Toby William Towler

Registration number 100395626

2025

Robot Mower Mapping and Pathing

Supervised by Edwin Ren

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The abstract of your report summarises your entire work () in no more than half a page. It should include the context of your work including its main objective, what methods you employed, how you implemented these, what the outcomes were and a final statement as a conclusion. It should not contain acronyms, abbreviations, elements of a literature review (though a statement of related work is permissible if it is crucial to your work) or future work. The abstract should be written when everything else has been written up and the project is finished! is this workng

Acknowledgements

Contents

List of Figures

1. Introduction

The robot mower is an already existing project developed by previous masters students from the University of East Anglia. Physically, the mower has 2 tracks for movement on the sides of a metal frame, it is controlled by raspberry pi 4 running the Robot Operating System (Macenski et al., 2022). Sensor wise, the robot is equipped with a 4G dongle, lidar and a GPS chip that was upgraded to an RTK chip in this iteration of the project. The existing code base was mostly written in python with very small amounts of C++. Because of this, all of my code will be written in python to slot into existing code without issue.

My contribution to this project this year will be, regarding the overall movement and guidance of the robot. For ease of planning, I have broken this down into 3 sections:

1. Basic map generation

2. complete coverage path planning

3. map generation from an aerial image

These sections are all modular meaning they can be developed, tested and function independently but still easily be integrated together for the final product. The specified use case of the robot will now be to cut golf courses, this is particularly relevant to the aerial map generation section of my work which will likely use a machine learning model and require relevant training data, while the other sections are not concerned with a real world use case as their algorithms will work be able to tweaked for any applicable use case of this robot.

2. Background and Related Work

Another section that is essential and should keep its title as is! Although you could perhaps call it "Literature Review" instead, this is not advisable as at this stage of your project we do not expect an extensive literature review since this was already done in the second formative assignment. The rationale is simply because you will lose valuable pages that could be used better in the next two sections that will cover the preparation and implementation of actual work done. So just provide the context in which your project operates here, and then provide a brief overview of similar work that is directly

relevant to yours. Try to avoid blatant copying and pasting from the formative literature review as it is bound to read awkwardly.

3. System Design

3.1. Map Generation

Map generation is an important part of testing this system, it is important to test on all scenarios that may occur in the real world. For this reason, random or parameter based map generation is very necessary to guarantee success in every environment. As the outputs of this section will mostly be used for testing the path planning algorithm on regions with differing area, number of corners and complexity, no excessive algorithmic complexity is needed. This program should also be able to create n obstacles within the main field, such areas would represent obstructions in the mowers desired path, for example trees or telephone poles in the real world. This means we need a function with 2 parameters:

- **K**, number of angles in the outer field

- **N**, number of obstacles within the field, since it would not be sensible to take a parameter for the number of corners for every hole, we can generate them randomly assuming 3–8 corners staying inline with the complexity of the rest of the field without being unreasonably overengineered.

3.1.1. Corners

The number and distance between corners could be thought to represent complexity of a shape. The number of corners in a shape
   KEEPGOING

3.1.2. Obstacles

Obstacles or holes, can be thought to represent real world obstructions for example trees or telephone poles in a real field. Generation of obstacles can be completed using the same function as the outer field generation, simply with different parameters. The

algorithm 2 takes 3 parameters, number of points in the shape, an origin point and the range new points. This allows for variable size, positioning and complexity.

3.1.3. Graham Scan

The Graham Scan (Graham, 1972) is an algorithm to find convex hulls, that is from a set of points the outline which contains all inner points. This algorithm does this by sorting the points by their polar angle to the lowest point, since this is always in the hull. There is then further calculations based on the angle between adjacent points to omit inner points from the outline, however for this use case that is not necessary since all points will be vertexes in a field. For this reason, the algorithm used is not strictly a Graham Scan but rather heavily based on the first stage, as shown in 3 Using this algorithm allows for consistent outlining of any set of points with no crossovers or intersections

3.2. Complete Coverage Path Planning

Complete coverage path planning(CCPP) is "the task of determining a path that passes over all points of an area or volume of interest while avoiding obstacles" (Zhao and Hwang, 2023) For this module, I have used the Fields2Cover library (Team, 2023), this library is open source. During the course of this project, I contributed to his library, fixing a bug during the build process. The library splits the task up into several sub-tasks.

3.2.1. Robot Configuration

Robot sizing has 2 important factors, track width, how wide the machine itself is, and blade width, how wide the utility object is. For farming equipment the utility object is usually larger than the vehicle, for example a combine harvester's wheels being narrower than blade, however for this project the blade is within the tracks. For this reason the functions will compute slightly differently to its probable intended use case as the tracks are likely to overlap however this should not cause an issue and all outputs should function as needed.

The robot has 2 more parameters regarding its turning circle, these are minimum turning radius and maximum differential curvature. Both of these are important when

calculating a path for the robot to follow as they ensure the robot can easily follow the path.

The robot is 22cm wide from outer edge to edge and the gap between the tracks is 17cm, with a track width of roughly 2.5cm.

3.2.2. Headland Generation

Headlands are the area in which a vehicle turns, think of the rough edges of a crop field, these exist so that agricultural vehicles do not trample their crops when turning. Although for the use case of a golf course this is not strictly needed as there are no "crops" or areas the robot cannot touch Headlands are also generated around obstacles to allow for suitable turning area around the obstacles if needed, this serves as an area to do a "U-turn", effectively the same as reaching the end of the field.

