

Pre-planning Intersection Traversal for Autonomous Vehicles

Master's Thesis in Computer Engineering

Ian Dahl Oliver

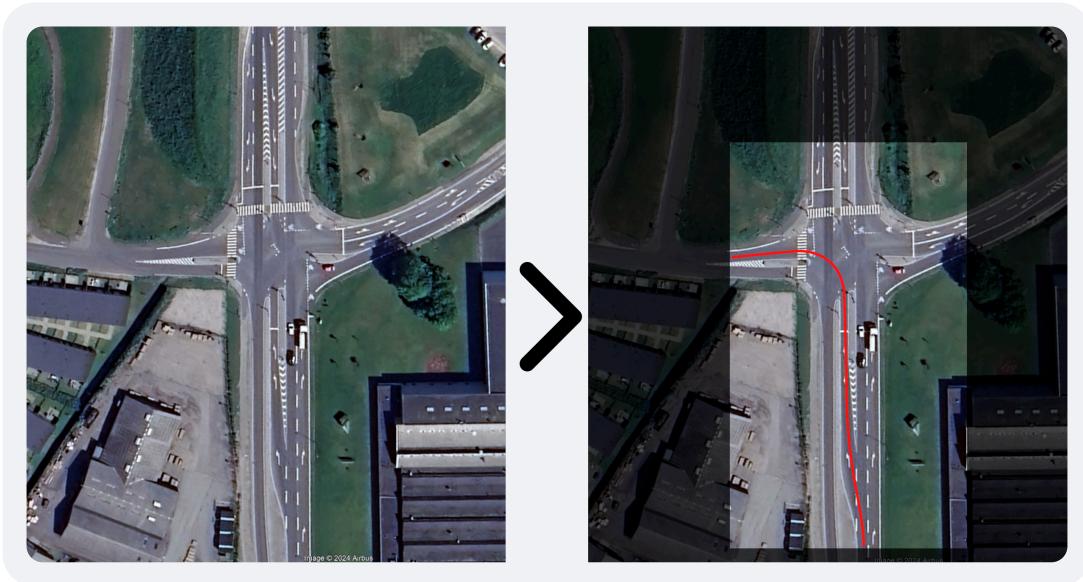
Department of Electrical and Computer Engineering

Aarhus University

Aarhus, Denmark

ian.oliver@post.au.dk

19-02-2025



Supervisor: Lukas Esterle

lukas.estlerle@ece.au.dk



**AARHUS
UNIVERSITY**

Preface

This master thesis is titled “*Pre-planning Intersection Traversal for Autonomous Vehicles*” and is devised by Ian Dahl Oliver. The author is a student at Aarhus University, Department of Electrical and Computer Engineering, enrolled in the Computer Engineering Master’s programme. The author has completed a Bachelor’s degree in Computer Engineering under the same conditions.

The thesis has been conducted in the period from 27-01-2025 to 05-06-2025, and supervised by Associate Professor Lukas Esterle. I would like to express my gratitudes to my supervisor for his support and advice throughout the project.

An additional thanks goes to Associate Professor at AU, Kaare Mikkelsen for his guidance in the early stages of this project.

All software developed in this thesis is released under the MIT license, and is provided as is without any warranty.

Enjoy reading,
Ian Dahl Oliver

Abstract

hello Robot Operating System 2 (ROS2)

Nomenclature

Some terminology and type setting used in this thesis may not be familiar to the reader, and are explained here for clarity.

`monospace`

- Inline monospace text is used for code function names, variables, or parameters.

`a.b`

- In inline monospace text, a period `.` is used to denote a method or property of an object. Can also be used outside of monospace text.

`listing:<int>`

- A reference to a specific listing, where `<int>` represents a line number.

`listing:<int>-<int>`

- Reference to a range of lines within a listing.

`file.ext`

- A reference to a specific file of given file type.

`file.ext:<func>`

- A reference to a specific function within a file.

Acronyms Index

Acronym	Definition
AIM	Autonomous Intersection Management
APF	Artificial Potential Field
API	Application Programming Interface
AUV	Autonomous Underwater Vehicle
AV	Autonomous Vehicle
BEV	Bird's Eye View
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DL	Deep Learning
DOF	Degrees of Freedom
FCN	Fully Convolutional Network
FL	Fuzzy Logic
GA	Genetic Algorithm
GNN	Graph Neural Network
GUI	Graphical User Interface
ID	Identifier
JSON	JavaScript Object Notation
MLP	Multilayer Perceptron
NN	Neural Network
PNG	Portable Network Graphics
PRNG	Pseudo-Random Number Generator
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNG	Random Number Generator
ROS2	Robot Operating System 2
RRT	Rapidly-exploring Random Tree
TOML	Tom's Obvious Minimal Language
Acronym	Definition
UI	User Interface
UX	User Experience
ViT	Vision Transformer
YAML	YAML Ain't Markup Language

Contents

Preface	ii
Abstract	iii
Nomenclature	iv
Acronyms Index	v
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Research Questions	3
2 Background	4
2.1 Autonomous Vehicles	4
2.2 Deep Learning	4
2.3 Satellite Imagery	4
2.4 Path Drawing	4
2.5 Pose Estimation	4
3 Related Work	5
3.1 Path-planning	5
3.2 Intersection Management	6
4 Methodology	7
4.1 Satellite Imagery	7
4.2 Loss Function Design	11
4.3 Dataset Creation	11
4.4 Model Design	15
5 Results	16
6 Discussion	17
6.1 Integration with existing systems	17
6.2 Shortcomings	17
6.3 Future Work	17
7 Conclusion	18
References	19
Appendix	vi

Introduction

The introduction to the thesis will be structured as follows. First, [Section 1.1](#) presents the motivation for this thesis with the problem statement following in [Section 1.2](#). With the motivation and problem statement in place, the research questions will be presented in [Section 1.3](#).

