Spring 2011

BME
FIT
RMA

REAL-TIME SMOOTH REALISTIC PATHS & NAVIGATION

An undergraduate thesis submitted to

Budapest University of Technology and Economics

in partial fulfillment of the requirements
for the degree of

Bachelor Degree
Computer Engineering

Aaron Zampaglione | azampagl2007@my.fit.edu
Kristel Verbiest | kristel.verbiest@rma.ac.be

Abstract

"Real-time Smooth Realistic Paths & Navigation"

Author: Aaron Zampaglione

Advisor: Kristel Verbiest

Mobile robotics is a popular subfield of robotics. The main objective for any mobile robot is to autonomously navigate a partially known environment until a goal is reached. Planning algorithms such as D* have been designed to find the optimal paths in a partially known environment. However, these paths typically lack realistic movement. To make movement appear more realistic, paths need to be smoothed and curved turns need to occur. This thesis presents methods in which smoothed paths and curved turns can be achieved.

Zigzags in paths cause unwanted "staircase" effects and unnecessary abrupt turns during traversal. To eliminate the zigzags, a naïve incremental method was chosen. Points were selectively eliminated until the original path produced by the planning algorithm was smoothed.

Curved turns make movement more aesthetically pleasing and require less power from the robot's drive shaft thereby reducing power consumption [1]. The first attempt at curved turns was to a use B-Spline curve. B-Spline curves are a polynomial curve with a given amount of control points. Ultimately, B-Splines were the wrong choice because it was extremely difficult to make the robot follow the spline precisely. Eventually Hermite curves were discovered. Hermite curves are a third-degree spline defined by two (control) points. They are very powerful and inexpensive to compute [2]. Using the points generated by the planning algorithm as control points, Hermite curves were generated along the path, as necessary.

After the main thesis objectives were accomplished independently, the presented global planning algorithm was replaced with D* Lite. D* Lite was chosen because integration of the components was much easier and it is one of the most efficient partially known global planning algorithms available to date.

Smoothed paths and curved turns were accomplished individually. All components, with the exception of curved turns, were properly integrated in the time allotted for this project.

Résumé

"En Temps Réel Smooth Chemins Réaliste et Navigation"

Auteur: Aaron Zampaglione

Conseiller: Kristel Verbiest

La robotique mobile est un domaine populaire de la robotique. L'objectif principal pour un robot mobile autonome est de naviguer dans un environnement partiellement connu jusqu'à ce qu'un but soit atteint. Des algorithmes planifiés tels que D* ont été conçus pour trouver les chemins optimaux dans un environnement partiellement connu. Toutefois, ces chemins manquent généralement d'un mouvement réaliste. Pour rendre les mouvements plus réalistes, les chemins doivent être lissées et les tournures courbées doivent se produire. Cette thèse présente des méthodes pour atteindre des trajectoires lissées et tournures courbées.

Des zigzags dans les chemins causent des inutiles effets de la façon "en escalier" indésirables et des virages brusques lors de la traversée. Pour éliminer les zigzags, une méthode naïve supplémentaire a été choisie. Des points ont été sélectivement éliminés jusqu'à ce que la voie originale produite par l'algorithme planifié a été lissée.

Les tournures courbées rendent les déplacements plus esthétiques et nécessitent moins de puissance de l'arbre d'entraînement du robot, réduisant ainsi la consommation d'énergie [1]. La première tentative pour produire des tournures courbées a été l'utilisation d'une courbe B-Spline. Les courbes B-Spline sont un type de courbe polynomiale avec une quantité donnée de points de contrôle. Finalement, B-Splines sont le mauvais choix, car il était extrêmement difficile de faire suivre précisément les splines au robot. En fin de compte, les courbes de Hermite ont été découverts. Ces courbes sont un type d'une spline Hermite de troisième degré définie par deux points (de contrôle). Ils sont très puissantes et peu coûteuses à calculer [2]. En utilisant les points générés par l'algorithme planifié comme points de contrôle, les courbes de Hermite ont été générées en bordant du chemin, ce qui est le cas échéant.

Après que les objectifs de la thèse principale ont été atteints de manière indépendante, l'algorithme de planification présenté mondiale a été remplacé par D * Lite. D * Lite a été choisi parce que l'intégration des composants a été beaucoup plus facile et il est l'un des plus efficaces globales algorithmes partiellement connus disponibles à ce jour.

Les chemins lissées et tournures courbées ont été réalisées individuellement. Tous les composants, à l'exception de tours incurvées, ont été bien intégrées dans le temps alloué pour ce projet.

Table of Contents

List of Figures

List of Tables

List of Equations

Acknowledgment

I would like to express my appreciation to my thesis advisor, Kristel Verbiest, for her support and time. Without her expertise and assistance, completion of many parts of this work would not have been feasible.

In addition, I would like to thank Dr. Balázs Csébfalvi of Budapest University of Technology and Economics for his course “Computer Graphics and Image Processing”. Many concepts learned in class, such as B-Splines and Incremental Ray Tracing, were used in this project.

I would also like to thank Dr. Bálint Kiss and Dr. Charles Bostater. Without their facilitation, the chance to undertake this project would not have been possible.

Finally, my deepest gratitude goes to my parents, Dennis and Teresa Zampaglione, for their love and encouragement. Without them, my continuous success would have not been achievable.

Chapter 1: Introduction

1.1 Motivation

Robots have infiltrated our lives from pop culture, to movies, to video games and are even aiding our everyday activities. What was once only seen on the television screen or as a hobby for super-nerds is now seen as an elegant and “cool” concentration for the brightest of the bright. Observing the robots we know today is quite astonishing in its own right, but designing the intelligence for such fascinating machines is an experience like no other.

Imagine trying to give a box made of plastic, metal, and a few wheels, a brain. As one aspect of the “brain” is solved, new challenges arise unexpectedly. It is for this very reason the field of robotics is so inspiring. As the field of robotics advances, even more unforeseen challenges will arise, awaiting anyone who is willing to accept them.

1.2 Background

One major category of robotics is mobile robots. A mobile robot is a machine that is able to move from a start position to a goal position, execute tasks, and react to input either from the environment or an external controller. Common types of mobile robots are unmanned aerial vehicles (UAV), underwater robots, and unmanned ground vehicles (UGV). Applications can range from military operations, such as the RQ-1 / MQ-1 Predator UAV [3], to ordinary consumer products, such as the Roomba floor cleaning robot [4].

A major challenge for mobile robots and UGVs in particular, is autonomous navigation. Typically, the robot’s environment is partially known, at best. The robot must discover obstacles and generate new paths "on the fly" [5]. To cope with this dilemma, autonomous navigation is broken into four simultaneous tasks in which “action and planning” are “interleaved” [5]:

- Perception: viewing the world and interpreting what it sees [6].
- Localization: recording the robot’s position [6].
- Local navigation: avoiding obstacles in a close proximity to the robot [6].
- Global path planning: Finding the most efficient, safest, and fastest way to go from start to goal [6].

Ultimately, the objective is to move from start to goal while avoiding all obstacles and minimizing the cost of path traversal [7].

For this project, all four tasks were previously accomplished. Perception and local navigation were achieved using the robot's onboard sonar and laser. Localization was achieved using odometry (change in sensors over time to estimate position). Finally, optimal paths (global path planning) were generated using Stenz's original D* (D-Star) algorithm [7].

1.3 Problem Statement

Although optimal paths had been generated, the paths typically lacked realistic appearance. The immediate path generated by the planning algorithm was consistent with a stereotypical robot: sharp and fast maneuvers with no elegance, which is typically undesired. To make movement appear more realistic, the path had to be smoothed (Figure 1) and curved turns needed to occur (Figure 2).

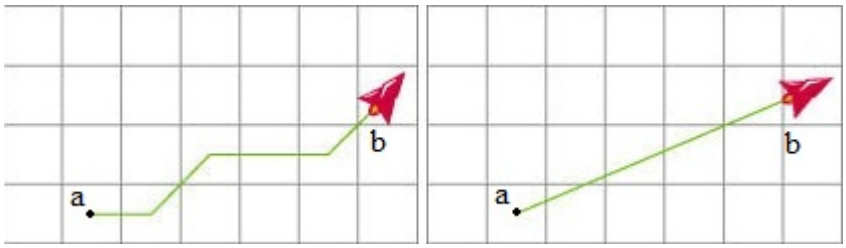

Figure 1: Smoothed path [8].

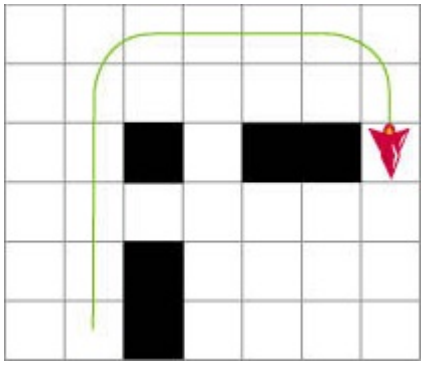

Figure 2: Curved turn [8].

Zigzags in paths were generated due to the local step by step (tile by tile) movement and the direct turns (typically 45°) of the planning algorithm. This effect would have been fine for simple environments, but did not suffice for the real world. As seen in Figure 1, the shortest distance from arbitrary point *a* to arbitrary point *b* should have been taken, regardless of the necessary angle. Fortunately, this was accomplishable by eliminating zigzags, which produced smoother paths.

In practice, smooth turns require less power for the robot's drive system. This reduces operational errors and helps maintain low energy consumption [1]. To achieve curved turns, "a trajectory or path through space" [9] needs to be specified along which a robot would be able to follow precisely.

Not only are both desired for their efficiency, but both make the final route more visually pleasing [10]. In order to achieve realistic movement, both path smoothing and curved turns are required to occur.

1.4 Objective

The objective of this thesis research was two-fold: smoothed paths (zigzag elimination) and curved turns. Both were to be accomplished via post processing, after the path was determined by the designated planning algorithm.

After the proper algorithms for path smoothing and curved turns were chosen, they were implemented. Once implemented, the algorithms were tested in three phases:

1. Test the algorithm with custom maps, simulating different terrains and gradients [6].
2. Test implementation with simulator [6].
3. Test implementation with physical robot [6].

1.5 Outline

The report is structured as follows. The *Approach* section introduces the work process of the project. It includes the software/hardware required, tools used to aid development, and the thought process and research used for accomplishing the objective.

The *Smoothed Path* section explicates how zigzags were eliminated from the original planned path. It introduces the algorithm used and possible caveats.

The *Curved Turns* section introduces the concept of making the robot turn smoothly from point to point. It explains all the algorithms that were implemented, why some of the attempted algorithms failed, and why the final algorithm was chosen.

The *D* Lite* chapter introduces an alternative global planning algorithm that was implemented. It explains the underlying concepts of D* Lite, why it was chosen over D*, how it was used, and what the benefits of using D* Lite were.

The *Integration* section describes how the research came together. It describes how D* Lite was integrated with smoothed paths, curved turns, and the robot's simulator.

Finally, the *Conclusion* section summarizes the thesis as a whole. It explicitly states what was accomplished, introduces limitations of the thesis work, and suggests future tasks for developers that decide to continue this project.

Chapter 2: Approach

2.1 Research

The foundation for any thesis project is always research. Information for accomplishing the objective of this thesis came from a variety of sources: the internet, textbooks received in class [9], and Florida Institute of Technology's "ProQuest" and "360 Search" online libraries [11]. Articles varied from scholarly journals, to blogger posts, to freelance papers. Once the original papers that were presented with the project were analyzed, research bloomed and continued throughout the entire project. Although not considered scholarly, the most useful information was acquired from freelancers trying to aid their fellow developers in the complicated world of robotics.

2.2 Environment

2.2.1 Work Repository

To collaborate, backup, and allow others to monitor progress, a version control system stored in a remote location was used. The version control system chosen was *git*: a fast, efficient, and distributed version control system that is ideal for collaborative development [12]. All data was stored on *GitHub*, a social coding website [13], under a private repository. The following credentials will enable read-only access to the repository:

Website	https://github.com/
Username	azampagl-pal
Password	Kn0ckKn0ck46

When logged in (Figure 3), the user can navigate to source files, download versions of the code and documents, view all branches (master, develop, feature/smooth-path-navigation, etc.) and view "commit" history.

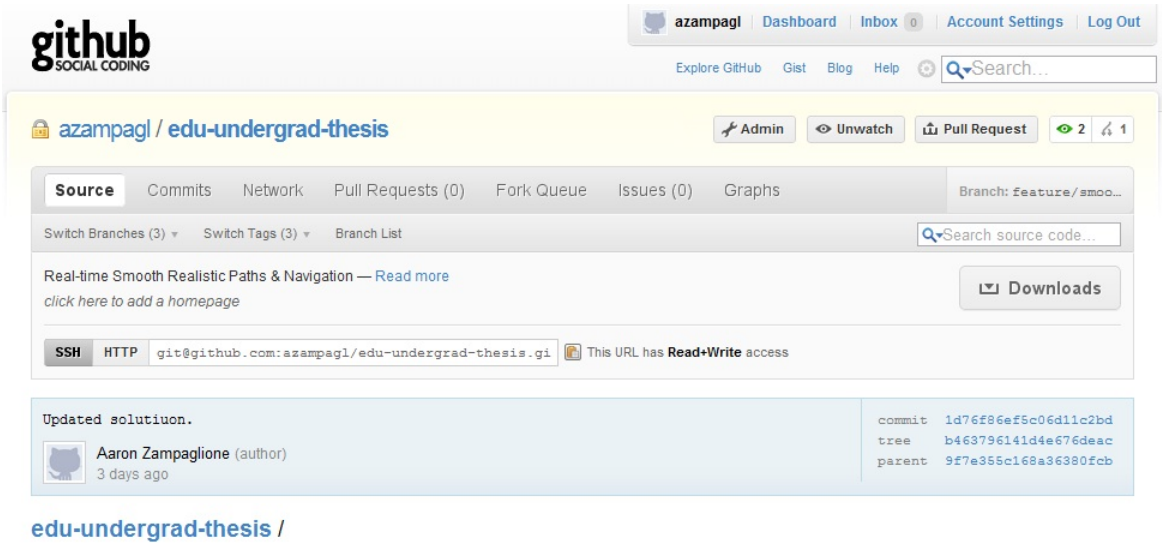

Figure 3: Project homepage on GitHub [13].

