UNIVERSITY OF AGDER

Development of Genetic and GPU-Based Brute Force Algorithms for Optimal Sensor Placement

by

Vegard Tveit

Supervisor

Geir Hovland, UiA

This master's thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder, 2018

Faculty of Engineering and Science

Department of Engineering Sciences

Abstract

Optimal sensor placement is a complicated task where several parameters have to be considered simultaneously. The problem has been a subject of much research in the last decades, but there does not seem to be a consensus regarding how to solve the problem. With the increasing use of sensors in a variety of applications, e.g. surveillance and motion tracking, optimal placement is desirable due to the possible reduction of the total cost.

In this thesis, a method for solving the static 3D Sensor Placement Problem is presented. From a 3D model of the environment, the constraints of the problem can be defined in the User Interface, including Regions of Interest, sensor parameters, possible sensor positions and discretization accuracy. The User Interface is developed in Matlab, utilizing a variety of functions and scripts.

Based on the output data from the User Interface, several optimization algorithms are developed and compared. First, a traditional Greedy Algorithm is developed in C++. This algorithm is extremely fast, but it has been proven to be sub-optimal. A Brute Force Algorithm is also developed in C++, to guarantee the global optimum. Since this algorithm computes the coverage of all possible sensor placement combinations, it will always produce the same result, which is guaranteed to be the global optimum. The Brute Force Algorithm requires vast amounts of computational power for more complex problems, and it has been shown that a threshold exists where the Brute Force Algorithm is not feasible due to hardware and computational time restrictions. To enable the use of the developed Brute Force Algorithm in more complex problems, it is converted to CUDA for utilization of a GPU. By converting the problem to CUDA, a considerable speedup was achieved, enabling the use of the GPU-Based Brute Force Algorithm on more complicated problems.

A Genetic Algorithm has also been developed in Matlab. The Genetic Algorithm is a meta-heuristic algorithm; hence it can not guarantee to produce the global optimum. By designing suitable genetic operators and investigating the effect of parameter tuning, a method has been developed which has proven to produce the optimal results for all verifiable tests. This algorithm converges to a solution much faster than the GPU-Based Brute Force Algorithm, also needing less computational power.

Preface

This thesis is written as the final compulsory part of the Master's Programme in Mechatronics at the University of Agder.It has been an interesting and challenging task, during which I have gained valuable knowledge in a variety of fields.

I would like to state my sincerest gratitude towards my supervisor, Professor Geir Hovland for his knowledge and guidance throughout the process of writing this thesis. Without the access to his computer, it would not have been possible to utilize the GPU as a vital part of this thesis.

A thank is also extended to Knut Berg Kaldestad for his expertise in CUDA programming and general discussions throughout the process of developing the Brute Force Algorithm. Also, I would like to thank Saber Derouiche and the rest of the Vision Systems team at National Oilwell Varco for suggestions regarding how to make the solution more practically usable.

Vegard Tveit

Vegard Tveit

Grimstad, May 24, 2018

Contents

Abstract **I**

Preface **II**

Contents **III**

List of Figures **VI**

List of Tables **VIII**

List of Algorithms **IX**

1 Introduction **1**
1.1 Research Background and Motivation . . . 1
1.2 Problem Description . . . 2
1.3 Report Outline . . . 2

2 Literature Review **4**

3 Background Theory **8**
3.1 Computational Geometry . . . 9
3.1.1 Polygons . . . 9
3.1.2 VRML and Indexed Face Sets . . . 10
3.1.3 Patch Objects . . . 11
3.1.4 The Art Gallery Problem . . . 12
3.2 Sensor Model . . . 13
3.3 Proposed Coverage and Visibility Model . . . 15
3.4 Environmental Model . . . 18
3.5 Matlab Graphical User Interface . . . 18
3.6 Optimization . . . 20
3.6.1 Optimization Categories . . . 21
3.7 Combinatorial Algorithms . . . 24
3.7.1 Heuristics . . . 24
3.7.2 Meta-heuristics . . . 25
3.7.3 Exact Algorithms . . . 28
3.8 Set-Cover Problem . . . 29
3.9 The JSON File Format . . . 32
3.10 Parallel Programming with CUDA . . . 33
3.10.1 The Graphical Processing Unit . . . 33
3.10.2 CUDA Programming . . . 34

4 User Interface, Case Studies and Results **38**
4.1 Matlab User Interface . . . 39
4.1.1 Adding the Floor and Camera Parameters . . . 40
4.1.2 Placement Lines . . . 42
4.1.3 Region of Interest . . . 42

4.1.4 Optimization Parameters and JSON Generation . . . 47
4.2 Sensor Model Verification . . . 49
4.3 Camera Placement Algorithms . . . 51
4.3.1 Greedy Algorithm . . . 51
4.3.2 Brute Force Algorithm . . . 53
4.3.3 Comparison . . . 55
4.4 Developing a Genetic Algorithm . . . 57
4.4.1 Algorithm Design . . . 57
4.4.2 Tuning the Variables in the Genetic Algorithm . . . 60
4.5 Speeding Up a Brute Force Algorithm . . . 64
4.5.1 Converting to CUDA . . . 64
4.5.2 Sizing the Problem for a NVIDIA GTX 1080 GPU . . . 65
4.5.3 Kernel Function . . . 66
4.5.4 Test Results . . . 67
4.6 Adding K-Coverage Functionality . . . 68
4.6.1 Implementation . . . 68
4.7 High Accuracy Final Test . . . 70
4.7.1 Case Analysis Using the Greedy Algorithm . . . 72
4.7.2 Genetic Algorithm . . . 73
4.7.3 Brute Force Algorithm . . . 77
4.7.4 Continuous Neighborhood Optimization . . . 80
4.7.5 Case Study Results . . . 82

5 Discussion and Further Work **84**
5.1 User Interface . . . 84
5.2 GPU-Based Brute Force Algorithm . . . 84
5.3 Genetic Algorithm . . . 85
5.4 Further Work . . . 86

6 Conclusion **88**

Bibliography **89**

A Appendix **A - 1**
A.1 Project Proposal . . . A - 1
A.2 GUI: User Manual . . . A - 2
A.2.1 Intended Use . . . A - 2
A.2.2 About . . . A - 2
A.2.3 Requirements . . . A - 2
A.2.4 File . . . A - 2
A.2.5 Edit . . . A - 3
A.2.6 Optimization . . . A - 5
A.2.7 Visualization . . . A - 5
A.3 Test Data for the Genetic Algorithm . . . A - 6
A.4 Test Data for the Final Test Brute Force Algorithm . . . A - 10
A.5 Matlab Scripts for the Set-Cover Problem . . . A - 12
A.5.1 Set Cover Greedy Algorithm . . . A - 12
A.5.2 Set Cover Exact Algorithm . . . A - 13
A.6 Matlab Functions for the Sensor Placement Problem . . . A - 14
A.6.1 Coverage Function . . . A - 14

A.6.2 Environmental Function . . . . A - 15
A.6.3 Function for Finding the Camera Position and Pan Angle . . . . A - 16
A.7 Matlab Scripts for the Genetic Algorithm . . . . A - 16
A.7.1 Genetic Algorithm for the Sensor Placement Problem . . . . A - 16
A.8 C++ Scripts . . . . A - 21
A.8.1 Sensor Model Test Case . . . . A - 21
A.8.2 Heuristic Greedy Algorithm . . . . A - 26
A.8.3 Combinatorial Brute Force Algorithm . . . . A - 32
A.9 CUDA C++ Scripts . . . . A - 39
A.9.1 C++ Program for Importing JSON and Exporting Text File Matrices for CUDA A - 39
A.9.2 CUDA Program for the Brute Force Algorithm . . . . A - 45
A.10 Scripts for the Final Test . . . . A - 49
A.10.1 Matlab Program for Genetic Algorithm with K-Cover . . . . A - 49
A.10.2 Matlab Program for Generation of Combinations Matrix . . . . A - 54
A.10.3 CUDA Program for Brute Force Algorithm with K-Cover . . . . A - 54
A.10.4 Matlab Program for Visualization . . . . A - 58
A.11 Matlab Scripts for the Continuous Neighborhood Optimization . . . . A - 60
A.11.1 Objective Funciton . . . . A - 60
A.11.2 Main Script . . . . A - 61

List of Figures

2.1 Different Methods of Optimization for the Sensor Placement Problem 7

3.1 Polygons: (a) Convex Polygon (b) Non-simple Polygon (c) Concave Polygon 9
3.2 VRML Coordinate System 10
3.3 Patch Representation of a Cube 11
3.4 3-Coloured Triangulated Art Gallery 12
3.5 Horizontal and Vertical Field of View 15
3.6 Camera Pan and Tilt Rotations 16
3.7 Objective Function with Multiple Minima 20
3.8 Visualization of Complexity Classes 23
3.9 Set Cover Problem Example [1] 30
3.10 Illustration of CUDA Threads and Blocks 35
3.11 GPU Memory Hierarchy 35
3.12 CUDA Example Code: C++ file 36
3.13 CUDA Example Code: CUDA file 37

4.1 Main Window User Interface 40
4.2 Example of Adding Floor in the User Interface 41
4.3 Displaying Added Sensors in the User Interface 41
4.4 Adding and Visualizing Placement Lines 42
4.5 Flowchart of the Region of Interest Generation 43
4.6 Definition of a Region of Interest 46
4.7 Adding a Region of Interest in the User Interface 46
4.8 Region of Interest Visualization in the User Interface 47
4.9 Problem Accuracy 48
4.10 Environment Model for Sensor Model Verification 49
4.11 User Interface Output : Sensor Test Model 50
4.12 Optimization Result Visualization for Sensor Model Verification 50
4.13 Visualization of the Rapid Increase of the Binomial Coefficient 53
4.14 Single Point Crossover Method 58
4.15 Mean Fitness Scores for Test A with Varying Mutation Factors 61
4.16 Mean Fitness Scores for Test B with Varying Mutation Factors 61
4.17 Mean Fitness Scores for Test C with Varying Mutation Factors 62
4.18 Mean Fitness Scores for Test D with Varying Mutation Factors 62
4.19 Comparison of Computational Times for the Developed Algorithms 63
4.20 CUDA Code Snippet for the Brute Force Kernel Function 66
4.21 3D Environment for the High Accuracy Test 70
4.22 Defining the Region of Interest and Placement Lines for the Final test 71
4.23 JSON Output Region of Interest and Obstacles Final Test 71
4.24 Computational Time for the Greedy Algorithm 72
4.25 Computational Time for the Genetic Algorithm 73
4.26 The Position and Fitness of the 30 Best Individuals for Various Generations 75
4.27 GA Coverage Results for 5 Cameras 76
4.28 GA Coverage Results for 6 Cameras 76
4.29 New Placement Lines for the BFA Final Test 77

4.30 Coverage per Chop Final Test 79
4.31 2D Comparison of BFA and Continous Optimization Results 82
4.32 3D Comparison of BFA and Continous Optimization Results 82

A.1 Problem Proposal . . . . A - 1
A.2 Directions of Grid Generation . . . . A - 4
A.3 Region of Interest Cube Coordinates and Indices . . . . A - 4

List of Tables

3.1 Increase of Combinatorial Equations with n=20 28
3.2 Subset Coverage for Set-Cover Example 30

4.1 Camera Parameters : Sensor Model Test 49
4.2 Brute Force Algorithm Test Case Results 56
4.3 Greedy Algorithm Test Case Results 56
4.4 Test Setup for Mutation Factor Tests 60
4.5 Variables and Timing for Mutation Factor Tests 60
4.6 NVIDIA GTX 1080 Device Query 65
4.7 GPU Test Results 67
4.8 GPU vs CPU Computational Results 67
4.9 Greedy Algorithm Analysis Results 72
4.10 Genetic Algorithm Parameters Final Test 73
4.11 Genetic Algorithm Final Test Results 74
4.12 Number of Combinations for the Final Test 77
4.13 C++ and CUDA Requirements and Results 78
4.14 Brute Force Algorithm Result for 5 Cameras 79
4.15 Results of Continuous Neighborhood Search 81
4.16 Result after Continuous Neighborhood Search 81
4.17 Results of the Final Case Study 83

A.1 Mutation Factor Test: A1 . . . . . A - 6
A.2 Mutation Factor Test: A2 . . . . . A - 6
A.3 Mutation Factor Test: A3 . . . . . A - 6
A.4 Mutation Factor Test: B1 . . . . . A - 7
A.5 Mutation Factor Test: B2 . . . . . A - 7
A.6 Mutation Factor Test: B3 . . . . . A - 7
A.7 Mutation Factor Test: C1 . . . . . A - 8
A.8 Mutation Factor Test: C2 . . . . . A - 8
A.9 Mutation Factor Test: C3 . . . . . A - 8
A.10 Mutation Factor Test: D1 . . . . . A - 9
A.11 Mutation Factor Test: D2 . . . . . A - 9
A.12 Mutation Factor Test: D3 . . . . . A - 9
A.13 BFA Best Coverage Result Chop 1-25 Final Test . . . . . A - 10
A.14 BFA Best Coverage Result Chop 1-25 Final Test . . . . . A - 11

List of Algorithms

Algorithm 1 Visibility Algorithm 17
Algorithm 2 Simulated Annealing Algorithm 26
Algorithm 3 Genetic Algorithm 27
Algorithm 4 Brute Force Algorithm for the Set-Cover Problem 31
Algorithm 5 Greedy Heuristic for the Set-Cover Problem 31

Algorithm 6 Discretize Walls 44
Algorithm 7 Fishnet Generation along Vertical Axis 45
Algorithm 8 Greedy Placement Algorithm 52
Algorithm 9 Brute Force Placement Algorithm 54
Algorithm 10 Genetic Algorithm for the Sensor Placement Problem 59
Algorithm 11 K-Coverage Algorithm 69

1 Introduction

1.1 Research Background and Motivation

A classical problem in computational geometry is the Art Gallery Problem, formulated by Victor Klee in 1974 [2] during a conference. It concerns placement of guards in a polygon-formed room. The guards have an unlimited range of sight and no angular limitations such as limited field of view. The problem is to determine the minimum number of guards and their position, to adequately cover the room. The Art Gallery Problem is often recognized as the predecessor of the Sensor Placement Problem.