1.1 Motivation

Since the dawn of time, humans have made strides in automating any and all systems that surround them. In the days before technology, humans used animals to help them with their daily tasks. As technology advanced and exploded during the Industrial Revolution, humans replaced animals with machinery with the intent of automating production tasks. As this mentality continued to grow and technologies improved at a rapid pace, automation spread to other areas of life, such as transportation. The first Autonomous Vehicles (AVs) were developed in the 1980s by both Americans at Carnegie Mellon University [1], [2] and Europeans at Mercedes-Benz and Bundeswehr University Munich [3]. Since then, the development has spread to the individual car manufacturers instead of universities, making it more of a competitive field than a cooperative one. Still, the development in recent years with the rise of more powerful computers and the invention of machine learning, has led the field to become a fiercely researched area with many companies doing their part to create reliable, efficient, and safe AVs.

Despite this rapid development, AVs still encounter many challenges in their deployment, chief amongst which is their ability to handle intersections [4]. Unlike motorway driving — where lane following and obstacle detection are relatively well-defined tasks with few obstacles — intersection introduce a lot of complexity. Challenges arise from many different factors, such as unpredictable driver behaviour of other drivers, a huge variety of intersection types and configurations, and the state of the intersection with regards to faded or obstructed lane markings. Current solutions rely heavily on on-board sensors for perception and reactive decision-making, which can struggle in certain situations. Other solutions to intersections rely heavily on infrastructure support, such as V2X communication, which is not yet widely deployed.

A potential alternative to purely perception-based or infrastructure-dependent approaches is pre-planned path traversal, where an AV generates an optimal path through an intersection before reaching it. By leveraging Deep Learning (DL) models trained on annotated satellite imagery, AVs can gain a significantly increased amount of understanding

about the intersection it is about to enter. This presents many potential improvements to the capability and efficient of AVs. Firstly, it can reduce the reliance on on-board sensors by taking away the need for real-time perception and decision-making. Secondly, it can reduce the reliance on infrastructure support by allowing the AV to make decisions based on the pre-planned path. Finally, it can increase the safety of the AV by reducing the number of unpredictable situations it may encounter and it give it a fair chance when it then does encounter foreign situations.

Beyond improving the performance of individual AVs, better intersection handling has even broader implications to the public as a whole. Optimized intersection traversal can lead to smoother traffic flow, reduced congestion, and improved urban mobility. By generating more efficient paths, AVs can reduce waiting times, minimize unnecessary stops, and create a more predictable and coordinated traffic environment. As the proportion of autonomous vehicles on the road increases, these optimizations scale exponentially, leading to fewer bottlenecks and a decrease in fuel consumption and emissions.

1.2 Problem Statement

Advancements in AV technologies have been at the forefront of tech innovations in the 21st century. A key challenge in the development of fully autonomous vehicles, is their ability to handle intersections. Intersections pose a wide variety of challenges to AVs: from those posed by complex structures, to those posed by the unpredictability of human drivers, to faded lines that make it difficult for on-board computer vision system to clearly identify lanes or paths. All of these hinder AVs from reaching their full potential and being able to navigate intersections safely and efficiently.

Current existing solutions are very infrastructure-dependent. The Car2X system by Volkswagen, for example, relies on a network of sensors and communication devices installed in the infrastructure to spread information to vehicles on the road [5]. Autonomous Intersection Management (AIM) also relies on infrastructure to provide vehicles with information regarding intersections, with an orchestrator monitoring and managing individual intersections [6], [7], [8], with active development moving towards a more decentralized and distributed approach [9]. Furthermore, reliance on camera-based vision is susceptible to environmental limitations, such as adverse weather, that reduce system reliability.

The challenges posed by intersections cause major problems for AV developers who want to push fully autonomous driving. AVs' inability to properly react to and handle intersections, leads to significant delays in real-world deployment as a consequence of the unreliability experienced by regulators and the general public. If AVs want to enter the market with full self-driving capabilities, full autonomy is a key challenge to be tackled, as it is an essential task experienced when driving.

This project aims to develop a solution that will help AVs to better handle intersections. With the use of DL and Computer Vision (CV) technologies, trained on and utilizing satellite imagery, this project aims to train a model that can accurately identify the proper path for an AV to travel through an intersection. The system is not meant to replace current

systems deployed in AVs, but rather assist the existing systems make better decisions when in self-driving mode, approaching an arbitrary intersection.

