The Development of a Relative Point and a Relative Plane SLAM algorithms

By

Jay Kraut

A Thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

Abstract

A current research question in robotics is how to map an area without accurate odometry in real time. This thesis looks into this question first from a system perspective, and then focuses on developing a real time simultaneous localization and mapping (SLAM) algorithm.

This thesis follows a previously successful project in developing a robot to map an indoor environment, and uses the previous project's design process as a template. Like the previous project, each part works individually but this time the parts do not work together. This is a textbook example of the “second system effect” described in software engineering literature. Following the attempt at developing a complete mapping system, the thesis is refocused on designing a SLAM algorithm. Planes are chosen to be the input to the SLAM problem and they are generated from 3D point clouds using the expectation maximization (EM) algorithm. After several failed attempts at creating an algorithm, the knowledge gained in those attempts led to the Relative Plane algorithm and then the Relative Point algorithm.

There are many different algorithms that have been shown to solve the SLAM problem depending on the type of input data. Many of these algorithms use some form of cumulative current position as a state variable and only store landmarks in their globally mapped form, discarding past data. This thesis takes a different approach in not using current position as a cumulative state variable and storing and using past data. Landmarks are mapped relative to each other in their untransformed states and use either three points or one plane to maintain translation and rotation invariance. The Relative algorithms can use both current and past data for accuracy purposes. Using this approach, the SLAM problem is solved by data structures and algorithms rather than probabilistic modeling.

The Relative algorithms are shown to be good solutions to the simulated SLAM problems tested in this thesis. In particular the Relative Point algorithm is shown to have a worst case computation complexity of $O(n_s \log n_s)$. n_s is the average quantity of points observed in a given observation and is not related to the total quantity of points on the map. The Relative Point algorithm is able to identify points with movement that is not correlated to the viewpoint at a low cost, and has comparable accuracy to a 6D no odometry Extended Kalman Filter.

Acknowledgements

I would like to thank all the people that have helped me from my undergraduate years to present as this thesis has been the culmination of my work from all those years. Starting with my Advisor Professor McLeod, Tom Kaminski, Marek Laskowski, Jeff Diamond, Professor Baltes, Professor Peters, Professor Kinsner ,Leonard Cohen, Doug Cornelson, Clint Stuart, U of M machine shop and tech shop staff, the UMGF fellowship, Google, TRLabs, Vansco Electronics, and without regards to the order, my Family.

Table of Contents

Table of Figures

Glossary

SLAM Simultaneous localization and mapping

EM Expectation maximization

ICP Iterative Closest Point

EKF Extended Kalman Filter

BSP Binary space partitions

API Application Programming Interface

PVS Potential Visible Set

CGS constructive solid geometry

SVD singular value decomposition

MFC Microsoft Foundation Class.

$O(n^2)$, O(nlogn), O(n), Big-O notation. Describes the relationship of the number of computations required versus the number of elements n to compute. For real time computation, generally the computation complexity cannot be worse than O(nlogn).
RltPlane a name given to the data structure (class) which stores untransformed planes. The key functionality is to match incoming planes to the last plane in storage and the quick retrieval of untransformed planes given an iteration for the RltXPoint.

RltXPoint a name given to the data structure (class) which stores links to RltPlanes and calculates their relative offsets from each other. The RltXPoints are linked to other RLXPoints with linking planes and starting at the first observed one they can be used to calculate the map.

Key points. The key point is the corner point which is static in a given interval.

Motion model. When only one partial plane is visible it is not possible to locate the robot with a full range of movements. So it is assumed that the robot uses a restricted motion model (no side to side movement) and doing this allows for the algorithm to continue to operate.

Chapter 1 Introduction

1.1 Problem definition

One of the goals in robotics research is for a robot to be able to map an area without being given accurate position data. This is generally referred to as the simultaneous localization and mapping (SLAM) problem. The problem can be stated as: how does one place objects on a map if the current position is unknown or inaccurate, and how does one know the exact current position if there is not map to reference?

There are many different solutions to the SLAM problem. This thesis looks at the problem from the perspective that the only inputs are points in 3D that can be generated by computer vision algorithms and there is no odometry data. The points can either be used an the input to the SLAM algorithm or they can first be transformed into planes which are then used as the input. This thesis takes a representational approach of solving SLAM using algorithms and data structures rather than relying on one of the known techniques.

Most SLAM algorithms typically perform only one task. Given landmark data they are able to filter noise to place the landmarks correctly on a map. Depending on the environment, an additional challenge can be present in identifying non static features. With planes, the edges of a plane are dynamic, growing or shrinking as the plane appears or leaves the viewpoint. With points it is possible that several of the point's movements are dynamic and not correlated with the movement of the viewpoint. This dual problem of both noise in the environment and dynamic changes to the landmarks is more difficult than solving either of these problems if they occur separately. The challenge increases further if the requirement is that the SLAM algorithm is to run at real time computational speed. The algorithms developed in this thesis, the Relative Plane algorithm and the Relative Point algorithm are able to solve these dual problem in real time.

1.2 Methodology

The original goal of the thesis was to construct a complete system for autonomous mapping. That is, build the robot hardware, run the AI software on it and have a map being made in real time. This consisted of designing the electronics for the robot, a FPGA design to do the low level vision processing, and a simulator to program the AI software. Individually each component worked but it was not possible to combine them to form a working system.

A previously successful robot project that had the robot achieve its goals, was used as a template for this project. Its scaled up design process was unsuccessful in meeting the goals of this thesis's more difficult problem. This phenomena is known as the "the second system effect" described in software engineering literature [Broo95]. Even though the design was not successful the process is quite interesting so it is described in this thesis.

After reevaluating this project it was decided to focus on the map building problem. There are many different algorithms that have been shown to solve the SLAM problem, depending on the type of input data. Many of these algorithms use some form of cumulative current position as a state variable and only store landmarks in their globally mapped form, discarding past data. This thesis takes a different approach in not using the current position as a cumulative state variable and storing and using all or as much possible past data. Landmarks are mapped relative to each other and the Relative algorithms can use both current and past data for accuracy purposes. Using this approach, the SLAM problem is solved by data structures and algorithms rather than probabilistic modeling.

It is interesting to see the path that led to the derivation of the Relative algorithms. The basic idea of being able to use both current and past data is established early but the relative aspect is not. The first attempt uses something similar to the Iterative Closest Point algorithm after the points have been converted to planes. This algorithm did not work but it was shown that if the current position estimate is accurate, a good map can be created. This led to several more attempts where the focus was to maintain an accurate position. These attempts worked when the plane size is static but in reality a plane grows and shrinks as the plane is first seen and later passed by. Approach after approach failed as they were attempting to treat this growing and shrinking problem as noise, which they were never able to filter out. Each attempt proved to be a learning experience on the limits of different types of algorithms and more importantly of the problem itself. It is in the refinement of the knowledge of the problem that leads to the Relative Plane algorithm. Subsequently, the Relative Point algorithm is developed that uses points as input and has the capability of identifying dynamic point movement. It is noted that individual parts of these algorithm have been seen before, but the algorithms in their entirety are new.

Both versions of the Relative algorithms use the principle of storing untransformed observations, as seen by the robot's viewpoint. The processing is performed exclusively on the untransformed observations rather than transform them into global space. New observations are matched to past untransformed ones and placed into efficient data structures. A single instance of the data structure contains many observations of the same landmark. The landmark represented by all of its untransformed observations is grouped together with other landmarks that are seen on the same observation interval. A relative map of a group can be calculated using the average of the relative locations of the untransformed observations, computed over the observation interval. Before computing the average, it is first required to make the comparison both translation and rotation invariant. In the planar version this is done by making one plane the basis of the average comparison. In the point version this is done by making three points the basis of the average comparison. After a group's relative map is created, a global map can be made by combining many group's relative maps using landmarks that are present in both. Current position can be calculated by comparing the difference of the current untransformed observation location's versus their global position's. Current position is used in a local sense when backtracking and only used globally when closing the loop.

The Relative Plane algorithm is unique in the way it is able to identify which edge of a plane is valid, and its use of a motion model when only one partial plane is visible. It is difficult to directly compare it against other algorithms due to its capabilities. This is why the formal evaluation is done using the Relative Point algorithm.

The Relative Point algorithm uses the more common 3D points as the input. There is a direct comparison to a 6D no odometry version of the Extended Kalman Filter (EKF). The comparison is not to rank which algorithm is better. Rather, the EKF results are used as a baseline to prove that the Relative Point algorithm has comparable accuracy given the same data. Further testing is done to evaluate the error of the Relative Point algorithm as a simulated robot travels the same path many times.

The Relative Point algorithm is thoroughly evaluated to show that its computation complexity is proportional to $O(n_s \log n_s)$ where n_s is the average quantity of landmarks seen in a given observation.

It is also shown that it is effective in removing dynamic landmarks with a low cost in terms of the average computation time of the algorithm. In fact it can be argued that the Relative Point algorithm approaches the minimum computation possible, given that each observed landmark must be registered every iteration.

It is not the intent to rank the Relative algorithm versus others. It is too earlier to exactly quantify how the Relative algorithms directly compares to others especially in accuracy. There is a potential unknown effect on accuracy due to the discretization of only comparing landmarks present in the same groups. This is why much of the evaluation attempts to identify possible sources of landmark error. Rather than rank the Relative algorithm versus others, it can be said that given a problem similar to the ones simulated in this thesis, the Relative algorithm can be an effective solution to the SLAM problem.

1.3 Outline

There are three distinct sections in this thesis: Chapter 2 and Chapter 3 are background chapters, Chapter 4 and Chapter 5 are on the Relative Plane Algorithm, and Chapter 6 is on the Relative Point Algorithm.

The figures in this thesis are best viewed in color. A pdf of this thesis is available online.

Chapter 2: The second system effect discusses the initial system designed to build a robot. 2.2 talks about a previous project and 2.3 uses the past design process on this thesis goals. 2.3.4 discusses the second system effect and 2.4 describes the next design process which is successful.

Chapter 3: The search for the Algorithm describes the implementation and results from several well known algorithms: Iterative Closest Point (ICP), Extended Kalman Filter (EKF) and Expectation Maximization (EM). 3.6 describes several attempts at a SLAM algorithm by trying to get the current position as accurate as possible and treating a plane's growing and shrinking as noise.

Chapter 4: The Relative Plane Algorithm introduces the new Relative Plane algorithm by going through the process step by step used to develop and implement it. The algorithm is stated as a collection of rules that are found in implementing the underlying concept of grouping planes. 4.2 introduces the basic concept of the algorithm and 4.3 discusses how the algorithm is derived. 4.4 discusses software architecture of the algorithm, which is important since having a good architecture made it possible for the algorithm to be built from a basic concept to a full algorithm. 4.5 goes through one full iteration of the algorithm corresponding the the input of the algorithm to the software architecture and stating where each rule is used. 4.6 has closing remarks of the algorithm including noting some related work in 4.6.2.

Chapter 5: EM in combination with the Relative Plane Algorithm shows the results of the algorithm as it is integrated into a simulation that starts with points, forming them into planes using the EM algorithm and then runs the Relative Point algorithm.

Chapter 6: The Relative Point Algorithm adapts the Relative Plane algorithm to using points as its input. 6.1 through 6.5 describe the algorithm. 6.6 examines the performance of the Relative Point algorithm in terms of landmark error and computational complexity. 6.7 compares the accuracy of the Relative Point algorithm to a 6D no odometry version of the EKF. 6.9 takes one further look of the accuracy of the Relative Point algorithm by examining the landmark and position error as the simulated robot loops the same area many times. 6.10 compares the two Relative algorithms, noting that the Relative Point algorithm implements much of the future work of the Relative Plane algorithm.

Chapter 7: Conclusion is the summary of the thesis.

Chapter 2 The second system effect

2.1 Introduction

Sometimes decisions of the past that were correct at the time are reused for future projects to unfortunately poor results. Sometimes following a straight forward path to achieve goals does not work and a new direction has to be chosen. This chapter examines some successful past work of the author that caused some misdirection for the Ph. D. work in terms of the "second system effect" as described by [Broo95].

Why is this chapter part of this thesis? One of the goals of this thesis is to provide information for anyone that wants to work on robotic projects, whether it is building the physical robot, designing a simulation or working on the artificial intelligence (AI) algorithms. Much can be learned by seeing what went right and what went wrong as described by this chapter.

The first section of this chapter is on the author's undergraduate thesis [Krau02]. The thesis consists of the construction of a robot from a radio controlled car chassis, implementation of a 2D map editor and simulator, and development of a wall following AI. The thesis is successful as it met its goals. It showed that it is valid to develop the AI on a simulator first and then port the code over to the robot that has the same application programming interface (API).

Following the success of the undergraduate thesis, a very similar process is used to develop a second robot. As with the "second system effect" many parts of the previous work are improved upon. This time the robot is designed all at once rather than built incrementally, and all the boards are manufactured together. A new 3D map editor and simulator are created to simulate an area for 3D vision AI algorithms. Although the second robot project has parts that work individually, together they do not achieve the project's goals.

The third project is a scaled down version of the second. Some code is reused and there is a much better focus on the actual project goals. The simulation tools of the third project are used for the Relative Plane algorithm developed in this thesis.

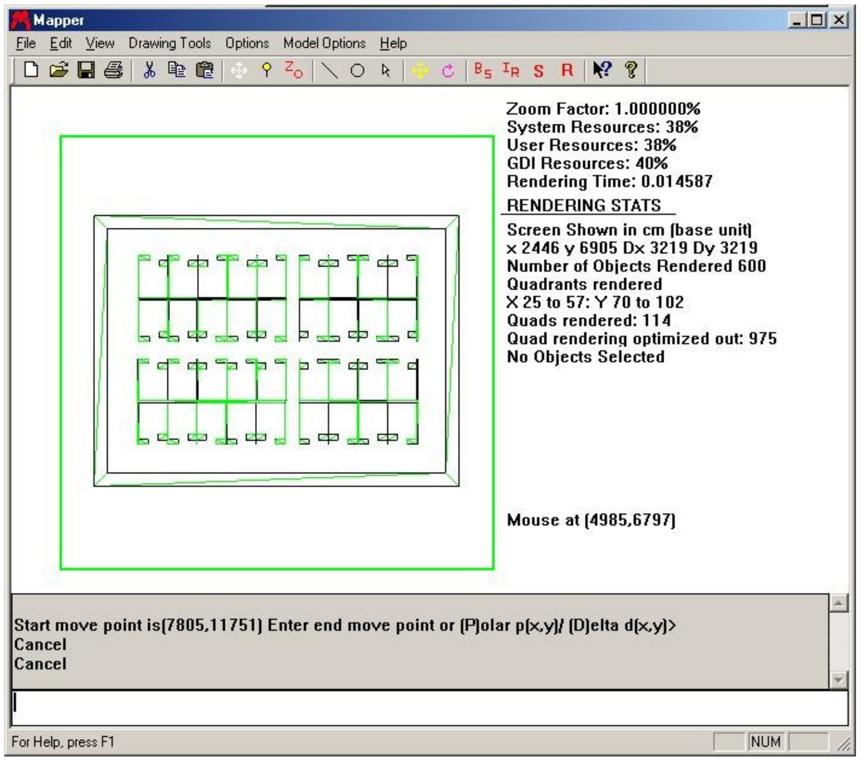
2.2 First Robot Project

When developing the hardware for the undergraduate robot thesis, it is realized that it would be easier to program the AI in a simulator, and then port the code to the actual robot. To compile and load the firmware, then setup a robot can take minutes whereas programming in a simulator can take seconds. Also depending on how good the wireless communication is, it can be difficult to debug an algorithm when the robot is running. First a map editor is created and then it is expanded into a simulator.

2.2.1 Map Editor

The map editor has similar functionality to a two dimensional CAD program. The base object is a line and it supports arcs which are composed of lines. It features.

- create line, arc, snap to end point, grid, zoom in/out.
- group/ungroup undo/redo, save/load print, cut/copy/paste
- for the robot, add bump sensor, ir sensor, sonar sensor, robot starting position
- add depth to line segments mostly for 3D rendering.
- GUI and console base user interface.



```
Zoom Factor: 1.000000%
System Resources: 38%
User Resources: 38%
GDI Resources: 40%
Rendering Time: 0.014587
RENDERING STATS
Screen Shown in cm (base unit)
x 2446 y 6905 Dx 3219 Dy 3219
Number of Objects Rendered 600
Quadrants rendered
X 25 to 57: Y 70 to 102
Quads rendered: 114
Quad rendering optimized out: 975
No Objects Selected

Mouse at (4985,6797)
```

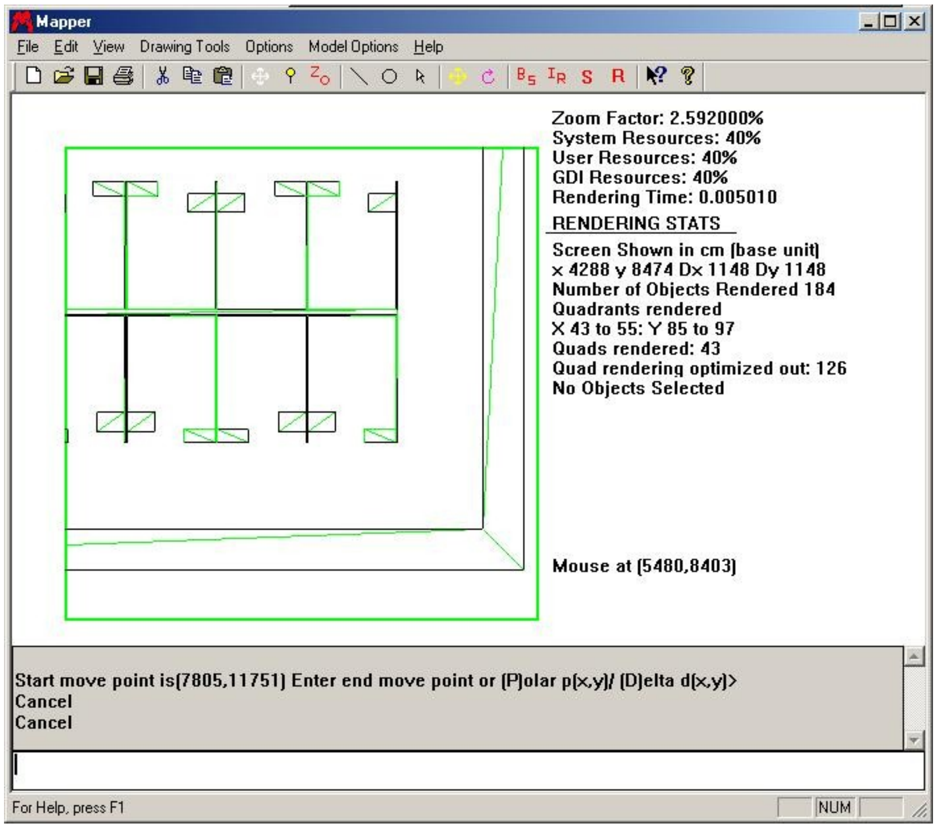
```
Start move point is(7805,11751) Enter end move point or (P)olar p(x,y)/ (D)elta d(x,y)>
Cancel
Cancel
```

Figure 1 Map created in the map editor

Figure 1 shows the map editor with the map that is used for the simulation. The software architecture is very basic with only a few classes.

- CchildView: a class created by the MFC wizard. It does the initial handling of user events.
- Uconsole: in addition to be able to add input using the mouse, it is possible to use text input with the console. This class parses the user input and performs the action. When the user uses the GUI to do an action, the corresponding command is displayed in the console.
- Udraw: this class is a an interface. When the user clicks on a button to draw a line, that event gets routed to this class. It has some variables for state, event handling, and translating screen to logical coordinates.
- UlinearAlgebra: the linear algebra library contains 2D math such as compute the intersection of two lines.
- UquadTree: not only is this class a quad tree which implies it is used to store objects on the map for efficient selection and rendering, it also contains all the information and operations for the

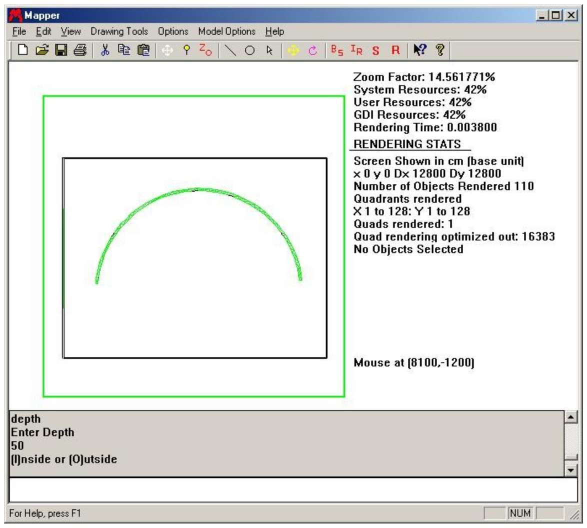
objects themselves! Everything to do with creating an object, moving it, saving it, copy/paste, snap to point, and rendering is performed here. The class is very large as it had the majority of lines of code for the project. One potential improvement to the map editor is that the architecture be redesigned so most of the functionality does not end up in one class.



```
Zoom Factor: 2.592000%
System Resources: 40%
User Resources: 40%
GDI Resources: 40%
Rendering Time: 0.005010
RENDERING STATS
Screen Shown in cm (base unit)
x 4288 y 8474 Dx 1148 Dy 1148
Number of Objects Rendered 184
Quadrants rendered
X 43 to 55: Y 85 to 97
Quads rendered: 43
Quad rendering optimized out: 126
No Objects Selected

Mouse at (5480,8403)
```

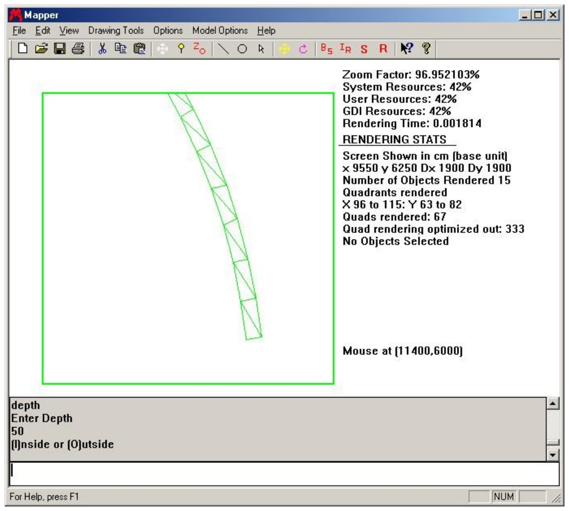
```
Start move point is(7805,11751) Enter end move point or (P)olar p(x,y)/ (D)elta d(x,y)>
Cancel
Cancel
```

Figure 2 Close up of the map used for the simulation



```
Zoom Factor: 14.561771%
System Resources: 42%
User Resources: 42%
GDI Resources: 42%
Rendering Time: 0.003800
RENDERING STATS
Screen Shown in cm (base unit)
x 0 y 0 Dx 12800 Dy 12800
Number of Objects Rendered 110
Quadrants rendered
X 1 to 128: Y 1 to 128
Quads rendered: 1
Quad rendering optimized out: 16383
No Objects Selected

Mouse at (8100,-1200)

depth
Enter Depth
50
(I)nside or (O)utside
```

Figure 3 Making a curve with depth



```
Zoom Factor: 96.952103%
System Resources: 42%
User Resources: 42%
GDI Resources: 42%
Rendering Time: 0.001814
RENDERING STATS
Screen Shown in cm (base unit)
x 9550 y 6250 Dx 1900 Dy 1900
Number of Objects Rendered 15
Quadrants rendered
X 96 to 115: Y 63 to 82
Quads rendered: 67
Quad rendering optimized out: 333
No Objects Selected

Mouse at (11400,6000)

depth
Enter Depth
50
(I)nside or (O)utside
```

Figure 4 Close up of curve with depth

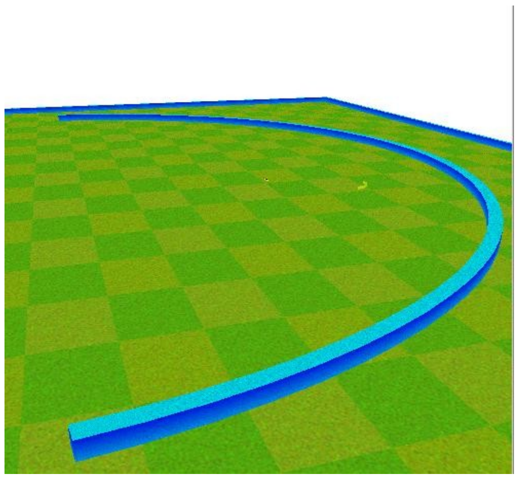
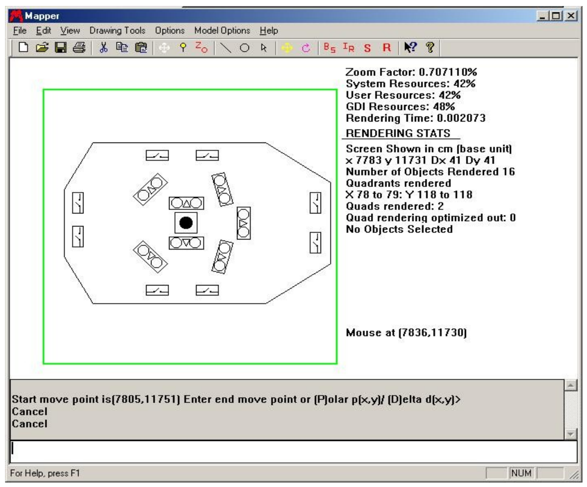
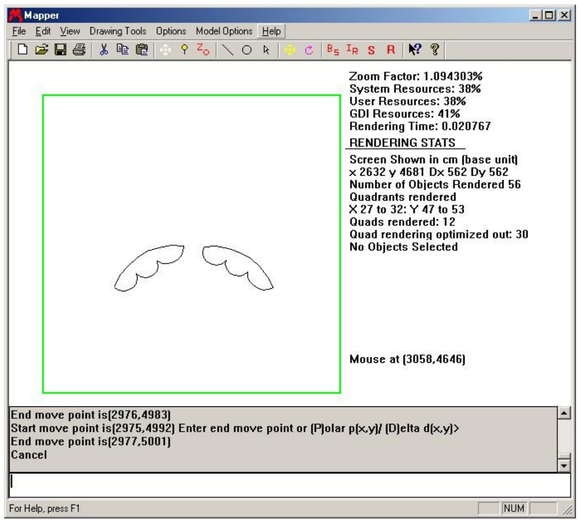


Figure 5 3D rendering



```
Zoom Factor: 0.707110%
System Resources: 42%
User Resources: 42%
GDI Resources: 48%
Rendering Time: 0.002073
RENDERING STATS
Screen Shown in cm (base unit)
x 7783 y 11731 Dx 41 Dy 41
Number of Objects Rendered 16
Quadrants rendered
X 78 to 79: Y 118 to 118
Quads rendered: 2
Quad rendering optimized out: 0
No Objects Selected

Mouse at (7836,11730)

Start move point is(7805,11751) Enter end move point or (P)olar p(x,y)/ (D)elta d(x,y)>
Cancel
Cancel
```

Figure 6 Editing the robot



```
Zoom Factor: 1.094303%
System Resources: 38%
User Resources: 38%
GDI Resources: 41%
Rendering Time: 0.020767
RENDERING STATS
Screen Shown in cm (base unit)
x 2632 y 4681 Dx 562 Dy 562
Number of Objects Rendered 56
Quadrants rendered
X 27 to 32: Y 47 to 53
Quads rendered: 12
Quad rendering optimized out: 30
No Objects Selected

Mouse at (3058,4646)
```

```
End move point is(2976,4983)
Start move point is(2975,4992) Enter end move point or (P)olar p(x,y)/ (D)elta d(x,y)>
End move point is(2977,5001)
Cancel
```

Figure 7 Creating a sonar beam

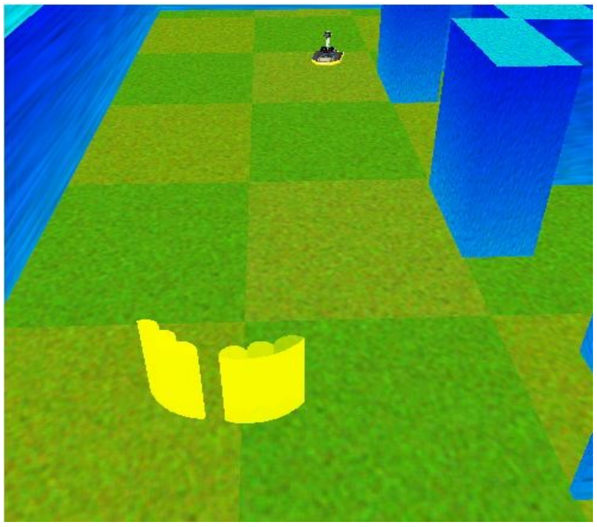


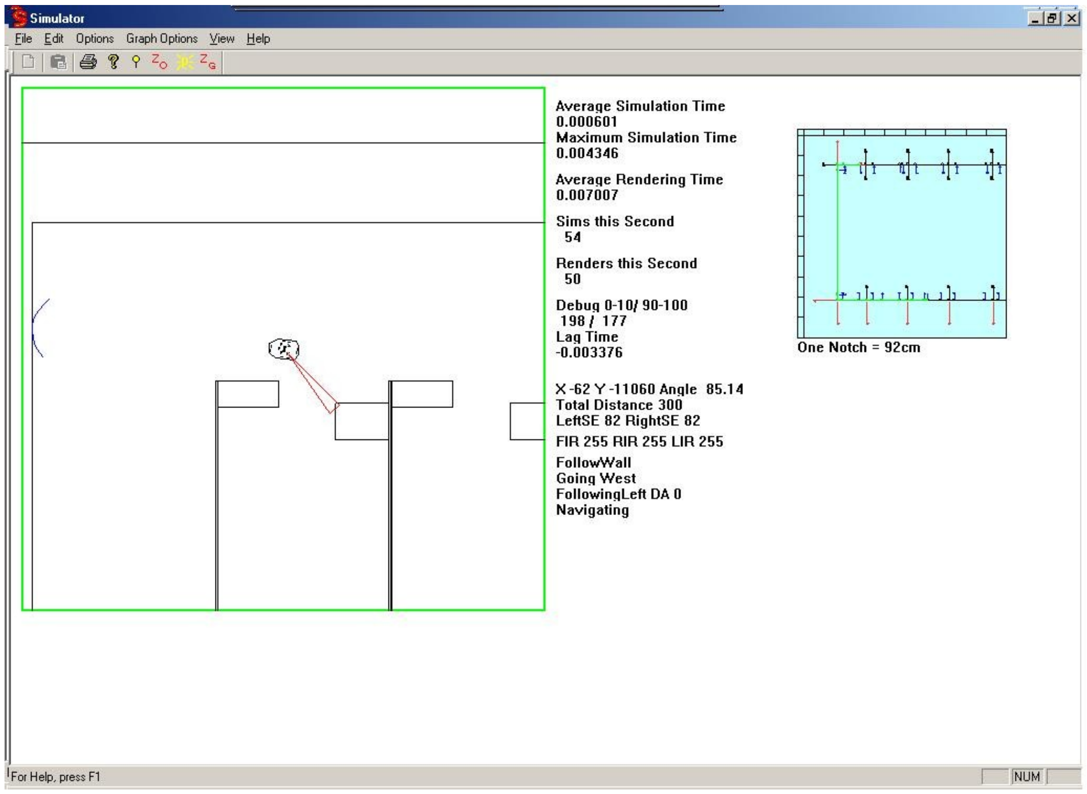
Figure 8 Sonar beam in the simulation

Figure 2, Figure 3, Figure 4, Figure 5, Figure 7, Figure 8, and Figure 6, show some of the various uses of the map editor. In addition to making maps, it can be used to make the shape of the sonar beam and to make the bounds of the robot and place sensors inside of it.

2.2.2 Simulator

The simulator is based on the code from the map editor. Its main class is still the UQuadTree and contains a class for each sensor, SbumpSensor, SIRSensor, SShaftEncoder, SsonarSensor, etc. Most of them are fairly simple but the sonar is more complex as it shoots out line segments to see if there is any contact. It also has a angular firing sequence.

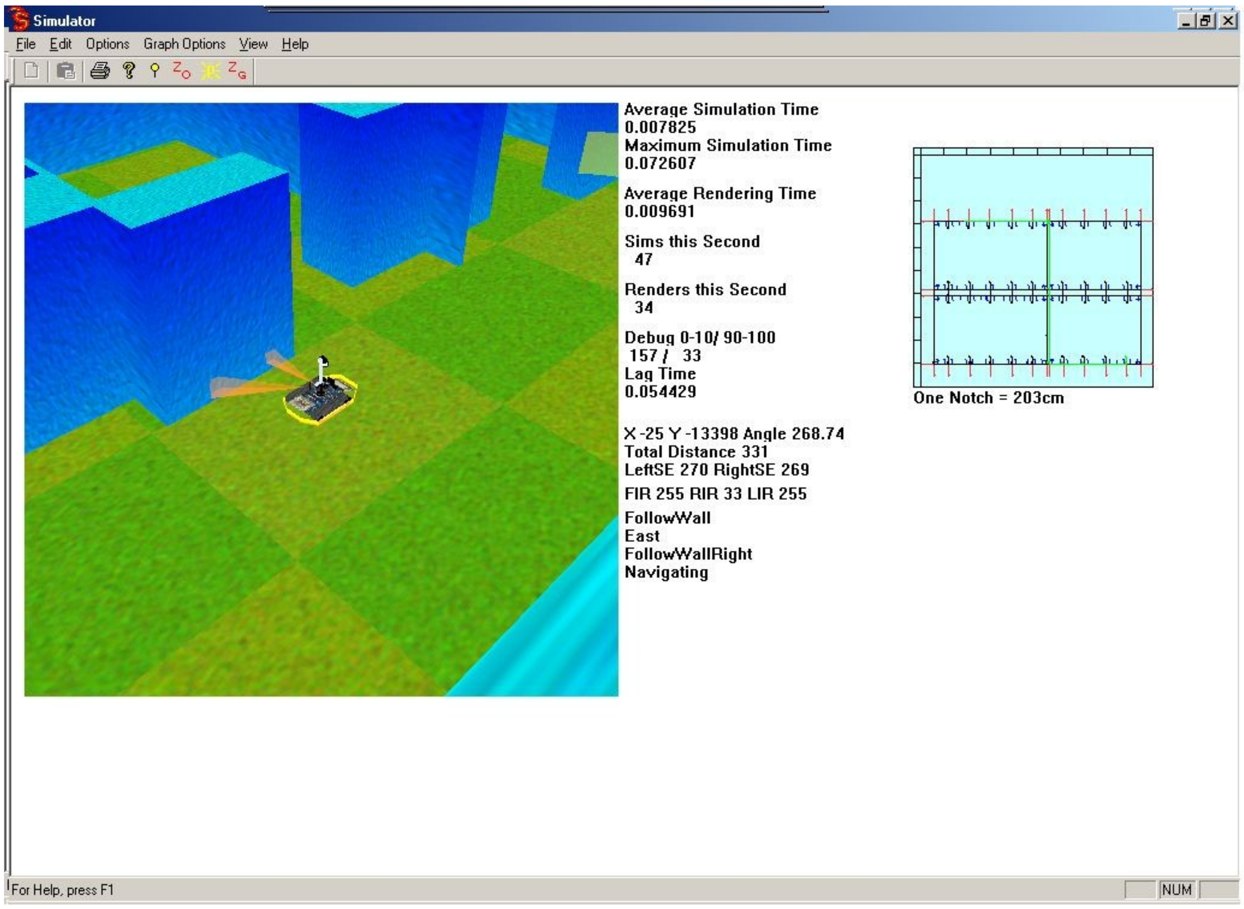
The simulator is in 2D, however it is possible to render in both 2D and 3D. As it turned out 3D is quicker, as the graphics card can rendering polygons quicker than the Windows API can render lines.



```
Average Simulation Time
0.000601
Maximum Simulation Time
0.004346

Average Rendering Time
0.007007

Sims this Second
 54

Renders this Second
 50

Debug 0-10/ 90-100
 198 /  177
Lag Time
-0.003376

X -62 Y -11060 Angle  85.14
Total Distance 300
LeftSE 82 RightSE 82
FIR 255 RIR 255 LIR 255
FollowWall
Going West
FollowingLeft DA 0
Navigating
```

One Notch = 92cm

Figure 9 Showing the simulation in 2D mode



```
Average Simulation Time
0.007825
Maximum Simulation Time
0.072607

Average Rendering Time
0.009691

Sims this Second
  47

Renders this Second
  34

Debug 0-10/ 90-100
 157 /  33
Lag Time
0.054429

X -25 Y -13398 Angle 268.74
Total Distance 331
LeftSE 270 RightSE 269
FIR 255 RIR 33 LIR 255
FollowWall
East
FollowWallRight
Navigating
```

Figure 10 Full simulation, right most window shows the map the AI is generating

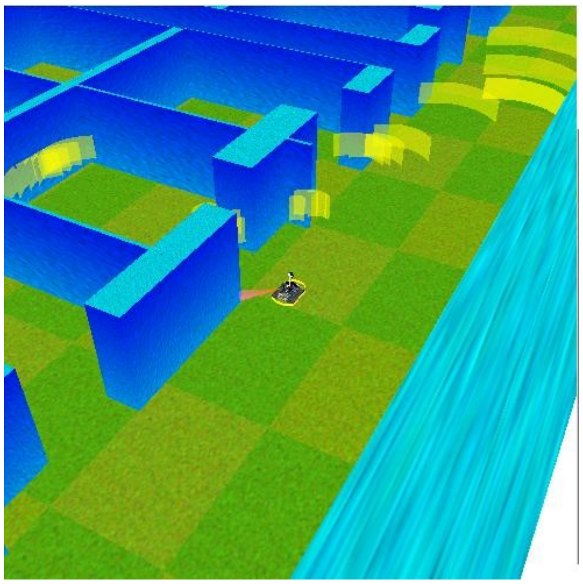


Figure 11 Sonar in a hallway

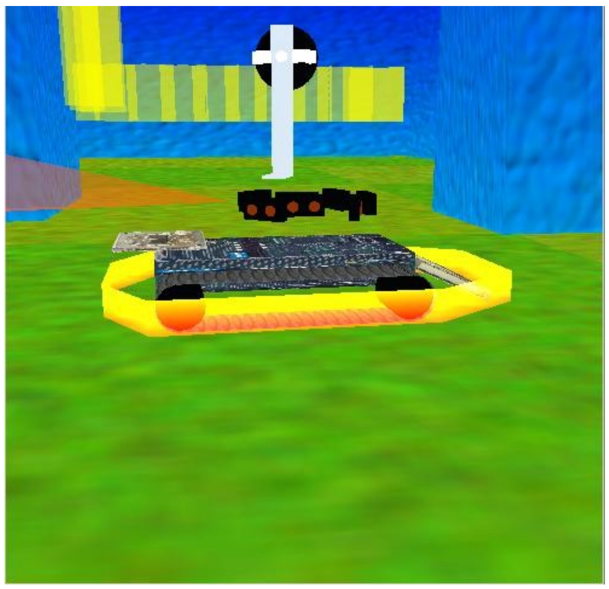


Figure 12 Sonar in a room

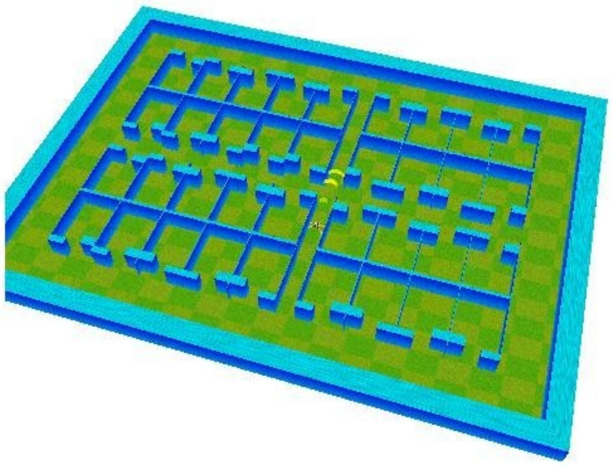


Figure 13 Total map

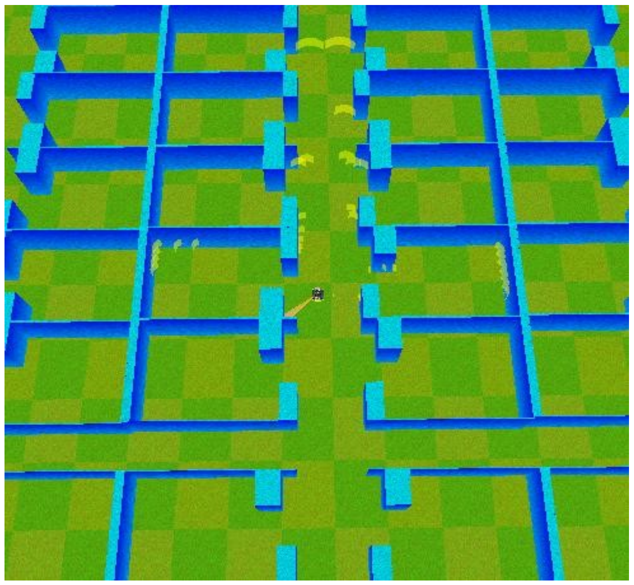


Figure 14 Sonar in a hallway

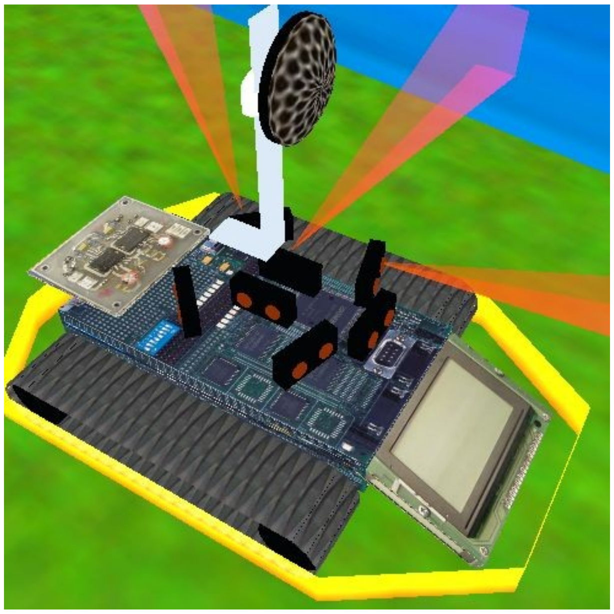


Figure 15 IR closeup

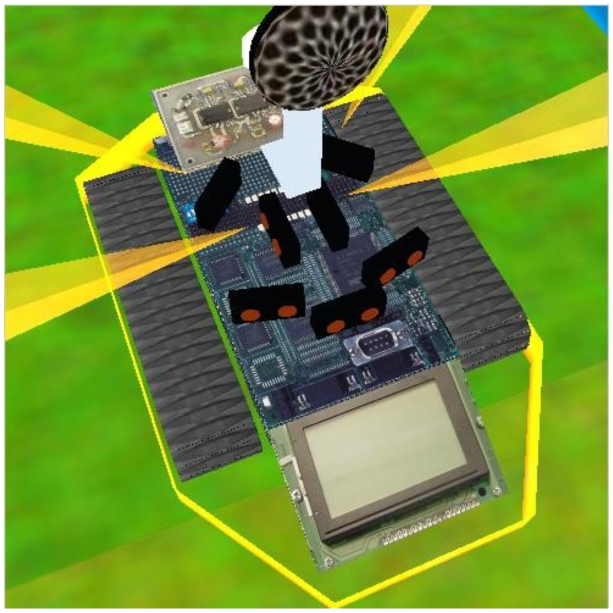


Figure 16 IR closeup

Figure 9 shows the simulation in two dimension (2D) mode and Figure 10 shows the simulation in three dimension (3D) mode. It is possible to swap modes instantly. In those two figures, the upper right portion of the window shows the map being made and it is possible to select nodes in the map for navigation. Figure 11, Figure 12, Figure 13, Figure 14, and Figure 15 show various screen shots of the simulation.

The first simulation worked really well. It simulates the robot accurately enough so that the AI code works in a live setting. One of the key features is that the simulator has the same API as is used in the robot's embedded system. This allowed the AI code to be ported within minutes. The AI code is fairly large and would have been very difficult to debug in a live setting.

2.2.3 Robot

The first robot is based on the rug warrior seen in Figure 18. It is powered by a 68HC11 processor and a custom PCB and chassis. The stock version has a limited amount of sensors. It comes with a dual IR emitter, a single IR detector, shaft encoders for the wheels and three bump switches. Figure 17 Shows the iRobot B21 which comes with 48 sonar sensors 24 IR sensors, 56 touch sensors and far better shaft encoders. The goal is to build a robot better than the rug warrior and to try approach the capability of the iRobot B21.

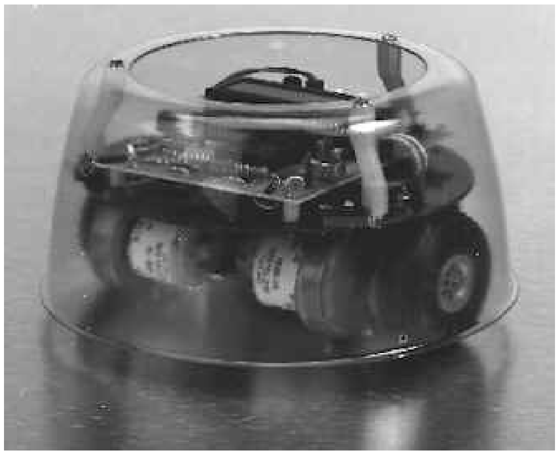


Figure 18 Rug Warrior picture from

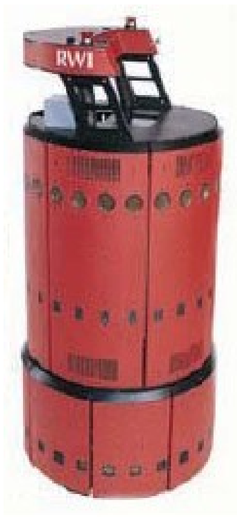


Figure 17 iRobot B21

The solution is to use a Radio Shack bedlam RC chassis seen in Figure 19, a more powerful micro controller, the Siemens 167, some custom electronics seen in Figure 20, a sonar that could rotate, five IR sensors, an optical mouse, and 8 bump sensors.

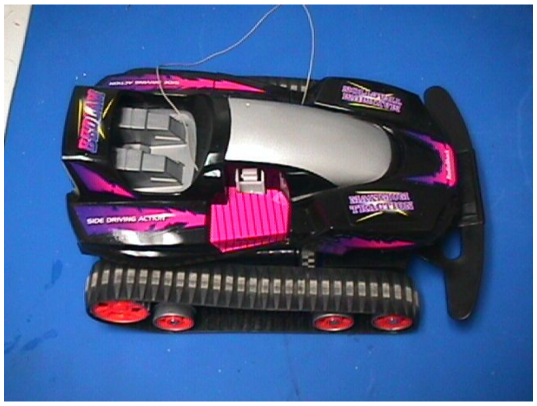


Figure 19 Bedlam chassis

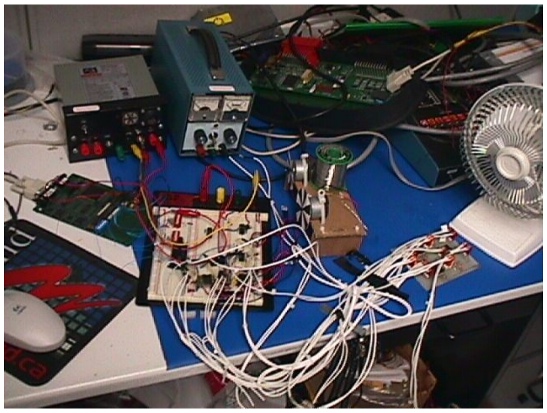


Figure 20 The custom electronics

After gutting the Bedlam chassis, the first step is to put in the custom motor board, the custom power board, and the optical mouse seen in Figure 21 Base assembly. The next step is to put in the LCD, the micro controller board and then IR sensors seen in Figure 22. Each sensor requires the addition of interfacing electronics. In between the micro controller board and the IR sensors is the LCD controller board and the optical mouse interface board.

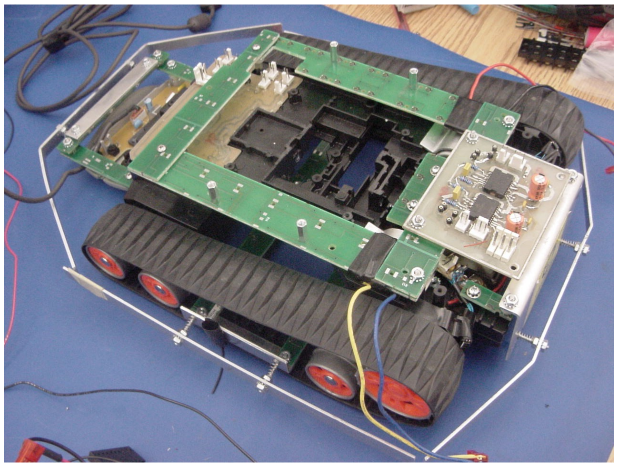


Figure 21 Base assembly

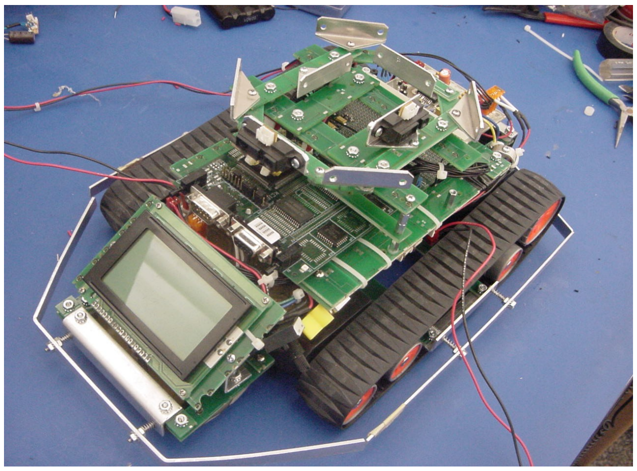


Figure 22 With IR assembly

The sonar and sonar interface boards are placed on top of the IR sensors. The sonar's electronics Figure 23 has four boards: the sonar kit board, the stepper driver board, a board which has chips to interface with the kit and an IR emitter detector to zero the stepper motor, and a board to change the routing of the wiring. There are also electronics hidden inside the robot such as the shaft encoder electronics and some circuitry for the batteries and a power switch. At this point the robot is getting quite tall and congested.

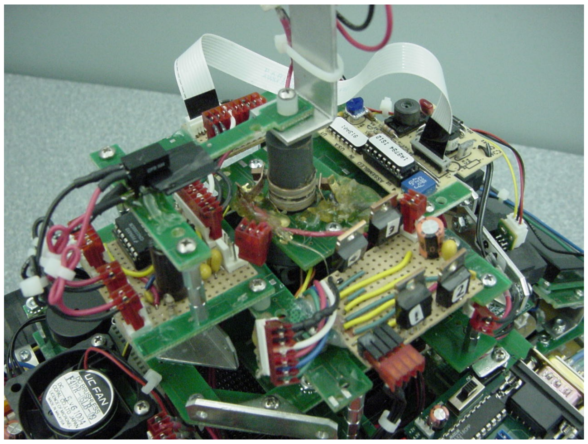


Figure 23 With sonar assembly

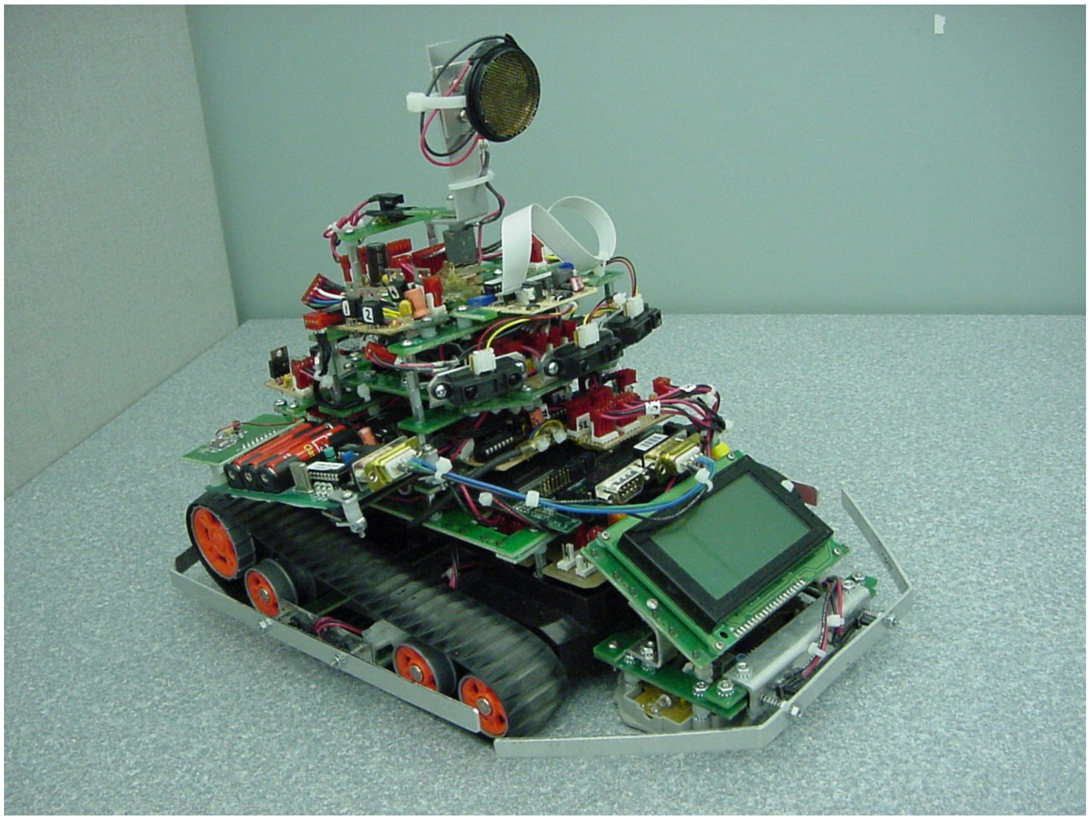


Figure 24 First robot

The end result in Figure 24 may not look pretty but the robot worked. Much space is wasted due to wiring and electronics which could be on the same board but are not. If the robot is designed all at once the design might be more compact and efficient.

2.2.4 Robot AI

The robot is limited by the 80 cm infrared detector range and the time it takes for one long range sonar sweep. The sonar sweep is too slow for continuous movement since there is only one sonar that is rotated and it takes over 1 second for a full scan. This led to the choice of using wall following as the means to localize the robot.

When map making, the robot follows the wall within the infrared detector range and looks for any discontinuities. If it finds one it charts it on the map. If it finds an empty space the robot is instructed to stop and perform a full sonar sweep. The sonar routine is calibrated to know if it is detecting a room or an open hallway. If the opening is a room, it goes past the room and keeps on following the wall. If the opening is a hallway the robot always turns right. Always turning right allows the loop to be closed using the shortest distance. After closing the loop, the robot would then go to new area to generate a map.

A generated topological map is shown in Figure 25. Circles with arrows represent discontinuities. The arrows direction shows if the detection is closer or further away than the previous wall. The size of the arrows indicate the length difference. Squares with a circle indicate locations where a sonar sweep is taken and squares with a triangle indicate a room. The landmarks are placed on the map using distance given by the shaft encoders which is very accurate going straight. The shaft encoders are unreliable during turns due to the slippage caused by the tracks having to be nearly at full power in order to turn. This necessitates the need to assume all turns are $90^{0.}$ After a turn the robot is nearly never 90^{0} from the previous location so it needs to reorient using the next wall.

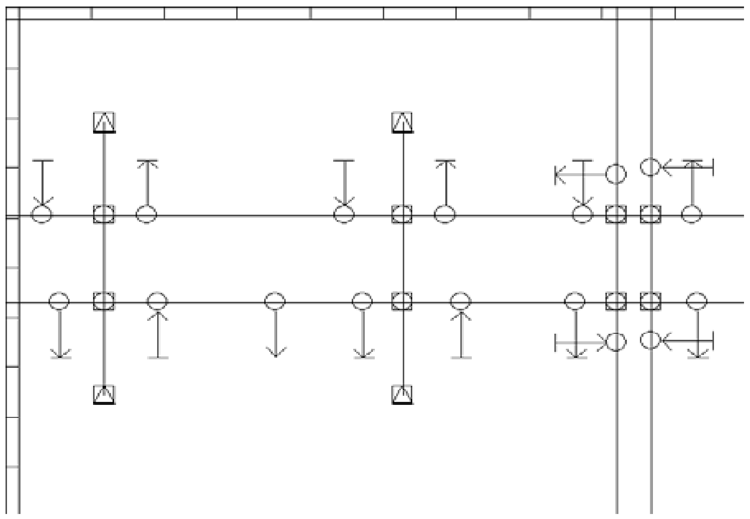


Figure 25: Topological map

When navigating, the user selects the current node and the destination node and then instructs the robot to go on that path. The robot uses a shortest path algorithm to find the route. Each landmark on the route is stored on the stack. When the robot detects a wall discontinuity, it references the stack to see if it has arrived at the next landmark. If it does not match, the stack is searched for a matching landmark. If a matching one is found that become the current position. This can happen because sometime a landmark can be close to the size threshold and not appear every time. Locations in between landmarks are charted uses the shaft encoder distance. When the stack is empty, the current node is the goal node and the navigation is complete. Navigating is quicker than map making since when navigating the sonar is not required.

2.2.5 Results

Overall, the project was a success. First the AI is tested in the simulation map shown in Figure 1. The result is shown in Figure 26. The AI is then ported to the robot in a few minutes.

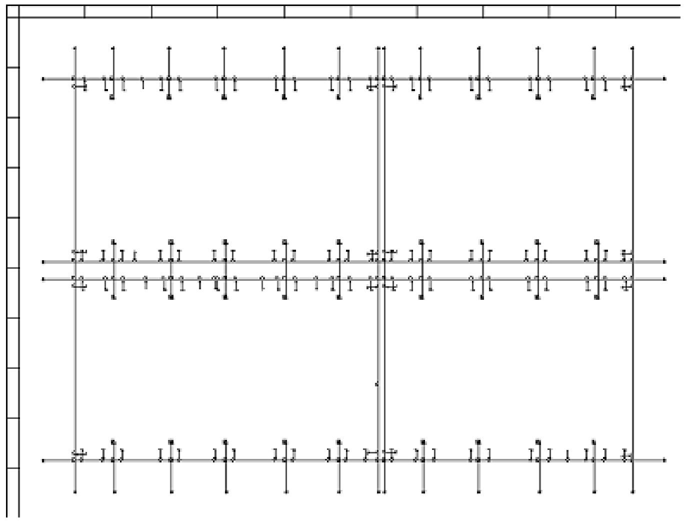


Figure 26 Simulation results

The first experiment was to see if the robot could map a section shown in Figure 27 and the results in Figure 28. The section in Figure 27 differs from Figure 26 due to some ongoing construction in the office that doubled the size of one cubicle.

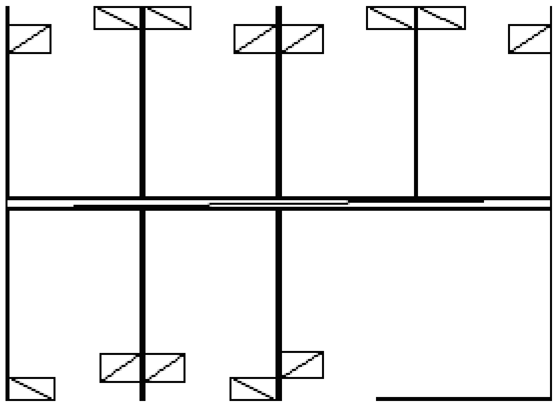


Figure 27 Section to map

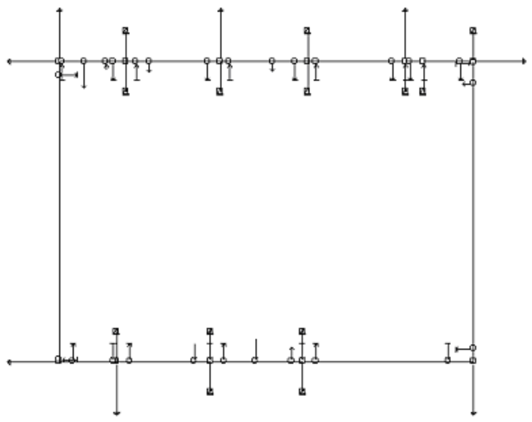


Figure 28 Map results

The results in Figure 28 are quite good. There were some erroneous readings on the map as can be seen by comparing the landmarks (the lines with arrows and perpendicular lines), to the simulated results in Figure 26. This is due to the IR sensors giving incorrect readings. The IR sensors were an issue as the detection threshold was raised and black filing cabinets were covered up with white paper as the IR sensors could not detect them. The robot was able to successfully create a map and close the loop.

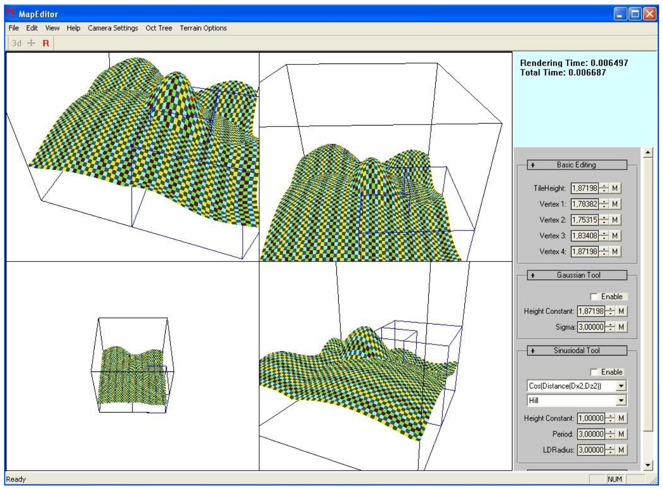
Next the navigation was tested, and the robot was able to go from one cubicle on the map to another across the office. When map making, the robot was able to complete the map about half the time. When navigating the robot was able to arrive at the destination nearly every time. When map making the robot would have to take a sonar reading when encountering any opening and the realignment to the wall would not always work, especially depending on the stagger of the cubicle walls. When navigating the robot does not have to stop and take a sonar reading as it is able to reference the map as to what the opening is. There is video of both the mapping and navigation.

The conclusion of this project was not only the success of the robot, but also the verification of the usefulness of the simulation. After about a month of developing the AI, the complete process of porting the AI, loading it into the robot, testing it, making a few changes to the sensors thresholds, and then taking the video took less than one evening.

2.3 Second Project

2.3.1 Map Editor

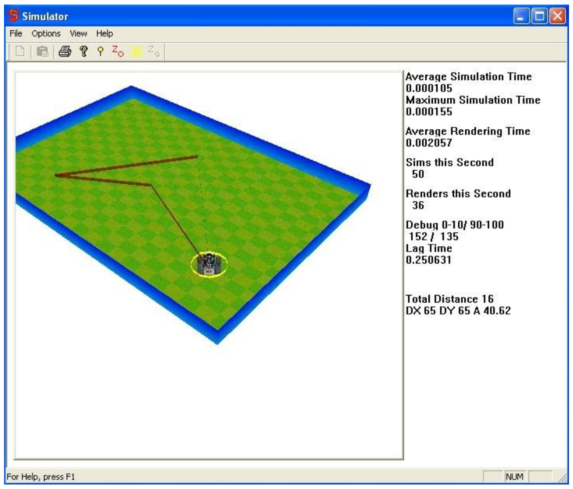
After the success of the first simulation there was a plan to create a second simulation. The second simulation is planned to be in full 3D and to be used to simulate navigation using vision. It took several research projects before the 3D simulation was created.



```
Rendering Time: 0.006497
Total Time: 0.006687
```

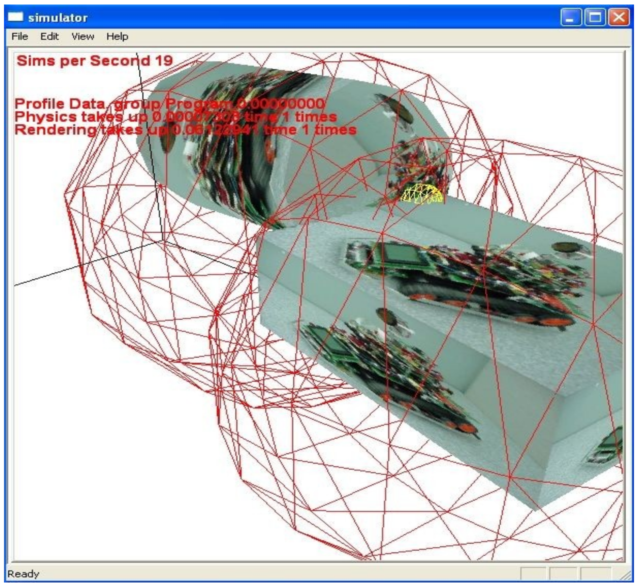
```
TileHeight:      1,87198  M
Vertex 1:        1,78382  M
Vertex 2:        1,75315  M
Vertex 3:        1,83408  M
Vertex 4:        1,87198  M
Height Constant: 1,87198  M
Sigma:           3,00000  M
Cos(Distance(Dx2,Dz2))
Hill
Height Constant: 1,00000  M
Period:          3,00000  M
LDRadius:        3,00000  M
```

Figure 29 Terrain map builder



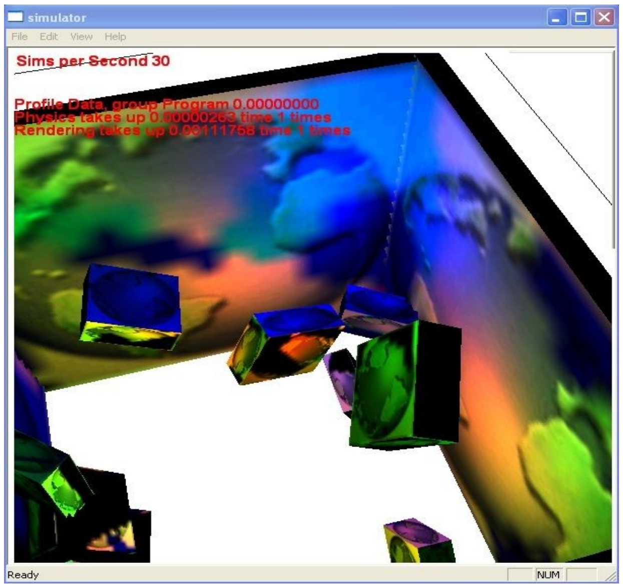
```
Average Simulation Time
0.000105
Maximum Simulation Time
0.000155

Average Rendering Time
0.002057

Sims this Second
 50

Renders this Second
 36

Debug 0-10/ 90-100
 152 /  135
Lag Time
0.250631


Total Distance 16
DX 65 DY 65 A 40.62
```

Figure 30 Line follower



```
Sims per Second 19

Profile Data, group Program 0.00000000
Physics takes up 0.00007308 time 1 times
Rendering takes up 0.06122941 time 1 times
```

Figure 31 Polygon collision physics



```
Sims per Second 30

Profile Data, group Program 0.00000000
Physics takes up 0.00000263 time 1 times
Rendering takes up 0.00111758 time 1 times
```

Figure 32 Light maps

Figure 29 shows a terrain editor that uses mathematical functions to create a height map for terrain. The terrain editor is not reused, but the octree spatial subdivision structure and the roll out controls are. Figure 30 contains the path following algorithm using the original simulator. That algorithm is reused for the second simulation. Figure 31 contains a physics demonstration of polygon to polygon collision detection and basic Newtonian physics to deal with the collision. Figure 32 contains the light maps that are used for the second simulation.

After a few projects it was decided to use binary space partitions (BSP) with light maps. BSP is a way to partition space into leaves using a splitting plane. Figure 33 shows the process. With a BSP, it is possible to do O(logn) collision detection and be able to draw a map without having to use a z buffer as the planes would be correctly sorted back to front.

Rather than use the BSP itself to do the rendering, there is a more efficient way using the results of the BSP. After the space is partitioned it is possible to generate a potential visible set (PVS). Given a location, the PVS is used to create a list of polygons that are potentially visible. When rendering, some addition frustum culling is performed to see if any of the PVS are visible and if so it renders them. There is some overdraw, but it greatly reduces the amount of polygons that have to be rendered.

The interesting thing about a BSP is that to create an optimum tree would take a lot of processing, as it is difficult to know what the best splitting plane is. BSP algorithms try to select a good splitting plane but not necessarily the best one. As long as the tree is mostly balanced the BSP should be efficient.

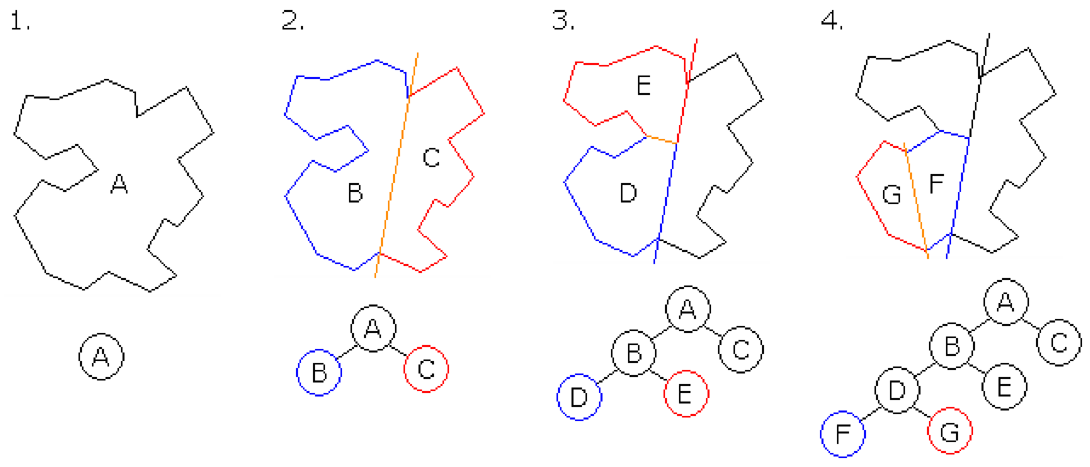


Figure 33 Shows how a BSP is created, figure from wikipedia [Wiki09a].

To create a BSP, there is a rule that there cannot be any "holes" in the map otherwise it is not possible to compile a map. To avoid this situation, constructive solid geometry (CSG) is used. Anything that is going to be used for the BSP is created out of geometry with addition and subtraction operations. To create a room, one large square may first be added to the map, then a slightly smaller square inside of it is subtracted from the large square.

