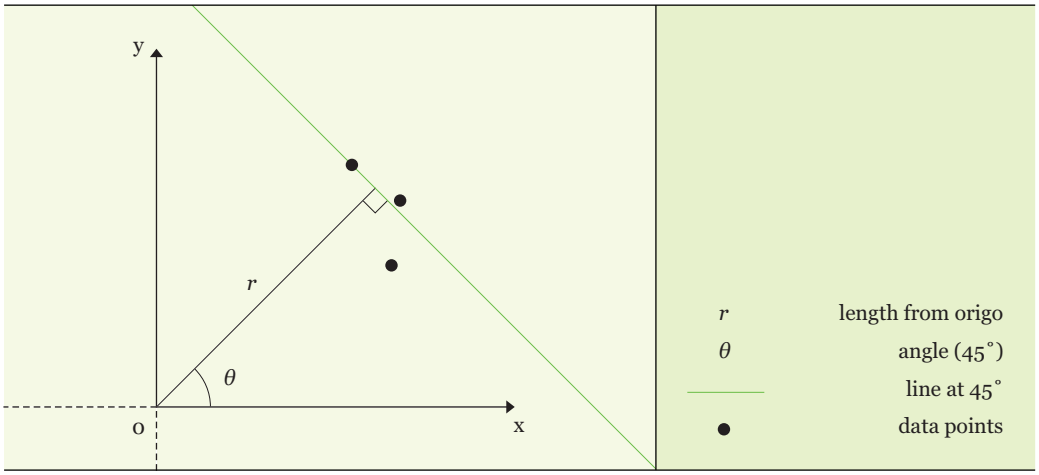


Figure 2.25: A single Hough line and its r and θ.

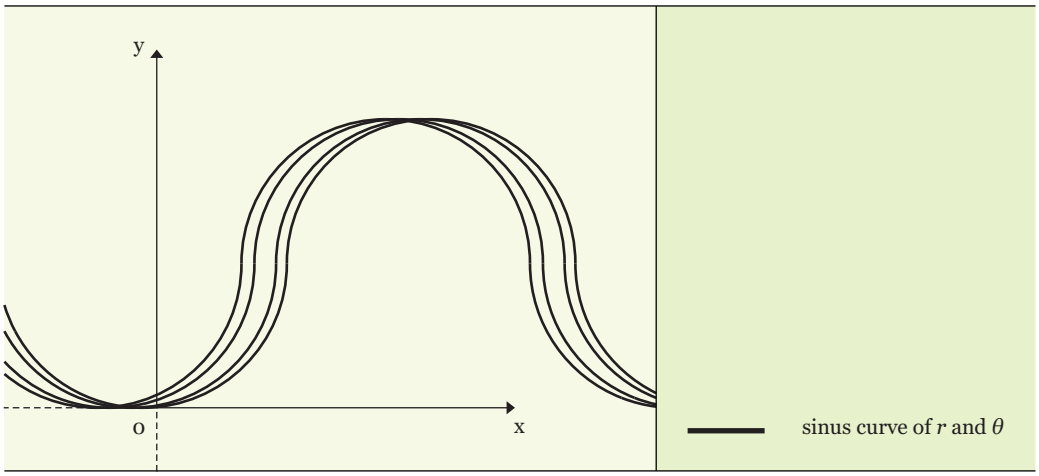


Figure 2.26: The Hough point-to-curve transformation.

where two or more curves intersect indicate that a line ($x\cos\theta + y\sin\theta = r$) exists for the values of r and θ where the curves intersect. This plot of sinusoidal curves is called the Hough point-to-curve transformation. The more curves that intersect in the same point the more certain is the indication of a line for these values of r and θ. Figure 2.26 shows a general Hough point-to-curve transformation.

2.7 Related Work

A lot of research has been conducted in the field of robotic navigation. For an overview of the available techniques, see [Borenstein et al. 1996], [DeSouza, Kak 2002] and [Thrun 2002]. The most recent work in this field has been published by Dr. Longin Jan Latecki at Temple University, Philadelphia, USA. For papers particularly related to the subject this thesis covers, see [Latecki et al. 2004] and [Latecki et al. 2006].

Chapter 3

Scenarios

3.1 Introduction

A robot is, according to the Cambridge Online Dictionary, "A machine used to perform jobs automatically, which is controlled by a computer".

What robots do usually falls into the category "automation". They perform tasks that humans find too hard, too dangerous or simply too dull to perform themselves. But while robots are excellent at doing repetitive and structured tasks from well defined instructions, they often fall short when it comes to adapting to unforeseen events and dynamically coming up with solutions to problems which they are not explicitly programmed to handle. In the context of robotic navigation such unforeseen events can be mobile objects moving around in the robots environment or sensed static objects not present in the robot's map. A robot working in such an environment is said to be working in a "dynamic environment".

This thesis presents a set of algorithms well suited to the problem of pathfinding in a dynamic environment and this chapter describes some scenarious in which this type of pathfinding can be useful.

3.2 Scenario 1 - Search and Rescue

Using robots in a search and rescue mission have several advantages. Robots can be smaller, faster, more mobile and more precise than human beings, but most importantly; they can easily be repaired and in the extreme case they are disposable. This property of being disposable makes them suitable for situations where it would not be feasible to use human personnel for the search and rescue mission. This could be search missions such as finding trapped or unconscious people in a building that is susceptible to collapse or in a building where radiation levels are too high to send in human personnel.

In situations such as these it is of high importance that the robot is capable of intelligent unassisted navigation in the case communication is unavailable for parts or the whole of the mission. The environment within such buildings or constructions can be expected to have changed as an effect of the accident that caused the dangerous situation. Known paths may have been blocked and new paths may have formed and the robot should ideally be able to cope with this new and unknown environment.
A robot suitable for search and rescue applications would need to possess the following characteristics:

1. **Detection of objects or personnel needing attention**
   Perhaps the most important task for a robot to be used in search and rescue operations is the ability to precisely detect an object or person that is in need of help. Although an interesting subject, this will not be covered in this thesis.

2. **Efficient operation**
   A search and rescue operation is in most cases time critical in nature and the robot performing the search should avoid having to spend needless time roaming the search site. The route planning algorithms should be both fast and accurate when predicting the most efficient route from point A to point B.

3. **Robust positional error handling**
   Using a robot to navigate ie. a damaged building means that it will have to handle moving around on contaminated surfaces containing both small and large obstacles. A robot, such the P3AT used as the basis for this thesis, is unable to detect obstacles that lay below or above

its laser range finders vertical position, approximately 35 cm in case of the P3AT. This means that it will be unable to detect low obstacles such as small inventory or thresholds. It is therefore probable that the unit will run into objects causing the actual trajectory to differ from the expected. The navigational algorithms in place must therefore be able to correct the estimated position using sensed range data.

4. **Logging and incorporation of uncharted objects**
   The objects and obstacles discovered should be persisted within the robots computer, both to facilitate optimal route planning throughout the mission and to provide valuable information back to the search and rescue operators.

5. **The ability to determine persistency of objects**
   When faced with an uncharted object, the unit needs to be able to determine if this is a persistent object that limits the navigational possibilities of the robot or if the object is transient or noise. If the object is persistent the robot needs to incorporate it into its route planning algorithms.

3.3 Scenario 2 - Automation, Guiding and Transport

A less dramatic scenario than the search and rescue setting is to use a robot for automation, guiding or transport. Thrun et.al. demonstrated an example application of such a guidance robot in [Thrun et al. 1999]. Their Minerva-robot successfully gave guided tours through the National Museum of American History for a period of two weeks. A number of similar automated tasks in dynamic environments can be thought of:

- Guide robots operating from the reception in a large facility, guiding visitors to their destinations.
- Transport units in a hospital moving patients and beds between locations.
- Automated cleaning robots covering larger areas within e.g. a production hall.

- Feeding-robots in an agricultural setting or in a zoo, feeding animals at regular intervals.

- Watchdog/security robots used for patroling large semi-structured areas and reporting possible security breaches.

The main difference between these proposed tasks and the search and rescue scenario is that most of these tasks are not time critical, but they are operations that span over a longer period of time, and longer geographical distances than what was the case in Scenario 1. This means that errors between estimated and actual position and heading can drift to substancial numbers when travelling in wide, open areas that provide few reference points for the robot's navigational algorithms.

Another important difference from Scenario 1 is the amount of dynamic objects within the robots "senseable" range, e.g imagine the crowd in a newly opened gallery. This places great importance on the robots ability to avoid collisions and further emphasizes its ability to differenciate persistent objects from transient objects, to be able to plan efficient routes within the dynamic environment.

3.4 Scenario 3 - Automated Reconnaissance

The third an final scenario we will examine is the use of an autonomous robot as a reconnaissance unit for military or security applications. Such a robot could be sent as a scout to scan an area for mines/bombs or simply to record detailed geometrical data of the area to facilitate precise planning of a forecoming operation. What kind of challenges does such a usage scenario present? The most important requirement is on the robots ability to record accurate data i.e. filter out possible noise and to record and report persistent objects discovered during the reconnaissance mission.

3.5 Informal Requirements

From the list of possible usage scenarios we have examined, we collect the following informal requirements for a robust robotic navigation system:

1. The system needs to be able to accurately determine its current position and heading, based on estimated numbers and recorded sensor readings.

2. The system needs to avoid collisions with both static charted objects and dynamic uncharted objects discovered during operation.

3. The system needs to be able to estimate the most efficient route from point A to point B, based on both static map data and discovered objects found during operation.

4. The system needs to distinguish between transient and persistent objects discovered and record the objects for optimal route planning and for post-operation analysis.

These four informal requirements form the basis for the rest of the work presented in this thesis. We will present a set of algorithms and structured approaches that facilitate these requirements and we will use a prototype-/simulator to determine the success of the proposed approaches.

Chapter 4

Implementation

4.1 Introduction

This chapter presents a set of strategies and algorithms suitable for autonomous robotic navigation. I will cover the practical challenges encountered when developing these strategies and explore their strengths and limitations.

4.1.1 Possim

To ease testing and experimentation, a robot simulator has been developed. This piece of software, called *Possim*, enables rapid prototyping of concepts and screenshots from this application are used extensively throughout this chapter to illustrate problems and ideas. *Possim* is written as a Java Swing application using a highly modular structure that enables users to load different navigation strategies and position estimation algorithms in runtime. As the robot-component of Possim is replaceable, it is also possible to control a robot using Possim as a graphical front end, but this has not been incorporated in the version of Possim as delivered with this thesis. The simulator enables the user to graphically draw and manipulate maps, as well as the possibility to save maps and restore them at a later point in time. To test how well the pathfinding and map matching algorithms work, the user can also place a robot in the map, set a destination and step through the robots route from start to destination, placing new obstacles in the way and seeing how the algorithms cope with this.

Java was chosen as the technical platform for this simulator based on its

multi-OS availability, its widespread use in education and because of Java's rich library of 2D geometrical functions that simplifies some of the more tedious work involved in implementing such a simulator. Full source code and a set of screenshot videos of Possim in action supplement this thesis. Figure 4.1 shows a screenshot of Possim.

4.1.2 Using Possim

Possim can be used to test various aspects of robot navigation, such a pathfinding algorithms, line estimation algorithms and map matching algorithms. The most common usage scenario is to draw or load a map, place the robot in the map, set a destination, calculate the route from start to destination and then to step through the route and see how well the robot estimates its own position from its simulated laser rangefinder readings. To do this in Possim, follow these steps:

1. Hit the keyboard shortcut Ctrl+R and click in the main area of the Possim window to place the robot.

2. Move the mouse pointer in the direction you want the robot's heading to be and click again to set the robot's heading.

3. Hit Ctrl+A to start a new map obstacle polyline and click in Possim's main window to add vertices to this polyline. Hit Escape when done.

4. Press Ctrl+D and click in Possim's main window to set the robot destination.

5. Press F5 to update the robot's laser rangefinder readings.

6. Press F6 to calculate the shortest route from robot position to destination.

7. Press F7 to move one step along the calculated route towards the destination.

To manipulate a map, use these shortcuts:

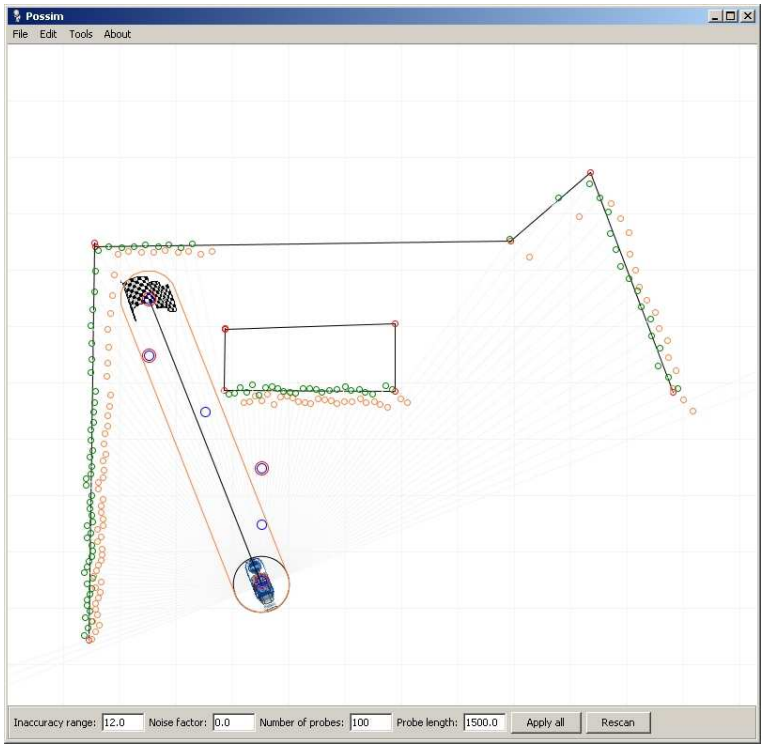


Figure 4.1: An example screenshot of Possim

1. Hit Ctrl+M and click a map obstacle vertex to select it. Move the mouse pointer to where you want to relocate the vertex to and click again to place the vertex in the new location.

