



NTNU – Trondheim
Norwegian University of
Science and Technology

Optimal trajectory planning for robotized tiling of floors

Stein Melvær Nornes

Master of Science in Engineering Cybernetics

Submission date: June 2013

Supervisor: Tor Arne Johansen, ITK

Co-supervisor: Håvard Halvorsen, nLink AS

Esten Ingar Grøtli, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

The candidate will consider the problem of path planning for a 4-wheel skid-steered mobile platform, intended for use in automated tiling. The following tasks are to be performed:

1. Literature study on trajectory planning for mobile robots.
2. Determine which performance criteria a path needs to fulfill to be considered optimal for an automated tilesetting system.
3. Develop and implement strategies for optimal path planning.
4. Verify the performance of the strategies by simulations, focusing on the quality of the paths, but also considering the calculation time.

Abstract

This thesis describes the strategy of a new approach for automated tilesetting currently under development by nLink AS in Sogndal. The main focus of the thesis is the development of an appropriate strategy for path planning for this robotic system.

A mathematical model of a 4-wheel skid-steering mobile platform with a model-based nonlinear controller is presented. A brief introduction to the basics of path planning is offered, before detailing some relevant algorithms. Three different optimization criteria, based on distance time and energy consumption, are defined and a MATLAB implementation of a Sampling-Based Model Predictive Optimization algorithm is presented. The mobile platform is implemented in SIMULINKTM with input from the SBMPO-algorithm. Simulation results of three different obstacle cases with both constant and variable speed are presented and the performance of the different optimization schemes are reviewed and compared.

The simulations clearly reveal that while optimizing with respect to energy consumption does present some very promising results, the current implementation is far too slow when computing paths with variable speeds. The most promising optimization method is an optimization with respect to time, but with a restriction on the turning radius to avoid sharp, energy-costly turns. This too has some drawbacks, but the ideal optimization strategy can be concluded to be a combination of time and energy efficiency.

Samandrag

Denne oppgåva beskriv strategien for ei ny tilnærming til automatisert flislegging som for tida er under utvikling hjå nLink AS i Sogndal. Hovudfokuset til oppgåva er å utvikla ein passande strategi for baneplanlegging for dette robotsystemet.

Ein matematisk modell for ein firehjuls differensialstyrt mobil platform med ei modellbasert ulineær styringseining blir presentert. Det blir gitt ei kort innføring i grunnleggjande baneplanlegging, før nokon av dei relevante algoritmene blir forklart i større detalj. Tre ulike optimaliseringskriterier blir definert og ein MATLAB-implementasjon av ei "Sampling-Based Model Predictive Optimization"-algoritme blir presentert. Den mobile plattformen er implementert i SIMULINK med inndata frå SBMPO-algoritma. Resultat frå simuleringar av tre ulike hindringsscenario med både konstant og variabel fart blir presentert, og yteevna til dei ulike optimaliseringsstrategiane blir vurdert og samanlikna.

Simuleringane viser tydeleg at sjølv om optimalisering med hensyn til energibruk viser lovande resultat, så brukar den noverande implementasjonen alt for lang tid på å berekna banene. Den mest lovande optimaliseringsmetoden er optimalisering med hensyn på tid, men med ein restriksjon på svingradiusen for å unngå skarpe energikrevande svingar. Denne strategien har også nokon ulemper, men vi kan konkludera med at den ideelle optimaliseringsstrategien må ta hensyn til både tid og energibruk.

Contents

1	Introduction	1
2	Automated tilesetting systems	3
2.1	Case: Rema 1000 Sogndal	4
2.1.1	Step by step scenario	4
2.2	Challenges	6
3	Equations of motion for a skid-steered mobile platform	9
3.1	Trajectory control	15
3.2	Segway RMP 440	18
4	Path planning for mobile wheelbased platforms	21
4.1	Operating Spaces	21
4.2	Sampling-Based vs Combinatorial motion planning	22
4.3	Discrete algorithms	22
4.3.1	Lifelong Planning A*	23
4.3.2	D* Lite	26
4.4	Sampling-Based algorithms	29
4.4.1	Sampling techniques: Halton points	30
4.4.2	Output Sampling: A* assisted Rapidly-expanding Random Tree (RRT)	30
4.4.3	Input Sampling: Sampling Based Model Predictive Control	36
5	The optimization problem	41
5.1	Optimization with respect to distance(Shortest path)	41
5.2	Optimization with respect to time(Quickest path)	42
5.3	Optimization with respect to energy consumption(Easiest path)	42
5.3.1	Defining a power model	43
6	MATLAB implementation of the SBMPO-algorithm	49
6.1	The structure of the path planner	49
6.1.1	The classes	49
6.1.2	The functions	51
6.2	The simplified model of the vehicle	52
6.3	Sampling the input	54
6.4	Smoothing the output	54
6.5	Obstacle and goal handling	56

6.5.1	The configuration space of the vehicle	56
6.5.2	Obstacle handling	57
6.5.3	Goal handling	58
7	Simulation	59
7.1	Specification	59
7.1.1	Assumptions and simplifications in the modelling . .	59
7.1.2	Model parameters	60
7.1.3	Simulation parameters	60
7.2	Simulator structure	61
7.2.1	Prepared functionality for future work	68
8	Defining the test cases	69
8.1	Test parameters	69
8.2	Case 1: No obstacles	69
8.3	Case 2: Simple cluster of obstacles	69
8.4	Case 3: Complex cluster of obstacles	70
9	Simulation results	73
9.1	Case 1: No obstacles (variable speed)	74
9.2	Case 2: Simple set of obstacles	78
9.2.1	Constant speed=1.0 m/s	78
9.2.2	Speed=1.0-1.4 m/s	81
9.3	Case 3: Complex cluster of obstacles	85
9.3.1	Constant speed=1.0 m/s	85
9.3.2	Speed=1.0-1.4 m/s, gridsize=0.2 m	89
9.3.3	Speed=1.0-1.4 m/s, gridsize=0.1 m	92
10	Discussion	95
10.1	Choice and implementation of the path planning algorithm	95
10.1.1	Implementing the SBMPO-algorithm	95
10.1.2	Controller performance	95
10.2	Comparison of the optimization schemes	96
10.2.1	The findings of Collins et.al.[29]	96
10.2.2	Energy vs Time with limited turning radius	97
10.3	Choosing a different algorithm	98
10.3.1	Possible modification: D* Lite	98
10.3.2	Completely different approach?	98
11	Conclusion	101

12 Further work	103
------------------------	------------

Bibliography	105
---------------------	------------

List of Figures

1.1	A draft of the nLink robot	2
2.1	Pattern examples formed by a software controlled robot . .	3
2.2	Blueprints of Rema 1000 Sogndal	4
2.3	Workers tiling the new Rema 1000 Sogndal	5
2.4	nLink animation of a multi-robot system at work	8
3.1	Free-body diagram of the vehicle	10
3.2	Active and resistive forces of the vehicle	12
3.3	Implementation of the Karnopp model for friction	14
3.4	Illustration of a Segway RMP 440 mobile platform	19
4.1	Lifelong Planning A* algorithm	25
4.2	D* Lite algorithm	28
4.3	Comparison of random sample generators	31
4.4	Rapidly-Exploring Random Tree (RRT) algorithm	32
4.5	The distribution of RRT candidate milestones	35
4.6	Illustration of the necessity of an implicit state grid	37
4.7	Sampling-Based MPC algorithm	39
5.1	Wheeled skid-steering vehicle during curving	43
5.2	The $\alpha(r)$ term of the power model	45
5.3	The $\beta(r)$ term of the power model	46
5.4	2D representation of power model	47
5.5	3D representation of power model	48
6.1	Class diagram of the MATLAB implementation	50
6.2	Accuracy of different simple models for the vehicle	53
6.3	Output smoothing functions	56
7.1	The complete implemented system	63
7.2	The implementation of the skid-steered model	64
7.3	The c_func-block from figure 7.2	64
7.4	The R_x-block from figure 7.3	65
7.5	Implementation of the modified sign function	65
7.6	The "Optional Deadzone w.rotation"-block from figure 7.2. .	65
7.7	The "Optional Deadzone"-block from figure 7.6.	66
7.8	The implementation of the controller	67
8.1	A simple cluster of obstacles	70
8.2	A more complex cluster of obstacles	71

9.1	The Shortest path for no obstacles	75
9.2	The Quickest path for no obstacles	76
9.3	The Easiest path for no obstacles	76
9.4	Comparison of paths for no obstacles	77
9.5	Simple obstacles, const. speed: Shortest path ($r_{min} = 5$) . .	78
9.6	Simple obstacles, const. speed: Easiest path	79
9.7	Simple obstacles, const. speed: Shortest path ($r_{min} = 11$) .	79
9.8	Comparison of the optimization schemes with simple obsta- cle set and constant speed	80
9.9	Simple obstacles, var. speed: Shortest path	81
9.10	Simple obstacles, var. speed: Quickest path ($r_{min} = 5$) . . .	82
9.11	Simple obstacles, var. speed: Easiest path	82
9.12	Simple obstacles, var. speed: Quickest path ($r_{min} = 11$) . .	83
9.13	Comparison of the optimization schemes with simple obsta- cle set and variable speed	84
9.14	Complex obstacles, const. speed: Quickest path ($r_{min} = 5$)	86
9.15	Complex obstacles, const. speed: Easiest path	86
9.16	Complex obstacles, const. speed: Quickest path ($r_{min} = 11$)	87
9.17	Comparison of the optimization schemes with complex ob- stacles and constant speed	88
9.18	Complex obst., var. speed, grid0.2: Quickest path ($r_{min} = 5$)	89
9.19	Complex obst., var. speed, grid0.2: Easiest path	90
9.20	Complex obst., var. speed, grid0.2: Quickest path ($r_{min} = 11$)	90
9.21	Comparison of the optimization schemes with complex ob- stacle set, variable speed and gridsize=0.2m	91
9.22	Complex obst., var. speed, grid0.1: Quickest path ($r_{min} = 5$)	92
9.23	Complex obst., var. speed, grid0.1: Quickest path ($r_{min} = 11$)	93
9.24	Comparison of the optimization schemes with complex ob- stacleset, variable speed and gridsize=0.1m	94

List of Tables

1	Specification for the Segway RMP440 LE	18
2	Model parameters used in SIMULINK	60
3	Algorithm tuning parameters	74
4	Comparison of the optimization schemes with no obstacles .	77
5	Comparison of the optimization schemes with simple obsta- cles and constant speed	80
6	Comparison of the optimization schemes with simple obsta- cles and variable speed	83

7	Comparison of the optimization schemes with complex obstacles and constant speed	87
8	Comparison of the optimization schemes with complex obstacles, variable speed and gridsize 0.2m	91
9	Comparison of the optimization schemes with complex obstacles, variable speed and gridsize 0.1 m	93

1 Introduction

Tiling floors is hard manual labour which takes its toll on the human body. Lifting and carrying heavy tiles, combined with a harmful work posture, make the workers prone to ergonomic injuries which in turn may lead to early retirement. Since the work is both heavy, damaging and repetitive, this makes it an ideal place to introduce robotics.

The global ceramic tiles market is estimated to reach 8,6 billion square meters by 2015 [1]. During the development of a floor tiling system, it is reasonable to focus this on the large and simple areas of for instance shopping malls, while leaving the more difficult sections to experienced manual workers.

This project will be based on a mobile robot under development by nLink AS (see fig. 1.1) and a real world case provided by Brødr. Olsen, the largest tiling contractor in Norway. There are too many challenges associated with this development to address in a single project report. The modeling of the platform is discussed thoroughly in [2]. The focus of this report will be on defining the work case, developing and implementing a path planning scheme for the mobile platform, and running simulations to test the implemented path planner.

The report will begin by outlining a specific work case and the challenges associated with it, before reviewing some path planning algorithms. Three different optimization criteria are proposed and detailed. The skid-steered platform-model implemented in SIMULINK in [2] is expanded with a MATLAB implementation of an SBMPO-algorithm to also cover path planning, and the results of the simulations are used as a basis for a discussion of the quality of the path planning schemes.



Figure 1.1: A draft of the nLink robot

2 Automated tilesetting systems

Industrial robots are typically mounted in a fixed position next to a conveyor belt. This creates a simple, easily describable working environment for the robot, but limits its uses to inside a factory. Moving the industrial robots out of the factories and workbenches would create numerous new areas of application, but not without introducing a significant increase in complexity.

The process of floor tiling is a useful stepping stone in the learning curve in the development of an automatic mobile manipulator system that could eventually carry out any task. This is because the workspace is 2-dimensional, which makes it a much simpler starting point than a full 3-dimensional workspace. As early as 1996, researchers at Carnegie Mellon University Designed a mobile robot system for automatic floor tiling [3]. The proposed robot was a four-wheeled robot with omni-directional wheels and a somewhat complex internal tile-feeding and -placement system. Positioning was to be achieved using a planar laser-scanning device mounted on the robot with active or retroflective targets attached to the facility.

A design more closely resembling the nLink system is shown in [4]. This robot consists of a robotic arm mounted on a work module, with the work module attached to a mobility module by a rotating plate. Both [3] and [4] focus primarily on the tilesetting itself and the demands for accuracy in the actual placement of the tiles.

Some floor tiling jobs will require intricate mosaic designs. These require much extra planning and are very labour intensive for a manual tiler, however for a robot this task is not significantly more complex than any other tiling job. In [5], Oral and Inal used a four degrees of freedom cartesian robot mounted on a fixed workstation to demonstrate how a robot can improve the tiling process of mosaics by increasing the speed, reducing the errors and allowing more flexibility for complex patterns, see Fig. 2.1.

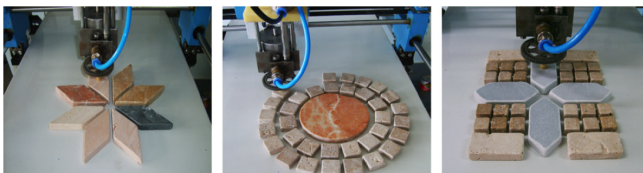


Figure 2.1: Pattern examples formed by the software controlled robot of Oral and Inal[5].

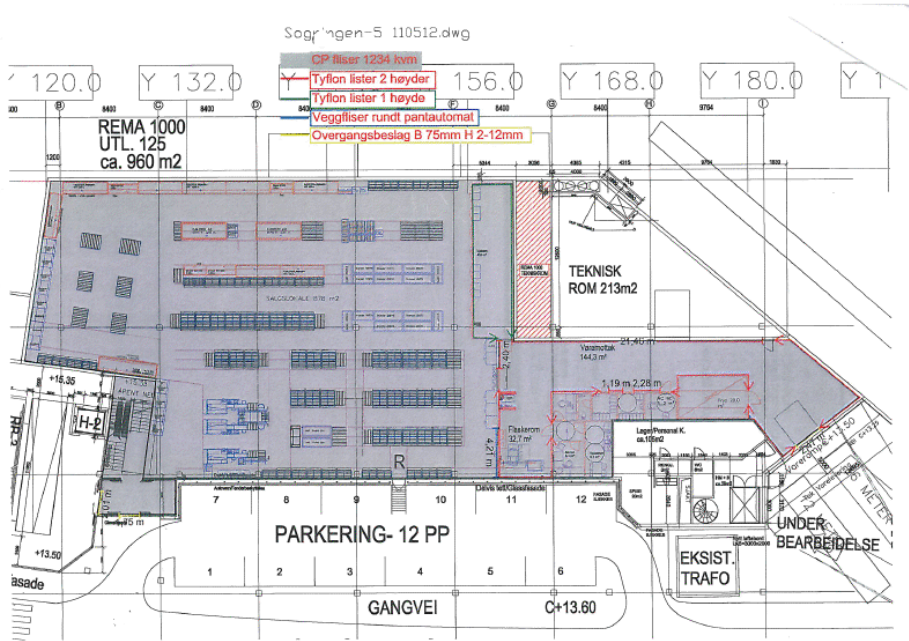


Figure 2.2: Blueprints of Rema 1000 Sogndal

2.1 Case: Rema 1000 Sogndal

In preparation of the development of the nLink robot, nLink CEO Håvard Halvorsen spent a day studying the tiling work process of Brødr. Olsen in the construction of a new Rema 1000 store in Sogndal (fig. 2.2). Based on video footage of the work (fig. 2.3) and conversations with the workers, we have defined the following step-by-step scenario for the operation of the tilesetting robot:

2.1.1 Step by step scenario

Prerequisites:

- The area to be tiled is empty and ready.
- Tiles are available on one ore more pallets.
- The workers have a clear plan for the work.
- The robot is inside the work area, and is fully loaded with 80 tiles .
- The worker laying the glue has already started.



Figure 2.3: Workers tiling the new Rema 1000 Sogndal

Work method:

- The whole area indicated is to be tiled. It is assumed that the workers leave enough room on the outside of the last stretch of the area for the robot to manoeuvre.
- The worker laying the glue has the remote, and stops the robot if it catches up with him.
- Tiles are laid out in two rows, and the robot places 2 by 2 tiles from each stopping point.
- The workers can add/remove pallets in the program during operation, when they run out of tiles.
- The tiles are (at this point in development) loaded manually by the workers.

Step by step:

1. Workers use a tripod or similar to indicate the work area.
 - (a) Corners
 - (b) Position(s) of tile pallet(s)
 - (c) Position(s) of obstacle(s) like columns, drains etc.
 - (d) The intended starting position and -direction of the tiling

- (e) The grid for the expansion joints (see section 2.2, paragraph Precision)
- 2. The system draws up the work area and the scenario in an application, and the worker confirms that the scenario is correct.
- 3. System startup via remote
- 4. Robot verifies it's position
- 5. Moves to the first position to lay tiles from
- 6. Verifies position
 - (a) If the deviation from ideal position is too large to compensate for using the robotic arm, system repeats from 5.
- 7. Lays 2 by 2 tiles, with the robotic arm compensating for any deviation from the ideal positioning of the platform
- 8. Moves to the next position
 - (a) As long as the robot has got tiles left, repeat from 6
 - (b) Continue when tile tray is empty
- 9. Moves to the nearest pallet with tiles, making sure to position and rotate itself to the position and orientation the operator has instructed
- 10. The operator loads the robot with tiles (the robot will do this automatically in a later stage of development)
- 11. Moves back to the next position to lay tiles from, then repeat from 6

2.2 Challenges

There are numerous challenges associated with developing such a system.

Human-Robot interaction

The robot will be operating in close collaboration with human workers, and this means a number of safety precautions need to be taken. It goes without saying that a robot that weighs in excess of 100 kg, is capable of moving freely around the room and has a large moving robotic arm mounted on top, would pose a serious hazard to the human workers if it is not properly designed to ensure their safety.

The robot will need to be fitted with several sensors capable of detecting any obstacles in the path of either the platform or the manipulator arm. Also, there will need to be set reasonable restrictions for operation

speeds, for instance restricting the drive speed of the mobile platform to walking speed (~ 1.4 m/s [6]).

Navigation and Obstacle avoidance

The working environment of the robot will contain several potential obstacles, for instance tile pallets, buckets of glue, strips of already applied glue, and most importantly the human workers. The obstacles will constantly be changing, requiring the system to reevaluate the optimum path. The mobile platform needs to effectively navigate to the correct position in the work area, so both path planning and trajectory tracking will be very important aspects to consider.

Precision

The tiles used are 30 cm by 30 cm and the distance between the tiles is to be 1.5 mm. The accuracy tolerance level is that the tiles do not overlap, so in the worst-case situation, they can be placed in contact with each other.

For every 6th meter in each direction, a 5 mm joint with a different type of grout (tiling glue) is added. This wide joint is added to allow the flexibility required to account for the difference in temperature-induced expanding between concrete and ceramic tiles.

The tolerance level for height deviation is 1 mm difference between adjacent tiles. This is the standard of Brødr. Olsen, the norwegian standard for this is 2 mm.

As stated in section 2.1.1, the robotic arm will compensate for any error in the position of the mobile platform, however this demands the position measurements of the platform to be very accurate.

Efficiency

Time is money, and particularly when dealing with such large areas, any small delay in a small part of the process will accumulate and may in the end become a significant factor. This means the system needs to run efficiently, stopping as little as possible and not making too many unnecessary turns.

Reliability

Even more important than moving the robot efficiently, is making sure the system is running properly as much of the time as possible. If the system breaks down frequently and the workers need to spend a lot of

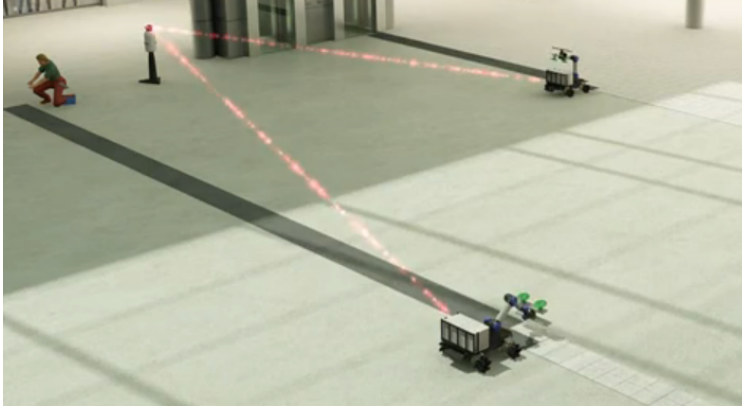


Figure 2.4: nLink animation of a multi-robot system at work

time repairing and maintaining the unit, it does not matter how efficiently the system works when it is up and running, the tiling company will be wasting time and money.

Simplicity

The average tiler does not have a degree in robotics. This needs to be taken into account when designing user interface, keeping it simple and intuitive to use, with a minimum of unnecessary functions. How a system performs in a laboratory with the designers is interesting for testing purposes only, what really matters is how it performs in the actual working environment being operated by the ones that are actually going to be using it.

Scaling

Being able to do a job quickly is a major business advantage, and because of this the system needs to be able to consist of multiple robots working in the same space with a larger crew (see fig. 2.4). This of course adds another layer of complexity, because we now also need to consider robot-robot interaction.

3 Equations of motion for a skid-steered mobile platform

The mobility module of the nLink robot is currently planned to be based on the Segway RMP 440 [7] presented in section 3.2.

One of the primary advantages of this platform as opposed to other platforms is the loading capacity. When looking at the proposed step-by-step work-scenario of the robot in section 2.1.1, it is easy to see that the efficiency of the tiling system will be highly dependant on how often the robot needs to be reloaded with tiles, so being able to carry a large payload of tiles is vital.

Another selling point for this skid-steered alternative is its mechanical simplicity. A simple and robust design is often more reliable, since there are fewer moving parts that can fail. In [8], Tlale and de Villiers point to the high wear a surface like concrete will induce on the omni-directional Mecanum wheels, and discourage operation of a Mecanum-wheeled platform on surfaces with a high coefficient of friction, like for instance concrete. The floors to be tiled using the nLink robot will naturally be made of concrete.

Skid-steering is a term for the steering of a vehicle with fixed wheels (no steering wheels) and using different wheel speeds on the opposing sides to induce a skidding rotation of the vehicle. A platform using skid-steering (like the one presented in section 3.2) can be made very simple and mechanically robust, since there is no need for steering wheels. However, as pointed out in [9], the varying tire/ground interactions and over-constrained contact makes it quite challenging to obtain accurate dynamic models and tracking control systems for such mobile robots. [9] uses a highly simplified model for friction, and still ends up with a very complex set of equations that are difficult to implement in a simulator. Since the tiling robot is only going to be operating at walking speed or lower (in order to guarantee the safety of the human workers in its workspace), it seems reasonable to base the mathematical model in the simulator primarily on the simpler model derived by [10] instead. In addition to this, we will utilize some of the elements and notation introduced by [11] and [9], and introduce a new approach for the modeling of the sliding friction. A thorough review of the behaviour of the proposed model and controller is given in [2].

Figure 3.1 shows the schematic of a skid-steered robot. We make the following assumptions:

1. The vehicle is rigid and moves on a horizontal plane with all four wheels always in contact with the ground surface.

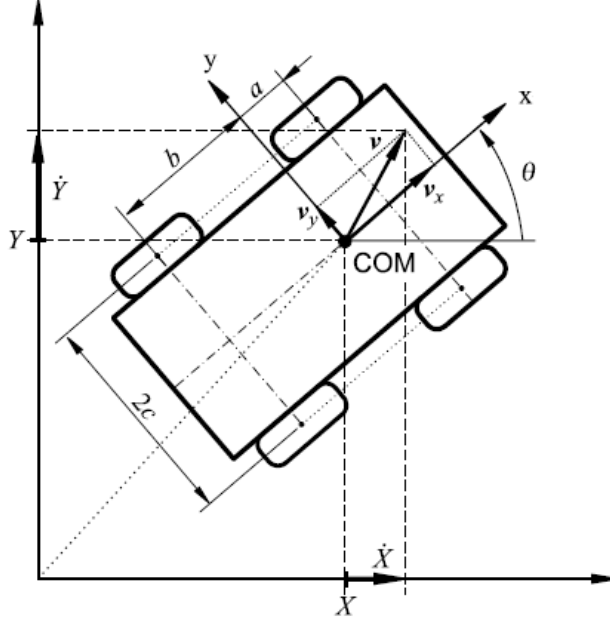


Figure 3.1: Free-body diagram[11].

2. The vehicle speed is below 2.5 m/s.
3. The longitudinal wheel slippage is negligible.
4. The lateral force on the tires is a function of its vertical load.
5. Each sides two wheels rotate at the same speed.

Define a fixed reference frame $F(X, Y)$ and a moving frame $f(x, y)$ attached to the vehicle body, with origin at the vehicle center of mass COM and angle θ with respect to the reference frame (see Fig. 3.1). The center of mass is located at distances a and b (usually, $a < b$) from the front and rear wheels axes, respectively, and is symmetric with respect to the vehicle sides (at distance c).

Let $\dot{x}, \dot{y}, \dot{\theta}$ be respectively, the longitudinal, lateral, and angular velocity of the vehicle in frame f . In the fixed frame F , the absolute velocities are

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \end{bmatrix} = \begin{bmatrix} \dot{x} \cos \theta - \dot{y} \sin \theta \\ \dot{x} \sin \theta + \dot{y} \cos \theta \end{bmatrix} = \mathbf{R}(\theta) \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix},$$

Differentiation with respect to time gives

$$\begin{bmatrix} \ddot{X} \\ \ddot{Y} \end{bmatrix} = \mathbf{R}(\phi) \begin{bmatrix} \ddot{x} - \dot{y}\dot{\theta} \\ \ddot{y} + \dot{x}\dot{\theta} \end{bmatrix} = \mathbf{R}(\theta) \begin{bmatrix} a_x \\ a_y \end{bmatrix},$$

where a_x and a_y are the absolute accelerations expressed in the moving frame f . At each instant the vehicle motion is a pure rotation around a point C , the instantaneous center of rotation (ICR), in which the linear velocity components in f vanish. Its coordinates are

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} -\dot{y}/\dot{\theta} \\ \dot{x}/\dot{\theta} \end{bmatrix}$$

The angular velocity $\dot{\theta}$ and the lateral velocity \dot{y} both vanish during straight line motion, and the ICR goes to infinity along the y -axis. On a curved path, the ICR shifts (forwards) by an amount $|x_c|$. When $\dot{y} = 0$, there is no lateral skidding. If x_c goes out of the robot wheelbase, the vehicle skids dramatically with loss of motion stability.

Finally, note that the longitudinal velocity \dot{x}_i and the lateral (skidding) velocity \dot{y}_i of each wheel ($i = 1, \dots, 4$) are given by

$$\begin{aligned} \dot{x}_1 &= \dot{x}_3 = \dot{x} - c\dot{\theta} & (\text{left}) \\ \dot{x}_2 &= \dot{x}_4 = \dot{x} + c\dot{\theta} & (\text{right}) \\ \dot{y}_1 &= \dot{y}_2 = \dot{y} + a\dot{\theta} & (\text{front}) \\ \dot{y}_3 &= \dot{y}_4 = \dot{y} - b\dot{\theta} & (\text{rear}) \end{aligned} \tag{3.1}$$

Figure 3.2 shows the active and resistive forces acting on the vehicle. The wheels develop the traction forces F_{xi} and are subject to longitudinal resistance forces R_{xi} , for $i = 1, \dots, 4$. We assume that wheel actuation is equal on each side so as to reduce longitudinal slip. Thus, the following always holds: $F_{x3} = F_{x1}$ and $F_{x4} = F_{x2}$. Lateral forces F_{yi} act on the wheels as a consequence of lateral skidding. Also, an active torque M around the center of mass is induced in general by the active F_{xi} forces, and a corresponding resistive moment M_r is likewise induced by the resistive F_{yi} and R_{xi} forces.