In today's industry, the demand for optimal sensor solutions is increasing. As the sensor technology advances along with the progress in both optimization and algorithms, the sensor solutions today should be as close to the optimal solution as possible. In this thesis 3D sensors are the considered sensor type, focusing mainly on the camera. Cameras are among the most utilized sensor technologies since it is usable in a vast variety of applications, e.g., surveillance, autonomous vehicles, augmented reality, object tracking, and people detection. In many of these applications, the sensor cost can quickly get excessive. Taking into account the cost of the sensor itself in addition to the required wiring and installation, minimizing the number of sensors for a given application can reduce the total cost of the system. Optimizing sensor placement is a difficult task since there are many variables and parameters to be taken into account. First, the position and pose of each camera have to be decided based on given positioning constraints. The process of finding the best location can be influenced by requirements and options such as coverage redundancy for specific areas, different sensor types with individual sensor parameters, and obstacles which block the sensor view. Considering the mentioned parameters and variables, designing a general solution to solve the sensor placement problem is a demanding task.

Today, many camera arrays are placed iteratively, by humans, using trial-and-error methods which is both a time-consuming and challenging task. There are, in the literature, numerous approaches to solving the problem for different problem formulations but still there does not seem to be a consensus regarding which method is the best. An intuitive solution would be to evaluate all possible solutions in the search space using a Brute Force method. This is a computationally massive task since the search space can quickly get very large for such problems. However, with the technology available today concerning both CPU and GPU computational power, it should be possible to design algorithms that can guarantee the global optimum given specific inputs below a given threshold size determined by the hardware and time limitations.

With the increasing focus on algorithms and optimization, especially driven by the Artificial Intelligence technology, smart algorithms are getting increasingly popular. Algorithms such as neural networks, genetic algorithms, and other learning algorithms are well documented to work for complicated optimization problems. The optimal Sensor Placement Problem is usually formulated as a discrete optimization problem, where the environment is modeled using voxels, and the possible camera locations are given as points with a fixed distance between each other along plausible location lines such as beams or walls.

1.2 Problem Description

As the project proposal states, given in App. A.1, the objective is to develop an optimization method for optimal placement of 3D sensors in a defined environment. The proposal states that a User Interface should be designed where the constraints of the problem can be specified. Based on the output from this User Interface, an optimization algorithm should be developed to determine the optimal placement. It was also stated that a literature study should be performed to identify the state-of-the-art methods for solving related or similar problems.

Based on the project proposal a more specific problem formulation is presented:

- A literature review should be performed to determine the state-of-the-art methods for solving the Sensor Placement Problem.
- Develop a Graphical User Interface (GUI) where the optimization problem can be specified. The GUI should be developed in Matlab/Simulink with support for VRML models. It should be possible to specify several constraints: sensor parameters, sensor price, regions of interest for redundancy and possible sensor locations.
- The main optimization method will be a Brute Force Algorithm using a GPU. The code generated from the GUI must be compatible with CUDA C/C++ to run on the GPU.
- It is also preferable to design another algorithm, using a different approach. The reason for this is that a Brute Force Algorithm may require too much computational power to be practical for larger problems. Also, for users who do not have access to a CUDA-supported GPU, another approach should be developed.

1.3 Report Outline

The thesis is divided into six chapters.

- **Chapter 1:** Chapter 1 is the introductory chapter. First, the background and motivation for the thesis are presented. Following is the formulation of the problem statement before the outline of the thesis is described.
- **Chapter 2:** In Chapter 2, a comprehensive literature study is presented. Here, several applications are shown along with a variety of approaches used to solve related problems. This chapter aims to provide a background to the problem as well as to show the challenges related to the problem statement.
- **Chapter 3:** In chapter 3, the necessary background theory is presented. First, the science of computational geometry is presented which connects geometrical concepts and computational algorithms. A presentation of the developed sensor and environmental model succeeds this before a brief description of User Interfaces is given. The fundamental background of optimization theory is recapitulated where the discrete optimization is emphasized. Algorithms to solve combinatorial optimization problems are presented before an example of a Set-Cover Problem is given to show an implementation of combinatorial algorithms in a discrete optimization problem. The chapter is finalized with an introduction of the JSON format and parallel programming in CUDA.

- **Chapter 4:** The method is presented as case studies in chapter 4. For each case study, both the methodical work and the relevant results are presented. Firstly, the design and development of the User Interface are presented. Thereafter, the sensor model and User Interface output is verified. The first algorithm for solving the sensor placement problem is the Greedy Algorithm which is given together with the Brute Force Algorithm. Then, the development of the Genetic Algorithm is presented along with a test to investigate the effect of parameter tuning. Then, the Brute Force Algorithm is converted into CUDA code for utilization of the GPU. When the algorithms are developed and presented, implementation of k-coverage (redundancy) is introduced and applied to both the Brute Force and the Genetic Algorithm. The final case study of this chapter is to combine all the previous work into a case. This includes problem description in the User Interface followed by an initial analysis of the problem using the Greedy Algorithm. The Genetic Algorithm is then used along with the Brute Force Algorithm to find the best sensor placement for the given problem. Finally, a neighborhood search is conducted to investigate if this can further improve the solution.
- **Chapter 6:** In chapter six, the methods and results are discussed along with a presentation of the suggested further work.
- **Chapter 7:** The final chapter of this thesis concludes the work and answers the problem statement based on the achieved results.

All required additional information such as code, test results and supplementary information is provided in the appendices. Also, a GitHub repository is made where all code can be accessed and downloaded as well as the full source code for the User Interface: `https://github.com/Vegardtve/sensorplacement`. The source code for the User Interface is too extensive to be appended with the thesis, and the reader is therefore encouraged to visit the repository for running the User Interface or investigating the source code.

2 Literature Review

In this chapter, a review of the current research will be presented, providing an overview of applications, methods, and results of previous work in Sensor Placement Problems (SPP) and problems closely related. The majority of the research has been focused on the 2D discrete problem formulation, but in the later years, the trend shows an increasing amount of research on discrete 3D problems. There is some research focused on continuous problem formulations; however, most of this research is limited to specific cases and not very useful in practice. Kirchhof [3] states that:

For almost every existing sensor technique a localization solution exists. These solutions are often in a prototype state since they are either developed as a proof of concept for the sensor technique or to be used in specialized individual applications.

This statement highlights a problem that can be seen in much of the work in the literature. Since around 2000, there has been a large number of publications on issues in sensor placement. However, for almost every new paper a new method of formulating and solving the problem is developed.

In 1995 Tarabanis et al. [4] published an extensive survey on sensor planning. At this point, the Sensor Placement Problem was mostly solved with trial-and-error solutions involving human interaction. In this work, several reasons for making sensor placement automatic was listed, such as reducing time and cost of sensor placement and achieving more robust solutions.

Hörster and Lienhart [5] formulated four different versions of the SPP to provide a consensus for problem formulations:

1. Given the number of sensors of one type and their specific parameters, determine the position and pose such that the coverage is maximized.
2. Given several types of sensors, their parameters and specific costs as well as the maximum total price of the sensor array, determine the sensor array, the sensor types and positions/poses that maximize coverage in the given space.
3. Given the fixed positions and respective types of some sensors determine their optimal poses such that coverage is maximized.
4. Given a minimally required percentage of coverage, determine the sensor array with the minimum cost that satisfies the coverage constraint.

Additionally, the authors presented a User Interface (UI) where users could apply the developed methods to all 2D problems within the limitations of the User Interface. In this UI, the possible sensor locations could be specified, as well as Regions of Interest (ROI) which are areas of higher importance. The authors found that a Binary Integer Problem (BIP) formulation was able to solve the SPP but had restrictions related to the total number of constraints due to the complexity of the algorithm. For more extensive problems, the authors suggested a Greedy Algorithm.

Bianco and Tisator [6] suggested a Direct Search (DS) based algorithm for solving the problem stated by [5]. The proposed algorithm outperformed approximate algorithms like the Greedy algorithm within a reasonable convergence time. Although the problem is formulated in 2D, the authors state that extending the SPP to 3D would only require minor changes.

Erdem and Sclaroff [7] formulated the 2D SPP as problem formulation number 4 stated by [5]. The problem was converted to a variant of the Set Cover Problem solved using BIP optimization. In the future work, the authors stated that a continuous formulation that guarantees the global optimum would be desirable.

Kirchhof [3] also focused on the 2D SPP but solved the problem for both the discrete and the continuous case. The discrete problem is solved by BIP and Mixed Integer Programming (MIP). The continuous case is briefly presented in this work, and solved using a Nonlinear Programming (NLP) method. These algorithms work together in a way that the continuous approach aims to improve the solution of the discrete problem formulation. The author states that although the work is formulated as a 2D problem, it is applicable also in 3D.

Altinel et al. [8] analyzed the BIP and stated that for large-scale problems the solution would be too computationally complex to achieve in a reasonable amount of time. The authors suggested using a Greedy Algorithm to provide a satisfactory solution to the 2D SPP in shorter time.

Hovland and Dybedal [9] further explored the Integer Programming methods and developed a Mixed-Integer Linear Program (MILP) for solving the continuous 3D SPP. By using linearization, the problem was converted from a nonlinear problem to a linear problem. For the linearized problem, it was possible to find the optimal solution. The drawback with this solution was the computational complexity of the MILP and the linearization, which limits the scalability of the solution.

Davis and Mittal [10] researched the SPP with extension to random occlusion. In this research, it is stated that a full search algorithm is too computationally expensive to be used in such problems. Instead, a Simulated Annealing (SA) algorithm is used to find a sub-optimal solution in a reasonable amount of time. Introducing random occluding objects presents another aspect of many problems such as surveillance and industrial automation where objects can move around in the volumes of interest.

Wireless Sensor Networks (WSN) have received a considerable amount of research in sensor placement in the latest years. In these problems, a large outdoor environment is often considered, which should be covered by an extensive sensor array. Akbarzadeh et al. [11] developed a probabilistic sensing model which has been implemented on different optimization schemes such as Covariance-Matrix Adaption Evolution Strategy (CMA-ES). The authors claim that this approach is novel when taking into account both the probabilistic sensor model and also using elevation maps to analyze the visibility in 3D environments. In a later article Akbarzadeh et al. [12] presented a Gradient Descent Algorithm to optimize the SPP based on probabilistic sensing models which gave as good or better results in much shorter computational time than the CMA-ES algorithm. Tam et al. [13] formulated the problem to solve a k-coverage 3D problem where each point in a Region of Interest needed to be covered by the sensing range of at least one sensor. The authors also used probabilistic sensing and a new method of determining if an obstacle is blocking the visibility of a point from a given sensor, based on linear regression. A modified Particle Swarm Optimization (PSO) algorithm is presented to solve the SPP. Topcouglou [14] focused on a more specific WSN, namely the wireless multimedia sensor network (WMSN). A simulation environment for large, outdoor, 3D environments was the main objective for this work. A Genetic Algorithm (GA) is suggested to solve the placement problem.

Sensor placement is vital in other applications as well, such as the reconstruction of known scenes. Fleishman et al. [15] stated that the global optimum was not necessary due to the practical application of the research. A Greedy Algorithm was developed to find satisfactory camera positions given a set of possible locations.

With the increased popularity of Virtual Reality (VR), Augmented Reality (AR) and Mixed Reality (MR), depth cameras have been a subject of much research. Chabra [16] focuses on optimizing placement of depth cameras for known dynamic 3D scenes. The intended use of this work is in medical studies and surgery practice. A Greedy Algorithm is presented to determine how many cameras are needed, and then a SA algorithm determines the best position by avoiding occlusion. Occlusion is vital in 3D reconstruction since it can lead to holes in the reconstructed scene.

Medical training was also the application for the work of State et al. [17]. In this work, a simulator was developed which supported interactive placement and manipulation of multiple cameras to see the effect of altering the positions of multiple sensors. The users could then place the cameras to cover as much of a medical operation as possible.

Rahiman and Kearney [18] studied the motion tracking issue in VR systems. Reference markers are often used to track motion, and then a 3D scene can be reconstructed by triangulation. To make this reconstruction as good as possible, a SA algorithm was designed to find a good solution in a reasonable amount of time.

Another area in which sensor placement can be a useful tool is barrier coverage. Zhang [19] formulated a barrier coverage problem where maximum coverage of the whole scene is not necessary, but rather to have full coverage of a particular barrier, e.g. to observe breaches. An Integer Linear Program (ILP) was developed to solve the SPP, which proved superior to a Greedy Algorithm.

Autonomous vehicles have been another area of much research for a large variety of applications. Cortés et al. [20] studied a sensing network where the sensors were mounted on autonomous vehicles moving in 2D. Autonomous vehicles present a dynamic aspect to the sensor placement problem, which is mainly static in the literature. Both continuous and discrete problem formulations have been researched, where both have been solved by solving the problem in a Voronoi partitioned area.

Schwager et al. [21] also focused on moving sensors but rather than agents moving in 2D; the agents can move in 3D, i.e. flying agents. Flying agents can be useful in many applications such as surveillance, object recognition and tracking. A decentralized control method was developed for deploying all agents in the sensing network.

As described throughout this chapter, there are numerous solutions to the Sensor Placement Problem, many of them problem-specific with a lack of generality. Fig. 2.1 shows an illustration of the various optimization methods described throughout this chapter.