KEEPGOING

3.2.3. Swath Generation

Swaths are the individual straight lines that make up the complete path, they are often parallel to each other, but this can vary depending on how the shape of the field is segmented into smaller shapes however they will always be parallel while in the same sub area. There are 2 ways of determining the optimal angle; the sum of the lengths of all swaths, or the number of swaths, where lower is better for each. Both of these have pros and cons, for example lower total length offers better time efficiency assuming a near constant travel speed while lower swath count gives reduced turning movements which can be better if turning requires a slower speed or other complexities. Similarly, lower sum length of course means less distance travelled and usually less fuel/energy consumed, however the opposite can be true if the acceleration and deceleration causes greater energy consumption compared to constant velocity. Swaths are generated within the bounds of the headlands to allow for independent handling of the turning and connection of swaths regardless of the rest of the generation process. To find the best angle, each angle is tested and compared. This brute force approach could be considered slow, but it is all the library offers and could be taken into consideration in a future release.

3.2.4. Route Planning

A route is the order in which swaths will be covered. These can be sorted in a number of ways:

- **Shortest route** to cover all swaths, order depends on field

- **Boustrophedon order, a zigzag like pattern covering swaths in order (1,2,3...)**

- **Snake order, Similar to Boustrophedon but skipping 1 swath each time (1,3,5...)**

- **Spiral order, a spiral pattern around the field, first swath then last (1, n, 2, n-1...)**

- **Custom order, user defined**

KEEPGOING

3.2.5. Path Planning

Path planning refers to the connection of all swaths in route order. There are a couple of ways the to link neighbouring swaths, these are the Dubins curve and Reeds-Shepp curve.

Both of these compute the shortest path between the 2 points, with the deciding factor between the 2 is the robots ability to move backwards. Reeds-Shepp allows backwards movement

3.2.6. Cell Decomposition

3.3. Aerial Map Generation

Previously, for the program to take in a real world map from an image of the real world, the user would have to upload the picture and manually trace the outline, this could cause some problems. For example, the outline is only as accurate as the user's mouse placement, likely skipping some smaller corners to save time or due to inability to be accurate to the necessary degree. Automating this labourious and time consuming task would positively increase user experience and usability, this can be done using algorithmic and machine learning approaches.

3.3.1. Algorithmic Approaches

There are several algorithmic methods that try to achieve this task from several different librarys. A popular library for image processing is scikit-image.

OpenCV's Canny edge detection implementation (Bradski et al., 2023) follows the classic algorithm developed by Canny (Canny, 1986). The algorithm works through several steps to detect edges in images:

- **Gaussian Blur**: First, the image is smoothed with a Gaussian filter to reduce noise, as edge detection is highly sensitive to noise.

- **Gradient Calculation**: The algorithm calculates the intensity gradients of the image using Sobel filters in both horizontal and vertical directions:

- **Non-maximum Suppression**: This step thins out the edges by keeping only the local maxima. For each pixel, it checks if it's a local maximum in the direction of the gradient. If not, it's suppressed (set to zero).

- **Double Thresholding**: The algorithm applies two thresholds:
    - High threshold: Pixels above this are considered "strong" edges
    - Low threshold: Pixels between low and high are considered "weak" edges

- Edge Tracking by Hysteresis: Finally, weak edges that are connected to strong edges are kept, while isolated weak edges are discarded. This helps ensure that the detected edges are continuous.

This approach works very well for high contrast images, particularly those black and white Figure 1. The clear separation between foreground and background elements allows the algorithm to detect distinct gradient changes and create clean, continuous edge lines.

As for the similarly coloured golf course images, it can often find the outlines but they are made up of many smaller lines Figure 2. This fragmentation occurs because the subtle color transitions between different areas of the golf course (like fairways, roughs, and greens) create weaker gradient signals that may fall inconsistently above or below the detection thresholds.

It may appear there is a simple solution to this problem; join the connected lines into one, however this proves to be rather problematic in itself for a number of reasons:

1. The desired line does not form a complete shape - Gaps in the detected edges mean that even sophisticated line-joining algorithms may fail to connect all segments belonging to a single boundary

2. The line is connected to other lines outside the desired shape - Many detected edges from shadows, texture variations, or other regions of the golf course can intersect with the boundary edges required to be isolated, creating unwanted connection points

3. Once lines are connected, it is impossible to tell which collection of lines is needed - Without prior knowledge of the expected shape or additional contextual information, there's no reliable way to determine which edge segments represent meaningful boundaries, like the edge of a green compared to an outline of trees or a rough area.

All of these means use of canny edge detection can often cause more problems than it solves, while it may have more accurate outlines in places, they may be connected to unwanted areas, Or they may not be complete and still require further user input to chose a selection or finish a shape.

3.3.2. Machine Learning Approach

Another possible solution to this problem is to use a machine learning network to predict the areas of the golf course instead. PyTorch (Paszke et al., 2019) is a powerful open-source machine learning library and has become one of the most popular frameworks for deep learning research and production applications since its initial release in 2016. It was developed to show that usability and speed can both be present in a python implementation of machine learning making it the perfect tool for this task.