2. Hit Ctrl+E and click a map obstacle line to add a new vertex to this line. Move the mouse pointer to where you want the vertex to be placed and click again to place the vertex in the new location.

3. Right-clicking a vertex removes the vertex from it's containing obstacle polyline.

4. Ctrl+L removes all obstacle polylines from the current map.

In addition, Possim enables the user to save maps to file, using File → Save (Ctrl+S), and to open maps, using File → Open (Ctrl+O).

4.2 Navigational strategy

In a 100% static and fully charted environment, robotic navigation is rather simple. At first one would give the robot its starting position, then its destination, and finally ask the robot to calculate the optimal route from start to destination and then to travel the route. This means that route calculations would only have to be performed once for every set of start- and destination points and the only travel time interruption would be doing occational positional corrections based on laser rangefinder readings. How does this change when the environment is only semi charted and contains dynamic objects? A main loop for traveling in a dynamic environment might look something like this:

1. Get laser rangefinder readings.
2. Update map with laser rangefinder readings.
3. Correct current position based on updated map.
4. Calculate optimal path from current position to destination.
5. Move towards destination.
6. If current position equals destination, exit loop, else go to top.

From this we can see that the computational complexity is severly higher than in the static environment. Where we earlier only had to calculate the route once, we now have to calculate it for every move. Although this might sound dramatic, section 4.3.4 presentes a set of techniques that can reduce this complexity somewhat, but in general we are still left with a main loop looking like this:

1. Recalculate own position.
2. Recalculate route.
3. Move towards destination.

And this is the navigational strategy that forms the basis for discussion in the next few sections.

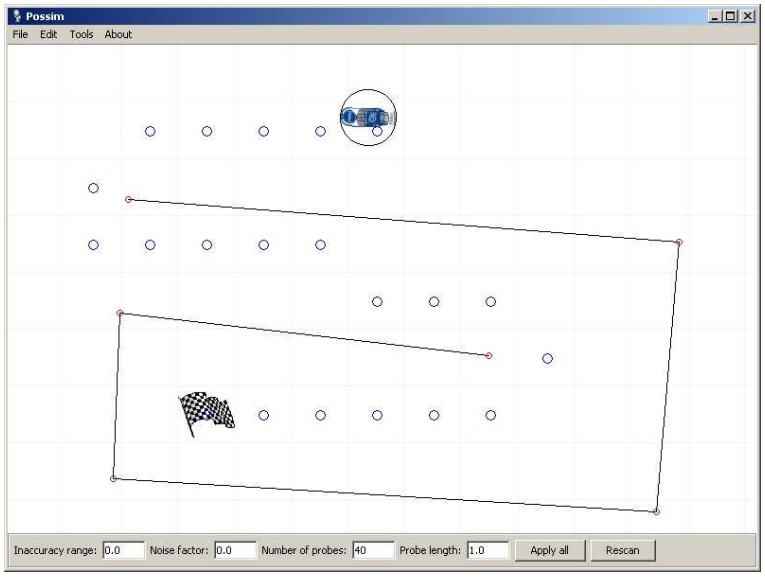


Figure 4.2: A tile route as calculated by the A-Star algorithm

4.3 Pathfinding

As discussed in section 2.5, pathfinding is the process of computationally trying finding a travelable route from point *A* to point *B* in a given map. While this process is trivial for humans, it has proven rather difficult to implement as a general and efficient algorithm. The most common algorithms employed for pathfinding today are *Dijkstra's algorithm* and the *A-Star algorithm*. Both of these algorithms divide the map into a grid of discreet tiles, similar to a chess board, and investigate each of a given tiles' neighbouring tiles in the direction of the destination to determine a travelable route. Both *Dijkstra's algorithm* and the *A-Star algorithm* are relatively complex algorithms and the fact that these algorithms operate in a discreet tile map forces us to create an approximation of the map that is more coarse grained than the original map to be able to calculate the route in reasonable time. The result of this approximation is a route that is suboptimal compared to what could have been achieved using a 1:1 map, since we now are limited to moving from tile to tile, instead of moving in the maps native units. Figure 4.2 illustrates such an example where it is clear that the route calculated by the *A-Star algorithm* is longer than what the route could have been is the robot was allowed to move

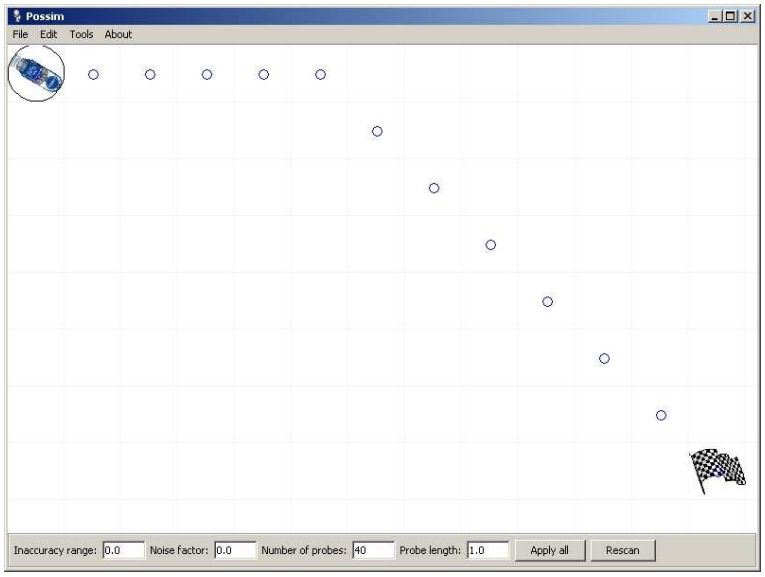


Figure 4.3: A suboptimal discreet tile route.

as close to the obstacles as allowed by the resolution of the map. Another negative impact of the tile approximation technique employed by Dijkstra's algorithm and the A-Star algorithm is that only horizontal, vertical or diagonal movement is calculated. The reason for this is that these algorithms consider the eight neighbouring tiles in turn to determine the single tile move that is most benefitial to achieving the shortest route from start to destination, and a single move from one tile to any of its neighbouring tiles can only result in horizontal, vertical or diagonal movement. Figure 4.3 shows how the combination of tile approximation and only horizontal, vertical or diagonal movement can lead to a significantly suboptimal route, compared to the route the robot could have followed if it was allowed to rotate directly towards the destination and travel there in a continuous straight motion.

The third and final drawback to the tile approximation technique used by the common pathfinding algorithms that we will discuss here, is the complexity involved in determining whether tile B is travelable coming from tile A. A very simple algorithm might assume that if tile A is travelable and neighbouring tile B is also travelable, it is possible to move from A to B. This is however not the case when factoring in the robots shape and size and the requirements as to how much the robot should be allowed to rotate at any point in the

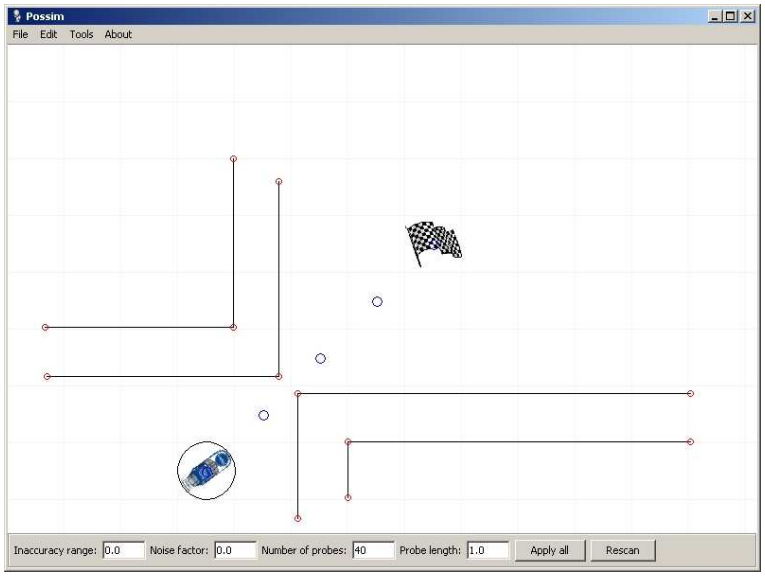


Figure 4.4: A tile route calculated without consideration to robot shape.

route. Figure 4.4 shows such a case where all the tiles from the start position to the destination are travelable, but when considering the robots shape and size it is clear that the route calculated is not travelable. To encompass this problem we need to extend the pathfinding algorithms callback function so that it answers not only "is tile B travelable?", but rather "is tile B travelable from tile A?" and within this extended function take the robots shape and size into consideration. Futhermore, for the pathfinding algorithms to find a route from tile A to tile N, there needs to be a continuous set of travelable discreet tiles available. It is not enough that there is an available area clearly big enough for the robot to pass through, since the pathfinding algorithms are not able to combine several semi obstacled tiles. Figure 4.5 illustrates this problem. We can see that the robot would be able to move from its starting position to its destination had it not been for the placement of the tiles, which unfortunately falls so that no continuous route of unobstacled tiles are available from start to destination.

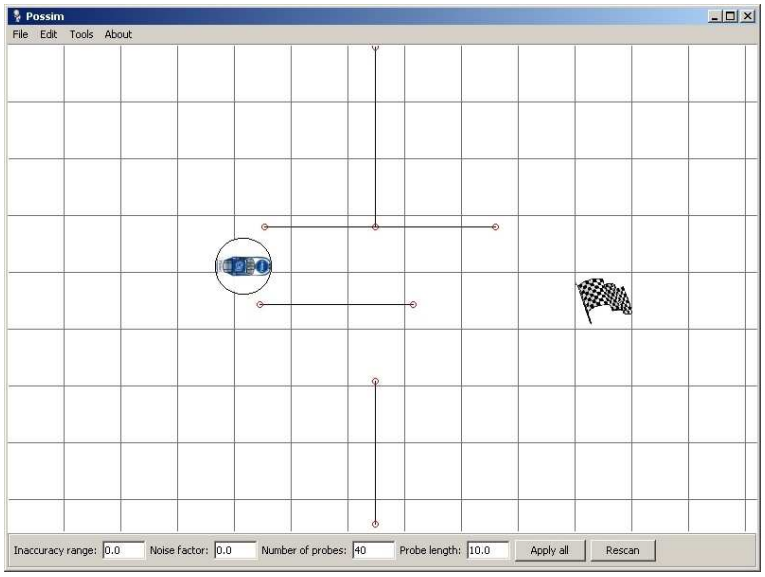


Figure 4.5: An untraveable route due to unfortunate tile placement.

4.3.1 Implementing the A-Star algorithm

In section 2.5.2 we described the A-Star algorithms history and basic logic. In this section we take a look at the inner workings of the algorithm and the complexities encountered when working with this algorithm.

The A-Star algorithm in itself is very elegant, well defined and independent from any specific technical features. Accompanying this thesis is a generalized implementations of the A-Star algorithm written in Java. The algorithm exposes a relatively simple interface, is takes a starting position, a destination and a callback function as its input parameters and return the optimal list of route points if a route can be found or *null* if there are no possible routes. The callback function takes a two-dimentional position as its input parameters and returns a *boolean* value to signal whether the specified position is travelable. The input object to the algorithm is a single object implementing the *AStarMap* interface. The definition of the *AStarMap* interface can be seen in figure 4.6.

Figure 4.7 shows a basic A-Star main loop similar to how it is implemented in Possim. We can see that the logic is fairly straight forward and that a program using this algorithm only needs to supply a small number of

```
1 import java.awt.Point;
2
3 public interface AStarMap {
4     public Point getStart();
5     public Point getDestination();
6     public boolean isWalkable(int x, int y, int parentX, int parentY);
7 }
```

Figure 4.6: The AStarMap interface

well defined methods; *getWalkableAdjacentNodes()*, *calculateManhattanDistance()*, *calculateEuclidianDistance()* and *createOrderedRoute()*. *calculateManhattanDistance()* and *calculateEuclidianDistance()* are self explanatory and *createOrderedRoute()* simply backtracks through the given Node objects *previous* references to find the route from start to destination. The more complex method is *getWalkableAdjacentNodes()*. This method tests all of *current*s neighbouring tiles to see if they are travelable from *current* and returns a list of those that are. This method calls *AStarMap.isWalkable()* to determine if a tile is travelable and as such all application-specific logic with respect to map scaling and robot shape and size is off-loaded to this method.