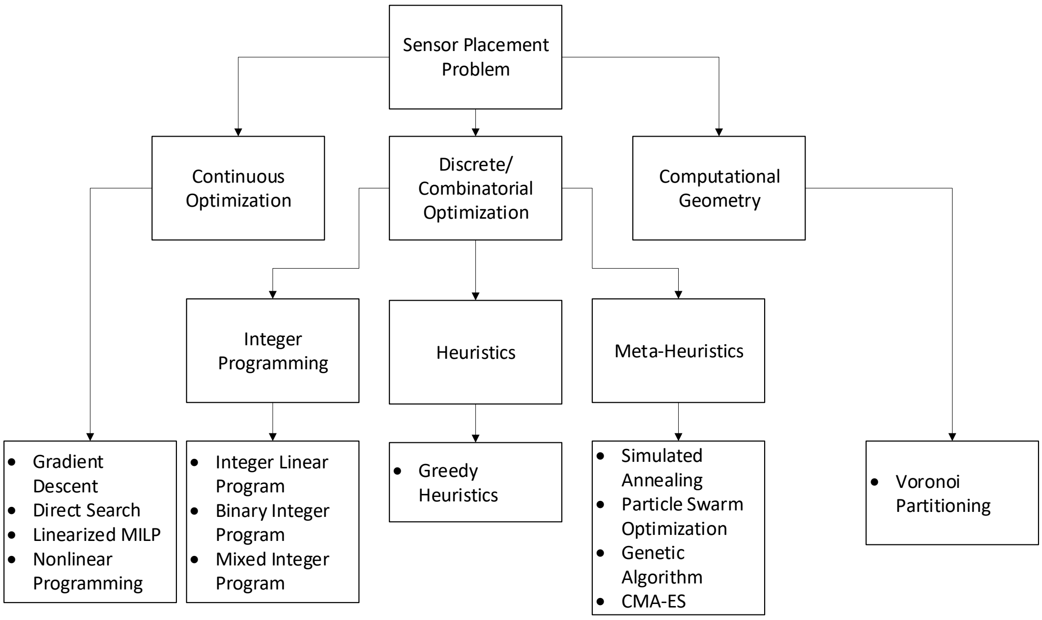


Figure 2.1: Different Methods of Optimization for the Sensor Placement Problem

3 | Background Theory

In this chapter, the relevant theory will be presented. Firstly in Sec. 3.1, the concepts of Computational Geometry, which is the science of combining geometrical concepts and algorithms, is introduced and presented in the context of sensor placement. Previous work on sensor model formulations is presented in Sec. 3.2. In Sec. 3.3 the proposed coverage and visibility model is presented before the environmental model is shown in Sec. 3.4. Then, an introduction to User Interfaces is given in Sec. 3.5.

The main research field of this thesis is optimization and algorithms, and a thorough introduction to optimization theory is presented in Sec. 3.6. Several algorithms to solve discrete optimization problems are presented in Sec. 3.7, where both heuristics, meta-heuristic and exact methods are presented. A Set-Cover example problem is formulated and solved in Sec. 3.8 to show the difference between exact and heuristic algorithms.

The main programming languages for algorithms are Matlab and C++. To use Matlab data in C++, and vice versa, the JSON file extension is used, which is presented in Sec. 3.9. The final section, Sec. 3.10 concerns parallel programming. With the technology available today, programmers can benefit greatly from using GPUs in complex optimization tasks. This section presents CUDA, which is a programming language developed by NVIDIA to utilize the parallel nature of GPUs, along with the necessary background to adopt CUDA to different applications.

3.1 Computational Geometry

Computational geometry connects algorithms and geometry to solve geometric problems. Forrest [22] defined computational geometry as

The mathematical representation, manipulation, analysis and synthesis of shape information in a computer.

Since 1970, computational geometry has been a field of much research due to its broad variety of application domains, such as:

- Robotics
- CAD/CAM software
- Computer graphics
- Pattern recognition

In many cases, extensive data sets cause problems challenging to solve due to the computational complexity of the problem. Therefore, an important aspect of algorithms in computational geometry is efficient programming while keeping the algorithms robust. Often in the literature, assumptions are made to avoid the challenges of creating such algorithms, resulting in unrealistic algorithms only applicable in particular cases [23].

3.1.1 Polygons

One of the main elements in geometry is the polygon. It is a figure that lies in a plane, i.e. it is two dimensional. It is defined as a chain of straight lines that can form a closed circuit. Two well-known examples are triangles and rectangles, being 3- and 4-gons, respectively.

Polygons can be classified in several ways. A convex polygon, S, is defined such that all two points $(p, q) \in S$ can be connected with a straight line inside the polygon. A non-convex polygon is called concave. Here the floor plan of the area to be observed is limited to a concave polygon. Also, the polygon should be simple, i.e. no lines should cross each other. The difference between a concave and non-simple polygon is that a concave polygon can not have intersecting lines. Three different polygons can be seen in Fig. 3.1.

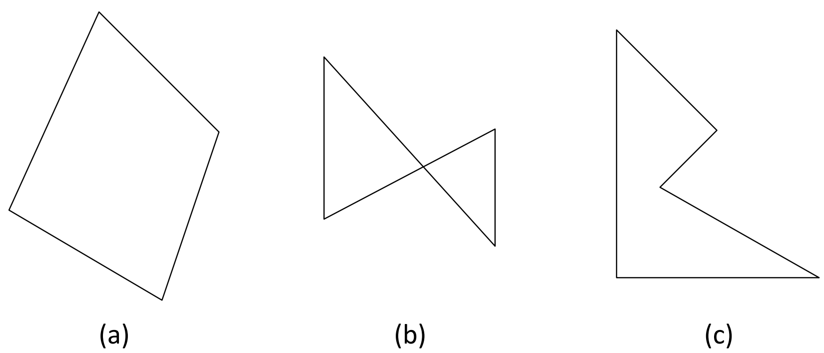


Figure 3.1: Polygons: (a) Convex Polygon (b) Non-simple Polygon (c) Concave Polygon

3.1.2 VRML and Indexed Face Sets

In 1995, Virtual Reality Modeling Language (VRML) was introduced to define 3D virtual worlds to be used in VRML browsers or on the world wide web. A VRML file is text-based and contains information regarding vertices, faces, colors, etc. for 3D polygons. The VRML file syntax and format is out of the scope of this thesis since the files are mostly generated by the modeling software, and will therefore not be described. The interested reader can find more information regarding the VRML language in [24].

A VRML file is often called a world, and it has the file extension **.wrl*. Objects can be described in several ways in VRML, but due to the Matlab support of the *Indexed Face Set* geometry node, this is considered here.

The *Indexed Face Set* node is specified with coordinates of vertices and indices to define the faces. It is not limited to triangles, but VRML can only draw convex faces; thus concave faces are split into several convex faces. Most CAD software can export the 3D models to **.wrl* to be used by the developed Matlab UI. If a 3D model is saved in any other file extension, it can be converted into VRML by using conversion software, for example MeshLab [25].

In Matlab, the support for 3D models is mainly based on the VRML format, but to a certain extent, X3D can also be used in the 3D editor and 3D animation tools included. With the VRML editor, 3D worlds can be edited in Matlab [26].

Throughout this thesis, the VRML coordinate system will be used unless otherwise is specified. The VRML coordinate system differs from the coordinates system used by Matlab and can be seen in Fig. 3.2.

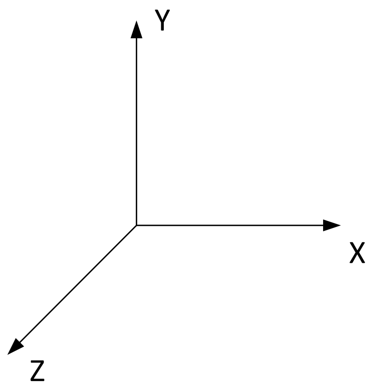


Figure 3.2: VRML Coordinate System

In Matlab, a function is provided for converting VR worlds modeled with indexed face sets into patches. These functions and the patch objects are covered in the following section.

3.1.3 Patch Objects

Patch objects are used in Matlab to draw real-world objects using polygons that may or may not be connected. The objects can be drawn in arbitrary shapes, which make patch objects a powerful tool when handling tasks with 3D objects [27].

Patches can be created from *Indexed Face Sets* using the Matlab function *vrifs2patch(ifs)* [28], where *ifs* is the variable where the *Indexed Face Set* is stored. Additionally, patches can be implemented in an existing *ifs* variable using the function *vrpatch2ifs(patches,ifs)* [29]. Since patch objects are easy to define and handle in Matlab by themselves, all objects made in the user interface will be created as patches.

A patch object can be defined in multiple ways, but in this thesis, the method of defining multifaceted patches will be used. This is described in the Matlab documentation [30], but it will be slightly altered since it is desirable to have triangular faces rather than rectangular. An example of how to set up a patch object will be demonstrated for a cube using low-level syntax.

In Fig. 3.3, a visualization of how the triangular faces will look can be seen. Also, the vertices are indexed from 1 to 8. The faces are defined as can be seen in Eq. 3.1.

$$f = [1,6,2;1,5,6;2,6,3;6,7,3;5,8,6;8,7,6;1,4,2;4,3,2;4,7,3;4,8,7;1,5,4;5,8,4] \tag{3.1}$$

The numbers in Eq. 3.1 are the vertices of the object. The coordinates of the vertices need to be specified as v. The patch object can then be made using the Matlab function *patch('Vertices',v,'Faces',f)*, which produces the cube seen in Fig. 3.3.

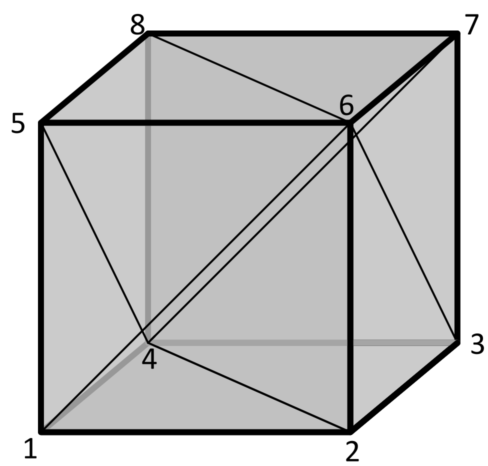


Figure 3.3: Patch Representation of a Cube

3.1.4 The Art Gallery Problem

The 2D case of the Sensor Placement Problem is closely associated with the Art Gallery Problem, stated by Victor Klee during a conference in 1974:

Consider a room shaped like a simple polygon with n vertices. Determine how many guards, able to survey 360° *about their own position, which is fixed, is the minimum to cover the whole polygon* [31].

From the problem formulation, it is seen that this problem can be extended to a Sensor Placement Problem by adding some constraints. It is much more theoretical than the sensor placement problem since the guards do not have limited range of sight, obstacles are not considered, and there are no limitations in the field of view. However, the solutions presented are interesting and have provided researchers with ideas that can be adapted to more complex problems.

Chvátal [2] presented a proof that $\frac{n}{3}$ guards can cover any simple polygon of n vertices. It is proved that this number is always sufficient and occasionally necessary.

In 1978, Fisk presented his proof of Chávatal's theorem, which was much more straightforward and elegant than the original proof. This proof is based on 3-colouring and triangulation:

Firstly, triangulate the polygon so that no new vertices are added. By 3-colouring every triangle, every vertex has one of three colors, and it can be shown that each color forms a set of guards placed at these vertices able to cover the whole polygon. Choosing the color with the fewest vertices implies that the number of guards must be smaller than or equal to $\frac{n}{3}$ [32].

A visual representation of the proof by Fisk can be seen in Fig. 3.4, where a polygon with $n = 11$ vertices can be seen. By applying the method presented above, it can be seen that the green vertices form a set of 3 vertices. By placing guards at each of these vertices, the whole art gallery is covered. This example also shows an important aspect of the proof presented by Chvátal. The art gallery in the figure can be covered by only using two of the three green vertices; hence the solution of three vertices is sufficient, but in this case not necessary.

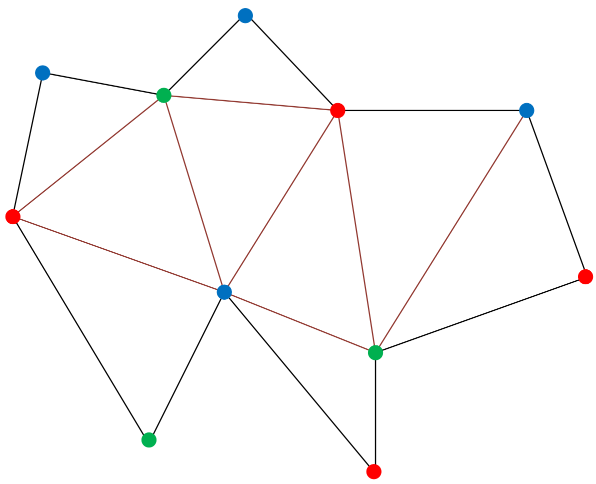


Figure 3.4: 3-Coloured Triangulated Art Gallery

3.2 Sensor Model

The Art Gallery Problem (AGP) has many similarities to the Sensor Placement Problem. However, the 3D SPP is more complex due to several important factors:

- The AGP is formulated in 2D, while the SPP considered here is a 3D problem.
- The sensor model in the AGP assumes an omnidirectional sensing range, whereas many sensors, in reality, have a directional field of view.
- The AGP assumes unlimited sensing range. This assumption can not be made for real sensors, which have limited range.

The Art Gallery Problem can be seen as a predecessor to the Sensor Placement Problem, and computational geometry inspires many of the solutions to Sensor Placement Problems. One of the main differences between the 2D Sensor Placement Problem and the Art Gallery Problem is the sensing model. In this section, some previous work on sensor models will be presented for both the 2D and the 3D case.