For a vehicle of mass m and inertia I about its center of mass, the equations of motion can be written in frame f as:

$$\begin{aligned} ma_x &= 2F_{x1} + 2F_{x2} - R_x \\ ma_y &= -F_y \\ I\ddot{\theta} &= 2c(F_{x2} - F_{x1}) - M_r. \end{aligned} \tag{3.2}$$

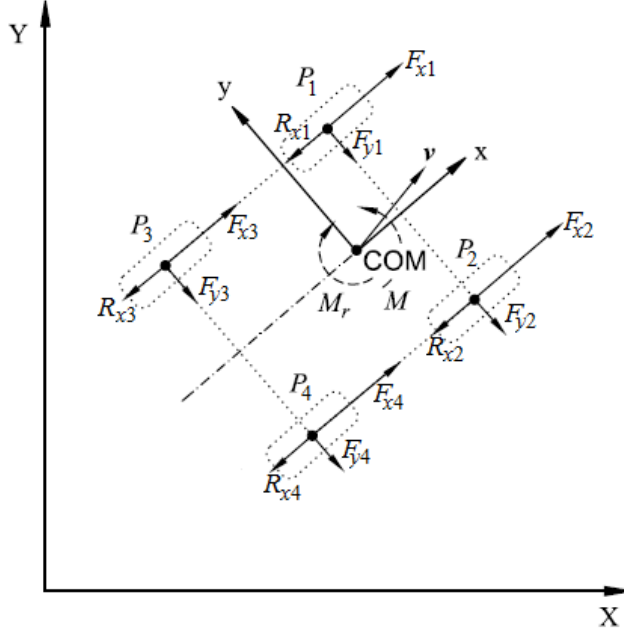


Figure 3.2: Active and resistive forces of the vehicle[11].

To express the longitudinal resistive force R_x , the lateral resistive force F_y , and the resistive moment M_r , we should consider how the vehicles gravitational load m_g is shared among the wheels and introduce a Coulomb friction model for the wheel-ground contact. We have

$$F_{z1} = F_{z2} = \frac{b}{a+b} \cdot \frac{mg}{2}$$

$$F_{z3} = F_{z4} = \frac{a}{a+b} \cdot \frac{mg}{2}.$$

At low speed, the lateral load transfer due to centrifugal forces on curved paths can be neglected. In case of hard ground, we can assume that the contact patch between wheel and ground is rectangular and that the vertical load of the tire produces an uniform pressure distribution. Under this condition, $R_{xi} = f_r F_{zi} \text{sgn}(\dot{x}_i)$, where f_r is the coefficient of rolling resistance, assumed independent from velocity[12]. The total longitudinal resistive force is then

$$R_x = \sum_{i=1}^4 R_{xi} = f_r \frac{mg}{2} \left(\text{sgn}(\dot{x}_1) + \text{sgn}(\dot{x}_2) \right). \quad (3.3)$$

Introducing a lateral friction coefficient μ , the lateral force acting on each wheel will be $F_{yi} = \mu F_{zi} \operatorname{sgn}(\dot{y}_i)$. The total lateral force is thus

$$F_y = \sum_{i=1}^4 F_{yi} = \mu \frac{mg}{a+b} \left(b \operatorname{sgn}(\dot{y}_1) + a \operatorname{sgn}(\dot{y}_3) \right) \quad (3.4)$$

while the resistive moment is

$$\begin{aligned} M_r &= a(F_{y1} + F_{y2}) - b(F_{y3} + F_{y4}) + c[(R_{x2} + R_{x4}) - (R_{x1} + R_{x3})] \\ &= \mu \frac{abmg}{a+b} (\operatorname{sgn}(\dot{y}_1) - \operatorname{sgn}(\dot{y}_3)) + f_r \frac{cmg}{2} (\operatorname{sgn}(\dot{x}_2) - \operatorname{sgn}(\dot{x}_1)) \end{aligned} \quad (3.5)$$

A major disadvantage with using the Coulomb friction model in its most common form, $F_f = F_c \operatorname{sgn}(v)$, is that it fails to model the correct physical behaviour when the velocity v is zero and the acting force F_a is in the interval $0 < |F_a| < F_c$. When simulating such a system with a fixed time-step solver, the numerical solution will suffer from strong oscillations, whereas a variable time-step solver will basically stop because the time-steps will approach zero in order to reach the specified accuracy[13]. In [11], this problem is avoided by replacing the sign function with the following approximation:

$$\widehat{\operatorname{sgn}}(v) = \frac{2}{\pi} \arctan(k_s v),$$

where $k_s \gg 1$ is a constant which determines the accuracy of the approximation according to the relation

$$\lim_{k_s \rightarrow \infty} \frac{2}{\pi} \arctan(k_s v) = \operatorname{sgn}(v).$$

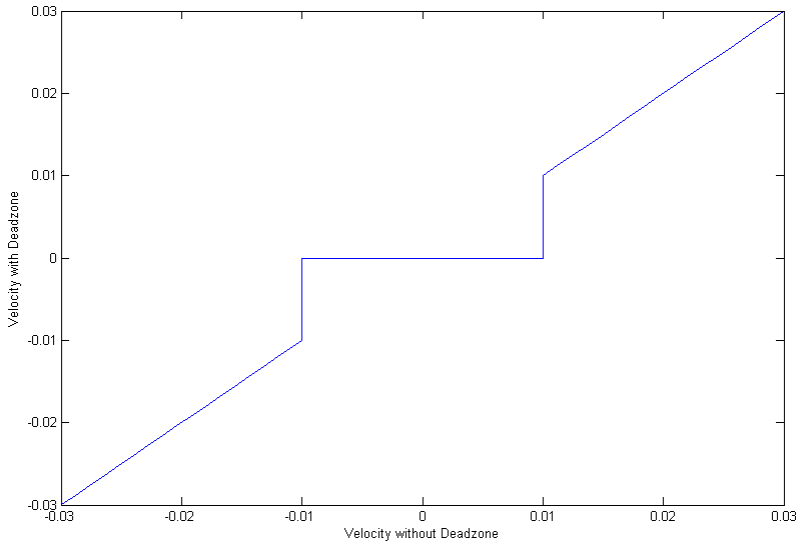
An alternate approach is to implement a Karnopp model as described in [14], by adding a deadzone of $\pm \delta_v$ in the velocity of the vehicle (figure 3.3a) and replacing the sign function with a gain of $\frac{1}{\delta_v}$ and a saturation keeping the output between ± 1 (figure 3.3b).

The dynamic model can be rewritten in frame F , introducing the generalized coordinates $\mathbf{q} = (X, Y, \theta)$ and matrix notation

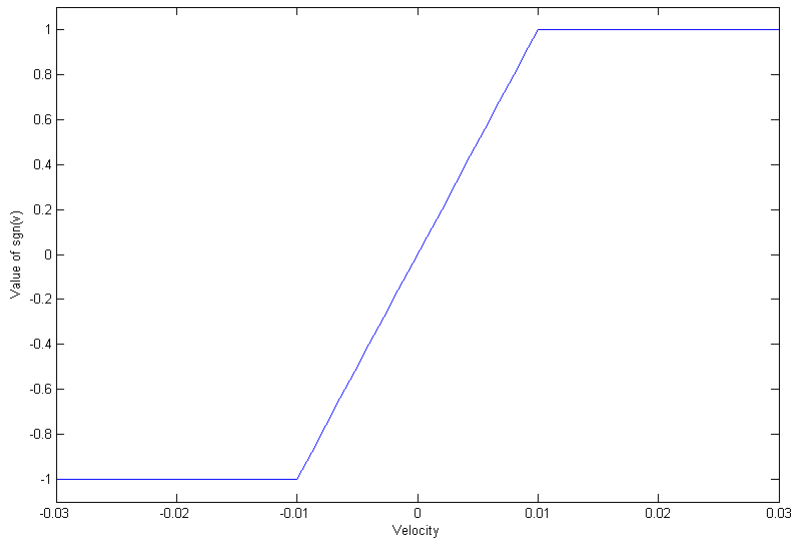
$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{E}(\mathbf{q})\boldsymbol{\tau} \quad (3.6)$$

with

$$\mathbf{M} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & I \end{bmatrix}, \quad \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) = \begin{bmatrix} R_x \cos \theta - F_y \sin \theta \\ R_x \cos \theta + F_y \sin \theta \\ M_r \end{bmatrix},$$



(a)



(b)

Figure 3.3: Implementation of the Karnopp model with $\delta_v=0.01\text{m/s}$. (a) The deadzone of the velocity, (b) Gain and Saturation as a replacement for $\text{sgn}(v)$.

and

$$\mathbf{E}(\mathbf{q}) = \begin{bmatrix} \cos \theta / r & \cos \theta / r \\ \sin \theta / r & \sin \theta / r \\ -c/r & c/r \end{bmatrix}, \quad \tau_i = 2rF_{xi} \ (i = 1, 2),$$

with r being the wheel radius, τ_1 and τ_2 the torques produced by the left and right side motors at the load side, respectively. An ideal transmission factor is also assumed.

3.1 Trajectory control

We start by observing that x_c (the x -axis projection of the instantaneous center of rotation) cannot be larger than a . If this happens, the vehicle would skid along the y -axis thus losing control. In order to have the vehicle move properly, the following should be fulfilled:

$$\left| -\frac{\dot{y}}{\dot{\theta}} \right| < a.$$

Thus, we can introduce the following operative constraint

$$\dot{y} + d_0 \dot{\theta}, \quad 0 < d_0 < a, \quad (3.7)$$

or, in terms of generalized coordinates:

$$\begin{bmatrix} -\sin \theta & \cos \theta & d_0 \end{bmatrix} \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \mathbf{A}(\mathbf{q})\dot{\mathbf{q}} = 0. \quad (3.8)$$

This relation represents a nonholonomic constraint that can be attached to the dynamic model (3.6) for control design purposes. When this constraint is enforced, the robot dynamics becomes

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{E}(\mathbf{q})\boldsymbol{\tau} + \mathbf{A}(\mathbf{q})\boldsymbol{\lambda}, \quad (3.9)$$

where $\boldsymbol{\lambda}$ is the vector of Lagrange multipliers corresponding to equation (3.8).

Admissible generalized velocities $\dot{\mathbf{q}}$ can be expressed as

$$\dot{\mathbf{q}} = \mathbf{S}(\mathbf{q})\boldsymbol{\eta}, \quad \boldsymbol{\eta} \in \mathbb{R}^2, \quad (3.10)$$

where $\boldsymbol{\eta}$ is a pseudo-velocity and $\mathbf{S}(\mathbf{q})$ is a 3×2 full rank matrix, whose columns are in the null space of $\mathbf{A}(\mathbf{q})$, e.g.,

$$\mathbf{S}(\mathbf{q}) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \\ 0 & -\frac{1}{d_0} \end{bmatrix}.$$

We can differentiate (3.10) and eliminate λ from equation (3.9) to obtain the reduced dynamic model (dropping dependencies)

$$\begin{aligned}\dot{\mathbf{q}} &= \mathbf{S}\boldsymbol{\eta} \\ \dot{\boldsymbol{\eta}} &= (\mathbf{S}^T \mathbf{M} \mathbf{S})^{-1} \mathbf{S}^T (\mathbf{E}\boldsymbol{\tau} - \mathbf{M}\dot{\mathbf{S}}\boldsymbol{\eta} - \mathbf{c}),\end{aligned}\tag{3.11}$$

If we apply the nonlinear static state-feedback law

$$\boldsymbol{\tau} = (\mathbf{S}^T \mathbf{E})^{-1} (\mathbf{S}^T \mathbf{M} \mathbf{S} \mathbf{u} + \mathbf{S}^T \mathbf{M} \dot{\mathbf{S}} \boldsymbol{\eta} + \mathbf{S}^T \mathbf{c})\tag{3.12}$$

where $\mathbf{u} = (u_1, u_2)$ is the vector of new control variables, system (3.11) becomes a purely (second-order) kinematic model

$$\begin{aligned}\dot{\mathbf{q}} &= \mathbf{S}\boldsymbol{\eta} \\ \dot{\boldsymbol{\eta}} &= \mathbf{u}.\end{aligned}$$

In our case, the control law (3.12) has the explicit form

$$\begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix} = \begin{bmatrix} \frac{r}{2} \left(mu_1 + \frac{m}{d_0} \eta_2^2 + R_x \right) + \frac{rd_0}{2c} \left(\left(m + \frac{I}{d_0^2} \right) u_2 - \frac{m}{d_0} \eta_1 \eta_2 + F_y - \frac{M_r}{d_0} \right) \\ \frac{r}{2} \left(mu_1 + \frac{m}{d_0} \eta_2^2 + R_x \right) - \frac{rd_0}{2c} \left(\left(m + \frac{I}{d_0^2} \right) u_2 - \frac{m}{d_0} \eta_1 \eta_2 + F_y - \frac{M_r}{d_0} \right) \end{bmatrix}\tag{3.13}$$

and gives

$$\begin{aligned}\dot{X} &= \cos \theta \eta_1 - \sin \theta \eta_2 \\ \dot{Y} &= \cos \theta \eta_1 + \sin \theta \eta_2 \\ \dot{\theta} &= -\frac{1}{d_0} \eta_2 \\ \dot{\eta}_1 &= u_1 \\ \dot{\eta}_2 &= u_2.\end{aligned}\tag{3.14}$$

We show next that, by choosing a particular output, equations (3.14) can be fully linearized and input-output decoupled by means of a dynamic state feedback.

We choose as linearizing outputs the position of a point D placed on the x -axis at a distance d_0 from the vehicle frame origin

$$\mathbf{z} = \begin{bmatrix} X + d_0 \cos \theta \\ Y + d_0 \sin \theta \end{bmatrix},\tag{3.15}$$

and add one integrator on the input u_1 (dynamic extension)

$$\begin{aligned} u_1 &= \xi \\ \dot{\xi}_1 &= v_1 \\ u_2 &= v_2. \end{aligned} \tag{3.16}$$

where ξ is the controller state and v_1 and v_2 are the new control inputs.

By applying the standard input-output decoupling algorithm (see [15]), we differentiate equation (3.15) until the input v explicitly appears. We obtain

$$\ddot{\mathbf{z}} = \begin{bmatrix} \cos \theta & \frac{1}{d_0} \eta_1 \sin \theta \\ \sin \theta & -\frac{1}{d_0} \eta_1 \cos \theta \end{bmatrix} \mathbf{v} + \begin{bmatrix} \frac{2}{d_0} \xi \eta_2 \sin \theta - \frac{1}{d_0^2} \eta_1 \eta_2^2 \cos \theta \\ -\frac{2}{d_0} \xi \eta_2 \cos \theta - \frac{1}{d_0^2} \eta_1 \eta_2^2 \sin \theta \end{bmatrix} = \boldsymbol{\alpha}(\mathbf{q}, \boldsymbol{\eta}) \mathbf{v} + \boldsymbol{\beta}(\mathbf{q}, \boldsymbol{\eta})$$

Since

$$\det[\boldsymbol{\alpha}(\mathbf{q}, \boldsymbol{\eta})] = -\frac{1}{d_0} \eta_1,$$

we have that the decoupling matrix $\boldsymbol{\alpha}$ is nonsingular iff¹ the vehicle longitudinal velocity η_1 is different from zero. Whenever defined, the control law

$$\mathbf{v} = \boldsymbol{\alpha}^{-1}(\mathbf{q}, \boldsymbol{\eta})[\mathbf{r} - \boldsymbol{\beta}(\mathbf{q}, \boldsymbol{\eta})], \tag{3.17}$$

where \mathbf{r} is the trajectory jerk reference, yields

$$\ddot{\mathbf{z}} = \mathbf{r}, \tag{3.18}$$

i.e., two independent input-output chains of three integrators. Combining equations (3.16) and (3.17) gives the following input-output decoupling and fully linearizing dynamic controller

$$\begin{aligned} \dot{\xi}_1 &= r_1 \cos \theta + r_2 \sin \theta + \frac{1}{d_0^2} \eta_1 \eta_2^2 \\ u_1 &= \xi \\ u_2 &= \frac{d_0}{\eta_1} (r_1 \sin \theta - r_2 \cos \theta) - \frac{2}{\eta_1} \xi \eta_2. \end{aligned} \tag{3.19}$$

We note that the limitation $\eta_1 \neq 0$ does not inhibit the ability to achieve good tracking performance by means of controller (3.19), as long as the trajectory is persistent.

¹If and only if

It is easy to complete the control design for eq. (3.18) using an exponentially stabilizing state feedback for each integrator chain with input r_i . For $i = 1, 2$, we choose

$$r_i = \ddot{z}_{di} + k_a(\ddot{z}_{di} - \ddot{z}_i) + k_v(\dot{z}_{di} - \dot{z}_i) + k_p(z_{di} - z_i), \quad (3.20)$$

where the gains are such that $\lambda^3 + k_a\lambda^2 + k_v\lambda + k_p$ is a Hurwitz polynomial, $z_d(t)$ is the desired smooth reference trajectory, and z , \dot{z} and \ddot{z} can be evaluated in terms of \mathbf{q} , $\boldsymbol{\eta}$ and ξ . In order to satisfy the Routh-Hurwitz criterion, a necessary and sufficient criterion for the polynomial to be Hurwitz, the gains need to satisfy $k_a, k_v, k_p > 0$ and $k_ak_v > k_p$.

The state-feedback control law (3.21) can be seen as an output-feedback linear controller having two (realizable) minimum-phase zeros, characterized by the gain ratios k_v/k_a and k_p/k_a , and a feedforward action depending on \ddot{z}_d . The resulting control scheme has the open-loop transfer function

$$F(s) = C(s)\dot{P}(s) = (k_as^2 + k_vs + k_p) \quad (3.21)$$

3.2 Segway RMP 440

The Segway RMP 440[16] is a commercially available mobile platform, and it is at present the preferred choice of nLink AS to comprise the mobile module of their planned tilesetting robot. This mobile platform represents the simplest type of wheel configuration, as shown in figure 3.4. It has four ATV wheels that can be controlled independently, giving it the ability to turn in place. Four lithium-ion battery packs give the platform the power to carry up to 180 kg payload.

Maximum Payload:	181.4 kg
All-Terrain Payload:	90.7 kg
Dimensions:	1105 mm x 842 mm x 533 mm (LxWxH)
Weight:	120.2 kg
Top Speed:	8 m/s
Run time:	Up to 20 hours (stand by)

Table 1: Specification for the Segway RMP440 LE[7]

One of the new features of the RMP 440 over the 400 version, is the integrated auxiliary power module allowing it to power the manipulator arm.

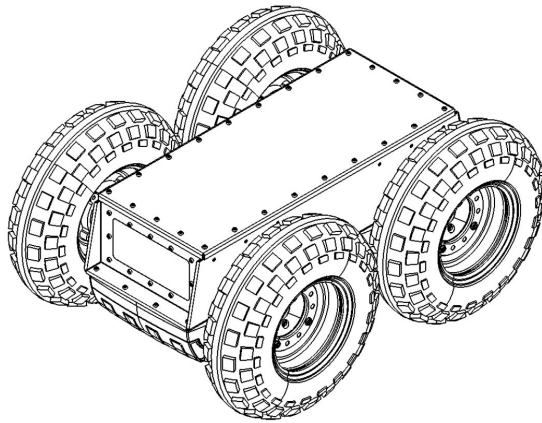


Figure 3.4: Illustration of a Segway RMP 440 mobile platform

4 Path planning for mobile wheelbased platforms

In order to make a mobile system autonomous, we need to be able to translate high-level specifications like "move from point A to point B" into a series of low-level inputs like longitudinal and angular speed. This translation is known as motion-, trajectory- or path planning, and algorithms for solving this type of problems is a fundamental part of robotics. This section will first present some basic theory (mostly from [17]) needed to formulate the motion planning problem, before outlining the two main philosophies for addressing the problem and some of the algorithms that can be used to solve it.

4.1 Operating Spaces

To define the path planning problem, we need to have a geometric representation of the world of the mobile platform. These representations are referred to as operating spaces, and are generally divided into two types: The workspace and the configuration space.

The workspace

The workspace \mathcal{W} is also sometimes referred to as simply "the world". Most problems are covered by two choices: a 2D world ($\mathcal{W} = \mathbb{R}^2$) or a 3D world ($\mathcal{W} = \mathbb{R}^3$). More complicated worlds, like for instance the surface of a sphere, are also possible, but their applications are quite limited.

This world generally contains two kinds of entities:

1. **Obstacles:** Portions of the world that are permanently occupied, for example the walls of a building.
2. **Robots:** Bodies that are modeled geometrically and are controllable via a motion plan.

Both types of entities are considered as (closed) subsets of \mathcal{W} . The *obstacle region* \mathcal{O} denotes the set of all points in \mathcal{W} that lie in one or more obstacles, $\mathcal{O} \subseteq \mathcal{W}$. Similarly, if we let \mathcal{A} refer to the robot, \mathcal{A} denotes the set of all points in \mathcal{W} that are occupied by the robot, $\mathcal{A} \subseteq \mathcal{W}$. In order to avoid collisions, the two sets cannot intersect, $\mathcal{O} \cap \mathcal{A} = \emptyset$.

The configuration space

For the purpose of planning, it is also important to define the state space. Commonly referred to as the configuration space, or \mathcal{C} -space, this

space is an important abstraction that allows numerous seemingly very different problems to be solved using the same planning algorithms.

The \mathcal{C} -space includes all the possible configurations of a physical system. If the system has n degrees of freedom, \mathcal{C} will usually be n -dimensional, with one dimension corresponding to each state.

Physical constraints like obstacles or angle restrictions on manipulator joints cause parts of \mathcal{C} to be inadmissible for our system. If we let $q \in \mathcal{C}$ denote the configuration of our vehicle \mathcal{A} , the configurations that are inadmissible can be defined as:

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} | \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\} \quad (4.1)$$

The leftover configurations are called the free space, which is defined and denoted as $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$. The path planning and obstacle avoidance problem has now been reduced to finding a path from q_{init} to q_{goal} inside \mathcal{C}_{free} .

4.2 Sampling-Based vs Combinatorial motion planning

There are two main philosophies in motion planning: Sampling-Based and Combinatorial[17].

Sampling-Based planning utilizes a sampling scheme to avoid explicit construction of the obstructed configuration space \mathcal{C}_{obs} . The sampling scheme probes the C-space using a collision detection module that the algorithm considers to be a black box, making the development of algorithms independent of the geometric models used to represent the obstructions.

Combinatorial planning does not resort to approximations. If the instances have certain convenient properties (e.g., low dimensionality, convex models), then a combinatorial algorithm may provide an elegant, practical solution. If the set of instances is too broad, then a requirement of both completeness and practical solutions may be unreasonable.

4.3 Discrete algorithms

In computing, there are several different algorithms that are able to find an ideal path in a discrete world of finite resolution. Unfortunately these will not work in our continuous world. However, as we will see in section 4.4, the use of sampling allows us to adapt the discrete algorithms into continuous methods.

4.3.1 Lifelong Planning A*

Lifelong Planning A* (LPA*) was first introduced by [18] and later discussed more thoroughly in [19] and [20], and solves the following path-planning problems: It applies to path-planning problems on known finite graphs whose edge costs increase or decrease over time. Such cost changes can also be used to model edges or vertices that are added or deleted. It was later extended to the D* Lite algorithm in section 4.3.2.

S denotes the finite set of vertices of the graph. $\text{succ}(s) \subseteq S$ denotes the set of successors of vertex $s \in S$. Similarly, $\text{pred}(s) \subseteq S$ denotes the set of predecessors of vertex $s \in S$. $0 < c(s; s') \leq \infty$ denotes the cost of moving from vertex s to vertex $s' \in \text{succ}(s)$. LPA* always determines a shortest path from a given start vertex $s_{\text{start}} \in S$ to a given goal vertex $s_{\text{goal}} \in S$, knowing both the topology of the graph and the current edge costs. We use $g^*(s)$ to denote the start distance of vertex $s \in S$, the cost of a shortest path from s_{start} to s . The start distances satisfy the following relationship:

$$g^*(s) = \begin{cases} 0 & \text{if } s = s_{\text{start}} \\ \min_{s' \in \text{pred}(s)} (g^*(s') + c(s', s)) & \text{otherwise.} \end{cases} \quad (4.2)$$

LPA* maintains two kinds of estimates of the start distance of each vertex: a g -value and an rhs -value (that is, right-hand side value, a term borrowed from [21]). The rhs -value of a vertex is based on the g -values of its predecessors and is thus potentially better informed than them. It always satisfies the following relationship:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{\text{start}} \\ \min_{s' \in \text{pred}(s)} (g(s') + c(s', s)) & \text{otherwise.} \end{cases} \quad (4.3)$$

A vertex s is called locally consistent if $g(s) = rhs(s)$, otherwise it is called locally inconsistent. If all vertices are locally consistent then all of their g -values are equal to their respective start distances, which allows one to find shortest paths from the start vertex to any vertex. However, LPA* does not make every vertex locally consistent after some of the edge costs have changed. Instead, it shares with A* the fact that it uses nonnegative and consistent heuristics $h(s, s_{\text{goal}})$ that approximate the goal distances of the vertices s to focus its search and updates only the g -values that are relevant for computing a shortest path from the start to the goal vertex. Consistent heuristics obey the triangle inequality $h(s_{\text{goal}}, s_{\text{goal}}) = 0$ and $h(s, s_{\text{goal}}) \leq c(s, s') + h(s', s_{\text{goal}})$ for all vertices $s \in S$ and $s' \in \text{succ}(s)$ with $s \neq s_{\text{goal}}$.

LPA* maintains a priority queue that always contains exactly the locally inconsistent vertices. These are the vertices whose g-values potentially needs to change to make them locally consistent. The key of vertex in the priority queue is a vector with two components:

$$k(s) = \begin{bmatrix} \min(g(s), rhs(s)) + h(s, s_{goal}) \\ \min(g(s), rhs(s)) \end{bmatrix} \quad (4.4)$$

Keys are compared according to a lexicographic ordering, meaning they are sorted first according to the first element, and any ties are then sorted according to the second element.

The algorithm

The pseudocode for LPA* is shown in figure 4.1, and the numbers in curly braces in the following explanation refers to the line numbers. Its main function Main() first calls Initialize() to initialize the search problem {17}. Initialize() sets the initial g-values of all vertices to infinity and sets their rhs-values according to equation (4.3) {03–04}. Thus, initially the start vertex is the only locally inconsistent vertex and is inserted into the otherwise empty priority queue {05}. Note that, in an actual implementation, Initialize() only needs to initialize a vertex when it encounters it during the search and thus does not need to initialize all vertices up front.

After calling Initialize(), Main() calls ComputeShortestPath() to find a shortest path from the start to the goal vertex. ComputeShortestPath() repeatedly recalculates the g-values of locally inconsistent vertices (“expands the vertices”) in nondecreasing order of their keys {10–16}.

A locally inconsistent vertex s is called locally overconsistent iff $g(s) > rhs(s)$. When ComputeShortestPath() expands a locally overconsistent vertex {12–13}, then it holds that $rhs(s) = g^*(s)$, which implies that $k(s) = [f(s); g^*(s)]$, where $f(s) = g^*(s) + h(s, s_{goal})$. During the expansion of the vertex, ComputeShortestPath() sets the g-value of the vertex to its rhs-value and thus its start distance {12}, which is the desired value and also makes the vertex locally consistent. Its g-value then no longer changes until ComputeShortestPath() terminates.

A locally inconsistent vertex is called locally underconsistent iff $g(s) < rhs(s)$. When ComputeShortestPath() expands a locally underconsistent vertex {15–16}, then it simply sets the g-value of the vertex to infinity {15}. This makes the vertex either locally consistent or overconsistent. If the expanded vertex was locally overconsistent, then the change of its g-value can affect the local consistency of its successors {13}.

The pseudocode uses the following functions to manage the priority queue: $U.\text{TopKey}()$ returns the smallest priority of all vertices in priority queue U . (If U is empty, then $U.\text{TopKey}()$ returns $[\infty; \infty]$.) $U.\text{Pop}()$ deletes the vertex with the smallest priority in priority queue U and returns the vertex. $U.\text{Insert}(s; k)$ inserts vertex s into priority queue U with priority k . Finally, $U.\text{Remove}(s)$ removes vertex s from priority queue U .

procedure CalculateKey(s)

1: **return** $[\min(g(s), rhs(s)) + h(s, s_{goal}); \min(g(s), rhs(s))];$

procedure Initialize()

2: $U = \emptyset;$

3: **for all** $s \in S, rhs(s) = g(s) = \infty;$

4: $rhs(s_{start}) = 0;$

5: $U.\text{Insert}(s_{start}; [h(s_{start}); 0]);$

procedure UpdateVertex(u)

if $(u \neq s_{start}), rhs(u) = \min_{s' \in pred(u)} (g(s') + c(s'; u));$

if $(u \in U), U.\text{Remove}(u);$

if $(g(u) \neq rhs(u)), U.\text{Insert}(u, \text{CalculateKey}(u));$

procedure ComputeShortestPath()

9: **while** $(U.\text{TopKey}() < \text{CalculateKey}(s_{goal})$ **or** $rhs(s_{goal}) \neq g(s_{goal}))$
do

10: $u = U.\text{Pop}();$

11: **if** $(g(u) > rhs(u))$ **then**

12: $g(u) = rhs(u);$

13: **for all** $s \in succ(u), \text{UpdateVertex}(s);$

14: **else**

15: $g(u) = \infty;$

16: **for all** $s \in succ(u) \cup u, \text{UpdateVertex}(s);$

procedure Main()

17: $\text{Initialize}();$

18: **loop**

19: $\text{ComputeShortestPath}();$

20: Wait for changes in edge costs;

21: **for all** directed edges $(u; v)$ with changed edge costs **do**

22: Update the edge cost $c(u; v);$

23: $\text{UpdateVertex}(v);$

Figure 4.1: Lifelong Planning A* algorithm

Similarly, if the expanded vertex was locally underconsistent, then it and its successors can be affected {16}.

