School of Electronics and Computer Science
Faculty of Engineering, Science and Mathematics
University of Southampton

Author: Christopher Patuzzo
(cp22g08@ecs.soton.ac.uk)

Date: April 27, 2011

Simulated Autonomous Exploration using Object-Detector Technology

Project supervisor: Dr Alex Rogers
Second examiner: Dr Seth Bullock

A project report submitted for the award of
BSc Computer Science

Abstract

It is often required that a robotic agent has an understanding of its environment, to complete its tasks efficiently. To create such an understanding, a process of exploration is required. Real-world phenomenon, such as noise and collisions should be considered as a part of this process. This report explains the implementation of a simulation for an obstructed environment containing a robot. Extensive software testing is carried out on the simulation to imply its validity. Multiple explorative strategies are developed, ranging from simplistic, hierarchical approaches to complex data-driven exploration. A formal process for the evaluation of strategies is introduced and explained. It is concluded that the 'implied quadrants' strategy performs the best for most applications, although greater analysis of the importance of criteria for a given application may yield different results.

Contents

Chapter 1

Introduction

1.1 Project description

1.1.1 Problem

Robots are often required to complete tasks within an obstructed environment. If the robot is unfamiliar with the environment, then it may have to map it out to complete its tasks efficiently. If the robot has limited resources, the success or failure of its operation may be dependent upon how rapidly and accurately it can learn its surroundings.

If we are to assume that the robot is capable of movement and has some kind of object-detection technology, such as RADAR, SONAR or LIDAR, then it is possible for the robot to gain an understanding of its surroundings [14]. To accomplish this efficiently, some level of artificial intelligence is required to control the movement of the robot and process feedback from its object-detector.

1.1.2 Goals

Building the simulation

The project aims to simulate an environment containing a robot. The simulation should be somewhat isomorphic to a real-world scenario, such that the robot's actions are affected by noise, and collisions are handled appropriately. Noise within the system is modelled by a Gaussian distribution, due to its many constituent origins [5].

Noise compensation

For the robot to succeed, it must first attempt to compensate for noise within the model. Kalman filtering is implemented, which is a statistic approach frequently used in the field of electronics and signal processing [10]. A 'beacon' entity is introduced to allow the robot to localise itself within the environment.

Develop explorative strategies

Once the robot is able to compensate for noise within the model, the project considers different strategies for the robot to explore its environment. The explorative strategies range from simplistic hierarchical approaches [4], to more complex, data driven exploration.

Analysis of strategies

The process of rating strategies is difficult without prior understanding of the context in which the strategies are to be used. The robot may be required to gain a highly detailed understanding of its environment, or a more complete, overall understanding, albeit of less detail. Therefore, the analysis of strategies aims to consider different applications of the robot, as opposed to focussing on one set of optimisation parameters. It seems reasonable that one would consider the merits and drawbacks of each strategy when choosing which to implement for a particular application.

1.2 Approach

1.2.1 Languages and packages

C++

The simulation is written in C++. The software is responsible for displaying a graphical animation as well as processing custom artificial intelligence. C++ is a fast, highly optimised language, making it an ideal choice. Another reason for choosing C++ is that real-world micro-controllers are commonly programmed in C-based languages. Therefore, if the software were to be ported to a real-world micro-controller, it could ease the transition.

OpenGL

The project uses OpenGL in addition to the OpenGL Utility Toolkit. The Open Graphics Library provides all of the drawing and animation functionality the project requires [1]. It is widely supported, cross-platform and interfaces readily with C++. The graphical output is predominantly two-dimensional, however, it is convenient to have the option to expand into three dimensions as an extension. Microsoft Visual Studio is used as the development IDE as it is the recommended tool for OpenGL development on Windows.

1.2.2 Development process

Waterfall

The project uses a modified version of the Waterfall software development process. The specific phases for this project are ‘Management’, ‘Design’, ‘Implementation’, ‘Testing’, ‘Empirical evaluation’ and ‘Conclusions’. The modification in the waterfall model arises from the iterable characteristic of the implementation and testing phases. Constant unit testing is carried out during the implementation stage as new features are introduced.

Justification

One reason for choosing the Waterfall development process is that it places an emphasis on documentation. The project is relatively complex, in that it incorporates a graphical animation with custom artificial intelligence. Therefore, it is crucial to have a clear understanding from the outset to ensure that design decisions are not changed late into the project.

Another reason for choosing the Waterfall development process is that it is well suited to projects with unchanging requirements. Generally speaking, research-based projects are not intended for a specific end-user. Therefore, the requirements are unlikely to change and the linear approach of the Waterfall model, provides an adequate foundation for the development process.

1.2.3 Test strategy

Unit testing

Unit testing is carried out constantly throughout the implementation stage. This consists of plenty of textual output from functions, to validate their behaviour as well as tracing variable values through Visual Studio's debugging tool. Unit testing is used in the project to ensure that individual modules are operating correctly.

User-acceptance testing

User-acceptance testing is carried out to ensure that the project adheres to its detailed requirements. The correspondence of each test will be evaluated by reasoning and deduction rather than formal proof. This allows for the functional and non-functional requirements to be evaluated using the same approach.

1.2.4 Management strategy

Gantt charts

An initial, intermediate and final Gantt chart have been produced for the project, appendix figures 1 through 6. Gantt charts help to plan work flow throughout the project and evaluate progress. It is especially useful for a project with a lot of dependent stages, such as this one.

Project reviews

Regular project reviews also help to manage the project. These check that the project is on track and ensure that the required work level is maintained. If any unforeseen circumstances arise, the project reviews help to resolve them. Both individual project reviews and supervisor meetings are carried out as management for the project.

1.3 Risk management

Throughout the project, there may be a number of problems that arise. These could be due to technical issues, or may be more general. These problems could be detrimental to the success of the project; therefore it is logical to plan in advance, in case such problems arise.

1.3.1 Problem identification

1. A problem may arise from a compromised development environment. This could be due to theft, or hard drive failure. In either case, it would delay the project as it would take time to set up a new machine.

2. A problem may arise from a breakdown in project management. This could be due to a change in supervisor, whose approach to project management is significantly different.

3. A problem may arise if there is unforeseen difficulty in integrating the object-oriented programming style with OpenGL and GLUT. These libraries are generally modelled on state machines.

4. A problem may arise if the software runs very slowly and has delayed response times due to the complexity of displaying the graphical animation as well as processing the custom artificial intelligence.

5. A problem may arise if it is difficult to define a quantifiable method to rate each explorative strategy's performance due to the wide range of possible applications.

6. A problem may arise if the implementation takes far longer than expected to complete. This could be due to an unrealistic understanding of programmer capability.

7. A problem may arise if an important requirement is overseen. If it is discovered late into the project, then it could be costly in time.

1.3.2 Risk table

Table 1.1 shows the risk table containing the problems identified previously. The 'Severity of Occurrence' column is a predicted value between zero and one that indicates how crippling an effect the problem would have on the project. A value of zero indicates no effect. These values are multiplied by the estimated 'Probability of Occurrence' for each problem to obtain the 'Calculated Risk' values.

Problem Number	Severity of Occurrence	Probability of Occurrence	Calculated Risk
1.	20%	1%	0.2%
2.	35%	5%	1.75%
3.	15%	10%	1.5%
4.	10%	20%	2%
5.	40%	5%	2%
6.	50%	15%	7.5%
7.	15%	10%	1.5%

Table 1.1: Estimated risk table. The highlighted rows are the problems with the highest calculated risk.

1.3.3 Prevention and recovery

Problem 4 was that of slow running software, with a delayed response. A preventative measure that could be taken to help with this problem would be to use a graphics library that has been extensively optimised, such as OpenGL. If the preventative measure is followed, and the problem still arises then a possible recovery measure would be to share the simulation's processing amongst threads.

Problem 5 was that it may be difficult to define a quantifiable method for rating the performance of explorative strategies. A preventative measure that could be taken to help with this problem is to consider this factor in the design of the model that will store the environment data. If the preventative measure is followed and the problem still arises, then a possible recovery measure would be to reason informally about the success and failure of each explorative strategy through observation.

Problem 6 was that the implementation of the project may take much longer than expected to complete. A preventative measure that could be taken to help with this problem is to start implementation early to allow as long as possible for this stage. If the preventative measure is followed and the problem still arises then a possible recovery measure would be to reduce the scope of the project.

1.4 Detailed requirements

1.4.1 Functional

1. The simulation must be able to represent environments of varying dimensions.
2. Environments must have the capability to store a list of obstacles.
3. The software must simulate a robot within the obstructed environment, with an API that may be used by each explorative strategy.
4. Additive white Gaussian noise must be applied on each robot movement.
5. Collisions must be handled appropriately such that the robot may not intersect objects.
6. An object-detector must be simulated that measures the distance to the nearest obstacle at a given angle.
7. The behavioural module must attempt to compensate for noise within the system.
8. An internalised representation of the environment must be maintained for the robot.
9. The robot must be able to interface with different strategies for exploring the environment.
10. When the simulation window is resized, its aspect ratio must be the same.
11. A graph window must be displayed showing the current progress of each explorative strategy.
12. The simulation must be able to use a configuration file to load environments and configure many aspects of the system.
13. The capability to automate testing must be provided as part of the configuration file.
14. The simulation must provide the capability to toggle different data elements to be displayed.
15. The user must be able to pause the simulation at any time and toggle the current display elements.

1.4.2 Non-functional

16. The simulation should run at near real-time speed. The complexity of its underlying operations should support this requirement.

17. The simulation should be frame-rate independent.

18. The system should be object-oriented for purposes of extensibility and reuse.

19. The software code should be well-commented such that it is accessible to other developers.

20. The explorative strategies should be sufficiently decoupled such that they may be modified with ease.

1.4.3 Costs, benefits and constraints

This section considers a few implications of the requirements in the context of their impact on the project. Requirement 4 states that noise must be applied on robot movements. The benefit of enforcing this requirement is that the simulation is more realistic as a robot in real scenario would be susceptible to noise. A cost of enforcing this requirement is that of complexity. Each robot movement requires a random value to be generated based on a Gaussian distribution. This places a constraint on the rate of moves of the simulation.

Requirement 7 states that the behavioural module must attempt to compensate for noise within the system. The process of noise compensation considers several sensory measurements at once, and therefore requires that previous data be kept in memory. If the simulation runs for an extended duration, the memory of the local machine could become expended. This places a constraint on the duration for which the simulation may run. On modern machines, the effect of this constraint is likely to be minimal due to the large quantities of memory they contain.

Requirements 10, 11, 14 and 15 are all related to the windowing interface, display of graphical elements and program flow of the simulation. An obvious benefit of including these requirements is that the simulation becomes more interactive for the end-user, improving interest, and the ability to assess the progress of an explorative strategy in greater detail.

Requirement 16 states that the simulation should run at near real-time speed. A cost of enforcing this requirement is that additional time must be spent to ensure the complexity of underlying processes is sufficiently minimal for the simulation to run efficiently. This places a constraint on the maximum complexity of operations in the system. A benefit of enforcing this requirement is that testing of the explorative strategies may be completed relatively quickly.

Requirement 16 states that the simulation should be frame-rate independent. A cost of enforcing this requirement is that the main display loop for the graphics engine becomes more complex as the time between frames must be passed to the relevant update functions of the robot and behaviour classes. A benefit of enforcing this requirement is that the simulation is more realistic due to real-world scenarios being continuous in time.

1.5 Class hierarchy

Figure 1.1 shows the class hierarchy diagram for the larger modules within the system. The 'Display' class is at the top-most level and contains an instance of 'Behaviour', 'Robot' and 'Environment'. The Environment class contains many instances of the 'Polygon' class. The Behaviour class contains an instance of the 'Grid' class and many instances of the 'Record' class.

The Behaviour class has visibility of the Robot class which, in turn, has visibility of the Environment class. The Grid class has visibility of every instance of the Record class. The membership and visibility properties are transitive. For example, the Display class could access all instances of the Record class via Behaviour, or via the combination of Behaviour and Grid [8].

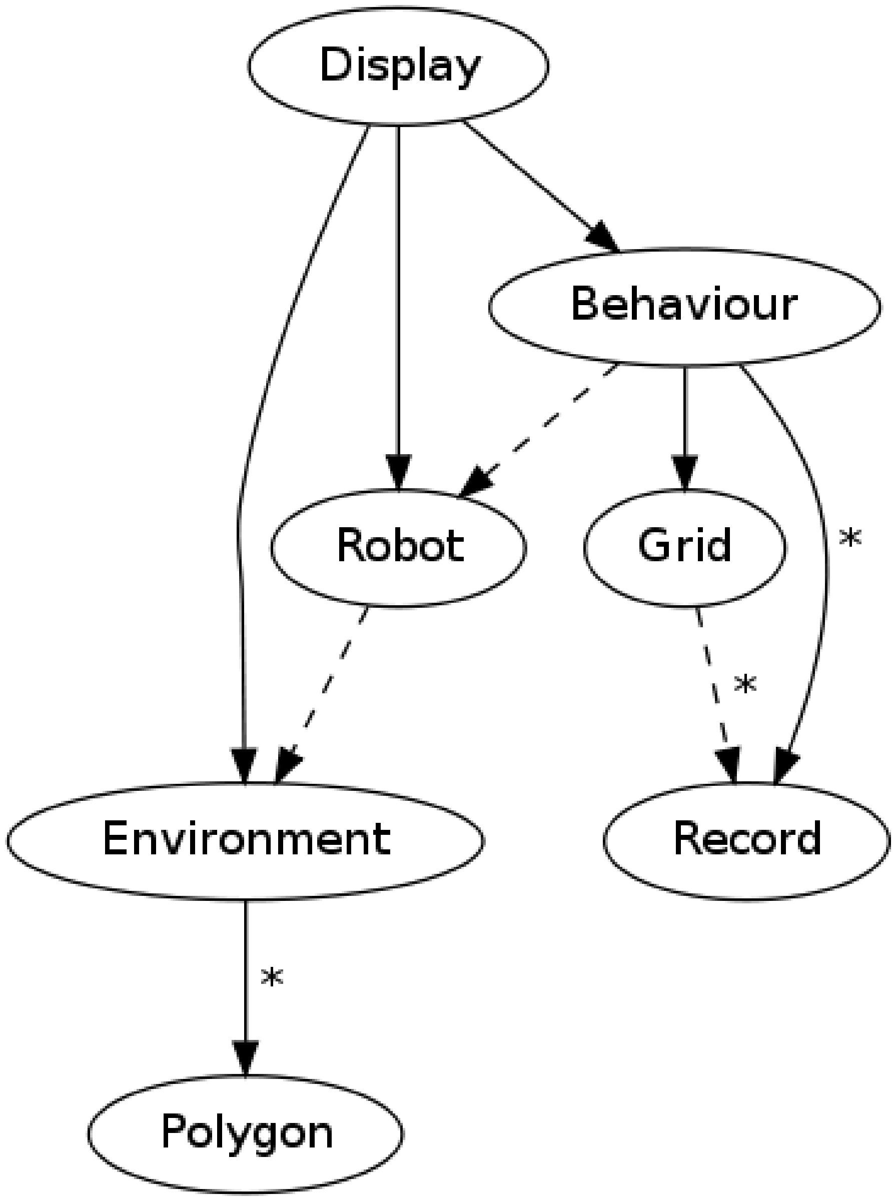


Figure 1.1: Class hierarchy diagram. The directed arrows denote membership, whereas the dashed arrows denote visibility. A * denotes a one-to-many relationship.

Chapter 2

Background

2.1 Gaussian noise

According to the central limit theorem, noise within a system may be modelled normally, due to the summation of its many independent origins [5]. Additive white Gaussian noise is the standard channel model for representing noise within a system. Its simplest form is shown in equation 2.1. The output signal level is equal to the input signal level plus a noise value, at time step i. Noise is distributed normally, shown in equation 2.2.

$$\text{Output}_i = \text{Input}_i + \text{Noise}_i \tag{2.1}$$

$$\text{Noise} \sim N(0, \eta) \tag{2.2}$$

2.2 Box-Muller transform

The Box-Muller transformation is a method to transform a uniform continuous two-dimensional distribution to a bivariate normal distribution, with a mean of zero and variance of one, shown in equations 2.3 and 2.4. u_1 and u_2 denote each dimension of the two-dimensional uniform distribution [2].

$$n_1 = \sqrt{-2 \ln u_1} \cos(2\pi u_2) \tag{2.3}$$

$$n_2 = \sqrt{-2 \ln u_1} \sin(2\pi u_2) \tag{2.4}$$

The Box-Muller transform may be used to generate random numbers efficiently, with a statistical distribution based on that of a bivariate normal distribution. This is useful for modelling systems containing Gaussian noise.

2.3 Kalman filter

The Kalman filter is an optimal recursive data processing algorithm [10]. It takes into account all sensory data, and estimates a new piece of data from a weighted combination. The weighting is dependent upon the certainty of each sensory measurement. It requires initial knowledge of the distribution of noise in the system, but it is often acceptable to assume a Gaussian distribution due to implications of the central limit theorem. The Kalman filter is said to be *recursive* because it is not a requirement that all previous data is kept in memory. This lends itself to the practicalities of the algorithm when deployed in a system with large quantities of data.

There are two main stages to Kalman filtering; prediction and correction. Equation 2.5 is used at the *prediction* stage and calculates the optimal estimation from a set of sensory measurements. Equation 2.6 is used at the *correction* stage and calculates the variance for the optimal estimation. This value is often referred to as the Kalman gain.

$$\mu = [\sigma_{z2}^2/(\sigma_{z1}^2 + \sigma_{z2}^2)]z1 + [\sigma_{z1}^2/(\sigma_{z1}^2 + \sigma_{z2}^2)]z2 \tag{2.5}$$

$$1/\sigma^2 = (1/\sigma_{z1}^2) + (1/\sigma_{z2}^2) \tag{2.6}$$

2.4 Rotation matrix

The two-dimensional rotation matrix is shown in equation 2.7. A point is rotated around the origin by θ degrees in an anti-clockwise direction, when the matrix is pre-multiplied with the point, shown in equation 2.8 and figure 2.1.