3.3.3. Data Set

A models training data needs to be closely related to its real world input data, for the Robot Mower that is satelite images or orthophotos from above the desired mowing area. Training data also needs to be clearly and consistently annotated to producde accurate, reliable results in practice. The only orthophoto data set publicly available is the danish golf course data set, but it lacks quality and consistency; the outlines do not cover the

whole area of single sections and often are missing entirely. Upon testing the data, it produced faulty results including both true negative and false positives. Figures 3 to 4

A custom data set was needed, or at least custom annotations of the plain images from the Danish data set since the raw images from the data set are still good quality, high resolution images and are suitable for training purposes with correct annotations

3.3.4. Training

Pytorch has several training parameters each with their own effect on the outcome. Some foundational hyperparameters are:

- **Batch size**, determines how many samples are at once, this is mainly limited by available memory

- **Image size**, the dimensions of the image, impacting detail perception and processing requirements

- **In channels**, depth of input data(RGB vs grayscale)

- **Learning rate**, how fast weights are updated during training

- **Number of classes**, the amount of classes the model can pick from

There are a few other options needed in the configuration of the model:

Class weights are distinctly assigned to each class, higher weights are given to less frequent classes to help balance the less frequent outputs.

Data augmentation enhances the model's robustness with random horizontal and vertical flips (probability $P = 0.5$ for each), random rotations $\theta \sim \mathscr{U}(-20°, 20°)$ with $P = 0.5$, color jitter transformations for brightness, contrast, and saturation $\sim \mathscr{U}(0.8, 1.2)$ with $P = 0.5$, and random crops to 80% of original size with $P = 0.5$.

The network architecture follows an enhanced UNet design with an encoder channel progression $[3 \rightarrow 24 \rightarrow 48 \rightarrow 96]$, bottleneck channels 192, and symmetric decoder. Memory efficiency is achieved through depthwise separable convolutions (kernel size 3×3, padding 1, groups equal to input channels) and squeeze-excitation blocks with reduction ratio 4.

For optimization, the model employs a combined loss function $\mathscr{L}_{\text{total}} = 0.7 \cdot \mathscr{L}_{\text{CE}} + 0.3 \cdot \mathscr{L}_{\text{Dice}}$, where $\mathscr{L}_{\text{Dice}} = 1 - \frac{2 \cdot \text{intersection} + 1.0}{\text{union} + 1.0}$. Training utilizes the AdamW optimizer

with learning rate $\eta = 1 \times 10^{-3}$ and weight decay $\lambda = 1 \times 10^{-4}$. A OneCycleLR scheduler manages learning rate progression with parameters pct_start $= 0.1$, div_factor $= 10$, final_div_factor $= 100$, and three_phase $=$ True.

The PyTorch Lightning trainer configuration includes max_epochs $= 100$, mixed precision training (precision $=$ "16-mixed"), gradient clipping (gradient_clip_val $= 0.5$), gradient accumulation (accumulate_grad_batches $= 2$), and validation metrics checked every quarter epoch (val_check_interval $= 0.25$) using half the validation data (limit_val_batches $= 0.5$). Early stopping monitors validation IoU with patience of 15 epochs, while the model checkpoint system saves the top 2 models based on validation performance.

Dataset partitioning allocates 70% for training, 20% for validation, and 10% for testing. Full-resolution inference processes images in patches of size 256 with 50% overlap, employing test-time augmentation by averaging predictions from the original image and its horizontal and vertical flips for improved robustness.

Random forest or not?

To ensure consistent training and results, the data set is split up into sections; 70% for training, 20% for validation and 10% for testing. Constant validation helps to make sure that the model is constantly improving, tweaking the hyperparameters and make modelling decisions.

4. Performance Evaluation

Performance is a key part of user experience therefore is crucial to the real world performance of this product. Performance in this case can be broken down into 2 key metrics:

- **Run time**: how long the process takes to complete in real time

- **Memory Usage**: how much system memory the process uses both average and maximum

Both of these can be measured in python consistently and reliably, meaning the code can be directly tested in the same file without needing system usage or other programs that may have their own overheads. Each module will be run with different parameters to see how they impact performance which then will be run 5 times, these 5 results will then be averaged and plotted against the chosen parameter providing clear graphs to visualise any impact varying inputs may have on the products performance. A baseline run will be provided using set parameters and run 10 times to show a good default.

Run time will be measured in milleseconds and track the total execution time from start to finish. The tool used to measure exectuion time is the **time** library and its **perf_counter** function. This is accurate to nanoseconds and causes negligable difference in runtime.

Memory Usage will be measured in MB and track the average and peak memory usage during the runtime of the program. Peak memory usage is tracked using the **tracemalloc** library. This library tracks the peak memory usage of the code exectuion.

For reference, these benchmarks were taken on a **Ryzen5 5600X** with **16GB** of R.A.M. at **3200 MT/s**.

4.1. Map Generation

4.1.1. Run Time

BaseLine

Figure 5 represents the baseline and tested with 10 corner points, 0-400 range and 0 holes. The runtime is fairly consistent having a very small range of 0.2 milleseconds, showing a very strong baseline to run further testing on.

Number of points

Figure 6 shows the impact of varying the number of points on runtime. The tests were run with a range or 0-400 and 0 holes. Clearly points are added in linear respect to execution time but are efficient in doing so, taking around 0.0075 milleseconds per point added. Points are added with O(n) complexity

Range of points

Figure 7 visualises the time taken when varying limits are placed on the range of points, this can be thought to represent the Physical size of the field. These runs were carried out with 10 points and 0 holes. There is no clear correlation between the range of points and the run time of the program, the graph closely resembles the baseline, having the same range and very similar bounds at each side. The quickest runtime from this test and all baseline runs actually occurs with a range 1200, three times the range in the baseline. This is a very good indication on top of everything else that range has no effect on runtime meaing the program should work for any size of field in good time.