Ma and Liu [33] consider 2D sensors with a limited field of view. The sensor model is a 4-tuple consisting of position, sensing range, centerline, and field of view. A point is said to be covered if and only if:

$$||L - L_1|| \leq R \tag{3.2}$$

$$\angle(L, L_1) \in (-\alpha, \alpha) \tag{3.3}$$

where:

L	-	Sensor location
L_1	-	Point location
R	-	Sensing range
α	-	Field of view

Peng et al. [34] developed a 3D sensor model with a limited field of view. Also, a hole detection method is presented. The model uses binarization with edge and feature extraction to determine the location of coverage holes.

PTZ (Pan-Tilt-Zoom) cameras are considered for fixed sensor positions. The sensor model is a 5-tuple consisting of position, sensing direction, maximal tilt angle and two offset angles to describe the vertical and horizontal field of view relative to the sensing direction.

Akbarzadeh et al. [35] use probabilistic sensor coverage instead of the commonly used binary coverage method for two reasons: 1) It is stated to represent real sensors better. 2) The coverage function is differentiable, which is a requirement for some optimization methods such as the Gradient Descent Algorithm.

By also introducing weighted data points, the importance of covering a given point is also considered. A Line-of-Sight (LoS) method is proposed to identify obstacles blocking the visibility. This method uses the elevation data of the environment.

A point is said to be covered if and only if:

1. It is visible by the given sensor.
2. The angle in the XY plane between the sensor and the point is inside the coverage area given the pan angle and the field of view.
3. The angle between the sensor and the point in the XZ plane is inside the coverage area given the tilt angle and the field of view.
4. The distance between the point and the sensor is below the sensor range.

In non-probabilistic formulations, the above constraints are modeled as stated above. In this work, probabilistic sensing is used, hence an uncertainty is introduced if the point is too close to the edge of the field of view or the limit of the range. To implement probabilistic sensing, sigmoid functions are used.

Tam et.al [13] presents a novel 3D sensor model based on [35]. Also, a novel LoS model is developed to determine the number of points between a given sensor and a point. A point is said to be covered by a given sensor if and only if the following conditions are met:

$$d(s_j, e_i) \leq r_s^j \tag{3.4}$$

$$\arctan\left(\frac{y_i^e - y_j^s}{x_i^e - x_j^s}\right) \in [\alpha_j, \alpha_j + \theta_j] \tag{3.5}$$

$$\arctan\left(\frac{h_i^e - h_j^s}{d(s_j, e_i)}\right) \in [\epsilon_j, \beta_j + \epsilon_j] \tag{3.6}$$

$$\text{There are no obstacles blocking the LoS} \tag{3.7}$$

where:

$d(s_j, e_i)$	-	Distance between sensor s_j and point e_i
(x_i^e, y_i^e, h_i^e)	-	Coordinates of point e_i
(x_j^s, y_j^s, h_j^s)	-	Coordinates of sensor s_j
r_s^j	-	Range of sensor s_j
α_j	-	Angle which defines the sensor orientation about the Z-axis
θ_j	-	Pan angle of the sensor about the Z-axis
β_j	-	Angle which defines the sensor orientation about the X-axis
ϵ_j	-	Tilt angle of the sensor about the X-axis

It should be noted that the presented work is formulated using the Matlab coordinate system (Positive Z-axis upwards, positive X-axis out of the screen). The LoS method is based on linear regression to find intersecting obstacles between the sensor and the point to be observed.

3.3 Proposed Coverage and Visibility Model

The proposed sensor model is inspired by [35] and [13]. It assumes binary coverage and obstacles are taken into account as opposed to much of the previous work [5, 9, 33, 36, 37].

The considered sensor type in this thesis is the camera, therefore the sensor model aims to resemble the camera. The field of view of a camera (FOV) is computed from the focal length and image size, and it is assumed that the vertical FOV is equal to the horizontal FOV. This assumption may be slightly inaccurate for some specific sensors, but introducing a difference in the vertical and horizontal FOV requires minor code changes and can be done without any complications. Further, both radial and tangential distortion are neglected, while assuming a rectilinear image. This assumption is realistic with the lens technology available today, together with proper camera calibration tools. The vertical and horizontal field of view for the proposed model is shown in Fig. 3.5.

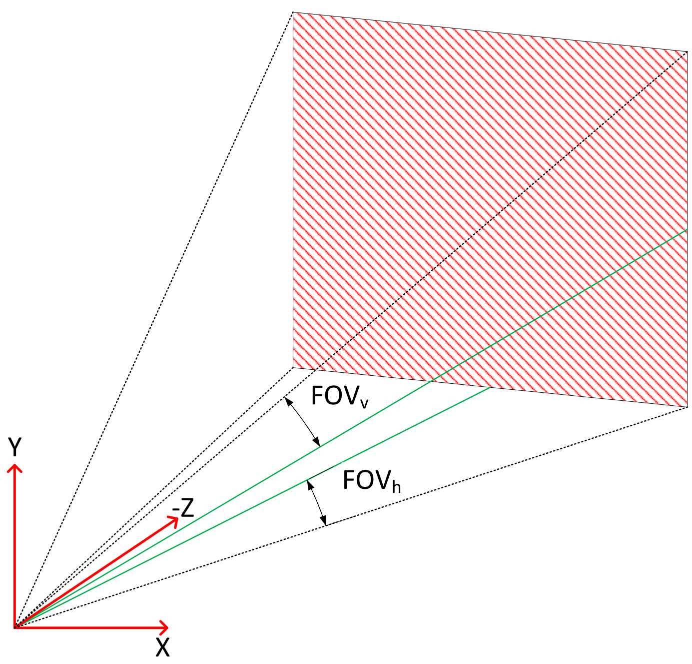


Figure 3.5: Horizontal and Vertical Field of View

A sensor, k, is defined by a 5-tuple $[\mathbf{P}_k, f_k, \alpha_k, \theta_k, r_k]$.

where:

$\mathbf{P}_k$	-	Sensor position (x_k, y_k, z_k)
f_k	-	Field of view of the sensor defined from the mid-line.
α_k	-	Sensor pan angle (Rotation about y-axis)
β_k	-	Sensor tilt angle (Rotation about z-axis)
r_k	-	Sensing range

The rotation about the x-axis of the sensor is not taken into account since most cameras are limited to pan and tilt movement. If $\alpha_k = \beta_k = 0$, the direction vector of the sensor is along the X-axis. The desired sensor orientation is realized by $RotY(\alpha_k) \cdot RotZ(\beta_k)$, where the coordinate system is the moving local coordinate system of the sensor such that the sensing direction is always along the local x-axis. The pan and tilt angles are shown in Fig. 3.6 along with the rotational planes.

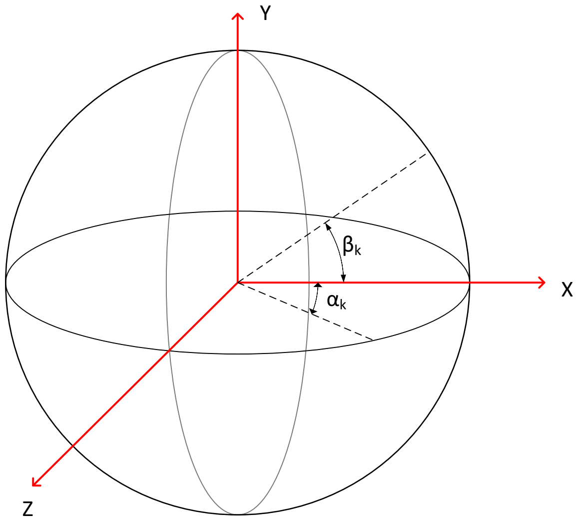


Figure 3.6: Camera Pan and Tilt Rotations

A point i (x_i, y_i, z_i) is said to be covered by a sensor k if and only if the following constraints are satisfied:

- $L_i^k \leq r_k$ - The Euclidean distance from sensor k to point i is smaller than, or equal to, the sensing range of sensor k.
- $\angle_{xz}(\theta_i^k - \alpha_k) \in (-f_k, f_k)$ - The angle between sensor k and point i in the xz-plane is in the field of view f_k of the α_k oriented sensor.
- $\angle_{xy}(\epsilon_i^k - \beta_k) \in (-f_k, f_k)$ - The angle between sensor k and point i in the xy-plane is in the field of view f_k of the β_k oriented sensor.
- $v_{i,j}^k = 1$ - Point i is visible from sensor k with respect to obstacle j. The algorithm used to compute visibility is described in Alg. 1. $v_{i,j}^k$ should be equal to 1 $\forall j$ for a point i to be visible from k.

The following equations apply for the mentioned constraints:

$$L_i^k = ||\mathbf{P}_k - \mathbf{P}_i|| = \sqrt{(x_k - x_i)^2 + (y_k - y_i)^2 + (z_k - z_i)^2} \tag{3.8}$$

$$\theta_i^k = \text{atan2}(z_i - z_k, x_i - x_k) \tag{3.9}$$

$$\epsilon_i^k = \text{atan2}(y_i - y_k, L_i^k) \tag{3.10}$$

The constraints can then be formulated as a function:

$$C_i^k = f[(L_i^k), (\theta_i^k), (\epsilon_i^k), (v_{i,j}^k)] \tag{3.11}$$

Algorithm 1: Visibility Algorithm

```
Input:
Obsx, Obsy, Obsz - Obstacle data points array
x_k, y_k, z_k - Sensor position
f_k, α_k, β_k - Sensor parameters
thr - Angular threshold
i - The index of the data point to be checked whether is visible or not
Result:
v_{i,j}^k - Boolean visibility variable
Data:
for j = 1:length(Obsx) do
    x_j = Obsx(j);
    y_j = Obsy(j);
    z_j = Obsz(j);
    L_j = ||P_j − P_k||;
    θ_j = atan2(z_j − z_k, x_j − x_k);
    ε_j = atan2(y_j − y_k, L_j);
    if (θ_j − α_k) ∈ (−f_k, f_k) then
        if |θ_i^k − θ_j| < thr then
            if (ε_j − β_k) ∈ (−f_k, f_k) then
                if |ε_i^k − ε_j| < thr then
                    if L_j < L_i^k then
                        v_{i,j}^k = 0;
                    else
                        v_{i,j}^k = 1;
                    end
                else
                    v_{i,j}^k = 1;
                end
            else
                v_{i,j}^k = 1;
            end
        else
            v_{i,j}^k = 1;
        end
    else
        v_{i,j}^k = 1;
    end
    if v_{i,j}^k == 0 then
        Point i is blocked from sensor k by obstacle j
    end
end
```

3.4 Environmental Model

The environment is defined in this thesis as the discrete static volume containing 3D grid points which should be covered by the sensor(s). Each point in the grid represents a voxel in the real scene. The grid point is the center point of the voxel. The voxel is said to be covered if the corresponding data point is within the sensing range and visible from one or more of the sensors placed in the environment.

Each point is annotated according to what it represents in the environment specified by the user:

0. Obstacle.
1. Data point to be observed by minimum one sensor.
2. Data point to be observed by minimum two sensors.
3. Data point to be observed by minimum three sensors.
4. Data point that does not need to be observed by any sensors.

The annotations are used to determine if a point is either an obstacle or a Region of Interest which allow for *k-coverage* functionality. This is often desirable, and in some cases required, for certain areas of the scene. A maximum of 3 sensors is said to be required to cover any grid point in a *k-coverage* region of interest, but this is easily extendable to a larger number. Obstacles are static, rigid objects which block the sensor line of sight if it is within the sensing range. Complicated obstacle models, e.g. cabinets where the content can only be seen from one side, are not considered. Also, transparent obstacles are not taken into account. If a closed chain of data points is defined as an obstacle, the content that lies within this chain is not possible to cover by definition in this thesis. The environment is imported and defined in the User Interface to determine the constraints for the optimization problem.

By changing the voxel size, the discretization accuracy can be modified. The user should be able to alter the environmental model by specifying Regions of Interest with *k-coverage* requirements, voxel size and possible sensor positions (along specific straight lines in the environment). This is important to make the solution applicable in a variety of environments with different problem specifications.

3.5 Matlab Graphical User Interface

A User Interface (UI) is a program that enables the user to communicate with the system, often a machine, device or program, to perform a specific task. User Interfaces are also known as HMIs (Human Machine Interface) or MMIs (Man-Machine Interface), and are often graphical (GUI). The UI is defined from the input to the output. The user specifies the input required to manipulate the program to the desired output which is the result of the actions and decisions made by the user.

The User Interface should help the user with decision-making by providing the relevant information in a simple, yet understandable manner. For most UIs to be usable in practice, they should include a guide, or user manual, to aid the user in making decisions and also to interpret the different messages provided by the UI. If users are not accommodated properly, many users can lose interest and feel that the program is too complex or cumbersome. Usability is the most important aspect of User Interface design. Other important aspects to consider when developing a UI are [38, 39]:

- **Conciseness:**

  It is of high importance that the information provided to the user is concise and limited to only the information relevant to perform specific tasks. Also, the result of any action should be as close to the intended outcome as possible.

- **Consistency:**

  For users to be able to get an understanding of how the program works, it is essential to be consistent with the coding style, notation, and comments. If users should troubleshoot the UI themselves, and the code is inconsistent with notations, etc., it is often practically impossible to troubleshoot for anyone except the developer.

- **Understandability:**

  If the user does not understand the software, it is not usable. A well-documented user manual can provide understandability, as well as a defined workflow for the user to follow.

- **Aesthetics:**

  Colors and graphics should be used to provide useful information only. An aesthetically pleasing UI is attractive and draws attention.

- **Robustness:**

  The User Interface should not have any random aspects to the produced input. For the same inputs and decisions, the output must always be the same.

Matlab is chosen as the programming platform for the UI. The main reasons for using Matlab is that it includes an extensive library of functions available for the software designer. The Matlab support for VRML models and patch objects makes the handling and interfacing of 3D objects more accessible than many other programs where such support is limited. Also, the programming language is easy to understand and interpret for other developers who desire to either troubleshoot or implement new functionality in the UI.