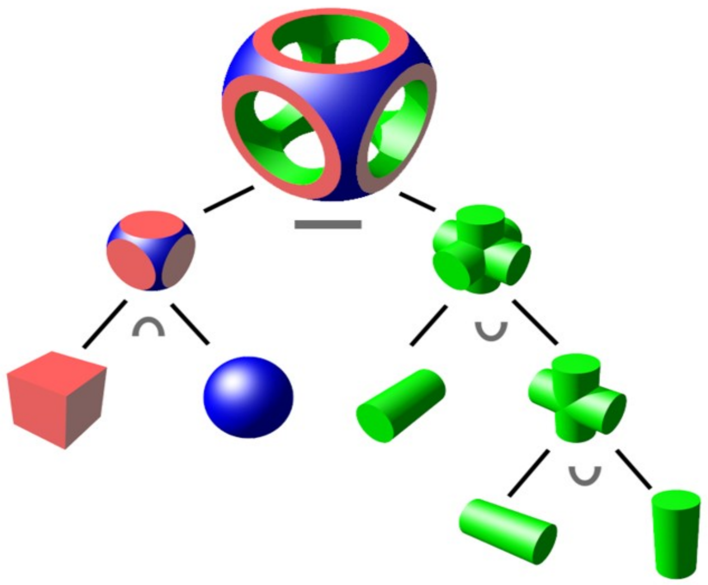
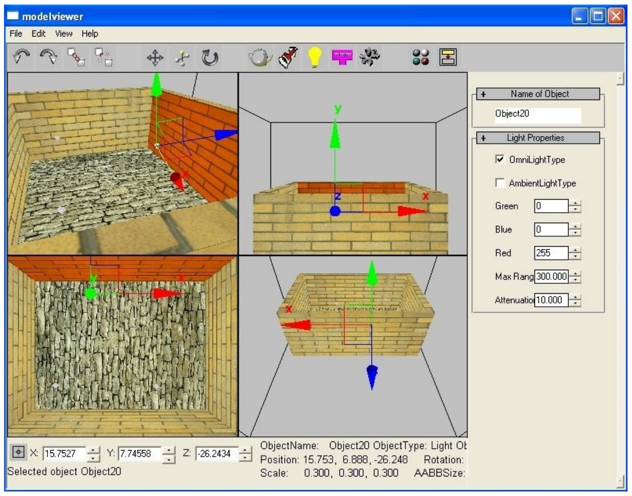


Figure 34 CSG demonstration, figure from wikipedia [Wiki09b]

Using the BSP as the basis of the map editor, a design process similar to the first project is followed. First a map editor is created, and then a simulator is made out of it. The original plan is for the map editor and simulator to use the same rendering technology as the current state of the art computer games but increase the realism by using high resolution textures taken from the area being simulated. The hope is that this would be photo realistic.

Figure 35 shows a room being created by cutting a large box with a slightly smaller one. Figure 36 shows that room after lighting is applied. Figure 37 shows multiple rooms and Figure 38 shows the same rooms lighted.



Object20

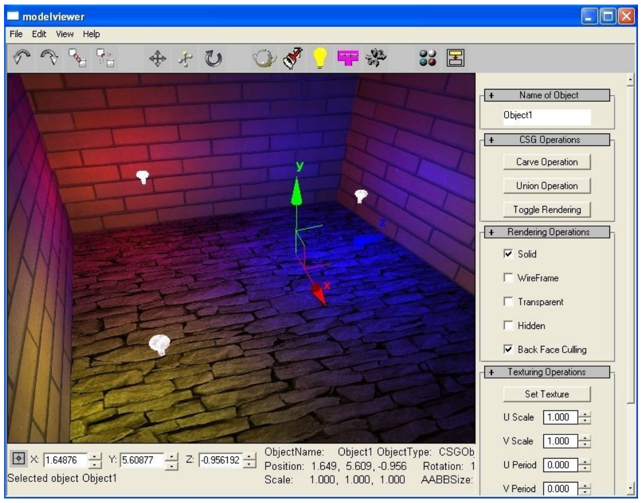
☑ OmniLightType
☐ AmbientLightType

Green 0
Blue 0
Red 255
Max Rang 300.000
Attenuatio 10.000

X: 15.7527 Y: 7.74558 Z: -26.2434
Selected object Object20

```
ObjectName:   Object20 ObjectType: Light Ob
Position: 15.753,  6.888, -26.248     Rotation:
Scale:     0.300,  0.300,  0.300     AABBSize:
```

Figure 35 Creating a room out of a box



Object1

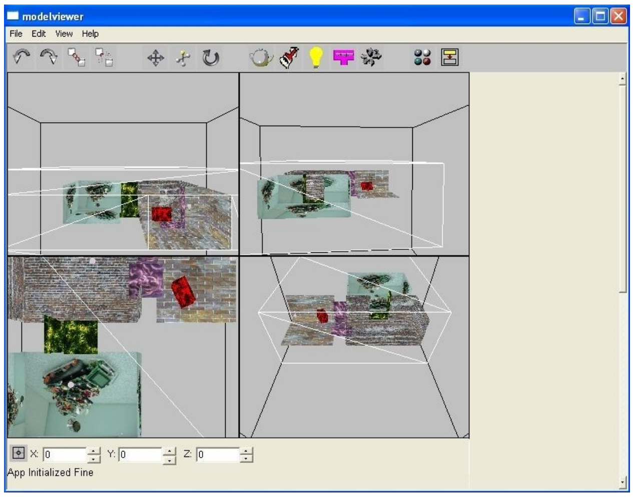
☑ Solid
☐ WireFrame
☐ Transparent
☐ Hidden
☑ Back Face Culling

U Scale 1.000
V Scale 1.000
U Period 0.000
V Period 0.000

X: 1.64876 Y: 5.60877 Z: -0.956192
Selected object Object1

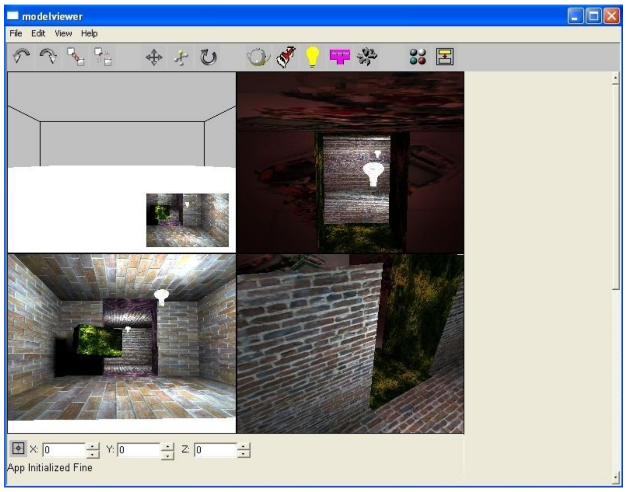
```
ObjectName:    Object1 ObjectType:  CSGOb
Position:  1.649,  5.609, -0.956     Rotation:  1
Scale:     1.000,  1.000,  1.000     AABBSize:
```

Figure 36 Lighting the room



X: 0 Y: 0 Z: 0
App Initialized Fine

Figure 37 Creating multiple rooms



X: 0 Y: 0 Z: 0
App Initialized Fine

Figure 38 Lighting multiple rooms

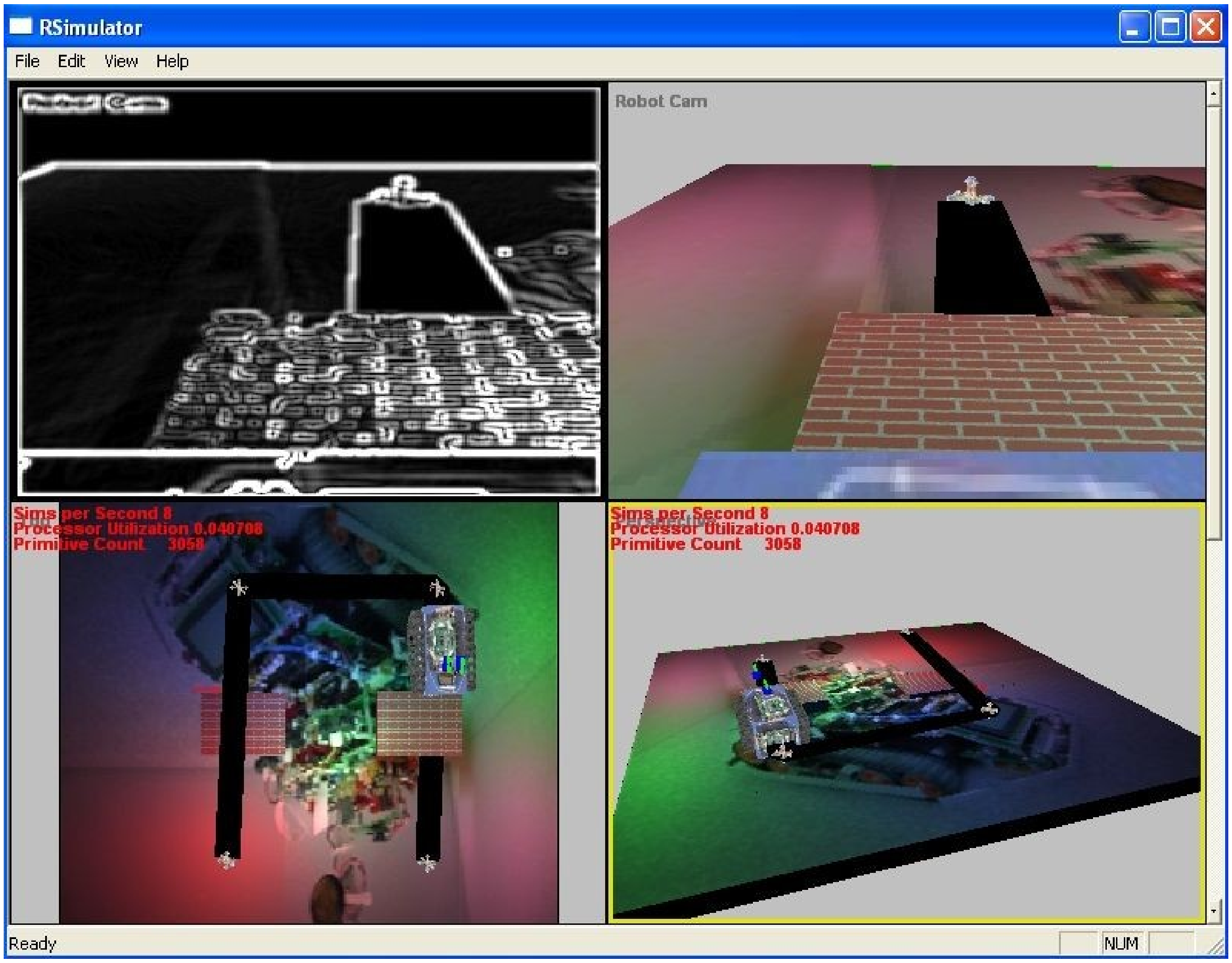


Figure 39 Using edge detection in the simulated environment

The architecture of this map editor is a large improvement from the previous version. The first map editor is loosely based on AutoCAD and this one is based on 3D studio max which has a plug in architecture. This map editor does not have a plug in architecture but it has a similar design philosophy. Each object has its own class inheriting from a base class. Each control has its own class inheriting from a base class. The user interface is implemented to interface with the base classes, so all the objects have access to create custom controls when they are selected. This makes it very simple to add new objects and controls to the map editor.

2.3.2 Result

The second map editor is a large improvement over the first one. The simulation borrows much of the code from the map editor and includes an addition of a robot and a path following algorithm. Figure 39 shows the simulation with an edge detection algorithm.

2.3.3 Second version of the robot

The plan was to design the second robot all at once rather than build it incrementally. All the electronics are to be manufactured together as shown in Figure 42. The boards consisted of motor drive electronics, a power supply board, a analog/digital interface board, and a speaker board. All of the boards are more powerful and/or more feature laden than the first robot. The electronics contains the same Siemens 167 board that is connected to a Via EPIA x86 1.3 GHz board. The Via board has a wireless Ethernet connection instead of the slow serial modem of the previous robot. The Via board is powerful enough to stream motion JPEG video from a web camera.

The robot is designed in Pro/E shown in Figure 40 and Figure 41, rather than being designed incrementally as it is built. The plan is for the robot to work both for autonomous indoor navigation and to be operated remotely for robocup rescue. Because of this requirement, it is developed to be rugged, featuring suspension and a powerful drive train from a Traxxax E-Maxx (Figure 43). An E-Maxx is gutted for the motors, transmission and suspension components. These are measured and digitized into Pro/E. A chassis is designed to hold the motors, batteries, electronic and feature a tracked suspension system.

Similar to the last project when the AI and the robot are built at the same time, the computer vision algorithm is developed at the same time as the robot is being built. Since the robot would require real time processing, a FPGA board is used to run edge detection on the output of a web camera [Krau06]. The FPGA is able to process 437 bitmaps a second which is much faster than unoptimized edge detection code can run on a laptop. Unfortunately the Ethernet used to transfer the image can only handle a few bitmaps a second.

The end result of the robots design is shown in Figure 44 and the FPGA design in Figure 45. The boards are assembled and tested. Figure 44 shows a computer connected to the Via board. A game pad on the computer is able to control the motors and by using a web cam with a built in servo, it is able to aim the camera. Unfortunately the chassis shown in Figure 41 was not built.

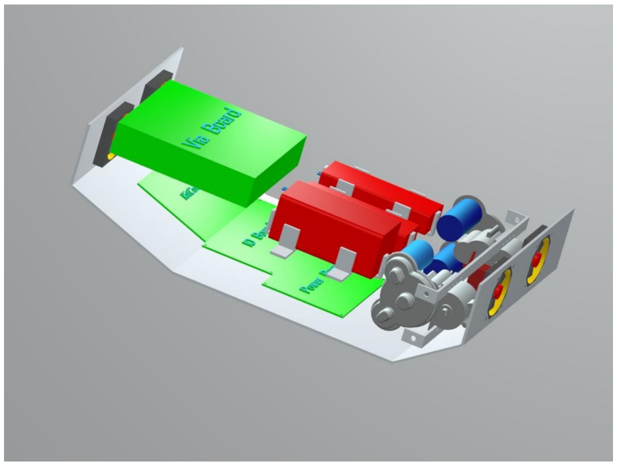


Figure 40 Pro/E rendering

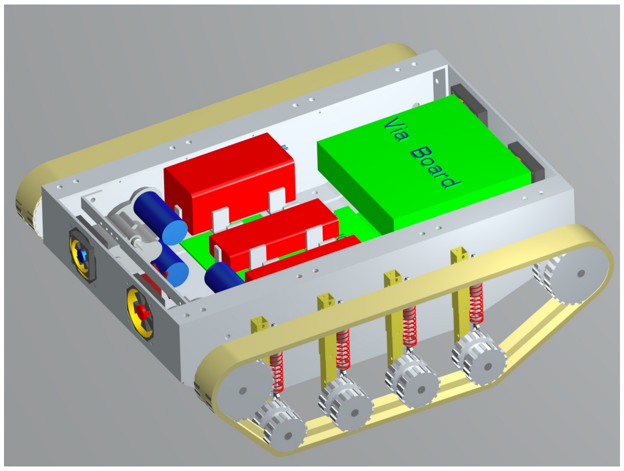


Figure 41 Pro/E rendering

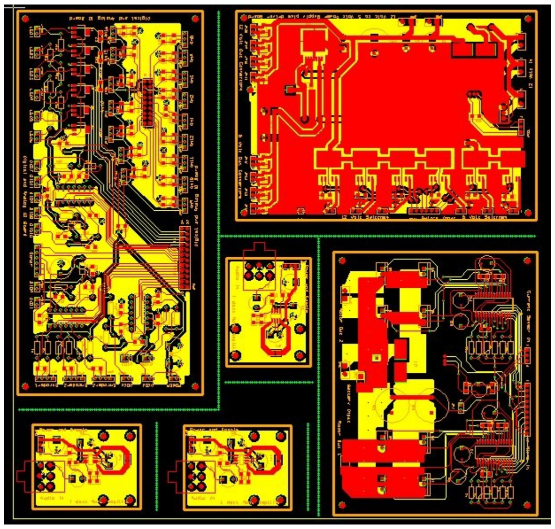


Figure 42 PCB layout

Figure 43 Traxxax E-Maxx

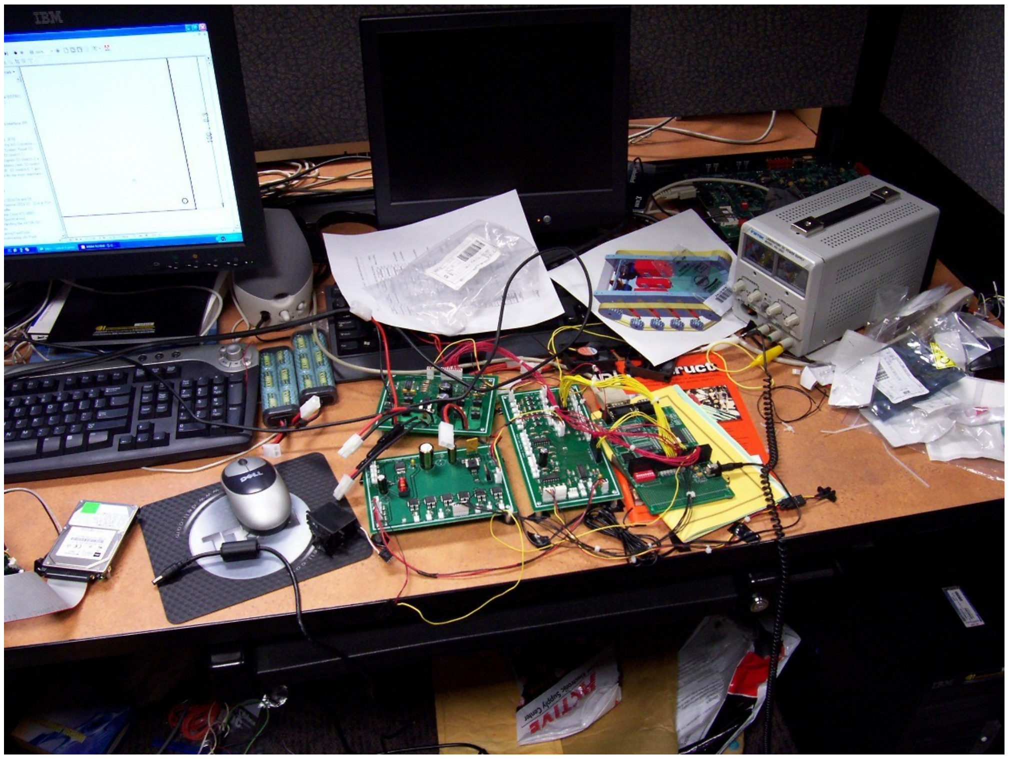


Figure 44 Hardware testing

Figure 45 FPGA edge detector development

2.3.4 Results of the second project, the second system effect

[Broo95] states: "As he designs the first work, frill after frill and embellishment after embellishment occur to him. These get stored away to be used "next time." Sooner or later the first system is finished, and the architect, with firm confidence and a demonstrated mastery of that class of systems, is ready to build a second system. <u>The second is the most dangerous system a man ever designs</u>. When he does his third and later ones, his prior experiences will confirm each other as to the general characteristics of such systems, and their differences will identify those parts of his experience that are particular and not generalizable." and "How does the project manager avoid the second-system effect? By insisting on a senior architect who has at least two systems under his belt."

It is clear to see how the second system effect doomed the second project. Rather than look at the goals of the second project, the intent was to create a better version of the first project. For the second robot, is all the extra capacity on the electronic boards really necessary? Is it necessary to have a tracked system with suspension? Is it a wise idea to try and make a photo realistic 3D simulator where game engines have large teams of developers and they themselves are not photo realistic?

After implementing the 3D simulation it was realized that much of the time spent on creating a game is not on the core engine itself. Looking through the credits on any modern game, more of the personel are devoted to the level design rather than the core engine. It would have been wiser to use a preexisting game engine as the simulator. When designing the electronics, it might have been more cost and time effective to use off the shelf parts. When designing the robot, it should have been known that trying to design for two goals, indoor autonomous navigation and robot rescue would doom the robot to do neither well.

Given that the main goal is to achieve 3D vision, how can the project be better managed to support this goal? Instead of trying to implement a photo realistic simulation why not use a game engine that has free education licensing [Unre09]? Or better yet why not record video and use that as input? Using recorded video forces the simulation on the given path, but for vision purposes it would suffice. Instead of designing a complicated robot why not just use a simple one with a laptop?

After reevaluating the work completed so far it was decided that it might not be realistic to be able to complete a robot that can autonomously map in real time. Instead, this thesis is refocused on the SLAM problem. A new simulator is created out of the code from the second simulator to simulate the output of vision processing, points and lines.

2.4 Third Simulation

After the thesis is refocused on a SLAM algorithm it was decided to use the code of the second map editor/simulator to develop a simpler simulation to simulate the output of point and line detector.

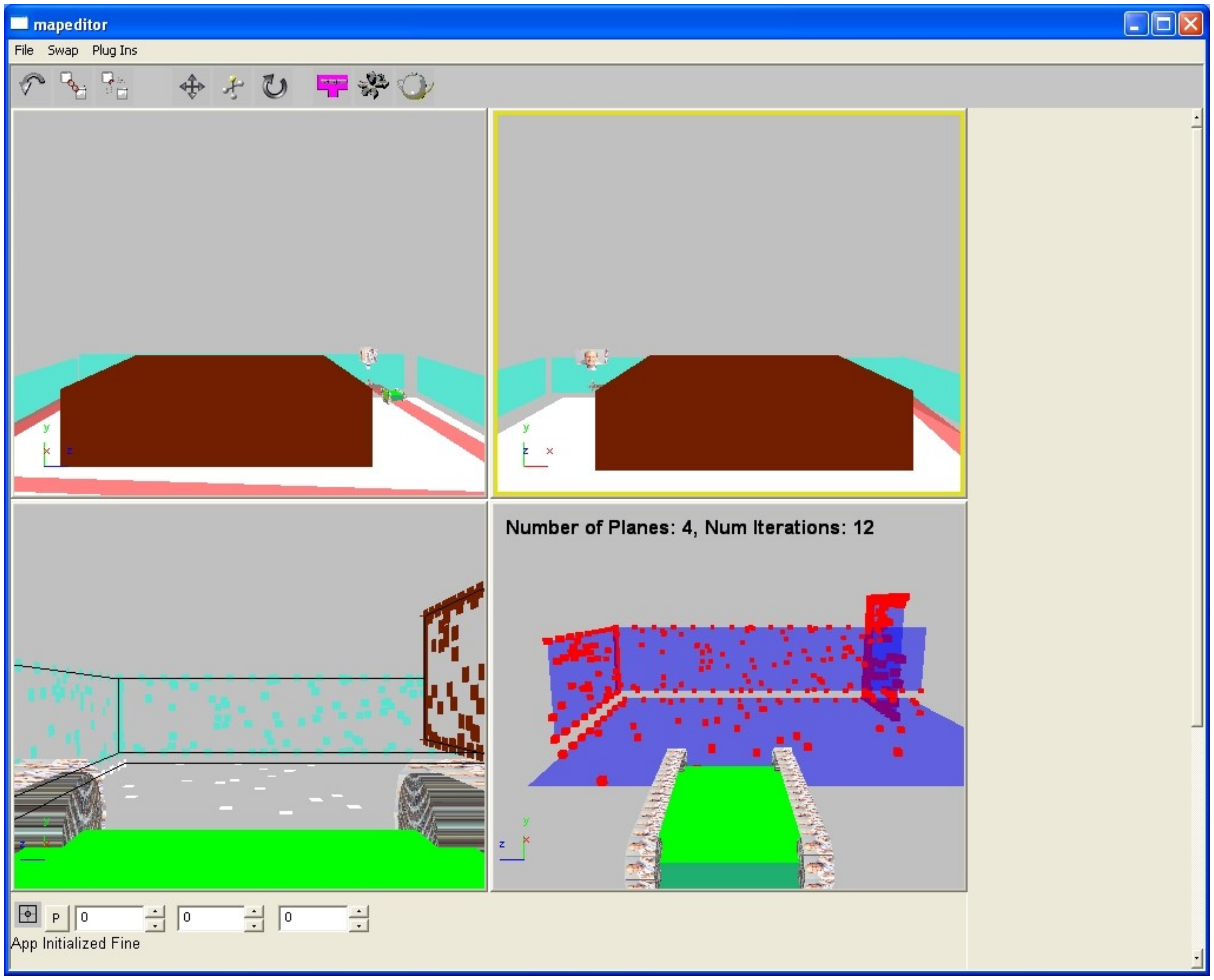


Figure 46 The simulated points and lines seen in the bottom left The bottom right has the results of the EM algorithm.

Figure 46 shows a simulator which given the robots viewpoint, produces 3D points and lines for use in vision algorithms. This simulation is far less complex than the second one, as some of the functionality such as CSG and PVS is no longer required.

Some new features are added to this version. This project uses a more advanced interface system such that it is possible to swap the program from being a simulation to being a map editor. It also uses a plug in AI system so that the AI is programmed in a separate project and then integrated into the simulation by loading a dll file.

The line algorithm rendering algorithm is interesting since it is similar to the earliest 3D renderers. A scene is sorted using a BSP and then all objects are rendered in back to front order. The lines of a polygon are extracted and placed in a software z buffer. The z buffer knows that any incoming lines are closer to the viewer than the previous ones. Any previous lines that fall inside of the lines of a closer polygon are overdrawn.

The points are rendered differently. Each polygon has points statically created. Then to see if a point belongs in the current view, a ray is projected from the points location to the robot viewpoint. If there are no collisions then the point is viewable. If there is a collision or if the point is not in the viewing frustum it is occluded.

The third simulation is used to evaluate the next chapters ICP and EM algorithms.

Chapter 3 The search for the Algorithm

3.1 Introduction

There are many different ways to provide a robot with autonomous mapping capability. This thesis starts with 3D points. 3D points can be generated from a stereo camera setup where features points are computed from each camera and then combined to form a 3D point cloud. Processing can occur directly on the points, or the points can be transformed into a higher order primitive. Different algorithms can then be used to register multiple readings of the same features. This chapter explores some of these methods and then attempts to create a new algorithm to meet this thesis's objectives.

The sections 3.2 to 3.5 describes the exploration of previous work done on this topic. The discussion then goes off in a different direction as the methodology chosen for this thesis is for the SLAM algorithm to operate on planes while maintaining as much past data as possible. Multiple attempts at creating an algorithm to achieve this goal is presented in section 3.6. This chapter does not arrive at a successful solution to planar SLAM, but through multiple attempts at creating the algorithm the problem is better defined. This makes it possible to solve the SLAM problem in Chapter 4: The Relative Plane Algorithm.

3.2 ICP

3.2.1 Introduction

Iterative Closest Point (ICP) is an algorithm that computes the difference of rotation and translation between two 3D point clouds. The algorithm is originally described in [BeMc92]. The least squared solution is given in [ArHB87] which includes a useful explanation of some special cases that did occur in simulation. A version of the ICP algorithm that is used successfully in a small environment is reported in [SaEs04].

There are various design choices in implementing the algorithm. An example of these are described in [RuLe01] such as which points to pick (all of them or take a sampling), and what matching criteria to use. Zinber et al, [ZiSN03] describes several refinements to improve performance. ICP can be used in some cases as the method to localize the robot's position [MiSi06] [SNJM04]. One caveat in using ICP, is that when two point clouds are compared, the distance traveled by the robot to generate the second point cloud cannot be so large as to create an aliasing effect [GaSo04]. The algorithm looked promising so it is implemented and tested in a simulation.

3.2.2 ICP Algorithm

1. Remove the mean from both point clouds.
2. Iterate until the error of the difference of the point clouds is within desired bounds, or if a certain number of iterations have passed.
3. Load all the points from the second point cloud into an overlapping quad tree An overlapping quadtree is a tree structure that is subdivided into a certain amount of cells at creation time (some quadtrees only divide the cells when objects are loaded into them). These cells overlap each other by a given amount. All points are loaded into the quadtree and are placed into every cell which it fits into. The advantage of the overlapping feature is seen when retrieving a list of points to be compared to a single point. The cell that contains the point without including the use of the overlap contains all points that are closest to that point given the overlap size. So it is guaranteed that all points given an overlap size are in the cell. This leads to very fast point retrieval as the quadtree does not need to be traversed if a point is located at the border of a cell.

4. For each point in the first point cloud, get a list of points that are close to it from the quad tree. Find the closet point match and save that point.
5. Load the matrices for the singular value decomposition (SVD) [ArHB87] and perform the SVD.
6. Perform the rotation on the second point cloud and reiterate the algorithm.
7. Rotate/UnRotate one of the means by the solved rotation difference and then compare it to the other to get the location difference for the iteration.

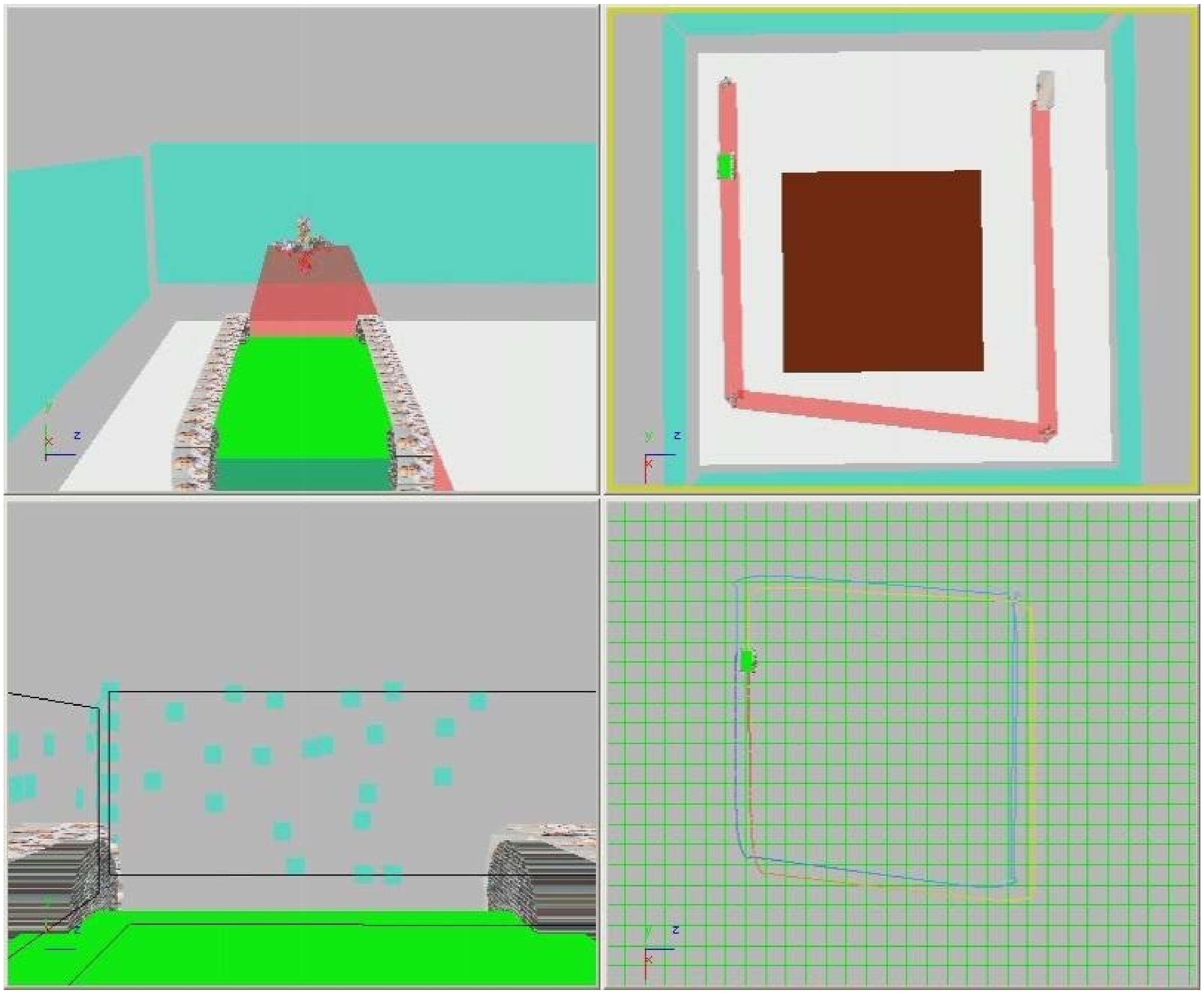


Figure 47 ICP algorithm

The ICP algorithm implemented in the simulation is shown in Figure 47. To verify that the ICP algorithm is 100% accurate in a simulation without noise an additional step is taken. If a point disappears or appears between two iterations, that point is not used in the calculation. Figure 47 bottom right window shows the robots true path in orange/yellow and the calculated path in blue. The blue path is initially offset so the two paths do not overlap. After the robot travels the path several times the blue path is consistent, showing ICP's accuracy. ICP is also attempted with noise but did not perform well as shown in Figure 48.

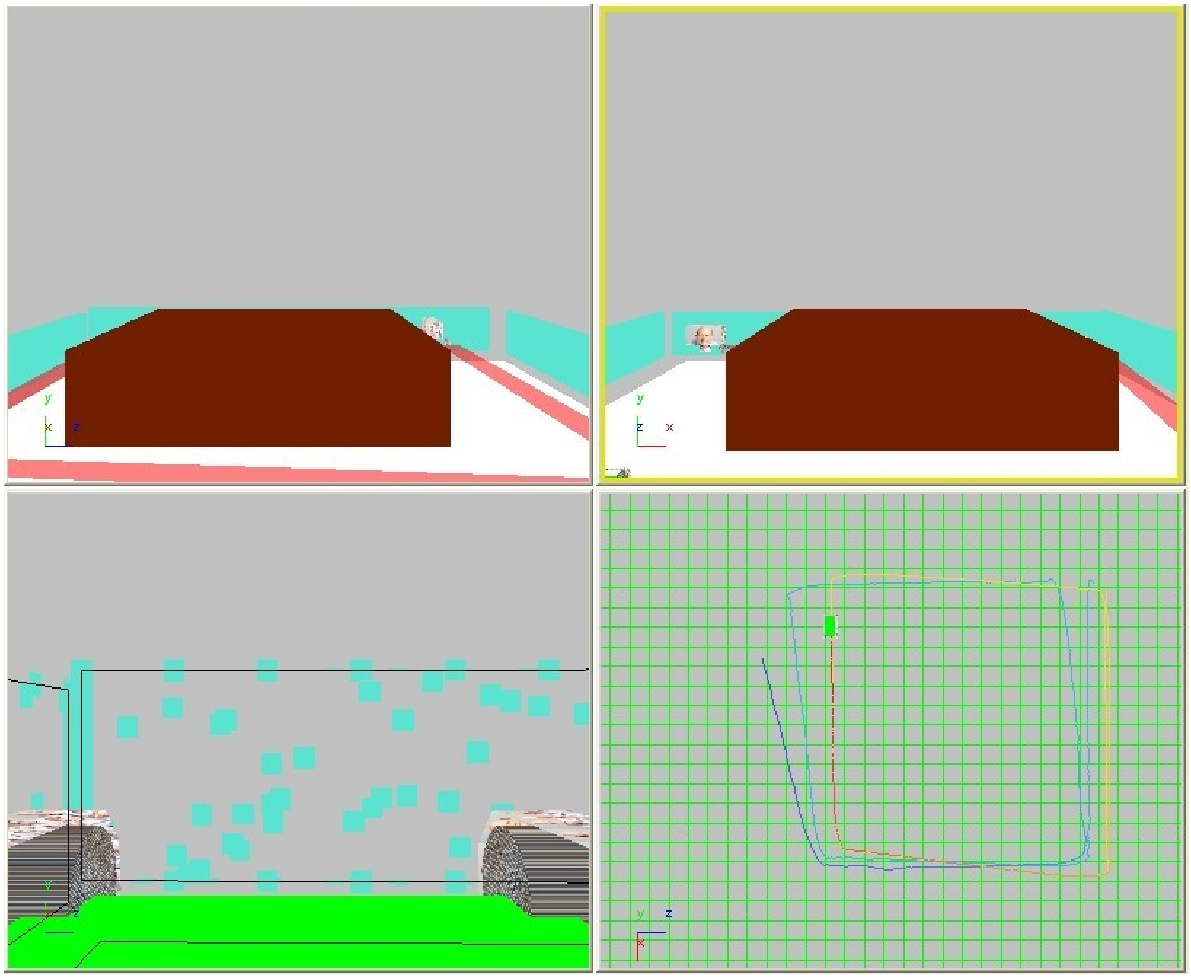


Figure 48 ICP with noise.

3.2.3 Remarks

Even with noise, ICP gives somewhat accurate position information if looking from a single iteration's perspective. If using it over a large distance it has the tendency for errors to accumulate.

3.3 Extended Kalman Filters with particle filters using landmarks

Extended Kalman Filters (EKF) and Particle Filters are covered in one section as they are combined in FastSLAM [MTKW02]. Two key papers for the use of the EKF are in [SmSC86] and [LeWh91]. Dissanayake et al, [DNCD01] shows how EKF SLAM could work for an outdoor environment. A good reference into the topic is the book [ThBF06] and a good starting point to learn the algorithm is given by [WeBi01].

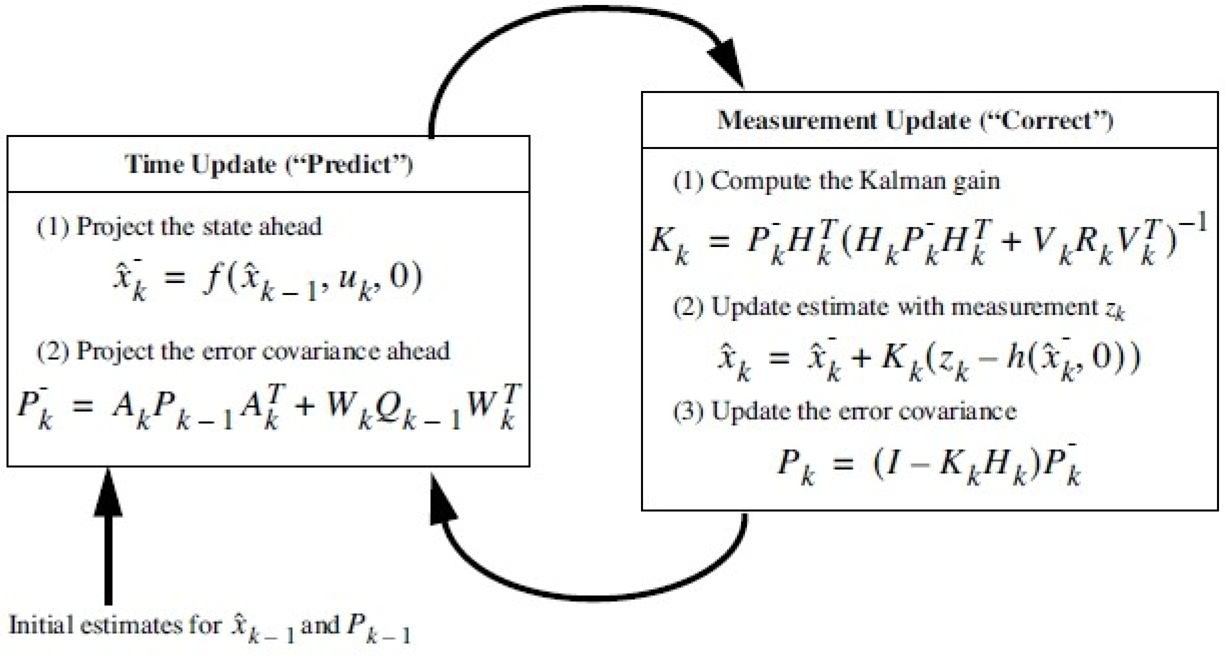

Time Update ("Predict")
(1) Project the state ahead
$\hat{x}_k^- = f(\hat{x}_{k-1}, u_k, 0)$
(2) Project the error covariance ahead
$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T$
Measurement Update ("Correct")
(1) Compute the Kalman gain
$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1}$
(2) Update estimate with measurement z_k
$\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, 0))$
(3) Update the error covariance
$P_k = (I - K_k H_k) P_k^-$
Initial estimates for $\hat{x}_{k-1}$ and P_{k-1}

Figure 49 The EKF algorithm. Diagram from [WeBi01]

The EKF algorithm is:

1. Predict the next state by using input from a motion model.
2. Calculate the error covariance using the process noise estimate Q. This is the expected noise of the dead reckoning system.
3. Calculate the Kalman Gain K. K can be thought as a measure of how much to trust the measured position to the accuracy of the landmarks. It uses R which is the estimated landmark noise.
4. Calculate the estimate location by mixing the predicted location with the one given by using the

landmark measurements using the Kalman gain *K*.

5. Update the error covariance for the next iteration.

Notice that the mixing matrix *K* is only dependent on the variables *Q* and *R*. This leads to the observation where "Under Conditions where *Q* and *R* are in fact constant, both the estimation error covariance P_k and the Kalman gain K_k will stabilize quickly and then remain constant" [WeBi01]. So if there is a flat noise model the Kalman filter can be thought of as simply mixing the result of the landmarks with a predict position of the robot's movement. Often the noise model used is that landmarks with a longer distance have more noise, so the closest landmarks would have a greater weight than the farther ones.

Using the tutorial example given in [WeBi01], a scalar random constant is estimated using the Kalman Filter. The results are shown in Figure 50, Figure 51, and Figure 52. The red dots shows the actual noisy values while the blue line shows the estimated value. When *Q* is much lower than *R* in Figure 50 the predicted model is more trusted than landmarks measurements and convergence takes a while. When *Q* and *R* are closer together in Figure 51 convergence is quicker. In Figure 52 *Q* is very high so the predicted model is hardly trusted and the landmarks are used. Note that the black line toward the bottom in all the graphs are the Kalman gains and they converge quickly.

A full EKF SLAM example is programed in Matlab to see how well it works. Figure 53 shows the results. Even though the prediction location has noise inserted in it, the resulting location is not that bad as the noise has no bias. The robot in Figure 24, like many robots had a bias when turning. This is simulated in Figure 54 which has an added bias to the odometry noise. The predicted location is far less accurate with the robot having a bias, but the EKF is able to keep the estimated position in the correct location.

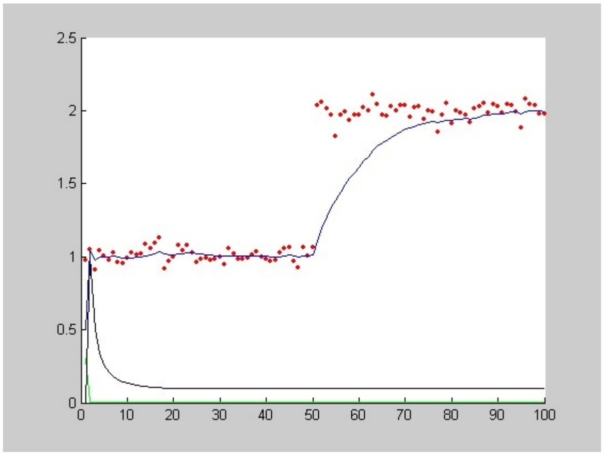


Figure 50 Low *Q*

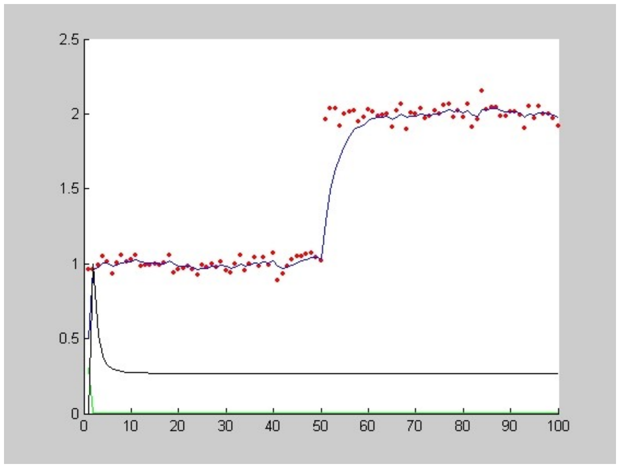


Figure 51 Mid *Q*

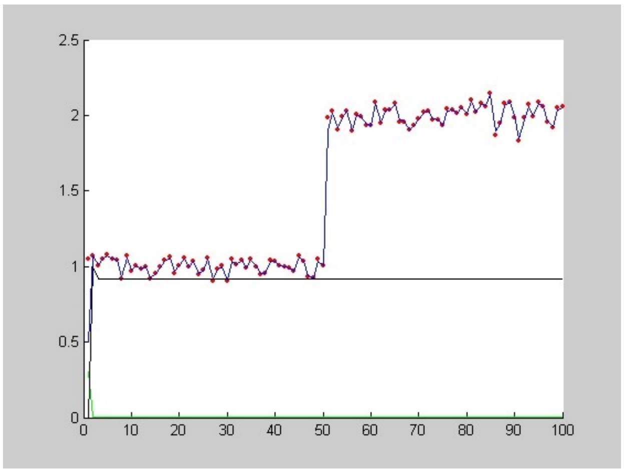


Figure 52 High *Q*

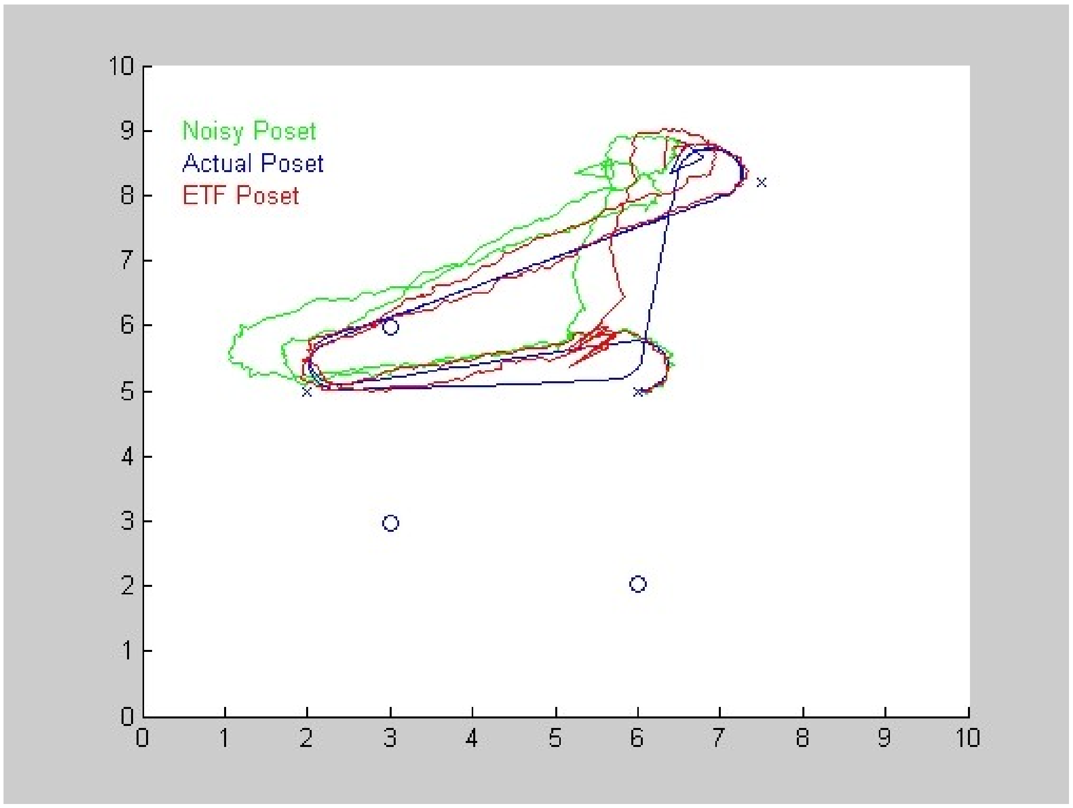


Figure 53 SLAM without any odometry bias

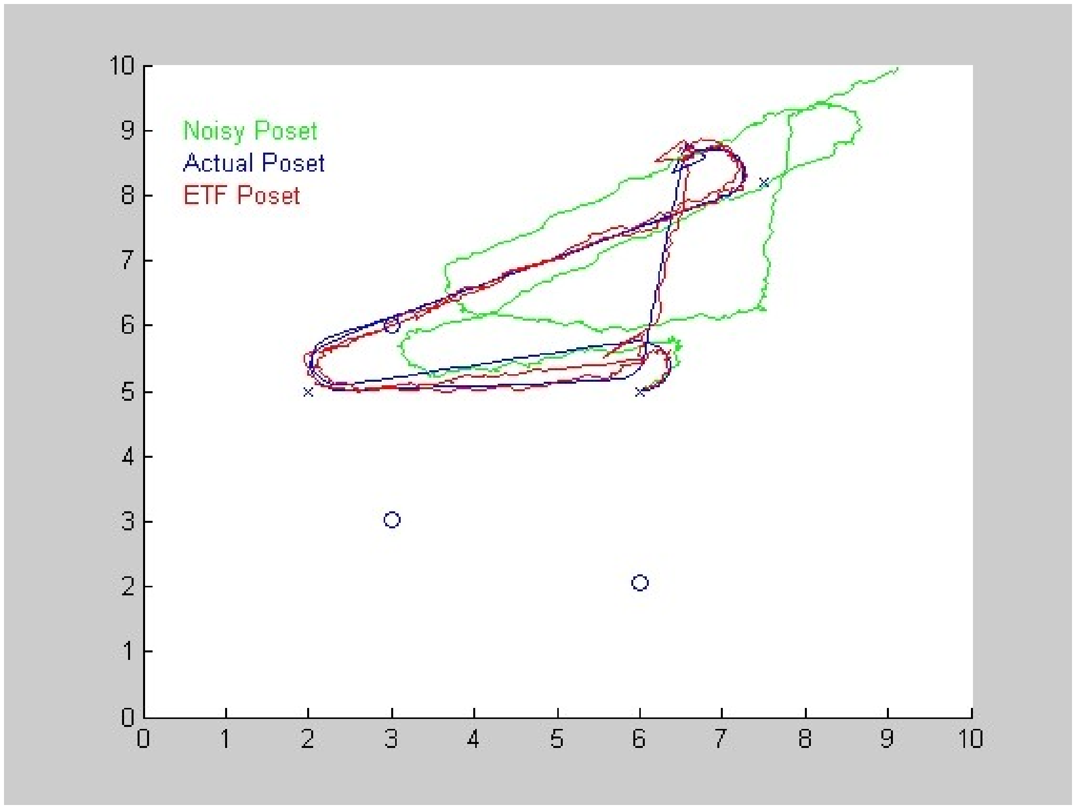


Figure 54 SLAM with odometry bias

One drawback to the EKF is the processing time. “The quadratic complexity limits the number of landmarks that can be handled by this approach to only a few hundred” [MTKW02]. It is further elaborated in [CGDM09] “the complexity is dominated... in the measurement vector size for the EKF update..... more than a hundred features are currently running at 1 Hz in an Intel Core 2 Quad at 2.83 GHz”.

An algorithm that can scale better than the EKF SLAM is FastSLAM [MTKW02] that uses particle filters. Rather than be computationally bound by the fact that the EKF matrices are the size of the number of landmarks, FastSLAM uses particles. Each particle uses an EKF for each landmark, so the EKF is no longer affected by quadratic complexity. The particles represents a probabilistic density field of where the robot could be.

In terms of accuracy [ThBF06] states “As the number of particles goes up the accuracy approaches the EKF” so the EKF is more accurate given ideal noise but it is possible for the particle filter to approach the same accuracy while taking far less computation time. FastSLAM outperforms the accuracy of the EKF given non ideal noise. The accuracy of EKF versus FastSLAM is examined in [SaMN08]. The high accuracy of the EKF is shown under Gaussian conditions but “how fragile the EKF-SLAM can be due to non-Gaussian implication”. The paper cites the accuracy of FastSLAM with non-Gaussian noise but does mention the potential of “there will not be enough particles left to incorporate for the rest of the path estimation... the filter diverges and the result will be catastrophic. This property that affects the performance of FastSLAM is called sample impoverishment.”

FastSLAM also has the advantage of, “It has been observed frequently that false data association will make the conventional EKF approach fail catastrophically, FastSLAM is more likely to recover thanks to its ability to pursue multiple data associations simultaneously” [MTKW02] One draw back may be that the computational complexity scales by the number of particles required which might “depend linearly on a particle-filter specific parameter (the number of particles), whose scaling with environmental size is still poorly understood. None of these approaches, however, offer constant time updating while simultaneously maintaining global consistency of the map” [TKGW02]

An interesting quote is “The robustness of FastSLAM becomes apparent... ...simply ignored the motion information. Instead, the odometry-based motion model was replaced by a Brownian motion model. The average error on FastSLAM is statistically indistinguishable from the error obtained before.” [ThBF06].

One disadvantage of FastSLAM without odometry is “Impractical to work without odometric data because it will require a lot of particles and a very large noise covariance” [Husa10]. [ElSL06] uses FastSLAM with visual odometry at the cost of an average time of above one second per iteration.

Perhaps the advantages and disadvantages of FastSLAM versus EKF is best described by [FuWi03] “Particle filters approximate the posterior distribution with a set of samples that simulate the probabilistic model of the system. Thus, they are applicable to a range of general, non-Guassian, non-linear models. However, with a few exceptions, these methods are purely simulational in the sense that they sample the complete state space. Hence, for large systems, the sample size is too large to be practical.” and the EKF “represent the belief state as a mixture of Gaussians.... Their Gaussian representation and focused search provide an efficient solution to high-dimensional problems. At the same time, non-linearities and merging can introduce significant bias in the estimate.”

It is also possible to speed up the EKF by using sub mapping. A map is divided into smaller areas and each area has its own EKF which improves the processing efficient of the algorithm by keeping the matrices small. [LMSK03] uses a layered approach combining EKF with topological maps. [Fres07] provides a framework called Treemap to subdivide an area into smaller ones based on which features are close together. [PiTa08] uses a single camera and divides the features obtained into sub maps.

One more approach to speed up the EKF is to use the inverse of the covariance matrix, the information matrix. The information matrix stores links between pairs of features which is sparse [TYDG04]. The result is a faster algorithm but there are issues in maintaining the sparcification of the matrix.

3.4 Non landmark algorithms

Rather than defining a map by a small number of landmarks, some algorithms define a map by labeling if a space is occupied or not. Occupancy grids consist of an area being mapped into 2D or 3D cells and each cell is probabilistically occupied or not. Algorithms that use occupancy grids can use EKF, particle filters, or just scan matching to register multiple readings.

It is interesting to observe that scan matching with raw laser output yields pretty good results by itself. Hahnel et al, [HaSB02] discusses laser scanning with a probabilistic occupancy grid and dynamic objects. It has a filter to compensate for dynamic objects. The filter improves the map however both maps, map the outline of the area correctly. Hahnel et al, [HBFT03] compares a map taken of a robot traveling in a loop several times. Although the scan matching map had inconsistencies in it, it is still mostly accurate as compared to the version that had a particle filter applied.

Vision based occupancy grids have been shown to work. Some work uses 2D grids [MuLi00] [ElSL06] while some use 3D grids [Mora96].

With vision based SLAM there is a question as to what primitives to use. A disadvantage in using points as landmarks is that the quantity of points may be high. It is possible to undergo additional refinements to reduce the number of points used [SSSD06]. Another way to reduce the disadvantage is to transform the points to higher level primitives that reduces the number of entities registered on the map. This is done using an EM approach, forming planes out of points [WeSi05] [TCLH04]. It is also possible to use general polygons [ShBa04], and other higher order primitives such as boxes in [SKYT02].

One note about any algorithm that uses scan matching is that sometimes scan matching is insufficient to create a map. Ellekilde et al, [EHMD07] uses a laser range finder to build a 3D point cloud map, but if there are large distances between scans there might be an aliasing effect. It might not be possible to reconstruct the map using scan matching alone since there is not enough information to correctly align the scans. The approach taken is to use vision sensors to generate approximate positions that are then used to align the scans.

3.5 EM Algorithm

3.5.1 Introduction

The expectation maximization (EM) algorithm [Demp77] uses as input a 3D point cloud and generates planes from the point cloud. The algorithm is a general purpose algorithm that contains two steps. The E step where the data is mapped to a given model, and then the M step where the parameters of the models are maximized to fit the data. The model itself is generated randomly at the start of the algorithm and is continuously adjusted with additional random parameters to fit any data that is not yet matched to the model.

The EM algorithm solves the dual problem of not knowing what planes exist in the 3D point cloud, and given the planes; which planes do each point map to? A very good tutorial that is worth reading on using the EM algorithm to match 2D points to lines is by [Weis97]. The EM algorithm can be used for 3D points and planes as shown by [TCLH04]. There are variations on the algorithm such as [WeSi05] that first subdivides the space into cubes of .25m of length before running the EM algorithm, so that plane size expansion is not an issue.

Perhaps the best way to explain how the EM algorithm works is to simply list the steps in the algorithm. There are two different implementations listed in the following sections.

3.5.2 EM Algorithm 1

1. Iterate the algorithm until ending conditions are met which are calculated later in the algorithm.
2. Each known plane contains a point list of all points that match to it. At the start of the iteration it contains the points from the previous iteration. This version of EM matches all the points every iteration so reset the list.
3. Match the points in the point list given as the input data of the algorithm. This list is static except for each point in the list having a counter which counts how many planes use that point. That counter is reset every iteration.
    - A point is said to belong to a plane if:
        - The point has to be within the bounding box of a plane which is expanded from the starting bounding box with an expansion constant. This is done since a

    plane's initial bounding box contains only the original three points used to create it. Without some expansion the plane cannot grow.

    - The point is at least a minimum distance to a plane given by the distance to plane equation.

- It is possible to either have a point match to every plane that it belongs to or to just the one plane that has the smallest distance of all the planes.
- If the point fails to match to any plane then add the point to the unmatched list that is later used to generate new random planes.
- Any time a point is assigned to a plane its usage counter goes up by one. This is used to cull planes that do not have any exclusive points.

4. Recalculate the bounds of each plane so that all the points fit in the normal rather than the expanded bounds.
5. Check for ending conditions
    - Most or all of the points have matched to planes.
    - A certain number of iterations have been run and it is time to give up so the algorithm does not run forever.
    - There are no new valid planes created in the last few iterations. If for several iterations that are no new established planes it can be said that the EM algorithm is done even if there are many points that have not yet matched.
6. If the ending conditions have not been met yet then pick new random planes out of the list of unassigned points
    - Pick a point at random from the unassigned point list.
    - Find the two closest points to the random point. The points have to be within a distance threshold. If they are not then a new plane cannot be made out of that random point.
    - If there are two valid nearby points then create a plane out of the three points.
    - Iterate the algorithm several times until the desired quantity of random planes is found. If the algorithm has iterated past a threshold then stop since it is unlikely that new

random planes can be found.

7. Perform the algorithm described in [DrGe03] that uses total least squares to maximize the plane parameters given the point list for each plane.
    - Find the centroid of all the points.
    - Make a 3 by N matrix features [x_1-x_c,y_1-y_c,z_1-z_c;.... x_n-x_c,y_n-y_c,z_n-z_c] for all points in the point list [1..n]
    - Multiply by the 3 by N by its transpose to get a 3 by 3 matrix called “M” and then takes its singular value decomposition (SVD)
    - The SVD of M is $M = USV^t$. The S matrix contains singular values on the diagonal. The lowest singular value corresponds to the normal of the plane given in the matrix.
8. Merge planes that have a similar plane equation parameters and bounding boxes that overlap. The merged plane contains the points from both planes that have been merged.
9. For every plane make sure it has at least a minimum number of points. If it does not remove that plane.
10. Remove planes that do not have enough exclusive points. Search through all the points for every plane and use their plane usage counter to calculate the percentage of points that are exclusive to the plane (have a count of one). If the percentage is too low then remove the plane. Since planes are randomly generated it is possible for extraneous planes to be created. An example of which is a plane that is created near the intersection point of two planes that connect the two planes together. In this case its exclusive percentage would be 0% so it could be easily removed.
11. After the EM algorithm is done iterating, convert the bounding box to rendering points given the plane equation. This is done by selecting the four points in the bounding box that are closest to the plane.

3.5.3 EM Algorithm 2

1. Load all the points into an overlapping quad tree
2. Iterate the algorithm until ending conditions are met which are computed later in the algorithm.

3. Generate a list of all points that are not attached to a plane yet. This has to be done by looking at the usage counter of each point since there is not an automatically generated unassigned list in this algorithm.
4. Generate one random plane. This is done by picking an initial point from the generated unassigned list and then searching a list returned by the quadtree for the closest two points. If it is not possible to pick one new random plane after a certain amount of iterations then end the algorithm.
5. Search for points to match to this new plane by doing an iterative process:
    - Expand the planes bound by an expansion constant.
    - Retrieve the point list from the quadtree given the initial random point of the plane.
    - Go through all these points adding them to the plane if they are less than a maximum distance to the plane (equation) and in the bounding box of the plane.
    - Perform the algorithm described in [DrGe03] that is used to maximize the plane parameters given the point list for each plane.
        - Find the centroid of all the points.
        - Make a 3 by N matrix features [x1-xc,y1-yc,z1-zc;.... xn-xc,yn-yc,zn-zc]
        - Multiply by the 3 by N by its transpose to get a 3 b 3 matrix called "M" and then takes its singular value decomposition (SVD)
        - The SVD of M is $M = USV^t$. The S matrix contains singular values (on the diagonal). The lowest singular value corresponds to the normal of the plane given in the matrix.
    - After the plane maximization.
        - Cull the plane if it fails to have a minimum amount of points.
        - Start from the random point again and see how the point density of the plane changes as the plane is gradually expanded. If the density becomes too low then stop expanding and remove any points that are not in the higher density portion of the plane. This is done because sometimes a plane attached to a point from

another plane because it is close enough even though it should not. By checking the density this case can hopefully be eliminated.

- End the iterative process after a few iterations.

6. For every point selected in the now completed plane, add to their counter so they are no longer selected to generate a new random plane. Then recompute the unattached points list.
7. Merge the new plane with any others if possible.
8. Iterate until there are only so many points left, or a certain amount of iterations have passed, or an exit condition is arrived at (cannot generate a new random plane).
9. Remove planes that do not have enough exclusive points. Search through all points for every plane and calculate the percentage of points that are exclusive to the plane. If the percentage is too low then remove the plane. Since planes are randomly generated it is possible for extraneous planes to be created. An example of which is a plane that is created near the intersection point of two planes that connect the two planes together. In this case its exclusive percentage would be 0% so it could be easily removed.
10. After the EM algorithm is done iterating, convert the bounding box to rendering points. This is done by first looking for the 4 points in the bounding box that are closest to the plane equation. If the plane is rotated about the Y axis the bounding box may not align with the actual plane. To align the bounding box, rotate the bounding box incrementally by a total of 180^0 and search for the best fit. The best fit is the one with the lowest absolute value of distance from each point to each of the four lines in the bounding box. Rather than run the algorithm many times incrementing the angle checked by a small amount, each time a large degree increment is used. The best value is then subdivided into a range with smaller increments, and the best angle can then be subdivided again. This only has to be done at a few different increments to get a good result. In addition to adjusting the angle, the size of the bounding box is shrunk during the iterations so it fits the points as tightly as it can.

3.5.4 Remarks

The first algorithm keeps more to the spirit of the EM algorithm by recalculating all the planes every single iterations. The second one executes about twice as fast and takes advantages of the fact that once a good plane is found it is unlikely that it is going to be removed.

Before the algorithms are tested in the robot simulation, they are first tested in a test bench shown in Figure 55. The test bench randomly generates planes and then generates random points inside of each plane. These points are sent to the EM algorithm which then computes the planes that the points match up to. The EM planes can then be visually compared to the actual planes to see if it worked.

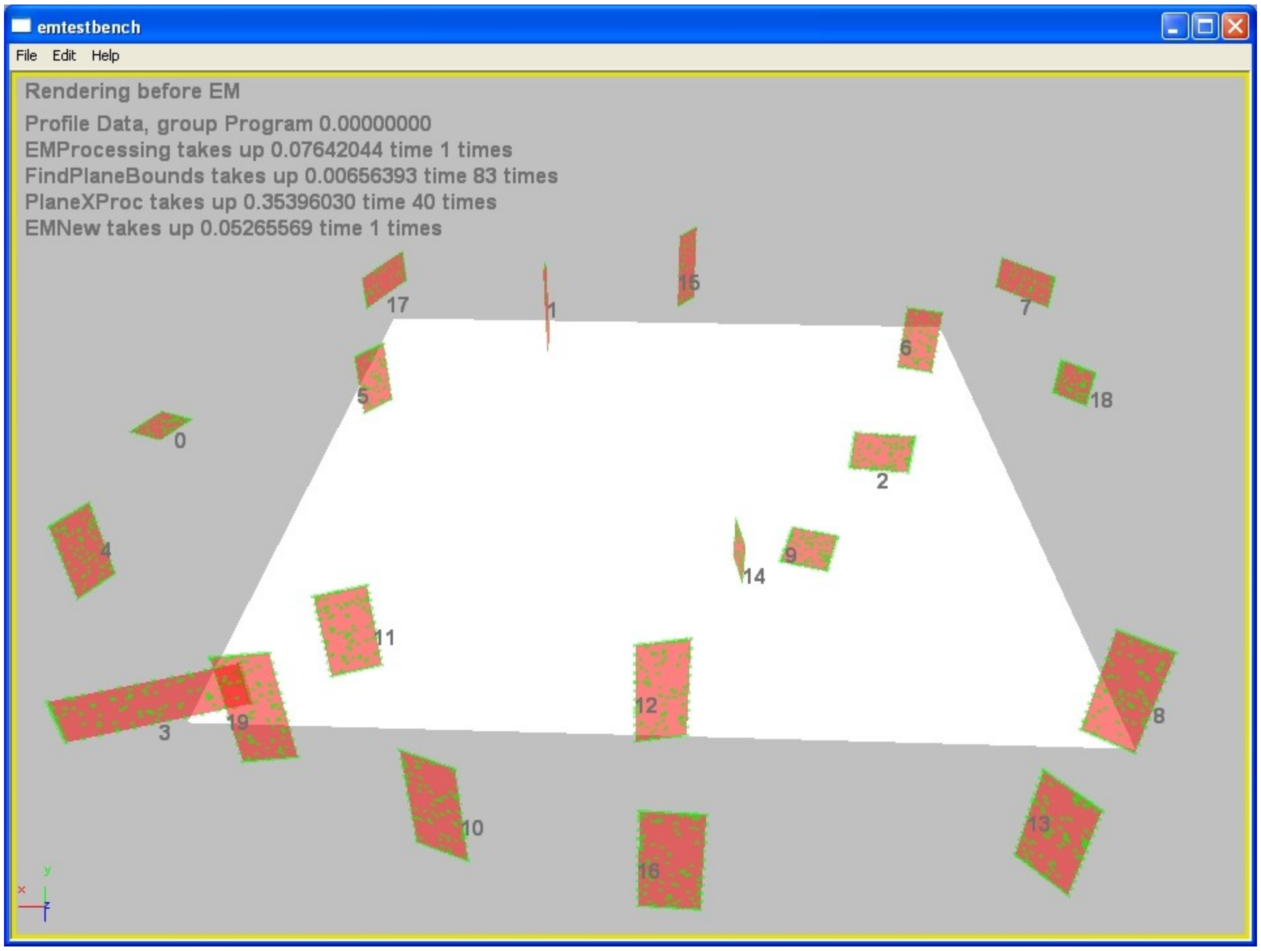
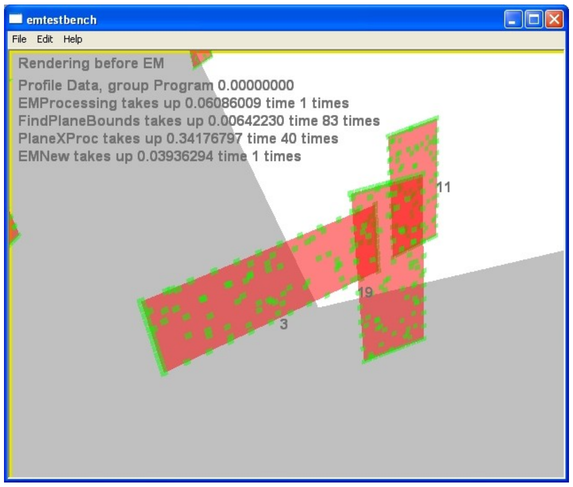
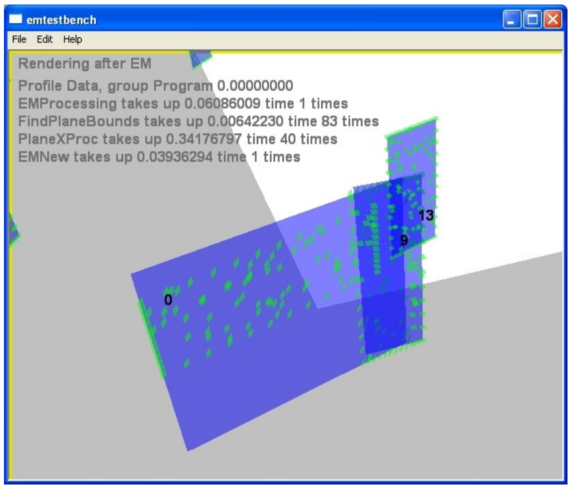

```
Rendering before EM
Profile Data, group Program 0.00000000
EMProcessing takes up 0.07642044 time 1 times
FindPlaneBounds takes up 0.00656393 time 83 times
PlaneXProc takes up 0.35396030 time 40 times
EMNew takes up 0.05265569 time 1 times
```

Figure 55 EM algorithm test bench



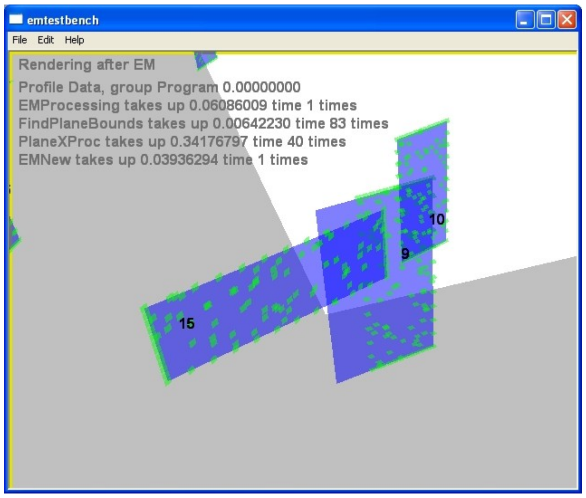
```
Rendering before EM
Profile Data, group Program 0.00000000
EMProcessing takes up 0.06086009 time 1 times
FindPlaneBounds takes up 0.00642230 time 83 times
PlaneXProc takes up 0.34176797 time 40 times
EMNew takes up 0.03936294 time 1 times
```

Figure 56 Actual planes



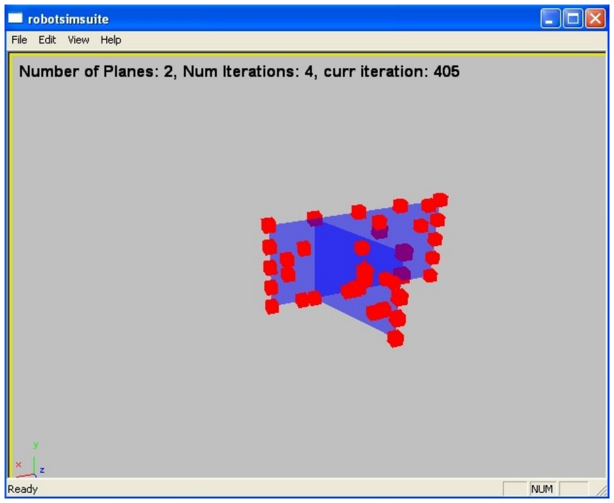
```
Rendering after EM
Profile Data, group Program 0.00000000
EMProcessing takes up 0.06086009 time 1 times
FindPlaneBounds takes up 0.00642230 time 83 times
PlaneXProc takes up 0.34176797 time 40 times
EMNew takes up 0.03936294 time 1 times
```

Figure 57 Plane stealing points



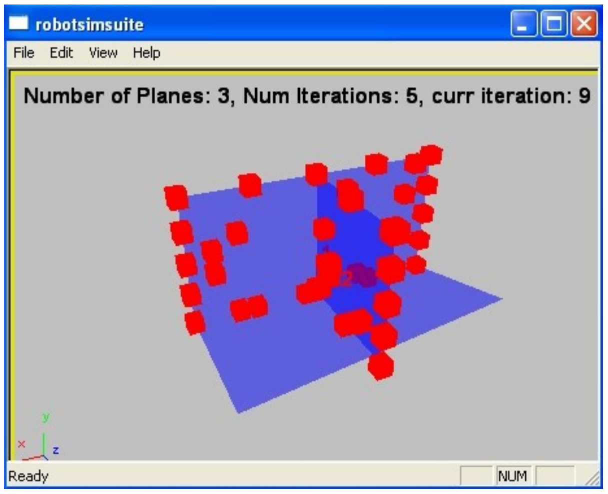
```
Rendering after EM
Profile Data, group Program 0.00000000
EMProcessing takes up 0.06086009 time 1 times
FindPlaneBounds takes up 0.00642230 time 83 times
PlaneXProc takes up 0.34176797 time 40 times
EMNew takes up 0.03936294 time 1 times
```

Figure 58 Other plane stealing points



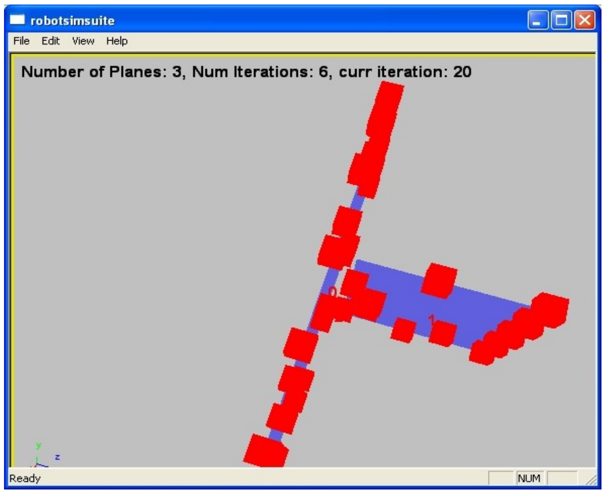
```
Number of Planes: 2, Num Iterations: 4, curr iteration: 405
```

Figure 60 Misaligned plane



```
Number of Planes: 3, Num Iterations: 5, curr iteration: 9
```

Figure 59 Extra plane



```
Number of Planes: 3, Num Iterations: 6, curr iteration: 20
```

Figure 61 Unattached plane

The algorithm is very sensitive to having two planes close together. Figure 56, Figure 57 and Figure 58 show how sensitive the EM algorithm can be when two planes are close together. In Figure 57 the horizontal plane uses points from the vertical plane and Figure 58 the vertical plane uses points from the horizontal plane.