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{2.7}$$

$$P^{'} = PR \tag{2.8}$$

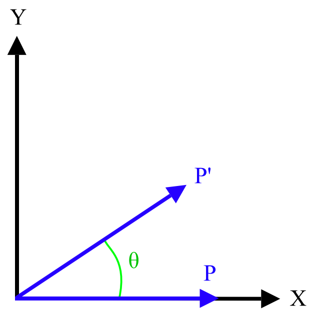


Figure 2.1: Application of the rotation matrix

2.5 Coordinate systems

To deal with abstract geometric objects, it is often easier to work in different coordinate systems. For example, a cell of a grid, displayed on a standard monitor may be defined in terms of grid coordinates or screen-pixel coordinates. To convert between the two systems, translation functions are required. Equation 2.9 translates grid coordinates to screen coordinates, whereas equation 2.10 translates screen coordinates to grid coordinates. S and G denote the screen and grid coordinate systems respectively. The subscripts w and h denote the respective width and height of each coordinate system.

$$S_{x,y} = G_{x,y} \ / \ G_{w,h} \times S_{w,h} \tag{2.9}$$

$$G_{x,y} = S_{x,y} \ / \ S_{w,h} \times G_{w,h} \tag{2.10}$$

2.6 Set comprehension

Set comprehension is a method of defining a set by stating the conditions that elements within the set must satisfy. The set {1, 2, 3} is described by equation 2.11. The first term denotes the superset from which to derive elements. The second term denotes the conditions that must be met for elements to appear in the set.

$$S = \{x \in \mathbb{Z} : 1 \leq x \leq 3\} \tag{2.11}$$

The semi-colon delimiter is used if there are several conditions that must be met for elements to appear in the set. For example, the set {1, 3} may be described by equation 2.12.

$$S = \{x \in \mathbb{Z} : 1 \leq x \leq 3; x \neq 2\} \tag{2.12}$$

Set comprehension may be used as an effective method for describing infinite sets, such as equation 2.13 for the set of integers greater than three, or equation 2.14 for the set of rational numbers between zero and one.

$$S = \{x \in \mathbb{Z} : x > 3\} \tag{2.13}$$

$$S = \{x \in \mathbb{R} : 0 < x < 1\} \tag{2.14}$$

2.7 Inverse tangent

The arctan2 function is a variation on the arctangent function that is used when the direction of a rotation around the origin is important. The standard arctangent function's range of principal values is limited from $-\pi/2$ to $\pi/2$. To calculate inverse tangents that span all four quadrants, the arctan2 function is required. It is defined in equation 2.15.

$$\text{atan2}(y, x) = \begin{cases} \text{atan}(y/x) & x > 0 \\ \pi + \text{atan}(y/x) & y \geq 0, x < 0 \\ -\pi + \text{atan}(y/x) & y < 0, x < 0 \\ \pi/2 & y > 0, x = 0 \\ -\pi/2 & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases} \tag{2.15}$$

Chapter 3

Simulation environment

3.1 Obstacles

Obstacles are represented within the simulation as a list of polygons, each containing at least three vertices. The environment class is responsible for maintaining this list, as well as defining the width and height of the simulation environment. The ordering of vertices within each obstacle is important. Starting from an arbitrary vertex, successive vertices must track the perimeter of the obstacle in either a clockwise or anti-clockwise direction. Figure 5.1 shows an example environment constructed from a set of obstacles.

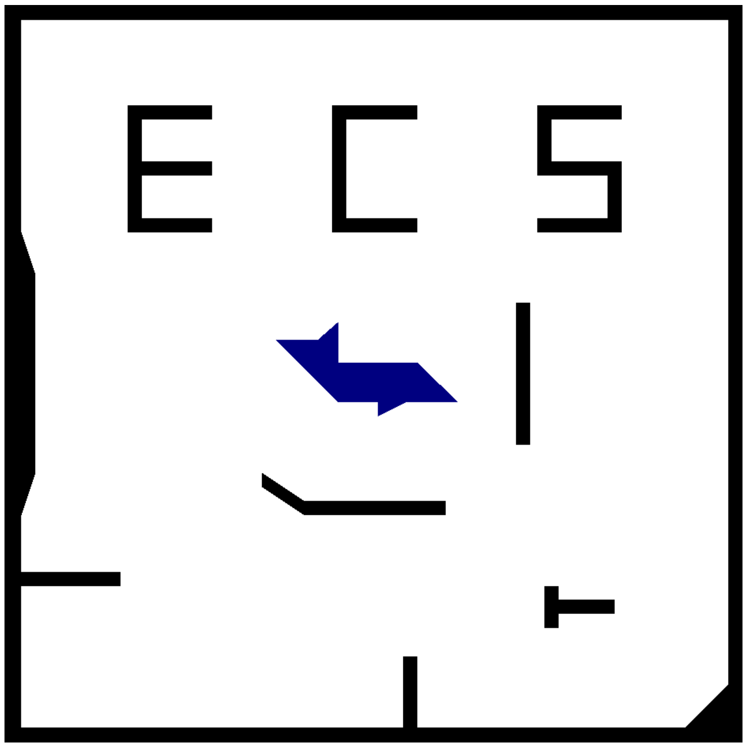


Figure 3.1: An example environment constructed from obstacles. The blue obstacle also serves as a beacon entity, explained in section 3.4.1.

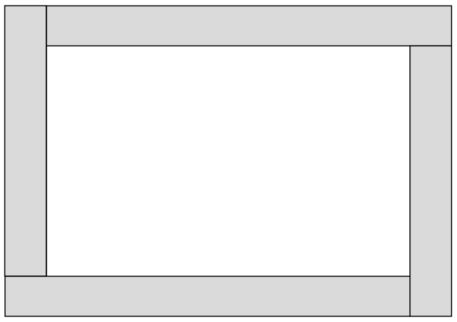


Figure 3.2: Generalising walls as obstacles

The walls of the simulation are represented by four additional polygons placed within the environment on construction, shown in figure 3.2. This means that the same drawing and obstacle evaluation functions may be used, instead of handling walls separately. There is no limitation on where obstacles may be placed; they can overlap, be contained within each other and be placed partially out-of-bounds, if required. These kinds of redundancies have a negligible effect on the efficiency of the simulation and may help to simplify the environment creation process for the user.

3.2 Robot

The robot class is used to represent the robot within the simulation environment. Static properties of the robot, such as width, height and movement rates, are passed into the constructor and an application programming interface is provided containing the basic operations to control the robot. These basic operations modify the dynamic properties of the robot, such as current location and angular direction. Gaussian white noise is applied on robot movements to promote the isomorphism between the simulated robot and a real-world equivalent.

3.2.1 API

The robot's API consists of four operations; forward, backward, left and right. Each operation takes a parameter of type float, which is the amount that the robot has been requested to move (or turn). The actual amount moved by the robot is susceptible to Gaussian white noise, which is generated using the Box-Muller method.

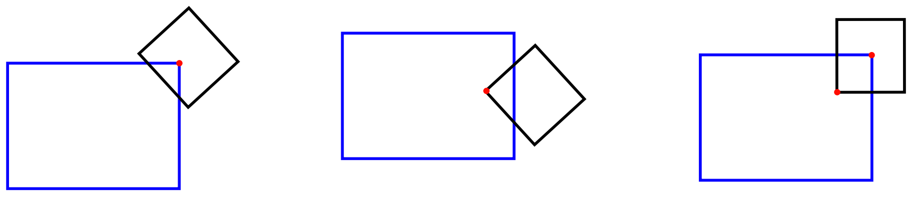
To preserve proportionality, the amount requested to move is multiplied by the static noise value to determine the variance of the noise function, shown in equation 3.1. The symbol r denotes the amount that the robot is requested to move and η denotes the static noise level within the system. Finally, the application of the homogeneous translation matrix, shown in equation 3.2, determines the new location of the robot in the case of forward and backwards movement and simple angular addition is used in the case of left and right turns.

$$D \sim N(r, \eta \times r) \tag{3.1}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = rand(D) \begin{pmatrix} 1 & 0 & cos\theta \\ 0 & 1 & sin\theta \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{3.2}$$

3.2.2 Collision handling

In a real-world situation, robots may not intersect obstacles. To simulate this phenomenon, some basic collision handling functionality is built into the robot class [7]. After each movement, the function 'updateVertices' is called, which re-calculates the corner coordinates of the robot. The function 'updateCollision' checks whether any of the vertices of the robot are within an obstacle or any vertices of an obstacle are within the robot. If they are, then the move is reversed. Figure 3.3 demonstrates some of the possible cases for collision between the robot and an obstacle.



(a) One vertex collision (b) One vertex collision (c) Two vertex collisions

Figure 3.3: Some possible cases for collision between the robot and an obstacle

3.3 Detector

The ‘lidar’ function in the robot class is responsible for updating the angular direction of the detector. This simply carries out angular addition, but also takes into account turns of the robot, as it would in a real-world scenario. The ‘updateLidar’ function is responsible for calculating the point at which the detector intersects with an obstacle [6]. When returning from this function, the value that is set, corresponds to the distance from the robot to the intercept and does not contain information concerning the obstacle or vertex of intersection.

A hill-climbing approach is implemented to determine the distance of the intersection [13]. The vertices of each obstacle are iterated over and every edge combination is passed to the ‘lidarIntercept’ function. This returns the vertex of the intersection between the detector and the current edge if one exists. The lidar distance is set to the nearest vertex returned from the ‘lidarIntercept’ function.

3.3.1 Intersections

To determine where the intersection occurs, the equation of the line for the object detector and the current edge are calculated. For there to be a valid intersection point, three criteria must be met. The first criterion is that the lines must not be parallel- in this case, a check can be made that the determinant of the two equations is not equal to zero. Secondly, the intersection must lie within the bounds of the edge. Finally, the intersection must lie on the correct side of the robot, corresponding to the emmision direction of the object detector.

3.4 Behaviour

The behaviour class takes low level sensory data and transforms it into more useful complex representations that may later be used by the explorative strategies. It carries out multi-dimensional Kalman filtering on the expected location of the robot as an attempt to localise itself. It maintains a grid representation of the environment and provides a framework for explorative strategies to query potential collisions.

3.4.1 Kalman filter

As the robot moves around its environment, uncertainty as to the location of the robot gradually builds due to the Gaussian white noise contained within the system.

This in turn causes degradation of the robot's understanding of its environment. The behaviour class tracks this uncertainty in the form of covariance in the x and y planes. Any obstacle may be set as a 'beacon' entity, which the robot may use to localise itself [10].

Prediction

When a beacon is detected, the behaviour class sets its current location to the robot's actual location within the environment. The current covariance is set to zero and the Kalman prediction process begins. A multi-dimensional array holds data about the previous moves of the robot. This is iterated over backwards until a previous measurement with zero covariance is found. On each iteration, an estimation is made as to the robot's location, by backtracking from the newly discovered certain location. The covariance as to the uncertainty of this reverse path is also tracked, which may then be used to combine the sensory results into an improved understanding of the robot's path.

Correction

The Kalman correction process calculates the improved location and covariance for each iteration by combining corresponding points on the original, forwards path, and the predicted backwards path. Equations 3.3 and 3.4 are calculated to estimate the new location based on the ratio of certainty between each pairing. The pairs $(x1, y1)$ and $(x2, y2)$ denote the location vectors on the forwards and backwards paths, whilst the pairs $(\sigma_{x1}^2, \sigma_{y1}^2)$ and $(\sigma_{x2}^2, \sigma_{y2}^2)$ denote the covariances associated with those locations. Equations 3.5 and 3.6 are used to calculate the covariance for the new location. This will always be less than that of the minimum input covariances, hence the robot's *understanding* of its path is improved.

$$\mu_x = [\sigma_{x2}^2/(\sigma_{x1}^2 + \sigma_{x2}^2)]x1 + [\sigma_{x1}^2/(\sigma_{x1}^2 + \sigma_{x2}^2)]x2 \tag{3.3}$$

$$\mu_y = [\sigma_{y2}^2/(\sigma_{y1}^2 + \sigma_{y2}^2)]y1 + [\sigma_{y1}^2/(\sigma_{y1}^2 + \sigma_{y2}^2)]y2 \tag{3.4}$$

$$1/\sigma_x^2 = (1/\sigma_{x1}^2) + (1/\sigma_{x2}^2) \tag{3.5}$$

$$1/\sigma_y^2 = (1/\sigma_{y1}^2) + (1/\sigma_{y2}^2) \tag{3.6}$$

3.4.2 Grid model

The grid model is a high level construct primarily designed to be used by the explorative strategies as a tool for reasoning about their current understanding of the environment. It assumes no prior knowledge and dynamically resizes and rescales in accordance with configurable criteria. Figure 3.4 shows an instance of the grid model constructed from the environment shown in figure 3.1.

Functionality is provided for mapping points onto the grid and translating between *world* and grid coordinates. Special consideration is made for compatibility with Kalman filtering to ensure that the grid model is a true representation of the updated sensory data, after a correction process has occurred. Mechanisms for feedback are built into the grid in the form of ‘completeness’ and ‘accuracy’ which may be tracked throughout the simulation.

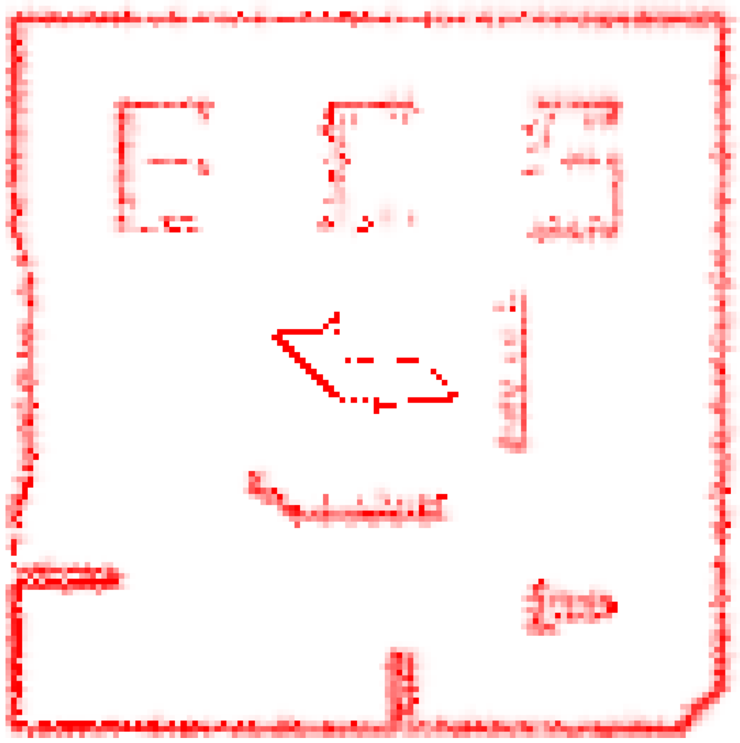


Figure 3.4: An instance of the grid model

Mapping points

To map a point on the grid, the location and covariance of the point are passed as parameters to the ‘mapPoint’ function. A bounding box is calculated that determines over which cells to iterate. Theoretically, every cell in the grid should be reconsidered due to the unbounded characteristic of the Normal distribution, but in practice, this is not necessary. The bounding box is limited to three standard deviations from the mean, covering approximately 99.9% of possible obstacle locations, and is considerably more efficient than a calculatory census of grid cells. For each cell within the bounding box, the probability that the obstacle measurement lies within the cell is calculated. If the calculated probability exceeds the existing probability then the value is replaced.

To calculate the probability, the normal distributions are set up for the x and y planes, shown in equations 3.7 and 3.8 respectively. The mean parameter for each distribution is the relevant component of the location passed to the function. These distributions are then used to calculate the probability that the obstacle lies in the subsection of each independent plane by differencing of the relevant cumulative distribution functions- equations 3.9 and 3.10. The w in the equations refers to a translation function that maps grid-cell coordinates to ‘world’ coordinates. These independent probabilities are multiplied together to gain the overall probability that the obstacle measurement lies within the current cell, shown in equation 3.11. This is equivalent to finding the probability of the intersection of the two areas, shown in figure 3.5.

$$X \sim N(\mu_x, \sigma_x^2) \tag{3.7}$$

$$Y \sim N(\mu_y, \sigma_y^2) \tag{3.8}$$

$$P_x = P(x < w(C_x + 1)) - P(x < w(C_x)) \tag{3.9}$$

$$P_y = P(y < w(C_y + 1)) - P(y < w(C_y)) \tag{3.10}$$

$$P_{x,y} = P_x \times P_y \tag{3.11}$$

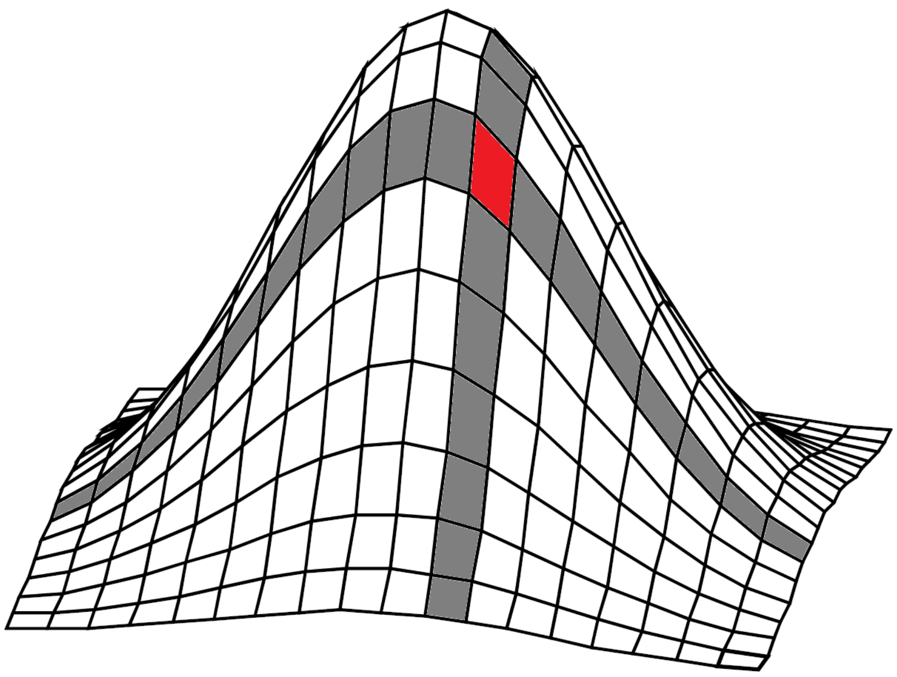


Figure 3.5: Calculating the probability that a measurement occurred in an arbitrary cell by finding the intersection of areas in orthogonal planes, for a grid mapped onto a bivariate Normal distribution

Dynamics

The dimensions of the grid are determined by the points it has been requested to map. After the bounding box has been calculated, the 'mapPoint' function checks whether the box is fully contained within the grid. If it is not, then a resize process takes place. The cardinality of the two-dimensional 'cells' array is increased, and all prior points are remapped. The resize operation has a time complexity of $O(n)$, where n is the number of points previously mapped to the grid. Resize operations tend to take place frequently at the start of the simulation and rarely as it progresses.

Therefore, the time spent resizing the grid has a minimal impact on the runtime of the simulation.

Rescaling the grid is a little more complex. As configuration for the simulation, three parameters are set; ‘start granularity’, ‘minimum granularity’ and ‘split determinant’. The size of each cell within the grid is initially set to the start granularity. After a point has been mapped to a cell, a check is carried out on its direct neighbours, including diagonals. If the probability of the current cell and any neighbouring cell is above the split determinant then the ‘divide’ function is called. This halves the current granularity and remaps the points within the grid. Rescaling the grid is a costly operation. Therefore, setting a minimum granularity ensures that the grid does not rescale indefinitely.