1.3 Research Questions

The following research questions have been formulated to address key challenges in AV path planning at intersections. The questions are designed to explore the effectiveness of different approaches and models in generating accurate and efficient paths for autonomous vehicles. The research questions are as follows:

- RQ-1** How can pixel-subset-based deep learning approaches be optimized to improve accuracy and efficiency in path planning for autonomous vehicles at intersections?
- RQ-2** How do Convolutional Neural Networks (CNNs), Fully Convolutional Networks (FCNs), Vision Transformers (ViTs) and diffusion-based models compare in terms of achieving effective performance for pre-planned intersection traversal?
- RQ-3** Is it possible to design a loss function that effectively captures the similarity between generated and desired paths for autonomous vehicles without forcing exact matches?
- RQ-4** Is it possible to create a dataset that allows for the training of a model, such that the data is not too stringent to a singular path?

2

Background

This section outlines the theory relevant to the thesis. It begins with...

2.1 Autonomous Vehicles

2.2 Deep Learning

2.2.1 Computer Vision

2.2.2 Datasets

2.3 Satellite Imagery

2.4 Path Drawing

2.5 Pose Estimation

3

Related Work

3.1 Path-planning

Path-planning is the task of having a certain amount of knowledge about the environment and finding a path from a starting point to a goal point. This task is one of the most fundamental tasks in the field of robotics and autonomous navigation, and has thus has a long history of improvement and evolution.

One of the first algorithms to be used for path-planning is the Dijkstra algorithm from 1959 [10], which is a graph search algorithm that finds the shortest path between nodes in a graph. The A* algorithm is another popular algorithm from 1968 that is used for path-planning, and is a combination of Dijkstra's algorithm and a heuristic function that estimates the cost of the cheapest path from a node to the goal node [11]. Some years later, the D* algorithm was introduced in 1994, which is an incremental search algorithm that finds the shortest path between nodes in a graph, and is an improvement over the A* algorithm [12]. D* has since become a very popular algorithm for path-planning in robotics, with improved alternatives like Focused D* the year after [13] and D* Lite from 2005 [14] proving use in real-world applications.

The concept of Artificial Potential Fields (APFs) was introduced in 1986 [15]. It assumes some repulsive field around obstacles to avoid and a pulling force towards the goal, resulting in autonomous robots navigating towards the goal while avoiding obstacles. This method is particularly effective in dynamic environments, where the obstacles are moving. It is, however, very prone to local minima and situations where it might get trapped [16], significantly reducing its effectiveness in complex environments, such as a tight hallway where the robot might fit but the calculated repulsive force is too great or if it encounters a dead-end or U-shaped obstacle, leading it to loop infinitely. Combined with other global path-planning algorithms, it has shown considerable success, especially in the field of swarm robotics [17], [18].

The Rapidly-exploring Random Tree (RRT) algorithm was introduced in 1998 [19], and is a popular algorithm for path-planning in robotics. It is a randomized algorithm that builds a tree of possible paths from the starting point to the goal point, and is particularly useful in high-dimensional spaces. A node is randomly chosen from the initial point. The intermediate node is then determined based on the movement direction and maximum section length. If obstacles are detected, the route in that direction is ignored. Otherwise, a new random point is selected. The RRT* algorithm was introduced in 2011 [20], improving

on the original with two small but significant modifications: a cost function that takes into account the distance between nodes and a re-wiring step that allows the tree to be restructured to find a better path. It has shown great usage in real-world applications regarding Autonomous Underwater Vehicles (AUVs) [21], [22], despite challenges regarding the need for information about large areas [23].

Other areas of research in path-planning include Genetic Algorithms (GAs) and Fuzzy Logic (FL). GA [24] is inspired by the process of natural selection where only the fittest organisms survive. Generally the algorithm works by generating a random population of solutions, and then selecting the most efficient ones by using some cost function. Then these selected solutions go through the crossover process where they are combined and mutated to generate new solutions. FL is another old method from 1965 used for path-planning [25], [26]. It depends on functions used in fuzzification, inference, and defuzzification. These functions are based on a descriptive classification of the input data, such as low, medium, or high collision risk. Based on the defuzzification process, the robot decides on the best path to take.

Neural Networks (NNs) are also finding their usage in the field. NNs are made to imitate the human brain's innate ability to learn. They are trained on data and learn how to react to it. They are used in the field, not necessarily for path-planning explicitly, but more in conjunction with other algorithms that use their output as input. I.e. a NN might be able to tell the controller where some obstacle is, meaning it is giving a helping hand to algorithms like APF. Akin to APF, Reinforcement Learning (RL) models are taught to react to their surroundings, driving towards a goal and being rewarded and penalized for the actions that it takes, like how APF is moving towards a goal and avoiding obstacles due to the repulsive forces.

In summary, the evolution of path-planning – from early graph search methods like Dijkstra and A* to more adaptive techniques such as D*, RRT, and learning-based models – illustrates a steady push toward efficiency and robustness. Approaches like APFs, GAs, and FL add further flexibility, each with its own trade-offs. Together, these methods highlight the ongoing effort to balance computational efficiency with real-world challenges.

3.2 Intersection Management

4

Methodology

This section covers the methodology and work produced as part of this thesis. The first part to be detailed is the retrieval of satellite images. This includes the Application Programming Interface (API) usage and method used for signing URLs.

4.1 Satellite Imagery

Satellite imagery is a key component of this thesis project. The imagery will be used for both training and testing the DL models, by creating a dataset detailed in [Section 4.3](#), and as input to said model during inference. This section covers the acquisition of satellite imagery, the process of signing URLs as required by the API, and the code created for these purposes.