From figure 4.7 we can see that this implementation of the pathfinding algorithm uses two lists of open and closed *Node* objects. These Node objects extend the *java.awt.Point* object with three fields; g, h and f (see section 2.5.2 for a detailed explanation of these values). The open list contains Nodes that are still possible candidates for the final optimal route, while the closed list contains Nodes that have been found unusable by the algorithm. The first method called in the A-Star main loop is *openList.getNodeWithLowestF()*. This method returns the Node object from the open list with the lowest f value, indicating that this Node is the one currently being considered as the most probable to lead to the shortest route to the final destination, based on its actual distance back to the starting position and its estimated distance to the final destination. Using a general purpose container like an array or a list for the open and closed lists would mean a search operation would have to be performed to find the Node with the lowest f value for each iteration of the main loop. Using a more specialised container for the open list could avoid this search operation and we will look at this more closely in section 4.3.4.

In the beginning of this chapter we mentioned that the complexity of the A-Star algorithm necessitates a tile approximation of the map. In Possim this grid size is configurable and values between 50 and 100, meaning a simplifi-

```
1  public ArrayList<Point> aStar(AStarMap map) {
2    Point start = map.getStart();
3    Point destination = map.getDestination();
4
5    openList.add(new Node(start));
6
7    while (true) {
8      Node current = openList.getNodeWithLowestF();
9
10     if (current == null)
11       return null; // No possible route!
12     else if (current.x == destination.x && current.y == destination.y)
13       return createOrderedRoute(current); // Destination found!
14
15     openList.remove(current);
16     closedList.add(current);
17
18     ArrayList<Node> walkableAdjacentNodes = getWalkableAdjacentNodes(map, current);
19
20     for (Node adjacentNode : walkableAdjacentNodes) {
21       if(closedList.contains(adjacentNode))
22         continue;
23
24       if(!openList.contains(adjacentNode)) {
25         adjacentNode.g = current.g + calculateManhattanDistance(current, adjacentNode);
26         adjacentNode.h = calculateEuclidianDistance(adjacentNode, destinationPoint);
27         adjacentNode.f = adjacentNode.g + adjacentNode.h;
28         adjacentNode.previous = current;
29         openList.add(adjacentNode);
30       } else {
31         if (current.g + calculateManhattanDistance(current, adjacentNode) < adjacentNode.g) {
32           adjacentNode.g = current.g + calculateManhattanDistance(current, adjacentNode);
33           adjacentNode.h = calculateEuclidianDistance(adjacentNode, destinationPoint);
34           adjacentNode.f = adjacentNode.g + adjacentNode.h;
35           adjacentNode.previous = current;
36         }
37       }
38     }
39   }
40 }
```

Figure 4.7: A Java method illustrating the A-Star algorithms main loop.

cation of 50:1 to 100:1, have been found to be sensible values. Converting from the original map scale to the simplified tile scale utilizes integer division;

$$x_s = \left\lfloor \frac{x_o}{s} \right\rfloor$$

$$y_s = \left\lfloor \frac{y_o}{s} \right\rfloor$$

where x_o and y_o indicates the original two dimentional coordinate, x_s and y_s are the scaled x and y values and s is the tile/grid size. To convert back from the simplified tile scale to the center coordinate in the original map scale the following formula is used;

$$x_o = x_s s + \frac{s}{2}$$

$$y_o = y_s s + \frac{s}{2}$$

What does such a simplification mean with respect to computational complexity? Let us assume that the robot is positioned in $(220, 300)$ and wants to move to $(810, 550)$. Scaling this down to a grid size of 40 leaves us with the start position $(5, 7)$ and the destination $(20, 13)$ giving an area of $15x6$ tiles and a route of 6 diagonal and 9 horizontal moves, assuming an unobstacled area between start and destination. Since this route does not contain any obstacles, the algorithm can always use the neighbouring tile with the lowest f thereby limiting the number of calls to *AStarMap.isWalkable()* to 15. If we had not performed this scaling the *AStarMap.isWalkable()* method would have been called 340 times. This might not seem much, but as we will see in section 4.3.2 the calculations done inside *AStarMap.isWalkable()* are relatively computationally expensive and we therefore wish to limit the calls to this function to the absolute minimum.

4.3.2 Considering robot shape and size

As discussed in section 2.5.3 and illustrated in figure 4.4 it is important to consider practical implications when computationally determining whether the robot can move from point A to point B in a given map. A physical robot will always have a width and length and it might have specific constraints when it comes to the space needed to rotate. A significant part of the

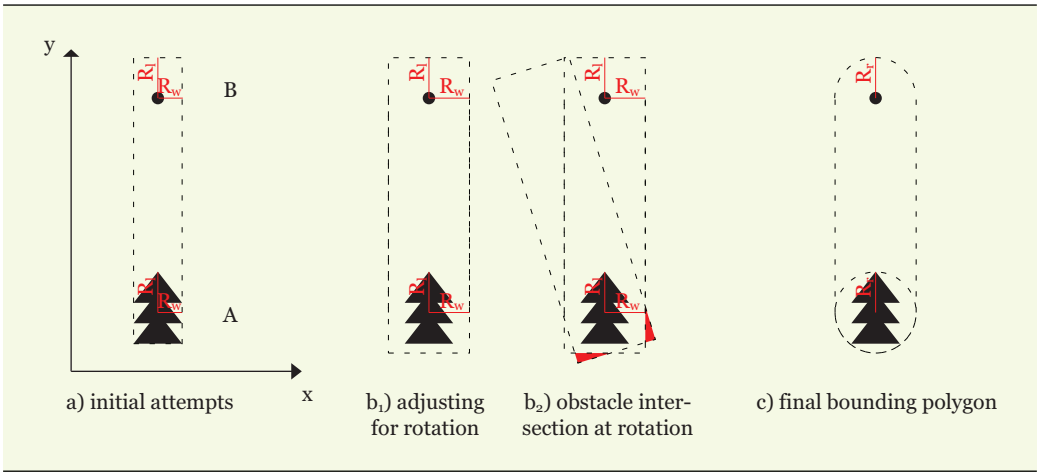


Figure 4.8: Various attempts at finding the best bounding polygon.

time writing this thesis has therefore been spent developing a robust, but efficient integration of the pathfinding algorithm and a *AStarMap* interface implementation that considers the robots shape, size and its demands with respect to rotation area.

The simplest implementation of *AStarMap.isWalkable()* might only take the destination coordinate as its input parameter and determine if any obstacles in the map intersects the specified tile and return the finding. A more advanced implementation would also need to know the source coordinate/start tile before answering whether the destination tile is travelable from the start tile. Only after getting the start and destination tiles and creating an internal representation of the area needed to move from start to destination and cross checking this area with the obstacle map, would the algorithm be able to determine whether the distance is travelable.

Initial attempts

At first I attempted to tackle the problem illustrated in figure 4.4. It is clear from this illustration that the robot has a certain width that needs to be taken into consideration when determining whether the next tile is travelable. Geometrically, this means that there needs to be an unobstacled rectangular area with width equal to the robot's width centered on the center

of point A, going to the center of point B. But this considers only the robots width. To consider its length as well, we need to extends the rectangle in each direction with half the robots length to ensure that the robot can stop perfectly centered in both the two points. Figure 4.8 *a*) illustrates such a rectangle and we can express the four corner point in the rectangle, starting in the top left corner and going clockwise, as follows;

$$x_1 = x_A - \frac{R_l}{2}, y_1 = y_A + \frac{R_w}{2}$$

$$x_2 = x_B + \frac{R_l}{2}, y_2 = y_B + \frac{R_w}{2}$$

$$x_3 = x_2, y_3 = y_B - \frac{R_w}{2}$$

$$x_4 = x_1, y_4 = y_A - \frac{R_w}{2}$$

Where x_A, y_A, x_B and y_B are the coordinates of points A and B, R_l is the robots length and R_w is the robots width. We call this the robots *bounding rectangle* from A to B. Note that these formulas assume an angle of 0 degrees from point A to point B. For other rotations, simply rotate the points x_1, y_1 to x_4, y_4 around point A by the angle from point A to point B, as explained in 2.2.3. After calculating this rectangle we can iterate through all the obstacle line segments in the map to see if any of them intersect any of the rectangle sides, or if any of the obstacle line segments are completely contained by the rectangle. If this is the case then the distance from point A to point B is not travelable. Initial testing showed that this worked quite well for avoiding situations where the the A-Star algorithm previously would report that the distance was travelable, but it was visually obvious that the move was impossible. However, what happens if the robot does not move from A to B in one continuous move, but stops and discovers an obstacle that neccessitates a rotation to continue towards the destination? Our simple strategy does not cover this and as such we might find the robot in a situation where it is not able to rotate and continue on its route. We therefore need to expand the shape and size considerations made in this section to also include the need for full rotation at any point in the route.

Adjusting for rotation

Incorporating the requirement for full rotation at any point actually makes the formulas in the previous section somewhat simpler. We no longer need to distinguish between robot length and robot width, we now only need to consider the robots rotational *radius*; $R_r = \frac{max(R_l, R_w)}{2}$, given that the robot can rotate 360° around its own center as is the case for the P3AT robot. We rewrite our formulas;

$$x_1 = x_A - R_r, y_1 = y_A + R_r$$

$$x_2 = x_B + R_r, y_2 = y_B + R_r$$

$$x_3 = x_B + R_r, y_3 = y_B - R_r$$

$$x_4 = x_A - R_r, y_4 = y_A - R_r$$

While this change makes the bounding rectangle larger and might thus limit the possible moves in the map for all cases, except where $R_l = R_w$, we now avoid the situation where the robot gets stuck because it cannot rotate. Figure 4.8 b_1) shows the extended bounding rectangle. Although this extension gets us a long way towards covering all considerations with respect to robot size when calculating travelable moves, it introduces one special corner case. What happens when the robot moves as close as possible in parallel with an obstacle and the robot needs to rotate? It now turns out that even if the robot would physically be able to rotate, our algorithms would indicate that the bounding rectangles corners would intersect the obstacle line segments, as illustrated in figure 4.8 b_2), and further processing would therefore stop.

Final adjustments

An extreme case of the scenario illustrated in the previous section is when the robot moves down a corridor with width $= 2R_r$. No matter how narrow a turn the robot tries to make, its bounding rectangle corners will intersect the corridor walls and the A-Star algorithm will report that the robot is stuck. To solve this limitation we need to promote the bounding shape used for intersection testing from a rectangle to a polygon shaped as a rectangle with one half circle in each end, as illustrated in figure 4.8 c). This way, the

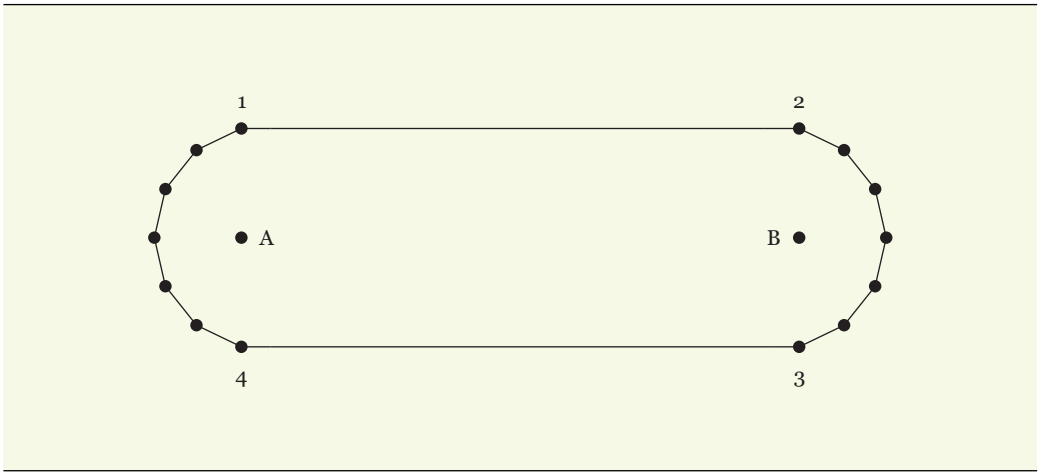


Figure 4.9: The final bounding polygon's individual vertices.

distance from the robot center to any of the bounding polygon sides can never exceed R_r and the *AStarMap.isWalkable()* method will never report any false negatives. In addition to points 1 to 4 in the previous section, we now need to insert these points between points 2 and 3;

$$x_{2n} = x_B + cos(\frac{\pi}{2} - \frac{n}{m}\pi)R_r, y_{2n} = y_B + sin(\frac{\pi}{2} - \frac{n}{m}\pi)R_r$$

And we need to insert the following points between point 4 and 1;

$$x_{4n} = x_B + cos(\frac{3}{4}\pi - \frac{n}{m}\pi)R_r, y_{4n} = y_B + sin(\frac{3}{4}\pi - \frac{n}{m}\pi)R_r$$

where m is number of points in the polygon half circle and $n \rightarrow [1, m-1]$. We call the resulting polygon the robots *bounding polygon* from point A to B. Figure 4.9 shows a simple representation of the final bounding polygon with it's individual vertices.