Kalman compatibility

After a Kalman correction process has occurred, many of the points that had been mapped to the grid will have changed. To ensure the grid is a true representation based on the sensory data, these points need to be remapped. This is not as simple as working the ‘mapPoint’ function in reverse, as the ‘max’ operand for combining the probabilities on a cell is non-commutative. A naive approach, would be simply to remap every point. This would be hugely inefficient as only a few points change between updates. Instead, an internalised stack is maintained within the grid class, that keeps track of changes since a previous Kalman update. Prior to the correction process, the grid is reverted back to its previous state by popping all changes. The modified points are then mapped to the grid.

Feedback

As a tool for the evaluation of explorative strategies, functions to measure ‘completeness’ and ‘accuracy’ are provided as part of the grid class. Completeness is calculated by counting the number of non-zero cells and dividing this summation by the total number of cells. Accuracy is calculated by finding the mean probability of all non-zero cells. The general idea is that the completeness metric gives a comparative value for an explorative strategy’s overall coverage of the grid, whereas the accuracy metric refers to its certainty of the representation.

3.4.3 Collision detection

A collision detection module is built into the behaviour class that may be used by the explorative strategies, to query the probability that a given move will result in collision [11]. The ‘overlay’ function estimates the corner coordinates of the robot, given a location and angular direction, and returns the set of grid cells that the robot would overlap.

Overlays are retrieved in the ‘collision’ function for the current location of the robot and its estimated location, after the given move takes place. The set complement of the estimated location cells in the current location cells is determined, shown in equation 3.12. The summation of the probabilities of the cells contained within the complement set is calculated. By clamping this value to one, a slight performance gain can be achieved by exiting from the function early with the same results, shown in equation 3.13.

$$S_\Delta = S_\beta - S_\alpha \tag{3.12}$$

$$\min\left(\sum_{i \in S_\Delta} i, 1\right) \tag{3.13}$$

3.5 Graph window

The graph window plots completeness and accuracy throughout the simulation. The relevant functions are polled every second and plotted to the graph. Polling is a better solution than re-plotting on each simulation update, due to the complexities of the functions. The x-axis rescales constantly such that the data is spread maximally, for better viewing. The graph is generated in pure OpenGL using a mixture of lines, text and colours, shown in figure 3.6.

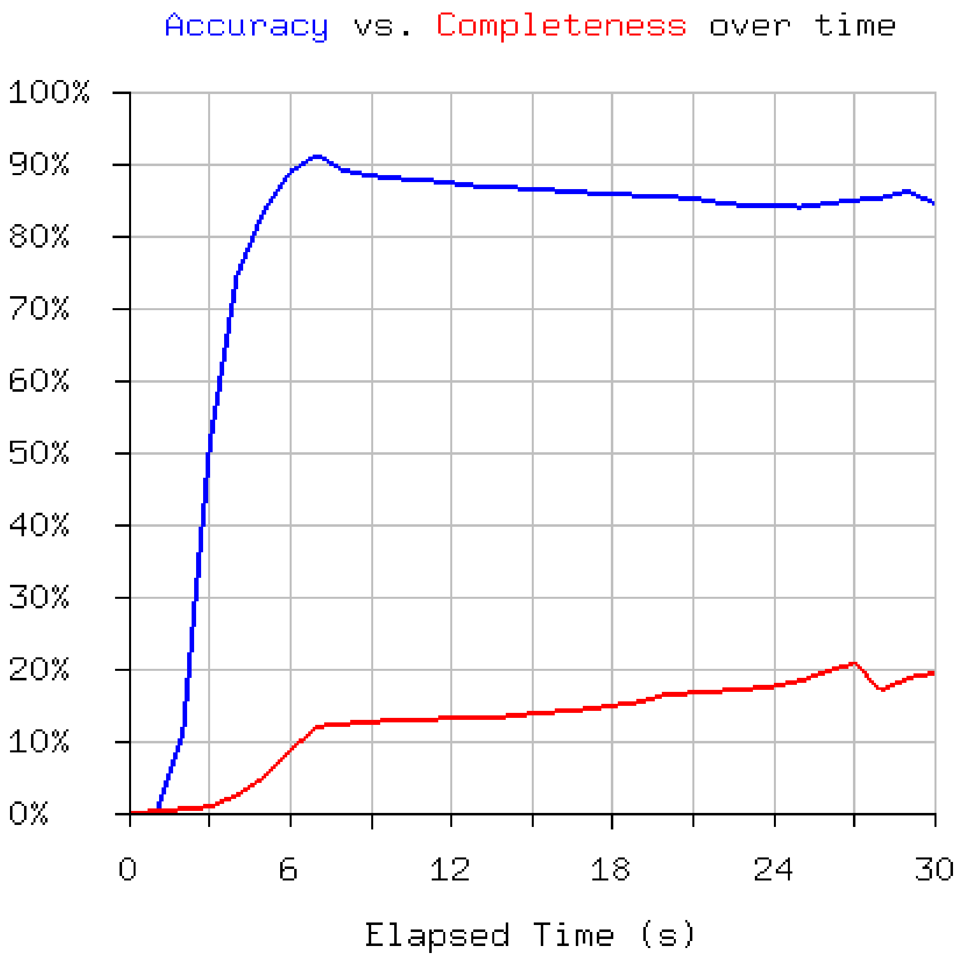


Figure 3.6: The graph window

3.6 Configuration

To set up the simulation, a bespoke configuration file is read. An XML reader library aggregates the data from this file and passes it to the relevant functions. The configurable elements are split into four sections; 'environment', 'robot', 'behaviour' and 'display'. The environment section is used to set up the width and height of the simulation and to define all obstacles and beacons within the environment. The robot section is used to set up the static properties of the robot, such as width, height, move rate and noise. The behaviour section defines the identifier of the explorative strategy to run, as well as the granularity parameters for the grid. Finally, the display section sets up the window dimensions and the default viewing elements, such as the robot path display or the grid model.

3.6.1 XML / DTD

The configuration file is an XML file that complies with a custom document type definition. XML is a widely used markup language that is well supported. It may be edited in a basic text editor and is human readable. The DTD is provided at the top of all configuration files and serves as a reference for the structure of the XML. Appendix listing 1 shows the Document Type Definition used for all configuration files.

3.6.2 Automation

Additional support for the automation of testing is built into the configuration file. A 'runtime' parameter may be set in addition to a 'tests' parameter. If there are remaining tests, the simulation will restart when the runtime has elapsed. After each test completes, a text file is appended with the final values for completeness and accuracy. If the robot becomes stuck, that particular test instance will exit early, and the time of exit will also be written to file. This means that software may be left for an extended duration to carry out testing without any additional human interaction.

3.7 Unit testing

3.7.1 Robot movements

Listing 3.1 demonstrates a unit test investigating robot movements. It uses textual outputs liberally to track the behaviour of variables at intermediate stages throughout the function.

```
/*

Unit test 4/18
Checks that forwards movement works as expected
Revision 1
15/03/11

*/
//Moves the robot unless there is a collision.
void Robot::forward(float amount) {
  cout << "Trying to move by " << amount << " units" << endl;

  amount = gaussianRandom(amount, amount * mNoise);

  cout << "After noise is applied: " << amount << endl;

  float radians = angle * (float)PI / 180;

  cout << "Current location: " << location.x << ", " << location.y <<
      endl;

  location.x += amount * cos(radians);
  location.y += amount * sin(radians);

  cout << "New location:" << location.x << ", " << location.y << endl;

  updateVertices();
  updateCollision();

  if (collision) {
    cout << "A collision has occurred, reverting move" << endl;

    location.x -= amount * cos(radians);
    location.y -= amount * sin(radians);

    cout << "Final location: " << location.x << ", " << location.y;
  }
  updateVertices();
  updateCollision();
}
```

Listing 3.1: Carrying out unit testing for robot movements

3.7.2 Window resizing

Listing 3.2 demonstrates a unit test for window resizing. The aspects for the window and model are compared and different outputs are generated based on the program flow.

```cpp
/*

Unit test 7/18
Checks that window resizing works correctly
Revision 1
17/03/11

*/
//Called when the window is reshaped. Maintains aspect ratio.
void reshapeCallback(int width, int height) {
  cout << "Window dimensions: " << width << ", " << height << endl;

  double windowAspect = (double)width / height;
  double modelAspect = (double)DisplayE->width / DisplayE->height;
  double offset;

  cout << "wAspect: " << windowAspect << ", mAspect: " << modelAspect
      << endl;

  if (modelAspect > windowAspect) {
    cout << "model aspect is greater";

    offset = (1 - windowAspect / modelAspect) * height;

    xFrom = 0; xTo = width;
    yFrom = offset / 2; yTo = height - yFrom;
  }
  else {
    cout << "window aspect is greater";

    offset = (1 - modelAspect / windowAspect) * width;

    xFrom = offset / 2; xTo = width - xFrom;
    yFrom = 0; yTo = height;
  }

  cout << "Setting viewport to (" << xFrom << ", " << yFrom << ", " <<
      xTo - xFrom << ", " << yTo - yFrom;

  glViewport(xFrom, yFrom, xTo - xFrom, yTo - yFrom);
}
```

Listing 3.2: Carrying out unit testing for window resizing

3.7.3 Kalman filtering

Listing 3.3 demonstrates a unit test to verify the behaviour of the Kalman filter. The covariance is checked for improvement and the size of the 'reverse' array is tested after it has been cleared.

```
/*

Unit test 12/18
Checks that the Kalman filter is working correctly
Revision 2
24/03/11

*/
//Calculate best estimate locations based on variance weightings.
for (unsigned int r = 1, d = data.size() - 2; r < reverse.size(); r++,
    d--) {
  //Mean location.
  float x = data[d].x + (data[d].xV / (data[d].xV + reverse[r].xV)) * (
      reverse[r].x - data[d].x);
  float y = data[d].y + (data[d].yV / (data[d].yV + reverse[r].yV)) * (
      reverse[r].y - data[d].y);

  cout << "Averaged " << x << ", " << y;
  cout << " from " << data[d].x << ", " << data[d].y << " and ";
  cout << reverse[r].x << ", " << reverse[r].y << endl;

  //Improved variance.
  float xVar = data[d].xV * reverse[r].xV / (data[d].xV + reverse[r].xV
      );
  float yVar = data[d].yV * reverse[r].yV / (data[d].yV + reverse[r].yV
      );

  if (xVar > data[d].xV || xVar > reverse[r].xV) cout << "Error: x
      variance has not improved" << endl;
  if (yVar > data[d].yV || yVar > reverse[r].yV) cout << "Error: y
      variance has not improved" << endl;

  //Update the data records.
  data[d].x = x;
  data[d].y = y;
  data[d].xV = xVar;
  data[d].yV = yVar;

  cout << "Attempting to map point..." << endl;

  //Map the improved points to grid.
  Vertex v = getVertex(data[d].x, data[d].y, data[d].l, data[d].d);
  grid.mapPoint(v.x, v.y, data[d].xV, data[d].yV, true);

  cout << "Point mapped";
}
reverse.clear();
if (reverse.size() != 0) cout << "Error: reverse contains left over
    elements";
```

Listing 3.3: Carrying out unit testing for Kalman filtering

3.8 Acceptance testing

1. The simulation must be able to represent environments of varying dimensions. This requirement is met successfully, see sections 3.1 and 3.6. The user is able to set the dimensions of the model in the configuration file which are passed to the environment constructor.

2. Environments must have the capability to store a list of obstacles. This requirement is met successfully, see section 3.1. The environment class contains a member variable that is a list of obstacles, where each obstacle is a list of vertices.

3. The software must simulate a robot within the obstructed environment, with an API that may be used by each explorative strategy. This requirement is met successfully, see sections 3.2 and 3.2.1. A robot is simulated within the environment with an API consisting of four basic operations; forwards, backwards, left and right.

4. Additive white Gaussian noise must be applied on each robot movement. This requirement is met successfully, see section 3.2.1. Noise is generated using the Box-Muller transform and applied on each movement. The noise is scaled according to its magnitude, preserving proportionality and ensuring it is additive.

5. Collisions must be handled appropriately such that the robot may not intersect objects. This requirement is met successfully, see section 3.2.2. If any of the robot's vertices are contained within an obstacle, or vice versa, the move is reversed.

6. An object-detector must be simulated that measures the distance to the nearest obstacle at a give angle. This requirement is met successfully, see sections 3.3 and 3.3.1. The 'updateLidar' function calculates the distance to the nearest obstacle.

7. The behavioural module must attempt to compensate for noise within the system. This requirement is met successfully, see section 3.4.1. A two-dimensional Kalman filter is implemented that calculates improved estimates for the robot's previous locations.

8. An internalised representation of the environment must be maintained for the robot. This requirement is met successfully, see section 3.4.2. The grid model is created and updated with sensory measurements which the robot may use as a high level construct for reasoning about its environment.

9. The robot must be able to interface with different strategies for exploring the environment. This requirement is met successfully, see section 3.6. The configuration file is used to specify which strategy to run, which is passed to the 'runStrategy' function inside the behaviour module.

10. When the simulation window is resized, its aspect ratio must be the same. This requirement is met successfully, see listing 3.2. The window and model aspect ratios are compared and the viewport is set appropriately [1].

11. A graph window must be displayed showing the current progress of each explorative strategy. This requirement is met successfully, see section 3.5. The completeness and accuracy functions for the grid are polled each second and plotted to a graph.

12. The simulation must be able to use a configuration file to load environments and configure many aspects of the system. This requirement is met successfully, see section 3.6. A wide array of elements are configurable, such as static robot values, obstacle locations and the current exploration strategy.

13. The capability to automate testing must be provided as part of the configuration file. This requirement is met successfully, see section 3.6.2. The 'runtime' and 'tests' values are set in the configuration file. When the runtime expires, a new test begins until all tests are complete.

14. The simulation must provide the capability to toggle different data elements to be displayed. This requirement is met successfully, see section 3.6.2. The default view elements can be set in the configuration file and then toggled from a right click menu as the simulation runs. Shortcut keys are also provided for toggling each view.

15. The user must be able to pause the simulation at any time and toggle the current display elements. This requirement is met successfully. The space bar key may be used to pause the simulation or by going through the right-click menu. Viewing elements may be changed as usual, in this state.

16. The simulation should run at near real-time speed. The complexity of its underlying operations should support this requirement. This requirement is met successfully, see section 3.4.2. The complexity of the resize, rescale and kalman update operations are optimised to allow for the simulation to run at near real-time speeds.

17. The simulation should be frame-rate independent. This requirement is met successfully, see section 1.2.1. The OpenGL framework is used which allows control over the frame and animation rates [12].

18. The system should be object-oriented for purposes of extensibility and reuse. This requirement is met successfully, see section 1.5. The system is divided into different objects shown in the class hierarchy diagram. Classes may be members of, or contain references to other classes.

19. The software code should be well-commented such that it is accessible to other developers. This requirement is met successfully, see listings 3.1, 3.2 and 3.3. The software code is commented and presented appropriately.

20. The explorative strategies should be sufficiently decoupled such that they may be modified with ease. This requirement is met successfully, see chapter 4 and section 3.6. The current strategy may be specified in the configuration file. To modify a strategy, the relevant case statement in the 'runStrategy' function is altered.

Chapter 4

Exploration strategies

4.1 Wall following

Wall following is frequently used as an algorithm for maze solving. When applied to an open environment, it may still provide good coverage [3] [15]. The wall following algorithm uses the grid model and begins by calculating the equation of the line that passes through the front-right corner of the robot at a 45 degree angle, equation 4.1. P denotes the location of the front-right corner and θ denotes the angular direction.

$$y = tan(\theta - 45)x + (P_y - tan(\theta - 45) \times P_x) \tag{4.1}$$

The next stage implements a Digital Differential Analyser, shown in figure 4.1. This involves stepping along the line in x and y, and adding all cells that intersect with the line to a set. The cell that contains the front-right corner of the robot determines where to begin stepping. The direction to step in x and y is determined by the direction of the ray emitting from the front-right corner and is shown by the orange arrows in the diagram.

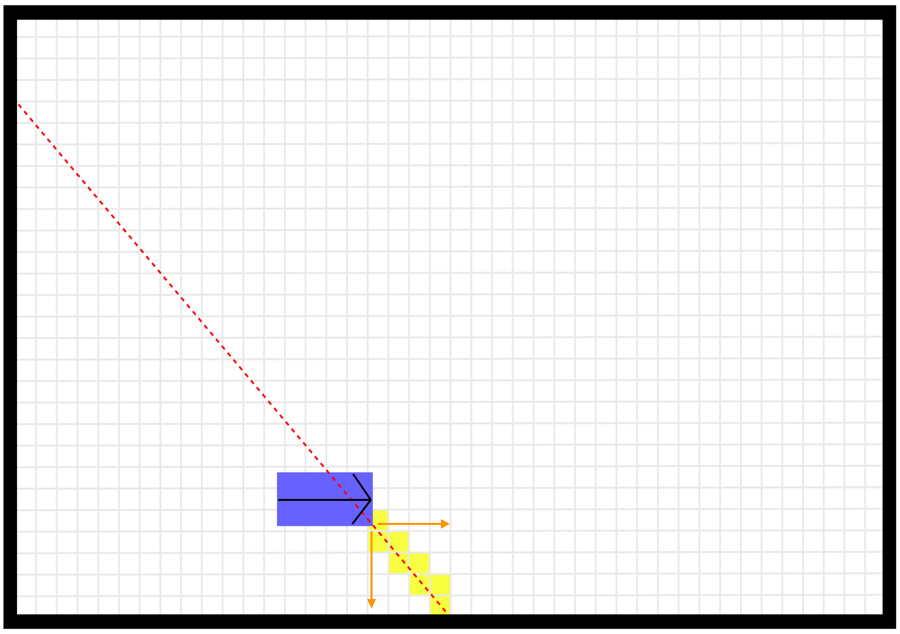


Figure 4.1: The implementation of a Digital Differential Analyser to determine which cells intersect a ray emitting from the front-right corner of the robot

For every cell in the set, the distance from its centre to the robot's location is calculated, if its grid probability is greater than zero, shown in equations 4.2 and 4.3. S denotes the set of cells to consider and r denotes the robot's location. All cells that are further from the robot than the non-zero cell with minimum distance are removed from the set, for the purposes of displaying a unique view related to the wall following strategy.

$$\beta = \{c \in S : c_p \neq 0\} \tag{4.2}$$

$$d = \min\left(\sqrt{(c_x - r_x)^2 + (c_y - r_y)^2}\right), \quad \forall c \in \beta \tag{4.3}$$

4.1.1 Controlling distance

The minimum distance from the wall is defined as one width of the robot, whereas the maximum distance from the wall is defined as two widths of the robot. If the value of d is less than the minimum then it turns away from the wall. If d is greater than the maximum then it turns towards the wall. Otherwise, the robot continues forwards.

Moves are alternated between forwards and turning so that the robot moves in arcs rather than turning on the spot. This prevents the robot from endlessly rotating when too far or too close to the wall. Figure 4.2 shows an example run of the wall following strategy after 30 seconds. The green line represents the robot's path.