This project utilizes Google Maps Static API as provided by Google Cloud Platform. The API allows for the retrieval of static map images at a given resolution and zoom level. This API was chosen due to its ease of use, the quality of the retrieved images, and the fact that it is free to use for a limited number of requests. The API is used to retrieve satellite imagery of a given location.

4.1.1 Image Acquisition

Google Maps Static API can retrieve images by forming requests with specific parameters that define the center, zoom level, size, and additional options for the map. For this project, images of type `satellite` are used, as they provide the highest level of detail for each retrieved image. Other types like `roadmap` or `terrain` do not provide enough detail to create a path that would realistically help navigate any kind of intersection as things like line markings are abstracted away.

To request an image, a URL is generated dynamically for the API, incorporating the required parameters. The parameters of the API request are as follows:

- `center`: The latitude and longitude of the center of the map (e.g. `41.30392, -81.90169`).
- `zoom`: The zoom level of the map. 1 is the lowest zoom level, showing the entire Earth, and 21 is the highest zoom level, showing individual buildings.
- `size`: The dimensions of the image to be retrieved, specified in pixels (e.g., `400x400`).
- `maptype`: Specifies the type of map to be retrieved. Options include `roadmap`, `satellite`, `terrain`, and `hybrid`.
- `key`: The API key used to authenticate the request.

- `[signature]`: Secret signing signature given by Google Cloud Platform through their interface.

Furthermore, the API allows for markers to be placed on the map, which can be used to highlight specific points of interest. This is, however, not relevant to this project.

4.1.1.1 URL Signing

While requests to the API can be made using only the API key, the usage is severely limited without URL signing. URL signing is a security measure that ensures that requests to the API are made by the intended user. The signature is generated using the API key and a secret key provided by Google Cloud Platform. The URL signing algorithm is shown in [Algorithm 1](#) and is provided by Google [27].

Algorithm 1: URL Signing Algorithm (`sign_url`)

Input: URL, secret

```
url ← urlparse(URL)
secret_decoded ← base64_decode(secret)
signature ← HMAC_SHA1(secret_decoded, url.path + "?" + url.query)
signature ← base64_encode(signature)
URL_signed ← URL + '&signature=' + signature
```

Output: URL_signed

As input is the URL with filled parameters and the secret key. The algorithm generates a signature using the HMAC-SHA1 algorithm with the secret key and the URL to be signed. The signature is then base64 encoded and appended to the URL as a query parameter. The signed URL can then be used to make requests to the API.

4.1.2 Implementation

The main functionality of satellite imagery retrieval can be seen in [Listing 1](#). An example of the output of the functionality can be seen in [Figure 1](#).

```

1  def get_sat_image(lat: float, lon: float, zoom: int = 15, secret: str = None, print_url: bool
2      = False) -> requests.Response:
3      if not secret:
4          raise Exception("Secret is required")
5      if not lat or not lon:
6          raise Exception("Both lat and lon are required")
7
8      req_url = f"https://maps.googleapis.com/maps/api/staticmap?center={lat},{lon}&zoom={zoom}"
9      &size=400x400&maptype=satellite&key={API_key}"
10     signed_url = sign_url(req_url, secret)
11     if print_url:
12         print(signed_url)
13
14     response = requests.get(signed_url)
15     return response
16
17 def save_sat_image(response: requests.Response, filename: str = "map.png") -> None:
18     if response.status_code != 200:
19         raise Exception(f"Failed to get image, got status code {response.status_code}")
20
21     with open(filename, "wb") as f:
22         f.write(response.content)

```

Listing 1. Python functions used to retrieve and save satellite imagery (`get_sat_image` and `save_sat_image`)

Listing 1 shows two functions, `get_sat_image` and `save_sat_image`, that are used to retrieve and save satellite imagery, respectively. The `get_sat_image` function first checks if a `secret` and coordinates are provided as they are required. It then constructs a URL for the Google Maps Static API request and signs it using the `sign_url` function detailed in [Algorithm 1](#). The signed URL is then used to make a request to the API, and the response containing the image is returned. This response can then be passed to the `save_sat_image` function, which saves the image to a file with the specified filename.

Listing 2 shows the Python implementation of [Algorithm 1](#) with details following beneath.

```

1  def sign_url(input_url: str = None, secret: str = None) -> str:
2      if not input_url or not secret:
3          raise Exception("Both input_url and secret are required")
4
5      url = urlparse.urlparse(input_url)
6      url_to_sign = url.path + "?" + url.query
7      decoded_key = base64.urlsafe_b64decode(secret)
8      signature = hmac.new(decoded_key, str.encode(url_to_sign), hashlib.sha1)
9      encoded_signature = base64.urlsafe_b64encode(signature.digest())
10     original_url = url.scheme + "://" + url.netloc + url.path + "?" + url.query
11
12     return original_url + "&signature=" + encoded_signature.decode()

```

Listing 2. Python implementation of the URL signing algorithm (`sign_url`)

- `sign_url:1`: Function declaration for the URL signing algorithm. It takes in two parameters, `input_url` and `secret`.
- `sign_url:2-3`: A check is performed to ensure that both `input_url` and `secret` are provided.
- `sign_url:5`: The provided URL is parsed using the `urlparse` function from the `urlparse` library.
- `sign_url:6`: The URL to be signed is extracted from the parsed URL object.
- `sign_url:7`: The secret key is decoded from base64 encoding.