Number of holes

Figure 8 demonstrates how the number of holes translate to runtime. The runs were carried out with 10 points in the main field, a range of 400 and 5 points in each hole. Similarly to the number of points, the number of holes has a linear effect on runtime, however runtime increaes in larger steps. This is likely because in this example, each hole has 5 points and an extra append operation, this is likely why the same number of points generated in the outside field, and in total considering holes differ in runtime. For instance, 200 points in a single field takes around 1.5ms but 210 points comprised of 10 in the outer field and 40, 5 point holes just under 2ms. This time increase does not line up and could be for a number of reasons such as the additional appending to the array or the random number generation process of the point generation function, or could be the extra overheads from the sorting of the points in each hole. Whichever of these is the cause, or if it is a combination of all 3, they are all needed operations so nothing can be done to improve this further, although they whole process still completes in an insignificant amount of time, even with 200 holes it only takes around 9 milleseconds. 200 holes is likely far more than any real world appliaction would need but still computes in excellent time.

4.1.2. Memory Usage

Baseline

Figure 9 is a baseline graph to show the memory usage of the map generation process with 10 points, 400 range and 0 holes. There memory usage is quite consisten except for the obvious outlier on run 1. There is no clear reason for this but it occured on every measurement taken, it is possible it is an issue with the measurement script although unlikely.

Number of points

Figure 10 shows the memory usage of the map generation process with variable points, 400 range and 0 holes. Points effect memory linearly which make sense as there is contasnt size need for each point. The amount of memory needed is very small and is almost guaranteed to be suitable for any computer system even a low end portable micro PC such as a raspberry Pi.

Range of points

Figure 11 draws the memory usage of the map generation process with 10 points, variable range and 0 holes. Increasing the range of points does have a positively correlated effect on memory when compared to the baseline. This could be for a number of reasons but essentially comes down to the higher number possible coordinate values. That could be the data structure of the point takes more memory to store the greater value, because the random number generation process requires more memory or becuase the sorting algorithm needs to memory to sort the higher values.

Number of holes

Figure 12 demonstrates the memory usage of the map generation process with 10 points, 400 range and variable holes. As mentioned in the runtime section, holes are closely related to points, they effect memory linearly, but at a higher rate than points as they are a collection of points. Where memory differs to runtime however, is there are no overheads to having points as holes or a single field, since memory allocation is constant. This concludes the number of holes is irrelevant and the number of total points is all that needs to be considered.

4.2. Complete Coverage Path Planning

y axis - memory usage x axis - map size

 y axis - run time x axis - map size

 y axis - memory usage x axis - route planning method

 y axis - run time x axis - route planning method

 y axis - memory usage x axis - map shape (cornes, sqaure vs star shape etc)

 y axis - run time x axis - map shape

4.3. Aerail Map Generation

y axis - accuracy x axis - image size

 y axis - accuracy x axis - image resolution

 y axis - accuracy x axis - training data size

 y axis - accuracy x axis - training epochs

 y axis - precision x axis - recall

5. Conclusion and Future Work

Another essential section that should keep its title as suggested. Briefly discuss your main findings, outcomes, results; what worked and what could have been done differently. Then summarise your work in a concluding statement by comparing your outcomes against the main and sub-objectives and/or MoSCoW requirements (if used) and suggest potential future work that could be done if more time would be available.

References

Bradski, G., Kaehler, A., and Pisarevsky, V. (2023). Opencv: Open source computer vision library. *https://opencv.org/*. Accessed: 2025-04-13.

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, PAMI-8(6):679–698.

Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133.

Macenski, S., Foote, T., Gerkey, B., Lalancette, C., and Woodall, W. (2022). Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Team, F. (2023). Fields2cover: Mission planning for agricultural vehicles. *https://fields2cover.github.io/*.

Zhao, S. and Hwang, S.-H. (2023). Complete coverage path planning scheme for autonomous navigation ros-based robots. *ICT Express*, 9(3):361–366.

A. Map Generation

Algorithm 1 Point Class Definition

1: **procedure** CLASS POINT
2:     $X \leftarrow -1$ ▷ X-coordinate initialized to -1
3:     $Y \leftarrow -1$ ▷ Y-coordinate initialized to -1
4:     $angle \leftarrow -10$ ▷ Angle initialized to -10
5:     **procedure** CONSTRUCTOR(x, y)
6:         $this.X \leftarrow x$
7:         $this.Y \leftarrow y$
8:         $this.angle \leftarrow -10$ ▷ Default angle value
9:     **end procedure**
10: **end procedure**

Algorithm 2 Generate random points

1: **function** GENPOINTS(*num*, *P*, *size*)
2:     $points \leftarrow []$
3:     **for** $i \leftarrow 0$ **to** $num - 1$ **do**
4:         $randX \leftarrow$ random_integer($P.X + 1, P.X + size$)
5:         $randY \leftarrow$ random_integer($P.Y + 1, P.Y + size$)
6:         $points$.append(Point($randX, randY$))
7:     **end for**
8:     **return** $points$
9: **end function**

Algorithm 3 Sort points by polar angle to origin

1: **function** SORTPOINTS(*points*, *origin*)
2:     *hull* ← [*origin*]
3:     Sort *points* by Y-coordinate
4:     **for** i ← 0 **to** length(*points*) − 1 **do**
5:         *points*[*i*].*angle* ← CALCANGLE(*origin*, *points*[*i*])
6:     **end for**
7:     Sort *points* by angle
8:     Append *points* to *hull*
9:     **return** *hull*
10: **end function**