To maintain equations (4.3) and (4.4), as well as the priority queue, `ComputeShortestPath()` therefore updates the rhs-values of these vertices, checks their local consistency, and adds them to or removes them from the priority queue as needed {06–08}. LPA^* expands vertices until the goal vertex is locally consistent and the key of the vertex to expand next is no smaller than the key of the goal vertex.

If $g(s_{goal}) = \infty$ after the search, then there is no finite-cost path from the start to the goal vertex. Otherwise, one can find a shortest path from the start to the goal vertex as follows: One always moves from the current vertex s , starting at the goal vertex, to any predecessor s' that minimizes $g(s') + c(s', s)$ until the start vertex is reached (ties can be broken arbitrarily). This way, one traverses a shortest path from the start to the goal vertex backward.

After calling `ComputeShortestPath()`, `Main()` waits for changes in edge costs {20}. To maintain equations (4.3) and (4.4) if some edge costs have changed, it calls `UpdateVertex()` {23} to update the rhs-values and keys of the vertices potentially affected by the changed edge costs as well as their membership in the priority queue if they become locally consistent or inconsistent, and finally recalculates a shortest path {19} by calling `ComputeShortestPath()` again and iterates.

4.3.2 D* Lite

D* Lite, also known as Focused Dynamic A* Lite, is described in [20] and builds on the LPA^* algorithm from section 4.3.1. As the name suggests, it was developed as a simpler version of D*. Nonetheless, it has been demonstrated to be at least as efficient as D*.

D* Lite repeatedly determines shortest paths between the current vertex of the robot and the goal vertex as the edge costs of a graph change while the robot moves toward the goal vertex. D* Lite does not make any assumptions about how the edge costs change, whether they go up or down, whether they change close to the current vertex of the robot or far away from it, or whether they change in the world or only because the knowledge of the robot changes.

The first version of D* Lite was essentially an inverted version of LPA^* , searching from the goal vertex to the start vertex. The heuristics $h(s_{start}, s)$ now describe an estimate of the start distance of the vertex, and as the robot moves, these heuristics will naturally change. Because

of this, the first version of D* Lite had to update all of the keys in the entire priority queue every time a change in edge costs were detected. This had the disadvantage of repeatedly reordering the priority queue, which can be expensive since the priority queue often contains a large number of vertices.

The second version of D* Lite, shown in figure 4.2, uses a search method derived from D* to avoid having to reorder the priority queue. The heuristics $H(s, s')$ need to be nonnegative and forward-backward consistent, that is, obey $h(s, s'') \leq h(s, s') + h(s', s'')$ for all vertices $s, s', s'' \in S$. They also need to be admissible no matter what the goal vertex is, that is, obey $h(s, s') \leq c^*(s, s')$ for all vertices $s, s' \in S$, where $c^*(s, s')$ denotes the cost of a shortest path from vertex $s \in S$ to vertex $s' \in S$.

The second version of D* Lite uses keys that are lower bounds on the keys that the first version of D* Lite uses for the corresponding vertices. It initializes them in the same way as the first version of D* Lite. After the robot has moved from vertex s to some vertex s' where it detects changes in edge costs, the first component of the keys can have decreased by at most $h(s, s')$. The second component does not depend on the heuristics and thus remains unchanged. Thus, in order to maintain lower bounds, D* Lite needs to subtract $h(s, s')$ from the first component of the keys of all vertices in the priority queue. However, since $h(s, s')$ is the same for all vertices in the priority queue, the order of the vertices in the priority queue does not change if the subtraction is not performed. Then, when new keys are computed, their first components are by $h(s, s')$ too small relative to the keys in the priority queue. Thus, $h(s, s')$ has to be added to their first components.

If the robot moves again and then detects cost changes again, then the constants need to get added up. We do this in the variable k_m (that is, key modifier) {30} (curly braces indicate line numbers in figure 4.2). Thus, whenever new keys are computed, the variable k_m has to be added to their first components, as done in {1}. Then, the order of the vertices in the priority queue does not change after the robot moves and the priority queue does not need to get reordered. The keys, on the other hand, are always lower bounds on the corresponding keys of the first version of D* Lite after the first component of the keys of the first version of D* Lite has been increased by the current value of k_m , that is, lower bounds on the values calculated by CalcKey() {1}.

```

procedure CalcKey( $s$ )
1: return  $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$ ;

procedure Initialize()
2:  $U = \emptyset$ ;
3:  $k_m = 0$ ;
4: for all  $s \in S, rhs(s) = g(s) = \infty$ ;
5:  $rhs(s_{goal}) = 0$ ;
6:  $U.Insert(s_{goal}, CalcKey(s_{goal}))$ ;

procedure UpdateVertex( $u$ )
7: if ( $u \neq s_{goal}$ ),  $rhs(u) = \min_{s' \in pred(u)} (c(u; s') + g(s'))$ ;
8: if ( $u \in U$ ),  $U.Remove(u)$ ;
9: if ( $g(u) \neq rhs(u)$ ),  $U.Insert(u, CalcKey(u))$ ;

procedure ComputeShortestPath()
10: while ( $U.TopKey() < CalcKey(s_{start})$  or  $rhs(s_{start}) \neq g(s_{start})$ ) do
11:    $k_{old} = U.TopKey()$ ;
12:    $u = U.Pop()$ ;
13:   if ( $k_{old} < CalcKey(u)$ ) then
14:      $U.Insert(u, CalcKey(u))$ ;
15:   else if ( $g(u) > rhs(u)$ ) then
16:      $g(u) = rhs(u)$ ;
17:     for all  $s \in pred(u)$ ,  $UpdateVertex(s)$ ;
18:   else
19:      $g(u) = \infty$ ;
20:     for all  $s \in pred(u) \cup u$ ,  $UpdateVertex(s)$ ;

procedure Main()
21:  $s_{last} = s_{start}$ ;
22: Initialize();
23: ComputeShortestPath();
24: while ( $s_{start} \neq s_{goal}$ ) do
25:   /* if ( $g(s_{start}) = \infty$ ) then there is no known path */
26:    $s_{start} = \arg \min_{s' \in succ(s_{start})} (c(s_{start}, s') + g(s'))$ ;
27:   Move to  $s_{start}$ ;
28:   Scan graph for changes in edge costs;
29:   if any edge costs changed then
30:      $k_m = k_m + h(s_{last}, s_{start})$ ;
31:      $s_{last} = s_{start}$ ;
32:     for all directed edges  $(u; v)$  with changed edge costs do
33:       Update the edge cost  $c(u; v)$ ;
34:       UpdateVertex( $u$ );
35:     ComputeShortestPath();

```

Figure 4.2: D* Lite algorithm

We exploit this property by changing `ComputeShortestPath()` as follows: After `ComputeShortestPath()` has removed a vertex u with the smallest key $k_{old} = U.TopKey()$ from the priority queue $\{12\}$, it now uses `CalcKey()` to compute the key that it should have had. If $k_{old} < \text{CalcKey}(u)$ then it reinserts the removed vertex with the key calculated by `CalcKey()` into the priority queue $\{13-14\}$. Thus, it remains true that the keys of all vertices in the priority queue are lower bounds on the corresponding keys of the first version of D* Lite after the first components of the keys of the first version of D* Lite have been increased by the current value of k_m . If $k_{old} \leq \text{CalcKey}(u)$, then it holds that $k_{old} = \text{CalcKey}(u)$ since k_{old} was a lower bound on the value returned by `CalcKey()`. In this case, `ComputeShortestPath()` performs the same operations for vertex u as `ComputeShortestPath()` of the first version of D* Lite.

4.4 Sampling-Based algorithms

When sampling the continuous world to discretize it, there are two typical approaches: Input sampling and output sampling. We will look at an example of each case, but first we need to look at some of the pros and cons of each approach:

Input sampling is done by sampling the available inputs of the system, and then calculating the corresponding outputs given a certain timestep. One of the benefits of this approach is that calculating the outputs is simply a matter of integrating a system model. If different inputs over some timesteps result in outputs that are nearly identical and the workspace contains a local minimum for the search algorithm, we run the risk of creating an infinite amount of nodes in a small area around the local minimum. In section 4.4.3, we will see examples of this, as well as one approach to avoid the problem.

Output sampling is done by sampling the outputs of the system, and calculate the inputs required to get from one sample to the next. As [22] points out, there are two primary disadvantages to using output (i.e., configuration space) sampling, a common approach in robotics. The first limitation lies within the vertex (sample) selection method, where the algorithm must determine the most ideal vertex to expand. This selection is typically made based on the proximity of vertices in the graph to a sampled output point, and involves a potentially costly nearest neighbor search. The local planning method (generating a path between two states) presents the second, and perhaps more troublesome problem, which is determining an input that connects a newly sampled node to the current

node. This problem is essentially a two-point boundary value problem (BVP) that connects one output or state to another. There is no guarantee that such an input exists. Also, for systems with complex dynamics, the search itself can be computationally expensive, which leads to a computationally inefficient planner.

4.4.1 Sampling techniques: Halton points

When sampling a space, we want to choose samples randomly, but at the same time the purpose is to cover the space such that the samples are uniformly distributed and minimizes gaps and clusters. Choosing samples truly randomly (or even pseudo-randomly) for a limited set of samples, will always run the risk of creating clusters of samples with gaps between them.

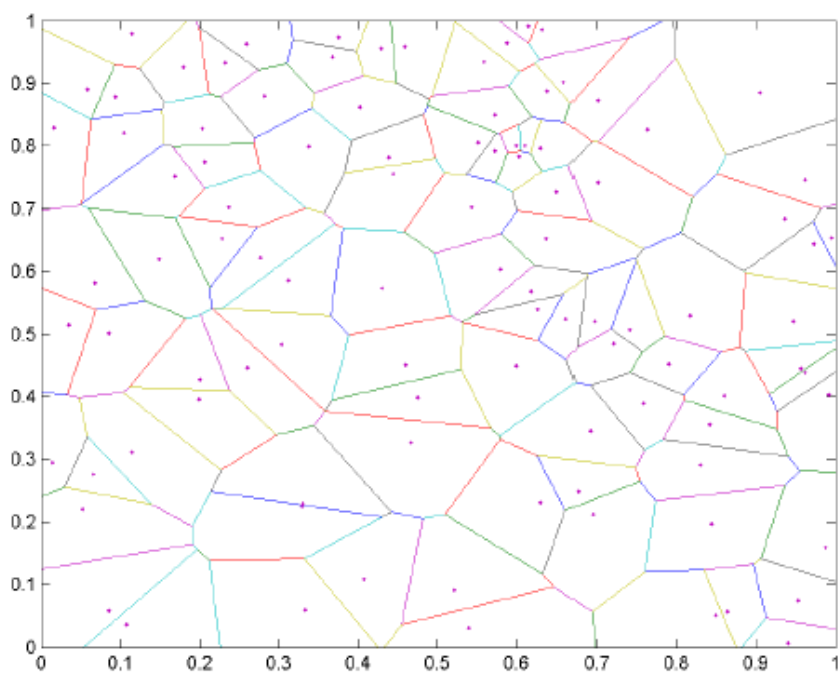
A Halton sequence[23] is a quasi-random number sequence that appears random, but has the added benefit of maintaining good uniformity. Figure 4.3 shows a set of coordinates generated by the Matlab *twister* pseudorandom number generator (4.3a), and by the Halton sequence (4.3b). The Voronoi diagrams² for the points are drawn to better visualize the distribution of the points. As seen in the figures, the area of the regions in (b) are more even than those in (a), clearly suggesting that the coordinates generated by the Halton sequence are more uniformly distributed than those generated by the ordinary randomized method in Matlab[24].

4.4.2 Output Sampling: A* assisted Rapidly-expanding Random Tree (RRT)

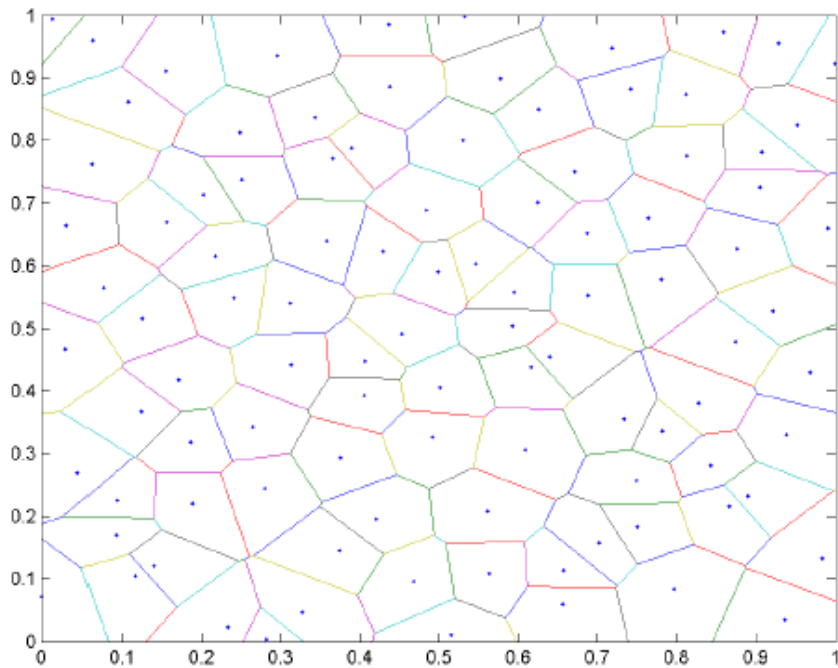
The Rapidly-Exploring Random Tree method (RRT), was introduced by LaValle in 1998[25]. It is a well known motion planning method that can plan a path while taking the dynamics of the vehicle into account. As a randomized method, the RRT can explore a space of possible solutions several orders of magnitude faster than a complete method. The solution provided by the RRT is sub-optimal, but will in most cases prove sufficient.

Several different papers have proposed different methods of biasing the randomization with the intention of guiding the search and improving runtime, for instance [26]. The following example, developed in [24], utilizes A* heuristics in order to increase the density of samples in areas likely to contain the optimal path.

²For a set of points S in a plane, the Voronoi diagram is a partition of the plane associating a region $V(p)$ with each point $p \in S$. The regions are set up so that all points in a region $V(p)$ are closer to p than any other point in S



(a)



(b)

Figure 4.3: The set of coordinates generated by the Matlab *twister* pseudorandom number generator (a), and by the Halton sequence (b)[24].

The RRT algorithm

The RRT method will as its name indicates, generate a tree structure during its execution. This tree starts at the state of our vehicle, and iteration after iteration explores larger portions of the configuration space \mathcal{C} of the vehicle. An important property of the RRT method is that it not only plans the workspace-coordinates of our vehicle, but all its states, i.e., coordinates, heading and velocities. The result is a path through the \mathcal{C} -space of the vehicle, or more specifically the admissible part of the \mathcal{C} -space, namely \mathcal{C}_{free} . If a path has been generated by the RRT method, we thus know it is a feasible path, at least if no changes have occurred in the environment after the path was generated.

The RRT algorithm is relatively simple to implement. Given the initial state of our vehicle x_{init} and a requested goal configuration x_{goal} , we get the algorithm in figure 4.4.

```

1:  $\mathcal{T} \leftarrow \text{TREE}(\{x_{init}, 0\})$ 
2: if CAN_CONNECT( $x_{init}, x_{goal}$ ) then
3:    $\mathcal{T} \leftarrow \{\text{CONNECT\_STATE}(), \text{CONNECT\_INPUT}()\}$  as child of  $x_{init}$ 

4:   goal found
5:   for  $k \leftarrow 1 \dots N$  do {The main loop}
6:      $\mathbf{m}_c = \text{CANDIDATE\_MILESTONE}()$ 
7:     if  $\mathbf{m}_c \in \mathcal{C}_{free}$  then
8:        $x_{near} = \text{NEAREST\_NEIGHBOUR}(\mathbf{m}_c, \mathcal{T})$ 
9:       if CAN_CONNECT( $x_{near}, \mathbf{m}_c$ ) then
10:         $u \leftarrow \text{CONNECT\_INPUT}()$ 
11:         $x_{next} \leftarrow \text{CONNECT\_STATE}()$ 
12:         $\mathcal{T} \leftarrow \{x_{next}, u\}$  as child of  $x_{near}$ 
13:        if CAN_CONNECT( $x_{next}, x_{goal}$ ) then
14:           $\mathcal{T} \leftarrow \{\text{CONNECT\_STATE}(), \text{CONNECT\_INPUT}()\}$  child of
             $x_{next}$ 
15:          goal found
16:   if goal found then
17:     return cheapest path from  $x_{init}$  to  $x_{goal}$  in  $\mathcal{T}$ 
18:   else
19:     return cheapest path from  $x_{init}$  taking us closest to  $x_{goal}$ 

```

Figure 4.4: Rapidly-Exploring Random Tree (RRT) algorithm

The concept of milestones is the heart of the algorithm. At each iteration, a candidate milestone \mathbf{m}_c is chosen. It represents a potentially

unexplored configuration of the controlled vehicle, for instance an unvisited location in the environment. The algorithm then tries to predict whether it can get the vehicle to \mathbf{m}_c from any of the configurations previously explored, i.e. the configurations currently in the tree. If the algorithm manages to do this, the candidate is added to the tree as an explored milestone/node. An attempt is then made to connect this new node to the goal configuration x_{goal} .

CAN_CONNECT(x, \mathbf{m}_c) is the function that does most of the work. It uses a local motion planner to try to connect the vehicle state/configuration x with the new candidate milestone \mathbf{m}_c . For the local planner, a compromise has to be made between its accuracy and computational complexity. An accurate planner will more often be able to connect x with \mathbf{m}_c , while a fast planner will be able to make more attempts with different candidate milestones in the same amount of time.

A vehicle state is connected to a milestone by integrating the vehicle from x while applying a series of inputs u . If the vehicle comes within a predefined distance of \mathbf{m}_c within a given time and without exiting \mathcal{C}_{free} , then the connection was successful, and the function returns true. CONNECT_INPUT() can later be called to get the series of inputs u that made this possible, and CONNECT_STATE() can be called to get the final state, which by definition must be in the vicinity of \mathbf{m}_c .

CANDIDATE_MILESTONE() returns a random state vector within the state space of our vehicle. There are many ways of generating these random states, and the way they are generated can have a serious impact on the performance of the algorithm. Using a uniform distribution makes sure all parts of \mathcal{C}_{free} are explored, while selecting a distribution biased towards a smaller area may speed up the algorithm significantly.

NEAREST_NEIGHBOUR(\mathbf{m}, \mathcal{T}) returns the node in \mathcal{T} whose state vector is closest to the milestone \mathbf{m} by some metric. Using the Euclidean distance metric is common since it ensures that the algorithm tries to connect nodes that are geometrically close, creating a short path through \mathcal{C}_{free} .

N is the maximum number of times to run the algorithm. Logic can be implemented to end the algorithm prematurely if a satisfying path to x_{goal} has been found.

The RRT method is not complete. Even though a solution to the planning problem exists, it cannot be guaranteed that the algorithm will find one in a limited time interval. This is however not as catastrophic as it might sound. The most obvious solution to the problem is to re-use the previously-generated path, which will likely still be valid. Another

way to solve the problem is to use one of the partial solutions existing in \mathcal{T} . All of these branches are feasible, even though they don't end up at the goal. By selecting a branch that makes progress towards the goal, the vehicle has something to do until the completion of the next iteration of the algorithm.

The milestone generator

At the heart of the RRT algorithm is the random state generator. Represented by the function `CANDIDATE_MILESTONE()` in figure 4.4, it is responsible for generating candidate milestones, or states, which the local planner of the RRT algorithm then tries to reach.

The distribution of the candidate milestones, and especially their spatial coordinates, can have a serious impact on the performance of the algorithm. It has been claimed that the distribution given by the Halton sequence in section 4.4.1 gives better results than ordinary randomized and biased distributions. The reason for this is probably that the near-uniform distribution allows the algorithm to quickly explore the entire search space before creating major branches impeding path optimality.

When trying to find a route through an environment, the search space for the RRT algorithm can often be very large. Using only the Halton generator, equal attention is paid to all parts of the search space. This is good because it allows the algorithm to find alternative and difficult routes through the environment, but it also has a downside. As the allowed computation time is limited, the uniform distribution often leads to less optimal paths since the attention of the algorithm is scattered over the entire search space.

What if we could direct the attention of the algorithm towards the area most likely to hold the most optimal path? The algorithm would then spend more time in this area, and perhaps come up with a better solution to the path planning problem.

As seen in [27], the A^* algorithm (a non-incremental version predecessor of the LPA^* algorithm presented in section 4.3.1) consistently found more optimal paths through the environment than the RRT algorithm. The idea here is that the optimal path through the environment probably lies around the route generated by the A^* algorithm, although this cannot be guaranteed since the A^* algorithm disregards the dynamics of the vehicle. We need to direct the attention of the RRT algorithm towards this route, and this can easily be accomplished by increasing the density of candidate milestones along the A^* route.

Before each run of the RRT algorithm, the A^* algorithm is run to find

the A* path through the environment. This generates a set of coordinates a_1, a_2, \dots, a_N where $a \in \mathbb{R}^2$. A milestone taken from the A* route is now generated as follows:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = a_q + R \cdot s, \quad (4.5)$$

where $q \in \{1, 2, \dots, N\}$ and $s \in \{x \in \mathbb{R}^2 \mid \|x\|_1 \leq R\}$ are randomly chosen.

This strategy will produce candidate milestones in close vicinity to the A* optimal path, but we also want the rest of the search space to be explored. Every time a candidate milestone is requested by the algorithm, the `CANDIDATE_MILESTONE()` function has to determine whether to respond with a candidate generated by the Halton sequence or one chosen along the A* route. In this implementation, the choice is done by chance:

$$generator = \begin{cases} Halton, & \text{if } \mathcal{X} > f \\ A^*, & \text{if } \mathcal{X} \leq f, \end{cases} \quad (4.6)$$

where $\mathcal{X} \in [0, 1)$ is randomly chosen. This means that an A* milestone is generated with a probability of f . If for instance $f = 0.1$, an A* milestone will be generated 10% of the time on average. Care has to be taken when choosing a value for f , as too high a value will prohibit exploration of the rest of the search space, possibly resulting in no path being found. Choosing f too small however, reduces the effect of the A* assistance to the algorithm. An example is shown in figure 4.5.

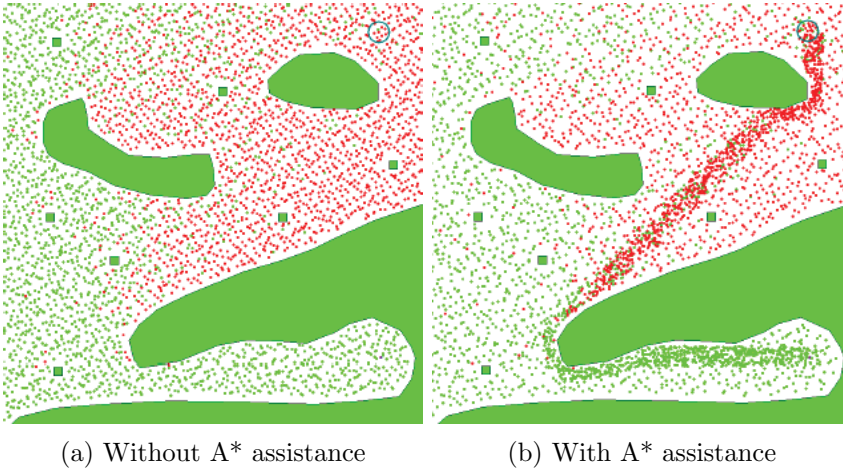


Figure 4.5: The distribution of RRT candidate milestones[24].

4.4.3 Input Sampling: Sampling Based Model Predictive Control

SBMPC is a method presented in [22] that can address Nonlinear Model Predictive Control (NMPC) problems. It effectively reduces the problem size of MPC by sampling the inputs of the system. The method also replaces the traditional MPC optimization phase with LPA* (see section 4.3.1) that can replan quickly (i.e. it is incremental). The objective is to determine a sequence of control inputs that cause the system to achieve a given set point while solving the optimization problem.

SBMPC optimization differs from the typical application of A* in that the nodes are configurations in the output space and an edge is the control input that links two output nodes. SBMPC retains the computational efficiency and has the convergence properties of LPA*[19] under two conditions:

1. The graph must be constructed such that a finite number of vertices exists within a finite state space
2. The implicit state grid (this will be defined shortly) must represent the full state space and not simply the states used in the planning objective

At the same time, SPMPC avoids some of the computational bottlenecks associated with sampling-based motion planners. In particular, the Local Planning Method (generating a path between two nodes) of sampling-based motion planners is simplified from what is effectively a two-point boundary value problem by sampling in the input space and integrating to determine the next output node. Although most sampling-based planners use proximity to a random or heuristically biased point as their vertex selection method, the proposed method bases the vertex selection on an A* criterion. This involves computing the cost to the node and an optimistic (i.e., lower bound) prediction of the cost from the node to the goal; this eliminates the need for a potentially costly nearest neighbor search while promoting advancement towards the goal.

The Implicit State Grid

Although extension of several of the existing sampling-based paradigms can lead to input sampling algorithms like SBMPC [17], input sampling has not been used in most planning research. This is most likely due to the fact that input sampling is seen as being inefficient because it can result in highly dense samples in the configuration space since input sampling

does not inherently lead to a uniformly discretized state space, such as a uniform grid.

This problem is especially evident when encountering a local minimum problem associated with the A* algorithm, which can occur when planning in the presence of a large obstacle while the goal is on the other side of the obstacle. This situation is considered in depth for discretized 2D path planning in the work of [28], which discusses that the A* algorithm must explore all the states in the neighborhood of the local minimum, shown as the shaded region of figure 4.6a, before progressing to the final solution. The issue that this presents to input sampling methods is that the number of states within the local minimum is infinite because of the lack of a discretized state space.

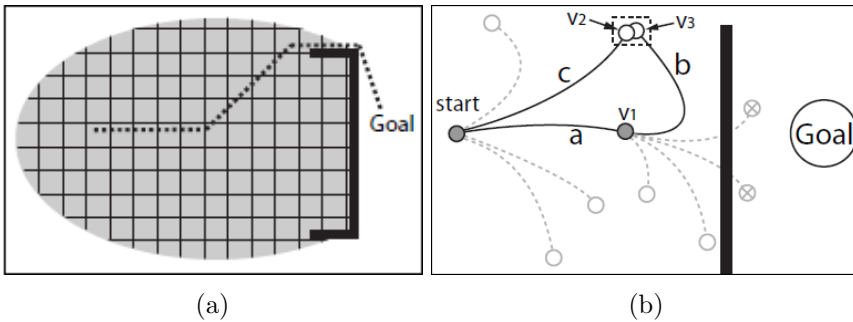


Figure 4.6: Illustration of the necessity of an implicit state grid[22].

The secondary effect resulting from the nature of input sampling as well as the lack of a grid, is that the likelihood of two states being identical is extremely small. All A*-like algorithms utilize Bellman's optimality principle to improve the path to a particular state by updating the paths through that state when a lower cost alternative is found. This feature is essential to the proper functioning of the algorithm and requires a mechanism to identify when states are close enough to be considered the same. The scenario presented in figure 4.6b is a situation for which the lack of this mechanism would generate an inefficient path. In this situation, vertex v_1 is selected for expansion after which the lowest cost vertex is v_3 . The implicit state grid then recognizes that v_2 and v_3 are close enough to be considered the same and updates the path to their grid cell to be path c since $c < a + b$.

The concept of an implicit state grid is introduced as a solution to both of these problems. The implicit grid ensures that the graph generated by the SBMPC algorithm is constructed such that only one active state ex-

ists in each grid cell, therefore limiting the number of vertices that can exist within any finite region of the state space. It also allows for the efficient storage of potentially infinite grids by only storing the grid cells that contain vertices, which is increasingly important for higher dimensional problems. The resolution of the grid is a significant factor in determining the performance of the algorithm with more fine grids in general requiring more computation time, due to the increased number of states, with the benefit being a more optimal solution. The resolution therefore is a useful tool that enables SBMPC to effectively make the trade off between solution quality and computational performance. However, it must be stated that increasing the resolution significantly beyond the accuracy of the prediction could result in the generation of infeasible solutions.

