AvoidBench

A high-fidelity vision-based obstacle avoidance benchmarking suite for multi-rotors.

by

R.R. Veder

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday December 16, 2020 at 15:30.

Student number:	4460464	
Project duration:	November, 2019 – December, 2020	
Thesis committee:	Prof. dr. G.C.H.E de Croon,	TU Delft, supervisor
	Ir. C. de Wagter,	TU Delft
	J.F.P. Kooij,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

After a year of hard work with many ups and downs, I am happy to finally present you with my final master thesis. This thesis was written to obtain a masters degree from the faculty of aerospace engineering at the Technical University of Delft. In this work, we present a new tool for benchmarking obstacle avoidance algorithms in simulation.

This document consists of three parts. The first part contains the main thesis written in the form of a scientific article. The second part contains the literature study that was performed at the start of this project. And finally, the third and last part contains the appendix, where some extra results are showcased.

I would like to thank all people who were involved in the development of this project. First of all, I would like to specifically thank my supervisor Guido de Croon for providing the initial idea and for the countless meetings and feedback sessions. These helped me tremendously in the development of both the project and the final paper. I would also like to thank the people from the MAVLab for joining my preliminary presentation and providing helpful feedback from an outsiders perspective. Finally, I would like to thank my friends and girlfriend for cheering me on when things got rough.

In the end, I feel like I learned a lot of new valuable skills this past year, especially related to doing academic research. Although I do not see myself pursuing an academic career, it was certainly an interesting and eye-opening experience. I hope that with the release of this work, we can leave a lasting impression in the field of obstacle avoidance research.

R.R. Veder
Delft, December 2020

Contents

I

Master Thesis

AvoidBench: A high-fidelity vision-based obstacle avoidance benchmarking suite for multi-rotors.

Veder. R.R*
Control & Simulation
TU Delft Aerospace Engineering
Delft, Netherlands

Croon. G.C.H.E. van†
Control & Simulation
TU Delft Aerospace Engineering
Delft, Netherlands

***Abstract*—Obstacle avoidance is an essential topic in the field of autonomous drone research. When choosing an avoidance algorithm, many different options are available, each with their advantages and disadvantages. As there is currently no consensus on testing methods, it is quite challenging to compare the performance between algorithms. In this paper, we present AvoidBench, a benchmarking suite capable of evaluating the performance of vision-based obstacle avoidance algorithms for multi-rotors in simulation. Utilising a set of performance metrics, AvoidBench assigns performance scores to obstacle avoidance algorithms by subjecting them to a series of tasks. Using both Airsim and Unreal engine under the hood, we are able to provide high-fidelity visuals and dynamics, leading to a relatively small gap between simulation and reality. AvoidBench comes included with a simple, but powerful C++ and Python API which provides functionality for procedural environment generation, custom benchmark design, and an easy-to-use framework for users to implement their own vision-based obstacle avoidance methods. Implementing an obstacle avoidance method can be done entirely in a single file, allowing anyone to share and compare their obstacle detection and avoidance algorithms with others.**

***Index Terms*—Benchmark, Obstacle Avoidance**

I. Introduction

The interest in autonomous UAVs has continued to grow in both the academic and commercial sector. Use cases such as package delivery and indoor search and rescue have gained more attention in recent years. One common theme between all these cases is that obstacle avoidance plays a large role in how safe and effective these vehicles are. Unfortunately, there is currently no shared consensus on how obstacle avoidance algorithms should be tested. This makes it not only difficult to see how the field of obstacle avoidance as a whole is progressing, but it also makes it hard to compare the performance of different obstacle avoidance algorithms. Benchmarking can offer a solution to this problem.

In the last ten years, we have seen a large increase in the number of benchmarks being published in other fields, such as computer vision and AI. Benchmarks like KITTI [1] and ImageNet [2] provide large datasets and proper metrics for evaluating and comparing different algorithms. This allows researchers not only to compare state-of-the-art algorithms but also to see how performance has increased over time. Furthermore, it offers an invaluable platform for researchers to showcase and share their algorithms and garner more interest from the community. If executed well, a benchmark can give a boost to the entire community and research field.

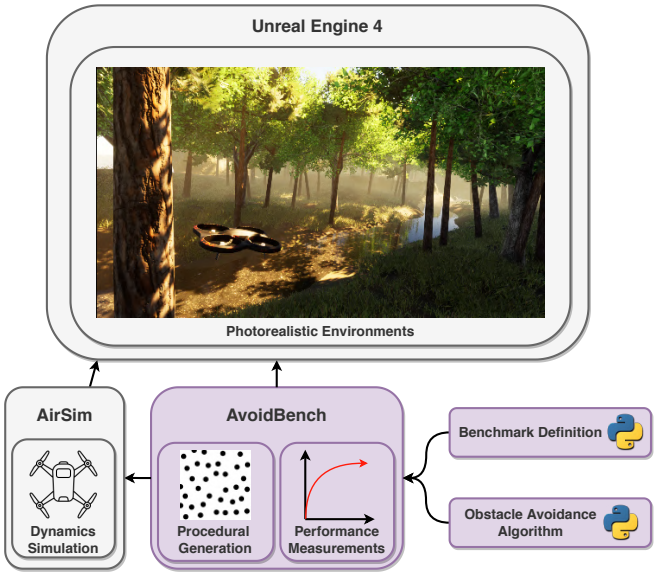


Fig. 1: Setup of AvoidBench. The user implements their obstacle avoidance algorithm in a Python script and selects the benchmark that should be performed. AvoidBench (purple elements) then automatically performs the benchmark using AirSim and Unreal (gray elements) for simulating the drone in visually realistic environments.

Unfortunately, we have not seen this same level of success in the more general field of robotics. One can only speculate on the reasons for this, but an important factor is that creating benchmarks for robotics is inherently more difficult than for fields such as computer vision. This is mainly because computer vision benchmarks are passive, requiring the algorithms only to observe the environment and not interact with it. This is in stark contrast with active robot tasks, in which the algorithm not only observes the environment, but

* Student
† Supervisor

also takes actions that will influence future observations. As one can imagine, this closed loop of observations and actions, also termed *"situatedness"*, makes it more challenging to design a centrally accessible benchmark. Creating such a benchmark with real robots would be timestaking, and - obviously - no two different robots are physically exactly the same, nor do they encounter the exact same real-world conditions. Hence, benchmarking for active robotic tasks is likely best done in simulation.

Although benchmarks in the field of robotics are definitely not as popular as computer vision benchmarks, there are still many noteworthy examples, both in real-world environments and simulation. SenseAct [3] focuses on reinforcement learning algorithms for continuous control of a real-life robot-arm, a ground-based robot and a motor actuator. Their tasks require users to use the same robots and have the same environmental conditions. Another benchmark is RLBench [4], which focuses on control of a robotic arm but in simulation. They provide more than 100 different tasks which all require the robot to interact with simulated objects on a table.

In addition to benchmarking suites, there is also the idea that robotics competitions can serve as a benchmark for performance. In [5], a framework is proposed for turning ordinary robotics competitions into benchmarking tools. This two-in-one approach of a competition also serving as a benchmark provides a really easy way to benchmark the bleeding edge of research. However, there are also disadvantages of using competitions in such a way. The performance of a robot in a competition is not only dependent on the robot itself, but also on factors such as the state of the hardware and the level of preparation of the team. If the team is having a bad day, it will show in the final performance evaluation.

Returning to the field of obstacle avoidance, some efforts have been made to allow for benchmarking of obstacle avoidance algorithms. In [6], several metrics were introduced for benchmarking obstacle avoidance algorithms. Environmental metrics were introduced in order to evaluate the difficulty of the obstacle avoidance task and allowing for fairer comparison and performance prediction in unknown, real-world environments. Various performance metrics were determined for different algorithms and environments of varying difficulty. The metrics were demonstrated with extensive real-world experiments. The high number of experiments required for a real robot confirms that ease of benchmarking will be best served by having a common simulation environment. In fact, a team from UGA introduced a simulation benchmark using ROS and RotorS called BOARR [7] [8]. Unfortunately, this benchmark is based on the Gazebo simulation environment, which leads to a large reality gap between simulated and real-world vision. This makes the simulator less suited for evaluating vision-based obstacle avoidance. Moreover, also the ease of use for implementing and benchmarking with BOARR could be further improved. Until now, this simulator has not gained much traction.

Our contribution addresses these issues in the hope that benchmarking becomes easier, faster and more reliable. In this paper we present ***AvoidBench***, a benchmarking suite capable of evaluating the performance of visual obstacle avoidance algorithms for multi-rotors in simulation. By making use of the broadly used Airsim simulator that relies on Unreal Engine 4 [9] [10], AvoidBench is able to provide photorealistic environments for development, testing and benchmarking purposes. AvoidBench comes included with a simple, but powerful C++ and Python API that provides functionality for custom benchmark design and procedural environment generation. With the main focus of AvoidBench being usability, it has been made as easy as possible to implement and share obstacle avoidance algorithms using only a single Python file.

The layout of this paper is as follows. First a brief overview is given of the related work. Afterwards, the AvoidBench benchmarking pipeline is explained. Subsequently, a section is dedicated to introducing the metrics that are used for the algorithm performance evaluation. Next, the software architecture and design decisions are discussed. And finally, a small experiment is presented where a custom benchmark is created and a generic obstacle avoidance algorithm is tested.

II. Related Work

Although the AvoidBench benchmarking suite was built from scratch, a lot of inspiration was drawn from existing projects during development. From the beginning, the decision was made to use an existing quadrotor simulator to save both time and put more focus on the benchmarking part. In this section, we will discuss some noteworthy simulators that were considered. Furthermore, we will take a closer look at some benchmarks in the field of robotics.

A. Simulators

Prior to the development of AvoidBench, a choice had to be made on the backend drone simulator. For this, we looked at a variety of different simulators. Gazebo based simulators Hector [11] and RotorS [8] were considered at first as both were mature simulators that served as a basis for many scientific studies. Both were ultimately rejected because of the low visual fidelity of the Gazebo simulator. This low fidelity would lead to a large sim2real gap, and hence reduce the generalisation of the benchmarking performances to real robotic platforms and environments.

Focusing more on simulators with photorealistic visuals, FlightGoggles and Airsim were considered. FlightGoggles uses both ROS and the Unity game engine [12]. At the time we considered it, the simulator was relatively new and had not yet garnered much support from the community. Furthermore,

we found that the ROS interface hampers usability, because (i) it limits the operating system of the user to specific versions of Ubuntu, and (ii) it requires users to program in C++ and be familiar with the ROS framework. For these reasons, the choice was made to not pick FlightGoggles as the simulator. The last considered simulator, Airsim, is developed by Microsoft and uses Unreal Engine 4. AirSim provides a pure C++ and Python API [9] to interface with the simulator. Additionally, the project includes an extensive documentation wiki, which is invaluable during development. AirSim also has a large active community, ensuring that there are always people available to help when issues arise.

Towards the end of our work for this article, a new photorealistic simulator was released named FlightMare [13]. Although we have no experience with this simulator, it looks promising and deserves a mention in this list. If it will be broadly used in the future, it may be incorporated as an option for the AvoidBench backend.

B. Benchmarks

The introduction already briefly mentioned passive benchmarks such as KITTI [1] and ImageNet [2]. Here, we would like to instead focus on some noteworthy active benchmarks, the first of which is AI Gym [14]. AI Gym is described as a toolkit for developing and comparing reinforcement learning algorithms. They offer a wide variety of so-called "environments" for the field of AI. Environments can be best described as singular reinforcement learning tasks where the user has to design a controller for different types of agents or robots with the goal of reaching a particular state. Although AI Gym's environments include some simulated robots and can be considered challenging, most tasks focus on a single aspect: The task performance and how many "samples" are required for reaching that performance. With AvoidBench we include multiple performance metrics for obstacle avoidance algorithms that we expect to generalise to real environments (cf. [6]).

Moving on to the field of autonomous UAVs, MAVBench [15] is a benchmarking simulator designed to measure the performance of drones executing different missions such as search-and-rescue, mapping and planning operations. Using AirSim and Unreal Engine 4 as a backend, MAVBench was - to the best of our knowledge - the first end-to-end photorealistic benchmarking suite for MAVs. Unlike AvoidBench however, MAVBench applied benchmarking to a much broader field and also chose to use ROS. Although many tasks are supported, MAVBench currently has no specific support for generating environments or measuring performance of obstacle avoidance algorithms.

Focusing on the field of obstacle avoidance, the most relevant work is BOARR [7]. BOARR is a benchmark suite specifically designed to evaluate the performance of obstacle avoidance algorithms. Unfortunately, as BOARR is using Gazebo as a simulator (in combination with RotorS and ROS), the visual fidelity is quite poor. We believe that in applications such as visual obstacle avoidance, visual realism plays a large role in how an algorithm performs. Real-world effects that would present problems for visual obstacle avoidance algorithms such as lens-flare and reflections can often not be properly simulated in low visual-fidelity simulators. By choosing a photorealistic simulator, we can ensure that simulated results transfer better to the real world.

The benchmark and simulator that was most influential to AvoidBench was CARLA [16]. CARLA is a software package for the development, training, and validation of autonomous cars. Also using Unreal Engine 4 as the main rendering engine, CARLA provided references and best practices that aided in the development of AvoidBench. These were related to the utilisation of remote procedure calls (RPC) and the design of the user-exposed API.

III. Benchmarking Pipeline

AvoidBench has a custom benchmarking pipeline that allows anyone to create, run and share their own benchmarks. Every benchmark requires at least two parts to function correctly. A set of **tasks** that need to be completed by an algorithm, and a set of **metrics** that evaluate how well the algorithm is able to perform those tasks. Getting these two concepts right is critical to the success and usefulness of the benchmark. In the case of obstacle avoidance benchmarking, it is not immediately clear what tasks the algorithms should perform and what should be measured. There are multiple approaches one could take when benchmarking obstacle avoidance algorithms. When looking at the original applications of these algorithms, they are often used in either an exploratory or a goal-based fashion. As an example, in [17], a flapping wing drone explored various environments, autonomously avoiding obstacles without specific goal waypoints. However, most work studies use goal-based navigation, such as the quadrotor in [18] that avoids obstacles in its way to a goal location. In AvoidBench, the decision was made to test using a goal-based approach. This ensures that all algorithms are indirectly forced to travel through an obstacle field on their way to the goal. By giving the benchmark creator control on how this obstacle field is generated, it becomes possible to create tasks of varying difficulty. During the execution of a task, data should be measured such that it becomes possible to create a performance evaluation. In [6], various metrics were introduced to quantify the performance of different aspects of obstacle avoidance algorithms in a way that can generalise to the real world. In AvoidBench, we build upon these metrics and present new and modified ones to create a performance evaluation.

In this section, we shall give an in-depth description of the complete AvoidBench pipeline. First, we shall introduce some key concepts that are required to understand AvoidBench terminology. Next, an explanation is given on how benchmarks are built, and finally, the metrics that evaluate the performance of algorithms are discussed.

A. Key Concepts

The AvoidBench benchmarking pipeline is not necessarily complex, but it does require the user to understand some basic concepts before being able to interact with it. Two of the main concepts are that of a ***benchmark*** and a ***task***. A ***benchmark*** is best defined as a container for multiple ***tasks***. Each ***task*** provides AvoidBench with specific parameters on how the simulator should be set up. These include descriptions on which ***map*** should be used, which ***mission*** the algorithm should be given and how the ***procedural generation system*** should be initialised. In the following paragraphs, these concepts will be introduced in more detail.

1) Map: The *map* in AvoidBench is defined as a user-created environment in which the obstacle avoidance algorithms are benchmarked. These maps can either be created as template maps, to be used in tandem with AvoidBench's procedural generation system, or as completed maps with all obstacle instances predefined. During a benchmarking session, multiple unique maps can be used to test the algorithm in a variety of different conditions. The initial release of AvoidBench includes two example maps that are modelled after a generic forest environment and an empty factory (as seen in Figure 2). These maps can be used to benchmark obstacle avoidance algorithms both in indoor and outdoor conditions respectively.

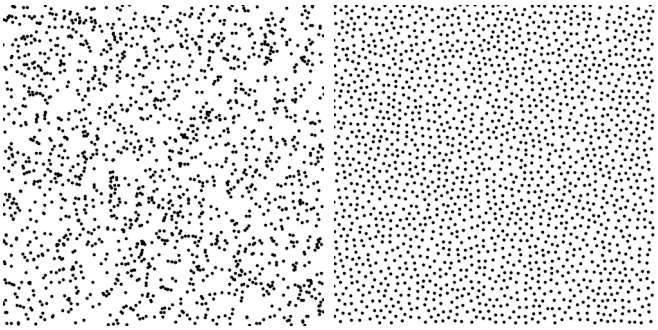
AvoidBench maps are created using the Unreal Engine 4 editor and are packaged with the executable during compile time. Currently, AvoidBench does not support sharing of custom maps as separate files. Therefore, user-created maps will have to be submitted to the central AvoidBench repository for everyone to benefit from them.

(a) The forest map serves as an outdoor testing environment for obstacle avoidance algorithms.

(b) The factory map serves as an indoor testing environment for obstacle avoidance algorithms.

Fig. 2: Screenshots of the included forest and factory maps from inside Unreal Engine 4

2) Mission: During each task, the obstacle avoidance algorithm is given a *"mission"* with the the purpose to safely move the simulated drone from a start location to a goal location in the map. If the algorithm is able to command the drone to the goal without colliding with any obstacles, the mission is marked as "completed". Conversely, if a collision occurs, the mission is marked as "failed". Apart from colliding, there can be a variety of situations that lead to mission failure. An algorithm might get stuck in a local minimum, unable to find or move to the goal. By setting a maximum mission time parameter, AvoidBench automatically stops the mission if the time limit has been exceeded. There is also the possibility of the drone leaving the mission area. This is addressed by defining a mission-specific bounding box that ensures the mission is terminated if the drone leaves the specified area.



(a) Obstacle locations when using uniform random sampling. With this type of sampling, the obstacle density - and hence the difficulty of performing obstacle avoidance - varies over the environment.

(b) Obstacle locations when using Poisson disc sampling. The obstacle density - and hence difficulty of performing obstacle avoidance - varies much less over the environment.

Fig. 3: Comparison between uniform and Poisson Disc sampling methods

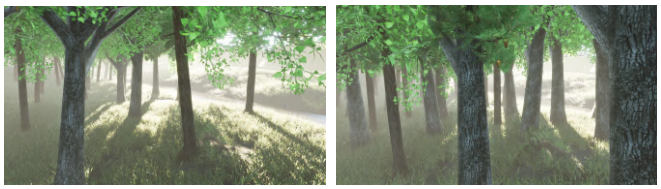
Currently, missions are defined inside the Unreal Engine 4 editor. The user is able to create multiple missions per map, where each mission can have a separate start and goal location. In a future version of AvoidBench, it will be possible to generate missions using the user-exposed API. This will allow users to dynamically generate new missions.

3) Procedural Generation System: In order to characterise obstacle avoidance performance in a statistically sound manner, an obstacle avoidance algorithm has to be tested in many different maps. Manual creation of a large number of maps (e.g., $>>$ 100) is an enormous task and may lead to unintended bias on the part of the human developer. Instead, the *procedural generation system* automatically generates obstacle fields inside a pre-designed empty environment based on a number of parameters.

Using the Unreal Engine 4 editor, every obstacle field that is to be procedurally generated must first be assigned a unique *Name*. Two parameters must then be defined which are the *bounding-area* and the *obstacle-type set*. The *bounding-area* specifies the space in which the procedural generation system will place the obstacles. These obstacles will be randomly selected instances of the obstacles defined in the *obstacle-type set*. Once these two parameters are defined, it becomes possible to generate an unlimited number of different obstacle fields. The user must inform AvoidBench on how the obstacle field is to be generated. This is done by specifying the *Name* of the procedural generation system with a *Radius* and a *Seed* parameter. The obstacles will then be

generated in such a way that the distance between obstacles is always within 1 and 2 *radius* units.

When generating obstacle fields, it is important that the placement of obstacles is uniform across the entire area to ensure the traversal difficulty is independent of location. This is especially important during benchmarking, as it makes it much easier to relate algorithm performance to environmental conditions. To generate the actual obstacle field, AvoidBench utilises a rudimentary and well-known method for procedural generation named *Poisson Disc Sampling* [19]. This method has a considerable advantage over random sampling as it creates a uniform distribution of obstacles. Furthermore, it allows us to control the minimum distance between obstacles. This fine control allows us to increase the difficulty of tasks incrementally, thus obtaining the performance of an algorithm over a wide array of obstacle densities. A visual example of the difference between random uniform sampling and *Poisson Disc Sampling* is given in Figure 3. Figure 4 shows how *Poisson Disc Sampling* can be used to procedurally generate both a sparse and dense forest.



(a) A procedurally generated forest using a Poisson disc radius of 10 meters.

(b) A procedurally generated forest using a Poisson disc radius of 6 meters.

Fig. 4: Example of Poisson disc procedural generation in the forest map.

4) Task: The *task* was already briefly introduced in the introduction of this section. Here we will give a more in-depth view on how it is defined and used within AvoidBench. A task can be best described as a data structure containing the parameters needed to initiate the AvoidBench simulation. These parameters are listed below.

- Task name
- Trials
- Map
- Mission
- Procedural Generation System Description

The *Task name* parameter gives each task a unique identifier. The *Trials* parameter allows the user to specify how many times the *Mission* should be performed in succession. As missions are bound to specific maps, one must ensure both the *Map* and *Mission* are compatible. Finally, the *Procedural Generation System Description* describes how the obstacle field(s) should be generated. For each obstacle field, the user has to supply the *Name*, *Radius* and optionally the *Seed* parameter. Although the generation of multiple obstacle fields per task is supported, it is not recommended as this makes it harder to relate algorithm performance to a specific setting.

B. Benchmarking Pipeline Description

With all the main concepts discussed, it is now possible to introduce the AvoidBench benchmarking pipeline. The first step in the pipeline is for the user to supply AvoidBench with an obstacle avoidance algorithm and a benchmark. Given that a benchmark consists of multiple *tasks*, AvoidBench starts a verification procedure to ensure that every task is well-defined. It performs the verification by confirming that the specified *map*, *mission* and *procedural generation system* names for each task exist.

Once the correctness of every task is verified, AvoidBench selects the first task in the list and instructs Unreal Engine 4 to load the specified *map*. The procedural generation system then populates the world with obstacles based on the parameters in the *procedural generation system description*. Finally, the selected *mission* is started, and the algorithm is allowed to control the drone. Throughout the mission, AvoidBench measures a set of metrics which are later used for the performance evaluation.

Once a *mission* is completed, either due to success or failure, AvoidBench will end the current *mission*. Depending on the number of specified trials, AvoidBench will either reload the same *mission* or move on to the next task in the list. Once all tasks are performed, the results of all metrics are saved to a JSON output file. A visual representation of the entire AvoidBench benchmarking pipeline can be seen in Figure 5.

C. Metrics

In order to properly assess the performance of each obstacle avoidance algorithm during a benchmark, AvoidBench measures a set of metrics throughout the missions which try to quantify the efficiency, speed, and accuracy of each algorithm. As the performance of the algorithm is highly dependent on the environmental conditions, AvoidBench also tries to quantify the complexity of the environment. By measuring both performance and environment metrics, it becomes possible to relate the performance of an algorithm to the type of environment and its corresponding difficulty.

In the following sections, different metrics will be introduced, and a brief explanation of each of them will be given.

1) Performance Metrics: AvoidBench currently measures five different types of performance metrics:

- Optimality Factor
- Average Goal Velocity
- Collision Percentage
- Mission Progress
- Algorithm Processing Time

a) Optimality Factor: The *Optimality Factor* measures how efficiently the algorithm is able to fly the drone from the start location to the goal location. In [6], the metric for total travelled distance was used as a way to measure path

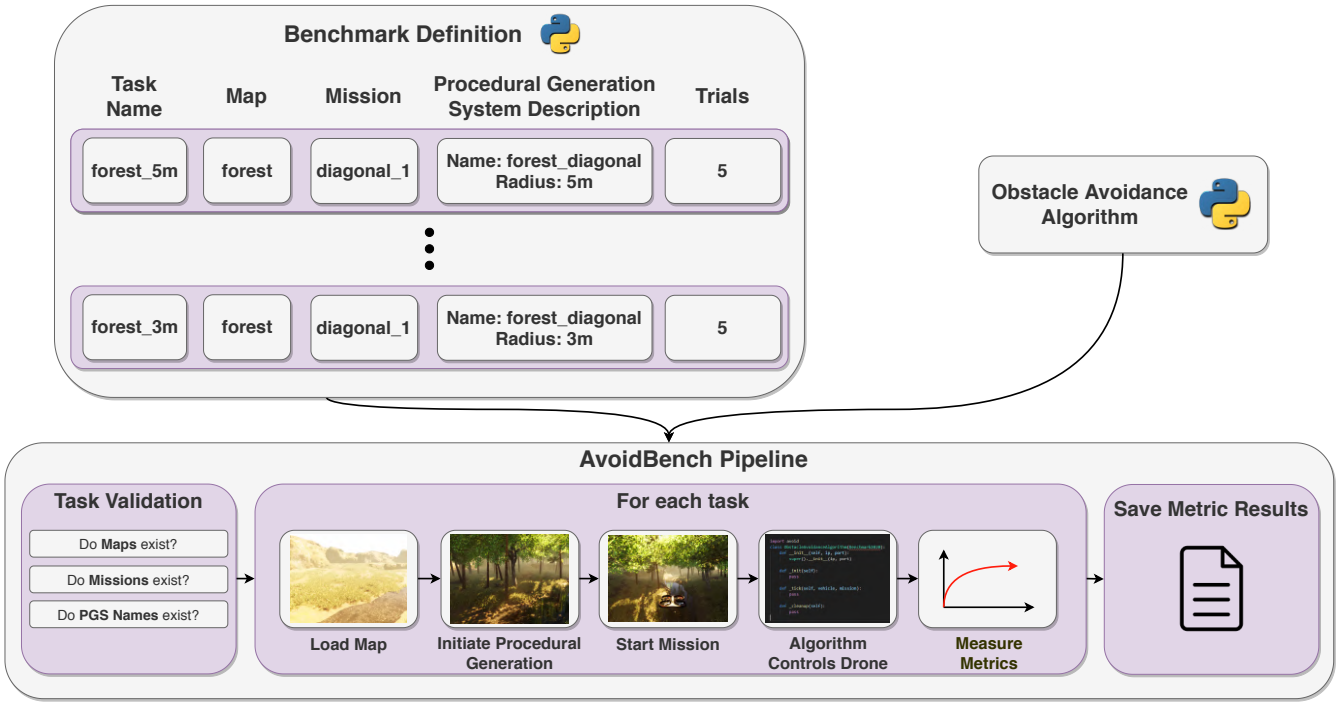


Fig. 5: A visual representation of the AvoidBench benchmarking pipeline. The user supplies AvoidBench with an (existing) benchmark and an obstacle avoidance algorithm. AvoidBench first verifies all the tasks. Afterwards, for each task, the simulation is initiated and the algorithm is allowed to control the drone. During each mission, AvoidBench measures multiple metrics to create a performance evaluation. Finally, once the benchmark is completed, all metric results are saved to a file.

optimality. Although this gives a good indication on how efficient the algorithm is at traversing the map to the goal location, due to the fact that the metric is dimensional, it is difficult to compare the performance between different maps and missions. This is the case because, for different missions, the distance between the start and goal location is almost never the same.

We take this issue into account and go a step further by adjusting the travelled distance by the optimal non-colliding path. Before the mission commences, AvoidBench determines the shortest non-colliding path to the goal and calculates the length d_{opt}. Once the mission has started, AvoidBench tracks the total distance travelled by the drone, d_{trav}. On successful completion of the mission, the *Optimality Factor* is calculated using:

$$OF = \frac{d_{trav} - d_{opt}}{d_{opt}} \times 100\% \tag{1}$$

Given that d_{opt} is the optimal path, and therefore the minimum distance required in order to reach the goal, the *Optimality factor* will always be larger than 0%.

b) Average Goal Velocity: The purpose of the *Average Goal Velocity* metric is to measure the speed at which an algorithm is able to traverse the obstacle field and reach the goal location. Usually, a simple measure of mission time, the time it takes for the drone to travel from the start location to the goal location, would suffice. However, to ensure that results are also comparable between different missions and maps, the mission time is converted to a velocity value using the optimal mission distance, as shown in formula 2. Here, $t_{mission}$ is defined as the mission time and d_{opt} is defined as the optimal, i.e., shortest distance.

$$AGV = \frac{d_{opt}}{t_{mission}} \tag{2}$$

The main reason d_{opt} is used instead of d_{trav}, is because using d_{trav} in Equation 2 would give the average velocity. Although the average velocity seems like a good metric to measure the traversal speed, it breaks down when the drone moves in different directions than the goal direction. A drone that would be stuck in the middle of the obstacle field for a given period, flying circles at a constant velocity of 1 m/s, would have the same average velocity as a drone flying straight to the goal without interruption at 1 m/s. Using d_{opt} better shows how fast the drone effectively moves towards the goal.

c) Collision Percentage: Every mission, there is the possibility of the drone hitting an obstacle and failing the mission. To quantify these occurrences, AvoidBench calculates a *Collision Percentage* for every task using equation 3.

$$CP = \frac{\#\text{Collisions}}{\#\text{Trials}} \times 100\% \tag{3}$$

As was already discussed, each *task* defines a *Trials* parameter that specifies how often a particular mission is performed in succession. As the number of mission trials increases, the statistical reliability of the *Collision Percentage* increases and becomes better suited as a metric to measure performance. However, this does come at the cost of increased benchmarking evaluation time. As AvoidBench is configured to run in real-time, it is currently not possible to speed up this process. In this paper, the choice was made to have 30 trials per task.

d) Mission Progress: The *Mission Progress* metric measures how far the drone is able to progress to the goal location. In the cases where the drone is unable to reach the goal, the *Average Goal Velocity* and *Optimality Factor* become unusable and the only information that is obtained during that specific trial is that the drone has failed the task. The Mission Progress metric allows us to capture how far the drone has advanced and thus gives a quantifiable method for measuring how well obstacles were avoided.

In AvoidBench, *Mission Progress* is calculated based on vector projection. Given vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ as seen in Figure 6, the travelled distance of the drone along the start-goal vector can be calculated. To obtain the final *Mission Progress* value, this distance is divided by the start-goal distance as seen in Equation 4.

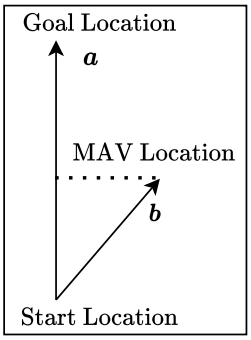


Fig. 6: Variable definitions used for the calculation of the *Mission Progress* metric. $\boldsymbol{a}$ signifies the vector spanning from the start location to the goal location, while $\boldsymbol{b}$ is the vector from the start location to the final drone location.

$$MP = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}|^2} \times 100\% \tag{4}$$

e) Algorithm Processing Time: The last performance metric, *Algorithm Processing Time*, directly measures how long a single iteration of the obstacle avoidance algorithm takes. During the mission, all processing periods are stored, which allows AvoidBench to calculate statistics such as the mean and standard deviation of these values. The processing time is machine-specific, and hence algorithms would have to run on the same machine for a fair comparison. As AvoidBench runs in real-time, it should be noted that obstacle avoidance algorithms with long processing times will therefore experience an inherent disadvantage.

2) Environment Metrics: AvoidBench currently measures two different environment metrics, namely *Traversability* and *Relative Gap Size*. In this section, a brief explanation on each of these metrics is given.

a) Traversability: Traversability was introduced in [6] as a metric to quantify how difficult it would be for a drone to traverse an environment. It was presented as an alternative to obstacle density which had the issue that it did not accurately capture avoidance difficulty as the dimensions of the drone were not taken into account. *Traversability* was originally calculated by picking N random positions p in the environment. For each position, a random heading h would be selected, and a ray was cast to calculate the free-flight distance s. All the results would then be averaged to obtain the uncorrected Traversability. The final step to make the metric dimensionless was to divide the result by the drone diameter d_{drone}.

$$TRAV = \frac{1}{d_{drone} \cdot N} \sum_{i}^{N} s(p(i), h(i)) \tag{5}$$

In AvoidBench, we apply some changes to the original traversability metric to make the results more repeatable. Since every free-path calculation s requires two random number samples, the results for different traversability calculations in the same environment can differ significantly. To combat this, we use a grid-based sampling method to sample locations, which removes the probabilistic component of the original metric. Additionally, at each location, instead of randomly picking a single direction vector for the free-path calculation s, N direction vectors are generated evenly across a 2D circle. For each direction vector, the free-path length is calculated, and all results are averaged to obtain a single $s_{average}$ per location. Finally, the results of all locations are averaged to find the *Traversability*.

The advantage of using *Traversability* is that this metric can be calculated for any environment. By being able to relate the performance of the drone to environmental metrics, it becomes possible to make educated guesses on how the drone will perform in unknown, real-world environments. Furthermore, by measuring the traversability in a given environment, tasks can be generated with similar levels of difficulty if the relation between *Poisson Disc radius* and *Traversability* is known. Unfortunately, *Traversability* can be quite computationally expensive, especially if a fine grid is chosen with a large number of samples per location. Calculating traversability in real-life is also tricky, as it requires an occupancy grid of the environment with all the obstacles.

b) Relative Gap Size: The final environment metric, *Relative Gap Size*, is directly related to the *procedural generation system*. This metric is, therefore, only available during missions where the generation system has populated the environment with obstacles. As the name implies, this metric uses the *Poisson Disc radius* parameter $r_{poisson}$ from the procedural generation system description. It adjusts this radius by subtracting the average obstacle width $\hat{w}_{obstacle}$ and divides it by the drone diameter d_{drone} to make it dimensionless. The *Relative Gap Size* value represents the minimum space between obstacles expressed in multiples of the drone's diameter. The primary purpose for calculating the *Relative Gap Size* is that it offers users a more intuitive measure of traversal difficulty than the *Poisson Disc radius*.

$$RGS = \frac{r_{poisson} - \hat{w}_{obstacle}}{d_{drone}} \tag{6}$$

The *Relative Gap Size* should never be below a value of 1. In that case, the drone would be unable to traverse through the obstacle field as the obstacles are positioned too close together. The relation between *Relative Gap Size* and *Traversability* is further explored in subsection V-C.

IV. Software Architecture

From the start, AvoidBench was designed to be an extendable and easy-to-use benchmarking suite. This required a modular system architecture with a simple method to interact with the software. The decision was made to utilise C++ for the AvoidBench backend systems, and Python for the user-exposed API. To simulate and control the drone, the already existing software package Airsim was used. This allowed us to focus the project purely on the benchmarking aspect without having to worry about the simulation aspect.

In this section, we will discuss the high-level software architecture of AvoidBench and briefly explain how benchmarks and algorithms are implemented in AvoidBench.

A. AvoidBench Modules

AvoidBench is a software suite that consists of three main modules: the Unreal Engine 4 Plugin, AvoidLib and the Python API. As was mentioned in the introduction, AvoidBench uses AirSim as the drone simulator backend. It provides functionality related to the simulation, control and data retrieval of the drone. However, for AvoidBench to work properly, more features were added in order to create a benchmarking suite. These were related to the loading of Unreal Engine levels, loading of missions and dealing with the procedural environment generation. With these features added, it became possible to implement the entire AvoidBench benchmarking pipeline. In the following sections, a brief explanation of each of the AvoidBench modules is given.

1) AvoidBench Unreal Engine 4 Plugin: The Unreal Engine 4 plugin is the module that directly interacts with the Unreal Game Engine. It implements all the necessary functionality needed for the benchmarking pipeline to work such as the Poisson Disc procedural generation, the concept of missions, metrics and level loading. This is also the location where AvoidBench interacts with the AirSim backend to control and reposition the drone. AvoidBench re-exposes a part of the AirSim API responsible for controlling the drone and retrieving images. This decision was made to ensure that if ever the need arises to switch to a different simulator backend, only a single part of the software would need to be rewritten. During compile-time, the AvoidBench plugin is packaged with the AirSim plugin and Unreal Engine to create a single executable.

2) AvoidLib: The AvoidLib module contains the source code for two programs. The first is the AvoidLib Client, which is the C++ application that users can use to communicate with the AvoidBench executable. The second, AvoidLib Server, is used within the AvoidBench Executable to receive commands from the client. Communication between the client and the server is performed using remote procedure calls (RPC), which is a form of Inter-Process Communication. The benefits of RPC is that latency is very low, in low-load cases often lower than 1 ms, allowing commands to be executed almost immediately.