At anytime, anyone could view every incremental change (commit) to the project. Each commit contains every code line change and is accompanied by a brief summary of the commit (Figure 4). A major benefit of using commit history was that it allowed management to track progress of the project without weekly summaries, saving headaches and unnecessary scheduled meetings. Management could easily view progress at any time that was convenient. Commit history also allows future developers to easily see previous development history so that progress can continue as fast as possible.

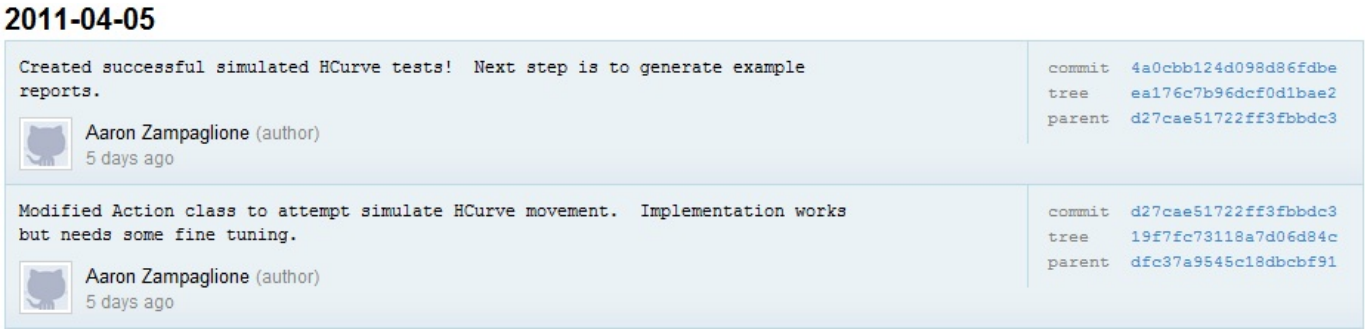

Figure 4: Example commit history [13].

Storing the data remotely on GitHub allowed the project to be exceptionally accessible. Current (and future) developers could collaborate simply by "pulling" (downloading) from GitHub, giving them access to the entire code base almost instantaneously. Once "pulled" (downloaded), the developer could continue progress and "push" (upload) their changes to GitHub's remote servers. Storing the data remotely also inadvertently acted as a backup in case of local data loss.

2.2.2 Development Environment

All development occurred within a virtual machine using VirtualBox. VirtualBox is an open source "x86 and AMD64/Intel64 virtualization product" for enterprise and home users [14]. The benefits of using a dedicated virtual machine include:

- **Dedicated resources** – the machine was allocated 1 gigabyte of RAM and 20 gigabytes of hard drive disk space.
- **Work separation** – all source code files and executables were stored on the virtual machine so they could not mix with personal files on the host machine.
- **Backups via Snapshots** (Figure 5) – at any time, a backup could be created by generating a snapshot of the current machine state.

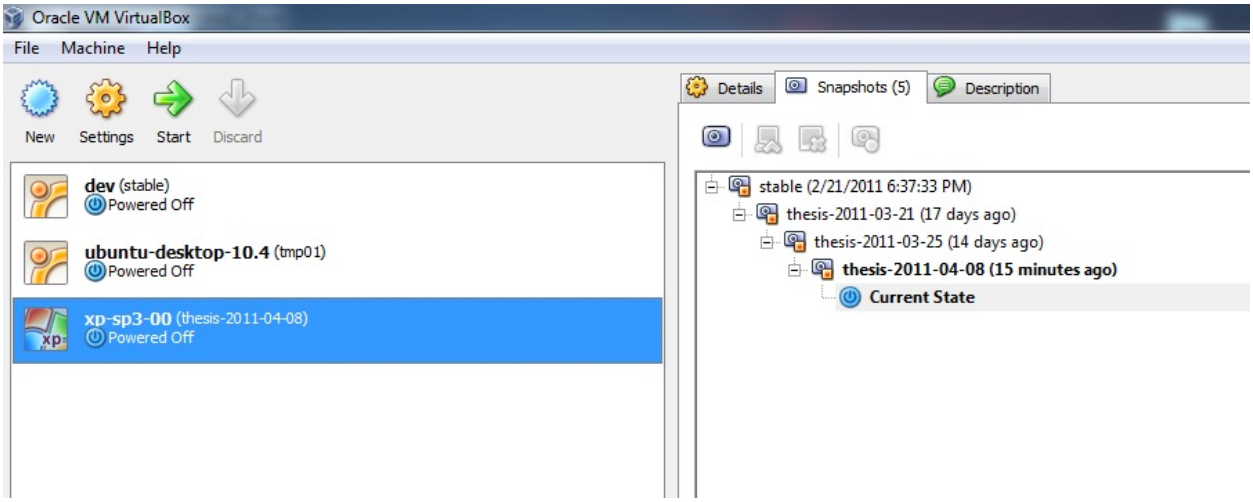

Figure 5: VirtualBox Windows XP environment with snapshots.

If requested, an exported virtual machine in Open Virtual Format (OVF) can be delivered to future developers. It can be imported by any virtualization product that supports OVF (VirtualBox, VMWare, etc.). Future developers will be able to continue development using the same environment, in exactly the same state as it was at the last time of use.

2.2.3 Software Requirements

The following software is required to build and execute the code included with this thesis.

- **Windows XP** – operating system selected for compatibility with the Aria software.

- **Visual Studio 2008** – integrated development environment (IDE) tool used to build the C++ code.
- **Visual Studio 2008 SP1** – service pack that includes additional required libraries such as unordered maps.
- **OpenCV 2.1** – C++ graphics library used to visualize robot maps, paths, and navigation.
- **Aria 2.7.2** – "The source code and libraries required to create new ARIA programs and some demo applications" [15].
- **MobileSIM 0.5.0** – "The stage-based simulator from ActivMedia" [15] that simulates the actions of the real robot.
- **Mapper3 2.2.5** – "A program for generating environments for the robot to navigate" [15].

2.2.4 Hardware Requirements

The hardware necessary for this project was quite limited: the robot, a Pioneer P3-DX and/or Pioneer 3-AT (Figure 6). The only other optional piece of hardware necessary was a wireless router to remotely connect to the robot's built-in operating system.

Figure 6: Thesis Robot a.k.a Little Red Riding Hood.

Chapter 3: Smoothed Paths

3.1 Zigzag Dilemma

Figure 7 is a quintessential example of a path containing zigzags. Although it was the result an A* search, the concept still applies to D* search. When D* produced a planned path, the path was never re-analyzed to see if there were more efficient intermediate paths from point to point.

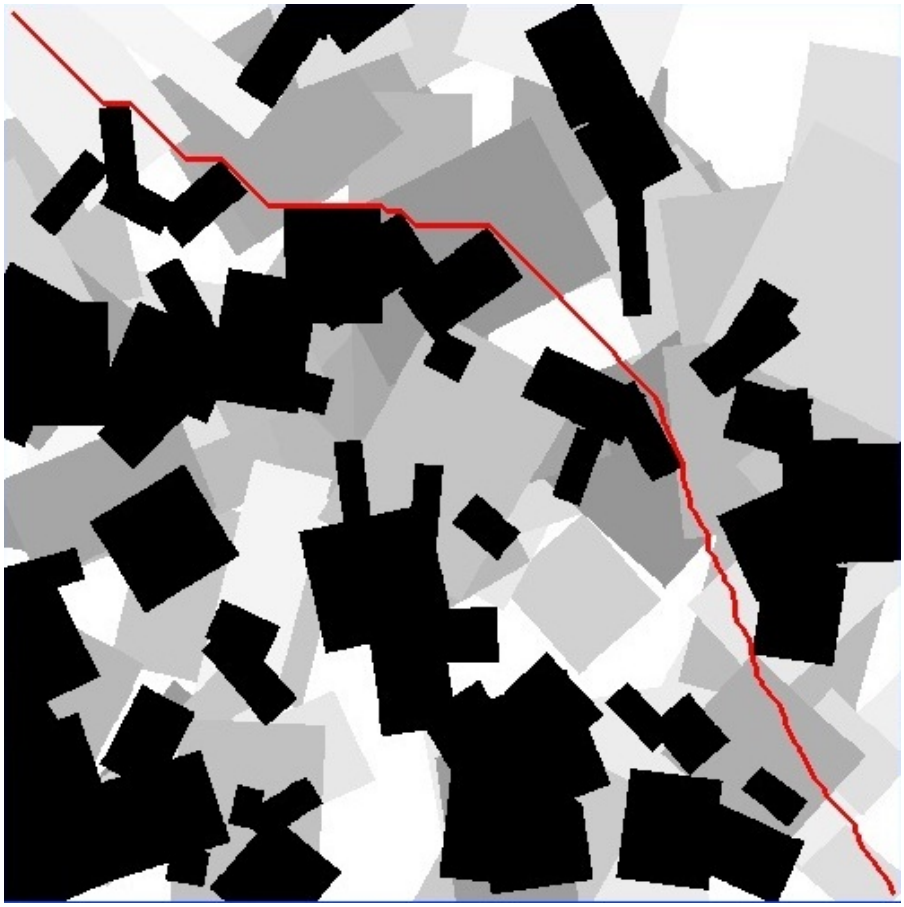

Figure 7: A* with zigzags.

3.2 Solution

The solution chosen was a simple point sampling algorithm. It would extract points, in order, from a queue and use a *walkable* [8] function to determine if a path between two points was traversable. If they were, any intermediate points between those two chosen points would be eliminated.

```
checkPoint = starting point of path
currentPoint = next point in path
while (currentPoint->next != NULL)
        if Walkable(checkPoint, currentPoint->next)
                temp = currentPoint
                currentPoint = currentPoint->next
                delete temp from the path
        else
                checkPoint = currentPoint
                currentPoint = currentPoint->next
```

Figure 8: Zigzag elimination algorithm pseudo code [8].

The *walkable* method was an integer based grid ray tracing algorithm [16]. An imaginary straight line was traced between two given points, intersecting tiles along the path (highlighted boxes in Figure 9). If any of the traversed tiles were considered forbidden, then the path between the two points was considered *unwalkable*.

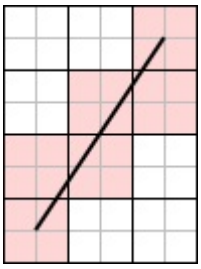

Figure 9: Example of grid ray tracing [17].

3.3 Results

The solution was implemented and then was tested against the results of an A* search, as seen in Figure 10. Testing against the results of an A* search was done because the points were static, unlike a D* search whose path points are updated every time an obstacle is discovered. A static point selection was not only easier to test against, but easier to graphically illustrate for visual verification. In theory, if the implementation tested successfully for the results of the A* search, it would test successfully when being applied to another search algorithm's results, such as D*.

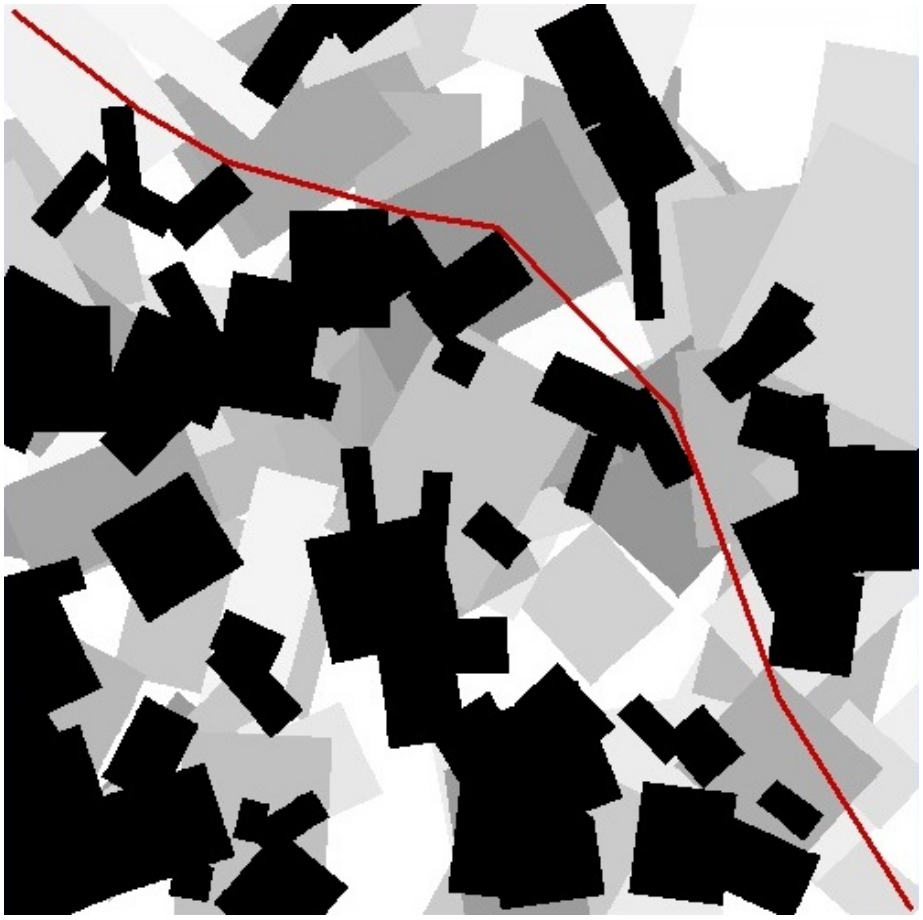

Figure 10: A* with zigzags eliminated.