The Matlab Graphical User Interface (GUI) is a vital part of this thesis. As previously stated, much of the work in the literature is limited to case-specific solution contributing little to provide a consensus in both environment generation and optimization solutions. Therefore, the GUI is made to provide a common platform for problems to be formulated on, thus enabling optimization programs to have the same inputs.

There are three main methods of designing GUIs in Matlab:

1. Matlab App Designer
2. Matlab GUIDE
3. Creating Matlab GUI Programmatically

Both the App Designer and GUIDE are interfaced solutions for creating custom User Interfaces with pre-defined UI objects such as sliders and buttons. This provides easier solutions for users that want to develop UIs. In this thesis, the third option is chosen to have more control over the workflow and design of the User Interface. Matlab has built-in functions for GUIs such as dialog boxes, buttons, and panels.

Callbacks are the functions that are executed when the user does any pre-defined function in the User Interface such as clicking on a menu item. Callback functions are then executed to produce the desired result of the action taken.

3.6 Optimization

In both science and everyday life, optimization is an important tool for humans, nature and machines. Optimization is related to making one or several decisions, whether it is to determine which material combination that provides the best characteristics in manufacturing, distribution of taxis to minimize the waiting time for customers, or choosing where to place cameras in a given environment. In this section, an introduction to optimization will be given to provide the relevant theory for the optimization in this thesis. For the interested reader, it is encouraged to visit one or several of the available textbooks [40–42]. Firstly, some fundamentally important notations will be discussed, namely *objective function, decision variables, constraints, feasible solution* and *local/global optimum.*

Optimization, in mathematics, is the selection of the best *decision variables* with regards to a set of given *constraints* for problems formulated mathematically. The *optimal solution* is, in most problems, either the maximum or minimum of the *objective function.* A simple optimization example is to minimize a real function with one adjustable variable, such as

$$\min_{x\in[0.1,1.5]} \quad f(x) = \frac{cos(5 \cdot \pi \cdot x)}{x} \tag{3.12}$$

Which translates to choosing the value of x that minimizes the objective function $f(x)$, where x is *constrained* to be any value between 0.1 and 1.5. For this given problem, the *optimal solution* is the *minimum* of the *objective function.* Optimization problems can often become more complicated than the presented problem, which can be solved intuitively without complicated algorithms. Algorithms are the method used to find the *optimal solution* of the given problem. Since optimization problems can take numerous different forms, there are several algorithms to be used for different problem formulations.

For almost all optimization problems, there can be several solutions x^* which can be so-called *local optima.* Fig. 3.7 shows Eq. 3.12 in the specified range:

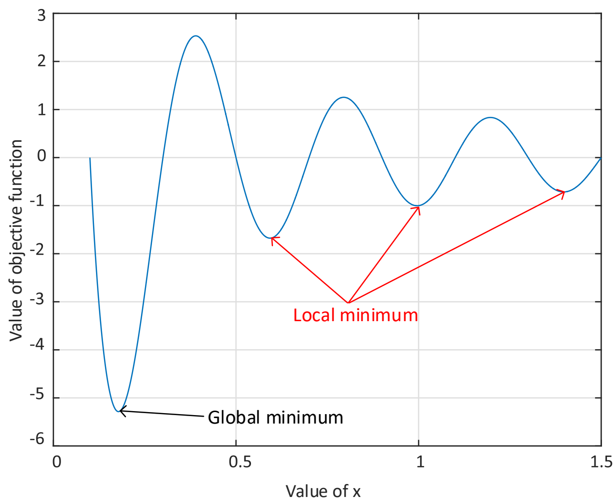


Figure 3.7: Objective Function with Multiple Minima

From Fig. 3.7, the *global minimum* can be identified easily by investigating the graph of the function. The solution x^* is a *global minimum* if $f(x^*) \leq f(x)\ \forall\ x$ inside the *feasible* set $\mathbb{S} \subseteq \mathbb{R}$. With the same notation, a solution x^* is a *local minimum* $f(x^*) \leq f(x)\ \forall\ x$ inside a neighbourhood of x^*.

A *feasible* solution is a solution where the desicion variables lie inside the *constraints*. For Eq. 3.12, the solutions that lies inside the *feasible* subset $\mathbb{S} = (0.1, 1.5) \subseteq \mathbb{R}$ are *feasible* solutions.

3.6.1 Optimization Categories

Optimization problems can be classified in several ways. This section aims to identify the distinction between continuous and discrete problems, constrained and unconstrained problems and convex optimization. Finally, problem complexity will be briefly presented.

Continuous and Discrete Optimization

When the optimization parameters can take any value inside a given constraint, the optimization problem is called a continuous problem. For continuous problems, an infinite number of *feasible* solutions exist.

A branch of mathematical optimization is discrete optimization which covers optimization problems where the variables are restricted to be discrete. These problems are often called combinatorial optimization problems. For many problems, it is practically impossible for variables to have anything else than integer or binary values. One example is to choose the number of antennas needed to cover a given area with adequate signal strength. In this example, the decision variable must take a non-negative integer value which can be mathematically formulated as $x^* \in \mathbb{Z}^{>}$. Problems, where the decision variables must have the values of integers, are called *integer programming problems* (IP). Binary decision variables do also often appear, and a thought case could be whether or not factories should be shut down or maintained in a set of cities to achieve the highest possible profit in a company. The decision variables must then take binary values and can be mathematically formulated as $x^* \in (0, 1)$. Such problems are called *binary programming problems* (BP).

In some problems, some variables are limited to take discrete values, but others may take any value. Such problems are called *mixed integer programming problems* (MIP).

A general distinction between continuous and discrete optimization problems is given in [40], where the discrete optimization problems are identified as where the decision variable(s) must be chosen from a finite set. These sets, however, tend to get very large. On the other hand, the continuous problems contain an infinite number of possible values for the decision variables, such as when they are limited to real numbers only.

Often, continuous problems are easier to solve than discrete problems. There are several reasons to support this claim:

- Gradient methods are not applicable in most discrete problem formulations.
- For most discrete problems, all combinations or permutations of feasible solutions have to be evaluated to guarantee the global optimum. This makes many discrete optimization tasks computationally hard.
- In many continuous problems, the solution often lies nearby the bounds of the constraints, and therefore the algorithm can often start searching in the vicinity of the bounds. Also, when the gradient can be computed at all points for a smooth function, it is easier to gather information regarding the neighborhood of a point in continuous problems. Since the discrete function is not smooth, this rule-of-thumb does not apply in discrete optimization.

Constrained and Unconstrained Optimization

Problems with no constraints on the decision variables are classified as unconstrained optimization problems. If the feasible set is $\mathbb{S} = \mathbb{R}^n$ for n decision variables, the problem is said to be unconstrained. Unconstrained optimization is used in several applications, e.g. pattern recognition [43] and linear regression [41].

However, most problems are constrained in one or several ways. Generally, for continuous optimization problems, the constraints have the form of either being *equality- or inequality-constrained*. The Rosenbrock function [44], commonly used to test optimization algorithms, is given as an optimization problem in Eq. 3.13.

$$\min_{\boldsymbol{x} \in \mathbb{R}^2} f(\boldsymbol{x}) = f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2 \tag{3.13}$$

With the following constraints:

$$x_1 - x_2 = 0 \tag{3.14}$$
$$x_1 + x_2 \geq 0 \tag{3.15}$$

Eq. 3.14 is called an *equality constraint*, and Eq. 3.15 is called an *inequality constraint*. The global minimum of Eq. 3.13 is $\boldsymbol{x}^* = (x_1, x_2) = (1, 1)$, which is feasible since it satisfies both constraints.

A particular case of constrained optimization is *linear programming*. An optimization task is defined as a linear program when both the objective function and all constraints are linear functions of the decision variable. On the other hand, when at least one of the constraints or objective function is nonlinear, the problem is defined as a *nonlinear program*.

Convexity

Convexity is an important aspect of optimization. If a problem is convex, it is generally easier to solve [40]. For a problem to be convex, both the objective function and the constraint function must be convex. A 1D function $f(x)$ is convex if and only if [45]:

$$\forall x_1, x_2 \in \mathbb{S}, \forall \alpha \in [0, 1] : f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha) f(x_2) \tag{3.16}$$

Where the domain of the function $\mathbb{S}$ is a convex set. The set $\mathbb{S} \in \mathbb{R}^n$ is said to be convex if a straight line drawn between each possible pair of points in $\mathbb{S}$ lies entirely inside the set [40]. Eq. 3.16 can only determine convexity for 1D functions. For functions of larger dimensions, convexity is often checked by the method described of drawing lines inside the set.

An example of a convex function is the quadratic functions $f(x) = x^2$.

Complexity Classes

Decision problems (problems that can be answered with a yes/no answer) are often classified into different complexity classes. Complexity classes are highly relevant in discrete optimization tasks since they can provide information about how complicated the problem is, and possibly how it can be solved. A short presentation of complexity classes related to time will be given here. Mainly, the P and NP classes will be presented.

A problem is in complexity class P if it can be solved in polynomial time with a deterministic Turing machine. Problems in P are often easy, and for most problems, there have been developed algorithms that solve the problems in polynomial time. Polynomial time complexity is often written as $O(n^k)$, which denotes the computational time required ($O()$). $O(n^k)$ means that a problem of input complexity n and a positive constant k can be solved in a predictable amount of time calculated from the polynomial function n^k.

The NP class is the set of problems that can not be solved by a deterministic Turing machine in polynomial time (but can be verified by a nondeterministic Turing machine in polynomial time). All problems in P are also in NP ($P \subseteq NP$).

NP-hard problems more difficult than NP problems, since some NP-hard problems may not be in NP. They are classified as problems where an algorithm for solving the problem is reducible (in polynomial time) to an algorithm that can solve all NP problems.

When considering decision problems, the term NP-complete is fundamental. A decision problem is NP-complete if the corresponding optimization problem is in both the NP and NP-hard set.

In practice, almost all combinatorial optimization problems are either NP-complete or NP-hard. This is an essential aspect of such problems, indicating the computational complexity. Fig. 3.8 shows the relationship between the described complexity classes.

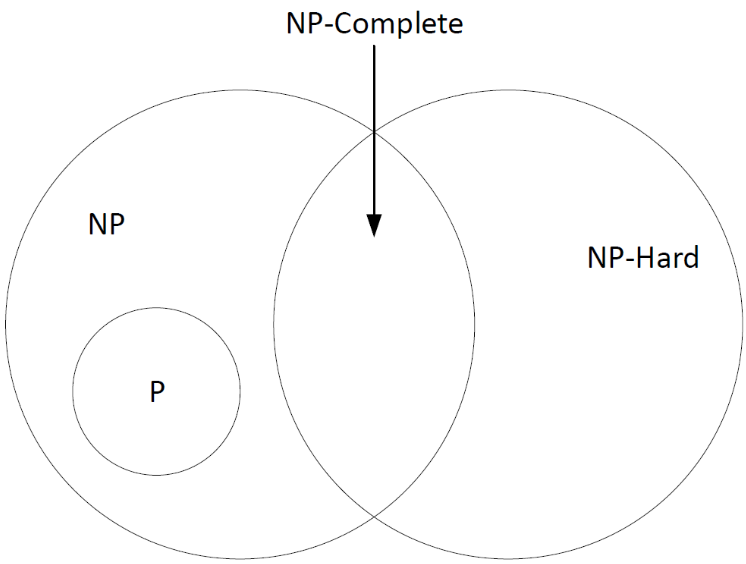


Figure 3.8: Visualization of Complexity Classes

3.7 Combinatorial Algorithms

Algorithms are computer programs written in a language that can be understood by the computer to solve a problem given a set of inputs. From these inputs, some instructions are provided on how to solve the given problem. The output of the algorithm is the solution to the problem. In this section, combinatorial algorithms will be presented.

Combinatorial problems arise in many practical optimizations when the search-space is discrete and often large, and the solution can be measured using an objective function. When the search space gets especially large, it is not possible to find the optimal solution to combinatorial problems in a reasonable amount of time, since the problems tend to be NP-hard/NP-complete. [46]. Combinatorial problems contain a large variety of different problems, including the following [47]:

- Knapsack problem
- Traveling salesman problem
- Set-cover problem
- Facility location problem
- Job scheduling problem

Since the problems are so complicated, approximation algorithms are often used to find an acceptable solution within a reasonable amount of time. In this section, both heuristics and meta-heuristics approximation algorithms will be presented as well as exact algorithm programming methods.

3.7.1 Heuristics

Heuristic algorithms are commonly used in combinatorial optimization. These algorithms are not guaranteed to provide the optimal solution but find a satisfactory answer in a reasonable amount of time. These algorithms take advantage of the knowledge of the programmer about the given task. Such knowledge may be a rule-of-thumb, an educated guess or even common sense. The algorithms are often developed for solving one particular problem. Since the algorithms need to be fast, they are often inspired by the Greedy Algorithm.

A Greedy Algorithm is defined in this thesis as an algorithm that finds a solution based on the best solution at each iteration. In most problems, this will provide a sub-optimal solution, making it a heuristic algorithm. Since the search space is reduced at each iteration, the Greedy Algorithm converges to a solution very quickly.

3.7.2 Meta-heuristics

Where heuristic algorithms tend to be made for one specific problem, meta-heuristics are, generally, more problem-independent. Most meta-heuristic algorithms are inspired by natural processes since these tend to reach an equilibrium, which is often optimal. Nature inspires many computational theories and methods since processes in nature often can be treated as computations [48]. One of the main problems for heuristic algorithms is that they tend to get stuck in local minima. Most meta-heuristic algorithms have features to avoid this problem. A conventional method of preventing such minima is by doing a neighborhood search [49].

