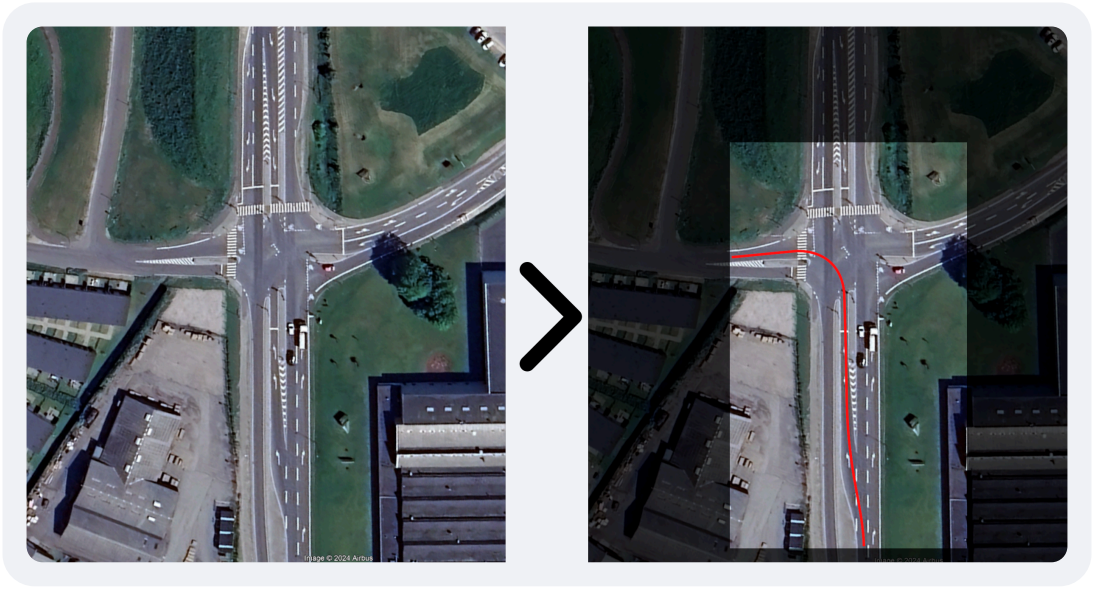
Pre-planning Intersection Traversal for Autonomous Vehicles

Master's Thesis in Computer Engineering

Ian Dahl Oliver

Department of Electrical and Computer Engineering
Aarhus University
Aarhus, Denmark
ian.oliver@post.au.dk

11-02-2025



Supervisor: Lukas Esterle
lukas.esterle@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 05-06-2024, 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...

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.

Acronyms Index

Acronym	Definition
AIM	Autonomous Intersection Management
API	Application Programming Interface
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
GNN	Graph Neural Network
GUI	Graphical User Interface
ID	Identifier
JSON	JavaScript Object Notation
MLP	Multilayer Perceptron
PNG	Portable Network Graphics
PRNG	Pseudo-Random Number Generator
RMSE	Root Mean Squared Error
RNG	Random Number Generator
ROS2	Robot Operating System 2
TOML	Tom's Obvious Minimal Language
UI	User Interface
UX	User Experience
YAML	YAML Ain't Markup Language

Contents

1 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. 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 [4]. Autonomous Intersection Management (AIM) also relies on infrastructure to provide vehicles with information regarding intersections, with an orchestrator monitoring and managing individual intersections [5], [6], [7], with active development moving towards a more decentralized and distributed approach [8]. 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 projects 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

RQ-1 How does a waypoint-based approach compare to a pixel-subset-based approach in terms of accuracy and efficiency in path planning for autonomous vehicles at intersections?

RQ-2 Which type of deep learning model produces the most optimal results for pre-planned intersection traversal?

RQ-3 Is it possible to design a loss function that effectively captures the similarity between generated and optimal paths for autonomous vehicles without it only allowing for exact paths?

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?

Research Question **RQ-4**

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

3.2 Intersection Management

3.3 On-board vs Cloud Computing

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 [9].

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.

```python
def get_sat_image(lat: float, lon: float, zoom: int = 15, secret: str = None, print_url: bool = False) -> requests.Response:
  if not secret:
    raise Exception("Secret is required")
  if not lat or not lon:
    raise Exception("Both lat and lon are required")

  req_url = f"https://maps.googleapis.com/maps/api/staticmap?center={lat},{lon}&zoom={zoom}&size=400x400&maptype=satellite&key={API_key}"
  signed_url = sign_url(req_url, secret)
  if print_url:
    print(signed_url)

  response = requests.get(signed_url)
  return response


def save_sat_image(response: requests.Response, filename: str = "map.png") -> None:
  if response.status_code != 200:
    raise Exception(f"Failed to get image, got status code {response.status_code}")

  with open(filename, "wb") as f:
    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.