The algorithm

SBMPC operates on a dynamic directed graph G which is a set of all vertices and edges currently in the graph. $\text{succ}(v)$ denotes the set of successors (children) of vertex $v \in G$ while $\text{pred}(v)$ denotes the set of all predecessors (parents) of vertex $v \in G$. The cost of moving from vertex v to vertex $v' \in \text{succ}(v)$ is denoted by $c(v; v')$, where $0 < c(v; v') < \infty$. The optimization component of SBMPC is called Sampling-Based Model Predictive Optimization (SBMPO) and is an algorithm that determines the shortest path from a given start vertex $v_{\text{start}} \in G$ and a given goal vertex $v_{\text{goal}} \in G$. SBMPO uses heuristics along with the start distance estimates to rank the *OPEN* list, a priority queue containing the locally inconsistent vertices and thus all the vertices that need to be updated in order to make them locally consistent.

As we can see in figure 4.7, the *Main()* and *SBMPO()* procedures are essentially the same as the *LPA** algorithm in section 4.3.1, the difference being that *LPA** works with a predefined graph, while SBMPC builds its own graph simultaneously. The main algorithm follows the general structure of MPC where SBMPO repeatedly computes the shortest path between the current state x_{current} and the goal state x_{goal} . After a single path is generated x_{current} is updated to reflect the implementation of the first computer control input and the graph G is updated to reflect any system changes.

procedure SBMPO()

```

1: while (OPEN.TopKey()  $< v_{goal}.key$  or  $v_{goal}.rhs \neq v_{goal}.g$  do
2:    $v_{best} \leftarrow \text{OPEN.TopKey}()$ 
3:   Generate_Neighbours( $v_{best}$ , B)
4:   if  $v_{best}.g > v_{best}.rhs$  then
5:      $v_{best}.g = v_{best}.rhs$ 
6:     for all  $s \in \text{succ}(v_{best})$  do
7:       Update the vertex,  $s$ 
8:   else
9:      $v_{best} \leftarrow \infty$ 
10:    for all  $s \in \text{succ}(v_{best}) \cup v_{best}$  do
11:      Update the vertex,  $s$ 

```

procedure Generate_Neighbours (Vertex v , Branching B)

```

1: for  $i=0$  to B do
2:   Generate sampled input,  $u \in \mathbb{R}^u \cap \mathbf{U}_{free}$ 
3:   for  $t = t1 : dt_{integ} : t2$  do
4:     Evaluate model:  $x(t) = f(v.x, u)$ 
5:     if  $x(t) \notin \mathbf{X}_{free}(t)$  then
6:       Break and jump to next iteration of  $i$ 
7:      $x_{new} = x(t2)$ 
8:     if  $x_{new} \in \text{STATE\_GRID}$  and  $x_{new} \in \mathbf{X}_{free}$  then
9:       Add  $\text{Edge}(v.x, x_{new})$  to graph, G
10:    else if  $x_{new} \in \mathbf{X}_{free}$  then
11:      Add  $\text{Vertex}(x_{new})$  to graph, G
12:      Add  $\text{Edge}(v.x, x_{new})$  to graph, G

```

procedure Main()

```

1:  $x_{current} \leftarrow \text{start}$ 
2: repeat
3:   SBMPO()
4:   Update system state,  $x_{current}$ 
5:   Update graph, g
6: until the goal state is achieved

```

Figure 4.7: Sampling-Based MPC algorithm

The neighbour generation method generates a set of pseudorandom samples in the input space that are then used to predict a set of paths to a new set of states with the $x_{current}$ being the initial condition. The branching factor B , determines the number of paths that will be generated. The path is represented by a sequence of states $x(t)$ for $t = t_1, t_1 + \Delta t, \dots, t_2$, where Δt is the model step size. The set of states that do not violate any state or obstacle constraints is called \mathbf{X}_{free} . If $x(t) \in \mathbf{X}_{free}$ then the new vertex x_{new} and the connecting edge can be added to the graph. If $x_{new} \in STATE_GRID$ then the vertex currently exists in the graph and only the new path to get to the existing vertex needs to be added.

One of the downfalls of SBMPC is that it uses a nonstandard optimization method. Therefore, it is not likely that SBMPC can be easily implemented using readily available optimization tools and involves a great deal of programming effort.

5 The optimization problem

After reviewing the literature presented in section 4, it was decided to base the path planning module in section 7 on the SBMPO algorithm from section 4.4.3. The calculation time of the algorithm is not a primary factor in the efficiency, the only demand being that it is fast enough that the vehicle does not have to stop and wait for the algorithm to finish. This, combined with the number of dimensions discourages a combinatorial approach. The reason for choosing the SBMPO-algorithm for the path planning module was primarily the very positive results in [29] with regards to planning of energy efficient paths. The use of input sampling rather than the more common output sampling was deemed an interesting area of research. The use of a state grid to control the amount of nodes generated, supposedly eliminating the drawback of input sampling, also made it seem like a more viable option than output sampling, since output sampling would include some costly search procedures. [29] also proposed numerous extensions which seemed interesting to explore.

Before we can run our algorithm to find the optimal path, we first need to define what we want to optimize. The primary goal for the tiling robot will be to work as efficiently as possible. A contractor using the system wants the job done as quickly as possible while still maintaining the necessary quality. Being able to do the job quicker than the competing contractors will not only allow them to ask a higher price for the service, but also decrease the number of working hours they need to pay their workers for the given job, thus further increasing their profit margin.

There are two major factors in the path choice that affects the efficiency of the robot. The most obvious factor is of course how quickly it can get from A to B. But because the robot will be battery powered, energy consumption may also become a critical factor. As will be shown in section 5.3.1, sharp turns are far more energy consuming than smooth curves for a skid-steered platform. If the robot constantly needs to recharge or switch the depleted batteries for fresh ones, the time saved by making many sharp turns is lost, and the performance of the system declines.

5.1 Optimization with respect to distance(Shortest path)

The most common optimization goal in path planning is probably to find the shortest path. In the case of the tiling robot, the traveling distance in itself is not really interesting. This optimization criterium is primarily discussed in this thesis because it is such a standard optimization goal,

and serves as a benchmark with which to compare the other two criteria.

In order to implement Shortest Path in the SBMPO-algorithm from section 4.4.3, the cost of the arcs needs to be defined as their length, and the heuristic function is defined as the straight line distance from the node to the goal:

$$\text{cost}(\text{arc}) = D_{\text{arc}}, \quad \text{heuristic}(\text{node}) = D_{\text{node} \rightarrow \text{goal}} \quad (5.1)$$

5.2 Optimization with respect to time(Quickest path)

When assuming constant speed like in [29], optimizing with respect to time and optimizing with respect to distance are identical. When adding the option of variable speed however, differences emerge.

In order to implement Quickest Path in the SBMPO-algorithm from section 4.4.3, the cost of the arcs needs to be defined as the time taken to traverse them, and the heuristic function is defined as the time it takes to drive the straight line distance from the node to the goal at top speed.

$$\text{cost}(\text{arc}) = t_{\text{arc}}, \quad \text{heuristic}(\text{node}) = t_{\text{node} \rightarrow \text{goal}} \quad (5.2)$$

5.3 Optimization with respect to energy consumption(Easiest path)

[29] showed the benefit of running an optimization based on energy consumption. While highly beneficial, this criterium is somewhat more complicated to implement, and it requires some more information about the system.

In [29], Easiest Path was implemented in the SBMPO-algorithm by defining the cost of the arcs as the energy required to traverse them, and the heuristic function as the energy it takes to drive the straight line distance from the node to the goal:

$$\text{cost}(\text{arc}) = P(r, v)t_{\text{arc}}, \quad \text{heuristic}(\text{node}) = P(\infty, v)t_{\text{node} \rightarrow \text{goal}} \quad (5.3)$$

where $P(r, v)$ is the power it takes to drive in a curve with radius r and speed v . We will derive an expression for this function in section 5.3.1

This heuristic was found to be a large underestimate of the actual cost of traveling from the given node to the goal, due to the high amount of energy required to change the orientation of the vehicle. In addition to the heuristic function in [29], we therefore add an extra term to the heuristic function, corresponding to the energy it would require to turn the vehicle to face the goal directly while standing still:

$$\text{cost}(\text{arc}) = P(r, v)t_{\text{arc}}, \text{heuristic}(\text{node}) = P(\infty, v)t_{\text{node} \rightarrow \text{goal}} + W(\tilde{\theta}) \quad (5.4)$$

where $\tilde{\theta}$ is the angle between the orientation of the current node and the straight line between the node and the goal, and $W(\theta)$ is the energy required to turn-in-place for an angle θ .

In order to implement this optimization criterium, we need to define a power model for the robot.

5.3.1 Defining a power model

Following some of the same procedure as [30], we first derive a function for the power consumption assuming pure rolling. This assumption can be reasonably applied to an Ackermann steered vehicle or a differentially steered vehicle, which have some wheels that are passive and used for balancing. However, this assumption is invalid for skid steered vehicles. The results of this analysis are used as an intermediate step in the analysis of power consumption for skid steered vehicles in curved motion.

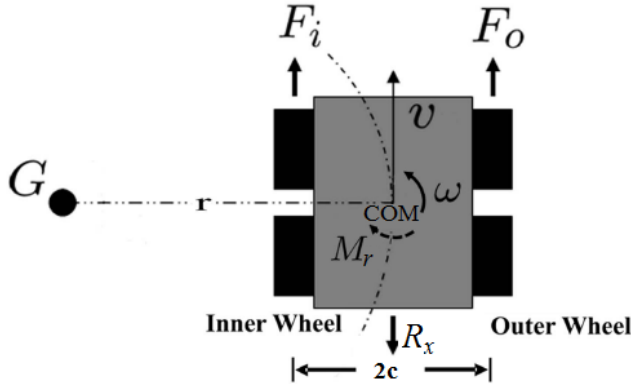


Figure 5.1: Representation of wheeled skid-steering vehicle force/torque and moment of turning resistance during curving[30].

As shown in figure 5.1, consider a skid-steered wheeled vehicle moving in circular motion of radius r about point G with a linear velocity v and an angular velocity of ω on a flat surface. In figure 5.1, as in section 3, COM denotes the geometric center of the vehicle, $2c$ is the width of the vehicle, F_i and F_o denote respectively the force of the inner and outer wheels, R_x denotes the force caused by rolling resistance, and M_r denotes the moment

of resistance that under the assumption of pure rolling is purely due to the longitudinal resistances. It follows that under constant linear and angular velocity

$$R_x = F_o + F_i = F_{\text{active}} \quad (5.5)$$

$$M_r = c(F_o - F_i) = M_{\text{active}} \quad (5.6)$$

where F_{active} and M_{active} are the active longitudinal force and angular torque developed by the wheels, respectively. Using the formula for instantaneous power

$$P = F_{\text{active}}v + M_{\text{active}}\omega$$

and recognizing that $\omega = v/r$, we see that the power consumption is the function of r and v given by

$$P(r, v) = \alpha(r)v \quad (5.7)$$

where

$$\alpha(r) = (F_o + F_i) + \frac{c(F_o - F_i)}{r} = R_x + \frac{M_r}{r} \quad (5.8)$$

Equations (5.7) and (5.8) show that, as would be expected, the effect of the moment of turning resistance decreases as the turning radius r increases. It should be noted that mass transfer during curving is not considered in the analysis since the vehicle is assumed to be moving at low speed.

The motion power consumption described by equation (5.7) was derived under the assumption of pure rolling and hence does not capture the effects of skidding. Now, we assume that the actual power consumption is given by

$$P(r, v) = \alpha(r)v + \beta(r) \quad (5.9)$$

where from equation (5.8) it is assumed that if $r_2 > r_1$, then $\alpha(r_2) < \alpha(r_1)$, and the term $\beta(r)$ has been added to take into account the power loss due to skidding.

In order to determine $\alpha(r)$ and $\beta(r)$, a large number of simulations were run on the simulator described in section 7. In these simulations, the path planning block was replaced by a block generating a path with a fixed linear velocity v_d and a fixed turning radius r_d . The simulation was set to run for 50 seconds, and the power consumption was recorded as the average of the last 30 seconds, in order to avoid any large influence of transient spikes. This basic experiment was performed for all combinations of v_d and r_d in the sets $\{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2,$

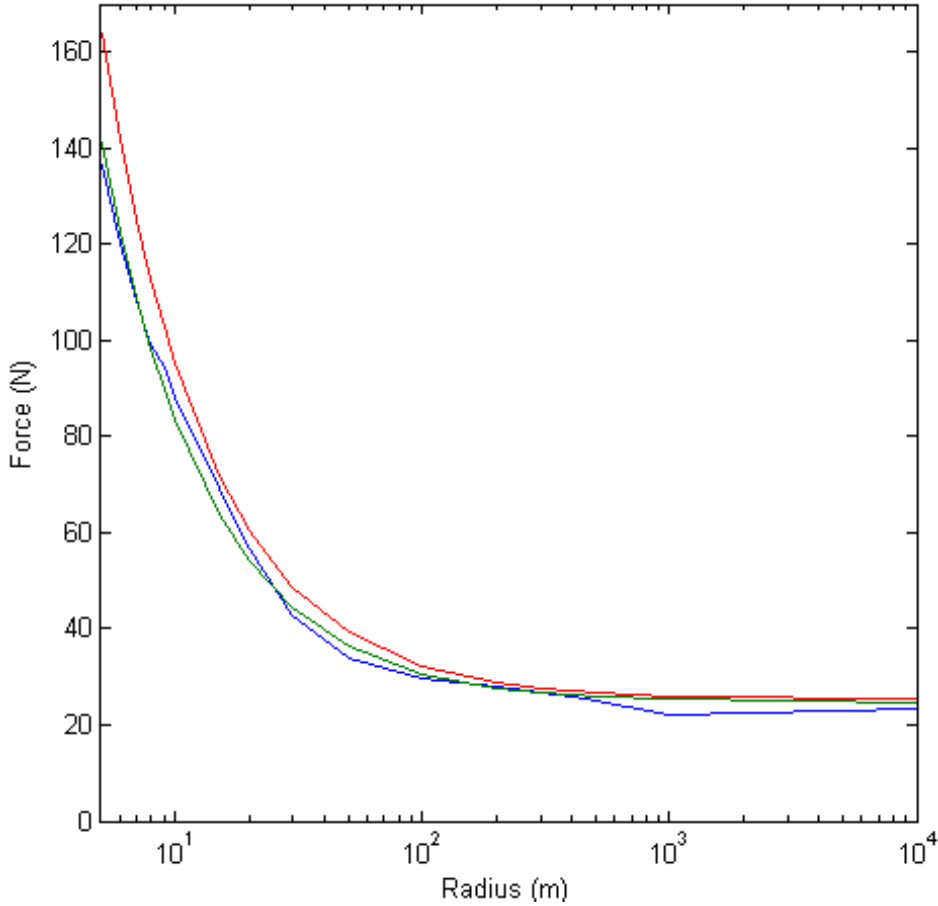


Figure 5.2: $\alpha(r)$ as obtained through simulation (blue), analytical analysis (red) and regression (green)

1.3, 1.4} m/s and {5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100, 200, 300, 400, 500, 1000, 10000, ∞ } m.

Analytically, we would expect $\alpha(r)$ to be the same as in equation (5.8), but when comparing the blue (simulation) and red (analytical solution) curves in figure 5.2, we clearly see a difference, although the shape of the curve is similar. If we assume that the function has the same form, only with different values for R_x and M_r , and run the matlab regression function `polyfit` on the curve, we get the values $R_x = 24.7442$ and $M_r = 586.818$, shown in figure 5.2 by the green curve. The value for R_x is reasonably close to the correct value of 25.3, whereas the M_r -value is quite far from

the correct 700.

The reason for this difference is assumed to lie in the shortcomings of the controller, as pointed out in [2]. The model does not drive in a smooth curve, but rather in a series of short straight lines, which may cause an artificially lower power value. Nevertheless, the goal of this power model is to describe the behaviour of our simulated system, and the results of `polyfit` is deemed to be the most accurate description of the currently implemented system with controller.

For the energy function in equation (5.4) for turning-in-place, we define the following:

$$W(\theta) = M_r \theta \quad (5.10)$$

where M_r is the value found through regression.

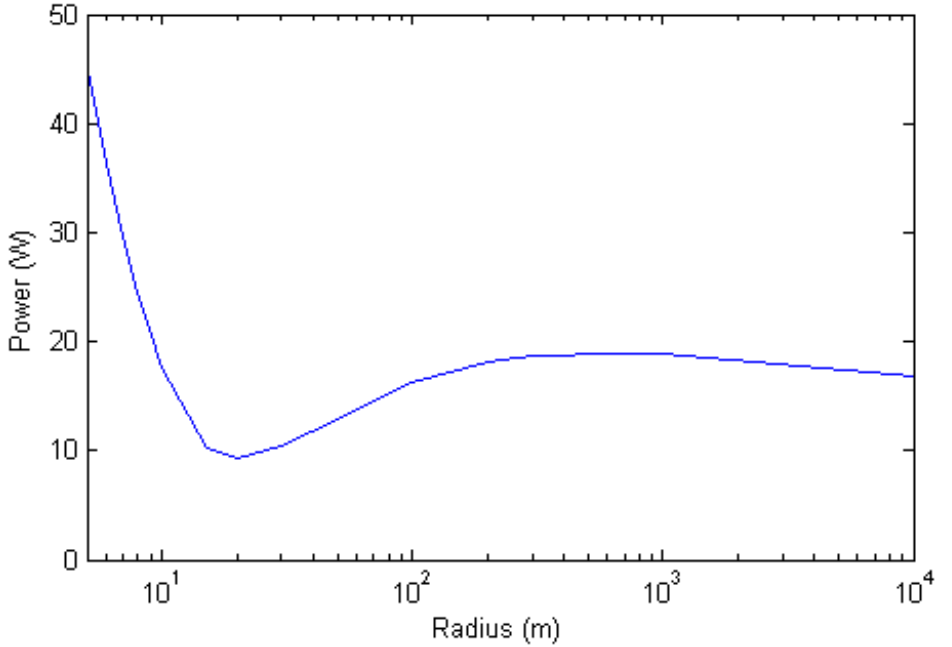


Figure 5.3: $\beta(r)$ as obtained through simulation.

In [30], $\beta(r)$ displayed the same steadily decaying characteristics as $\alpha(r)$ and was negligible for straight line motion. From figure 5.3, it is quite clear that this is not the case for our system. Once again, this is likely due to the weaknesses of the implemented tracking controller. When travelling in a straight line, the vehicle turns back and forth across the designated

trajectory, causing it to consume more energy. The controller struggles more with this at lower speeds, up to the point where 0.1 m/s actually had to be omitted from the experiment set, because the simulator simply failed to run the full 50 seconds.

While it naturally would have been better to work with a more ideal controller, the time it would take to find and implement a new controller seems inappropriate when the performance of the controller is not a primary goal of this thesis. The controller works reasonably well and the $\beta(r)$ -function describes the power behaviour satisfactorily. Since no continuous function to accurately describe $\beta(r)$ could be found, it was decided to interpolate it using the values found in the simulation with the matlab function `interp1`.

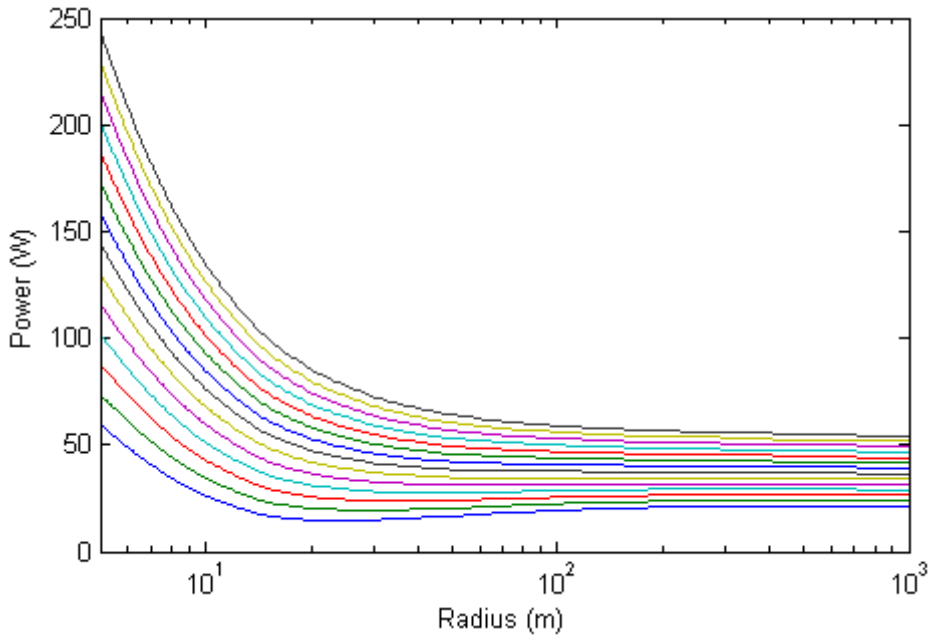


Figure 5.4: Power consumption for different speeds with respect to turning radius. Each curve represents a speed ranging from 1.4 m/s for the upper line to 0.1 m/s for the lower line in 0.1 m/s increments.

Figures 5.4 and 5.5 display the behaviour of our final power model. We clearly see that the cost of sharp turns are far higher than for gentler curves. As a result of the peculiar behaviour of the $\beta(r)$ -function, some sharper turns ($r \approx 11$) will be more energy efficient than gentler turns, particularly at low speeds. At speeds over 0.6 m/s, this effect is over-

shadowed by the larger $\alpha(r)v$ term, making the power curve a steadily decreasing function with respect to turning radius. Since the simulations in section 9 use speeds ranging from 1.0 to 1.4 m/s, this effect will not have an impact on these simulations.

It should be noted that these power curves will be highly dependant on the friction properties between the wheels and the surface. The simulations used to derive this power model assumed rubber tires and concrete floor. Other combinations will naturally have different friction properties, resulting in different power models. The general shape of the power model, with power consumption increasing exponentially with decreasing turning radius, will remain the same.

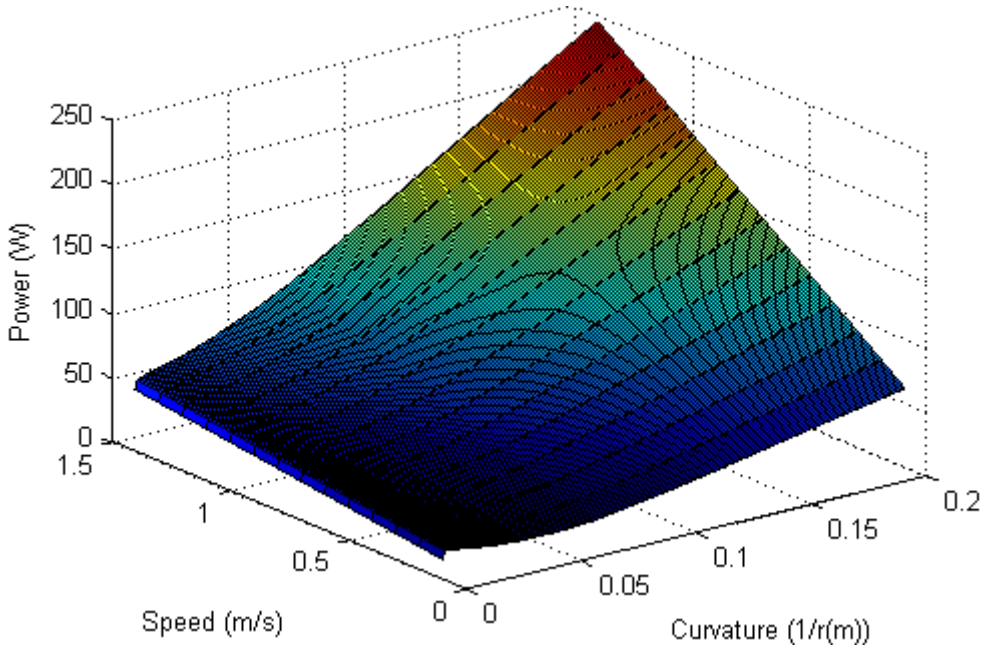


Figure 5.5: Power consumption with respect to curvature and speed

6 MATLAB implementation of the SBMPO-algorithm

In the conclusion of section 4.4.3, it was mentioned that implementation of the algorithm would involve a great deal of programming effort. In an attempt to lighten the required programming effort and make the code more readable and intuitive, it was decided to implement the algorithm in MATLAB. This choice was also made for compatibility reasons, since the simulator described in section 7.2 was already implemented in SIMULINK. The use of MEX-files (MATLAB EXecutable C/C++ code) was considered to boost speed, but simplicity and readability was deemed more important than computation speed.

Choosing a high-level programming language like MATLAB over lower-level languages like for instance C++, will obviously have a massive impact on the runtime of the algorithm. [31] shows a comparison of MATLAB versus C++ runtime, where C++ consistently achieved runtimes of approximately 1/500th that of MATLAB. Obviously, this means we will abandon any hope of making an online path planner in this implementation.

No previous implementations of SBMPO could be found in any available programming language. While MATLAB offers many large and useful libraries and tools, they unfortunately do not cover all the needs of the SBMPO algorithm either, and as a result, all necessary classes and functions had to be implemented from scratch. While the pseudo code in figure 4.7 is only 29 lines long, the need to implement custom built libraries caused the final MATLAB code to exceed 1000 lines of code. Granted, this number could probably have been significantly reduced by removing functions needed for debugging and plotting, and sacrificing some readability for a lower number of lines. Again, the need for visual demonstrations and simple, intuitive code was deemed more important.

In this section, the structure and functionality of the implementation will be explained, along with some of the modifications that had to be developed to make the algorithm work satisfactorily.

6.1 The structure of the path planner

6.1.1 The classes

At the heart of the implementation are the four classes shown in figure 6.1. All of them are derived from the handle superclass, so each instance of a class is a pointer to an object containing the attributes. This means a single object can be stored in several arrays without demanding memory space for more than one object.

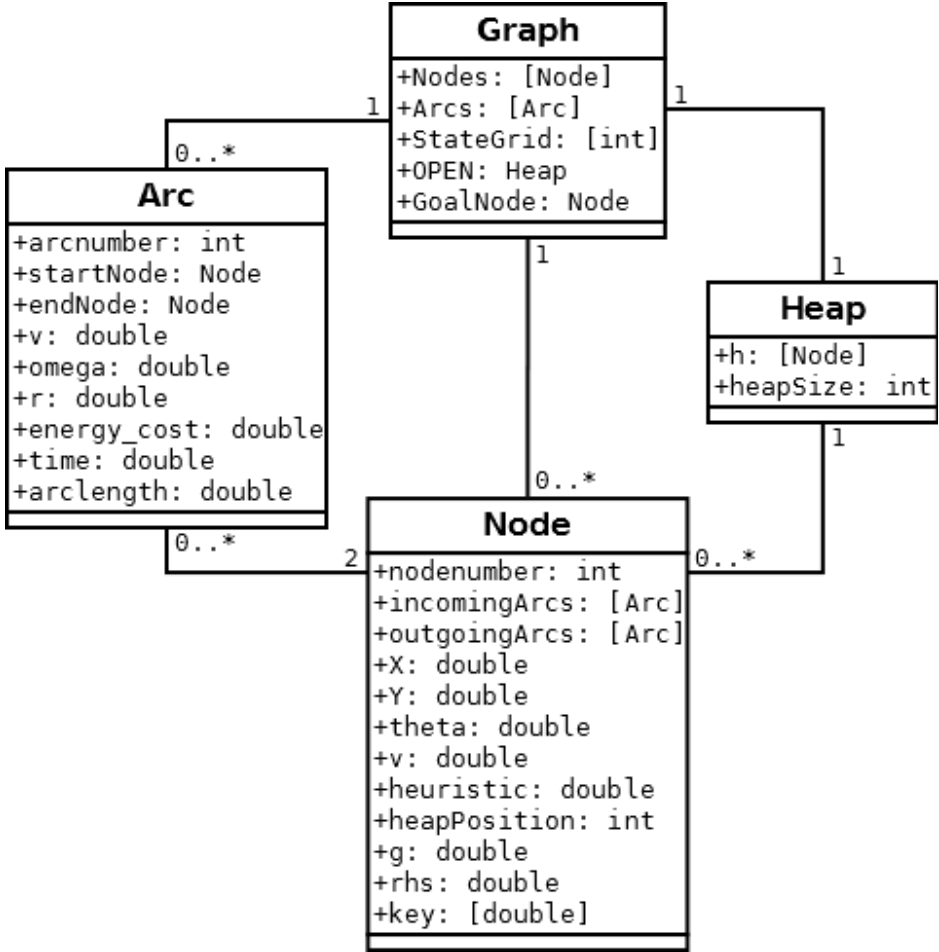


Figure 6.1: Class diagram of the MATLAB implementation

The Node and Arc classes are the basis classes. In addition to variables concerning the states in a node and on an arc, each arc contains a pointer to its start and end node, while each node contains an array of all the arcs leading to it and one array for all arcs leaving from it.