This final bounding polygon covers all the considerations we have to make with regards to the robots size and rotational area demands.

4.3.3 Optimizing paths using route climbing

Pathfinding algorithms such as Dijkstras algorithm and the A-Star algorithm consider only the eight neighbouring tiles of the current tile as possible route

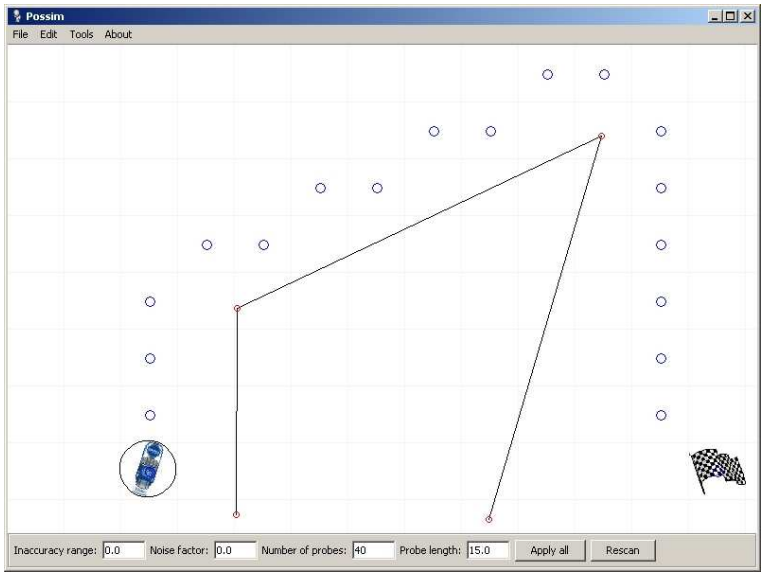


Figure 4.10: A tile route consisting of many unneeded rotations.

targets. This leads to a final route where every tile is at a multiple of 45° angle relative to its previous and next tiles. We call this the *continuous tile route*. For a robot capable of rotating in 1° increments such a route can be suboptimal with respect to both the travelled distance, as illustrated in figure 4.3, and to the number of stops and rotations needed to travel the route, as seen in illustration 4.10. From the route calculated in figure 4.10 it is clear that this route ideally could be optimized down to only two rotations, one at the top left corner of the obstacle polyline and one top right corner. This way we could both shorten the travelled distance and reduce the number of stops/rotations. How do we proceed to achieve such an optimized route?

First we need to consider what we want the resulting route to be; we want it to be a route free of redundant route points so that every route point in the optimal route can only be reached from its previous and next route points in the optimal route. These route points are considered *significant route points*. The optimal route is calculated from the continuous tile route, starting at the first route point and continuing to the last route point, assuming that these are the start and destination points respectively. To determine the significant route points in a continuous tile route we have developed a technique we have called the *route climbing* technique. The route climbing algorithm

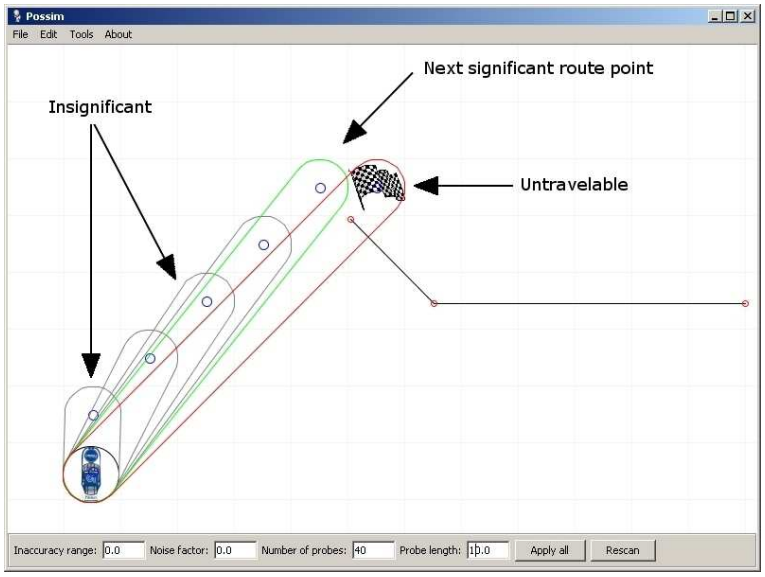


Figure 4.11: The route climbing technique illustrated.

applies parts of an obstacle avoidance technique known as *points-of-visibility*, described in [DeLoura 2001]. But instead of using points-of-visibility to do pathfinding on the actual map, we use the technique on the travelable tile route already found using the A-Star algorithm. The route climbing algorithm is defined in Algorithm 1, where p_{n-1} is the previous succeeding point of *current* with respect to p_n. p_{n-1} will always be defined if the line from *current* to p_n intersects an obstacle, since the distance from *current* to the immediate following p always is travelable in a well defined continuous tile route. More informally the route climbing algorithm can be thought of as probing ("climbing") every succeeding point of a significant route point until it finds a route point that is not directly travelable from the significant route point, it then sets the previous route point as the next significant route point and continues the algorithm from this point. Figure 4.11 illustrates the route climbing technique in action, with black bounding polygons to insignificant route points, a red bounding polygons to an untravelable point and the green bounding polygon from the starting point to its next significant route point. Figure 4.13 shows the resulting route after applying the route climbing technique to a full continuous tile route. Note that to determine whether a point p_n is travelable from *current* we employ the same bounding polygon as described

```
1.1  set current = the starting route point in the continuous tile route;
1.2  set destination = the final route point in the continuous tile route;
1.3  set significantRoutePoints = an empty list;
1.4  while current != destination do
1.5      add current to significantRoutePoints;
1.6      foreach point p_n after current in the continuous tile route do
1.7          if p_n == destination then
1.8              add p_n to significantRoutePoints;
1.9              set current = p_n;
1.10             break loop;
1.11         else if !isWalkable(current, p_n) then
1.12             set current = p_{n-1};
1.13             break loop;
1.14         else
1.15             continue to next p_n;
1.16         end
1.17     end
1.18 end
1.19 return significantRoutePoints;
```

Algorithm 1: The route climbing algorithm.

in 4.3.2. Figure 4.12 lists the route climbing algorithms as implemented in Possim.

4.3.4 Performance considerations

Introduction

Pathfinding is inherently complex, and although the pathfinding algorithms themselves are not very complex in terms of logic, they are computationally complex and thus time consuming. The worst case complexity of the A-Star algorithm is $n_f{}^2$, where n_f is the number of route points in the final route, and when considering the number of geometric operations we suggested earlier in this chapter applied to every one of those points, it is clear that a few optimalizations are needed to make this usable in a real world scenario. In this section we look at the most time consuming operations[1], what makes them complex and how we can reduce their complexity. Figure 4.14 shows

[1] The most time consuming operations have been identified using a profiler toolkit, revealing the execution time of the algorithms on a per method-level.

```
1  private ArrayList<Point> calculateOptimalRoute
2  (ArrayList<Point> route, Point start, ArrayList<Line2D> obstacleList, int gridRadius) {
3    ArrayList<Point> optimalRoute = new ArrayList<Point>();
4
5    int routeSize = route.size();
6    for (int i = 0; i < routeSize - 1;) {
7      Point currentPosition = route.get(i);
8
9      // Use actual robot position rather than first
10     // tile as starting point
11     if (i == 0) {
12       currentPosition = new Point(start);
13       optimalRoute.add(new Point(currentPosition));
14     }
15
16     for (int j = i + 1; j < routeSize; j++) {
17       Point nextPosition = route.get(j);
18
19       boolean onLastRoutePoint = (j == (routeSize - 1));
20       boolean canTravelWithoutCollision = GeometryUtils.canTravelWithoutCollision(currentPosition
             , nextPosition, gridRadius, obstacleList);
21
22       if (onLastRoutePoint && canTravelWithoutCollision) {
23         optimalRoute.add(new Point(nextPosition));
24         i = route.size();
25         break;
26       } else if (!canTravelWithoutCollision) {
27         Point previousNonIntersectingPoint = route.get(j - 1);
28         optimalRoute.add(new Point(previousNonIntersectingPoint));
29         i = j - 1;
30         break;
31       }
32     }
33   }
34
35   return optimalRoute;
36 }
```

Figure 4.12: The *route climbing* algorithm as implemented in Possim.

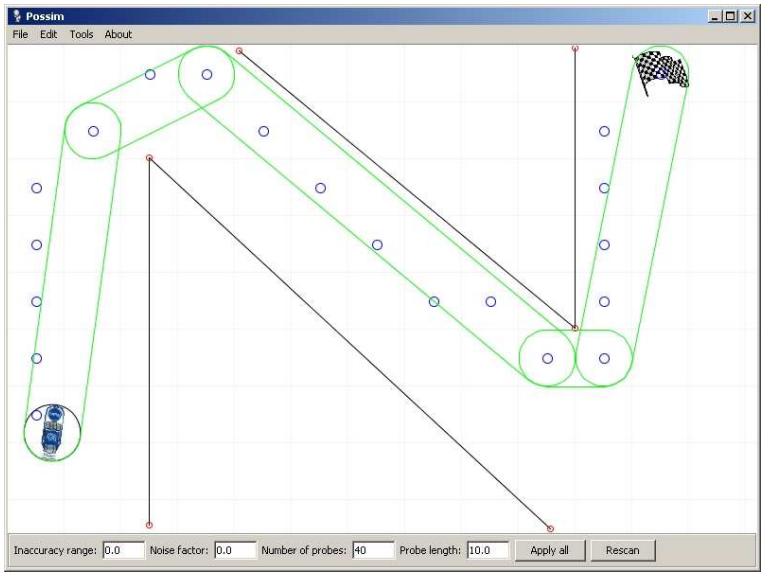


Figure 4.13: The route climbing technique applied to a longer route.

the maze test map used for testing the various performance optimizations suggested in this section. Informal testing is done by placing the robot in the bottom left-most tile and setting the destination as the bottom right-most tile and calculating the route ten times and considering the lowest run time, with and without the suggested optimalizations. The runtimes given in the following sections are calculated using a robot- and tile-size of 10 (i.e. a simplification of 10:1) and 60 (simplification 60:1) and calculating the route using these sizes and with no calculations takes 92500 ms (10) and 220 ms (60).

Avoiding duplicate calls to *isWalkable()*

From our earlier analysis we can see that the single most logically complex method that is being called during the pathfinding algorithm is the *AStarMap.isWalkable()* method that determines whether a tile is walkable from a specific tile. This method contains lots of trigonometric operations when calculating the movement bounding polygons and even a polygonal intersection test for each possible tile. Limiting the number of times this method is called is a good starting point for performance optimization.

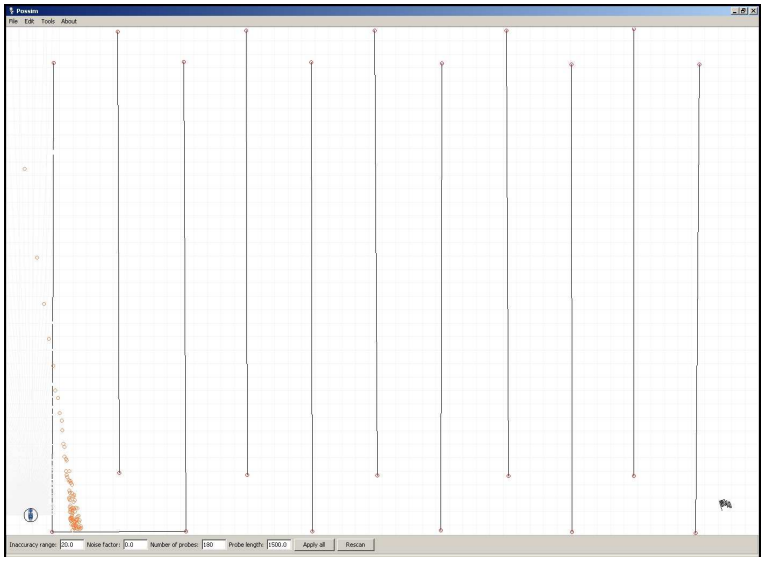


Figure 4.14: The maze test map used for performance testing.

From the A-Star implementations seen in 4.7 we see that the *getWalkableAdjacentNodes()* method is called to analyse the neighbouring tiles of *current* and return the tiles that are walkable. After calculating these tiles, the main loop iterates over the walkable neighbouring tiles, but skips the tiles that are in the *closed* list. This means that *isWalkable()* will be called for a lot of tiles that will simply be skipped later on because they are marked as being closed. Making *getWalkableAdjacentNodes()* aware of the list of closed tiles and avoid calling *isWalkable()* for the tiles in this list should improve performance quite a bit.

Testing gives a runtime of 56500 ms for tile size 10 and 156 ms for tile size 60, indicating runtimes now being 61% and 70% with respect to the original.