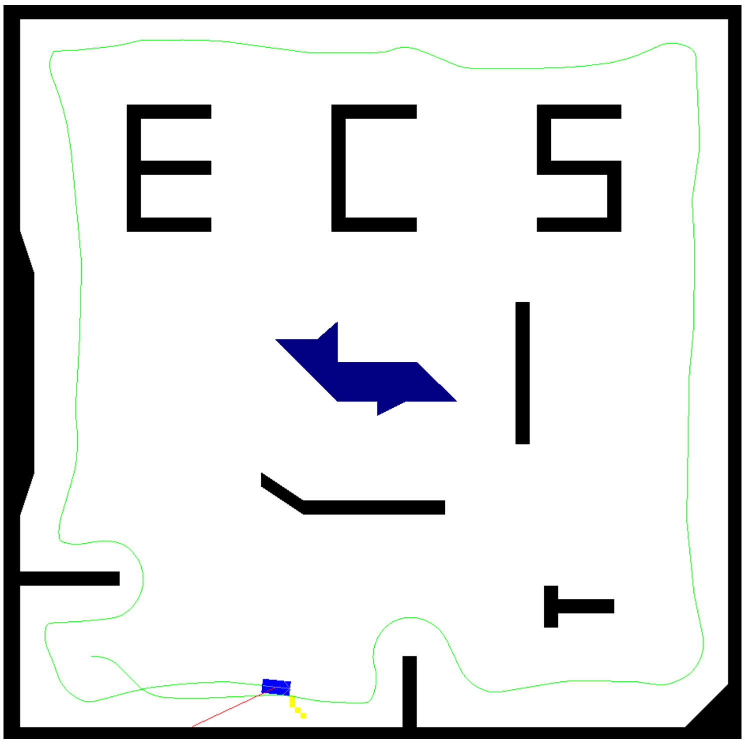


Figure 4.2: The robot's path, shown in green, after a 30 second run of the wall following strategy. The 'tracker' overlay is enabled, displaying the set of cells, derived from the digital differential analyser, up to the first non-zero cell.

4.1.2 Collisions and cycles

Collision prevention and cycle detection are built into the strategy. On each forward movement, the strategy queries the probability of collision using the detection function contained within the behaviour class. If this probability is greater than zero, the forwards movement is skipped and the robot continues to turn. To prevent cycles, the previous move is tracked. The strategy may not choose a move for the current turn that undoes the move from the previous turn. This constraint is shown in equation 4.4.

$$move(t) \neq \neg move(t-1), \quad t > 1 \tag{4.4}$$

4.2 Random avoidance

The random avoidance strategy instructs the robot to travel in a straight line until it is about to collide. It will then turn a random distance and continue to travel forwards. This strategy tends to explore further into the centre of the environment than the wall following strategy so, in theory, should gain better coverage. It uses the collision detection function to query when it is about to collide based on the grid model. If the robot becomes stuck, it will attempt to move in any direction that it has not been previously tried for the current move, as an attempt to free itself. Figure 4.3 shows an example run of the random avoidance strategy after 30 seconds.

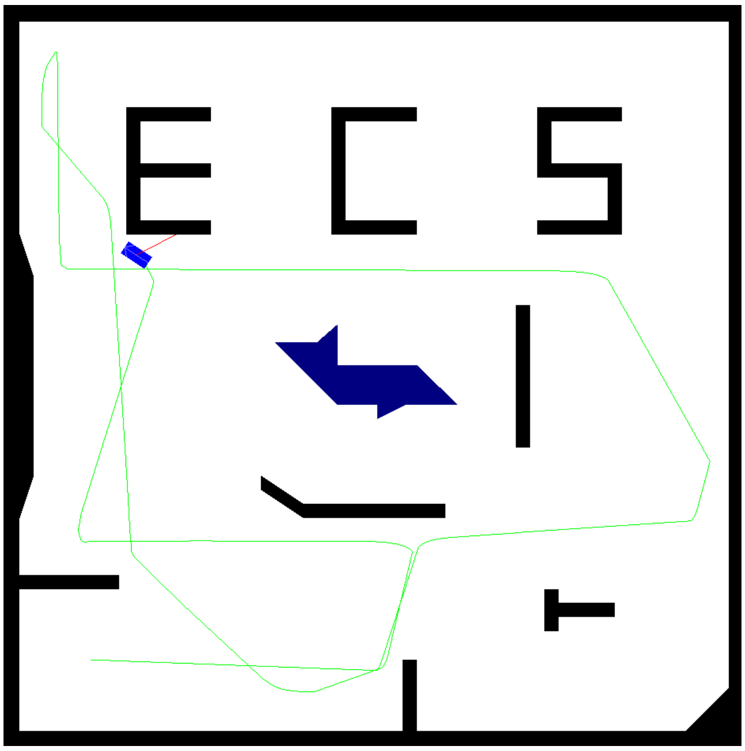


Figure 4.3: The robot's path, shown in green, after a 30 second run of the random avoidance strategy

A cyclic problem arises with this strategy if the robot becomes stuck and backwards is selected as the valid move to free itself. The next move the robot will attempt is forwards. This will lock the robot into a forwards-backwards cycle and no explorative progress will be made. To counter this problem, a flag is set to true if the previous move was backwards. This may then be checked on the successive move. If it is true then the robot will not choose to move forwards.

A slightly more complicated problem arises related to the random direction for turning. If the robot collides again shortly after it has turned away from the wall, it may choose the opposite direction in which to turn. This is inefficient as the robot will spend most of its time backtracking over areas it has already explored. To counter this problem, a 'remaining' variable is maintained. When a new turning direction is chosen, the remaining value is set to two lengths of the robot. If it collides again before it has travelled the distance stored in the remaining variable, then it will choose to turn in the same direction.

4.3 Direct quadrants

The direct quadrants strategy attempts to build on the random avoidance strategy by directing the robot towards unexplored areas. An additional abstract data structure is maintained, containing the number of turns the robot has spent in different sections of the simulation environment. This data structure may then be used to direct the robot towards quadrants with fewer turns. It uses the same collision handling and cycle prevention as the random avoidance strategy, based on the grid model.

Equation 4.5 is used to calculate in which quadrant the robot is located. $w^{'}$ denotes a function that maps world coordinates to grid-cell coordinates, Q_w and Q_h denote the dimensions of the quadrant data structure and G_w and G_h denote the dimensions of the grid model. Initially, the number of turns that the robot has spent in the current quadrant is incremented, shown in equation 4.6.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \lfloor w^{'}(r_x) \times Q_w \;/\; G_w \rfloor \\ \lfloor w^{'}(r_y) \times Q_h \;/\; G_h \rfloor \end{pmatrix} \tag{4.5}$$

$$Q_{x,y} = Q_{x,y} + 1 \tag{4.6}$$

4.3.1 Second ordering

The direct neighbour with the minimum turns determines the next quadrant to navigate towards. When the robot is located in a boundary quadrant, the choice of neighbours is reduced. If more than one neighbour has the minimum number of turns, a second ordering is required. The set containing the centres of the neighbours with the minimum turns is constructed, shown in equation 4.7. M denotes the set of neighbours with minimum turns.

The set containing the unit vectors from the robot's location to the centres of the quadrants with minimum turns is constructed, shown in equation 4.8. The angular offsets from the robot's current heading are calculated for each unit vector- equation 4.9, and the quadrant corresponding to the minimal offset is chosen. This means that the time spent turning towards quadrants will be minimised, improving the efficiency of the strategy. Figure 4.4 demonstrates the quadrant selection process.

$$C = \left\{ c : \begin{pmatrix} c_x \\ c_y \end{pmatrix} = \begin{pmatrix} w((m_x + 0.5) \times G_w \;/\; Q_w) \\ w((m_y + 0.5) \times G_h \;/\; Q_h) \end{pmatrix} ;\; m \in M \right\} \tag{4.7}$$

$$U = \left\{ \vec{u} : \begin{pmatrix} u_x \\ u_y \end{pmatrix} = \widehat{\begin{pmatrix} c_x - r_x \\ c_y - r_y \end{pmatrix}} ;\; c \in C \right\} \tag{4.8}$$

$$\alpha = \min(\arctan_2(\sin\theta, \cos\theta) - \arctan_2(u_y, u_x)), \quad \forall u \in U \tag{4.9}$$

The 'turn' flag is set to false if α is within an acceptable range, otherwise it is true. The 'turnLeft' flag is set to true if α is between 180 and 360 degrees, otherwise it is false. These flags are used within the random avoidance strategy as a form of encapsulation. If 'turn' is true, the robot will turn on the spot, in the direction determined by the 'turnLeft' flag, instead of moving forwards by default. Figure 4.5 shows an example run of the direct quadrants strategy.

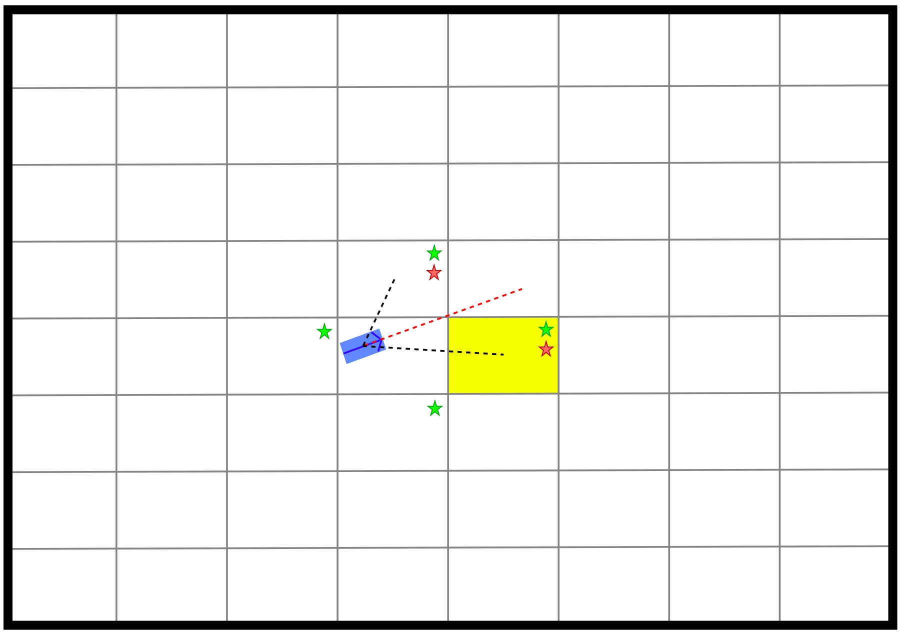


Figure 4.4: An example selection process of the direct quadrants strategy. Green stars denote neighbour quadrants, red stars denote neighbour quadrants with minimal turns, the shaded quadrant is selected.

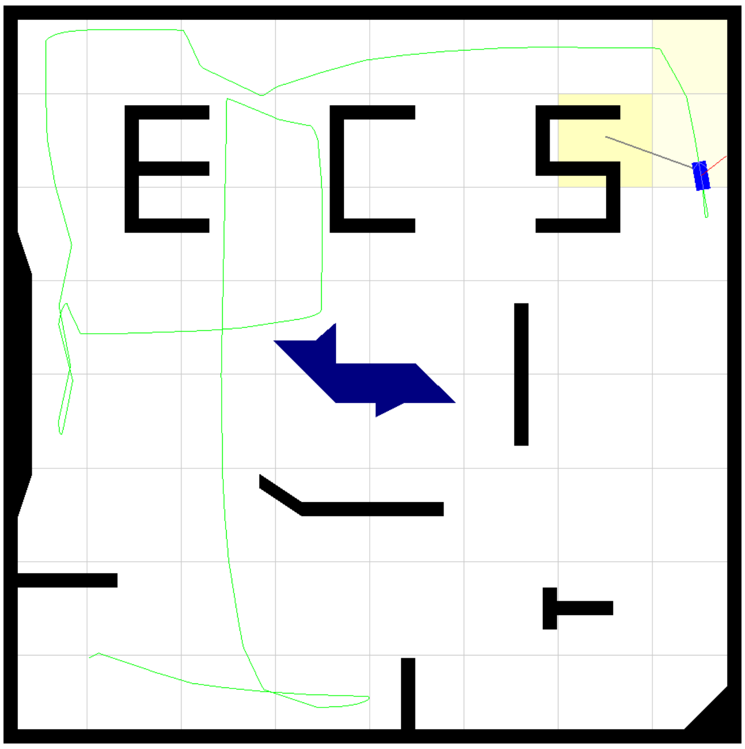


Figure 4.5: The robot's path, shown in green, after a 30 second run of the direct quadrants strategy. The quadrant overlay is enabled displaying the neighbouring cells the strategy is considering. The grey line denotes the vector to the selected quadrant.

4.4 Implied quadrants

The need for a modification of the direct quadrants strategy is demonstrated in figure 4.6. When an obstacle spans many quadrants, the robot will tend to attempt to head towards the quadrants on the other side of the obstacle. With the direct quadrants strategy, the robot has a fixed understanding of where it wants to go and turns to face that direction. Therefore, the strategy will constantly direct the robot into the wall and trigger the collision prevention code. Usually, the robot will get round the obstacle due to the 'remaining' variable as part of the collision prevention, but it is somewhat inefficient.

The implied quadrants strategy builds upon direct quadrants, but alternates between forwards and turning movements. Additionally, the second ordering criterion for selecting quadrants is different. When there is more than one quadrant with a minimum number of turns, the quadrant that is nearest to the robot's location is selected. In the case of figure 4.6, the robot will still attempt to head towards the quadrant on the other side, but it has more flexibility in its pathing.

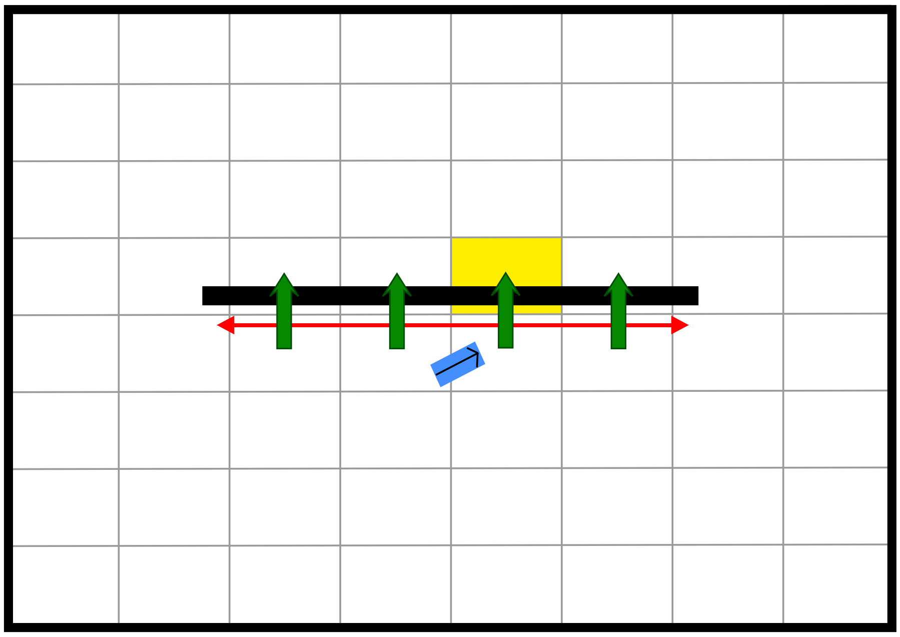


Figure 4.6: A demonstration of a problem that arises with the direct quadrants strategy. The green arrows denote the direction the strategy chooses for the cells under the obstacle. The red arrow denotes the path that is likely to be repeated by the robot, causing inefficiency.

4.4.1 Second ordering

The minimum distance from the robot's location to the centres of the cells with minimum turns, is calculated, shown in equation 4.10. The quadrant corresponding to the minimum is selected. Figure 4.7 demonstrates the difference in quadrant selection. Figure 4.8 shows an example run of the implied quadrants strategy.

$$d = \min\left(\sqrt{(c_x - r_x)^2 + (c_y - r_y)^2}\right), \quad \forall c \in C \tag{4.10}$$

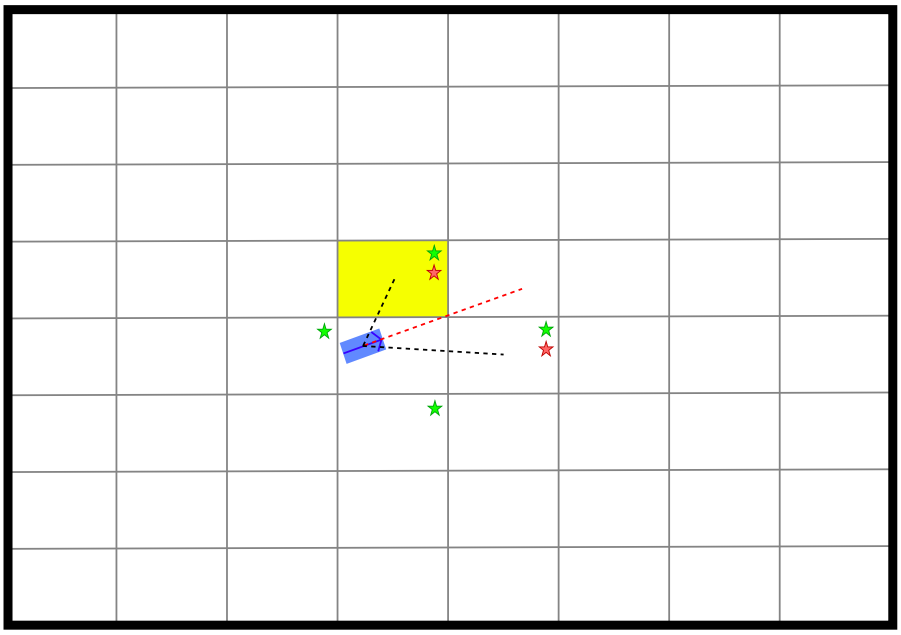


Figure 4.7: An example selection process of the implied quadrants strategy. Green stars denote neighbour quadrants, red stars denote neighbour quadrants with minimal turns, the shaded quadrant is selected.