Algorithm 4 Main function

1: **function** MAIN
2:     *hull* ← []
3:     *origin* ← Point(20, 20)
4:     *field* ← GENPOINTS(20, *origin*, 400)
5:     *hull*.append(SORTPOINTS(*field*, *origin*))
6: **end function**

Algorithm 5 Main function with holes in shape

1: **procedure** MAIN
2:     *hull* ← empty list
3:     *origin* ← Point(20, 20)
4:     *field* ← GenPoints(20, *origin*, 400)
5:     add SortPoints(*field*, *origin*) to *hull*
6:     *hole*1*Base* ← Point(100, 100)
7:     *hole*1*Points* ← GenPoints(5, *hole*1*Base*, 50)
8:     add SortPoints(*hole*1*Points*, *hole*1*Base*) to *hull*
9:     *hole*2*Base* ← Point(150, 50)
10:     *hole*2*Points* ← GenPoints(3, *hole*2*Base*, 30)
11:     add SortPoints(*hole*2*Points*, *hole*2*Base*) to *hull*
12: **end procedure**

B. Aerial Mapping
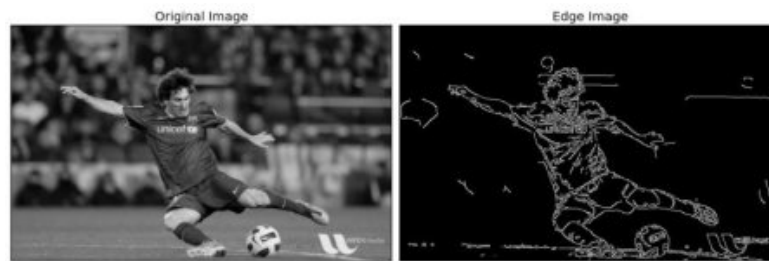
Figure 1: Opencv example of canny detection, source (Bradski et al., 2023) docs
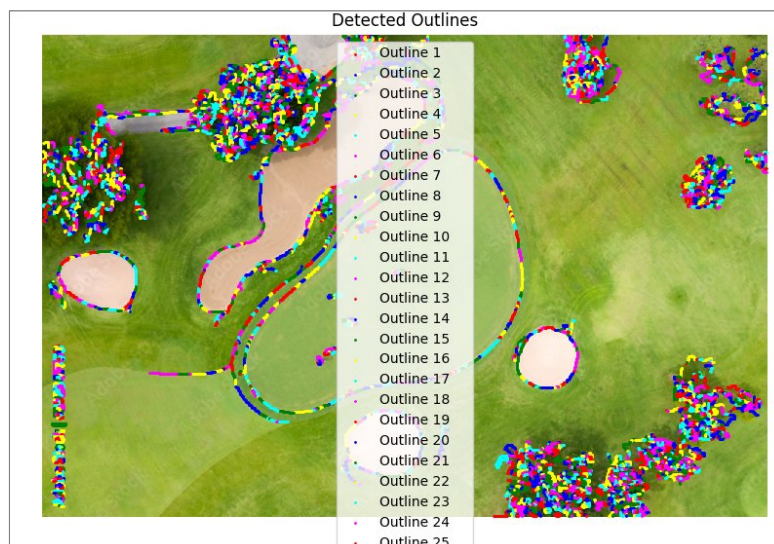
Figure 2: OpenCV canny edge detection on an image of a golf course
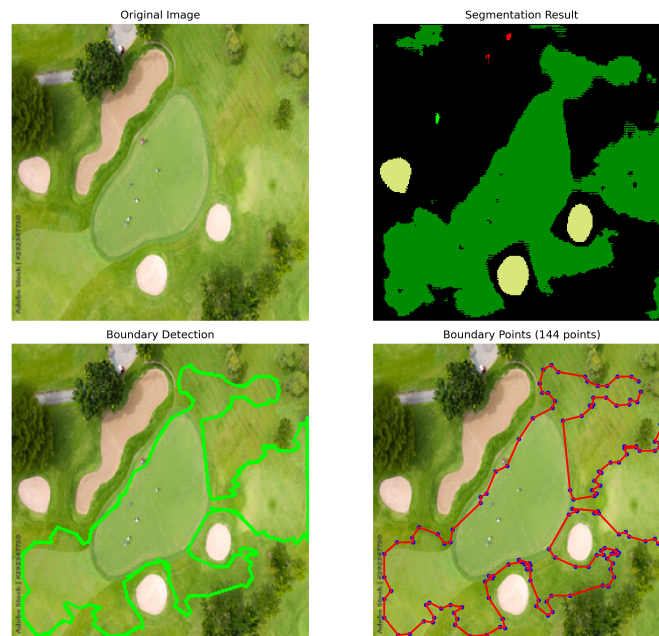
C. Performance and Evaluation

C.1. Mapping

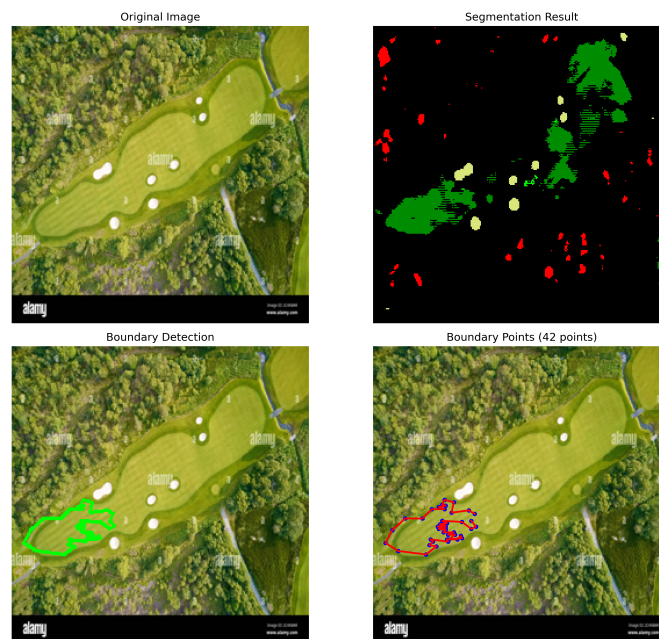Figure 3: An example ouput of pytorch model trained on the Danish Golf Course data set