Improving open and closed tile collection access times

Avoiding duplicate calls to *isWalkable()* definitely helps, but profiling a route calculation shows that a considerable amount of time is spent iterating over the collections of open and closed nodes to find tiles matching certain criteria or to determine if a specific tile is contained within one of the collections. Our naive first implementation applied lists of type *java.lang.ArrayList* for the open

and closed lists, resulting in linear searches through the lists. By analyzing how the open and closed lists are used, we can apply better suited data structures for these collections, leading to more efficient route calculations.

The collection of open tiles is used in two ways;

- To extract the tile with the lowest estimated distance to the destination.
- To search for a given tile to see if it is contained within the collection.

From this we can see that the collection should be ordered by f (the destination distance heuristic) to enable $O(1)$ complexity when extracting the tile with the lowest f, it should have a unique primary key enabling $O(1)$ complexity when checking if the collection contains a specified tile and as both the collections holding open and closed tiles are constantly manipulated throughout the pathfinding algorithm, the data structures should offer fast object insertions and deletions. A data structure offering $O(1)$ complexity on retrieving the element that lies first in an ordered structure, $O(1)$ complexity on insertion and $O(log n)$ complexity on containment checking and deletion is the *binary heap*, specifically the *min heap*. The Java Collections Framework offers the *java.util.TreeSet* as a general implementation of the binary heap, guaranteeing maximum $O(log n)$ on *add()*, *remove()* and *contains()*. Although the *TreeSet* offers a fast implementation of *contains()*, this method checks for containment based on the sort criterion f, while we want to check for containment based on tile coordinates. Testing various solutions to this mismatch has lead us to a compromise, where we maintain two collections of open tiles; one *TreeSet* for fast extraction of the tile with the lowest f and a *java.util.HashMap* containing string concatenated tile coordinates as keys for $O(1)$ containment checking.

The collection of closed tiles is used in only one way; to check whether a specific tile has been visited. This requires fast containment checking and fast insertion. The *hash table* data structure is ideal for this as it offers $O(1)$ *get()* and *put()* operations. We have used The Java Collections Frameworks *HashMap* class for the collection of closed tiles.

Applying the *TreeSet* and *HashMap* duo for the collection of open tiles and a single *HashMap* for the collection of closed tiles reduces the runtimes on our test map to 142 ms for tile size 60 and 16250 ms for tile size 10. These are runtimes being respectively 64.5% and 17.6% of the original un-optimalized pathfinding algorithm runtimes.

Speeding up collision detection

Our performance tweaking so far has been quite successful, we have lowered the runtime of the pathfinding algorithm on our chosen test map from over 95 seconds to approximately 16 seconds, but there are still a few improvements we can implement. Re-profiling our application reveals the next performance bottleneck; 67% of the time it takes to calculate the route is spent within the *GeometryUtils.canTravelWithoutCollision()* method. This method takes a start position and a destination as parameters, calculates the bounding polygon of the move, as described in section 4.3.2, and checks if any of the obstacle lines in the map are contained within or intersects the bounding polygon. Looking closer at this method, the profiler shows that 17% of the time in *canTravelWithoutCollision()* is spent calling *GeometryUtils.getBoundingPolygon()* and the remaining time is mostly spent in *java.awt.Polygon.contains(java.awt.geom.Point2D)*, which makes sense since determining if a point lies within a polygon is a rather complex operation. If almost all the time in *canTravelWithoutCollision()* is spent calculating the movement bounding polygon and determining whether any obstacles intersects this bounding polygon, we need to seriously reconsider the approach used here. The following geometrical characteristics of the bounding polygon are known:

- It consists of a rectangular area with the start position and destination are placed in the middle of two parallel walls.
- The parallel walls are extruded to convex half circles, to encompass full robot rotation in these locations.

Two conditions with regard to the bounding polygon need to be true for the move to be considered walkable; no obstacle lines can intersect the edge of the bounding polygon, and no obstacle line end points can lie within the bounding polygon.

It seems that if we simplify the bounding polygon to a rectangle and two circles, we could simply check for containment or edge intersection on these three objects, which is dramaticly less expensive with respect to prosessing time than containment checking on a polygon. However, this is only simple for a rectangle with purely vertical and horizontal edges. So in the cases where the angle between the starting position and the destination is not 0°

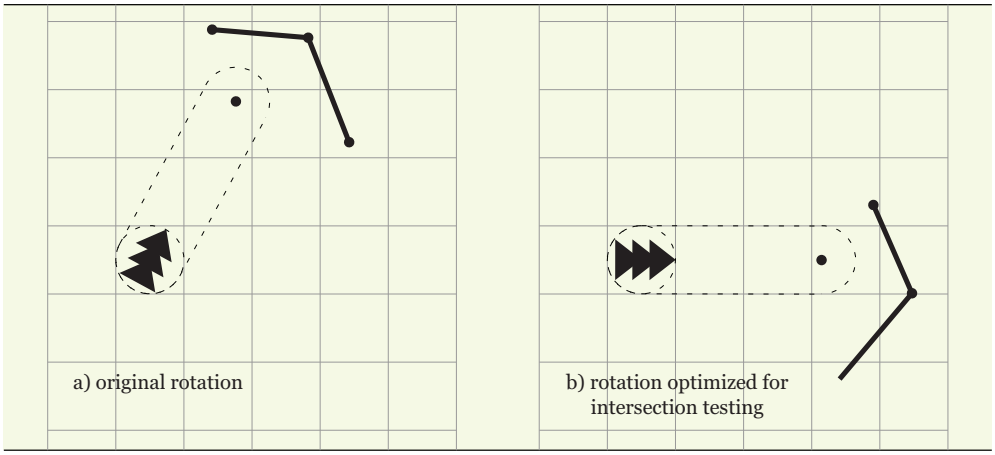


Figure 4.15: Optimizing bounding polygon intersection testing.

or a multiple of 90°, we need to rotate the rectangle, the destination circle and all the obstacle lines around the starting position by the negative of the angle between the positions, as seen in figure 4.15.

While rotating all the obstacle lines in the map may seem awkwardly complex, it is actually a considerably improvement from the previous method. On the test map used, the pathfinding algorithm now takes 20 ms for tile size 60 and 1840 ms for tile size 10. These are runtimes being respectively 9% and 2.18% of the original un-optimalized pathfinding algorithm runtimes.

Caching calculated routes

Having optimalized the pathfinding algorithm from a runtime of over 92 seconds down to below 2 seconds is a good improvement, but a 2 second pathfinding algorithm is still too timeconsuming to be run while the robot is moving. Looking back at section 4.3.3 however, we know that the route the robot actually moves by consists of only the "extreme" route points of the route originally calculated by the A-Star algorithm, that is the positions where the robot has to rotate. This also means that the next point in the optimal route at any point in time is as far as the robot is able sense at

that time. Knowing this, we can simplify the pathfinding algorithm that runs during movement to simply check if the distance from the current position to the next position in the route is walkable. If this distance is walkable, the robot can continue moving without recalculating the route. If the distance is not walkable, then the robot needs to recalculate the entire route. The movement logic is now:

1. Calculate route.
2. While current position != destination:
   (a) If the distance from current position to the next significant route point is travelable, move towards this route point and resync the pose.
   (b) Else, recalculate route.

This optimization is not as measurable as the earlier improvements we have made, but from experimenting in Possim it is clear that route caching can make a crucial difference when it comes to maintaining efficient flow in the robots movements.

Summary

Incorporating all the performance optimizations proposed in the preceeding sections leaves us with a lean and efficient pathfinding strategy. Figure 4.16 gives a visual overview of the impact of every optimization. The two figures at the left show the actual times spent calculating the routes in our test map, starting from tile size 10, going to tile size 60 in size increments of 5. The figures at the right show the approximated curves of the left hand size plots. The two upper graphs are shown with a linear time scale, while the two at the bottom are shown in a logarithmic time scale. From the logarithmic curve we can see that just skipping *isWalkable()* comes with the penalty of having to search through the list of closed tiles, which becomes significant for tile sizes smaller than 30. Adding the hashtable overcomes this and we can see that the optimizations are mostly linear, meaning that they don't lower the overall complexity of the pathfinding algorithm, rather they lower the runtimes of the steps involved.

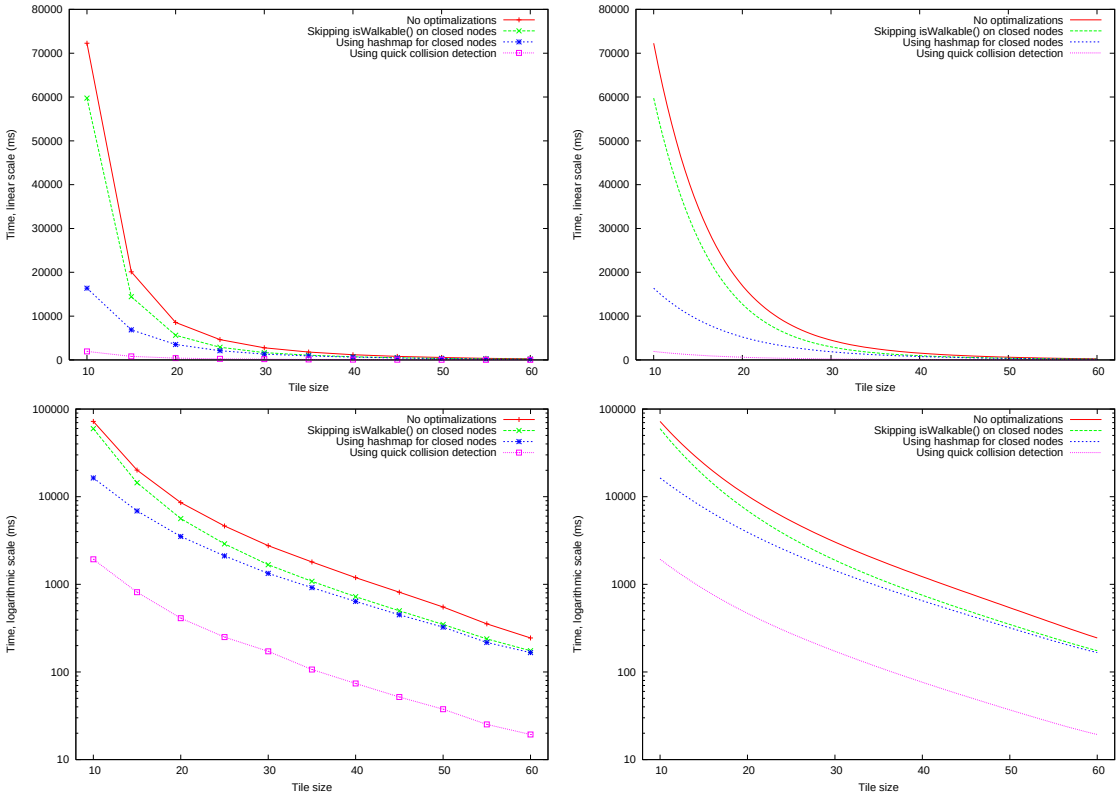


Figure 4.16: Graph of performance optimizations.
This graph shows the cumulative effects of applying the optimizations described in this section.

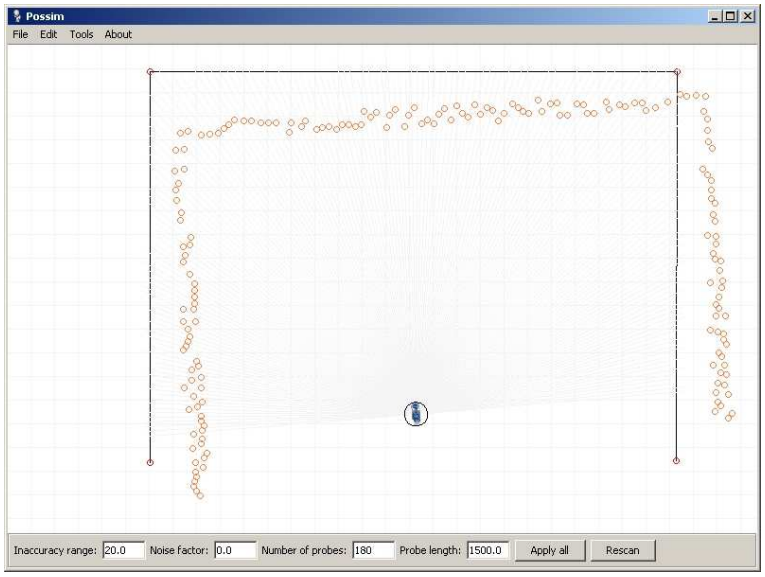


Figure 4.17: A typical laser rangefinder scan.
This laser rangefinder scan has been plotted on top of the robot's percieved map position.

4.4 Map Matching

4.4.1 Introduction

So far we have concerned ourselves with pathfinding and the algorithms needed to enable the robot to navigate within a known environment, assuming that the robot always makes perfect moves without any rotational or translational errors. This is, however, not the situation in real life. When the robot moves or rotates there will always be a small error factor in the amount actually moved or rotated. This can be due to inaccuracies in the robot software and hardware, or it can be caused by external factors such as slippery surfaces, gravel, carpets etc. To compensate for these inaccuracies we need to synchronize the robot's calculated position/orientation to its actual position/orientation after movement/rotation. To do so we need sensor-data to tell us what the environment around the actual robot looks like and this is just what a laser rangefinder can provide for us.