Chapter 4: Curved Turns

4.1 B-Spline Curves

4.1.1 B-Spline Curve Introduction

Generating curves given only a few discrete points was a daunting task. Fortunately, it has already been solved in the field of computer graphics. The idea was to generate B-Spline curves. B-Spline curves are "a generalization of the Bézier curve" [18] with a predetermined polynomial degree and amount of control points.

The only parameter necessary to generate a B-Spline curve was the set of control points. The set of control points used were the points the D* algorithm produced as the robot progressed. The polynomial degree of the curve was always set to *n-1* (the number of control points minus one) so that the curve would be as close to the control points as possible.

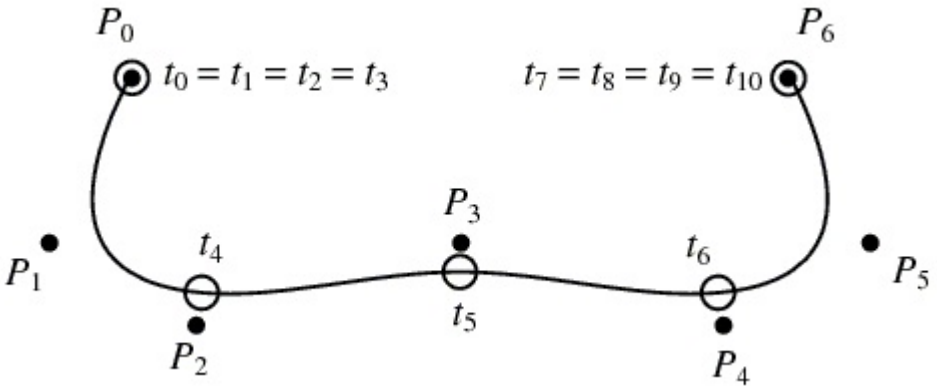

Figure 11: B-Spline with control points [18].

4.1.2 Full Path B-Spline

The first approach using B-Spline curves was generating one large B-Spline for the entire projected path. After a few hours of working with the idea, it was determined that this approach was futile. The first problem was that the B-Spline needed to be regenerated every time the map was updated, which was computationally expensive. Second, the B-Spline curve never really touched the control points due to its inherent nature to remain within the convex hull, as seen in Figure 12. This would have caused issues if the robot needed to make sharp turns, since it doesn't follow the control points precisely.

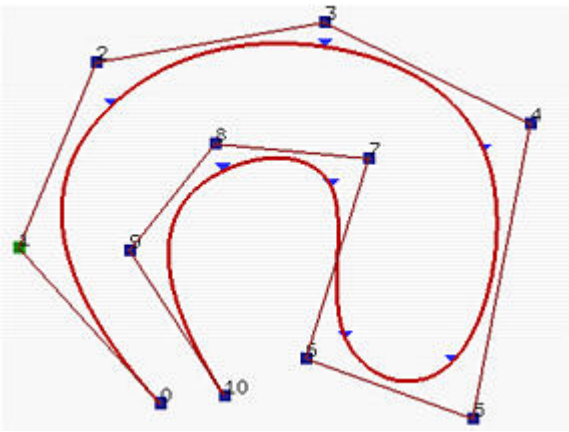

Figure 12: B-Spline convex hull [19].

4.1.3 Fragmented B-Spline

The next approach was to generate B-Splines only between points where the robot needed to turn. Figure 13 is a graphical representation of the results of that approach. Once intermediate points were generated between any two control points, the robot would follow from point to point along the curve. The hope was that, given enough intermediate points, the movement of the robot would appear smooth. Unfortunately, this did not occur. No matter how many points were generated the attempted solution did not work. If there were too few intermediate points, the robot had very jagged movement because the robot moved from point to point directly without any smoothness. When there were to too many intermediate points, the robot would miss some points completely due to its velocity, which then caused the robot to immediately cease movement because it was unaware of which point to progress to next.

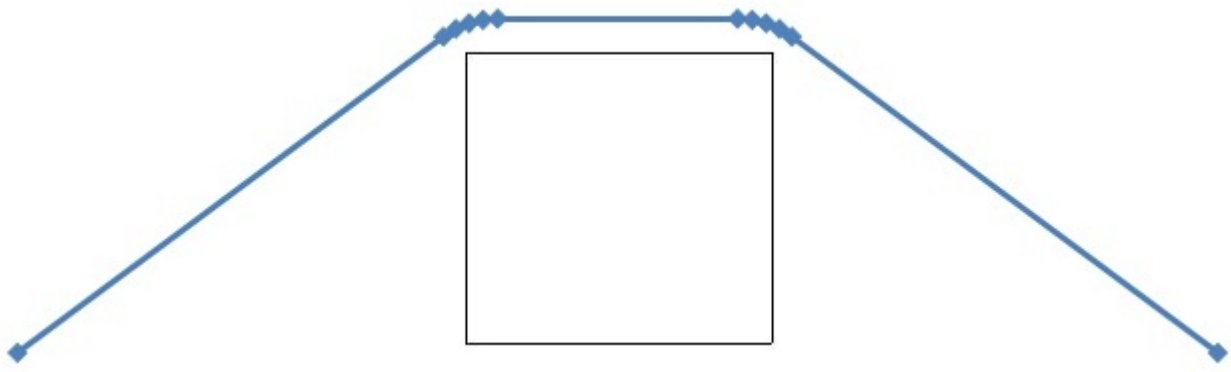

Figure 13: Projected fragmented B-Spline trajectory.

4.1.4 B-Spline Conclusion

Ultimately, the B-Spline approach was abandoned. Although B-Spline curves were generated and could been visualized it did not work for the following reasons:

- Degree of the curve was too low (convex hull dilemma), so for tight turns the curve generated would force the robot to run into a forbidden area.
- Every time D* updated, new B-Spline curves would need to be generated, which is computationally expensive.
- No effective way to make the robot follow intermediate points along the B-Spline curve.

4.2 Hermite Curves

4.2.1 Hermite Curve Introduction

All of the complications with the BSpline curves brought the realization that an even simpler approach was necessary. From the most abstract level, how does one make a curve? Ultimately, a curve is nothing more than a finite sequence of line segments with increasing (or decreasing) angles. How can one tell the rate of change of the angles? Simply take the derivative and obtain rotational velocity. Research then began on finding an algorithm in which the rotational velocity could be computed at any instance of time. Finally a solution was found: Hermite curves.

A Hermite curve is third-degree spline defined by two (control) points and two (control) tangents, as illustrated in Figure 14. They are commonly used in computer graphics to smoothly interpolate between two points. Hermite curves are a popular choice because they are very powerful when properly applied and are easy to calculate [2].

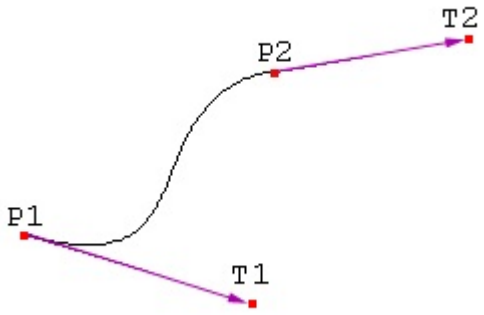

Figure 14: Hermite curve defined by two starting points (P1 and P2) and two tangents (T1 and T2) [2].

The technique used in this work was a special case and an overly simplified application of Hermite curves [1]. A curve could be generated, given an initial and final point's coordinates, speed, and angle. Using the given variables, an estimated time for traversing the curve was calculated. With the estimated time of traversal, a multitude of additional variables were produced based on an instance of time along the curve. This included the most important variable: rotational velocity. The Hermite curve allowed for a very flexible approach, thus solving the problem of sharp turns, mild turns, and even near straight paths.

4.2.2 Hermite Curve Implementation

The first determination was to see if a curve was even necessary. If the angle between a point (initially, the robots current heading) and a destination point was below a certain threshold, the robots built in methods for navigating to a point were used. If the difference of angles was larger than the threshold, a Hermite curve was used. As the robot executed its cycles, the translational velocity and rotational velocity would be calculated and sent to the robot using its built-in methods *setVelocity()* and *setRotationalVelocity()*, respectively. A simplified pseudo code version of the approach is as follows:

```
startPoint = starting point
startAngle = current heading of the robot
startSpeed = current speed of the robot

endPoint = end point
endAngle = desired heading when end point is reached
endSpeed = desired speed when end point is reached

useCurve = false

if (angleDifference(startAngle,endAngle) > threshold)
        useCurve = true
        initialTime = currentTime()
        curve = HermiteCurve(startPoint, startAngle, startSpeed, endPoint, endAngle, endSpeed)

while ( ! close(startPoint, endPoint))
        if useCurve
                time = currentTime() - initialTime
                setRotationalVelocity(curve.getRotationalVelocity(time))
                setVelocity(curve.getSpeed(time))
        else
                setDeltaHeading(angleDifference(currentHeading(), endAngle))
                setVelocity(endSpeed)
```

Figure 15: Hermite curve algorithm pseudo code.

4.2.3 Hermite Curve Applied Equations

The following section goes in depth on the equations applied to generate the Hermite curves.

Table 1: Specified (Given) variables for Hermite curve [1].

$(x_0, y_0), (x_1, y_1)$	Initial and final point coordinates.
θ_0, θ_1	Initial and desired final angles.
s_0, s_1	Initial and desired final speeds.

Table 1 specifies the required parameters to make a Hermite curve. The coordinates (points) necessary were provided by whatever point set the planning algorithm generated. The angles and speeds required preprocessing.

To generate the angles, the path point set was completely analyzed. If a curve was isolated between two straight lines on the path as seen in Figure 16, the initial angle and final angle were easy to calculate. The initial angle was the robot's current heading. The final angle was the heading from the final point of the curve, to whatever point was next on the path.

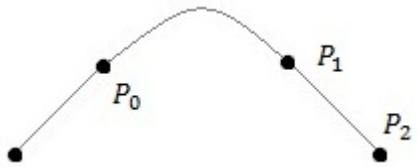

Figure 16: Curve isolated between two straight lines.

$$\theta_0 = robot's\ current\ heading$$

$$\theta_1 = Heading(P_1, P_2)$$

If multiple consecutive curves were necessary, the intermediate point angles needed to be calculated as well as the final angle. Although technically any angle could have been used, half the angle between the prior point and the next point seemed to be most logical, as seen in Figure 17.

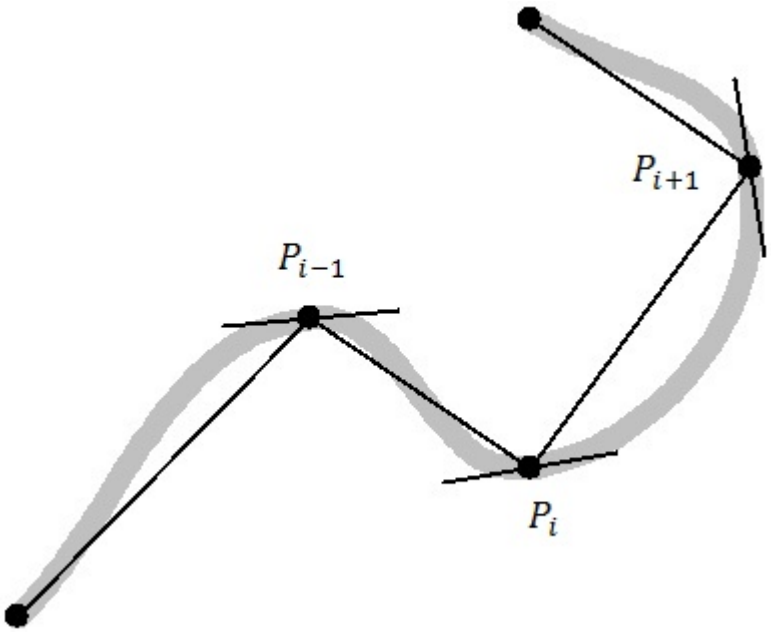

Figure 17: Four Hermite curves with pre-calculated intermediate angles [1].

Computationally, the following formula was applied to a point set:

$$-\pi \leq \theta_i \leq \pi$$

$$\theta_0 = robot's\ current\ heading$$

$$\theta_i = \frac{Heading(P_{i-1}, P_i) + Heading(P_i, P_{i+1})}{2}$$

Equation 1: Intermediate point angle formula.

It is important to note that it was assumed that points existed after the final point of the curve (or curves). If the final point was the goal point, then an arbitrary angle could be chosen because it did not matter, as long as the goal had been reached.

The speed parameters for the Hermite curve varied, depending on the sharpness of the turn and the distance between the initial point and final point of the curve. Sharp turns and curves, where the initial and final points were close, were passed slower speed parameters, with an absolute minimum of 50 millimeters per second. Smoother turns and long initial and final point distances were passed speed parameters that were relative to the speed the robot was moving when it entered the curve. Variations in initial and final speeds during turns were not experimented with further, as they were not a priority.

Once the parameters for the Hermite curve were available, it was necessary to calculate the polynomial coefficients of the cubic equations (Equation 2 and Equation 3) for future calculations.

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

Equation 2: X-coordinate cubic equation [1].

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

Equation 3: Y-coordinate cubic equation [1].

The simplest coefficients to calculate were d_x and d_y, which were the initial x-coordinate and y-coordinate, respectively. This only left a_x, b_x, c_x, a_y, b_y and c_y to be calculated.

To do so, first the velocity vectors of the initial and final points were calculated, using Equation 4 and Equation 5.

$$(v_{x0}, v_{y0}) = s_0 \cdot (\cos\theta_0 , \sin\theta_0)$$

Equation 4: Initial velocity components [1].

$$(v_{x1}, v_{y1}) = s_1 \cdot (\cos\theta_1 , \sin\theta_1)$$

Equation 5: Final velocity components [1].

c_x and c_y are the first derivative constants of Equation 2 and Equation 3. These values were obtained by Equation 4 by calculating the initial velocity vector, v_{x0} and v_{y0}, respectively.