It is important when two planes are perpendicular that they are constructed correctly. Figure 60, Figure 59, and Figure 61 are supposed to have one larger plane perpendicular and attached to a second smaller one. Figure 60 shows when the constant used to calculate the distance from point to plane is too large and the larger plane uses too many of the smaller plane's points; the smaller plane is computed incorrectly. Figure 59 shows when points can match to multiple planes and there is no check to see if a plane has enough exclusive points. An extra plane is created that has no exclusive points but technically is correct otherwise. Figure 61 Shows what happens if points can only be used once so the planes are unattached when they should be attached.

The EM algorithm is also sensitive to the fact that the random points used to pick the seeds for each plane are going to be different each time the algorithm is ran. In fact the the ordering of the incoming points can be different each time too. This is why the implementation of the algorithm contains such functionality of making sure each plane has enough exclusive points, and growing out the planes by expanding the bounds incrementally. The hope is that regardless of the random points used, each run the EM algorithm returns the correct result.

3.6 Journey to the Algorithm

3.6.1 Introduction

Earlier work on autonomous robotics [Krau02] showed that a robot's dead reckoning sensors are very accurate going forwards and backwards but much less accurate turning. Using planes as the lowest level primitive instead of points is attractive since the orientation problem can be solved by maintaining the orientation of the planes in a map.

This led to the desire that the SLAM algorithm used for this thesis would use planes. Planes can be created from points with the EM algorithm shown in the previous section . Watching a simulation of the EM algorithm is very intriguing. It seems obvious what the global map is when watching the simulation. But what algorithm can be used to generate a map given planes?

Much of the current literature on SLAM algorithms using 3D points use the EKF or a Particle Filter (FastSLAM). Using planes as a primitive add extra information to the problem as each plane has a normal that can be used to compute the robot orientation. However, it also has a disadvantage as the plane has both translation and rotation noise and a plane grows and shrinks as the robot passes it. EKF and FastSLAM are known not to work with dynamic movement. Perhaps a different approach can be used that stores past untransformed readings.

The goal is to create a SLAM algorithm that uses planes as the input, and can solve the dynamic movement by using past data. In addition, the computational complexity should be better than $O(n^2)$.

3.6.2 First Attempt

The first attempt at creating a SLAM algorithm is to combine the ICP and the EM algorithm. The difference in a plane's location is computed between subsequent iterations, and that is used to generate the position of the robot. Each untransformed plane is transformed given the current location of the robot to compare them to planes in a global map. If they are close to a known plane, they are placed in storage of that known plane. Then each known plane is recomputed as the average of all planes stored. If a plane does not match up to an existing one, it is added to the global map. There is functionality to remove and merge planes if necessary.

Unfortunately even without noise the ICP using planes is not accurate as shown by Figure 62. In the upper right corner, the blue line is the position given by the ICP and the red/yellow line is the true position. The error is possibly caused by that fact that the ICP uses the plane midpoint, and the midpoint changes as the plane grows and shrinks.. To see what would happen with correct ICP, the true global coordinates are fed into the algorithm and the results are shown in Figure 63. In that figure the map is built correctly.

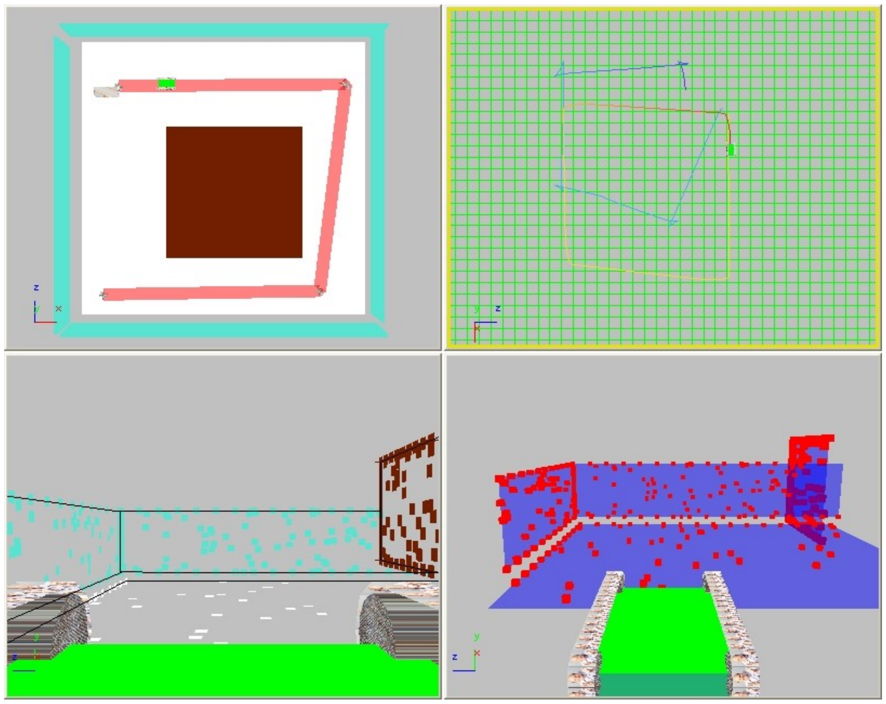


Figure 62 Combination of EM and ICP

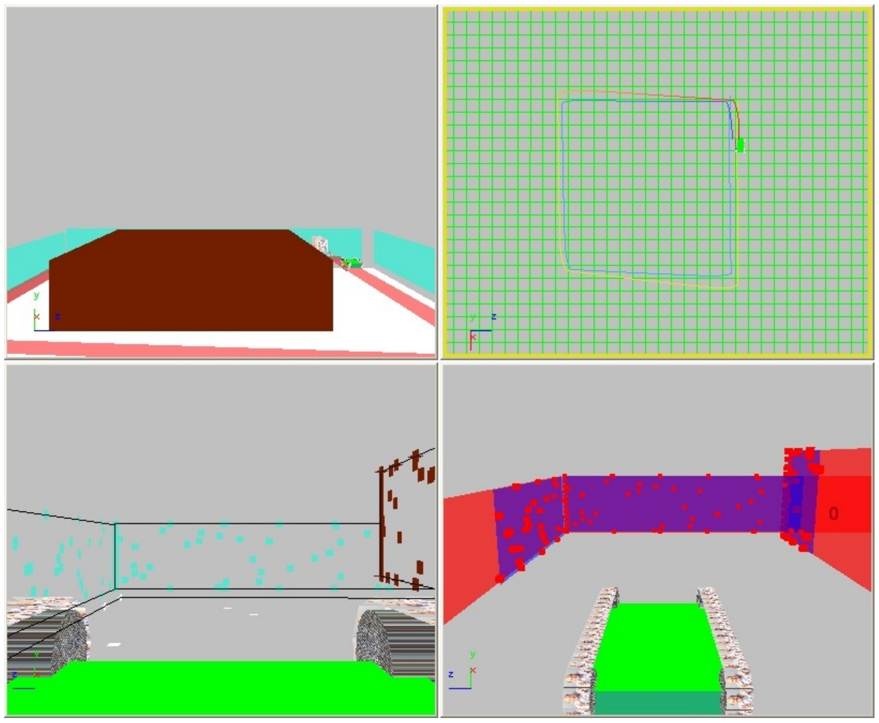


Figure 63 Combination of EM and fake ICP mapping

The results seem to show that the focus should be on trying to locate the robot as accurate as possible. This unfortunately is the wrong approach as described at the end of this chapter.

To further look into this problem a simpler simulation is used as shown by Figure 64. In the four planes problem the planes are always visible. The goal is to figure out a way to generate accurate position data in a way that does not use the planes midpoint.

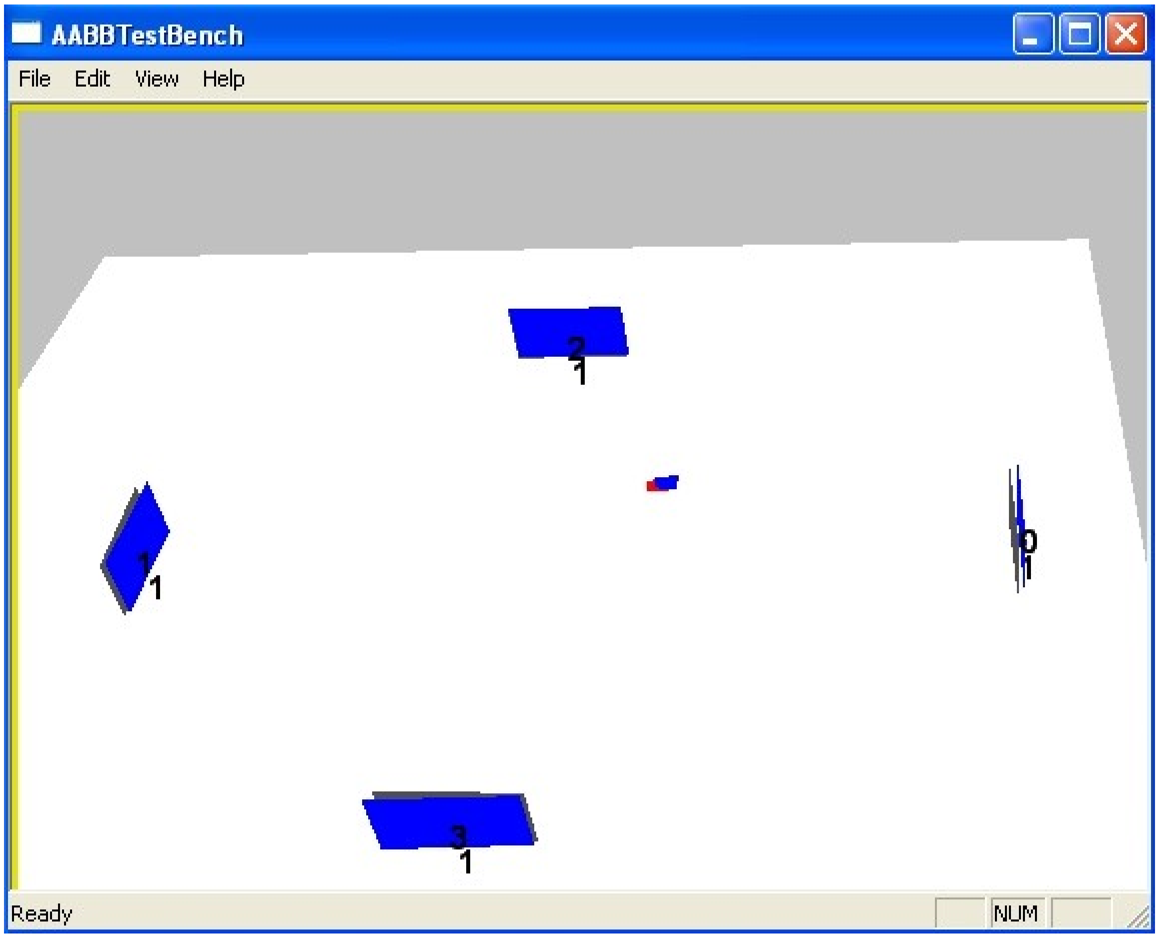


Figure 64 Four planes problem.

3.6.3 Second Algorithm

The second algorithm attempts to solve the problem by averaging the untransformed planes given an interval range.

A binary multiple, say 2^n observations of a given plane is chosen to be used to create the plane's average. Each plane is compared to its nearest plane (e.g plane 1 to 2, plane 3 to 4) assuming each is the starting plane and averaging the results. Then each of the computed averages are combined with their computed average of the same level and so on until only one plane is left. This plane is now the global average. If there are insufficient planes available say 3 when 4 are required, then the 3rd plane is upgraded a level, as it is considered to be the output of the merging of plane 3 and 4. The first observation can no longer be used as the current observation if the current iteration is more than 2^n beyond the first observation. In that case a saved value of the calculated plane average at n - 2^n is then used as the first observation. This will correctly orient the mapped plane to its proper location on the map.

This algorithm is shown to mostly work in the simulation As the simulation runs, the mapped planes are correct at least in the short term. For some reason in time, the mapped plane would gradually drift away from the correct position. As soon as the drift is large enough new readings of a plane no longer match to the global one, which causes the algorithm to fail. The drift can be fixed by locking the planes location after a certain amount of iterations have been seen.

The algorithm is then converted to using the distance to plane equation rather than the midpoint. It would only use planes that are approximately perpendicular to the robots viewpoint. It did work under ideal conditions that perpendicular planes are always available, but it is an unsatisfying solution as it did not use the edges of any parallel plane which is useful information.

A few things are learned implementing this algorithm. When there is a small amount of error over time it creates a large enough error so that the algorithm fails. This is due to using the current position to match current planes to their corresponding planes in the map. Another thing learned is that the planes growing and shrinking is a more difficult problem than anticipated earlier.

3.6.4 Third Algorithm

The third algorithm uses a particle filter algorithm. It takes the estimated odometry given by comparing the plane location between two iterations. Many particles are generated using multiple planes. If there are an insufficient quantity of particles more can be generated by using an error model. Then the difference of plane's locations given a span of iterations is used to determine which particles are more likely and reweigh or regenerate the particles based on this. The theory is that particles which are created due to planes growing and shrinking would in the long term be found to be invalid and the true location would be found after those particles are removed. As long as there is one particle in each point in time that contains the correct movement the algorithm should work.

The representation of a plane is changed from the middle point and angle to the angle and the plane's four corners. Each rotation/corner pair creates a particle. The particles are created by comparing corner points in subsequent iterations. The particles created using short term readings are compared against the long term readings. This process can be repeated for increasing number of iterations of the long term readings, for example the 5th reading, the 10th and then the 15th. It is necessary to do this since when there is abrupt movement, say a rotation, using a long interval in the first correction would cause the movement to be smoothed over time which would not be correct.

The algorithm worked well as it could solve the robot's position even after putting artificial particle error to simulate a planes changing size. However as implemented, it is extremely computationally expensive and still a small amount of error results. Instead of continuing to work on this algorithm that would reweigh each individual particle, one that alters the center of mass itself it attempted next.

One good improvement over the previous algorithm is that when attempting to match a plane to the list of current planes the global map is not used. Instead the previous plane reading in local coordinates is used. This prevents errors in global position affecting the plane matching algorithm.

3.6.5 Fourth Algorithm

The previous algorithm attempted to solve the plane growing and shrinking problem by assigning each corner point a particle. The side of the plane that is growing and shrinking would give an incorrect particle and the hope is that this particle would in time be given no weighting. The way it was implemented proved to be too computationally complex to be used.

For this algorithm the center of mass given all the particles, is generated. After it is created the individual particles would not be referenced again. A hill climbing algorithm is then used to minimize the short term error given a longer term more accurate reading.

The first attempt at this created a new back allocation problem. Let's say the algorithm is minimizing over 10 steps and the first nine are already corrected. The newest, the 10th is incorrect. This would cause the error in the 10th iteration to be under corrected and the center of mass in the previous 9 steps to be corrected when it should not. This would cause long term drift based on any input error.

A solution to the back allocation problem is to minimize the error over several different ranges at the same time. To make sure the algorithm does not oscillate before permanently changing a value, any past or future reading that would be effected by that value is checked to see if overall, the error increased or decreased. If the error increased that change is rejected and the value is rolled back. Then the percentage, that is how much to change the value, is decreased so the next change would be smaller.

This algorithm had the best performance so far. When injecting error particles without noise it is able to correct that error. However, after adding in planes growing and shrinking to the simulation rather than using biased noise particles, the algorithm would no longer work. The reason being is that the biased noise was removed by using the long term readying which did not contain the accumulated biased noise. Using planes growing and shrinking, any error created by an edge of a plane changing in the short term is also there in the long term. Now it is fully understood that the planes growing and shrinking cannot be treated as a filtering problem, but rather must be identified and dealt with.

3.6.6 New Direction

It is interesting to understand the mindset when working on the algorithms. Perhaps there is an error in the interpretation of the first attempt in Figure 63. The mapping worked in Figure 63 after the location error is eliminated so the thought was to reduce the position error to get the mapping to work. Later it is found that the planes growing and shrinking creates an error that cannot be regarded as just being noise and then removed by filtering it out.

When observing the EM simulations it is obvious which plane is growing or shrinking but how to convert this obviousness into algorithm form? Some logic can be used such as noting when a plane is changing size and simply ignoring that plane. Ignoring planes might not work if the total number of planes is small. Perhaps comparing a plane against the global movements of all the other planes can identify which edge is the one growing or shrinking. However comparing the movement of an edge of a plane to the average movement can be problematic, since many planes can be growing and shrinking at the same time. Also noise in one iteration may give the wrong results that would then be propagated to future iterations. Another choice can be to compare each plane against every other plane but this would be inefficient and approach $O(n^2)$.

The requirements of a planar SLAM algorithm is to be to identify the static edges of growing and shrinking planes while having better than $O(n^2)$ computational complexity. Thinking about the problem in terms of this requirement led to the elegant solution described in the next chapter.

Chapter 4 The Relative Plane Algorithm

4.1 Introduction

The previous chapter describes several algorithms that are unable to solve the planar SLAM problem. After a while there is a realization that it is not necessarily that the algorithms are faulty, rather it is how the SLAM problem is defined. The previous attempts focused on using the planes to generate the current location as accurately as possible since this information is carried forward in subsequent computations. The focus was on modeling the plane shrinking and growing as noise and attempted to filter it out. Perhaps there is a different way

Regardless of the SLAM algorithm, planes or objects in general are first observed untransformed in the robots viewpoint. That is, they are viewed from the robots position being at coordinates (0,0) before being translated to a global position on the map. Most algorithms concentrate on using a filtering algorithm to determine the current position as accurate as possible. That position is then used to register the objects on the map. There is another possibility though. Since objects are seen together, their relative locations can be found based on their untransformed relative locations. If an algorithm can represent how objects are grouped together using only their untransformed locations as input then a map can be constructed. An algorithm such as this does not maintain current position as a state variable.

The Relative Plane algorithms works with the principle of storing planes in their untransformed state, and using the average of location and rotation differences in comparisons with other untransformed planes to form a relative map. Planes are grouped together by observation interval. Growing and shrinking edges are identified using the pair wise comparisons as the dynamic edges have a higher standard deviation than static ones. Since all the comparisons are pairwise, the algorithm is much better than O(n^2). The algorithm does not need to use current position except locally for backtracking and globally for closing the loop. There are additional features to this algorithm such as being able to work with only one partial plane using a motion model, and a roll back mechanism to correct errors.

Compared to the Relative Point algorithm in Chapter 6, the planar version is much more complex. Three sections, 4.2-4.5 are used to describe every aspect of the Relative Planar algorithm for reproducibility. However spreading the description over three sections makes it difficult to understand how the parts of the algorithm work together. This chapter starts with a general overview of the algorithm in 4.2 Algorithm Description.

4.3 The Derivation shows how the algorithm is developed chronologically and forms the rules that describe the algorithm. The first problem to be solved is the simple four plane problem in which there are four planes that are always fully visible at perpendicular angles to each other. This problem of mapping these four planes is a good starting point to test the basic concept of the algorithm. This step is described in 4.3.1 The Beginning, the four planes problem.

The next problem to be solved is the 4.3.2 Considering plane visibility, the hallway problem. What happens when planes appear/disappear from visibility? The simulation is expanded to be able to simulate this problem. After it is solved it is possible to fully traverse an area.

4.3.3 The enduring problem of calculation intervals talks about the the main issue affecting accuracy for this algorithm. The fact that the accuracy in position is dependent on the size of interval used to calculated the relative plane location.

4.3.4 Non uniform plane size, the addition of Key Points finally solves the problem of having planes shrink and grow which is previously unsolvable by the other attempted algorithms. Adding key points allow for the shrinking and growing problem to be solved but add a number of other issues that are solvable but have to be taken care of.

4.3.5 Closing the loop and deciding the current RltXPoint talks about an issue when the robot closes the loop and introduces the need to calculate the current RltXPoint.

4.3.6 Motion Model, when only one partial plane is visible, when only one partial plane is visible it is not possible to use the regular functionality to solve for location, and it is not possible to continue mapping in the case where a plane expands during a rotation. The motion model solves this problem by restricting the robot to a restricted although realistic movement and calculates the location using a simple point to plane distance equation. It then computes plane expansion by using the calculated location. Several issues such as key points in different intervals are found when testing the motion model and their solution is in this chapter.

4.3.7 Comparison pairs, going back in time, saving the RltXPoint state, pseudo plane merging.. This chapter introduces the concept of going back in time to fix an error that is discovered in the future.

It contains the solution to the issue of having a finite amount of storage by saving the RltXPoint state. It discusses rearranging the pairing of planes for comparisons. Finally it has pseudo plane merging when closing the loop.

4.4 Class Architecture describes each class that is used in both the simulation and the Relative Plane algorithm.

4.5 One iteration of the algorithm does a complete walk through of one iteration to show how the rules are executed in the class architecture.

4.6 Closing Remarks ends the chapter by discussing some additional features that are not implemented, some potential issues, some similar previous work and closes with a summary.

4.2 Algorithm Description

The section describes the algorithm in steps: starting from when only two planes are visible, to a full map, to a full map that has growing and shrinking planes, to a full map that has all of the previous features but also has instances where only one partial plane is visible.

4.2.1 Two Planes

The algorithm receives untransformed planes as input from the viewpoint's perspective. A data structure called rltplane shown in Figure 65 is used to store a plane's untransformed observation by iteration using a circular array. During the start of a new iteration each newly seen untransformed plane is compared to the last added untransformed plane in every rltplane. If a match is found it is added to that rltplane. If not a new rltplane can be created.

Iteration	Untransformed location	Untransformed Orientation
1	(x_{a1},y_{a1})	(nx_1,ny_1,nz_1)
2	(x_{a2},y_{a2})	(nx_2,ny_2,nz_2)
3	(x_3,y_3)	(nx_3,ny_3,nz_3)
..		
N	(x_n,y_n)	(nx_n,ny_n,nz_n)

Figure 65: The rltplane structure

The relative location of two planes is translation invariant but not rotation invariant as shown in Figure 66. To make the comparison rotation invariant it is required to set one of the planes to having a consistent orientation, say aligned with the x-axis.

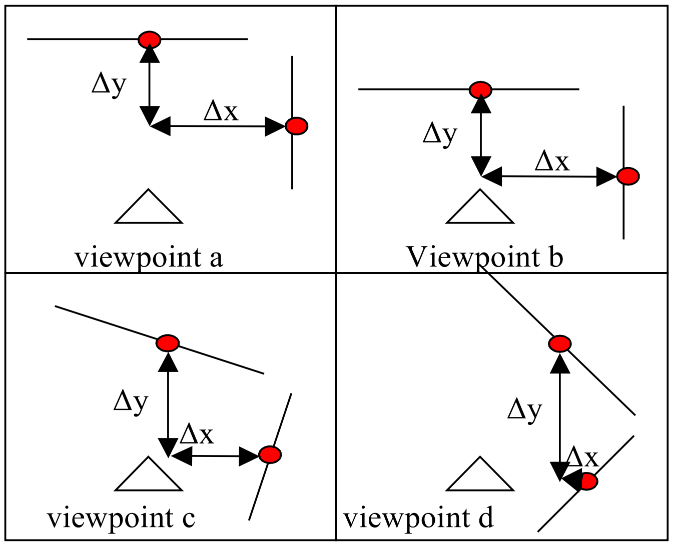


Figure 66: Invariance issue

Given two planes rltplane 1 and rltplane 2, solve the location average difference, Lav, (Δx, Δy) and average orientation average difference, Oav, over the interval where they are both visible.

The algorithm is as follow:

```
Initial $L_{av}$ and $O_{av}$ to zero
For each iteration in the given interval
    Obtain the untransformed planes from each rltplane
    Rotate both planes so rltplane 1 is at 0 degrees
    Add the Δx and Δy to $L_{av}$
    Add the orientation difference to $O_{av}$
Divide $L_{av\ and}$ $O_{av}$ by the size of the interval
```

When there is more than two planes a chain is formed by having every rltplane being in at least one comparison pair and choosing the comparison pairs so there is connectivity from every plane to every other plane. Say compare rltplane 1 against rltplane 2 and rltplane 2 against rltplane 3, etc.

Combining the comparison pairs together, a relative map is formed of rltplanes seen together. The relative map can be converted to a global map by aligning the relative map to its first observations assuming the viewpoint is initially at (0,0) or some other inputted coordinates. To calculate the current position, the last stored untransformed plane of an rltplane is compared against the global location of that rltplane. Note that the current position is regenerated every single iteration and does not propagate from the previous one. When registering untransformed planes, they are compared to the previous iterations untransformed planes in each rltplane. This means that the current position in this example is not used except for display purposes. Any issues, such as variance in the readings and how it affects current position does not apply to this algorithm, provided the next iteration's untransformed planes can be matched to their previous iteration's untransformed planes.

4.2.2 Many Groups

As the viewpoint changes, planes enter and leave the current view. This leads to grouping planes where all the planes are visible in the same time interval.

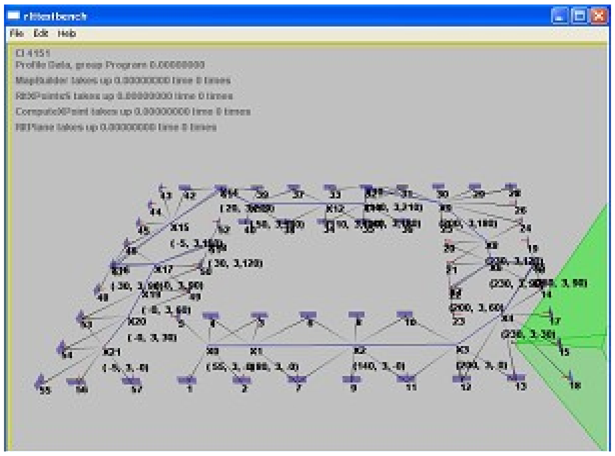
An rltxpoint group is created when a new untransformed plane is seen and is not visible in the interval of every rltplane in the current grouping. When this happens, a new grouping is created that includes the newly seen plane and ones from the previous grouping that are observed with the newly seen plane. The rltplanes that are present in both groups are used to link the two rltxpoint groups together when generating the map.

What happens when an untransformed plane is seen that is already being stored but does not belong to the current grouping? This can happen when backtracking or arriving at a previously seen area when closing a loop. It is desired not to make a new rltplane, rather the algorithm should identify which rltplane this untransformed plane should be put into. To do this, the current position is used to transform the untransformed plane into its global location. The planes global location is then compared against every rltplane in the generated map. If it matches to an rltplane, it is placed into that rltplanes storage which will cause it to automatically match to that rltplane in next iteration. This may seem like a conventional use of current position but it is not. The current position used is generated from the previous iterations observations and not a state variable that is propagated forward from previous iterations. The current position only needs to be accurate locally for the backtracking to work.

The backtracking mechanism allows for a loop to be automatically closed as long as the calculated current position is within the bounds of plane matching. This requires the current position to be globally accurate. If the loop is able to be closed the calculated current position automatically reverts back to the likely more accurate current position given by the first set of planes.

In terms of computation, rltxpoints only need to have their relative map calculated if any one of their rltplanes have had a untransformed plane added to it in the current iteration. If not, no further calculations need to take place. This way a map can be very large, however only local information would be processed. This automatic subdivision means that the algorithm is only as slow as the maximum number of planes it can see in an interval, and due to the pair based comparison this itself is an O(n) algorithm with a constant depending on the quantity of planes used to make the comparisons.

A full map is shown in Figure 67.



```
CI 4151
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
```

Figure 67: A completed map

4.2.3 Key Points

Using the algorithm described so far, it is possible to create a map if it can be assumed that planes are only seen at their maximum size, never the less due to shrinking as a plane leaves the view and growing as a plane is newly seen. The next section describes the algorithm to identify and make use of planes that are growing and shrinking.

There is a simple way to determine which corner points on the plane are the ones that are static and which are growing or shrinking. Since rltplanes are compared against each other, rather than use the untransformed planes midpoints to calculate the translation differences, all four corners, or two since height is static in the simulation, can be used. Each of the two corners can be compared against the other rltplanes two corners to form four comparison pairs. Since the rltplanes stores many untransformed planes it is possible to take the standard deviation of each of the four comparison pairs average location difference. The pair with the lowest is the one that is comparing two static corner points. The corner which is static is identified as the keypoint. This is shown in Figure 68.

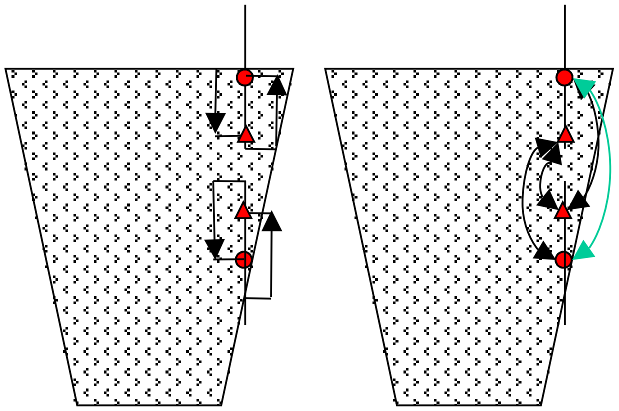


Figure 68: Showing two planes inside of a viewing frustum

There is an issue if there are two parallel planes, potentially on opposite sides of a hallway growing or shrinking at the same time. In this case there will be two pairs of keypoints that have a low standard deviation and without making an assumption of direction of travel it would not be possible to know which one is correct. To solve this, the rltplane chain comparison has to be altered so that non parallel pairs of rltplanes are compared against each other.

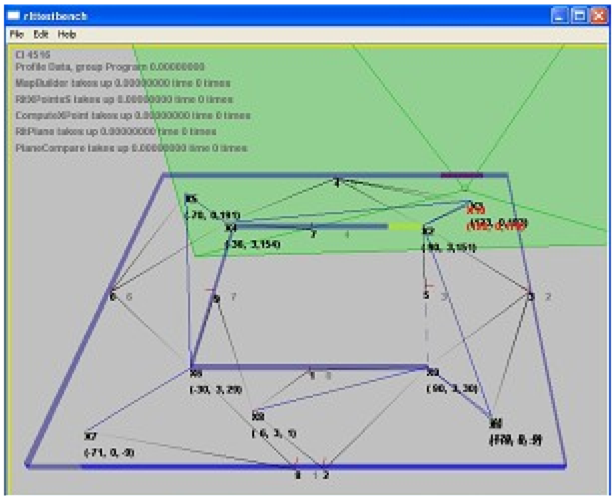
When using keypoints, the interval with two rltplanes are compared must be considered. It is possible for a plane to first be growing when it is first seen and then shrinking as the viewpoint passes it by. In this case two different keypoint pairs are valid at different times. The way to avoid having this effect the standard deviation, is to split up the calculation interval when the different keypoints are valid.

A plane when first spotted can be said to be in its growing phase. When it is detected to be shrinking, the interval is split. Either interval can be used to calculate the rltplanes relative position. This adds to the complexity to the algorithm as the interval used is the one that is considered the best, e.g has the most iterations. It is possible that when processing the rltxpoint such as when comparing plane 1, to plane 2 and plane 2 to plane 3 that the two pairs relative locations are computed in different intervals with different key points. This has to be taken into account when chaining together the relative positions.

4.2.4 Motion Model

Arriving at the end of the hallway as in Figure 69 and having only one plane visible can be a common occurrence encountered when traveling in a hallway environment. One solution might be to use the distance of point to plane equation to solve the current position. This unfortunately would lead to relying on using the previous location to calculate the current location which is not ideal. There is a better solution than this that is similar.

The plane at the end of the hallway would be correctly placed on the map earlier when compared to planes passed by that are no longer visible. The problem therefore is not the placement of the plane. The only unknown information is the planes maximum size as it expands as the viewpoint rotates. So the problem is not to calculate the current position, rather how to track how the plane expands as the viewpoint moves. Notice that this means that the rltplanes orientation is not affected by any data taken when the viewpoint is rotating since it is fixed to its position given by the rltplanes that have been passed by.



```
CI 4516
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 69: Only one partial plane is visible (seen in red in the top right).

4.2.5 Plane comparison pairs

The simplest way of choosing the plane's comparison pair is to select them in order of arrival. The first seen plane gets compared to the second seen plane, etc. This can sometimes lead to inefficient groupings in terms of the interval size available to compare the rltplanes. The current implementation uses a $O(n^2)$ algorithm to reorder the planes, however there is a faster version discussed in the future work.

4.2.6 Roll Back system

One benefit of maintaining all or as much possible data is that it is possible to roll back time and change some part of the algorithms calculations and then roll time forward again. This can be useful. For example, it can be found that a plane is in a rltxpoint that it should not belong to since its comparison interval is too small. It would be hazardous to attempt to remove it from the rltxpoint since it may be used in other places. It is possible to send back a hint that a rltplane should not join a specific rltxpoint or perhaps join any rltxpoint and then roll back time, use the hint and roll time forward again thus correcting the problem.

4.2.7 Save System

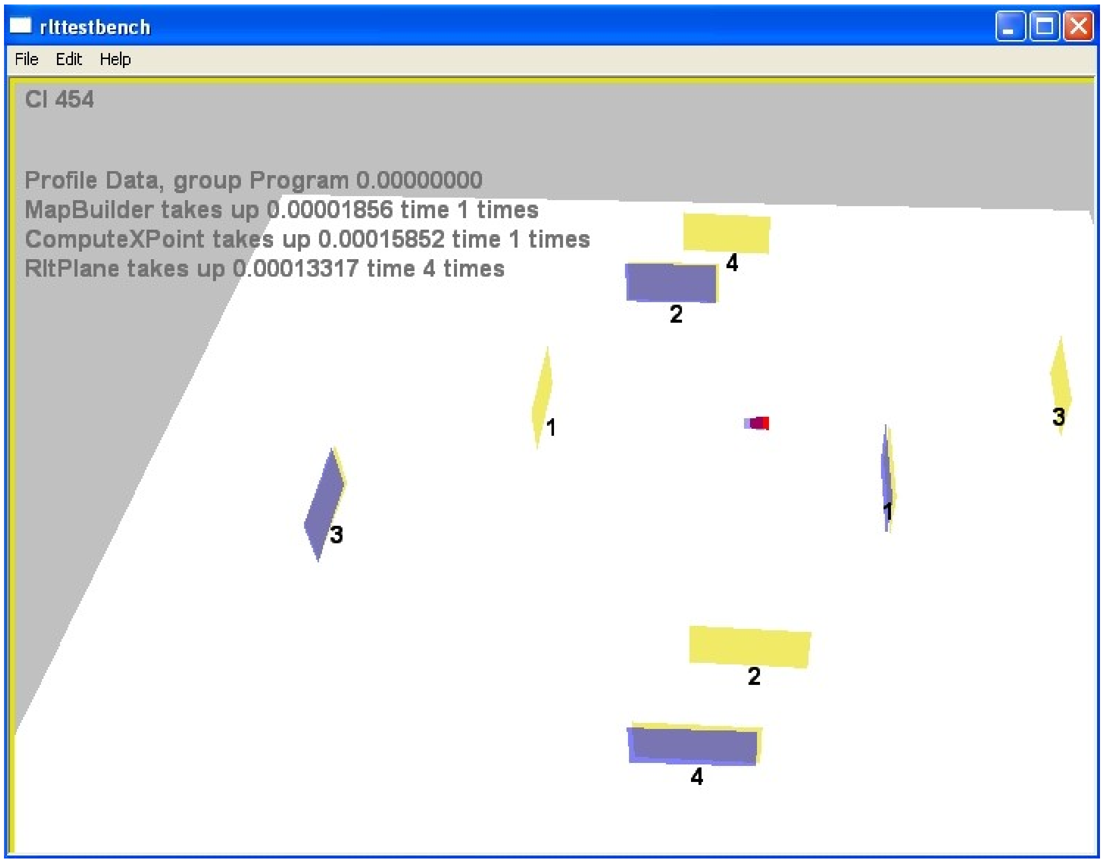
In case the memory available is not large enough to store every untransformed plane reading, it is possible to save the best rltplane comparison and to use that instead of relying on potentially less accurate plane comparisons.

4.2.8 Where errors can occur

Since this algorithm uses the untransformed plane readings and does not have any in between calculation that can be the source of error, the accuracy of this algorithm is related to the quality of the intervals used to do the plane calculation. If the intervals available are too small it is possible for there to be a small noise offsets on the map.

It is also dependent on the input quality of planes it received from the front end vision system. It may be possible to compensate for larger errors not simulated in this paper such as adding filtering to remove noise spikes in untransformed planes.

4.3 The Derivation



```
CI 454


Profile Data, group Program 0.00000000
MapBuilder takes up 0.00001856 time 1 times
ComputeXPoint takes up 0.00015852 time 1 times
RltPlane takes up 0.00013317 time 4 times
```

Figure 70 The four planes problem

4.3.1 The Beginning, the four planes problem

The first simulation to be solved is the four planes problem shown by Figure 70. There are four planes that are visible at all times and the robot travels in a repeating counter clockwise path inside of the area. The blue planes are the global position of the planes after they are generated into a map. The blue planes are compared against barely visible adjacent yellow planes which are the current global position of the last reading. There is some noise generated with each iteration which is shown by the fact that the adjacent yellow planes are slightly offset from blue planes. There is another set of yellow planes that are significantly offset from the other planes. These planes are the ones that are in relative space to the robot's position. They are rendered as if the robot is fixed at the starting position and planes move around the robot. They are just there to confirm that the robot is viewing the relative planes and not the global ones.

So the question is, given the concept of wanting to map the plane location by using relative position, how is that accomplished? In the start of every iteration, the untransformed data (a plane's location relative to the robot) is received and then processed. A class is created that stores untransformed planes called RltPlane. When a new plane is first seen, a new instance of a RltPlane is created. RltPlane stores the untransformed data in a circular array. It also has functionally to compare a new plane against the last plane entered. This is an important feature of the algorithm. Planes are sent to the correct RltPlane without any regard to the current global position. It is only dependent on matching to the last reading. This is Rule 1. Even if the global position is incorrect for a moment it will not affect the mechanism to place each plane in its correct RltPlane.

Rule 1:

Add untransformed planes to the known RltPlanes by comparing them to the previous planes that are stored in each RltPlane. Do not use the previous generated global position for this operation.

If we could see all the planes on a map at every given moment regardless of which ones the robot can actually see, how could we generate their relative position. Do we have to process every plane against every other plane which would be $O(n^2)$? The answer is no. If all the planes are visible for the same interval we could use differential signally theory [Wiki09c]. If we compare one plane against another one in any order for the exact same interval we should arrive at the identical map. This is Rule 2.

Rule 2:

If all the RltPlanes are visible in the same time, a map that is created by chaining together pairs of planes comparisons should be identical to a map created by chaining together a different combination of pairs.

Given untransformed planes how should they be processed? There are two choices, using the planes midpoint and its normal (its angle) or just using its midpoint. When using both the midpoint and the normal it is possible to describe the relative position given only two planes since the normal can be used to describe the angular offset. Using only the midpoint would require a third midpoint to describe the angular offset, since the angular offset is undefined for only two points. Since the planes do have normals they are used.

Can we directly compare two untransformed planes to get their position offsets? The answer is no. Remember that the planes are constantly changing position and rotation relative to the robot so we can not simply say that plane is always at 0^0 or 90^0. The comparison takes a look at the first plane's normal and calculates the angle the plane is currently at. Then both planes are rotated by the negative of this angle to set the first plane as being always 0^0. This is Rule 3 that allows for the position offset to be consistent no matter what angle the planes are viewed at. In Figure 1 if the robot is directly facing plane 1, comparing plane 1 to plane 2 would yield a negative x value and a position z value. However if the robot is facing plane 2 doing the same comparison would yield a positive x and z value, which would cause the average position location to be wrong. However if on the second comparison plane 1 is rotated such that it is the same angle as it is in the first comparison, the location difference would be identical to both comparisons.

Rule 3:

Compare individual planes, say plane 1 to plane 2 by first rotating plane 1 and plane 2 so plane 1 is on the x axis (0^0). Its position offset to plane 2 will be consistent regardless of the angle plane 1 is viewed by the robot.

Lets say plane 1 is compared to plane 2, and then plane 2 is compared to plane 3 etc, how is this combined to form a single map? Remember, the first plane in every comparison is assumed to be at 0^0. To calculate the map assume plane 1 is the starting position and its located at 0^0. We could use that to calculate where plane 2 is relative to plane 1. Then we take that value and save it as the current total offset. Adding the current total offset to plane's 2 comparison to plane 3 yields plane 3's offset from plane 1. Plane 4 could then be put into plane's 1 comparison space by adding plane's 3 offset to its comparison with plane 4. This process would return a map that is correct from planes 1 perspective. It is the same as the global map except it is offset as it assumes plane 1 is the starting position. To align it with the global map plane 1 uses Rule 4 that uses the first reading to align the entire map.

After the map is generated the location can be derived by looking at the last iteration untransformed reading and comparing it to the map, Rule 5. Note that the location is just used for display purposes as currently it has no usage in this part of the algorithm. This is a powerful concept. In an earlier algorithm the location is used to add untransformed planes to the RltPlane and if the location is off by a sufficient quantity, the plane comparison would fail and a new plane would be added. Without using the current location for any computation purposes, any errors in location can be recovered.

Rule 4:

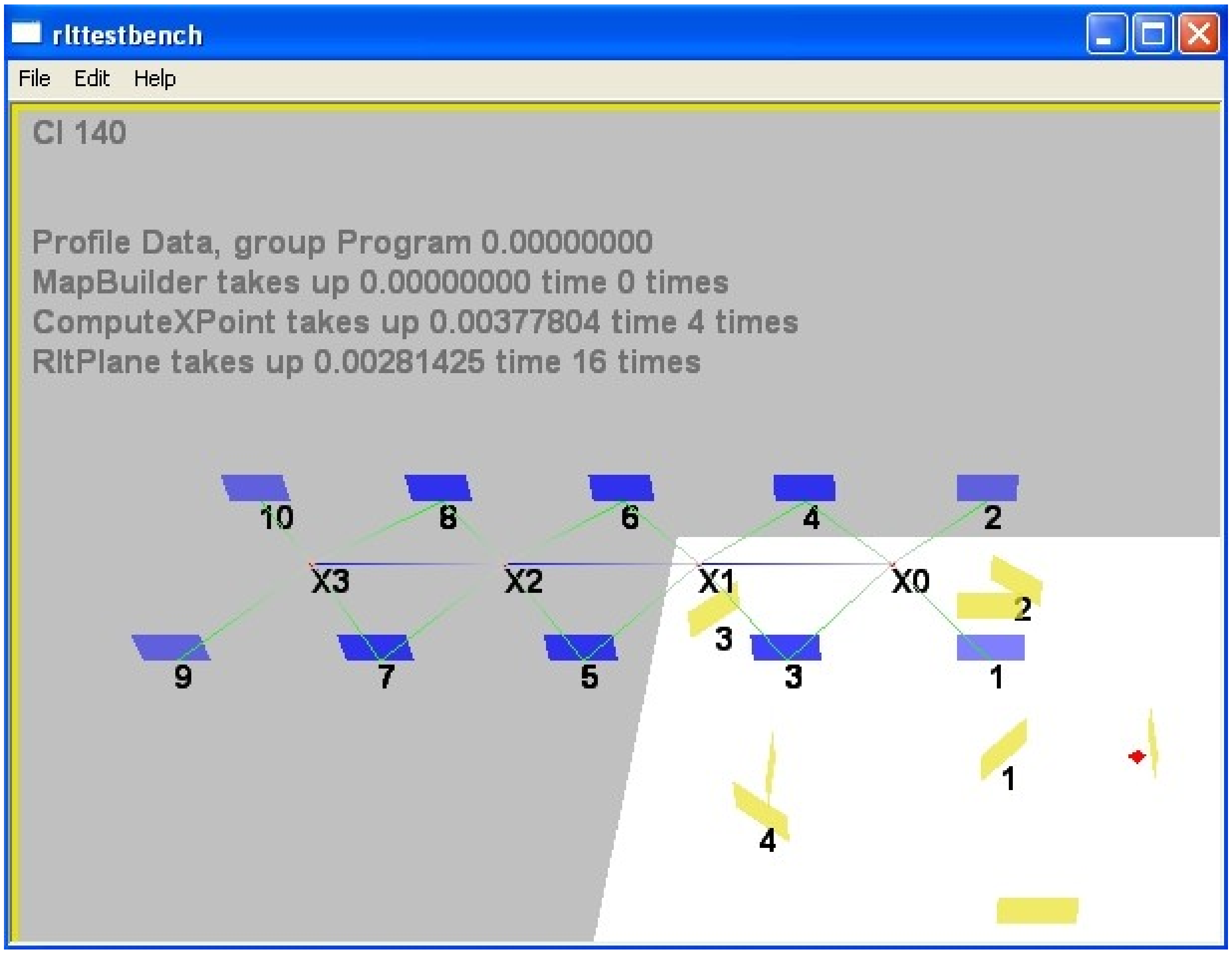
To align the planes relative offset to the global map, the initial location of a plane(s) is used and compared to its position in relative space. The difference of position is then used to align all the planes to their global location.

Rule 5:

To generate the current location, take the last iteration untransformed plane(s) and compare to the same plane on the map. The difference in location is the current location.

Using this simple algorithm the four planes problem is solved. The algorithm uses all the plane information from the start and is really quick. The processing time increases by O(n) for each plane added and by O(n) for each extra iteration calculated, but does not contain any O(n^2) functions. It also runs for a long time and does not have any drift.

4.3.2 Considering plane visibility, the hallway problem



```
CI 140


Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00377804 time 4 times
RltPlane takes up 0.00281425 time 16 times
```

Figure 71 The hallway problem

The previous section requires that all planes are to be visible at all times. So how does the algorithm work when only some of the planes are visible at any time? The solution is to form groups of planes visible at the same time. When computing the relative position, only planes in groups are compared to each other. These groups are then linked together by using planes which are present in both groups to calculate the offset of one group to another. The map building follows this iterative linking process until all the groups are placed on the global map, Rule 6. The class that forms the group is called RltXPoint.

A unit test is used before adding a viewing frustum to the simulation to restrict visibility. The unit test controls the visibility of planes based on an iteration. This is what is shown in Figure 71. Doing it this way made it easier to debug, as the exact iteration when each plane is visible is known when stepping through the code.

Let's start defining some rules for the RltXPoint.

Rule 6:

First compute the relative location of the RltPlanes in each RltXPoint and then combine the RltXPoints to form a map by using the offsets of the linking planes.

Rule 7:

All RltPlanes in an RltXPoint must be visible to each other, such that there is a interval that all the planes are visible to each other.

Rule 8:

There must be at least 2 planes linking each RltXPoint to another RltXPoint

Rule 7 states that in an RltXPoint all RltPlanes must be visible to each other. This is important because otherwise it would be impossible for the relative position to be calculated for planes that do not see each other.

Rule 8 states that when linking RltXPoints that there should be at least two planes linking them together. It would be possible to use only one plane but it is decided two is better and the offset is averaged between them.

To implement Rule 7, when adding a new plane, the RltXPoint generates the minimum interval where all the planes are visible to each other. If the RltPlane fits in the interval it can be added to the RltXPoint. If it does not fit a new RltXPoint is created. To create a new RltXPoint, linking planes have to be found. They are found by looking at all RltPlanes from the first RltXPoint and looking for ones that match Rule 9. After the best linking RltPlanes are found a new RltXPoint is created with the initial RltPlane and the ones that are used to do the linking.

Rule 9:

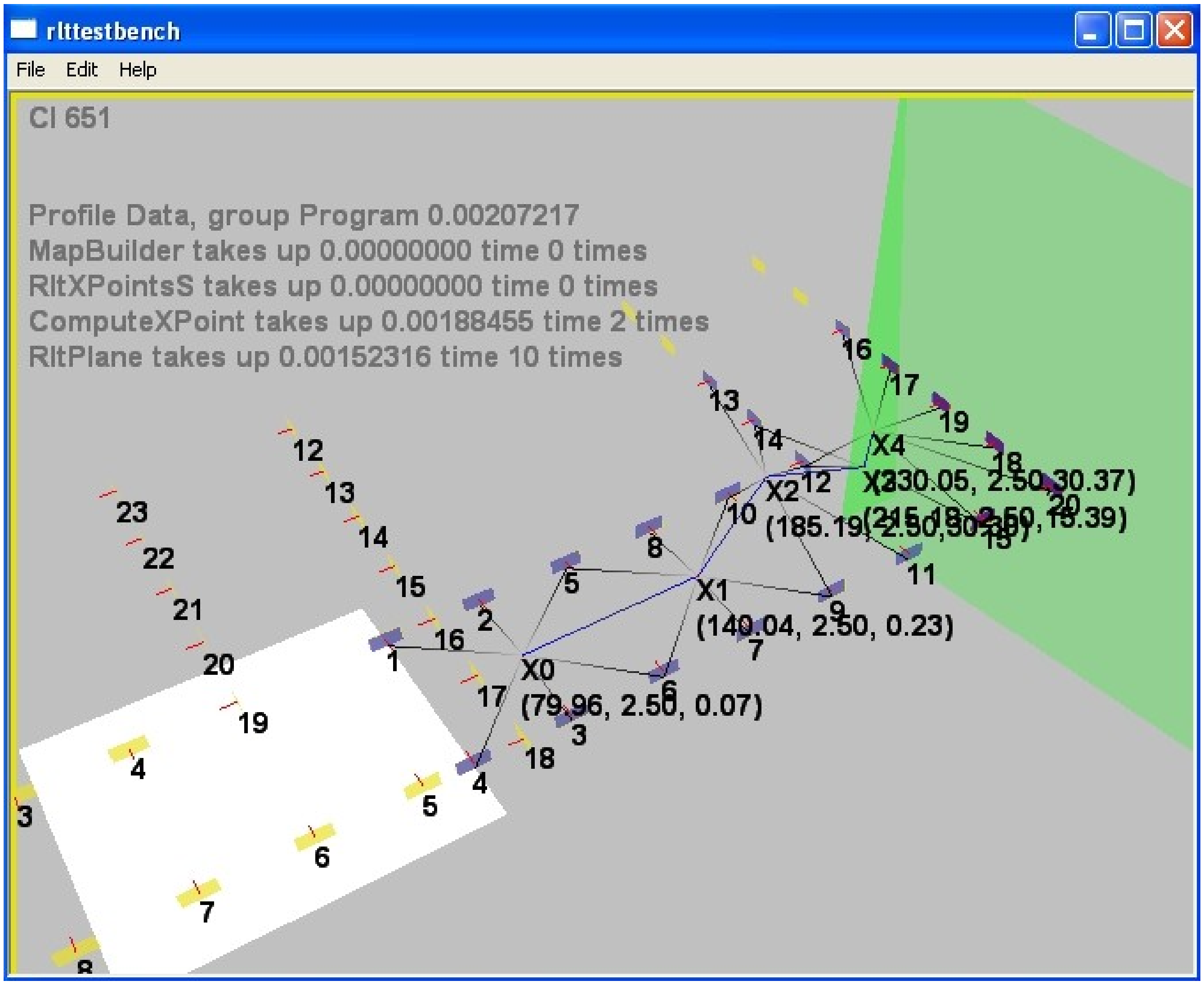
When choosing linking RltPlanes choose the one with the latest starting iteration in the hope that it will be the best link between two RltXPoints

Now it is possible to form groups. Before moving foward another issue has to be settled

Rule 10:

A plane could only be matched if the normal of the plane faces the correct direction. If it is the opposite direction a new plane is created.

Rule 10 is found to be important when a simulation had the robot rotate exactly when it is in line with a hallway, which is a fortunate occurrence since it allowed for the normal issue to be sorted out. The early simulation did not have any normal culling of a plane, that is the plane is viewable on both sides. When there is noise and the robot is in line with a perpendicular plane, to the robot the plane appears to be switching its normal. This occurs since the robot is viewing two different sides of a infinitely thin wall. The reason it is a problem is that the plane's normal is used to calculate the robot position when comparing the currently viewed plane to a mapped plane. When the normal is vague there are two possible positions for the robot to be in. In Figure 72 the normals of each plane are shown by a red line coming out of each plane.


rlttestbench

File Edit Help

CI 651

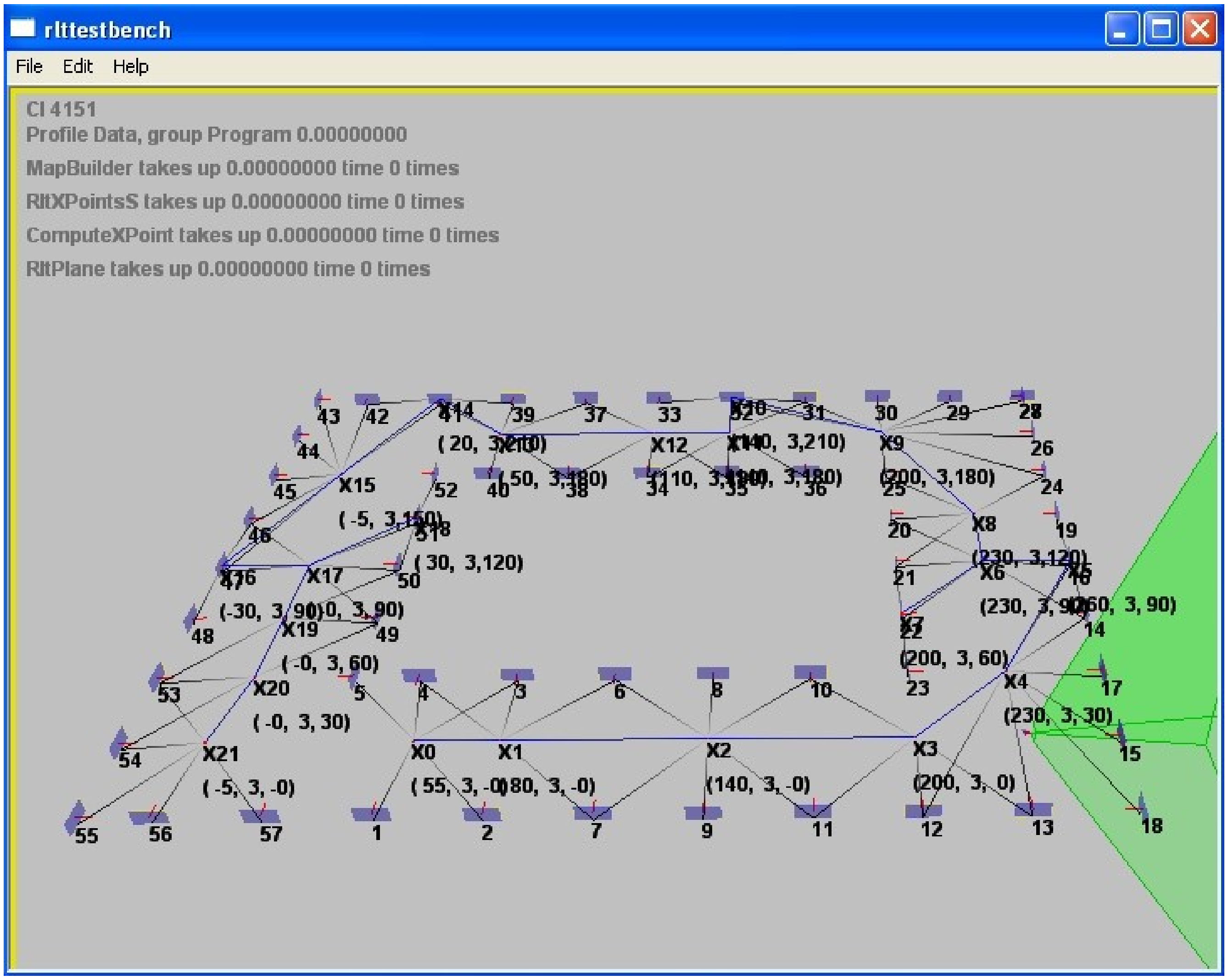
```
Profile Data, group Program 0.00207217
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00188455 time 2 times
RltPlane takes up 0.00152316 time 10 times
```

Figure 72 Next simulation used to debug the normal direction plus backtracking

This leaves one more issue before this phase of simulation is complete. What happens when the robot turns around and heads back to return to the starting position? This leads to the next rule.

Rule 11:

For backtracking ,when a plane cannot be matched to one of the current RltPlanes, the plane is transformed using the current position and then compared against the global map to see if it matches against any preexisting RltPlanes. If it does match, the plane is simply linked to the matching RltPlane. Unfortunately this is the one place where a large location error would effect things, although only temporally since the situation would self correct if the location error corrects.



```
CI 4151
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
```

Figure 73 The full simulation

When the robot turns around it sees the same planes over again. The question is, how to integrate them? The solution is to take the map from the last iteration and to use last iteration's location to place the newly seen plane on the global map. If it matches then place the plane into the matching RltPlane, Rule 11.

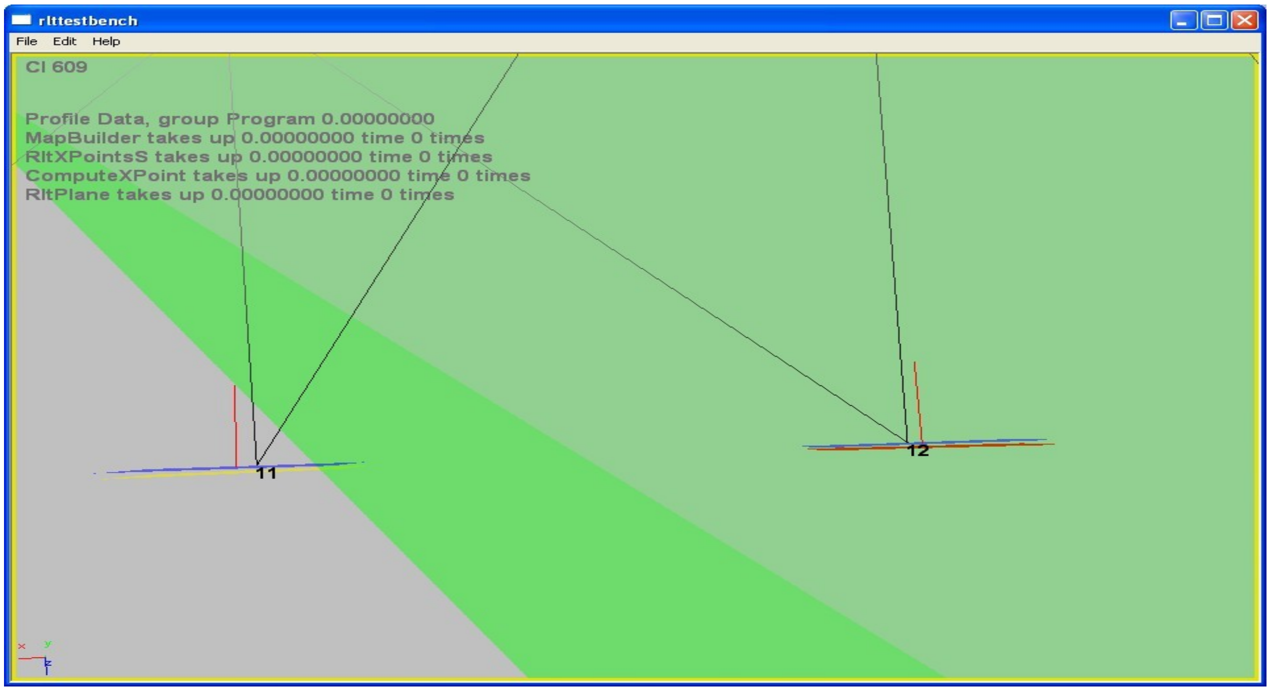
There is one more rule required to generate Figure 73. Rule 12 came about when it appears that the RltXPoints are getting linked together incorrectly. Sometimes a new RltXPoint is created, say X7 (seen in the middle of plane 21,22,23) in Figure 73 due to visibility issues. It indeed should be created, but after the visibility issues are over a new RltXPoint is created. In this case X8 is created and linked to X6 in Figure 73, as X7 is not the best linking RltXPoint. Rule 12 causes a routine to check for the best RltXPoint to link to rather than the current one.

Rule 12:

When linking RltXPoints make it possible to backtrack to possibility select a better link.

One issue to note is that the RltXPoints that are generated and the choice of planes which they are linked to is not guaranteed to be optimum. A poor choice of a linking plane could mean that another RltXPoint is required sooner than it might otherwise be required. The solution to this problem is that it does not matter that it is not optimum. It would be nice to have an optimum selection but it is not necessary. This is very similar to a BSP in the way they select planes to split the world. It is also not optimum but as long as the choices are somewhat good and the BSP is mostly balanced, it works.

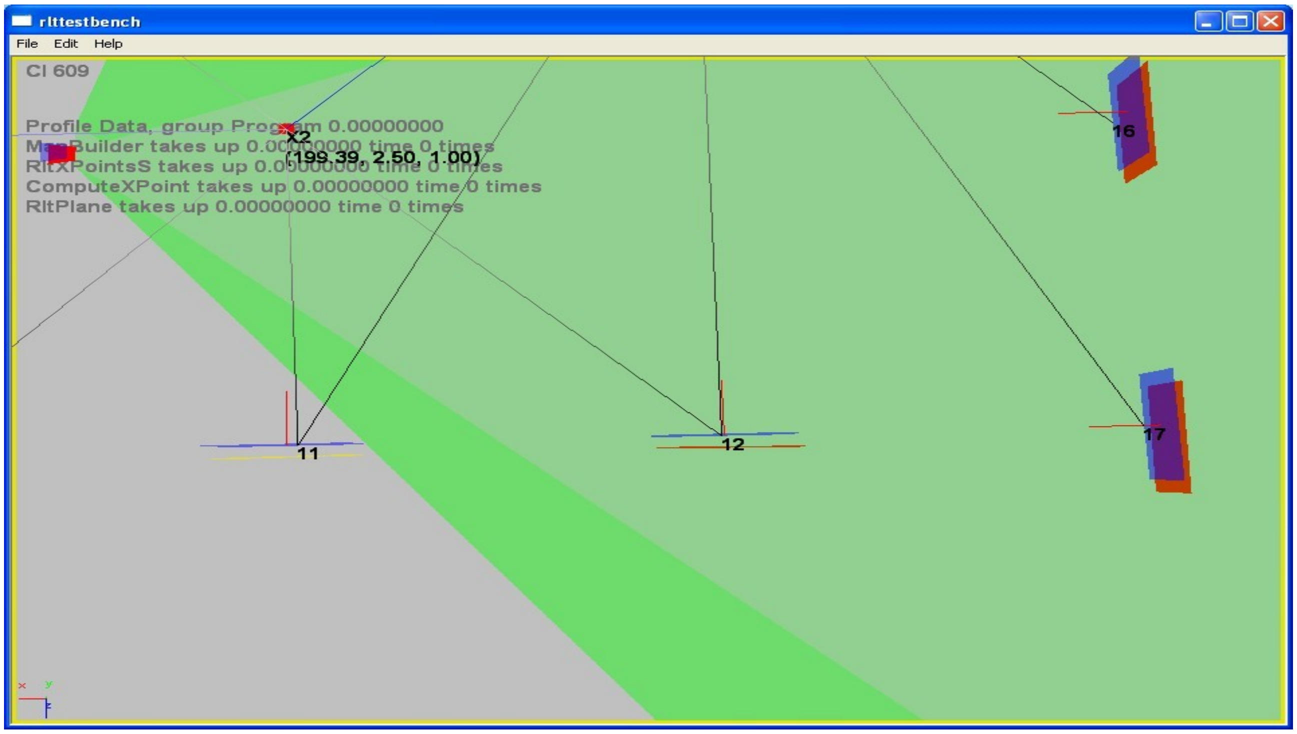
4.3.3 The enduring problem of calculation intervals



CI 609

Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times

Figure 74 Using the minimum RltXPoint interval for calculations



CI 609

Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times

Figure 75 Using the minimum RltPlane interval for calculations

Figure 74 and Figure 75 show an interesting issue. There are several choices to choose which interval is used for the RltPlanes to calculated their relative offset. Figure 74 uses the minimum interval from the RltXPoint which all of the RltPlanes in the RltXPoint are visible together. Figure 75 uses the minimum interval that two RltPlanes are visible to each other regardless of the RltXPoint minimum interval. This is possible because RltPlanes are stored independently of RltXPoints. When using the minimum RltPlane interval in Figure 75 there is more noise in the map compared to Figure 74. The reason being is that the differential signal theory is no longer in effect because the intervals are different. The reason behind using the minimum RltPlane interval is sometimes the RltXPoint is such that the minimum RltXPoint interval covers very little iterations between two RltPlanes and this causes the interval to be too small.

It is possible to reduce the map noise in Figure 75 by doing some error correction when computing the RltXPoints. Also when looking at why the noise causes problems it is found that the map noise only occurs with both rotations and positional noise and furthermore it was found to be due to non zero mean noise since the calculation interval is sometimes small. When the rotation noise is switched from alternating between three values: a small negative angle error, zero angle error and a small positive angle error (both errors of the same magnitude as before) the noise disappeared.

The Relative Plane algorithm is dependent on having a sufficient size of intervals for planes to be compared against each other. If the interval is too small the map can lose accuracy. This is an enduring problem since the interval used to do the calculations is an issue that occurs many times in this algorithm.

4.3.4 Non uniform plane size, the addition of Key Points

Up to now it is assumed that if a plane is seen, then the plane in its entirety is seen, rather than the plane size changing due to growing and shrinking. The plane growing and shrinking problem has been the hardest problem to solve and caused the breakdown of the previous algorithms. The very nature of this algorithm made this problem easy to solve. The solution to this problem is to use key points.

Because planes now are compared in pairs it leads to a very elegant solutions. All planes are assumed to be of their maximum size and if they are only partially visible then the midpoint is wrong and must be adjusted. Each plane has four corner points but since the height is assumed to be static there are two unique points per plane. Estimated midpoints can be generated by offsetting the maximum size of the plane from both of the two points. If the plane is not at its maximum size only one of the two assumed midpoints is correct. When comparing two planes, each has two estimated mid points to give a total of four possible comparisons.

In time, if a plane is growing or shrinking only one of its corner points, now called key points is static. When comparing two planes, even if both are non static, one out of the four comparisons should have a standard deviations lower than the rest. The rule is stated as Rule 13.

Rule 13:

Instead of using the actual plane's midpoint to do a comparison, for each plane generate two estimated midpoints and compare them to the other plane's two estimated midpoints. This generates four possible offset values for every iteration and use the one with the lowest standard deviation.

Rule 13 is a very elegant solution to the problem of shrinking and growing planes. It is much easier to use key points than some sort of global correlation algorithm. It is also necessary since sometimes only a few partial planes are visible. An algorithm that compares a plane's corner points against the average movement may not work since the average movement may be too incorrect to be used. Using key points creates all sorts of other issues as seen in the rules below.

Key point numbering follow a convention as seen by Rule 14. Without this rule when the four estimate midpoints are calculated, there would not be consistency as to which key points are used. This is due to the fact that the plane could be rotating.

Rule 14:

By convention when a plane is mapped to 0^0 the left side is considered key point 1, and the right side is considered key point 2. If no key point is valid after the calculation then it gets mapped to key point 0.

It is possible for a plane comparison pair to be accurate but the key point not be defined. This occurs when two parallel planes are compared that are growing or shrinking at the same time. This is possible in the hallway scenario if the two sides of the hallway are symmetric. If this occurs two out of the four the standard deviations are identically low and the other two are high. When this situation is detected the key point is mapped to 0. In this case it is required to use a different plane comparison. This is Rule 15. If only two partially parallel planes are viewable without another plane to generate the key points the location would not be possible to compute.

Rule 15:

If two parallel symmetric growing or shrinking planes are compared, the offset will be correct but the key point is undefined. This situation can be discovered if it is found that two specific key point pairs have a low standard deviation. In this case map the key point to 0 and have it recalculated.

Key points are again required with partial planes to compare the plane as viewed by the robot to the globally mapped plane. This is due to the fact that the global plane will be its maximum size and the viewed plane might not so the following rule is required Rule 16.

Rule 16:

When calculating the robot's location from a partial plane, that plane is compared to its maximum size global plane. To do the comparison, grow the viewed plane to its maximum size using the key point as a base.

A similar situation occurs when finding out the offset between two RltXPoints and one plane is larger than the other. If this happens Rule 17 is required.

Rule 17:

When linking RltXPoints with planes it is possible that one of them is not the maximum size. Expand the smaller one to the larger size using the key point as the base for the expansion.

When traveling down a hallway, at first, planes will grow from nothing as they are first seen and then they will shrink to nothing when they are passed. If the entire range is used to calculate the key points, it will not work properly due to the fact that in different times, different key points are valid. The solution to this is to split the calculation interval as soon as it is detected that a plane goes from growing to shrinking. This is done in a function that implements Rule 18.

Rule 18:

Make sure the interval used to compare planes uses the interval when both planes are the largest possible size where either plane can not both be growing or shrinking. Since there are multiple intervals, pick the one that has the most iterations.

The key point is stored each time it is calculated. It is important to note that the key point could only be used if it is generated on the same interval as any computation that requires it. If it is generated form a different interval Rule 19 is to be used.

Rule 19:

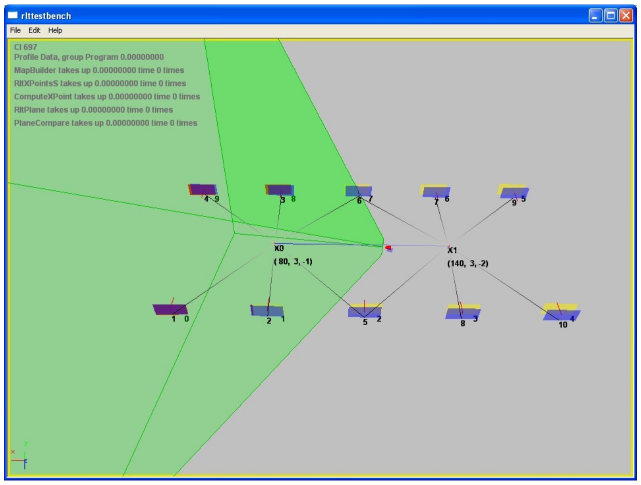
When using a key point for a calculation, make sure it is in the same interval as when the calculation is to happen. If not recalculate it, forcing it to be computed in a given interval.

There is one further issue with Rule 18. What happens if there is some noise and there is one iteration where there is a lot of noise and the plane is much bigger than it should be. This is going to cause errors in the map. The solution is to have a filter on the plane size to regulate the maximum plane size. If the plane size is too large due to noise, simply remove the iteration where that occurred from the calculation. This is not implemented due to this not occurring in the simulation and not too relevant to the algorithm itself. This is stated as Rule 20.

Rule 20:

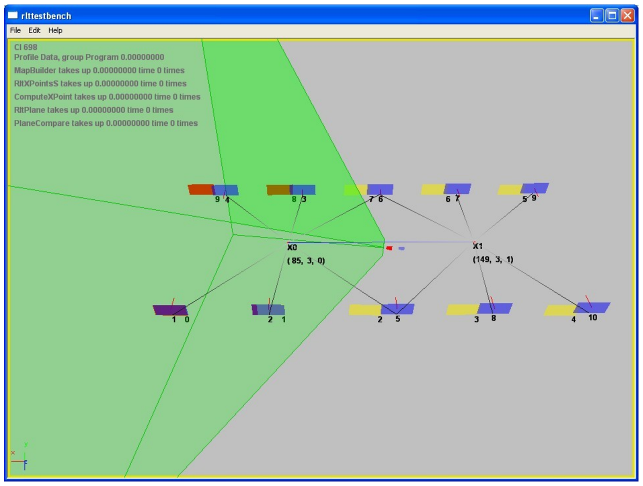
Have a filter that looks at the maximum plane size over time and remove any iteration where the plane size is too large due to noise. (not implemented)

A further note in relation to 4.3.3 The enduring problem of calculation intervals. Key points force the intervals to be split and that the key points must be recalculated if they are in a different interval. The interval splitting causes further issues that are seen and resolved in the following sections.



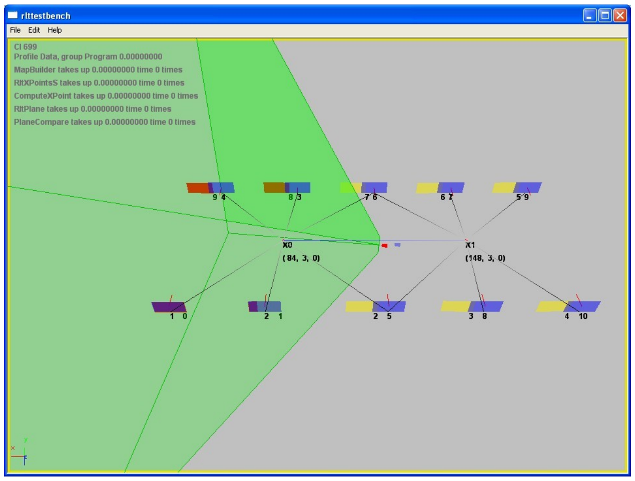
```
CI 697
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 76 Bad key points 1