AvoidLib Client provides API endpoints for all the previously mentioned functionality. This functionality is accessed through the main four classes:

- Client
- Vehicle
- Mission
- World

The *Client* class handles the actual connection to the AvoidBench executable. Once connected, the user can use the *Client* class to obtain instances of the *Vehicle*, *Mission* and *World* classes. The *Vehicle* instance allows the user to send control commands to the drone. Currently, these control commands can be in the form of position reference, velocity reference, attitude reference, rate reference and direct motor references. These are then all forwarded to the AirSim module inside the AvoidBench executable. The user can also use the *Vehicle* class to obtain the drone state and forward-facing images. When loading a mission, the user is returned a *Mission* instance that allows them to start and stop the mission. While the mission is running, the user can query information about the mission such as the current status, mission time and the start and goal location.

3) Python API: The last module in AvoidBench is the Python API. Utilising PyBind11*, a C++ function wrapper, the original AvoidLib Client C++ API is converted to a

*https://github.com/pybind/pybind11

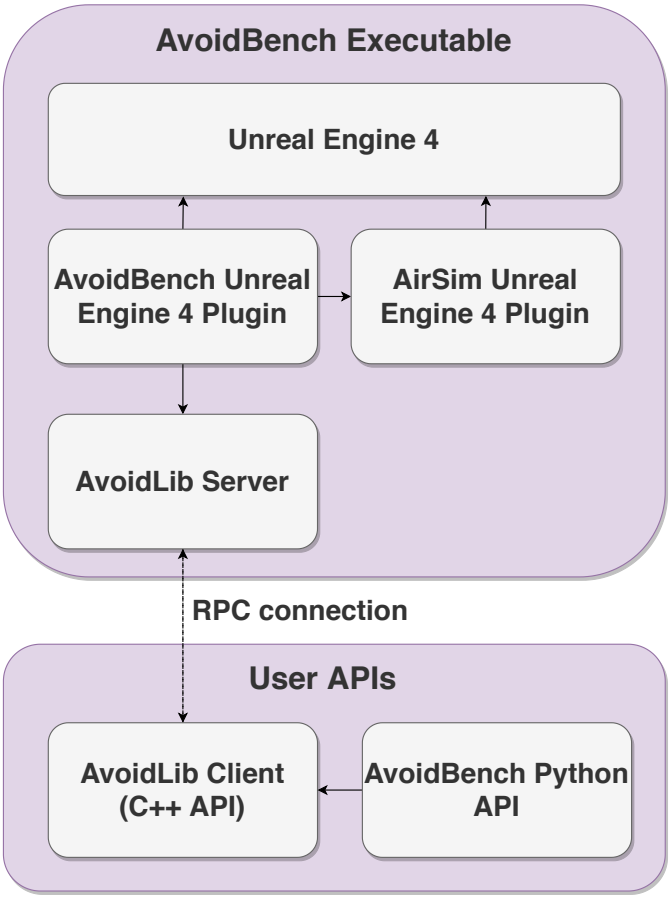


Fig. 7: A high level overview of the AvoidBench software architecture. The user must download two programs to be able to use AvoidBench. The first is the AvoidBench executable, responsible for running the simulation, benchmarks and performance evaluations. The second program is the user API, which is used to instruct the AvoidBench executable. The user can choose to work with the original C++ API, or the easier to use Python API.

Python API. This module is entirely self-contained and works similarly to the C++ API. Interacting with the simulator using Python only requires the user to download the Python AvoidBench package (which includes AvoidLib) and the AvoidBench executable. This will then allow to the user to create benchmarks, implement obstacle avoidance algorithms and freely fly in the environments. A full high-level architecture overview of the AvoidBench software suite can be seen in Figure 7.

B. Benchmarking Architecture

Now that the general architecture of AvoidBench has been discussed, the last section will focus on how benchmarks are implemented and executed. The general Python and C++ APIs contain all the functionality needed for anyone to interact with the AvoidBench benchmarking suite. However, when running a benchmark, only certain parts of the API are exposed. The user should not be able to utilise the API in such a way that they can modify the benchmark while it is running. To properly convert the existing API into one that can support benchmarking, two modules are needed. One to define the benchmark, and one to run it.

This is achieved by introducing a separate benchmarking interface class to both the C++ and Python APIs. The benchmarking interface provides the user with functionality to create custom **benchmarks** and define **tasks**. As discussed earlier, these tasks are described by a map name, mission name, procedural location settings and the number of trials. The interface also provides functionality for implementing obstacle avoidance algorithms. It does so by exposing three virtual functions called *Init*, *Tick* and *CleanUp*. Once a task is started, AvoidBench will take care of calling the *Init* and *Cleanup* functions at the start and end of the mission respectively. While the mission is running, the *Tick* function is called in a loop. Every iteration AvoidBench passes the *Vehicle* instance class and mission data information to the *Tick* function. This allows the algorithm to control the drone, receive images and obtain its state, without it having access to all of the API. This ensures that the algorithm is only in control of the drone and cannot e.g., access the obstacle locations in an extraneous way.

Although the *Benchmark Interface* class can be fully implemented with only one child class, this is not the recommended approach. AvoidBench uses a two-step system for defining benchmarks and implementing algorithms. First, a *Benchmark Definition* class is created which inherits from the *Benchmark Interface* class. The purpose of this class is only to define the tasks and register them with the interface. It does not implement the virtual functions, thus making the class incomplete. The second step is where the actual implementation of the algorithm occurs. The *Algorithm Implementation* class inherits from the *Benchmark Definition* class and implements the virtual *Init*, *Tick* and *CleanUp* functions. This makes it possible to quickly write different algorithms and benchmark them using the same tasks. A visual representation of this can be seen in Figure 8.

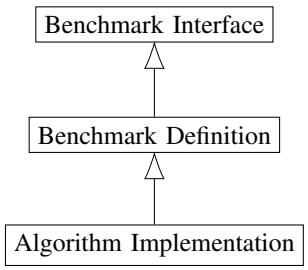


Fig. 8: Recommended inheritance approach when defining benchmarks and implementing algorithms

V. Experiment

With the publication of AvoidBench, a custom benchmark definition and algorithm is included to show the basic working principles of the benchmarking suite. We will briefly present details on both the structure of the benchmark and the obstacle avoidance algorithm. Finally, we will discuss the results of the benchmark.

A. Benchmark Definition

For this experiment, a custom benchmark is created to test obstacle avoidance algorithms on their ability to avoid obstacles with different levels of contrast. We choose to focus on this because most vision algorithm heavily rely on contrast. This is for instance the case for stereo vision algorithms or for optical flow algorithms. The obstacle set used for this benchmark consists of coloured cylindrical pillars with a checkerboard pattern as seen in Figure 9. The distinction is made between six levels of contrast where each level has a solid colour overlay with a percentage of 0%, 20%, 40%, 60%, 80% and 100%. In total six obstacle sets are created where each set contains all four colours with a fixed overlay percentage. These sets are given the names L0, L1, L2, L3, L4 and L5.

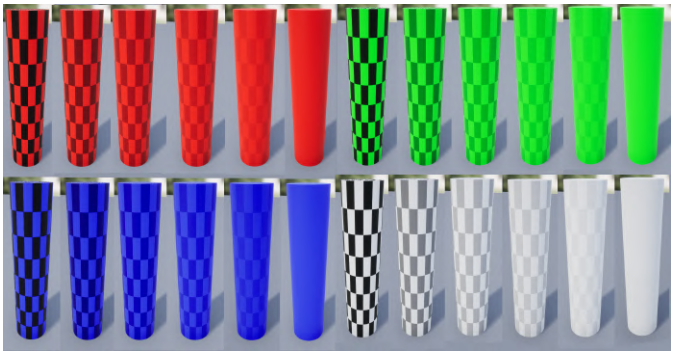


Fig. 9: Checkerboard obstacles with four solid colour overlays in red, green, blue and white. For each colour, the respective overlay is applied from left to right with an opacity of 0%, 20%, 40%, 60%, 80% and 100%.

For each obstacle set, five environments with increasingly smaller Poisson disc radii are generated, making the obstacle field progressively more dense. This benchmark uses the following five radii: 4m, 3.5m, 3m, 2.5m and 2m. Here, the difference between each successive radius is equivalent to the drone's diameter. The decision was made to have a combined number of trials of 30 for each radius and obstacle set type. Every three trials, the seed for the random number generator is changed while keeping the Poisson radius constant. This ensures that the performance is not tied to a single obstacle field pattern. As the Poisson radius remains unchanged between seeds, the system will guarantee that environments of similar difficulty are generated.

With six obstacle sets, five levels of difficulty and 30 trials per task, a total of 900 trials are performed during the benchmark. The maximum mission time is set to 180 seconds and given that the distance from the start to the goal is 26.5 meters, an algorithm must be able to fly towards the goal with a speed of at least 0.15 m/s. As each trial can last 180 seconds, the maximum required time to complete the benchmark is 45 hours. In practice, this will always be lower because the algorithm will not always reach the goal location due to collisions. All tasks in our example are set to take place on the *factory* map. An image of this testing area can be seen in Figure 10.

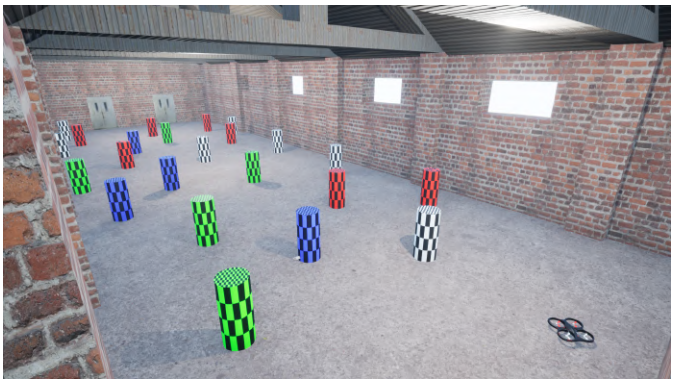


Fig. 10: A procedurally generated obstacle field using a 2.5m Poisson radius inside the factory environment with obstacles from the L0 set. This is one of the 50 unique environments that is generated during the benchmark. The drone that can be seen in the image has a diameter of 0.5m.

B. Algorithm

For this experiment, a monocular obstacle avoidance algorithm was implemented based on the principles of sparse optical flow. The algorithm is by no means novel or complex, but as the primary purpose of the algorithm is to showcase the benchmarking procedure, this does not pose a big issue.

Every iteration, the algorithm starts with the calculation of the reference yaw based on the current position of the drone and the goal location. This ensures that the drone is always looking directly at the goal. Throughout the mission, the algorithm keeps a constant forward reference velocity of 1 m/s. To avoid any obstacles the drone may encounter, the algorithm relies on the comparison of optical flow between the left and right halves of the image. It utilises FAST features and the Lucas-Kanade method to calculate the optical flow [20] [21]. Depending on the total sum of optical flow in each half, the decision is made to either set the lateral reference velocity positive or negative. Once set, the lateral reference is kept active for a brief period before reverting it to zero. Afterwards, a timeout is applied until the actual lateral velocity of the drone is close to zero. Although this hinders reactivity of the algorithm, it simplifies the procedure as the

lateral movement of the drone has a significant impact on the total sum of optical flow in each half. Finally, all control references are applied. The algorithm runs at a consistent update rate of 10Hz. A flowchart depicting the obstacle avoidance algorithm can be seen in Figure 11.

The performance of the algorithm heavily relies on the constants Tc, Tt, Kl, Kr, Kd, Kv and Kf. Picking values that ensure the best performance can be quite difficult as different environmental conditions might require a completely different set of constants. During development, these values were determined by trial-and-error in the factory environment with procedural generation seeds that were different from the test set. A final benchmarking could involve keeping a set of environments out of reach for the researchers, so that the final performance is not contaminated by such "training on the test set".

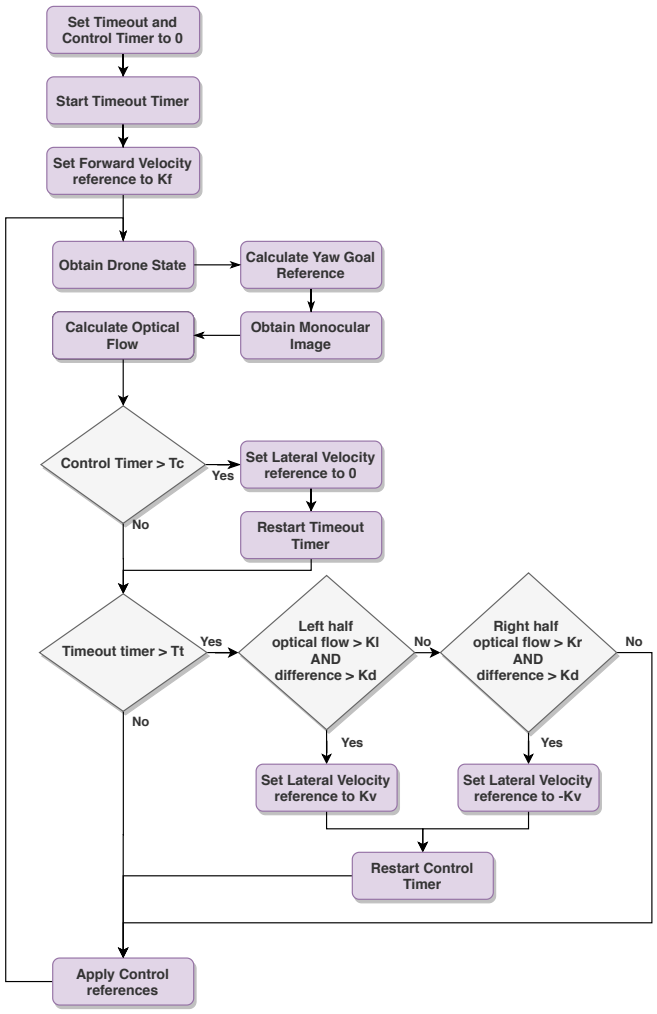


Fig. 11: A flowchart depicting the workings of the obstacle avoidance algorithm used in this paper.

C. Results & Discussion

For this experiment, the benchmark was performed on a machine with an *AMD Ryzen 7 2700X* CPU and a *Nvidia 1080Ti* GPU. The total time needed to complete the benchmark was 6 hours and 37 minutes. The results from the benchmark are split into two sections. First, the obtained performance metrics will be discussed and following that the environment metrics.

1) Performance Metrics: Table I reports the results of the performed benchmark. Each row shows the average results for a fixed *Poisson Disc Radius*. The columns show the *Collision Percentage*, *Mission Progress*, *Average Goal Velocity* and *Optimality Factor* for each contrast level. In cases where the algorithm was unable to complete multiple mission trials, the metrics for *Average Goal Velocity* and *Optimality Factor* lose their inherent meaning and are therefore changed to "N/A".

We compare the performance metrics against the *Relative Gap Size* for different contrast levels. These sub-figures can be seen in Figure 12. Each graph also contains a *Straight Flight* (SF) line that shows the results for an algorithm that flies straight to the goal without any avoidance manoeuvres.

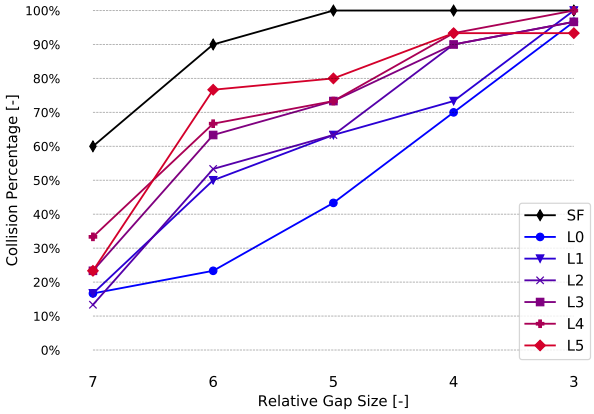
a) Collision Percentage: The *Collision Percentage* graph seen in Figure 12a shows there exists a general trend of the *Collision Percentage* increasing as the *Relative Gap Size* decreases. It can also be noticed that as the contrast on the obstacles decreases, the performance becomes worse. Looking at the *Straight Flight* line, it can be seen that for the tasks with the higher *Relative Gap Size* values, there exist a few generated obstacle fields where it is possible to reach the goal by only flying straight. Only at a *Relative Gap Size* of 5 do we see that there is always at least one obstacle with which the drone will collide when travelling on the line from the start to the goal location.

One important detail to note is that in some cases such as the L1 and L2 obstacle sets, the *Collision Percentage* at a *Relative Gap Size* of 5 is exactly the same. This can be attributed to the fact that the number of trials is set to 30, allowing the collision percentage to change only in steps of 3.33%. If the number of performed trials were to be increased, better statically reliable results would be obtained.

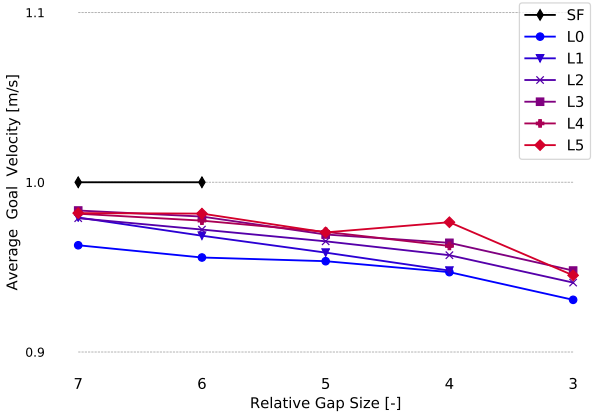
b) Average Goal Velocity: Looking at Figure 12b, it can be seen that the *Average Goal Velocity* stays relatively constant throughout all *Relative Gap Size* values and obstacle sets. This can be explained by the fact that the algorithm always controls the drone to travel forward with a fixed velocity of 1 m/s. The slight downwards trend can be attributed to the fact that more lateral movement is required to avoid a higher density of obstacles. This increases the total mission time and therefore results in a lower *Average Goal Velocity*. An interesting thing to note about the graph is that the *Average Goal Velocity* is higher for obstacles

TABLE I: Quantitative benchmarking results for the implemented obstacle avoidance algorithm subjected to the benchmark presented in subsection V-A.

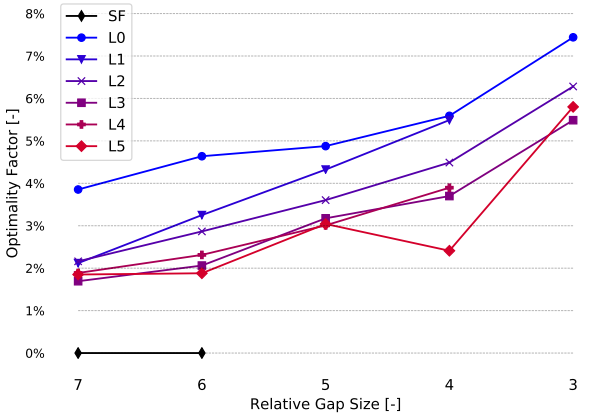
Procedural Generation System Radius [m]	Relative Gap Size	L0				L1				L2				L3				L4				L5			
		CP	MP	AGV	OF	CP	MP	AGV	OF	CP	MP	AGV	OF	CP	MP	AGV	OF	CP	MP	AGV	OF	CP	MP	AGV	OF
4	7	16.7%	87.7%	0.96	3.9%	16.7%	88.2%	0.98	2.1%	13.3%	91.2%	0.98	2.2%	23.3%	83.9%	0.98	1.7%	33.3%	76.5%	0.98	1.9%	23.3%	83.5%	0.98	1.9%
3.5	6	23.3%	85.2%	0.96	4.6%	50.0%	81.7%	0.97	3.3%	53.3%	71.4%	0.97	2.9%	63.3%	65.8%	0.98	2.1%	66.7%	65.5%	0.98	2.3%	76.7%	57.2%	0.98	1.9%
3	5	43.3%	78.4%	0.95	4.9%	63.3%	69.0%	0.96	4.3%	63.3%	59.6%	0.97	3.6%	73.3%	57.5%	0.97	3.2%	73.3%	50.5%	0.97	3.0%	80.0%	41.6%	0.97	3.0%
2.5	4	70.0%	63.6%	0.95	5.6%	73.3%	58.7%	0.95	5.5%	90.0%	41.8%	0.96	4.5%	90.0%	52.3%	0.96	3.7%	93.3%	48.6%	0.96	3.9%	93.3%	41.9%	0.98	2.4%
2	3	96.7%	28.9%	0.93	7.4%	100.0%	31.6%	N/A	N/A	96.7%	31.4%	0.94	6.3%	96.7%	26.3%	0.95	5.5%	100.0%	28.2%	N/A	N/A	93.3%	29.2%	0.95	5.8%



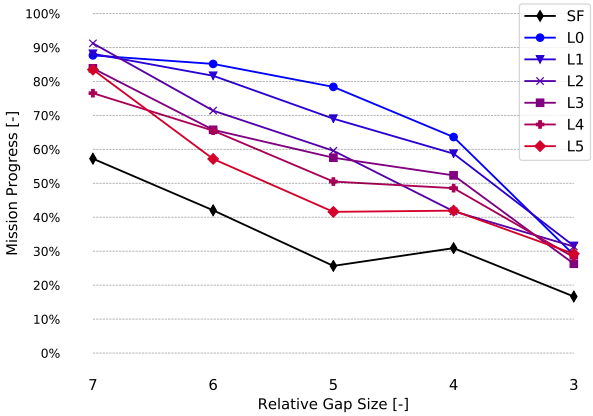
(a) *Collision Percentage* results for different *Relative Gap Sizes* and obstacle sets.



(b) *Average Goal Velocity* results for different *Relative Gap Sizes* and obstacle sets.



(c) *Optimality Factor* results for different *Relative Gap Sizes* and obstacle sets.

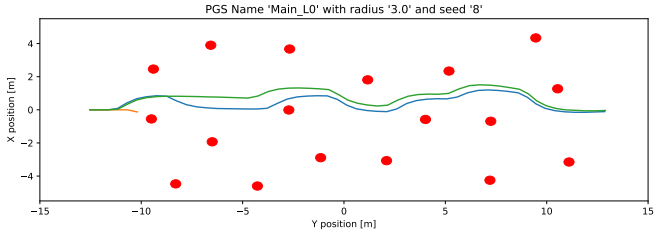


(d) *Mission Progress* results for different *Relative Gap Sizes* and obstacle sets.

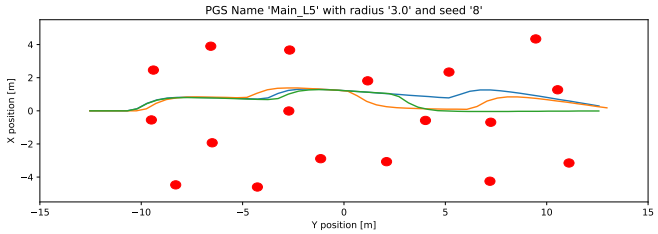
Fig. 12: Performance metric graphs generated from the benchmark results. The black *SF* line shows the results for an algorithm that flies straight to the goal without performing any avoidance manoeuvres.

with less contrast. Because the algorithm is highly sensitive to the perceived optical flow, the lack of features on the obstacles results in the algorithm executing fewer avoidance manoeuvres, thus reaching the goal faster.

c) Optimality Factor: Figure 12c shows how the *Optimality Factor* trends upwards with increasing *Relative Gap Size*. Similarly to the *Average Goal Velocity*, this can partly be attributed to the fact that a higher obstacle density requires more lateral movement adjustment to steer clear of the obstacles, therefore increasing the total travelled distance. Again, it can also be seen in this graph that the performance is better for the tasks where the obstacle contrast is low. Although this might seem counter-intuitive, it is again caused by the fact that less optical flow is measured and therefore, fewer avoidance manoeuvres are performed compared to tasks with high contrast obstacles. From the images in Figure 13, it can be seen that the algorithm makes many more avoidance manoeuvres in the environment with high contrast obstacles. This causes more lateral movement and thus results in a higher *Optimality Factor*.



(a) Three recorded trials taken during the benchmark with obstacle set L0. The high contrast on the obstacles results in the algorithm making some unnecessary avoidance manoeuvres (e.g. the blue trajectory), thereby increasing the *Optimality Factor*.



(b) Three recorded trials taken during the benchmark with obstacle set L5. Due to the lower contrast on the obstacles, the algorithm performs fewer avoidance manoeuvres.

Fig. 13: Comparison between benchmarking runs for different contrast levels

d) Mission Progress: Looking at Figure 12d, it can be seen that the *Mission Progress* decreases as the *Relative Gap Size* is decreased. When comparing the performance between different obstacle sets, it can be noticed that the algorithm is able to traverse further to the goal in environments with high levels of contrast than with low contrast levels. At a *Relative Gap Size* of 3, the results for all obstacle sets start to converge, indicating the point where the algorithm is unable to traverse the obstacle field irrespective of contrast levels.

e) Algorithm Processing Time: The final performance metric, *Algorithm Processing Time*, is not listed in Table I because the result always is a constant value of 100ms. This has to do with the fact that the algorithm runs at a fixed update rate of 10Hz. This performance metric is, therefore, better suited to algorithms that utilise a variable update rate.

f) On Metric Redundancy: Looking at the current results, it might seem as if some metrics can be considered redundant. For example, there exists a clear correlation between the *Collision Percentage* and the *Mission Progress* as seen in Figure 14. This might make it seem like one of these metrics is redundant. Although this is a valid concern, we do not believe this is the case. All metric results are highly dependent on the type of obstacle avoidance algorithm that is being evaluated. Therefore, the relations between metrics that exist in this example will not necessarily transfer to other algorithms. Depending on the behaviour of a specific algorithm, the graphs might look completely different.

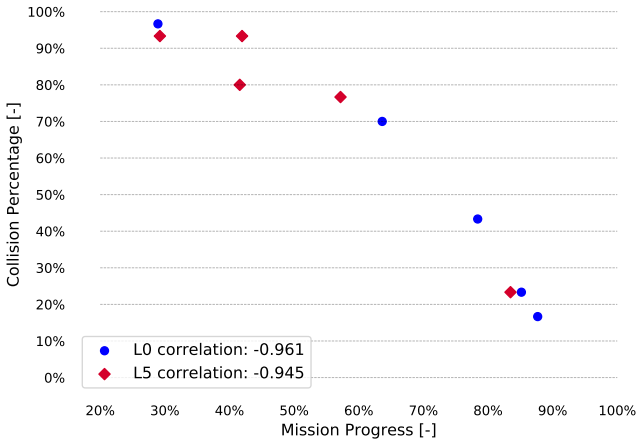


Fig. 14: Correlation between *Collision Percentage* and *Mission Progress* for the L0 and L5 obstacle sets. It can be seen that for both extremes, there is a highly negative correlation.

2) Environment Metrics: One of the benefits of measuring environment metrics is that the performance of the algorithm can be generalised better. So far, we have only compared the performance metric results against the *Relative Gap Size* metric. By establishing the relationship between *Relative Gap Size* and *Traversability*, it should in theory be possible to generate environments of the same difficulty level simply by obtaining a maps *Traversability*.

Plotting the relationship between *Traversability* and *Relative Gap Size* in Figure 15, it can be seen that the *Traversability* values do not have a high statistical variance. This helps to confirm the workings of the procedural generation system, showing that the generated environments are of the same difficulty for a given *Poisson Radius*. The figure shows that the relationship between *Traversability* and *Relative Gap Size* is not linear. One of the reasons for this is that, as more obstacles are generated, more grid locations will fall within these obstacles. As the N headings are evenly split in a 2D unit circle, the average free-path distance s is significantly lower than when a sample is taken in a non-obstacle location.

VI. Recommendations

Currently, AvoidBench is capable of automatically evaluating the performance of obstacle avoidance algorithms. However, as with any initial release of a software package, only the absolutely necessary required features are implemented. This means that there is still plenty of room for AvoidBench to grow, both as a benchmarking suite and a development suite.

Here we would like to make some recommendations on extra features, metrics and content that could be added in

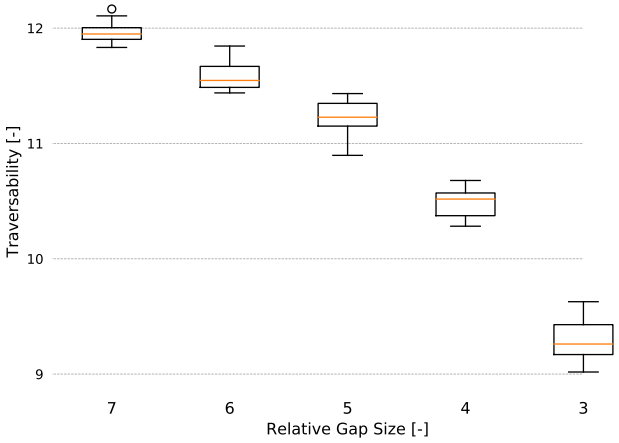


Fig. 15: The relationship between *Traversability* and *Relative Gap Size*. For each *Relative Gap Size* value, *Traversability* is measured for all ten unique environments.

the future. First of all, over time, we expect that more maps will be added, allowing for a greater variety of benchmarks to take place. New mission types can also be added such as a *multiple-waypoint* mission. This could make it possible to design a racing track benchmark where the purpose is to pass through multiple gates while avoiding other types of obstacles. As a last example, more environment metrics could be added to generalise the results of the performance evaluation better. Current environment metrics only take into account the topography of a map. An environment metric such as the *average perceived level of texture* would make it possible to also take into account the visual properties of a map.

Perhaps one of the most important recommendations is to set up a benchmarking server and a tracking website like KITTI [1]. This would allow users to submit their algorithms to the AvoidBench website so that they can automatically be benchmarked. The results could then be posted to a public leaderboard with references to the algorithm's source code. This would make the entire field of obstacle avoidance even more accessible.

VII. Conclusion

In this paper we presented AvoidBench, a benchmarking suite capable of evaluating the performance of visual obstacle avoidance algorithms for multi-rotors. We discussed the general design decisions, the software architecture and explained the AvoidBench benchmarking pipeline. Using the AvoidBench benchmarking suite, a benchmark was then created to test obstacle avoidance algorithms on their ability to traverse an obstacle field with varying levels of contrast. A generic obstacle avoidance algorithm was implemented and exposed to more than 50 unique obstacle fields. During this benchmark, a variety of different performance and environment metrics were measured to evaluate the performance of this algorithm. We hope that with the introduction of AvoidBench, it will gradually become known as one of the main benchmarking suites for obstacle avoidance algorithms.

References

[1] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving?the kitti vision benchmark suite," 2014. [Online]. Available: http://www.cvlibs.net/publications/Geiger2012CVPR.pdf

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[3] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, "Benchmarking reinforcement learning algorithms on real-world robots," *CoRR*, vol. abs/1809.07731, 2018. [Online]. Available: http://arxiv.org/abs/1809.07731

[4] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, "Rlbench: The robot learning benchmark & learning environment," *CoRR*, vol. abs/1909.12271, 2019. [Online]. Available: http://arxiv.org/abs/1909.12271

[5] G. Fontana, M. Matteucci, F. Amigoni, V. Schiaffonati, A. Bonarini, and P. U. Lima, "Rockin benchmarking and scoring system," in *RoCKIn-Benchmarking Through Robot Competitions*. IntechOpen, 2017.

[6] C. Nous, R. Meertens, C. de Wagter, and G. de Croon, "Performance evaluation in obstacle avoidance," *IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7759532

[7] T. Tezenas Du Montcel, A. Negre, J.-E. Gomez-Balderas, and N. Marchand, "Boarr : A benchmark for quadrotor obstacle avoidance based on ros and rotors." [Online]. Available: https://hal.archives-ouvertes.fr/hal-02142571

[8] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Cham: Springer International Publishing, 2016, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26054-9_23

[9] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[10] Epic Games, "Unreal engine." [Online]. Available: https://www.unrealengine.com

[11] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. von Stryk, "Comprehensive simulation of quadrotor uavs using ros and gazebo," in *3rd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, 2012, p. to appear.

[12] W. Guerra, E. Tal, V. Murali, G. Ryou, and S. Karaman, "Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality," *CoRR*, vol. abs/1905.11377, 2019. [Online]. Available: http://arxiv.org/abs/1905.11377

[13] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," 2020.

[14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[15] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, M. Almeida, K. Mansoorshahi, A. Faust, and V. Reddi, "Mavbench: Micro aerial vehicle benchmarking," *CoRR*, vol. abs/1905.06388, 2019. [Online]. Available: http://arxiv.org/abs/1905.06388

[16] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[17] S. Tijmons, G. C. H. E. de Croon, B. D. W. Remes, C. De Wagter, and M. Mulder, "Obstacle avoidance strategy using onboard stereo vision on a flapping wing mav," *IEEE Transactions on Robotics*, vol. 33, no. 4, p. 858–874, Aug 2017. [Online]. Available: http://dx.doi.org/10.1109/TRO.2017.2683530

[18] L. Matthies, R. Brockers, Y. Kuwata, and S. Weiss, "Stereo vision-based obstacle avoidance for micro air vehicles using disparity space," in *2014 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2014, pp. 3242–3249.

[19] R. Bridson, "Fast poisson disk sampling in arbitrary dimensions," in *ACM SIGGRAPH 2007 Sketches*, ser. SIGGRAPH '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 22–es. [Online]. Available: https://doi.org/10.1145/1278780.1278807

[20] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443.

[21] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'81. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1981, p. 674–679.

II

Literature Study

Literature Study

Obstacle avoidance benchmarking suite

by

Rano Veder

Student number: 4460464

TU Delft

Contents

1 Introduction

The interest in obstacle avoidance algorithms, whether for cars, robots, or drones, has always been an important subject in robotics research. If autonomous agents are to be working among people and processes, it is vital that they can avoid obstacles, to ensure continuity of operation, and prevent costly damages. Currently, obstacle avoidance in micro air vehicles (MAVs) and drones has become quite significant. In this age, where it is not uncommon to hear of applications for drones such as package delivery and human transportation, the importance of obstacle avoidance becomes even more pronounced.

Currently, there are many possible options when looking for different obstacle avoidance algorithms. One issue, however, is the validation and comparison of different algorithms. Most algorithms are developed by researchers at different universities and are tested in special drone-safe rooms. These tests, however, often do not represent real-life scenarios. Each university has a different idea of how these algorithms should be tested. Often, these environments contain basic shapes which serve as obstacles with highly unnatural colours and texture patterns. The environments are meant to test the algorithm under extreme unnatural conditions. This makes it difficult to predict in which types of real-life environments the algorithms would perform well, and in which they would fail.

The research to be presented in the thesis is to solve this problem. By creating a benchmark suite in simulation, it will be possible to measure the performance of different obstacle avoidance algorithms in a real-life setting. These measurements then allow us to compare the algorithms and possibly predict the performance given a description of the environment. Furthermore, the benchmark will allow researchers to implement their own algorithms, and measure their performance against state-of-the-art obstacle avoidance algorithms.

In this document, a literature review is performed in order to come up with a design for an obstacle avoidance benchmark suite. Chapter 2 shall present the main research question and lay out the objectives of this research. In Chapter 3, an overview shall be given on obstacle avoidance methods in general. Chapter 4 looks at drone simulators and presents a trade-off between different candidates in order to decide the final simulator that will be used in the benchmark suite. Chapter 5 gives an overview of different performance measures that can be used in order to evaluate the performance of an obstacle avoidance algorithm. Finally, Chapter 6 discusses the concept of benchmarking in general.

2

Research Question and Objectives

The obstacle avoidance evaluation framework by Nous, Meertens, de Wagter, and de Croon (2016) introduced a proper method for evaluating the performance of planar, vision-based obstacle avoidance algorithms. Although this framework was proposed three years ago with promising results, it has currently not yet seen widespread usage. One can, of course, only speculate on the reasons for this, but one apparent reason is the fact that the barrier to entry is too high. Utilising the methods presented in the framework requires a lot of manual labour. One has to calculate the environmental metrics for multiple different environments, run multiple independent tests, and process all the data. All this extra work can play a factor in the current adoption rate.

To lower the barrier to entry for other researchers, the need arises to come up with a solution to this problem. Although it is a challenging task to simplify the real-life evaluation of the framework, it can, however, be largely automated by performing all the tasks in simulation. The researcher implements the algorithm and chooses a MAV model, and the benchmark suite takes care of running all the tests and generating output data. This brings us to the main objective of this thesis:

'The objective is to create a benchmark suite to validate and compare obstacle avoidance algorithms in a controlled environment by designing and implementing a method in simulation to "objectively" measure the performance of the aforementioned algorithms using a set of metrics.'

This objective consists of two parts. First, the benchmark suite has to be designed. This consists of determining what performance and environment metrics ought to be used to measure the performance of obstacle avoidance algorithms. Next, the missions and environments need to be determined. Finally, all the relevant software which is necessary to build the benchmark suite needs to be selected. The last part of this objective is programming the benchmark suite, and designing the environments. This all leads to the following set of (sub-)research questions:

- *What benchmark metrics should be used to measure the performance of obstacle avoidance algorithms?*
- *What metrics should be used to describe a complex environment?*
- *What are the expected relationships between the performance and environment metrics?*
- *What types of environments should be used in the benchmark?*
- *What missions should be evaluated in the benchmark simulation?*
- *What should the software architecture of the benchmark suite look like?*

To answer these questions, it is possible to split the research objective into multiple sub-goals. These are all quickly listed below and further explained in the following paragraphs.