Second, an estimated time for a complete curve traversal was calculated. The time was an average of the arc lengths generated by the initial and final velocity vectors.

Equation 6 was used to find the bearing between the initial and final points based on the initial velocity vector.

$$\vartheta_0 = \cos^{-1} \frac{(\overrightarrow{P_1} - \overrightarrow{P_0}) \cdot \overrightarrow{v_0}}{|\overrightarrow{P_1} - \overrightarrow{P_0}||\overrightarrow{v_0}|}$$

Equation 6: Bearing based on initial velocity [1].

Once the bearing based on the initial velocity vector was determined, the arc length could be calculated using Equation 7.

$$L_0 = \left| \frac{\left|\overrightarrow{P_1} - \overrightarrow{P_0}\right| \vartheta_0}{\sin \vartheta_0} \right|$$

Equation 7: Arc length based on initial velocity [1].

Since the curve might have differentiating initial and final velocity vectors, it was necessary to calculate the arc length based on the final velocity vector as well, using Equation 8.

$$\vartheta_1 = \cos^{-1} \frac{\left(\overrightarrow{P_1} - \overrightarrow{P_0}\right) \cdot \overrightarrow{v_1}}{\left|\overrightarrow{P_1} - \overrightarrow{P_0}\right| \left|\overrightarrow{v_1}\right|}$$

Equation 8: Bearing based on final velocity [1].

Once the bearing based on the final velocity vector was determined, the arc length could be calculated using Equation 9.

$$L_1 = \left| \frac{\left|\overrightarrow{P_1} - \overrightarrow{P_0}\right| \vartheta_1}{\sin \vartheta_1} \right|$$

Equation 9: Arc length based on final velocity [1].

To determine the time, the average lengths previously obtained were divided by the speeds at the initial and final points (Equation 10).

$$\Delta t = \frac{L_0 + L_1}{s_0 + s_1}$$

Equation 10: Estimated time based on initial and final arc lengths and speeds [1].

The estimated time is a major determining factor in the shape of the curve. For any given Hermite curve, the shorter the time, the more direct the route. The curve is sharper closer to the initial and final points. If the time is larger, the curve is not as direct and has a longer curved segment near either end.

Figure 18 is an example of Hermite curves with different estimated times of traversal. Both curves were given identical parameters (initial position, initial speed, initial heading, final position, final speed, final heading). The only value that was modified for both curves was their estimated time of traversal. Instead of calculating the time of traversal, the time of traversal was hard coded for both curves. The blue curve was given a large estimated time of traversal (20 seconds) and the red curve was given a smaller estimated time of traversal (10 seconds).

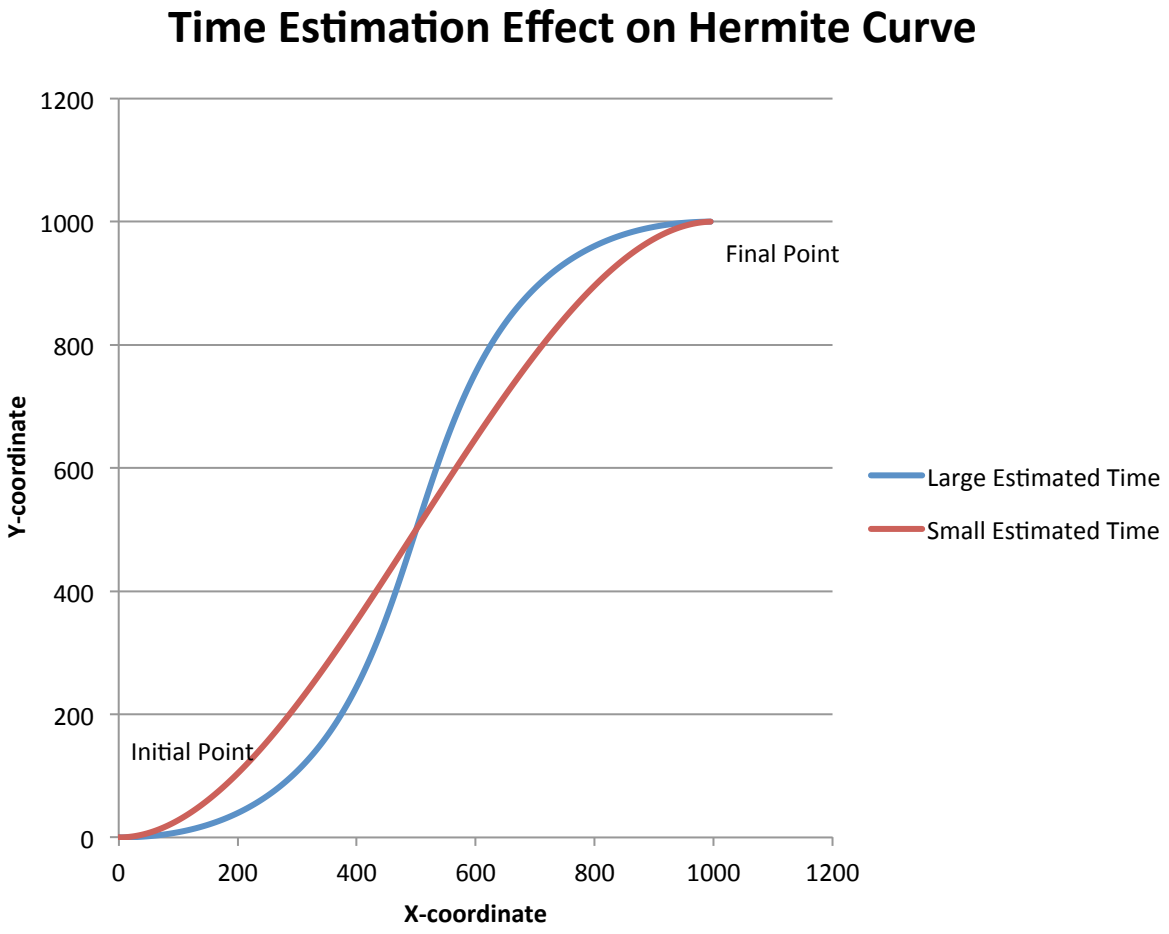

Figure 18: Time estimation effects on Hermite curve.

Upon the availability of the velocity vectors and the estimated time of traversal, the coefficients for the polynomial were calculated using Equation 11, Equation 12, Equation 13, and Equation 14:

$$a_x = 6\frac{(v_{x1} + v_{x0})\Delta t - 2 \cdot (x_1 - x_0)}{\Delta t^3}$$

Equation 11: Ax X-component coefficient [1].

$$b_x = -2\frac{(v_{x1} + 2 \cdot v_{x0})\Delta t - 3 \cdot (x_1 - x_0)}{\Delta t^2}$$

Equation 12: Bx X-component coefficient [1].

$$a_y = 6\frac{(v_{y1} + v_{y0})\Delta t - 2 \cdot (y_1 - y_0)}{\Delta t^3}$$

Equation 13: Ay Y-component coefficient [1].

$$b_y = -2\frac{(v_{y1} + 2 \cdot v_{y0})\Delta t - 3 \cdot (y_1 - y_0)}{\Delta t^2}$$

Equation 14: By Y-component coefficient [1].

The velocity vectors and the polynomial coefficients were only calculated once inside the constructor of the Hermite curve class and stored as member variables. This saved a large amount of computation, as they were constantly referenced by other methods of the class.

Calculating the speed (Equation 15) and the rotational velocity (Equation 16) were the two crucial values needed. After computation, the values were sent to the robot via *setVelocity()* and *setRotationalVelocity()*, as in the pseudo code presented earlier.

$$s(t) = \sqrt{\left(\frac{1}{2}a_x t^2 + b_x t + v_{x0}\right)^2 + \left(\frac{1}{2}a_y t^2 + b_y t + v_{y0}\right)^2}$$

Equation 15: Speed as a function of time [1].

$$\omega(t) = \frac{(a_y t + b_y) \cdot \left(\frac{1}{2}a_x t^2 + b_x t + v_{x0}\right) - (a_x t + b_x) \cdot \left(\frac{1}{2}a_y t^2 + b_y t + v_{y0}\right)}{\left(\frac{1}{2}a_x t^2 + b_x t + v_{x0}\right)^2 + \left(\frac{1}{2}a_y t^2 + b_y t + v_{y0}\right)^2}$$

Equation 16: Rotational velocity as a function of time [1].

Equation 17 was used to obtain the X-coordinate of a Hermite curve given an instance of time. It was used for post processing, presented later in the report.

$$x(t) = \frac{1}{6}a_x t^3 + \frac{1}{2}b_x t^2 + v_{xo}t + x_0$$

Equation 17: X-coordinate as a function of time [1].

The following equation (Equation 18) was used to calculate the Y-coordinate of a curve at any given instance of time. Like the X coordinate equation, it was used for post processing.

$$y(t) = \frac{1}{6}a_y t^3 + \frac{1}{2}b_y t^2 + v_{yo}t + y_0$$

Equation 18: Y-coordinate as a function of time [1].

Although calculating the angle of the curve using Equation 19 wasn't used in production, it was used during testing to see how far off the robot's actual heading was during simulation.

$$\theta(t) = \tan^{-1}\frac{\frac{1}{2}a_y t^2 + b_y t + v_{y0}}{\frac{1}{2}a_x t^2 + b_x t + v_{x0}}$$

Equation 19: Orientation as a function of time [1].

4.2.4 Traversable Hermite Curve Check

The path the robot took following the curve was not always similar to the one generated by the planning algorithm, as seen in Figure 19. Therefore, the path had to be re-checked for obstacles and map boundaries.

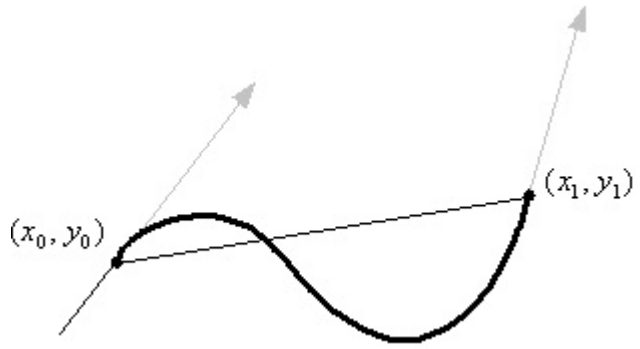

Figure 19: Hermite curve path.

Using Equation 17 and Equation 18, an iterative stepping algorithm was used to predict the path of the curve. By using a small time step, points along the curve could be determined and checked for obstacles or out-of-bound errors. The pseudo code is as follows:

```
curve = Hermite curve
startPoint = initial point
endPoint = final point
currentPoint = startPoint
time = 0
timeIncrement = time step
error = false

while ( ! error && ! close(currentPoint, endPoint))
        time += timeIncrement

        currentPoint = Point(curve.getX(time), curve.getY(time))

        if (currentPoint == Obstacle || ! inMap(currentPoint))
                error = true
```

Figure 20: Curve path check pseudo code.

If an error was found, a more compact curve was generated, as needed. This was accomplished by either moving the initial or final point closer to the other and by optionally decreasing the robot speed. Changing the points or the speed required the calculation of a whole new curve, but fortunately generating Hermite curves is computationally inexpensive.

Figure 21 is non-scaled graphical representation of the strategy. The assumption is that if the curve between P_0 and P_2 is not valid an intermediate point can be used (P_1) to generate a more compact curve.

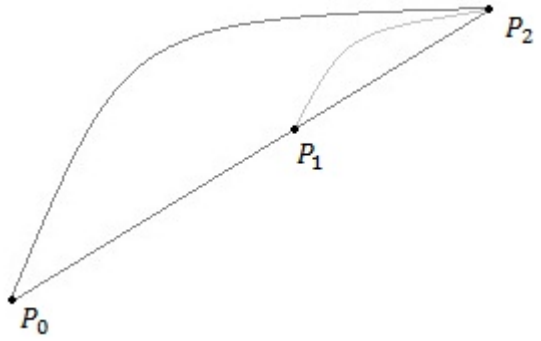

Figure 21: Example of a recalculated curve.

4.3 Results

4.3.1 Introduction

The outcome of the Hermite curve approach is best illustrated by the usage of an example. The results were produced using calculated (exact) results of a Hermite curve, and simulated results based on output from MobileSIM. The parameters from Table 2 were used to construct the example curve.

Table 2: Parameters for example Hermite curve.

Initial Point	(0, 0) (millimeters)
Initial Speed	100 (millimeters per second)
Initial Angle	0°
Final Point	(1000, 1000) (millimeters)
Final Speed	100 (millimeters per second)
Final Angle	0°

Note that for both the calculated and simulator results, the curves were time constrained by the value calculated by Equation 10. For the given scenario, that time calculated to 15.7 seconds.

4.3.2 Calculated Results

The calculated version used a loop to produce results that were written to a file for post analysis. Inside the loop, a counter was incremented by 50 milliseconds up until the estimated time of completion (15.7 seconds). Figure 22 is an illustration of the calculated curve path.