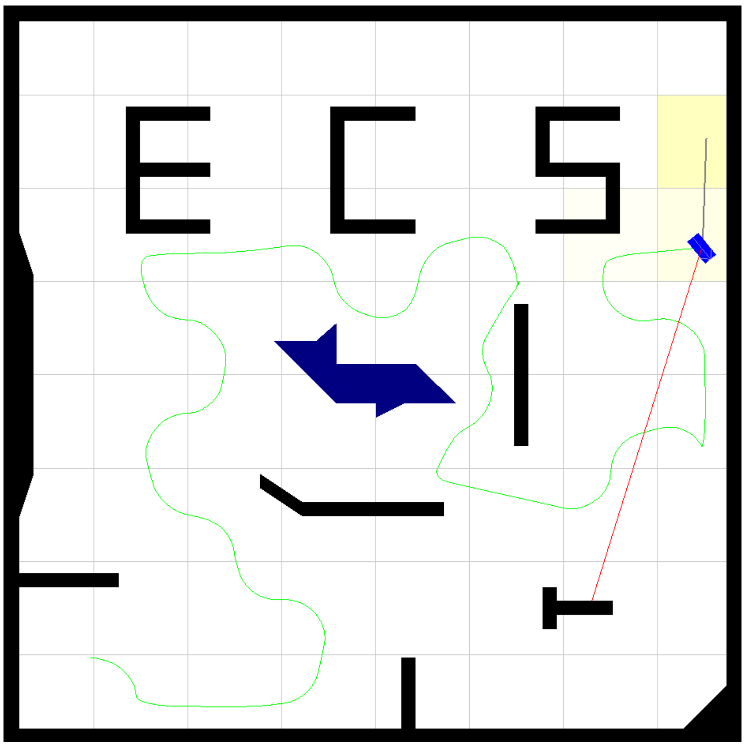


Figure 4.8: The robot's path, shown in green, after a 30 second run of the implied quadrants strategy. The quadrant overlay is enabled displaying the neighbouring cells the strategy is considering. They grey line denotes the vector to the selected quadrant.

Chapter 5

Empirical evaluation

5.1 Test environments

Three environments have been created to test how well the strategies perform in different scenarios. Test environment one, figure 5.1, is fifty units in width and height. It is a relatively open environment and has a beacon entity located centrally.

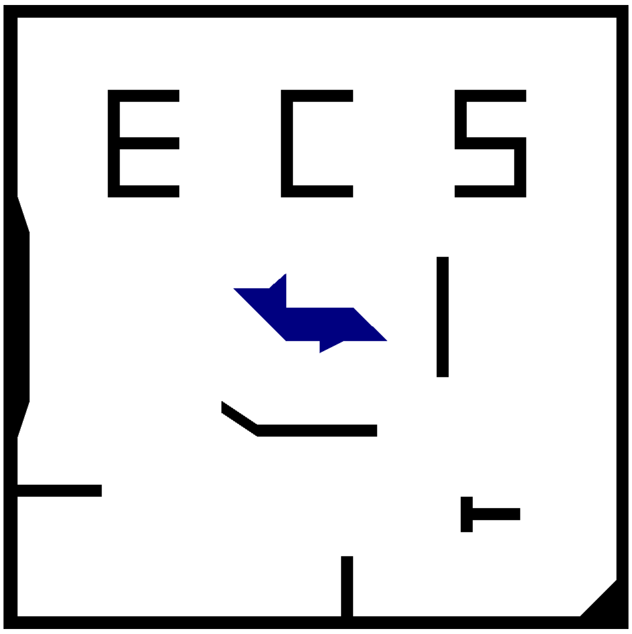


Figure 5.1: Test environment one

Test environment two, figure 5.2, is far more occluded and contains more areas without direct line of sight to a beacon obstacle. It has the same dimensions as test environment one.

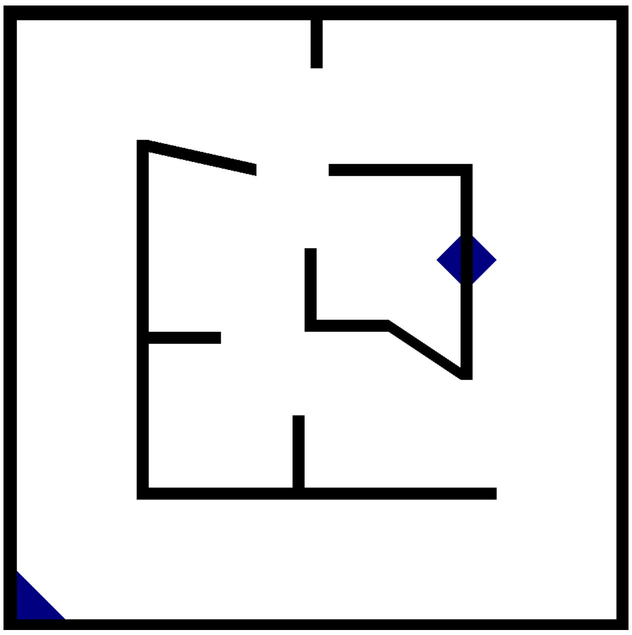


Figure 5.2: Test environment two

Test environment three, figure 5.3, is much larger, containing roughly twice the area of the other environments. It is seventy units in width and height and has a greater number of areas in which the robot could potentially become immobilised. There are two beacon entities located relatively central.

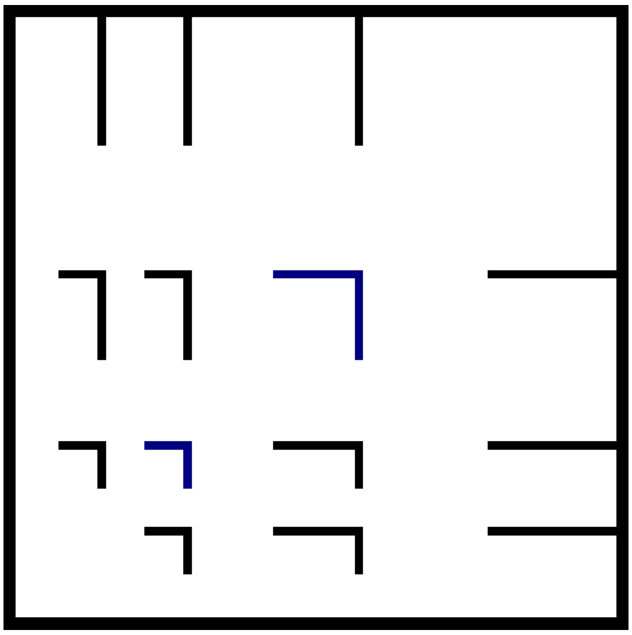


Figure 5.3: Test environment three

5.2 Collecting results

The automated testing routines built into the configuration file are used to collect results. A test configuration file is created for each strategy on each environment. The length of time the strategies may explore each environment is set to two minutes. This is a long enough time for the robot to get round each environment multiple times. On the other hand, it is not too long such that the strategies can explore every environment in huge amounts of detail. The number of tests is set to one hundred for each configuration file, but this can be increased if the results are inconclusive [9]. This equates to forty machine hours to collect results, assuming that each strategy runs at real-time speeds, shown in equation 5.1.

$$120 \text{ seconds} \times 100 \text{ tests} \times 4 \text{ strategies} \times 3 \text{ environments} = 40 \text{ hours} \tag{5.1}$$

5.3 Early exits

One apparent criterion for rating the performance of strategies is how frequently they cause the robot to become stuck. Figure 5.4 shows the number of tests that did not reach the end of their two minute time. Figure 5.5 shows the average early exit times for each environment and strategy.

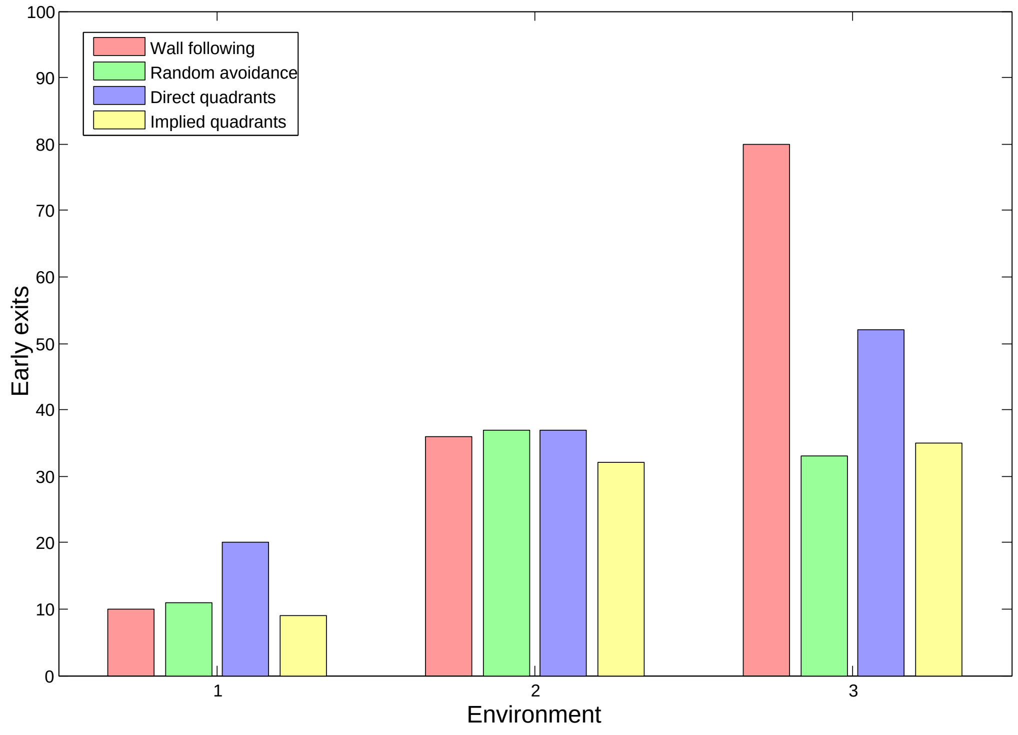


Figure 5.4: A bar chart showing the number of tests that exited early due to the robot getting stuck, for each strategy and environment

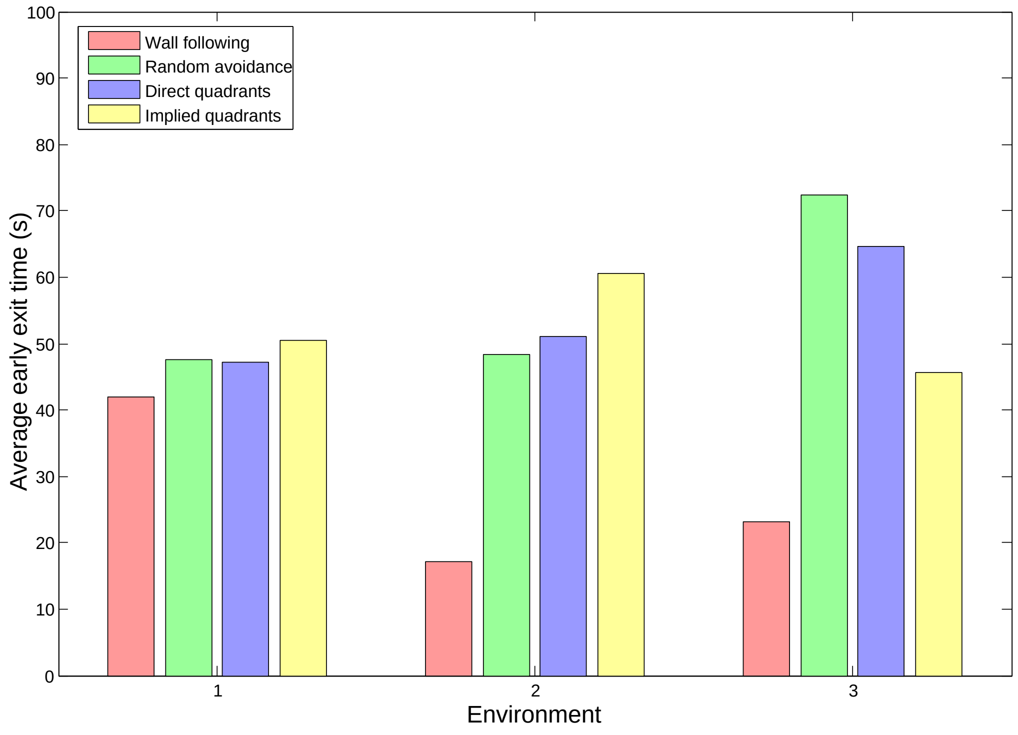


Figure 5.5: A bar chart showing the average time that the robot became stuck, for each strategy and environment

5.4 Completeness vs. accuracy

5.4.1 Environment one

Figure 5.6 shows the completeness and accuracy results for test environment one. Table 5.1 shows many statistical properties of the data sets corresponding to environment one. The table has been coloured to show the strategy rankings for each criteria. Wall following has the highest mean completeness and accuracy, but gets stuck more frequently and earlier than the implied quadrants strategy. The direct quadrants strategy performed the worst for almost all criteria.

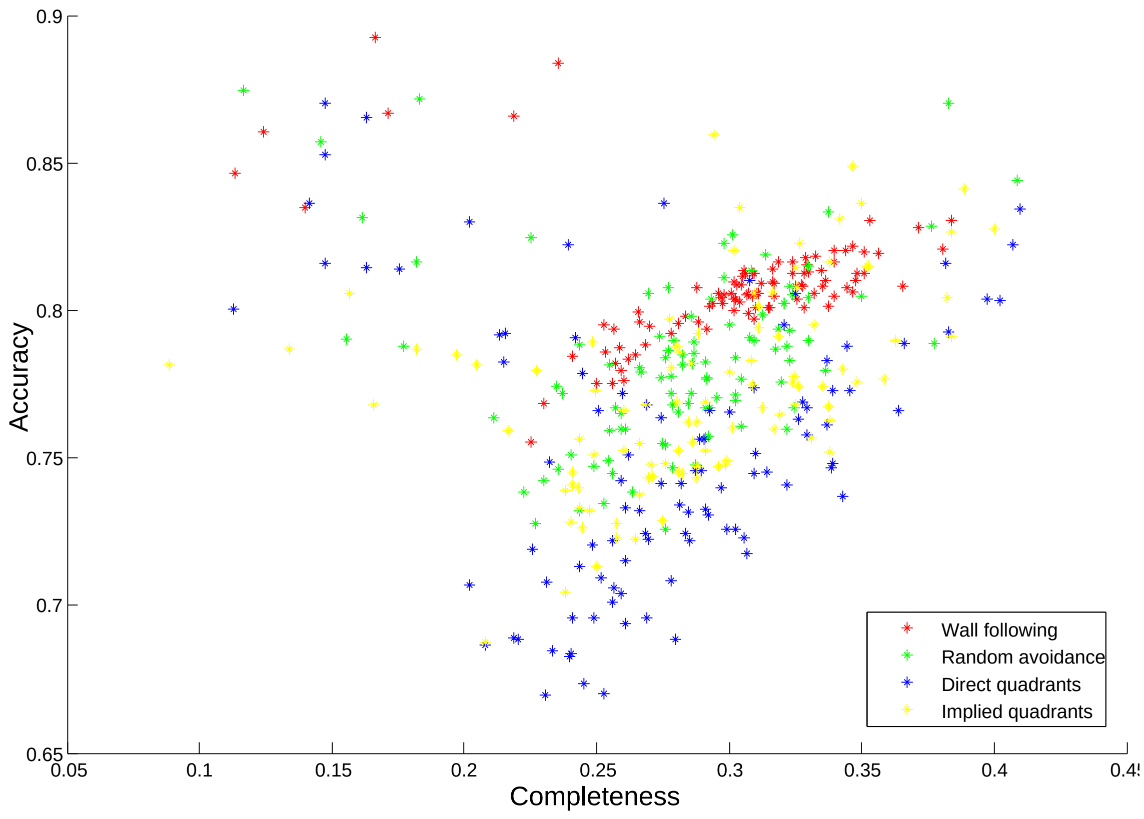


Figure 5.6: Environment one's scatter graph, plotting completeness vs. accuracy for each strategy

	Wall following	Random avoidance	Direct quadrants	Implied quadrants
C_μ	0.2982	0.2805	0.2747	0.2874
C_{σ^2}	0.0025	0.0024	0.0036	0.0029
A_μ	0.8082	0.7838	0.7523	0.7726
A_{σ^2}	0.00042703	0.00091907	0.0022	0.0011
E_μ	42.0143	47.6528	47.1826	50.425
$E_\%$	0.1	0.11	0.2	0.09

Table 5.1: Environment one, strategy statistics. The elements in the far left column are: completeness mean, completeness variance, accuracy mean, accuracy variance, early exit time mean, amount of early exits.

5.4.2 Environment two

Figure 5.7 shows the completeness and accuracy results for test environment one. Table 5.2 shows many statistical properties of the data sets corresponding to environment one. The table has been coloured to show the strategy rankings for each criteria. The implied quadrants strategy came out on top, performing the best for almost all criteria. The wall following strategy was the worst whilst the random avoidance and direct quadrants strategies came out about even. The direct quadrants and implied quadrants are tied for first place for the mean completeness.

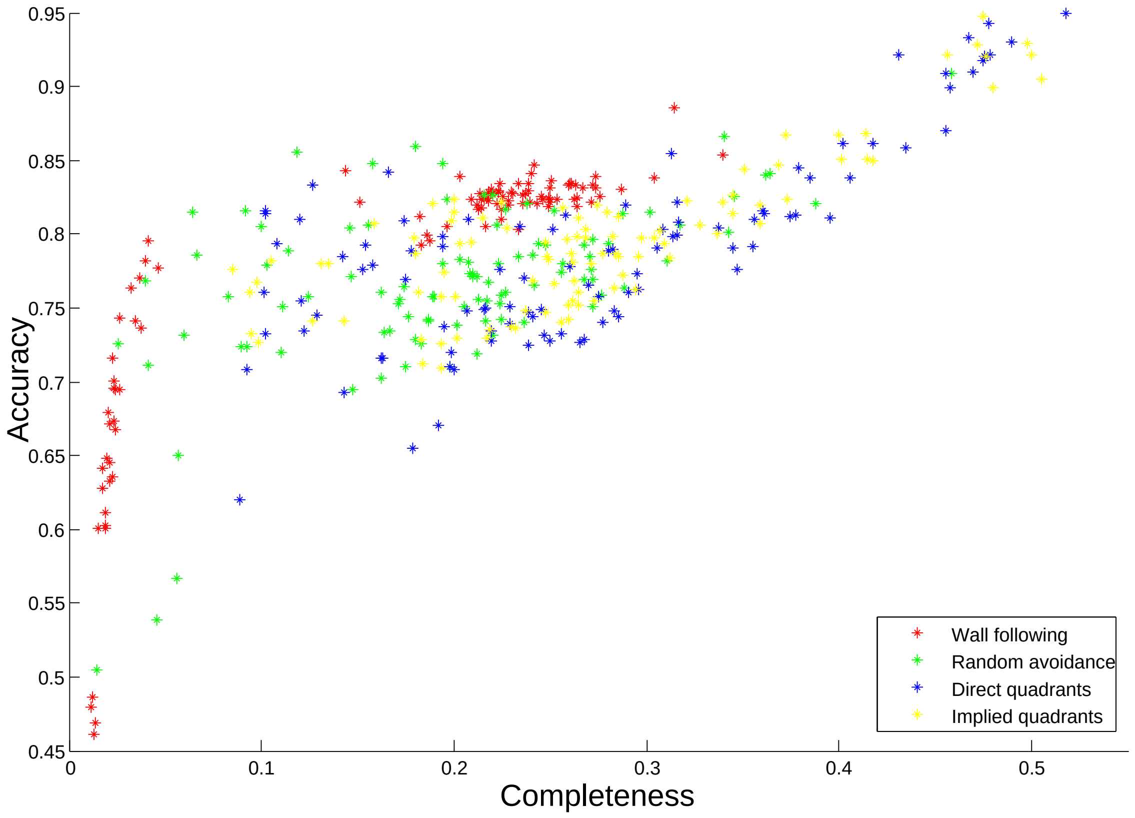


Figure 5.7: Environment two's scatter graph, plotting completeness vs. accuracy for each strategy

	Wall following	Random avoidance	Direct quadrants	Implied quadrants
C_μ	0.1703	0.1992	0.2708	0.2708
C_{σ^2}	0.105	0.0071	0.123	0.0095
A_μ	0.7743	0.7674	0.7903	0.7967
A_{σ^2}	0.0087	0.0034	0.0042	0.0026
E_μ	17.0862	48.2655	51.1066	60.6272
$E_\%$	0.36	0.37	0.37	0.32

Table 5.2: Environment two, strategy statistics. The elements in the far left column are: completeness mean, completeness variance, accuracy mean, accuracy variance, early exit time mean, amount of early exits.