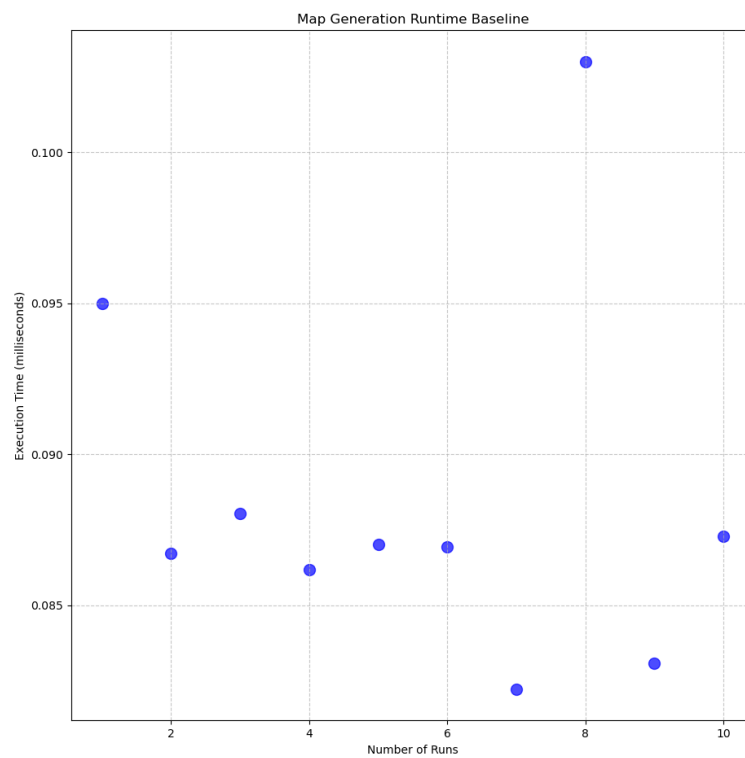Figure 4: An example ouput of pytorch model trained on the Danish Golf Course data set

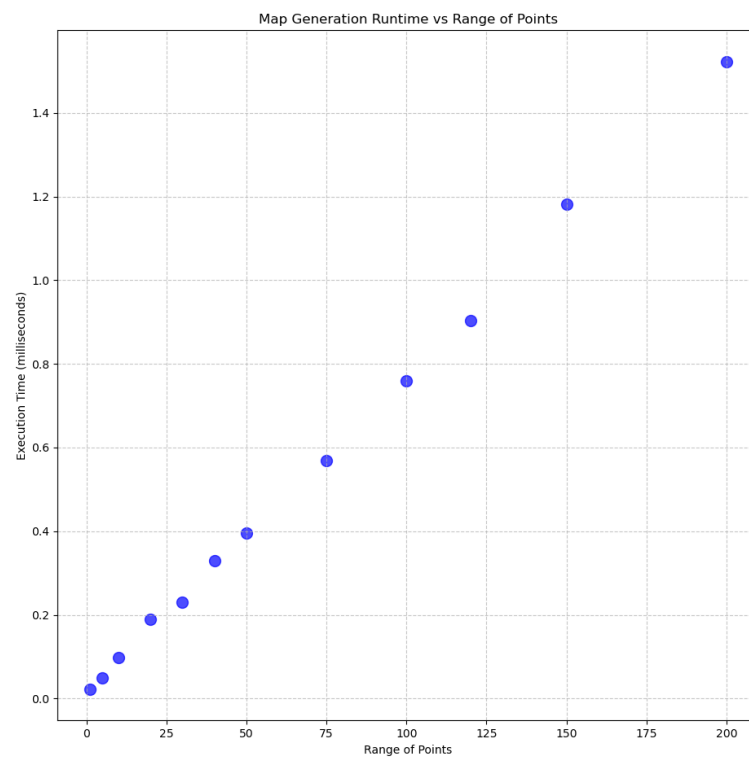Figure 5: Performance of the map generation algorithm with 10 points, 400 range, 0 holes