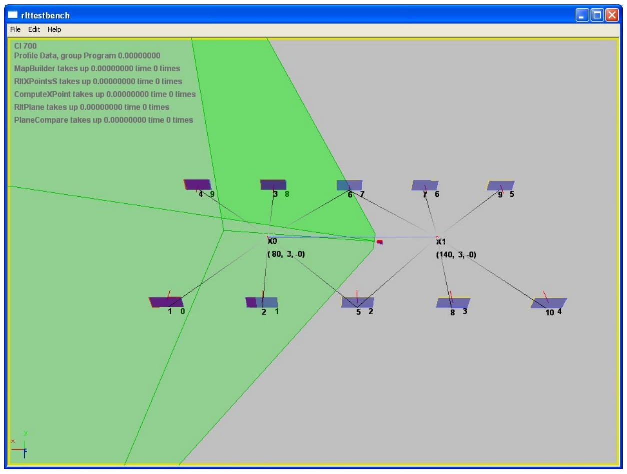
```
CI 698
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 77 Bad key points 2



```
CI 699
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 78 Bad key points 3



```
CI 700
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 79 Bad key points 4

There is unfortunately one case where the key points fail and but it is not necessary to have a solution. The sequence shown by Figure 76, Figure 77, Figure 78, and Figure 79 show the robot traveling and a large error occurring in the map. The blue plane is the mapped plane, yellow planes are the global position of a plane in the current iteration (with a bit of noise so slightly offset from the blue planes). If the plane is visible with the frustum it is red instead of yellow. Notice how the error propagates in the RltXPoints that are linked from the first one.

The error occurs when the robot is rotating and the plane in the bottom, second from the left is just coming into view. The actual problem had to do with using an incorrect interval to do the plane comparisons which reduced the accuracy, but posed an interesting question why the plane relative offset has such a large miscalculation. This simulation has a small amount of noise, and with the small interval there is enough noise so that the key point with the smallest standard deviation is the incorrect one and causes the error. Notice that the error self corrected after the fourth iteration as enough data is available to correct the key points.

This is an interesting problem because it is not possible to guarantee to have the correct key point when a plane is first seen. The good news is that the problem is self correcting in a small amount of time and does not cause any cumulative error. The only potential for propagating errors is when at an instant of an incorrect key points (and current locations) a global match is to occur and it fails. This might not be possible to occur as the above case only happened due to an incorrect interval. The key point failure should only occur in a newly explored area that should not have any global matches.

Despite this, there is a possible solution. There is a mechanism that is described later on (it was not conceived at this stage of development) that can use the future knowledge of a key point to correct this error. If a key point is noted to change on a plane when it should not, a hint can be sent back to assign the correct key point and then the past incorrect iterations can be rerun. It is not known if it is theoretically possible for lasting damage to the map to occur due to this issue so this is not implemented.

Using key points solved a problem that could not be solved properly with many previous attempts at a SLAM algorithm using planes as the input data. The main drawback is an increase in the processing time as the key point require four times the calculation than just using midpoints. Also key points must be generated at different points in times due to interval mismatches. The key point solution is elegant but it does come with issues such as processing time and splitting up intervals but these would have to resolved regardless of the type of algorithm used.

4.3.5 Closing the loop and deciding the current RltXPoint

Without any extra code when the robot enters a location where it has been before the global matching mechanism works and the loop is effectively closed. There is strange case that is created due to the global matching.

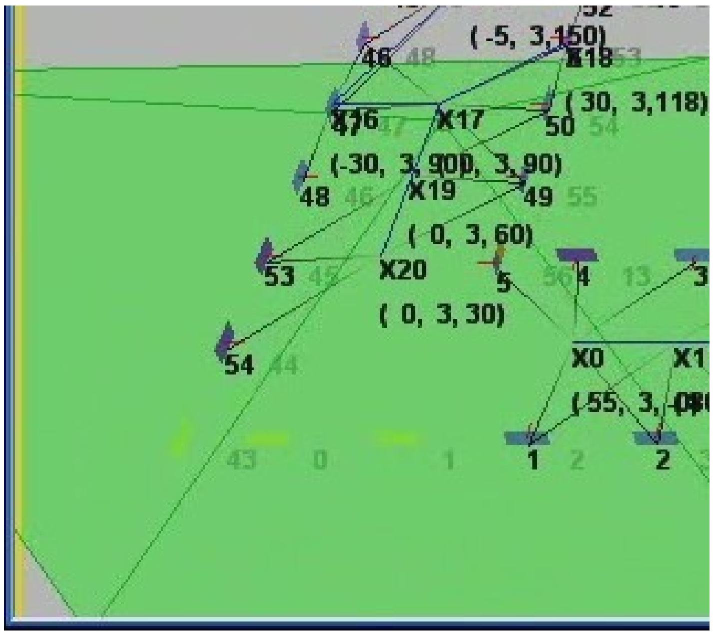


Figure 80 Plane 5 not calculated at full size 1

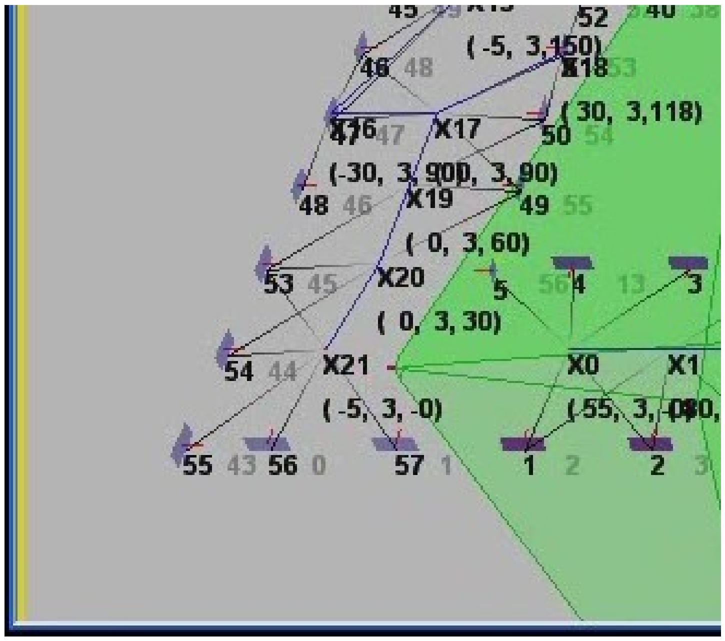


Figure 81 Plane 5 not calculated at full size 2

Looking at Figure 80 and Figure 81 something does not quite work with the global matching. Looking at plane 5, notice that it does not fill up. Looking at Figure 80 half of it is red signifying that a full plane is seen but only half of it is mapped. Plane 5 is being globally matched so X20 never gets a link with it. X20 never does any processing with plane 5, and even though X0 has plane 5 when it is full size it does not have any other planes that are in the same interval. Without another plane visible when plane 5 is at its maximum size, the maximum size is never used for processing.

The easiest solution would be when discovering a globally matched plane to first see if it could be added to the current RltXPoint. This solved the plane 5 issue but created another issue. Once in the simulation, a plane appeared during a rotation for a few iteration and it was not added to the current RltXPoint since the global matching picked it up. After this change in matching order is implemented, that plane did match to the current RltXPoint. This caused the current RltXPoint to have poor accuracy due to the now decreased calculation interval.

Instead, the solution is when a global match occurs to create a new RltXPoint with this global match which would have a new interval to work with. This solved the problem but created a new one in Figure 82 of having many extra RltXPoints, from 21 without the extra code to 30. To reduce the quantity of new RltXPoints, a new RltXPoint is not created if the new global matched plane already has at few RltXPoint references. This is stated in Rule 21

Rule 21:

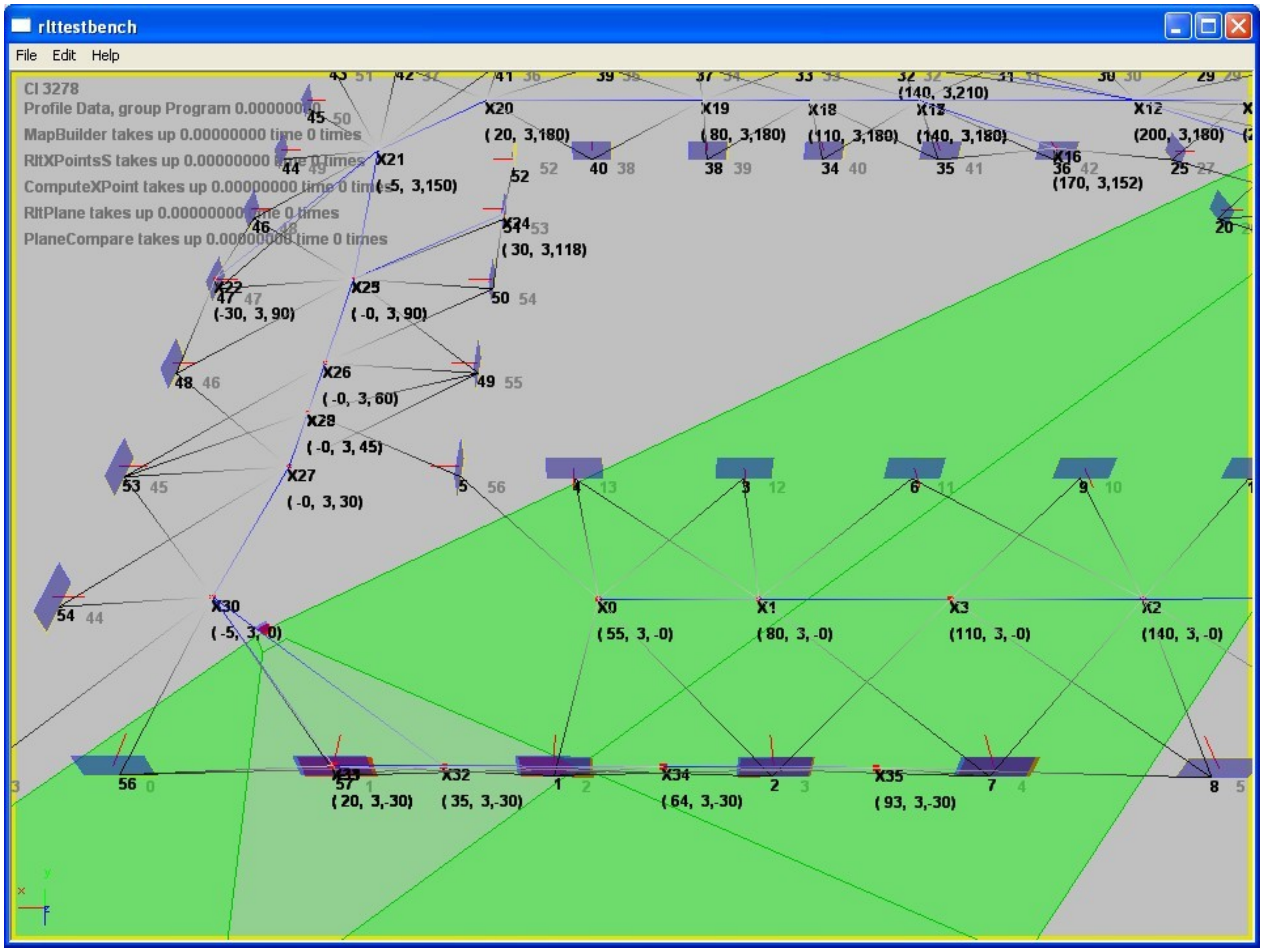
When encountering a plane that is globally matched, create a new RltXPoint with it rather than adding it to the current RltXPoint. If the plane has already a few RltXPoint references just ignore it as it should be mapped correctly already. (This rule is somewhat obsolete due to the roll back capability that removes RltPlanes if they do not have a sufficiently large interval to be used for a comparison, seen later in this section)

Much later in the development another solution is developed that prevents a globally matched plane from ruining the comparison intervals of a RltXPoint but for now this worked.

One further issue to look at, is when backtracking, all the matching uses the global plane matching routine. Although the routine could be optimized, it has to check a plane against every plane in the global map, and it is no where near as quick as having the correct current RltXPoint. Due to this, mainly for processing efficiently there is Rule 22. It states that the current RltXPoint should be the one that is the most relevant. When this is added, it improved the total algorithm processing time by about 5% when the robot visits a previously viewed area.

Rule 22:

Every time a plane is successfully added to a RltPlane, increment a counter for every RltXPoint that is linked to that RltPlane. At the end of an iteration the RltXPoint with the highest count is considered the current one and gets priority over matching planes.



```
CI 3278
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 82 Too many RltXPoints

4.3.6 Motion Model, when only one partial plane is visible

When traveling in a hallway, the robot can reach the end of the hallway and only have a single plane partially visible. The robot then continues on its journey by turning into a new path. How would all of the previously given rules operate in this occurrence? Well, they would not be able to.

Having only one partial plane available means that there would not be enough information to generate any key points. A plane that is only partially visible cannot be used to generate a location without knowing which side has the key point. If before the end of hallway is reached, the full size of the plane is known, the algorithm will recover after the turn when other planes become visible. But if the plane expanded as the robot turns, the algorithm would not have any ability to note the size increase. This would cause a mapping error.

Arriving at the end of the hallway and having only one plane visible can be a common occurrence and it is useful to be able to solve this problem. One solution might be to use the change of distance and change in angle in the one plane to compute the traveled distance. This unfortunately would lead to relying on using the previous location to calculate the current location which is not ideal. There is a better solution that is similar.

The plane at the end of the hallway would correctly be placed on the map earlier when compared to planes that are no longer visible. The problem therefore is not the placement of the plane. The only unknown information is the planes maximum size. So the problem is not the robots position, rather how to track how the plane expands as the robot moves, Rule 23

Rule 23:

When only one partial plane is visible, the only relevant information necessary to be solved is if the plane size expands as previously hidden parts become visible.

How could the robot track distance given only one plane which does not have any bounds? The only information that is known is the distance of the robot to the plane and any change in its angle. This leads to the condition that the robot can only be tracked if it restricted to x movement. Any side to side movement cannot be solved since this movement will not cause any change in observations given the planes unbound size. Using the point to plane distance equations, x distance can always be tracked accurately. Figuring out the current robot angle is simple as it only requires a comparison of the previous angle to the current angle. If the robot goes forwards, turns a bit, goes forward again, z distance is generated which is solved by looking at the change of angle compared to the x distance. This is Rule 24

Rule 24:

In order to calculate movement when only one partial plane is visible a motion model is to be used. The model states that the robot is only capable of x movement and no z movement. It is still possible to rotate in any dimension (although the algorithm currently only supports y angle).

At a certain point as the robot becomes oriented parallel to the plane, the z distance which is tracked using the robots angle becomes prone to large errors. So care must be taken to restrict z movement as the plane becomes parallel with the robot. This is Rule 25.

Rule 25:

The motion model is only valid as long as the plane is not parallel to the robot. As the plane becomes closer to being parallel, noise starts to affect the reading. As this occurs restrict the movement calculated to the x dimension only.

So now the question is how to integrate in the motion model with the rest of the algorithm. It turns out, this is not that difficult. Using the previous architecture it would make sense to use some form of the RltXPoint class to represent the motion model, and to have it fully integrated such that any of the functionality that uses RltXPoint would not need any code changes. This is exactly how it is accomplished, as the RltXPoint class becomes the super class for the RltXMMPoint class.

The first thing that is necessary to integrate the motion model is a way to detect when a motion model is required. The mechanism is stated by Rule 26. When a motion model is detected, a new RltMMPoint is created. It is given the location of the last known good location Rule 27. This does allow for some error as this location could contain noise. The RltXMMPoint calculates the estimated location with Rule 24. That information is used to expand the plane if necessary as stated by Rule 23.

Rule 26:

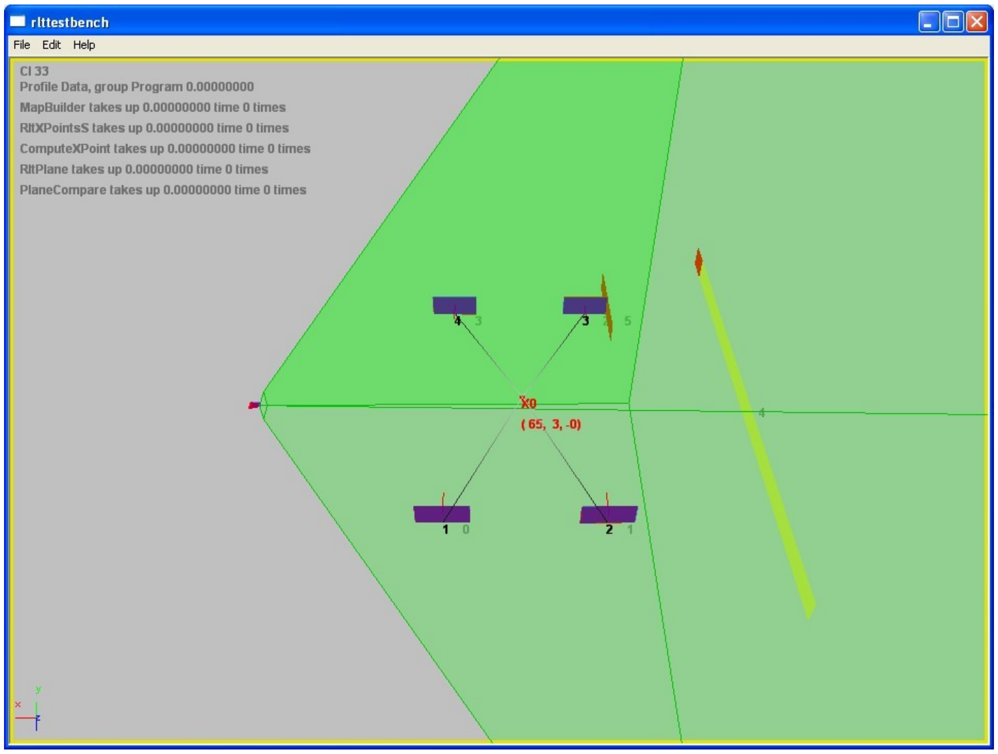
A motion model is detected when it is no longer possible to get the current location from the global map. After a few iterations of not having a location the motion model turns on by creating an instance of RltXMMPoint. The motion model ends whenever a second plane is seen.

Rule 27:

The motion model is fixed at the last known global position. This should be safe as at this point any computation that is done in the previous RltXPoint cannot change since it will receive no new information.

In order to implement the motion model a new simulation map is created. This is shown in Figure 83. The simulation is setup just to test the one partial plane at the end of hallway scenario. This simulation turned out to reveal quite a few situations are not anticipated earlier and led to the refinement of some of the rules in relation to the key points.

The first situation seen is similar to the one observed in Figure 76, Figure 77, Figure 78, and Figure 79. Figure 84, Figure 85, and Figure 86 show a situation where in a small instance in time due to noise the key point on the large plane is miscalculated. Since the plane is very large, it causes a very a large error in mapping. However like the previous time, the error only occurs over a few iteraitons and self corrects when enough readings occur.



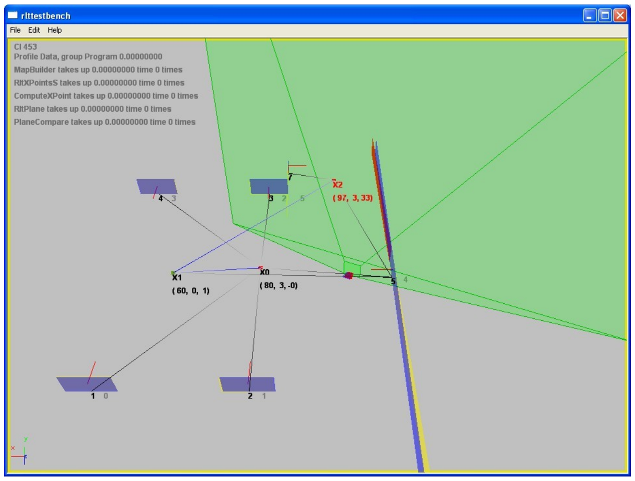
```
CI 33
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 83 The first motion model test

Figure 87 shows an interesting problem due to the way intervals are selected and objects are compared to each other. When calculating X3 plane 3,5,6 are compared to each other. When comparing the two planes, the interval that is chosen for the comparison is hopefully the most accurate one. In this case, the interval that is chosen is the one before the robot did the rotation since there are more observations. At this point plane 5 is its full size. When plane 5 and 6 are compared against each other plane 5 is not the same size as it is in the previous calculation. When linking the planes together an error occurs due to the size mismatch. Rule 28 resolves this issue.

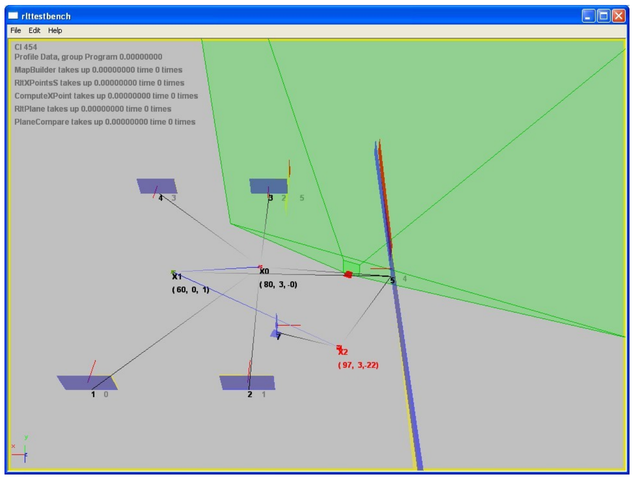
Rule 28:

When calculating an RltXPoint, when chaining the results together say plane 1,2, and 3, the size of plane 2 in the 1 vs 2 comparison must be looked at in the 2 vs 3 comparison in case there is a size mismatch. If there is a difference, the offset of the 2 vs 3 comparison has to be adjusted



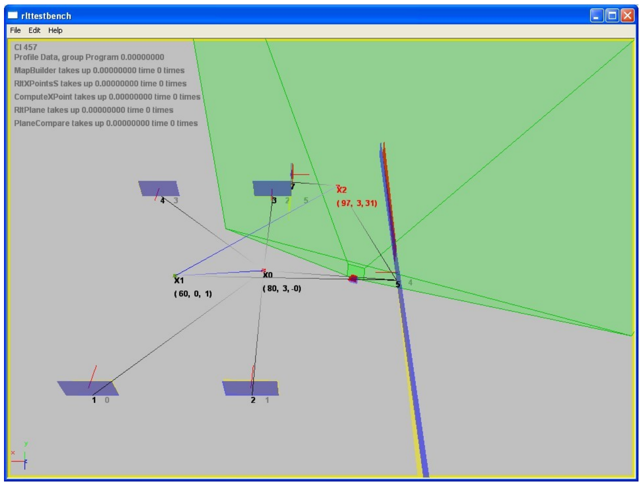
```
CI 453
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 84 Key point failure 1



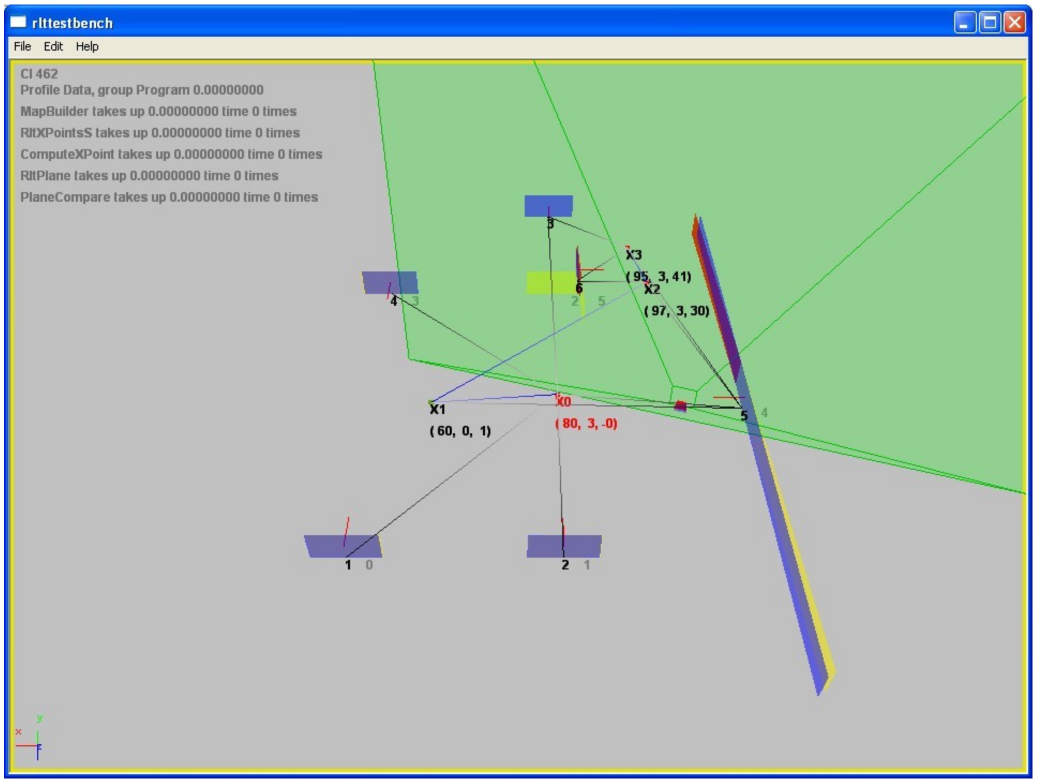
```
CI 454
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 85 Key point failure 2



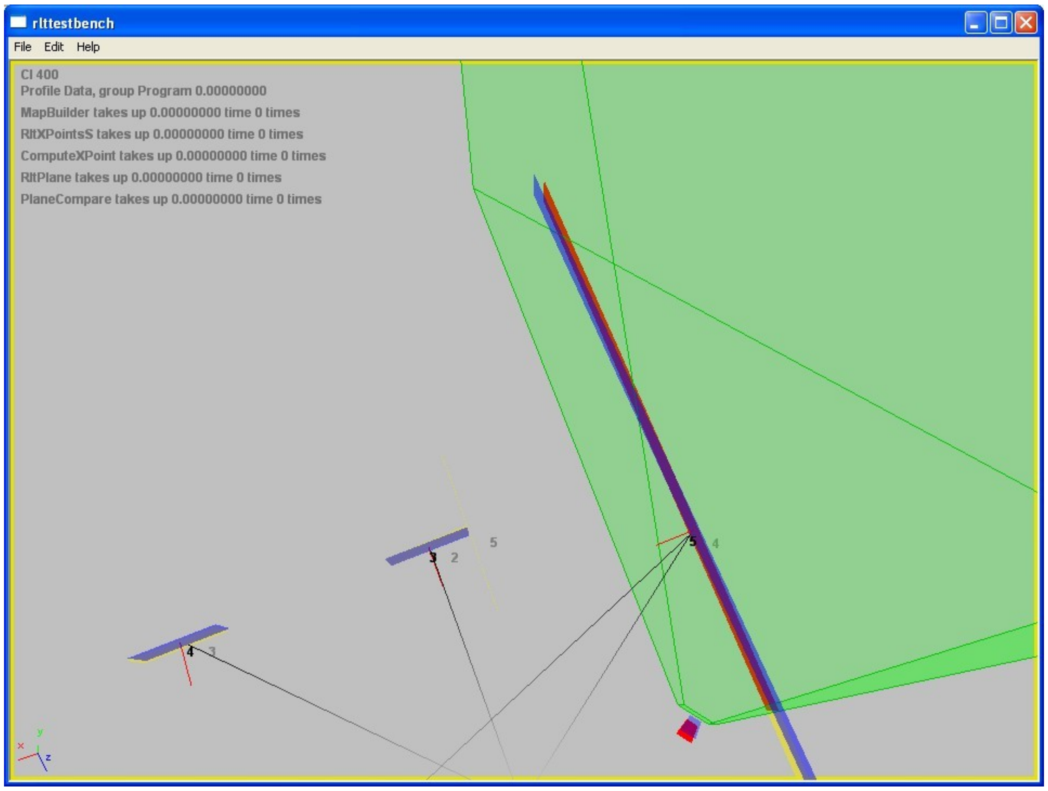
```
CI 457
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 86 Key point failure 3



CI 462
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times

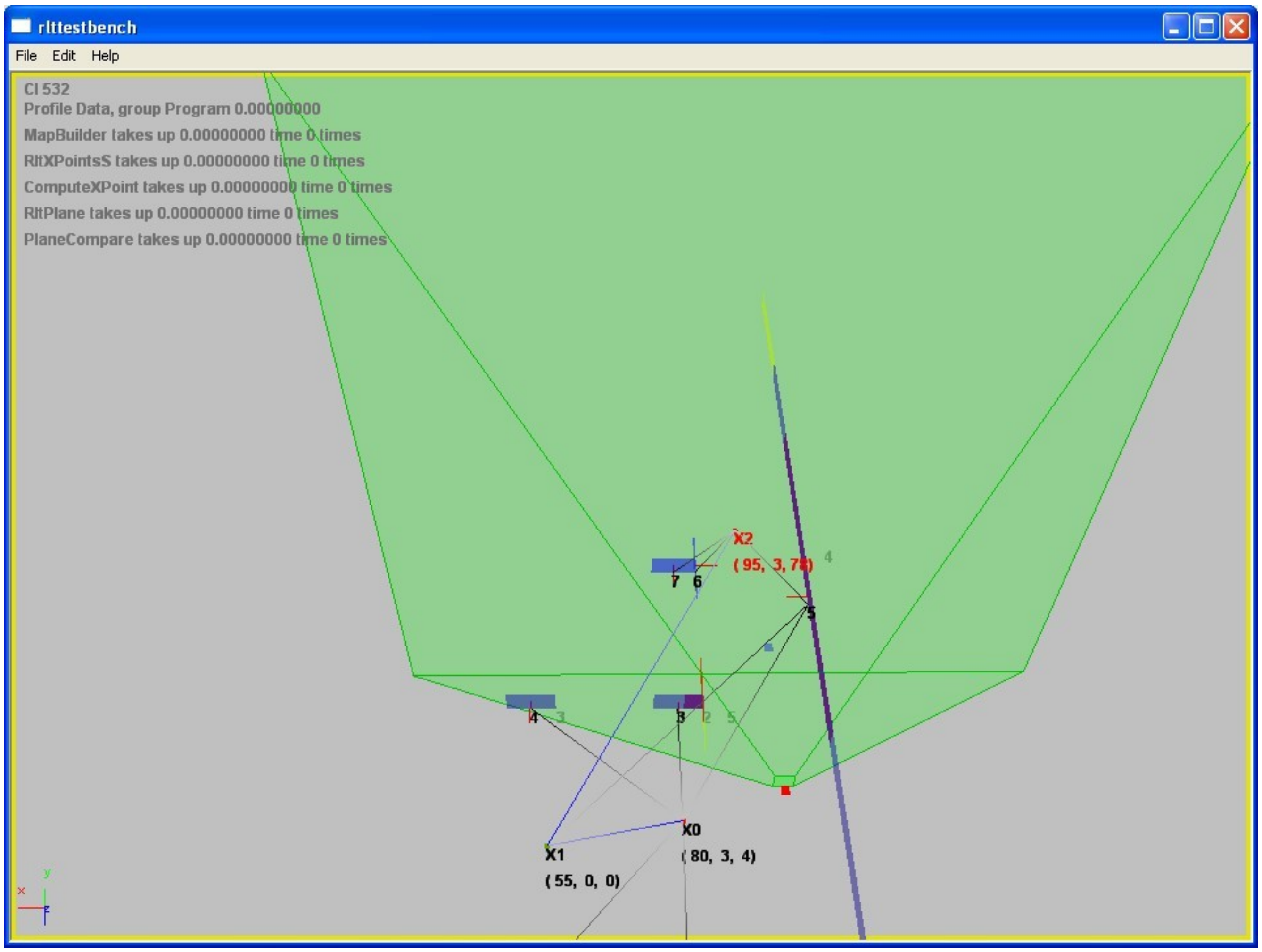
Figure 87 Error due to plane size mismatch when calculating a RltXPoint



CI 400
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times

Figure 88 Plane slightly too big for noise

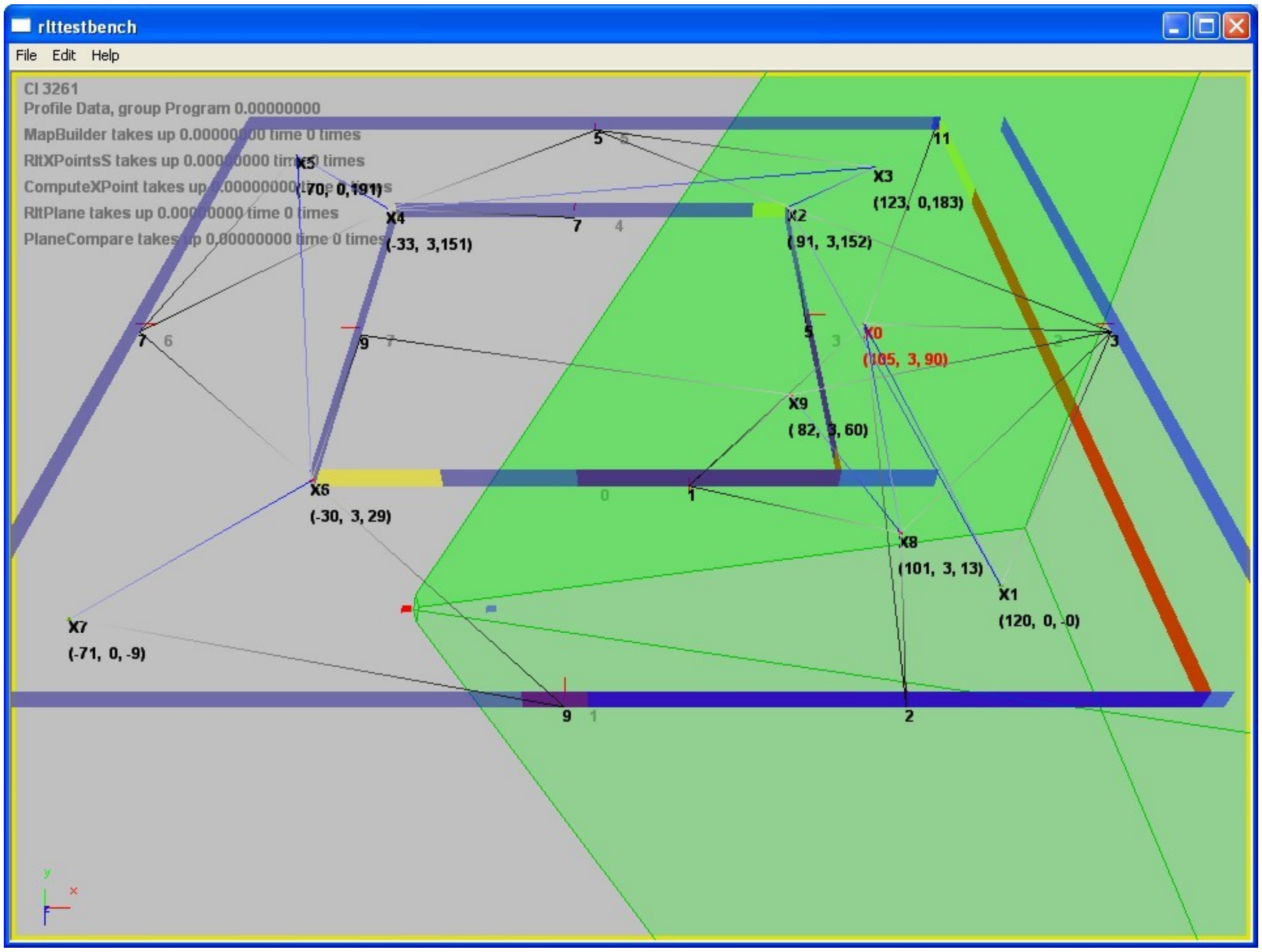
Figure 88 presents and interesting issue. Due to noise, when rotating it is possible for the plane to be expanded beyond its maximum size. It might be possible to solve this by using a filter, however the motion model is always going to be prone to some errors so it is left as is.



```
CI 532
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 89 Frustum failure

Figure 89 shows an interesting problem that is not physically realistic. The viewing frustum longest dimension is the distance between the origin to any corner point on the far plane. This distance is larger than the distance of the frustum looking straight ahead. When a robot is rotating, it first sees the big plane expand and then it shrinks a bit. This is observed in the picture as the large plane is mapped larger than it is currently viewed and this creates an error. This is not a realistic situation as a real persons' viewing distance is not longer on the edge of the viewing area than the center. This problem is solved by expanding the frustum size so this does not occur.



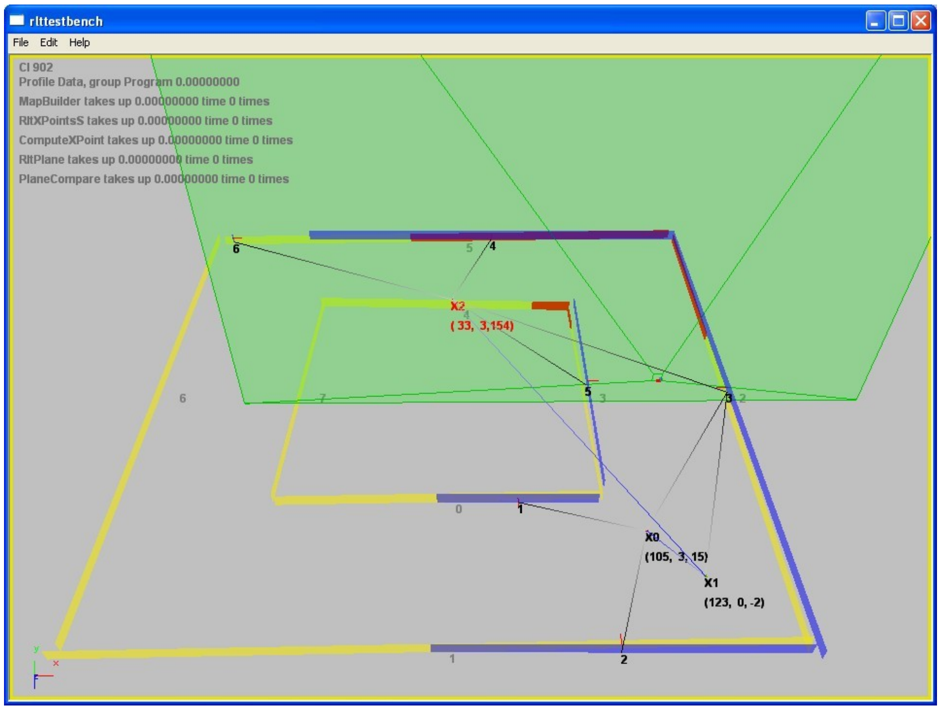
```
CI 3261
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 90 Errors caused by three issues .

Figure 90 contains one of the first simulation of the full motion model test. It worked until the loop was closed. It has three interesting issues that previously are not seen. The first is that plane 9 and plane 2 are the same plane, so should they be merged? The reason there are two planes is that the robot started pretty much where it is located on the map presently so it only saw a smaller version of plane 2. Plane 9 which is generated on the return trip is the larger correct version. To merge them the global map would have to be used and each plane compared to each other one to see if any are the same. That operation is not too difficult but there is the issue should they be merged? When there is no noise it would appear that they should however when there is noise, that opens up questions on how it will effect the accuracy? Due to the noise issue the plane is not merged however there is some loose connection described later.

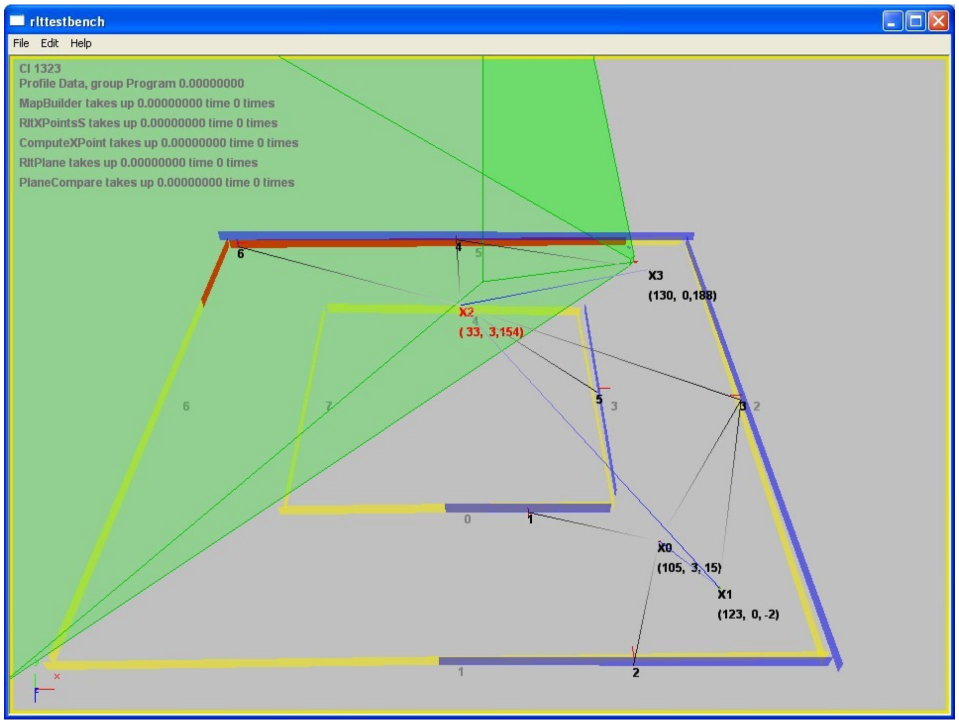
The second issue in Figure 90 is that the size of the first reading of the planes used to align the map with the global version is different as plane 2 expanded on the return trip. The size used in the comparison to align the map to the global map has to be adjusted to compensate for this.

The third issue in Figure 90 has to do with the key points, in that some are not available since they are not required in the first pass but required in the second one. Figure 91, and Figure 92 contain a related problem with the key points being incorrect.



```
CI 902
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

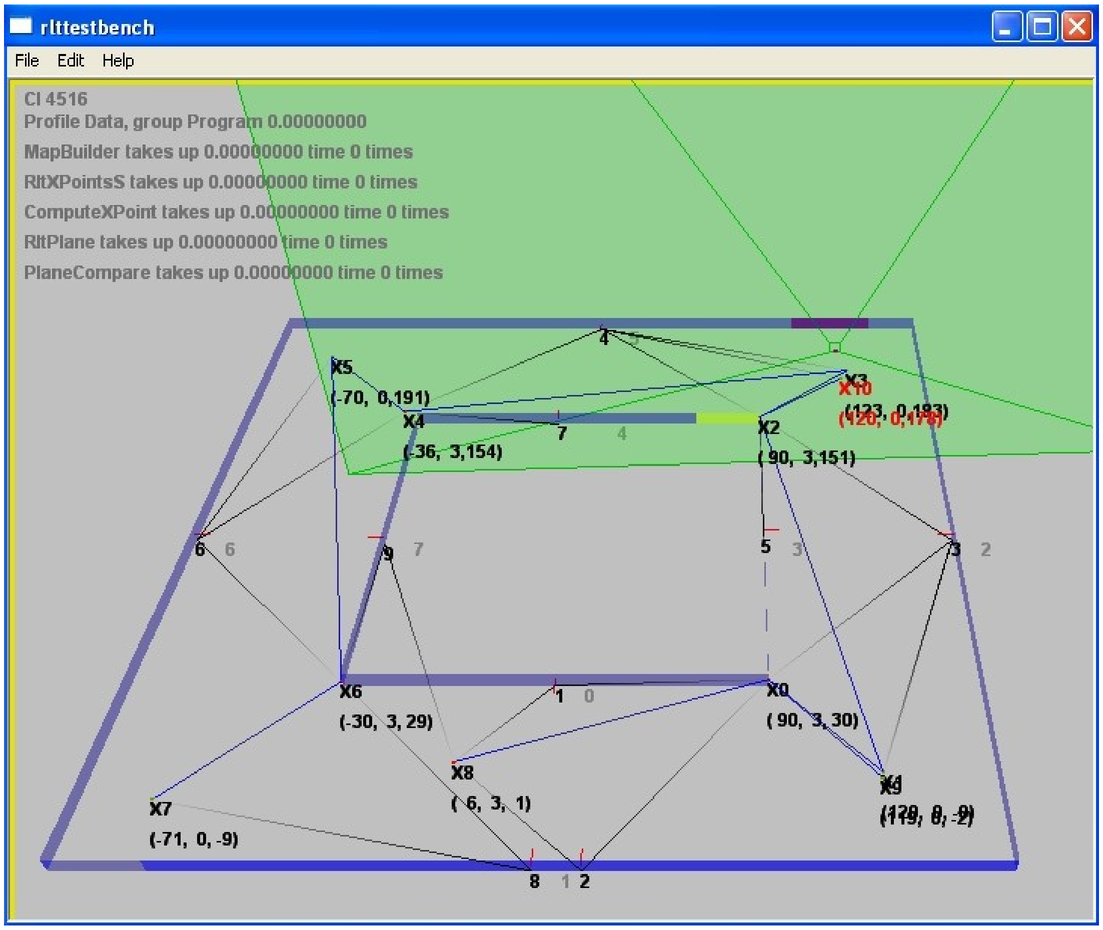
Figure 91 Plane 4 should be bigger



```
CI 1323
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 92 Plane 6 not calculated well

The problem in Figure 90, Figure 91, and Figure 92 led to the realization that is is possible that the key point available is the wrong one since it is generated in an earlier interval. There is also the possibility of a key point not existing. It is at this point that the need for Rule 19 is required, in which key points are recalculated if they are on the incorrect interval.



```
CI 4516
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 93 A finished simulation run of the motion model

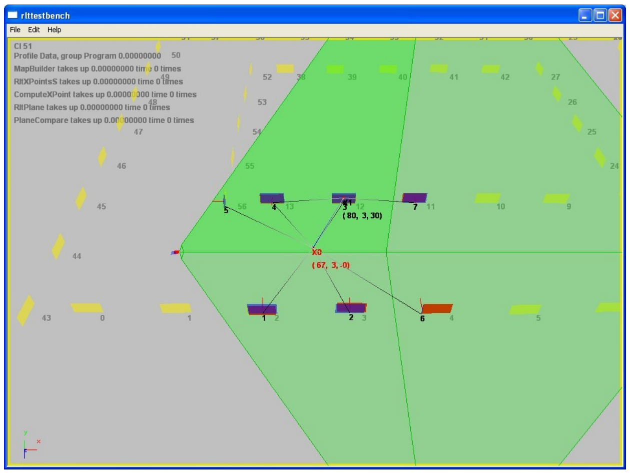
Figure 93 is the completed simulation for testing the motion model. It contains the solution to one final issue. Normally when exiting a motion model a new RltXPoint is created. However when returning to the same place as before why create a new RltXPoint? Rule 29 is created to resolve this issue. Note that when entering a motion model it is always required to create a new RltXMMPoint because it is unknown if the robot is going to travel the same direction as the previous RltXMMPoint.

Rule 29:

When exiting the motion model, if there are any preexisting RltXPoints that contain all of the viewed planes based on global matching, switch to that RltXPoint. If this RltXPoint does not exist, create a new one.

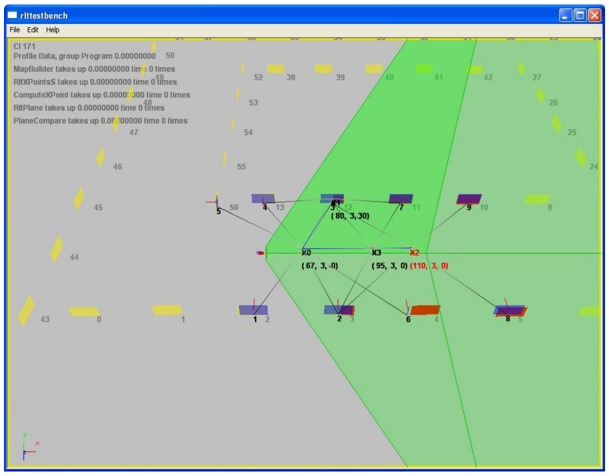
4.3.7 Comparison pairs, going back in time, saving the RltXPoint state, pseudo plane merging.

4.3.7.1 The problem of bad plane comparison pairs



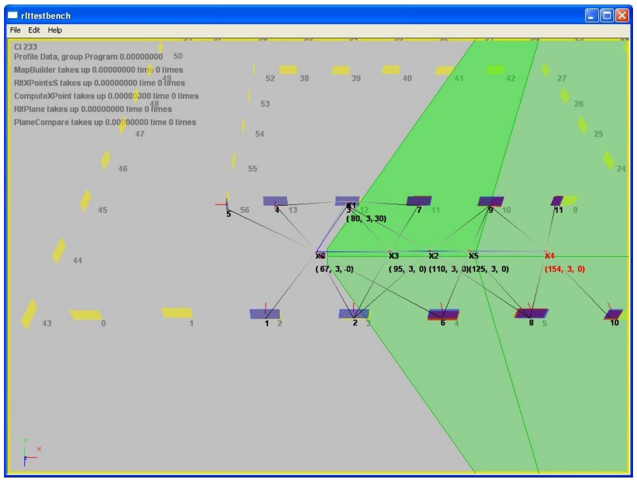
```
CI 51
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 94 Plane 6 should not be in X0



```
CI 171
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 95 Plane 6 should not be in X0

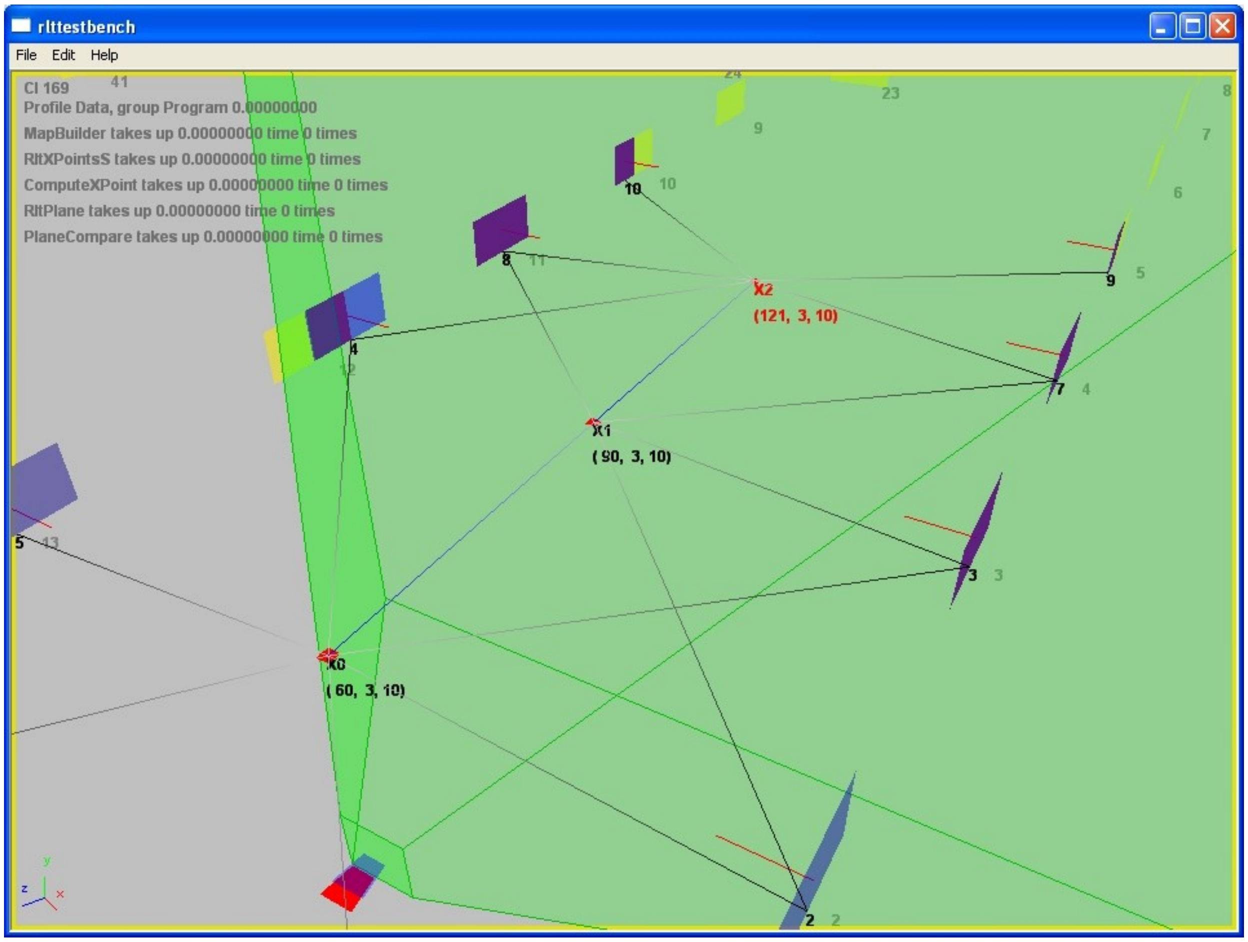


```
CI 233
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 96 Plane 6 now fully mapped

Figure 94, and Figure 95 show a sequence where the bottom right plane (in red) is added to the starting RltXPoint. As it turns out the bottom right plane, plane 6 appears nearly exactly when the top left plane, plane 5 is leaving the view. It only is in the RltXPoint for a very brief time, and to make matters worse it is being compared to plane 5. Since plane 6 and plane 5 are only visible together for a very short time, the calculation uses a very small interval and only when plane 6 is of a small size. That is why plane 6 is not really visible in Figure 94, and Figure 95 although it is mapped as seen by the link. The problem is compounded in that plane 6 is also used to link the RltXPoints together. Plane 6 is later visible in Figure 96 after a restriction is placed on planes with small intervals cannot be used to link together RltXPoints. But then in Figure 96 there are now many more RltXPoints than necessary.

Currently planes are compared by the order in which they are first observed. This would normally work but in this case these planes 1 to 5 are all viewed together in the start of the simulation so the order in the RltXPoint is the same order that they are stored in the simulation. There is a possible solution in Rule 30.



```
CI 169
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 97 Bad key point calculation due to plane comparison used

A similar problem is seen in Figure 97. In this figure, plane 4 is misplaced. This is due to plane 4 being compared to plane 10 and the two are coplanar. Plane 4 is shrinking as plane 10 is growing. This causes two of the key points comparisons to be correct. Key point 1 of plane 4 when being compared to key point 2 of plane 10 returns the correct value. However key point 2 of plane 4 is shrinking at the exactly the same rate that key point 1 of plane 10 is growing which gives a valid result. This is exactly the same issue as Rule 15 of two parallel planes except in this case the offset is not correct. Possible solutions would be to assume the direction of travel which would cause the correct key point combination to be chosen, or doing a distance test and in the case of the parallel one the distance would be the same. If one of the two are shorter as in this case, use the shorter one. A better solution might be to implement Rule 30.

Rule 30:

During a RltXPoint computation the plane pairs should be chosen so that the pairs maximize the total amount of intervals used to compute the RltXPoint. (only partially implemented)

The first attempt to implement Rule 30 is a simple algorithm. First an n by n matrix is created comparing every plane in an RltXPoint to each other and if their key point is valid then record the size of the interval used in the plane comparison. If the key point is not valid, then record a negative one so that combination is not used. The matrix construction is O(n^2). Then a O(n^2) search starts at each plane and goes through the matrix in every possible path and saves the path if the total interval size exceeds the previous. The path with the largest interval size should be the best. The algorithm is implemented and fixed the problem seen in Figure 97. Even though the algorithm is O(n^2) the number of planes in an RltXPoint is usually small and this algorithm only has to be run periodically for every RltXPoint rather than every iteration.

The problem is that this algorithm does not fully solve Rule 30 as the initial guess at the solving algorithm is incorrect. It finds the best path if every plane is only passed through once. The true solution would be to allow a plane to be used for multiple paths. This would mean that each plane can be a comparison base to as many as all the other planes. This is identical to the traveling salesman problem where the salesman can revisit any city. The brute force solution is O(n!) which is impractical [Wiki09d]. Knowing the difficulty of solving the traveling salesman problem Rule 30 is left as only partially implemented since it works well enough as is, and is left as future work.

4.3.7.2 Roll back system

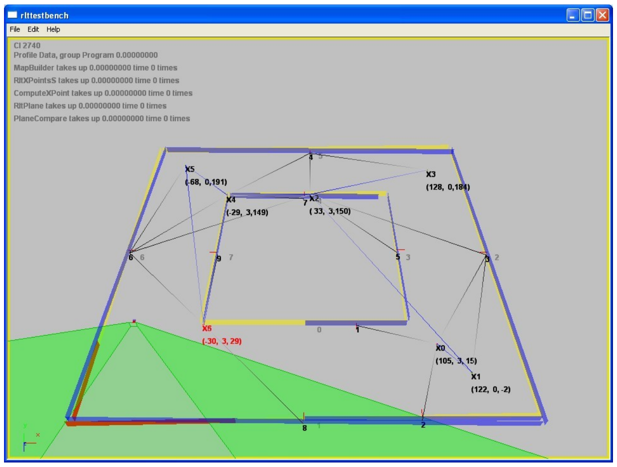

```
CI 2740
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 98 Before planes go missing

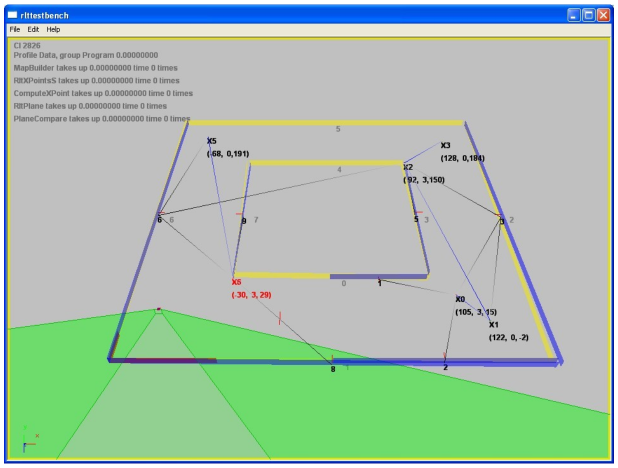

```
CI 2826
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 99 After planes go missing

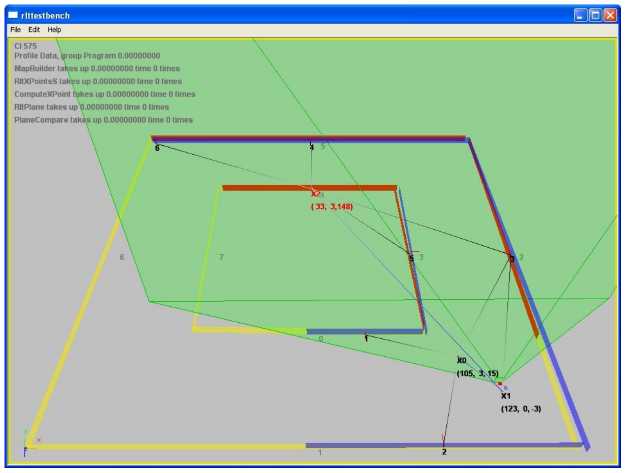

```
CI 575
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 100 Plane 6 first seen

Figure 98 and Figure 99 show an interesting sequence in which some planes simply disappear as the robot moves. The explanation is partially in Figure 100 as the left outside vertical plane, plane 6 is first seen when the robot is in the lower right due to the large frustum. Note that this simulation does not have any occlusion. There is nothing wrong with it having been seen early but it causes a large error later on. The problem with the planes disappearing is due to the limited amount of plane storage. At this point, the plane storage size is 1200. Ideally all planes in an RltXPoint are seen together so even if the plane storage rolls over, all the planes that are relevant to each other roll over together.

When some planes are viewed for a much longer duration, the planes that are necessary to compare to other planes may be lost to a roll over. This is precisely what happened, as without plane 6, the top planes could not be calculated since they are in the same RltXPoint. Also note that there is a question that if a plane is only seen for a very brief time should it be added to an RltXPoint?

It is possible to chose a better order to do plane comparisons stated in Rule 30. It might also be possible to save the best previous plane comparison as seen in the next section. But is there another larger problem? Rule 7 states that in a RltXPoint all planes should be viewable together at the same time. Perhaps the real problem is that plane 6 should not be in the first RltXPoint at all. This is similar to the problem seen in Figure 94 and Figure 95 when its plane 6 comparison interval with plane 5 is too small. The solution is to use Rule 31.

Rule 31:

All planes in an RltXPoint should be viewable together for a minimum amount of time.

The problem with implementing Rule 31 is that it could only be implemented in the future. When a plane is first observed it is not possible to estimate what its interval will be in a RltXPoint. To detect this situation Rule 32 can be used. The question is, how to ensure that the RltPlane is no longer in the RltXPoint. Can the RltPlane just be unlinked from the RltXPoint and linked to another RltXPoint or have a new RltXPoint created? Arbitrary unlinking the RltPlane could be dangerous and unpredictable and can lead to many special cases. There is a much more elegant solution available in Rule 33 and then the general Rule 34 which can be applied to any situation if required. Rule 34 is inspired by [Mema09].

Details of how to implement Rule 33 are best left to the software architecture chapter. Suffice it so say when it is found that a RltPlane should not belong to a RltXPoint, a hint is created that tells the algorithm not to add a RltPlane with a certain first plane, at a certain iteration, to a certain RltXPoint.

Rule 32:

After a specific number of iterations, verify that a RltPlane is recorded to have been compared to its comparison plane for a minimum number of iterations. If not, ensure that the RltPlane is no longer in that RltXPoint. If there is another RltXPoint available, first try adding it to that one. If this rule repeats itself again for the same RltPlane then have a new RltXPoint created. If this rule repeats itself one more time them simply do not place the RltPlane into an RltXPoint which effectively removes it from the map.

Rule 33:

The algorithm must be able to roll back time from the current iteration to one in the past. All RltPlanes should be rolled back to the iteration, and any extra structure that is created in the meantime (RltXPoint) that should no longer exist should be removed. The algorithm will also store all inputs for every iteration so that the algorithm is able to roll forward in time.

Rule 34:

If a situation occurs that is undesirable, the algorithm should be able to roll back time to the point where the undesirable situation occurred and be able to send a hint to prevent that situation from occurring.

The only draw backs to rolling back time are the processing time and running out of previous planes inside of RltPlane. The rolling back part is quick but to go forwards again requires the complete computation of all the intervals rolled back and can be somewhat intensive depending how many iterations are rolled back. The other issue is that circular array in RltPlane that store raw plane data has a finite limit of data and that limits how much time could safely be rolled back.

The rolling back and hint mechanism could be used for other purposes to improve the accuracy. For example, in the situation where due to noise the key point is incorrect, it would be possible for the correct one in the future to be sent to the past, in case any global mismatch problems occur.

4.3.7.3 Save system

The other problem of Figure 98 and Figure 99 is the fact that the quantity of untransformed planes stored in an RltPlane is a finite quantity. Previously in testing, any problems were solved by increasing the storage. It would be ideal to have the best copy of the computed RltXPoint available.

Rule 35:

Save the best available computed RltXPoint from a single iteration in case it there is a situation where it is no longer possible to compute the RltXPoint. In that case, simply load the previous saved copy.

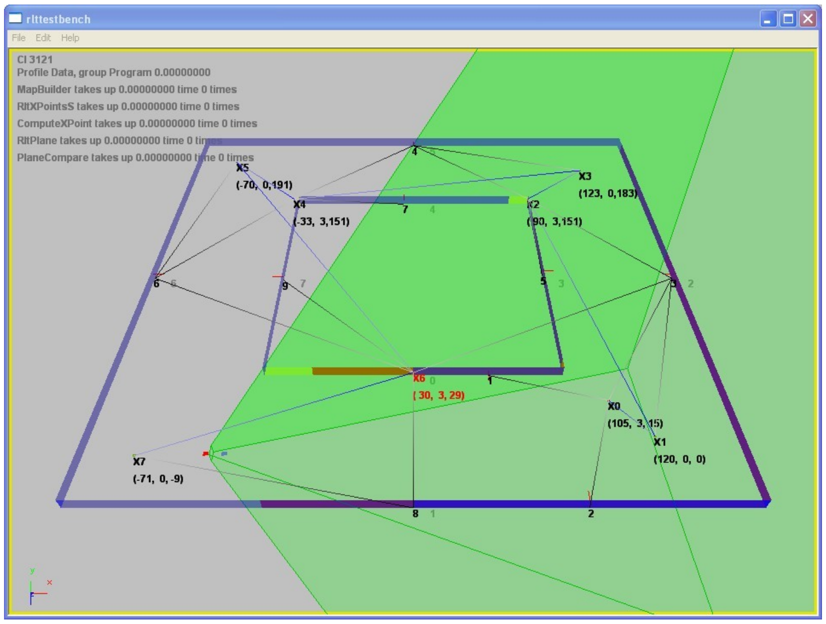
Rule 36:

Save the best available intermediate information for a RltXPoint computation when a plane is compared to another plane. This rule is more complex than Rule 35 since it needs to deal with the fact that there might be size mismatches since the best intermediate information might be generated over different intervals.

Rule 35 saves a complete RltXPoint from a single iteration. This may not be the best available information since any given iteration may have planes at less than the maximum size.

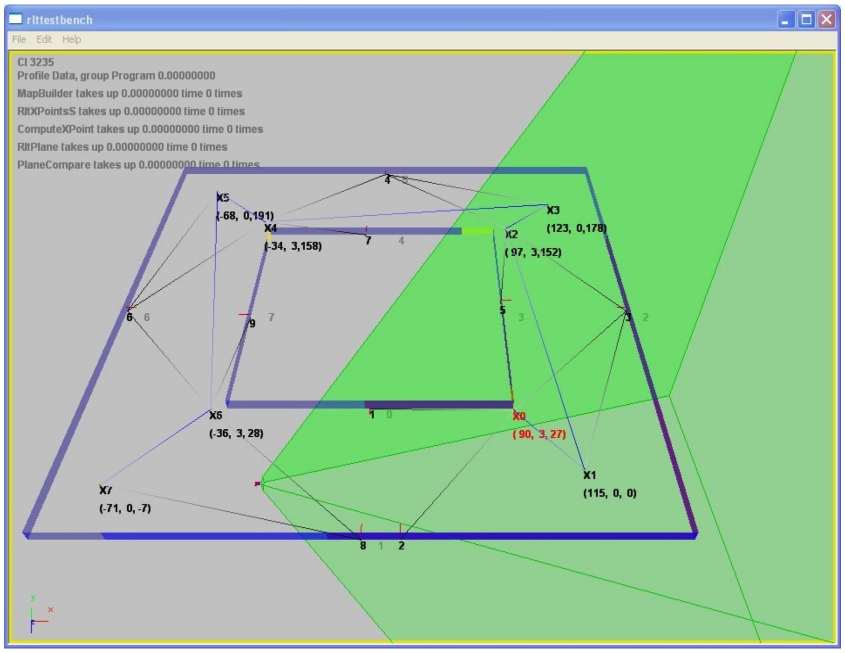
Rule 36 has the same functionality as Rule 35 but it is more complex to implement. It uses intermediate values that could be more accurate but it also requires more storage and processing. Only one of the two rules is required to be implemented. Both are currently implemented.