1. Design of the benchmark suite
   (a) Design metrics
      i. Generate a list of potential metrics to measure algorithm performance.
      ii. Generate a list of potential metrics to describe the environment.
      iii. Make a final selection.
      iv. Create hypotheses between algorithm performance and environment metrics
   (b) Design tasks that must be tested
      i. Make a list of potential missions/tasks that could be used
      ii. Select final set of tasks
   (c) Decide types of environments
      i. Inspect what types of environments are commonly seen in real-life applications of obstacle avoidance algorithms
      ii. Specify visual description of what each environment should look like
   (d) Create software architecture
      i. Decide on how algorithms interface with the benchmark suite
      ii. Decide on how the simulator and benchmark suite communicate
      iii. Design the internal structure of the benchmark suite
2. Implement Benchmark suite
   (a) Program the benchmark suite and simulator
      i. Unknown
   (b) Design environments
      i. Get familiar with 3D environment generation inside Unreal
      ii. Create 3D environments
   (c) Package software as bundle
      i. Upload software to Github
      ii. Write wiki/readme
3. Analyze results
   (a) Benchmark set of existing avoidance algorithms
      i. Make a list of algorithms which could be implemented
      ii. Make a selection of 2-3 algorithms
      iii. Convert algorithms such that they can be evaluated by the benchmark
   (b) Analyze if the results make sense (performance metrics environment metrics)
      i. Run the benchmark suite for different algorithms
      ii. Use data to see if hypotheses from 1.a.iv are correct

Based on the research questions and their respective objectives, it is possible to roughly get an idea of what subjects are important in order to finish this research. The four main subjects that will be discussed in this literature study are listed below:

- Obstacle avoidance algorithms
- Simulators
- Performance measures
- Benchmarks

3

Obstacle Avoidance Overview

Obstacle avoidance can be described as the process of detecting and preventing the collision between the agent and potential obstacles. The field of obstacle avoidance is quite vast and certainly not limited to Micro-air-vehicles (MAVs). In the automotive and aviation industry, it is more commonly known under the name collision avoidance systems (CAS). Although the names may differ, there exists a large amount of overlap between these two methods. Where they differ is that in the MAV sector, obstacle avoidance mostly refers to the avoidance of static objects, such as walls, pillars, trees, windows, etc. With CAS, on the other hand, this is generally not the case. The focus of CAS is more on the avoidance of other agents which exist in the same operating space.

The specific inner workings of each obstacle avoidance algorithm may be different, but they all seem to follow the same general approach. Albaker and Rahim (2009) describe how a fully automated collision avoidance system must address four problems. Namely, sensing the environment, performing conflict detection, calculating escape trajectories and manoeuvre realisation. These problems can be seen in Figure 3.1. Although their survey is focused on CAS, many of the same problems still apply to obstacle avoidance in a MAV context.

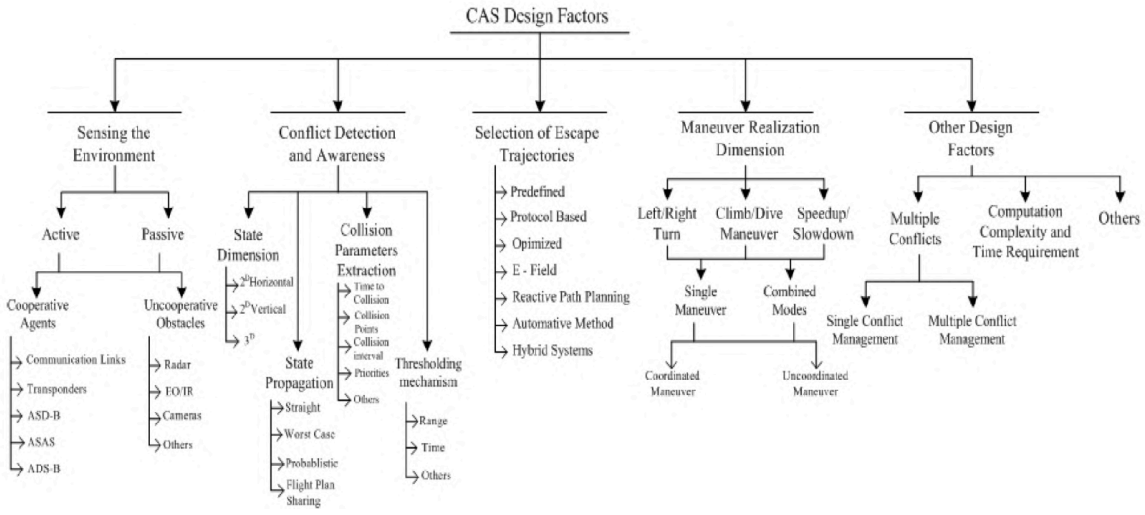


Figure 3.1: The main problems related to a CAS (Albaker & Rahim, 2009)

The first problem, sensing, refers to the ability of the algorithm to take in information on the environment around it. This obtained data is used in the second step to detect obstacles and potential conflicts that may arise, given the current state of the vehicle. If a conflict occurs, an escape trajectory is calculated in step 3, and send to the control system of the vehicle in step 4.

In the following sections, we will explore the current state of these four problems, look at some of the principles used to solve them, and explore some solutions found in literature.

3.1. Sense & Detect

This section covers the first two problems in the obstacle avoidance pipeline, namely sensing the environment and the detection of obstacles. First, a brief overview is given on sensors in general, and their applications for MAVs. Afterwards, an overview is given on obstacle detection techniques commonly seen for mono and stereo vision cameras.

The observation of the environment is performed by devices called sensors. Sensors provide the detection algorithm with raw data, which must be processed in order to find if there are any obstacles. There are two types of sensors which can be used, that is active and passive sensors. Active sensors use their own energy to transmit signals, which are then measured after being reflected, refracted or scattered. Passive sensors, on the other hand, transmit no signals. Instead, they measure signals which exist due to natural transmission. Commonly, for MAV applications, not all available sensors can easily be used. Constraints such as power usage and dimensions may prevent a sensor from being carried on board. Fortunately, many active sensors currently have low-power variant created specifically designed for robotics applications. A list of sensors which are commonly used can be seen in Table 3.1.

Table 3.1: List of sensors commonly used for obstacle avoidance applications.

Sensor	**Type**
Monocular Cameras	Passive
Stereo Cameras	Passive
Event Cameras	Passive
Sonar	Active
Lidar	Active
Radar	Active

Although all these sensors have their own advantages and use cases, in this literature review and thesis, we limit the sensor types to monocular and stereo-vision cameras. The main reason for this is as follows. Due to the low cost, low power consumption and high information density of vision-based sensors, they are perfect candidates for MAVs to sense their environment. Focusing on these types of sensors ensures that the benchmark applies to as many algorithms as possible. Furthermore, the sensor performance metrics developed by Nous et al. (2016) form the basis for the to be developed benchmark. As these were only defined for visible spectrum vision-based sensors, it goes without saying that more metrics need to be developed if extra sensors are added. This, unfortunately, is unfeasible given the available time frame for this thesis.

It should, however, still be important to acknowledge the limitations of vision-based methods. Cameras are passive sensors, requiring a certain amount of light in the scene in order to produce useful measurements. Flying in areas which are poorly lit can thus cause the obstacle avoidance algorithm to produce more errors. Fortunately, if necessary, an external light source can be added to the MAV, allowing the camera to also work better in dim environments.

In the following sections, a brief overview will be given of some of the methods used for the detection of obstacles with mono and stereo vision camera sensors.

3.1.1. Monocular Vision

Monocular vision obstacle avoidance utilises a single camera combined with image processing to detect obstacles in a scene. The use of a single camera in obstacle avoidance inherently has some shortcomings. One issue is that, without any knowledge of the scene itself, the camera is unable to get the absolute scale of the scene. There will always exist an uncertainty between the dimensions of an object, and how far away it is from the camera. Generally, two different principles are used to find potential obstacles. These are based on motion parallax, and monocular cues (Mori & Scherer, 2013).

Motion parallax Motion parallax utilises optical flow, which relies on the assumption that the intensity of objects in a scene remains unchanged. A point moving through the image frame over time will be assumed to keep the same brightness level. Using this assumption, the *brightness constancy constraint* can be formulated stating the dependency between these points during multiple frames. (Baker et al., 2007)

$$I(x(t), y(t), t) = constant \tag{3.1}$$

By assuming small changes between frames and a small time-step (which is often the case for video), Equation 3.1 can be formulated as seen in Equation 3.2. Here, u and v describe the velocity of a given point, and δt described the time increment.

$$I(x + u\delta t, y + v\delta t, t + \delta t) = I(x, y, t) \tag{3.2}$$

Often, Equation 3.2 is linearised to bring it in a more usable form as seen in Equation 3.3. The value of Δt can be calculated based on the frame rate at which the camera was recording. The other two values u and v, however, are unknown and must be calculated. As there is only a single equation, this system is under-determined and cannot be solved. Often, an extra assumption is added, which states that the optical flow in a small region around the target pixel is equal. This assumption forms the basis for the Lucas-Kanade method, which uses a window of a fixed size to calculate the optical flow for an entire region. This almost always leads to an over-determined system, which is solved using a least-squares method. (Lucas & Kanade, 1981).

$$\frac{\delta I}{\delta x}u\Delta t + \frac{\delta I}{\delta y}v\Delta t + \frac{\delta I}{\delta t}\Delta t = 0 \tag{3.3}$$

The Lucas-Kanade method is not the only available method to solve the optical flow equation. The Horn-Schunck method is a global algorithm which ensures smoothness among the entire flow-field. Because a global optimisation is necessary in order to solve the entire flow field, this method has worse performance than methods such as Lucas-Kanade. Depending on the amount of available computing power and image size, it might thus not be possible to run this on some MAVs. (Horn & Schunck, 1981)

Once the flow field for the entire frame is calculated, it can be used in a variety of different ways for the detection of obstacles. Sebesta and Baillieul (2012) utilise the flow field to estimate the time-to-contact (TTC) for obstacles in the current frame. In the algorithm proposed by Marlow and Langelaan (2011), the optical flow in the image, together with the velocity of the vehicle is used to calculate the distance to the obstacles. These distances are then used to create a local occupancy map, thus providing the algorithm with a map on where the obstacles are located.

Issues with optical flow for obstacle avoidance mainly occur when the camera is approaching the obstacle frontally. In this case, the optical flow is close to zero, which makes it impossible to detect obstacles straight ahead. By zig-zagging the vehicle perpendicular to the velocity vector, a parallax effect can be created, which makes it again possible to detect frontal obstacles. This is, however, not always a possible option for MAVs due to higher energy consumption. It is also difficult for obstacle avoidance algorithms to fly in areas with homogeneous textures. In these cases, it becomes difficult for the calculate the flow field, as there are not enough features to track. (Mori & Scherer, 2013)

Apart from issues that are fundamentally hard to solve, optical flow can also cause issues for platforms with low computation power. Nowadays, this is less of an issue than before, with computers being released that are specifically designed for robotics/computer vision applications. NVIDIA[1] especially is releasing embedded computers with internal graphic processing units (GPU). Optical flow methods are highly parallelisable and thus benefit greatly from these hundreds of cores. Unfortunately, these computers are meant for mid to high power robots. Small MAVs may thus still have issues when trying to run methods such as optical flow.

[1] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

Monocular cues Obstacle detection based on monocular cues relies on using predefined rules about the ”look” of the environment. Based on known visual features, the algorithm might be able to detect obstacles from a still frame. There are many different subcategories which fall under *monocular cues*. Mori and Scherer (2013) describes a list of five total categories. That is perspective, relative size, known object size, texture gradient, Occlusion and depth from focus.

Perspective methods rely on the identification of vanishing lines to detect obstacles such as walls and ceilings. Bills, Chen, and Saxena (2011) present an algorithm which allows a MAV to traverse through unknown corridors, stairs and corners based on perspective cues. Although perspective methods generally work well for indoor environments, they are less suited for outdoor environment due to the lack of available perspective cues.

The next category, relative size, can detect incoming obstacles by tracking the area of an object throughout multiple frames. As the relative size for an object increases, it means the camera is getting closer to the obstacle. An algorithm which utilises the concept of relative size is presented by de Croon, de Weerdt, De Wagter, Remes, and Ruijsink (2012). By calculating the Shannon entropy, it becomes possible to quantify the *appearance cue* which is created by two effects. When the camera gets closer to an obstacle, the relative size of the frontal obstacle increases while other objects go out of view, and the texture detail increases. The authors noticed that entropy, in most cases, decreases when approaching an obstacle. There were, however, cases in which it went up. To deal with this inconsistency, the algorithm was extended with optical flow. A different method was proposed by Nègre, Braillon, Crowley, and Laugier (2006). By calculating the relative size of the object and rate at which it changes object changes, it becomes possible to calculate the time-to-contact. Often, optical flow algorithms are extended with appearance-based/monocular methods. This provides a solution to the issue of zero optical flow for frontal obstacles.

The next category, known object size, is based on the principle of matching features in the image frame to a set of known features stored in an internal database. These methods rely on predefined knowledge of the scene in which operation will take place and often have a list of object sizes for most, if not all, objects in the scene. By matching objects in the frame which have a known object size, the scale ambiguity, which is present by default in monocular vision is solved. This allows the algorithm to obtain the distance from the object to the camera.

3.1.2. Stereo Vision

Stereo vision methods rely on the known distance between two cameras to find the disparity map for a given set of stereo images. This disparity map contains a depth value from each pixel in the disparity frame. The depth to an object from the camera is related by the triangulation equation. Here, f is the focal length of the camera (Assuming both cameras are identical), B is the horizontal distance between the cameras, and x_l and x_r are the pixel locations of a corresponding point in both frames. The problem that remains is finding the correspondence between points in both images. In order words, finding the (x_r, y_r) for a given (x_l, y_l). (Hartley & Zisserman, 2003)

$$Z = \frac{f \cdot B}{x_l - x_r} \tag{3.4}$$

The process of finding the correspondence between points is known as matching. Fortunately, this process does not require an exhaustive search of all features to find a match. One only needs to search for features along the epipolar line. Given that stereo-cameras are often horizontally aligned, this means that features are found along the same horizontal line. It’s not always possible to find a correspondence, as some features may be occluded, that is, present in one image but not in the other. Homogeneous textures can also pose issues, as there are no apparent features which can be matched.

Matching algorithms can be classified into two groups, namely, global and local methods. Global methods obtain the disparity map by finding a minimum solution to the disparity problem, which is dependent on all pixels in the frame. Local methods, on the other hand, use a windowing technique to find the disparity map. As local methods don’t require a global optimisation over all pixels, they are less

complex, and therefore also have better performance. An advantage of global methods, however, is that they can deal better with depth estimation in regions with Occlusion and homogeneous textures. Furthermore, since a smoothness parameter can be specified for global methods, a cleaner disparity map can be obtained. Unfortunately, these benefits come at the cost of performance. These global methods are often not used on MAVs, as they are not able to run in real-time. (Hamzah & Ibrahim, 2016)

A popular local algorithm is known as block-matching. In this method, given a window of pixels in the left image, the algorithm will walk along the epipolar line in the right image to find the best match. The score for each match can be calculated in a variety of different ways. Most commonly, however, a sum of absolute differences is used.

Once the matching step is completed, it is relatively straightforward to obtain the distance to each object in the frame. These distances can then be used to detect potential obstacles. Barry, Florence, and Tedrake (2018) present a modified block-matching algorithm used to avoid obstacles during fixed-wing flight at speeds around 14 m/s. This method differs from conventional block-matching, as it only matches blocks along the epipolar line for a specified distance. This reduces the run-time of the algorithm, as only a single comparison is needed for each block. Using the obtained disparity map, and combining it with the estimated state of the drone, a point-cloud is created. This cloud is then used in combination with a trajectory planner to avoid the obstacles. Liu, Watterson, Tang, and Kumar (2016) present a method in which they use the depth of a stereo vision camera to build up a volumetric occupancy grid. A free path is then generated using a local planner.

One issue with stereo methods is that the resolution of the depth measurement decreases as the distance increases. Depth measurements of objects far away from the camera will, therefore, be less accurate. This should be considered when building up a map of the environment by not putting a lot of confidence in large magnitude depth values.

3.2. Escape & Realisation

In a traditional sense, once a potential obstacle is detected, the obstacle avoidance algorithm tests if this new obstacle gives rise to a collision conflict. This is done by projecting the current state of the MAV a specific time delta into the future. If an obstacle intersects such a projection, a collision conflict is generated. The obstacle avoidance algorithm will try to find an escape trajectory, with the goal of resolving the conflict. Finally, when the trajectory is determined, control inputs are generated to move the vehicle off the collision path, onto the desired trajectory.

This traditional type of obstacle avoidance is more applicable to aircraft in large-scale air traffic, where the purpose of obstacle avoidance is mostly concerned with other agents instead of static obstacles. In the context of drones and MAVs operating in human-created surroundings, the detection and escape phase is often more simplified. Instead of complex state projection, the algorithms often utilise a simple rule-based approach in which they change heading when the measured distance to an obstacle becomes too small, or make use of local planner to avoid the obstacles based on a sensor-created map.

For completeness, the formal approach shall briefly be discussed. In the following section, we shall first discuss the methods for conflict detection. Afterwards, methods related to escape trajectory generation will be discussed, and finally, the last section will briefly touch on manoeuvre realisation.

3.2.1. Conflict Detection

Conflict detection in a formal sense is the process of determining whether the current vehicle is on a collision course with the detected obstacles. In situations where this is not the case, the vehicle can keep on travelling without making any adjustments. If, however, a conflict is detected, it must be resolved in order to not collide with the obstacle. The process of determining whether the vehicle is on a collision course is performed by state extrapolation.

As Albaker and Rahim (2009) discuss in their survey, there are four main methods of projecting the state of the UAV into the future. These are *straight projection*, *worst case projection*, *probabilistic*

projection and *flight plan sharing*. In the case of single-agent environments, which will be a property of the benchmark in this thesis, only the first three methods are relevant. In the following few paragraphs, each of these methods shall be briefly discussed.

Straight path projection Straight path projection can be categorised as the simplest method of the three. As the name implies, it projects the vehicle along its current state vector, assuming no changes in control inputs or uncertainties that may occur due to external conditions. This method has good performance, as only a single trajectory must be calculated. However, because the projection method does not take into account outside uncertainties, it means that the validity of the projection decreases as the lookahead time increases.

Worst case projection Worst case projection, in a sense, is the complete opposite of *straight path projection*. Instead of only evaluating one possibility, this method tries to evaluate *all* possible projections. By having the vehicle perform all possible manoeuvres in the lookahead time, a set of trajectories can be generated. If any of these intersect with the obstacle, a conflict is generated. This method will generally be able to more robustly detect real conflicts, at the cost of more performance and more false positives.

Probablistic projection Probabilistic projection generates, like worst-case projection, a set of possible trajectories. The main difference, however, is that each of these trajectories is weighted by a probability. By having a probability assigned to each trajectory, a conflict will only have to be generated when the overlapping trajectory has value above a certain threshold. Based on the risk strategy, this method can reduce the number of false positives, at the cost of even more processing than *worst-case projection*.

As was already briefly mentioned, the previous methods of conflict detection are not often seen in the drone and MAV sector. Instead of state extrapolation, the algorithms rely on raw depth, or time to contact measurements to detect potential conflicts passively. That is, there are no actual conflict events generated. The details of such algorithms work will be presented in the next section.

3.2.2. Escape Trajectory Generation

When an obstacle is detected, and a conflict generated, the obstacle avoidance algorithm is responsible for generating an escape trajectory which will take the vehicle off the collision path and onto the desired trajectory. There are many different methods which can be used to create such trajectories. Some do not create a trajectory at all, and simply change the reference heading of the drone. Others may utilise the obstacle detection information to build a custom map of the environment, either local or global and use a planner to create a new trajectory for the drone.

The first type of trajectory generation method can be classified as rule-based methods. These often perform very well and don't require a lot of computation power, as they are often evaluating a set of if-else statements. For this reason, these methods are often applied to small and low-power drones. One such method is incorporated in the work presented by Wagter, Tijmons, Remes, and de Croon (2014). First, a disparity map is generated from a set of stereo-camera images. Next, the number of pixels is counted in both the left and right half of the image where the disparity is higher than 5 m. If the total number of such pixels is smaller than a certain threshold, the drone will keep flying straight. Else, the drone will change its reference heading to the half that has fewer of such pixels.

Rule-based methods by themselves are often only able to perform pure avoidance of obstacles. That is, they are only able to generate commands such as turn left, turn right or continue. Although this is not specifically an issue, it will require some sort of communication between the planning phase, which is responsible for the execution of the mission (if any exist), and the rule-based system. Between these two systems, the rule-based algorithm will need priority over the planner, given that the planner by itself is not aware of the existence of obstacles.

There also exist obstacle avoidance methods which include a planner. In these systems, the location of the obstacles is directly fed to the planner, which in turn plans its route with the obstacles in

mind as such to avoid them. It is possible to separate these algorithms into two types: Global and local planners. Global planners create a trajectory directly from the current location of the drone to the goal. If along the way obstacles are detected, these are added to the environment map, and the trajectory is regenerated.

Another type of obstacle avoidance algorithm relies on the use of generated force and potential fields. In these methods, the agent can be seen as a negatively charged particle which is attracted and repelled by different forces in the environment. Often, the goal objective is given a positive potential field, attracting the agent, while the obstacles are given negative potential fields, repelling the agent. The agent then follows a path in which the least amount of resistance is encountered. This still requires the agent to generate a local map with all the detected obstacles, as this is often not known beforehand.

3.2.3. Manoeuvre realisation

Manoeuvre realisation is the last step in the execution chain of an obstacle avoidance algorithm. In the case of MAVs, there is often already a flight controller onboard which takes care of stabilisation and control. These flight controllers can have different working modes such as attitude control, rate control, position control or trajectory control. Depending on the type of algorithm and controller, some conversion may be needed in order to properly pass the manoeuvre command to the flight controller.

In some cases, this conversion can be quite difficult as one may need to convert from a local coordinate frame used by the algorithm to a global coordinate system used by the flight controller. If no global coordinate system is available (small MAV with no GPS), a conversion will need to be performed to generate a set of relative movement commands which can be sent to the flight controller.

4

Simulators

A simulation can be defined as ”The imitation of the operation of a real-world process or system over time” (Banks, 2010). As the word ”imitation” describes, a simulation is almost always an approximation of real-life. The fidelity of which mostly depends on how well this reality is modelled. Due to the complex nature of some problems, a perfect model might not always be feasible to obtain. For this reason, a set of assumptions can be introduced to lower the complexity, thus making the world easier to describe.

A model can be seen as a description of the behaviour of a process. Once a model is obtained, a simulation can be performed using a *simulator*. This could, for example, be a physical setup, or a digital piece of software. What is used depends entirely on the problem itself.

Although a simulation is an approximation of the process in real life, given good knowledge of the utilised assumptions, one may still obtain valuable information which can be applied to the real world. It is essential that the users of a simulator know what the assumptions are such that they can also determine the limitations of the simulation. When an assumption is violated, the results may not depict an accurate representation of the real-world anymore.

4.1. Simulation in the context of MAVs

In the context of MAVs, simulations are used for a variety of different applications. Often, they are used for the preliminary development of control systems, as this can be a costly procedure if multiple drone crashes occur. Furthermore, simulations are also used in cases where a lot of different agents are needed, such as in swarming.

Most MAV simulations contain a model of the dynamic behaviour of the drone being investigated. Most dynamic models assume the drone to be a point-mass with inertia and actuators (motors) located at a certain distance from the centre. The fidelity of such models again depends on the application. Simulations focused on assessing the performance of a MAV in high gust environments, will require additional work to ensure the behaviour of the drone in response to high-velocity wind gusts is properly modelled. In contrast, a MAV which only travels at low speeds inside a building might completely discard the effect of gusts.

If the task requires the usage of sensors, these also need to be simulated. For most sensors, this can be as rudimentary as adding white noise to the true value. For sensors such as cameras, however, an entire 3D world will have to be created and simulated. The visual fidelity of these 3D worlds can differ quite a lot, depending on how it is implemented. Depending on the importance of the visual fidelity of the environment, a lot of extra work might be necessary to reach a high enough level.

4.2. Open Source or Build

Given that the benchmark suite will be implemented in simulation, an important question arises. What simulator should be used? There are two options which are available. Either a simulator can be built

from scratch, or an existing open-source solution can be used and adapted. Each option has its advantages and disadvantages.

A self-made simulator allows the user to design their environment with certain assumptions and constraints of the main problem in mind. Depending on the requirements, a self-made simulator might even be a necessity if no other options are available. A few issues with this approach, however, is that creating a physically correct simulator is a time expensive undertaking. It requires general knowledge of simulator design, and proper use of verification and validation methods. Furthermore, numerous weeks will need to be spent creating the simulator, which consists mainly of programming and debugging. All this might make it unfeasible to design and implement a simulator.

Existing solutions, on the other hand, can often be installed and used within days. It then, however, becomes the responsibility of the user to make sure that the underlying assumptions on which the simulator was built, align which those of the main problem. Depending on the software, it might be possible to adapt the source code in order to add or remove assumptions. However, most of the time, this is only possible when choosing open source projects. These are projects created by a community of developers and maintainers, with a license that allows anyone to inspect, adapt and distribute the source code of the project. Although these projects are available for free, one must remain cautious about blindly using open-source software. Anyone can create and distribute open-source software, regardless of the quality of the underlying code. Furthermore, not all open-source projects are actively being maintained by a community of developers. When choosing to utilise an open-source project which is currently unmaintained, the user itself becomes responsible for fixing the bugs and issues. As this may require a fundamental understanding of the underlying source code, a lot of time will have to be spent familiarising with the codebase.

For this thesis, the choice will be made to choose an already existing simulator solution. Building a custom simulator, which is also able to render a 3D world would take too much design and development time. Furthermore, the focus of this thesis is on the creation of a benchmark suite. Creating a custom simulator would only distract from this topic.

4.3. Requirements

For the selection of a suitable simulator. A trade-off will be performed between a set of open-source simulators. This trade-off will depend on a variety of different criteria, represented by both qualitative and quantitative measures. This is by no means supposed to be an objective trade-off. Performing an objective trade-off with both qualitative and quantitative measures can be quite difficult. Therefore, the relative importance of the criteria will be used to make the final trade-off. The criteria listed below act as a guide to help the reader understand what aspects of a simulator ought to be important. In the paragraphs below, each criterion shall be presented and discussed. They are listed in no specific order.

Cross Platform Support Nowadays, there are three main operating systems that are being used around the world, namely *Windows*, *MacOS* and *Linux*. With each of these operating systems bringing their own advantages to the table, it is not uncommon to see all three of them being used inside the same lab. For this reason, the more platforms the simulator supports, the more potential users the benchmark can reach.

Ease of Installation Given that the benchmark suite will run atop of an open-source simulator, the user experience related to the setup and installation of the underlying simulator becomes an interesting point to inspect. The best-case scenario would be a one-click installation which does not require any input from the user.

Code Quality Code quality is an essential aspect of open-source software, especially when the plan is to make modifications to the existing codebase. Working with a codebase where variables and functions are given abstract names can be a hassle, and severely increase development time. It thus should come as no surprise that this an important aspect to look at.

Extendibility Extendibility of a simulator can be described as how easy it is to add additional functionality to the codebase. This must not be confused with code quality. Code quality focuses on the quality of the already existing code. In contrast, extendibility focuses on well the software is modularised. A project with a modularised codebase (Where systems such as model dynamics, sensors, rendering are all independently defined) will make it easier for external developers to add additional features.

Headless support A small but important criterion is that the simulator should have the possibility of running headless. By running a simulator headless, it does not require a display to be connected to the computer. This opens up the possibility of simulating a server rack. This will potentially make it possible to create an interactive website where users can submit their algorithms, which are, in turn, processed by the simulator running in the backend.

Language Although a programming language by itself is no way related to the quality of a simulator (or any piece of software in general), having a simulator written in a performant, well-known language can help a lot with development in general. This language should, however, not be too low-level, as this will hinder the speed at which development can take place. Ideal candidates are languages such as C++ or Rust.

Community As explained in section 4.2, the community plays a large role in open source projects. When looking at open source simulators, we are specifically interested in how active the community surrounding the project is. By looking at the commit history, issue status, and community forums, a general idea of how active the community is can be formed.

Simulation Fidelity The level of fidelity in the simulation is typically dependent on the domain which is being simulated. In the case of MAV simulation, not all factors which can influence the drone are always taken into account. Support for gust vector fields may, for example, be missing.

Visual Fidelity Visual fidelity is a criterion which plays an important role in the benchmarking of obstacle avoidance algorithms. Especially in this thesis, where the focus is put on visual obstacle avoidance, it becomes clear that, in order to best guarantee results which accurately reflect the real world, the visual fidelity must be quite realistic.

Performance Performance, in theory, is a quantitative measure which depicts the efficiency in which a program can utilise the bounded resources of a computer. However, to accurately calculate this number is quite hard. There are many factors which influence the calculation, such as operating system scheduling, background tasks and I/O operations. Although in theory, it would be possible to account for this, it is outside the scope of this research. Thus, this criterion will mostly concern itself whether the simulator can run on medium to high-end laptops.

From this list of criteria, the most important ones are *Visual Fidelity*, *Simulation Fidelity* and *Extendibility*.

4.4. Simulator Review

This section will cover multiple simulators in detail. A brief introduction shall be given for each simulator, after which all the criteria are evaluated for that specific simulator.

4.4.1. Airsim

Airsim is a simulator built by Microsoft for simulation of cars and drones (Shah, Dey, Lovett, & Kapoor, 2017). It is built on top of unreal engine (Epic Games, n.d.), which can be classified as a semi open-source engine (requiring an account at epic games to access the source code). Build with contributions in mind, the simulator features a modular architecture, thus easily allowing contributors to adapt and improve the Engine. Remarkable is that Airsim provides the opportunity to connect external hardware on which the flight controller will run. This is also known as hardware-in-the-loop and allows the user to inspect how the custom flight software will perform in a safe, simulated environment.

Figure 4.1: A screenshot of the Airsim simulator. (Shah et al., 2017)

Visual Fidelity Airsim is written atop of Unreal Engine, a 3D game engine with a visual fidelity that is among the best of what is currently available in terms of real-time rendering. Unreal Engine supports both pre-computed and dynamic lighting, resulting in realistic lighting for static scenes, while still allowing for dynamic lights to present and influence the environment. A realistic example scene generated in unreal can be seen in Figure 4.2.

Figure 4.2: Photorealistic sample of a forest environment in Unreal Engine (Studios, 2019)

Cross Platform Support Airsim is officially only supported for both Windows and Linux. However, given that Unreal Engine is also available on macOS, an attempt was made to compile and run it on this platform. So far, it seems the macOS build is stable, and that most features are available. It does, however, require the user to compile LLDB, which can take between 30 and 60 minutes.

Ease of Installation The installation of Airsim and Unreal Engine for development purposes can be quite difficult. There are a lot of steps that need to be performed. Fortunately, once all the development work is done, the final benchmark suite can be compiled as a single binary and distributed. This makes it possible to distribute the entire benchmark suite as a single executable, thus allowing users to run it without any manual compilation.

Code Quality The code quality of Airsim is very clean. The source code is well structured and clear to read. Looking at sections of code, it immediately becomes obvious what purpose they serve. In addition to good code quality, the developers also created an extensive archive of documentation, containing

information on design choices, API usage, concept explanation, install instructions and much more. The project itself is also structured in a clear and concise manner.

Extendibility Airsim is as created as a plugin for Unreal Engine and is responsible for setting up the simulation environment and performing the actual simulation inside unreal. The simulator is built up from a set of extendable components which all work in unison to properly simulate a dynamic vehicle. These can be seen below. A visual overview of these components can be seen in Figure 4.3.

- Environment model
- Physics engine
- Vehicle model
- Sensor model
- Flight controller
- Public API

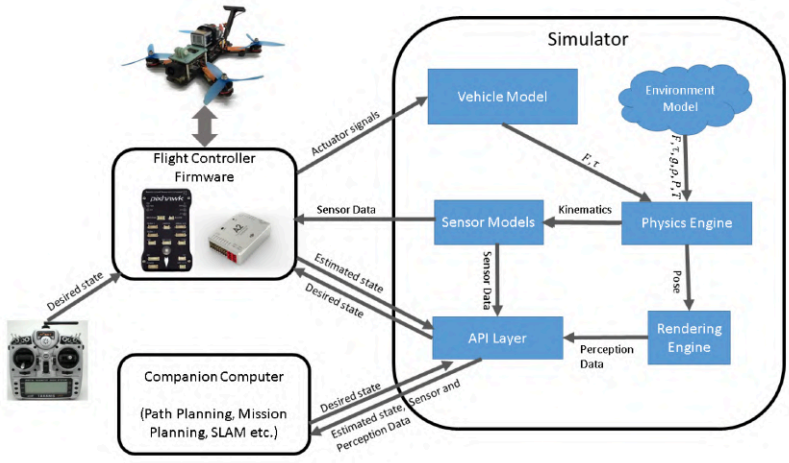


Figure 4.3: The modular architecture of the Airsim Simulator. (Shah et al., 2017)

The environment model is responsible for modelling the conditions in which the drone is operating. Airsim utilises mathematical models which are used to model the effect of gravity, air pressure, air density and magnetic field. Next, the physics engine is responsible for stepping the simulation forward. Internally, it keeps track of the position, orientation, linear velocity, linear acceleration, angular velocity and angular acceleration of the body being simulated. Every iteration, it receives signals containing the forces and torques acting on the body. Furthermore, apart from standard kinematics integration, the physics engine also offers the ability to compute linear and angular drag. This computation relies on an estimation for the drag coefficient, which is computed based on the dimensions of the vehicle. The vehicle model is responsible for outputting thrust and torque magnitudes to the physics engine based on the actuator input signals. Airsim provides an interface to define rigid body vehicles with an arbitrary amount of thrust and torque actuators. Parameters such as inertia, mass and drag coefficients can all be changed, which directly influence the integration step in the physics engine. Currently, Airsim provides one existing drone model which resembles a Parrot AR. Drone 2.0. It is, however, unknown if the model is also physically based on this drone. The sensor models are responsible for simulating sensor readings which can be passed to either an internal or external flight controller. By default, Airsim provides basic sensor models for GPS, accelerometers, gyroscopes, barometers and magnetometers. Camera sensors are implemented using unreal. The simulator comes packaged with a simple flight controller based on a cascade of PID controllers able to take in attitude, angle rate, velocity or position inputs. The controller can be run either on the simulating computer, or an external hardware-in-the-loop device such as the Pixhawk. Finally, the public API is responsible for allowing the user to interact

with Airsim core. By exposing endpoints, which the user can access either using Python or C++, the simulation can be controlled, and data such as images, attitude can be obtained. An obstacle avoidance algorithm would interact with this API to make decisions, and to send new headings.

Headless support Headless support is possible, as Airsim utilises Unreal Engine which has this feature natively available.

Language The core of Airsim is written in the C++ programming language. Airsim also provides a public API in both Python and C++, which can be used to interact with the simulation.

Community Due to the fact that Airsim is created by Microsoft, it quickly became very popular. Currently, although active development has been halted by the original developers, issues and bug-fixes are still being addressed. During the last three years, more than 2500 people have forked the project, and a total of 1500 issues have been resolved.

Simulation Fidelity The Airsim physics simulation runs at 1000 Hz. Each iteration, the engine checks for collisions, and updates the drone state based on its dynamic model. This dynamic model accounts for drag using the linear drag equation but does not take into account engine spooling. A trajectory evaluation was performed in which the simulated drone was to fly a circular and square trajectory in both the real world and simulation. This circle had a radius of 10m, and the square had 5m long sides. The Hausdorff distance between simulated and real-world for both trajectories respectively were 1.47m and 0.65m. These errors were explained by the Airsim authors to exist due to model inaccuracies, integration errors, and small random wind fields.

Performance The high visual fidelity of Unreal Engine does come with a performance cost. During real-time rendering, scenes must be rasterised and shaded every 16.6 ms (assuming a minimum of 60 frames per second are required). Given that it is not uncommon for complex scenes to contain thousands of triangles, this process can be quite expensive in terms of performance. Fortunately, this process runs almost entirely on the GPU (graphics processing unit). What this means, is that the performance cost for rendering is not in conflict with the cost of running the actual simulation, as this occurs on the CPU (central processing unit). However, as the simulation and rendering are both equally important, it requires the user to have an adequate high-performance graphics card installed. Users trying to run unreal Engine without a dedicated graphics card, which is often not found on basic laptops, will probably be unable to. It is, of course, possible to reduce the graphics quality in Unreal Engine, but as this directly affects the simulation results, it does not make for a good solution.