```python
def sign_url(input_url: str = None, secret: str = None) -> str:
  if not input_url or not secret:
      raise Exception("Both input_url and secret are required")

  url = urlparse.urlparse(input_url)
  url_to_sign = url.path + "?" + url.query
  decoded_key = base64.urlsafe_b64decode(secret)
  signature = hmac.new(decoded_key, str.encode(url_to_sign), hashlib.sha1)
  encoded_signature = base64.urlsafe_b64encode(signature.digest())
  original_url = url.scheme + "://" + url.netloc + url.path + "?" + url.query

  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&zoom=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.

```python
def rotate_image(image_path, angle) -> None:
  image_obj = Image.open(image_path)
  rotated_image = image_obj.rotate(angle)
  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=wefhuwvjwekrlbvowilerbvkebvlearufhbebwe&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.

```
nearest_coords = np.zeros((path.shape[0], path.shape[1], 2), dtype=int)
for i in range(path.shape[0]):
  for j in range(path.shape[1]):
    if binary[i, j] == 0:
      min_dist = float('inf')
      nearest_coord = (i, j)
      for x in range(path.shape[0]):
        for y in range(path.shape[1]):
          if binary[x, y] != 0:
            dist = hypot(i - x, j - y)
            if dist < min_dist:
              min_dist = dist
              nearest_coord = (x, y)
      nearest_coords[i, j] = nearest_coord
    else:
      nearest_coords[i, j] = (i, j)
```

Listing 4. Non-parallelized code for finding the nearest point on the path.

The algorithm in Listing 4 iterates over every pixel of the aforementioned grid. For each point in the grid, the first thing checked is the value of a binary map `binary`, which is also a grid of the same size as the input image, if there is something there. This is done to avoid calculating the distance for points that are already on the path and are simply assigned their current coordinates. If the point is not on the path, i.e. it as a value of 0, the algorithm iterates through the entire path image, and if it encounters a non-zero value, it calculates the distance between the current point and the point on the path. There is no guarantee that this is the closest point however, so the algorithm saves the coordinates of the found point in a variable if it currently is the closest point. Finally, the closest point at the end is saved to that grid entry's coordinates. This is then repeated for every single 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 $\mathcal{O}(n^4)$, where n is the size of the input image. This is due to the nested loops that iterate over the entire image to find the closest point. This operation is done for every pixel in the image, resulting in a high complexity. The actual implementation of this algorithm is parallelized, but the non-parallelized form is shown here. While the parallelized version is significantly faster by assigning each core its own chunk of the data, going from 73 minutes on a 400×400 down to 8 minutes on an 8-core CPU, the complexity remains the same. Further improvements could be made both to the complexity of the implementation and 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:

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

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

where $c = \texttt{coords}$, $c_{ij0} = \texttt{coords[i, j][0]}$, 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. The resulting grid is then saved as a cold map. The resulting cold map can be seen in the rightmost image in Figure 3.

```python
def coords_to_coldmap(coords, threshold: float, exponent: float):
  rows, cols = coords.shape[0], coords.shape[1]

  distances = np.zeros((rows, cols), dtype=np.float32)
  for i in range(rows):
    for j in range(cols):
      distances[i, j] = hypot(i - coords[i, j][0], j - coords[i, j][1])

  distances_c = distances.copy()
  mask = distances > threshold
  distances_c[mask] = threshold + (distances[mask] - threshold) ** exponent

   distances_c_normalized = 255 * (distances_c - distances_c.min()) / (distances_c.max() - distances_c.min())