The laser rangefinder supplied with the ActivMedia P3-AT robot gives

a 180° scan in approximately 1° intervals directed in the robots forward direction and this is what we simulate in Possim. We can use the data from the rangefinder to create a detailed two-dimensional representation of the robot's environment and by matching this representation to a map, we can estimate the robot's actual position and rotation in the environment. While the readings from the laser rangefinder are significantly more accurate than the readings from the P3-ATs sonar rangefinder, we still observe some inaccuracies and noise in the data and the combination of inaccurate laser readings and errors in the robots movements makes for a complex pose estimation scenario. Figure 4.17 shows how a typical range scan looks like just after the robot has moved. The red circles indicate the individual laser readings and we can see from the readings that there is a slight error in pose as the readings are tilted left by a few degrees and that they have a horizontal and vertical offset to the map. This tells us that the robot is actually closer to the obstacles in the map than what its internal pose indicates and that its pose rotation should be a few more degrees to the right. Analysing 4.17 with respect to the problem of pose correction and map matching, we divide the problem into three subtasks:

1. Extract vector polylines from raw laser rangefinder readings.

2. Find rotation of the polylines that correlate best to the map.

3. Find translation of the rotated polylines that correlate best to the map.

4.4.2 Extracting vector polylines

To be able to do map matching of sensor rangedata to vector maps we need to convert the rangedata to a common format. As the laser rangedata is in reality an approximation of a larger structure, we choose to convert these sensor readings to more meaningful structures such as vector polylines. This is a two-step process, first we need to group the rangefinder readings into groups that are likely to correspond to structures in the map and secondly we approximate the polylines that these point groups resemble.

```
2.1  set k = the maximal distance allowed between two points;
2.2  set groups = an empty list;
2.3  set readings = the rangefinder readings;
2.4  while readings is not empty do
2.5  |   set r = the first reading in readings;
2.6  |   set group = an empty list;
2.7  |   call collectGroup(readings, group, r, k);
2.8  |   add group to groups;
2.9  end
2.10 return groups;
```

Algorithm 2: The recursive euclidean distance clustering algorithm.

Recursive Euclidean Distance Clustering

When trying to extract meaningful vector map data from the laser rangefinder readings, the first step is to group the readings by some sort of criterion, a process called *clustering*. We have chosen to group the readings by neighbouring euclidian distance. This means that that the two points p_1 and p_2 belong in the same group if the distance between p_1 and p_2 is less than or equal to threshold distance k, or if a segment S of one or more points exists between p_1 and p_2 with every point $\in S$, p_1, p_2 having one or more neighbouring points within a distance of no more than than k. This consept is easy enough to comprehend, but rather complex to implement as an algorithm without resorting to high runtime complexities. A naive implementation would be to iterate over all rangefinder readings and for each reading find all other readings within the distance k, and then finally iterate over every reading again to collect the readings that share common neighbours. However, this involves several $O(n^2)$ passes over the data and also demands a lot of memory heap space and we thus set out to find a more efficient solution. The algorithm we have come up with is a recursive solution that we have called *the recursive euclidean distance clustering algorithm (REDCA)*. This algorithm minimizes the number of passes through the rangerfinder data by removing readings from the original set as they are grouped with their related readings. *REDCA* is defined in algorithm 2.

collectGroup(), as defined in algorithm 3, is the actual recursive element in the REDCA algorithm and it takes the following parameters; the list of un-

```
3.1  add r to group;
3.2  remove r from readings;
3.3  set neighbours = the rangefinder readings in readings that are within k distance from r;
3.4  foreach neighbour in neighbours do
3.5  |   if group contains neighbour then
3.6  |   |   continue to next neighbour;
3.7  |   else
3.8  |   |   call collectGroup(readings, group, neighbour, k);
3.9  |   end
3.10 end
```

Algorithm 3: The *collectGroup* method of the REDCA algorithm.

grouped rangefinder readings (*readings*), the reading whose neighbours should be grouped (*r*), a list to add the grouped readings to (*group*) and the max distance threshold (*k*).

There are two complex operations in *collectGroup*, the identification of neighbouring readings in step 3, and the check to see if *group* contains a given neighbour in step 4a. The identification of neighbours is hard to optimize away, but its complexity steadily decreases as the size of *readings* decrease as more points are removed from it in step 2. The neighbour containment check on *group* is easily optimized down to $O(1)$ complexity by using a HashSet as the datastructure for the grouped points. This is perfectly acceptable as there is no need to maintain ordering of these points. In reality this comes with the penalty of having to implement a special subtype of *java.awt.Point* that allows points with identical values in the HashSet, but disallows objects to be inserted twice. Figure 4.19 shows how the *Recursive Euclidean Distance Clustering Algorithm (REDCA)* is implemented in Possim.

Polyline Approximation

After grouping related laser rangefinder data together, the grouped data needs to be approximated into polylines. The subject of fitting lines to two-dimensional data has been thoroughly covered in science and perhaps the most known method of doing this is the method of *least squares* as described in section 2.3.2. However, these line fitting techniques are often meant for fitting statistical data, where one has ascending x-values and corresponding y-values and one wants to find the ascending or descending tendency in these data.

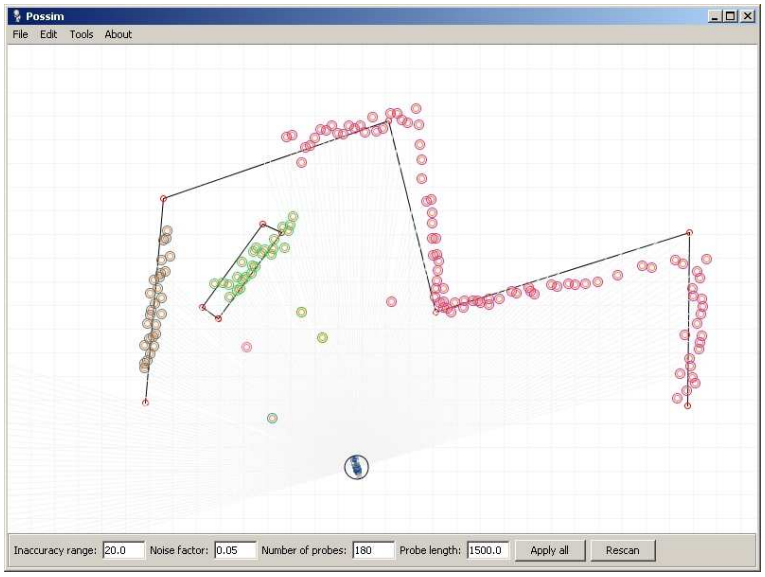


Figure 4.18: Laser rangefinder readings grouped by REDCA.

This does not match our scenario where we might have a 100% vertical line, consisting of 10 identical x-values with 10 different y-values. Applying the least squares line fitting algorithm to these rangefinder readings will, rather counterintuitively, result in a horizontal line in a height such as to minimize the sum of the vertical distances from the readings to the line. While this is perfectly correct from the least squares estimation standpoint, it is clear that this algorithm is not ideally suited to our specific problem.

The line fitting algorithm developed for this thesis is highly specific to the problem at hand. It takes the laser rangefinders position into consideration and, as such, is not very general, but it solves the problem we are facing quite efficiently. The algorithm executes the following steps to extract a polyline from the grouped rangefinder readings in *points*:

1. Sort the readings (ascending or descending) in *points* by their angle to the rangefinders position.

2. If there are consecutive readings in *points* with a distance greater than *maxDistance* between them, split *points* between these two points and treat the groups of points separately. [2]

[2]This is to handle the case where a line crosses the x-axis to the left of the rangefinders

```
1  public class PointGrouper {
2
3    public ArrayList<HashSet<Point>> groupPoints(ArrayList<Point> points, double maxDistance) {
4      // Create working list
5      ArrayList<GroupablePoint> pointList = new ArrayList<GroupablePoint>();
6      for (Point p : points) {
7        pointList.add(new GroupablePoint(p));
8      }
9
10     // Collect neighbouring points
11     ArrayList<HashSet<GroupablePoint>> groups = new ArrayList<HashSet<GroupablePoint>>();
12     while (pointList.size() > 0) {
13       GroupablePoint head = pointList.get(0);
14       HashSet<GroupablePoint> group = new HashSet<GroupablePoint>();
15
16       collectGroup(pointList, group, head, maxDistance);
17       groups.add(group);
18     }
19
20     // Map back to type ArrayList<HashSet<Point>>
21     ArrayList<HashSet<Point>> returnGroups = new ArrayList<HashSet<Point>>();
22     for (HashSet<GroupablePoint> hs : groups) {
23       returnGroups.add(new HashSet<Point>(hs));
24     }
25
26     return returnGroups;
27   }
28
29   void collectGroup(ArrayList<GroupablePoint> pointList, HashSet<GroupablePoint> group,
         GroupablePoint parent, double maxDistance) {
30     group.add(parent);
31     pointList.remove(parent);
32
33     for (GroupablePoint neighbour : getNeighbours(pointList, parent, maxDistance)) {
34       if (!group.contains(neighbour)) {
35         collectGroup(pointList, group, neighbour, maxDistance);
36       }
37     }
38   }
39
40   private ArrayList<GroupablePoint> getNeighbours(ArrayList<GroupablePoint> pointList,
         GroupablePoint point, double maxDistance) {
41     ArrayList<GroupablePoint> neighbours = new ArrayList<GroupablePoint>();
42
43     for (GroupablePoint p : pointList) {
44       if (p == point) {
45         continue;
46       }
47
48       if (point.distance(p) <= maxDistance) {
49         neighbours.add(p);
50       }
51     }
52
53       return neighbours;
54   }
55 }
56
57 class GroupablePoint extends Point {
58
59   GroupablePoint(Point p) {
60     super(p.x, p.y);
61   }
62
63   public int hashCode() {
64     return System.identityHashCode(this);
65   }
66
67   public boolean equals(Object o) {
68     return this == o;
69   }
70 }
```

Figure 4.19: The REDCA algorithm in Java.

3. Divide the list of points into sub-segments of size *testSegmentLength*. For each sub-segment, create a line from the first point in the segment to the last. Iterate over these lines and where two consecutive lines have a difference in angle of more than *maxAngleVariance*, a new point is added to the polyline at the coordinate corresponding to the center of the second sub-segment line.

```
4.1  set i = 0;
4.2  set testSegmentLength = desired test segment length;
4.3  set segmentStart = 0;
4.4  set segmentList = an empty list;
4.5  set previousAngle = 0;
4.6  while i ¡ (points.size() - testSegmentLength) do
4.7      set testSegment = line from points[i] to points[i + testSegmentLength];
4.8      set angle = angle(testSegment);
4.9      if i == 0 then
4.10         set previousAngle = angle;
4.11         increment i;
4.12     else if |previousAngle - angle| > maxAngleVariance then
4.13         add line from points[segmentStart] to points[i + testSegmentLength/2] to
             segmentList;
4.14         set previousAngle = angle;
4.15         set i = i + (testSegmentLength / 2);
4.16     else
4.17         increment i;
4.18     end
4.19 end
4.20 add line from points[segmentStart] to points[points.size() - 1] to segmentList;
```

Algorithm 4: The polyline approximation algorithm.

Step 3 in this approach contains the bulk of the polyline approximation technique and its pseudocode is shown in Figure 4 and the Possim implementation can be seen in Figure 4.20.

Figure 4.21 shows this polyline approximation algorithm applied in Possim. We can see that where the angle of a cyan-green line is significantly different from its preceding lines (sorted in clockwise order with respect to angle to

position, leaving the readings with an angle to the rangefinder around $-180°$ in the start of the list of sorted points and the readings with an angle to the rangefinder around $180°$ in the end of the list.

```
1  private static void splitActualLineSegments(ArrayList<Point> points, ArrayList<PossimLine>
       lineSegments, final double maxAngleVariance) {
2    int testSegmentLength = 10;
3    int segmentStartIndex = 0;
4    int finalSegmentStartIndex = points.size() - testSegmentLength;
5
6    double previousLineEstimatorAngle = 0;
7
8    for (int i = 0; i < finalSegmentStartIndex;) {
9      Point testSegmentStartPoint = points.get(i);
10     Point testSegmentEndPoint = points.get(i + testSegmentLength);
11     double lineEstimatorAngle = GeometryUtils.angleBetweenPoints(testSegmentStartPoint,
           testSegmentEndPoint);
12
13     if (i == segmentStartIndex) {
14       previousLineEstimatorAngle = lineEstimatorAngle;
15       i++;
16     } else if (Math.abs(previousLineEstimatorAngle - lineEstimatorAngle) > maxAngleVariance) {
17       int segmentEndIndex = i + (testSegmentLength / 2);
18
19       Point segmentStartPoint = points.get(segmentStartIndex);
20       Point segmentEndPoint = points.get(segmentEndIndex);
21       lineSegments.add(new PossimLine(segmentStartPoint, segmentEndPoint));
22
23       previousLineEstimatorAngle = lineEstimatorAngle;
24       segmentStartIndex = segmentEndIndex;
25       i = segmentEndIndex;
26     } else {
27       i++;
28     }
29   }
30
31   // Add final point
32   lineSegments.add(new PossimLine(points.get(segmentStartIndex), points.get(points.size() - 1)));
33 }
```

Figure 4.20: The polyline approximation algorithm in Java.