4.4.2. Hector

Hector is a simulator for quadrotor UAVs developed by researchers at TU Darmstadt (Meyer, Sendobry, Kohlbrecher, Klingauf, & von Stryk, 2012). It's built on top of the Gazebo simulator and ROS (robot operating system). Gazebo is a high-fidelity simulator created at the University of Southern California. It's able to simulate a variety of different vehicles/devices in various conditions. Over the years, Gazebo has become the de-facto simulator used within ROS. ROS, although called an operating system, is actually not one. The ROS website describes ROS as:

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. (Quigley et al., 2009)

Visual Fidelity As Hector is using Gazebo as the main rendering engine, the graphical fidelity is quite low. Gazebo was never meant to produce photorealistic renders, as the main focus of the software is to provide a simple interface to design environments for the simulation of robots. A screenshot of the Hector simulator can be seen in Figure 4.4.

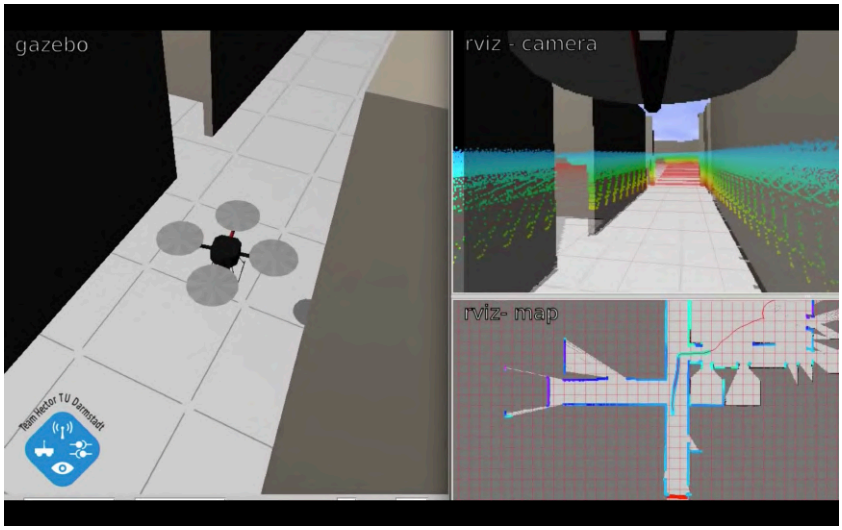


Figure 4.4: A screenshot of the Hector simulator. (Meyer et al., 2012)

Cross Platform Support The hector simulator is only available on Linux. This is due to the fact that ROS is only designed to run on Linux-based machines.

Ease of Installation Unfortunately, due to the lack of any available Linux machines, it was not possible to test the actual installation. Based on the provided installations instructions, and previous experience with ROS, it seems that installation is not very difficult for experienced ROS users. For new ROS users, however, it can be quite hard, as Hector will need to be compiled within the special ROS ecosystem, which can cause some confusion. Unfortunately, it is not possible to distribute the entire software package as a binary.

Code Quality The source code is readable, and the developers added some comments to clarify some difficult sections. Unfortunately, there exists almost no documentation apart from two pages on the ROS website.

Extendibility The Hector project is completely designed as a set of ROS modules. Each of these modules has a specific function during the simulation. To name a few, we can find modules related to the flight controller, drone dynamics simulation, gazebo initialisation, and pose estimation. Interaction with the simulator is performed using a feature provided by ROS, namely the messaging system.

Headless support It is possible to run Gazebo in headless mode. This will, however, disable any type of rendering, which makes it impossible to utilise sensors such as cameras.

Language All the Hector ROS modules are completely written in the C++ programming language.

Community The community surrounding Hector is not that large. Currently, a lot of issues are being left unanswered, and bugs are not being fixed by the maintainers. A lot of advantages come with Hector being build using ROS and Gazebo. The community surrounding these two pieces of software is quite large. Both are actively being maintained, and because many developers, both in industry and research, work with ROS, a vast number of external plugins are available which can tightly integrate with the simulator. Hector was able to use existing ROS plugins which model sensors such as RGB-D cameras and LIDAR. Sensors models for barometers, GPS receivers and sonar rangers were created by the Hector developers and shipped with the simulator.

Simulation Fidelity The Hector developers decided to create a custom drone dynamics model as they felt existing models lacked in fidelity. They noticed that the existing models were not taking into

account factors such as the motor and propeller dynamics, aerodynamics, external disturbances (e.g. wind), and noisy sensor signals. The model they created was that of a generic quadcopter, with additional focus on the propulsion model. This model considers factors such as bearing friction and load friction caused by the drag from the propeller. Fortunately, if necessary, custom user-made dynamic models can still be implemented.

Performance The run-time performance of a Gazebo-based simulator is much better than simulators using Unreal Engine or Unity3D. It is more likely that a user would be able to run both the simulation and the graphics engine on a low-spec laptop given that Gazebo utilises far fewer resources.

4.4.3. Flightgoggles

Flightgoggles is a MAV simulator created by researchers at MIT University (Guerra, Tal, Murali, Ryou, & Karaman, 2019). It is built on top of Unity3D, a game engine created by Unity Technologies. Although this Engine is not open-source, it has been taken into consideration, as it is free for personal usage, and all code related to the simulation is openly available in the Flightgoggles repository. Interaction with the simulator is performed using ROS. Like Airsim, Flightgoggles allows the user to attach external hardware-in-the-loop devices. However, ROS needs to be installed on the embedded device for this to work properly. Given that the simulator makes heavy usage of ROS, which is only available on Linux, Flightgoggles is not cross-platform compatible.

Visual Fidelity As Flightgoggles uses Unity3D as the underlying rendering engine, the visual quality of the simulation is, like Unreal Engine, among the best available in the industry. Both engines are able to create photorealistic environments and feature an asset store rich with content. The *"abandoned factory"* environment created in Unity3D by the Flightgoggles team can be seen in Figure 4.5.

Figure 4.5: A screenshot of the *abandoned factory* environment inside the Flightgoggles simulator. (Guerra et al., 2019)

Cross Platform Support As Flightgoggles is using ROS as a backend for communication between services, it is only possible to run the simulator on Linux.

Ease of Installation For the same reasons as discussed in subsection 4.4.2, using Flightgoggles can be somewhat more challenging for people with no ROS experience. Fortunately, once everything is installed, it is straightforward to start the simulator.

Code Quality The Flightgoggles project is made up of two different repositories. The Unity3D *FlightGogglesRenderer*[1] repository and the *Flightgoggles ROS*[2] repository. The *FlightGogglesRenderer*

[1] https://github.com/mit-fast/FlightGogglesRenderer
[2] https://github.com/mit-fast/FlightGoggles

repository is a unity project containing multiple assets such as Unity scripts, materials and much more. Both projects have clean code and comments where necessary. Documentation for Flightgoggles does exist, but it is very minimal. A total of only 12 pages exist, describing installation and usage instructions, but no development instructions.

Extendibility The *FlightGogglesRenderer* repository is responsible for setting up the simulation environment inside Unity3D. The *Flightgoggles ROS* repository is responsible for all the ROS related features such as hardware-in-the-loop support, UAV dynamics simulation, and status reporting. The UAV model can be replaced by a custom UAV model. However, the sensor implementation is part of the model file in Flightgoggles, which means that this requires the user to also re-implement the sensor models.

Headless support It is possible to run Flightgoggles in headless mode. Even better is that the developers have detailed instructions for setting up the Flightgoggles simulator in the cloud.

Language Flightgoggles makes use of multiple languages. For interfacing with the Unity engine, C is used. The other modules are all implemented using C++ and Python.

Community At this moment in time, the Flightgoggles community is still relatively small. The project itself is also currently not actively being worked on, apart from minor bug-fixes and issue handling.

Simulation Fidelity The Flightgoggles dynamics simulation runs at 960Hz. These dynamics are simulated based on a generic quadcopter model, taking into account factors such as aerodynamic drag, motor spooling and a stochastic variable to model uncertainties such as turbulence and vibration. Only a single dynamic model is currently available, but given that Flightgoggles uses ROS, it should easily be possible to use an already existing ROS model.

Performance For the same reason as discussed before, high-fidelity simulation comes at the cost of worse performance. Fortunately, the performance hit is mainly caused by the GPU, which does not affect the physics simulation. As long as any modern computer is used, the dynamics simulation should not give any issues.

4.4.4. RotorS

RotorS is a simulator for MAVs developed by researches at ETH Zurich (Furrer, Burri, Achtelik, & Siegwart, 2016). Like Hector, the simulator is built on top of the Gazebo simulator and the ROS framework. The explanation on ROS gazebo was already discussed in subsection 4.4.2. The RotorS simulator is described as a simulation framework, allowing the user to quickly swap between different quadcopter models, flight controllers, sensors and state estimators.

Visual Fidelity With RotorS using Gazebo as the main rendering engine, the graphical fidelity is relatively low. As mentioned in subsection 4.4.2, Gazebo was never meant to produce photorealistic renders. A screenshot of the Hector simulator can be seen in Figure 4.7.

Cross Platform Support The rotors simulator is only available on Linux. This is because ROS is only designed to run on Linux-based machines.

Ease of Installation RotorS has very detailed installation and usage instructions listed on their main webpage. This makes it very easy, even for people with no previous ROS experience, to install and run the simulator. Unfortunately, it is not possible to distribute the software package as a binary, given that ROS is being used.

Code Quality The RotorS project is very well structured. The source code looks clean, and comments are listed in some files which may be difficult to understand. RotorS provides a lot of documentation for installation, usage and development purposes.

Figure 4.6: A screenshot of the RotorS simulator. (Furrer et al., 2016)

Extendibility The RotorS project is completely designed as a set of ROS modules. These modules are responsible for subsystems such as Communication, Dynamics simulation and HID readout. A large focus of RotorS is that it allows users to easily add extra functionality to the simulator. A visual description of the architecture can be seen in Figure 4.7

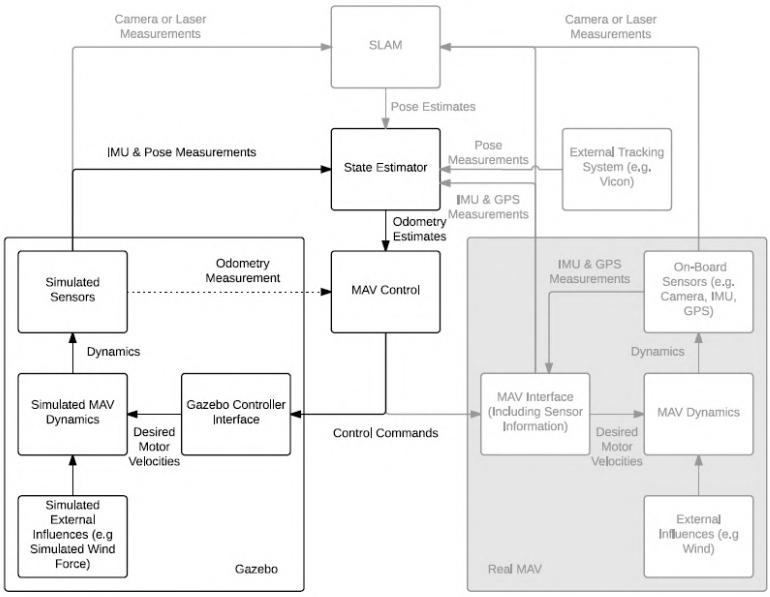


Figure 4.7: The RotorS simulator architecture. (Furrer et al., 2016)

Headless support It is possible to run Gazebo in headless mode. This will, however, disable any type of rendering, which makes it impossible to utilise sensors such as cameras.

Language RotorS almost entirely uses C++ to create their ROS modules. Interfacing with the simulator to design scenarios and analyse data is all done using Python.

Community There is a large community surrounding RotorS. It is currently one of the most popular Gazebo-based drone simulators based on GitHub activity. The project still seems to be maintained but

not very actively. Issues, however, are still being resolved, which shows that the original maintainers are still supporting the existing community.

Performance The run-time performance of a Gazebo-based simulator is much better than simulators using Unreal Engine or Unity3D. It's more likely that a user would be able to run both the simulation and the graphics engine on a low-spec laptop given that Gazebo utilises far fewer resources.

4.5. Trade-off Conclusion

Having reviewed the most relevant potential simulators, the actual trade-off is now presented in this section. The winner will be used for the obstacle avoidance benchmark suite. This trade-off is structured as follows. The simulators are compared based on the most relevant criteria in section 4.3.

Visual fidelity is among the most important criteria in this benchmark. Given that the focus of this suite is the benchmarking of mono and stereo-vision obstacle avoidance algorithms, the best way to ensure that the results obtained from simulation are comparable to real-life is to create photorealistic environments for the algorithms to be simulated in. An unfortunate side effect from this is that it makes the benchmark suite less accessible to users without a powerful graphics card. Fortunately, it is nowadays possible for such heavy programs to run in cloud instances such as Amazon AWS[3] or Microsoft Azure[4].

Of course, visual fidelity by itself is not enough to be able to produce realistic results. A good physical model (simulation fidelity) of the drone is also necessary in order to properly simulate manoeuvres and escape trajectories. All simulators discussed in the previous section all have access to at least one proper drone model by default. The fidelity of these models is, however, quite different, which is hard to express with a quantitative value.

Finally, extendibility and code quality are in this case very important. The benchmark suite will be an extra layer on top of the simulator, responsible for setting up different scenarios and keeping track of different metrics during the mission. This means that a lot of communication is necessary between these two systems. It is therefore very important that the simulator is extendable, such that extra software modules can easily be added. Good documentation and code quality are also a must. This can reduce the development time of the benchmark suite by a large amount, as the developer will need less time to understand the entire code base and its behaviour.

With these main criteria being responsible for the final choice, it quickly becomes obvious that Hector and RotorS are no good options. The constraint for high visual fidelity makes it difficult to consider gazebo-type simulators. This is an unfortunate decision, as the ROS framework provides a lot of different features such as sensor and drone models. This leaves both Airsim and Flightgoggles. The Simulation fidelity of both these systems is roughly the same. They both operate their physics engine at 1000Hz and 960Hz respectively and include a generic quadcopter model with atmospheric drag incorporated. These models will probably not be very accurate when simulating complex manoeuvres and fast velocities, but will not cause any issues as this will not occur during an obstacle avoidance benchmark. Finally, based on the extendibility and code quality, it quickly becomes obvious that Airsim has the upper hand here. Airsim has more than 60 pages of documentation, providing information on subjects such as program architecture, API usage, scenario creation, hardware-in-the-loop setup, and much more. This is a large difference in contrast to Flightgoggles, which only has ten pages.

There certainly more advantages which Airsim has compared to Flightgoggles. One of these is the ability to package all the software in a single binary makes it easy to distribute the software to more users. This is not as easily possible with Flightgoggles, as it relies on ROS, which cannot be bundled.

Overall, Airsim seems to be the clear winner of this trade-off, and will, therefore, be used as the basis for this benchmark suite. The final overview of all the trade-off criteria and their results for the discussed simulators can be seen in Table 4.1

[3] `https://aws.amazon.com/`
[4] `https://azure.microsoft.com/en-us/`

Table 4.1: Results of the simulator trade-off

Simulator	Cross Platform Support	Ease of Installation	Code Quality	Extendibility	Headless support	Language	Community	Simulation Fidelity	Visual Fidelity	Performance
Airsim	MacOS, Windows, Linux	Very easy	Good	Very good	Yes	C++, Python	Large Community	Good	Very Good	Good (GPU required)
RotorS	Linux	Easy	Good	Very good	No	C++	Large Community	Very good	Bad	Very Good
Flightgoggles	Linux	Easy	Good	Good	Yes	C++, C#	Small Community	Good	Very Good	Good (GPU required)
Hector	Linux	Easy	Average	Good	No	C++	Small Community	Very good	Bad	Very Good

5 Performance

An important question that quickly arises when thinking about benchmarking any type of algorithm is: *How is performance measured?*. It is evident that this question can have a different answer depending on what exactly is being benchmarked. An algorithm developed to control traffic lights at an intersection might use metrics such as total throughput and average waiting time, while a racing drone algorithm might look at lap time and the percentage of correct loop detection. Metrics are thus very domain-specific. In this chapter, we will be looking at some of the metrics for obstacle avoidance algorithms that have already previously been used in research.

5.1. Performance metrics

Performance metrics for obstacle avoidance algorithms can be categorised into multiple different ways. In this thesis, they are categorised in *task-related metrics* and *Obstacle avoidance-related metrics*.

The first, *task-related metrics*, measure how well a MAV equipped with an obstacle avoidance algorithm can carry out a particular task. Such a task may be flying across a field of obstacles or following a set of way-points without colliding. Metrics related to these types of tasks could, for example, be task-completion time and total energy consumption. The general idea is that by measuring these high-level metrics, they intrinsically tell us something about the performance of the underlying algorithm. This, in a sense, is of course correct. If algorithm A performs better than algorithm B by traversing the field faster and with less energy, then it must mean that A is better than B at that specific task. The issue with this approach is that algorithms may perform differently for different environments or tasks. If details like these are not taken into account, the results obtained by such a benchmark are useless at best, as they can not be used to extrapolate to different scenarios. If we can, however, find a way to adequately describe these tasks and environments, we can relate performance to these descriptions, and thus get a better picture of what influences the performance for these algorithms.

The other method, *Obstacle avoidance-related metrics*, focuses specifically on the algorithm itself, and not so much on the environment and tasks. In the case of obstacle avoidance algorithms, obstacle avoidance-related metrics might look at the percentage of correctly detected obstacles or the optimality of the generated avoidance manoeuvre. A potential issue with these types of metrics is that they require direct interfacing with the algorithm on a software level. In a benchmarking environment, this means that the software will need to have access to the intermediate steps of an algorithm written by a user. This could be achieved by utilising abstract classes, that contain functions such as *fn detect_obstacles -> detected_obstacles* which must be implemented by the user. These functions could then be called by the benchmarking suite to compare the results to the ground truth before returning the data back to the user. Although utilising abstract classes for the benchmarking of intermediate algorithm steps is a good solution, an issue that quickly arises is that not all algorithms are equal. This means some algorithms may not even need to implement some abstract function, as it is not an intermediate step that needs to be calculated for the algorithm to function.

As was already briefly discussed in the paragraph on task-related metrics, performance measurements of obstacle avoidance algorithms require the environment to be properly described in order to extrapolate the results to other scenarios. The concept of environment descriptions is certainly not new and will be discussed in more detail in section 5.2. The main difficulty in describing environments is picking the correct type of metrics to represent the environment. Henceforth, these will be called *environment-metrics*.

For both Task-related, Obstacle avoidance-related and environment metrics, it is also possible to make a distinction between local and global measurements. Whereas local measurements try to capture a specific metric at each time-step, global measurements look at the entire mission. An example of local and global types of measurements are task-related metrics such as instantaneous velocity, and average velocity, respectively.

5.2. Commonly used performance measures in literature

In this section, we will discuss some of the metrics seen in current literature used to measure the performance of algorithms in MAVs.

First, we shall look MAVBench, a closed-loop benchmarking suite for MAVs developed by Boroujerdian et al. (2019). It was created to help researchers identify bottlenecks between hardware and software in MAV applications, and providing a benchmark suite for some common MAV tasks such as, e.g. search-and-rescue missions, 3D mapping, and indoor navigation. A more detailed explanation of this suite will be discussed in Chapter 6. For now, though, we will focus on the metrics that are used to measure performance during and after flight. MAVBench uses a variety of different ”Quality-of-Flight (QoF) Metrics” to measure the performance of a MAV performing a mission. Due to the fact that not all missions are equal, these metrics are not always the same. However, there are two which are common for every mission. These are *total energy consumption* and *mission time*. The total energy consumption measures, as the name implies, is the energy consumption of the MAV throughout the mission. For this, the authors of MAVBench implemented an energy model in simulation, which is able to estimate the power consumption of the motors and hardware. The mission time metric measures how long it takes for the MAV to complete a certain goal, from the moment the mission simulation started. MAVBench provides no algorithm-related metrics.

Next, we will look at the work published by Nous et al. (2016). They propose a performance framework for the evaluation of obstacle avoidance algorithms. In this framework, the bring forth the distinction between *detection* and *avoidance* performance.

Detection performance is concerned with evaluating the performance of how well the obstacle avoidance algorithm is able to detect potential obstacles. In the paper, they predominantly focus on stereo vision cameras as the input sensor. The performance of the detection step is measured by having the algorithm measure the distance to a fixed obstacle under different conditions and comparing the result to the actual distance. By changing variables such as the actual distance to the obstacle, the illumination, and the texture contrast, it is possible to get a quantitative measure of how large the error is under different conditions. An image of the obstacle that was used for these tests can be seen in Figure 5.1

For the avoidance performance evaluation part of the framework, the objective is to find under which conditions detected obstacles could be avoided. Here Nous et al. (2016) discusses how, due to the variety in avoidance methods, it is difficult to find metrics that are applicable to all. For this reason, they decide to utilise task-related metrics such as *Success rate* and *Path optimality*. Furthermore, they argue that the performance of the algorithm is directly related to environmental conditions. Therefore, they also introduce a set of novel environment metrics to better relate the performance of avoidance algorithms to the conditions in the environment.

We shall first discuss the task-related metrics before touching upon the environment metrics. As

Figure 5.1: Test obstacle used for the evaluation of detection performance in (Nous et al., 2016)

was already briefly mentioned. The framework uses *Success rate* and *Path optimality* to evaluate the performance of the avoidance algorithm. The success rate specifies whether the MAV was able to reach the goal condition. Three possible scenarios are distinguished. Flights in which the MAV reaches the goal safely, flights in which the MAV gets stuck in a local minimum (goal is not reached, but no collisions occur), and flights where a collision occurs. The path optimality metric contains two sub-metrics which are used to measure the performance. These are *travelled distance* and *average velocity*.

Moving on to the environment metrics, these serve as a way to categorise the environment into a set of independent variables. In the paper, they propose the following metrics, namely *Traversability*, *Collision state percentage* and *Average avoidance length*. Traversability is related to the density of the obstacles in the environment. This metric is seen as an improvement to the standard *obstacle density* metric, as it accounts for the size of the MAV, and thus gives a better image of how well the MAV can manoeuvre through the obstacle field. Traversability is calculated by picking several random points in the area with random headings and finding the distance to the closest obstacle for each point along the chosen heading. Afterwards, these distances are averaged and divided by the radius of the drone to make the metric non-dimensional. The formal equation for traversability is given in Equation 5.1, where N is the number of samples, r is the radius of the drone, and s is the measured distance from the sample to the closest obstacle along a random heading.

$$Traversability = \frac{1}{N \cdot r} \sum_{0}^{N} s \tag{5.1}$$

The collision state percentage metric tries to find the percentage of collision states for different parts of the environment. By taking into account the MAVs dynamics, the authors of the paper argue that it becomes possible to calculate the number of states in a given area that gives rise to unavoidable collisions. Thus, if the MAV would be at a point where the value for this metric would be 100%, it would mean that a collision with the obstacle is unavoidable. Of course, depending on the initial conditions for heading and velocity, this measure can differ radically for the same environment.

Finally, the last metric, average avoidance length, tries to quantify the difference between forest-like environments (environments with small circular obstacles), and environments with more wide obstacles. The metric looks at the lateral distance required, with respect to the velocity vector of the MAV, to avoid obstacles at each time-step of the flight. The final result is calculated by averaging these values and dividing them by the radius of the MAV to make the metric non-dimensional. A visual representation of these three environment metrics can be seen in Figure 5.2.

Next, we will take a look at a paper published by Lampe and Chatila (2006) on performance measures for mobile robots. In this paper, the authors discuss some new metrics, and ones we are already familiar with. All these metrics, however, are task-related, and thus directly relate the performance of a given algorithm with how well the task is performed. To name a few, metrics such as *average velocity*, *average distance*, *mission duration* and *mission success rate* are mentioned. The purpose of

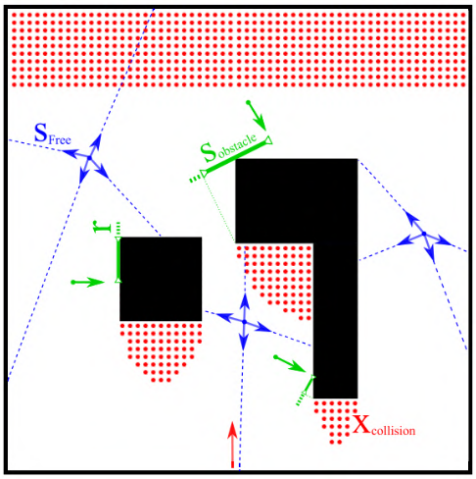


Figure 5.2: Visual representation of the Traversability, collision state percentage, and average avoidance length environment metrics (Nous et al., 2016)

these metrics speaks for themselves. The addition of this paper is how they then go on to relate these metrics to the environment. For this, they introduce the environment metric called *world complexity*. This metric utilises information theory to compute the entropy of the entire environment. First, a binary occupancy grid is created based on the environment, as seen in Figure 5.3. Next, a window moves over the entirety of the grid, and for each position calculates the obstacle occupancy percentage. For a window size of 2, the possible combinations are: 0, $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$ and 1. The total number of each combination is counted, and the entropy is then calculated by Equation 5.2, where p_i corresponds to the frequency of each combination.

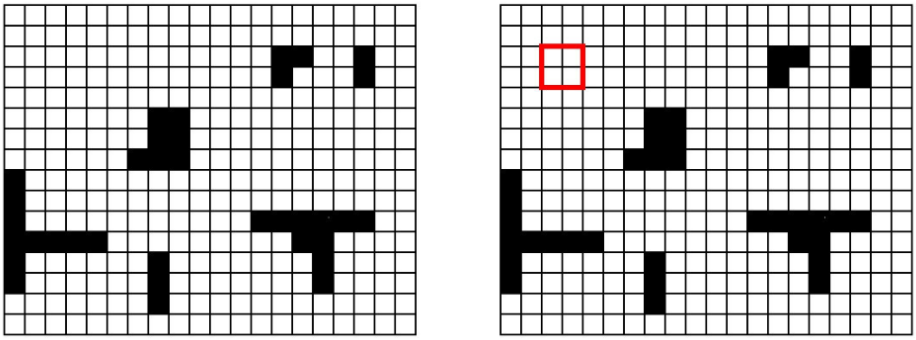


Figure 5.3: Example of an occupancy grid with window size 2 (Lampe & Chatila, 2006)

$$H = \sum_i p_i log(\frac{1}{p_i}) \tag{5.2}$$

They go on to show that the task performance for a navigation mission, namely the *mission duration*, increases, as the world entropy becomes higher as can be seen in Figure 5.4. There is a lot of variance in the results, which again shows that it is probably difficult to describe an environment using only global metrics.

5.3. Performance metrics discussion

Having mentioned some of the different metrics for obstacle avoidance algorithms in literature, we move on to the discussion of them, where we lay out potential shortcomings and potential methods for how to solve them.

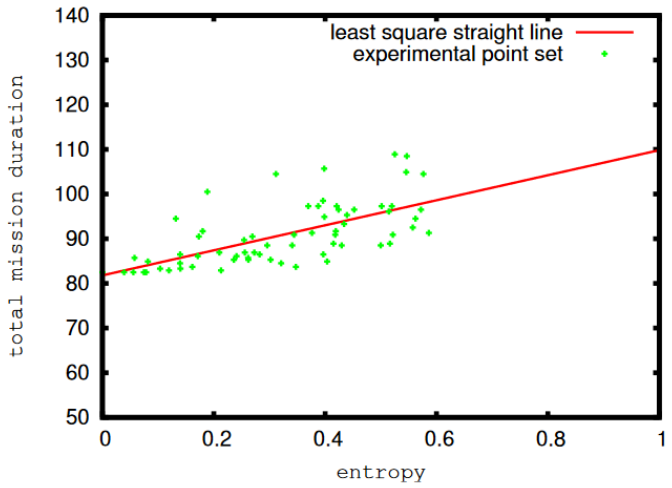


Figure 5.4: Simulated results for a navigation mission in different environments. (Lampe & Chatila, 2006)

In the last section, we mentioned a total of three different types of metrics, namely task-related metrics, obstacle avoidance-related metrics and environmental metrics. All of these can be seen summarised in Table 5.1. It quickly becomes apparent that there exist plenty of task-related metrics, while obstacle avoidance-related metrics are seen less often. One reason for this, as was already discussed in section 5.1, is that not all obstacle avoidance algorithms are equal. Due to the fact that these types of metrics are most often calculated based on intermediate values, it can become quite hard to find ones which apply to all obstacle avoidance algorithms. For this reason, it has currently decided to halt the usage of obstacle avoidance-related metrics and only implement them if time permits it.

5.3.1. Task-related metrics

First, we shall briefly discuss the task-metrics. Most of these metrics apply to any type of mission, whether that may be an exploration-oriented or goal-oriented mission. The *Mission completion time* metric is only applicable to situations where it is possible to complete the mission. This is often not the case in exploration-oriented missions, where the goal, most of the time, is for the MAV to create a map of the environment based on its sensor measurements. As this type of mission requires a special algorithm for map generation, which is not always needed for obstacle avoidance algorithms, exploration-oriented missions, and thus exploration-oriented metrics, are for now irrelevant for this research.

The *success rate percentage* is a bit of an unusual metrics, as the value is based on multiple mission runs. At the end of each mission, the end state is stored (completed, crashed or stuck). This vector of values is then used to calculate the percentage of completed, crashed and stuck mission. For such a metric to make sense, we expect that the missions which are taken into account are of similar difficulty, and take place in the same environment. Otherwise, the numerical results will start to lose meaning, as the result of different scenarios are mixed, making it impossible to relate the value to other metrics.

The *Total energy consumption* metric is also an exciting metric, as it gives the user an idea of how energy efficient an algorithm is. Implementing such metrics, however, can be quite tricky, as it requires a model of the power usage of the drone for components such as the motors and computer. In the case that the algorithm runs on the host computer, it is even impossible to measure the exact power usage of the computer part.

The *average velocity* and *instantaneous velocity* are relatively straight forward metrics. The instantaneous velocity, however, can be quite useful in confirming relationships between different environment metrics. This is because instantaneous velocity is sampled every time-step.

Finally, the *travelled distance* metric could potentially be used for exploration-based tasks where the goal is to fly as far as possible in a given time-window.

5.3.2. Environment metrics

There are a few concerns I have with the traversability metric. First is the fact that the measure is inherently very random, requiring three independent random number samples. This means, especially in large environments, that two separate measurements for traversability require many samples to get similar values. Furthermore, because traversability is calculated by sampling the complete environment, the metric is only applicable to environments which have a uniform distribution of obstacles. It may be possible to subdivide each environment in a set of windows, but this would then require a good understanding of how window size affects the overall result.

A potential solution would be to make the traversability metric more predictable, and local instead of global. By, for example, iterating in a grid-like fashion over the entire environment, taking multiple free path measurements for point, and averaging them, it becomes possible to calculate a local traversability-like metric. Generally, it is more difficult though to relate global measures to local ones, as one is a singular value, while the other is a time-series.

The *collision state percentage* metric can be quite difficult to compute in simulation, as it requires the worst-case projection of the drone state. This is not only very expensive computationally, but it can also be challenging to implement. A more simplified method, such as limiting the possible number of projections, can make this more feasible.

The *World complexity* metric again has some of the same issues as traversability. The fact that it is a global metric requires the environment to be uniform; otherwise, it becomes impossible to define an accurate relationship between performance and the environment. Furthermore, because world complexity relies on a window size, one must choose a value for this beforehand. Choosing the correct size, if such a thing even exists, can be quite hard and might be dependent on the scale of the environment.

Task-Related Metrics	Description	Global/Local
Total energy consumption	The total energy consumed by the MAV at the end of the mission	global
Mission completion time	The time it took for the MAV to complete the mission	global
Success Rate	Specifies whether the MAV reached its goal, collided, or got stuck in a local minimum	global
Travelled distance	The distance the MAV travelled at the end of the mission	global
Average velocity	The average velocity of the MAV during the mission	global
Instantaneous velocity	The velocity of the MAV at every timestep	local
Obstacle avoidance-Related Metrics		
Detection performance with distance	The measured distance to the obstacle detection measured for different distances	Unit-test metric
Detection performance with illumination	The measured distance to the obstacle detection measured under different illuminations	Unit-test metric
Detection performance with texture contrast	The measured distance to the obstacle detection measured for values of texture contrast	Unit-test metric
Trajectory optimality	A metric measuring the optimality of the generated avoidance path	global
Percentage correctly detected obstacles	A metric measuring the percentage of correctly detected obstacles during a mission	global
Environment-Related Metrics		
Traversability	A non-dimensional metric to indicate how difficult it is for a MAV to manoeuvre between obstacles	global
Collision state percentage	Metric to calculate the number of collision states in the environment	semi-global
Average avoidance length	The average lateral distance required to avoid obstacles at every timestep	local
World complexity	A measure of the entropy in the world	global

Table 5.1: Table summarising metrics discussed in Section 5.2

6

Benchmarking

The last chapter of this literature study will focus on the process of benchmarking in general. First, we will explore some of the current benchmarks often used in research. These include will include benchmarks both robotics-related and non-robotics-related fields. Afterwards, we shall discuss the necessary parts that ought the be included in an obstacle avoidance benchmarking suite.

6.1. Current state of benchmark suites in literature

In the context of algorithms, benchmarking is the concept of trying to measure the performance of an algorithm and comparing it with similar algorithms to get an idea of their relative performance. A distinction can be made between two types of benchmarks, namely active and passive. Passive benchmarks work by supplying the algorithms with a specifically chosen dataset. Such datasets are created beforehand and are the only data needed in order to perform the benchmark. In passive benchmarks, the environment is only observed, and no interaction takes place. This is in contrast to active benchmarks, where the algorithm can directly influence the environment, thus affecting the observed data. This gives rise to a feedback loop, which can only be evaluated in simulation. Therefore, active benchmarks are much more involved to create.

A popular application of benchmarks is seen in the computer vision sector. Most computer vision algorithms by themselves are passive. The datasets fed to these algorithms mostly consists of lists of images or videos. This makes it far easier to write benchmarks for computer vision algorithms, as they only require a dataset and no simulation of vehicles or robots.

A good example of a very successful computer vision benchmark is the KITTY vision benchmarking suite by Geiger, Lenz, and Urtasun (2014). This suite provides the possibility to benchmark algorithms related to visual odometry, stereo disparity map generation, object recognition and much more. These benchmarks are performed on a dataset created by the KITTY team, generated using their automotive platform. A comprehensive leaderboard is shown on their website, listing all submitted algorithms together with the measured performance of each. These leaderboards provide researchers with an excellent overview of what are currently the best algorithms available for a specific task.

Unfortunately, no benchmarks in the field of robotics currently exist with the level of attention and detail like KITTY. The fact that robotics tasks, including obstacle avoidance, are closed-loop systems, makes it so that designing a benchmark must allow for the algorithm to control the robot. As one can imagine, this makes it more challenging to design a reliable benchmark, as suddenly, many different situations can occur due to the algorithms making different decisions, causing a wide array of different states. If two algorithms cause the robot to take different paths throughout the environment, it becomes harder to compare the performance of the algorithms, as they have not been through the exact conditions. Nevertheless, there are still many attempts at creating robotic-type benchmarks.

One attempt at creating a drone benchmark was made by (Boroujerdian et al., 2019). They created

MAVBench, a benchmark designed to measure the performance of drones executing different missions such as search-and-rescue, mapping and planning operations. Built on-top of Unreal engine and Microsoft Airsim (Shah et al., 2017), the benchmark suite offers researchers a test platform to develop their algorithms and compare them to other algorithms in the field. The benchmark currently has no support for measuring the performance of obstacle avoidance algorithms. Nevertheless, it is still a high-quality benchmark suite utilising up-to-date technologies.

Another example is the RoboBench benchmark created by Weisz, Huang, Lier, Sethumadhavan, and Allen (2016). This is a more general benchmark meant for the evaluation of any type of robot, doing any type of task. The paper introduces a method to generate ”evaluation” containers, which specify all the relevant models, environments and sensors necessary to evaluate a particular mission. Although it is called a benchmark, it is better to define it as a framework, as they do not include any actual benchmarks. It is designed to allow researchers to quickly generate benchmarks using their framework. This concept of ”evaluation” containers is a promising framework which might be used in the obstacle avoidance benchmark suite to easily create different benchmark sessions.

Quite recently, in 2019, a benchmark suite for obstacle avoidance algorithms showed up. Although this paper was not published, it still is a valuable resource as the content closely aligns with the subject of this thesis. The suite, named BOARR, utilises both RotorS, a 3D drone simulator based on Gazebo, and ROS. The BOARR suite performs the obstacle avoidance benchmarks inside a randomly-generated forest environment, which contains sections with high obstacle density, medium obstacle density, and low obstacle density (shown in Figure 6.1). To measure the performance of the algorithm, a total of 1060 different missions are required to be performed for statistical significance. Each mission consists of the drone following a set of randomly generated waypoints. The mission is stopped when either the drone collides with an obstacle, or when it has flown a distance of 1km. Based on the result of these missions, the benchmark suite generates a performance report. The main metric is expressed using a single variable called the collision probability. (Tezenas du Montcel, Negre, Gomez-Balderas, & Marchand, 2019)

Figure 6.1: Top-down view of a generated benchmarking environment in BOARR. (Tezenas du Montcel et al., 2019)

Unfortunately, although the paper lists a performance metric which can be used to compare different algorithms, the results do not necessarily extrapolate to different real-world scenarios. As all the tests are performed inside a forest-like environment, it is hard to predict if algorithm A is still better than B

when placed inside a city-like environment. This issue arises because the authors have not thought of a way to relate algorithm performance to an environment description. Furthermore, the fact that BOARR utilises RotorS to perform the simulation means that the visual fidelity in lacking. As can be seen in Figure 6.2, the benchmark does not produce a photorealistic environment. This makes it more challenging to relate benchmark performance to real-life performance.