  return distances_c_normalized.astype(np.uint8)
```

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.

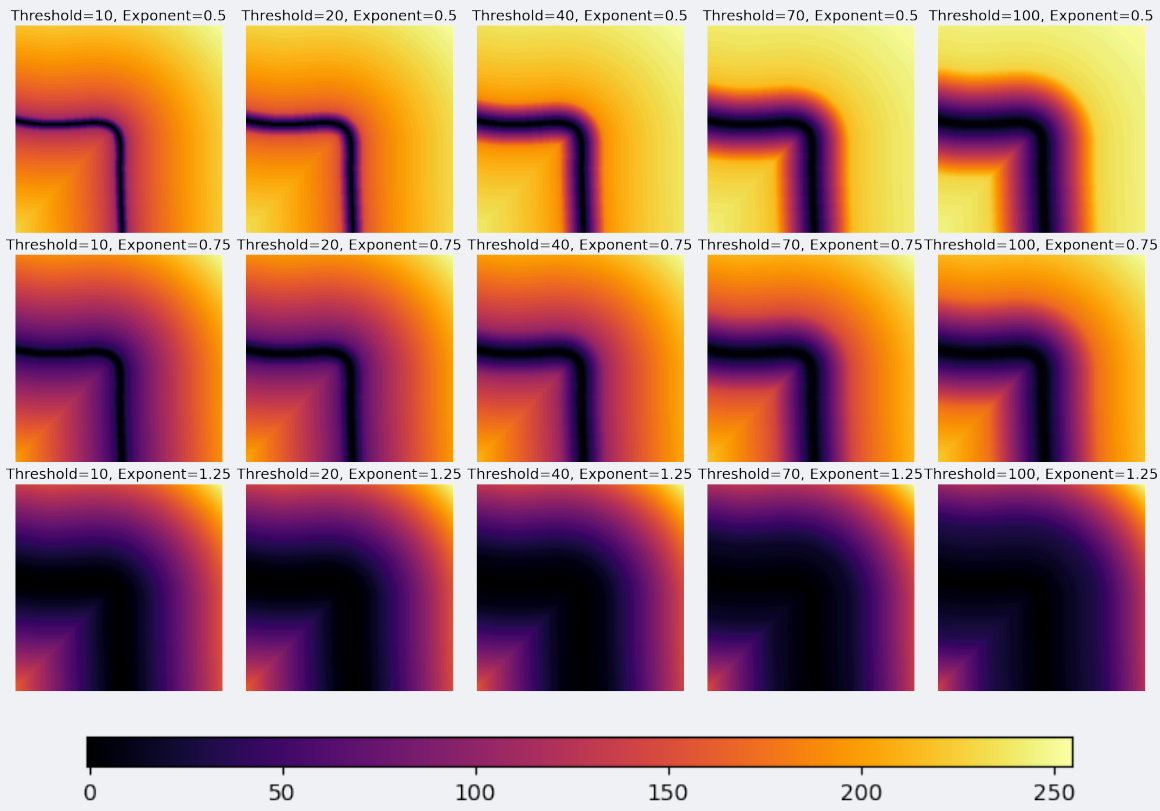


Figure 2. Results of testing threshold values $t \in \{10, 20, 40, 70, 100\}$ and exponent values $e \in \{0.5, 0.75, 1.25\}$. The colour map is shown beneath the results[1].

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.

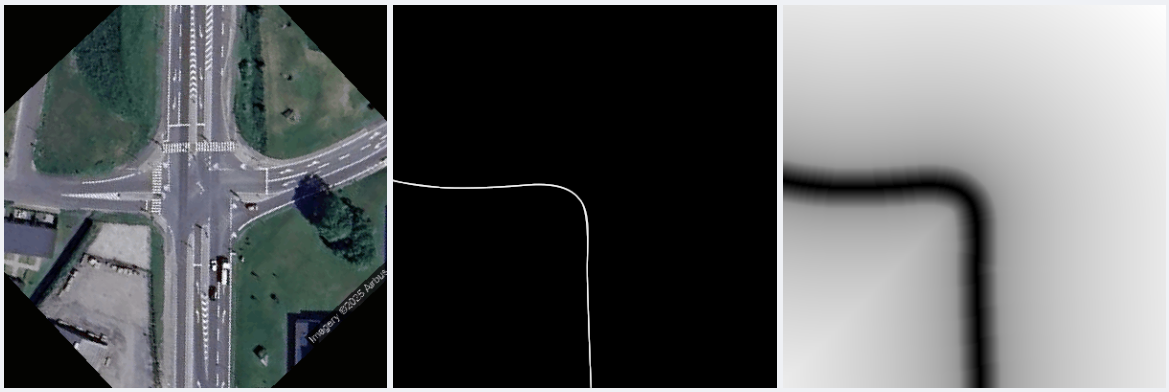


Figure 3. Example of satellite image, next to the desired path through with. To the far right, the generated cold map is shown with threshold $t = 20$ and exponent $e = 0.5$.

[1]The colour map is retrieved from the matplotlib docs: https://matplotlib.org/stable/users/explain/colors/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.

[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] “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

[5] K. Dresner and P. Stone, “A multiagent approach to autonomous intersection management,” *Journal of artificial intelligence research*, vol. 31, pp. 591–656, 2008.

[6] 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.

[7] 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/MITS.2020.3014074.

[8] 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

[9] “Use a Digital Signature.” [Online]. Available: https://developers.google.com/maps/documentation/maps-static/digital-signature

Appendix

A: Appendix 1