5.4.3 Environment three

Figure 5.8 shows the completeness and accuracy results for test environment one. Table 5.3 shows many statistical properties of the data sets corresponding to environment one. The table has been coloured to show the strategy rankings for each criteria. The random avoidance strategy performed the best, followed closely by the direct quadrants strategy. The wall following strategy collected the most accurate data, but was the least complete, whereas the direct quadrants strategy collected the most complete data, but was the least accurate.

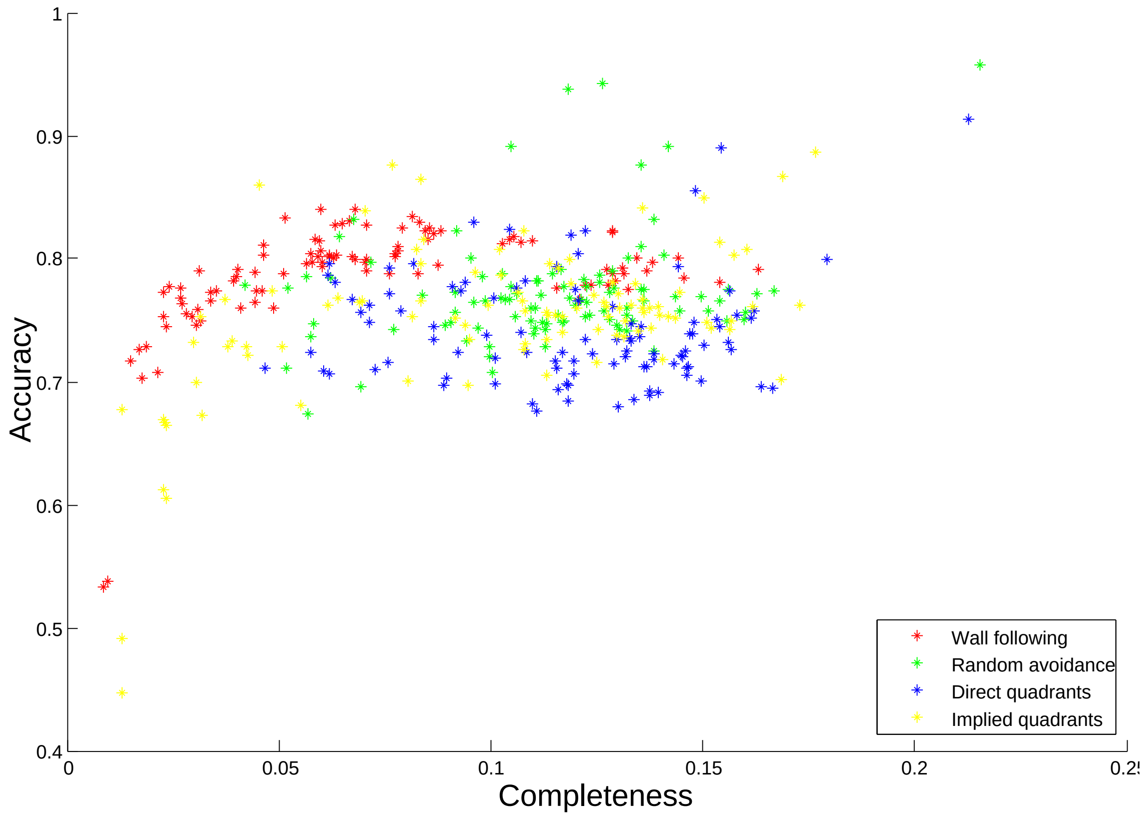


Figure 5.8: Environment three's scatter graph, plotting completeness vs. accuracy for each strategy

	Wall following	Random avoidance	Direct quadrants	Implied quadrants
C_μ	0.071	0.1156	0.1189	0.1017
C_{σ^2}	0.0015	0.0012	0.001	0.0019
A_μ	0.7855	0.7739	0.7423	0.7516
A_{σ^2}	0.0021	0.0021	0.002	0.004
E_μ	23.2453	72.3422	64.6144	45.6485
$E_\%$	0.8	0.33	0.52	0.35

Table 5.3: Environment three, strategy statistics. The elements in the far left column are: completeness mean, completeness variance, accuracy mean, accuracy variance, early exit time mean, amount of early exits.

5.4.4 Combined environments

Figure 5.9 shows the completeness and accuracy results for the combined environment data. Table 5.4 shows many statistical properties of the combined data sets. The table has been coloured to show the strategy rankings for each criteria. The wall following strategy performed the worst for all criteria except the average accuracy, for which it excelled considerably. The random avoidance strategy has minimal variances, but did not do well for the average completeness and accuracy values. The implied quadrants strategy was complete, but lacked accuracy. The implied quadrants strategy is the least likely to become stuck.

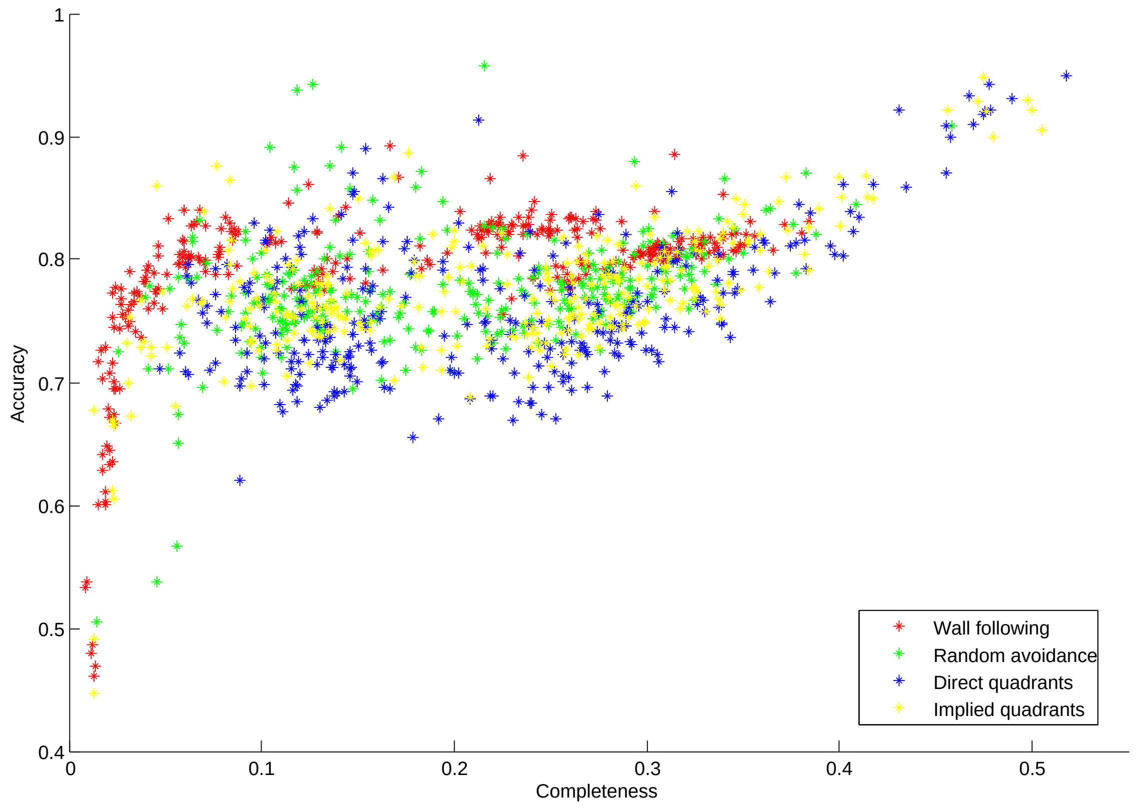


Figure 5.9: Combined environment scatter graph, plotting completeness vs. accuracy for each strategy

	Wall following	Random avoidance	Direct quadrants	Implied quadrants
C_μ	0.1798	0.1984	0.2215	0.22
C_{σ^2}	0.0135	0.0081	0.0109	0.0118
A_μ	0.7893	0.775	0.7617	0.7736
A_{σ^2}	0.0039	0.0022	0.0032	0.0029
E_μ	22.9752	57.9913	56.8307	52.5210
$E_\%$	0.42	0.27	0.3633	0.2533

Table 5.4: Combined environments, strategy statistics. The elements in the far left column are: completeness mean, completeness variance, accuracy mean, accuracy variance, early exit time mean, amount of early exits.

5.5 Ranking strategies

It is important to consider the performance of each strategy in the context of their intended application [9]. Without prior knowledge of the task, it is difficult to compare the strategies and definitively claim which is the best. Instead, a performance ranking system is created that scores each strategy according to a weighted set of parameters. This means that strategies may be evaluated for suitability to a given task by determining the importance of constituent criteria, relevant to completion of the task.

5.5.1 Parameters

The parameters for evaluation of strategies are the same as demonstrated in tables 5.1 through 5.4. A ranking of first place for a particular category scores one point, second place scores two points, etc. These totals are summed up for each criteria based on the individual environment data, shown in table 5.5. This leads to the scoring matrix, equation 5.2, which may be used to score each strategy's performance for a given application by defining the weighting values of C_u, C_{σ^2}, A_μ, A_{σ^2}, E_μ and $E_\%$. It is important to note that the minimum scores are the better performing strategies, whereas the maximum scores are the worst.

	Wall following	Random avoidance	Direct quadrants	Implied quadrants
C_μ	9	8	6	6
C_{σ^2}	8	4	9	9
A_μ	5	8	10	7
A_{σ^2}	7	6	8	8
E_μ	12	6	7	5
$E_\%$	8	7	10	4

Table 5.5: The summation of scores for each criteria and strategy from the individual environment data. The elements in the far left column are: completeness mean, completeness variance, accuracy mean, accuracy variance, early exit time mean, amount of early exits.

$$\begin{pmatrix} Score(\text{Wall following}) \\ Score(\text{Random avoidance}) \\ Score(\text{Direct quadrants}) \\ Score(\text{Implied quadrants}) \end{pmatrix} = \begin{pmatrix} 9 & 8 & 5 & 7 & 12 & 8 \\ 8 & 4 & 8 & 6 & 6 & 7 \\ 6 & 9 & 10 & 8 & 7 & 10 \\ 6 & 9 & 7 & 8 & 5 & 4 \end{pmatrix} \begin{pmatrix} C_\mu \\ C_{\sigma^2} \\ A_\mu \\ A_{\sigma^2} \\ E_\mu \\ E_\% \end{pmatrix} \tag{5.2}$$

5.5.2 Example applications

Path finding

If a robot is assigned to solving the task of path finding, it should attempt to find the shortest path between two points. It may be more efficient for the robot initially to explore the environment before its primary task. The completeness mean and variance criteria are most important for generating an overall understanding as opposed to accuracy which maps smaller areas in greater detail. It is also important that the robot does not become immobilised. The scores are calculated for each strategy based on the estimated weightings matrix, shown in equation 5.3.

$$\begin{pmatrix} 8.3 \\ 6.4 \\ 8.5 \\ 6.1 \end{pmatrix} = \begin{pmatrix} 9 & 8 & 5 & 7 & 12 & 8 \\ 8 & 4 & 8 & 6 & 6 & 7 \\ 6 & 9 & 10 & 8 & 7 & 10 \\ 6 & 9 & 7 & 8 & 5 & 4 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.3 \\ 0 \\ 0 \\ 0 \\ 0.4 \end{pmatrix} \tag{5.3}$$

With a score of 6.1, the implied quadrants strategy should perform the best. This is closely followed by random avoidance with a score of 6.4. Wall following scores 8.3 and was therefore third, whilst the direct quadrants strategy placed last with a score of 8.5. Therefore, for the application of initial exploration, prior to path finding, the implied quadrants strategy is the better choice.

Moving hazardous materials

If a robot is assigned to solving the task of moving hazardous materials, it's main priority should be not to spill the materials. Spills would most likely be caused by colliding with an obstacle at speed. This is most likely to happen if the robot did not expect the obstacle to lie on its path. Therefore, accuracy is far more important than completeness for this application. If the robot became stuck, it would have failed its task, but this should be of less importance than protecting the environment from the hazardous materials. It is preferable that it should become stuck as late as possible to maximise progress. The scores are calculated for each strategy based on the estimated weightings matrix, shown in equation 5.4.

$$\begin{pmatrix} 6.8 \\ 6.9 \\ 8.9 \\ 6.9 \end{pmatrix} = \begin{pmatrix} 9 & 8 & 5 & 7 & 12 & 8 \\ 8 & 4 & 8 & 6 & 6 & 7 \\ 6 & 9 & 10 & 8 & 7 & 10 \\ 6 & 9 & 7 & 8 & 5 & 4 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0.4 \\ 0.4 \\ 0.1 \\ 0.1 \end{pmatrix} \tag{5.4}$$

With a score of 6.8, the wall following strategy should perform the best. The random avoidance and implied quadrants strategies are equally as good for second place, with scores of 6.9. The direct quadrants strategy is, once again, the worst strategy with a score of 8.9. Therefore, the best strategy for the application of moving hazardous materials is wall following.

Automated cleaning

Many robots are used for automated cleaning, such as the Roomba. Similar applications could be for cutting grass, or polishing floors. These robots often have a charging station to return to, therefore, it is important that they do not become stuck whilst carrying out their tasks. If they do become stuck, the time at which they do so is not of great importance as it will still fail its task. Completeness and accuracy are equally important. Completeness is required so that the robot can get around most of the environment. Accuracy is required so that it carries out its task up to the edges of obstacles appropriately. The scores are calculated for each strategy based on the estimated weightings matrix, shown in equation 5.5.

$$\begin{pmatrix} 8.1 \\ 6.7 \\ 9 \\ 5.5 \end{pmatrix} = \begin{pmatrix} 9 & 8 & 5 & 7 & 12 & 8 \\ 8 & 4 & 8 & 6 & 6 & 7 \\ 6 & 9 & 10 & 8 & 7 & 10 \\ 6 & 9 & 7 & 8 & 5 & 4 \end{pmatrix} \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.5 \end{pmatrix} \tag{5.5}$$

With a score of 5.5, the implied quadrants strategy came out on top. The random avoidance strategy came second with a score of 6.7. The wall following strategy came third with a score of 8.1. The direct quadrants strategy came last again, with a score of 9. Therefore, for the application of automated cleaning, the implied quadrants strategy should perform the best.

Safety critical repairs

If a group of robots is assigned the task of repairing some safety critical system, it is important to have a clear understanding of how well the robots will perform. It is hard to judge how many robots to apply to a given area if its results are considerably varied. Therefore, it is arguably better to deploy several poor performing robots than, fewer robots with unpredictable behaviour. Accuracy should be regarded as more important, as the completeness of the environment will be shared amongst the swarm. It does not matter if a robot becomes stuck, because others may take its place and transport it back for repairs. The scores are calculated for each strategy based on the estimated weightings matrix, shown in equation 5.6.

$$\begin{pmatrix} 7 \\ 5.6 \\ 8.8 \\ 8.2 \end{pmatrix} = \begin{pmatrix} 9 & 8 & 5 & 7 & 12 & 8 \\ 8 & 4 & 8 & 6 & 6 & 7 \\ 6 & 9 & 10 & 8 & 7 & 10 \\ 6 & 9 & 7 & 8 & 5 & 4 \end{pmatrix} \begin{pmatrix} 0 \\ 0.4 \\ 0.2 \\ 0.4 \\ 0 \\ 0 \end{pmatrix} \tag{5.6}$$

With a score of 5.6, the random avoidance strategy came out on top. Wall following placed second with a score of 7. The implied quadrants strategy was third with a score of 8.2 and direct quadrants was the worst performer, with a score of 8.8. Therefore, for the application of safety critical repairs, the random avoidance strategy should perform the best.

Chapter 6

Conclusions

6.1 Choosing strategies

From the evidence gathered in section 5, it is apparent that the choice of strategy is not a simple decision. Instead, it requires some analysis of the context in which the strategy is to be applied. For the general case, it may be useful to examine the rankings in performance of each strategy, for the example applications presented in section 5.5.2. This makes the assumption that the example applications are somewhat representative of real-world scenarios and that their frequency of occurrence is uniform.

If no prior knowledge of the application of the strategies is known, then the implied quadrants strategy should be chosen as this performed the best for the example applications, scoring the highest in all four tests. The random avoidance and wall following strategies each came out on top for one example application. However, the random avoidance strategy ranked above wall following for applications that were won by the implied quadrants strategy. Therefore, in the general case, random avoidance is slightly better than wall following. The direct quadrants strategy performed the worst in every example application, indicating that it should rarely be chosen for exploration.

It seems curious that the two quadrant-based approaches to exploration scored at opposite ends of the spectrum. Intuitively, it would seem that this phenomenon arises due to the strategies' paradigms for exploration. The direct quadrants strategy attempts to travel towards neighbour quadrants, with minimal turns, by rotating on the spot and travelling in straight lines. The implied quadrants strategy alternates between forwards and turning towards the preferred quadrant. This may be perceived as being more relaxed in the context of control over its heading. The direct quadrants strategy is likely to run into more collisions as a result of this behaviour which is supported by the bar chart showing the number of early exists, figure 5.4.

For larger, more occluded environments, the gap in performance between random avoidance and wall following, tends to widen. This is supported by the combination of statistical tables 5.1, 5.2 and 5.3. The occlusion in environment two is greater than that of environment one, and environment three is larger than environment two. It seems likely that random avoidance strategy performs better on large, occluded environments because wall following is ultimately flawed in that it rarely ventures into the centre of environments.

The general procedure for choosing a strategy for a known application, involves analysing the task for the importance of constituent criteria. A weightings matrix should be estimated for the application, based on the importance of each criteria. The weightings matrix should be post-multiplied with the score matrix to obtain 'performance' scores for each strategy. The strategy with the minimal score should be selected for the given application. To calculate the score matrix for an additional strategy, it should be ranked in terms of each criteria based on statistical comparison with the other strategies. Section 5.5 explains this process in greater detail.

6.2 Critical Evaluation

The validity of the work presented here is dependent upon the realism incorporated into the simulation. The careful application of noise within the system means that it is somewhat representative of a real-world scenario. Basic collision handling is simulated, although this could be made more realistic by considering the momentum and angular direction of the robot on collision [11]. Additionally, the robot movements do not take into account acceleration, which would affect robots in the real-world.