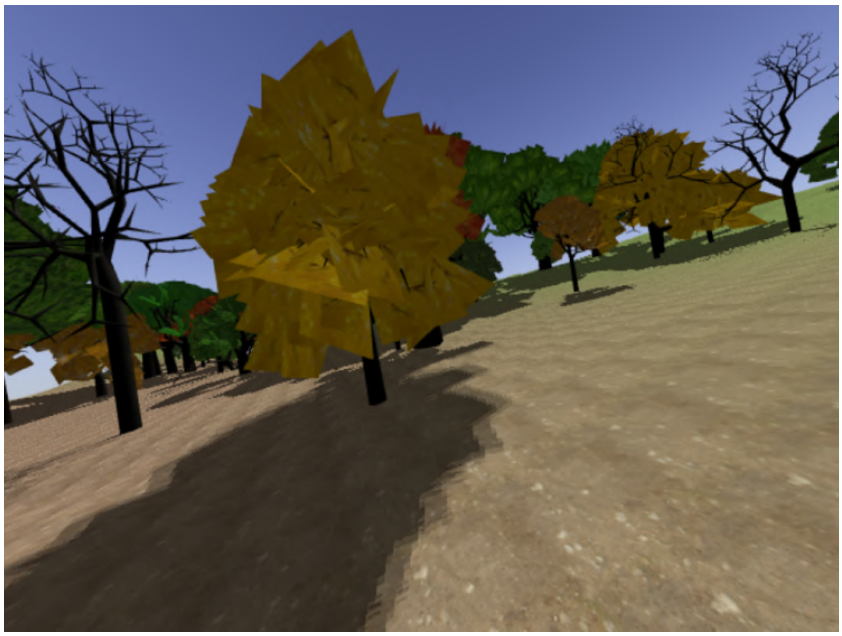


Figure 6.2: First-person view of a generated benchmarking environment in BOARR. (Tezenas du Montcel et al., 2019)

Finally, we will be discussing CARLA created by Dosovitskiy, Ros, Codevilla, Lopez, and Koltun (2017), a self-driving car simulator. This simulator is actively being developed in order for researchers to test, train and validate their self-driving algorithms. It offers the possibility to develop custom self-driving benchmarks. CARLA does this by exposing a python interface which is responsible for setting up the environment and agents. At this moment in time, CARLA currently only offers a single benchmark in which is measures four types of metrics: Distance travelled, average velocity, number of collisions, and number of times the car left the lane by accident. These metrics are tested in two different environments, and the results are compiled in a final performance analysis.

6.2. Obstacle avoidance benchmark anatomy

A benchmark suite consists of many different independent parts, all working together to evaluate a specific algorithm. In the case of an obstacle avoidance benchmark, these are parts such as:

- Tasks/Objectives
- Environments
- Performance measurements

A single benchmark session consists of an obstacle avoidance algorithm trying to complete a particular task in a specified environment. During this session, the benchmark suite continuously measures a set of metrics which are later used to generate a performance measurement. In this section, we shall discuss the anatomy of such an obstacle avoidance benchmark suite in more depth. We shall look at what types of tasks are available, and discuss potential environments which will be flown in.

6.2.1. Benchmark tasks

The benchmark suite is responsible for providing the obstacle avoidance algorithms with a goal.

There are many possible types of tasks that could be given to an obstacle avoidance algorithm. To list a few, one could give a *goal-related task*, where the objective is to reach a specific goal in the least amount of time while avoiding obstacles. Another task could be an *exploratory-related task*, where there the goal is to fly for the longest possible interval without colliding. These tasks can also be subdivided into different categories.

A *goal-related task* may specify its goal precisely using 2D/3D coordinates or a fixed-size area. In the case of a precise goal, a planning algorithm is also necessary for the drone to reach the goal location. This can pose an issue, as not all obstacle avoidance algorithms use a planning algorithm to find the final goal. These ”pure” algorithms detect if there is an obstacle in front of the drone, and can take reactive measures to ensure that no crash occurs. If these types of algorithms support reference headings, it might still be possible to make use of fixed-size area goals.

An *Exploratory-related* mission can also be posed in different ways. The drone can be allowed to free fly for a certain amount of time. In this case, the direction in which the drone travels does not matter. The goal is to keep moving without hitting any obstacles. A potential issue with this type of mission is that the algorithm might try to avoid complex situations by keeping to areas without obstacles. Without a goal objective, it becomes hard to ”force” the algorithm through complex parts of the environments. This would thus make it difficult to measure the performance of such an algorithm over a range of environmental metrics.

6.2.2. Benchmark environments

The benchmark tasks will take place in a variety of different environments, which will likely be made up of varying obstacle types. There are two possible ways to go about creating these environments. They could either be procedurally generated or designed by hand.

Procedurally generated environments are perfect for ensuring that the performance of an algorithm is measured for the entire range of environmental metrics. The environments can be created such that they have, for example, a varying traversability for each generation. The only potential issue with this is that generating photorealistic environments is very hard. It is not as simple as placing obstacles at the right locations. They also need to fit in well with the rest of the environment by having the proper textures. Furthermore, to create a feeling of a real environment, many different assets need to be placed. In an indoor environment, one may think of assets such as plants, desks, lamps, etc. Finding a good way to place these assets in generated environments can be difficult.

The exceptions to this are forest-like environments. These are relatively easy to generate, given that forest environments are often only made up of trees, bushes, grass and soil. The placement location of these assets does not really matter. Most of the time, it will still look like a real forest. Unfortunately, the obstacles found in such environments are always cylindrical. In order to get a good view of how the algorithm performs under different circumstances, it is therefore not satisfactory to only test in forest environments.

The other option is to manually design specific environments. Instead of generating many environments, a set of specific environments is chosen, which is then designed by hand to have multiple different sections of varying difficulty. Such a process is much more work-intensive, but it should produce better photorealistic results, as the designer is in full control of the art style.

Of course, it is possible to have a combination of both methods in the benchmark. Two or three environments can be manually designed, testing the algorithms in common cases often seen in real life, while a lot of more basic environments are procedurally generated to collect data in regions that were not covered by the handcrafted environments.

7 Conclusion

In this literature study, we introduced the main topic of this thesis, which is the development of an obstacle avoidance benchmarking suite. In order to complete this task, a set of research questions were first formulated.

- *What benchmark metrics should be used to measure the performance of obstacle avoidance algorithms?*
- *What metrics should be used to describe a complex environment?*
- *What are the expected relationships between the performance and environment metrics?*
- *What types of environments should be used in the benchmark?*
- *What missions should be evaluated in the benchmark simulation?*
- *What should the software architecture of the benchmark suite look like?*

Based on this list of questions, a handful of important subjects could be deduced on which more knowledge would be required. These were Obstacles avoidance algorithms, simulators, performance evaluation and benchmarking.

First, we reviewed obstacle avoidance algorithms in general, looking at what steps are necessary throughout the entire algorithm to avoid an obstacle. The first step of any obstacle avoidance algorithm is to take in information about the surrounding world and detect obstacles that are in the immediate surroundings. Here we focused mainly on both mono and stereo vision-based cameras and reviewed a list of methods that can be used to detect obstacles. Afterwards, based on the detected obstacles, the algorithm needs to decide on what action should be performed. We reviewed a handful of methods which are responsible for this task, such as rule-based methods and planners. Finally, the algorithm passes the action to the flight controller, which causes the MAV to perform the specified manoeuvre.

Next, we briefly discussed the role of simulators in the field of MAVs and listed the common subsystems which are present in those simulations. Afterwards, we introduced a set of trade-off parameters which would then be used to perform a trade-off between a set of simulators. After a detailed discussion of Airsim, Flightgoggles, Hector and RotorS, it was decided that Airsim would be the simulator of choice for this thesis due to its high visual fidelity and good software quality.

Next, we looked at some performance metrics commonly used for evaluating obstacle avoidance algorithms. Here, we made the distinction between task-based metrics and obstacle avoidance-based methods. The former type of metrics does not directly evaluate the algorithm, but rather assess how well it does in performing a specific task. The obstacle avoidance-based methods, on the other hand, directly evaluate the algorithm, assessing the performance of how well it, e.g. detects obstacles or generates trajectories. Because obstacle avoidance-based metrics do not apply to all algorithms, it

was decided that these would not be implemented for now. Finally, to relate the performance to the environment, a set of environmental metrics was discussed to quantify the properties of the environment.

In the last section, we focused on the process of benchmarking and discussed some noteworthy benchmarks in current literature. The anatomy of an obstacle avoidance suite consists of three parts: tasks, environments, and performance metrics. A single benchmark session consists of an obstacle avoidance algorithm trying to complete a particular task in a specified environment. During this session, the benchmark suite continuously measures a set of metrics which are later used to generate a performance measurement.

With the literature study now complete, the next step in the entire process is the development of the obstacle avoidance suite.

References

Albaker, B. M., & Rahim, N. A. (2009). A survey of collision avoidance approaches for unmanned aerial vehicles. *2009 International Conference for Technical Postgraduates (TECHPOS)*. doi: 10.1109/techpos.2009.5412074

Baker, S., Scharstein, D., Lewis, J. P., Roth, S., Black, M. J., & Szeliski, R. (2007). A database and evaluation methodology for optical flow. In *Iccv* (p. 1-8). IEEE Computer Society. Retrieved from http://dblp.uni-trier.de/db/conf/iccv/iccv2007.html#BakerSLRBS07

Banks, J. (2010). *Discrete-event system similation*. Pearson/Prentice Hall.

Barry, A. J., Florence, P. R., & Tedrake, R. (2018). High-speed autonomous obstacle avoidance with pushbroom stereo. *Journal of Field Robotics*, *35*(1), 52-68. Retrieved from https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21741 doi: 10.1002/rob.21741

Bills, C., Chen, J., & Saxena, A. (2011). Autonomous mav flight in indoor environments using single image perspective cues. *2011 IEEE International Conference on Robotics and Automation*, 5776-5783.

Boroujerdian, B., Genc, H., Krishnan, S., Cui, W., Almeida, M., Mansoorshahi, K., … Reddi, V. (2019). Mavbench: Micro aerial vehicle benchmarking. *CoRR*, *abs/1905.06388*. Retrieved from http://arxiv.org/abs/1905.06388 doi: 1905.06388

de Croon, G. C. H. E., de Weerdt, E., De Wagter, C., Remes, B. D. W., & Ruijsink, R. (2012, April). The appearance variation cue for obstacle avoidance. *Trans. Rob.*, *28*(2), 529–534. Retrieved from https://doi.org/10.1109/TRO.2011.2170754 doi: 10.1109/TRO.2011.2170754

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017). CARLA: An open urban driving simulator. In *Proceedings of the 1st annual conference on robot learning* (pp. 1–16).

Epic Games. (n.d.). *Unreal engine.* Retrieved from https://www.unrealengine.com

Furrer, F., Burri, M., Achtelik, M., & Siegwart, R. (2016). Robot operating system (ros): The complete reference (volume 1). In A. Koubaa (Ed.), (pp. 595–625). Cham: Springer International Publishing. Retrieved from http://dx.doi.org/10.1007/978-3-319-26054-9_23 doi: 10.1007/978-3-319-26054-9_23

Geiger, A., Lenz, P., & Urtasun, R. (2014). Are we ready for autonomous driving?the kitti vision benchmark suite. Retrieved from http://www.cvlibs.net/publications/Geiger2012CVPR.pdf

Guerra, W., Tal, E., Murali, V., Ryou, G., & Karaman, S. (2019). Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality. *CoRR*, *abs/1905.11377*. Retrieved from http://arxiv.org/abs/1905.11377

Hamzah, R. A., & Ibrahim, H. (2016). Literature survey on stereo vision disparity map algorithms. *J. Sensors*, *2016*, 8742920:1-8742920:23.

Hartley, R., & Zisserman, A. (2003). *Multiple view geometry in computer vision* (2nd ed.). USA:

Cambridge University Press.

Horn, B. K. P., & Schunck, B. G. (1981). Determining optical flow. *ARTIFICAL INTELLIGENCE*, *17*, 185–203. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.66.562

Lampe, A., & Chatila, R. (2006, 06). Performance measure for the evaluation of mobile robot autonomy. In (p. 4057 - 4062). doi: 10.1109/ROBOT.2006.1642325

Liu, S., Watterson, M., Tang, S., & Kumar, V. (2016). High speed navigation for quadrotors with limited onboard sensing. In *2016 ieee international conference on robotics and automation (icra)* (p. 1484-1491). Retrieved from https://app.dimensions.ai/details/publication/pub.1094764430 doi: 10.1109/icra.2016.7487284

Lucas, B. D., & Kanade, T. (1981). An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on artificial intelligence - volume 2* (pp. 674–679). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1623264.1623280

Marlow, S., & Langelaan, J. (2011, 4 1). Local terrain mapping for obstacle avoidance using monocular vision. *Journal of the American Helicopter Society*, *56*(2). doi: 10.4050/JAHS.56.022007

Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., & von Stryk, O. (2012). Comprehensive simulation of quadrotor uavs using ros and gazebo. In *3rd int. conf. on simulation, modeling and programming for autonomous robots (simpar)* (p. to appear).

Mori, T., & Scherer, S. (2013, May). First results in detecting and avoiding frontal obstacles from a monocular camera for micro unmanned aerial vehicles. In *Proceedings of international conference on robotics and automation* (p. 1750 – 1757).

Nègre, A., Braillon, C., Crowley, J. L., & Laugier, C. (2006). Real-time time-to-collision from variation of intrinsic scale. In *Experimental robotics, the 10th international symposium on experimental robotics [ISER '06, july 6-10, 2006, rio de janeiro, brazil]* (pp. 75–84). Retrieved from https://doi.org/10.1007/978-3-540-77457-0_8 doi: 10.1007/978-3-540-77457-0_8

Nous, C., Meertens, R., de Wagter, C., & de Croon, G. (2016). Performance evaluation in obstacle avoidance. *IEEE International Conference on Intelligent Robots and Systems (IROS)*. Retrieved from https://ieeexplore.ieee.org/document/7759532 doi: 10.1109/IROS.2016.7759532

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., … Ng, A. Y. (2009). Ros: an open-source robot operating system. In *Icra workshop on open source software.*

Sebesta, K., & Baillieul, J. (2012). Animal-inspired agile flight using optical flow sensing. *CoRR*, *abs/1203.2816*. Retrieved from http://arxiv.org/abs/1203.2816

Shah, S., Dey, D., Lovett, C., & Kapoor, A. (2017). *Airsim: High-fidelity visual and physical simulation for autonomous vehicles.* Retrieved from https://arxiv.org/abs/1705.05065

Studios, J. (2019). *Unreal engine - late summer wild meadow*. Retrieved from https://www.artstation.com/artwork/nQW401

Tezenas du Montcel, T., Negre, A., Gomez-Balderas, J.-E., & Marchand, N. (2019, May). *BOARR : A Benchmark for quadrotor Obstacle Avoidance based on ROS and RotorS.* Retrieved from https://hal.archives-ouvertes.fr/hal-02142571 (working paper or preprint)

Wagter, C. D., Tijmons, S., Remes, B. D. W., & de Croon, G. C. H. E. (2014). Autonomous flight of a 20-gram flapping wing mav with a 4-gram onboard stereo vision system. *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 4982-4987.

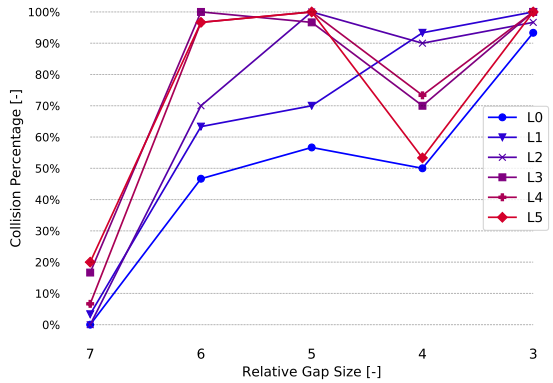
Weisz, J., Huang, Y., Lier, F., Sethumadhavan, S., & Allen, P. K. (2016). Robobench: Towards sustainable robotics system benchmarking. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 3383-3389.

III

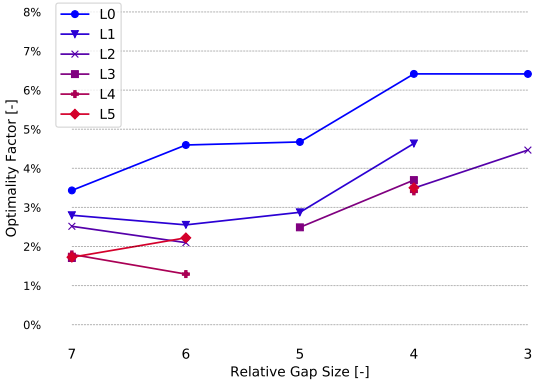
Appendix

A

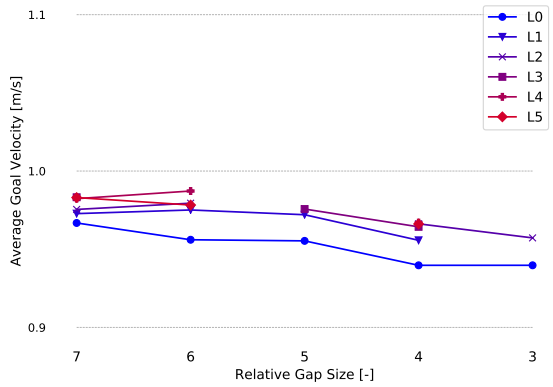
Additional Results



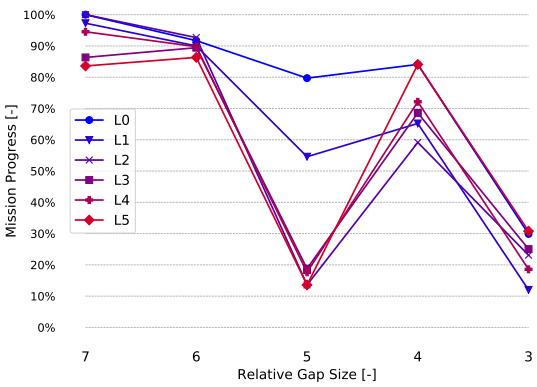
(a) Collision Percentage results for different Relative Gap Sizes and obstacle sets.



(b) Optimality Factor results for different Relative Gap Sizes and obstacle sets.

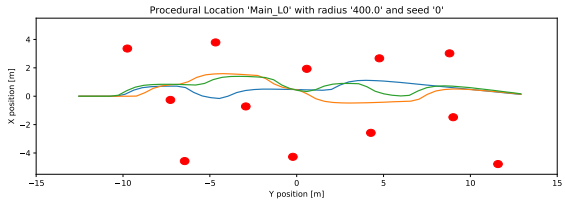


(c) Average Goal Velocity for results for different Relative Gap Sizes and obstacle sets.

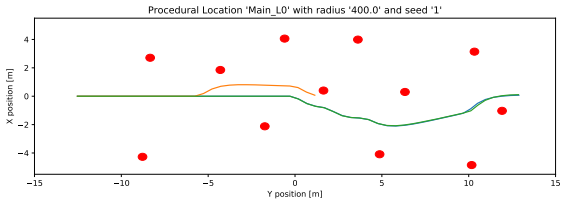


(d) Mission Progress results for different Relative Gap Sizes and obstacle sets.

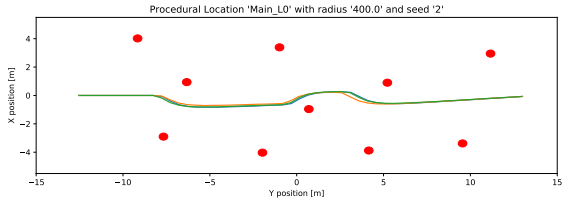
Figure A.1: Performance metric graphs generated from the texture benchmark when the **seed remains unchanged**. These graphs show the importance of why generating multiple obstacle fields for the same Poisson Radius is so important.



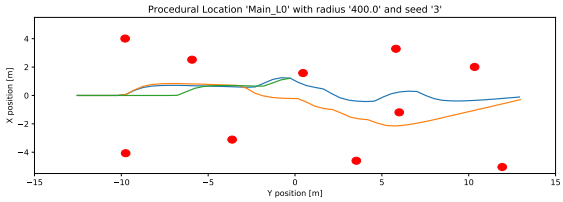
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



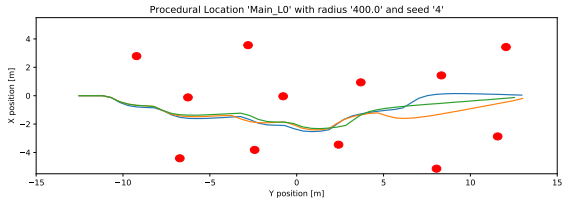
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



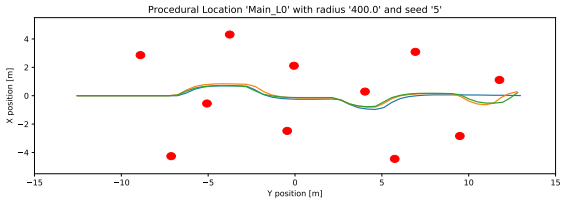
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



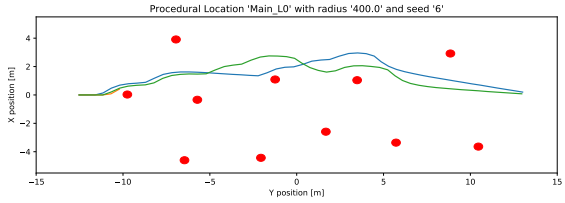
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



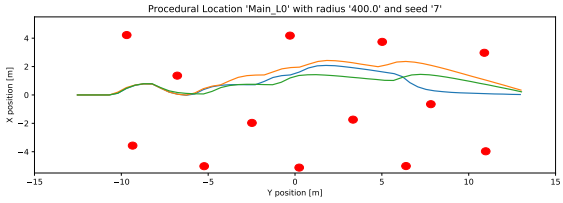
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



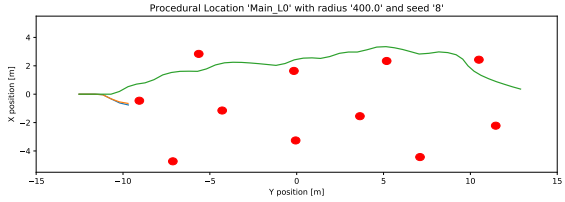
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



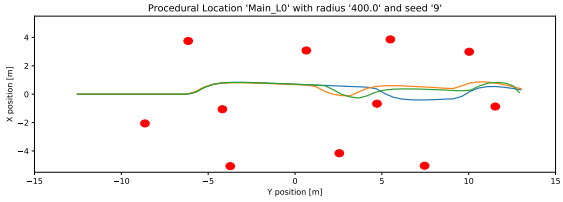
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

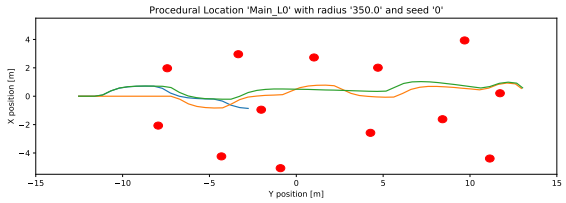


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

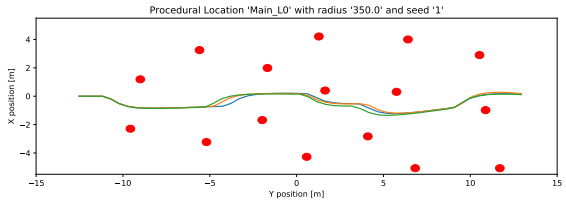


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

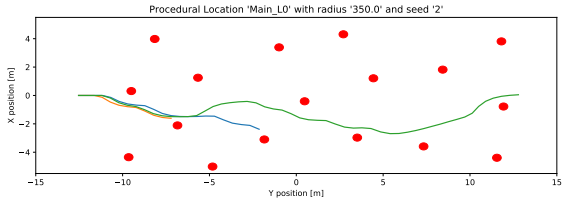
Figure A.2: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 4m, performed on obstacle set L0.



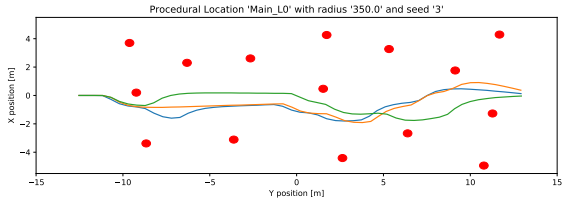
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



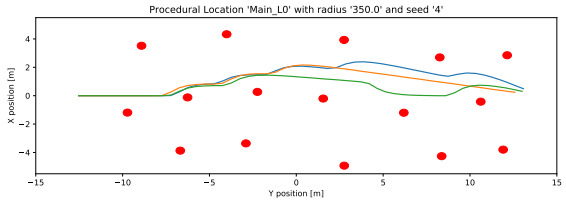
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



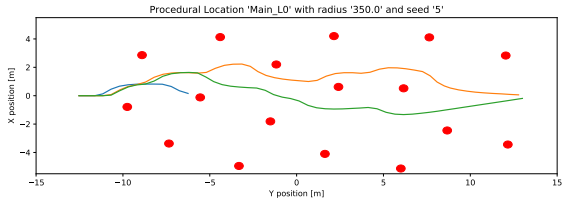
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



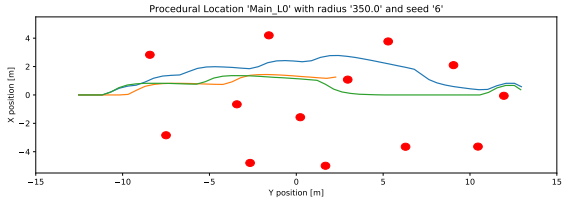
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



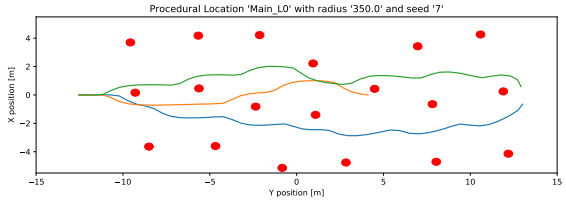
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



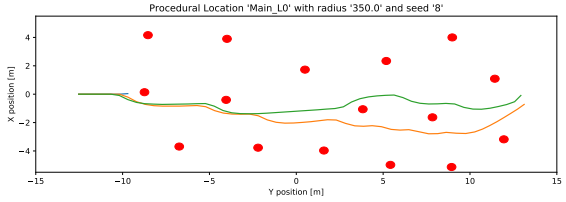
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



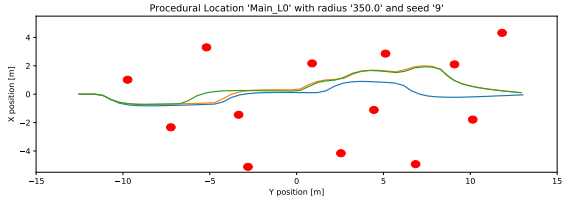
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

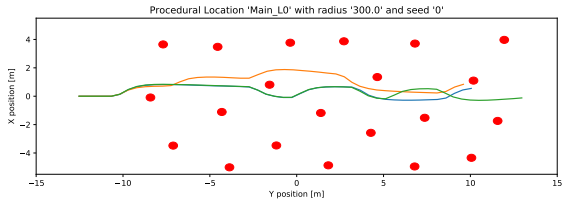


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

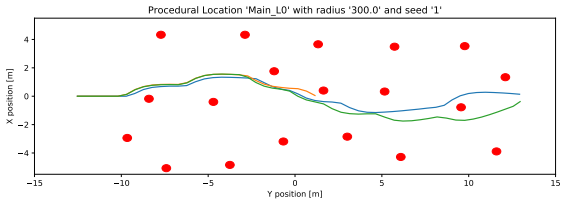


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

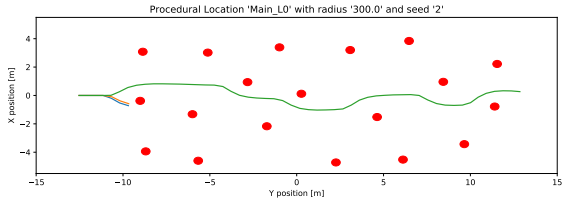
Figure A.3: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3.5m, performed on obstacle set L0.



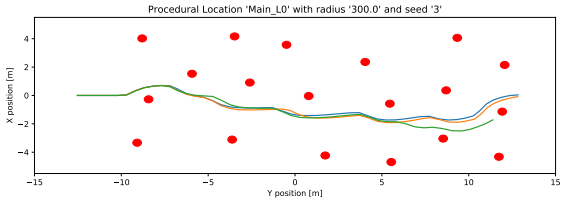
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



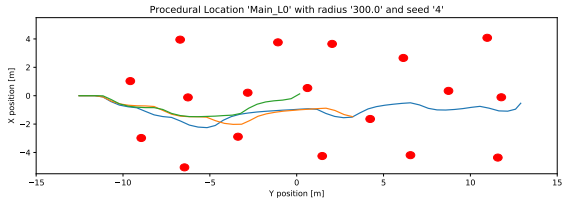
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



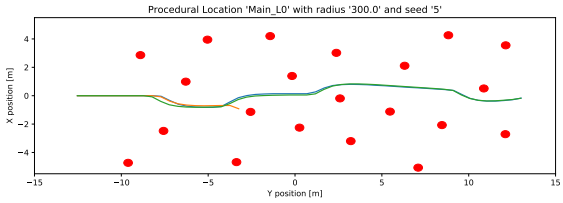
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



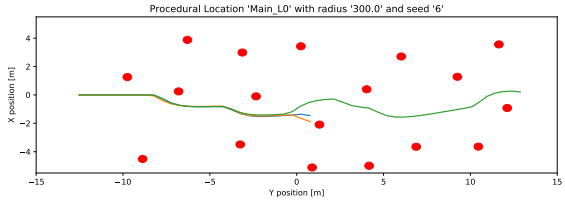
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



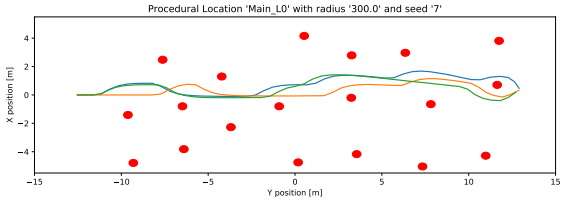
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



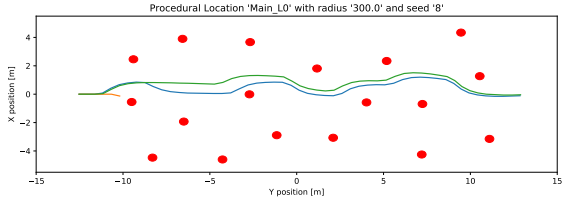
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



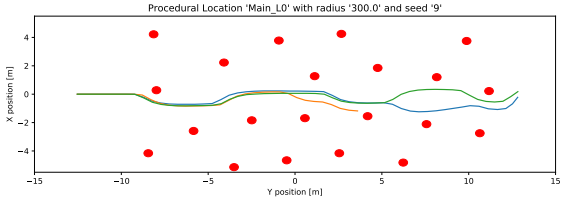
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

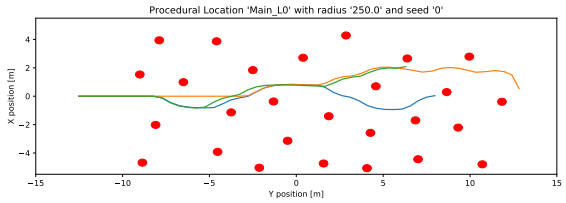


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

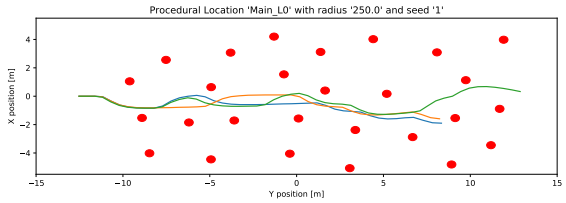


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

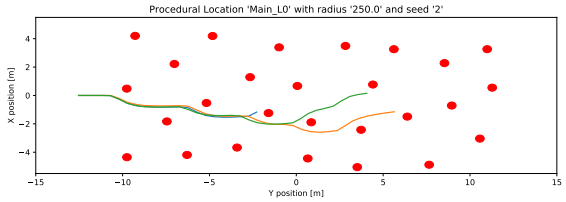
Figure A.4: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3m, performed on obstacle set L0.



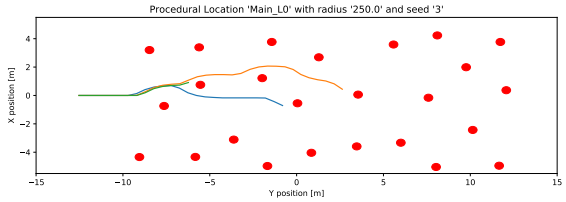
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



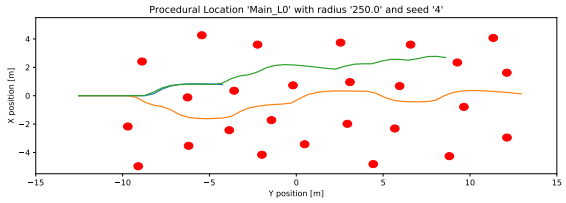
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



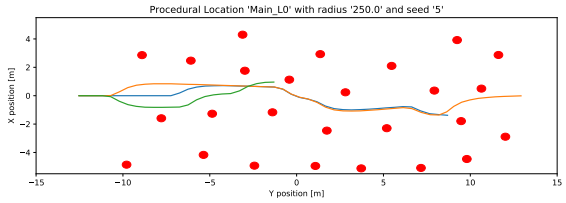
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



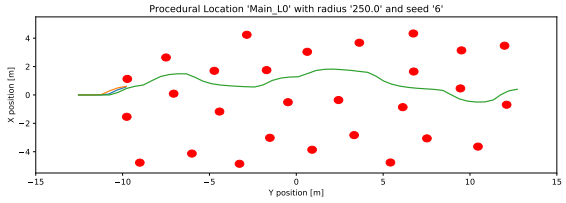
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



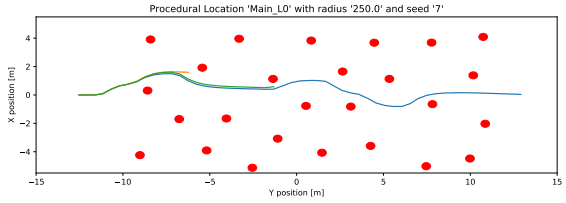
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



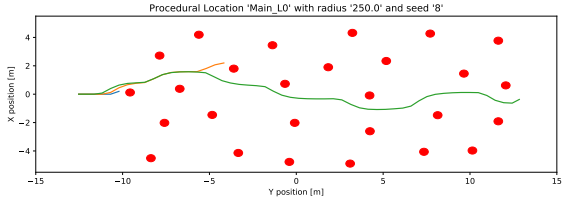
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



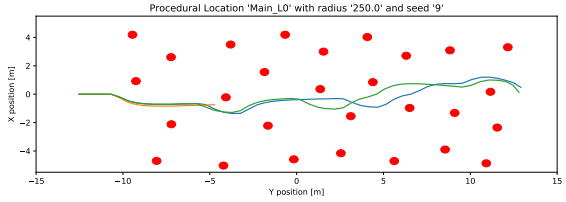
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

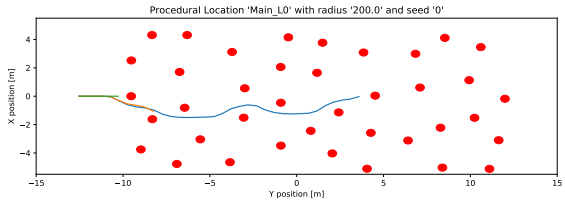


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

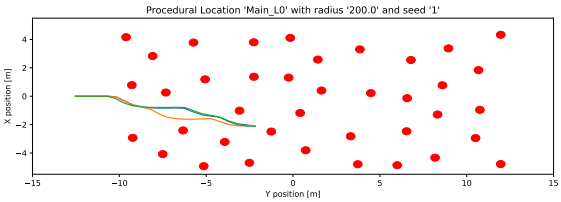


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

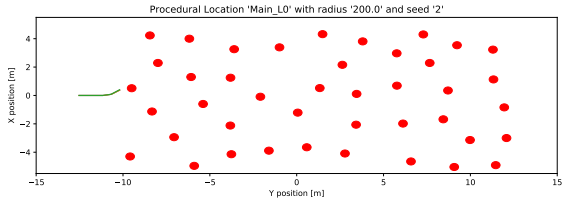
Figure A.5: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2.5m, performed on obstacle set L0.