Examples of methods inspired by nature are:

- Simulated Annealing [50]
- Particle Swarm Optimization [51]
- Genetic Algorithms [52]
- Artificial Neural Networks [53]
- Tabu search [54]

In this section, the simulated annealing (SA) and genetic algorithm (GA) algorithms will be briefly presented. For the interested reader, [48,49] are comprehensive sources of information on meta-heuristics.

Simulated Annealing (SA)

The SA algorithm is inspired by the annealing process in metallurgy which is the process of heating a metal to a specific temperature, and after a given period at that temperature, the metal is cooled down slowly to allow it to get the desired properties. It is often done to soften a metal or improve the conductivity. It is a popular method commercially because it can restore the ductility of metals that have been strain-hardened [55].

By introducing a probability of accepting a worse solution than the current best, the algorithm aims to avoid being stuck in local minima. The objective function is evaluated by an *energy function*, E, and the temperature is denoted t. The change in energy in the current iteration is denoted ΔE. The acceptance probability is then described using Eq. 3.17.

$$e^{-\Delta E/t} \geq r \tag{3.17}$$

where r is a randomly generated number between 0 and 1 ($r \sim U(0,1)$). A new solution is also accepted if the energy function is better than the previous best solution.

There are different ways of defining the cooling schedule of the SA algorithm, such as [56]:

- Exponential cooling schedule

$$T(k) = T_0 \cdot \alpha^t \tag{3.18}$$

  where $\alpha \in (0,1)$ is a constant that determines how rapidly the temperature decreases.

- Linear cooling schedule

$$T(t) = T_0 - \eta t \tag{3.19}$$

Where η is constant.

- Logarithmic cooling schedule

$$T(t) = \frac{c}{\log(t+d)} \tag{3.20}$$

Where c, d are constants. In many algorithms d is equal to one, and c is often a large constant.

The cooling scheme is very important in Simulated Annealing since it greatly influences the way the algorithm converges. The general SA algorithm for a problem where the energy function should be minimized can be seen in Alg. 2.

Algorithm 2: Simulated Annealing Algorithm

```
Input:
K_max : Maximum number of iterations
T_f : Final temperature
Ω : System to be optimized
Data:
while T > T_f do
    while K < K_max do
        Do a random change to the system Ω, resulting in a new system Ω′
        Evaluate the change in the energy function ΔE for the new system
        Generate a random number r between zero and one
        if ΔE ≤ 0 then
            Ω = Ω′
        else
            if e^(−ΔE/t) ≥ r then
                Ω = Ω′
            end
        end
        K = K + 1
    end
    Use a cooling schedule to set the new temperature T
end
```

Genetic Algorithms (GA)

The Genetic Algorithm is among the most popular algorithms in combinatorial problems, and it is a type of evolutionary algorithms which are the most influential meta-heuristics for optimization [48]. Where the SA Algorithm only has one solution that is altered at each iteration, the GA has a population of several solutions. There are several reasons to why this algorithm has gained so much interest, e.g. since the GA is:

- Largely parallel
- Suitable for large, complicated problems
- Guaranteed to find very good or even globally optimal solutions
- Easy to adapt to different problems

Compared to the SA Algorithm, the GA is much faster for problems with a large search space. Before presenting the GA, some concepts from natural evolution will be briefly introduced:

Mutation is the genetic operator that randomly alters the genetic material (often without any practical effect). It is important in genetic algorithms since it introduces a random effect which aids in avoiding local minima.

Crossover is the genetic operation that first breaks two parent chromosomes and then reconnects them, creating children chromosomes that inherit features from both parents. In GA, this can be done by selecting common traits between two parents and passing these traits onto the next generation, resulting in a child with the best features of both parents.

Selection is done by evaluating the *fitness function* which describes the fitness of an individual and then selecting individuals by a selection scheme based on these fitness function values.

The general idea of a Genetic Algorithm is given in Alg. 3.

Algorithm 3: Genetic Algorithm

Input:
$\mathbb{P}(0)$: Initial population
Data:
while *stopping criterion is not met* **do**
  Evaluate the fitness function of each individual in the population $\mathbb{P}(k)$
  Based on the selection scheme, choose the best individuals as parents for the next generation
  Apply crossover to the parents, to generate the offspring
  Mutate some of the offspring with a probability factor
  Generate the new generation $\mathbb{P}(k+1)$
  $k = k + 1$
end

The stopping criterion is highly problem-dependent, and for many problems, there could be several stopping criteria, such as:

- Maximum number of generations
- Convergence limits
- Changes in a fitness function

When designing Genetic Algorithms, the programmer needs to consider several factors that affect the results of the algorithm. Firstly, the fitness function must be designed; then a selection scheme must be made to choose which individuals to select as parents for the next generations. Then, both mutation and crossover methods must be made.

Determining appropriate schemes in the GA can be a difficult task since it requires considerable knowledge of the problem. Also, multiple aspects need to be considered for the algorithm to work in the best possible way, such as maintaining a random aspect while preserving a sufficiently high selection pressure for the GA to converge to the best possible solution.

3.7.3 Exact Algorithms

Contrary to the heuristic and meta-heuristic algorithms, the exact algorithm guarantees to produce the optimal results. For large scale problems, the trade-off is computational time. Since the gradient method is impossible to use in combinatorial problems, every feasible solution must be investigated. This is usually done by listing either all possible combinations, permutations or permutations with repetition depending on the problem formulation.

$$C(n,k) = \frac{n!}{k!(n-k)!} \tag{3.21}$$

$$P(n,k) = \frac{n!}{(n-k)!} \tag{3.22}$$

$$P_r(n,k) = n^k \tag{3.23}$$

Eq. 3.21 shows how many combinations are needed for a problem with n possibilities, and k number of variables. Eq. 3.22 shows the number of possible permutations with n possibilities and k variables, and 3.23 shows the number of permutations if repetition is allowed. As can be seen from the equations, these numbers rapidly increase as the size of the problem increases. Tab. 3.1 shows the rapid increase of the equations above.

Table 3.1: Increase of Combinatorial Equations with n=20

k Value	**1**	**2**	**3**	**4**	**5**
Combinations	20	190	1140	4845	15504
Permutations	20	380	6840	116280	1860480
Permutations with Repetition	20	400	8000	160000	3200000

3.8 Set-Cover Problem

In this section, the Set-Cover Problem (SCP) is presented as an introductory example to combinatorial optimization. A Greedy Algorithm is used to find an approximate solution, which will be proved to be sub-optimal for a given problem. The SCP is chosen due to its similarities to the discrete sensor placement problem. An exact algorithm will also be used to determine the optimal solution.

The Set-Cover Problem is one of the most popular and traditional problems in combinatorial optimization. It is applicable in many practical applications, such as [57]:

- Vehicle routing
- Facility location allocation
- Crew scheduling
- Distributing of broadcasting frequencies

The decision variant of the SCP was proven to be NP-complete by Karp in 1972 [58], whereas the optimization variant is NP-hard [59]. There have been developed numerous heuristics for the SCP, many of which are also relevant for the Sensor Placement Problem due to the similarity between the two problems. The SCP can be formulated in the following way:

Given a set $\mathbb{X}$, called the universe, and a number of subsets $\mathbb{S} = \{s_1, s_2, \ldots, s_n\} \subseteq \mathbb{X}$, find the minimal set $\mathbb{Z}$ that includes one or more subsets of $\mathbb{S}$ and $\mathbb{Z} \subseteq \mathbb{X}$, such that:

$$\bigcup_{i=1}^{k} \mathbb{Z}_i = \mathbb{X} \tag{3.24}$$

Which translates as minimizing the number of subsets required to cover the whole universe, where k is the number of subsets in $\mathbb{Z}$. The objective function to minimize is then the number of subsets. By rewriting the problem, given a set of indices with the same length as numbers of subsets, $\mathbb{C} = \{1, 2, \ldots, n\}$, where each element can be either zero or one. If $\mathbb{C} = \{0, 0, 1\}$, subset number three is placed in $\mathbb{Y}$ where $\mathbb{Y} \subseteq \mathbb{C}$.

The optimization problem could then be formulated as

$$\min |\mathbb{C}| = \sum_{j=1}^{m} C_j \tag{3.25}$$

subject to

$$\bigcup_{i=1}^{n} \mathbb{Y}_i = \mathbb{X} \tag{3.26}$$

Example:

A universe is given in Fig. 3.9 as a grid, where the subsets are also indicated.

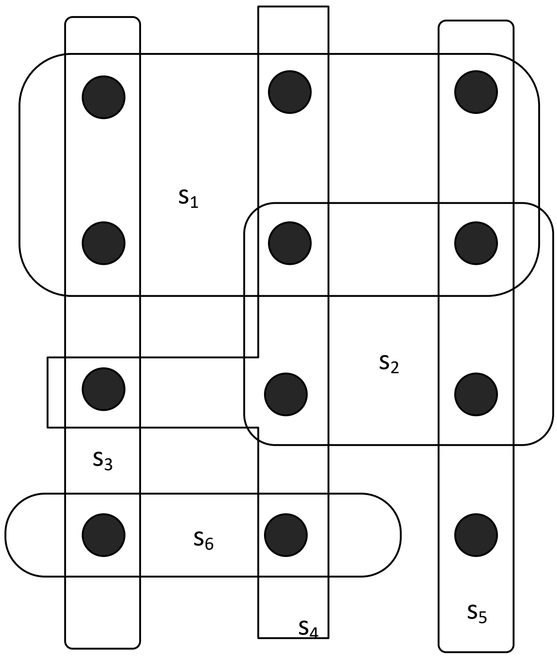


Figure 3.9: Set Cover Problem Example [1]

The value of the optimal set can be found intuitively by looking at the figure as $\mathbb{C}_{opt} = \{0, 0, 1, 1, 1, 0\}$ which translates to subsets (s_3, s_4, s_5). The objective function is then $|\mathbb{C}|_{opt} = 3$. All subsets can be listed in a table, such as Tab. 3.2 where the points in the universe are listed from top left (1) to bottom right (12) horizontally.

Table 3.2: Subset Coverage for Set-Cover Example

Subset	**1**	**2**	**3**	**4**	**5**	**6**
P. 1	1	0	1	0	0	0
P. 2	1	0	0	1	0	0
P. 3	1	0	0	0	1	0
P. 4	1	0	1	0	0	0
P. 5	1	1	0	1	0	0
P. 6	1	1	0	0	1	0
P. 7	0	0	1	1	0	0
P. 8	0	1	0	1	0	0
P. 9	0	1	0	0	1	0
P. 10	0	0	1	0	0	1
P. 11	0	0	0	1	0	1
P. 12	0	0	0	0	1	0

By using an OR-operation on the subsets, the combination with the fewest subsets can be found with a Brute Force Method. Since this problem is NP-hard (or NP-complete if formulated as a decision problem), the Brute Force Method is the only way of guaranteeing an optimal solution. The Brute Force Algorithm for the SCP can be seen in Alg. 4. This algorithm is shown in Matlab code in App. A.5.2.

Algorithm 4: Brute Force Algorithm for the Set-Cover Problem

Input:
$\mathbb{X}$: Universe
$\mathbb{S}$: Subsets
n_{ss} : Initial guess on number of subsets to cover the universe
Result:
$\mathbb{Y}$: Array of indices with chosen sets
Data:
while $\mathbb{X} \neq \mathbb{Y}$ **do**
    Calculate all combinations required to explore all possible solutions
    Evaluate the coverage of all possible subset combinations $\mathbb{S}^{n_{ss}}$ given n_{ss}
    Select the best combination $\mathbb{S}^{n_{ss}}_{best}$
    if $\mathbb{S}^{n_{ss}}_{best}$ *covers the whole universe* **then**
        $\mathbb{Y} = \mathbb{S}^{n_{ss}}_{best}$
    else
        Increase the number of subsets to cover the universe $n_{ss} = n_{ss} + 1$
    end
end

In practice, since the problem can be so complex, the SCP is often solved using approximation algorithms that are much faster, but produces approximate solutions. Detailed descriptions of these algorithms for the SCP can be found in the literature [59–61]. The most used approximation algorithm is a Greedy Algorithm shown in Alg. 5 [1]. Note that the notation $\{\mathbb{F}\}$ in line **7** indicates the corresponding index in $\mathbb{S}$ that is represented by $\mathbb{F}$.

Algorithm 5: Greedy Heuristic for the Set-Cover Problem

   Input:
1 $\mathbb{X}$: Universe
2 $\mathbb{S}$: Subsets
   Result:
3 $\mathbb{Y}$: Array of indices with chosen sets
   Data:
4 **while** $\mathbb{X} \neq \emptyset$ **do**
5   select the set $\mathbb{F} \subseteq \mathbb{S}$ such that $\mathbb{X} - \mathbb{F}$ is minimized
6     $\mathbb{X} = \mathbb{X} - \mathbb{F}$
7     $\mathbb{Y} = \mathbb{Y} \cup \{\mathbb{F}\}$
8 **end**

The Greedy Algorithm will, for the problem given in Fig. 3.9, choose the sets s_1, s_4, s_5, s_3, in that order, such that $\mathbb{C} = \{1, 0, 1, 1, 1, 0\}; |\mathbb{C}| = 4$. This indicates that for the given problem, the Greedy Algorithm does not provide the optimal solution since it only considers the best subset at each iteration. This algorithm can be found as a Matlab script in App. A.5.1.