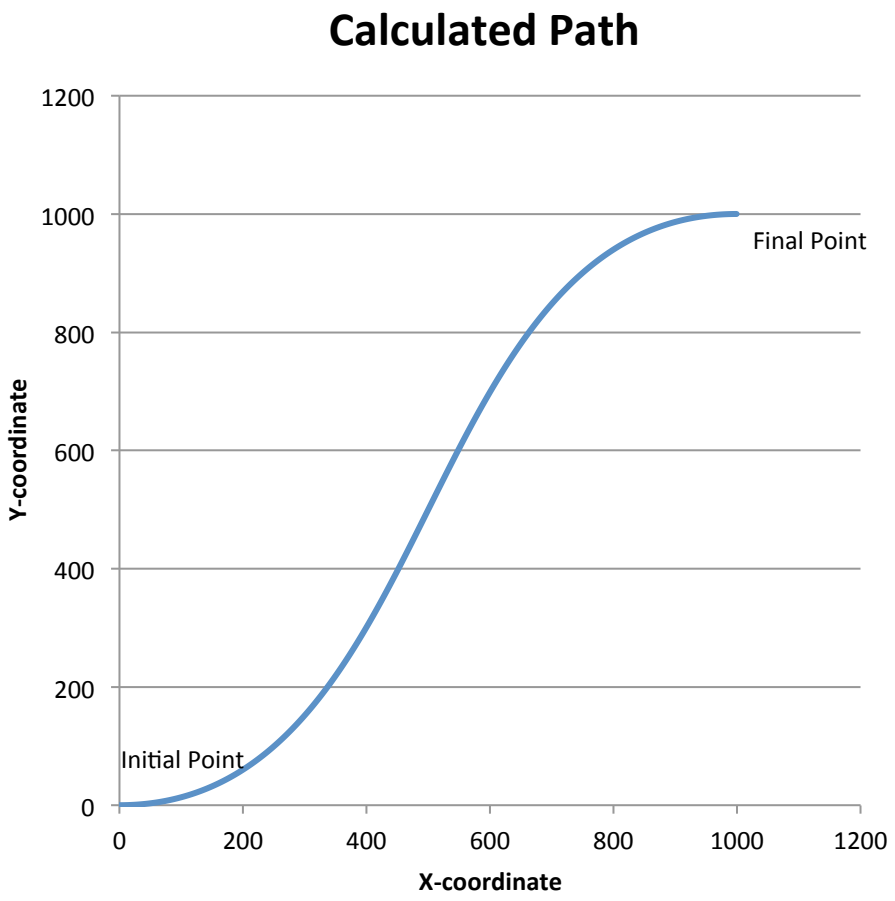

Figure 22: Calculated Hermite curve path.

Although the initial and final speeds were the same, the speed fluctuated as it traversed the path, as seen in Figure 23. This was why it was crucial that the *setVelocity*() method be used simultaneously with the *setRotationalVelocity*(). If the speed was not adjusted properly the robot would not follow the planned trajectory and would likely come in contact with an obstacle or map boundary.

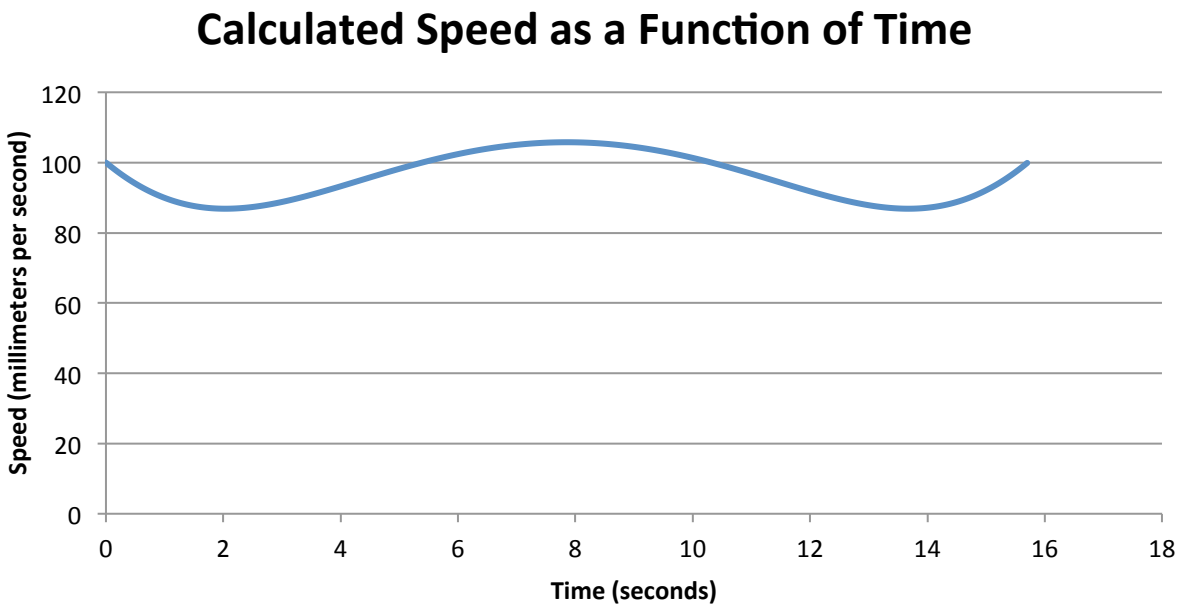

Figure 23: Calculated Hermite curve speed as a function of time.

Figure 24 is a graph of the rotational velocity for the path. As this figure illustrates, the first few seconds and the last few seconds have the highest absolute rotational velocity. This explains why in Figure 22, the "curvature" is greatest at these time instances. The segment towards the middle of the curve (coordinates (400, 300) to (700, 850)) is mostly straight. This clearly correlates to the near 0 degrees per second value of the rotational velocity during the same time frame.

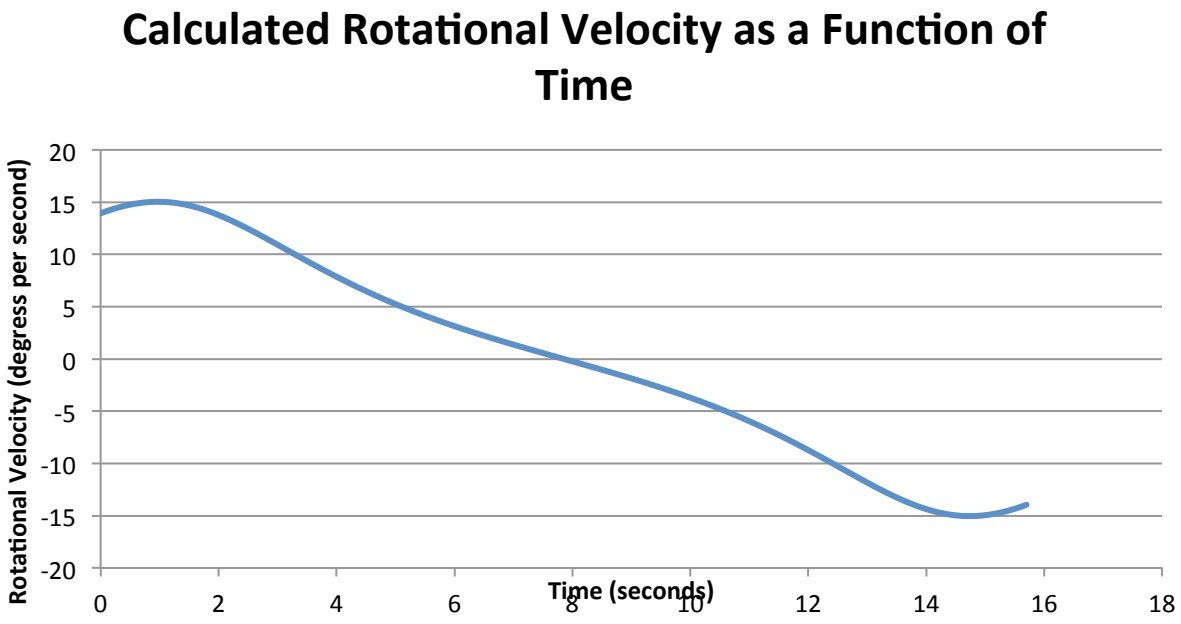

Figure 24: Calculated Hermite curve rotational velocity as a function of time.

Figure 25, as to be expected, started and ended with the same values as the initial and final angles, respectively. From a mathematical stand point, it is obvious that Figure 25 is the integration of the rotational velocity in Figure 24.

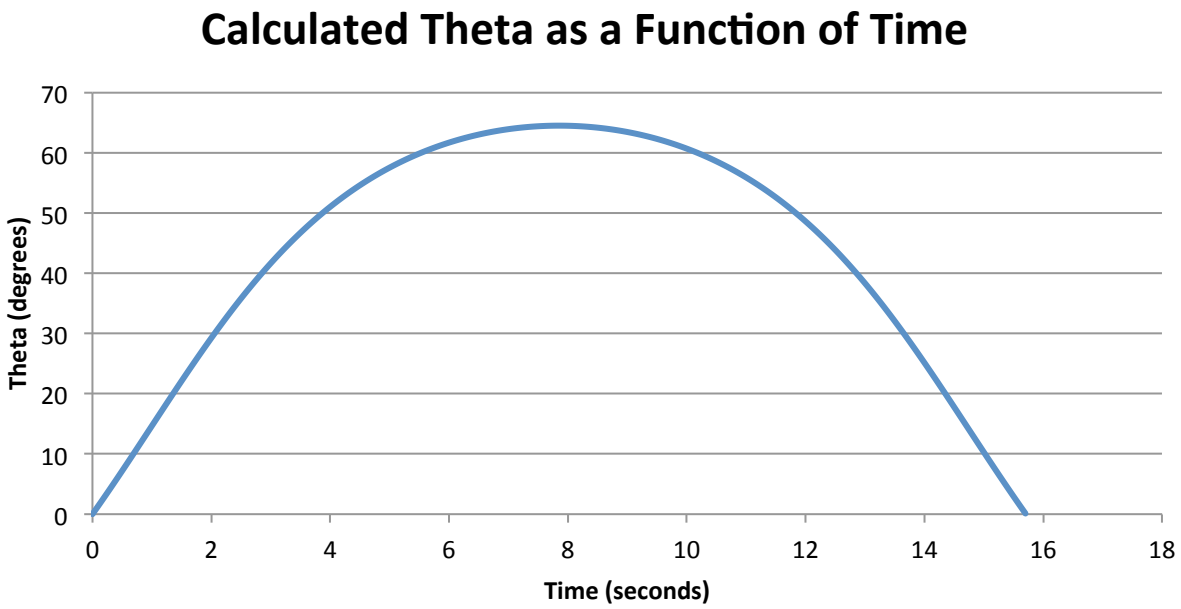

Figure 25: Calculated Hermite curve theta as a function of time.

4.3.3 Simulated Results

The simulated version took a similar approach to that of the calculated. Instead of a loop, the robot's action cycles were used. The cycle frequency was set to 1 cycle every 50 milliseconds. Inside the action was a stored instance of the Hermite curve object and the initial time. Every time a cycle executed, it determined the elapsed time and used it to obtain the necessary speed and rotational velocity at that instance. The results were obtained directly from the robots sensors instead of the calculated values from the curve and stored in a file for post analysis. Figure 26 illustrates the simulated curve path.

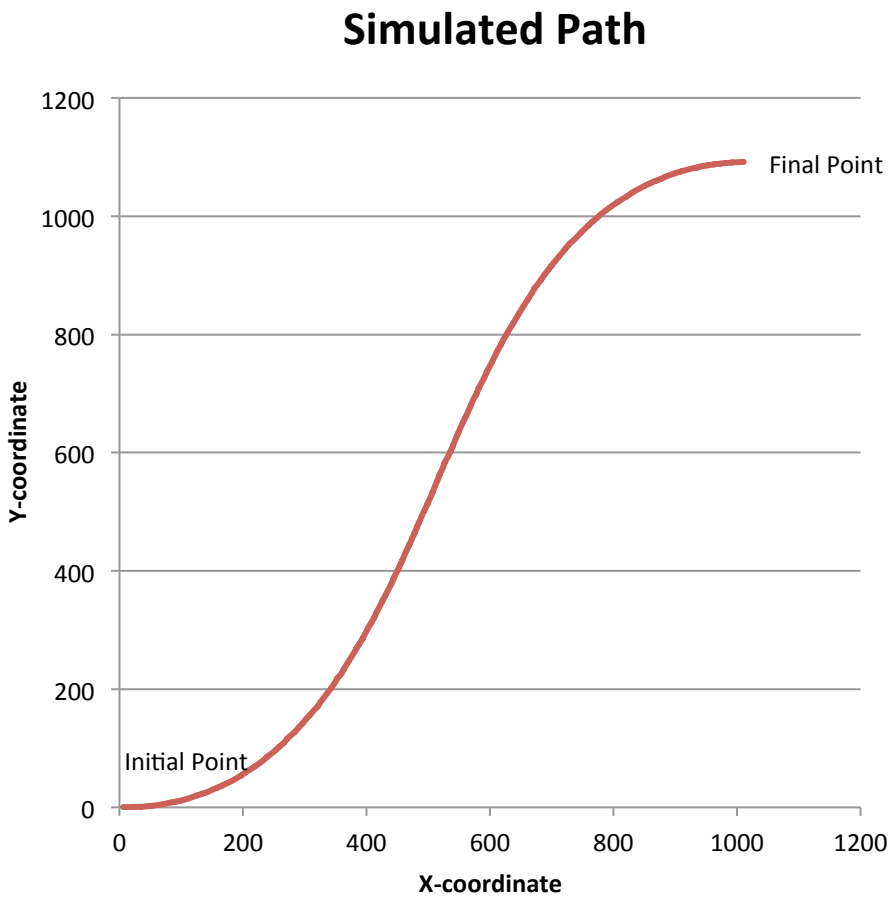

Figure 26: Simulated Hermite curve path.

The simulated speed in Figure 27 is very similar to that of the calculated speed in Figure 23. However, it is not as smooth due to the level of precision. In the simulator, values were sent at the millimeter level (3 decimal points) whereas the calculated version used the maximum amount of decimal points a floating point number would allow on the operating system.

Fortunately, in production, this had very little effect on the outcome of traversed curve because the millimeter precision was high enough.