The Graph class is the main class, containing an array of all the arcs and an array of all the nodes in the graph, a State Grid to avoid infinite density of nodes, and an OPEN Heap to prioritize the locally inconsistent nodes ($node.g \neq node.rhs$).

The StateGrid of the graph is a 4D array (corresponding to the four states X, Y, theta and v of the nodes). Each cell of the array represents

an interval in position, angle and speed. If there is a node in this interval, the cell contains the nodenumber of the node in the interval, otherwise the cell contains 0 indicating an empty interval.

The Heap class is an array implementation of a binary min heap that sorts the nodes in it according to the key variable. The heap structure is ideal for priority queues, since it remains sorted while inserting or removing a node has a runtime³ of only $O(\log n)$.

6.1.2 The functions

In addition to multiple smaller functions, there are 5 major functions that are implemented either as separate functions or as part of one of the classes: **sbmpo_alg**(Graph), **generate_neighbours**(Graph, Node), **get_path**(Graph), **translate_path**([Arc], Node) and **path_planner**(time, state).

Both **sbmpo_alg** and **generate_neighbours** is implemented according to the pseudocode in figure 4.7, with a few tweaks that will be outlined in the following subsections.

get_path traverses the graph by starting at the goal, always choosing the arc to the predecessor node that minimizes $(node.g + arc.cost)$, until the start node is reached. The output of the function is an array of the arcs that compose the optimal path.

The controller presented in section 3.1 takes input in the form of $\mathbf{z}_{des} = [X + d_0 \cos \theta, Y + d_0 \sin \theta]^T$ and its derivatives. It was decided to translate the array of arcs from **get_path** into \mathbf{z}_{des} in two steps, with the intention of making it easier to switch to a better controller with a different input at a later stage.

The first step is **translate_path**, which converts an arc into a 5-by- N array containing $[X, Y, \theta, v, \omega]^T$ for each timestep of the arc (with N being the number of timesteps in an arc). This function also performs a smoothing of the output, as will be further explained in section 6.4.

The second step is **path_planner**, which is the function being called by the SIMULINK model presented in section 7. This function takes as its input the time and state of the model, along with the array of arcs from **get_path**. It keeps an array of $[X, Y, \theta, v, \omega]^T$ -values which it updates every time it reaches the end of an arc using **translate_path**, and translates the appropriate $[X, Y, \theta, v, \omega]^T$ -values into an 1-by-8 vector

³Big O notation ($O(f(n))$) defines an upper limit for a function. If an algorithm has a runtime of $T(n) = O(f(n))$, this means that $T(n) \leq C \cdot f(n)$ holds for some constant C .

of \mathbf{z}_{des} and its first, second and third derivatives.

6.2 The simplified model of the vehicle

In order to use the sampled input of the system to generate the output nodes in the configuration space, we need a mathematical model for the system. The model derived in section 3 is quite extensive and requires a lot of calculation, primarily because we need to consider the effects of friction. If we were to use the complete model in the path planning algorithm, the algorithm would end up with a runtime far too high to be of any use. Therefore, we need to define a much simpler model.

As demonstrated in [2], the implemented Simulink model with controller is able to follow a trajectory given the the input of longitudinal speed and angular speed. This means we can make a simple model with inputs $[v_x, \omega]$, and outputs $[X, Y, \theta]$. The simplest and most intuitive model is probably the one derived by the euler method:

$$\begin{aligned} X(t_2) &= X(t_1) + v_x \Delta t \cos \theta(t_1) \\ Y(t_2) &= Y(t_1) + v_x \Delta t \sin \theta(t_1) \\ \theta(t_2) &= \theta(t_1) + \omega \Delta t \end{aligned} \tag{6.1}$$

where $t_2 = t_1 + \Delta t$. While this model is very simple, there are some accuracy issues: The red line in figure 6.2 represents a path with $v_x = 1$ and $\omega = 0.3$. While this is a sharper turn than the path planner is intended to allow, it is used to demonstrate the weakness of the model in equation (6.1) (represented by the blue line) as it makes the flaw so obvious. The accuracy is directly tied to the size of the timestep Δt .

Since the model is intended for use in creating arcs with a fixed v_x and ω , we can exploit the fact that we know the arcs will be part of a circle with radius $r = \frac{v_x}{\omega}$. As demonstrated in figure 6.2, the model in equation (6.1) finds the next point in the iteration using a tangent to the curve with length $v\Delta t$, and immediately diverges from the correct curve.

If we instead use a chord of the curve (represented by the green lines), we can make the accuracy of the model independent of the timestep Δt . This would allow us to choose the timestep Δt solely based on how often we wish to check for obstacles, and not how accurately we want the output to correspond to the given input.

For a circular sector with radius $r = \frac{v_x}{\omega}$ and central angle $\tilde{\theta} = \omega \Delta t$, we have

$$\text{chordlength} = 2r \sin \frac{\tilde{\theta}}{2} = 2 \frac{v_x}{\omega} \sin \frac{\omega \Delta t}{2}, \forall \omega \neq 0 \tag{6.2}$$

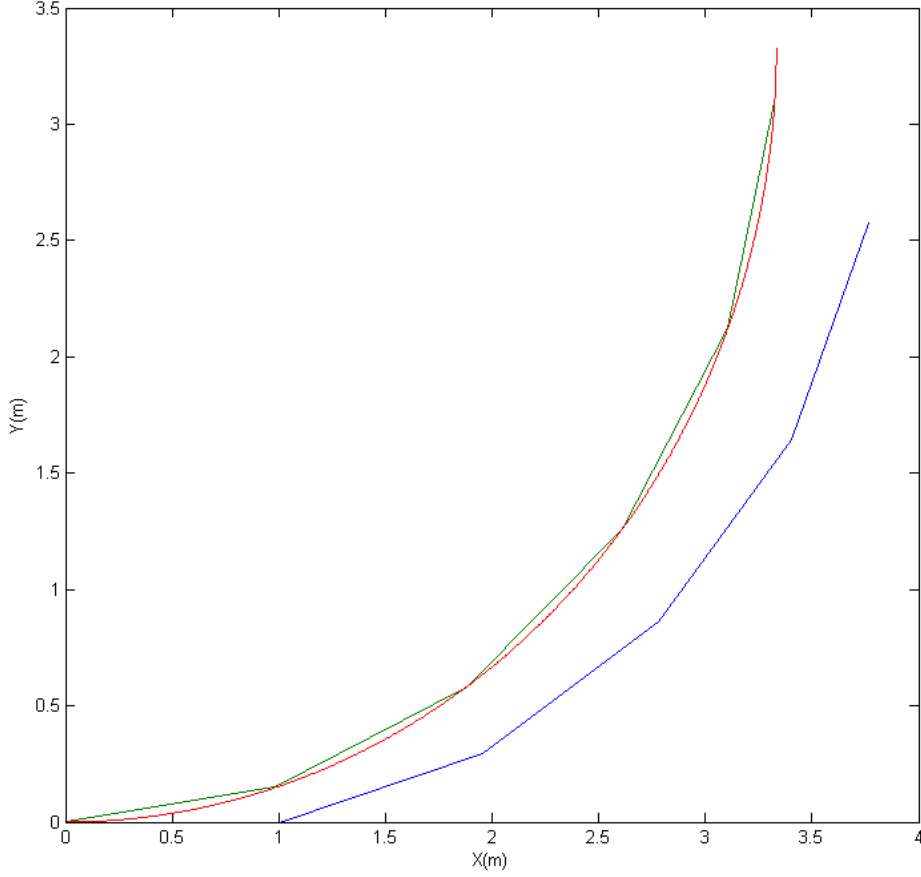


Figure 6.2: Accuracy of different simple models for the vehicle

In the case of $\omega = 0$, the chordlength reduces to $v_x \Delta t$. The angle of the chord line can be found by taking the average of the vehicle orientation θ before and after the chord. Combining this, we arrive at the following model:

$$\begin{aligned}
 X(t_2) &= X(t_1) + 2 \frac{v_x}{\omega} \sin \frac{\omega \Delta t}{2} \cos \frac{\theta(t_1) + \theta(t_2)}{2} \\
 Y(t_2) &= Y(t_1) + 2 \frac{v_x}{\omega} \sin \frac{\omega \Delta t}{2} \sin \frac{\theta(t_1) + \theta(t_2)}{2} \\
 \theta(t_2) &= \theta(t_1) + \omega \Delta t
 \end{aligned} \tag{6.3}$$

6.3 Sampling the input

To avoid generating paths that the full system is unable to follow without using excessive amounts of power, we put the following restrictions on the inputs of the model in (6.3):

$$\begin{aligned} v_x &\in [v_{x0} - a_x, v_{x0} + a_x] \\ \omega &\in \left[-\frac{v_x}{r_{min}}, \frac{v_x}{r_{min}} \right] \iff r \in [r_{min}, \infty) \end{aligned} \quad (6.4)$$

where v_{x0} is the speed at the start node, with $a_x = 0.2$ and $r_{min} = 5$ chosen after some experimentation with the simulator.

To limit the amount of reachable speed states, it was decided that the speed input v_x would only take three values when expanding a vertex: maximum acceleration ($v_{x0} + a_x$), maximum deceleration ($v_{x0} - a_x$), and no acceleration (v_{x0}).

Since moving in a straight line is by far the least energy demanding, it is natural to assume that straight lines will be part of the optimal path. Also, for a vehicle with a constraint on the turning radius, Dubins demonstrated in [32] that the shortest path will be made up of arcs with maximum curvature and/or straight line segments. So for each longitudinal speed, a set of B angular speeds were sampled using a halton set (B being the branching factor), and in addition it was decided to add the two angular speeds corresponding to maximum turning in each direction as well as $\omega = 0$ for moving in a straight line.

6.4 Smoothing the output

The vehicle controller presented in section 3.1 and implemented in section 7 is designed to track a smooth trajectory. As a result of the necessary introduction of the state grid (see sections 4.4.3 and 6.1), there is a high chance that there will be discontinuities between some of the arcs that make up the computed optimal path. What happens is that the end of one arc is close enough to the startingpoint of the next arc to be considered the same in the state grid, and the algorithm considers this point to have been reached, although this might not be the case. The risk for this behaviour increases if the number of nodes and arcs is high. Naturally, the coarser the resolution of the state grid, the larger the discontinuities. Also, because of the choice of $[v_x, \omega]$ as constants on the arcs, they will have step changes between arcs.

While these discontinuities don't have a large impact on the final distance travelled and the time it takes, the impact on the energy consump-

tion is massive. The first simulations of an optimal path with respect to energy consumption, revealed the actual energy consumption to be over 300% of the estimated consumption in some cases. The cause of this is of course that a sudden step in position error causes the controller to command a large power surge to correct the error. The sudden changes in v_x and ω also cause some violent spikes in the motor torque.

It would have been preferable to have modified the controller to handle this, but due to time constraints it was decided to smooth the output of the SBMPO-algorithm instead.

To smooth the discontinuities, we want the error between the endpoint of an arc and the startpoint of the next arc to be distributed over the entire arc. To do this, we will derive two smoothing functions $f_1, f_2 : \{1, \dots, n, \dots, N\} \rightarrow [0, 1]$ that are to be multiplied with the errors in the following scheme:

$$\begin{aligned} X_{smooth}(n) &= X(n) + X_{error}f_1(n) \\ Y_{smooth}(n) &= Y(n) + Y_{error}f_1(n) \\ \theta_{smooth}(n) &= \theta(n) + \theta_{error}f_1(n) \\ v_{smooth}(n) &= v(n) + v_{error}f_2(n) \\ \omega_{smooth}(n) &= \omega(n) + \omega_{error}f_2(n) \end{aligned} \tag{6.5}$$

where n is the number of the timestep and N is the total number of timesteps. This approach is not ideal, and will yield some outputs that are conflict with each other (for instance changes in θ that don't correspond to the value of ω and vice versa), but the performance improvement of the controller with this approach over the unsmoothed output makes it a viable option.

We want the f_1 -function to be as smooth as possible, with no abrupt changes. The requirements of this function, in addition to being continuously differentiable, can be summarised as follows:

$$\begin{aligned} f_1(0) &= 0 \\ f'_1(0) &= 0 \\ f_1(N) &= 1 \\ f'_1(N) &= 0 \end{aligned} \tag{6.6}$$

where the first two expressions ensure the start of the arc does not make any abrupt changes to position and orientation, and the last two expression ensure the same for the end of the arc. One function that meets these criterea is

$$f_1(n) = -2 \left(\frac{n}{N} \right)^3 + 3 \left(\frac{n}{N} \right)^2 \tag{6.7}$$

which is shown in blue in figure 6.3.

The speeds should ideally stay the same for the entire arc, but to avoid spikes in the motor torque the acceleration needs to be distributed to a certain degree. In our case, we choose to implement a constant acceleration in the final 0.05 seconds of a 1 second arc, as shown in green in figure 6.3.

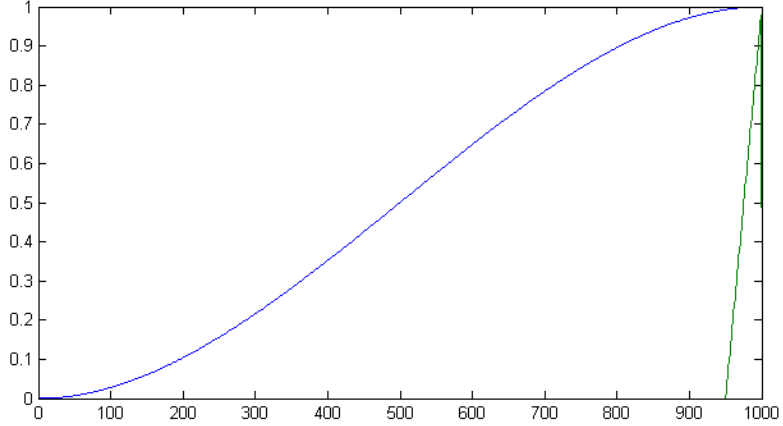


Figure 6.3: Output smoothing functions for X, Y, θ (blue) and v_x, ω (green).

6.5 Obstacle and goal handling

How we choose to handle the regions around the obstacles and the goal can have a major impact on the performance of the algorithm. In the following paragraphs, we will present an approach to require a minimum of calculation.

6.5.1 The configuration space of the vehicle

The mobile platform has three degrees of freedom, resulting in a configuration space $\mathcal{C} \in \mathbb{R}^3$. In order to simplify the obstacle handling, we will approximate the configuration space by collapsing the vehicle orientation in \mathcal{C} , resulting in $\mathcal{C} \in \mathbb{R}^2$. In essence, we now consider the vehicle to be a circle centered in the origin of the vehicle with a large enough radius to cover the entire vehicle.

This approximation does come with a few tradeoffs: In stead of a roughly rectangular vehicle measuring 110 by 85 centimeters, we now have a circular vehicle measuring approximately 135 centimeters in diameter. This will obviously mean that while the platform could easily drive through a meter wide gap, the path planning algorithm will consider it to be too tight. Since the tiling system is intended to be working in a large open space with relatively few obstacles, this does not seem like a severe disadvantage. The platform will also not be able to get as close to the position to be tiled since the tiling glue will be defined as an obstacle. The robotic manipulator should be able to compensate for this, but this obviously needs to be assesed when the type, range and placement of the robotic manipulator are to be decided.

On the plus side, the added distance required will create a buffer, reducing the risk of minor collisions due to inaccuracies.

In MATLAB, the configuration space is implemented by adding the radius of the vehicle to the obstacles, so in stead of considering a vehicle with radius 67.5 cm and obstacles with radius 50 cm, the algorithm considers the vehicle to be a point and the obstacles to have a radius of 117.5 cm.

6.5.2 Obstacle handling

If we look back to the `Generate_Neighbours` function of the SBMPO algorithm in figure 4.7, we see that the obstacle check in line 5 ($x(t) \notin \mathbf{X}_{free}(t)$) is inside a double for loop, so the function will be called frequently, and optimizing it will be essential in improving the runtime of the algorithm.

In section 6.5.1, we made some simplifications to the configuration space. The most important with respect to obstacle handling is that we decided that the vehicle center of mass need to be at least a specific distance from the center of any obstacle to be in \mathcal{C}_{free} . Finding the specific euclidian distance⁴ requires a potentially costly square root operation, but we can avoid the squareroot entirely using a simple trick: Instead of comparing the actual distance to the required minimum distance, we can compare the square of the actual distance to the square of the required minimum distance. The relation is the same, and we avoid having to use the sqrt function.

We can reduce the computational load further by reducing the number of obstacles we consider the distance to. As we start expanding a node using the `Generate_Neighbours`, we already know that the obstacles that

⁴ $d = \sqrt{\Delta x^2 + \Delta y^2}$

are too far away to collide with after driving straight towards it at full speed for the time length of one arc, will not be reached at any point between the nodes either, and can therefore be ignored in this running of `Generate_Neighbours`.

Another opportunity to reduce the computations even further is the case where no obstacles are within collidable distance. In this case, we can skip the obstacle test entirely, and because of the accuracy properties of the model in equation (6.3), we can even skip the intermediate points between nodes and just calculate the state of the new node directly by simply exchanging Δt with the time length of an entire arc.

6.5.3 Goal handling

Since the arcs have a certain length, it is possible that a node expansion close to the goal node would create an arc going through the goal node, but overshooting it. In [29] this was solved by expanding the goal node to a larger goal region. We wish to be able to travel to a specific position, so we therefore use a different approach:

If the node picked for expanding is close enough in distance to the goal node that it should be able to reach it within a single arc, a new method `Calculate_Goal_Arc` is called instead of `Generate_Neighbours`. This function calculates the distance and turning radius required to reach the goal node using equation (6.3), and if the turning radius is large enough to be admissible creates an arc for each admissible speed, and then performs the same tests as `Generate_Neighbours` on the arcs.

Note that this implementation does not require the vehicle to reach the goal with a specific orientation. This is proposed as one of the possible extensions in section 12.

7 Simulation

In this section, a simulator of a mobile platform model with a tracking controller is presented. The simulator is implemented in SIMULINK, and was built from scratch and thoroughly tested in [2], using the equations of motion derived in section 3.

7.1 Specification

Before implementing the system in SIMULINK, some considerations need to be made, and the input parameters need to be chosen.

7.1.1 Assumptions and simplifications in the modelling

Further assumptions have been made in addition to the ones already covered in section 3:

- The platform and robot arm is modeled as a single cuboid with the measurements $84.2 \times 110.5 \times 53.3$ cm, weight $109 + 28.9 = 137.9$ kg and uniform density.
- The robot is assumed to be carrying 80 tiles, each measuring 30×30 cm, weighing 1.5 kg each and being stacked exactly above the vehicles' center of mass.
- The friction between the wheels and surface is assumed to be uniform, with reasonable values selected from [33]. The friction will be implemented using the Karnopp model described in [14], with a combination of deadzone, gain and saturation blocks.

The specifications[34] unfortunately do not offer the maximum torque of the wheel motors on platform, only the maximum payload of 181 kg and that the platform has an ability to turn-in-place. Based on equations (3.5) and (3.6), with $\mu = 0.8$ and $f_r = 0.01$, this gives an absolute minimum differential torque ($|\tau_1 - \tau_2|$) of 737 Nm. A reasonable assumption based on these numbers would have been $\tau_i \in [-400, 400]$.

If the torque output of the controller exceeds the maximum torque of the wheel motors, saturation will add a nonlinear effect to the system. If a controller's output is saturated and the controller contains an integral effect, we may experience the integral windup effect, where integral terms accumulate excessive error when the outputs are limited.

It has not been possible to procure any definite numbers for maximum torque, and the proposed controller from section 3.1 does not employ any anti-windup strategies. Because of this, it has been decided not to implement saturation of the torques in the simulator itself, but rather leave them unconstrained and instead comment on the validity of the outputs.

7.1.2 Model parameters

The implemented model is the one derived in section 3. The model parameters are shown in table 2. The values for the vehicle are based mostly on the numbers of the Segway RMP400 (the predecessor of the RMP440 LE discussed in section 3.2), since the full specifications found in table 1 unfortunately were not made available until after all the simulations had been run and documented.

Description	variable	Value
Mass of platform[34] and arm[35]	m_0	137.9 kg
Moment of inertia of platform and arm	L_g	22.1427 kgm ²
Weight per tile	tile_weight	1.5 kg
Distance in x between COM and wheels	a,b	0.3456 m
Distance in y between COM and wheels	c	0.2859 m
Radius of wheel	R_wheel	0.265 m
Deadzone in friction model	delta_v	1 cm/s
Coefficient of sliding friction	mu	0.8
Coefficient of rolling resistance	f_r	0.01
Gravitational acceleration	g	9.81
Acceleration error gain	k_a	10
Velocity error gain	k_v	65
Position error gain	k_p	97.5
x -position of ICR	d_0	0.18

Table 2: Model parameters used in SIMULINK

7.1.3 Simulation parameters

The system is simulated with SIMULINK using the Bogacki-Shampine ode3 fixed-step solver with step size 0.001, and a runtime of 50 seconds.

7.2 Simulator structure

The overall structure of the simulator is presented in figure 7.1. The **path_planner**-function described in section 6.1 is implemented in the interpreted matlab function block. All relevant values are stored in MATLAB using the "To Workspace"-blocks, and then plotted using a script. The vector \mathbf{q} is the same as in section 3, $\mathbf{q} = (X, Y, \theta)$, with $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ representing its derivative and second derivative, respectively. The $\mathbf{z_d_vect}$ variable is a collection of the desired smooth reference trajectory variables with 1st-3rd order derivatives, and can be summarized as follows:

$$\mathbf{z_d} = \begin{bmatrix} \mathbf{z_{d1}} \\ \mathbf{z_{d2}} \end{bmatrix}, \quad \mathbf{z_{di}} = \begin{bmatrix} z_{di} \\ \dot{z}_{di} \\ \ddot{z}_{di} \\ \ddot{\ddot{z}}_{di} \end{bmatrix}$$

Figure 7.2 shows the implementation of the Skid-steered mobile platform using equation (3.6). The **mass_calc**-block takes the number of tiles as an input, and adds the mass and moment of inertia of the stack of tiles to the total mass and moment of inertia of the platform to construct the necessary \mathbf{M}^{-1} -matrix.

The **c_func**-block in figure 7.2 is further elaborated in figure 7.3. The **find_eta**-block calculates the vehicle-frame velocities from the general coordinates. The three blocks representing the friction forces and moment (R_x , F_y and M_r) are implemented from equations (3.3)-(3.5).

In figure 7.4, we see the contents of the R_x -block (the F_y - and M_r -blocks also contain a similar structure). Note that where Caracciolo et al.[10] used a regular sign-block, we have a **modsgn**-block instead (details in figure 7.5). This is implemented with a switch in order to easily toggle between the two modes, with mode 1 being the standard sign-block, and mode 2 being the gain/saturation strategy discussed in section 3.

Combined with the gain/saturation strategy, we also need a dead-zone on the velocity of the vehicle. Since this needs to be implemented on the vehicle-frame velocities, the "Optional Deadzone w.rotation"-block has been constructed (see figure 7.6). This block transforms the fixed-reference-frame velocities to vehicle-frame velocities, implements the dead-zone, and then transforms the velocities back to the fixed-reference-frame. Like the **modsgn**-block, the implemented deadzone (figure 7.7 has been implemented with a switch in order to easily toggle between a model with or without a deadzone on the vehicle-frame velocities.

In figure 7.8, we see the implementation of the controller from 3.1.

Equation (3.13) is implemented in "Partially linearizing static feedback", equation (3.19) in "Linearizing dynamic feedback", and equation (3.21) in "Linear stabilization for tracking". The `Z_vect`-block calculates the linearizing outputs \mathbf{z} (see equation (3.15)) along with its first and second derivatives, and sends them on the form $\mathbf{z}_{\mathbf{v}i} = (z_i, \dot{z}_i, \ddot{z}_i)$

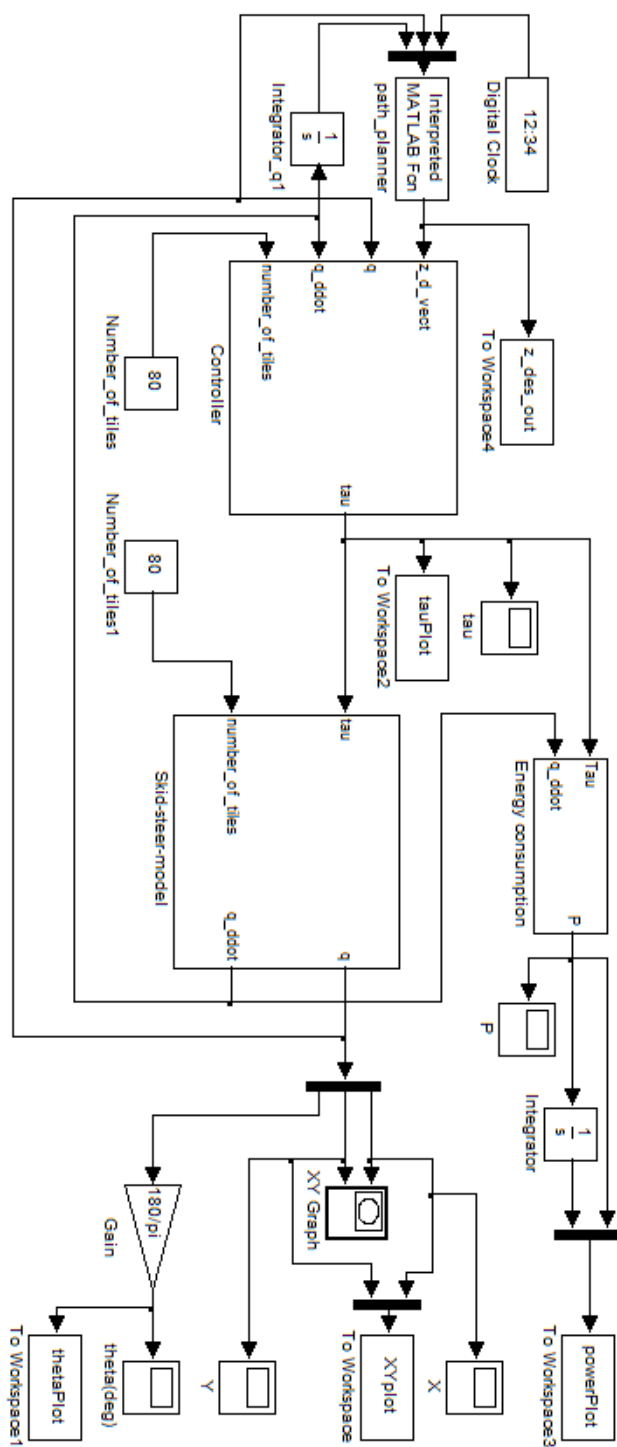


Figure 7.1: The complete implemented system

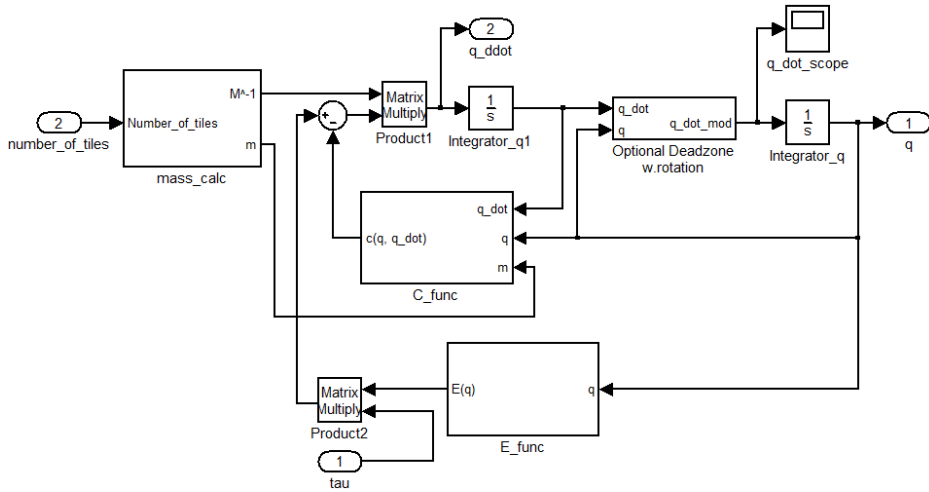


Figure 7.2: The implementation of the skid-steered model

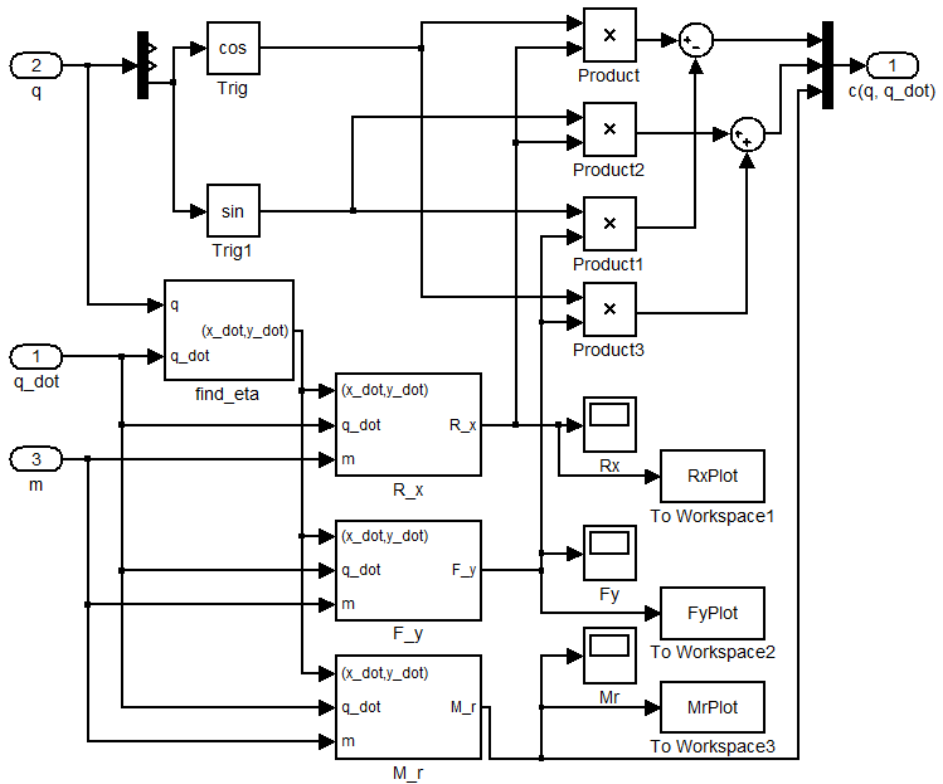


Figure 7.3: The c_func-block from figure 7.2

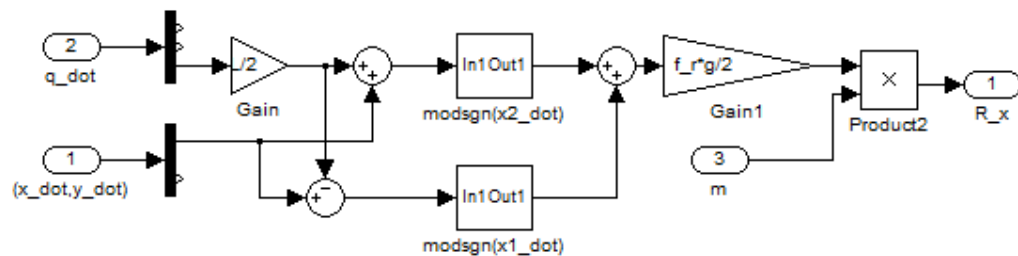
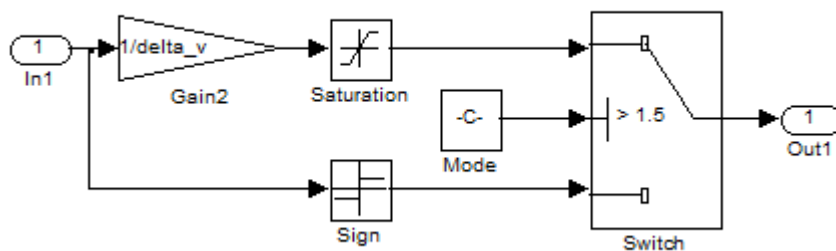
Figure 7.4: The R_x -block from figure 7.3