- `sign_url:8`: The signature is generated using the HMAC-SHA1 algorithm with the decoded secret key and the URL to be signed.
- `sign_url:9-10`: The resulting signature is then base64 encoded and the URL is reconstructed.
- `sign_url:12`: The signed URL is returned with the signature appended as a query parameter. An example output could be

```
https://maps.googleapis.com/maps/api/staticmap?center=41.30392,-81.90169&zoo
m=18&size=400x400&maptype=satellite&key=<api_key>&signature=<signature>
```

with filled `<api_key>` and `<signature>`.

A small `rotate_image` function was also created to rotate the retrieved image by some degrees, as the orientation of the satellite images can vary. The code can be seen in [Listing 3](#). This is meant to help simplify the task performed by the model, as it alleviates the need to handle poorly angled images.

```
1 def rotate_image(image_path, angle) -> None:
2     image_obj = Image.open(image_path)
3     rotated_image = image_obj.rotate(angle)
4     rotated_image.save(image_path)
```

[Listing 3](#). Python function to rotate an image by a specified angle (`rotate_image`)

<https://maps.googleapis.com/maps/api/staticmap?center=55.780001,9.717275&zoom=18&size=400x400&maptype=satellite&key=wefhuwvjwekrlbvowilerbvkebvlearufhbew&signature=aqwhfunojlksdcnipwebfpwebfu=>



[Figure 1](#). Example of a signed URL and satellite image retrieved using the Google Maps Static API.

4.2 Loss Function Design

4.3 Dataset Creation

The creation of a proper dataset is crucial for making sure the model learns the task desired for it to perform. The dataset will have to work hand-in-hand with the model architecture and the loss function to ensure that the model learns the task effectively. Many aspects are to be considered when creating a dataset for a task as specific as this project sets out to create:

- › It should be large enough to capture the complexity of the task. Size can be artificially increased through data augmentation.
- › It should be diverse enough to capture the variety of scenarios that can occur at an intersection.
- › It should allow for some leniency when it comes to generating paths, as the model should not be too stringent to a singular path.
- › For the purposes of this project, its creation should seek to answer Research Question **RQ-4** by providing a dataset that allows for the training of a model that can generate paths that are not too stringent to a singular path.

4.3.1 Cold maps

The deduced method for training the model, as detailed in [Section 4.2](#), revolves around the use of a cold map. A cold map representation of the desired path was chosen for a small simplification in the loss function. It penalizes points that are further from the desired path, and does not do this for points that are on the path. Creating this cold map was done in several steps. First, a grid of the same size as the input image is created. The input image is the path drawn in white on a black background, as shown in centre [Figure 3](#). This means that the only occupied pixels are those taken up by the path. In this grid, the coordinates of the closest non-zero pixel is found by iterating over the entire input image containing the path. The complexity of this operation will be covered in the following sections. Next, the distance between the current pixel and the closest non-zero pixel is calculated. This distance is then compared to a threshold value to determine its value. If it is further away, the resulting penalty from the loss function should be higher. Different values for the threshold and the exponent of the distance calculation were tested to find the best combination. Lastly, the cold map is saved in a structured folder format for later use in training. Later, the created data is put through augmentation to inflate the size of the dataset and increase its diversity.

4.3.1.1 Finding the distance to the desired path

The algorithm for finding the distance to the closest point on the desire path is shown in [Listing 4](#).

```

1  occupied = []
2  for i in range(binary.shape[0]):
3      for j in range(binary.shape[1]):
4          if binary[i, j] != 0:
5              occupied.append((i, j))
6
7  h, w = binary.shape
8  nearest_coords = np.zeros((h, w, 2), dtype=int)
9
10 for i in range(binary.shape[0]):
11     for j in range(binary.shape[1]):
12         if binary[i, j] == 0:
13             min_dist = float('inf')
14             nearest_coord = (i, j)
15             for x, y in occupied:
16                 d = hypot(i - x, j - y)
17                 if d < min_dist:
18                     min_dist = d
19                     nearest_coord = (x, y)
20             nearest_coords[i, j] = nearest_coord
21         else:
22             nearest_coords[i, j] = (i, j)

```

[Listing 4](#). Non-parallelized code for finding the nearest point on the path.

The algorithm in [Listing 4](#) starts by creating an array of coordinates based on the `binary` map created with the `threshold` function from the OpenCV library. This `binary` map contains every non-black pixel in the input image, which in this case is the path drawn on a black background. With these occupied pixels stored in an array, the algorithm then iterates over every grid point of the `nearest_coords` grid, created to be the same size as the input image. For every point in the grid, the algorithm checks if the point is on the path. If it is, the algorithm assigns the current point's coordinates to the `nearest_coords` grid. If the point is not on the path, the algorithm iterates over every occupied pixel and calculates the distance between the current point and the occupied pixel. If the distance is less than the current minimum distance, the minimum distance is updated and the coordinates of the closest point are saved. This is repeated for every occupied pixel, and the coordinates of the closest point are saved in the `nearest_coords` grid. This process is repeated for every point in the grid until every point has been assigned the coordinates of the closest point on the path. This grid will later be used under the name `coords`.