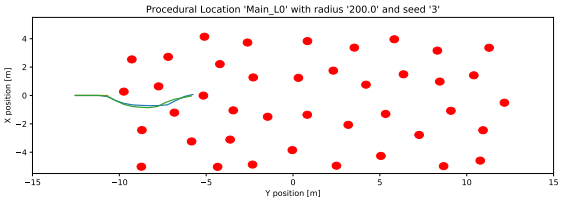
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



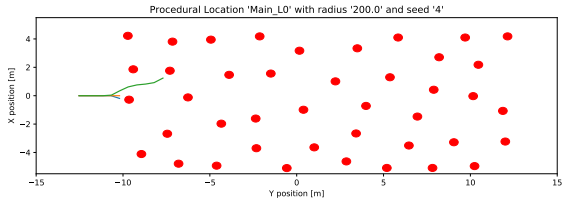
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



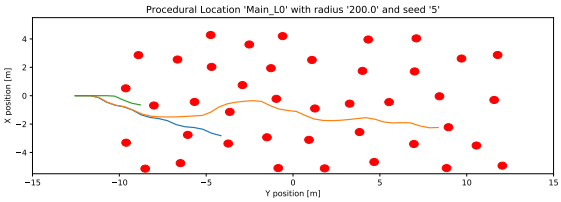
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



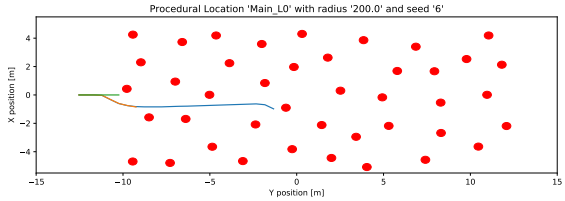
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



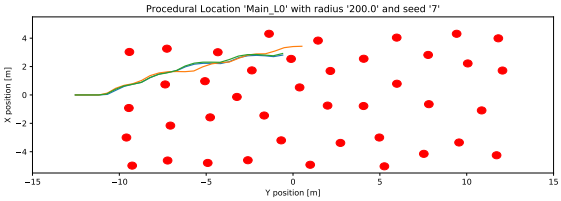
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



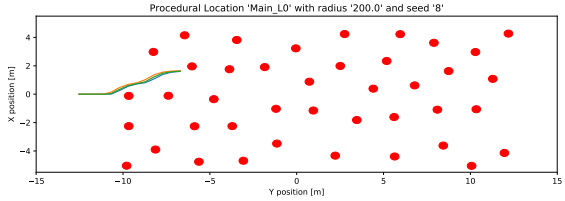
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



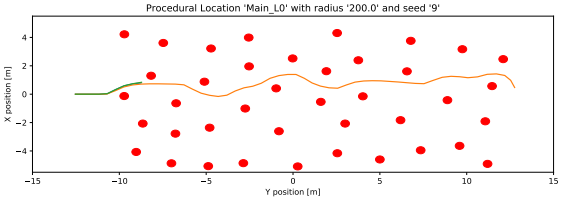
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

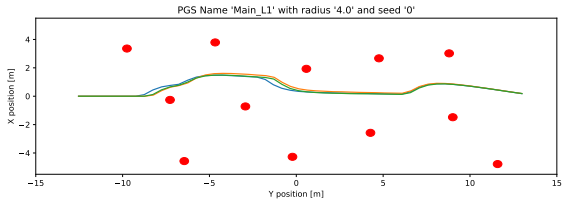


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

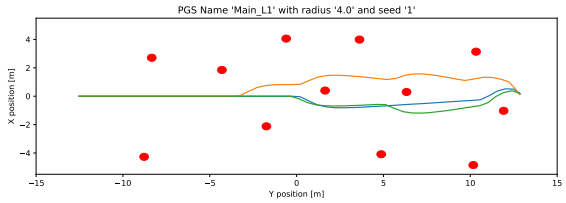


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

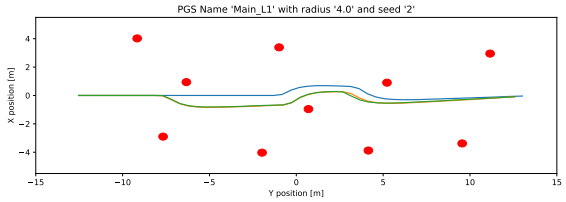
Figure A.6: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2m, performed on obstacle set L0.



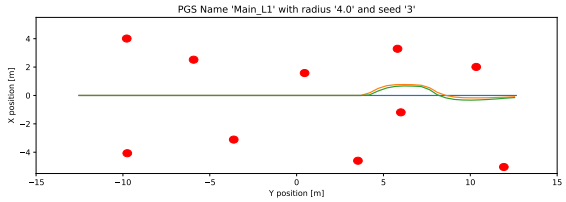
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



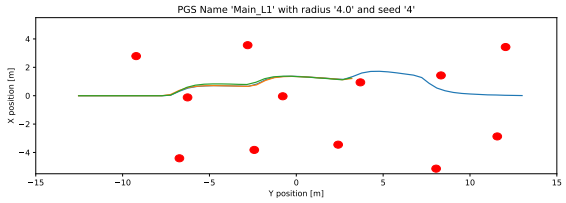
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



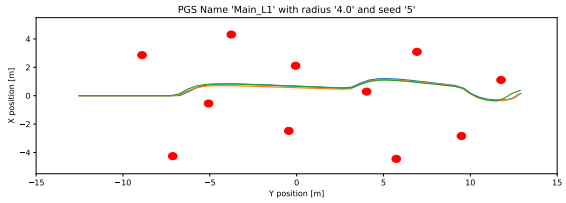
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



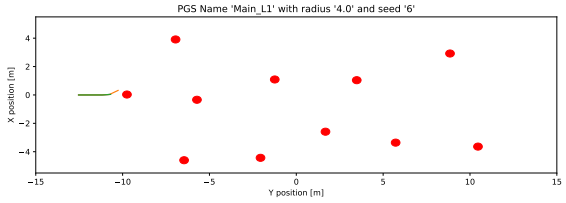
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



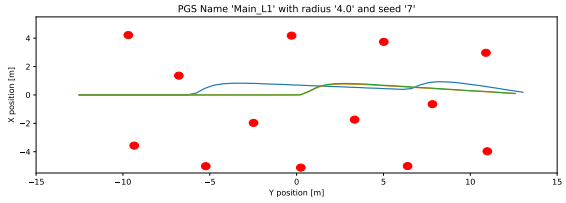
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



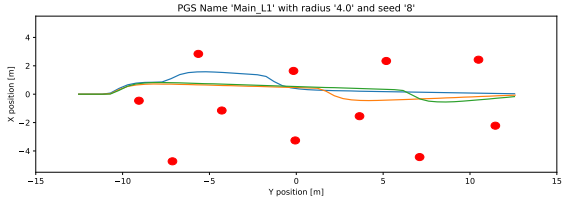
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



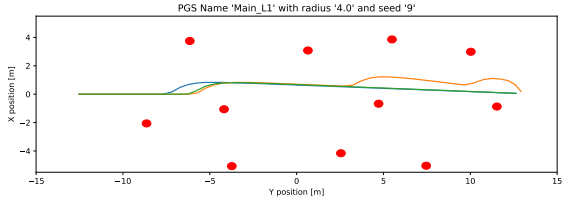
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

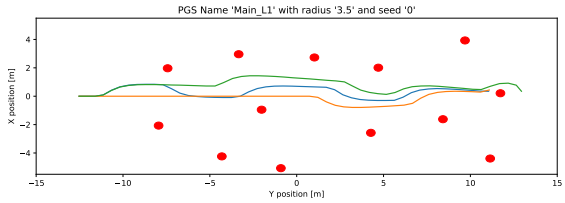


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

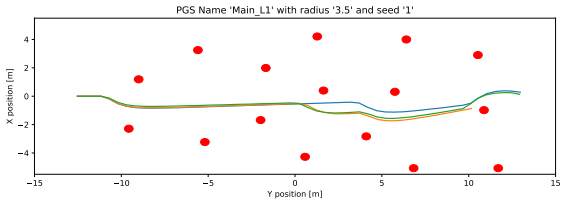


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

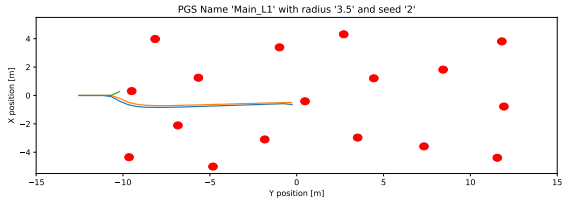
Figure A.7: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 4m, performed on obstacle set L1.



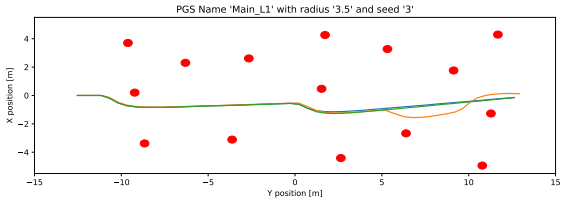
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



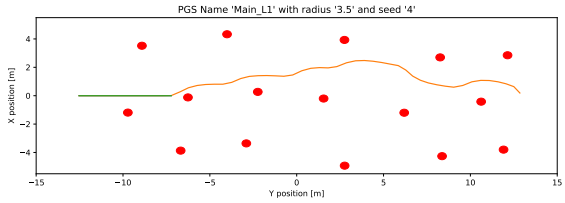
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



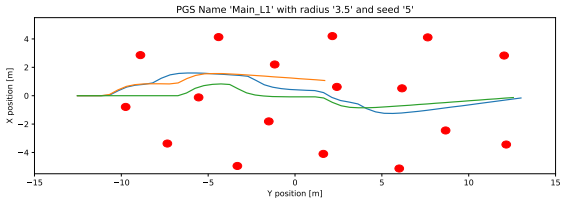
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



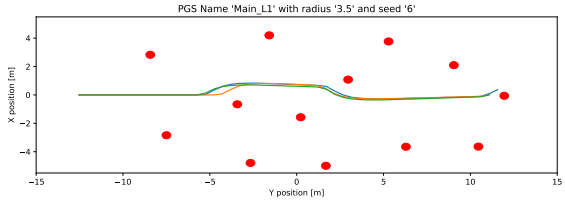
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



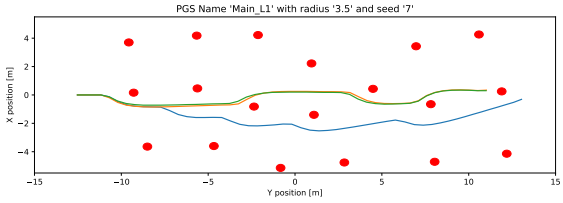
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



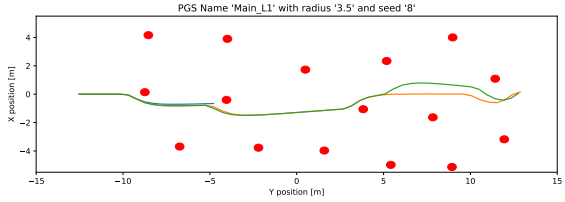
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



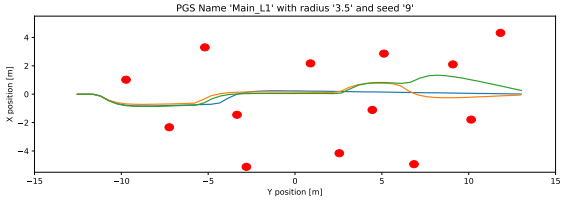
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

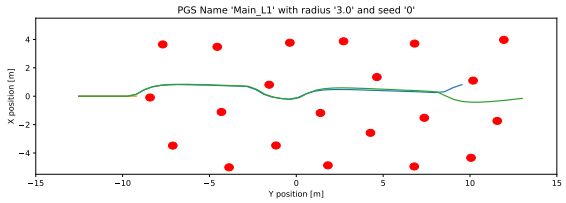


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

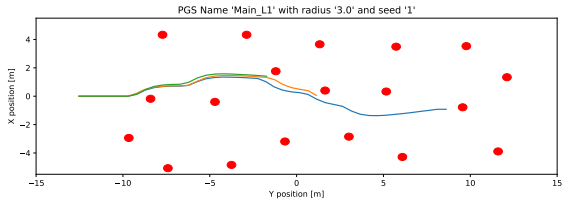


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

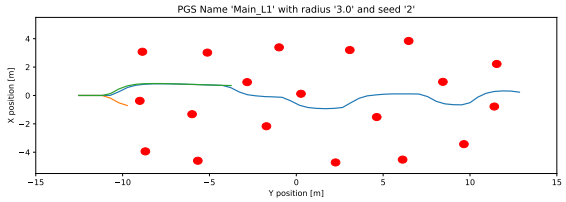
Figure A.8: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3.5m, performed on obstacle set L1.



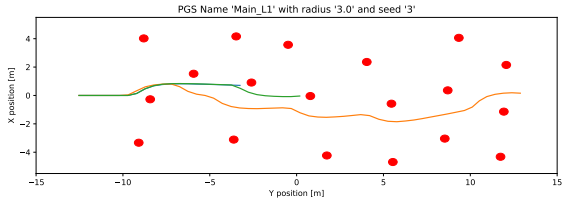
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



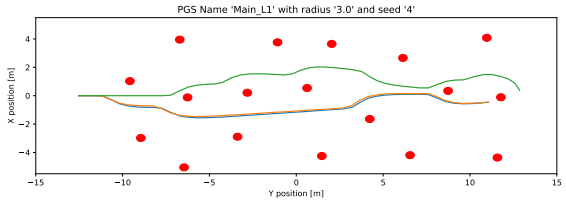
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



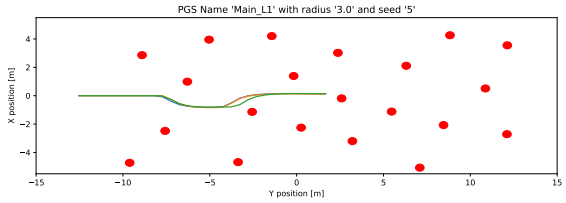
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



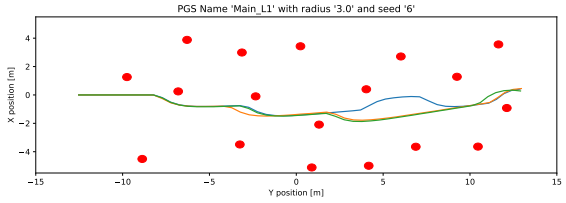
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



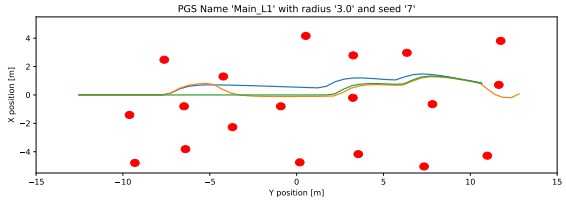
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



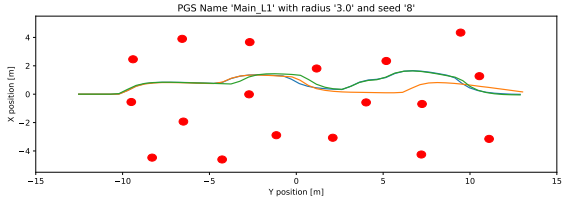
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



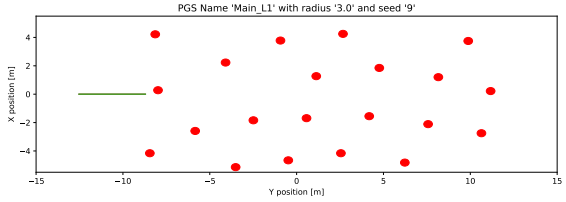
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

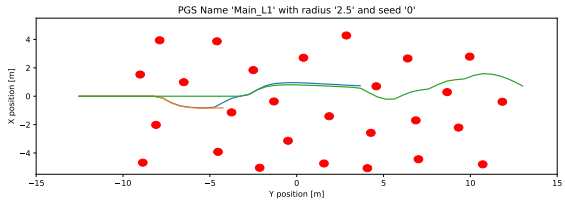


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

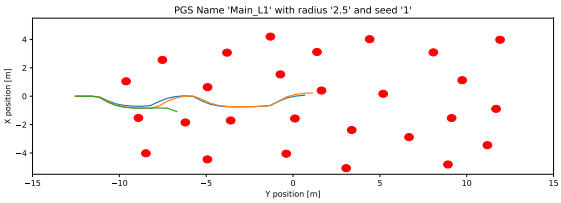


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

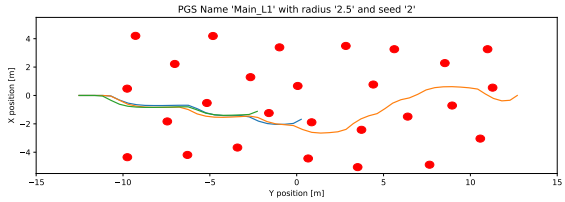
Figure A.9: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3m, performed on obstacle set L1.



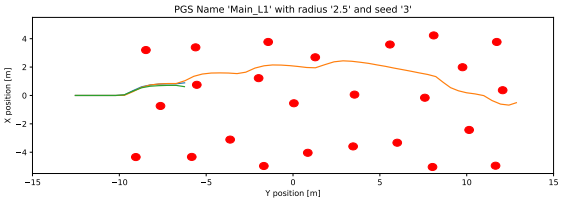
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



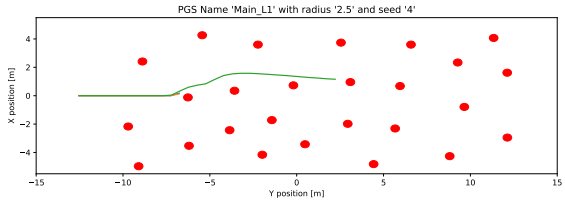
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



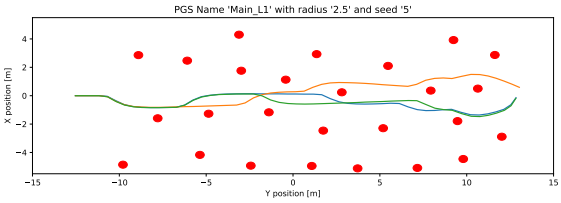
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



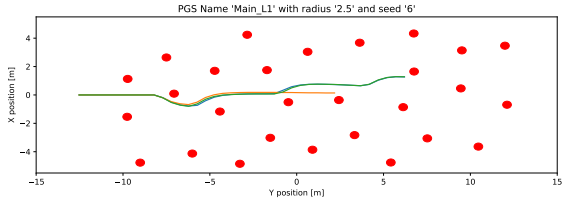
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



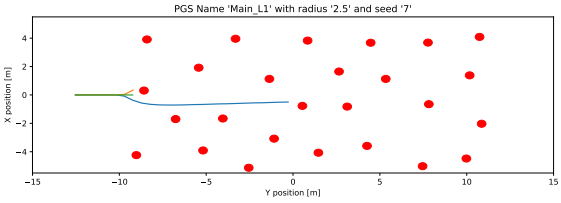
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



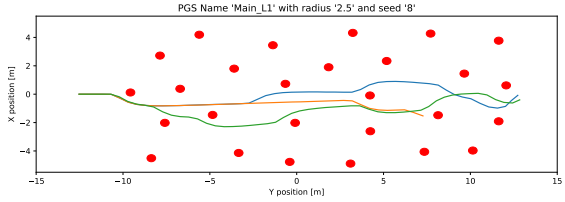
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



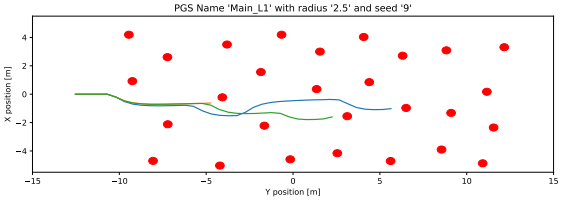
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

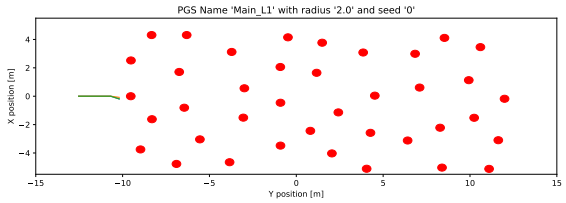


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

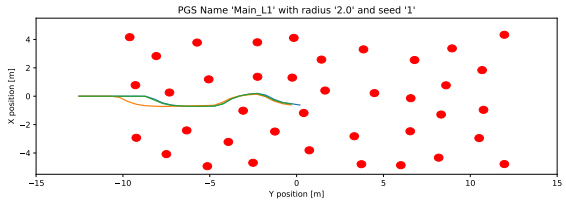


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

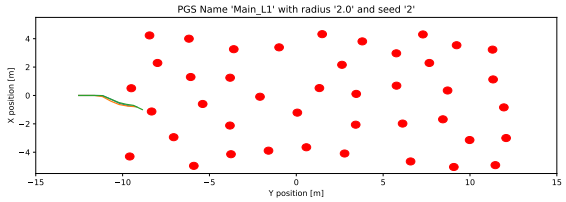
Figure A.10: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2.5m, performed on obstacle set L1.



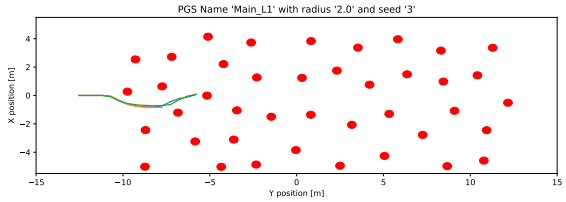
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



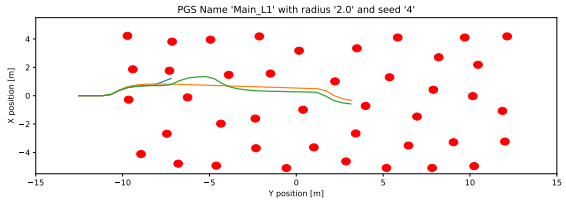
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



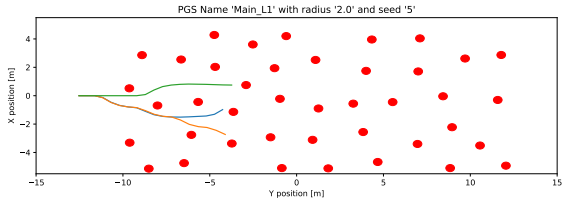
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



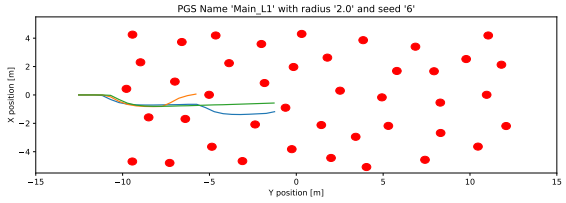
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



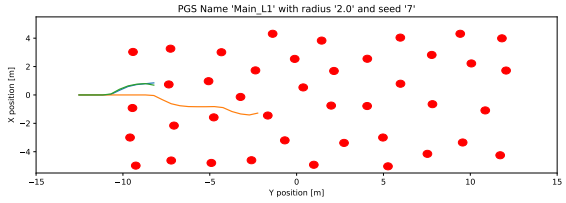
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



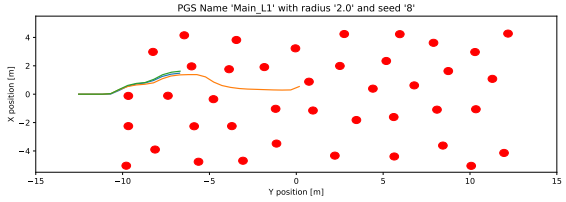
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



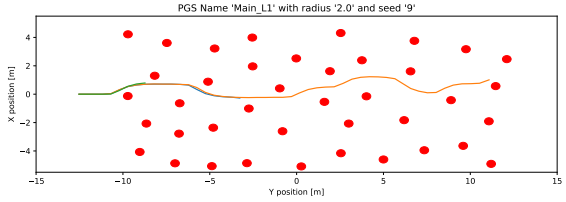
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

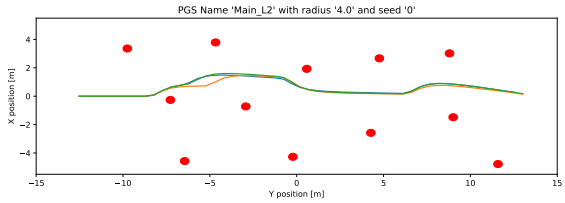


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

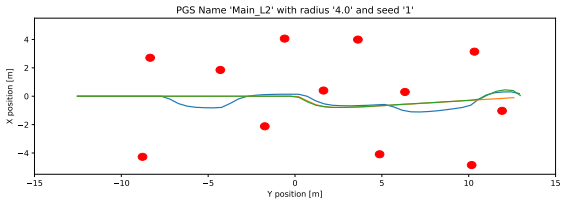


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

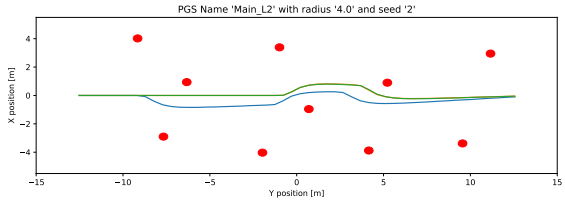
Figure A.11: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2m, performed on obstacle set L1.



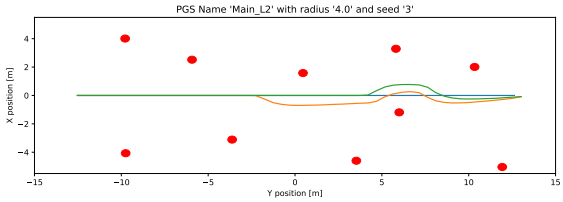
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



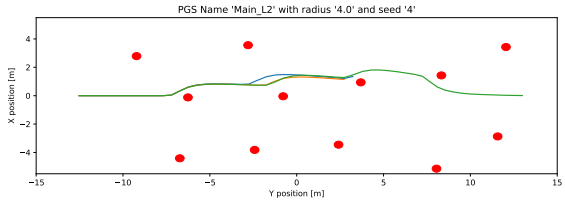
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



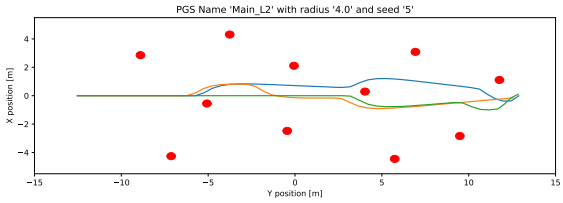
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



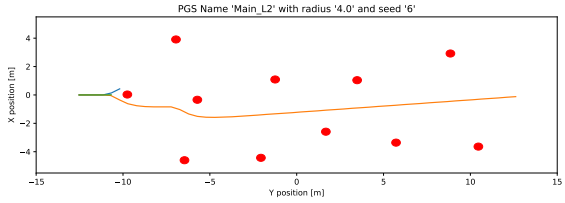
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



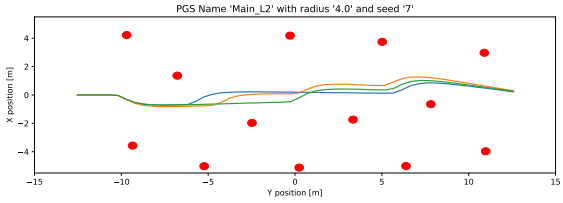
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



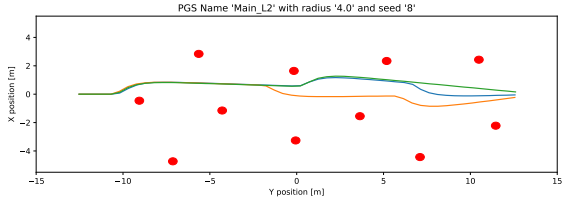
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



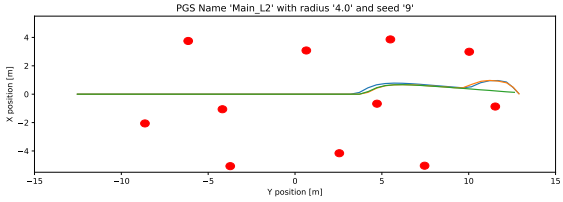
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

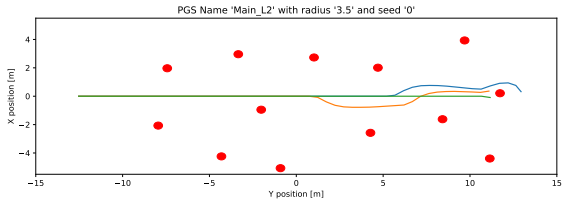


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

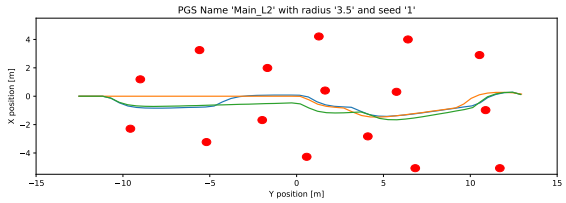


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

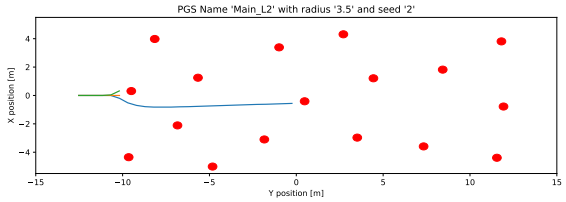
Figure A.12: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 4m, performed on obstacle set L2.



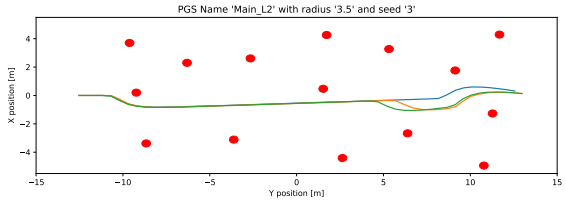
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



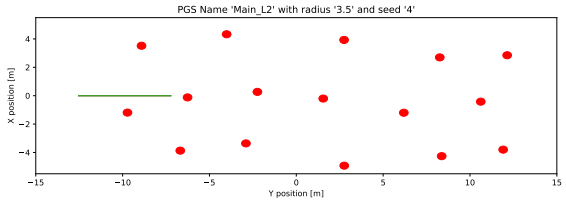
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



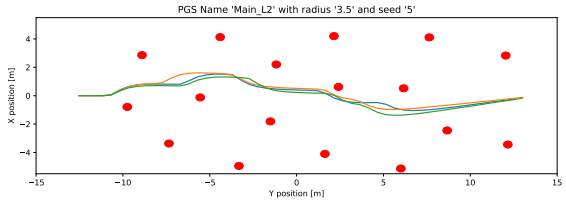
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



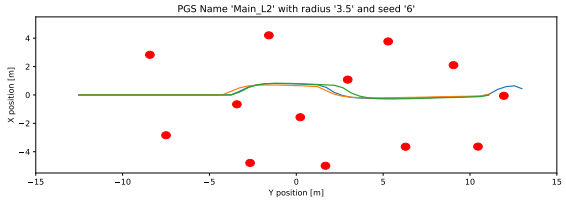
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



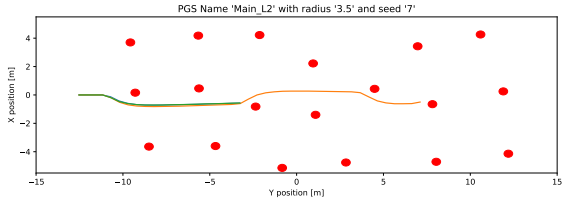
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



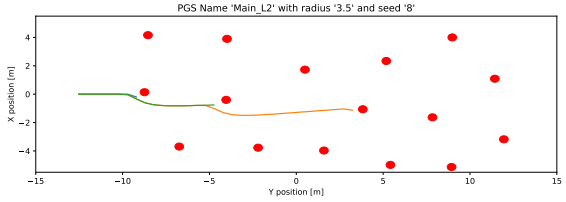
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



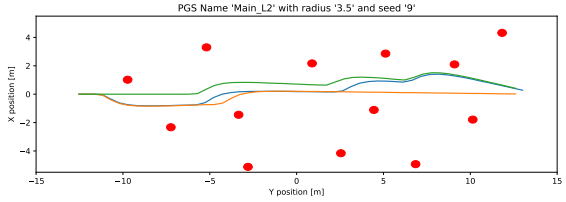
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

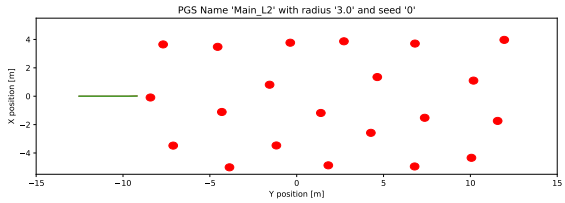


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

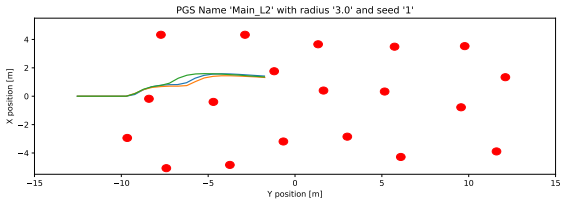


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

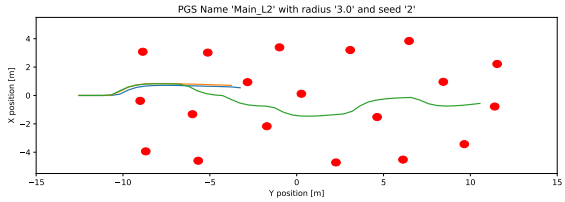
Figure A.13: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3.5m, performed on obstacle set L2.



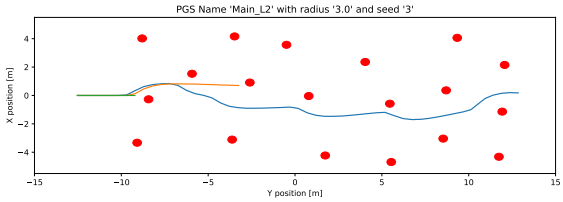
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



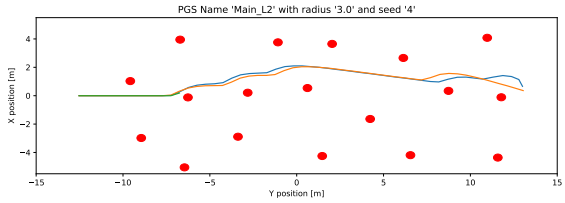
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



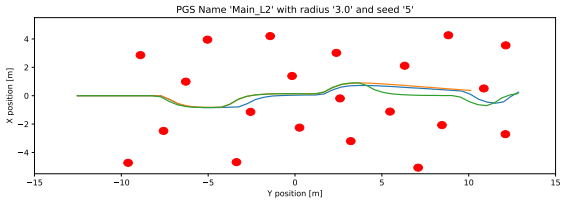
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



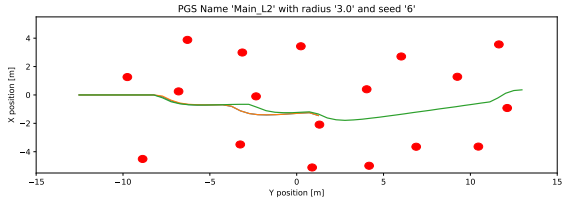
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



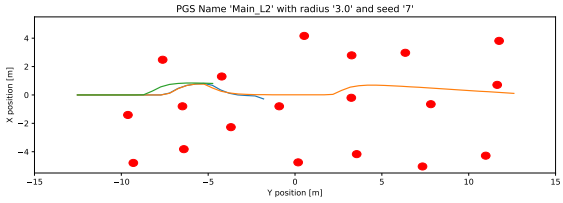
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



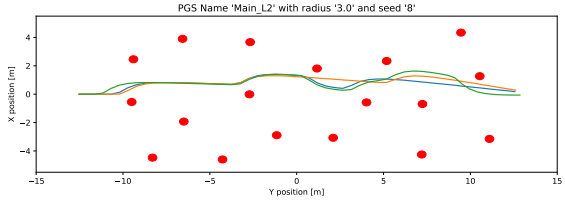
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



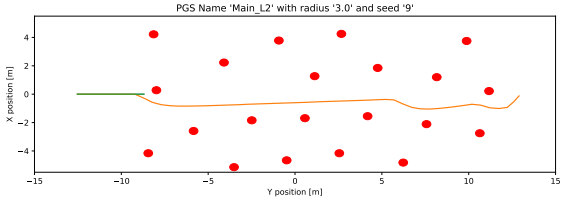
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

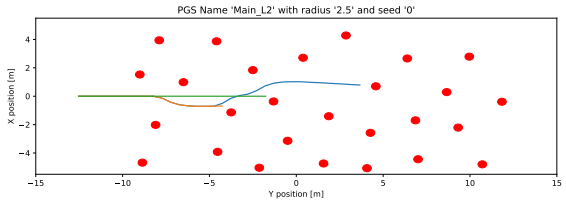


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

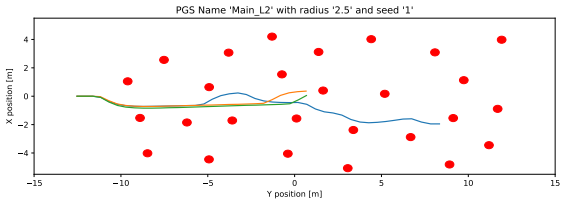


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

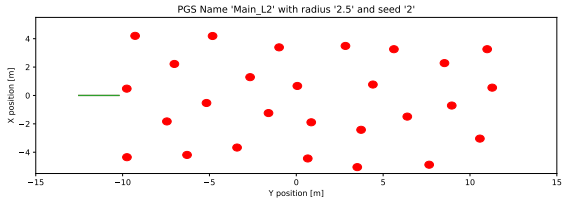
Figure A.14: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3m, performed on obstacle set L2.