4.3.7.4 Pseudo plane merging



```
rlttestbench
File  Edit  Help
CI 3121
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 ti
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 101 A problem with intervals and plane 8,2



```
rlttestbench
File  Edit  Help
CI 3235
Profile Data, group Program 0.00000000
MapBuilder takes up 0.00000000 time 0 times
RltXPointsS takes up 0.00000000 time 0 times
ComputeXPoint takes up 0.00000000 time 0 times
RltPlane takes up 0.00000000 time 0 times
PlaneCompare takes up 0.00000000 time 0 times
```

Figure 102 The problem with planes 8 and 2 fixed.

Figure 101 shows an issue with plane 1 not being mapped properly and Figure 102 shows the same simulation after the issue is fixed. The problem has to do with the bottom planes, plane 8 and plane 2 are the same plane. When the motion model ends, the algorithm first searches for a previous RltXPoint that contains all of the currently visible planes. Since after the turn, plane 8 is seen and not plane 2, the algorithm incorrectly assigned which RltXPoint is the current one. When this happens all of the untransformed planes are routed into plane 8. While plane 1 is updated in the correct RltXPoint X0, in X0 when comparing plane 1 to plane 2, plane 2 does not exist in the interval where plane 1 is bigger since all of plane 2's updates went to plane 8.

There are several solutions such as merging plane 8 and 2, which is not desired. Rule 30 could be implemented to change the comparison order which would compare plane 1 to plane 3 (the rightmost plane) and that would solve it. Turns out the best solution is a pseudo merge of just altering some of the links. Rule 22 charts which RltXPoint should be the current one. To solve this problem Rule 37 is used. When a loop is closed, it checks every viewed plane against all global matches. For these global matches, if they do not contain a link to the original plane, a link is added that allows the charting. The link is one way so the plane is not added to the RltXPoint which would cause it to be used when comparing planes.

Rule 37:

After a motion model, compare any viewed plane to all of its global matches. If a global matched plane does not contain a link to the RltPlane that means that there is a merging issues and simply link the RltPlane to the RltXPoint, but not the RltXPoint to the RltPlane. This prevents the RltXPoint from using it for processing. Now when charting, all RltXPoints that would contain the plane if it is merged are correctly charted.

4.3.8 Summary

The relative algorithm has some very good properties.

- It can make use of all available plane data.
- It is possible to go back in time and correct errors.
- The processing is O(n) in number of planes and O(n) in number of iterations used for processing. Only when reordering planes, there is a $O(n^2)$ function.
- Since the plane's relative angle is determined before a turn, even if the turn is noisy it will not create any angular errors.
- It can identify and make use of planes that are growing and shrinking.
- Since the algorithm is not dependent on the past location (except for global matching) many issues with algorithm that require very accurate current location do not occur.

The only requirements of this algorithm is that planes that are visible together can be grouped together and that these groups can be linked together with shared planes. If this holds the algorithm should work.

4.4 Class Architecture

4.4.1 Introduction

Figure 103 and Figure 104 show the class listing of the Relative Plane algorithm and the simulation. Each class is described in detail in the following sections.

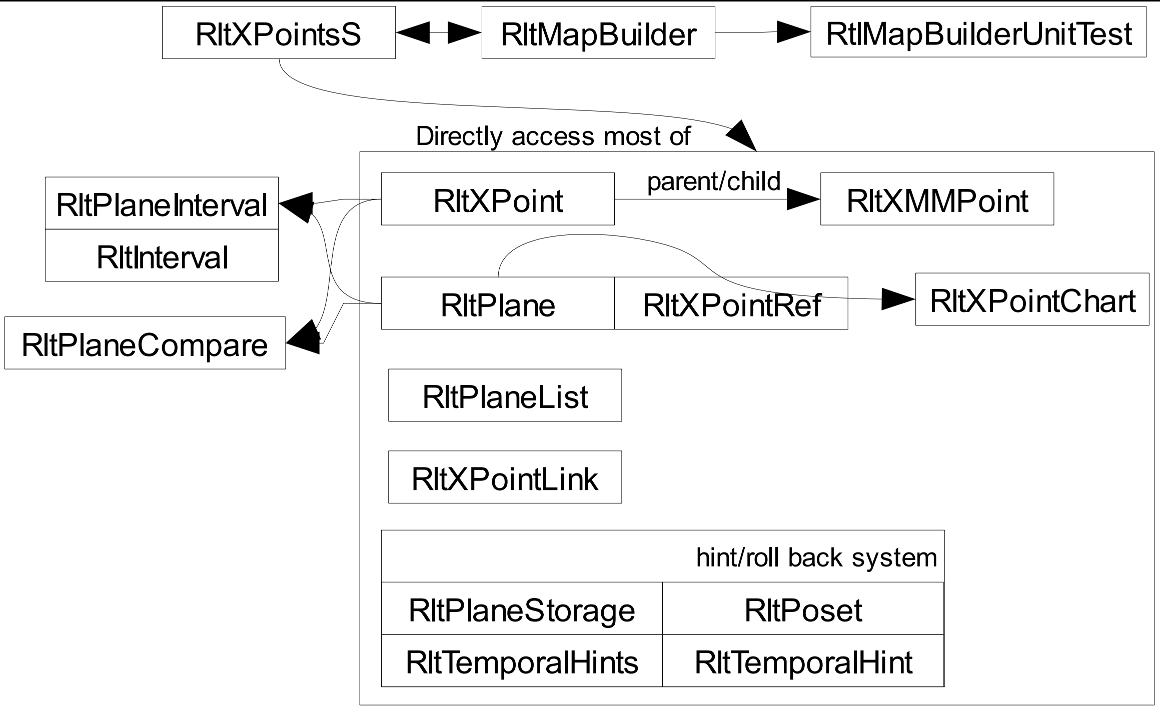


Figure 103 Relative Algorithm class listing

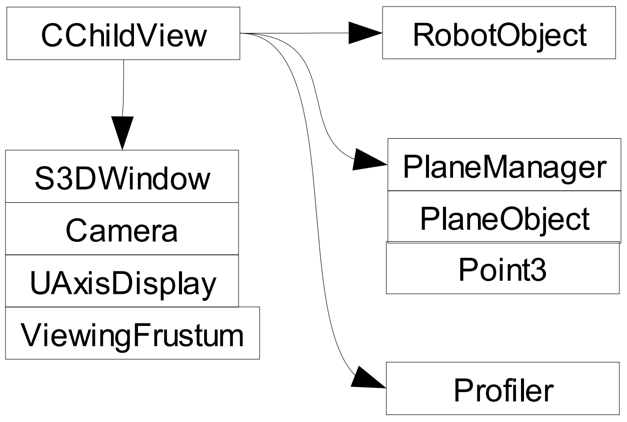


Figure 104 Simulation class listing

4.4.2 Simulation Classes

4.4.2.1 CChildView

The simulation is programmed in C++ under Windows using the Microsoft Foundation Class (MFC) API. MFC is a wrapper around the Windows API which simplifies the development of a project. For this project, the framework is custom, and MFC is only used for convenience as it is faster to start a new project this way. In addition to the CChildView class, there are several other classes that are created by the new project wizard which are not listed but they do not contain any project specific functionality.

CChildView is important to the simulation as it is the class that drives it. The class is the parent of the simulation and contains the thread which calls the simulation and then the rendering functionality.

The class receives user input which is used to control the simulation such as stopping, starting, and stepping through a single iteration. It also passes any non consumed user input to the rendering window to let it receive user controls.

4.4.2.2 S3DWindow and the rendering subsystem

The S3DWindow class contains all of the 3D rendering framework. It creates the Direct3D objects which are used for rendering, and has a rendering loop which calls the rendering function of any class wanting to render to the screen. It also accepts user controls to change the view of the window, and to toggle the display of the profiler.

The Camera class contains functionality to adjust the viewpoint given user input.

UAxisDisplay is used to display the axes in the corner of the screen which show the current view orientation.

ViewingFrustum is used to cull polygons that are not visible to the robot. For rendering the viewing frustum is built into the Direct3D pipeline and handled automatically. Since the simulation wants to simulate the viewpoint of the robot it needs to do its own viewpoint culling.

4.4.2.3 Robot Object

The robot object performs the actual simulation. First it moves the robot along a predesignated path. It then calls PlaneManager and enumerates all of the planes in the map. It uses the frustum to cull planes so only planes that are viewable are used. The used planes are then sent to the Relative Plane algorithm that computes the map

4.4.2.4 PlaneManager, the map subsystem

PlaneManager contains a collection of PlaneObjects which stores the map of the simulation. The class also has the code which creates the map.

The PlaneObject class is also used in the relative algorithm to store planes. It and the Point3 class are described in more detail when describing the classes of the relative planer SLAM algorithm.

4.4.2.5 Profiler

The profiler is used for performance reasons. It logs the time spent in each function and then displays this time.

4.4.3 Relative Algorithm Classes

4.4.3.1 RltXPointsS

RltXPointsS stands for RltXPoints storage. Its original purpose is to store RltXPoints but as the project evolved from the four plane problem onwards it became the entry class of the algorithm. It is involved with adding untransformed planes to the algorithm, managing RltXPoints and rolling back the algorithm. Being the main class of the algorithm, it also maintains the count of the current iteration and does maintenance at the start of every iteration. The interesting parts of this class are how this class is used to store data.

Early on it is realized that an RltPlane can be linked to multiple RltXPoints. This does not lead to a natural home to store the RltPlanes in the RltXPoints. If they are primarily stored in the RltXPoints, there would need to be a function that traverses the RltXPoints and only returns each RltPlane once for purposes such as matching. An easier way to do this is to store a master list that is used for such purposes and the RltXPoints simply have a reference to the RltPlanes. This led to the RltPlanes being stored in a vector inside of the RltXPointsS class, which made enumeration simple.

In addition to storing the master list of RltXPoints and RltPlanes, this class also stores a master list of RltXPointLink. Each RltXPoint maintains a list of other RltXPoints they are linked to for adjacency purposes. When building a map, the links have to be traversed, and a very similar issue to having multiple references of RltPlanes occurs. It is possible to enumerate every link given a graph of RltXPoints but it would take a stack and a complete traversal which is kind of a waste. Also when linking together two RltXPoints, it would also be necessary to know the list of planes that link together the RltXPoints. A better solution is to have the RltXPointLink class which stores information about two RltXPoints that are linked together and the planes that link the two. This makes the enumerations far simpler. An instance of a RltXPointLink is created when a new RltXPoint is created linking it to the previous one. Then a linear traversal is used to enumerate the links to generate the map which is far simpler than traversing a graph.

The key functions are:

- *AddPlane,* which adds a raw plane relative to the robots viewpoint to the algorithm.
- *ManageXPoints,* when there are new planes that cannot be matched against existing ones, it

adds them to RltXPoints and creates new RltXPoints if necessary.

- *ComputeXPoints,* the entry point function that computes the relative relationship between RltPlanes.
- *RollBack,* rolls back the simulation to a certain iteration and then rolls it forward to the current iteration.

4.4.3.2 RltXPoint

The RltXPoint mostly functions as a container to its references of its RltPlanes. It has a list of the RltPlanes that are contained inside of it, and most of the processing deals with traversing this list and calling RltPlane functionality.

The key functions are:

- Match/Add planes, traverses the RltPlane list to see if there are any matches. It also does something similar to see if a new plane belongs in the RltXPoint by traversing the list and seeing if the new plane is contained in all of the other plane's intervals.
- *ComputeXPoint*, enumerates the list finding the relative offsets between pairs of planes and then combining the relative offsets from pairs of planes together to form the offsets between all of the planes. Note that the individual computation is handled by another class. There are two other important functions:
  - After the computation it checks each planes to see how many iterations are used when forming the offset. After the RltPlane has been present in the RltXPoint for a certain number of iterations, if the interval that it is calculated in is too small, a hint is created to have the RltPlane not added to this RltXPoint and the algorithm is rolled back.
  - If for whatever reason a previous calculation is better than a current one, the previous one is used.
- Roll back functionality, the class has several functions that are necessary for the roll back functionality.
- Pair reordering, periodically it can be checked if there is a more optimum pairing of planes to generate the relative offsets.

4.4.3.3 RltXMMPoint

RltXMMPoint is the motion model version of a RltXPoint. When the motion model first turns on, an instance of RltXMMPoint is created. It is a subclass of RltXPoint and its interface is nearly the same as RltXPoint.

The key function is:

- ComputeXPoint. This has the same function prototype as RltXPoint so it is called the same way, but it only works on the single RltPlane that the RltXMMPoint has.

4.4.3.4 RltPlane

RltPlane primary purpose is to store and provide access to data. The data consists of a circular array of PlaneObjects that contain the untransformed readings given by the robots viewpoint, and a list of RltXPointRef which reference the offset of the plane from potentially multiple RltXPoint. The RltXPointRef are computed in ComputeXPoint and used when building the map.

The key functions are:

- *Match/Add Plane,* The class could be queried to see if the last matched plane matches up to a given one, and then to add the raw plane to the storage.
- *GetPlane*, Gets a plane object (raw plane) at a given iteration. Originally the correct plane is found using a linear search. However with it being linear in combination with the linear processing of the plane comparison it made the algorithm O(n^2). Instead a binary search is used, which causes the first plane comparison to be O(nlogn). Since the planes are referenced in order, after the binary search is used to find the first plane it is an O(1) enumeration.
- *GetRltXPointRef*, The RltPlane may have several RltXPointRef since it can be linked to many RltXPoints. This function returns the correct one given either a RltXPoint reference or an iteration. The iteration version is used when requiring a key point for a given iteration to calculate plane expansion.
- *GetInterval,* Returns the interval where there is plane data present by doing a full enumeration of the plane storage.
- Roll back functionality, the class has several functions that are necessary for the roll back functionality.

4.4.3.5 RltXPointRef

Each time a RltPlane is bound to a RltXPoint, a new RltXPointRef is created. The RltXPointRef contains data such as

- The position and rotation offset of the RltPlane to the RltXPoint.
- A reference to the RltXPoint.
- A boolean value if it is valid or not.
- The interval at which it is computed.
- The key point index, 0, 1 or 2 which is not valid, left or right.
- The max length and height.
- The reserve reference which is used if the plane comparison cannot be calculated. It is an exact copy that stores the previous best computed reference values. It can substituted for the RltXPointRef if need be such as the RltPlane runs out of untransformed planes to compare to. A second reserve reference is used to store the plane that it is compared to. This second reference is used in case the plane that it is compared to is a different size in a previous calculation which would mean that the offset needs to be adjusted for the change of size.
- A boolean if the RltPlane has been confirmed to belonging to the RltXPoint for enough iterations. This avoids rechecking the RltPlane once it is found to belong..

Most of the functions in the class are accessors. Functions that are not accessors are.

- *AdjustXRefSize* which adjusts the size of the plane. This is used when the plane is smaller than another version of itself so it has to be up sized for a comparison to be valid. It uses the key point and the new size to do this.
- DoBackupCheck, The goal is to store the most valid comparison with the plane it is compared to. This function does a comparison of the current plane size and then the stored reference to see which length is the largest. It also checks the interval length if the size comparison is similar.

4.4.3.6 RltPlaneCompare

This class does much of the important work for the algorithm. It is the one that compares two RltPlanes and generates their relative offset and their key points. All the functionality in this class are originally in RltPlane, but after a certain point it became large enough so that it was put into its own class.

There is only one external function in this class: *ComparePlanes* that finds the best interval to compare the two planes and then uses that interval to calculate the offset and the key point.

4.4.3.7 RltInterval

This class stores an interval so it has only two member variables, start and end. It has four operations that are self explanatory S*ize, InInterval, MergeInterval* (or operation)*, Shrink* (and operation).

4.4.3.8 RltPlaneInterval

Since it is possible a RltPlane to have more than one interval the RltPlaneInterval class stores many RltIntervals. It is very similar to RltInterval except all operations occur over multiple intervals. Its primary operations are *InInterval, AddInterval, Enumerate,GetMinimumInterval,TotalSize*.

4.4.3.9 RltXPointLink

This class is used to store a linking structure when two RltXPoints are linked together. It simply contains accessors for its two RltXPoints and the vector of RltPlanes that link them together.

4.4.3.10 RltMapBuilderUnitTest

This class is used to test the initial algorithm to add RltXPoints. It is not necessary anymore, but as a unit test it is kept around.

4.4.3.11 RltMapBuilder

RltMapBuilder along with RltXPointsS are the top level classes. This class is responsible to build a map by enumerating each RltXPointLink from RltXPointsS and to get the current location given the last iterations raw plane readings. For global plane matching purposes, it is used by RltXPointsS to match raw planes by: obtaining the current location, offset an untransformed plane reading by this location, and then seeing if it matches a plane on the global map. Also because it can detect when a current location is not valid it is used to determine if a motion model should start.

4.4.3.12 PlaneObject

This class is used in both the simulation map generation and the relative algorithm. It stores raw planes two different ways, first in terms of its plane equation and bounding box, and then in terms of the four rendering points used to draw it.

Its key functions are:

- GenerateNoisyPosition which is used to create the noisy plane to enter into the algorithm.
- AdjustSize which is used to adjust a plane object to a new size given the new size and which key point to base the expansion.
- GetKeyPoints, which first rotates the plane so that it is at 0^0 so it is completely on the x axis. The left most top point is key point 1 and the right most top point is key point 2.
- CalculateOffset which computes the offset for two plane objects. Used in map builder when comparing a raw plane to a global mapped one.

But by far the most important one is:

- MatchPlane. This function determines if and by how much two plane objects are matched together. It is very important as the whole concept of the RltPlane is that a new plane belongs if it matches to the last known plane.

4.4.3.13 Point3, Matrix

The Point3 class stores three float variables x,y and z, and contains a number of mathematical operations that can be performed. The Matrix class is a 4x4 matrix use for transformation. The Point3 class can be multiplied by the Matrix class to transform the Point3.

4.4.3.14 RltXPointChart

The RltXPointChart is used in the calculation of the current RltXPoint. For every iteration, when an untransformed plane is added to a RltPlane, all the RltXPoint references of that RltPlane are charted. During the ManageXPoint routine, the RltXPoint with the highest chart number is declared as the current RltXPoint.

4.4.3.15 RltTemporalHint

The RltTemporalHint class is a data structure that stores the hint type, and information on where to apply it. That information is in the form of the untransformed plane object, iteration, and the index of the RltXPoint not to add the plane to. Given this information, when rolling back time it is possible to avoid adding a RltPlane to an RltXPoint.

4.4.3.16 RltTemporalHints

This class stores instances of RltTemporalHint. It has functionality to add hints and retrieves hints given an iteration. It also has some helpful functions *CanAddToXPoint and CanMakeNewXPoint* which search for a certain hint given a plane and iteration. This functionality is here rather than in RltXPointsS.

The only tricky part of this class is that when adding a hint all other hints that are stored in the future have to be removed since they might not be valid anymore. This can cause a roll back to cause several other roll backs before the current state stabilizes.

4.4.3.17 RltPlaneStorage

This class stores all of the untransformed planes for each iteration. To store information for each iteration, a circular array similar to RltPlane is used storing everything in a PlaneRecord structure. This structure contains.

- A vector of PlaneObjects (all input planes for an iteration)
- A vector of RltPlane pointer as the unassigned list. Notice how convenient it is that the RltPlanes have a master storage. Otherwise it would be difficult to keep track of when to delete references.
- The current plane counter.
- The current RltXPoint index.

- The current iteration.
- Three variables to do with storing motion model information.

It is interesting to know that these variables completely describe all non computed state variables for a iteration.

Most other functionality in this class are accessors for the PlaneRecord circular array.

4.4.3.18 RltPoset

This contains three variables, position, rotation and valid. It is used in the RltMapBuilder to store previous locations.

4.4.3.19 RltPlaneList

This class is very similar to the RltXPoint class. It is used for the unassigned plane list in RltXPointsS. The unassigned plane list needs to remove RltPlanes that have an insufficient amount of readings to be used. This extra function meant that RltPlanes would be internally stored as a list and not a vector, since RltXPoint only has to remove RltPlanes during a rollback. A vector removal is more expensive as the vector needs to be cleared and reloaded with all the RltPlanes except the one not wanted. Due to this reason a new class is created that is very similar to RltXPoint. It contains the same matching/adding functionality and the same enumeration functionality.

4.5 One iteration of the algorithm

The previous sections: 4.1 Introduction describes the concept behind the algorithm, 4.3 The Derivation describes the development and rules of the algorithm, and 4.4 Class Architecture describes how the classes of the project are developed. Now lets put everything together to show how the algorithm runs through a single iteration of receiving untransformed input from the robot to generating the map.

Relative Plane Algorithm walk through

- The algorithm starts by calling RltXPointsS *StartIteration* which does some housekeeping such as clearing the RltXPointChart for the start of the iteration.
- Then each plane that is visible is sent to the RltXPointS *AddPlane.*

  RltXPointS *AddPlane*

  - First each plane is saved into RltPlaneStorage for roll back purposes
  - Then the plane is compared to every plane in the current RltXPoint by calling RltXPoint *MatchAddPlane*.

    RltXPoint *MatchAddPlane*

    - Enumerates each plane in the RltXPoint to find the greatest match by calling RltPlane *MatchTPlane.*

      RltPlane *MatchTPlane*

      - Returns the value of PlaneObjects *MatchPlane* using the previous iterations plane object compared to the input plane object

PlaneObjects MatchPlane algorithm.
• Takes the dot product of the two plane's normals and compares against a constant, returns false if smaller than the constant. *Rule 10* • Compares the planes *d* to a constant, returns false if larger than a constant. • Compares the bounding boxes by looking at the 8 points on the bounding box plus the middle

since there is a special case if one plane is very tall and the other very wide where they overlap but do not have any of the corner points inside of the other bounding box (there still could be a failure case even with the middle one and would require an edge check however this case is unrealistic). The function compares each bounding box points against the other bounding box min and max to see if it belongs. At least one point needs to return true.

- One final check to make sure that they are only separated by a small number of iterations. It would be possible to have an aliasing effect where one plane goes behind the robot view and a new one appears that matches the old one. By checking the iteration this ensures that the new plane would not be incorrectly assigned to the old one.
- If at any time a check fails return -1. If it passes then form a combined score of the four checks out of 100 in case more than one plane matches the best one can be selected.

▪ After the enumeration, if there is a good match call RltPlanes AddtoTPlane to formally add the untransformed plane to the RltPlane and return true. If there is no match then return false. Rule 1

RltPlanes *AddtoTPlane* Rule 22

- In AddtoTPlane, in addition to adding the plane to the circular array: If it is the first plane added to the RltPlane then save its starting location for potential use when aligning the plane to the global map. Also call RltXPointChart *ChartXPoints* which increments all RltXPoint references connected to the RltPlane.

○ If the current RltXPoint cannot match the plane then try any RltXPoints adjacent to the current one. This is done by calling *EnumXPConnection* of the RltXPoint and using the same match procedure as above.

○ If there still is not a match search for a global match calling RltMapBuilder *MatchPlanetoMap*.

RltMapBuilder *MatchPlanetoMap* Rule 11

▪ Given a location from the previous iteration, transform the untransformed plane to global space and do a match with all known planes. If there is a good match return it. There is also a version that returns all good matches for the loop closing pseudo merge.

- ○ If there is a global match then add it to that RltPlane. In addition, add the RltPlane to a list which stores RltPlanes that globally matched in an iteration. This is used later in ManageXPoints.
- ○ If there still is not a match, then see if the plane can be added to a RltPlane in the unassigned list. If not create a new RltPlane with the raw plane and add a plane to the unassigned list which is processed in ManageXPoints.

- After all the planes have been added RltXPointS *ManageXPoints* is called that processes the unassigned list, any global matches, and calculates the current RltXPoint

  <u>RltXPointS *ManageXPoints*</u> Rule 7 Rule 32

  - ○ Enumerate the unassigned RltPlane list looking for any RltPlanes that have more than the minimum number of readings. Also check to see if there is a hint which say that this plane should not be added to the map.
  - ○ Check each enumerated RltPlane to see if it belongs to the current RltXPoint and make sure there is not a hint which says it cannot be added to the RltXPoint. If it can be added then add it.
  - ○ If it still cannot be added, enumerate the RltXPoint connection list to see if any adjacent RltXPoint can accept the enumerated RltPlane, that does not have a hint saying it cannot be added.
  - ○ If it still cannot be added then make a new RltXPoint calling the function *MakeNewXPoint*

    <u>RltXPoint *MakeNewXPoint*</u>

    - ■ Do some basic variable setups then call *TraverseToLink*

      <u>RltXPointsS *TraverseToLink*</u> Rule 12

      - • Do a complete traversal of all of the RltXPoint starting a the current one. This requires a stack and a boolean flag in each RltXPoint so each RltXPoint is only visited once.

        - For each RltXPoint call *LinkToNewXPoint*

        <u>RltXPointsS *LinkToNewXPoint*</u> Rule 8 Rule 9

            - Enumerate each RltPlane in the RltXPoint that we are looking to link to.
            - First check if the enumerated RltPlane is in the same interval as the RltPlane that is the one that triggers the new RltXPoint.
            - If it is, since we are only looking for two linking RltPlanes compare the distance and the starting iteration to see if it should be used rather than other RltPlanes
            - If there are two good linking planes then add them to the new RltXPoint and return true, otherwise return false.

        - As soon as *LinktoNewXPoint* returns true, return with the reference of the linking RltXPoint and the two planes to link to
        - If we are in a motion model this is a special case where only one plane will match, since only one plane is previously viewable. Rule 29
        - Make the connection between the new RltXPoint and the one that is returned from *LinktoNewXPoint*.

- Now enumerate the global match list to look for any planes that matched to a global reference to see if they should be added to a known RltXPoint.
- Call *CheckForLoopClosing* which is used to do a pseudo merge. When a loop is closed there could be two planes on the map that are the same.   If there are two instances of the same plane, then only the one used will get charted so the normal get current RltXPoint might not be accurate. This function checks for that case so the current RltXPoint is the proper one.

<u>RltXPointsS *CheckForLoopClosing*</u> Rule 37

- ▪ Enumerate all planes seen this iteration.
- ▪ Match each enumerated plane to the global match receiving a list of all global matches.
- ▪ Store the count of each plane match for every RltXPoint seen.

        - After the enumeration, if there is an RltXPoint that is linked to every plane make that the current one. Also end the motion model if there is one, and call *MendMergeAfterLoopClosing*.

          RltXPointsS *MendMergeAfterLoopClosing*

            - This is nearly identical to CheckForLoopClosing though this time if it is found that there is plane that should be charting an RltXPoint but is not, add a one way link for charting purposes. The link is one way so it does not effect *ComputeXPoint..*

    - If *CheckForLoopClosing* returns false then check if the RltPlane can be linked to the current RltXPoint, and then to the adjacent RltXPoint.
    - If there still is not a match then make a new RltXPoint as long there are less than a certain amount of RltXPoint links for the RltPlane. Rule 29
    - Clear the global list and do some maintenance on the unassigned list since the plane should only be kept there for a certain amount of time after they are visible.
    - Calculate the current RltXPoint by calling the RltXPointChart *GetBestXPoint*

      RltXPointChart GetBestXPoint Rule 22

        - Return the RltXPoint index of the RltXPoint with the most RltPlanes charted to it this iteration. If there is a tie with last iterations RltXPoint then use the previous one.

    - Add the unassigned list, plane counter,and current RltXPoint to the plane storage for use for roll backs.

- Now call RltXPointsS *ComputeXPoint* that generates the relative offsets of all the planes in an RltXPoint.

  RltXPointsS ComputeXPoint

    - Iterate through every RltXPoint and call its *ComputeXPoint* function.

      RltXPoint *ComputeXPoint*

        - Check if any of the RltPlanes in the RltXPoint have new observations. If not do not continue in this function since there is no computation to perform.

- Call *ReorderPlanes* after a certain amount of time from the creation of the RltXPoint

  <u>RltXPoint ReorderPlanes</u> Rule 30

  - For each RltPlane in the RltXPoint create an n by n matrix storing the results of the *ComparePlanes* interval size. If there is not a valid key point between a pair of planes store the result as negative one so the pairing is not used.
  - Check each possible path through the matrix without allowing for multiple visits to a RltPlane to calculate the path with the highest total interval count.
  - Use the best calculated path to adjust the pairing of planes for comparisons.
  - If using new pairings clear any saved previous pairings.
- Enumerate through every RltPlane in the RltXPoint and compare planes in pairs.
- Set the RltXPointRef valid member to false of the second plane in the comparison (the first is the reference) and call RltPlaneCompare *ComparePlanes*

  <u>RltPlaneCompare ComparePlanes</u>

  - Calls *GetCombinedInterval* to get the combined interval of the two planes

  <u>RltPlaneCompare GetCombinedInterval</u>

    - Gets each of the intervals of the two RltPlanes and then return the value of RltPlaneInterval *GetMinimumInterval* using both those intervals.

      <u>RltPlaneInterval *GetMinimumInterval.*</u>

      - Do a nested enumeration of both of the two intervals given. When one interval is in the other, take the smallest combination and add it to the return RltPlaneInterval.
  - Then call *GetMaxSize* that gets the best interval available out of the ones returned by *GetMinimumInterval*

  <u>RltPlaneCompare *GetMaxSize*</u> Rule 18

    - Enumerate the combined interval

    - Get the raw plane for each iteration of the interval to get its max length.
    - Place the max length into a moving average for noise reasons.
    - Detect if the moving average is increasing or decreasing, If it has been noted as increasing and then decreasing or vice versa then split the interval. There is some hysteresis in the detection for noise reasons.
    - For each interval store the one with the greatest total size of both planes as the best one.
    - When generating key points this function could be forced to return a certain interval. If it is forced and the current interval is the one to be returned then do that.
  - If the interval size is 0 then exit and at which point the RltXPointRef to not valid.
  - Otherwise enumerate the interval. For each iteration get the raw planes and then the two key points of each of the two planes to give four points. Rotate these four points so that the first plane has an angle of 0^0. Each of the four points generates four midpoints given each planes maximum size in the calculation interval. Compare these four mid points to get four offsets. The ordering is consistent over time since the key point have a ordering convention, so store the four results in an array.
    Rule 3 Rule 14
  - After the interval enumeration take the standard deviation of the array which is the size of the interval length by four. Rule 13
  - Do a check to see if the there is a specific two standard deviations that are lowest and close in size. If this is the case the key points cannot be found. Otherwise use the standard deviation that is the is lowest to figure out the offset and the key points. Fill in the RltXPointRef information and return Rule 15
- Call *CheckIntervalSize* which checks if the RltPlane has enough iterations to be consider in the RltXPoint.

RltXPoint *CheckIntervalSize* Rule 31 Rule 32

    - Check a boolean in the RltXPointRef to see if we have confirmed if we belong to the RltXPoint yet so we do not have to repeat the processing. Return if we already confirmed this RltPlane.
    - Check if the interval the RltPlanes RltXPointRef is calculated in is larger than a certain size, if it is this set the check to true and exit.
    - If it is not, then check the starting iteration of the RltXPointRef and subtract it from the current iteration. If we are below the minimum just exit for now since we cannot make a judgment.
    - If we are above the minimum then the RltPlane should not belong to this RltXPoint. Create a RltTemporalHint. This hint should either be a hint that says that the RltPlane should not be added to the RltXPoint it is currently in, or if the RltPlane has been already used to create a new RltXPoint it should not be added to any RltXPoint which means it is not included in the map. Then add the hint to the RltTemporalHints.
    - Call RltXPointsS *SetRollBackTo* to have the algorithm roll back time to when the RltPlane is first added to the RltXPoint so that the hint can be used. Rule 34

- After *ComparePlanes* check if the key point is valid. If not call *CalculateMissingKeyPoints*.

  <u>RltXPoint *CalculateMissingKeyPoints*</u>

    - Go through every RltPlane in the RltXPoint except the one that is missing the key point and call RltPlaneCompare *ComparePlanes* using dummy RltXPointRef variables, and forcing the *ComparePlanes* to use the current interval. If there is a valid key point then load that into the actual RltXPointRef.

- Call the RltXPointRefs *DoBackupCheck* to see if we have the best available calculated offset or there is a better one stored in the RltXPointRef variable.

  <u>RltXPointRefs *DoBackupCheck*</u> Rule 35 Rule 36

    - Check the maximum size of the current reference and the previous best one. If the previous best one is better, then copy it over to the current reference. Otherwise copy the current reference over the previous best one.

- If we are loading the previous reference, then change the return value of the base RltXPointRef that when the function is called it has the current information of the the plane that this one is compared to. Change it to the one used when the previous best referenced is backed up. This is important as there could be a difference of plane size of the comparison plane which has to be adjusted for when grouping together all of the offsets.

▪ After we have processed all of the plane comparisons for the RltXPoint, chain together the individual comparisons by taking the result of one and use that as the new offset reference for the next one. There may be a size difference when the results are chained together. Say plane 1 compares against plane 2, then plane 2 compares against plane 3. If the two references of plane 2 are not the same size there needs to be an adjustment to the offset to account for this. Rule 2 Rule 28

▪ At this point everything is in the space of the first plane which is at the origin point. For rendering purposes only it is better if the RltXPoint is in the middle of all its planes it is compared to (makes no difference for map building). So take the center of all the planes and offset the planes so that the RltXPoint is in the midpoint of all the planes.

▪ RltXMMPoint *ComputeXPoint* is called instead for motion model RltXPoints

RltXMMPoint *ComputeXPoint* Rule 24 Rule 25

- The RltXMMPoint takes a look at the reference of the previous RltXPoint (in *ComputeXPointOffset)* to the one seen plane and compares it to the location of the RltXMMPoint. It computes the relative offset of the RltXMMPoint to its one RltPlane. This relative offset will be held static and is used to generate the location of the robot given a raw plane.

- Iterate through every raw plane. Use the point to plane distance equation to determine the x distance. Use the cumulative angular difference with the x distance to compute the z distance.

- If we know the location of the robot, we know where an iterations plane is compared to the stored plane reference derived from the previous RltXPoint. If the plane happens to be bigger expand the reference of the RltPlane to account for the bigger size (in *ExpandPoints)*. Rule 23

- Now call RltMapBuilder *BuildMap* to generate the map.

  RltMapBuilder *BuildMap*

  - Each RltXPointLink is enumerated Rule 6
  - The first one is a special case and is marked as such. In this case the RltPlane is compared against the first instance of its plane object which is used to align the computed map to the actual global map. For this special case there is another special case when the first instance of the plane object is no longer stored since the RltPlane storage rolled over. This is handled by having the RltPlane store information from a time where the first offset is calculated. Rule 4
  - If the RltXPointLink linking RltXPoint is the motion model version, it contains the global location already so use that.
  - For every other case look at the link structures RltPlanes which link the two RltXPoints. Get the two RltXPointRefs from the RltPlane for each of the RltXPoints. If the reference have different sizes then expand the one that is smaller using the key point. Rule 17
  - Use the two references to figure out the offset from one RltXPoint to another. There are two RltPlanes normally and one if the base reference is a motion model. Average the offset and add it the current total global location. This is the global location of the linked RltXPoint.
  - Enumerate each plane in the linking RltXPoint and its RltXPointRef variable. Transform the information in the RltXPointRef using the global location of the RltXPoint. If that plane is not mapped yet then add it to the map. If it is currently mapped then replace the previous plane on the map only if it is larger than the previous on.

- Now call RltMapBuilder GetPoset to get the current location

  RltMapBuilder *GetPoset*

  - Get every raw plane/RltPlane pair that is available at the current iteration.
  - The RltPlane index matches up with the corresponding plane on the global map. However

it might not be the same size since the global map will always have the largest known size.

  - ○ If the size is different then we could expand the raw plane only if the RltPlane has a key point that is generate in the same interval as the current iteration. If it does, add the plane to a good list. If it does not add it to a bad list. Rule 16