3.9 The JSON File Format

The JSON (JavaScript Object Notation) file format will be used in this thesis to store the UI data and import it into C++. JSON is an intuitive format which is easy to understand and use. It is language independent, which is one of the main reasons for it being so popular. JSON files can, however, get quite extensive, e.g. compared to binary files. JSON is written for easy interpretation and understanding, whereas the binary files are made to be understood by a computer but not necessarily being readable by humans. A JSON file is recognized by its extension, which is **.json*. There are several available values for elements to take in JSON:

- Object
- Array
- String
- Number
- True/false/null

An object is enclosed by curly brackets and contains pairs of names and values. Names and values are separated by colons, and commas separate the pairs. Arrays are enclosed by squared brackets and separated by commas. An element is defined as a string if it is wrapped in double quotation marks (, *i.e.*,"*abc*"). Contrary to C-like languages, octal and hexadecimal formats are not used in JSON. Instead, values are defined by numbers, and dots specify decimal points.

A JSON example can be seen in the code snippet below.

```
{"Camera":
{
"Number of cameras": 2,
"Camera parameters": [
{"Field of view":120,"Range":8,"Cost":20},
{"Field of view":90,"Range":12,"Cost":22}
],
"Array of numbers":[1,2,3,4,5,6,7,8,9,10,11]
}
}
```

Here, a JSON object is made, called camera. This contains a set of objects. The first object stores information regarding the number of cameras. The second object is an array of objects for each camera, which specifies the camera parameters. Finally, an array of numbers is included.

Matlab includes functions for encoding and decoding JSON files [62]. This makes the encoding of Matlab objects into JSON objects straightforward. There are several available methods of parsing JSON files in C++. In this thesis, the JSON11 library [63] is used due to its simple syntax.

3.10 Parallel Programming with CUDA

Since CUDA (Compute Unified Device Architecture) was released in 2007, general-purpose computing on graphics processing units (GPGPU) has accelerated. CUDA is a programming language developed to enable programmers to utilize the parallel architecture of GPUs. GPUs are an essential part of many accelerated applications in fields such as [64]:

- Artificial intelligence
- Cars
- Drones
- Robotics
- Hashing algorithms

In this section, the fundamental theory on graphical processing units and the essential functions of the CUDA language will be presented. For the interested reader information regarding GPUs and CUDA can be found at [65, 66].

3.10.1 The Graphical Processing Unit

When NVIDIA released the first GPU in 1999 [67] the intended use was mainly to improve the graphics of computer applications, but nowadays the GPU is also used in other fields and sciences. Throughout this section, the GPU term will be understood as NVIDIA GPUs. One of the main differences between a CPU and a GPU is understood by comparing the differences in cores and memory latency. While modern day CPUs usually have 4-8 cores optimized for minimal memory latency and quickly being able to switch between different operations, GPUs have thousands of cores optimized to process as many operations as possible, simultaneously. The trade-off is the memory latency, but since operations are done in an overlapping parallel manner, the memory overhead is often not noticeable for the user.

If a CPU is a Leatherman, a GPU is a very sharp knife. You can't tighten a hex bolt with a knife, but you can definitely cut some stuff [68].

This quote highlights the main difference in usage between the GPU and the CPU. Where the CPU is versatile, and superior in doing sequential, demanding tasks, the GPU outperforms the CPU in cases where more straightforward computations need to be done numerous times, and each iteration is independent.

The primary function of the GPU is 3D video rendering, which consists of vast amounts of matrix, and floating point operations to describe coordinates, light, textures, transparency, etc. Since each pixel is independent, the calculations can be done in parallel.

The GPU is built up around a scalable array of Streaming Multiprocessors (SMs). A function that should be executed on the GPU is called a kernel function. This kernel is distributed onto a grid of blocks, where the required blocks for the given kernel operation is allocated to the available multiprocessors. Multiple thread blocks can execute at the same time on the same multiprocessor since each multiprocessor can handle hundreds of threads concurrently [65].

3.10.2 CUDA Programming

The terms *kernel, threads, blocks* and *grid* are essential to understand when programming CUDA applications.

Kernels

To execute functions in parallel, kernels are used. To specify a C++ function as a kernel, a specifier has to be used as can be seen in line **1** in the code snippet below where **__global__** specifies that *mykernel* is a kernel function. In line **8** the kernel is invoked and executed on N parallel threads where all threads are inside the same block.

```
1  __global__ void mykernel()
2  {
3      // Some function
4  }
5
6  /*
7
8  ....
9
10 */
11 int main()
12 {
13     // Define the inputs to the function
14     mykernel <<< 1,N >>>()
15 }
```

Threads

Threads are organized in blocks which again are processed by SMs. The threads can be spread out into 1D-, 2D- or 3D-arrays, where the total number of threads is limited by the available threads on the GPU. Each thread is responsible for one kernel call. The thread id can be accessed inside the kernel using the *threadIdx* function. If the threads are distributed in a 1D array, the thread number is accessed using *threadIdx.x*, whereas for 2D- and 3D-arrays the thread number for the other dimensions are accessed using *threadIdx.y* and *threadIdx.z*.

Blocks

Multiple threads are collected into blocks. On current GPUs, one block may contain up to 1024 threads. Kernels can be executed on several blocks, meaning that the total number of kernel calls available is the number of blocks times the number of threads per block. The identity of the block can be found using the *blockIdx* function.

Grid

Blocks are again grouped into a grid. The grid is handled by the GPU, distributed to a set of SMs. Each SM is responsible for one or more blocks in the grid.

The number of blocks per grid and threads per block is specified by the user in the kernel call. Inside <<< ... >>> the user can specify the number of blocks and threads per blocks using <<< numBlocks, numThreadsPerBlock >>>. Fig. 3.10 shows the hierarchy of blocks and threads inside a grid. In this example, both the blocks and threads are defined as one-dimensional.

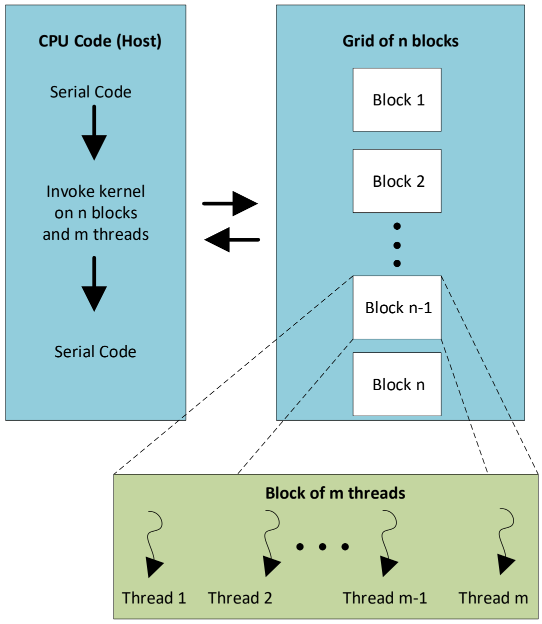


Figure 3.10: Illustration of CUDA Threads and Blocks

Memory

To reduce the slow memory transfer from the global memory of the GPU, each thread has its own dedicated memory. Also, each block has a shared memory which all threads in the block can access. Finally, all threads in the grid have access to the global memory. A visualization of the GPU memory hierarchy can be seen in Fig. 3.11.

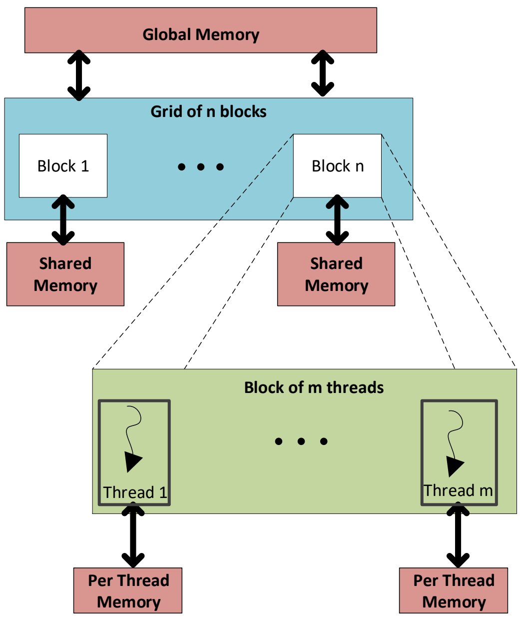


Figure 3.11: GPU Memory Hierarchy

CUDA Syntax

The CUDA language is basically C++ with extensions. NVIDIA has produced their own compiler, the *nvcc* compiler which can compile CUDA files. These are recognized as files with a **.cu* extension.

The CUDA files contain the functions for both the host (CPU) and the device (GPU) code. Also, a C++ file is made, which includes the reference to the CUDA file inside the *main* function.

The workflow of CUDA programming can be described in three steps:

1. Transfer data from the host memory to the device memory to be executed in the kernel.
2. Invoke the kernel to load the GPU program and execute it.
3. Copy results from device memory to host memory and clear device memory.

To present the key commands and provide a template for applications, a minimal CUDA example is given in Fig. 3.13. The CUDA code example fills an array where each element has the value of its corresponding index. Each thread is responsible for filling in the value of the array element corresponding to the identity of the thread. First, the CUDA header files are included along with the iostream library which allows the usage of *std::cout* for printing values to the display. Line **5-11** defines the kernel function which is invoked in line **38**. The GPU used to execute this code is a NVIDIA GTX 1080, and the grid size used in this example is defined in line **17-20**. In line **22-32**, memory is allocated on both the host and the device as well as initialization of the host array, where all elements in the array are set to zero. Before the kernel is invoked, the memory is transferred from the host to the device (line **35**), and when the kernel call is finished, the result data is transferred back from the device to the host (line **41**). To ensure that the results are correct, the first 100 values of the host array is displayed before the memory is freed on both the host and the device (line **50-51**).

The code is compiled in Linux using

nvcc -ccbin -clang-3.8 -lstdc++ cudafile.cu main.cpp -o outres

This means that *nvcc* is linked with the *clang-3.8* compiler which is responsible for compiling the C++ file. The *main.cpp* file is simply a main function which calls upon the function in the CUDA file. For the presented problem, the *main.cpp* file can be seen in Fig. 3.12.

```
void calcul();

int main()
{
    calcul();
    return 0;
}
```

Figure 3.12: CUDA Example Code: C++ file

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <iostream>
__global__ void kernel(int* data)
{
    // Get the global id of the thread
    int thid = blockDim.x*blockIdx.x + threadIdx.x;
    // Fill the array with the index of the current thread
    data[thid] = thid;
}

void calcu()
{
// Allocation, transfer and free an integer array of size 20

int n_blocks = 50;  // Number of blocks in the grid
int n_threads_per_block = 256; //Threads per block
int data_n = n_blocks*n_threads_per_block; //Number of elements
size_t size = data_n * sizeof(int); //Size (in bytes) of arrays

int* data_host = (int*)malloc(size); //Allocate host memory
int* data_dev; //Pointer to the device memory

// Fill the host array with zeros
for(int i = 0; i < data_n; i++)
{
    data_host[i] = 0;
}

// Allocate device memory
cudaMalloc(&data_dev,size);

// Transfer array from host memory to device memory
cudaMemcpy(data_dev, data_host, size, cudaMemcpyHostToDevice);

// Invoke kernel
kernel <<< n_blocks, n_threads_per_block >>> (data_dev);

// Transfer memory from device memory to host memory
cudaMemcpy(data_host, data_dev, size, cudaMemcpyDeviceToHost);

// Print the first 100 elements in the host array
for(int i = 0; i < 100 ; i++)
{
    std::cout << data_host[i] << std::endl;
}