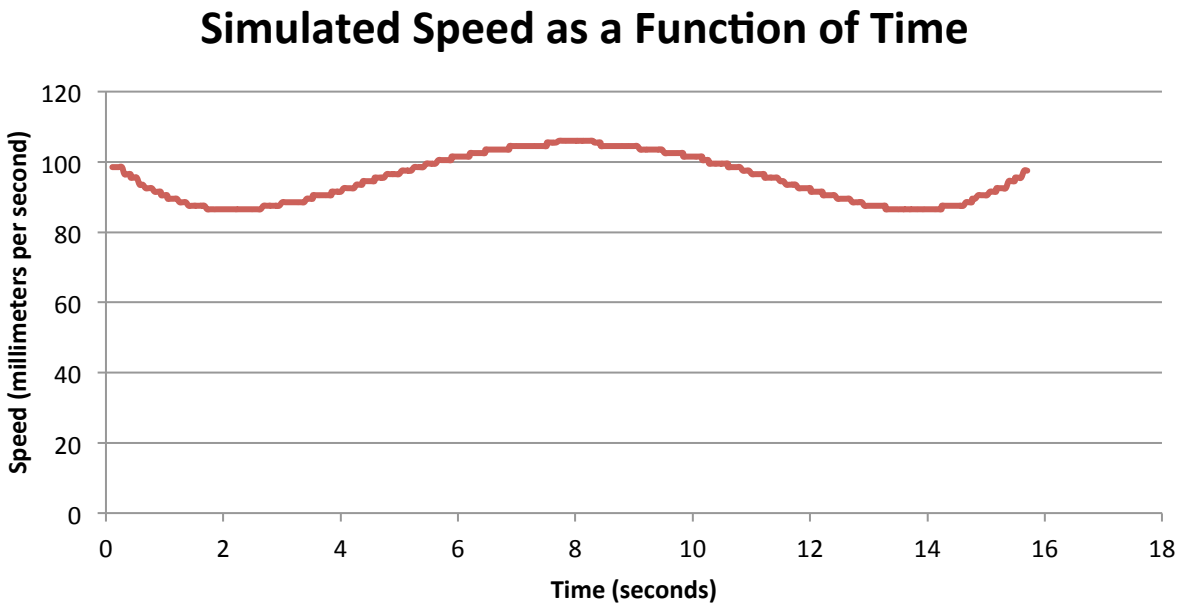

Figure 27: Simulated Hermite curve speed as a function of time.

Figure 28 is a graph of the rotational velocity results obtained from the robot sensors. Although it followed a similar pattern to the calculated version (Figure 24), an extreme "staircase" effect is noticeable. As was the problem with the speed, the level of precision (degrees at the integer level) was limited, visually creating a "staircase" effect when graphed.

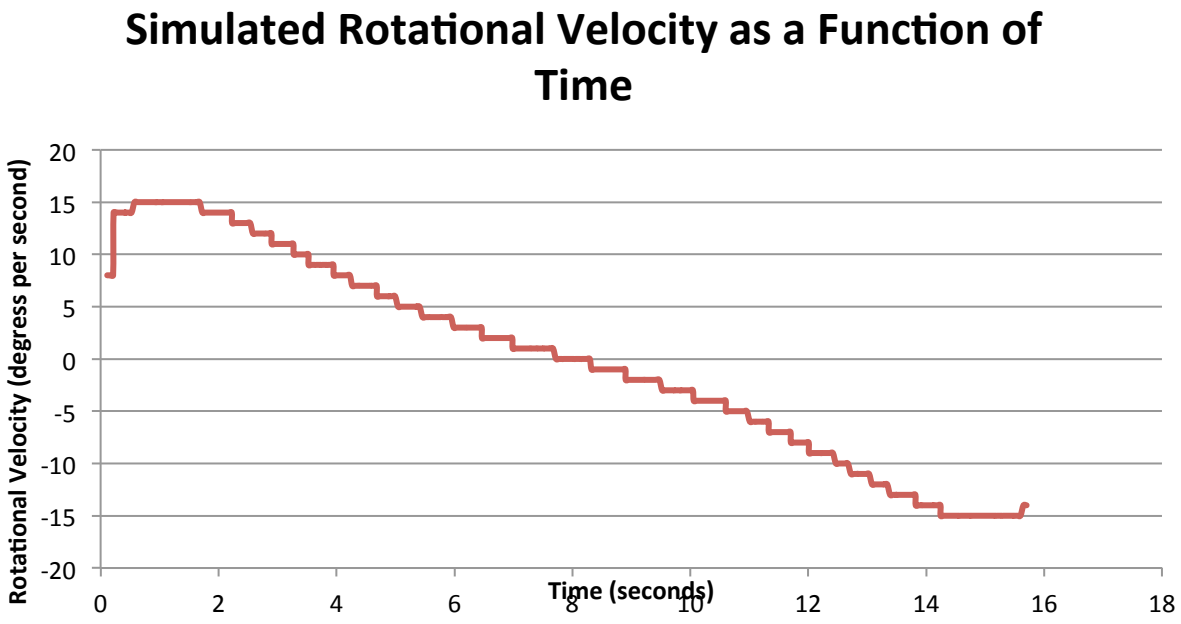

Figure 28: Simulated Hermite curve rotational velocity as a function of time.

As to be expected, the heading of the simulated results (Figure 29) had a "staircase" effect as well, due to the precision of the simulated rotational velocity.

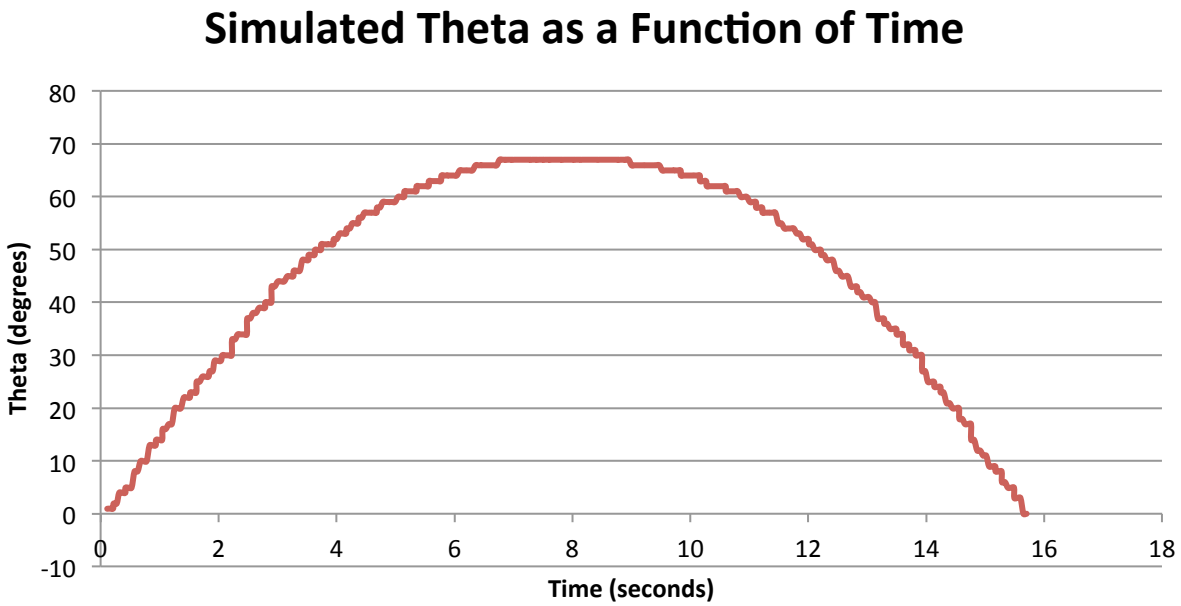

Figure 29: Simulated Hermite curve theta as a function of time.

4.3.4 Comparison of Results

Figure 30 is a comparison of the calculated path versus the simulated path results.

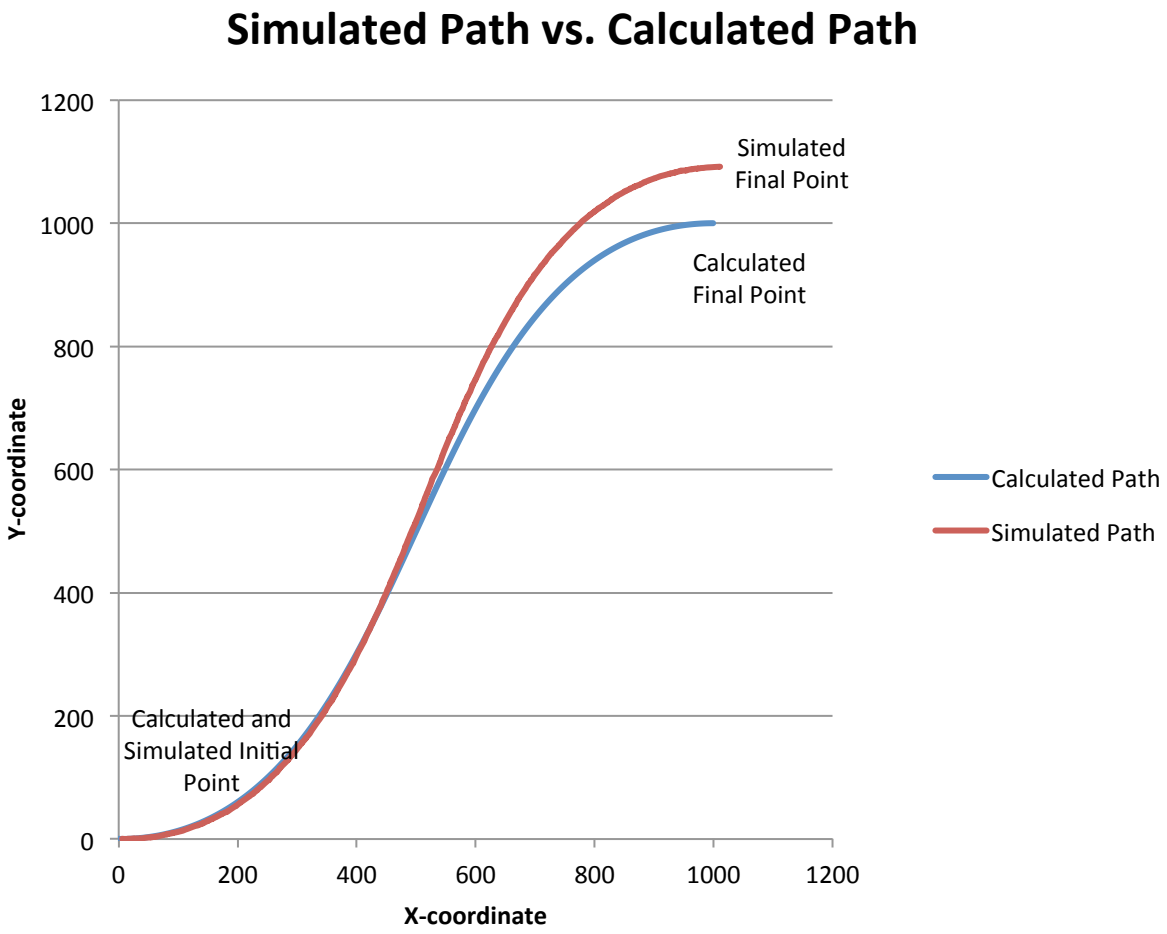

Figure 30: Comparison of a calculated Hermite curve and a simulated Hermite curve.

The simulated path's end position varies slightly from the desired calculated goal position. This is likely due to three main factors:

- Precision of the *setRotationalVelocity()*: the method provided by ARIA accepts a double precision value, but when the rotational velocity value is queried from the sensors, an integer value is returned. This was enough reason to believe that the robot only processed integer level values instead of the desired floating point precision. Unfortunately, this cannot be solved without a software update from ARIA.
- Discrete time increments: the only way to follow the curve was to sample the calculated curve at periodic times. Because continuous sampling is not possible, loss of precision occurred.

- Discrete time of traversal: The time to traverse the curve is only an estimation based on arc length. Estimations cause errors. As more time passed, the accumulation of errors caused larger variations between the calculated and simulated curves. To solve this, curve recalculation was necessary. When the robot's location deviated too far from the Hermite curve's calculated position at any instance of time, a new Hermite curve was calculated. The new Hermite curve used the robot's current position, speed, and bearing as the new initial parameters and kept the old final position, speed, and bearing.

In general, over an average of multiple test cases, the simulated version was approximately 125 millimeters off from the desired goal.

4.3.5 Conclusion

Hermite curves provide a simple, yet effective way of enabling a robot to make curved turns. By using a simple incremental method, they also indirectly provide a means of verifying if the curve's projected path runs into any obstacles or boundaries. If the curve is invalid due to obstacles or boundaries, they can be sharpened by moving either the initial or final point closer to the other and repeating the process.

Chapter 5: D* Lite

5.1 Introduction

D* is an elegant approach for mobile robotics. It successfully meets the objective of a global path planning algorithm for the partially known exploration problem [20]. However, it is a very complex solution. It was quite difficult to understand, let alone modify. Post processing of the algorithm was necessary, but not cost effective, due to the complexity of the implementation. A "lighter" algorithm was imperative. After a few hours of research, a solution was found: D* Lite.

The behavior of D* Lite is the same as that of D*; it aims to efficiently reach a goal while discovering obstructions along the way. However, D* Lite is algorithmically different from D* [21]. Its foundation is built on LPA*, an efficient A* searching algorithm. Due to its LPA* heritage, D* Lite included a heuristic based search for path planning, as demonstrated in Figure 31. D* only used cost of traversal for estimating path desirability. In addition, D* Lite was also beneficial because it was simpler to understand and used fewer lines of code.