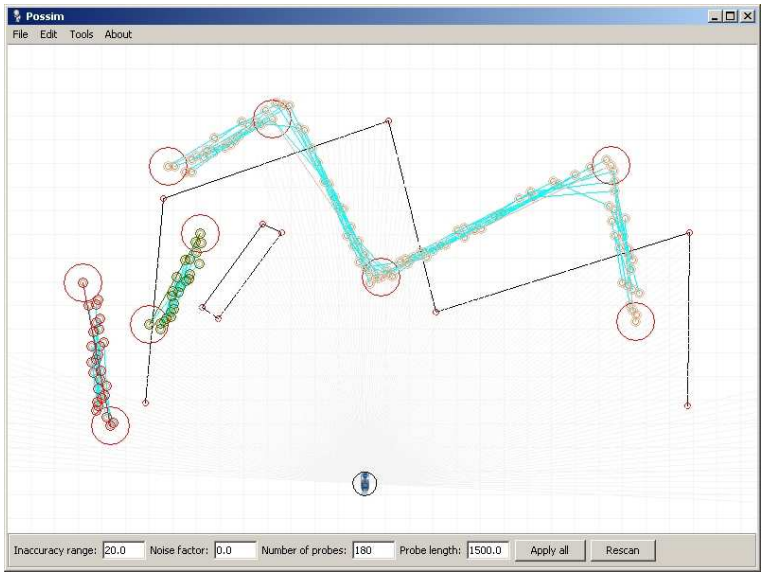


Figure 4.21: Approximated polylines in Possim.
Larger red circles shows corner points in the polylines and cyan-green lines show the sub-segments being used in the polyline approximation algorithm.

the robot), the line has been broken and a large red circle has been drawn to indicate where the polyline was split.

The proposed algorithm works very well when tuned to its working environment, but it does depend on the two variables *maxAngleVariance* and *testSegmentLength* to be adjusted to the rangefinders accuracy and resolution. We have found $maxAngleVariance = 30^\circ$ and $testSegmentLength = 10$ to be sensible starting values.

4.4.3 Correcting rotational error

After having converted the laser rangefinder data into more manageable vector data we can start the actual process of matching this data to our predefined map to determine the rotational and translational error the robot might have. Trying to do both things at once is too complex, so we have to split it into two tasks; to determine the rotation that makes the lines in our vectorized rangefinder data lie most parallel with the lines in the map and the translation of these rotated lines that minimizes the rangefinder data to

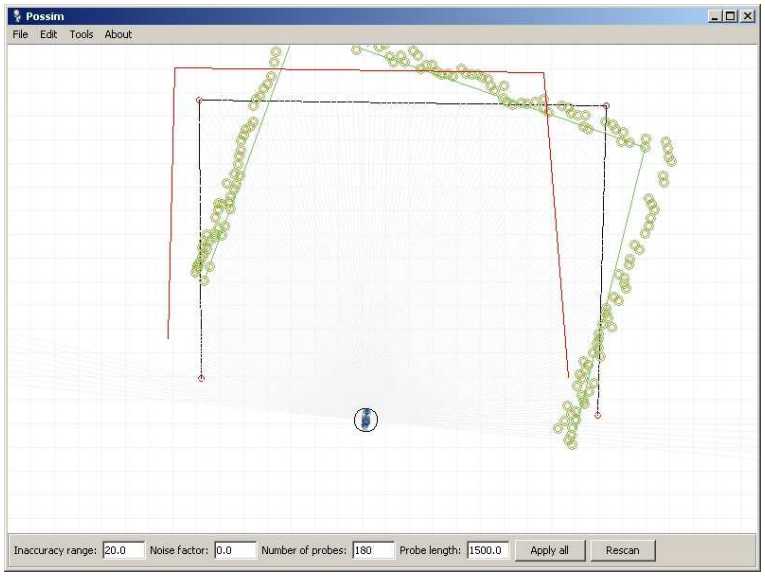


Figure 4.22: Calulating the best rotation of the laser rangefinder data. Green circles are the original rangefinder data and the red lines are the rotated line approximations.

map-line distances.

The algorithm proposed in this thesis for finding the best rotation scans through a number of rotations to find the angle where total *variance* is lowest. This variance is comprised of the rangefinder data line's lowest rotational difference with respect to a map line L, its distance from L, and its length with respect to the longest rangefinder data line. The main loop is described in algorithm 5, where θ is the range of angle rotations to try and *longestRangeFinderLine* is the longest of the vectorized rangefinder lines. The critical step when implementing this algorithm is determining the weighting of the variance-components. Optimal weighting needs to be tailored for the robot rangefinder combination. If the robot often has large rotational errors, one would weight the angle variance more, and if translational errors are more prominent, one would weight the distance variance more. A possible point of future work would be to see if the rotational error correcting algorithm could find and maintain this weighting dynamically. Figure 4.23 shows the algorithm as implemented in Possim and figure 4.22 shows this algorithm applied to a simple map. Notice the weighting applied on lines 44-50 in figure 4.23; angle

```
5.1  set bestAngle = 0;
5.2  set lowestTotalVariance = ∞.
5.3  foreach angle from −θ/2 to +θ/2 do
5.4      set totalVariance = 0;
5.5      foreach rangeFinderLine in the list of vectorized rangefinder readings do
5.6          set lowestVariance = ∞;
5.7          set rotatedLine = rangeFinderLine.rotate(angle);
5.8          set rotatedLineAngle = rotatedLine.angleBetweenEndpoints();
5.9          foreach mapLine in the map do
5.10             set mapLineAngle = mapLine.angleBetweenEndpoints();
5.11             set variance = |mapLineAngle - rotatedLineAngle| + (rotatedLine.length() /
                 longestRangeFinderLine.length()) + rotatedLine.distance(mapLine);
5.12             if variance < lowestVariance then
5.13                 set lowestVariance = variance;
5.14             end
5.15         end
5.16         set totalVariance += lowestVariance;
5.17     end
5.18     if totalVariance < lowestTotalVariance then
5.19         set bestAngle = angle;
5.20         set lowestTotalVariance = totalVariance;
5.21     end
5.22 end
5.23 return bestAngle;
```

Algorithm 5: The rotational error correcting algorithm.

variance is weighted 1:1 giving a range of 0 to 2π, line length is weighted as a ratio giving a range of 0 to 1 and distance variance is weighted $\frac{1}{1000}$ of its unit distance. These values have been found through empirical tests in Possim and should provide a good starting point for adaption to a particular rangefinder and robot.

4.4.4 Correcting translational error

From 4.22 it can be seen that we are now getting close to correcting the robots perceived pose to its actual pose. The robots heading have been synchronized, but we need to calculate how much the robot is off in the vertical and horizontal directions. To do this, we employ a similar technique as in 4.4.3, but rather than iterating over a range of angles, we here iterate over a range of values for horizontal and vertical translation, and consider the values that minimize the sum of polyline endpoint to closest map line distances as the best translation. In other words, we translate every line by a given set of values and try to find the translation where the combined distance from the translated lines endpoints to its closest map line is lowest. As with the the rotational error correcting algorithm, this is a linear search, but due to the relative low amount of rangefinder readings the only parameter to limit is the range over which to search. The default Possim implementation of this algorithm uses a window size of 100×100 and a step size of 4 to limit the number of iterations. Algorithm 6 details the algorithm pseudocode and figure 4.24 shows the algorithm as implemented in Possim.

From figure 4.25, showing the algorithm applied in a simulation, we can see the beige lines as polyline approximations of the rangefinder readings, the red lines show the same polylines after correcting for rotational error, and the blue lines after correcting for translational error. This figure sums up the result of applying the algorithms we have described in section 4.4 and shows that when properly tuned to the robot and rangefinder at hand, these algorithms can be quite successful.

4.5 Handling uncharted objects

For a robot to be able to navigate in a dynamic environment where objects move around and not every surrounding object is present in the robots map,

```
1  public static double getBestRotation(ArrayList<ArrayList<Point>> map, ArrayList<PossimLine>
       estimators, Point referencePosition) {
2    double bestRotation = 0.0;
3    double lowestVariance = Double.MAX_VALUE;
4
5    // Get rotation of lineestimator that gives the lowest variance with respect to map line
6    double angleRange = Math.toRadians(45);
7    double minAngle = -(angleRange / 2.0);
8    double maxAngle = (angleRange / 2.0);
9    double angleStep = Math.toRadians(1); // Try rotations in 1 degree steps
10
11   // Determine longest estimator line
12   double longestLineEstimatorLength = 0.0;
13   for (PossimLine lineEstimator : estimators) {
14     double lineEstimatorLength = lineEstimator.getLength();
15     if (lineEstimatorLength > longestLineEstimatorLength) {
16       longestLineEstimatorLength = lineEstimatorLength;
17     }
18   }
19
20   for (double angle = minAngle; angle <= maxAngle; angle += angleStep) {
21     double totalAngleVariance = 0.0;
22
23     for (PossimLine lineEstimator : estimators) {
24       // Rotate estimator points around referencePosition
25       Point lineEstimatorP1 = GeometryUtils.rotateAroundPoint(lineEstimator.getP1(),
             referencePosition, angle);
26       Point lineEstimatorP2 = GeometryUtils.rotateAroundPoint(lineEstimator.getP2(),
             referencePosition, angle);
27       double estimatorLineLength = lineEstimatorP1.distance(lineEstimatorP2);
28       double estimatorLineAngle = GeometryUtils.angleBetweenPointsPositive(lineEstimatorP1,
             lineEstimatorP2);
29
30       double bestEstimatorVariance = 1000000;
31
32       // Find map line with lowest variance at this rotation
33       for (ArrayList<Point> polyLine : map) {
34         for (int i = 0; i < polyLine.size() - 1; i++) {
35           Point mapLineP1 = polyLine.get(i);
36           Point mapLineP2 = polyLine.get(i + 1);
37           Line2D mapLine = new Line2D.Double(mapLineP1, mapLineP2);
38
39           double lineEstimatorP1Distance = mapLine.ptSegDist(lineEstimatorP1.x, lineEstimatorP1.y
                 );
40           double lineEstimatorP2Distance = mapLine.ptSegDist(lineEstimatorP2.x, lineEstimatorP2.y
                 );
41           double totalDistance = lineEstimatorP1Distance + lineEstimatorP2Distance;
42
43           double mapLineAngle = GeometryUtils.angleBetweenPointsPositive(mapLineP1, mapLineP2);
44           double variance = Math.abs(estimatorLineAngle - mapLineAngle);
45
46           // Weight longer lines more than short lines
47           variance /= 1.0 / (estimatorLineLength / longestLineEstimatorLength);
48
49           // Weight lines closer to their parallel line more than lines farther from their
                 parallel line
50           variance *= totalDistance / 1000.0;
51
52           if (variance < bestEstimatorVariance) {
53             bestEstimatorVariance = variance;
54           }
55         }
56       }
57       totalAngleVariance += bestEstimatorVariance;
58     }
59
60     if (totalAngleVariance > 0 && totalAngleVariance < lowestVariance) {
61       lowestVariance = totalAngleVariance;
62       bestRotation = angle;
63     }
64   }
65
66   return bestRotation;
67 }
```

Figure 4.23: The rotational error correcting algorithm in Java.

```
6.1  set lowestTotalDistance = ∞;
6.2  set bestXTranslation = 0;
6.3  set bestYTranslation = 0;
6.4  set startTranslation = -(windowSize/2);
6.5  set endTranslation = (windowSize/2);
6.6  foreach dx from startTranslation to endTranslation do
6.7      foreach dy from startTranslation to endTranslation do
6.8          set totalDistance = 0;
6.9          foreach approximatedLine do
6.10             set distanceToClosestMapLine = ∞;
6.11             set translatedLine = approximatedLine.translate(dx, dy);
6.12             foreach mapLine do
6.13                 set currentDistance = distance(translatedLine, mapLine);
6.14                 if currentDistance < distanceToClosestMapLine then
6.15                     set distanceToClosestMapLine = currentDistance;
6.16                 end
6.17             end
6.18             set totalDistance += distanceToClosestMapLine;
6.19         end
6.20         if totalDistance < lowestTotalDistance then
6.21             set lowestTotalDistance = totalDistance;
6.22             set bestXTranslation = dx;
6.23             set bestYTranslation = dy;
6.24         end
6.25     end
6.26 end
6.27 return bestXTranslation, bestYTranslation
```