Figure 6: Performance of the map generation algorithm with variable points, 400 range, 0 holes
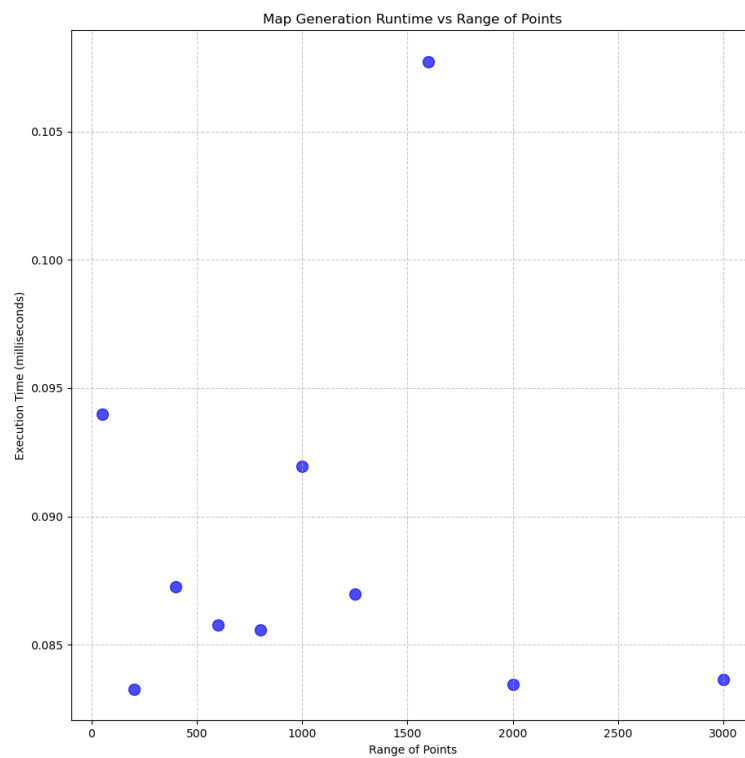
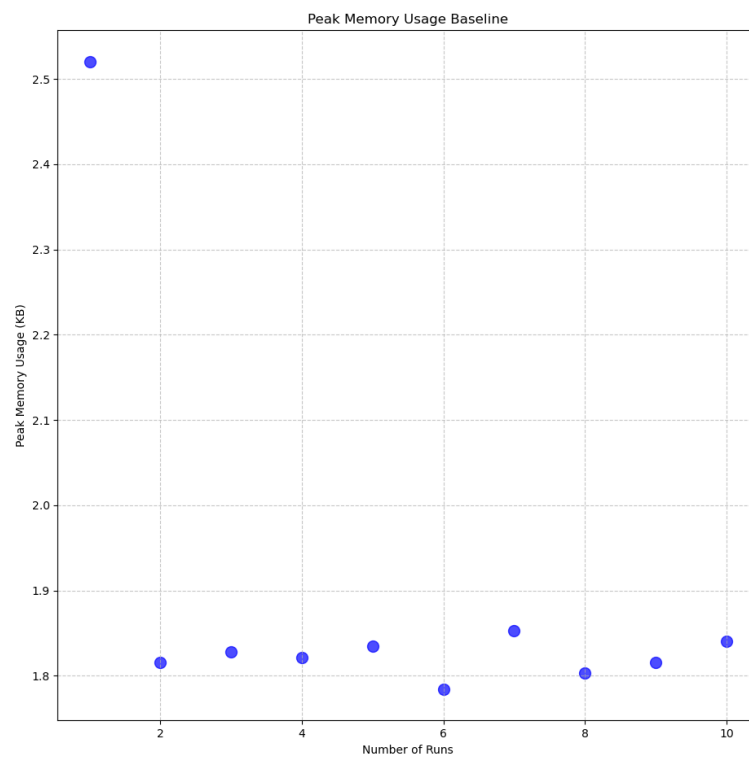Figure 7: Performance of the map generation algorithm with 10 points, variable range, 0 holes
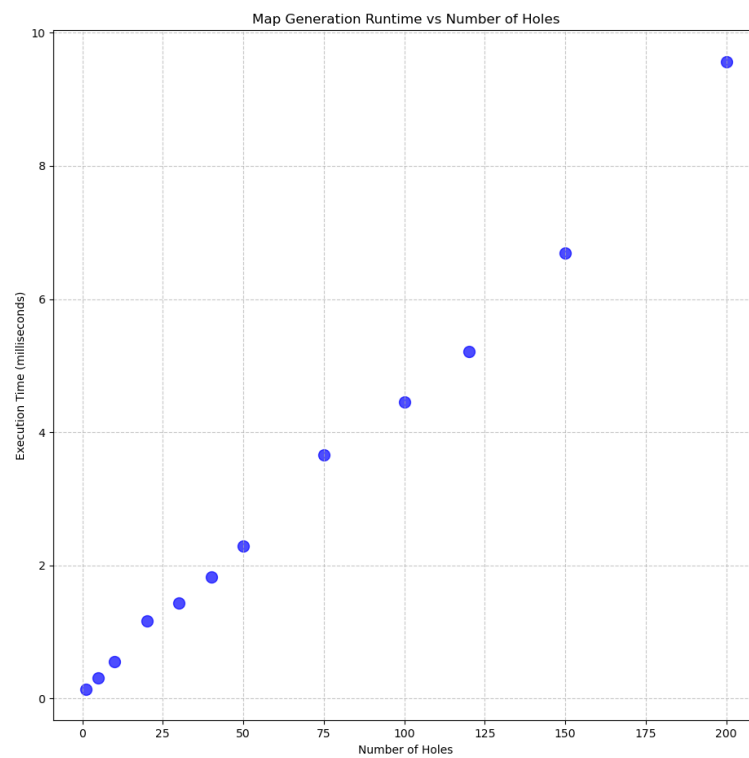
Figure 8: Performance of the map generation algorithm with 10 points, 400 range, variable holes