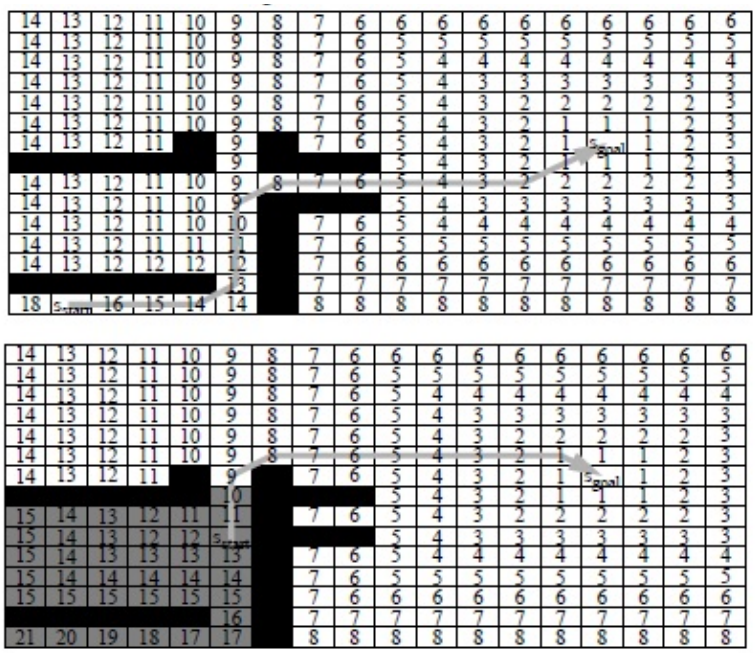

Figure 31: D* Lite using heuristics [21].

The algorithm implemented was based on optimized D* Lite pseudo code, which can be found in Koenig's and Likhachev's report as Figure 6 [21].

5.2 Usage

Usage of D* Lite has been illustrated in Figure 32.

```
dstar = DStarLite(startPoint, goalPoint)

path = dstar.replan()
currentPoint = path.pop()

while (currentPoint != goalPoint)
        if (changeInMap())
                dstar.update(getChangedPoints())
                path = dstar.replan()

                // Smooth path

                // Handle Curved Turns

        currentPoint = path.pop()
```

Figure 32: D* Lite usage pseudo code.

The approach is quite straight forward. Updates in path costs were as simple as sending the new cost to the area to be updated. If an update did occur, the *replan()* method was called, and the newly calculated path was retrieved. Once the path was obtained, path smoothing and determination of curved turns could occur.

5.3 Results

D* Lite was not only easier to implement and easier to handle post processing with, but it was also more efficient. Although Figure 33 compares D* Lite to Focussed D* (an improved derivative of D*), its message is still sound: D* Lite outperforms its counter parts on three levels:

1. Percent of extra vertex expansions [21].
2. Percent of extra vertex accesses (modifying heap node values) [21].
3. Percent of extra heap percolates (heap exchanges) [21].

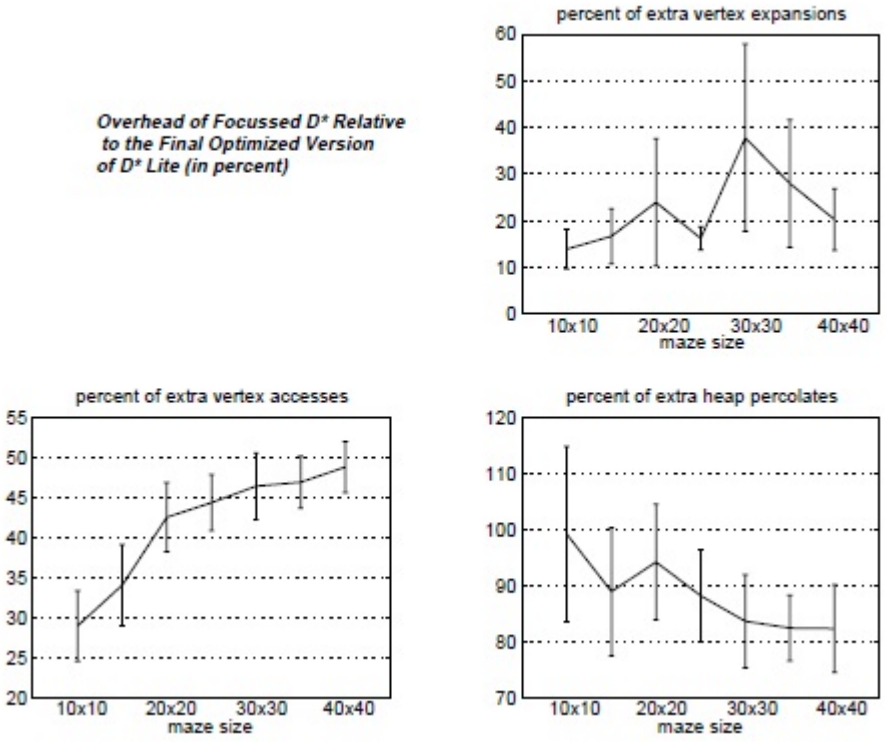

Figure 33: Performance of D* Lite compared to Focussed D* [21].

Chapter 6: Integration

6.1 Introduction

Once the individual components of the thesis objective were completed, they were selectively integrated. Smoothing the path generated by D* Lite was chosen first for post processing for three reasons:

1. Smoothed paths are simpler and more efficient, which made the other components of the thesis objective easier to work with.
2. Smoothing the path reduced the amount of necessary turns.
3. The smoothing algorithm generated the points that were necessary parameters for Hermite curves.

After the path was smoothed, the process discussed in "Chapter 4: Curved Turns" was attempted.

6.2 D* Lite & Smoothed Paths

Following the process outlined in "Chapter 3: Smoothed Paths", the points produced by D* Lite were fed to the smoothing algorithm.

To verify that the process was successful, a custom simulator was made using OpenCV [22]. The custom simulator used basic bitmap images for maps. Black areas were considered *unwalkable* whereas white areas were considered open and traversable. The robot navigated the map pixel by pixel and discovered obstacles along the way until it reached the goal point.

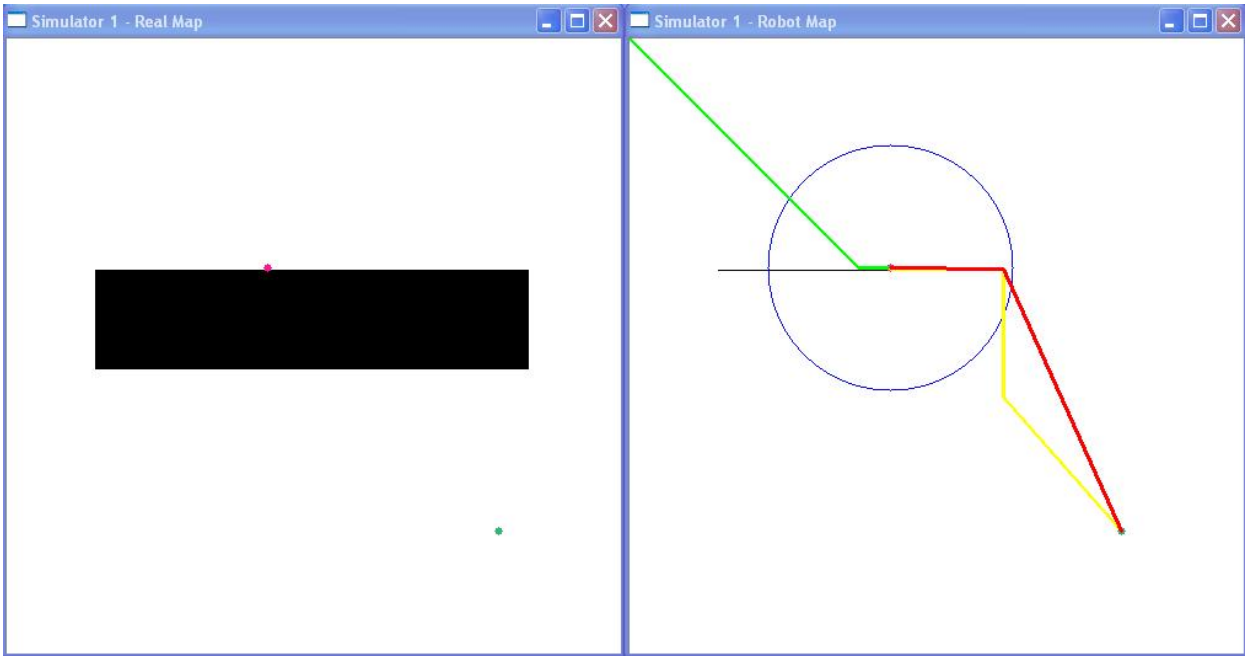

Figure 34: D* Lite smoothing.

Figure 34 is an illustration of the smoothing process. The figure was captured mid way through the simulated robot's journey to the goal position.

On the left side of the figure is the real map the simulated robot was navigating through. It was used for visual verification of the robots location in the "real world". The red dot towards the center was the robot's location at the time the figure was generated. The green point was the goal position the robot was attempting to reach.

On the right side of the figure was the robot's map. It illustrated what the robot was able to "see". The green line was the path already traversed by the robot. The yellow line was the original path produced by the D* Lite planning algorithm. The red line was the D* Lite path smoothed. Finally, the hollow blue circle towards the center was the robot's scanning radius (line-of-sight).

The custom simulator was a distraction from the core of the thesis, but it was worth the time. It became an excellent debugger tool for both the implemented D* Lite algorithm and the smoothing process. After hours of debugging, the smoothing process was successfully integrated with the D* Lite algorithm.

6.3 ARIA, D* Lite, & Smoothed Paths

Integrating the smoothed D* Lite process with ARIA was straight forward once it was accomplished with the custom simulator. All that was necessary was to create a few custom action classes using ARIA's libraries.

The most difficult part of the integration was the communication between the D* Lite algorithm and ARIA's libraries. The D* Lite algorithm used a grid like map which was based on cells/tiles. The map which ARIA's simulator, MobileSim, used was a "physical" map based on millimeters.

To convert between the two, a common scale was required. The scale chosen was the physical robot's radius. Converting between D* Lite's grid map and MobileSim then became nothing more than simple division and multiplication. For example, if the robot's radius was 200 millimeters and the dimensions of the map were 20,000 by 20,000 millimeters, the map D* Lite used was 100 by 100 cells. Using division, point (400, 400) on MobileSim was cell (2, 2) on the D* Lite map. Multiplication was used to convert back to points MobileSim recognized so that the robot could be properly directed.

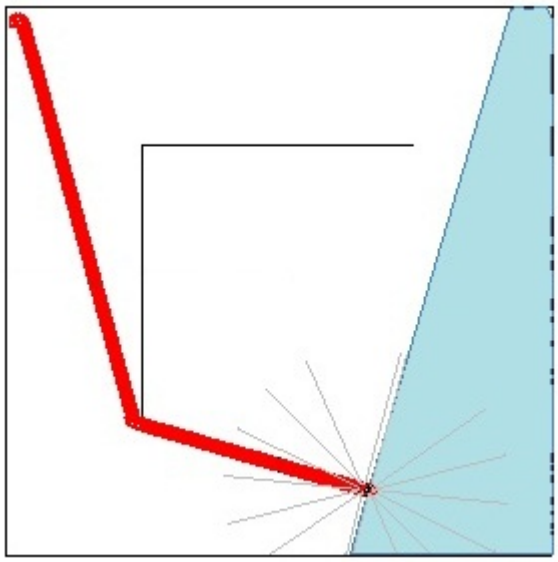

Figure 35: Robot traversing a path in ARIA simulator.

Figure 35 is a snapshot of MobileSim while a robot attempted to reach a goal during a test run. The bold red line was the path the robot already traversed at the time the figure was generated. The blue section was the scanning radius of the robot.

6.4 ARIA, D* Lite, Smoothed Paths, & Curved Turns

Curved turn integration was not accomplished in the time available. The basic idea would have been a combination of the pseudo code presented in Figure 15, Figure 20, and Figure 32. Some possible caveats after the basic process was implemented would have been:

- Assuming the robot hit a dead end, the robot would be required to turn around. Although Hermite curves could have been used to smoothly turn around by making a U-turn, reversing might be more efficient. Preference to reverse or make a U-turn would have been based on a multitude of parameters; including how far back the robot needed to go and the end user requirements (e.g. is it legal for the robot to go in reverse?).
- The turning radius of the robot was never directly taken into account. If the turn was too sharp, a possible solution would be to move points that were close to a wall farther away, as illustrated in Figure 36 by moving P1 to P1'. This would hopefully reduce the chances of the robot colliding with an obstacle while turning.

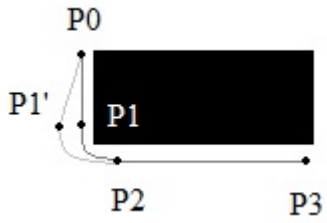

Figure 36: Moving a Hermite curve control point.

Chapter 7: Conclusion

7.1 Accomplishments

The two objectives of this project, smoothed paths and curved turns, were achieved individually. Each objective was developed in their own environment to prevent unwanted interference. Handling the tasks separately ensured each objective was tested thoroughly before integration began.

Along the way D* Lite was implemented. Although not a core objective, it was extremely advantageous. D* Lite was not only easier to integrate with smoothed paths, but was an overall better choice for a global planning algorithm compared to the original D* [21].

Integration of the individual components of the project was not completed. Integrating ARIA's simulator (MobileSim), D* Lite, and path smoothing was achieved almost flawlessly because each of the tasks were properly developed before integration. Unfortunately, time ran short before curved turns could be properly integrated with the rest of the components.

7.2 Limitations

7.2.1 Virtual Machine Resources and Performance

During development, the virtual machine needed to be rebuilt multiple times due to unexpected disk space requirements. Between Windows XP SP3, Visual Studio 2008 SP1, and all of the ARIA software, the virtual machine used just under 15 gigabytes of the allotted 20 gigabytes after being rebuilt.