The shown algorithm is not parallelized and has a complexity of $\mathcal{O}(n^2)$, where n is the size of the input image. This is due to the nested `for`-loops used in the algorithm. While not a great complexity, it is a vast improvement over its earlier iteration which was $\mathcal{O}(n^4)$ ¹. The actual implementation of this algorithm is parallelized, but the non-parallelized form is shown here. The first iteration of the algorithm took 73 minutes to complete on a 400×400 image, while the parallelized version took 8 minutes on an 8-core CPU. This non-parallelized version takes roughly 30 seconds to complete on the same image, with the parallelized version taking just a few seconds on a full 400×400 image. Further improvements are likely possible to be made both to the complexity of the implementation and

¹The original implementation can be seen in `dataset/lib.py:process_rows` in the GitLab repository.

parallelization could be distributed to a GPU or the cloud for even faster computation, but this remains future work.

4.3.1.2 Creating the cold map

To start the creation of the cold map, a distance grid is created using Pythagoras' theorem between the coordinates of the point of the grid and the coordinates saved within, retrieved from the aforementioned `coords` variable. A masking grid is then created by comparing the distance grid to a threshold value. This results in each grid point being calculated using:

$$d_{ij} = \sqrt{(i - c_{ij0})^2 + (j - c_{ij1})^2} \quad (1)$$

$$dt_{ij} = \begin{cases} d_{ij} & \text{if } d_{ij} < t \\ t + (d_{ij} - t)^e & \text{otherwise} \end{cases} \quad (2)$$

where $c = \text{coords}$, $c_{ij0} = \text{coords}[i, j][0]$, t is the threshold value, and e is the exponent value. All three of these can be seen as function parameters in the function declaration in [Listing 5](#). The distance grid is then normalized to a range of 0 to 255 to minimize space usage such that it fits within a byte, i.e. an unsigned 8-bit integer. This is done by subtracting the minimum value and dividing by the range of the values. Alternatively, the `normalize` parameter can be set to another value, as usage within a loss function would prefer a value between 0 and 1 (as detailed in [Section 4.2](#)). The resulting grid is then saved as a cold map. The resulting cold map can be seen in the rightmost image in [Figure 3](#).

```

1 def coords_to_coldmap(coords, threshold: float, exponent: float, normalize: int = 255):
2     rows, cols = coords.shape[0], coords.shape[1]
3
4     distances = np.zeros((rows, cols), dtype=np.float32)
5     for i in range(rows):
6         for j in range(cols):
7             distances[i, j] = hypot(i - coords[i, j][0], j - coords[i, j][1])
8
9     distances_c = distances.copy()
10    mask = distances > threshold
11    distances_c[mask] = threshold + (distances[mask] - threshold) ** exponent
12
13    distances_c_normalized = normalize * (distances_c - distances_c.min()) / (distances_c.max()
14 - distances_c.min())
15
16    return_type = np.uint8 if normalize == 255 else np.float32
17
18    return distances_c_normalized.astype(return_type)

```

[Listing 5](#). Non-parallelized code for finding the nearest point on the path.

To figure out the optimal values for the threshold and exponent, a grid search was performed. The grid search was done by iterating over a range of values for both the threshold and the exponent. The resulting cold maps were then evaluated by a human to determine which combination of values resulted in the most visually appealing cold map. For a 400×400 image, the optimal values were found to be $t = 20$ and $e = 0.5$. The grid of results can be seen in figure [Figure 2](#). In testing, the value of $e = 1$ was excluded as it had no effect on the gradient produced in the cold map, meaning all values of t produced the same map.