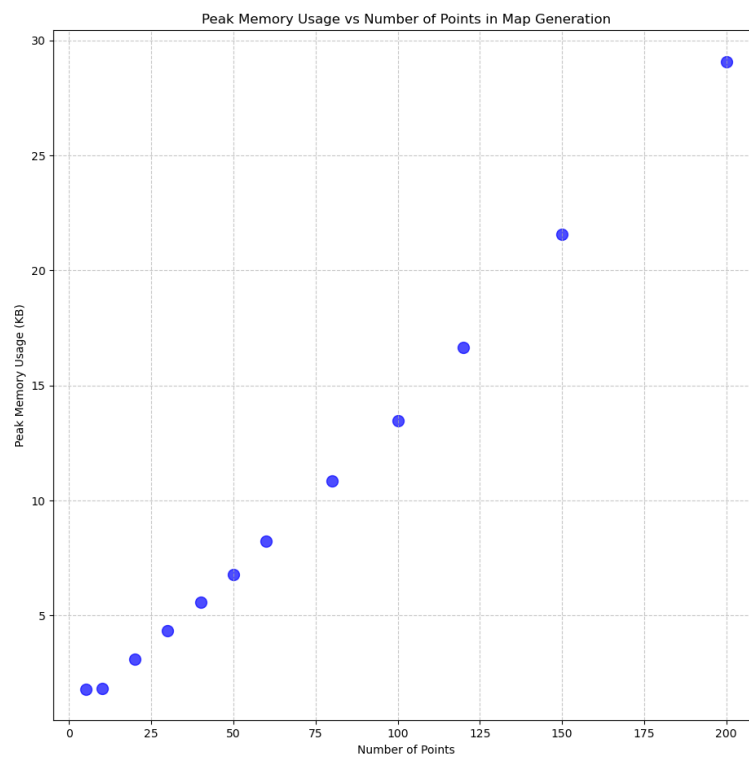Figure 9: Memory usage of map generation algorith with 10 points, 400 range, 0 holes

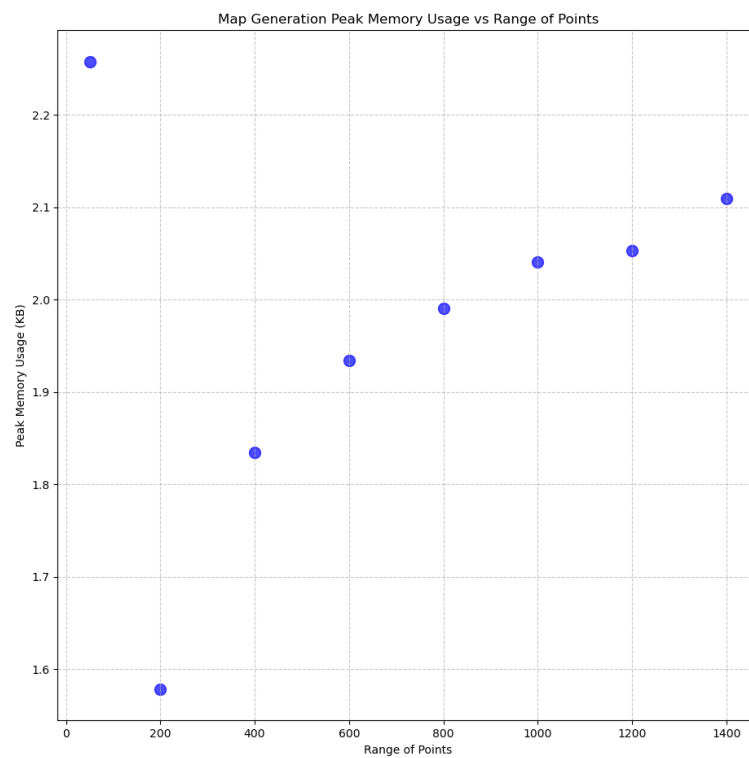Figure 10: Memory usage of map generation algorith with variable points, 400 range, 0
holes

Figure 11: Memory usage of map generation algorith with 10 points, variable range, 0 holes
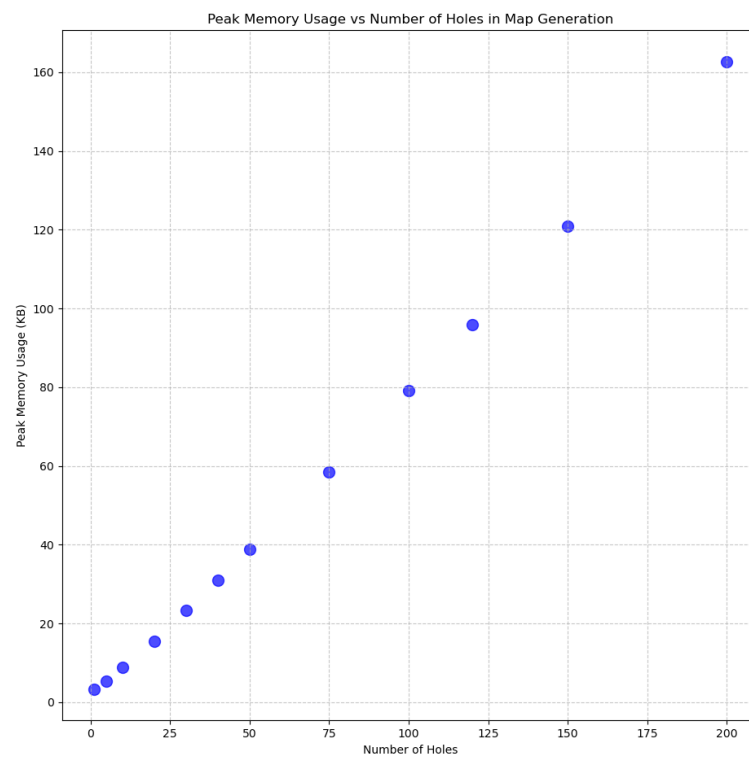
Figure 12: Memory usage of map generation algorith with 10 points, 400 range, variable holes