Figure 7.5: Implementation of the modified sign function discussed in section 3

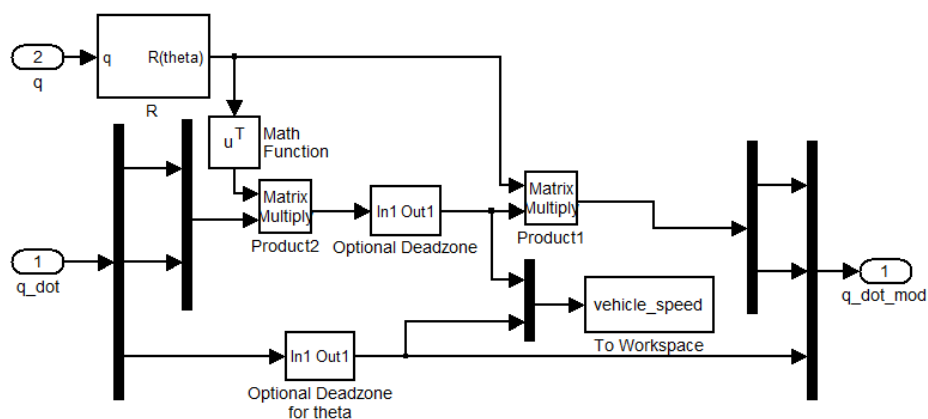


Figure 7.6: The "Optional Deadzone w.rotation"-block from figure 7.2.

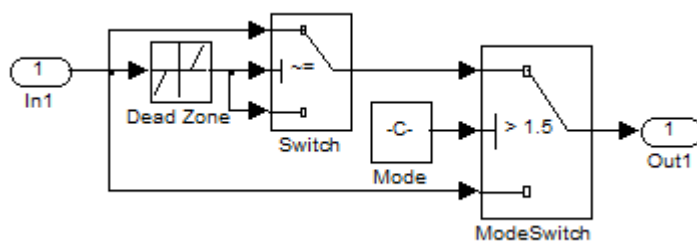


Figure 7.7: The "Optional Deadzone"-block from figure 7.6.

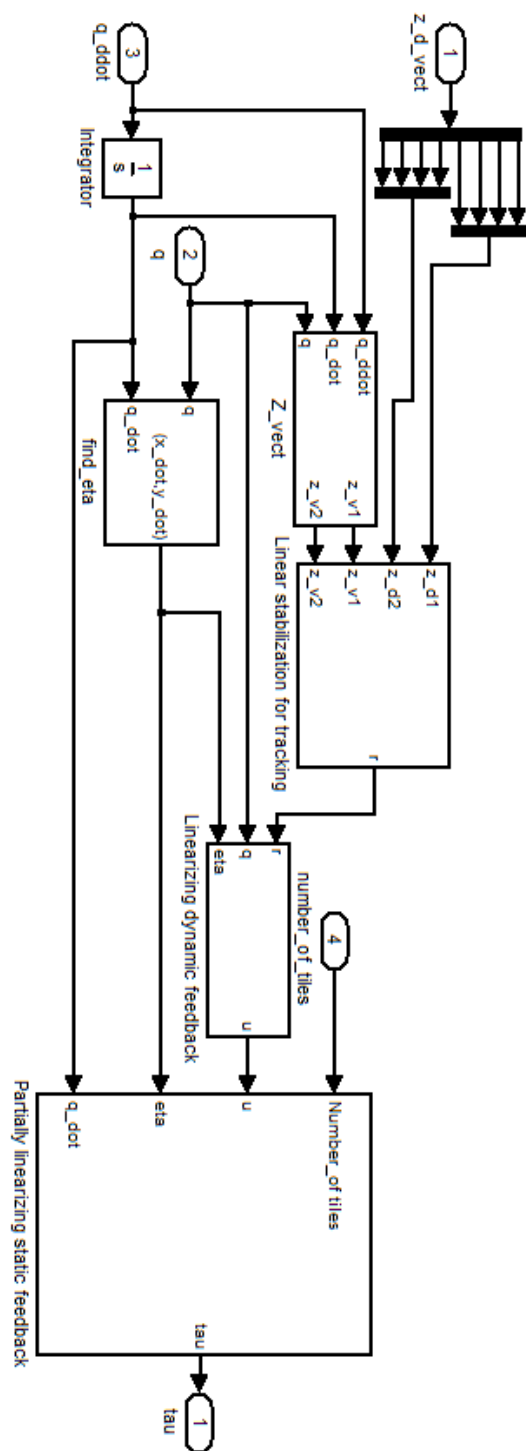


Figure 7.8: The implementation of the controller

7.2.1 Prepared functionality for future work

The model has been implemented with the intention of expanding it to a simulation of the entire system described in section 2.1.1. In the full simulation, the robot's mass and moment of inertia will be varying depending on the number of tiles it is currently carrying. Presently it is merely simulated as being loaded with a constant number of tiles at all times.

8 Defining the test cases

All of the chosen test cases start at position $(X, Y) = (1, 1)$ with orientation $\theta = 0$ and speed $v = 1\text{m/s}$. They also end at the same goal, $(X, Y) = (25, 15)$, with no demands on speed and orientation.

8.1 Test parameters

For each case, we will be looking at the performance of our three optimization criteria: Time, distance and energy consumption. Naturally, we need to consider both the quality of the chosen paths and the computational time required to find them.

There are also several tuning factors implemented in the SBMPO-algorithm (step-length, number of steps per arc, branching factor, and the resolution of the Implicit State Grid).

While originally not intended as a tuning parameter, altering the minimum turning radius of the vehicle turned out to yield very interesting results for the time (and distance) optimization. The optimizations with respect to time or distance tend to choose very sharp turns in order to correct the orientation quickly. If we look back to figure 5.4, we see that the power consumption is very high for sharp turns. The power consumption decreases very rapidly with increasing turning radius for small radii, but for radii above 11 meters, the decrease is much less significant. An extra optimization method based on time with an increased minimum turning radius of 11 meters is therefore added, since it is assumed this could give good results both with respect to time and energy consumption.

8.2 Case 1: No obstacles

We include this trivial case, merely to prove the algorithm actually works, and to use as a basis for placing the obstacles in the subsequent cases.

8.3 Case 2: Simple cluster of obstacles

This case contains 3 obstacles blocking the ideal path in case 1, creating a large obstacle/local minimum for the cost function. How well the algorithm copes with such local minima is crucial to avoid excessively high computational runtimes.

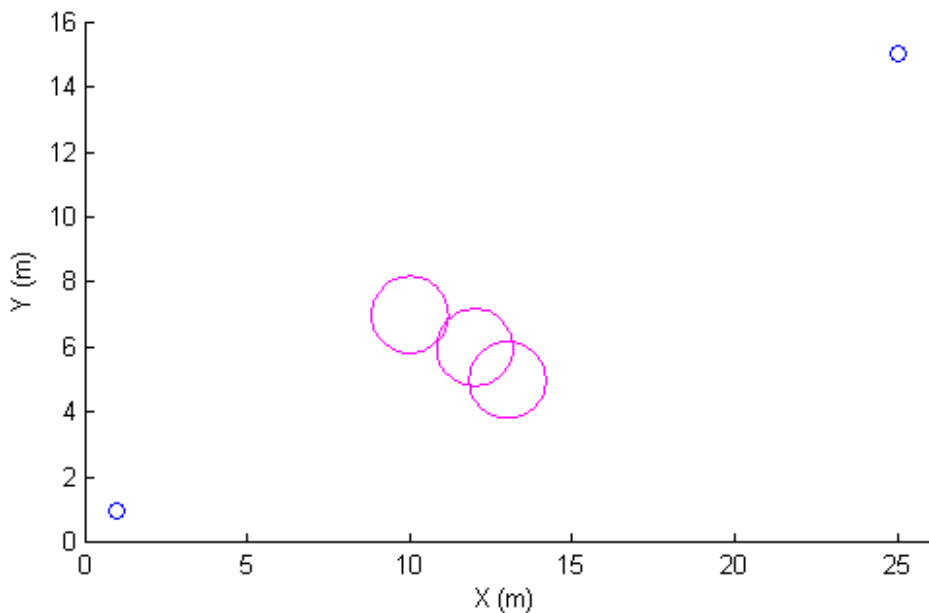


Figure 8.1: A simple cluster of obstacles

8.4 Case 3: Complex cluster of obstacles

This case contains a larger collection of obstacles chosen so as to resemble the obstacle set used in [29], in order to confirm their findings for fixed speed operation, and investigate how the addition of variable speed affects these findings. This case is specifically designed to contain a shortest path with several sharp turns, which will cause high power consumption. This will therefore demonstrate the fundamental differences of the different optimization criteria.

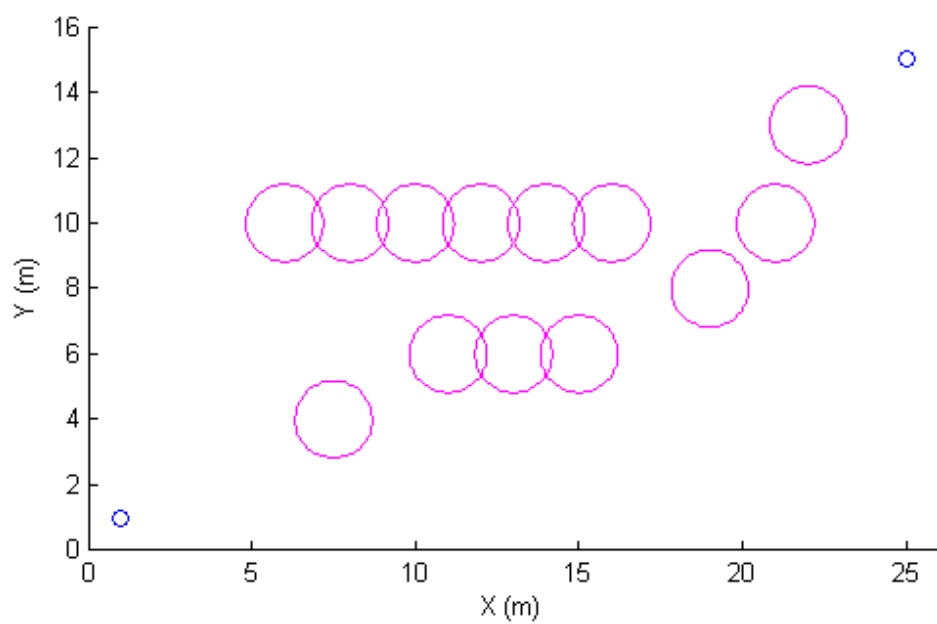


Figure 8.2: A more complex cluster of obstacles

9 Simulation results

All of the simulations in this section were done using SIMULINK, the model described in section 7 and the path planner described in section 6.

In the case of no obstacles, both the arcs and the nodes of the graph are plotted, to display the entire graph. In the more complicated cases, the arcs are omitted, simply because the amount of nodes and arcs is so high that adding the arcs only increases file size without adding any relevant information.

When using variable speed, the speeds are restricted to the set $\{1.0, 1.2, 1.4\}$ m/s. The reason for this is simply to get the simulations to terminate within reasonable time. The time-based algorithm can handle larger sets of possible speeds, because the algorithm naturally will be biased towards choosing the nodes with higher speeds for expanding, since these will lead to the goal in shorter time. The algorithms for distance and energy don't have the same incentive to prefer one speed over the other, and therefore end up expanding many unnecessary nodes. Initial tests suggested that each added speed-option doubled the number of nodes in the final graph, even though the added speed-options did not appear in the final computed path.

The number of nodes is very closely related to the algorithm runtime. As mentioned in section 6.1, the priority queue of the algorithm is implemented as a min heap. From the properties of a min heap, we can expect a runtime of $O(n \log n)$ for n insertions. In addition, every time a nodes priority key changes, we can expect a runtime of $O(\log n)$ for the updating of the queue. A very rough estimate based on the initial tests with the energy optimization algorithm would indicate a runtime of $O(n^2)$ (the runtime is also likely to be affected by the number of arcs), so controlling the amount of nodes being created is crucial. To avoid the algorithm running infinitely, a breakoff criteria of a maximum of 20,000 nodes was added to the SBMPO-procedure.

Unless specifically stated otherwise for the separate subcases, the tuning parameters of the algorithm are as presented in table 3.

Due to the SBMPO-algorithm being based on randomizing of the inputs, some runs of the algorithms might get lucky and find a very good path, while other runs might get unlucky and end up with a considerably worse path. To avoid chance having a too large effect on the results, all cases that terminated within reasonable time were run multiple times.

All tables display the average parameter values of at least three runs, while all figures display the path that was deemed most representative of

Tuning parameter	Value
Branching factor B	6
Steps per arc	8
Total arc length [s]	2
State grid position gridsize [m]	0.1
State grid angle gridsize [degrees]	10

Table 3: Algorithm tuning parameters

the runs (closest to the average). When viewing the graphs, keep in mind that in this representation, the obstacle circles represent the necessary distance to the center of the vehicle, meaning the paths are allowed to come arbitrarily close to the obstacle circles.

9.1 Case 1: No obstacles (variable speed)

Already in the case of no obstacles we start to see some characteristic differences between the different optimization schemes. The time scheme has an incentive to choose the nodes with the highest speeds, and as a result, ends up with much fewer nodes and significantly lower computation time than the other two schemes.

As we would expect, the time and distance schemes both compute the shortest path, the time scheme computes the lowest vehicle runtime, and the energy scheme gives the lowest estimated energy consumption.

However, when running the computed paths in the simulator, the actual energy consumption tells a very different story. While the distance and time schemes have an energy consumption of $\sim 110\%$ of the estimate, the energy scheme ends up consuming 133% of the estimate.

While the actual consumption being higher than the estimate can be attributed to the issues with the controller and the output smoothing discussed in section 6.4, the fact that the energy scheme has a much worse ratio of actual to estimated energy consumption (from here on referred to as the *estimate ratio*) is very interesting. In this case it is so bad that the energy scheme actually gives the poorest performance of all the algorithms in terms of actual energy consumption. In figure 9.3 and 9.4, we also see that the path seems to angle a bit back and forth, while the paths of the other schemes seem to move in straighter lines. This would indicate that the resolution of the angle part of the state grid is too low, but while this seemed to improve the estimate ratio, increasing this resolution led to such an increase in the amount of nodes that it was decided to keep

the grid size at 10 degrees. The energy scheme's apparent tendency to choose paths with discontinuities in the orientation of the vehicle will be discussed further in section 10.

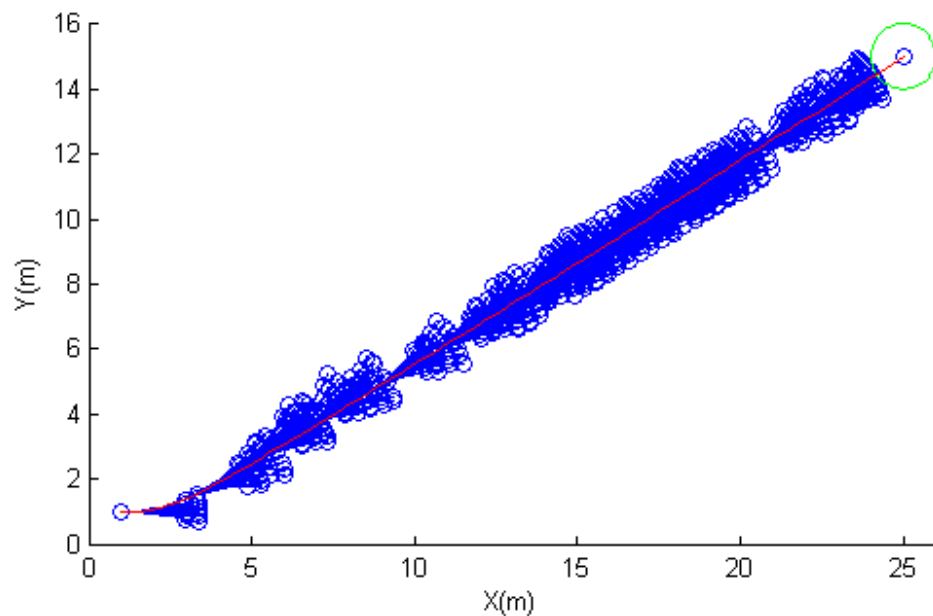


Figure 9.1: The graph for finding the Shortest path(blue) and the path itself(red).

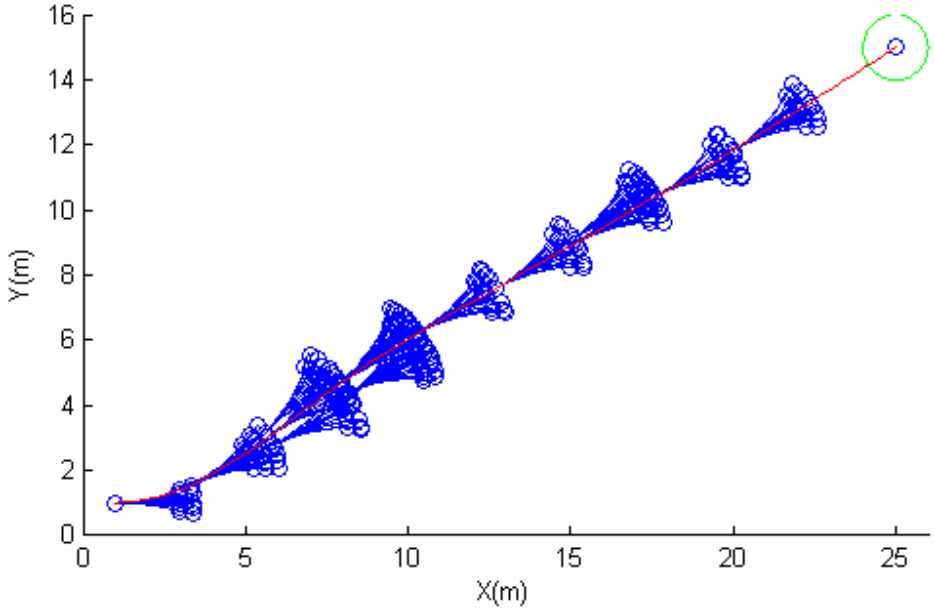


Figure 9.2: The graph for finding the Quickest path(blue) and the path itself(red).

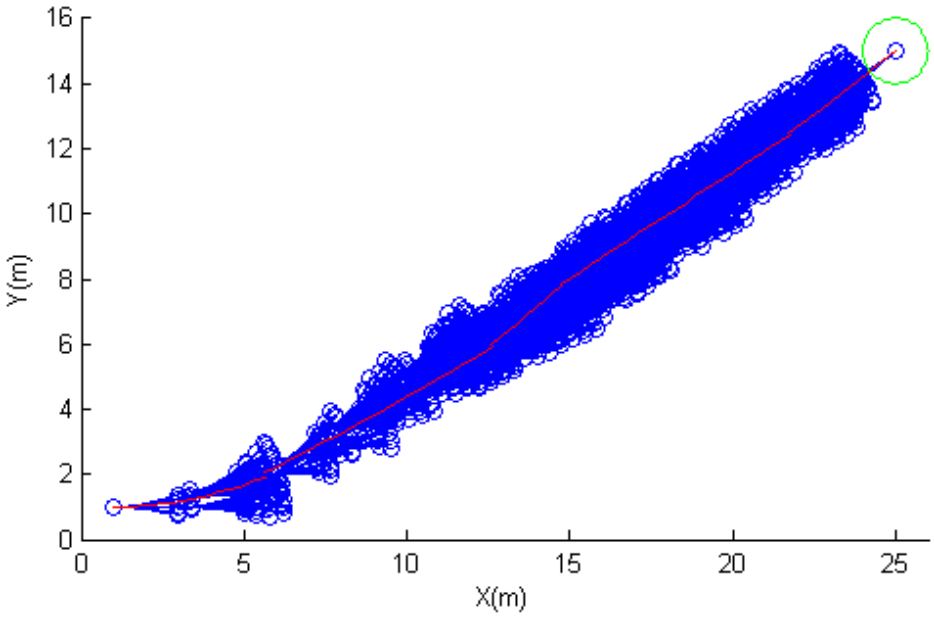


Figure 9.3: The graph for finding the Easiest path(blue) and the path itself(red).

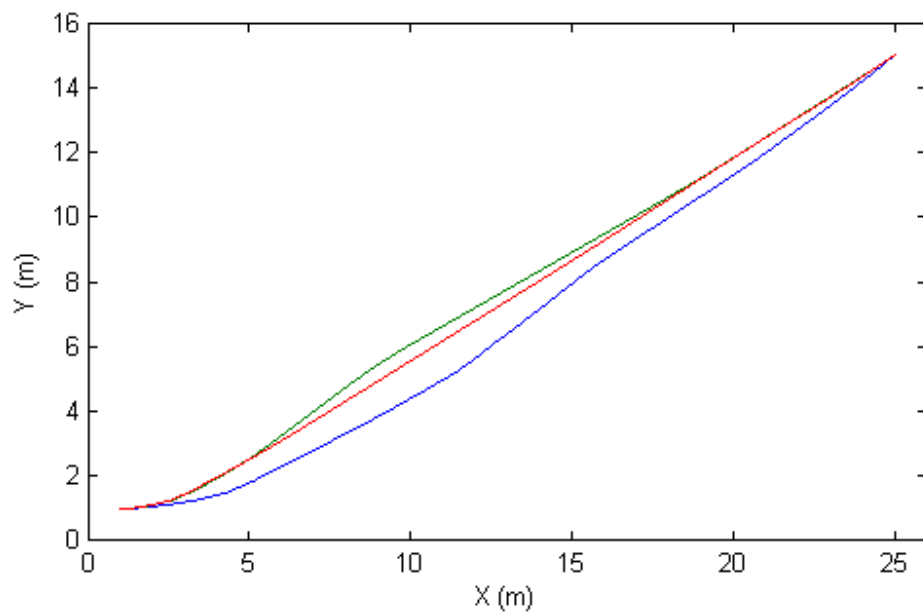


Figure 9.4: Comparison of Shortest (red), Quickest (green) and Easiest path (blue).

Optimization scheme Parameters	Distance	Time	Energy
Distance travelled [m]	27.93	27.93	28.15
Vehicle runtime [s]	23.91	20.24	21.46
Estimated energy consumption [J]	1506	1439	1257
Actual energy consumption [J]	1669	1544	1673
Actual/Estimated ratio	111%	107%	133%
Number of nodes	1166	238	4572
Number of arcs	2140	302	8441
Algorithm runtime [s]	8.79	0.598	105.26

Table 4: Comparison of the optimization schemes with no obstacles

9.2 Case 2: Simple set of obstacles

9.2.1 Constant speed=1.0 m/s

Since the speed is constant, optimizing with respect to time or distance reduces to the same problem. The table values for time therefore apply to optimizing with respect to distance as well. The constant speed also cause the nodes to cluster together in lines, since all the arcs will have the same length.

In figure 9.8, we see that the paths found by the energy and time($r_{min} = 11$ m) schemes take a very similar route. If we look at table 5 instead, we see that the energy scheme gives a path that is approximately 10% more energy efficient than time($r_{min} = 11$ m) and 33% more efficient than time($r_{min} = 5$ m) in this case.

As algorithm runtimes, time($r_{min} = 11$ m) is clearly faster than the other two.

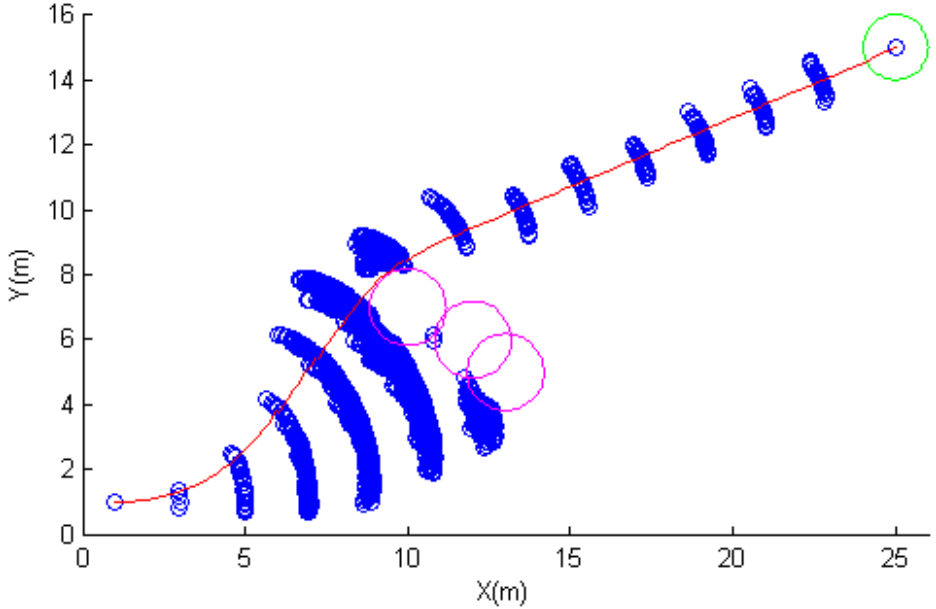


Figure 9.5: The Shortest path for simple obstacle set with constant speed and $r_{min} = 5$ m.