  - ○ Take a look at the bad list to see if it is possible to generate a key point using one of the known planes on the good list. If it is we could then expand the plane accordingly and then add it to the good list. Rule 19
  - ○ If a bad plane cannot get a good key point using the good plane list then try the same key point generation using the bad plane list.
  - ○ If it still does not have a valid key point then we cannot use it.
  - ○ For all good planes, match them against the global mapped version by calling PlaneObjects *CalculateOffset*. Rule 5

PlaneObjects *CalculateOffset.*

  - ▪ Compare the two plane's normal to figure out the rotation difference. Then use that difference to rotate the second plane by the negative of that amount and then subtract the two midpoints to get the position offset.

  - ○ Take the average of all the locations to get the current location.
  - ○ At this point if there are no good planes to get the location, check to see if we should be in a motion model by calling *CheckForMotionModel*

RltMapBuilder *CheckForMotionModel*

  - ▪ If we have no good planes, increment a counter.
  - ▪ If the counter is over a certain amount and the start of the count occurred less than a certain number of iterations then we should have a motion model occurrence. Call RltXPointsS *CreateMotionModelXPoint* Rule 26

    RltXPointsS *CreateMotionModelXPoint*

    - • Get the location of the last known location. This is the global position of the RltXMMPoint. Rule 27

        - Link the RltXMMPoint with the previous RltXPoint and load in the one linking RltPlane reference.

    - If the start of the count is higher than a certain amount reset the counter.

- Now call RltXPointsS *RollBack* to check if we have to roll back the algorithm due to a hint.

  <u>RltXPointsS *RollBack*</u> Rule 33

  - If a function requested a roll back it calls a function to tell RltXPointsS the iteration it wants the algorithm to be rolled back to. If there is not a roll back request, then exit the function.
  - Go through every RltPlane stored in the master list. Call the RltPlane *RollBackTo.*

  <u>RltPlane *RollBackTo*</u>

    - Find the plane at or before the given roll back iteration using the *GetPlane,* which does a binary search of the circular array looking for the iteration. If the roll back would cause the RltPlane to have no plane object left then return false. If true simply change the write index to the new iteration. Check every RltXPoint the RltPlane is linked to, to see if it still belongs. If it does not then remove it from the RltXPoint and remove the local reference.

  - Add RltPlanes that still exist to a good list. Add ones that should not, to a bad list to be deleted later.
  - Go through every RltXPoint in the master list. If its creation iteration is greater than the roll back then add that RltXPoint to a removal list. Also have it remove any two way references to any RltPlane that it is linked to. Otherwise add it to a good list.
  - Go through every RltXPointLink and if either of its links contain a RltXPoint that is on the delete list then add the RltXPointLink to the delete list. Otherwise add the RltXPointLink to a good list.
  - Since all the master lists are stored as vectors, clear those vectors and add any objects that are in the good list back to the master list. Any objects in the to delete list should be deleted to clear the memory.
  - Call RltXPointsS *ComputeXPoints* and RltMapBuilder *BuildMap* to reset all of the

computed structure. Load additional information from the PlaneStorage such as the unassigned list.

- Now iterate for every iteration from the roll back iteration to the previous current one. First calling *StartIteration,* then *AddPlane* for every plane object stored in the PlaneStorage for each iteration. Then *ManageXPoints, ComputeXPoints,* RltMapBuilder *BuildMap* and then *GetPoset.*

4.6 Closing Remarks

4.6.1 Potential issues

One issue not discussed so far is what happens if the noise model is not flat. What if the noise increases with the distance of objects. This situation is not neglected because it is a problem, rather it is not a problem at all. The reason other SLAM algorithms prioritize closer objects due to lower variance (but still the same mean) is because it would create larger short term errors in global location using objects farther in distance. In those algorithms, if there is a large enough error the algorithm could lose tracking and get lost.

This algorithm does not rely on the global location so any large short term errors do not matter. As long as the plane maintains the same mean regardless of any extra noise due to distance it should work. If for whatever reason the extra noise does cause issues there is a very simply solution. When choosing the interval used to calculate the plane comparison, the plane distance to the robot can be used to choose the interval that is closer.

This algorithm does not consider planes that move dynamically. There is a solution to this. If a plane has large standard of deviations in all its key points perhaps it could be removed from any calculation. This would require multiple comparisons to figure out which planes in the comparisons are static and which are dynamic.

Another issue not discussed is what happens when the RltPlane *MatchPlane* fails due to large short term noise. As it turns out this is not a big issue. The worse case scenario is that there is a run of noisy planes in one direction causing the last entered plane to be well off from its true position. In the next iteration lets say the untransformed plane is accurate but too far from the previous untransformed one to be matched correctly. In this case the global match will pick it up and match it into the correct RltPlane.

4.6.2 Related work

This algorithm is conceived from the unsuccessful previous algorithms described in this thesis. After its completion it is noted that many of the underlying concepts are seen in other papers. Several papers having some of the underlying concepts but none have all of them together. Mei et al

[MSCN09] uses the concept of relative space to register the output of stereo cameras. Newman [Newm99] stores landmarks in terms of relative position and has an algorithm to constrain the landmarks since there might be multiple links to each landmarks causing different global locations. Csorba [Csor97] is perhaps the closest to this work, as it links point based landmarks using a third point for relative angles. Lu et al [LuMi97] does scan matching between frames and stores all of the scans so that they could be combined to minimize error. The concept of subdividing space for calculation purposes is common and previously referenced in this thesis as [LMSK03] [Fres07] [PiTa08].

Chapter 5 EM in combination with the Relative Plane Algorithm

5.1 Introduction

For verification, it is important that the Relative Plane algorithm is tested using the results of the EM simulation. There are some differences in the simulation used in the previous chapter and the EM simulation. The EM algorithm works on points that are occluded so it is not possible to see through walls.

In the previous chapter's simulation, the planes growing and shrinking is based on the planes intersection with the frustum. In the EM algorithm, the planes growing and shrinking is based on the visibility of points in a plane. This means that a plane grows and shrinks in discrete steps as new points are seen and past ones leave. In the previous chapter's simulation, the growing and shrinking is continuous.

The past EM simulation in 3.5 uses a plug in architecture which has advantages and disadvantages. For a programming team with multiple people, a plug in architecture allows the work to be developed separately with a common interface to allow for interoperability. It requires that each programmer have only the code they are developing themselves which allow for lower compilation time. A disadvantage is the extra work to load the plug in. For medium sized programs, like the simulator used, the compilation time is small compared to the extra work. A larger problem is that when debugging, it is only possible to either change the code in the simulation or for a single plug in at any given time. With multiple plug ins, if debugging, it is not possible to change the code in both of the plug ins at the same time.

For these reasons, it is decided it would take less time to make a new simulation that would contain code from the old EM simulation without using the plug in architecture. The amount of time copying and pasting the code is minimal and it did save time later on.

This chapter shows the results of running the algorithm in the full simulation of generating points, running the EM algorithm to generate planes, and then using those planes as input for the Relative Plane algorithm.

5.2 Results

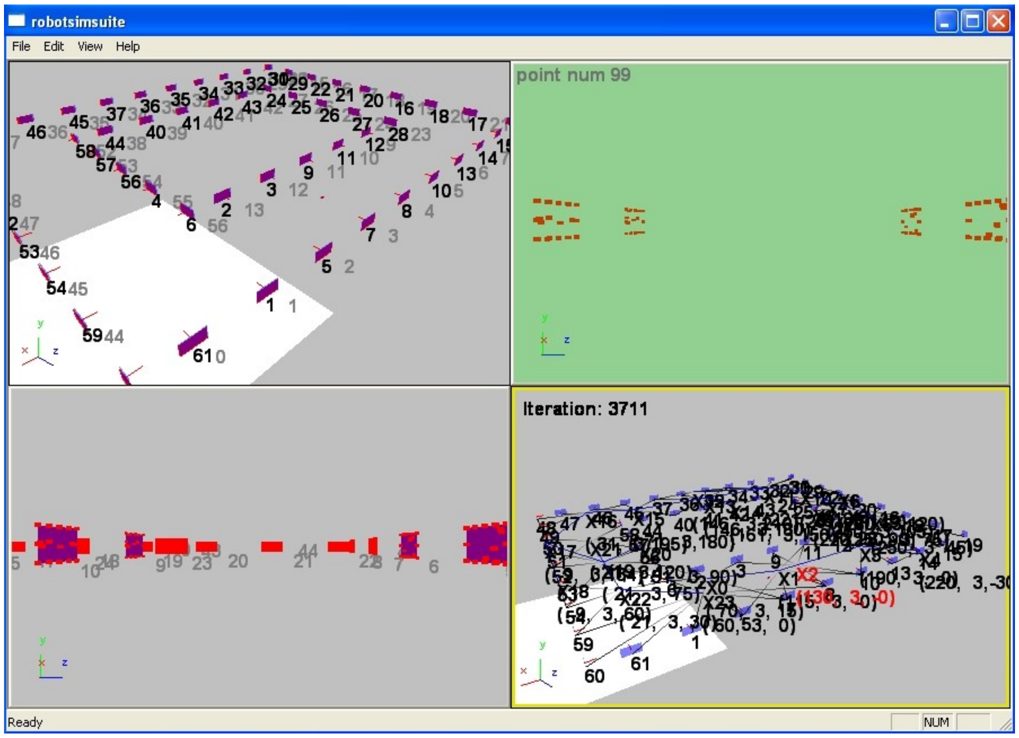


Figure 105 No noise

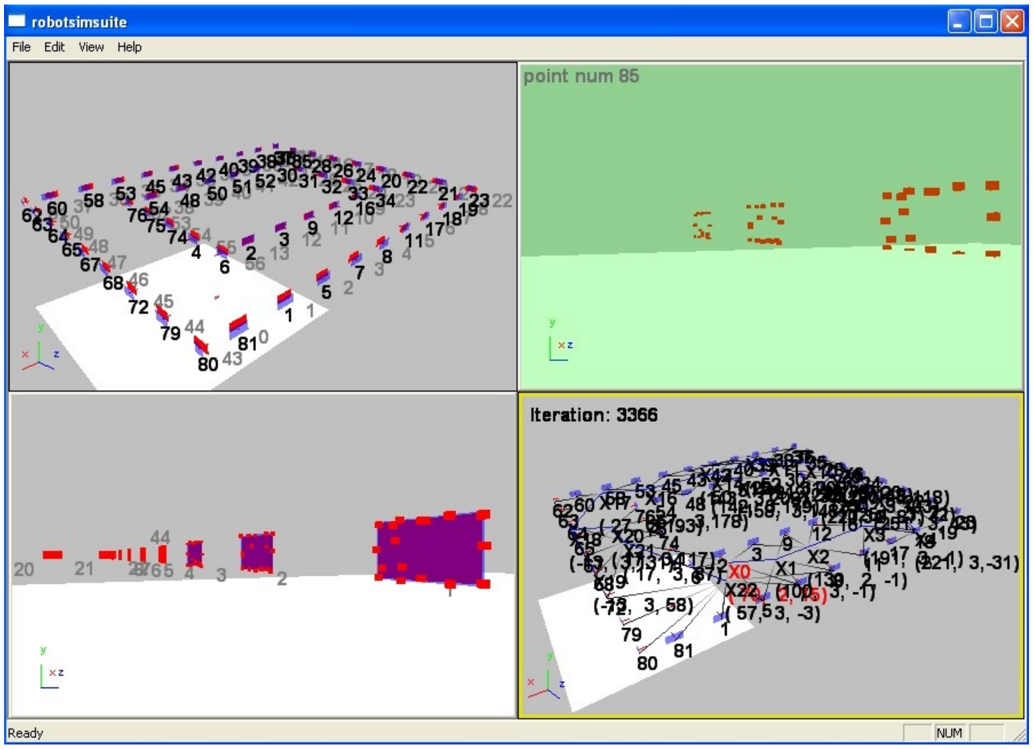


Figure 106 With noise

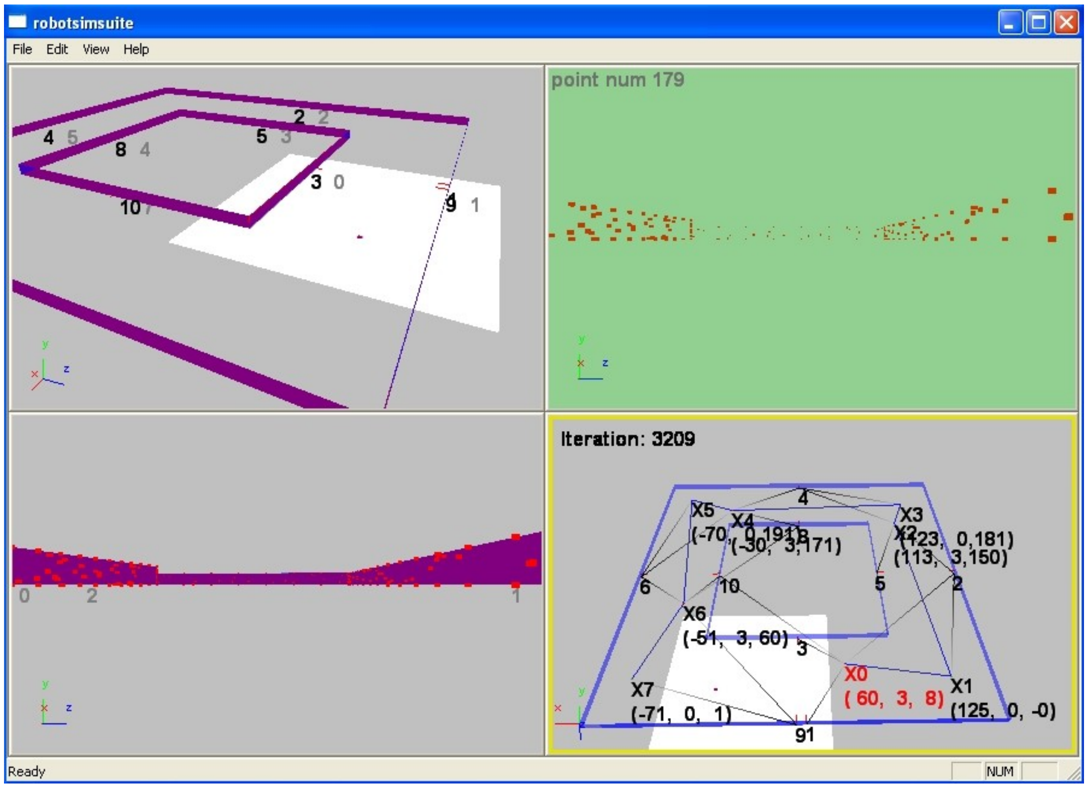


Figure 107 No noise

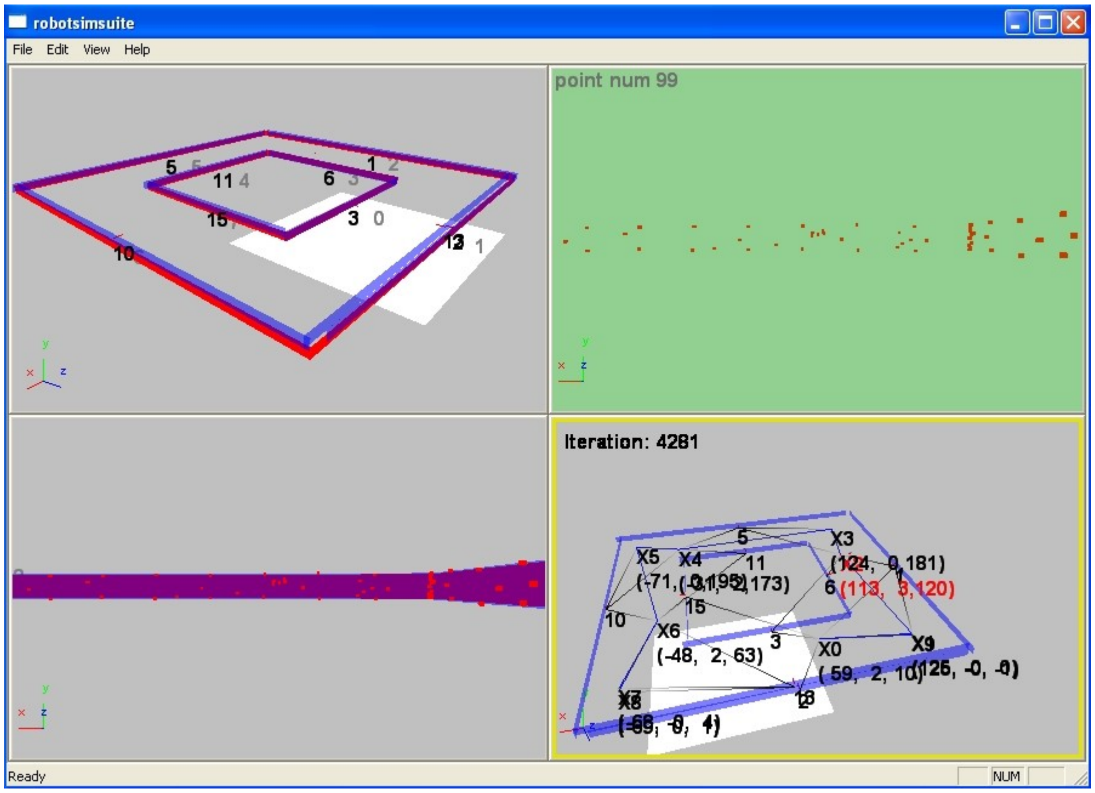


Figure 108 With noise

Figure 105 and Figure 106 show the results of a simulation with many planes but a small frustum and Figure 107 and Figure 108 show the results of a large frustum testing the motion model. The maps are identical to the ones used in the previous chapter's simulations. The only differences is that the planes are create out of points and each point has an independent amount of noise generated for it rather than having the noise generated for each plane itself.

In all those figures, the top left window shows the actual map in comparison to the generated map. The top right window shows the points currently visible to the robot. The bottom left shows the planes the EM algorithm generate in comparison to the planes viewed by the robot at the time. The bottom right show the results of the relative algorithm by itself, along with the RltXPoints connectivity and plane numbering.

Figure 105 does not have any noise in it and it does appear to generate the map correctly. However there is a slight amount of error in the map. The error happened as the robot was turning and is similar to the issue in Figure 89. When the robot turned, due to the viewing frustum being longer in the sides than the center, some planes were seen for a brief period of time. This caused the function which splits an interval when a plane is growing or shrinking to not have enough iterations to correctly identify when the interval should have been split. The last two untransformed planes seen should have been split to a second interval but were not. When doing the plane comparisons, the standard deviation was wrong so the wrong key points were selected. A possible solution is tested that changes the amount of planes for the moving average calculation and allowing the interval split to happen when it should occur. Although this solves the problem, it would have an adverse effect when working with noisy data. A better solution to this issue would involve a better RltXPoint reordering algorithm.

Figure 107 contains no noise and generates a visually perfect map. Figure 108 contains noise and looks similar to the results generated by the previous simpler simulation. The error is most likely due to planes over expanding during a turn when the motion model is on.

Overall the Relative Plane algorithm is shown to work as intended that while not completely accurate, it still produces good results.

5.3 Recommendations for future work

1. Adding a caching system to store in between calculations. The bottlenecks of the Relative algorithm is in calculating the interval in which to compare two planes and to calculate the position offsets of the four key point pairs and then compute the standard deviation. In these calculations, most of of the computation is identical to the iteration before it except for the new information. It should be possible to save the previous computation and simply add the new information and recompute the sums. The implementation is somewhat more tricky than it first appears as the cache system should be able to recall in between calculations based on pairs of planes and a given iteration, and operate with the roll back system. If necessary, this system should discard the last used cached information to free up memory.
2. Currently, only movement in the x and z axis is allowed and only rotation in the y axis is allowed. It should be possible to allow for full 6 degree of freedom movement. In order to do this four corner points per plane have to been used which would mean 16 comparisons. To make this four times increase of computation feasible the caching system should be implemented first.
3. Currently the RltXPoint RltPlane comparison ordering algorithm is only partially implemented since fully implementing it using a brute force method would require an O(n!) algorithm. It might be possible to arrive at a good enough algorithm that is computationally faster and more accurate than the $O(n^2)$ currently used. One possible solution is to sort each RltPlane according to its seen interval using an O(nlogn) sorting algorithm. Then the pairs can be chosen based on adjacent matches and if they are not parallel to each other. It should be possible to use this system to better choose which planes belong in each RltXPoint or better yet when an RltXPoint should be created. This can be done in hindsight coupled with the rollback system. Implementing a RltXPoint selection algorithm that works on past data could improve the map significantly, and reduce the interval problem. This might also eliminate the small error seen in Figure 105.
4. The original intention of this thesis is to use actual point data from a vision algorithm or a laser scanner. It is realized that those two sensors may work depending on the environment, however in an area with a low density of features, say a long hallway, it might be required to have a vision algorithm that can directly find regions/planes of interest, say a poster on a wall. In fact

any object should be able to be classified as a plane or a bounding box. This algorithm should work with any vision system that is able to classify an object consistently.

5. Adapt this algorithm to use in dynamic environments. This would require identifying objects with a higher standard deviation in key points comparisons and then classify them as dynamic. Because dynamic objects may move together a requirement would be to do multiple plane comparisons to identify them. This can lead to a worst case scenario that it is not possible to identify what is dynamic and what is static. In this case the vision system would have to classify objects.

5.4 Summary

It is not possible to directly compare the results to another algorithm such as the EKF or FastSLAM due to the uniqueness of this problem in terms of identifying planes that grow and shrink. In addition, to test this algorithm on real data, it is required to either have a vision system that is able to segment planes, or a point data set that has consistency in terms of allowing the EM algorithm to segment planes from it. Also, the current implementation only works on 2½D which may be insufficient using real data.

After the completion of the implementation of the Relative Plane algorithm there was an interesting observation. The goal of the Relative Plane algorithm is to accurately map a planar environment and to do this it is required to identify planes that are growing and shrinking. The Relative Plane algorithm both is able to filter out noise and identify dynamic edges of planes. The dynamic edge detection comes at a small computation cost and is inherent in the architecture of the algorithm.

It became evident that if the Relative Plane algorithm can both handle noise and a type of dynamic movement perhaps it can do something similar using points as the input. Chapter 6 The Relative Point Algorithm uses the Relative Plane algorithm as a template to implement a Relative algorithm using points. Since points are used, it can be directly compared against a six degrees of freedom (6D) no odometry EKF.

It is interesting to note that the Relative Point algorithm architecture benefits from the knowledge gained by implementing the Relative Plane algorithm. Most of the recommendations of the future work of the Relative Plane algorithm is directly designed into the architecture of the Relative Point algorithm.

Chapter 6 The Relative Point Algorithm

6.1 Introduction

The relative point algorithm is based on the Relative Plane algorithm that was originally implemented first. The main principle of both algorithms is that landmarks, be it points or planes are stored in their untransformed state. Most of the processing occurs using the relative positions of the untransformed landmarks. The untransformed landmarks are used to match subsequent observations of the same landmarks avoiding the use of current position. Landmarks that are observed together in the same time interval are placed into groups. In these groups, the relative locations of landmarks is computed by using the stored untransformed landmarks in the time interval where the landmarks are observed together. For each iteration before any comparisons can occur, the untransformed landmarks are transformed with a matrix for rotation and translation invariance. To maintain the invariance it is required to use three points or one plane as a basis. The difference of a landmark's position is then consistent regardless of the robot's viewpoint, and can be averaged from all the untransformed observations to obtain an accurate relative position inside of a group of landmarks.

To form a map from the relative position of many groups, each group's relative map is combined together. This is done by comparing landmarks that are present in more than one group. To align the map to the actual map, the observations seen at the first iteration are assumed to place the robot at the starting position of (0,0).

Notice that there is no mention of current position so far. In this algorithm, current position is not used as a state variable that is updated cumulatively every iteration. Instead it is generated by comparing the current observations of landmarks to the global map that is generated. Other than display purposes, current position is only used when a landmark has not been matched against a previous untransformed observation and is already present on the map. This occurs when backtracking or closing the loop.

When the initial untransformed matching fails, a landmark is transformed to global coordinates using the current position and compared to the global map. If this still fails to find a match then it is assumed to be a new landmark. In a sense, all the global matching is doing, is using the previously known relative relationship between landmarks and the last iteration's observations. Although when closing a loop global matching is still dependent on the overall accuracy of the map, backtracking is local and is invariant to global mapping errors.

The Relative Plane algorithm has additional responsibilities to identify if a plane is growing or shrinking. This is done by comparing the standard deviation of corner points over time to identify which ones are changing. Similarly the Relative Point algorithm has functionality to determine if any points are moving independently of the viewpoint and it decides whether or not to use them for mapping purposes.

The Relative Plane algorithm is developed for use in a 2½D simulation. The Relative Point algorithm is full 6D. It is able to distinguish both movement and rotation in all three dimensions. One of the main implementation differences, is that the plane version needs to identify planes that grow and shrink in different intervals. This identification requires splitting up the intervals used to calculate the relative positions. The interval splitting greatly increases the complexity of the algorithm. In the point version, either the point is visible or it is not. Rather than use pair wise comparisons as in the planar version, the point version uses three points as the basis for rotational invariance for the entire group. The planar version uses a formal linking of groups to generate the global map. The point version instead processes the groups in order of time first created and uses the Iterative Closest Point (ICP) algorithm to link together any points that have already been mapped from any other group. The point version is simpler than the plane version of the relative algorithm and this is reflected in the smaller sections to describe it in this chapter. The following sections make up this chapter:

- 6.2 Registering Points: describes the processes to store the untransformed points and match them to the following iteration observations.
- 6.3 Map Creation: describes the process of creating a map given that the points have already been organized into groups.
- 6.4 Group Creation: describes the process of decided which points should be formed into a group.
- 6.5 Other Functionality: describes the rest of the functionality of the algorithm such as: Global point matching, RltPoint merging, Dynamic point detection, Basis point optimizing, and Group optimizations.

Following the description of the algorithm, section 6.6 examines the properties of the algorithm.

- Section 6.6.1 looks into the main issue of accuracy performance in this algorithm, as well as how to improve the group selection. This section also compares different group creation constants versus the total landmark error.

- Section 6.6.2 examines the run time efficiency of the algorithm over both a long simulation run and with simulations varying the amount of points.

Section 6.6.1 looks at the algorithms accuracy with different constants for the group creation but there still is a question of how accurate this algorithm is, given that there is an unknown theoretical maximum accuracy given the noise in the data. To verify that the relative algorithm is reasonably accurate it is compared against a 6D EKF Slam algorithm that does not use odometry in section 6.7.

There is an additional accuracy comparison in subsection 6.9, Forty loops through the figure eight, where the error is evaluated after many runs through the same simulation. The section undertakes the question, as the quantity of iterations increases, does the error go towards zero?

Section 6.10 Comparison of the two relative algorithms, compares the relative point algorithm to the relative plane algorithm and notes the improvement. The relative point algorithm completes much of the relative plane's algorithm future work.

This chapter's conclusion is section 6.11.

6.2 Registering Points

At the start of every iteration the untransformed points as seen from the robot's viewpoint are matched to previously mapped points. Most algorithms use current position to transform points into their global positions and these are compared against globally matched points. Instead, this algorithm stores the previous iteration's untransformed points and uses those to do the matching.

Several (object oriented) classes are used to perform this operation: RltPointCharting, OVLPQuadTree, RltPoint, and Point.

6.2.1 Point

This class represents a single point in a single iteration. It stores a location (x,y,z) and the iteration it is observed in. Point contains a function to compare a point to another point to see how close it is. The comparison function compares two points distances versus a maximum bound constant and returns a number from 100 to -1 representing how close the match is. The comparison function can use the point's iterations so that there is a maximum allowed difference between their iterations for a positive match. Point also contains several utility functions for rendering.

6.2.2 RltPoint

A point that is on a map consists of many observations of a point seen in different iterations. The RltPoint class contains many observations (iterations) of the point stored in a circular array. Its main functionality involves a quick enumeration given a time interval. Since a point may be observed in several time intervals and may be missing in iterations inside of those intervals the enumeration requires a bit of logic. The enumeration consists of a O(logn) binary search to find the first iteration to enumerate and then a O(1) enumeration using the first iteration as a starting point. The RltPoint also contains: links to RltGroups described later, its globally mapped location, several boolean flags of its status and the interval range where it has been observed that is used for group creation.

6.2.3 RltPointCharting

Every point seen in an iteration is either added to a preexisting RltPoint or made into a new RltPoint. After the RltPoint has either been identified or created, a references is added to

RltPointCharting for the given iteration. This class stores every RltPoint that has been observed for each iteration. It uses a vector of RltPoint references for every iteration, and a circular array to store the vectors.

The Relative Point algorithm contains a single instance of RltPointCharting that is used to add RltPoints and to enumerate RltPoint from. For point matching, at the start of every iteration RltPointCharting is asked to generate a list of all points seen in the previous iteration(s). In addition to simply enumerating the last iteration's vector it can also enumerate over a range of iterations. This is useful since a point may be missing in several iterations in an interval. If only one iteration is returned that point may be missing. There is a limit of how many iterations can be used to look back, since the distance between untransformed points may be too great after a few iteration for a match. Global matching will be described later that matches points that have not been seen for a while.

It is important for computational efficiency reasons for the enumeration over several iterations not to return any duplicates. It is likely that each iteration contains nearly the same RltPoints so not removing duplicates would increase the matching processing cost. To do the enumeration as efficiently as possible, a hash table is used to store RltPoint's indexes. When a RltPoint is first enumerated it is added to the hash table that is then referenced for each subsequent RltPoint. Regardless of which iteration the RltPoint is found, the most recent untransformed point in it is the one used for point matching, not the untransformed point from the iteration the RltPoint is found in. Since the hash table duplicate check can be considered O(1) the creation of the enumeration list can then be considered being $O(n_s I)$ where I is the number of iterations checked. n_s is a subdivision of the total points on the map, which is the average amount of points visible in a given iteration. The point matching uses a small I of only a few iterations so it can be considered an $O(n_s)$ algorithm.

The RltPointCharting enumeration is used in several other places. When generating the current position it is used to enumerate each point seen in the current iteration to compare against their globally mapped locations. When creating groups, given an ungrouped point, it is used to enumerate points before and after the iteration where the ungrouped point is first seen to obtain points can be grouped with this point. It also used by the log to track how many points are seen in each iteration.

6.2.4 OVLPQuadTree

OVLPQuadTree stands for Overlapping Quadtree. A quadtree is a data structure [Wiki11a] which spatially divides a location into cells, each of which is a subdivision of the total amount of space. When doing point comparisons it is more efficient to compare against a small spatial area rather than against the entire area. If n points are to be compared against n previous points, the algorithm would be O(n^2). The quadtree can reduce this comparison to approximately O(nlogn).

The overlapping quadtree is a variant of a normal quadtree in which each cell overlaps neighbouring cells. If the cell's overlap is the same as the maximum bounding distance used for the point comparison, only one cell needs to be checked to guarantee all of the possible matching points are queried. This is more efficient than checking neighbouring cells since points in the neighbouring cells may not be within the matching bounds. The additional cost to create the overlapping structure is that when adding a point to the quadtree, it is added to the bin it matches without the overlap and then the point is checked against each neighbouring bin with their overlaps to see if it belongs in those bins.

6.2.5 The point matching

In the start of each iteration, the RltPointCharting structure is used to generate a list of all RltPoint in a given iteration range. This interval is only a few iterations in the past and given the duplicate detection with the hash table it can be considered an O(n_s) operation. The list is then enumerated and added to the OVLPQuadTree which is also O(n_s).

Each point that is seen in the current iteration is first compared against the OVLPQuadTree to see if it matches with any preexisting RltPoints and if so it is added to that RltPoint. If the point does not have a match after the OVLPQuadTree comparison, it is then checked against global matching described later. If the point still does not have any matches a new RltPoint is created. The checks against the quadtrees (both the local and global) can be considered O($n_s$logn_s). After a RltPoint is found or created ,it is added to the current iteration in RltPointCharting. The total computational complexity of this part of the algorithm is:

- O(n_s) for the duplicate check enumeration
- + O(n_s) to create the quadtree of previous seen RltPoints
- + O($n_s$logn_s) for every point to match against points in the quadtree
- + O($n_s$logn_s) for every point to match against points in the global quadtree if the previous check fails to find a match

The computational complexity of the point matching is $O(n_s) + O(n_s) + O(n_s \log n_s) + O(n_s \log n_s)$ with the worst case being global matching occurring for every point. The point matching is an $O(n_s \log n_s)$ algorithm provided the data structures used such as the hash table and quadtree do not degrade.

6.3 Map Creation

This section describes the process of generating a map given that all of the points are already grouped together. The next section 6.4 Group Creation describes the group creation process.

Given a group of RltPoints, three points in the group are chosen as the basis of the relative locations. Given this consistent basis it is possible to compute the relative distances between points for each iteration that is translation and rotation invariant. This allows the average of the distances to be valid regardless of the robot's changing viewpoint. If every group's relative map is combined, a global map is created that is aligned to the first observations of the first group. This section introduces the RltGroup, RltGroupRef, and RltICP classes.

6.3.1 RltGroupRef

Each RltPoint can belong to multiple groups. In order to keep track of this, the RltPoint stores a vector of RltGroupRef. A RltGroupRef stores a reference to the RltGroup it is linked to and the relative location of the RltPoint in that RltGroup. This structure also contains a boolean flag which specifies, if when the RltPoint is added to the RltGroup it is already grouped or not. This is described later in 6.4 Group Creation but for now this flag is used to determine if the RltPoint can be used as part of the ICP or not.

6.3.2 RltICP

This class performs an Iterative Closest Point between two sets of point clouds to determine the transform to transform one of the point cloud's location to the other's. It requires a minimum of three points to solve the transform. This version of the algorithm assumes the points in the two point clouds are already matched together which they are guaranteed to be at this point. This allows only one pass through the algorithm. The algorithm [BeMc92] first removes the mean from the two points clouds. It then forms the covariance matrix between the two clouds and uses the Singular Value Decomposition (SVD) to solve the rotation. The translation is then solved by rotating the second mean by the solved rotation and subtracting from the first mean.

6.3.3 RltGroup

The RltGroup class stores a reference to every RltPoint that is contained in the group. When a RltPoint is matched against an untransformed point, it notifies every RltGroup it is linked in that there is the potential of a new iteration to be used for its computation. The RltGroup updates its update interval with the new iteration, and places itself on a list of RltGroups to be computed in the current iteration.

The RltGroup is used slightly differently than the planar version of the algorithm. In this version of the algorithm, a RltGroup only performs a computation if every RltPoint has a valid observation. RltGroups also do not undergo any changes once they are constructed. If a change is required, the RltGroup is removed and a new one is added. This allows the RltPoint processing to be simple. Since it can be assumed that the RltPoints inside of the RltGroup are always the same, it can safely store a running average of each RltPoint's relative location in the group. It is also known that the three basis points (Figure 109) never change and that any dynamic points are already removed during the group creation. This eliminates the needs of a cache system to speed up computation or a system to save in between calculations that are required in the planar version.

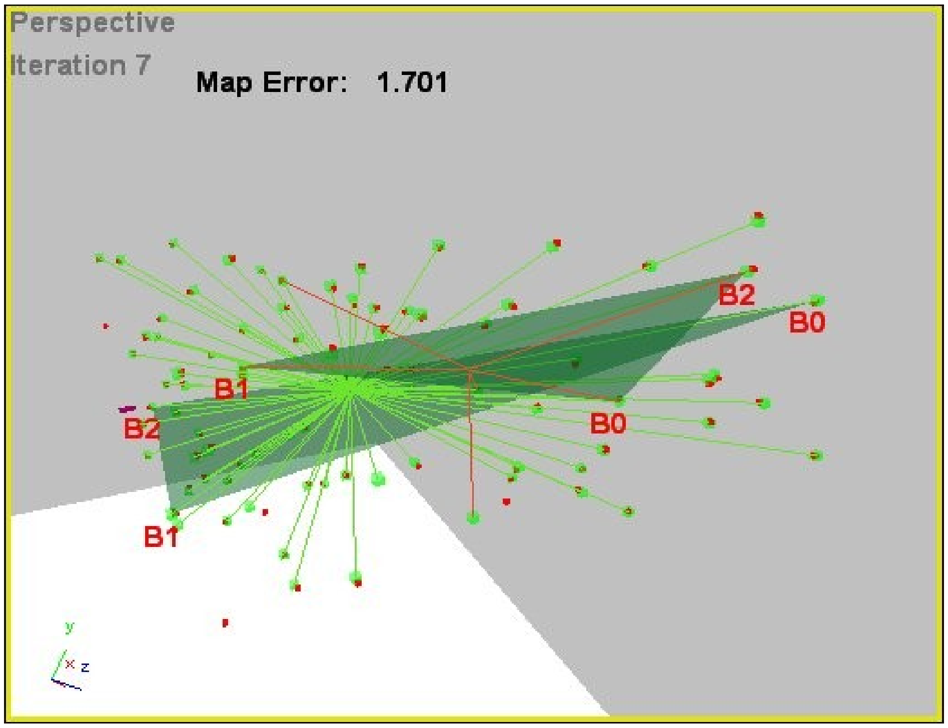


Figure 109: Illustration of the basis points of two groups. B0 is the (0,0,0) point B1 on the -x axis and B2 on the xz plane.

The main function in RltGroup is the ComputeRltGroup. The algorithm is as follows:

1. Determine if there is at least one RltPoint update for the current iteration by querying the update interval structure. There might be a range of iterations if the group is created in the current iteration.
2. For each iteration, obtain the untransformed point from every RltPoint in the group, and add to its relative location average. If even one RltPoint is missing a point for the iteration, skip the iteration. To calculate the average relative location:
    - Form the transform matrix for the three basis points so that the first point is transformed to (0,0,0) the second one transformed to being on the negative x axis and the third point transformed so that it on the xz plane. This is shown in Figure 109. Note that after the points are transformed into global coordinates the plane formed by the three points is no longer aligned to the axes.
    - Transform every point by the matrix. Add the x,y,z location to the average for that point.

    - Add the iteration to the calculated on interval variable.

3. For each RltPoint, divide the total location difference by the amount of observations to obtain the average. Place the average into the RltGroupRef structure from the RltPoint for this RltGroup.
4. Perform the ICP to generate the global coordinates.
    - If the group is created at iteration 0, then use the RltPoint's first observations in one point cloud and their relative position in the group in the other.
    - If the group is not created at iteration 0, it must have been created with at least a minimum number of RltPoints from previous groups. Use those RltPoint's RltGroupRef for this RltGroup in one point cloud and their global location in the other point cloud to solve the ICP. Then for the remaining points use the matrix returned by the ICP to find their global position. The RltPoints from the previous group do not have their global position recalculated.

6.3.4 Summary

It is straight forward to find the relative location of the RltPoints inside of a group. In terms of computation per iteration, first the transform matrix is found from the three basis points. Then for all of the RltGroup's RltPoints, their untransformed observations for that iteration are transformed by the transform matrix and then added to that points average relative location. This is an $O(n_s)$ operation. The ICP takes $O(n_s)$ to form the covariance matrix and then $O(1)$ to calculate the SVD. The algorithm is $O(n_s) + O(n_s) + O(1)$, thus $O(n_s)$ as a worst case.

The main difference between the point and planar version of the relative algorithm is that the planar version has extra logic due to the fact that the plane comparisons are pair wise. There is additional logic in case the pairings are changed and to account for different calculation intervals for each pair. For the point version the interval is the entire interval and any reassignments is taken care of higher up in the architecture. This simplifies the RltGroup class.

6.4 Group Creation

Since a robot's viewpoint moves and points enter and leave the view, groups are required to organize points that are seen in the same time interval. How optimally the groups are created in terms of maximizing the quantity of iterations in every group is crucial to the performance of this algorithm. This section only covers the creation algorithm. The performance analysis is in a later section, 6.6.1 Odometry analysis in the figure eight simulation. This section describes the RltMapper, RltInterval and RltUngroupedList classes.

6.4.1 RltInterval

A given RltPoint can be valid in different intervals, so the intervals are stored by start, end iteration and run length. Each interval is stored in a vector inside of RltInterval. When a point is added to a RltPoint the iteration is added to the RltInterval. To do this efficiently the RltInterval only allows iterations to be added in sequence as to not have a search to find where the iteration is supposed to go in the vector.

The main functionality of the RltInterval is used in the RltMapper when forming groups. There is a function to calculate the number of untransformed points a RltPoint has in a given interval. It is much quicker to reference the interval structure than to do a manual count of a RltPoint's untransformed point circular array.

6.4.2 RltUngroupedList

The RltUngroupedList stores a list of RltPoints. When a RltPoint is newly created it is added to this list. This list is then referenced in the group creation to see if any RltPoints are ungrouped. The group creation is delayed slightly to allow for the RltPoint to accumulate untransformed points. The RltUngroupedList is used in several places to allow for delayed processing on RltPoints. Having a list structure hold RltPoints used for future processing rather than doing an enumeration over the main RltPoint list keeps the relative algorithm only working on n_s points rather than all of them.

The RltUngroupedList maintains a sorted list, so the group creation checks the earliest created RltPoint first. After a RltPoint is placed in a group that RltPoint is removed from the list. There is a maintenance function to remove RltPoints that have not been able to be placed in groups after a certain amount of time. Due to the potential removal of RltPoints, this data structure is the only one to use a standard template library (STL) list. Most of the project was originally developed using STL lists but then converted to STL vectors for performance reasons.

6.4.3 RltMapper

In addition to performing group creation, the RltMapper class performs much of the high level functionality of the Relative Point algorithm. This class is used to perform the initializing of each iteration which builds the matching quadtree. RltMapper is then used to add each new untransformed point to the algorithm and either performs or calls other classes to do the remaining processing.