One issue to consider, is whether the behavioural module and explorative strategies could be deployed on a micro-controller. The programming language selected for the software was intentionally chosen to simplify the porting of code, as a lot of low-level micro-controllers support C-based languages, see section 1.2.1. The object-oriented approach to software development means that it is much easier to port specific modules to devices, as they are mostly decoupled from each other.

The behavioural strategies presented here range from simplistic wall following, or random avoidance, to relatively complex quadrant-based approaches. Although the rankings of strategies in various situations are useful for potential application design, the more useful ideas are presented in the process of empirical evaluation, chapter 5. It is arguably more useful for application designers to construct their own strategies, and then rate their performance based on the guidelines presented here.

The use of the grid model shows how a high level construct, may be created from simple sensory data, and applied to the field of autonomous exploration. It may not be the best model for the task, but it provides one possible solution. A benefit of using the grid model is that it is finite. It can indefinitely map points and will maintain a fixed size in memory. Intuitively, this is supported by the concepts of information theory, as repeated data does not carry additional information.

The significance of the completeness and accuracy metrics requires consideration. Ideally, they should be entirely independent variables, but in practice this is unrealistic. If accuracy is poor, the number of cells affected by a sensory measurement is large as its covariance causes the bounding box to span more cells. As a consequence the completeness value increases as there are more cells containing non-zero values. It may be more beneficial to calculate the completeness metric based on the sparsity of information over the grid. In practice, the bias in completeness is present for all strategies, so its affects are negligible.

6.3 Management account

The project went to schedule, requiring almost no changes from the initial Gantt chart, see appendix figures 1 through 6. Extra time was available towards the end of the project to carry out additional software testing. 18 unit tests were carried out, instead of 10. The graph window feature was decided fairly late into the project as an extension to help the user visualise the progress of each strategy as a metric. The time spent implementing this feature would not have been available if it were not for careful time management.

The use of supervisor meetings was incredibly useful throughout the project. They helped to focus the work-flow to ensure the project was on track. They helped to define the scope of the project and decide on the order of implementation for software modules. Additional perspective as to the direction of the project was valuable.

6.4 Reflection

Overall, a great deal was learnt over the duration of the project, in both software development and the field of autonomous exploration in robotics. At the beginning of the project, the testing of strategies was ambiguous. In contrast, the empirical evaluation, chapter 5 is one of the most formal and precise processes presented as part of this report. For the study of autonomous exploration, the process of evaluation of strategies is arguably, the most significant piece of work presented here.

If the project were repeated, additional realism would be incorporated, with the most emphasis placed on collision handling. A more accurate model for collisions could be supported such that the angular direction and momentum of the robot are considered. Collision detection should have been handled at a much lower level than the grid model for greater success. The most recent points should have had a higher impact on the collision detection as opposed to treating all sensory data uniformly.

If an existing software simulation could have been utilised for the testing of strategies, this would have allowed much longer for strategy development and analysis. However, a suitable simulation could not be located that provided such low level control over noise and collisions within the system. If the time spent creating additional data views had been reallocated, extra time could have been spent developing strategies.

A slight annoyance with the testing stage was the process of automation, for two reasons. The first is that a file must be created for each strategy on each environment. This is wasteful as many of the tests share the same environment. It would have been more efficient to reference another file for the environment in each test configuration, which would reduce the level of redundancy. The second annoyance with automation was that user interaction was required to start each test configuration file for the project. The level of automation could have been improved by implementing a top-level test file containing references to a list of configurations.

If the project were to be developed further, the obvious extension would be to develop new explorative strategies and increase the quantity of testing. Attempts could be made to create a three-dimensional view of the simulation as it runs. This could lead to more interesting views for the simulation data. The usefulness of the grid model could be analysed in greater detail, potentially leading to a formalised approach for building high level constructs that may be used by autonomous exploration. The three-dimensional exploration problem could be considered. A quadrotor flying robot could be simulated for the three-dimensional case, perhaps using the Kinect as a sensory input device.

References

[1] E. Angel, *Interactive Computer Graphics, A Top-Down Approach Using OpenGL*, Pearson Education Inc., 2009.

[2] G.E.P Box and M.E Muller, *A Note on the Generation of Random Normal Deviates*, Statistical Techniques Research Group, Princeton, 1958.

[3] R.A Brooks, *Intelligence without Representation*, MIT Artificial Intelligence Laboratory, Cambridge, 1987.

[4] R.A. Brooks, *Elephants Dont Play Chess*, MIT Artificial Intelligence Laboratory, Cambridge, 1990.

[5] H. Choset, K.M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L.E. Kavraki and S. Thrun, *Principles of Robot Motion, Theory, Algorithms, and Implementations*, The MIT Press, Cambridge, 2005.

[6] R.W. Ehrich, *2D Liang-Barsky Clipping*, Pixel Education, 2009.
Available: http://goo.gl/E8WwQ

[7] C. Ericson, *Real Time Collision Detection*, Morgan Kaufman Publishers, London, 2005.

[8] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modelling Language*, Addison Wesley, Boston, 2003.

[9] T.Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning, Data Mining, Inference, and Prediction*, Springer, California, 2008.

[10] P.S. Maybeck, *Stochastic models, estimation and control*, Academic Press, New York, 1979.

[11] I. Millington, *Game Physics Engine Development*, Morgan Kaufman Publishers, London, 2007.

[12] H.D. Ruiter, *Frame-Rate Independent Animation using GLUT*, HDR Lab, 2008.
Available: http://goo.gl/DMNi4

[13] S.J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, Pearson Education International, London, 2003.

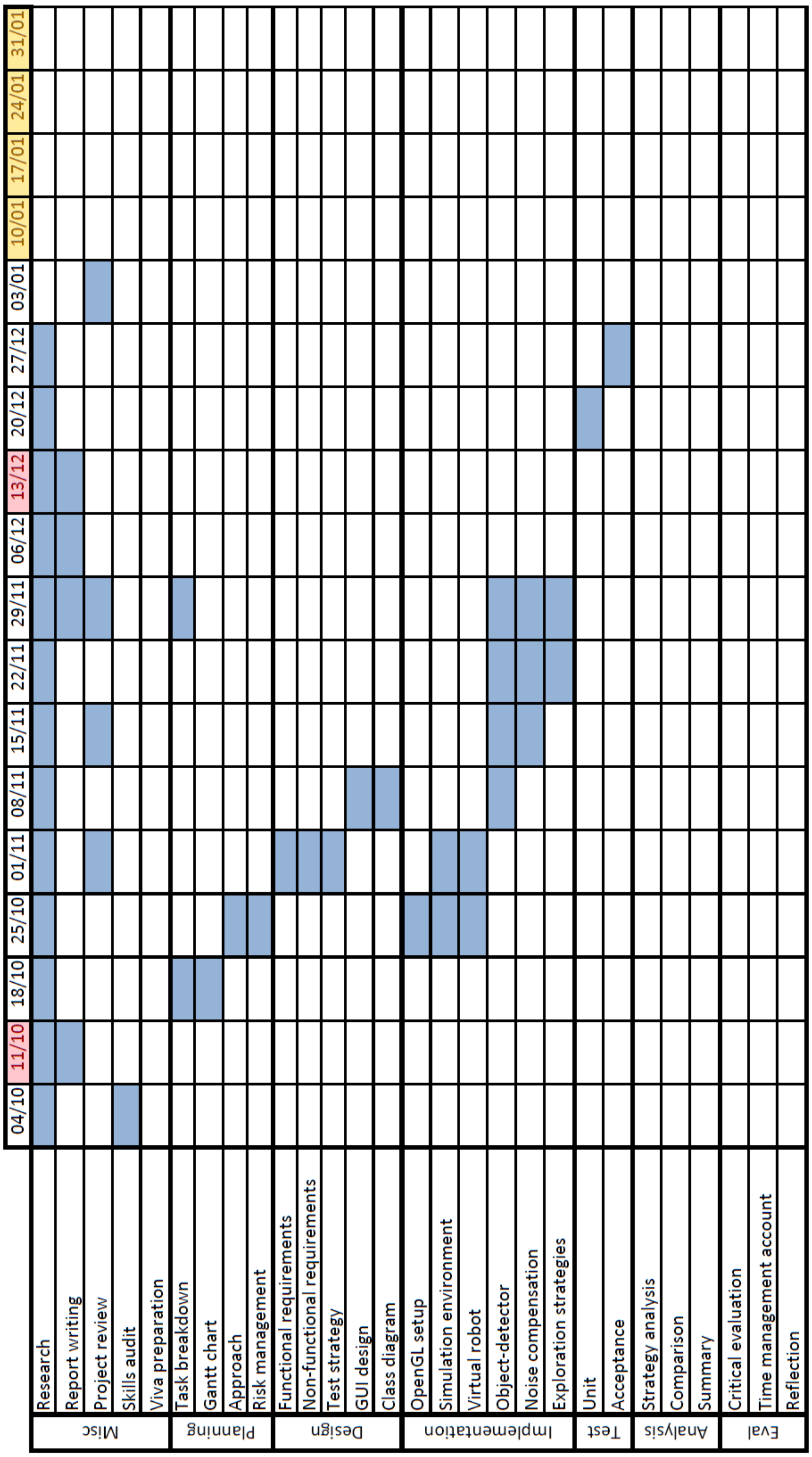
[14] S. Thrun, W. Burgard and D. Fox, *Probabilistic Robotics*, The MIT Press, Cambridge, 2005.

[15] B. Webb, *Animals versus animats: or why not the real iguana?*, School of Informatics, Edinburgh, 2009.

Appendices

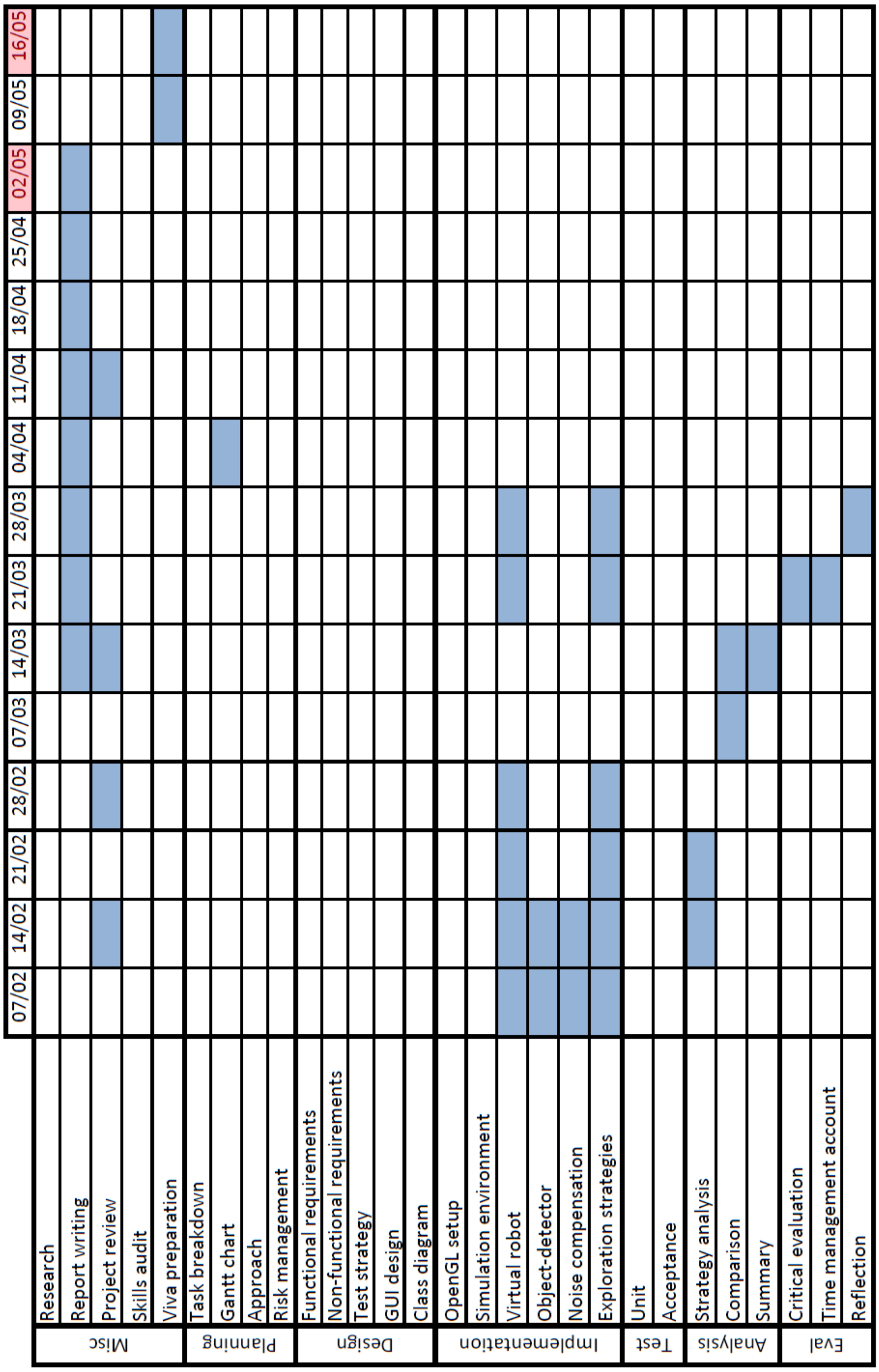
```
<!DOCTYPE root [
<!ELEMENT root (tests, environment, robot, behaviour, display)>
<!ELEMENT tests (#PCDATA)>
<!ELEMENT environment (polygon*)>
<!ELEMENT polygon (vertex+)>
<!ELEMENT vertex EMPTY>
<!ELEMENT robot (start, moveRate, turnRate, lidarRate, noise)>
<!ELEMENT start (vertex)>
<!ELEMENT moveRate (#PCDATA)>
<!ELEMENT turnRate (#PCDATA)>
<!ELEMENT lidarRate (#PCDATA)>
<!ELEMENT noise (#PCDATA)>
<!ELEMENT behaviour (grid, strategy)>
<!ELEMENT grid (startGran, minGran, split)>
<!ELEMENT startGran (#PCDATA)>
<!ELEMENT minGran (#PCDATA)>
<!ELEMENT split (#PCDATA)>
<!ELEMENT strategy (#PCDATA)>
<!ELEMENT display (default, runtime)>
<!ELEMENT default (robotV?, lidarV?, obstaclesV?, beaconsV?, overlayV?,
     collisionV?, detectorV?, ellipsesV?, pathV?, gridV?, mappingsV?,
    verticesV?, trackerV?, debugV?)>
<!ELEMENT robotV EMPTY>
<!ELEMENT lidarV EMPTY>
<!ELEMENT obstaclesV EMPTY>
<!ELEMENT beaconsV EMPTY>
<!ELEMENT overlayV EMPTY>
<!ELEMENT collisionV EMPTY>
<!ELEMENT detectorV EMPTY>
<!ELEMENT ellipsesV EMPTY>
<!ELEMENT pathV EMPTY>
<!ELEMENT gridV EMPTY>
<!ELEMENT mappingssV EMPTY>
<!ELEMENT verticesV EMPTY>
<!ELEMENT trackerV EMPTY>
<!ELEMENT quadrantsV EMPTY>
<!ELEMENT seekV EMPTY>
<!ELEMENT debugV EMPTY>
<!ELEMENT runtime (#PCDATA)>

<!ATTLIST environment width CDATA #REQUIRED>
<!ATTLIST environment height CDATA #REQUIRED>
<!ATTLIST polygon beacon (true|false) 'false'>
<!ATTLIST vertex x CDATA #REQUIRED>
<!ATTLIST vertex y CDATA #REQUIRED>
<!ATTLIST robot width CDATA #REQUIRED>
<!ATTLIST robot height CDATA #REQUIRED>
<!ATTLIST display width CDATA #REQUIRED>
<!ATTLIST display height CDATA #REQUIRED>
]>
```

Listing 1: Document Type Definition for the configuration file



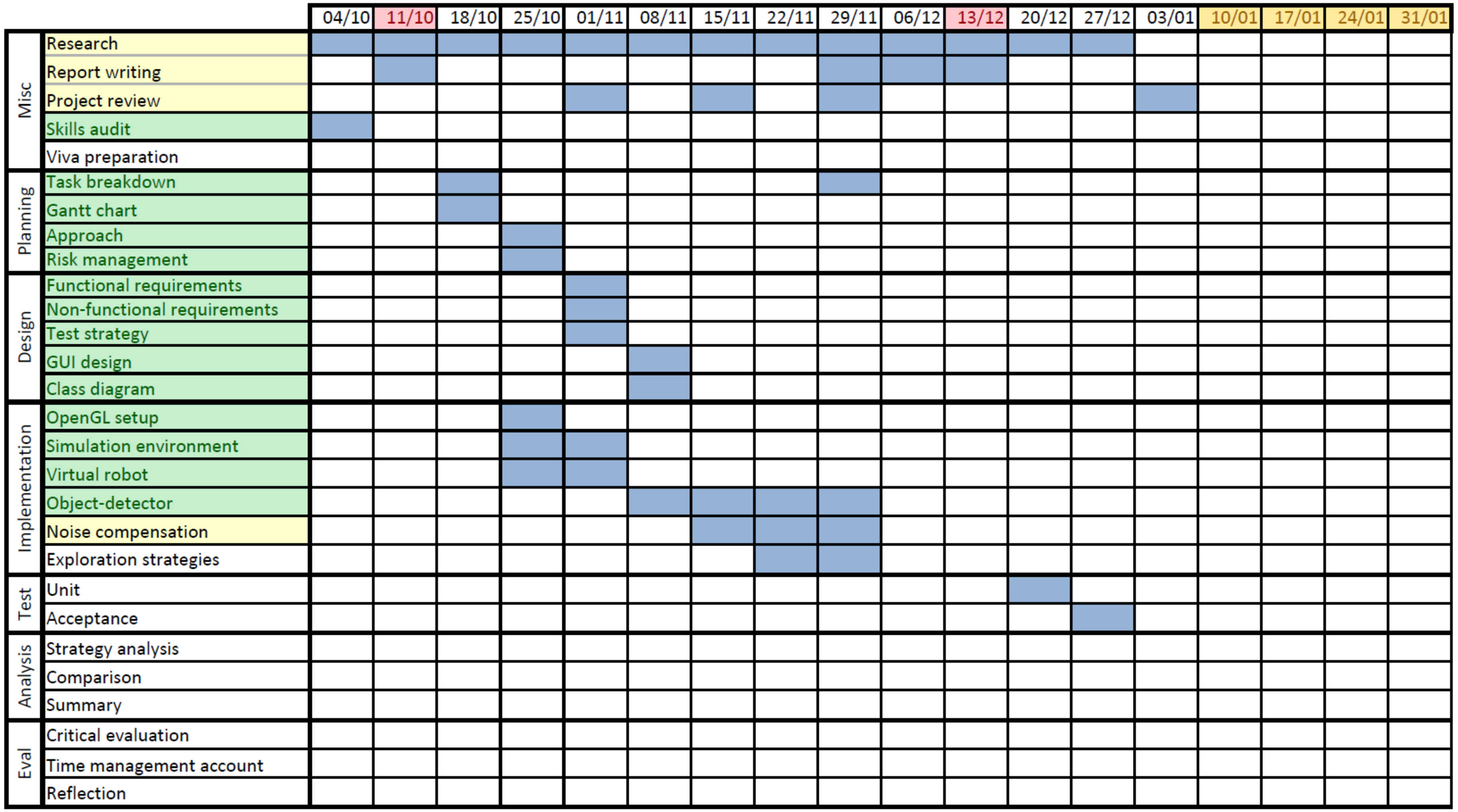
		04/10	11/10	18/10	25/10	01/11	08/11	15/11	22/11	29/11	06/12	13/12	20/12	27/12	03/01	10/01	17/01	24/01	31/01
Misc	Research	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦					
	Report writing		🟦							🟦	🟦	🟦							
	Project review					🟦		🟦		🟦					🟦				
	Skills audit	🟦																	
	Viva preparation																		
Planning	Task breakdown			🟦						🟦									
	Gantt chart			🟦															
	Approach				🟦														
	Risk management				🟦														
Design	Functional requirements					🟦													
	Non-functional requirements					🟦													
	Test strategy					🟦													
	GUI design						🟦												
	Class diagram						🟦												
Implementation	OpenGL setup				🟦														
	Simulation environment				🟦	🟦													
	Virtual robot				🟦	🟦													
	Object-detector						🟦	🟦	🟦	🟦									
	Noise compensation							🟦	🟦	🟦									
	Exploration strategies								🟦	🟦									
Test	Unit												🟦						
	Acceptance													🟦					
Analysis	Strategy analysis																		
	Comparison																		
	Summary																		
Eval	Critical evaluation																		
	Time management account																		
	Reflection																		