Algorithm 6: The translational error correcting algorithm.

```
1  public static Point getBestTranslation(ArrayList<ArrayList<Point>> map, ArrayList<PossimLine>
       estimators) {
2    int windowSize = 100;
3    int stepSize = 4;
4
5    int startTranslation = -(windowSize / 2);
6    int endTranslation = (windowSize / 2);
7
8    double lowestTotalDistance = Double.MAX_VALUE;
9    int bestXTranslation = 0;
10   int bestYTranslation = 0;
11
12   for (int dx = startTranslation; dx <= endTranslation; dx += stepSize) {
13     for (int dy = startTranslation; dy <= endTranslation; dy += stepSize) {
14
15       double totalDistance = 0;
16
17       for (PossimLine estimator : estimators) {
18         Point translatedEstimatorP1 = new Point(estimator.getP1().x+dx, estimator.getP1().y+dy);
19         Point translatedEstimatorP2 = new Point(estimator.getP2().x+dx, estimator.getP2().y+dy);
20
21         double distanceToClosestMapLine = Double.MAX_VALUE;
22
23         for (ArrayList<Point> polyLine : map) {
24           for (int i = 0; i < polyLine.size() - 1; i++) {
25             Point mapLineP1 = polyLine.get(i);
26             Point mapLineP2 = polyLine.get(i + 1);
27             Line2D mapLine = new Line2D.Double(mapLineP1, mapLineP2);
28
29             double lineEstimatorP1Distance = mapLine.ptSegDist(translatedEstimatorP1.x,
                   translatedEstimatorP1.y);
30             double lineEstimatorP2Distance = mapLine.ptSegDist(translatedEstimatorP2.x,
                   translatedEstimatorP2.y);
31             double distanceToMapLine = lineEstimatorP1Distance + lineEstimatorP2Distance;
32
33             if(distanceToMapLine < distanceToClosestMapLine) {
34               distanceToClosestMapLine = distanceToMapLine;
35             }
36           }
37         }
38         totalDistance += distanceToClosestMapLine;
39       }
40
41       if(totalDistance < lowestTotalDistance) {
42         lowestTotalDistance = totalDistance;
43         bestXTranslation = dx;
44         bestYTranslation = dy;
45       }
46     }
47   }
48
49   return new Point(bestXTranslation, bestYTranslation);
50 }
```

Figure 4.24: The translational error correcting algorithm in Java.

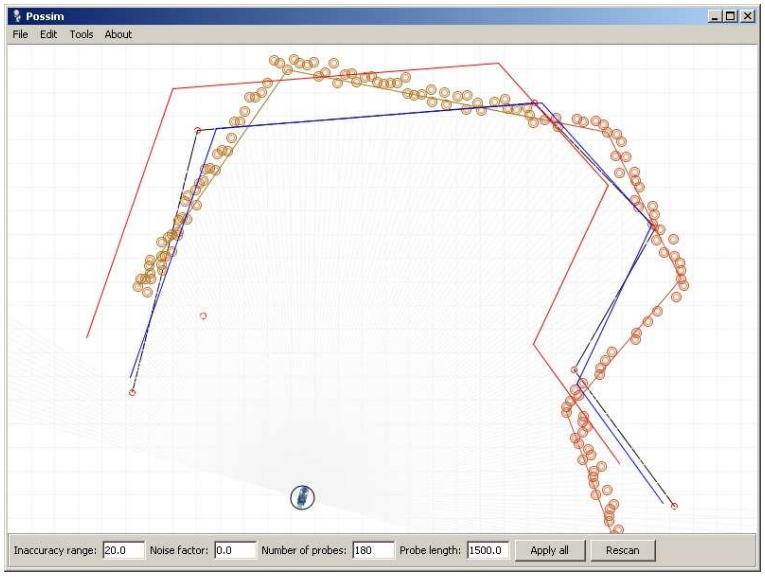


Figure 4.25: Calculating the translational error.

the robot needs to be able to detect un-mapped objects in it's laser rangefinder readings and take these objects into consideration when it calculates routes and moves along these routes. The subject of real time mapping and object detection has been covered in a number of research papers. In more recent years by Latecki and Lakämper in their papers [Latecki et al. 2004] and [Latecki et al. 2006]. Due to time constraints, this thesis will not cover handling uncharted object, it is rather suggested as a point of future work in Chapter 5.

4.6 Summary

In this chapter I have presented a strategy and set of algorithms suitable as a basis for autonomous robotic navigation. We have looked into the overall strategy for navigation and studied in detail the complex aspects of pathfinding, route optimization and map matching.

A number of measures to make the A-Star pathfinding algorithm usable in a near real-time environment have been proposed, including algorithmic optimizations such as using optimized data structures for the lists of open

and closed nodes and more functional optimizations such as partial route caching.

All the phases of map matching has been covered in detail. Starting off by grouping related laser rangefinder readings, then estimating the geometric primitives that these groups represent and finally matching these primitives to the robots internal map representation and thereby estimating the robots position and heading in the map.

In the process of writing this chapter, three new algorithms have been developed: the route climbing technique described in section 4.3.3, the recursive euclidian distance clustering algorithm for laser rangefinder data clustering described in section 4.4.2 and the polyline approximation technique presented in section 4.4.2. More thoughts on the significance of these algorithms are presented in the conclusion of this thesis.

Chapter 5

Conclusions

This chapter sums up the work presented in this thesis and a few thoughts on the process of writing it are presented.

5.1 On the development of this thesis

The initial focus, when starting this thesis, was to develop a set of algorithms and techniques needed for a working, stable navigation system for the ActivMedia Robotics P3AT robot belonging to the faculty of Computer Sciences at Østfold University College. The goal was to develop, deploy and test such an application on the robot and to submit a set of videos showing the software in action, as part of the thesis. The spring and summer 2006 was spent working with the P3AT. Interfacing with the C++ interface supplied with the robot was rather time-consuming and not enough time was spent on the theoretical aspects of developing the system. In hindsight it is clear that the intial problem statement was somewhat too broad for one person, without specific domain knowledge, to be able to complete in six months. I believe a subset of the initial problem, such as focusing on optimizing pathfinding algorithms or developing an optimal feature recognition algorithm for laser rangefinder data, would have been more appropriate. When I decided to revisit and complete this thesis in May 2008, one of the first decisions I made was to not focus on the actual P3AT, but rather develop a robot simulator mimicing the behaviour of the P3AT. This significantly simplified development and enabled me to focus on the interesting aspects of robotic navigation.

Secondly, I believe it would have been benefitial to have been two people writing this thesis. A lot of the discoveries I made along the way, have come when I have tried to explain or discuss a specific problem with a friend or colleague. This is a common effect, but it's importance should not be understated. Having a thesis partner would have also eased the process of getting an overview of the domain research available. The field of robotic navigation has been greatly researched and the amount of scientific papers available on this topic can be overwhelming.

5.2 The state of robotic navigation research

As stated in the previous section, a lot of work has been done in the field of robotic navigation. A plethora of research papers are available, as well as a number of books. This makes for a steep learning curve when trying to get an impression of the current state of the art, as it seems a lot of unrelated research papers are being published. As background for this thesis, summaries and surveys such as [Borenstein et al. 1996], [DeSouza, Kak 2002] and [Thrun 2002] have proved very valuable, but there is no de facto "bible" for robotic navigation that takes a qualitative evaluation of the different available techniques. This makes it difficult to choose which direction, statistical or geometrical, and what technique within that direction to pursue when setting out to implement a navigation system for a robot.

In contrast to the large amount of available literature on robotic navigation, very little actual source code is available. This further complicates the process of choosing a navigational strategy, as one can not simply download the source code and try it, one often has to thoroughly examine the relevant research papers and implement the proposed algorithms from scratch.

5.3 Conclusion

In the introduction to this thesis, I set out to develop a set of algorithms and techniques that enables a robot to navigate in a preloaded map, possibly containing dynamic and un-mapped objects. In section 3.5 I narrowed this problem statement down to four informal requirements. Of these requirements, requirement number one has been covered by the map matching

technique decribed in section 4.4 and requirement number two and three have been partially covered by the pathfinding and route optimization algorithms in section 4.3. When solving and implementing these requirements, it became clear that the problem of handling dynamic objects discovered during robot operation was too complex to be covered in this one semester thesis. Requirement number four, and the dynamic aspect of requirement two and three, are therefore not covered in this thesis.

In terms of contributions to the field of robotic navigation and route planning, this thesis contains three significant sections:

- The route climbing technique described in section 4.3.3. This algorithm, combined with the robot's bounding polygon as described in 4.3.2, greatly improves the efficiency, in terms distance travelled and rotations made, on a tile route as calculated by the A-Star or Dijkstra algorithm.

- The structured analysis of the A-Star algorithm optimization in section 4.3.4. Although a lot has been written on the A-Star algorithm, I have not been able to find a structured analysis on how to significantly increase its performance. Hopefully this section will be of use to future implementors.

- The recursive euclidean clustering algorithm described in section 4.4.2. This two-method, one variable (max distance between points in a cluster), and $O(n \log n)$ clustering algorithm is ideally suited to clustering rangefinder readings to groups that represent continuous objects in the map.

To summarize, this thesis presents a good toolbox of algorithms for getting started with pathfinding and route planning for mobile robots. The subject of efficient pathfinding has been throughly covered, and all the discussed algorithms are available as part of this thesis.

With respect to the problem of map matching and robot localization/-position determination, the algorithms presented in sections 4.4.3 and 4.4.4 work well for low complexity scenarios, but their high runtime complexity and somewhat inadequate handling of noise and small objects might make them unsuitable for very noisy real world usage. It is my impression that the system presented by Latecki and Lakämper in [Latecki et al. 2004] and [Latecki et al. 2006] is better suited for such noisy scenarios.

5.4 Further work

As pointed out in the previous section, the subject of map building and handling a dynamic environment is not covered in this thesis due to time constraints, and this would be an interesting point of further work. By first implementing Latecki and Lakämper's map building technique from [Latecki et al. 2004], one could possibly extend this technique with a time aspect, so that one could track moving objects in the map and plan routes that avoid the moving object by calculating it's trajectory.

Another interesting scenario is the possiblity of extending map matching to three dimensions. In the background chapter of this thesis, we briefly touch upon a few techniques for surface smoothing and edge detection. If the robot was fitted with one or more cameras, it could photograph it's surroundings, extract the edges/lines in the photograph and compare these line tendencies to a two-dimensional projection of a onboard three-dimensional model of it's surroundings and thereby calculate it's own position in the model. Some related work can be found in [David et al. 2003].

Bibliography

[Dijkstra 1959] Dijkstra, E. W.
A note on two problems in connexion with graphs
Numerische Mathematik 1, 1959

[Borenstein et al. 1996] Borenstein, J.; Everett, H. R.; Feng, L.
"Where am I?" - Sensors and Methods for Mobile Robot Positioning
The University of Michigan, April 1996

[Thrun et al. 1999] Thrun, S.; Bennewitz, M.; Burgard, W.; Cremers, A. B.; Dellaert, F.; Fox, D.; Hähnel, D.; Rosenberg, C.; Roy, N.; Schulte, J.; Schulz, D.
MINERVA: A Second-Generation Museum Tour-Guide Robot
Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

[Flato 2000] Flato, E.
Robust and Efficient Construction of Planar Minkowski Sums
Master thesis, Department of Computer Science, Tel-Aviv University 2000

[DeLoura 2001] DeLoura, M.
Game Programming Gems 2
Charles River Media, 2001

[Lo et al. 2002] Lo, S. C.; Akos, D.; Houck, S.; Normark, P. L.; Enge, P.
WAAS Performance in the 2001 Alaska Flight Trials of the High Speed

Loran Data Channel
IEEE Position Location and Navigation Symposium, 2002

[Thrun 2002] Thrun, S.
Robotic Mapping: A Survey
School of Computer Science, Carnegie Mellon University, February 2002

[DeSouza, Kak 2002] DeSouza, G. N.; Kak, A. C.
Vision for Mobile Robot Navigation: A Survey
IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 24, No. 2, February 2002

[David et al. 2003] David, P.; DeMenthon, D.; Duraiswami, R.; Samet, H.
Simultaneous Pose and Correspondence Determination using Line Features
Department of Computer Science, University of Maryland, 2003

[Latecki et al. 2004] Latecki, J. L.; Lakaemper, R.; Sun, X.; Wolter, D.
Building Polygonal Maps from Laser Range Data
ECAI Int. Cognitive Robotics Workshop, Valencia, Spain, August 2004

[Wing et al. 2005] Wing, M. G.; Eklund, A.; Kellogg, L. D.
Consumer-Grade GPS Accuracy and Reliability
Journal of Forestry, Volume 103, Number 4, June 2005

[Latecki et al. 2006] Latecki, J. L.; Lakaemper, R.
Polygonal Approximation of Laser Range Data Based on Perceptual Grouping and EM
IEEE Int. Conf. on Robotics and Automation, Orlando, Florida, May 2006

Appendix A

Included files

Included with this thesis document is a CD with the full source code for Possim application described in chapter four, as well as a set of videos showing how Possim works and how to use it.

Possim/possim.jar

This runnable jar file contains a precompiled version of Possim, ready to be run from the CD.

Possim/possim-src.jar

This jar file contains the full source code for Possim, complete with all the algorithms presented in this thesis.

Videos/

This folder contains a set of commented videos of Possim in action. It is a good idea to see these videos before trying to run Possim, to get an idea of what the application can do and how to control it.