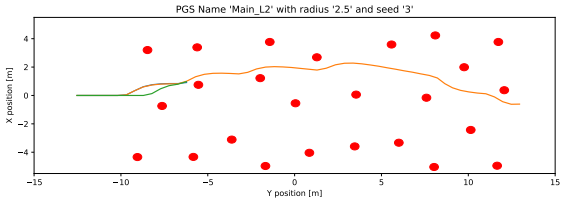
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



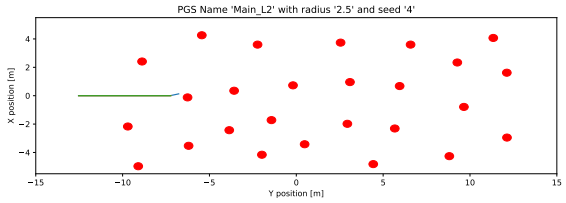
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



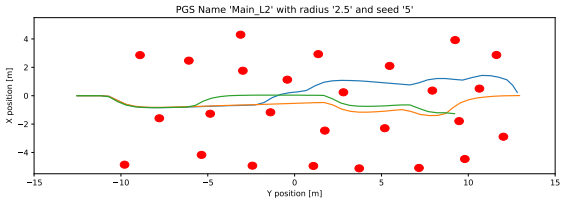
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



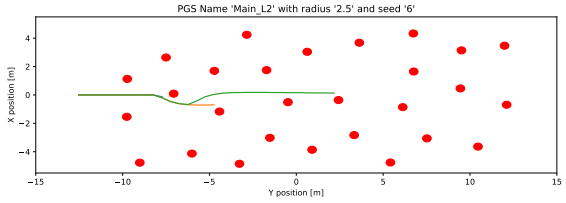
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



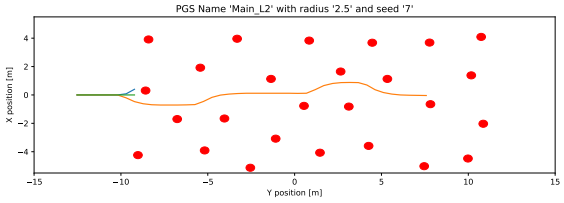
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



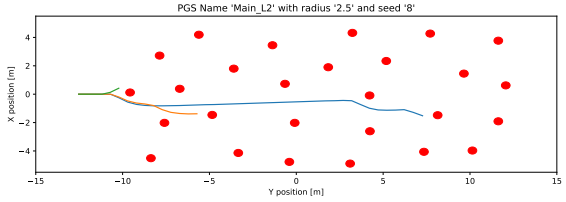
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



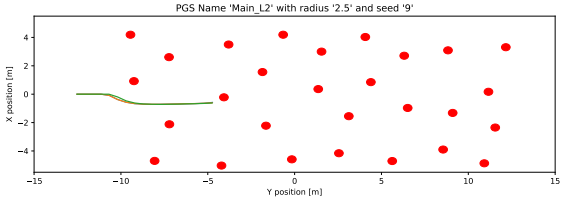
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

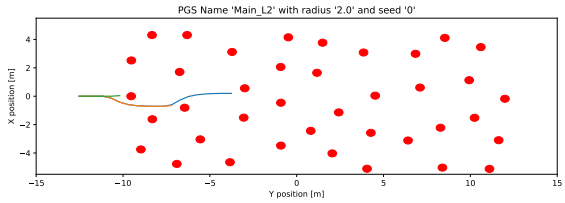


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

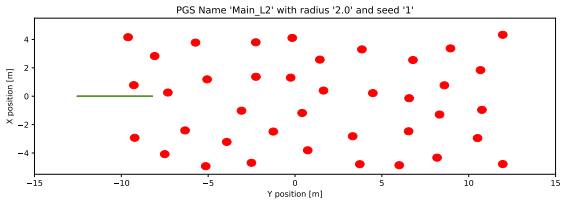


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

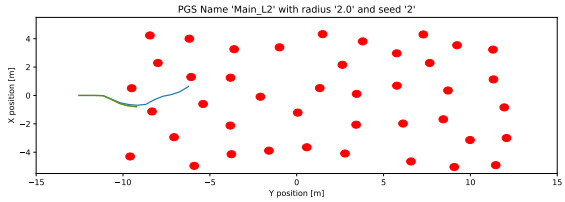
Figure A.15: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2.5m, performed on obstacle set L2.



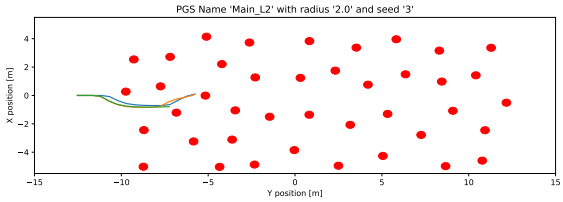
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



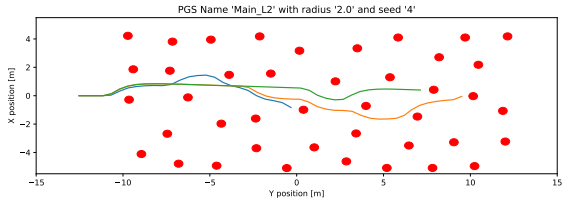
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



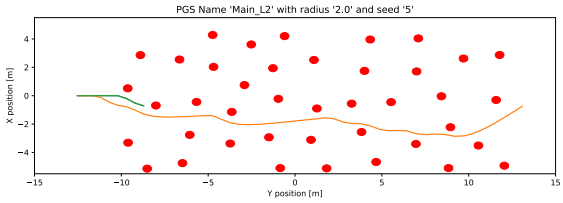
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



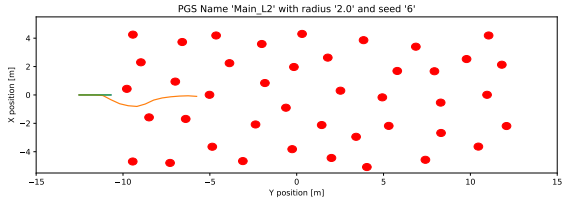
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



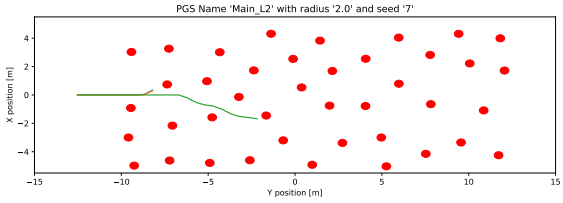
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



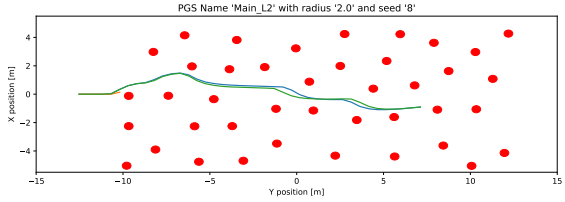
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



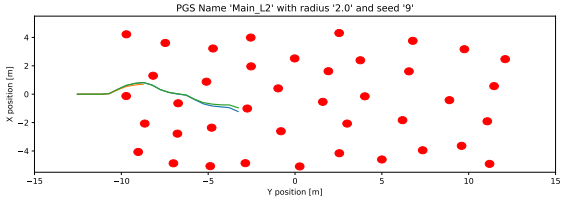
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

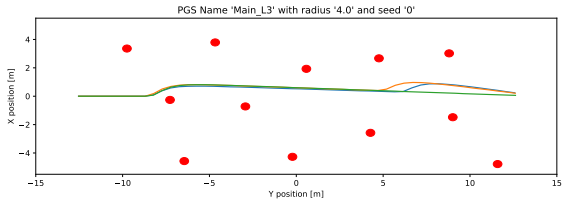


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

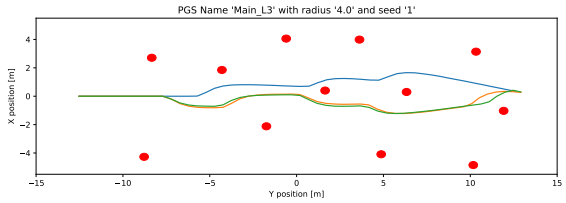


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

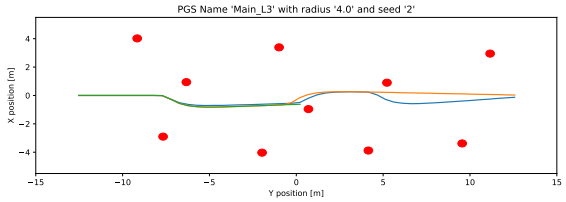
Figure A.16: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2m, performed on obstacle set L2.



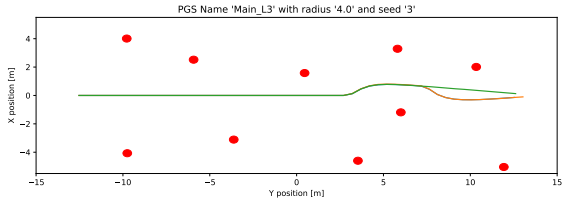
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



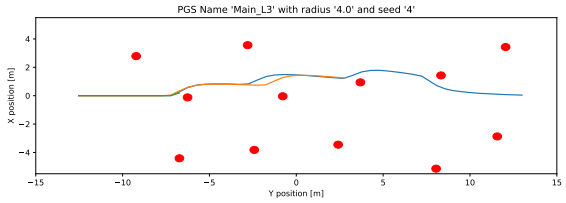
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



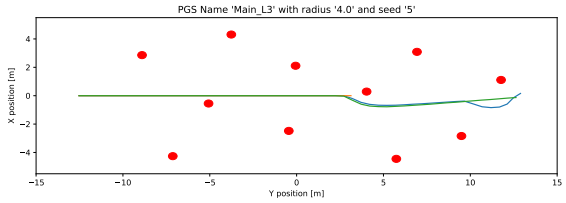
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



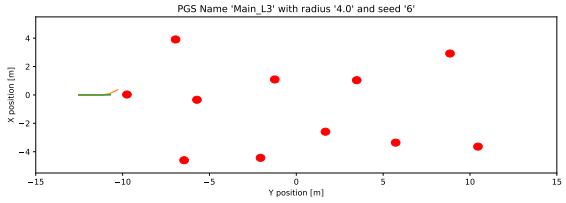
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



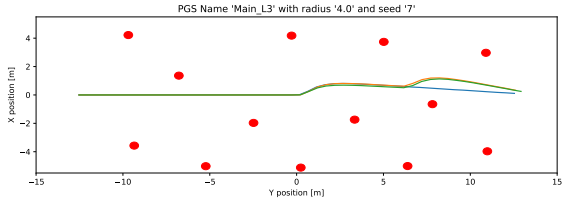
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



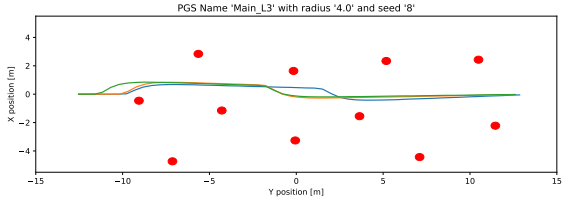
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



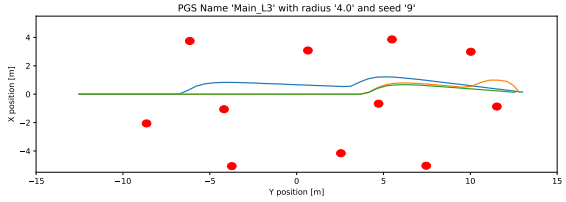
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

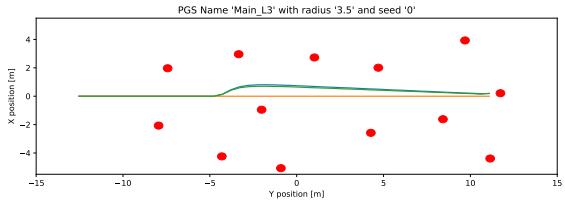


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

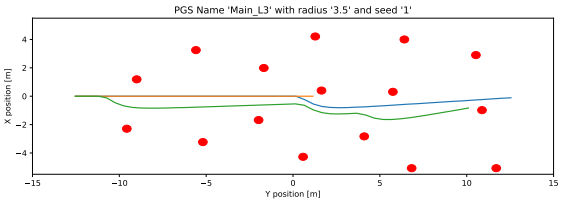


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

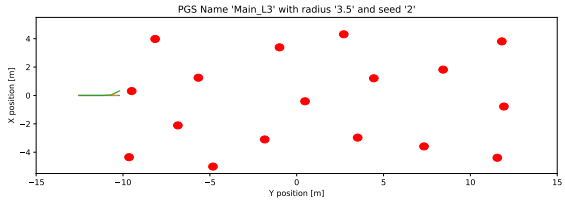
Figure A.17: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 4m, performed on obstacle set L3.



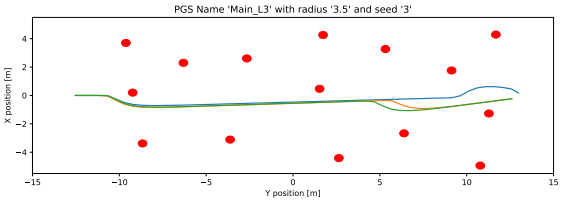
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



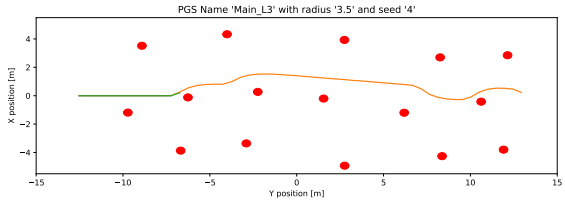
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



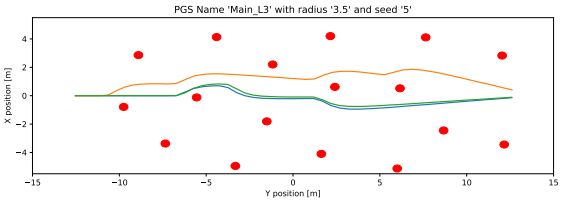
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



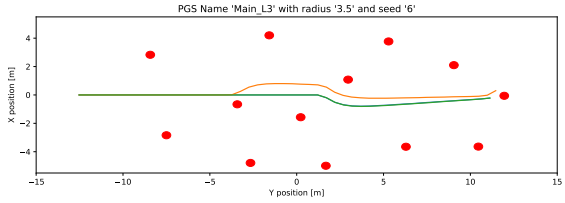
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



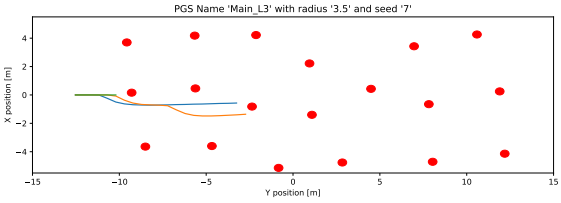
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



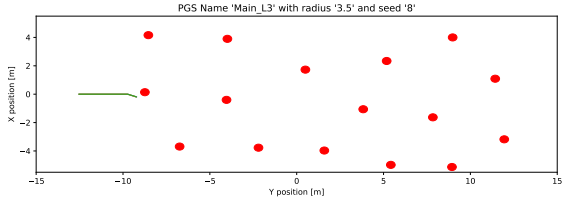
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



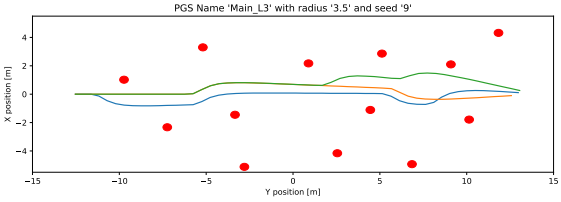
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

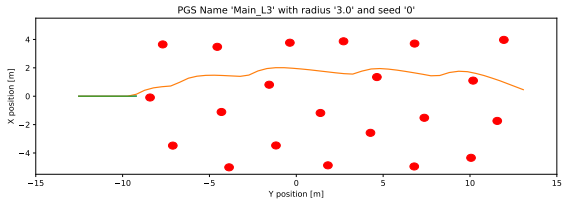


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

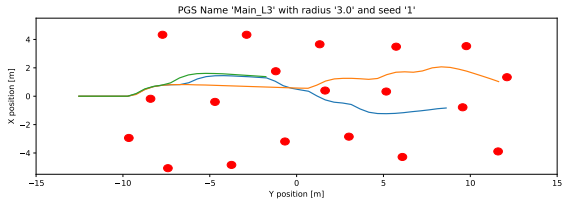


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

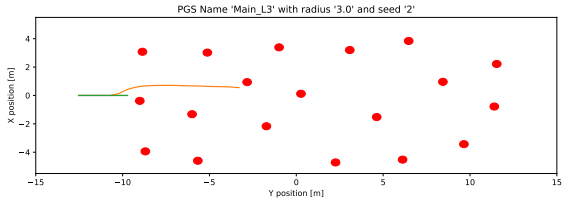
Figure A.18: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3.5m, performed on obstacle set L3.



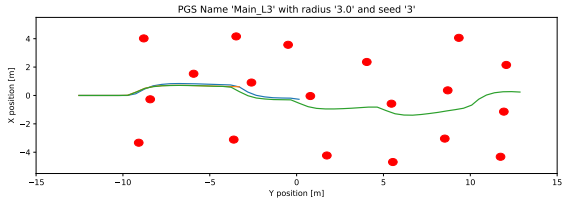
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



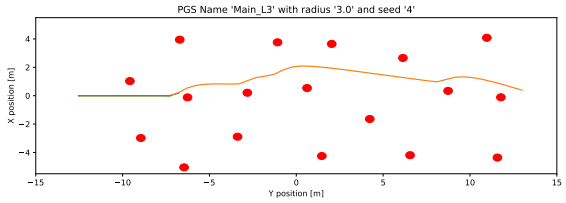
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



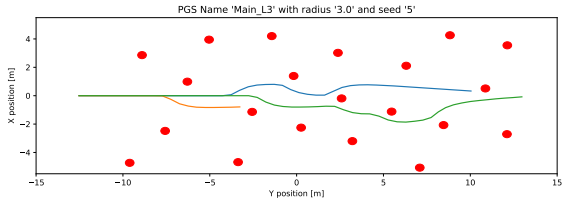
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



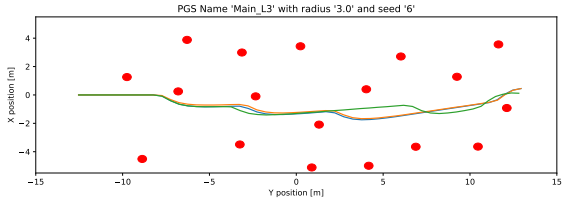
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



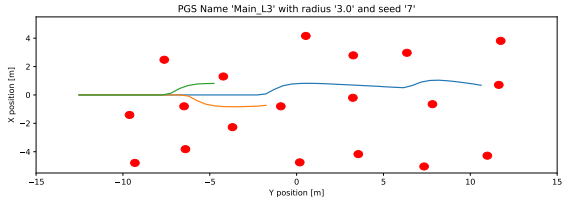
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



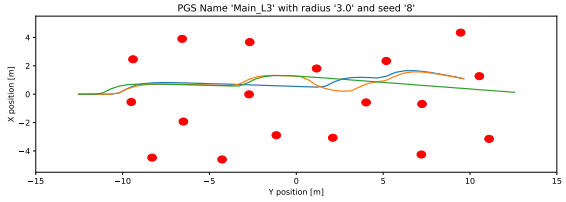
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



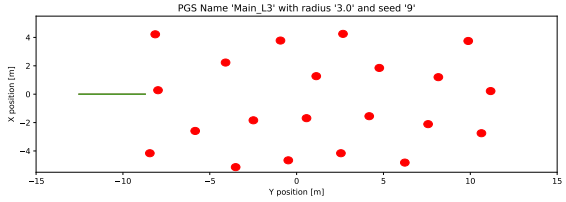
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

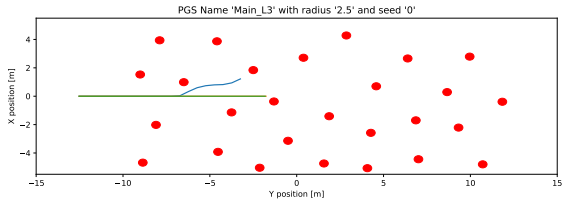


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

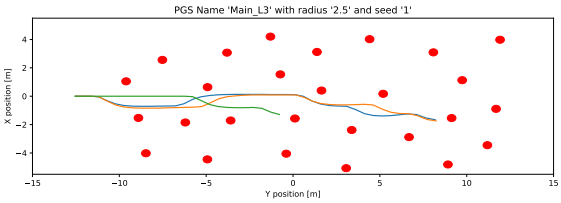


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

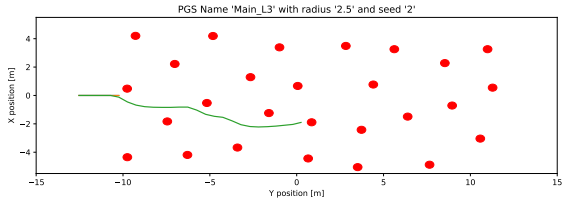
Figure A.19: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3m, performed on obstacle set L3.



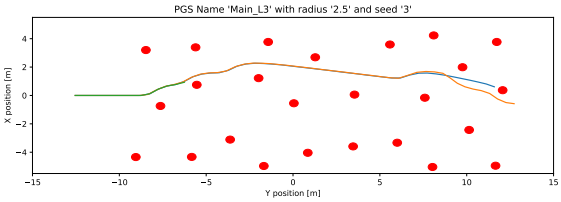
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



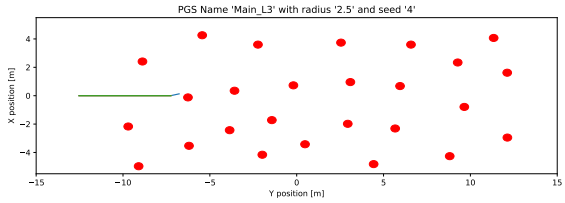
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



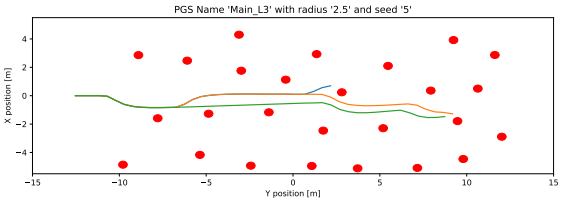
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



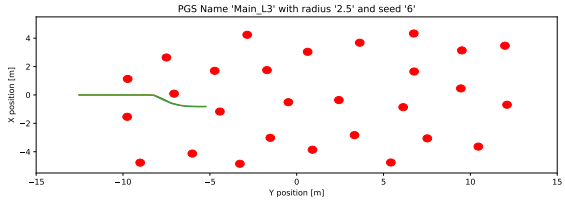
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



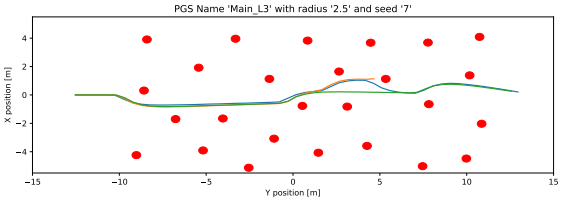
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



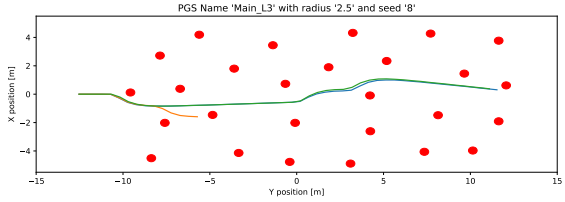
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



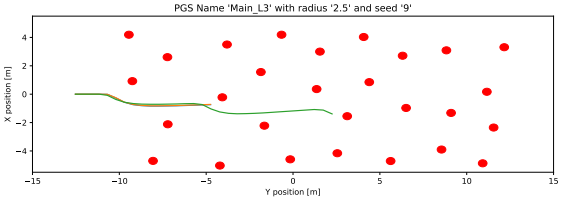
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

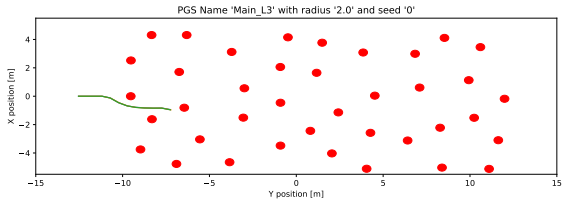


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

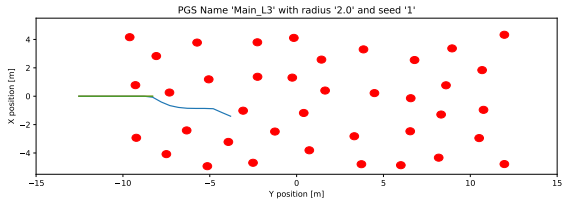


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

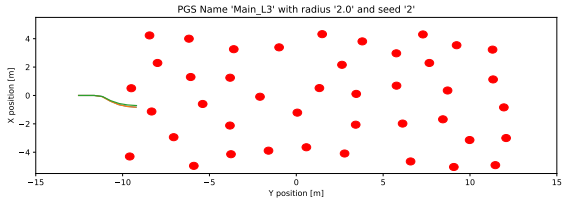
Figure A.20: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2.5m, performed on obstacle set L3.



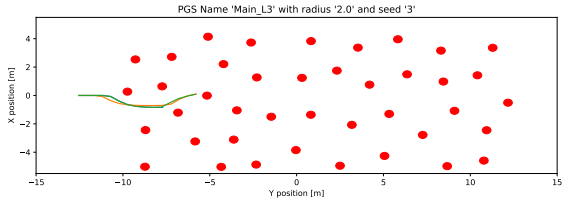
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



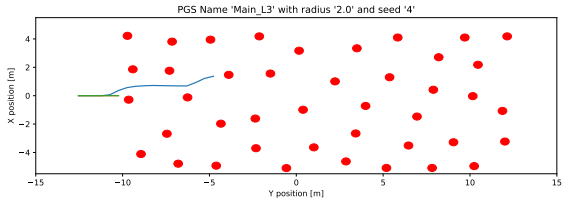
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



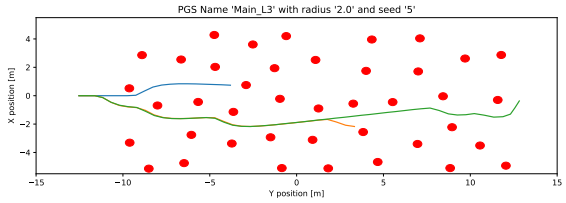
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



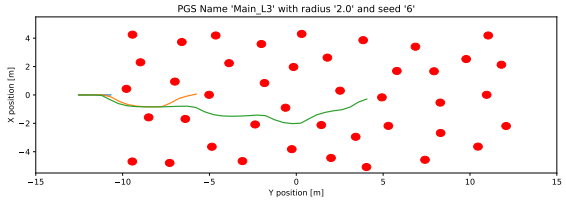
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



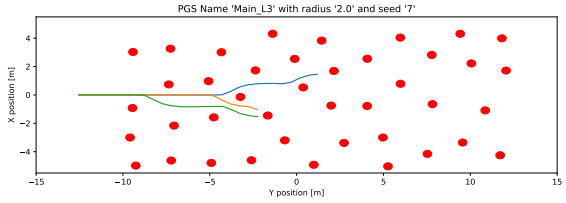
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



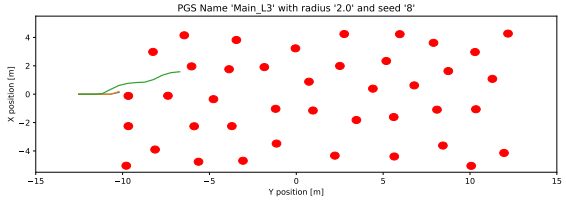
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



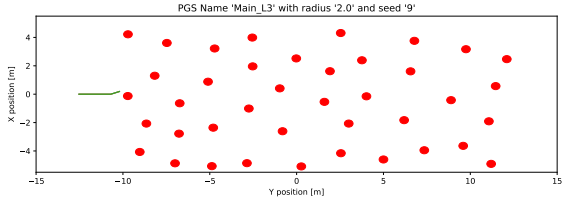
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

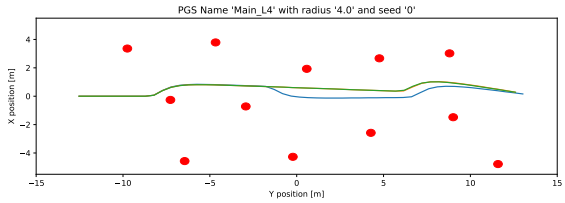


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

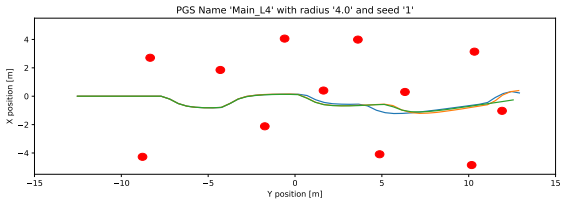


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

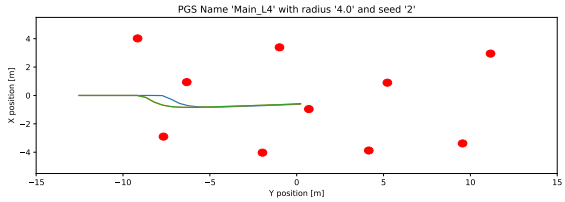
Figure A.21: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2m, performed on obstacle set L3.



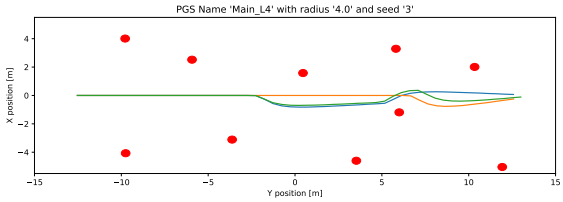
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



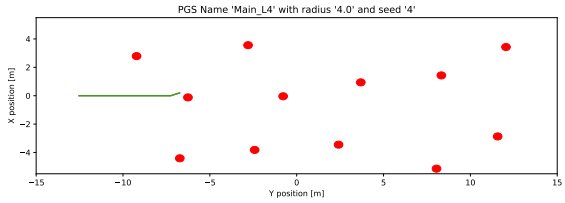
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



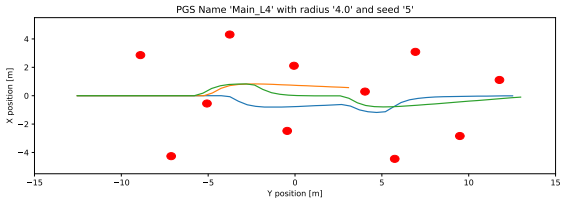
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



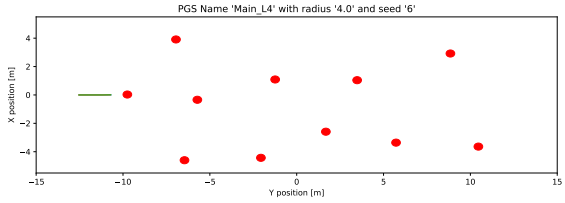
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



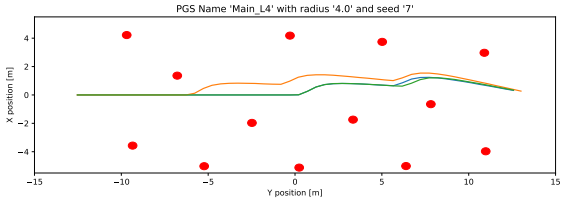
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



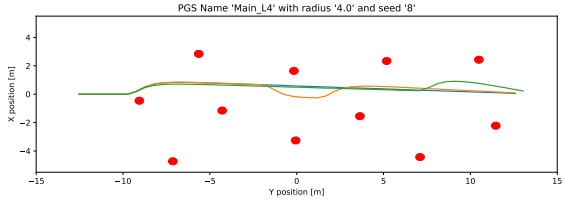
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



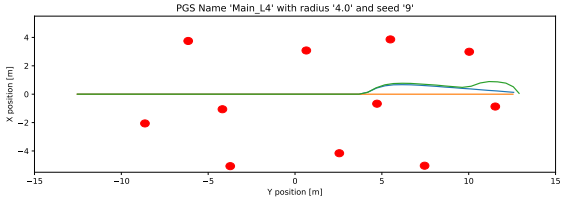
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

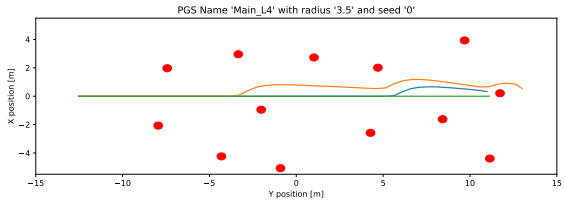


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

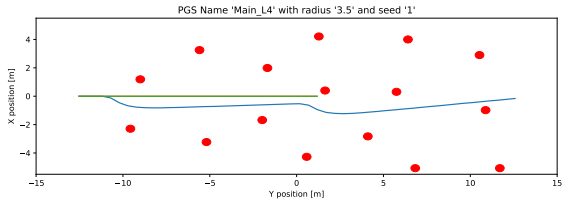


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

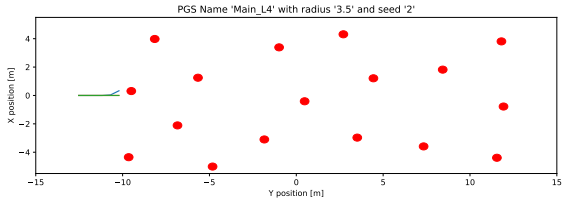
Figure A.22: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 4m, performed on obstacle set L4.