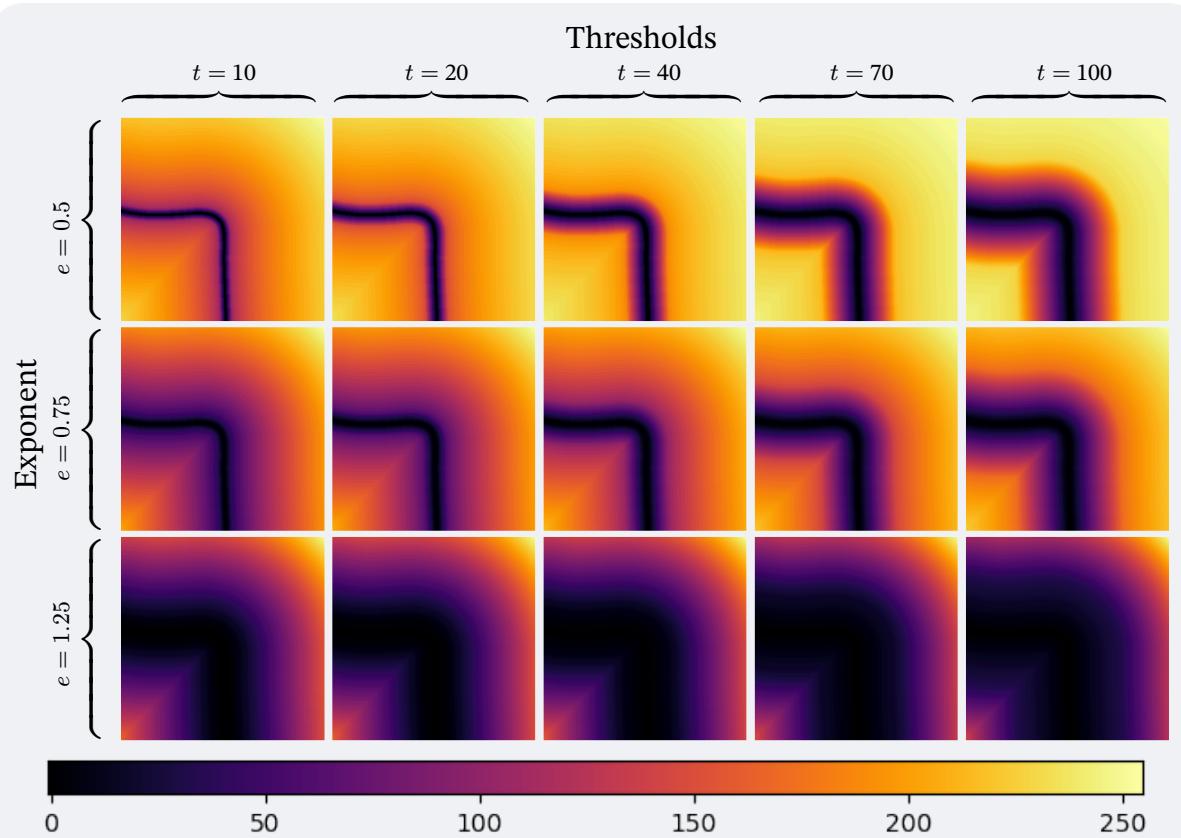


Figure 2. Results of testing threshold values $t \in \{10, 20, 40, 70, 100\}$ and exponent values $e \in \{0.5, 0.75, 1.25\}$. The colour map is shown beneath the results².

Finally, a comparison between the retrieved satellite image of an intersection, the optimal path through it, and the cold map generated by the process described above are shown in [Figure 3](#). This highlights the importance of the cold map in the training process as opposed to the single line path. The cold map allows for a more lenient path to be generated, as the model is not penalized for deviating slightly from the path.

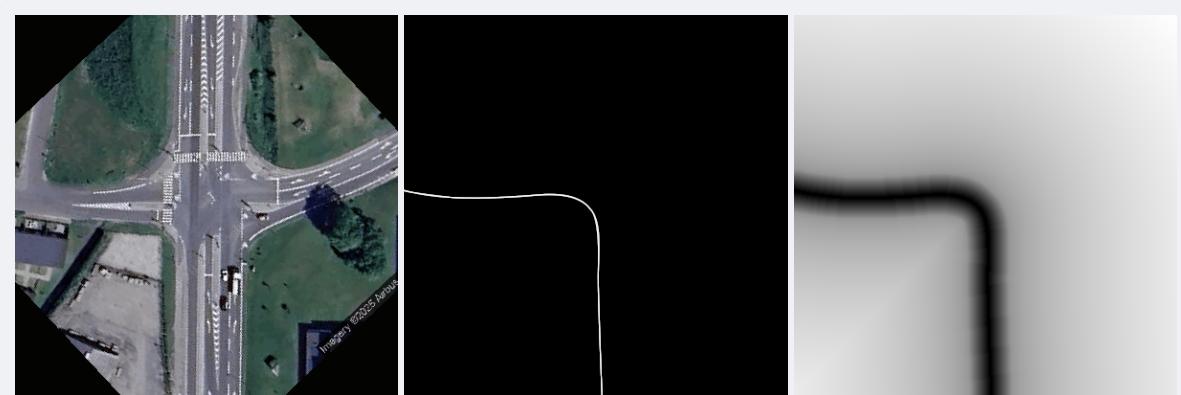


Figure 3. Example of satellite image, next to the desired path through with. To the far right, the generated cold map is shown with threshold $t = 20$ and exponent $e = 0.5$. Notice how it is only the points very close to the path that are very cold, while the rest of the map is warmer the further away it is.

²The colour map is retrieved from the matplotlib docs: <https://matplotlib.org/stable/users/explain/colorbars/colormaps.html>

4.3.2 Data Augmentation

4.3.3 Dataset Structure

4.4 Model Design

5

Results

This section details the experiments conducted

6

Discussion

In this section...