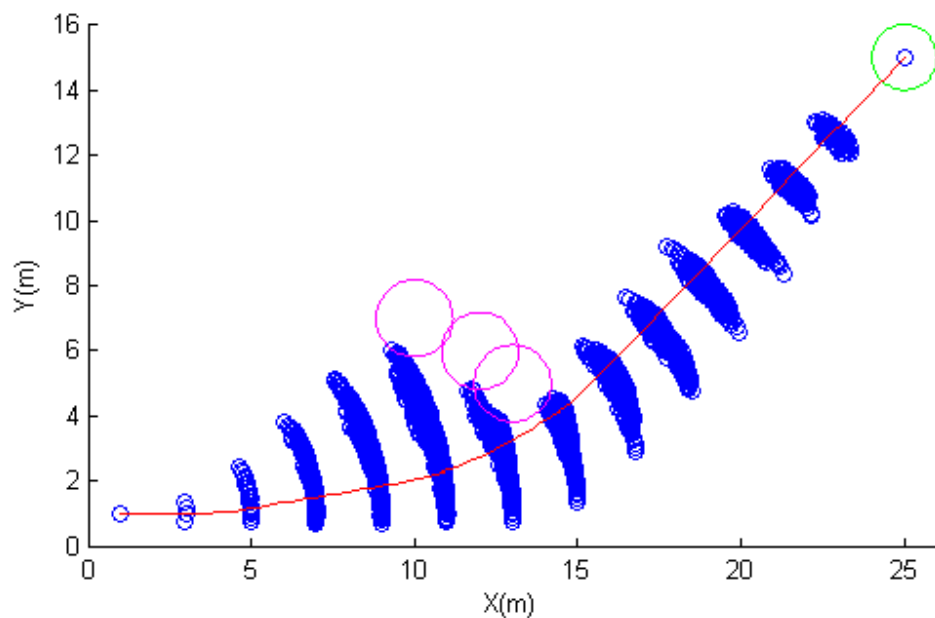


Figure 9.6: The Easiest path for simple obstacle set with constant speed.

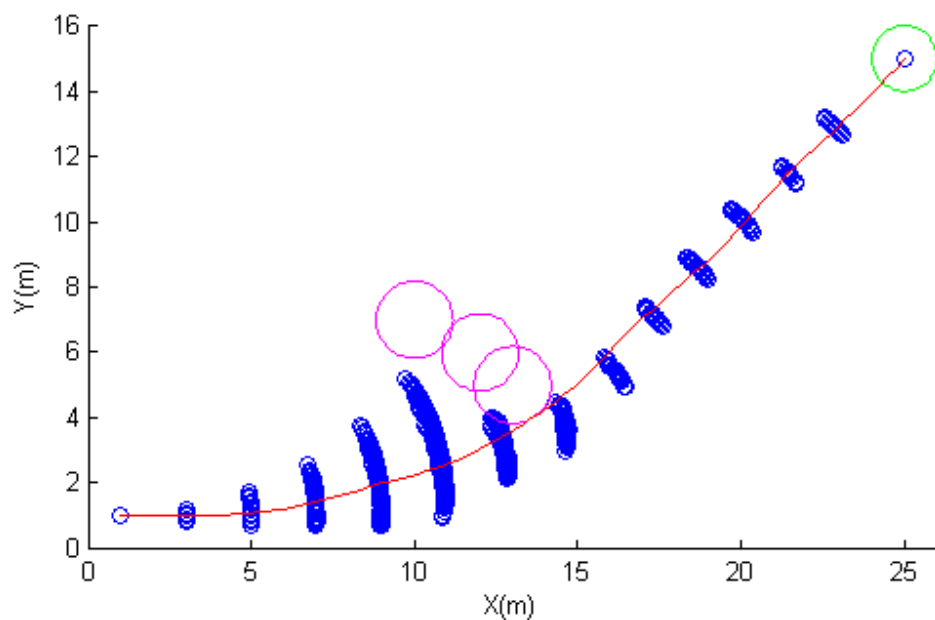


Figure 9.7: The Shortest path for simple obstacle set with constant speed and $r_{min} = 11$ m.

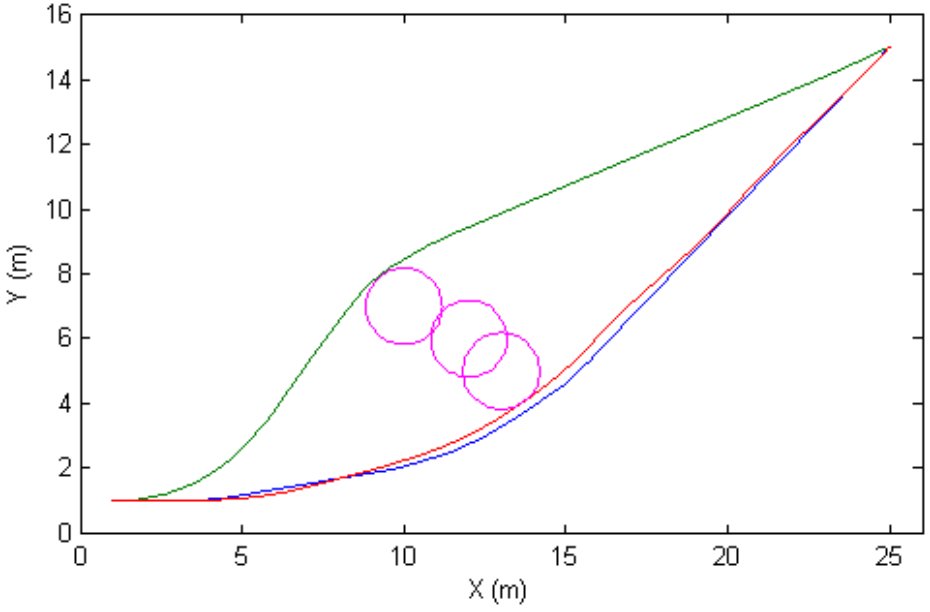


Figure 9.8: Comparison of the paths of the optimization schemes with simple obstacleset and constant speed. Blue=energy, green=time($r_{min} = 5$ m), red=time($r_{min} = 11$ m).

Optimization scheme Parameters	Time ($r_{min} = 5$)	Time ($r_{min} = 11$)	Energy
Distance travelled [m]	28.56	29.04	29.24
Vehicle runtime [s]	28.50	28.98	29.27
Estimated energy consumption [J]	2314	1759	1498
Actual energy consumption [J]	2470	1822	1655
Actual/Estimated ratio	107%	104%	110%
Number of nodes	2422	669	3095
Number of arcs	5347	2365	5722
Algorithm runtime [s]	47.15	8.39	61.14

Table 5: Comparison of the optimization schemes with simple obstacles and constant speed=1 m/s.

9.2.2 Speed=1.0-1.4 m/s

With variable speed enabled, we once again see a massive difference in number of nodes created and subsequent computation times in table 6. Both of the time schemes terminate in less than 100 seconds, while the distance and energy schemes need over 10 times as long.

We see that the $\text{time}(r_{\min} = 11 \text{ m})$ and energy schemes choose to drive on the right side of the obstacles, and have much better energy efficiency ($>25\%$) than the schemes choosing the left side. The energy scheme has an estimated path that is $\sim 12\%$ more energy efficient than the $\text{time}(r_{\min} = 11 \text{ m})$ scheme, but due to the aforementioned bad estimate ratio of the energy scheme, the actual energy consumption is actually $\sim 8\%$ lower for the time scheme than for the energy scheme. As for the driving time, this is naturally lower for the time scheme, by almost 15%.

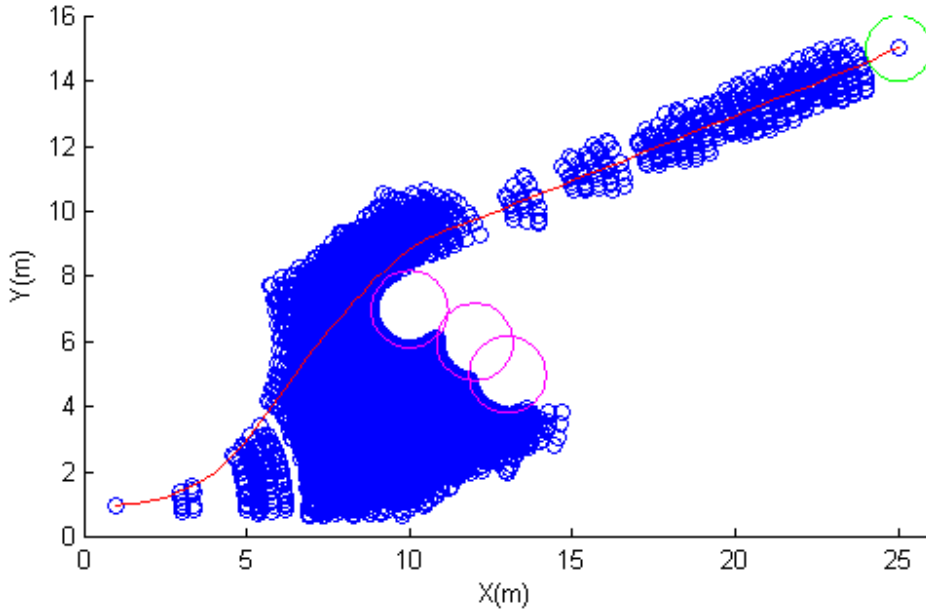


Figure 9.9: The Shortest path for simple obstacle set with variable speed.

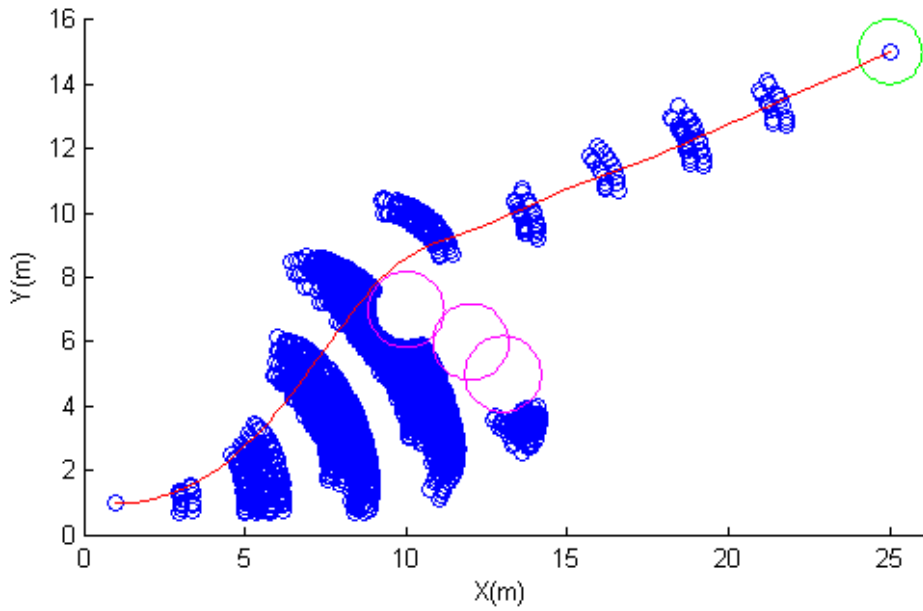


Figure 9.10: The Quickest path for simple obstacle set with variable speed and $r_{min} = 5$ m.

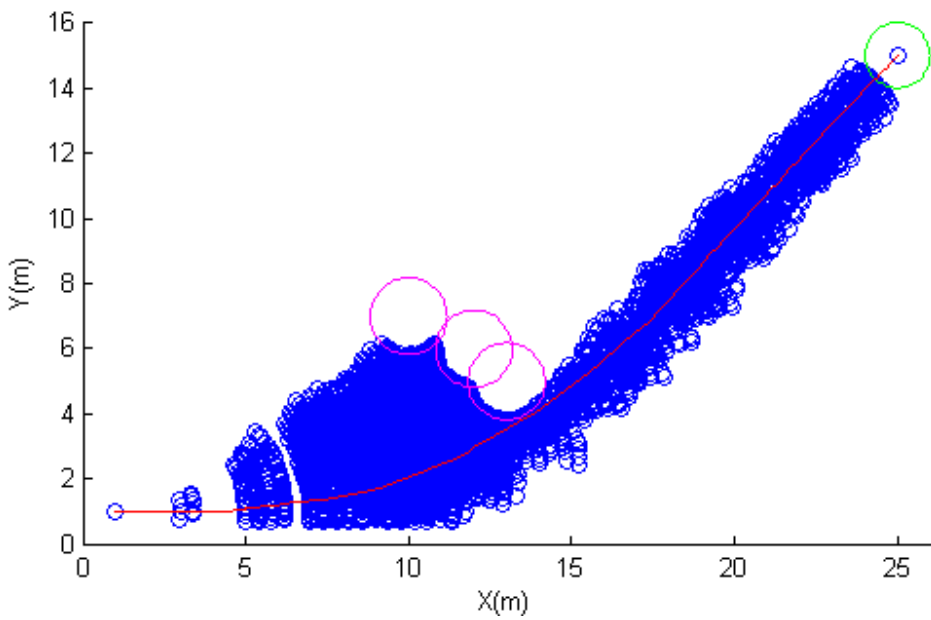


Figure 9.11: The Easiest path for simple obstacle set with variable speed.

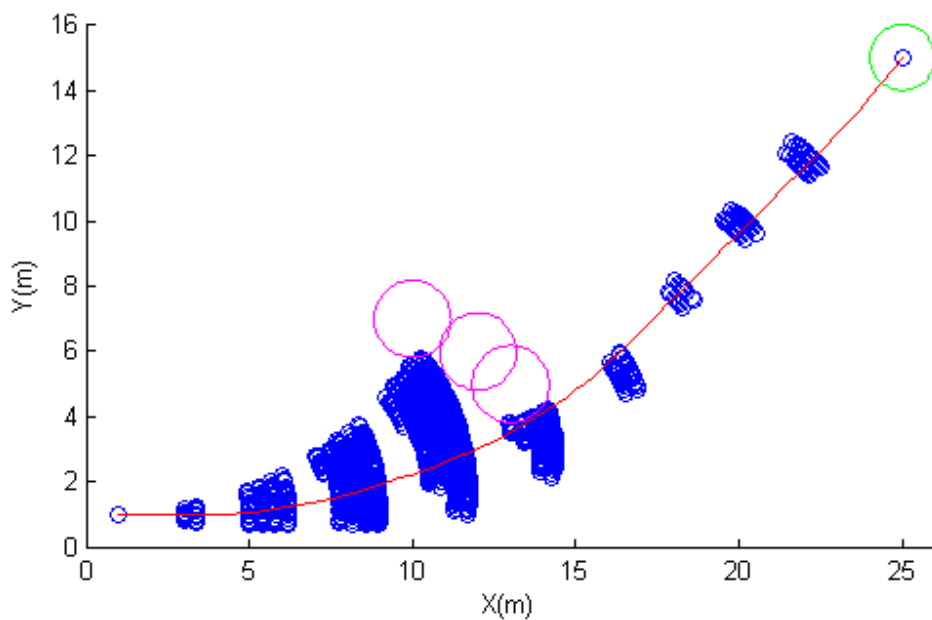


Figure 9.12: The Quickest path for simple obstacle set with variable speed and $r_{min} = 11$ m.

Opt. scheme Par.	Distance	Time ($r_{min} = 5$)	Time ($r_{min} = 11$)	Energy
Distance travelled [m]	28.44	28.51	28.99	29.09
Vehicle runtime [s]	22.81	20.64	20.96	24.50
Est. energy cons. [J]	2189	2359	1526	1349
Act. energy cons. [J]	2468	2569	1683	1827
Actual/Estimated ratio	113%	109%	110%	135%
Number of nodes	11261	3909	1918	13774
Number of arcs	41516	6295	5350	23106
Algorithm runtime [s]	2773	79.01	37.00	1315

Table 6: Comparison of the optimization schemes with simple obstacles and speed 1.0-1.4 m/s

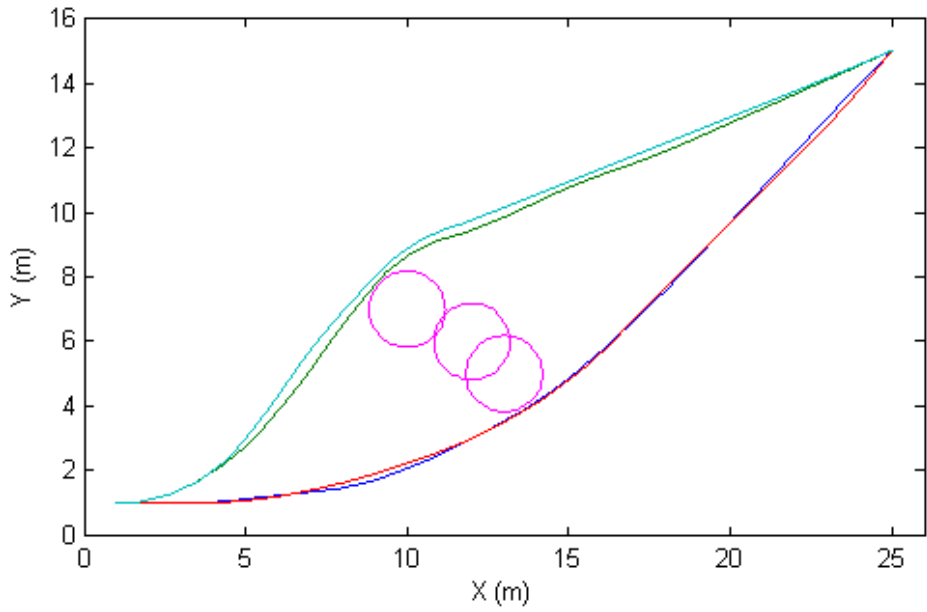


Figure 9.13: Comparison of the paths of the optimization schemes with simple obstacle set and variable speed. Blue=energy, green=time($r_{min} = 5$ m), red=time($r_{min} = 11$ m), cyan=distance.

9.3 Case 3: Complex cluster of obstacles

9.3.1 Constant speed=1.0 m/s

Once again, the table values for time apply to optimizing with respect to distance as well, since the speed is constant. All in all, the differences between the algorithms display the same behaviour as in the constant speed case with a simpler obstacle set in section 9.2.1.

The main point of this case was to compare it to the findings of [29], and if we compare the energy scheme with the time($r_{min} = 5$ m) scheme in table 7, we see a very similar behaviour:

The distance/time algorithm has a much longer computational time than the energy algorithm. If we compare figure 9.14 to figure 9.15, we see that while the energy scheme only searches the outer path, the distance scheme searches both the inner path and a large part of the outer path. The inner path is also obviously more complex.

Just like in [29], we also see that the energy scheme computes a somewhat longer path that demands considerably less energy. In fact the difference in our case is even more extreme: In [29], a 6.48% increase in distance led to a 34.6% decrease in energy consumption, while in our corresponding case, a 3% increase in distance led to a 58% decrease in energy consumption.

When comparing to the time($r_{min} = 11$ m) scheme instead, we see that the difference is far less pronounced, with a difference in estimated energy consumption of $\sim 12\%$ and a difference in actual energy consumption of less than 5%. The computational time of time($r_{min} = 11$ m) is even faster than for the energy scheme.

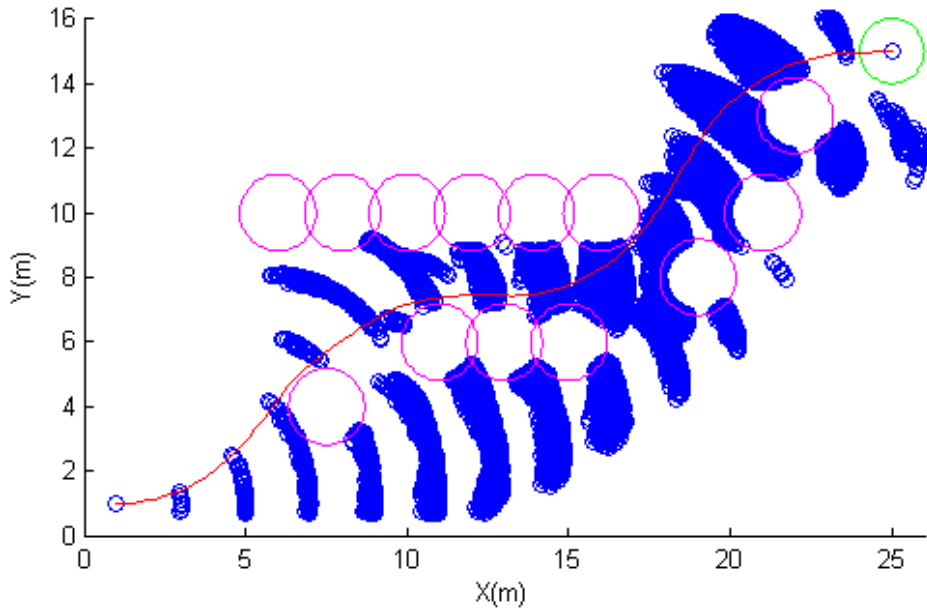


Figure 9.14: The Quickest path for complex obstacle set with constant speed and $r_{min} = 5$ m.

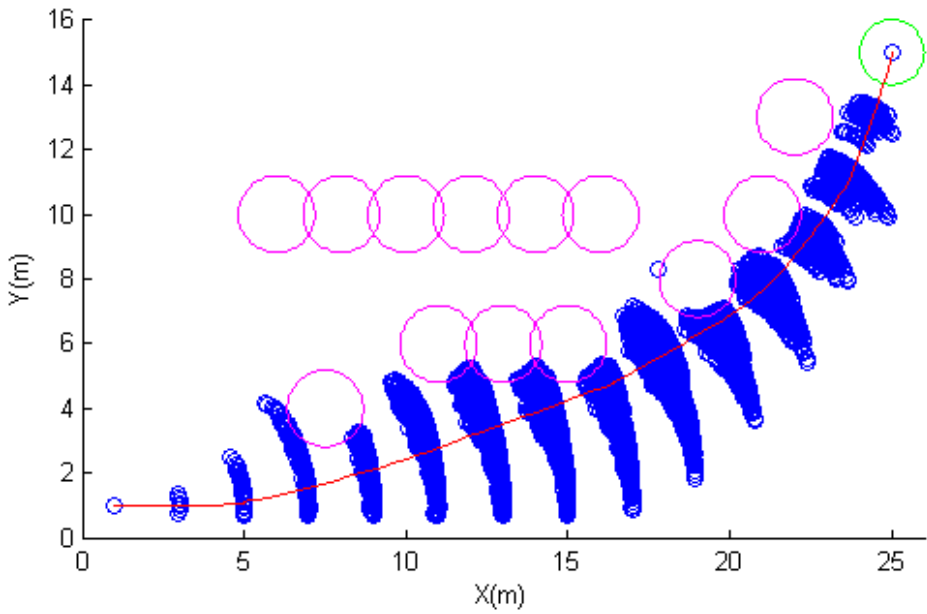


Figure 9.15: The Easiest path for complex obstacle set with constant speed.

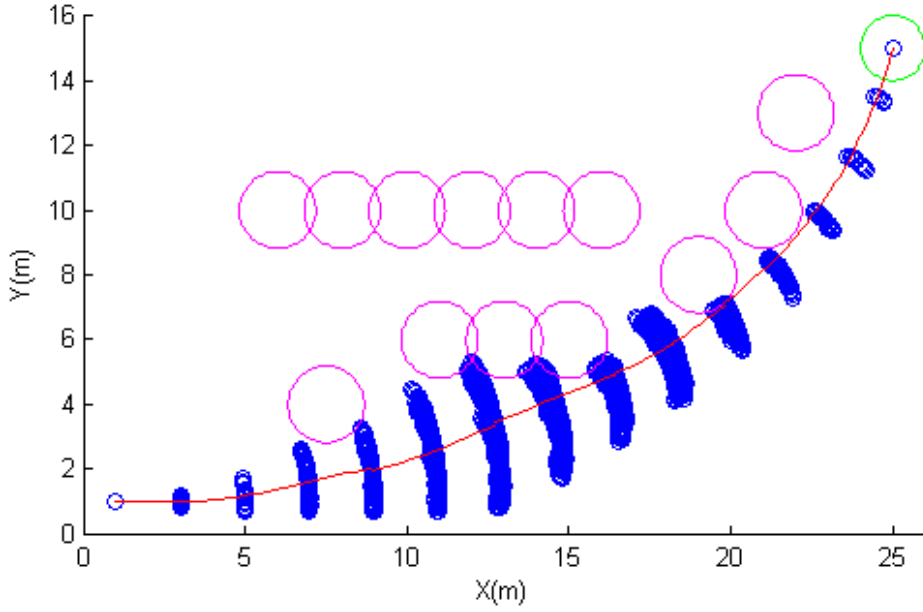


Figure 9.16: The Quickest path for complex obstacle set with constant speed and $r_{min} = 11$ m.

Optimization scheme Parameters	Time ($r_{min} = 5$)	Time ($r_{min} = 11$)	Energy
Distance travelled [m]	29.40	29.73	30.25
Vehicle runtime [s]	29.45	29.61	30.27
Estimated energy consumption [J]	4131	1880	1658
Actual energy consumption [J]	4564	2026	1932
Actual/Estimated ratio	110%	108%	117%
Number of nodes	12505	2193	7255
Number of arcs	35070	8900	16653
Algorithm runtime [s]	2080	101.7	462.3

Table 7: Comparison of the optimization schemes with complex obstacles and constant speed=1 m/s.

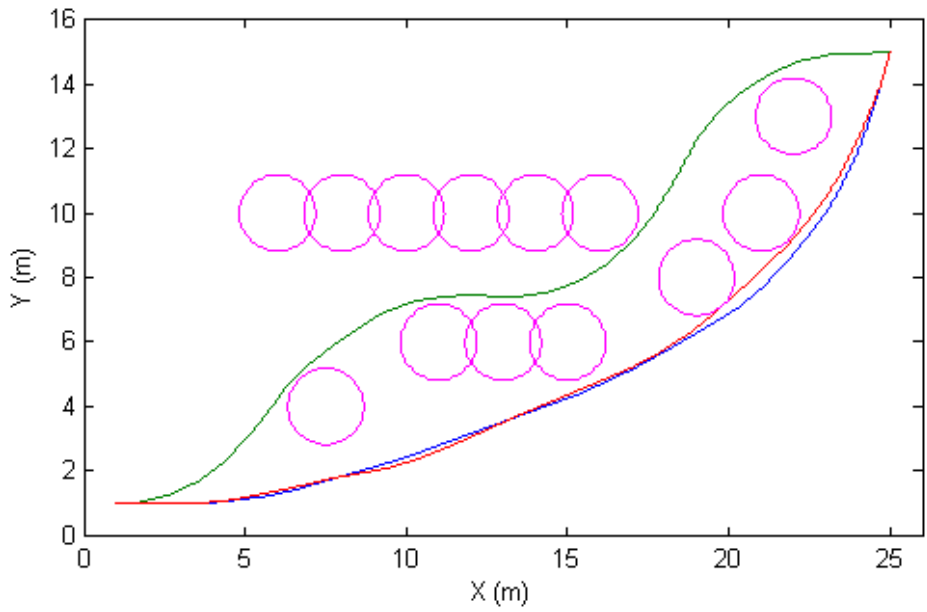


Figure 9.17: Comparison of the paths of the optimization schemes with complex obstacle set and constant speed. Blue=energy, green=time($r_{min} = 5$ m), red=time($r_{min} = 11$ m).

9.3.2 Speed=1.0-1.4 m/s, gridsize=0.2 m

The gridsize for position had to be increased to 0.2 m for the energy optimization to terminate within reasonable time. Even with this decrease in resolution, the distance optimization did not terminate within reasonable time. Since it has already been established in the previous cases that the distance optimizations behave like the time optimizations, only with longer algorithm runtimes, the distance optimization is omitted in the remaining subcases.

Again, we recognize the same relationships between the optimization schemes as in the previous variable-speed cases. The energy scheme has an estimated path that is $\sim 18\%$ more energy efficient than the time ($r_{min} = 11$ m) scheme, but due to the now even worse estimate ratio of the energy scheme, the actual energy consumption is $\sim 2.5\%$ lower with the time scheme than with the energy scheme. The driving time for the time scheme is once again almost 15% lower than the energy scheme, and the computational time is much faster.

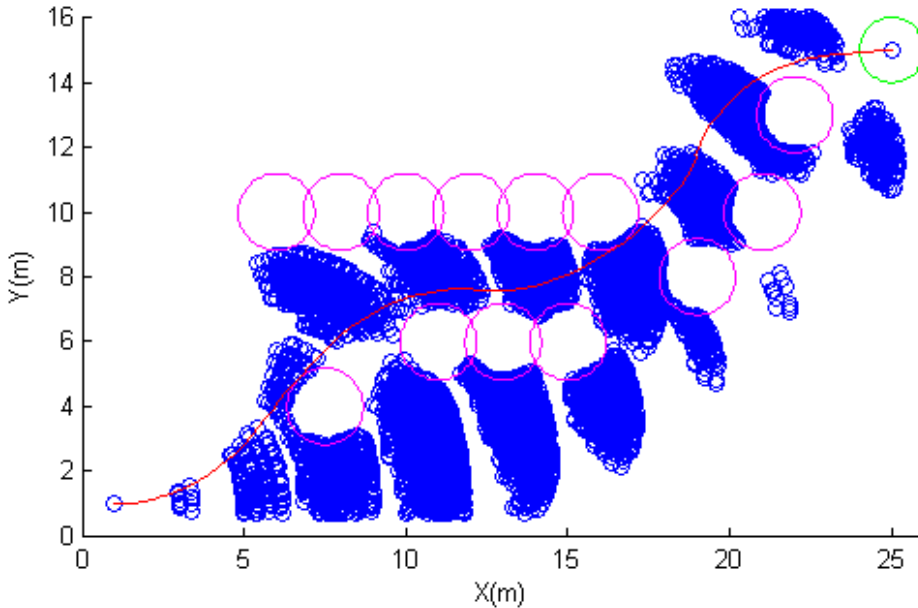


Figure 9.18: The Quickest path for complex obstacle set with variable speed, gridsize=0.2 m and $r_{min} = 5$ m.