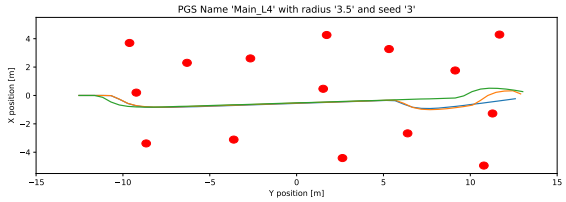
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



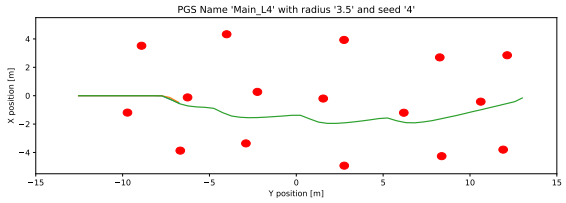
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



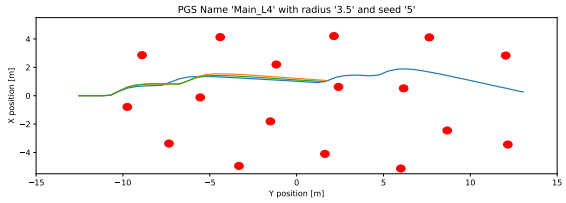
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



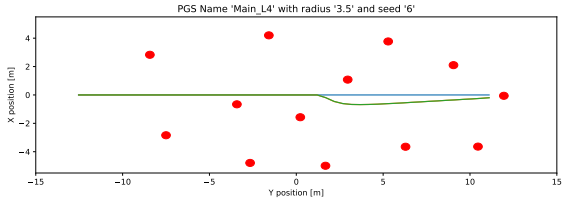
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



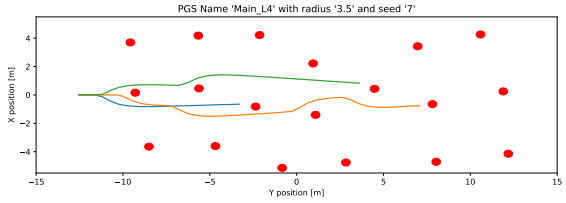
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



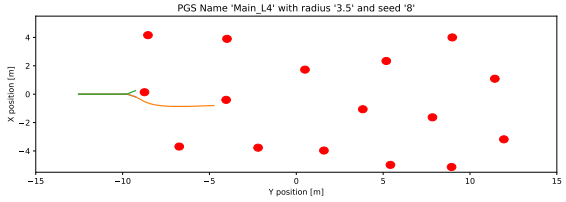
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



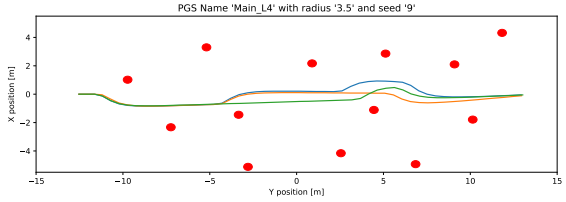
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

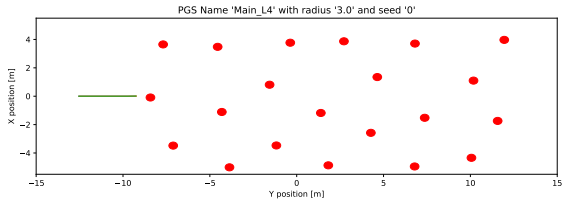


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

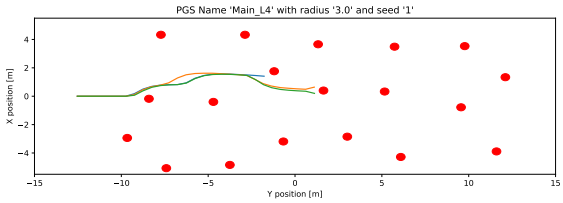


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

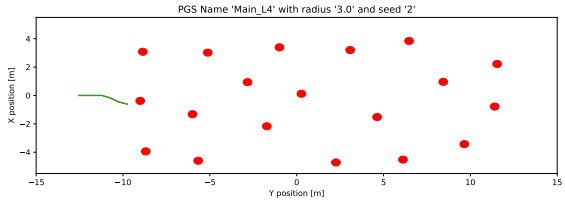
Figure A.23: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3.5m, performed on obstacle set L4.



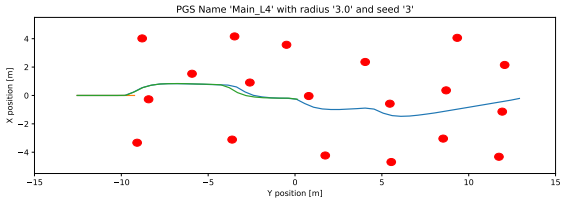
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



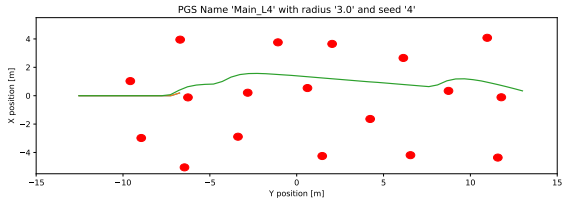
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



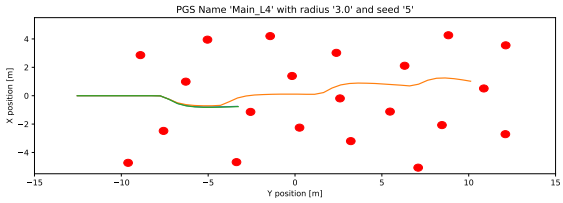
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



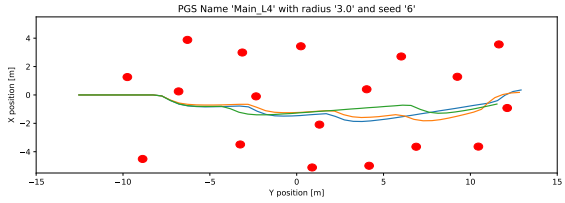
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



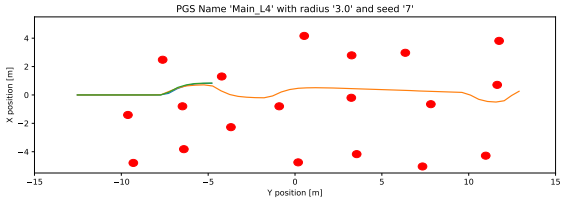
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



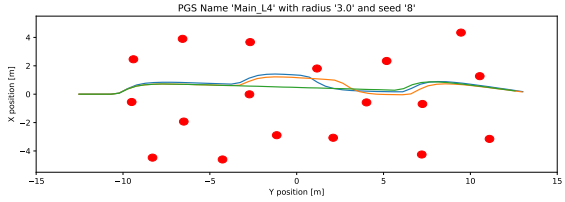
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



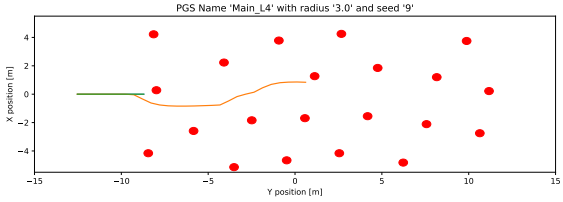
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

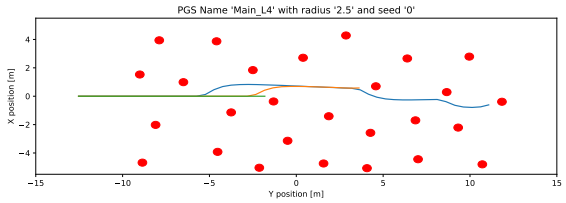


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

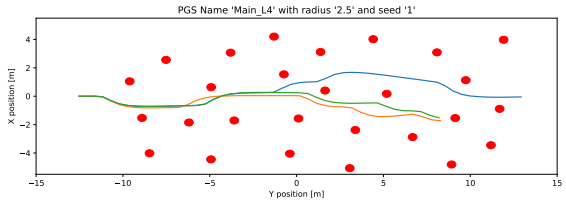


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

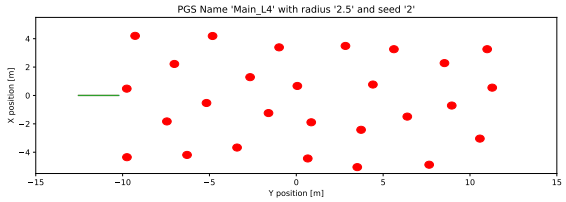
Figure A.24: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3m, performed on obstacle set L4.



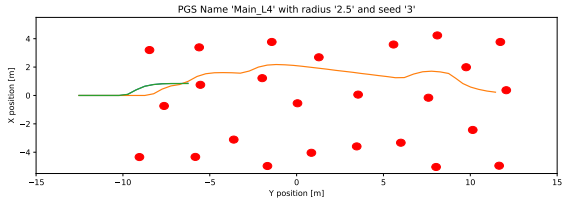
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



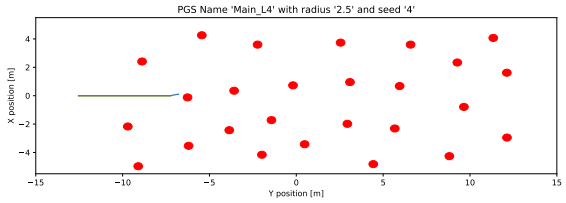
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



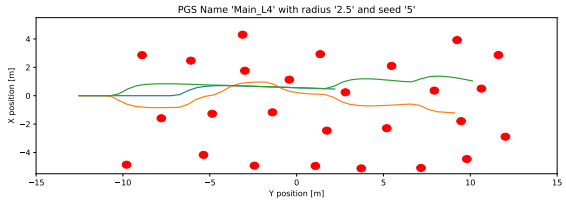
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



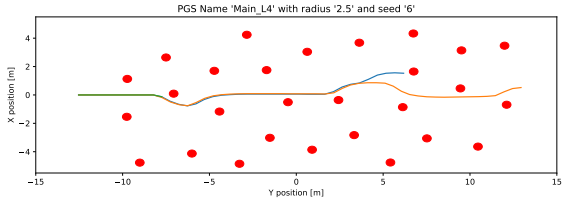
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



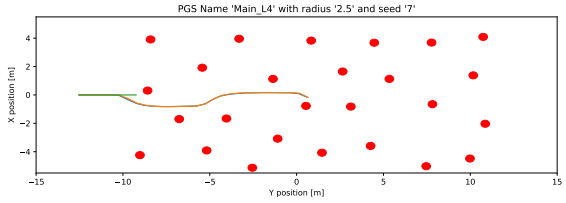
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



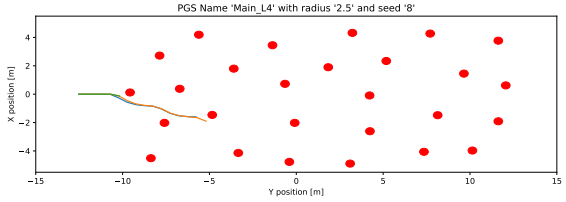
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



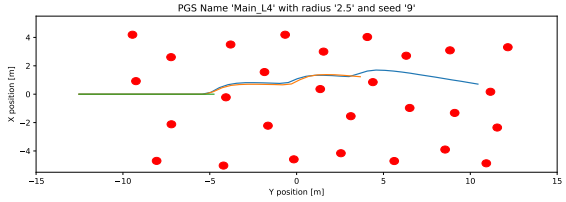
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

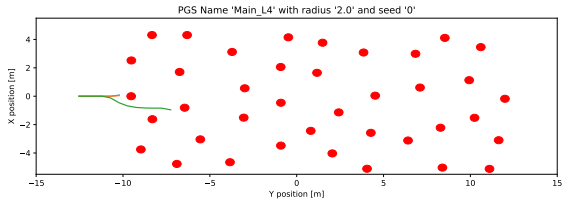


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

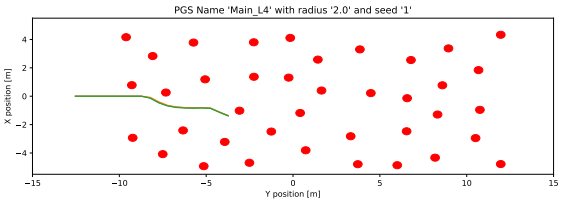


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

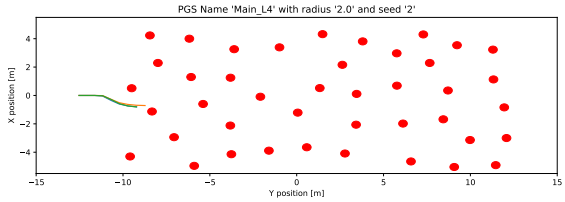
Figure A.25: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2.5m, performed on obstacle set L4.



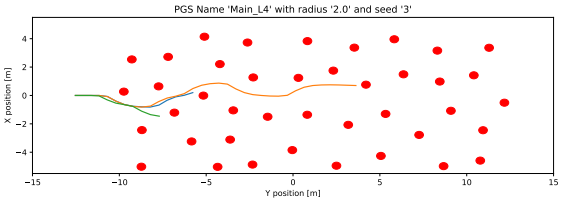
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



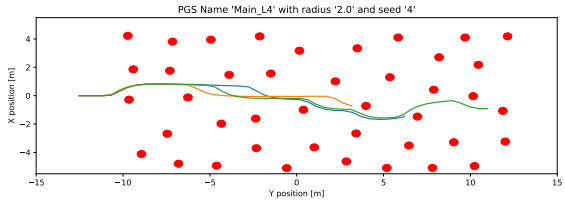
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



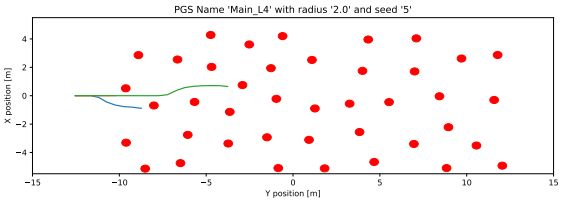
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



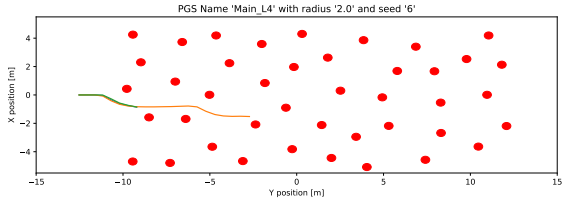
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



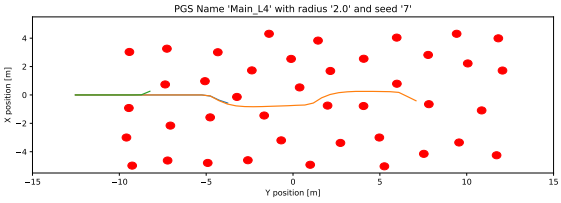
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



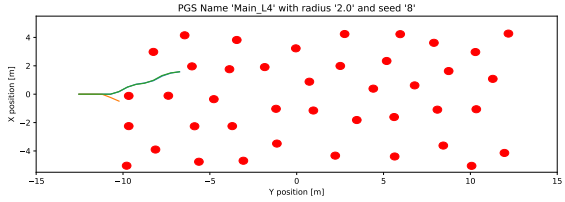
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



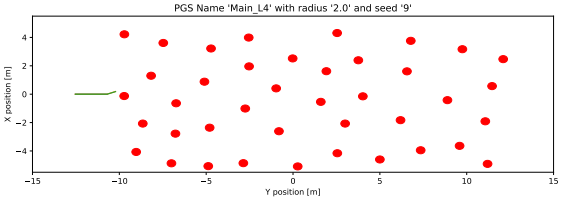
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

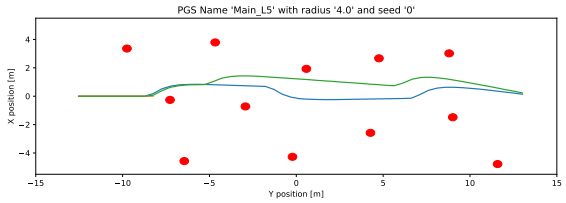


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

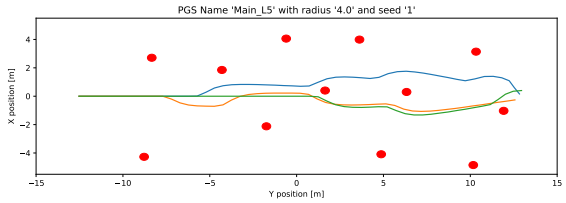


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

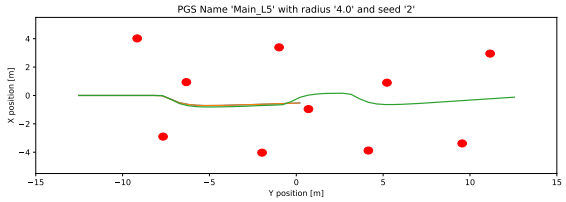
Figure A.26: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2m, performed on obstacle set L4.



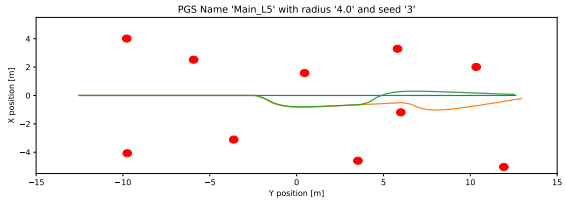
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



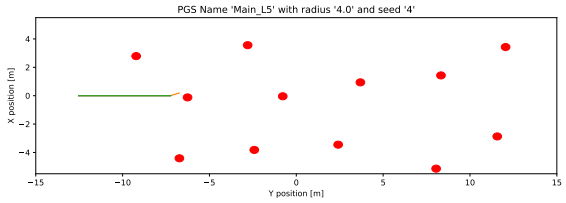
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



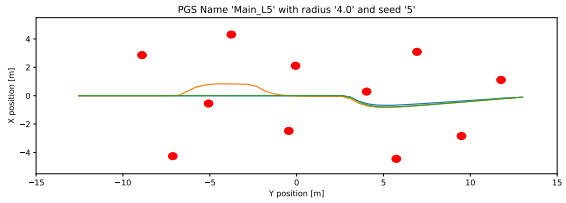
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



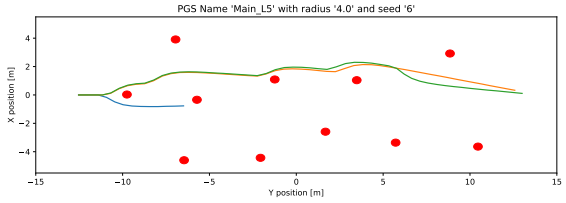
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



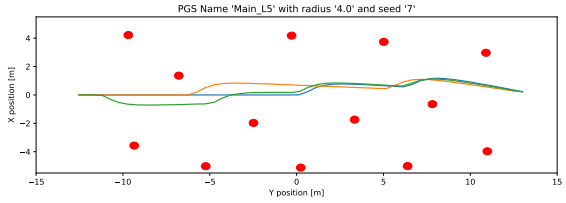
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



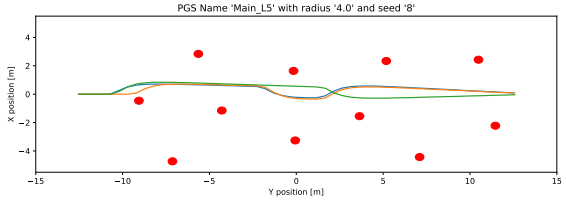
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



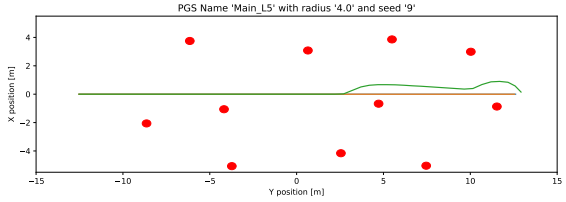
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

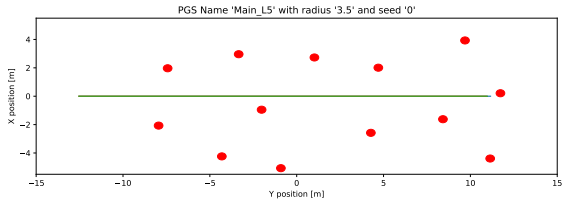


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

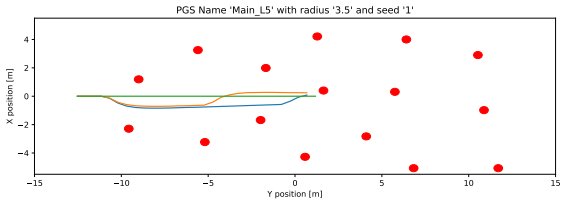


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 4m. The three benchmark trials are displayed in green, blue and orange.

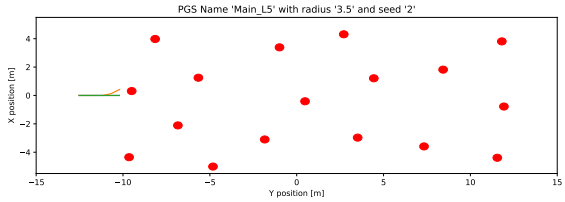
Figure A.27: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 4m, performed on obstacle set L5.



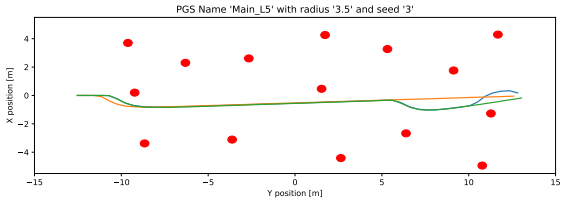
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



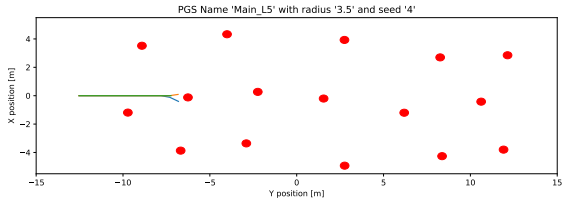
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



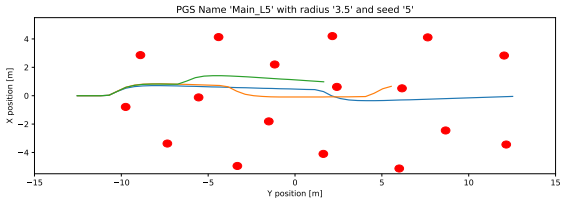
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



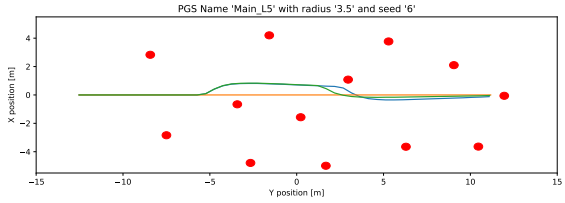
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



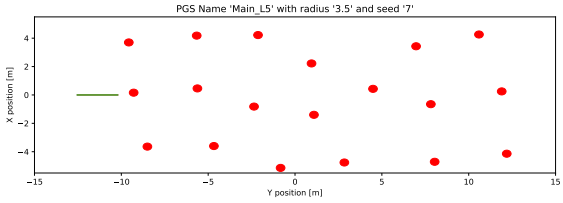
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



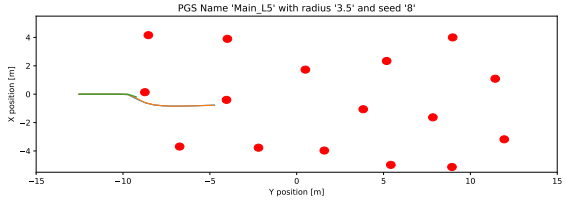
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



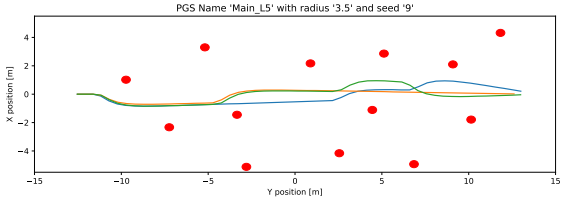
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

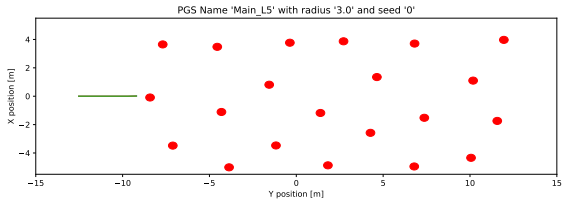


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

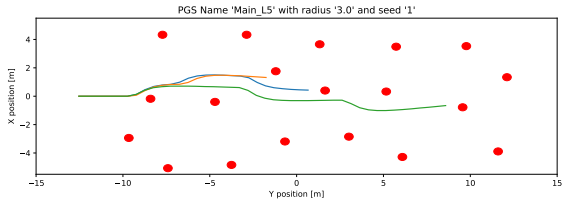


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3.5m. The three benchmark trials are displayed in green, blue and orange.

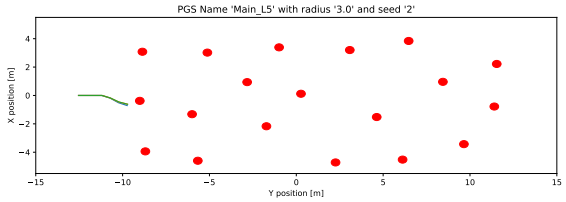
Figure A.28: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3.5m, performed on obstacle set L5.



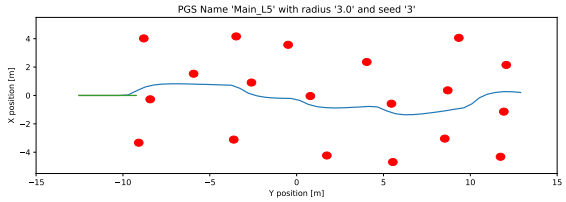
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



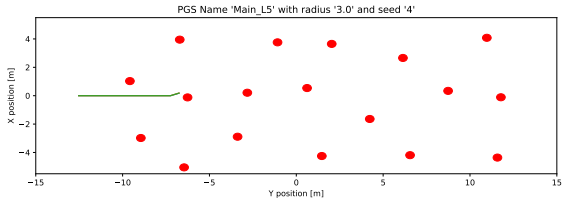
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



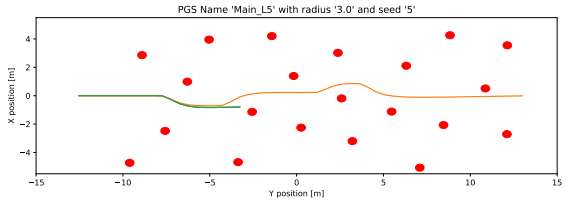
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



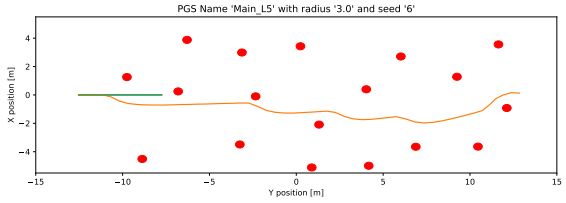
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



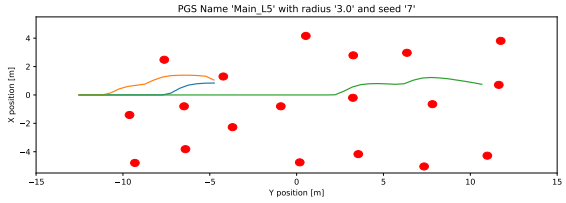
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



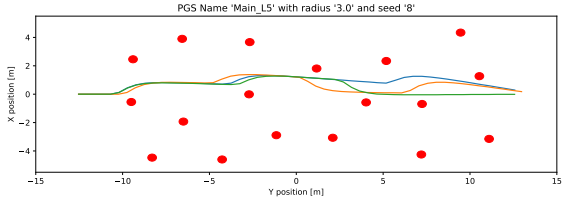
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



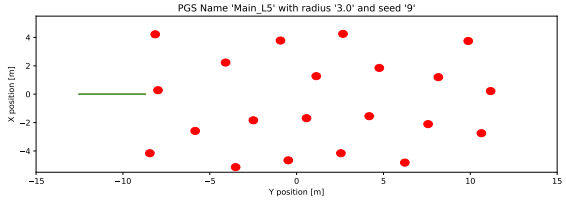
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

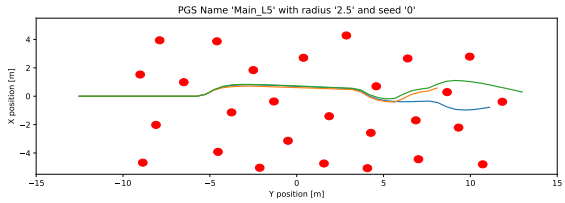


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

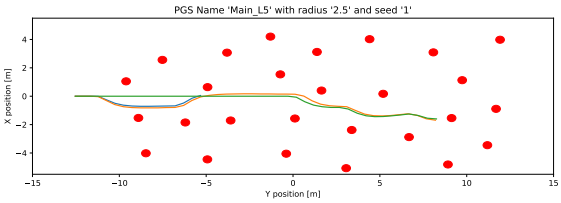


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 3m. The three benchmark trials are displayed in green, blue and orange.

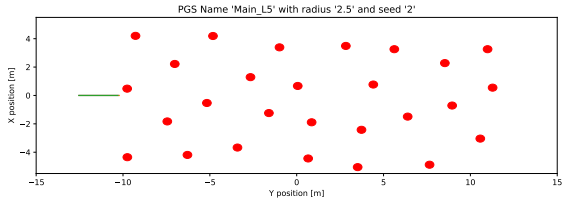
Figure A.29: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 3m, performed on obstacle set L5.



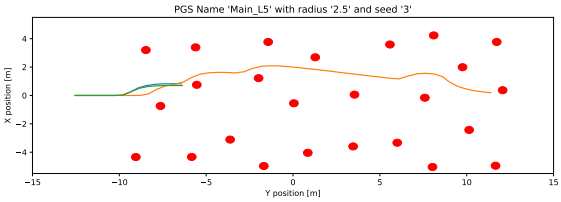
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



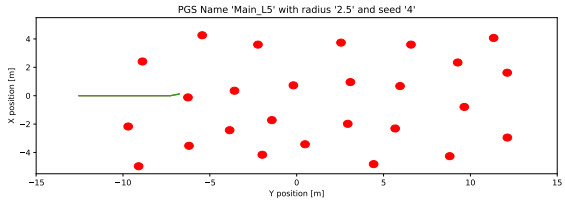
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



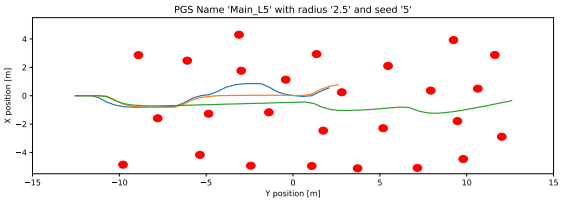
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



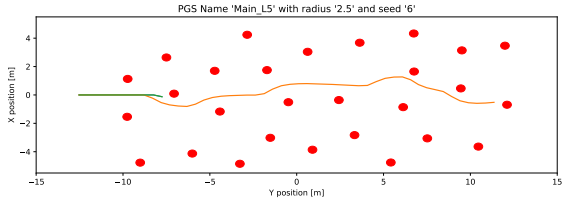
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



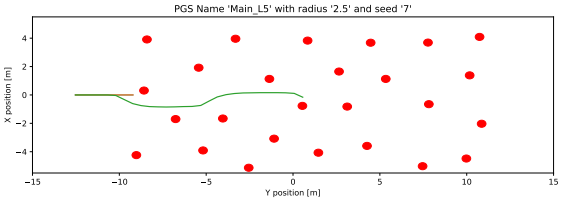
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



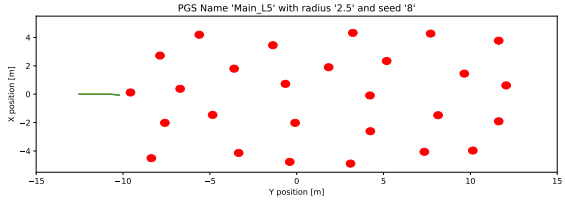
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



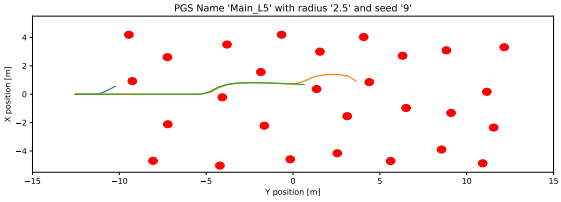
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

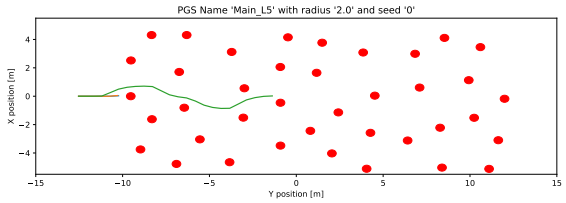


(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

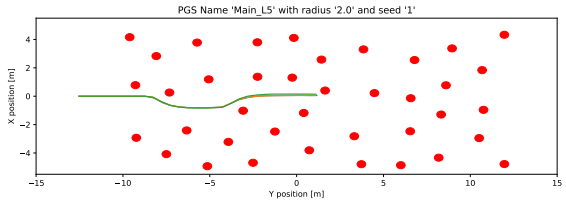


(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2.5m. The three benchmark trials are displayed in green, blue and orange.

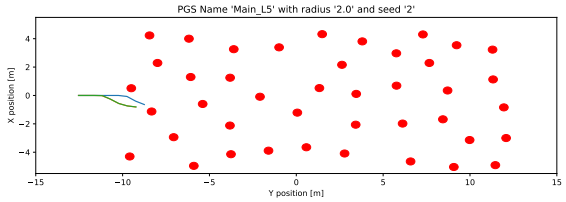
Figure A.30: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2.5m, performed on obstacle set L5.



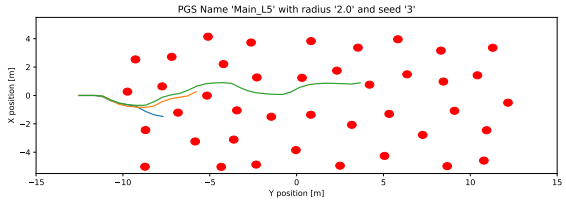
(a) The generated obstacle field with a seed of 0 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



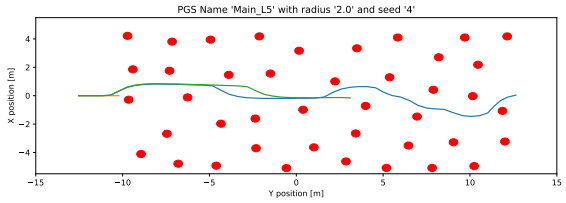
(b) The generated obstacle field with a seed of 1 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



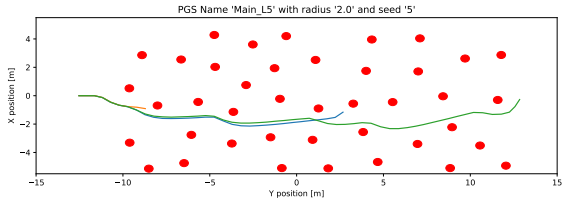
(c) The generated obstacle field with a seed of 2 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



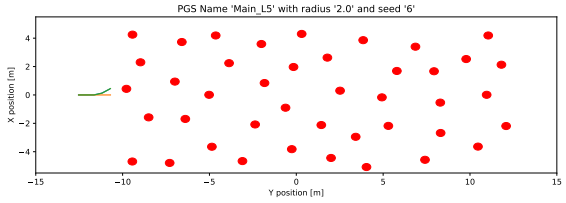
(d) The generated obstacle field with a seed of 3 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



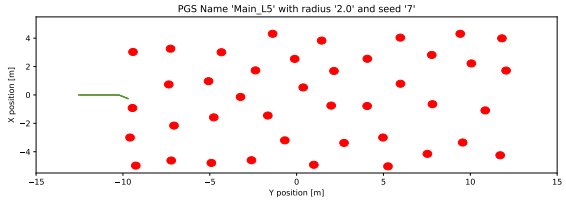
(e) The generated obstacle field with a seed of 4 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



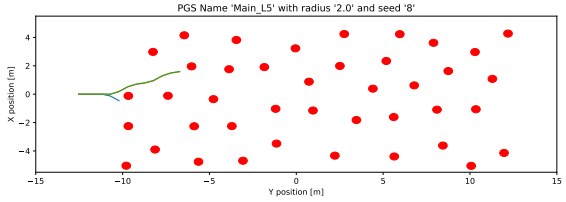
(f) The generated obstacle field with a seed of 5 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



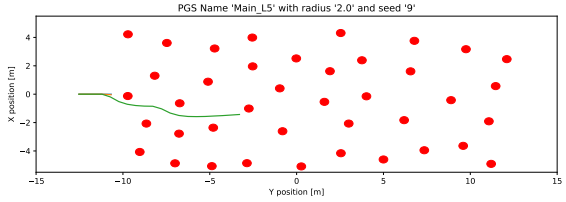
(g) The generated obstacle field with a seed of 6 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(h) The generated obstacle field with a seed of 7 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(i) The generated obstacle field with a seed of 8 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.



(j) The generated obstacle field with a seed of 9 and a Poisson Disc radius of 2m. The three benchmark trials are displayed in green, blue and orange.

Figure A.31: All generated obstacle fields and trial trajectories for a Poisson Disc radius of 2m, performed on obstacle set L5.

Table A.1: Traversability results for different Poisson Radii on the factory map.

Poisson Radius [m]	Adjusted Poisson Radius	Traversability									
		Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	Seed 6	Seed 7	Seed 8	Seed 9
4	7	11.89	11.87	11.93	12.17	11.97	12.11	11.83	12.02	11.95	11.94
3.5	6	11.58	11.44	11.51	11.69	11.47	11.49	11.84	11.6	11.48	11.76
30	5	11.11	11.36	11.15	11.24	11.36	10.9	11.43	11.22	11.32	11.15
2.5	4	10.28	10.4	10.57	10.68	10.52	10.66	10.36	10.52	10.57	10.37
2	3	9.46	9.63	9.23	9.03	9.55	9.02	9.16	9.29	9.18	9.34