The class also contains the master storage vectors of RltPoints and RltGroups. Storing all the RltPoints and RltGroups in a master list simplifies the architecture. If the RltPoints were only referenced in RltGroups then when RltGroups are changed it would be difficult to figure out if the RltPoint should be deleted or not. Having a master list allows for a garbage collection routine. However currently there are no memory issues, so no garbage collection is performed.

Currently there are two types of groups that are created. The initial group is created after only a few iterations and the final group is created after many iterations. Since the initial group is only created after a few iterations it is not possible to determine the best points to include in it. The final group is created after many iterations so that the previous points that are added maximize the amount of iterations available for group processing. There is a look back constant that determines how many iterations of untransformed planes a RltPoint has to contain to consider adding it to the group. The group creation algorithm is:

1. Look at the ungrouped list to see if there are any RltPoints in it that have sufficient untransformed points to be considered to be placed in a group. If there are no valid RltPoints then exit. If there are valid RltPoints extract the first RltPoint. This RltPoint's creation iteration is the iteration used in the rest of the algorithm.
2. Use RltPointCharting to form a list of all RltPoints that exist a look back amount of iterations before and after the creation iteration.
3. For every RltPoint in this list that is already mapped:
   - Find out the quantity of iterations in the interval between the creation iterations plus

and minus the look back. Add this number to an array.

- After all of the RltPoints have been evaluated sort the mapped RltPoints by their count of iterations in the interval.
- Create a previously mapped list out of the sorted list starting from the ones with the highest total count. There is a minimum threshold for the iteration count. There is also a maximum number of previous points constant and minimum number of previous points constant. If the minimum is not met, remove the creation RltPoint from the ungrouped list as it is determined that it is a not a good point to be added to the map and exit.

4. Restart the RltPointCharting enumeration and enumerate every ungrouped point.
   - Calculate the total number of iterations that is present in the interval from the creation iteration to the creation iteration plus the look back.
   - Sort the list and determine which ones should be added to the next list using a threshold constant. The minimum is one and there is no maximum.
5. Combine the previously grouped list and the ungrouped list to one list and run the dynamic bin checking as described later in 6.5.3 Dynamic point detection. Only continue if there are sufficient non dynamic points.
6. Find the optimum three points to use as the basis when calculating the relative location described in 6.5.4 Basis point optimizing.
7. Create a new group with the combine list. Label the RltPoints that are previously grouped as such so that they are used with their globally mapped location for the ICP calculation. Remove any ungrouped RltPoint from the ungrouped list that has been added to a RltGroup.

6.4.4 Summary

The grouping algorithm works well given a range where it can look back and look forward. It is possible to change this constant for accuracy purposes and there is analysis in 6.6.1 Odometry analysis in the figure eight simulation. In terms of computationally efficiency:

- $O(n_s)$ for the point charting 6.2.3 RltPointCharting. If there are a large number of iterations it has to look over, it would be considered $O(n_sI)$. However in this case I is set to a small amount.
- $O(n_s\log n_s)$ for the sorting of the RltPoints for both the previous grouped and ungrouped list. It is likely that the n_s is smaller than the RltPointChart enumeration so the sorting is not a significant part of the the algorithm.
- The Dynamic point binning 6.5.3 Dynamic point detection is considered to be $O(n_s\log n_s)$ average case and $O(n^2)$ worse case. There is considerable analysis of its performance in that section, and generally the dynamic point detection performs at $O(n_s\log n_s)$ or better.
- Finding the basis points 6.5.4 Basis point optimizing is unoptimized at $O(n_s^2)$ but since it has an insignificant run time it is left this way. If thousands of points are in a single group this would have to be optimized.

The grouping algorithm's computational complexity in the average case is hard to estimate. It is demonstrated to be $O(2n_s)$ in Figure 141 in 6.6.2 Performance testing, which is still $O(n_s)$.

6.5 Other Functionality

6.5.1 Global point matching

When the robot backtracks or completes a loop, previously mapped points are observed. First the previously seen untransformed points are compared to the ones seen in the last few iterations. After a match is not found the point is then compared to globally mapped RltPoints. This requires the untransformed point to be first transformed using current position then compared to RltPoints stored in the global location quadtree. If there is a match the untransformed point is added to the matching RltPoint and will be correctly matched to that RltPoint in future iterations using the normal point matching found in section 6.2 Registering Points.

At first this algorithm may seem $O(n_s \log n_s)$ with the use of 6.2.4 OVLPQuadTree. However if the global quad tree is created the same way as the untransformed version in 6.2 Registering Points it will require loading n_t points every iteration, where n_t is the total number of points on the map. If working with n_t is not improved upon then the global point matching will be be dominated by the $O(n_t)$ initialization which is not desired.

The OVLPQuadTree is altered so that it is possible to add and remove RltPoints incrementally, rather than creating a new global OVLPQuadTree every iteration. There is a higher overhead cost to doing this but it is quickly overtaken by the larger cost of reconstructing the global OVLPQuadTree with every iteration. This is shown in Figure 110 that shows the difference of computation time in using the incremental versus the normal OVLPQuadTree. In the chart, the robot revisits the starting point at about iteration 4000 and closes the loop and no new RltPoint are created after that. The figure shows that provided new RltPoint are being added, the non incremental version computational time increases linearly in time. In the simulation after the loop is closed the average run time is about 5ms of which the the non incremental versions initialization takes a noticeable percentage of this time.

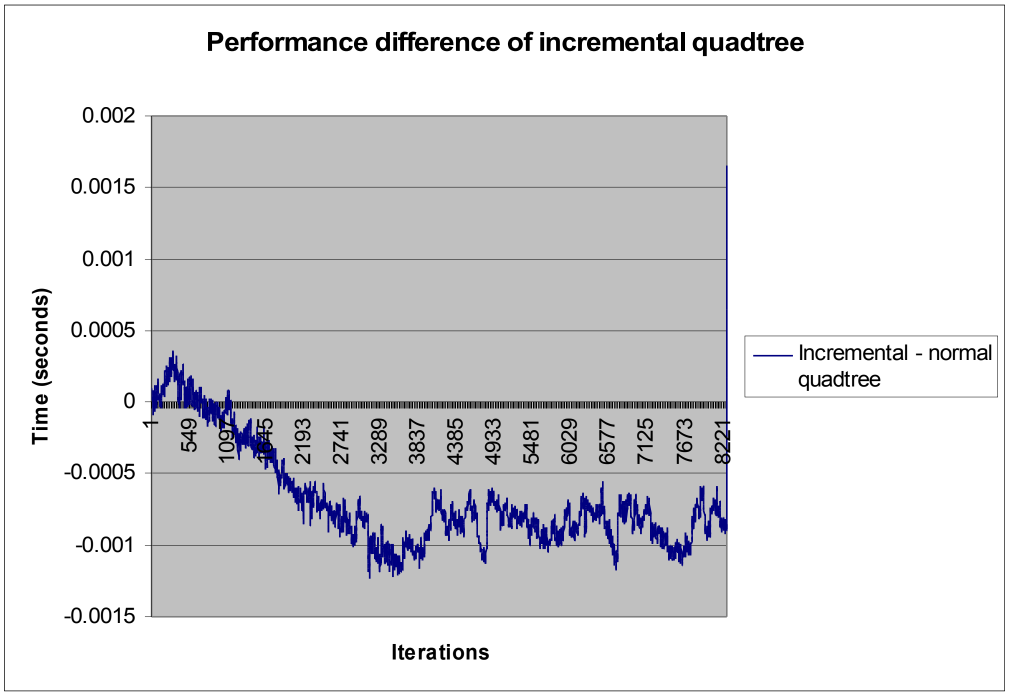


Figure 110: Difference between the incremental and normal quadtree. Negative means faster run time for the incremental quadtree

6.5.2 RltPoint merging

The global matching uses the nearest neighbours approach with a bounding constant. When an untransformed point is very far away, a small angular difference can lead to the untransformed point not being matched. It is possible to increase the bounding constant, but it is there to prevent points from being globally matched to the wrong RltPoints. Another solution is to allow for merging. Using the same RltUngroupedList as before, new RltPoints are evaluated after a constant amount of iterations to see if their global locations match to another RltPoint. If they match, then the two can be merged. The merge consists of copying all of the untransformed points of the newer RltPoint to the older one and then telling the RltGroup to recompute over the copied interval range. In practice, after a few iterations, RltPoints created with the same points are in nearly the same location and the merge seems to work consistently.

There is an issue related to dynamic points. When a dynamic point is globally mapped, its position changes as the point moves around. This may cause a false merge to occur. Currently this is not a big issue since the dynamic point will move on leaving the merged RltPoint to have about the same average. In addition, the dynamic binning may remove the merged RltPoint. There may be a way to avoid this by either having the merging take place after the dynamic check or having a roll back mechanism to fix this issue.

Currently neither of these are implemented. There is a slightly larger error when there are dynamic points however it is not known if this is due to merging or due to dynamic points getting added to RltPoints for a few iteration when their paths intersects. It is not known if this will become an issue when using real data. Real data may not contain dynamic points intersecting normal ones so for now this issue is not pursued further.

6.5.3 Dynamic point detection

It is possible to add in at a low cost dynamic point detection to the algorithm. The goal of the dynamic point detection algorithm is to detect which points in a group are static and which ones are moving around or have a high standard deviation compared to the rest of the points. This would be a simple problem if there is a stationary point as a reference. The stationary point can be used as a basis and a dynamic check could be done in O(n) time comparing every point to the stationary one. However without knowing which points are stationary and which are dynamic, without an optimized algorithm the dynamic check would require O(n^2) time. Every point would need to be compared against every other point to figure out which ones are correlated. Of groups that are correlated, one group, perhaps the largest has to be chosen as the static group. The algorithm used in the RltBinning class uses an approximation of an O(n^2) correlation for dynamic detection. The approximation is that points that are correlated can be grouped or binned together and only one point in the bin has to be compared against other bins for subsequent correlation computations.

The RltBinning class receives a list of RltPoints towards the end of the RltGroup creation. Its goal is to use the stored untransformed planes over an iteration range to find out which RltPoints are dynamic and which are not. Each RltPoint is compared in pairs to other RltPoints using the standard deviation of the distance between their untransformed points over an iteration range. If the standard deviation is higher than a threshold, then one of the RltPoints is dynamic. If it is lower, then the RltPoints are considered to be correlated and the bins are combined. Every bin has a hash table of every RltPoint that it has already been compared to, and this hash table is also merged when the bins are combined. When every bin has been compared to every other bin then the algorithm is complete.

If there is a log(n) number of bins compared to n RltPoints at the end of the computation, then the algorithm is O(nlogn). If there are many uncorrelated points, then the algorithm degrades to O(n^2).

There is a optimization that takes into account that the algorithm is only interested in finding the bin with the most points. When one bin has a higher than a threshold percentage of the total points, the pairwise comparisons can stop. Instead of continuing comparing every bin to each other, the bin with the highest number of total point can be compared against every remaining bin. Any RltPoint that does not belong into the highest total bin can be considered dynamic.

There is the question of how to chose the standard deviation threshold. Instead of doing this manually with a constant, the threshold can be generated automatically at a low cost. The first round of comparisons occurs normally until every pairs standard deviation is calculated. Then instead of comparing against a threshold, a function is called to decide the threshold. The standard deviations are sorted and then traversed from the lowest to the highest. When the difference between two standard deviations exceeds a percentage and a constant value, the lower of those standard deviations is determined to be the threshold.

The algorithm is as follows:

1 To start, place every RltPoint into a separate bin. The bin contains references to RltPoints and a list and hash table of RltPoints that the bin has already been compared to. Place all the bins into a valid list.

2 Perform pair wise comparisons until a finishing condition that either all valid bins have been compared to all other valid bins, or one bins RltPoint count exceeds a threshold.

3 For each bin in the valid list, find another bin to compare to.

- Before comparing bins, check to see if the first RltPoint in the first bins RltPoint list has already been compared to any RltPoint in the second bin. This is done by referencing the hash table of the second bin with the index of first RltPoint.
- If it has already been compared to, keep on enumerating the valid list to find a comparison pair. If no pair is available then this bin is done comparing and by not adding it to next valid list it will not be used for comparisons in the following iteration

4 If a bin is found, do the comparison between the first RltPoints of each bin

- Find the end iterations of the comparison by looking at the intervals of each RltPoint. Use the RltGroup creation iteration as the start of the comparison.
- For each iteration, obtain the untransformed point, calculate and save the distance and add to the average distance. After the average is calculated, do another pass to calculate the standard deviation.
- If the threshold has not been calculated yet, exit at this point. The threshold calculation function is then called. After the threshold is calculated the algorithm resumes at this point.
- If the standard deviation is below the threshold then merge the two bins and merge the hash table. Set the second bin to being not valid.
- If the standard deviation is above the threshold, merge the hash tables so the bins or any bins that merge with the two bins are not compared to each other again.

5 If the threshold has not been calculated, calculate the threshold by:

- Querying the bin's saved standard deviation,
- Sort the saved standard deviations from lowest to highest.
- Enumerating from lowest to highest to pick a threshold after the difference in two adjacent standard deviations are higher than a constant and percentage.
- After the threshold is found go back and complete the RltPoint pair comparison function for the first iteration.

6 Bins that have been compared to and successfully merged can be added to the next valid list.

7 At the end of the iteration, check to see if one bin has more RltPoints than the threshold. If this is the case the binning can end after one more pass of comparing every remaining valid bin that has not been compared to the highest bin, to the highest bin.

8 Copy the valid list from the next valid list and iterate again.

To evaluate the performance of the binning consider the best and worse cases. The best case is if every RltPoint goes into one bin then the algorithm performs n-1 comparisons. If every RltPoint stays in its original bin the algorithm will require $n^2/2$ comparisons. If there is a percentage of RltPoints that are dynamic then the algorithm will be in between. The computation complexity between the best case and worse case is displayed in Figure 111. This figure shows the total number of comparisons given the number of points and percentage of points with a high standard deviation.

Sometimes the numbers especially on the lower percentage are not increasing at a consistent rate. This is probably due to the clustering of the dynamic points versus the normal ones. Perhaps for some runs the highest count bin gets larger quicker than others so that run executes with less comparisons. Note that 90% line is inconsistent. What is happening is that in the 90% dynamic point test, the function that finds the standard deviation threshold does not work correctly since the data feeding into it does not contain enough low standard deviations. This causes the threshold to be set too high and all RltPoints to be binned in the high group. This effect is also noticeable at lower percentage with a smaller amount of points. Figure 112 Shows how many RltPoints per percentage are binned in the highest bin. Clearly the 90% line in that figure is incorrect. In fact the 70% line is incorrect for 20 and 40 points but correct for 60 points and up. The 100% line is artificial as the binning would not work due to the standard deviation being set too high so it is set as the worst case of $n^2/2$.

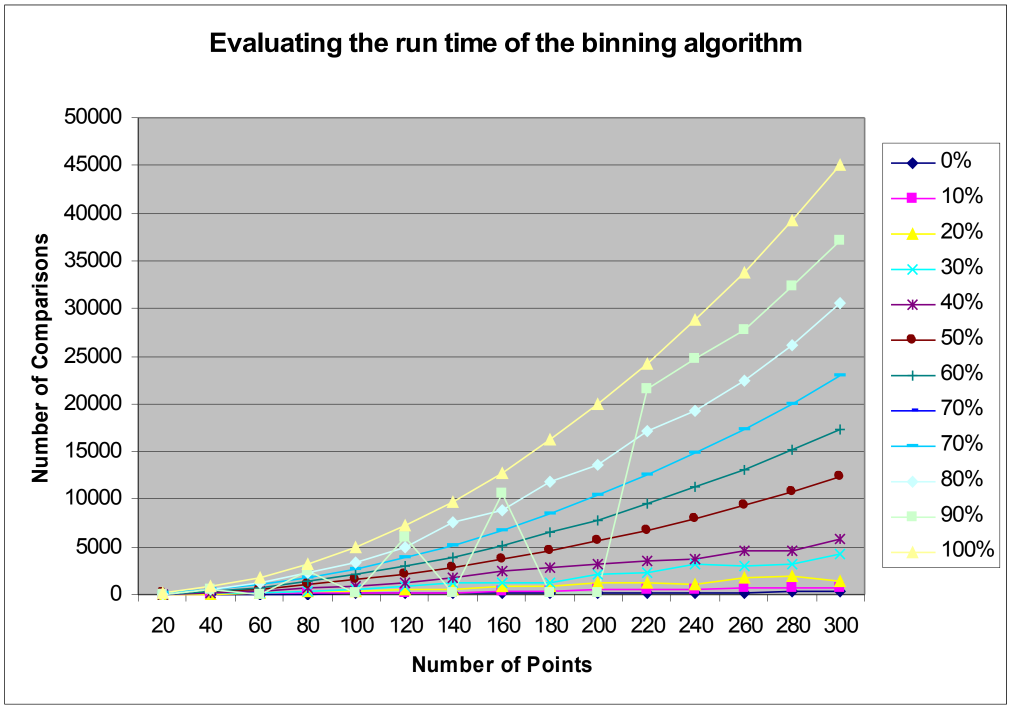


Figure 111: Verifying the run time complexity of the binning algorithm

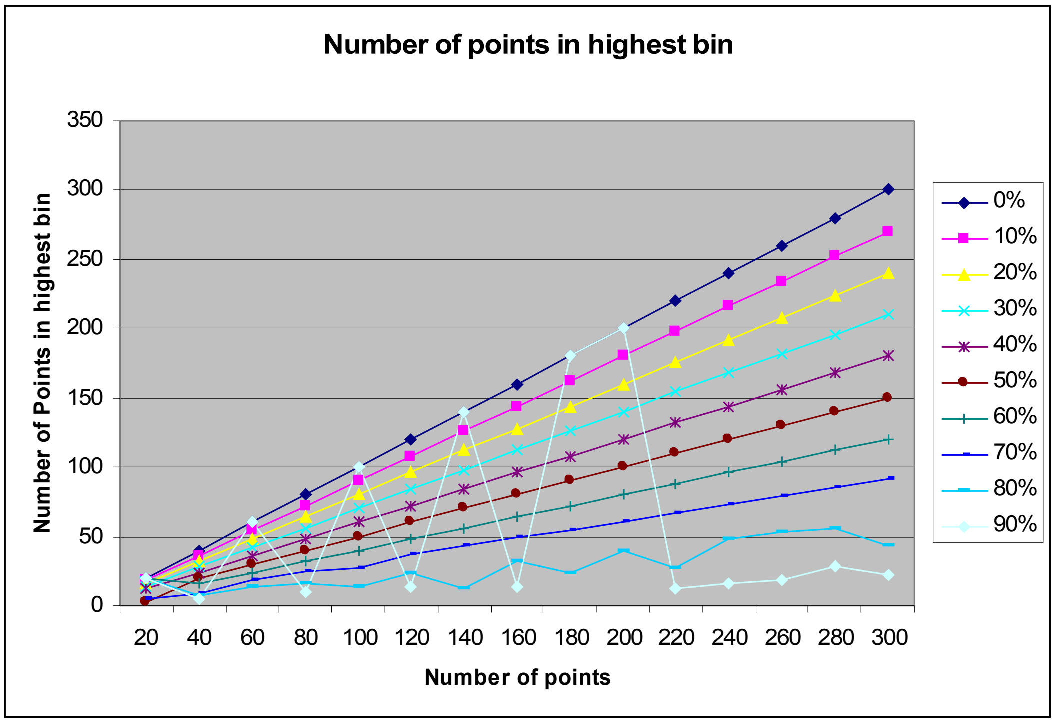


Figure 112: Number of points in highest bin

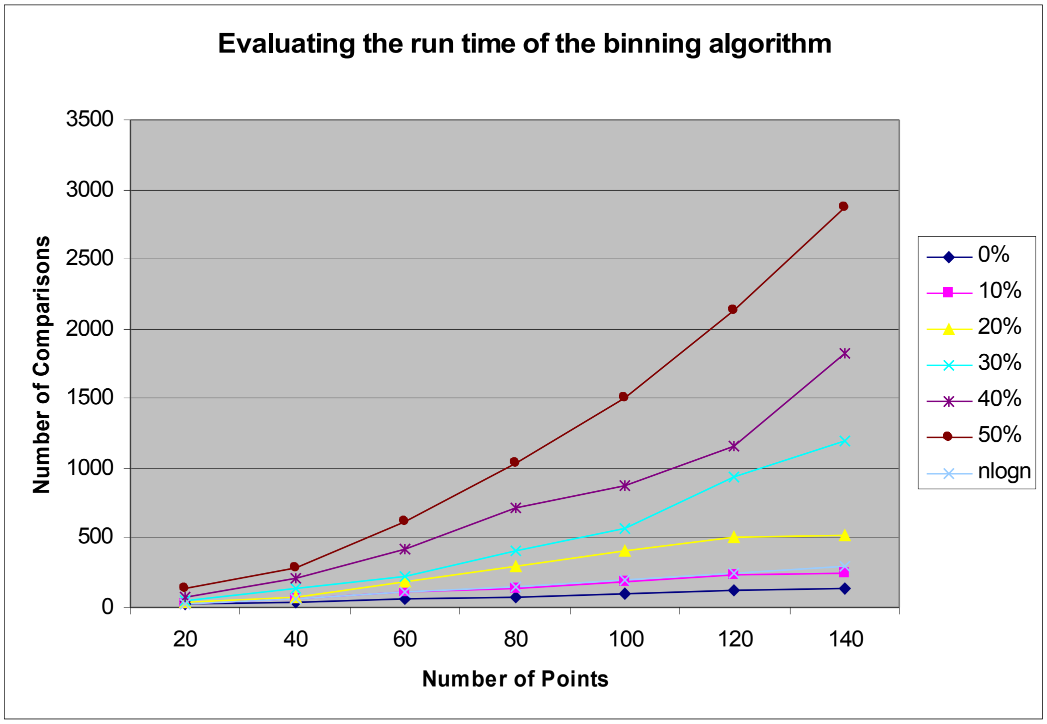


Figure 113: Evaluating the run time of the binning algorithm

So the question is where is the binning algorithm O(nlogn) and where is it worse than that? In Figure 113 it appears that there is the start of polynomial growth between the 20% and 30% line with the 10% noise line being nearly identical to the O(nlogn) line.

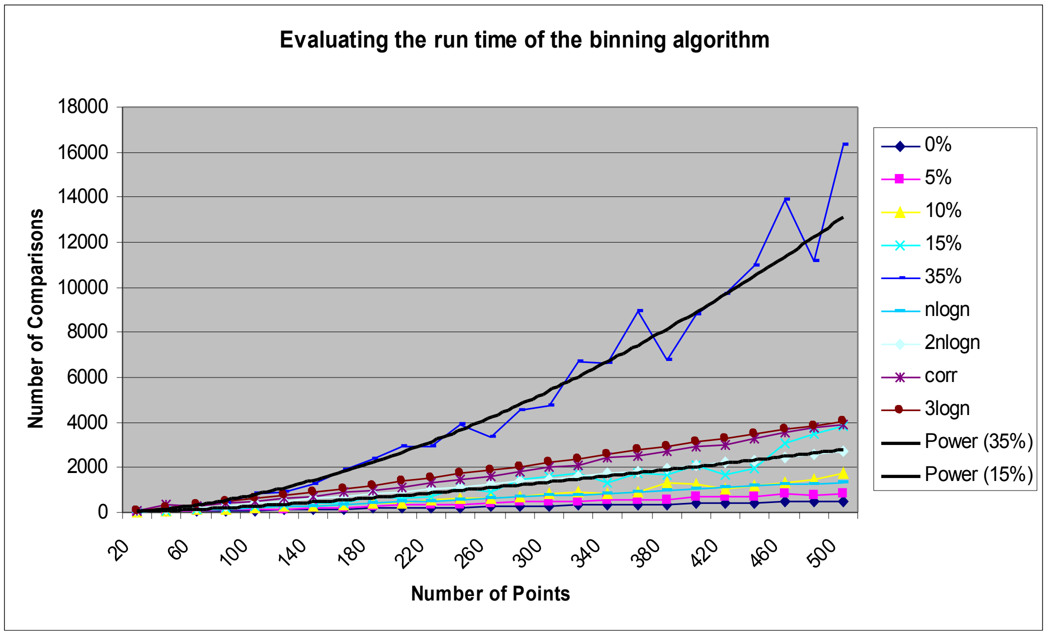


Figure 114: Performance with 50% normal 10% noise and 4 bins of 10% correlated noise

The 10% noise threshold is still approximately valid given correlated dynamic movement. This can occur when a group of points are attributed to an object that is moving in the viewpoint. The grouped points would have a high standard deviation compared to static points but they would have correlated movement inside of the group. Since the standard deviation between these points would be small they would be binned together causing the algorithm to complete faster. Figure 114 contains some of the same data as Figure 111 with the addition of the “corr” line. This line consists of 50% of the points belonging to the normal group, 10% being points that move with the viewpoint but have a high standard deviation and 4 bins of 10% each representing 4 objects that are moving in the viewpoint. While the correlation test has 50% of the points being considered as noise the fact that most of the end up in a bin together speeds up the computation. The “corr” line tracks the 3nlogn line.

It would appear that the binning algorithm is approximately O(nlogn) given that a maximum of

10% of points can belong to uncorrelated noise. It was thought that this was as good as the algorithm can perform until a simple heuristic is used. Before the addition of the heuristic, in each iteration, every bin gets to compare itself to another bin. The heuristic is to use a kind of priority queue to chose which bins are used to do the comparisons.

Perhaps the best priority mechanism would be to use the quantity of RltPoint in each bins. Given the current architecture of the algorithm, a simple method is used that has good results. The addition to the algorithm are as follows:

- For every iteration other than the first, place all bins that have a positive comparison into the first next valid list. Place all bins that have a negative comparison to the second next valid list.
- At the end of the iteration, combine the two lists into the main valid list, however copy all of the first lists members before the second. It is important to combine the list rather than not using the second list since members in the second list may have positive comparisons later.
- If there are any members in the first list, set an iteration stop flag to the number of points in the first list. If the first list is empty, set the stop flag to the total amount of points.
- During the next iteration, end the comparisons after the stop flag is reached so only the members of the first list are used to initiate comparisons. Note that points from the first list can still be compared to points from second list. If the first list is empty, the entire second list is used for the comparisons.

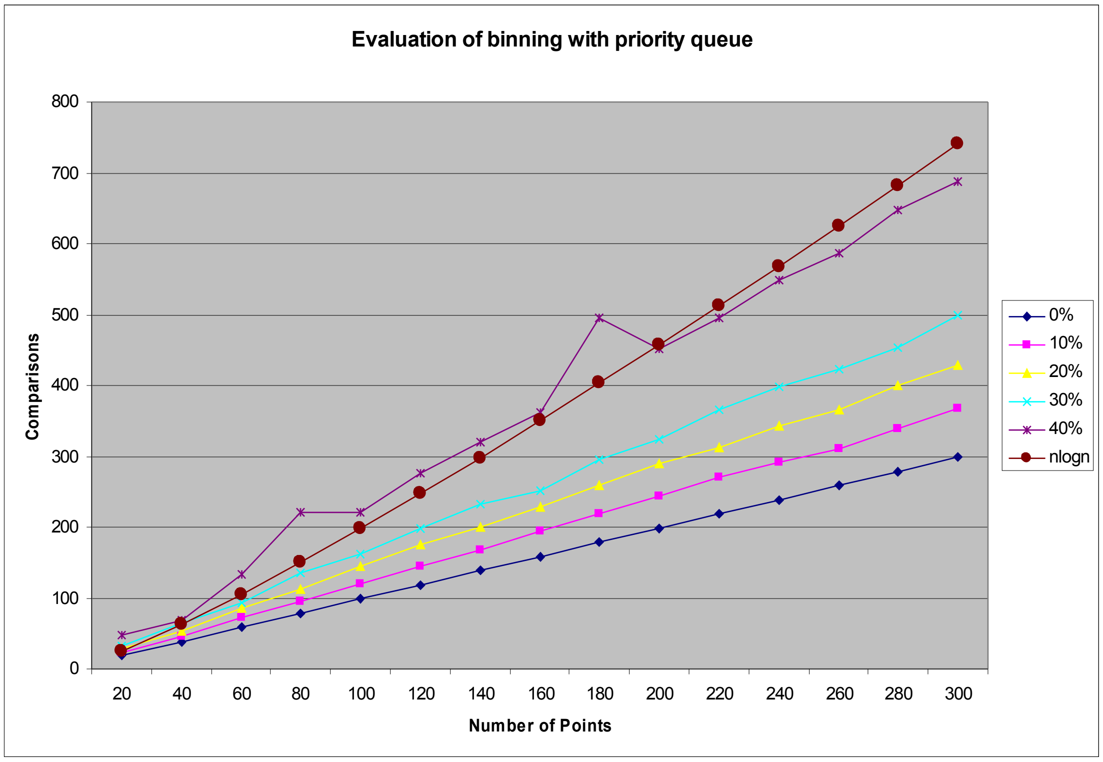


Figure 115: Performance of binning using priority queue

The performance improvement shown in Figure 115 using the priority heuristic is astounding. The figure uses 50% as the maximum bin threshold. The 50% line cannot be shown since the 50% threshold is not reached so the algorithm degrades to performing every pair wise comparison. Charting the 50% line would make the rest of the lines hard to see. Notice that nearly every line is at or below the O(nlogn) line. The 0% error line can be considered the O(n) line. Figure 116 examines the threshold of the heuristic by charting lines between 40%-50% noise. It is not until the 47% line where the algorithm becomes significantly worse than O(nlogn).

The binning algorithm can be considered O(nlogn) if the percentage of uncorrelated noise is a few percentage points below the threshold for the highest bin. In terms of total computation time Figure 117 compares two runs of the algorithm one with binning and 3% noise and one without binning and no noise. The linear trend line is near zero and the extra cost of binning is on average .048ms where the algorithm run time is on average is 3.5ms. The total binning time is below 2% of the total execution time.

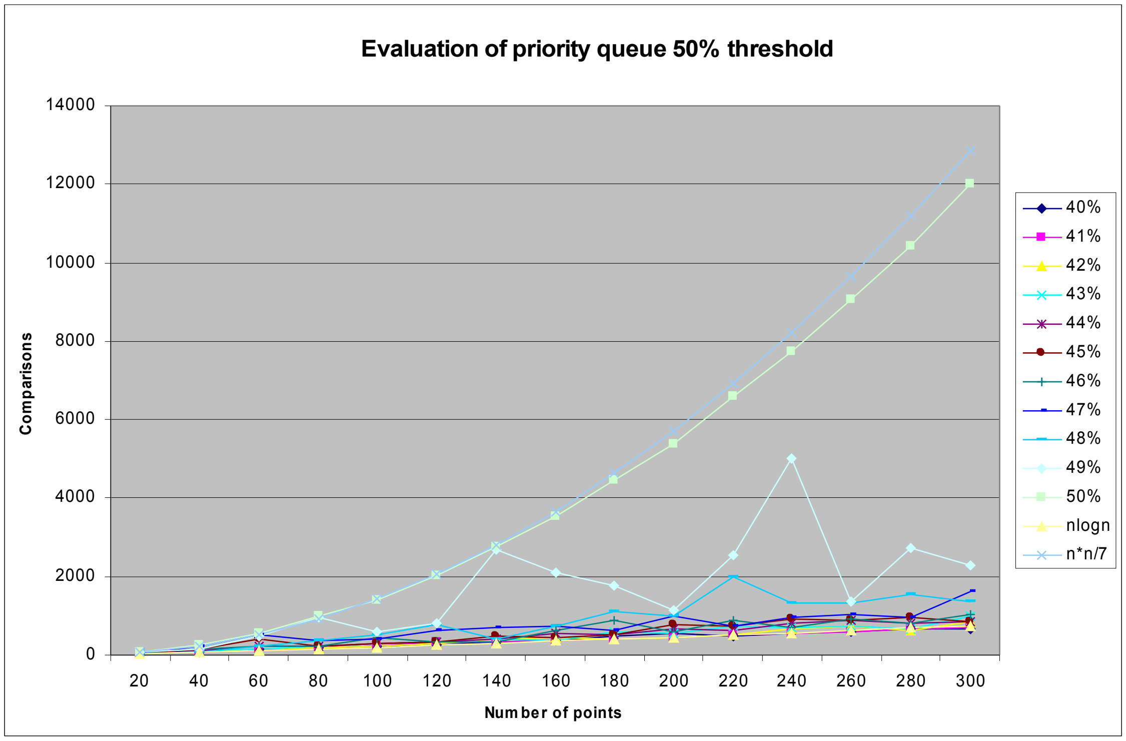


Figure 116: Evaluation of priority queue with a 50% threshold

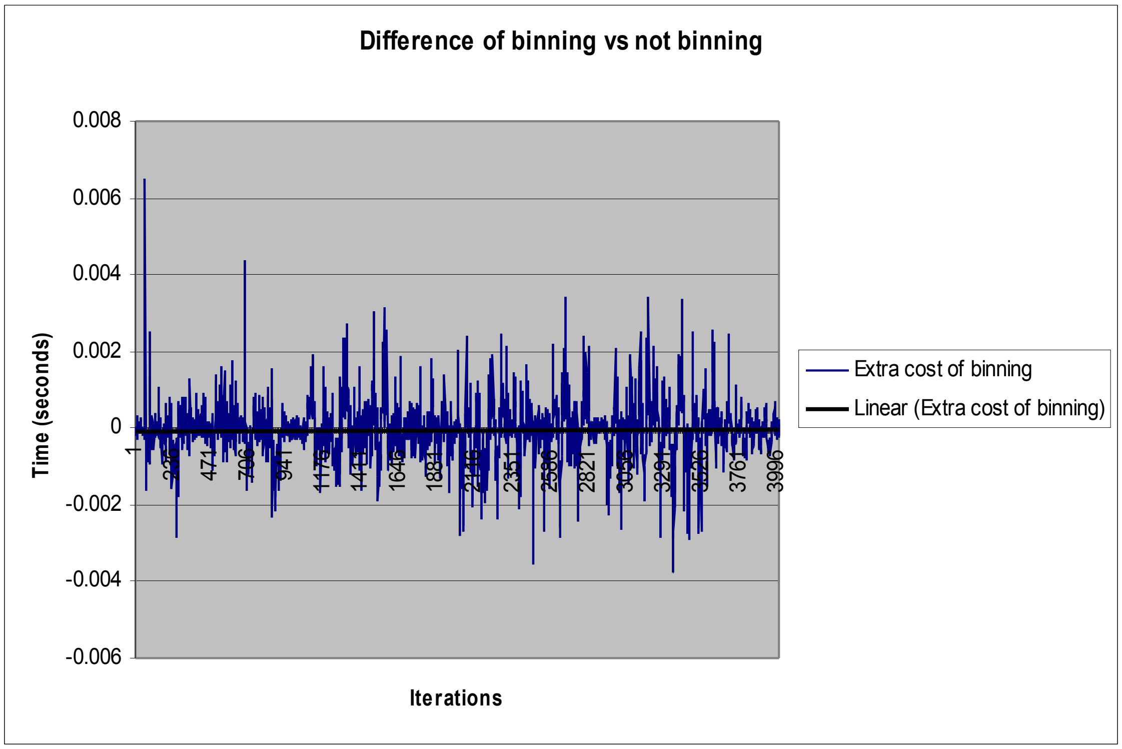


Figure 117: Extra computation cost of binning

One potential improvement to dynamic point detection is to be continuously checking for dynamic points rather than just check once at group creation time. Currently in the simulation either a point is moving or it is not so this is not an issue. It may be possible to calculate standard deviations at a low cost every iteration after computing the average relative positions. If a dynamic point is found it can be removed from the group and if there are insufficient points the group will have to be recreated with other points. If there are many RltPoint in the RltGroup that suddenly have a high standard deviation than the binning algorithm can be run again. Perhaps this feature could be implemented with group creation auto tuning (discussed as future work in 6.5.5Group optimizations) to make the algorithm more robust to changing conditions.

6.5.4 Basis point optimizing

When computing the relative location of a group, three points are used as the basis of the computation (Figure 118). These three points have to be chosen carefully. If the three points are collinear they will be missing a rotational component and will not be rotational invariant. As three points approach being collinear, noise in the readings will cause the basis to degrade. It is important to chose the three points well.

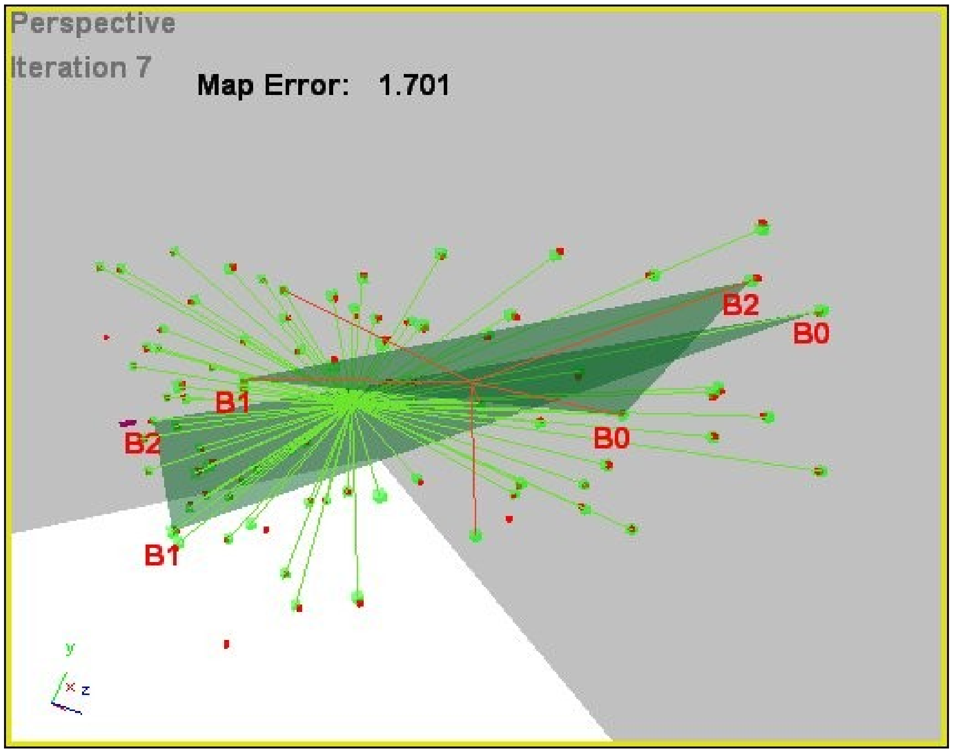


Figure 118: Showing the basis points, this is the same picture as in Figure 109

When developing the algorithm in stages, an early version of the algorithm placed the basis points in order of arrival so the first three points added to the RltGroup would become the basis points. This mostly worked with only occasionally there being three points that were collinear or close to being collinear. It was observed those groups had inconsistent relative locations.

The algorithm used to chose the three basis points is as follows:

1. Find the two points that are furthest apart in the group by doing an $O(n^2)$ search comparing every RltPoint to every other RltPoint.
2. Combine the two furthest points into a vector and do a O(n) search to find the point with the largest distance to the vector. These three points are then assigned as the basis points.

The basis point algorithm is designed to find the three best points to be the basis. Unfortunately it is also $O(n^2)$. However due to the group size and low amount of computation per comparison the algorithm is not a significant amount of the processing time so it is left unoptimized. One possible optimization would be to find the bounding box of the RltPoint in the group and then chose the RltPoint closest to the corners instead of doing the $O(n^2)$ search. This algorithm would run in O(n) time but perhaps due to setup time may take longer with small groups.

Figure 119 shows the mapping error for the two basis point selection routines. The random selection uses the first three points in the group as the basis points. The optimized selection has the basis points chosen using the selection algorithm. The basis point optimization greatly reduces the mapping error.

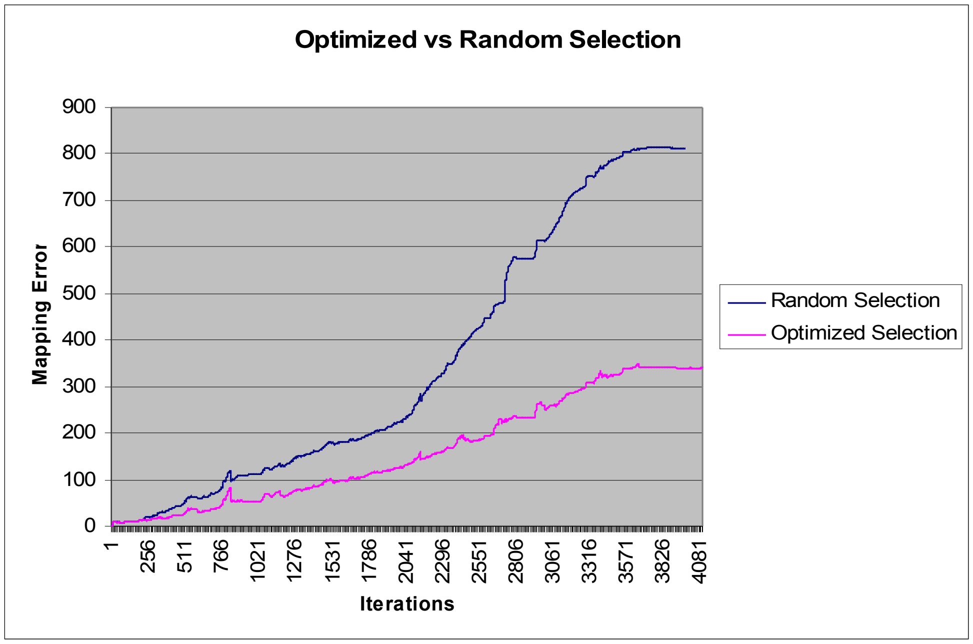


Figure 119: Comparing the mapping error of the basis point selection

6.5.5 Group optimizations

One of the largest issues of this algorithm is how to group together points that maximize the group interval size that then minimizes the noise. Section 6.4 Group Creation describes the algorithm for the group creation. This section elaborates on the details of the creation algorithm and charts the performance of using different constant values in the algorithm. All testing is done on the figure eight simulation seen in Figure 127, and Figure 128

When a RltPoint is first seen it is desirable to immediately place it in the map. The RltPoint is placed in the RltUngroupedList which is referenced after a few iterations. At this point the RltPoint is placed into a group. Due to only being able to look back a few iterations it is not possible to determine the best previously grouped RltPoints to place into that group. The RltPoint is also placed in a second RltUngroupedList with a much longer look back. When the RltPoint is enumerated from the second list after sufficient iterations have passed, a much better RltGroup is created. The previously grouped points selected are more likely to be present in more iterations than the ones in the initial group. The

RltGroups that are created are denoted as either being temporary groups created with a small look back or final groups with a longer look back. Final groups only use previous grouped points that are also in final groups. This means that as long as there are final groups, the overall accuracy of the map is dependent only on RltPoint in the final groups.

Periodically the temporary RltGroups are checked to see if all of the RltPoints referenced in it have been added to a final group. If this is the case, then the temporary RltGroup is redundant as it does not have any information that is used for mapping purposes. In the simulation most temporary RltGroups are removed, however there are a few that contain points that are judged as not being suitable for a final group so they are left out. RltPoints that are only in a few iterations are not placed in a final group so they do not affect the accuracy of the map.

There are several constants that may affect the performance of the group creation:

- The main constant is the look back / look ahead. This is the constant that determines how many iterations to wait in order to add the RltPoint to a group. This allows the group creation to look ahead or back this constant to look for points to place in the group.
- Accuracy Reduction. This constant determines the threshold for adding RltPoints to the RltGroup. The threshold to add RltPoints to the new group is the look back constants minus the accuracy reduction.
- PrevGroupMin, NextGroupMin. These minimums determine the minimum amount of previous grouped points that have to be available to form a group and the minimum amount of ungrouped points that have to be available to form a group.
- PrevGroupMax. The maximum amount of previous points that can be added to a new group.

Since there are many constants to test, each one is tested individually to get an idea how it affects the accuracy. All of the testing is this section is performed on the figure eight simulation where a full loop is about 3800 iterations. Figure 120 shows the testing with the highlighted pink line belonging to the baseline. Figure 121 relates the corresponding quantity of groups to the error. The base line has 60 look back / look ahead, 20 minimum previous group, 5 minimum next group, and 10 accuracy reduction.

The mapping error is the sum of the absolute difference in distance between the actual point and the globally mapped location given by RltPoints. The error is cumulative so that a single error early will carry on until the loop is closed. A single error appears over time as a line as the error propagates forward. When the mapping error has more of an increasing curve this indicates that there is the continued addition of mapping errors in that time iteration. It is also possible to have mapping errors in offsetting direction that will reduce the propagation of errors.

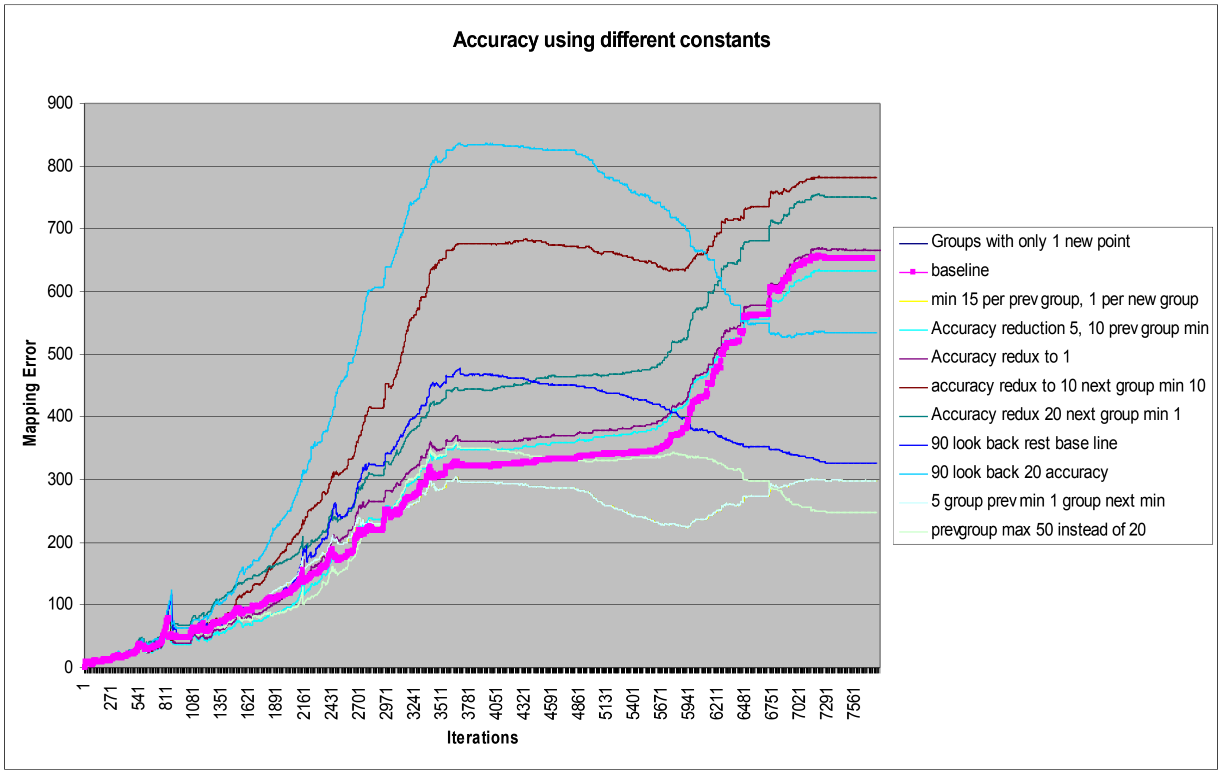


Figure 120: Accuracy using different constants

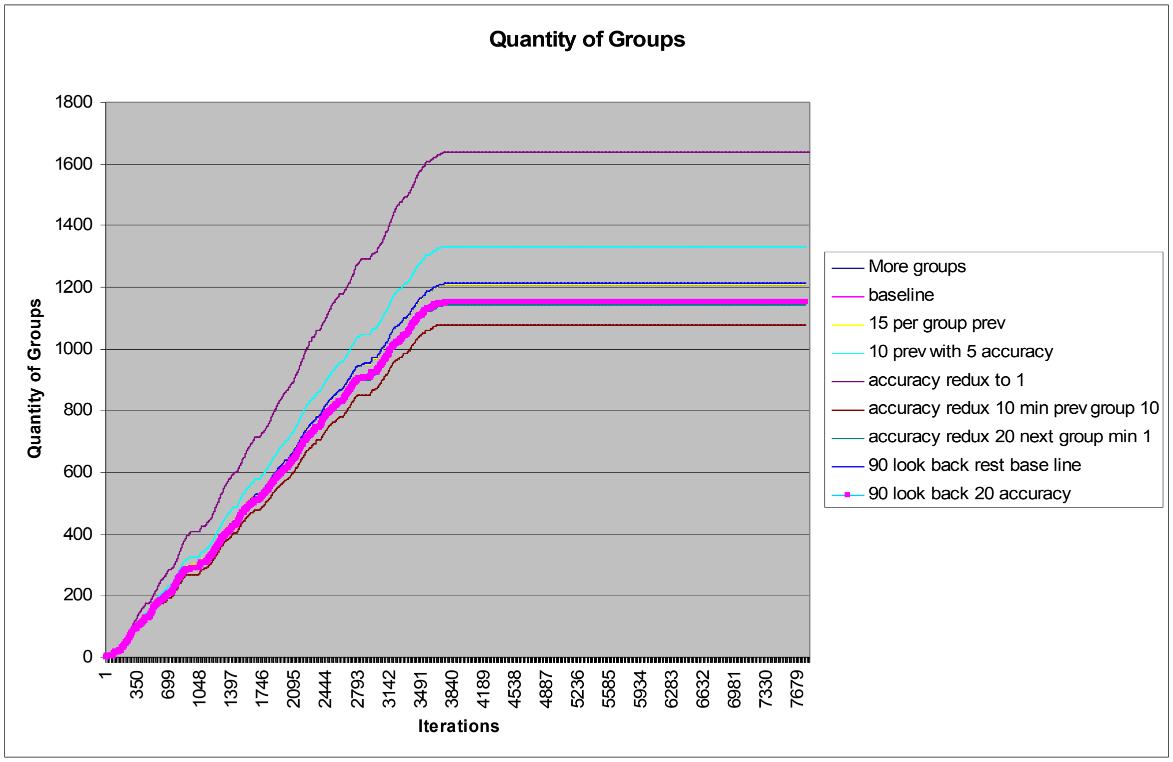


Figure 121: Quantity of Groups

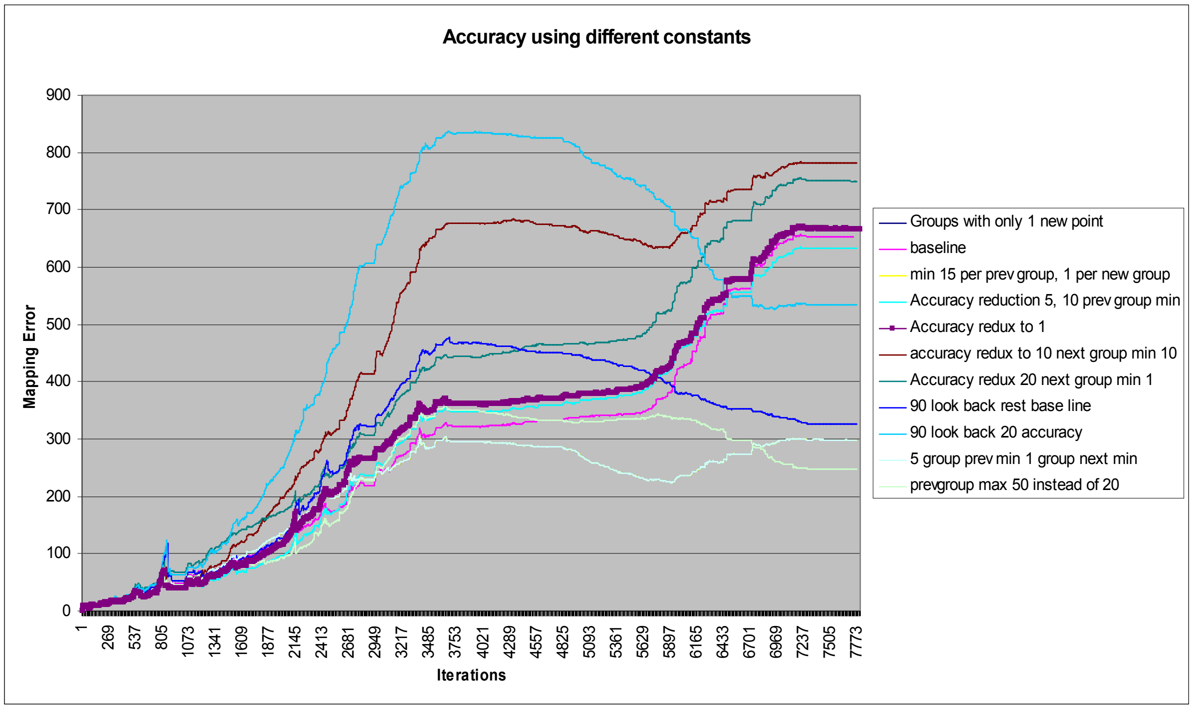


Figure 122: Evaluating the accuracy of setting accuracy reduction to 1

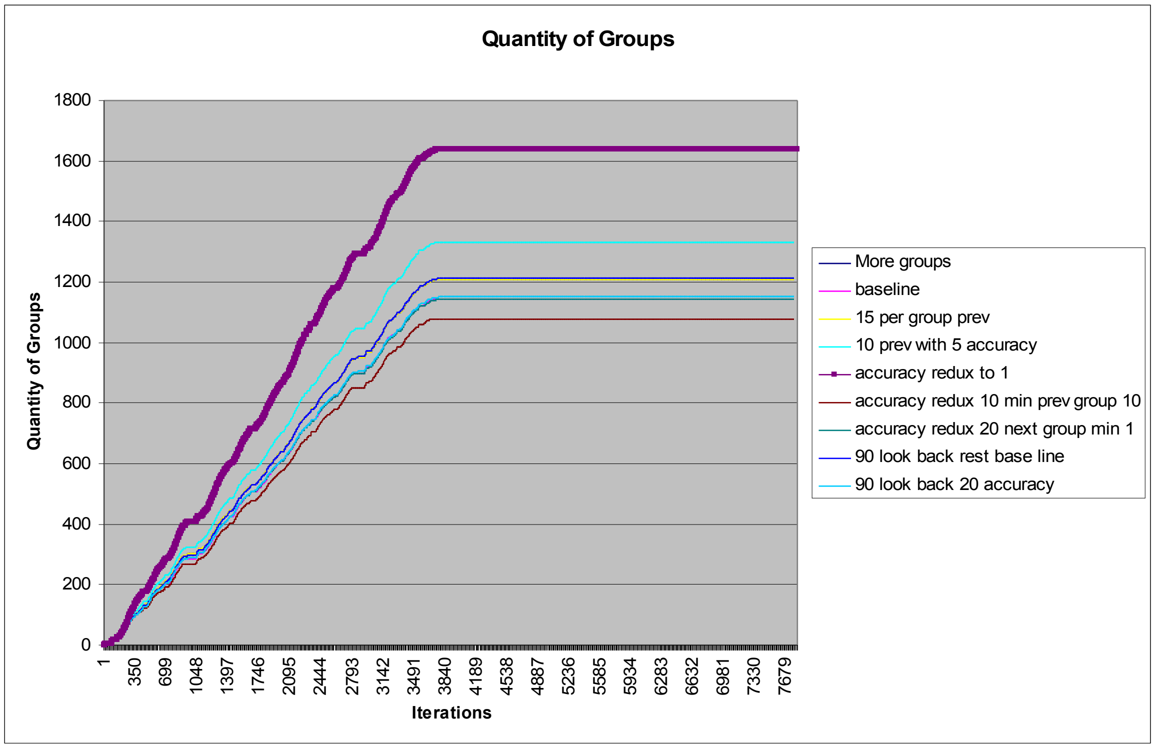


Figure 123: Group size with accuracy reduction set to 1

The first question is what happens when the accuracy reduction is reduced from 10 to 1. This will force smaller groups, as less ungrouped RltPoints will qualify to be in a RltGroup since they need to be in nearly the complete group interval. Figure 122 and Figure 123 highlight with a thicker line the affect of altering the accuracy reduction to 1. Notice that is does not have an affect on mapping error but it does increase the number of groups which is expected. This is a bit surprising as the initial guess was that less groups would correspond to higher accuracy.

Figure 124 illustrates the affect of increasing the look back / look ahead and altering the accuracy reduction. The top line is when the look back has been increased to 90 from 60 and the accuracy reduction increased to 20. The bottom highlighted line is with the look back / look ahead at 90 with the accuracy reduction being at 10. It was a surprise that having a higher look back / look ahead causes higher error than the baseline until a second loop of the figure eight is performed. Using 20 as the accuracy reduction makes the error significantly worse despite having more iterations guaranteed to be in the RltGroup.

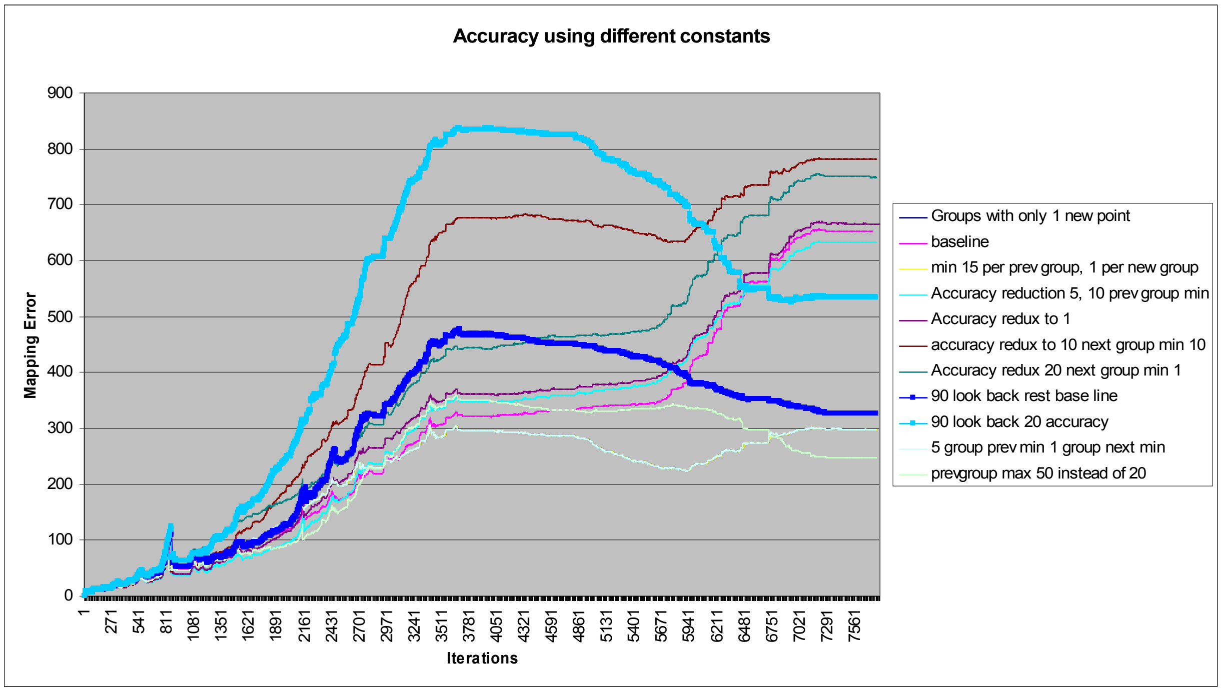
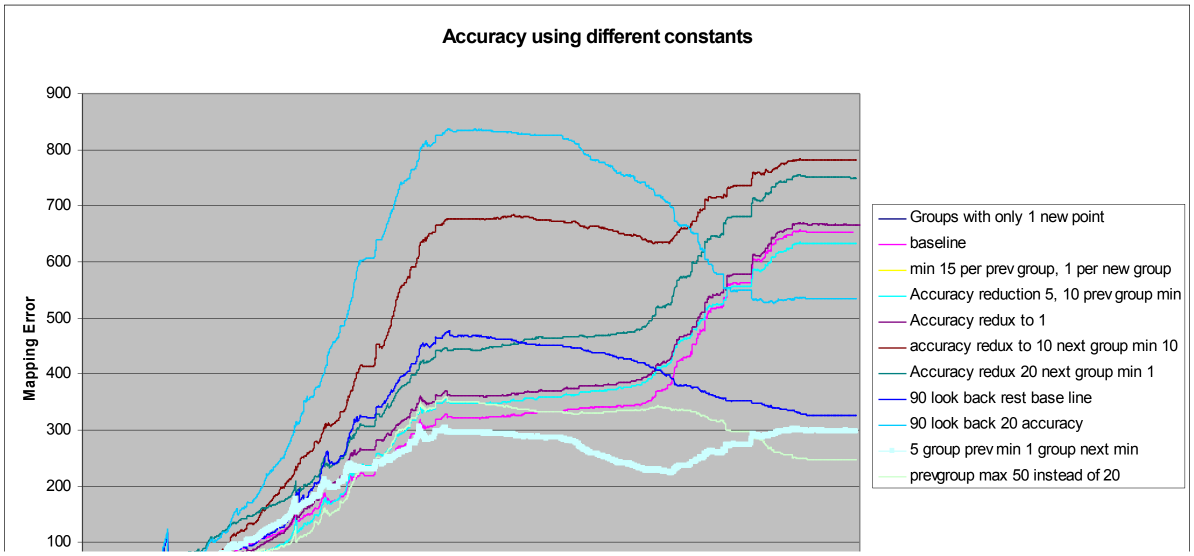


Figure 124: Highlighting higher look back / look ahead

After performing many tests, perhaps the best improvement comes from reducing the minimum required amount of ungrouped RltPoints to be added to a group to 1. This is shown in Figure 125. This makes intuitive sense. The grouping algorithm is always better off without a minimum size of new ungrouped RltPoints since there is always a minimum previous grouped size which should guarantee the accuracy of the RltGroup. Every fully mapped RltPoint can be said to increase the accuracy, rather than leaving an RltPoint to only being mapped in a temporary group.



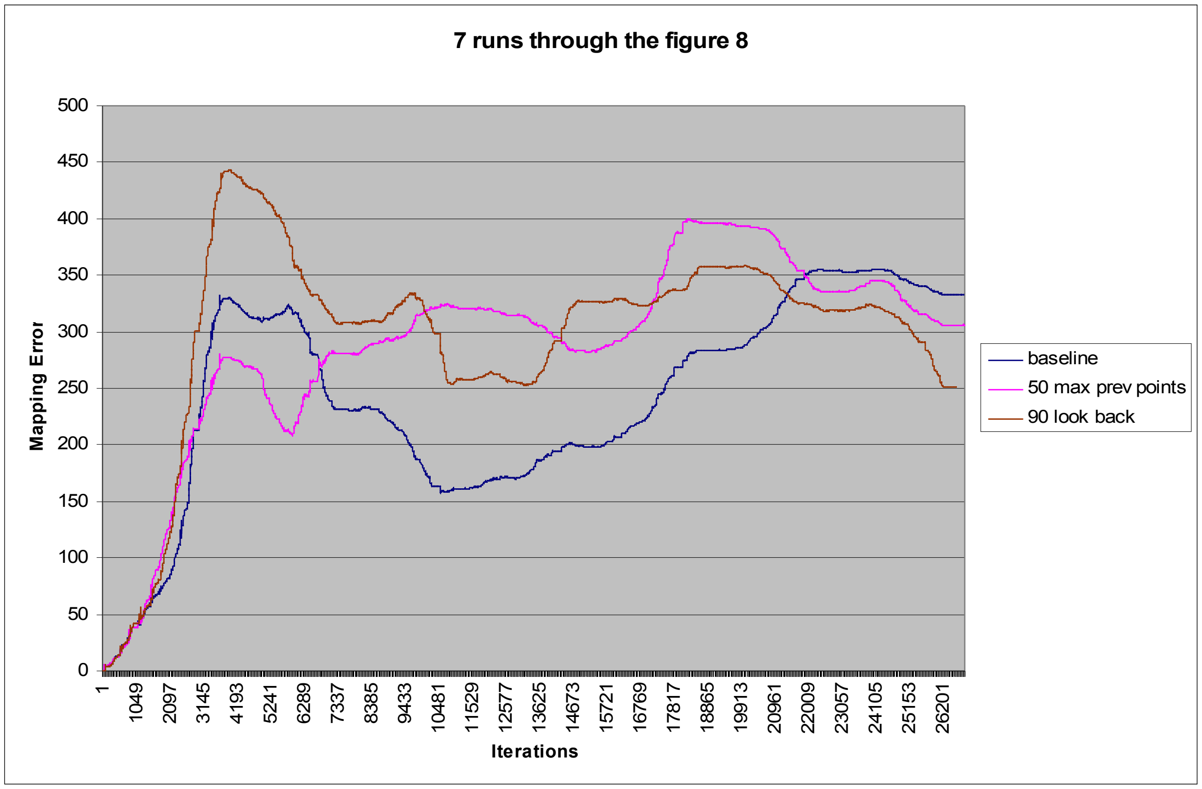


Figure 126: 7 runs through the figure 8 with 3 different settings

To get greater clarity into the accuracy versus the constant values Figure 126 was created running through the figure eight 7 times. All three lines use a minimum ungrouped RltPoint quantity of 1. One line is the baseline of 60 iterations look back and a accuracy reduction of 10. One line is the baseline with a 90 iteration look back / look ahead instead. One line is the baseline with a maximum of 50 previously grouped points added per new RltGroup rather than 20. These three lines are the lowest in Figure 125.

After examining Figure 126 it was decided that it is not worthwhile to further evaluate constant values versus mapping error. Perhaps the three chosen for Figure 126 are better than the ones not used from Figure 125 but it is very difficult to judge which one is better.

Perhaps the overall issue is not with altering the constant values but rather, reducing the use of constant values. The binning algorithm in 6.5.3 Dynamic point detection uses a routine to generate the standard deviation threshold. The binning algorithm should be adaptable to different environments without having to alter a constant value manually. Perhaps it would be better to have some routine that is able to automatically adjust the constant values as the robot explorers a new area.

One potential algorithm to create a RltGroup without constants is as follows:

1. Wait until a creation RltPoint is no longer visible before adding it to a final group.
2. Given the creation RltPoint start and end interval, find all points that are present and put them into sorted lists. One list has points that are already in a final group and the other of points not in a final group. These lists are sorted by the amount of iterations present in the observation interval of the creation RltPoint.
3. Add the three best already final grouped RltPoints and the creation RltPoint to the new group.
4. Keep on adding RltPoints obtained from the descending order of both sorted lists as long as the total amount of iterations goes up.
    - To check if a RltPoint should be added: perform an "and" operation of its observation interval with the current observation interval of the new group.
    - Multiply the total iterations in the new observation interval by the number of points in the group plus one.

- If the new total iteration is larger than the previous, add the RltPoint to the group and copy over the new observation interval.

6.6 Examining the odometry and run time performance

6.6.1 Odometry analysis in the figure eight simulation

The figure eight simulation used in the previous section 6.5.5 Group optimizations was tested again. The results of the simulation are shown in Figure 127 and Figure 128 from two view points. The robot travels in a figure eight path starting at the middle and going to the bottom right. The robot travels up a gradual incline which reaches its peak at the starting point and then starts to decline. When the robot returns to the starting point after completing the right side loop it is above the original starting point and does not close the loop until it completes the loop on the left side.

There are a total of 1795 RltPoint in the simulation and about 1250 groups. Only 249 of those groups are final groups and the ones used to propagate the map. There are many groups since as discussed in 6.5.5 Group optimizations, RltPoints that are spaced more than 10 iterations apart get placed into their own groups. A single loop of the figure eight is about 3800 iterations.

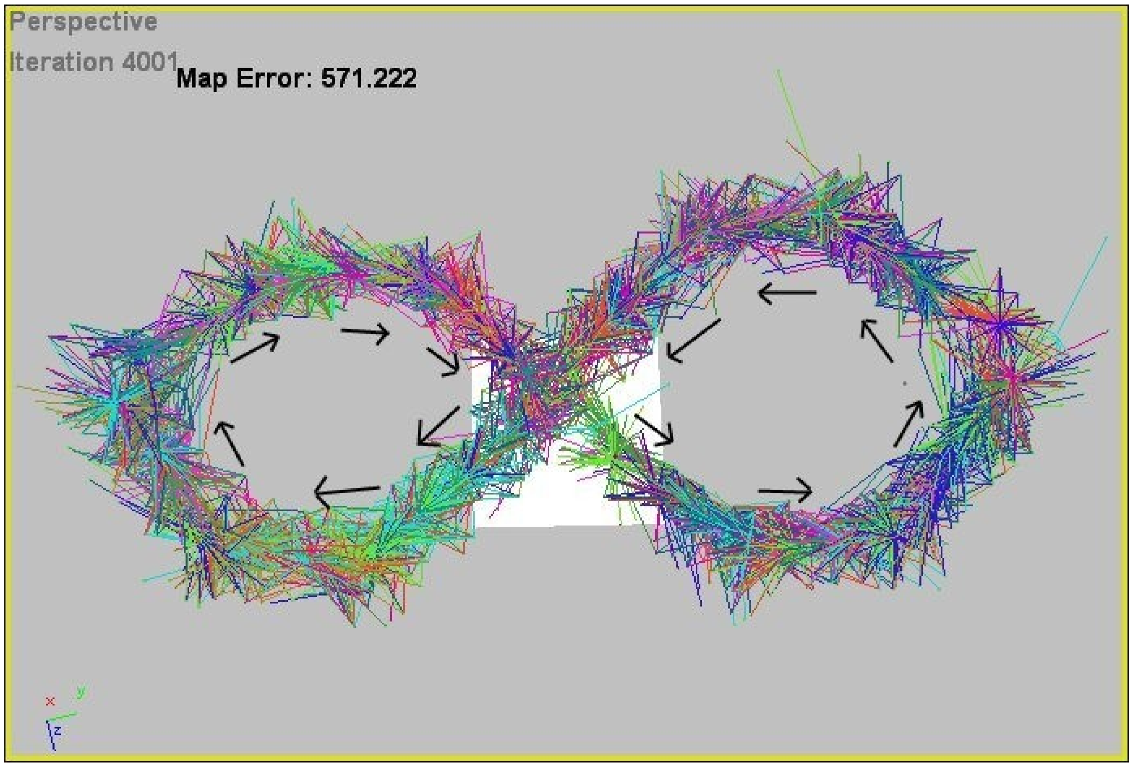


Figure 127: Figure eight top view. Every group has its own color of lines to each point from the centroid of the group.

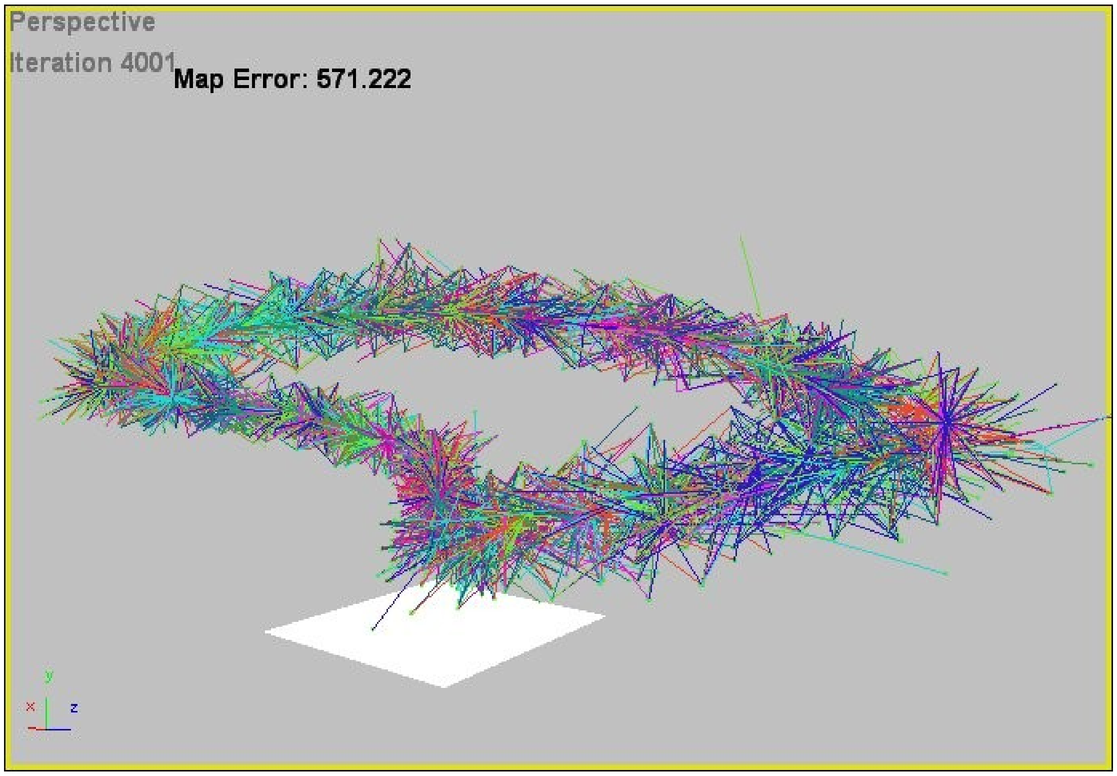


Figure 128: Figure eight side view

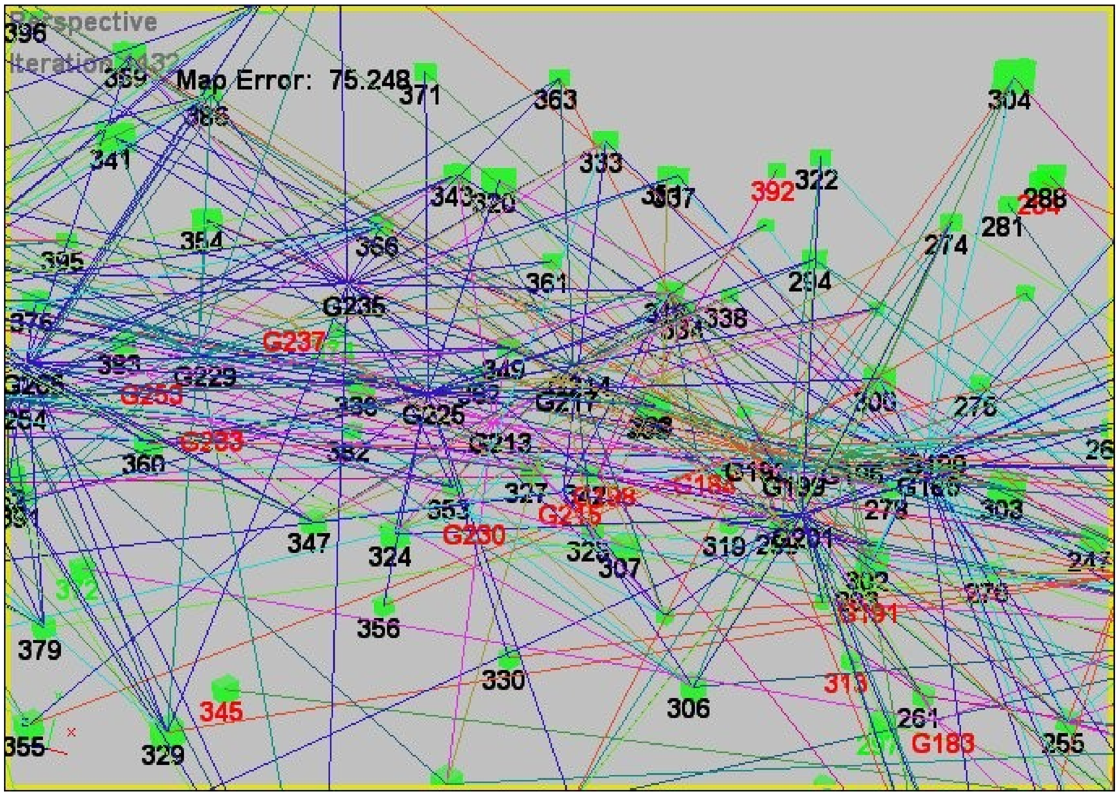


Figure 129: Figure eight close up

Figure 129 shows a close up of the figure eight. Green boxes represent the global location of RltPoints. If the number beside the box is in black then that RltPoint is fully mapped. If it is red it is only temporarily mapped. Each RltPoint has a line to each RltGroup it is linked to. The lines are from the global location of the RltPoint to the centroid of the RltGroup. The centroid has no functionality other than rendering. The color assignment of each line is based on the group it is a member of. The groups are labeled Gnnn where nnn is its index number with the text being black for a final RltGroup and in red for a temporary group.

Mapping errors can come from three places: noise with less than infinite iterations, frustum bias and algorithm inefficiency. Depending on the noise and the quantity of iterations there will likely always be a small error. This can be seen in Figure 152: Stationary viewpoint landmark error, which tests the EKF versus the relative algorithm with landmark noise given a stationary position. These small errors can accumulate on the map to create a bias and become noticeable.

There are errors due to the frustum. Some of the earlier testing had the robot have a steadily increasing y error in one direction. It was not obvious what was causing it, especially since the robot's path was set without changes in the y direction. The algorithms is 6D so there is not anything that would appear to only cause a bias in one direction.

It turned out the error was due to points being generated from y being zero to a height constant and the frustum was from a negative height to positive height. When the points at the top of the frustum had positive y noise they left the frustum causing the RltGroup to not be recalculated in that iteration. If a point at the bottom of the range had noise in the negative direction it would still be in the frustum. This led to a steady downwards bias as more noise in the negative y direction would get computed than from the positive y direction. When the point generation was balanced using a negative to positive height constant the problem went away. Frustum bias can still occur during turns when the frustum travels in one rotation direction.

The third place for error could be error caused by the algorithm itself. Since RltGroups discretize which iterations are used to calculate a RltPoint global position this can perhaps cause errors. There is the possibility that any discretization error can be made worse if the RltGroups are created inefficiently. This can happen if groups have misplaced RltPoints that cause too small of a calculation interval. There is a chance that using three points as a basis can cause noise, as the noise in these points may translate into noise in other points.

To try to get an idea where the noise is coming from Figure 130, Figure 131, Figure 132, Figure 133, Figure 134, and Figure 135 display the error in each of the x,y,z direction corresponding to robot's location. Each full loop of the figure eight occurs at about 3800 iterations. In the simulation used, the robot is always within the bounds for the loop to be closed automatically so the error goes to zero. The total length of a loop is about 1145 units.

It is interesting to see how the error is periodic especially during changing in direction of travel. This can relate to the fact that there is bias in the noise during turns due the frustum removing noise in one direction of rotation. Also during turns there is often less RltPoint available to form groups so that may be a source. It can not be said for certain how much of the error is due to the frustum issue versus the other two sources but it can be said that it is likely that there is a limit of accuracy due to these issues.

It is not going to be possible to compare the output of the Relative algorithm against a map that is optimally created having the minimum amount of landmark error given noise. Instead in order to evaluate the accuracy of the Relative Point algorithm further it is compared against a 6D Extended Kalman Filter (EKF) in 6.7 Comparison with the EKF in simulation. The Kalman filter does not discretize the map according to groups and has a different basis, the current location. If the Relative algorithm has much larger landmark noise than the EKF then it can be said that the Relative algorithm is the cause of the difference of landmark error from the EKF's result. However if the landmark error is similar for both algorithms that would seem to imply that the error is inherent to the map due to the noise.

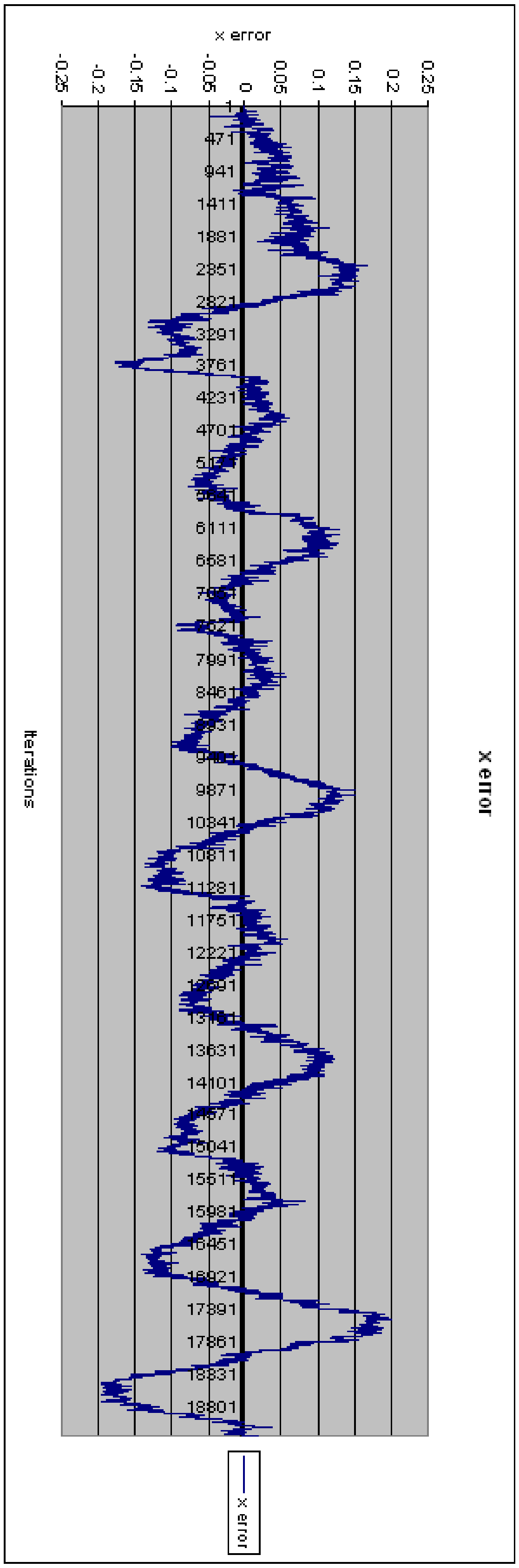


Figure 130: x error

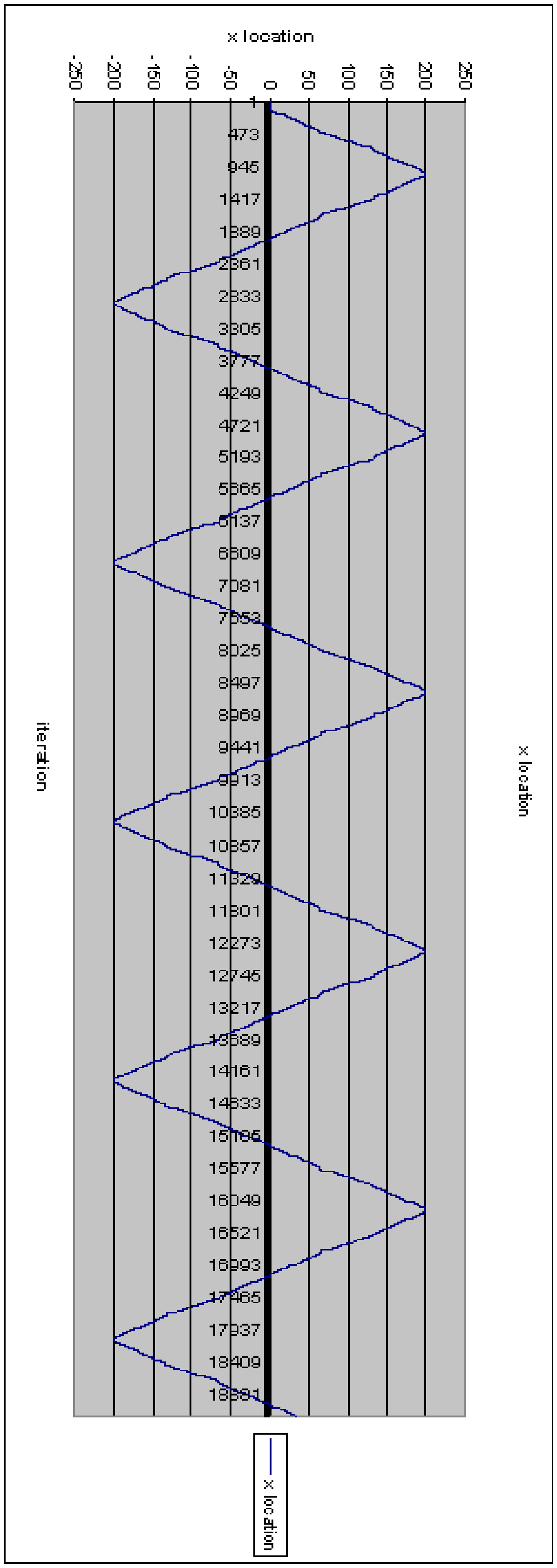


Figure 131: x location

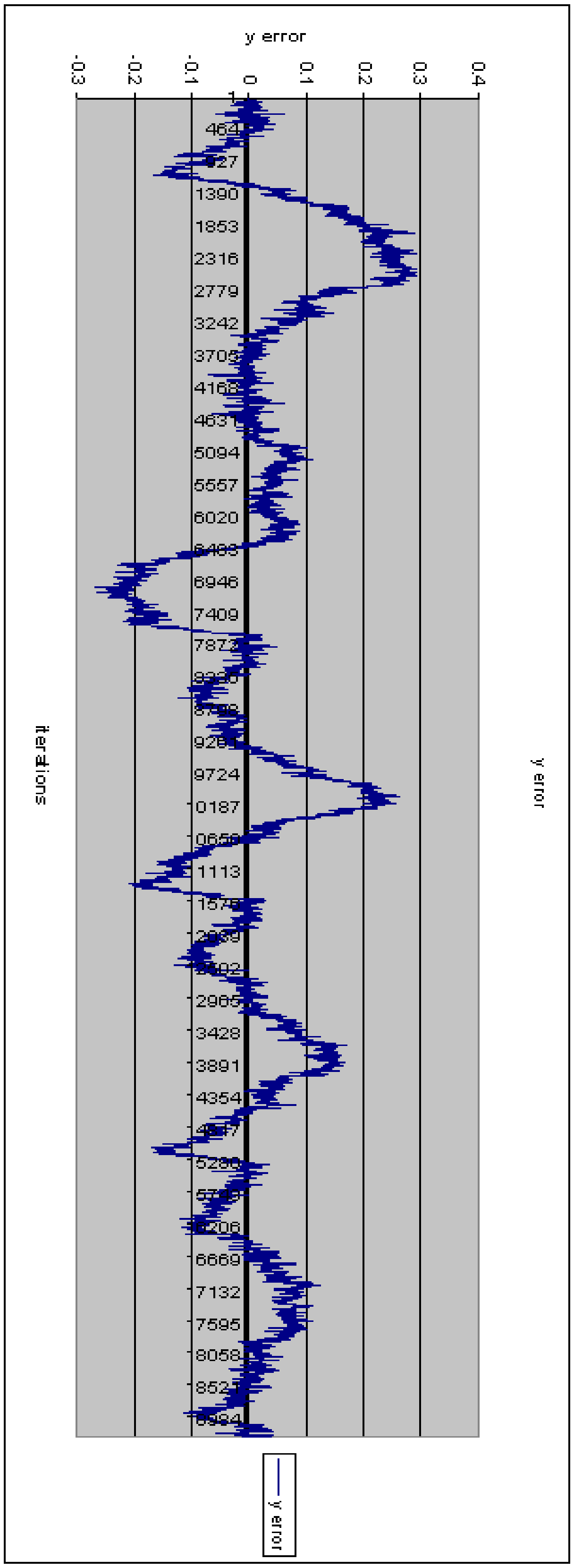


Figure 132: y error

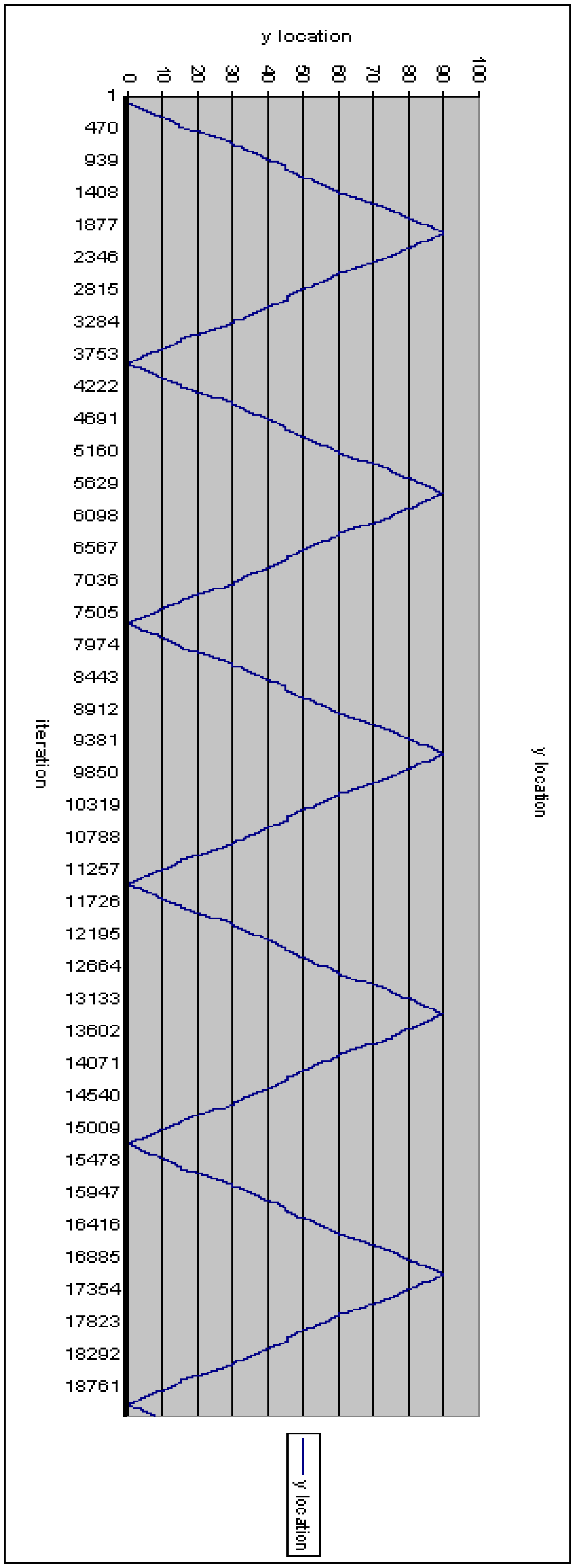


Figure 133: y location

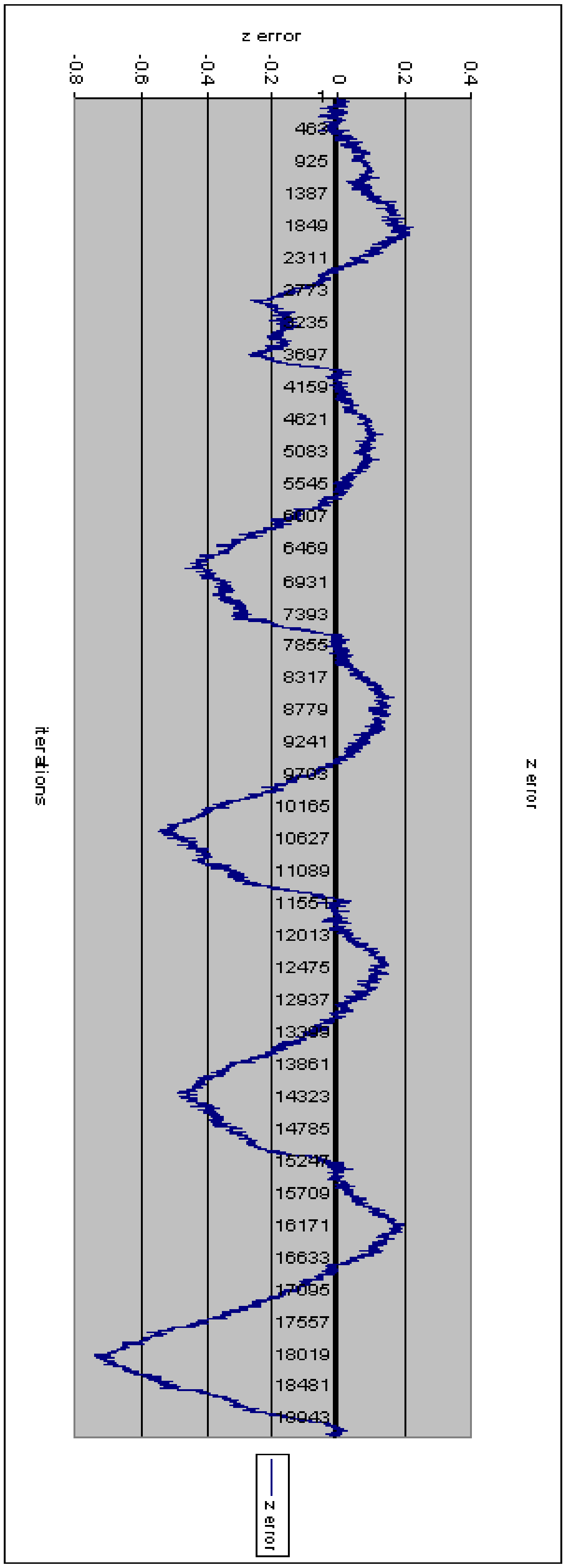


Figure 134: z error

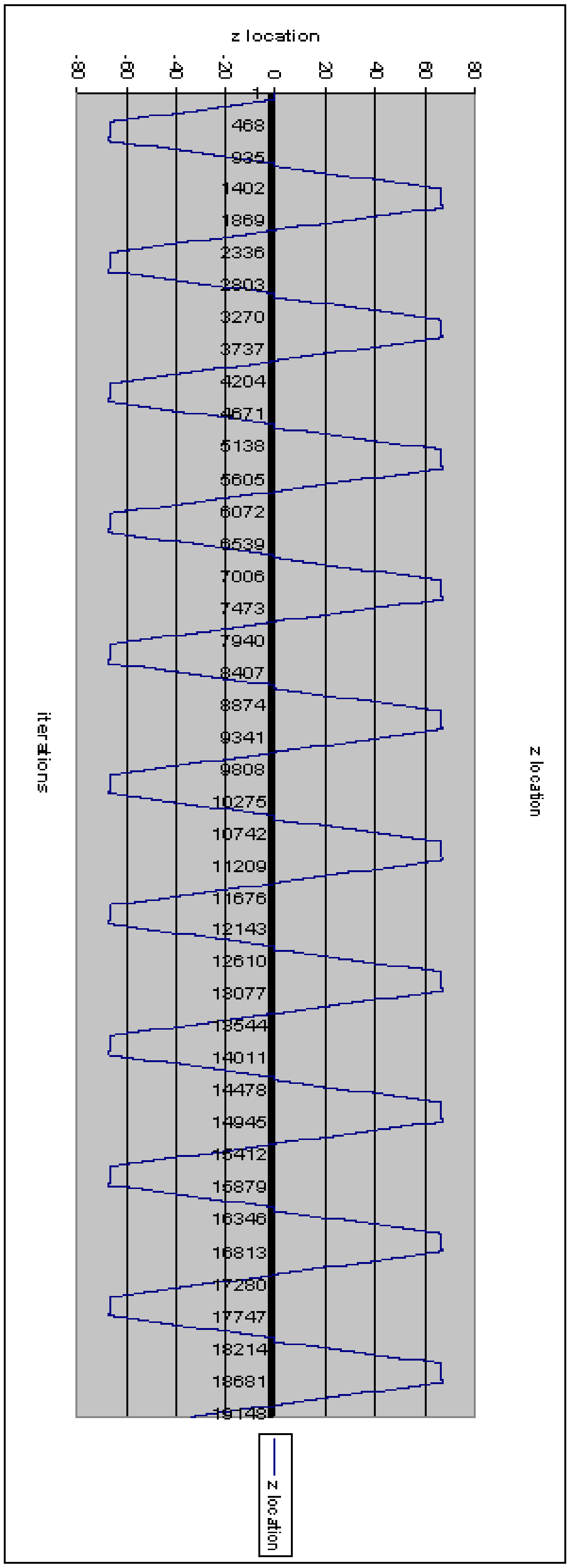


Figure 135: z location

6.6.2 Performance testing