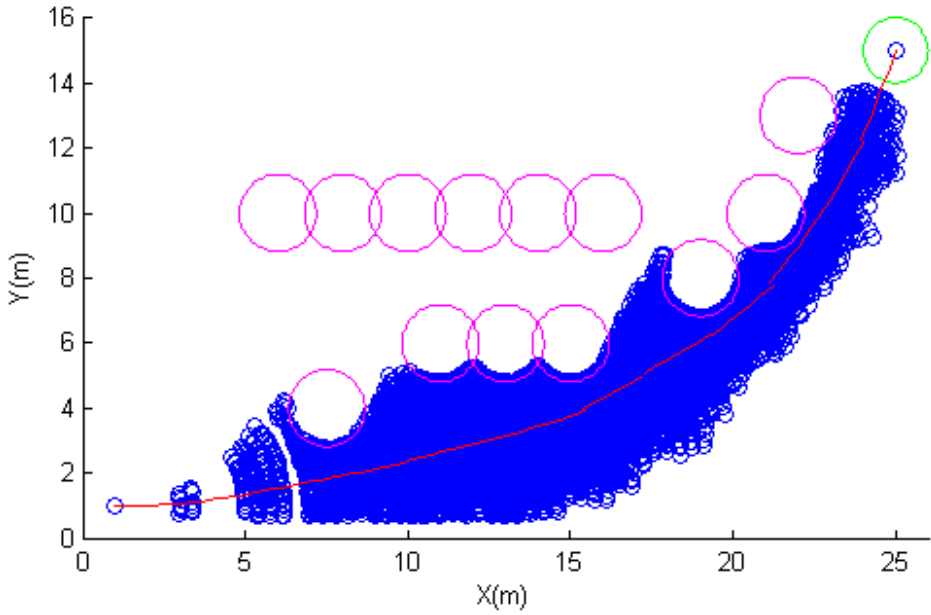


Figure 9.19: The Easiest path for complex obstacle set with variable speed and $\text{gridsize}=0.2$ m.

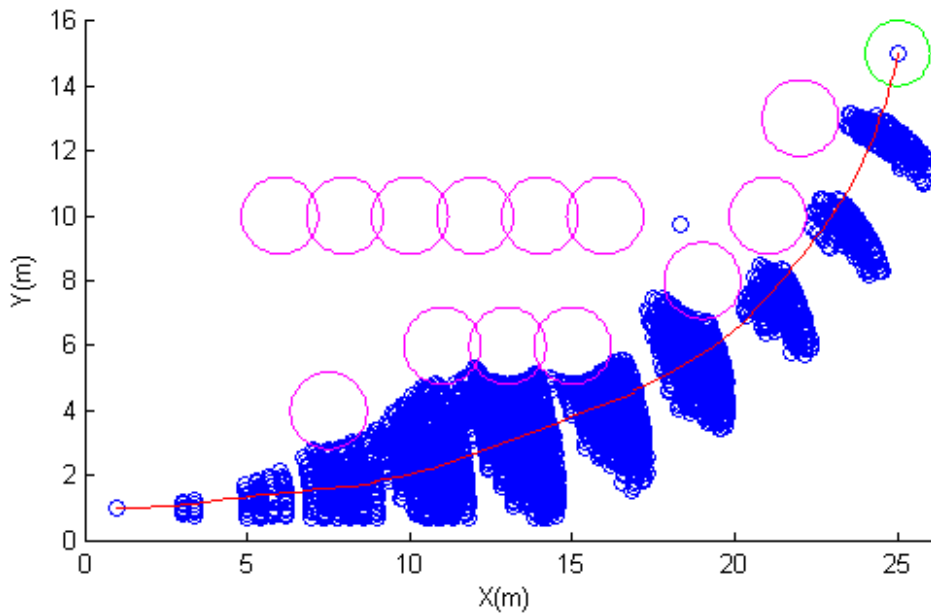


Figure 9.20: The Quickest path for complex obstacle set with variable speed, $\text{gridsize}=0.2$ m and $r_{\min} = 11$ m.

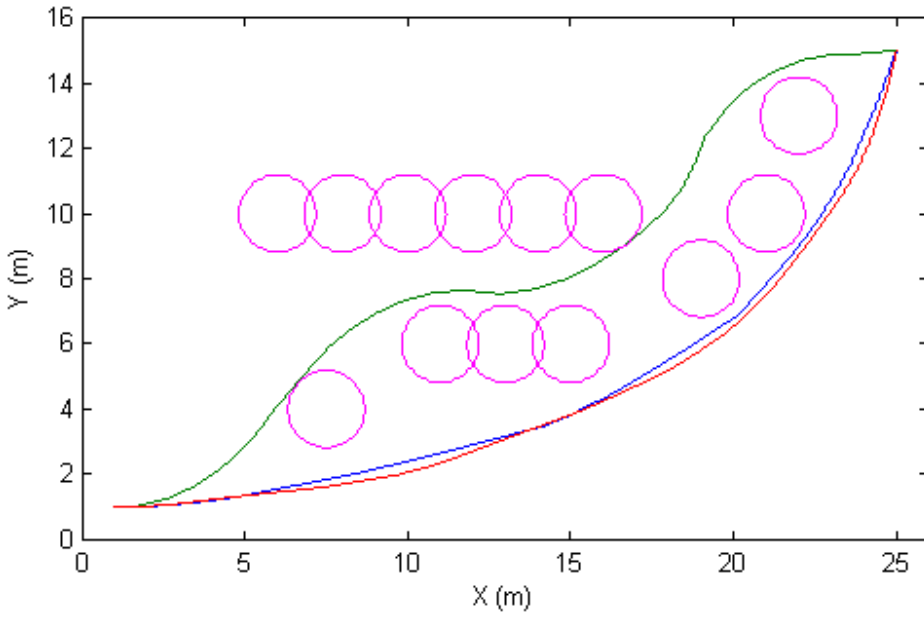


Figure 9.21: Comparison of the paths of the optimization schemes with complex obstacle set variable speed and gridsize=0.2m. Blue=energy, green=time($r_{min} = 5$ m), red=time($r_{min} = 11$ m).

Optimization scheme Parameters	Time ($r_{min} = 5$)	Time ($r_{min} = 11$)	Energy
Distance travelled [m]	29.45	29.93	29.84
Vehicle runtime [s]	21.30	21.59	25.16
Estimated energy consumption [J]	4050	1767	1447
Actual energy consumption [J]	4352	2040	2092
Actual/Estimated ratio	107%	115%	145%
Number of nodes	9790	3915	16579
Number of arcs	28470	20228	61341
Algorithm runtime [s]	1370	587.8	6995

Table 8: Comparison of the optimization schemes with complex obstacles, speed 1.0-1.4 m/s and gridsize 0.2m.

9.3.3 Speed=1.0-1.4 m/s, gridsize=0.1 m

Optimizing with respect to time could still terminate with the original resolution, and is therefore presented here in order to compare it to the corresponding values with lower grid resolution.

While we do see some minor improvements in the performance of the paths, the nodecount and algorithm runtime basically doubles, indicating that the increased resolution may not be worthwhile.

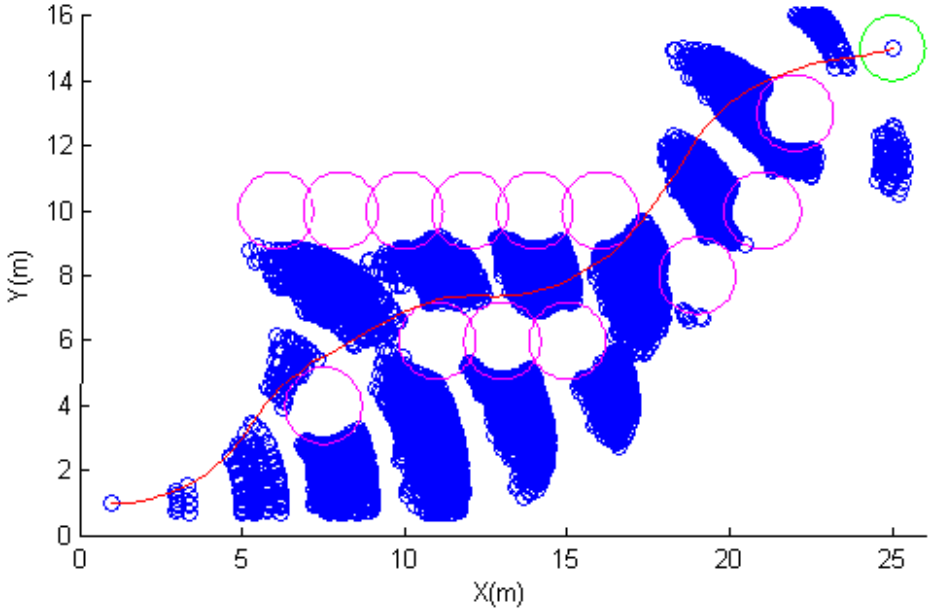


Figure 9.22: The Quickest path for complex obstacle set with variable speed, gridsize=0.1 m and $r_{min} = 5$ m.

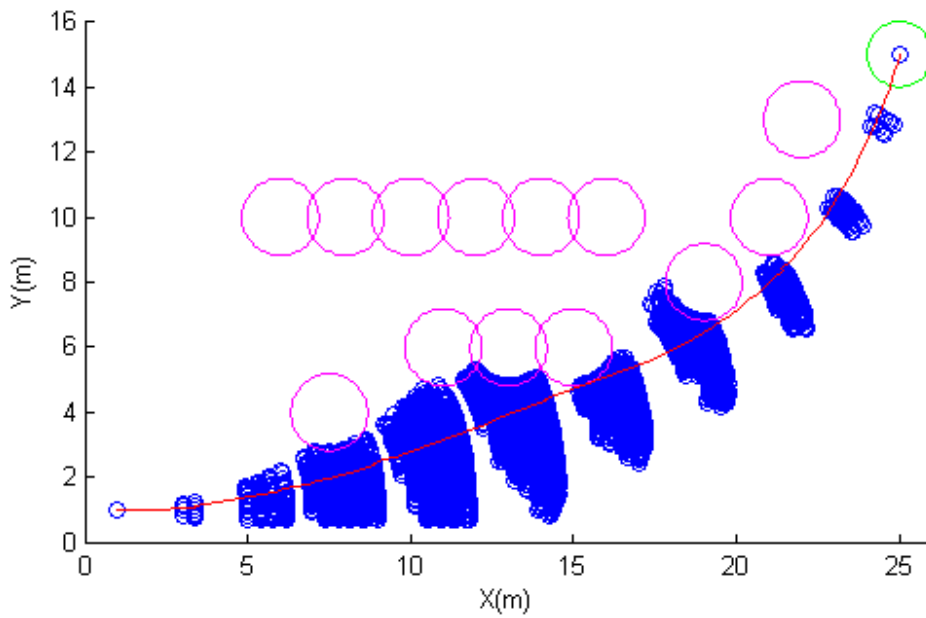


Figure 9.23: The Quickest path for complex obstacle set with variable speed, gridsize=0.1 m and $r_{min} = 11$ m.

Optimization scheme Parameters	Time ($r_{min} = 5$)	Time ($r_{min} = 11$)
Distance travelled [m]	29.15	29.67
Vehicle runtime [s]	21.08	21.42
Estimated energy consumption [J]	3947	1797
Actual energy consumption [J]	4312	2018
Actual/Estimated ratio	109%	112%
Number of nodes	16108	7635
Number of arcs	33236	31708
Algorithm runtime [s]	2562	1393

Table 9: Comparison of the optimization schemes with complex obstacles, speed 1.0-1.4 m/s and gridsize 0.1 m.

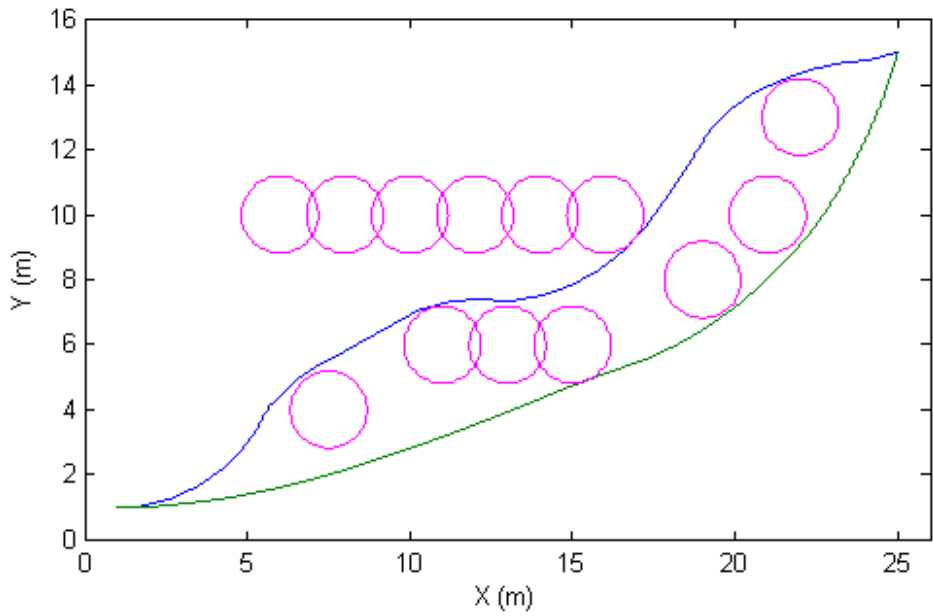


Figure 9.24: Comparison of the paths of the optimization schemes with complex obstacle set variable speed and gridsize=0.1m. Blue=time($r_{min} = 5$ m), green=time($r_{min} = 11$ m).

10 Discussion

The original task received from nLink AS was very open and without any particular guidelines on what would be an optimal path, which approach should be used to find it, and how quickly it needed to be computed. As a result of this, the main purpose of this report became to gather relevant information, test different approaches, and review which approaches that seem worth to look further into.

10.1 Choice and implementation of the path planning algorithm

The positive results achieved in [29] was the primary reason for choosing the SBMPO-algorithm. While the simulations in section 9 do confirm these findings, they also demonstrate some weaknesses to the approach, particularly with respect to excessive node generation.

10.1.1 Implementing the SBMPO-algorithm

The programming effort required to implement the SBMPO-algorithm proved to be much more monumental than anticipated. The 29 lines of pseudocode became 1000 lines of MATLAB code before the algorithm was operational.

Implementing the algorithm in MATLAB as opposed to a lower-level language like C++ came with both pros and cons. In the development phase, the familiarity of the MATLAB language, the useful function libraries and the instant SIMULINK compatibility made the daunting programming task somewhat more manageable.

In the testing phase, however, the slow performance of the MATLAB-implemented algorithm made testing with many different parameter modifications a very time consuming task. Any hope of testing with dynamic obstacles was also eradicated. Of course, the initial planning of the graph will take longer time than the subsequent replannings, but the performance of the MATLAB-implementation is still far too slow.

10.1.2 Controller performance

From [2], it was already known that the implemented controller of the SIMULINK model was far from perfect. Unfortunately, the requirement of a smooth trajectory had not been considered when deciding to keep working with this controller.

The results of the simulation clearly shows that the controller is unable to achieve the energy efficiency the algorithms estimate, particularly in the case of energy optimization. How much of this difference in energy consumption can be attributed to the controller using more power than necessary, and how much can be attributed to the algorithm giving an underestimate of the true cost, is uncertain, but both definitely have an effect.

One possible addition to the energy cost function that should be considered is the cost of acceleration. In the current implementation, the algorithm assumes that the vehicle instantly changes speed and angular velocity from one arc to the next without any cost. Obviously, with a large, heavy robot, this is not the case, and adding a term corresponding to overcoming the moment of inertia might improve the estimates.

Adding this estimate for the change in speed should be trivial with the current implementation, since the start speed of a node is part of the state of the node. However, for the change in angular velocity it might require adding the initial angular velocity to the state of the node, which would lead to an extra dimension in the state grid, causing even more nodes, which is not a desired outcome.

10.2 Comparison of the optimization schemes

We have been looking at four different optimization schemes: Distance, time($r_{min} = 5$ m), time($r_{min} = 11$ m) and energy.

10.2.1 The findings of Collins et.al.[29]

As mentioned before, the positive results achieved in [29] was a major factor in choosing the SBMPO-algorithm for the path planner, so naturally we would hope to replicate the results of this paper.

There are several differences in the approaches, notably the choice of programming language and simplified model of the vehicle. One difference that should be noted is the change in the heuristic function: [29] only used the cost of travelling in a straight line to the goal as the heuristic. When attempting to use this heuristic function with the setup in this paper, the algorithm would not reach the goal even in the case with no obstacle. The algorithm was unwilling to try nodes with any significant change in orientation, and ended up considering only the nodes more or less straight in front of it.

A possible reason why this was not an issue in the original article, may be the increased speed in this thesis. With a speed of 1.0 m/s as opposed

to 0.2 in [29], the α term of the power model in equation (5.9) becomes much more prominent, making turns much more costly. Adding the extra term for changing the orientation to face the direction of the goal caused the algorithm to behave as desired.

10.2.2 Energy vs Time with limited turning radius

The distance scheme, only included for comparison purposes, was shown to both have far too high computation times for variable speed and use excessive amounts of energy. We can therefore immediately exclude this scheme as an area of interest.

The unrestricted time scheme ($r_{min} = 5$ m) does have far more reasonable computation times, however, it too uses excessive amounts of energy, particularly in the case in section 9.3.

This leaves us with the time($r_{min} = 11$ m) and energy schemes.

It seems that the energy algorithm is prone to choose paths with many discontinuities. This might be because the cost of moving in a straight line is so much lower than turning, that the algorithm chooses a path that has many straight lines and exploits the discontinuities caused by the state grid to turn, rather than choosing arcs that actually turn. This will cause an artificially low estimate of the energy cost of a path.

The excessive amount of nodes and high computational times of the energy scheme demonstrate its inability to handle variable speed in the current implementation. If the energy scheme is to be implemented as a variable speed planner, some way of biasing nodes with higher speeds needs to be implemented. One approach might be to have a combined time-and-energy scheme where the start distance g and the heuristic h are both a weighted sum of the start distances and heuristics of the time and energy schemes. This scheme should in theory give a tunable compromise between speed and efficiency.

The time($r_{min} = 11$ m) scheme does seem to present itself as a very reasonable alternative with good computation time, low driving time, and very reasonable energy efficiency. The major drawback is that the self-imposed limitation on the turning radius may cause the algorithm to not find a path if the only possible route demands a sharp turn. If this scheme is to be used, some form of exception routine needs to be defined, so that the robot may use sharper turns if the obstacles require it.

Another possibility for a modification of the time scheme is to have a maximum power limit instead of a minimum radius limit. This would allow the vehicle to perform sharper turns if it slows down first. This

would probably not save much energy compared to making the turns at full speed, since the vehicle would take a longer time to make the turn. But since the scheme optimizes with respect to time, the arcs with lower speeds and sharp turns should only be chosen when all full-speed options fail. This will obviously require some testing and verification, but should not be too difficult to implement when the main algorithm is already implemented.

Since the primary objective of the system is to work as fast as possible, time is our primary concern and energy consumption is only interesting with regards to the extra downtime the charging or switching of batteries will cause. The battery capacity, the charging time and the energy demands of the rest of the system are all factors that will need to be determined before deciding how much the energy efficiency of the path should count.

Also, it is apparent that the structure of the obstacle set has a large impact on how the different optimization schemes perform. The obstacles in [29] and subsequently this thesis are designed to favor the energy algorithm. Some other sets that are more randomized, or preferably illustrate actual cases that the robot is likely to encounter, should be tested with the different schemes before deciding on which scheme to implement.

10.3 Choosing a different algorithm

The SBMPO-algorithm in its currently implemented form does not seem like it is ideal for the given task. It requires substantial modifications if it is to handle dynamic obstacles.

10.3.1 Possible modification: D* Lite

Switching the LPA* part of the SBMPO-algorithm with a D*-Lite approach should make replanning much simpler and less time consuming.

10.3.2 Completely different approach?

After seeing how good the time scheme with limited turning radius performed compared to the energy scheme, one can not help but wonder if the SBMPO-algorithm is a bit overkill for this task. When the project is to be implemented in a prototype, the entire algorithm will have to be implemented in a different programming language. Whether the performance of the SBMPO-algorithm is good enough to warrant the large programming

effort required to implement it, will have to be assessed, and looking into some simpler algorithms might be a good use of time.

11 Conclusion

This project has described a concept for a tilesetting robot currently under development. A mathematical model of a 4-wheel skid-steered vehicle has been presented, along with several different path planning algorithms.

A complete simulator of the model of the skid-steered vehicle has been constructed in SIMULINK with a Sampling-Based Model Predictive Optimization algorithm for path planning. A series of simulations has compared the performance of the algorithm with different optimization criteria: Distance, time and energy.

Optimization with respect to energy consumption does present some very promising results, particularly for fixed-speed operation. However, the current implementation is clearly unable to handle varying speed. With variable speed, the method generates an excessive amount of nodes, with a correspondingly high computation time.

The optimization strategy that performed the best was based on time with a lower limit on the turning radius. This path planning scheme produced paths that were fast, reasonably energy efficient and had by far the fastest computation times. It does have some drawbacks, since it has a higher risk of not finding a feasible path if sharp turns are required.

From this, we conclude that the ideal path planning strategy needs to incorporate elements from both time optimization and the energy optimization. Possible candidates include a time optimization with a maximum limit on power consumption, and an optimization based on a weighted compromise between time and energy consumption.

12 Further work

Even though the SBMPO-algorithm with limited turning radius and optimization with respect to time has been shown to be able to plan fast and efficient paths for the vehicle with good computational runtimes, some modifications need to be made and tested. The most obvious is the need to use a faster programming language than MATLAB, but the already implemented MATLAB code might be useful for some initial testing. Some of the possible modifications include:

1. Improving the model and controller by building on the work of [11].
2. Rewrite the time optimization to utilize a maximum power limit rather than a minimum radius limit.
3. Combine the time and energy optimization to an optimization scheme that could result in both low computational time and energy efficient paths.
4. Switching the LPA* part of the SBMPO-algorithm to instead use the D*-Lite algorithm from section 4.3.2, which would simplify fast replanning for dynamic obstacles.
5. Adding a criteria for the final orientation. A simple solution for this is to add three virtual obstacles surrounding the goal: If the vehicle is to face north, virtual obstacles to the north, east and west would ensure the final path approaches from the south.
6. Replacing the SBMPO-algorithm with a simpler algorithm, while maintaining the principle of avoiding sharp turns in order to increase energy efficiency.

When the system is nearing the prototyping stage, it might be a good idea to add an online estimator for the power consumption for different turning radii and speeds. Thus, the system would be able to continually update the power lookup tables and improve the accuracy of them. Keep in mind that different concrete floors will have different friction properties, and the wear and tear of the wheels will also impact this.

Eventually, the tiling system should be expanded to have several robots tiling the same area. Implementation of multiple robots will mean we will be adding a new obstacle for which we know the planned trajectory. The current approach will be able to avoid colliding with any obstacles, but since the robots know eachothers trajectories, they should exploit this

knowledge to find a more optimal trajectory. A scenario where this would be particularly useful, could for instance be the scenario where two robots meet head on, one from the east and one from the west, and they both intend to turn north.

In Sogndal, nLink are currently working with a newly acquired robotic arm from Universal robots[35] on a separate project. This will be used as a testing platform for different approaches for the tilesetting itself.

Bibliography

- [1] Prweb. global ceramic tiles market to reach 92.78 billion square feet by 2015, according to a new report by global industry analysts, inc. <http://www.prweb.com/releases/ceramic/tiles/prweb4447044.htm>. Accessed 2012.09.11.
- [2] S. M. Nornes. Modeling and simulation of a mobile platform for a tilesetting robot. Department of Engineering Cybernetics, Norwegian University of Science and Technology, 2012.
- [3] D. Apostolopoulos, H. Schempf, and J. West. Mobile Robot for Automatic Installation of Floor Tiles. *Robotics Inst., Carnegie Mellon Univ., Pittsburgh, USA*, 1996.
- [4] R. Navon. Process and quality control with a video camera, for a floor-tilling robot. *Dept. of Civil Engineering, Technion, Israel Inst. of Technology, Haifa, Israel*, 1999.
- [5] A. Oral and E.P. Inal. Marble mosaic tiling automation with a four degrees of freedom cartesian robot. *Mech. Eng. Dept., Balikesir Univ., Balikesir, Turkey*, 2009.
- [6] http://en.wikipedia.org/wiki/Preferred_walking_speed.
- [7] http://en.manu-systems.com/SegwayRMP_LE_and_SE_2012_web.pdf.
- [8] N. Tlale and M. de Villiers. Kinematics and dynamics modelling of a mecanum wheeled mobile platform. *Council for Sci. & ind. Res., Pretoria, South Africa*, 2008.
- [9] D. Song J. Yi, J. Zhang, and Z. Goodwin. Adaptive trajectory tracking control of skid-steered mobile robots. *Dept. of Mech. Eng., San Diego State Univ., CA, USA*, 2007.
- [10] L. Caracciolo, A. De Luca, and S. Iannitti. Trajectory tracking control of a four-wheel differentially driven mobile robot. *Dipt. di Inf. e Sistemistica, Rome Univ., Italy*, 1999.
- [11] K. Kozlowski and D. Pazderski. Modeling and control of a 4-wheel skid-steering mobile robot. *Inst. of Control & Syst. Eng., Poznan Univ. of Technol., Poland*, 2004.

- [12] J.Y. Wong. *Theory of Ground Vehicles*. John Wiley, New York, 1978.
- [13] O. Egeland and J.T. Gravdahl. *Modeling and Simulation for Automatic Control*, pages 191–199. Marine Cybernetics AS, Trondheim, Norway.
- [14] L. Ravanbod-Shirazi and A. Besançon-Voda. Friction identification using the karnopp model, applied to an electropneumatic actuator. *Lab. d'Automatique de Grenoble, ENSIEG, Saint Martin d'Heres, France*, 2003.
- [15] A. Isidori. *Nonlinear Control Systems*. Springer-Verlag, London, 3rd edition, 1995.
- [16] <http://en.manu-systems.com/RMP-2316100001.shtml>.
- [17] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.
- [18] S. Koenig and M. Likhachev. Incremental a*. *Proceedings of the Neural Information Processing Systems*, 2001.
- [19] M. Likhachev S. Koenig and D. Furcy. Lifelong planning a*. *Artificial Intelligence*, v 155, n 1-2, p 93-146, 2004.
- [20] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *IEEE TRANSACTIONS ON ROBOTICS*, VOL. 21, NO. 3, 2005.
- [21] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21, 1996.
- [22] C. V. Caldwell D. D. Dunlap and E. G. Collins Jr. Nonlinear model predictive control using sampling and goal-directed optimization. *Proceedings of the IEEE International Conference on Control Applications*, 2010.
- [23] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 1960.
- [24] Ø. A. G. Loe. Collision avoidance for unmanned surface vehicles. Master's thesis, NTNU, 2008.

- [25] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Computer Science Dept., Iowa State University, 1998.
- [26] C. Urmson and R. Simmons. Approaches for heuristically biasing rrt growth. *IEEE International Conference on Intelligent Robots and Systems*, v 2, p 1178-1183, 2003.
- [27] Ø. A. G. Loe. Collision avoidance concepts for marine surface craft. Norwegian University of Science and Technology, 2007.
- [28] M. Likhachev and A. Stentz. R* search. *AAAI*, pp. 1–7, 2008.
- [29] N. Gupta A. Sharma and E. G. Collins Jr. Energy efficient path planning for skid-steered autonomous ground vehicles. *Proceedings of the SPIE - The International Society for Optical Engineering*, v 8045, 2011.
- [30] W. Yu O. Chuy Jr., E. G. Collins Jr. and C. Ordonez. Power modeling of a skid steered wheeled robotic ground vehicle. *Proceedings - IEEE International Conference on Robotics and Automation*, p 4118-4123, 2009.
- [31] T. Andrews. Computation time comparison between matlab and c++ using launch windows. *California Polytechnic State University San Luis Obispo*, 2012.
- [32] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics* 79 (3): 497–516, 1957.
- [33] http://www.roymech.co.uk/Useful_Tables/Tribology/co_of_frict.htm.
- [34] http://rmp.segway.com/downloads/RMP_400_Specsheet.pdf.
- [35] http://media1.limitless.dk/UR_Tech_Spec/UR10_GB_2012.pdf.