Figure 1: Initial Gantt chart, page 1



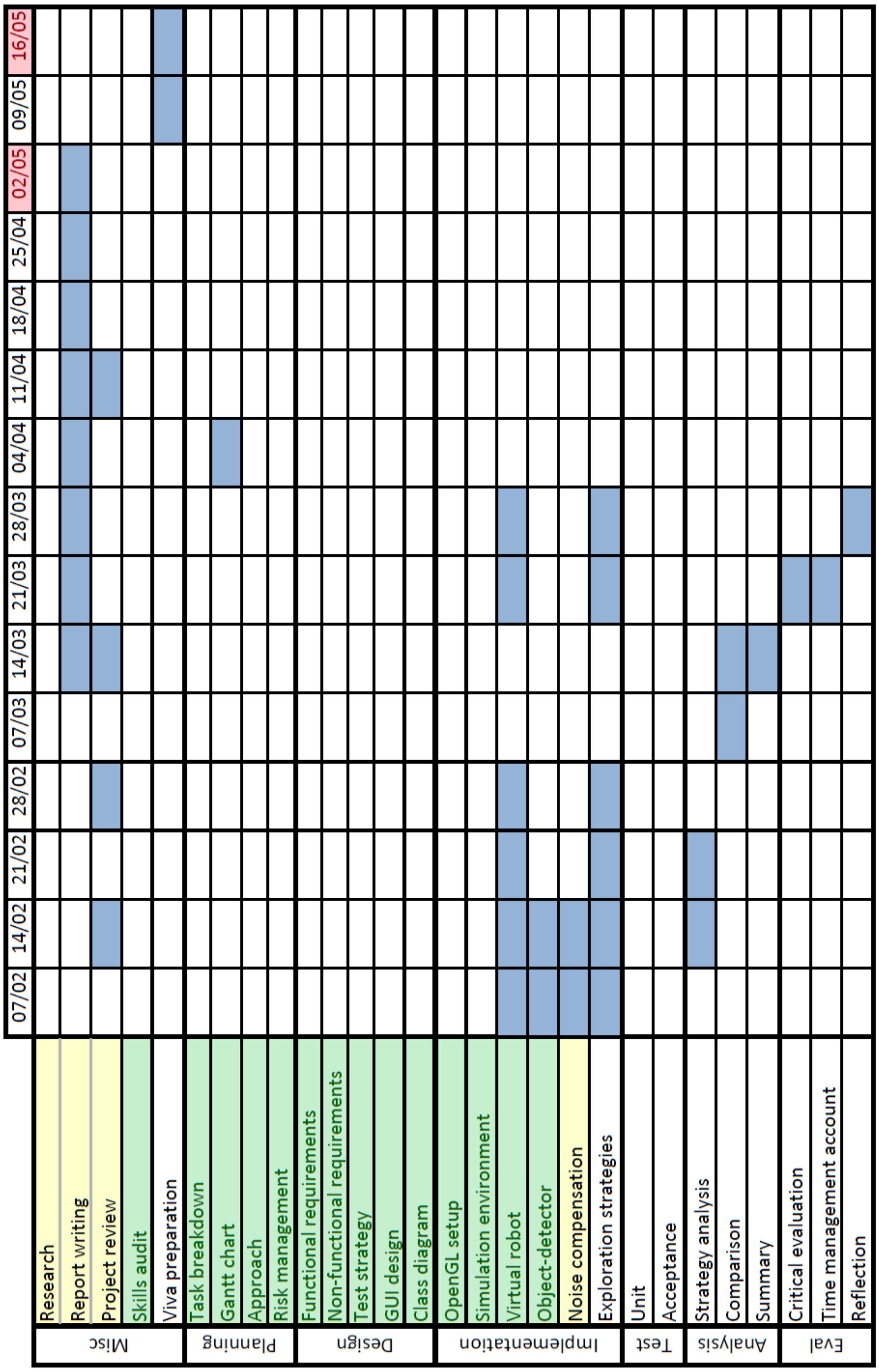
		07/02	14/02	21/02	28/02	07/03	14/03	21/03	28/03	04/04	11/04	18/04	25/04	02/05	09/05	16/05
Misc	Research															
	Report writing						■	■	■	■	■	■	■	■		
	Project review		■		■		■				■					
	Skills audit															
	Viva preparation														■	■
Planning	Task breakdown															
	Gantt chart									■						
	Approach															
	Risk management															
Design	Functional requirements															
	Non-functional requirements															
	Test strategy															
	GUI design															
	Class diagram															
Implementation	OpenGL setup															
	Simulation environment															
	Virtual robot	■	■	■	■			■	■							
	Object-detector	■	■													
	Noise compensation	■	■													
	Exploration strategies	■	■	■	■			■	■							
Test	Unit															
	Acceptance															
Analysis	Strategy analysis		■	■												
	Comparison					■	■									
	Summary						■									
Eval	Critical evaluation							■								
	Time management account							■								
	Reflection								■							

Figure 2: Initial Gantt chart, page 2



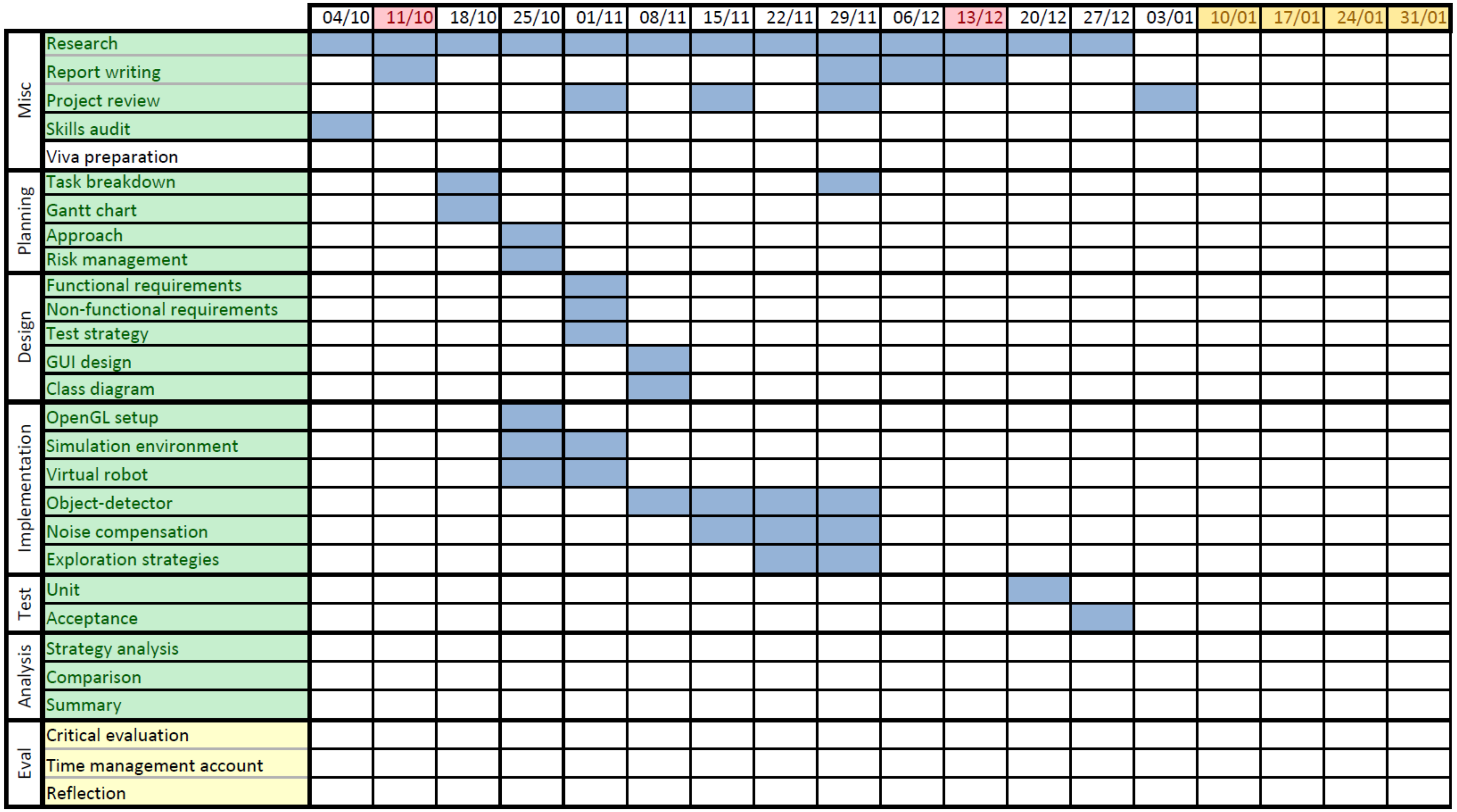
		04/10	11/10	18/10	25/10	01/11	08/11	15/11	22/11	29/11	06/12	13/12	20/12	27/12	03/01	10/01	17/01	24/01	31/01
Misc	Research	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦					
	Report writing		🟦							🟦	🟦	🟦							
	Project review					🟦		🟦		🟦					🟦				
	Skills audit	🟦																	
	Viva preparation																		
Planning	Task breakdown			🟦						🟦									
	Gantt chart			🟦															
	Approach				🟦														
	Risk management				🟦														
Design	Functional requirements					🟦													
	Non-functional requirements					🟦													
	Test strategy					🟦													
	GUI design						🟦												
	Class diagram						🟦												
Implementation	OpenGL setup				🟦														
	Simulation environment				🟦	🟦													
	Virtual robot				🟦	🟦													
	Object-detector						🟦	🟦	🟦	🟦									
	Noise compensation							🟦	🟦	🟦									
	Exploration strategies								🟦	🟦									
Test	Unit												🟦						
	Acceptance													🟦					
Analysis	Strategy analysis																		
	Comparison																		
	Summary																		
Eval	Critical evaluation																		
	Time management account																		
	Reflection																		

Figure 3: Intermediate Gantt chart, page 1



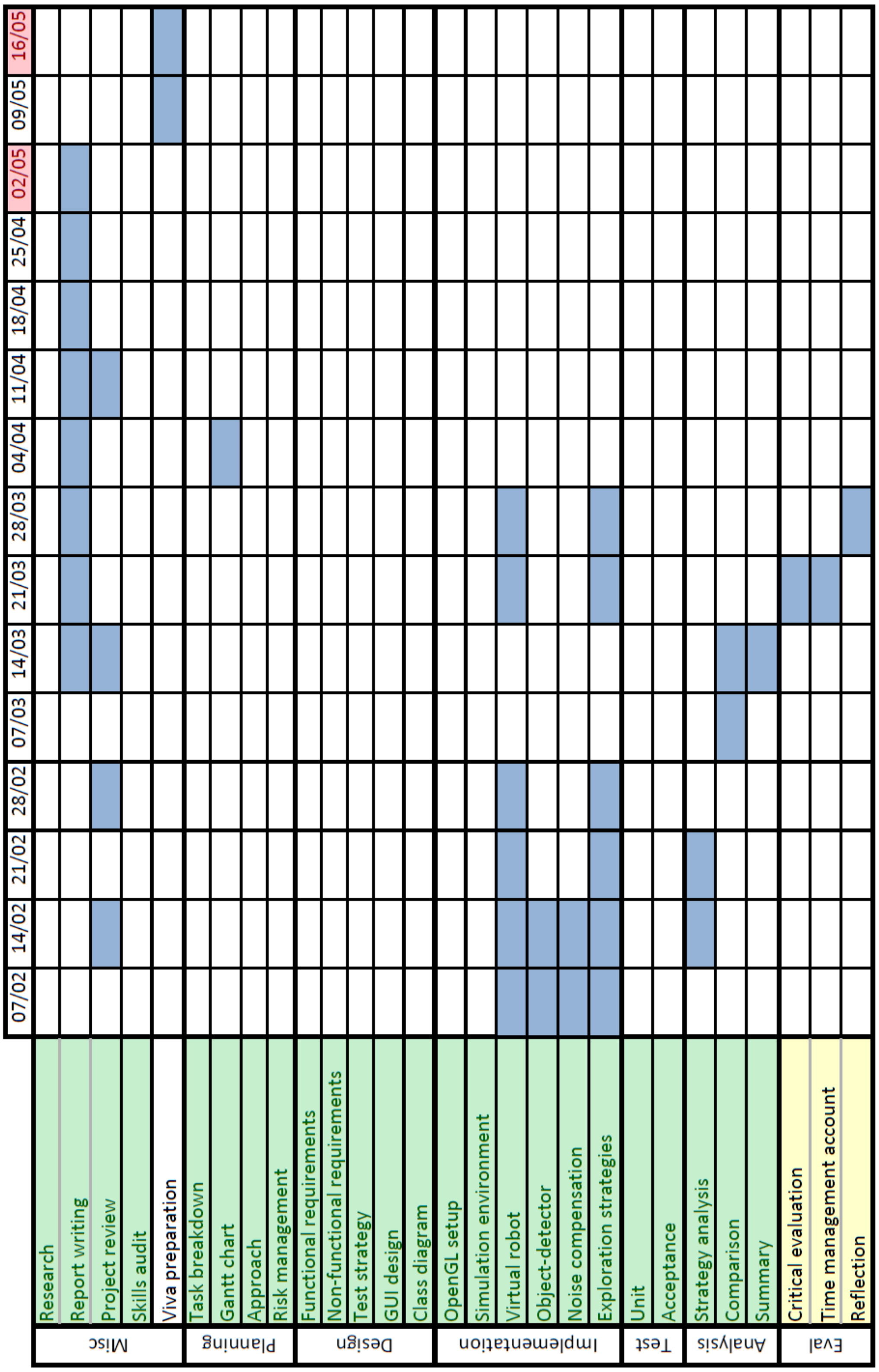
		07/02	14/02	21/02	28/02	07/03	14/03	21/03	28/03	04/04	11/04	18/04	25/04	02/05	09/05	16/05
Misc	Research															
	Report writing						🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦		
	Project review		🟦		🟦		🟦				🟦					
	Skills audit															
	Viva preparation														🟦	🟦
Planning	Task breakdown															
	Gantt chart									🟦						
	Approach															
	Risk management															
Design	Functional requirements															
	Non-functional requirements															
	Test strategy															
	GUI design															
	Class diagram															
Implementation	OpenGL setup															
	Simulation environment															
	Virtual robot	🟦	🟦	🟦	🟦			🟦	🟦							
	Object-detector	🟦	🟦													
	Noise compensation	🟦	🟦													
	Exploration strategies	🟦	🟦	🟦	🟦			🟦	🟦							
Test	Unit															
	Acceptance															
Analysis	Strategy analysis		🟦	🟦												
	Comparison					🟦	🟦									
	Summary						🟦									
Eval	Critical evaluation							🟦								
	Time management account							🟦								
	Reflection								🟦							

Figure 4: Intermediate Gantt chart, page 2



		04/10	11/10	18/10	25/10	01/11	08/11	15/11	22/11	29/11	06/12	13/12	20/12	27/12	03/01	10/01	17/01	24/01	31/01
Misc	Research	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦					
	Report writing		🟦							🟦	🟦	🟦							
	Project review					🟦		🟦		🟦					🟦				
	Skills audit	🟦																	
	Viva preparation																		
Planning	Task breakdown			🟦						🟦									
	Gantt chart			🟦															
	Approach				🟦														
	Risk management				🟦														
Design	Functional requirements					🟦													
	Non-functional requirements					🟦													
	Test strategy					🟦													
	GUI design						🟦												
	Class diagram						🟦												
Implementation	OpenGL setup				🟦														
	Simulation environment				🟦	🟦													
	Virtual robot				🟦	🟦													
	Object-detector						🟦	🟦	🟦	🟦									
	Noise compensation							🟦	🟦	🟦									
	Exploration strategies								🟦	🟦									
Test	Unit												🟦						
	Acceptance													🟦					
Analysis	Strategy analysis																		
	Comparison																		
	Summary																		
Eval	Critical evaluation																		
	Time management account																		
	Reflection																		

Figure 5: Final Gantt chart, page 1



		07/02	14/02	21/02	28/02	07/03	14/03	21/03	28/03	04/04	11/04	18/04	25/04	02/05	09/05	16/05
Misc	Research															
	Report writing						▨	▨	▨	▨	▨	▨	▨	▨		
	Project review		▨		▨		▨				▨					
	Skills audit															
	Viva preparation														▨	▨
Planning	Task breakdown															
	Gantt chart									▨						
	Approach															
	Risk management															
Design	Functional requirements															
	Non-functional requirements															
	Test strategy															
	GUI design															
	Class diagram															
Implementation	OpenGL setup															
	Simulation environment															
	Virtual robot	▨	▨	▨	▨			▨	▨							
	Object-detector	▨	▨													
	Noise compensation	▨	▨													
	Exploration strategies	▨	▨	▨	▨			▨	▨							
Test	Unit															
	Acceptance															
Analysis	Strategy analysis		▨	▨												
	Comparison					▨	▨									
	Summary						▨									
Eval	Critical evaluation							▨								
	Time management account							▨								
	Reflection								▨							

Figure 6: Final Gantt chart, page 2