This section evaluates the execution speed of the algorithm. The first test is to see if the algorithm execution time increases as it continues to explore new areas. Figure 136 shows the execution time as the robot does a figure eight of about twice the size of the previous testing. Figure 136 excludes the first 200 iterations since the run time is near zero for the first iterations and it distorts the chart. The execution time uses a 12 iteration moving average to make it easier to read.

The execution time linear regression is flat. There are currently no parts of the algorithm that use global processing and work on anything except RltPoints and RltGroups that are recently seen. It would be very evident if any parts of the algorithm were using global processing as seen in Figure 110 in 6.5.1 Global point matching.

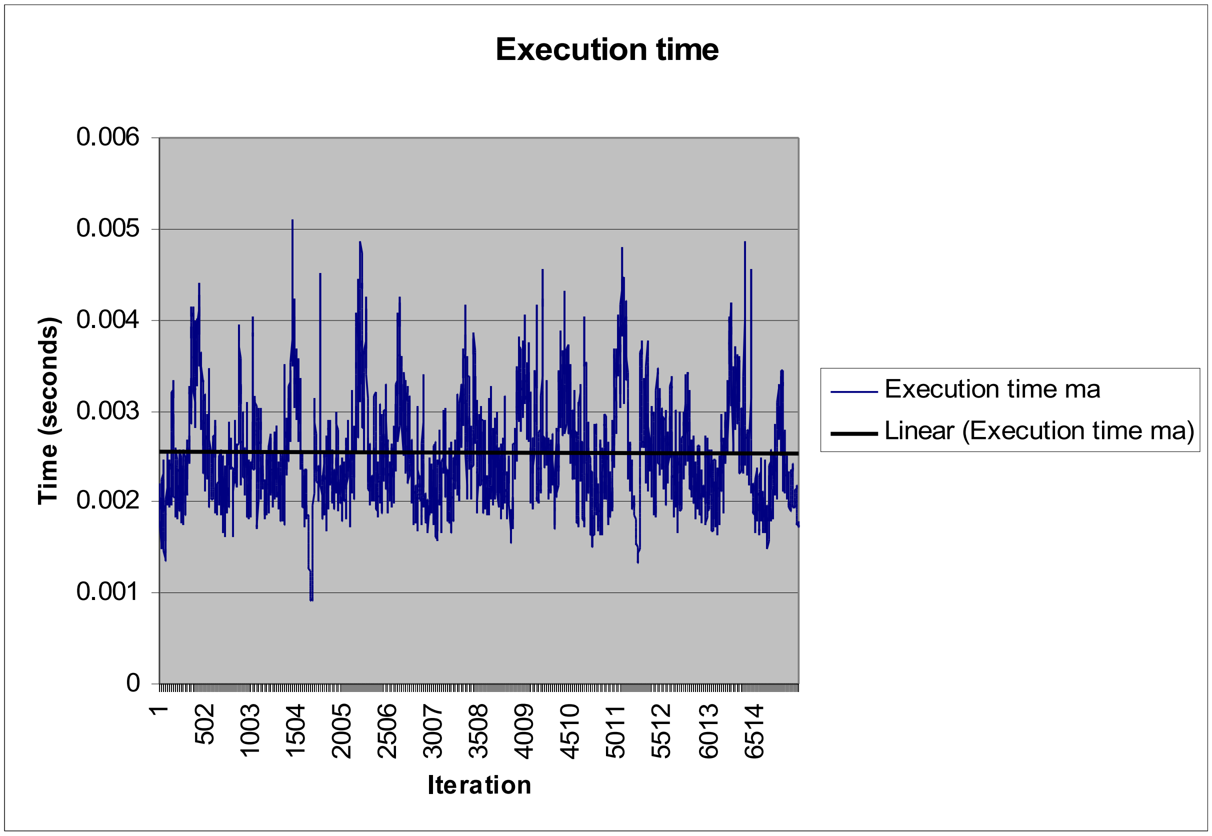


Figure 136: Execution time

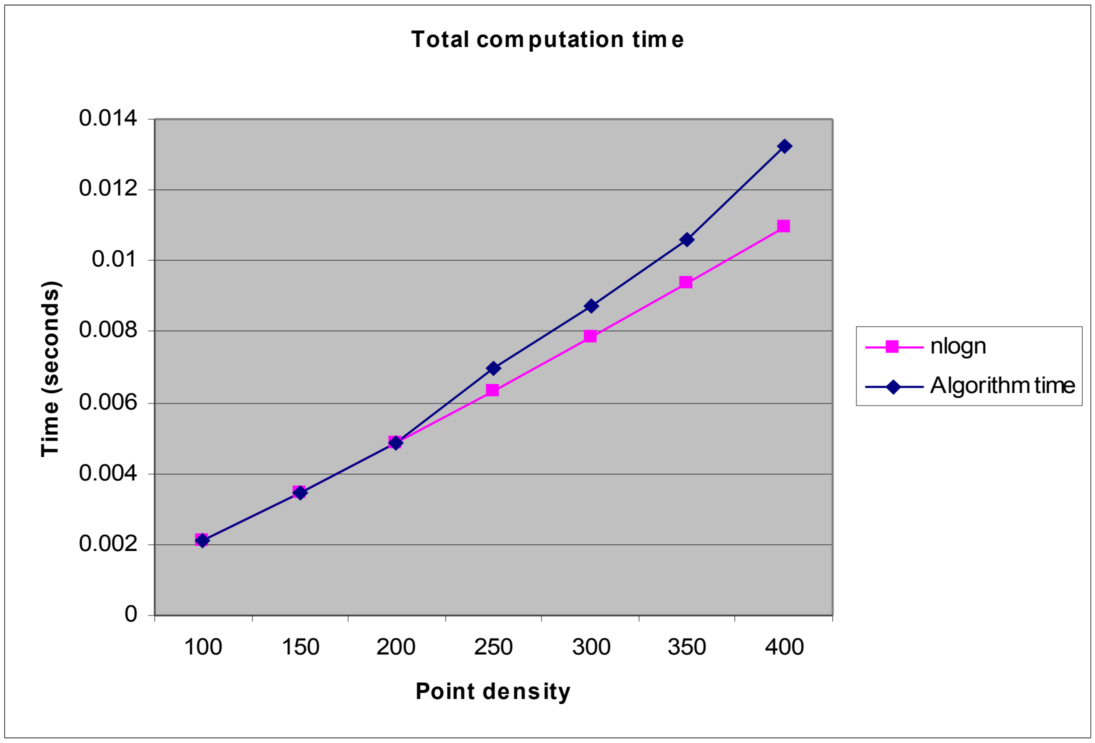


Figure 137: Testing the linearity of point density

The next test evaluates the effect of increasing the amount of points in the simulation. The point density is increased in the figure eight simulation from 100 points per 100 units of path length to 400 points per 100 units of path length. The step size is 50 point per 100 units. The results are shown in Figure 137. At first the run time is identical to O(nlogn) but then it has an polynomial component.

In order to further evaluate the algorithm, the execution time for each component of the algorithm is charted. This consists of.

- The initialization. Points are enumerated from what is observed in the last few iterations from 6.2.3 RltPointCharting and added to the 6.2.4 OVLPQuadTree.
- Load Time: Points are added by first comparing them to the local quadtree 6.2.4 OVLPQuadTree and then the global. New RltPoints are created in this step.
- The RltGroups are computed 6.3.3 RltGroup and combined with 6.3.2 RltICP to generate a map.
- New RltGroups are created 6.4 Group Creation, and 6.5.3 Dynamic point detection is performed

Each of these four are charted in Figure 138, Figure 139, Figure 140, and Figure 141.

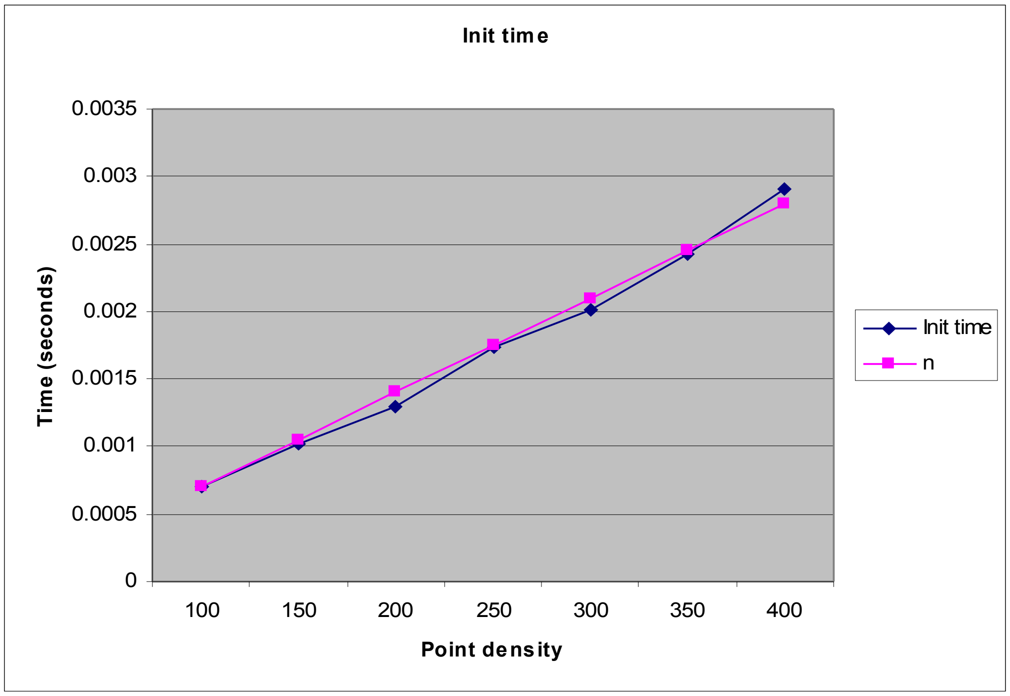


Figure 138: Initialization time

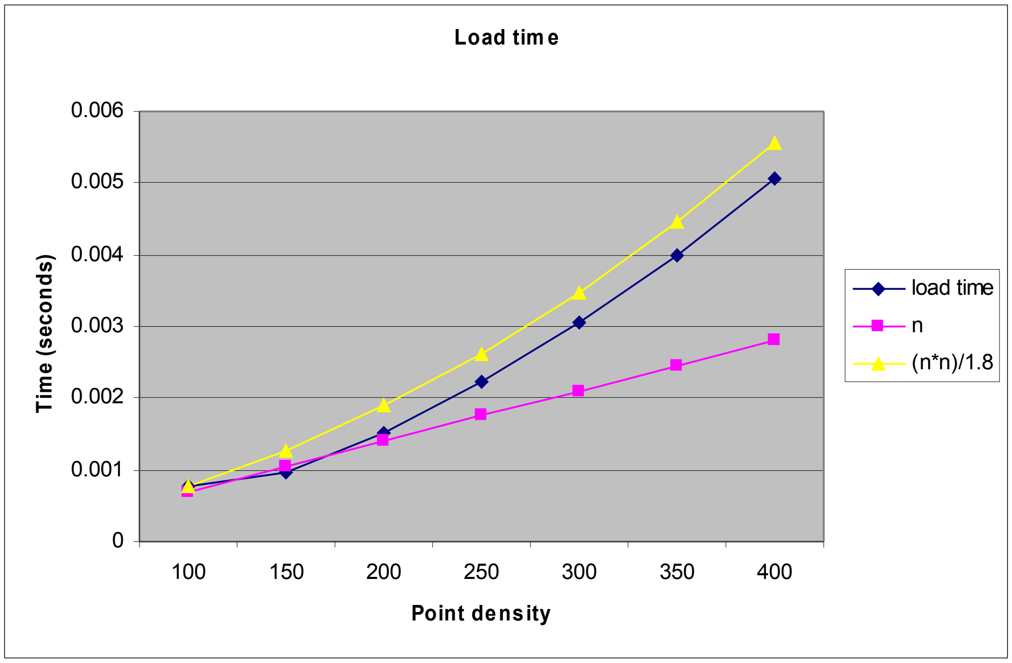


Figure 139: Point comparison (load time)

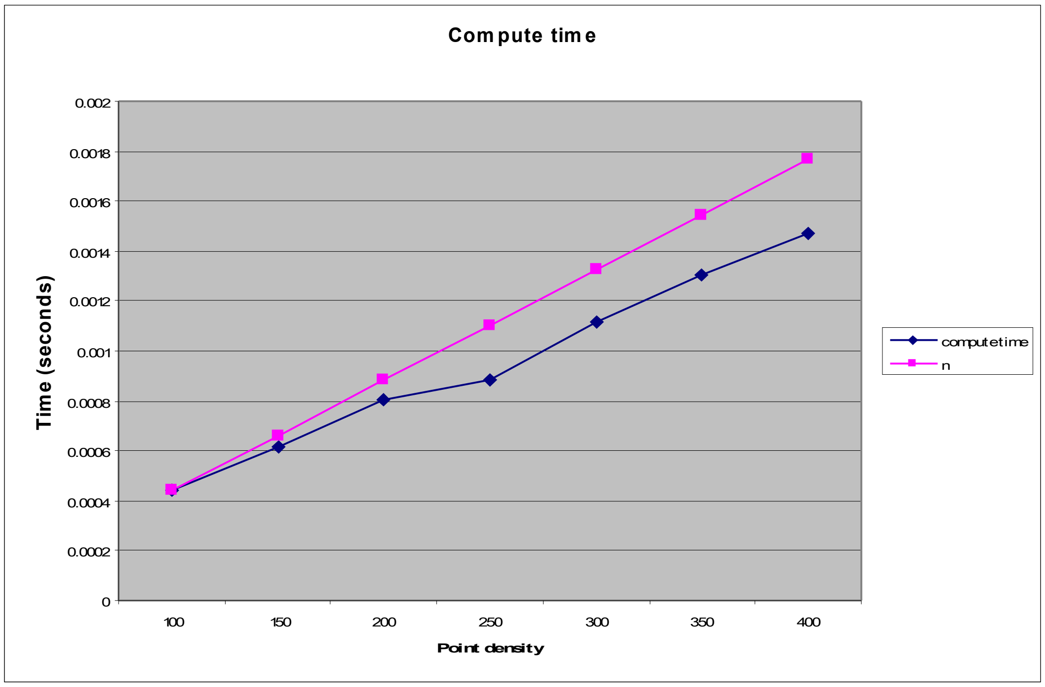


Figure 140: RltGroup computation time

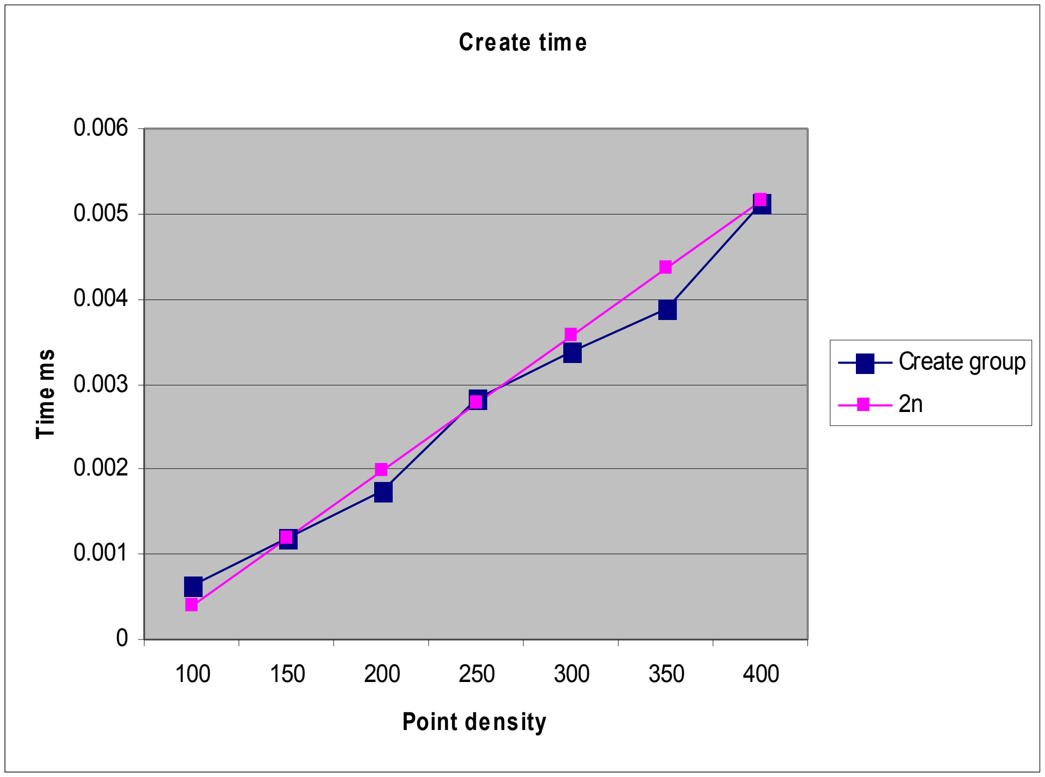


Figure 141: RltGroup creation time

- Figure 138 shows the initialization time is O(n)
- Figure 139 shows the point comparison is O(n^2).
- Figure 140 shows the RltGroup calculation time to be better than O(n). As the point density increases, the group per point ratio increase which lesson the amount of time performing ICP, thus lowering execution time.
- Figure 141 Shows the group creation time at about O(2n). It is compared to both a linear and polynomial regression and appears to be linear. The creation included both the RltPoint search for RltPoints that belong in the group and the binning algorithm.
- Figure 142 Shows the percentage breakdown of each of the parts

Figure 139 shows the load time is the only part of the algorithm that is not O(n). To verify that it is responsible for the polynomial part in Figure 137 the difference between the O(n) line and the actual line is plotted for both the total execution time and the point addition execution time. Figure 143 shows the point addition is responsible for all or nearly all of the polynomial behaviour.

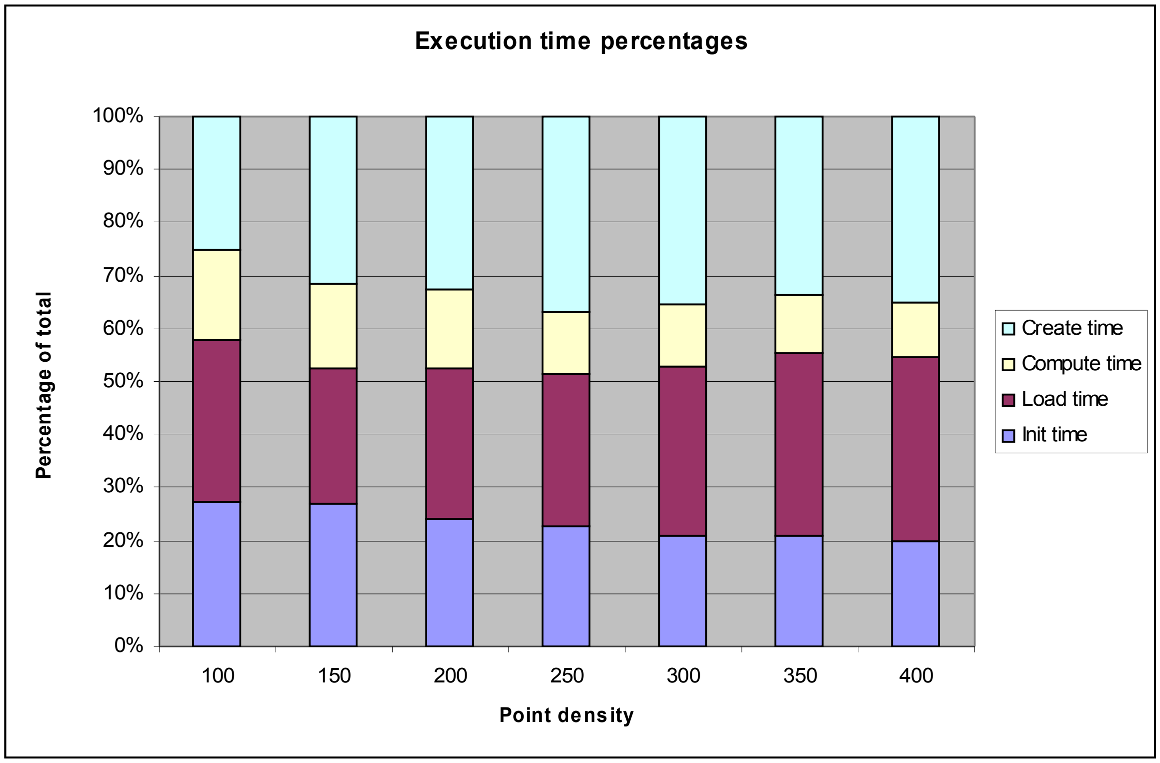


Figure 142: Execution time percentages

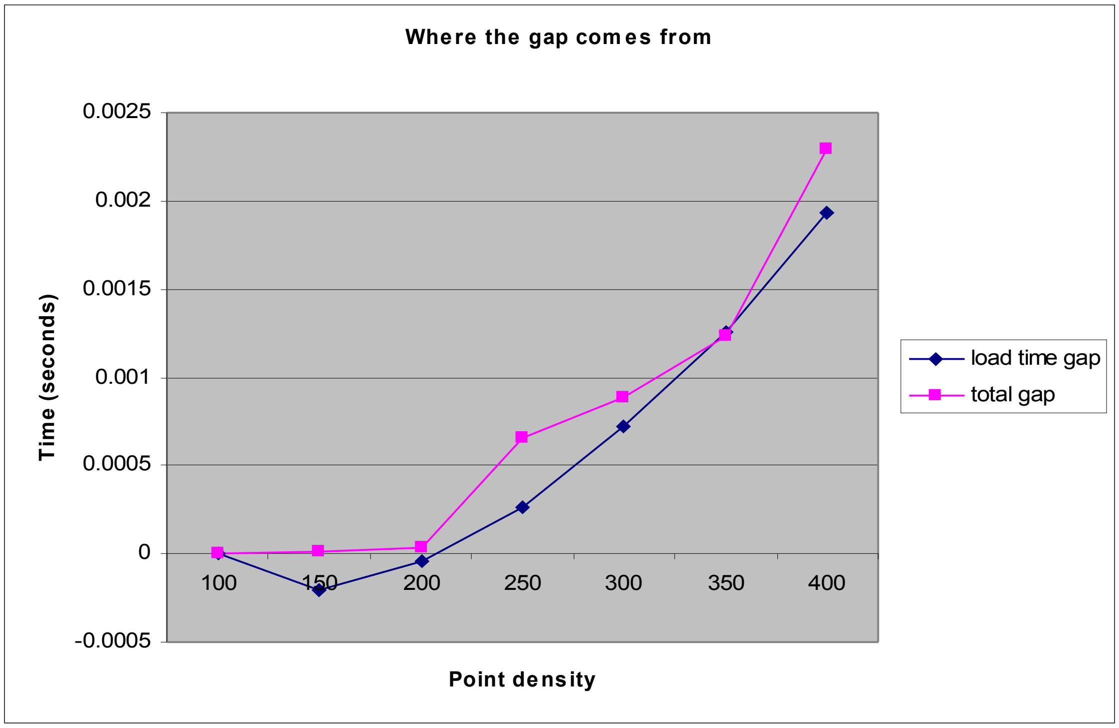


Figure 143: Almost the entire non O(n) part of the algorithm comes from the adding points part

Figure 144 shows the same test being performed, this time with the quadtree having smaller bins. It is compared to the data from Figure 137, nlogn lines for each data set and polynomial regression for each data set. The quad size is reduced on the local quadtree to near the minimum size given the noise. The computation time is still slightly polynomial as shown by the polynomial regression but it is an improvement.

There are other structures to consider octree [Wiki11a] and kd-tree [Wiki11b]. Both were attempted and using the octree it was found that the load time sees an improvement but the initialization time is slower. The octree has to search 3 times as many bins on the initialization as the quadtree does. A kd-tree was also tested. The kd-tree was found to be slower at both initialization and adding points in both the 100 and 400 point density. The kd-tree creation is unoptimized using a full sort to find the median, however that should insure that the tree is well balanced for the search. Even if the kd-tree creation can be improved, the search cannot be optimized further so the other two data structures are better choices.

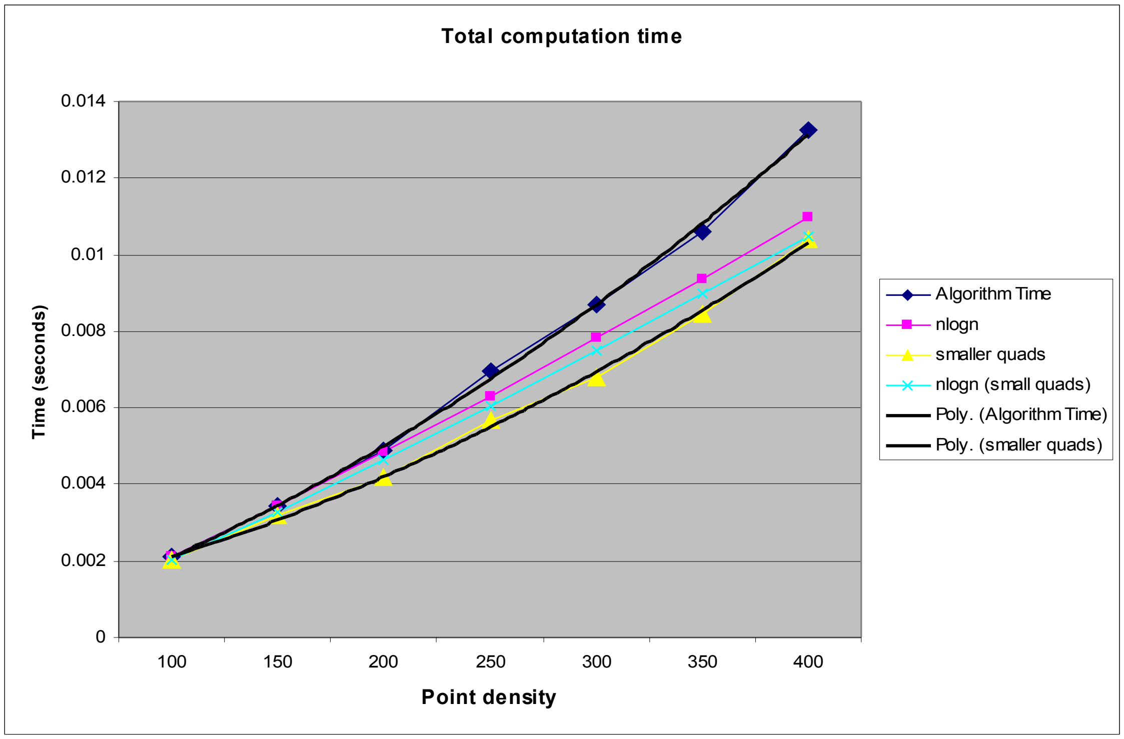


Figure 144: Comparison of using small quads

In Figure 145 an octree is used from the second 400 (there are two entries at 400 the first is the last quadtree test the second is the first octree test) to 550 which is near the maximum given the noise and point spacing. At the 400 density the octree takes on average 7.8ms to initialize which is about 3 times larger than the quadtree's 2.5ms. The load time using the quadtree is 4.5ms and 1.3ms using the octree. The octree charts between the trend lines of n and nlogn showing that the octree is has a reduced logn O(nlogn).

The point of combining them is to demonstrate that even near the maximum point density given the noise and matching bound, the execution time can still be considered approximately O($n_s$logn_s). This depends on the proper selection of the data structure used for point matching. There is a tradeoff of computation complexity versus actual execution speed. The Relative algorithm can be considered worse case O($n_s$logn_s) where n_s is the amount of observable points that is a subdivision of the total points on the map.

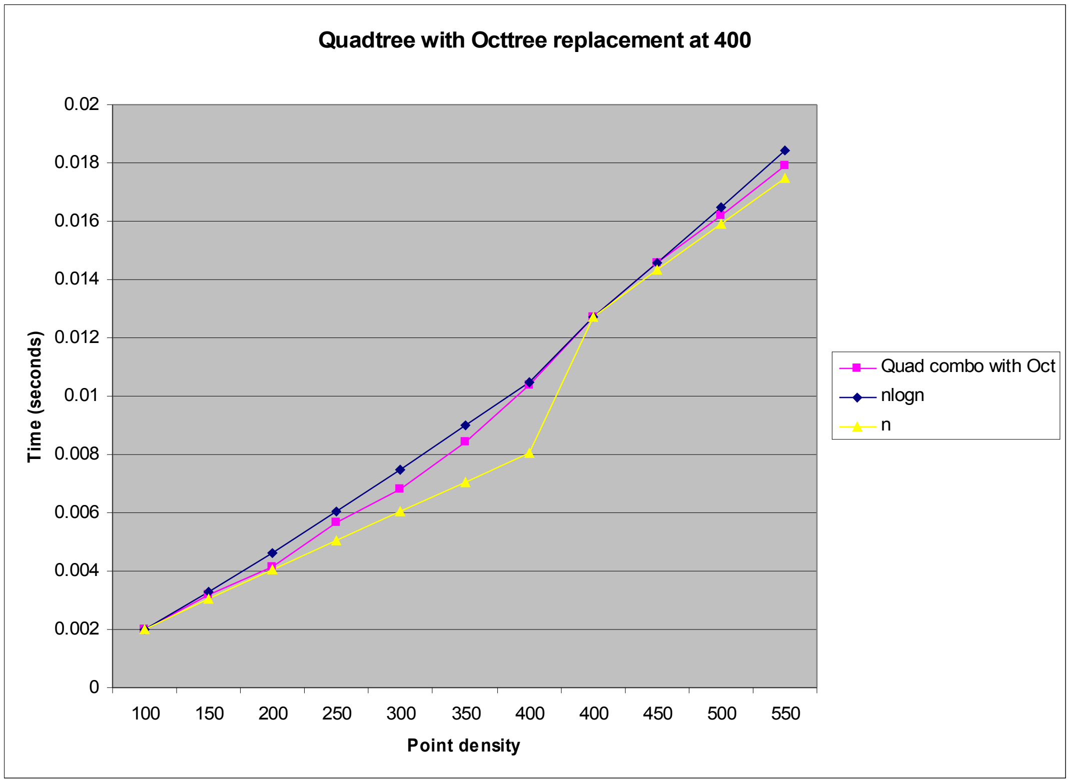


Figure 145: Comparison of using small quads and octree replacement at 400

6.6.3 Summary

In terms of accuracy there are three sources of noise: small errors accumulating due to the finite number of iterations, frustum bias during turns and inefficiency in the algorithm. 6.5.5 Group optimizations looks at landmark error in regards to group creation optimization but did not come to any conclusions given Figure 126 illustrates that it is not clear how to optimize the selection of the group creation constants. 6.6.1 Odometry analysis in the figure eight simulation closely looks at the odometry and finds that the errors in odometery are correlated with the robot changing direction. It can be said that given the raw data of the simulation there is a limit to how accurate the map can be made. It may be difficult to quantify how well this algorithm performs against the theoretical limit. However, it is possible to compare the Relative Point algorithm to an implementation of the EKF to get an idea if the source of error is the Relative Point algorithm or from the data. This comparison is in the next section 6.7 Comparison with the EKF in simulation.

It terms of computation complexity two scenarios are tested. The first is seeing if the algorithm shows a degradation in speed as the total number of points in the map increase. Figure 136 shows that the algorithm does not have any increase in computation as the total number of points in the map increase. Every part of the algorithm is designed to work with only the points that are viewable in the current iteration and as such this result is no surprise.

In terms of increasing the point density, there are several factors discussed that would cause the execution speed to degrade: the quadtree, the binning and the basis point optimization. As shown in Figure 143, the quadtree is the one responsible for the small polynomial growth. Figure 145 shows the polynomial growth can be eliminated with the proper choice of the spatial subdivision data structure used for the point matching. Provided the data structures are correctly tuned, the Relative algorithm can be considered $O(n_s \log n_s)$ as a worse case where n_s is the amount of observable points.

There is one more observation that is interesting to discuss. If the minimum required computation per iteration is to register every observed point, what is the minimum computation required to do this? The minimum would require a transformation step, a comparison step, and some sort of average or correlation step. Either the point needs to be transformed to its assumed global coordinates to be matched, or the untransformed point is used and the transformation occurs later. The point needs to be compared against possible matches and once matched need to be integrated to the map. This analysis does not look at adding new points or groups.

The Relative algorithm first performs the matching step using the untransformed point. Then when computing the relative map of a group, each point is transformed using the basis transformation and then averaged. ICP is used to compute the transformation to global coordinates which is then applied. It may be possible to avoid the second ICP transformation if the three basis points are already in global coordinates at the potential cost of accuracy. One computation that is unavoidable is that due to RltPoints belonging to multiple RltGroups the same processing may occur more than once on a given RltPoint.

The Relative algorithm performs its minimum computation at approximately 65% of its total average run time as shown in Figure 142. The rest of the run time is the group creation and the dynamic point detection. It is interesting to note that the computation of the Relative algorithm can be related to the minimum amount of computation required to do a SLAM algorithm.

6.7 Comparison with the EKF in simulation

The Relative Point SLAM algorithm is capable of 6D SLAM. 6D means there are no restrictions on movement in the (x,y,z) directions or the three rotations. It also does not use odometry. The only requirement is that at least 3 points are visible at all times. The previous chapter examines accuracy and notes where potential error may occur, but it is unclear how close the accuracy of the Relative Point algorithm is to the minimum amount of error, given the input data.

To verify that the accuracy is reasonable compared to other algorithms, a 6D EKF SLAM described in [Blan08] is integrated into the simulation and tested. The 6D EKF implementation has the option not to use any odometry so the comparison should be valid as they use the same data. The two algorithms are compared in a figure eight, straight line, and with dynamic points simulations. The noise used when evaluating the figure eight and straight line test is white Gaussian noise which is ideal for an EKF [ThBF06].

The 6D EKF-based SLAM is obtained from the Mobile Robot Programming Toolkit [Mrpt11] . The algorithm is integrated into the simulation the Relative Point algorithm uses. There are several options in its configuration file.

- Method: there are two methods Full EKF and EKF 'a la' Davison which is more efficient. Both were tested and the Davison method is found to be much faster which is noted in the paper and more accurate so it is used.
- force_ignore_odometry: is set to true so the odometry that is sent into the function is not used.
- There are three options for the sensor model. These are looked at experimentally and values are found that minimized the landmark error are chosen.

6.7.1 Figure eight tests

The robot is tested on the figure eight simulation used in the earlier section for performance testing. The point density is reduced from 100 points per 100 unit length to 25 due to the EKF's computation complexity. Figure 146, Figure 147, Figure 148, and Figure 149 show the result. It is interesting to see in Figure 146 how both algorithms have error increases that occur at about the same time. It is not feasible to do a full figure eight since the EKF algorithm's in Figure 148 run time becomes over a second an iteration and only getting larger. The average run time of the Relative algorithm is about 1 ms in the simulation.

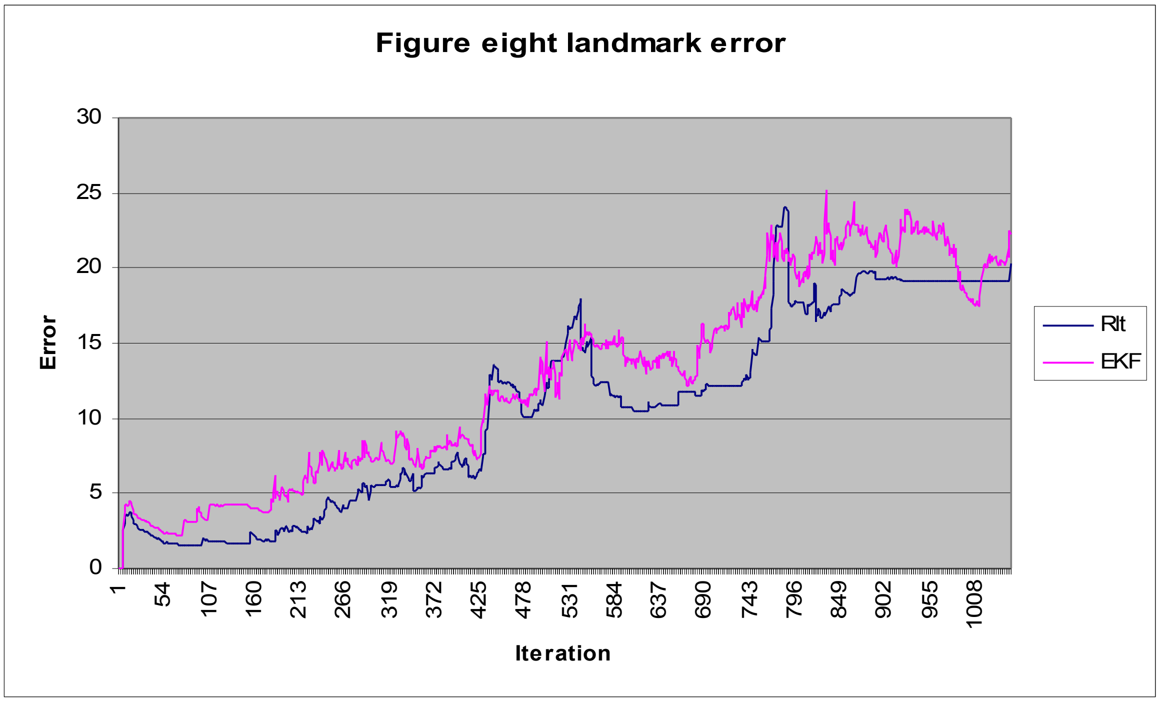


Figure 146: Figure eight landmark error

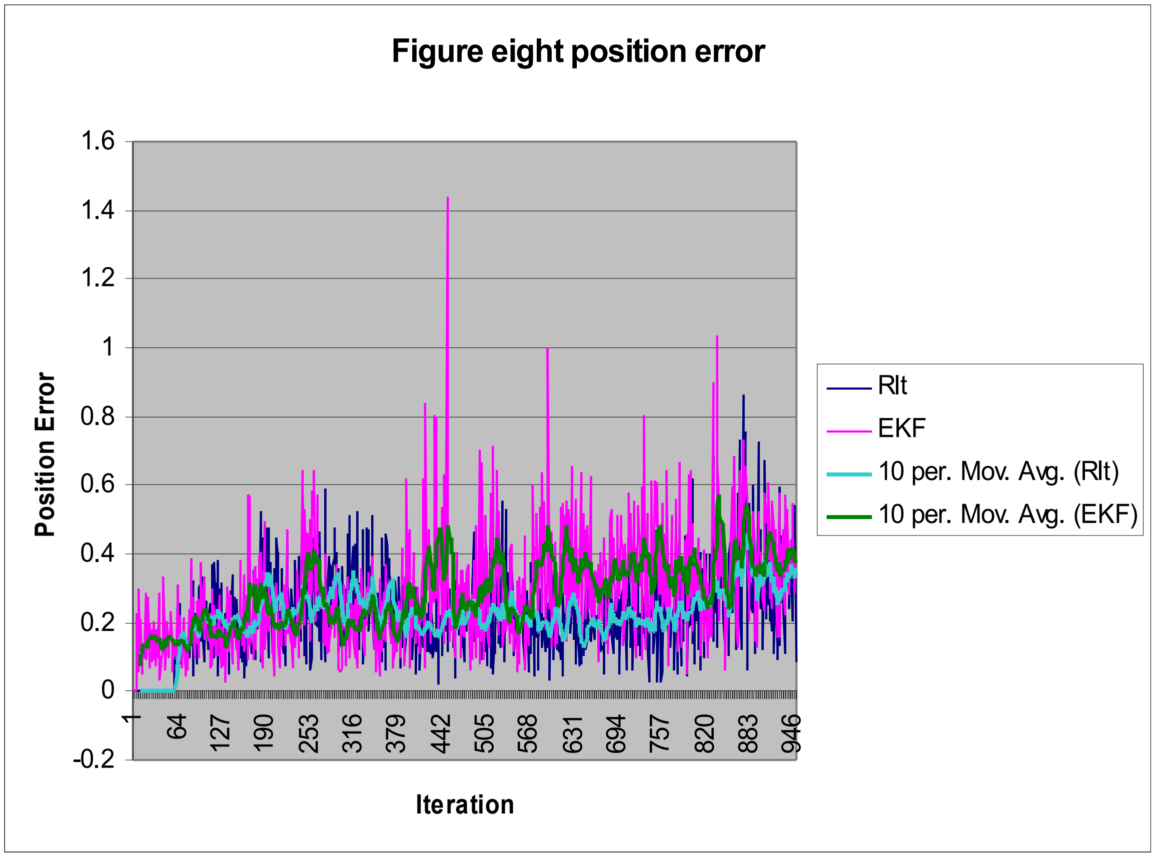


Figure 147: Figure eight position error

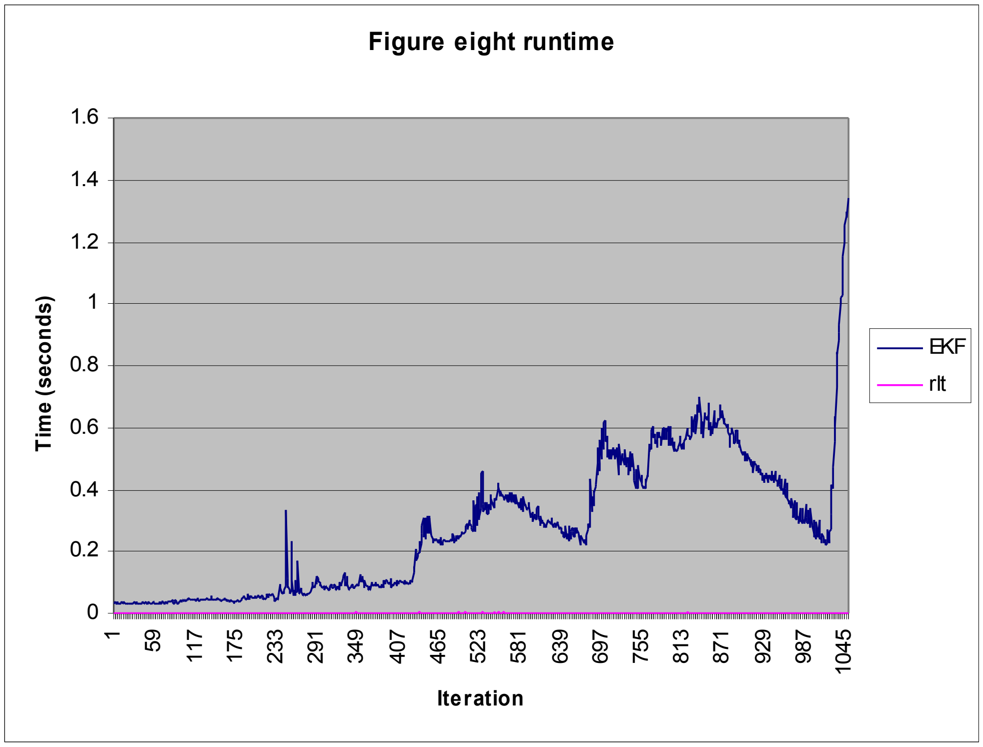


Figure 148: Figure eight run time. The Relative Point algorithm's run time average is 1 ms so it is hard to see.

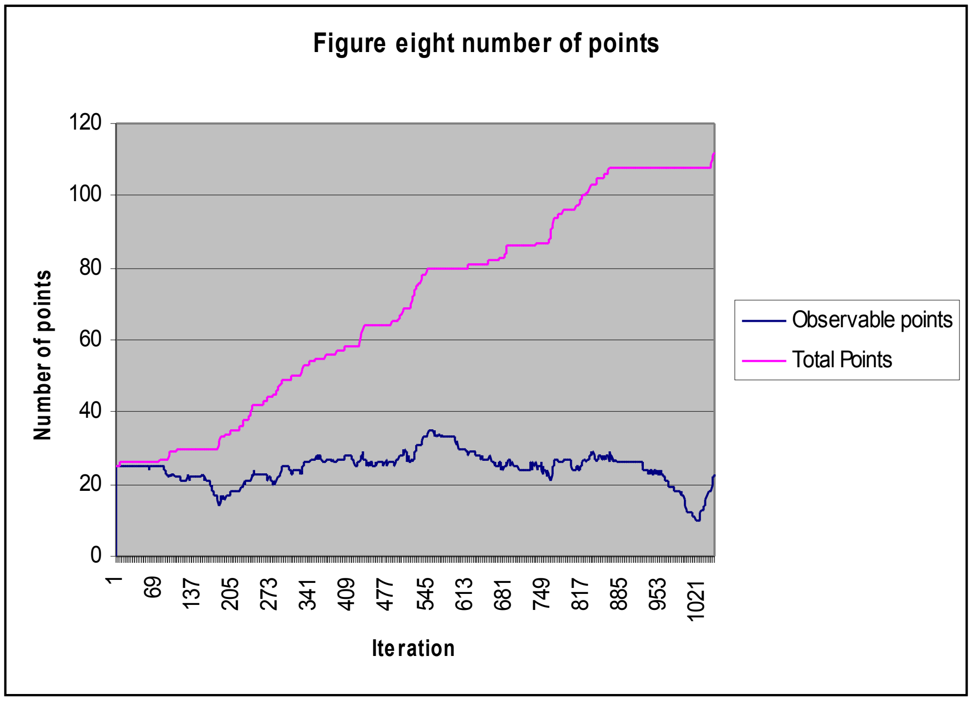


Figure 149: Figure eight point count

6.7.2 Straight line test

The next test is the straight line test. Since there are no turns, the errors on the map cannot be attributed to frustum bias. Either the error is from an inherent noise bias or algorithm inefficiency.

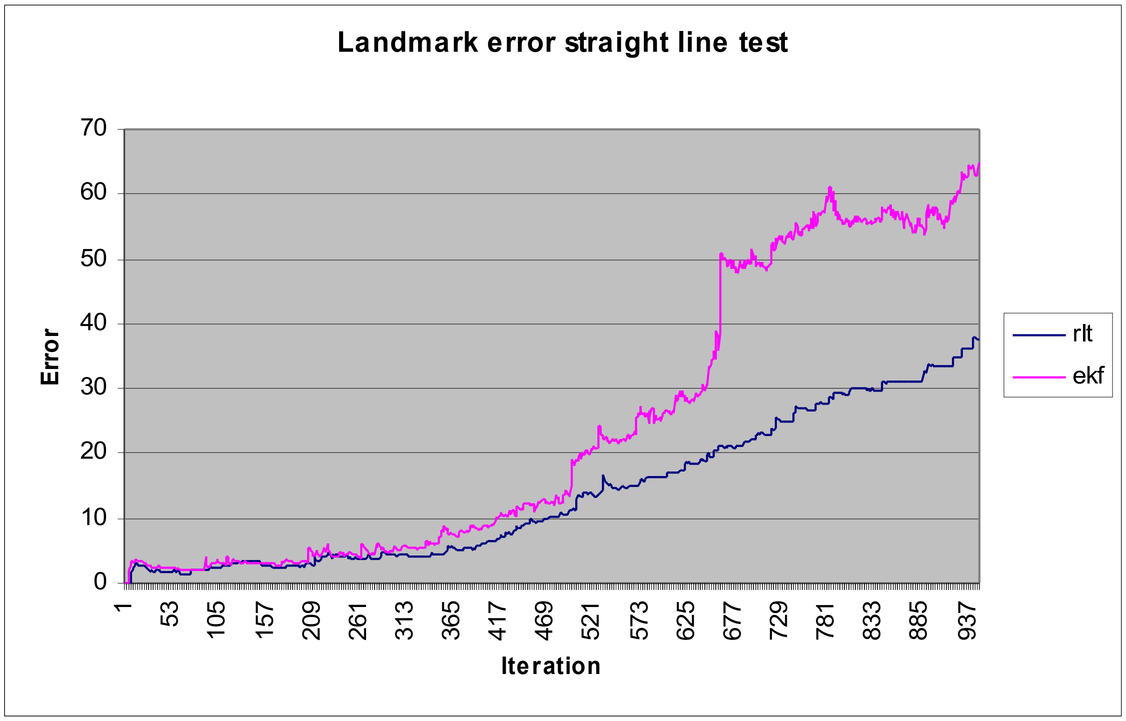


Figure 150: Landmark error in straight line test

Figure 150 and Figure 151 show the result of the straight line test. The error is very similar except for two single errors that cause the total error of the EKF to go up.

It is interesting to see the landmark error as the robot stays stationary for a long period of time. This is shown in Figure 152. The very slow convergence is likely due to the fact that the landmark errors offset each other when looking at position error. However, to have zero landmark error every landmark needs to receive the offsetting noise to be balanced. In addition the noise in every landmark must also be balanced at exactly the same time for there to be a zero landmark error.

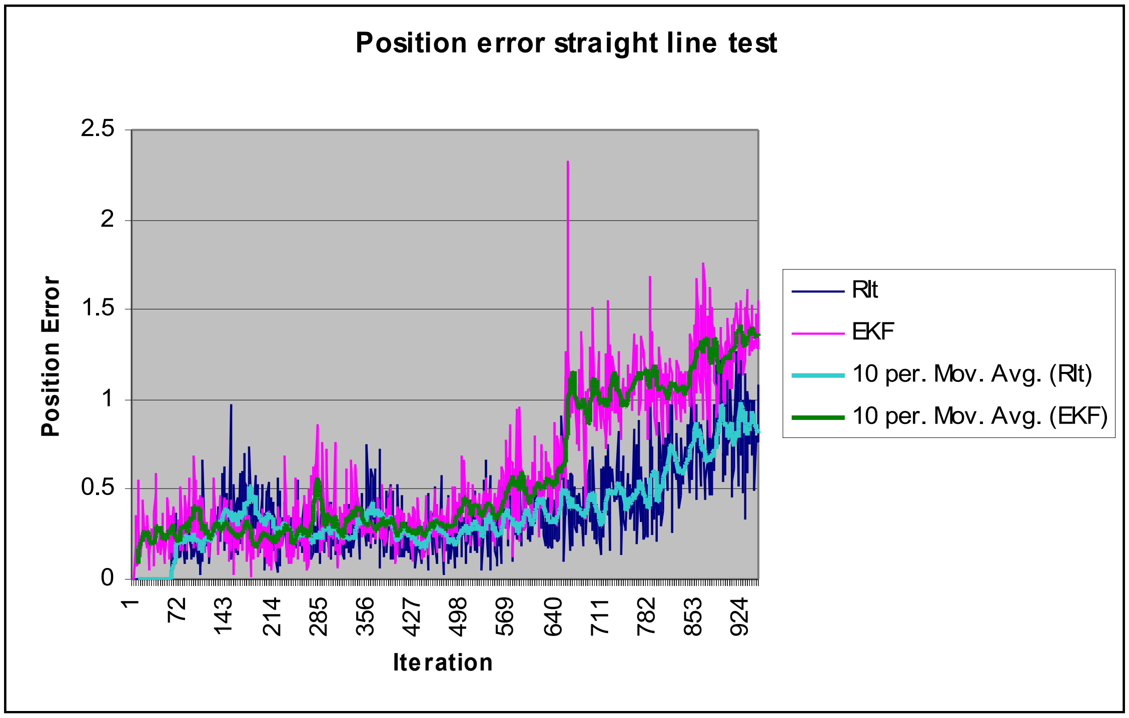


Figure 151: Position error in straight line test

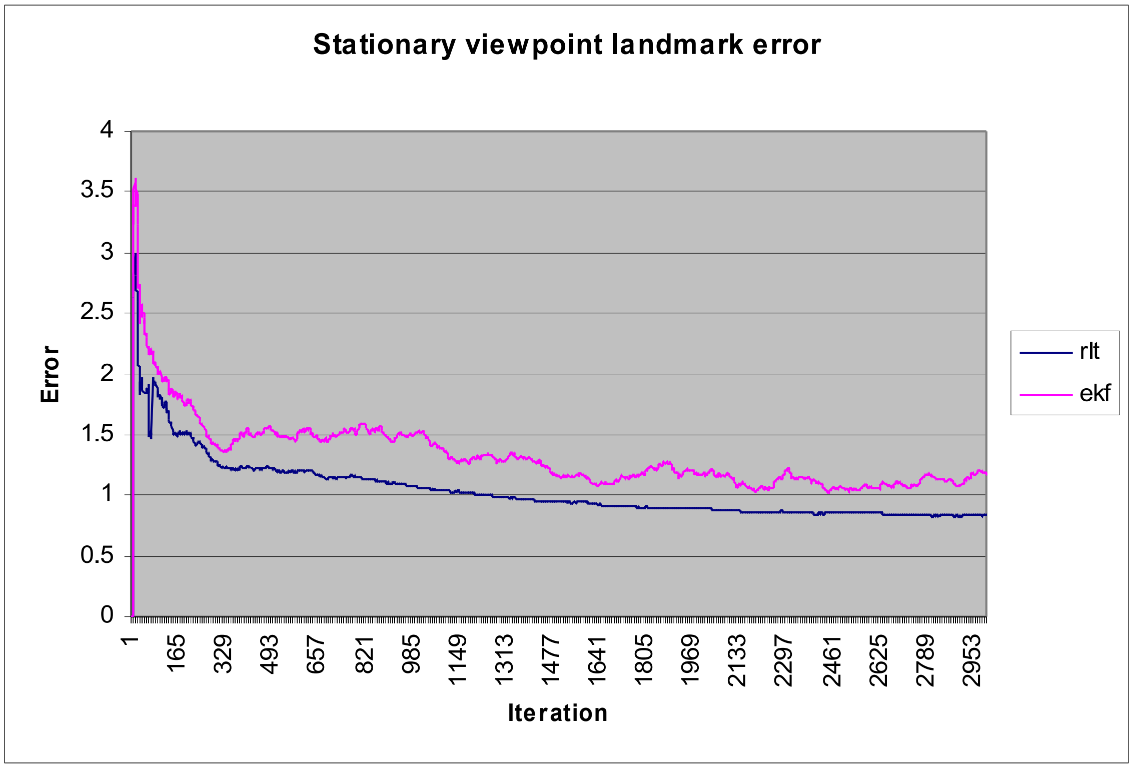


Figure 152: Stationary viewpoint landmark error

6.7.3 Dynamic point testing

It is known that the EKF does not have a mechanism to deal with dynamic points. But for the sake of completeness the figure eight simulation is tested using dynamic points. Figure 154, Figure 153, Figure 155, and Figure 156 show a sequence where two points that are dynamic causes the EKF to lose position and fail. The landmark error stopped working since it uses the nearest neighbour and the tracking was lost. The red robot is the estimated position and the blue robot is the actual position.

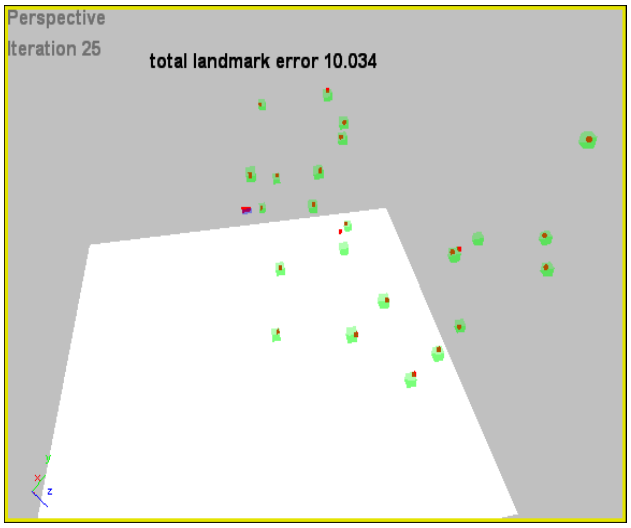


Figure 154: Dynamic error 1

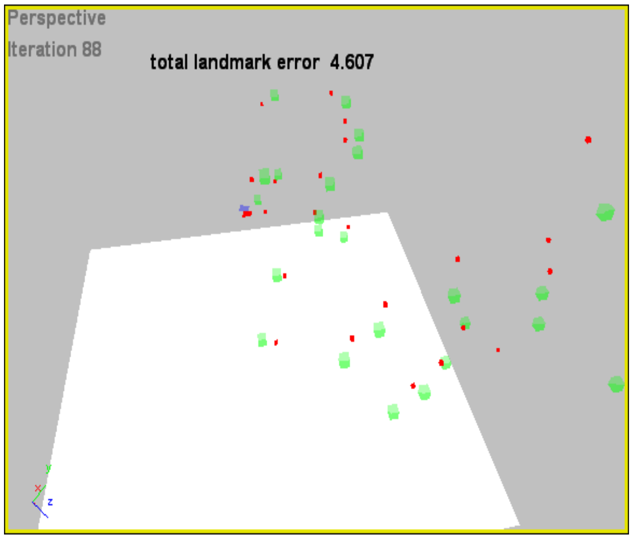


Figure 153: Dynamic error 2

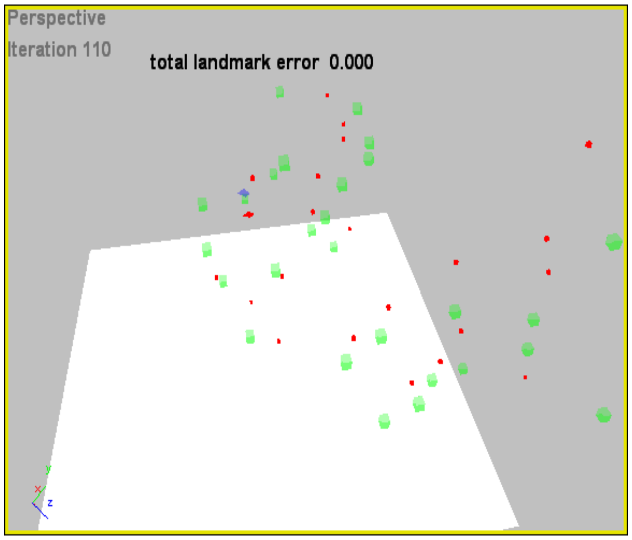


Figure 155: Dynamic error 3

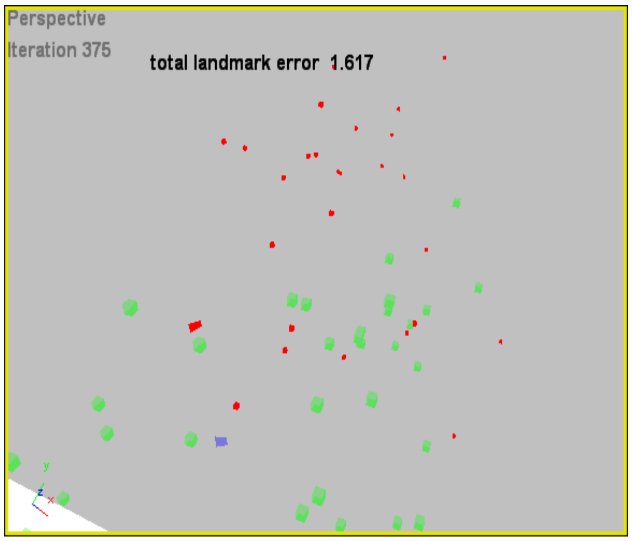


Figure 156: Dynamic error 4

The same sequence is run with the Relative Algorithm Figure 157, Figure 158, Figure 159, Figure 160, Figure 161, and Figure 164. The points as shown in Figure 157 that are dynamic are points 24 and 18. Notice in Figure 159, right before the final group creation and the binning algorithm, the landmark error is high. However the robots position is approximately correct as the current position is far less affected by noise than the EKF which uses current position as a cumulative state. In Figure 159 the binning algorithm is run and the dynamic points are identified. Notice that points 24 and 18 are in green text which signifies that it is dynamic. Figure 161 has the new landmark error update and the error is down significantly from 16.414 in the previous iteration to 1.841. Figure 164 shows the algorithm at about the same time as the EKF dynamic point test shown in Figure 156.

It is known that the EKF is susceptible to dynamic points: from [BaDu06] "The problem is that a single incorrect data association can induce divergence into the map estimate, often causing catastrophic failure of the localisation algorithm."

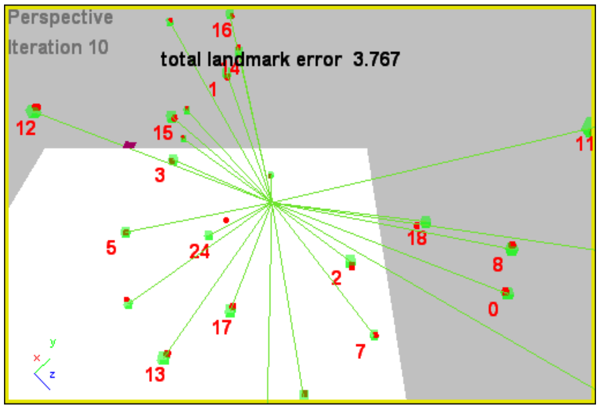


Figure 157: Dynamic detection 1

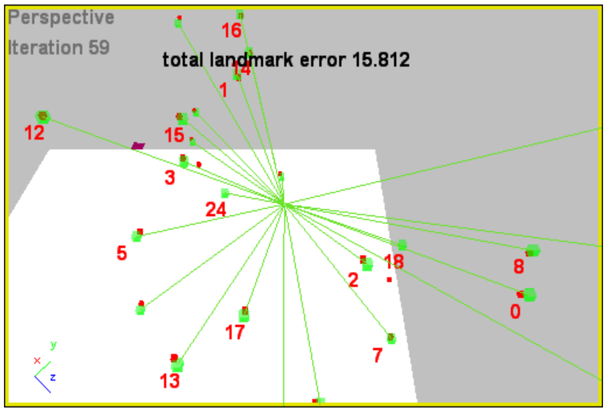


Figure 158: Dynamic detection 2

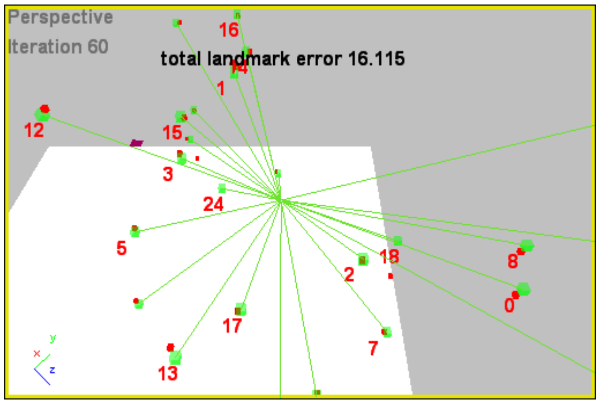


Figure 160: Dynamic detection 3

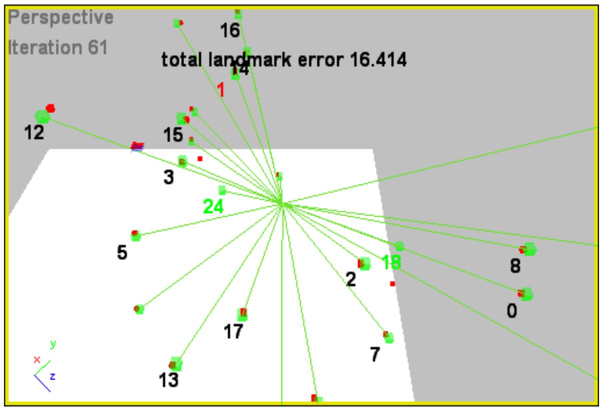


Figure 159: Dynamic detection 4

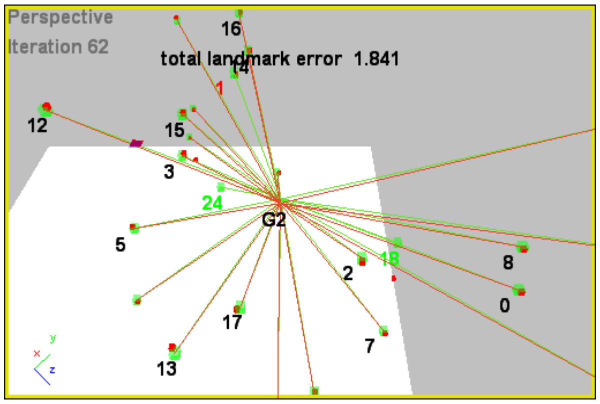


Figure 161: Dynamic detection 5

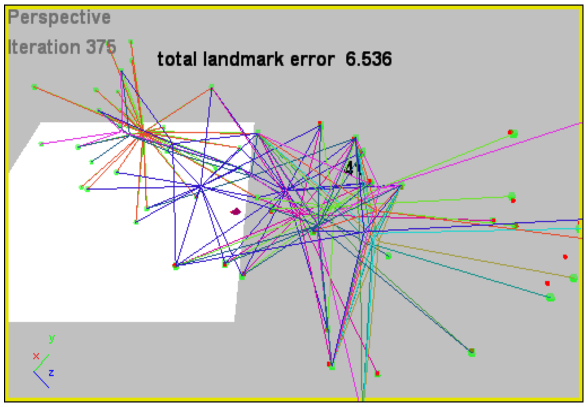


Figure 162: Dynamic detection 6

6.7.4 Summary

The purpose of this section is not to rank the EKF versus the Relative algorithm. It is too early to tell in general which algorithm is more accurate, due to the limited testing and the usage of only a single implementation of the EKF algorithm. The purpose is to provide a baseline of mapping error to determine the probable cause of the mapping error in the Relative algorithm. Since the mapping errors are comparable, the landmark error will be further explored in the next section 6.9.

It is known that a regular EKF implementation suffers from a polynomial computational complexity. There are many implementations that use sub maps such as [PaTN08] which can perform in O(n) time or even O(logn) [DaLe09]. It is likely that the Relative algorithm compares favourably to other algorithms. This is due to the Relative Point algorithm's computation complexity not being related to total points on the map rather the average count of points being observed, and its millisecond computation time.

6.8 Comparison to the EKF using the Victoria Park data set

One of the standard data sets used to test SLAM algorithms is the Victoria Park data set. The Victoria Park data set contains many iterations with less than four points. The relative point algorithm requires 3-4 points visible so it is unable to use this data set without the use of a motion model. However, there are two intervals that have four or more points and those intervals can be evaluated. The results are compared to [PiPa08]. The two intervals are from iteration 1181 to 1642 and from iteration 4220 to 4634.

The results of iterations 1181 to 1642 are shown in Figure 163 and Figure 164. Figure 164 shows the same landmark locations given in Figure 163 manually stretched over the results from [PiPa08].

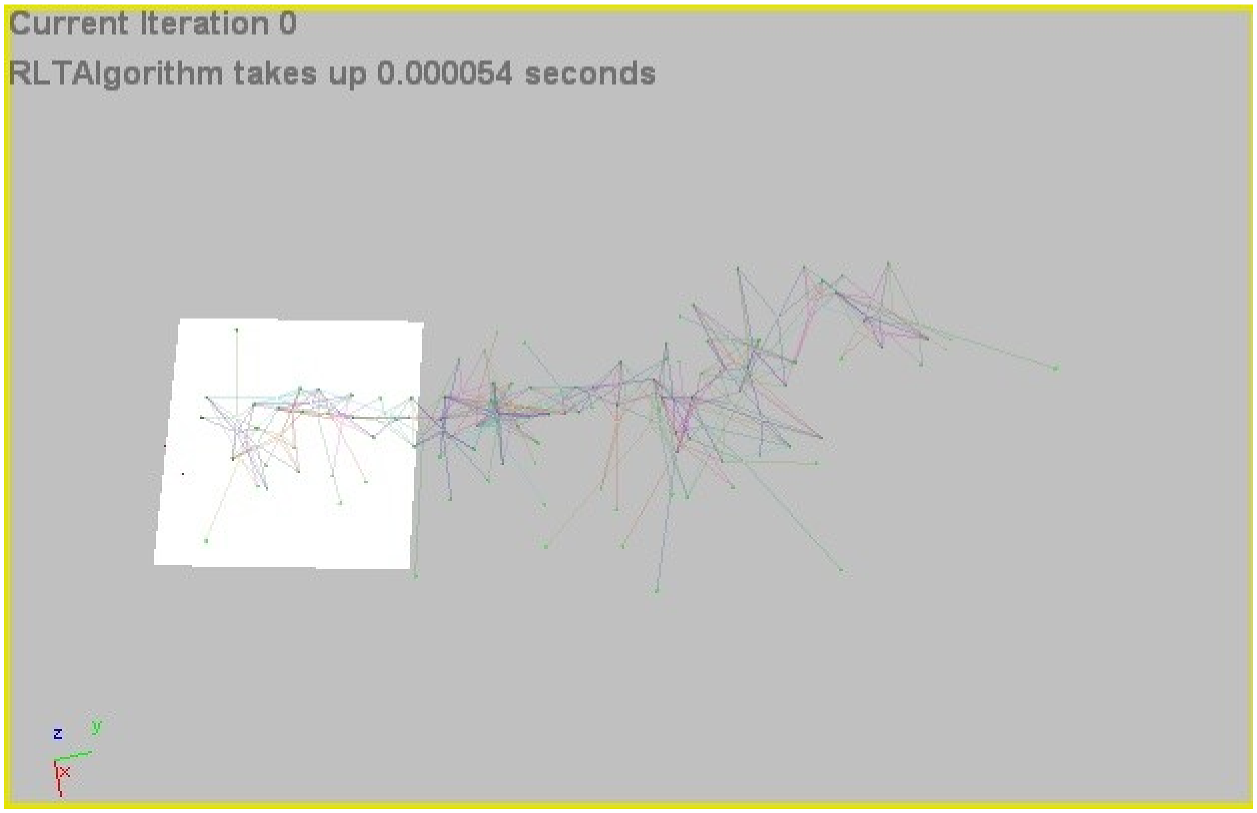

Current Iteration 0
RLTAlgorithm takes up 0.000054 seconds

Figure 163: Results of the relative algorithm form iteration 1181 to 1642

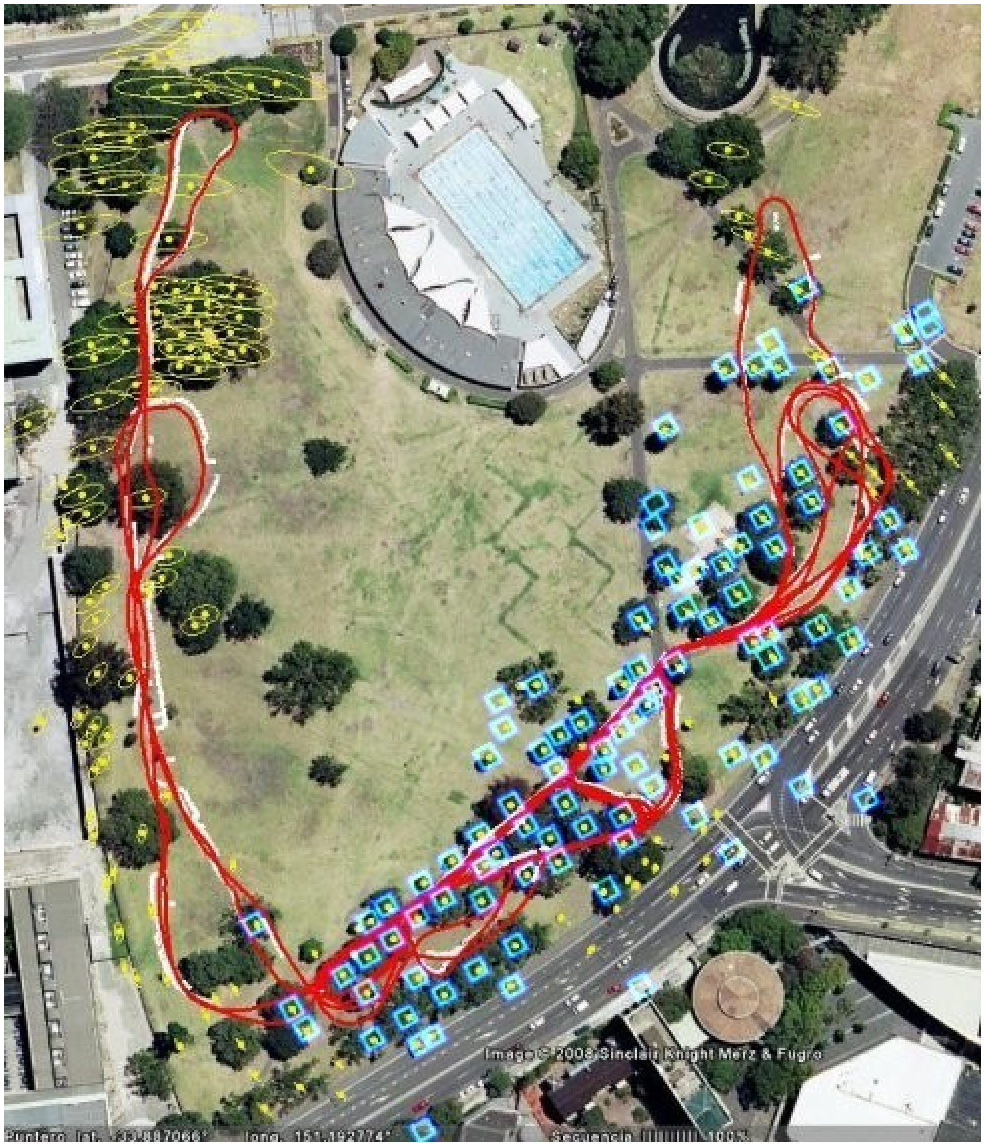


Figure 164: Results of the relative algorithm from iterations 1181 to 1642. The relative algorithm's landmark locations are shown by the blue squares. The results given by [PiPa08] are shown by the yellow circles.

The results of iterations 4220 to 4634 are shown in Figure 165 and Figure 166. Figure 166 shows the same landmark locations given in Figure 165 manually stretched over the results from [PiPa08].

The results are generally consistent with [PiPa08]. The full Victoria Park data set is 7249 iterations so the relative algorithm performs well given only a portion of the data set.

Evaluating the computation time: if only one level of grouping is used, the average time per iteration is 0.000296 seconds. If a second level of grouping with dynamic detection is used, the average time per iteration is 0.0004. If a third grouping which is formed after a point is no longer visible is used, the average time is 0.00057 seconds. Multiplying the averages by 7249 gives a total of 2.15 seconds, 2.9 seconds, and 4.13 seconds.

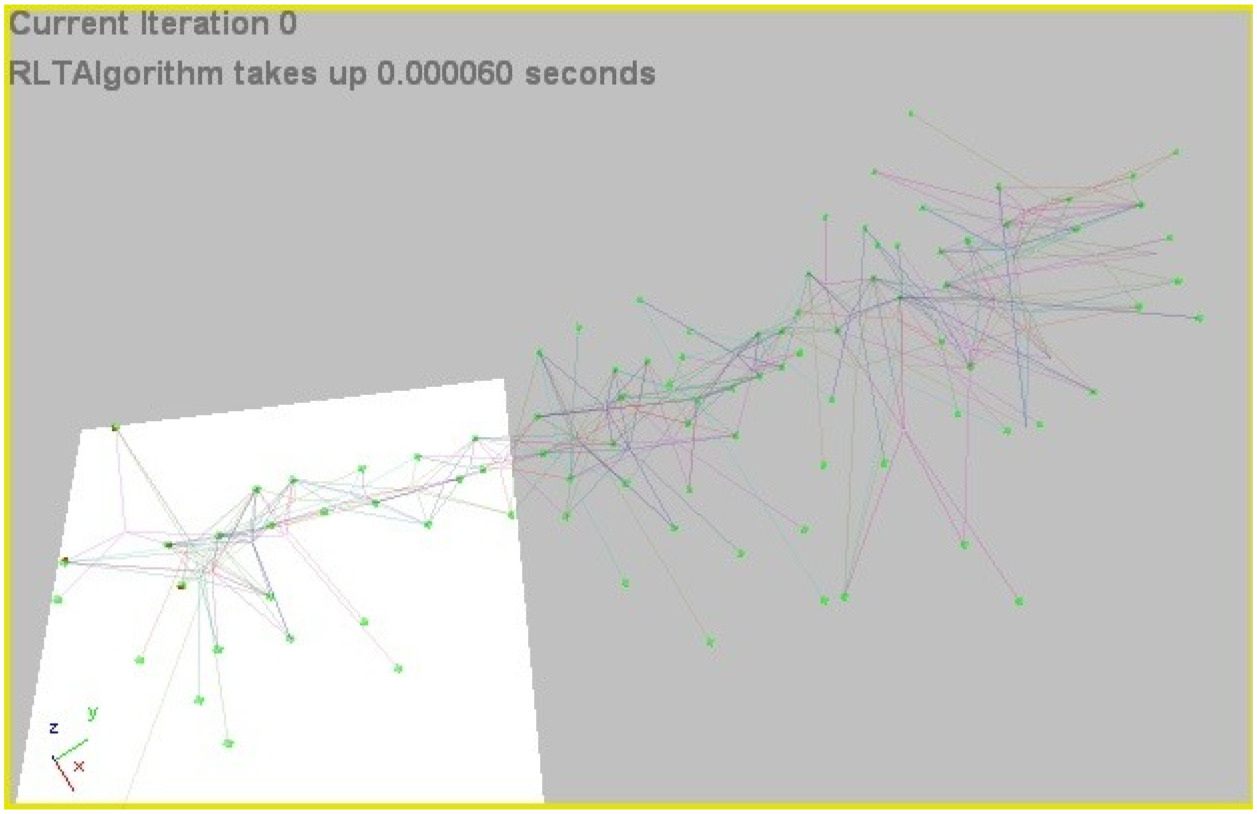

Current Iteration 0
RLTAlgorithm takes up 0.000060 seconds

Figure 165: Results of the relative algorithm from iterations 4220 to 4634

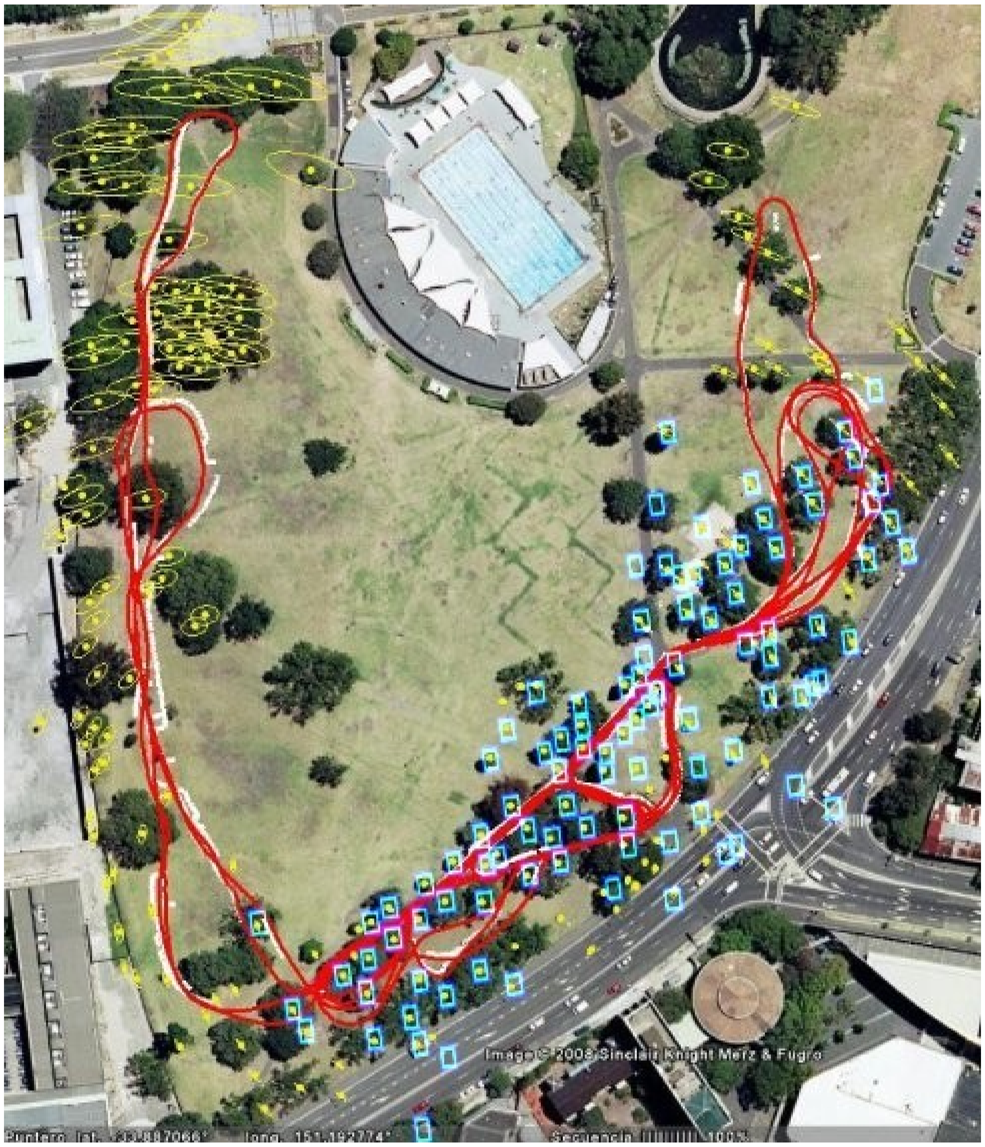


Figure 166: Results of the relative algorithm from iterations 4220 to 4634. The relative algorithm's landmark locations are shown by the blue squares. The results given by [PiPa08] are shown by the yellow circles.

6.9 Forty loops through the figure eight

Figure 152, stationary viewpoint landmark error, is interesting as it shows the landmark error steadily decreasing to zero. If error can be shown to approach zero by having the robot stay stationary, will the error approach zero after many loops through the figure eight simulation? For this test, the robot traverses the figure eight about 40 times without any dynamic points. Dynamic points are not used since the dynamic points may cross normal ones and may create a bias.

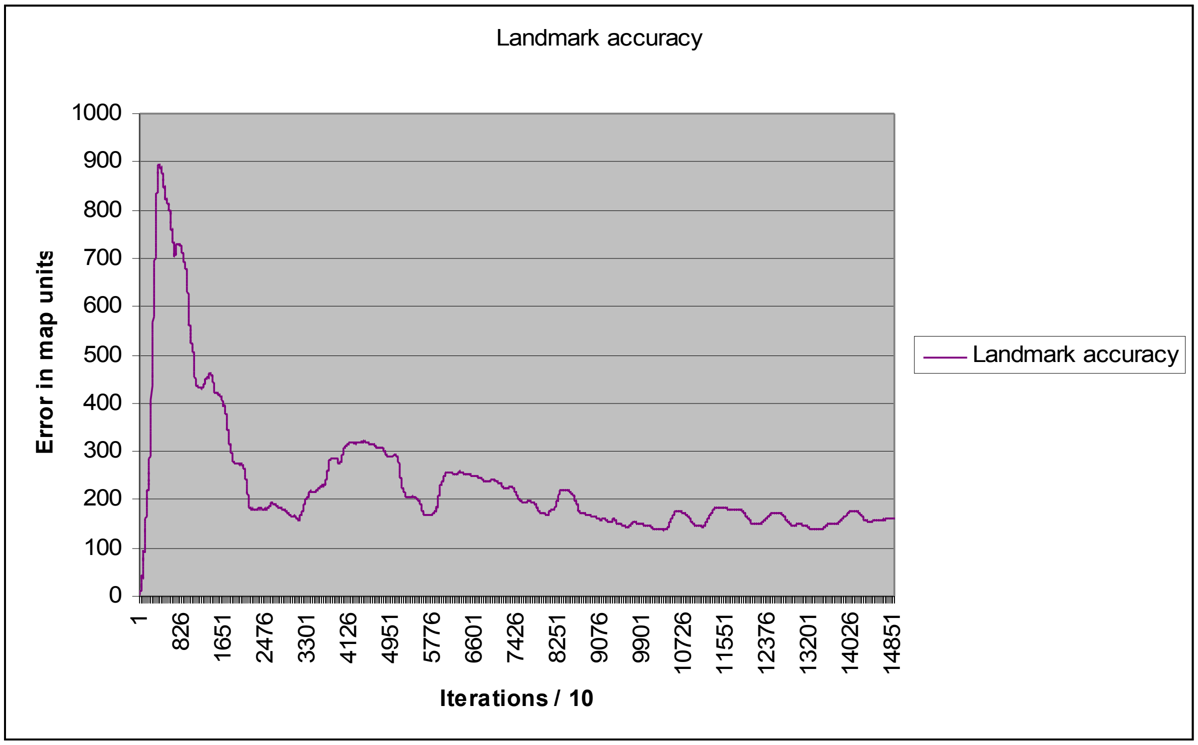


Figure 167: Landmark accuracy of 40 runs

Figure 167 appears similar to Figure 152. Since the landmark error is decreasing slowly perhaps it is balanced in that the position error is near zero. Figure 168 shows the position error for the 40 loops. The position error will never be zero since the noise in a single iteration will always cause some sort of error. It does appear periodic though, so perhaps looking at the x, y and z position error will be helpful.

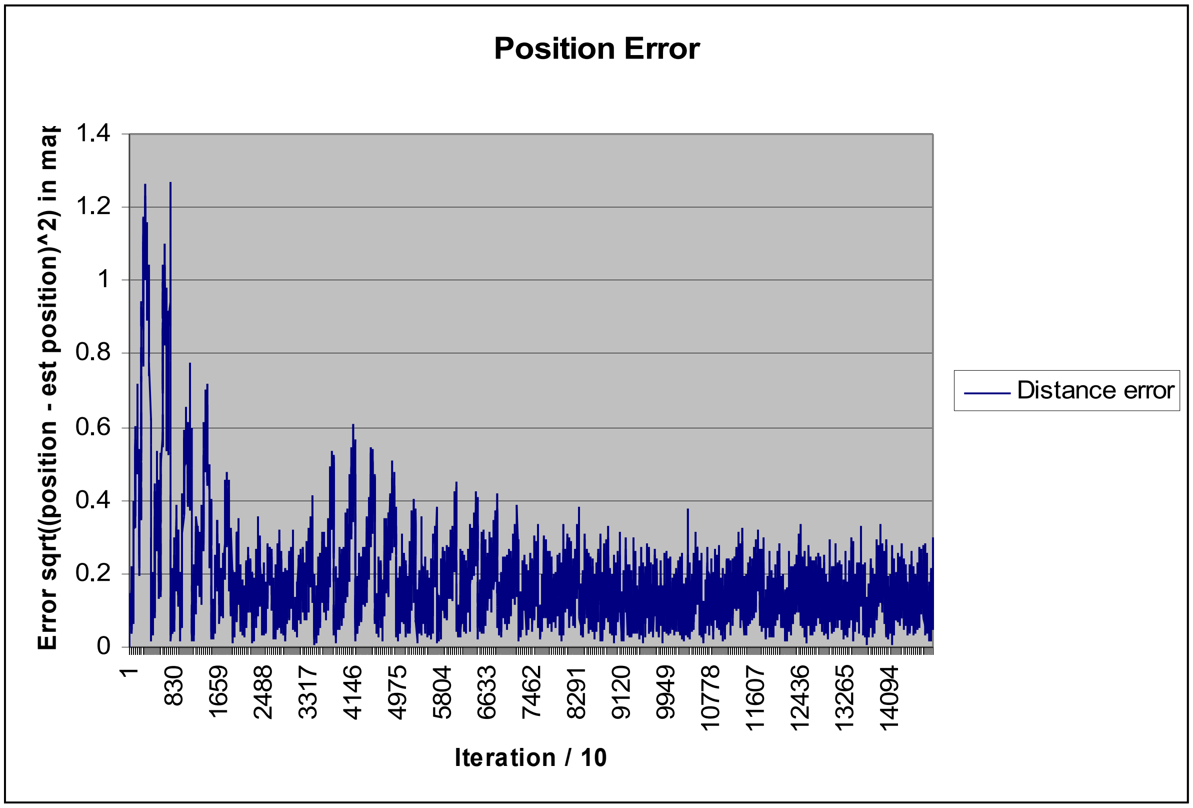


Figure 168: Position error of 40 runs

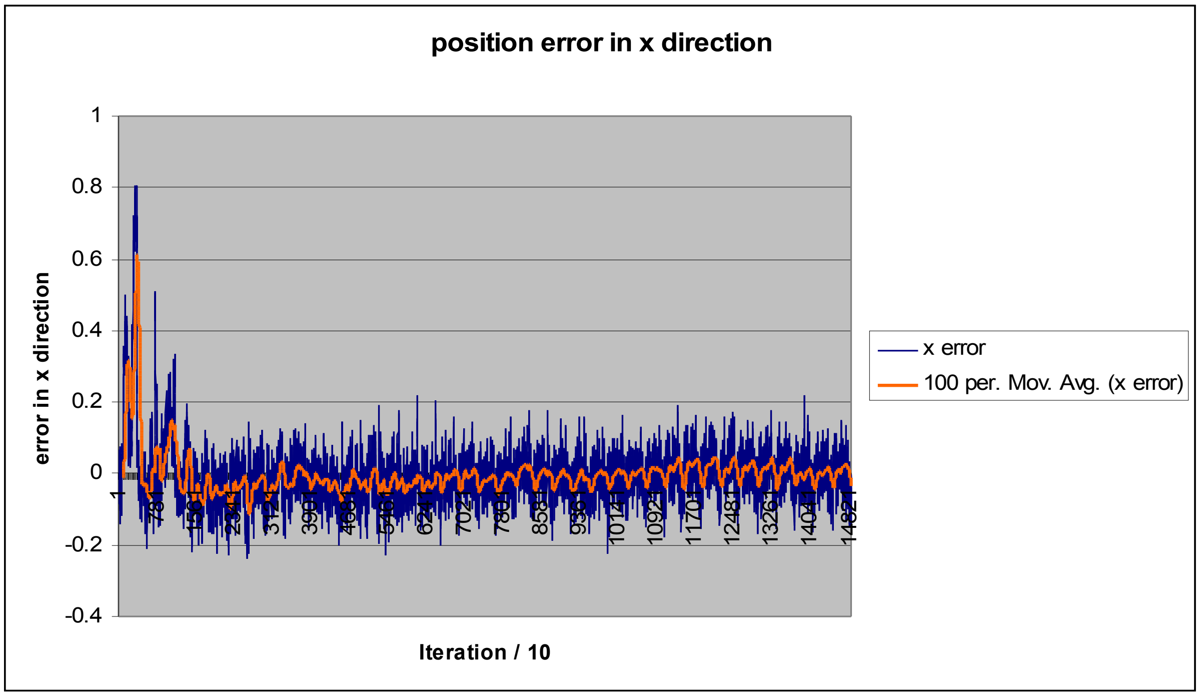


Figure 169: Error in x direction of 40 loops

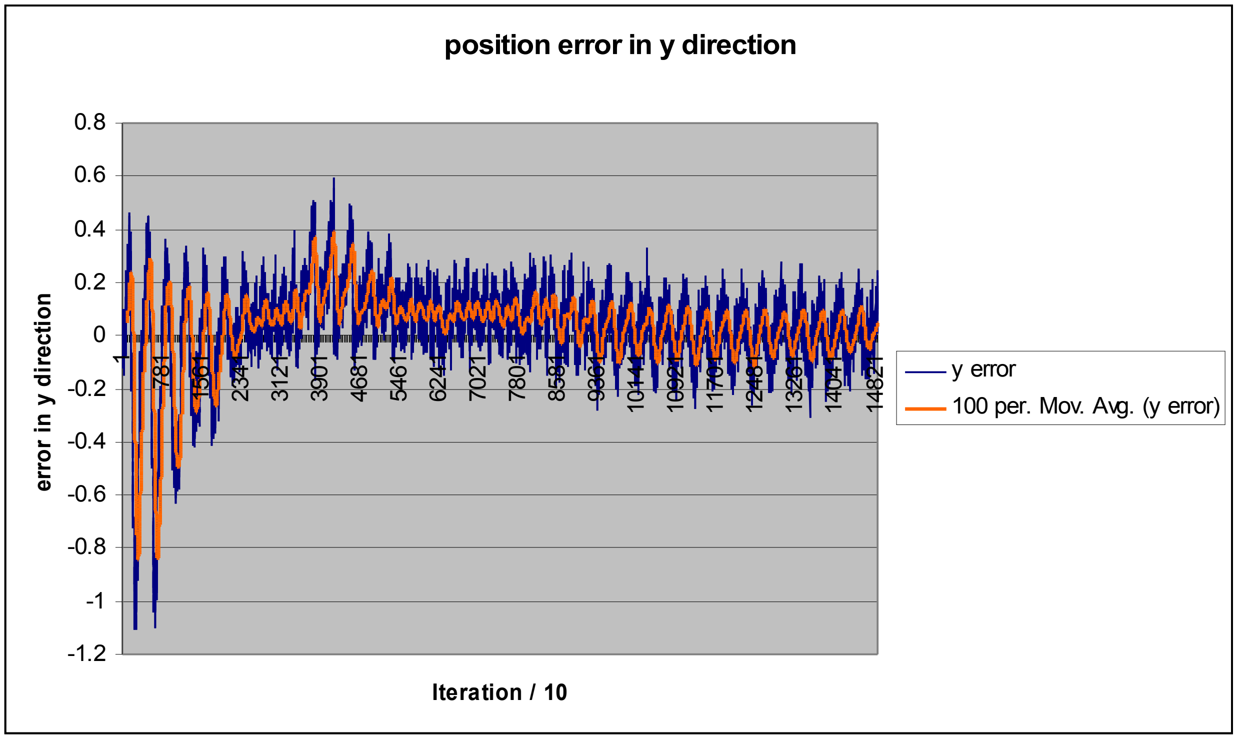


Figure 170: Error in y direction of 40 loops

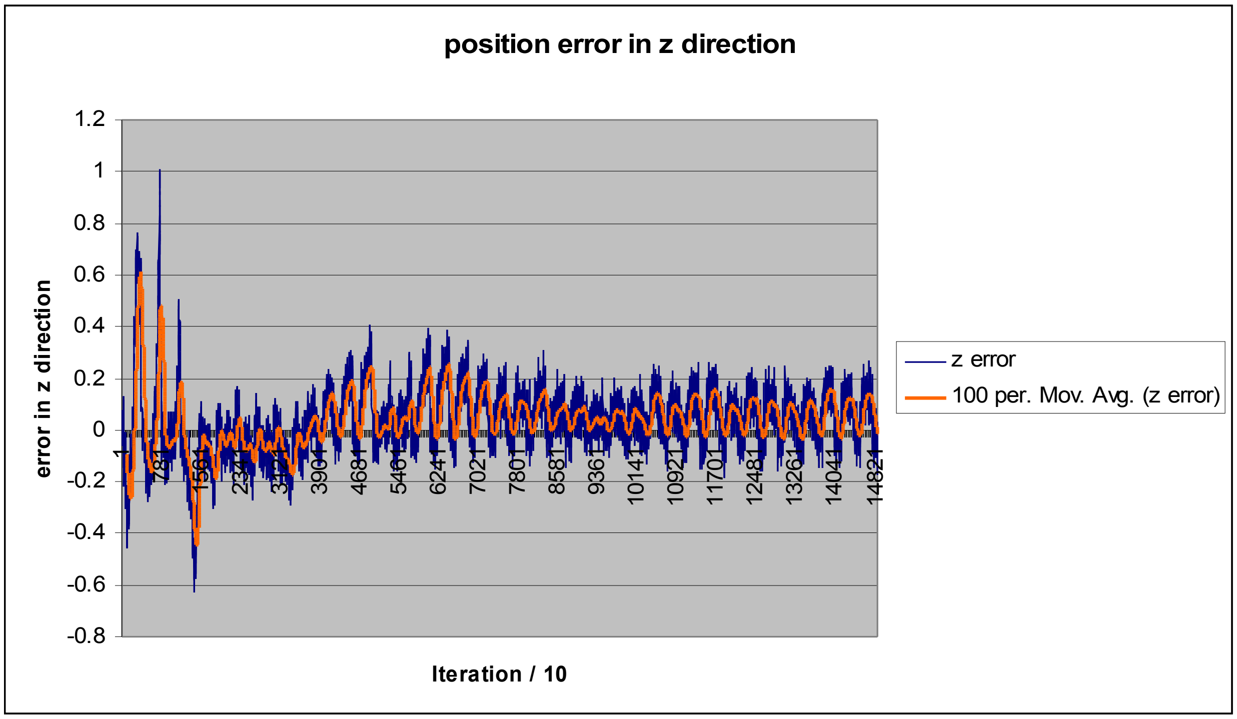


Figure 171: Error in z direction of 40 loops

Figure 169, Figure 170, and Figure 171 show the x,y, and z error for the 40 simulation loops. Each loop is about 3800 iterations with only every tenth iteration being plotted on the chart due to limits in the charting software. It is clear that the errors are periodic with some random but decreasing amplitude. Towards the middle of the charts the amplitude stabilizes. It is not clear where the error comes from but due to the slope of the error being constant it is likely only occurring during a turn and propagating forward. The error might be from the Relative algorithm or it might be from a bias in the environment used in the simulation.

6.10 Comparison of the two relative algorithms

The Relative Point algorithm's architecture is designed with the benefit of knowledge gained from implementing the Relative Plane algorithm. The improvements are:

- Rather than compare current observations of untransformed planes against previous ones from the current RltXPoint, untransformed points are compared against all previously viewable RltPoint stored in RltPointCharting. This solves some issues with maintaining the best current RltXPoint.
- There is no linking of RltGroups (RltXPoints) as they are soft linked together using the RltGroupRef structure. This allows any changes of RltGroups to happen without breaking any links. Any changes to past data can be safely done without requiring a roll back.
- The reordering is properly done in the way RltGroups are put together. Many issues with RltPlanes not being in a large enough intervals are settled.
- Much of the complexity is taken out of the RltGroups, in that all of the high level processing is done by RltMapper. If there are issues with changing groupings, a new RltGroup is created. Since the groups are static, it is possible to save the average relative location per RltPoint. This acts as both the cache system and the save system.
- The algorithm is now full 6D rather than just 2½D.

The Relative Plane algorithm took considerably more code and implementation time than the Relative Point algorithm. Most of the extra work is due to the complexity of identifying which planes are growing and shrinking. There would be improvements in reimplementing the Relative Plane algorithm, but much of the extra complexity would likely remain. This extra complexity is why the Relative Plane algorithm's description is broken into three sections with 72 pages (without counting 4.2), while it only takes 37 pages to describe the Relative Point algorithm

It is also curious to note that in regards to the "Second System Effect", the Relative Point algorithm seems to be implemented approximately correct considering it is the second Relative algorithm. Of course, this cannot be confirmed without seeing what the third implementation would look like.

6.11 Summary

The Relative Point algorithm matches points to the previous iterations untransformed observations. All of the iterations of a single point instance are stored in a data structure, the RltPoint. RltPoints are grouped together in RltGroups based on observation interval. To create a relative map, a RltGroup calculates the average untransformed relative point locations, using every iteration all of the RltPoints are observed together. Three points in the group are used as the basis for rotational and translation invariance. Common points between groups are used to performed an ICP to join groups together to form a global map. The only requirement of the algorithm is that three points are visible at all times.

The Relative Point algorithm has a worst case computation complexity of $O(n_s \log n_s)$ where n_s is a subdivision of the total points in the map. n_s is the average quantity of points visible at the same time. The computational complexity is dependent on the data structures used, in particular the structure used to do the point matching. It is important to tune the quadtree or octree so that it maintains its $O(n \log n)$ computation complexity. The average computation time of the 550 points per 100 units figure eight simulation is 18 ms per iteration, where there is an average of 537 points per observation and 8980 total points on an AMD 64 3400+.

The accuracy of the Relative Point algorithm is compared to a 6D EKF implementation that does not use odometery. The accuracy of the relative algorithm is shown to be comparable to the EKF. It is too early to tell in general which algorithm is more accurate, due to the limited testing and the usage of only a single implementation of the EKF algorithm.

The accuracy of the Relative Point algorithm is also evaluated by having the robot loop through the same figure eight many times to see if the error approaches zero. After many runs it appears to be reduced to having a small bias. It is not known if this bias is due to the algorithm or due to bias in the noise in the environment.

Chapter 7 Conclusion

This path taken to arrive at this point is an interesting one. The original intent is to implement a full vision system that can produce accurate maps in an indoor office environment. This thesis follows a previous project in indoor environments that is successful but limited. One of the issues with the previous project is the inaccuracy of the robots position after a turn. It is this issue that posed the hallway problem. If the robot is traveling down a hallway where the wall at the end is at a known angle to the current hallway, why cannot the orientation of the robot be trivially computed after the turn? This led to the use of planes, which store orientation that should inherently solve the problem.

This thesis started by repeating the design process of the previous work by producing a 3D simulation, the robot design and the FPGA design. Each worked individually but could not be integrated into a full project which came as a surprise due to how well the integration worked in the previous project. This is a textbook example of the "second system effect" discussed in software engineering literature.

What may be considered a failure, led to the focus of the thesis on Relative algorithms. There are several attempts in this thesis of SLAM algorithms using planes as the primitive and storing as much data as possible. These attempts had the misguided intention that the key to the algorithm was to reduce error in position as much as possible and modeling plane growing and shrinking as noise. Finally the Relative Plane algorithm is created that is based on maintaining the relative relationships of planes without regards to maintaining the current position. Position is only generated by corresponding an iteration readings to a map, and only used locally for backtracking and globally for closing the loop.

The fact that the Relative Plane algorithm compares planes in pairs led to the solution of the planes growing and shrinking problem. By using the top two corner points, assumed midpoints can be created using the maximum size of the plane. When comparing two planes, the assumed midpoints can be used creating four comparison pairs. The pair with the lowest standard deviation identifies the static edges of the two planes.

The Relative Plane algorithm also has a good solution for when only one partial plane is visible. With only one partial plane, it is shown that it is only important to track how the plane expands. The position of the partial plane is previously calculated when it was seen in the past with the other planes in its grouping. This solves the initial question posed of the hallway problem.

After the completion of the implementation of the Relative Plane algorithm, there was an interesting observation. The goal of the Relative Plane algorithm is to accurately map a planar environment and to do this, it is required to identify planes that are growing and shrinking. The Relative Plane algorithm is able to both filter out noise and identify dynamic edges of planes. The dynamic edge detection comes at a small computation cost and is inherent in the architecture of the algorithm.

It became evident that if the Relative Plane algorithm can both handle noise and some dynamic movement perhaps it can do something similar using points as the input. The Relative Point algorithm uses the Relative Plane algorithm as a template to implement a Relative algorithm using points. The Relative Point algorithm is shown to have a worst case computation complexity of $O(n_s \log n_s)$. n_s is the average quantity of points observed in a given observation, that is, a subdivision of the total quantity of points on the map. The Relative Point algorithm is able identify points with movement not correlated to the viewpoint at a low cost. It is compared against an implementation of a 6D no odometry EKF and is shown to have similar accuracy.

After thorough testing it is not clear if the errors in the algorithm are inherent to the algorithm, or inherent in the input data.. There is the potential drawback that the landmark comparisons are discretized into groups. It is clear though, given data with dynamic features such as dynamic edges on planes or dynamic movement in points and a requirement of real time processing, using a Relative algorithm can be a good solution.

Bibliography

[ArHB87] K.S. Arun, T.S. Huang, and S. D. Blostein. Least Square fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intellgience,* 9(5):698-700, 1987

[BaDu06] T. Bailey, H. Durrant-Whyte, H. Simultaneous localization and mapping (SLAM): part II. *IEEE Robotics & Automation Magazine,* September 2006.

[BeMc92] P. Bsel, N. McKay, A Method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol 14, NO .2 Febuary 239-256 1992.

[Blan08] J. L. Blanco. Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM. *Technical Report, Perception and Mobile Robots Research Group*, University of Malage, Spain, 2008

[Broo95] F. Brooks, The Mythical Man-Month Anniversary Edition. *Addison Wesley Longman, Inc.* 1995.

[CGDM09] J. Civra, O. Grase, A. Davison, J. Montiel, 1-Point RANSAC for EKF-Based Structure from Motion. *IROS.* 2009

[Csor97] M. Csorba, Simultaneous Localisation and Map Building, *Phd Thesis, University of Oxford* 1997

[DaLe09] C.Dario, C. Lerma, J. Niera, SLAM in O(log n) with the Combined Kalman - Information filter, *International Conference on Intelligent Robots and Systems* 2009.

[Demp77] A. P. Dempset; N. M. Laird; D.B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodolofical),* Vol. 39, No. 1 pp 1-38, 1977.

[DNCD01] M. Dissanayake, P. Newman, S. Clark, H. Durrant-Whyte, M. Csorba, A Solution ot the simultaneous localization and map building (SLAM) problem. *IEEE Transactions on Robotics and Automation* V 17 I 3 Jun 2001.

[DrGe03] Doctor George, Fitting a plane into a set of points. DR. Math Forum http://mathforum.org/library/drmath/view/63765.html 2003.

[EHMD07] L. Ellekilde, S. Haung, J. Miro, G. Dissanayake, Dense 3D map construction for indoor Search and Rescue. *Journal of Field Robotics (JFR 2007),* vol. 24, no. 1/2, Feb 2007, pp 71-89

[ElSL06] P. Elinas, R. Sim, J. Little, sigmaSLAM: Stereo Vision SLAM Using the Rao-Blackwellised Particle Filter and a Novel Mixture Proposal Distribution. *International Conference on Robotics and Automation,* May 2006.

[Fres07] U. Frese, Efficient 6-DOF SLAM with Treemap as a Generic Backend. *2007 IEEE International Conference on Robotics and Automation,* Roma, Italy, April 2007

[FuWi03] S. Funiak, B.C. Williams, Multi-modal particle filtering for hybrid systems with autonomous mode transitions. *Proc. 14th Int. Workshop Principles Diagnosis (DX03)*, 2003

[GaSo04] M. Garcia, A Solanas, 3D Simultaneous Localization and Modeling from Stereo Vision, *Proceeding of the 2004 IEEE International Conference on Robotics & Automation.* 2004

[HaSB02] D. Hahnel, D. Schulz, W. Burgard, Map Building with Mobile Robots in Populated Environments., *Proceedings of the 2002 IEEE/RSJ Intl, Conference on Intelligent Robots and Systems, EPFL.* Lausanne, Switzerland October 2002.

[HBFT03] D. Hahnel, W. Burgard, D. Fox, S. Thrun, An Efficient FastSLAM Algorithm for Generating Maps of Large-Scale Cyclic Environments from Raw Laser Range Measurements. *Proceedings of the 2003 IEEE/RSJ Intl, Conference on Intelligent Robots and Systems.* Las Vegas, Nevada, October 2003.

[Husa10] S. F. Husain, Evaluational of Methods for 3D Environment Reconstruction with Respect to Navigation and Manipulation Tasks for Mobile Robots. *Masters of Science Thesis,* Blekinge Institute of Technology November 2010.

[Krau02] J. Kraut, An Autonomous Navigational Mobile Robot, *University of Manitoba, Bsc Thesis,* 2002.

[Krau06] J. Kraut, Edge Detection Using an Altera Stratix Nois2 Development Kit, *Canadian Conference on Electrical and Computer Engineering CCECE*, Ottawa, May 2006.

[LeWh91] J. Leonard, H. Whyte, Simultaneous Map Building and Localization for an Autonomous Mobile Robot, *International Workshop on Intelligent Robots and Systems,* November 1991.

[LMSK03] B. Lisien, D. Morales, D. Silver, G. Kantor, I. Rekleitis, H. Choset, Hierarchical Simultaneous Localization and Mapping. *Proceedings of the 2003 IEEERSJ Intl. Conference on Intelligent Robots and Systems* Las Vegas, Nevada. October 2003.

[Lowe04] D. Lowe, Distinctive Image Features from Scale-Invariant Key points, *International Journal of Computer Vision,* 60,2, pp. 91-110, 2004.

[LuMi97] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots,* 4:333-349, 1997.

[Mema09] Star Trek memory alpha http://memory-alpha.org/en/wiki/Cause_and_Effect_(episode), Retreived December 2009

[MiSi06] A. Milella, R. Siegwart, Stereo-Based Ego-Motion Estimation Using Pixel Tracking and Iterative Closest Point, *Proceedings of the Fourth IEEE International Conference on Computer Vision Systems (ICVS),* 2006.

[Mora96] H. Moravec, Robot Spatial Perception by Stereo Vision and 3D Evidence Grids. *CMU Robotics Institute Technical* Report CMU-RI-TR-96-34 1996

[MSCN09] C. Mei, G. Sibley, M. Cummins, P. Newman I. Reid, A Constant Time Efficient Stereo SLAM System. *In Proceedings of the British Machine Vision Conference* September 2009

[Mrpt11] Mobile Robot Toolkit 6D-SLAM http://www.mrpt.org/6D-SLAM, Retrieved January, 2011.

[MTKW02] M. Montemerlo, S. Thrun, D. Killer, and B. Wegreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. *In Proceedings of the AAAI National Congference on Artificial Intelligence,,* Edmonton, Canada, 2002. AAAI

[MuLi00] D. Murray, J. Little, Using Real-Time Stereo Vision for Mobile Robot Navigation. *Autonomous Robots 8. p 161-171* 2000

[Newm99] P. Newman On the Structure and Solution of the Simultaneous Localization and Map Building Problem. *PhD Thesis University of Sydney* 1999.

[PaTN08] L. M. Paz, J. D. Tardos and J. Neira, Divide and Conquer : EKF SLAM in O(n). *IEEE Transactions on Robotics*. Volume 24, No. 5, October 2008.

[PiPa08] P. Pinies, L. M. Paz and J.D Tardos, CI-Graph: an efficient approach for Large Scale SLAM, *In Proceedings IEEE International Conference on Robotics and Automation 2009*, Kobe, Japan.

[PiTa08] P. Pinies, J. Tardos, Large-Scale SLAM Building Conditionally Independent Local Maps: Application to Monocular Vision, *IEEE Transactions on Robotics* V 24 I 5 Oct 2008.

[SaEs04] J. Saez, F. Escolano, A Global 3D Map-Building Approach Using Stereo Vision, *Proceedings of the 2004 IEEE International Conference on Robotics & Automation,* New Orleans, April 2004.

[SaMN08] J. Z. Sasiadek, A. Monjazeb, D. Necsulescu, Navigation of an autonomous mobile robot using EKF-SLAM and FastSLAM. *16th Mediterranean Conference on Control and Automation* Ajaccio, France 2008.

[Seeg09] Seegrid Corporation, http://www.seegrid.com 2009

[ShBa04] D. Shaw, N. Barnes, Regular Polygon Detection as an Interest Point Operator for SLAM, *Australasian Conference on Robotics and Automation* 2004.

[SKYT02] Y. Sumi, Y. Kawai, T. Yoshimi, F. Tomita, 3D Object Recognition in Cluttered Environments by Segment-Based Stereo Vision. *International Journal of Computer Vision 46(1) 5-23* 2002.

[SmSC86] R. Smith, M. Self, P. Cheeseman, Estimating uncertain spatial relationships in robotics. *UAI' 86 Proceedings of the Second Annual Conference on Uncertainty in Artificial Intelligence. University of Pennsylvania,* 1986

[SNJM04] S. Se, H. Ng, P. Jasiobedzki, T. Moyung, Vision Based Modeling and Localization for Planetary Exploration Rovers. *Proceedings of International Astronautical Congress, IAC 2004,* Vancouver, Canada, 2004.

[SSSD06] P. Sala, R. Sim, A. Shokoufandeh, S. Dickinson, Landmark Selection from Vision-Based Navigation, *IEEE Transactions on Robotics,* VOL. 22, NO. 2, April 2006

[RuLe01] S Rusinkiewicz, Marc Levoy, Efficient Variants of the ICP Algorithm. *Third International Conference on 3D Digital Imaging and Modeling (3DIM).* 2001

[TCLH04] S. Thrun, C.Martin, Y.Liu, D. Hahnel, R. Emery-Montemerlo, D. Chakrabarti, and W.

Burgand. A real-time expectation maximization algorithm for acquiring multi-planar maps of indoor environments with mobile robots. *IEEE Transactions on Robotics*, 20(3):433-443, 2004.

[TKGW02] S. Thrun, D. Koller, Z. Ghahramani, H. Durrant-Whyte, and A.Y. Ng. Simultaneous mapping and localization with sparse extended information filters: theory and initial results. *Technical Report CMU-CS-02-112,* Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, 2002.

[ThBF06] S. Thrun, W. Burgard, D. Fox. Probabilistic Robotics. *MIT Press*, Cambridge, MA, 2005.

[TYDG04] S. Thrun, Y. Liu, D. Killer, A. Ng, Z. Ghahramani, H. Durrant-Whyte. Simultaneous localization and mapping with spares extended information filters. *International Journal of Robotics Research,* 23(7-8), 2004.

[Unre09] Unreal Technology *http://www.unrealtechnology.com/licensing.php?ref=faq,* Epic Games Retrieved November 2009.

[WeBi01] G. Welch, G. Bishop, An Introduction to the Kalman Filter, SIGGRAPH 2001.

[Weis97] Yair Weiss. Motion Segmentation using EM – a short tutorial. http://www.cs.huji.ac.il/~yweiss/tutorials.html, 1997

[WeSi05] J. Weingarten, R. Siegwart, EKF-based 3D SLAM for Structured Environment Reconstruction *in Proceedings of the IEEE/RSH International Conference on Intelligent Robots and Systems (IROS),* Edmonton, August 2005

[Wiki09a] Binary Space Partitions http://en.wikipedia.org/wiki/Binary_space_partitioning, Retrieved November, 2009.

[Wiki09b] Constructive solid geometry *http://en.wikipedia.org/wiki/Constructive_solid_geometry*, Retrieved November, 2009.

[Wiki09c] Differential signaling http://en.wikipedia.org/wiki/Differential_signaling Retrieved December, 2009

[Wiki09d] Traveling salesman problem *http://en.wikipedia.org/wiki/Traveling_salesman_problem*, Retrieved November, 2009

[Wiki11a] quadtree http://en.wikipedia.org/wiki/Quadtree, Retrieved January, 2011.

[Wiki11b] octree http://en.wikipedia.org/wiki/Octree, Retrieved January, 2011.

[Wiki11c] kd-tree http://en.wikipedia.org/wiki/Kd-tree, Retrieved January, 2011.

[ZiSN03] T. Zinber, J. Schmidt, H. Niemann. A Refined ICP Algorithm for Robust 3-D Correspondence Estimation. *Proceedings of the IEEE International Conference on Image Processing,* v 2 p 695-698, Barcelona, Spain, September 2003.