6.1 Integration with existing systems

6.2 Shortcomings

6.3 Future Work

7

Conclusion

References

- [1] C. M. University, “The Carnegie Mellon University Autonomous Land Vehicle Project.” [Online]. Available: <https://www.cs.cmu.edu/afs/cs/project/alv/www/index.html>
- [2] C. Thorpe, M. Hebert, T. Kanade, and S. Shafer, “Vision and navigation for the Carnegie-Mellon Navlab,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 3, pp. 362–373, 1988, doi: [10.1109/34.3900](https://doi.org/10.1109/34.3900).
- [3] J. Billington, “The Prometheus project: The story behind one of AV's greatest developments.” [Online]. Available: <https://www.autonomousvehicleinternational.com/features/the-prometheus-project.html>
- [4] A. S. Francisco, [Online]. Available: <https://abc7news.com/post/wrong-waymo-driverless-car-goes-oncoming-traffic-tempo-arizona/15238556/>
- [5] “Technical milestone in road safety: experts praise Volkswagen's Car2X technology.” [Online]. Available: <https://www.volkswagen-newsroom.com/en/press-releases/technical-milestone-in-road-safety-experts-praise-volkswagens-car2x-technology-5914>
- [6] K. Dresner and P. Stone, “A multiagent approach to autonomous intersection management,” *Journal of artificial intelligence research*, vol. 31, pp. 591–656, 2008.
- [7] A. P. Chouhan and G. Banda, “Autonomous Intersection Management: A Heuristic Approach,” *IEEE Access*, vol. 6, no. , pp. 53287–53295, 2018, doi: [10.1109/ACCESS.2018.2871337](https://doi.org/10.1109/ACCESS.2018.2871337).
- [8] Z. Zhong, M. Nejad, and E. E. Lee, “Autonomous and Semiautonomous Intersection Management: A Survey,” *IEEE Intelligent Transportation Systems Magazine*, vol. 13, no. 2, pp. 53–70, 2021, doi: [10.1109/MITTS.2020.3014074](https://doi.org/10.1109/MITTS.2020.3014074).
- [9] M. Cederle, M. Fabris, and G. A. Susto, “A Distributed Approach to Autonomous Intersection Management via Multi-Agent Reinforcement Learning.” [Online]. Available: <https://arxiv.org/abs/2405.08655>
- [10] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numer. Math. (Heidelb.)*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
- [11] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968, doi: [10.1109/TSSC.1968.300136](https://doi.org/10.1109/TSSC.1968.300136).
- [12] A. Stentz, “Optimal and efficient path planning for partially-known environments,” in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1994, pp. 3310–3317. doi: [10.1109/ROBOT.1994.351061](https://doi.org/10.1109/ROBOT.1994.351061).
- [13] A. (Tony) Stentz, “The Focussed D* Algorithm for Real-Time Replanning,” in *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI '95)*, Aug. 1995, pp. 1652–1659.

- [14] S. Koenig and M. Likhachev, “Fast replanning for navigation in unknown terrain,” *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005, doi: [10.1109/TRO.2004.838026](https://doi.org/10.1109/TRO.2004.838026).
- [15] O. Khatib, “Real-Time Obstacle Avoidance for Manipulators and Mobile Robots,” in *Autonomous Robot Vehicles*, I. J. Cox and G. T. Wilfong, Eds., New York, NY: Springer New York, 1990, pp. 396–404. doi: [10.1007/978-1-4613-8997-2_29](https://doi.org/10.1007/978-1-4613-8997-2_29).
- [16] Y. Koren and J. Borenstein, “Potential field methods and their inherent limitations for mobile robot navigation,” in *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, 1991, pp. 1398–1404. doi: [10.1109/ROBOT.1991.131810](https://doi.org/10.1109/ROBOT.1991.131810).
- [17] V. Gazi, “Swarm aggregations using artificial potentials and sliding-mode control,” *IEEE Transactions on Robotics*, vol. 21, no. 6, pp. 1208–1214, 2005, doi: [10.1109/TRO.2005.853487](https://doi.org/10.1109/TRO.2005.853487).
- [18] N. Leonard and E. Fiorelli, “Virtual leaders, artificial potentials and coordinated control of groups,” in *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, 2001, pp. 2968–2973. doi: [10.1109/CDC.2001.980728](https://doi.org/10.1109/CDC.2001.980728).
- [19] S. M. LaValle, “Rapidly-exploring random trees : a new tool for path planning,” *The annual research report*, 1998, [Online]. Available: <https://api.semanticscholar.org/CorpusID:14744621>
- [20] S. Karaman and E. Frazzoli, “Sampling-based Algorithms for Optimal Motion Planning.” [Online]. Available: <https://arxiv.org/abs/1105.1186>
- [21] R. Cui, Y. Li, and W. Yan, “Mutual Information-Based Multi-AUV Path Planning for Scalar Field Sampling Using Multidimensional RRT*,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 7, pp. 993–1004, 2016, doi: [10.1109/TSMC.2015.2500027](https://doi.org/10.1109/TSMC.2015.2500027).
- [22] M. Xanthidis *et al.*, “Navigation in the Presence of Obstacles for an Agile Autonomous Underwater Vehicle,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 892–899. doi: [10.1109/ICRA40945.2020.9197558](https://doi.org/10.1109/ICRA40945.2020.9197558).
- [23] C. S. TAN, “A Collision Avoidance System for Autonomous Underwater Vehicles.” [Online]. Available: <https://pearl.plymouth.ac.uk/secam-theses/302/>
- [24] C. Lamini, S. Benhlima, and A. Elbekri, “Genetic Algorithm Based Approach for Autonomous Mobile Robot Path Planning,” *Procedia Computer Science*, vol. 127, pp. 180–189, 2018, doi: <https://doi.org/10.1016/j.procs.2018.01.113>.
- [25] L. Zadeh, “Fuzzy sets,” *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965, doi: [https://doi.org/10.1016/S0019-9958\(65\)90241-X](https://doi.org/10.1016/S0019-9958(65)90241-X).
- [26] L. Zadeh, “Fuzzy algorithms,” *Information and Control*, vol. 12, no. 2, pp. 94–102, 1968, doi: [https://doi.org/10.1016/S0019-9958\(68\)90211-8](https://doi.org/10.1016/S0019-9958(68)90211-8).
- [27] “Use a Digital Signature.” [Online]. Available: <https://developers.google.com/maps/documentation/maps-static/digital-signature>

Appendix

A: Appendix 1	vii
----------------------------	------------

A: Appendix 1