A virtual machine is inherently "slower" than the host machine. The combination of a near full hard disk, low memory (768 megabytes), and only one dedicated processor made matters worse. Poor performance was noticeable when the robot had a large scanning line-of-sight on a large map. It was hard to determine whether there were problems with the implemented code or the virtual machine itself.

It is suggested that future development use a dedicated PC or a virtual machine with a minimum of 30 gigabytes of hard disk space, 1+ gigabytes of RAM, and more than one dedicated processor core.

7.2.2 Precision of ARIA

As mentioned throughout section "4.3 Results", the precision of ARIA caused some unforeseen problems. The main problems were:

- Magnitude of rotational velocity: the *setRotationalVelocity()* method provided by ARIA accepted a floating point number. However, when the sensors were queried for the rotational velocity immediately after being set, a rounded floating point value was returned. This was enough reason to believe that the robot was receiving a rounded value instead of the desired original value. After searching through the source code of ARIA, it was found that the value being sent was rounded (Figure 37). This slight difference was the main reason the robot was slightly off course when it finished traversing a Hermite curve.

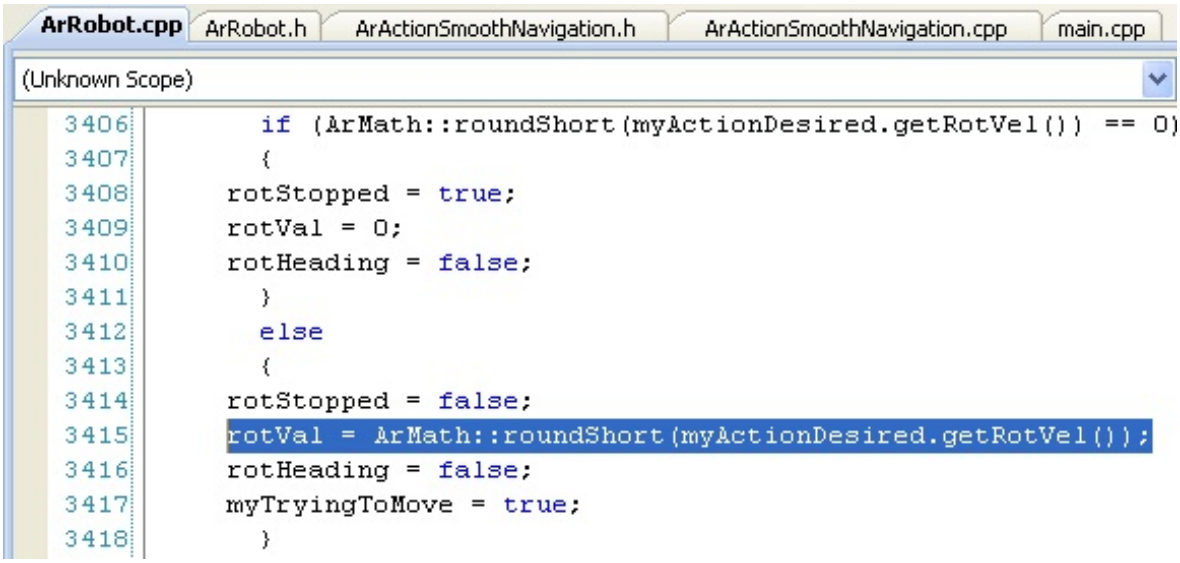

Figure 37: ARIA internally rounding rotational velocity.

- Absolute minimum cycle time: ARIA works based on executing actions on a repeated basis (cycle time). Unfortunately, the cycle time seemed to be capped at a minimum of 50 milliseconds. This meant that the rotational velocity and translational velocity could be sent to the robot at a maximum rate of once every 50 milliseconds. This became a problem when the robot was traveling at a high velocity. The rotational velocity needed to be set at a higher rate, otherwise the robot would go slightly off course, due to its rate of speed.

7.2.3 Time Management

Although more of a personal limitation, poor time management became a limitation in the overall project's success. A lot of time was wasted trying to optimize D* Lite and not enough time was spent focusing on integration. Looking back, one of the fundamental laws of software engineering was broken: make it work first, and then make it work fast.

"Make it run, then make it right, then make it fast". - Kent Beck [23]

7.3 Future Tasks

7.3.1 Complete Integration and Test

The obvious first future task would be to finish integration. This should entail nothing more than integrating the curved turns with the rest of the project components and debugging some of the possible caveats mentioned in section "6.4 ARIA, D* Lite, Smoothed Paths, & Curved Turns".

Once all the components are integrated, they will need to be thoroughly tested. It is likely that many small predicaments will be encountered and will have to be dealt with upon discovery. Testing must also occur with the real robot. Although the simulator is supposedly in accordance with the real world, variation of results in a real environment will most certainly occur.

7.3.2 Verify Varying Path Costs

Most of the tests ran during development used a map that had no varying costs. The left side of Figure 38 was a typical map used during testing. The right hand side of Figure 38 is a map more consistent with a real world environment. Black areas represent obstacles, whereas white areas represent locations that cost near nothing to traverse. The gray-scaled areas are varying costs where the darker the area, the higher the cost of traversal. D* Lite was developed with varying costs in mind, and should need no further verification. The smoothing algorithm would need to be tested and slightly modified if necessary.

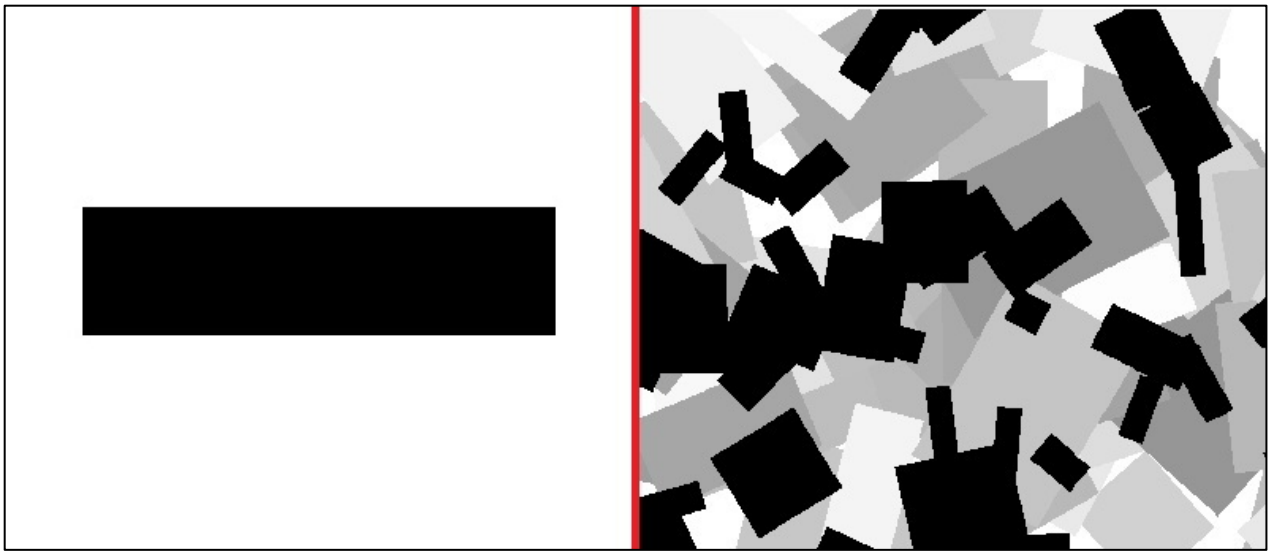

Figure 38: A map with only center section unwalkable (left) versus a map with varying path costs (right) [6].

7.3.3 Modify D* Lite

One unique alternative approach would be to integrate the smoothing process into the D* Lite algorithm. This would require proper manipulation of the costs values, which are illustrated in Figure 39. By integrating the smoothing processes, the need for post processing would be eliminated, thus hypothetically make the approach computationally more efficient. However, this probably would not be considered undergraduate level of work as it would require intensive knowledge of not only D* Lite, but theoretical knowledge of global planning algorithms.

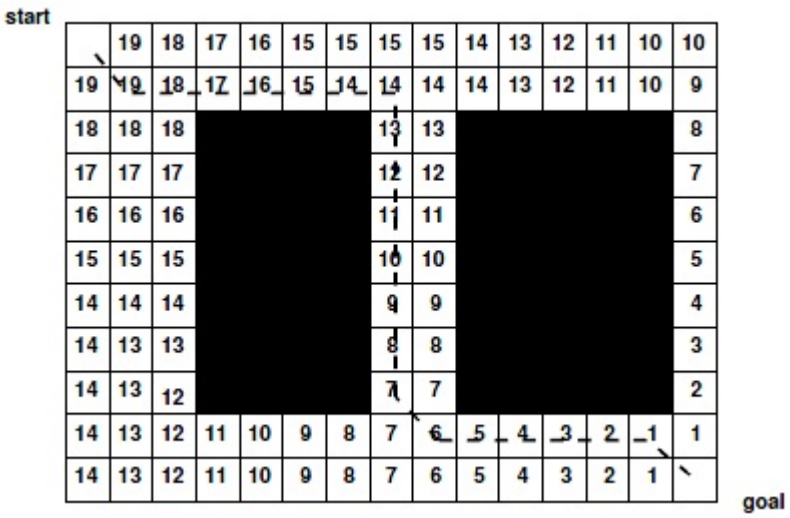

Figure 39: Path costs of a typical planning algorithm [5].

Curved turns should not be integrated with D* Lite. Curved turns in a partially known environment is a local planning issue, and should not be integrated with the global planning algorithm.

7.4 Summary

Mobile robotics is an extremely interesting and inspiring field. The challenges of trying to make an autonomous robot have realistic movement are unconceivable, unless the task is taken on personally. Although a complete polished product was not achieved, individual components were. The research presented in this thesis will hopefully be a useful stepping stone for continued research in the ever-expanding field of Real-Time Smooth Realistic Paths & Navigation.

References

[1]. **Lucas, G.** A Path Based on Third-Degree Polynomials, Constrained at its Endpoints by Position, Orientation, and Speed. [Online] January 16, 2011. [Cited: March 30, 2011.] http://rossum.sourceforge.net/papers/CalculationsForRobotics/CubicPath.htm.

[2]. **Pipenbrinck, N.** Hermite Curve Interpolation. [Online] Hamburg, march 30, 1998. [Cited: May 28, 2011.] http://www.cubic.org/docs/hermite.htm.

[3]. **General Atomics.** General Atomics - Aeronautical - Predator. [Online] http://www.ga-asi.com/products/aircraft/predator.php.

[4]. **iRobot.** Roomba Robot. [Online] [Cited: April 27, 2011.] http://www.irobot.com/.

[5]. **Giesbrecht, J.** *Global Path Planning for Unmanned Ground Vehicles.* Suffield : Defence R&D Canada, 2004. Technical Memorandum.

[6]. **Verbiest, K.** *Real-time Smooth Realistic Paths & Navigation.* Brussels : Royal Military Academy, 2011. Research Proposal.

[7]. **Stentz, A.** *Optimal and Efficient Path Planning for Partially-Known Environments.* Pittsburgh : The Robotics Institute; Carnegie Mellon University, 1994. Technical Paper.

[8]. **Pinter, M.** Toward More Realistic Pathfinding. [Online] March 14, 2001. [Cited: April 5, 2011.] http://www.gamasutra.com/view/feature/3096/toward_more_realistic_pathfinding.php.

[9]. **Baudoin, Y.** *Mechatronics - Part I.* Brussels : Royal Military Academy, 2009.

[10]. **Lucas, G.** A Path Following a Circular Arc. [Online] January 11, 2006. [Cited: March 29, 2011.] http://rossum.sourceforge.net/papers/CalculationsForRobotics/CirclePath.htm.

[11]. **Florida Institute of Technology.** Evans Library - Databases/Indexes. [Online] 2010. [Cited: April 12, 2011.] http://lib.fit.edu/documents/cfm/links/?currentpage=2&id=aac&search=&orderby=Title&sublinks=0&hitsperpage=40&searchtype=1.

[12]. **Chacon, S.** Git - Fast Version Control. [Online] [Cited: April 08, 2011.] http://git-scm.com/.

[13]. **GitHub Inc.** Secure source code hosting and collaborative development - GitHub. [Online] [Cited: April 08, 2011.] https://github.com/.

[14]. **Oracle.** VirtualBox. [Online] [Cited: April 12, 2011.] http://www.virtualbox.org/.

[15]. **Oates, R.** *boB's Guide to Using ARIA*. Nottingham : University of Nottingham, 2006. Technical Guide.

[16]. **Csébfalvi, B.** Ray Tracing. *Computer Graphics and Image Processing.* Budapest, Hungary : s.n., 2010.

[17]. **McNeil, J.** Ray Tracing On a Grid. [Online] March 26, 2007. [Cited: April 15, 2011.] http://playtechs.blogspot.com/2007/03/raytracing-on-grid.html.

[18]. **Weisstein, E.** B-Spline. [Online] [Cited: May 2, 2011.] http://mathworld.wolfram.com/B-Spline.html.

[19]. **Michigan Tech.** B-Spline Curves: Important Properties. [Online] [Cited: May 2, 2011.] http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/bspline-curve-prop.html.

[20]. **Russel, S and Peter, N.** *Artificial Intelligence - A Modern Approach, Second Edition.* Upper Sadle River : Prentice Hall, 2003.

[21]. **Koenig, S and Likhachev, M.** *Improved Fast Replanning for Robot Navigation in Unknown Terrain.* Atlanta : Georgia Institute of Technology, 2002.

[22]. **OpenCV.** Open Source Computer Vision Homepage. *Open Source Computer Vision.* [Online] [Cited: May 29, 2011.] http://opencv.willowgarage.com/wiki/.

[23]. **Advameg Inc.** Basics of the Unix Philosophy. *FAQ.com.* [Online] [Cited: May 31, 2011.] http://www.faqs.org/docs/artu/ch01s06.html.