// Free memory
cudaFree(data_dev);
free(data_host);
}
```

Figure 3.13: CUDA Example Code: CUDA file

4 User Interface, Case Studies and Results

In this chapter, the methodical work of the thesis will be presented based on the given theory in Ch. 3. The chapter is divided into case studies where different aspects of the Sensor Placement Problem is considered. For each section, the results are presented along with the methodical work.

The first section (Sec. 4.1) concerns the development and design of the User Interface. Here, the emphasis is on the methods used to fulfill the desired goals rather than how the UI should be used. A user manual is appended (App. A.2) where a description of how the program should be used can be found.

In Sec. 4.2, a case study is made to verify the output of the UI and the sensor model. Here, two sensors with fixed poses are to be placed in an environment defined in the UI. Next, both a Greedy Algorithm and a Brute Force Algorithm are presented and compared to each other to investigate the optimality of the Greedy Algorithm and the computational time for the two algorithms (Sec. 4.3).

Sec. 4.4 regards development of a Genetic Algorithm (GA) for the Sensor Placement Problem. Here, genetic operators are designed to give the best result in the Sensor Placement Problem. The determination of parameters in the algorithm is vital for producing the desired result, and an assessment of parameter tuning is given, where the mutation rate is emphasized as the most important tuneable parameter along with the population size and the number of generations. Finally, the algorithm is compared to the Brute Force Algorithm using the same tests as in Sec. 4.3.

A Brute Force Algorithm (BFA) is desirable since it is deemed to produce the optimal result. However, since the problem size increases so rapidly with increasing input size, a threshold exists where solving the problem with the presented Brute Force approach becomes practically impossible due to the required computational time. Therefore, a case study is made in Sec. 4.5 where the BFA is converted from C++ to CUDA with the purpose of increasing the mentioned threshold, making the algorithm usable for even more complex problems. The developed GPU-based Brute Force Algorithm is compared to the CPU implementation concerning computational time and to ensure that the results are the same.

In Sec. 4.6, a method of introducing k-covered Regions of Interest is presented, along with implementation details for both the BFA and GA.

Finally, all the above case studies are combined into a final test where optimal sensor placement should be conducted in a robotic test laboratory. This test includes several obstacles and a Region of Interest. First, the problem is defined in the UI, before an initial analysis is conducted using the Greedy Algorithm. With increased knowledge about the problem, the Genetic Algorithm is used to determine optimal, or at least very good, solutions to satisfy the requirements. Then, the problem is reduced to be handled by the BFA where a global optimum can be ensured. Finally, a neighborhood search is done to evaluate the neighborhood of the solutions, potentially improving the solution further.

4.1 Matlab User Interface

As formulated in the problem statement, one of the main intentions of this thesis is to develop a platform where a 3D file can be imported into a User Interface and provide the constraints for an optimization problem given this 3D model. The User Interface (UI) should include functions to define the Sensor Placement Problem in different ways depending on the intention of the user. The user should be able to:

- Define possible placement points for the sensors
- Define multiple sensors with different parameters and price
- Define regions of interest where k-coverage is required
- Define the optimization parameters to get the required accuracy and problem formulation
- Output the optimization task to a JSON file
- Visualize the optimized placement

To include all the desired functions mentioned above, the UI is made with a defined workflow:

1. Import VRML model
2. Add floor to imported model
3. Add camera parameters
4. Define placement lines
5. Define regions of interest
6. Define optimization parameters
7. Generate JSON file
8. Visualize results

Using the theory presented in Sec. 3.1, the model is converted from a VRML model to a patch object interpretable in Matlab. Often, these models are made without any floor which is added in the UI, partly for visualization but most importantly it is necessary to know the polygonal shape of the environment. It is assumed that the walls surrounding the environment are straight, i.e. the x- and z-coordinates are not dependent of the height.

The sensor model is calculated by defining the camera parameters as described in Sec. 3.3. Further, the lines where the sensors can be placed is defined as well as any Regions of Interest. Finally, the optimization accuracy is defined to generate the JSON file which can be processed in C++. As a verification option, the user can visualize the optimization result by importing the generated parameters such as camera position and pose. It should be noted that the VRML coordinate system is being used in the User Interface. Also, a standard length unit is not specified since different problems can be formulated using different units. The user should, therefore, ensure that the length unit is consistent, e.g. the sensor model and the room should correspond to each other.

By using the drop-down menus, the user can specify all details of the problem. In Fig. 4.1, the main window of the UI can be seen. This section aims to describe the different methods that have been used to develop the user interface. For a description of how to use the UI the appended user manual, seen in App. A.2, should be used.

The full source code of the UI is too extensive to append and is therefore uploaded to a GitHub repository which can be found at `https://github.com/Vegardtve/sensorplacement`. Here, the full documentation to the UI can be found, along with the VRML file which is used throughout this section. The UI is pre-defined for this environment for new users to better understand the workflow. It should be possible to perform all actions without any modifications from this repository. The code is sufficiently commented for every function and script to be understandable for any experienced Matlab programmer. The source code of the UI is also made available to enable other users to modify or improve it if any limitations or errors are found. Also, a video is made where the User Interface is demonstrated. The video can be found in either the GitHub repository or at `https://youtu.be/XAflsneC-x4`.

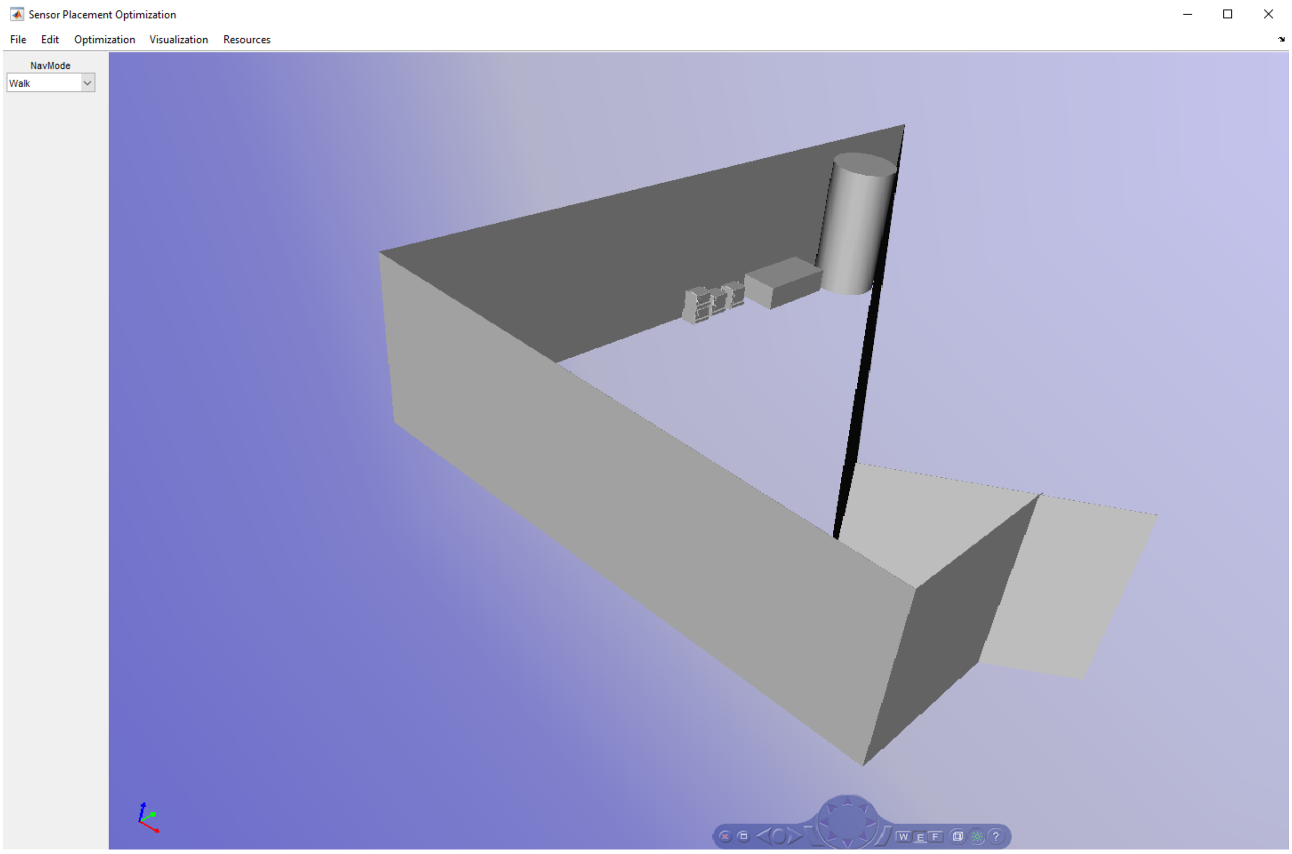


Figure 4.1: Main Window User Interface

4.1.1 Adding the Floor and Camera Parameters

When the user clicks *Add Floor*, an input window is opened. The floor is added by specifying all vertices of the polygon defining the floorplan. By using the figure tracer in Matlab, the coordinates of each vertex can be retrieved. The polygon is made at a user-specified height which must correspond with the height at floor level. Since some models are made with a negative height from the roof to the floor, no standard floor height can be defined. Fig. 4.2 shows the method of adding a floor to the VRML model in the UI.

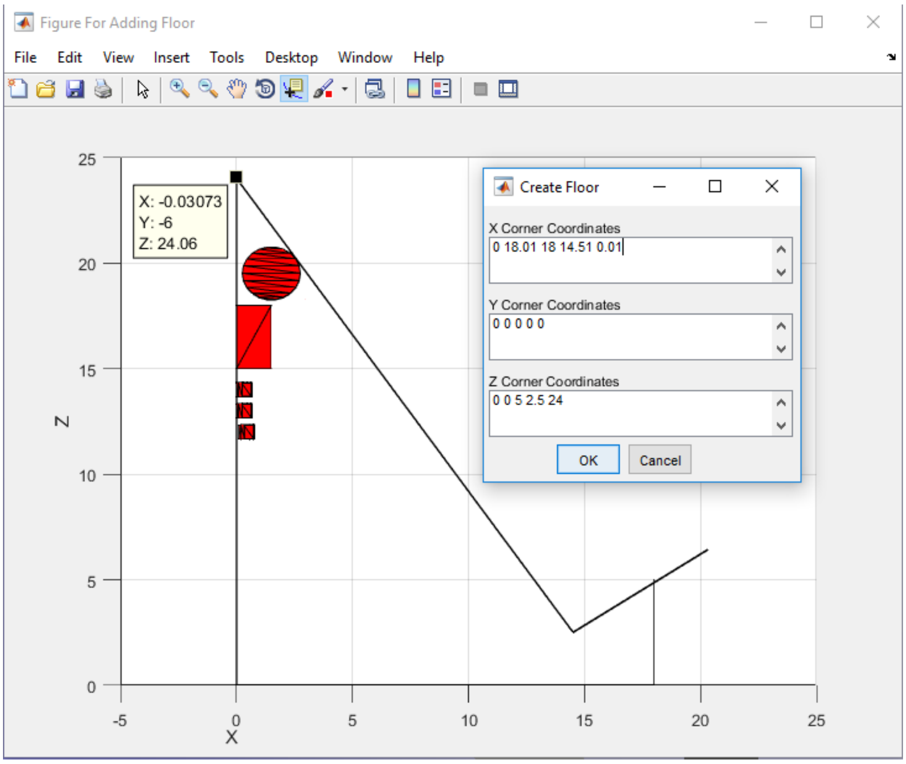
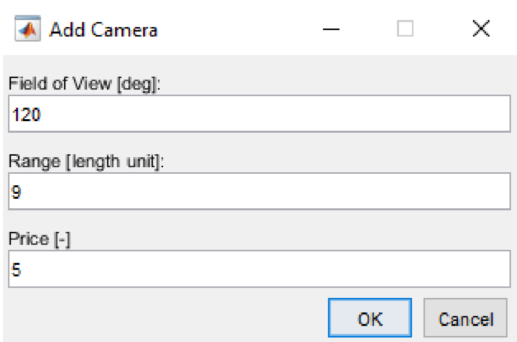
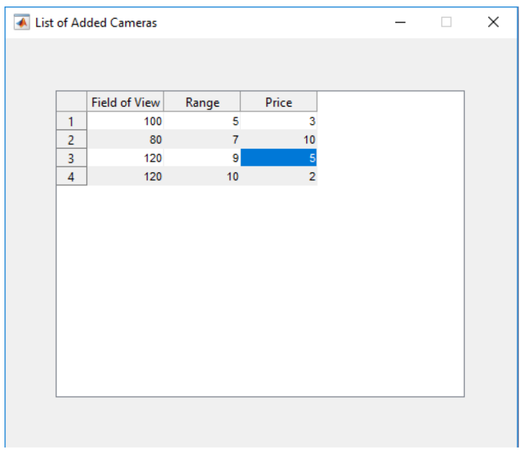


Figure 4.2: Example of Adding Floor in the User Interface

The user can add several cameras with different camera parameters. This is desirable since, in many situations, several cameras with different parameters and price are available. Cameras can be added by clicking *Add Camera*, and all added cameras can be seen using the *List of Added Cameras* button. The menus for adding sensors and displaying the added sensors can be seen in Fig. 4.3a and Fig. 4.3b, respectively.



(a) Adding a Sensor in the User Interface

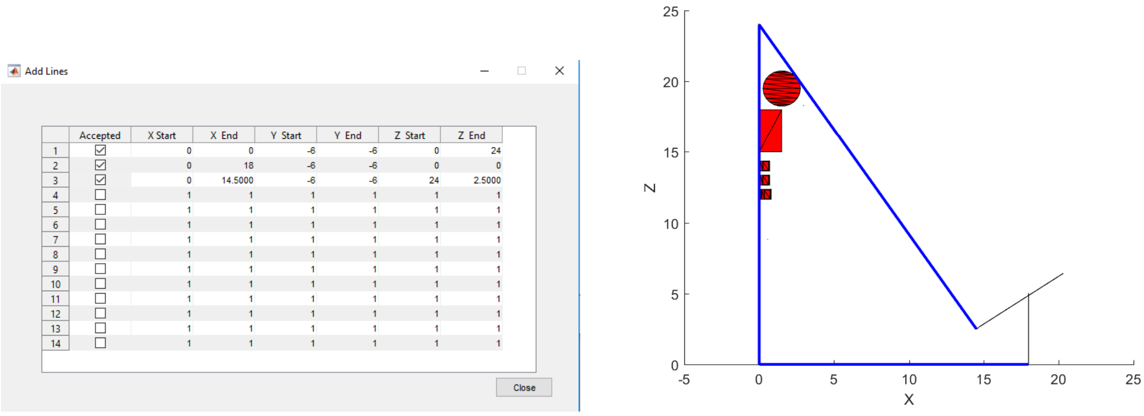


(b) List of Added Cameras

Figure 4.3: Displaying Added Sensors in the User Interface

4.1.2 Placement Lines

When the *Add Lines* item is chosen, two windows are opened. The figure window displays the patch model and can be used for tracing to get the coordinates of the desired data points. The placement lines are limited to being straight lines defined using the (x, y, z) coordinates of the starting- and end-points. Multiple placement lines can be specified, and the user can choose which lines actually to choose by checking the *Accepted* box. The accepted lines can be visualized with the *Display Placement Lines* menu item, which opens a figure window showing the placement lines together with the patch model. An example of three added lines is first defined in Fig. 4.4a, and then visualized in Fig. 4.4b which is exported from the figure window in the *Display Lines* menu item.



	Accepted	X Start	X End	Y Start	Y End	Z Start	Z End
1	☑	0	0	-6	-6	0	24
2	☑	0	18	-6	-6	0	0
3	☑	0	14.5000	-6	-6	24	2.5000
4	☐	1	1	1	1	1	1
5	☐	1	1	1	1	1	1
6	☐	1	1	1	1	1	1
7	☐	1	1	1	1	1	1
8	☐	1	1	1	1	1	1
9	☐	1	1	1	1	1	1
10	☐	1	1	1	1	1	1
11	☐	1	1	1	1	1	1
12	☐	1	1	1	1	1	1
13	☐	1	1	1	1	1	1
14	☐	1	1	1	1	1	1

(a) Example of Defining Placement Lines (b) Visualizing Added Placement Lines

Figure 4.4: Adding and Visualizing Placement Lines

4.1.3 Region of Interest

The room is discretized into equal sized voxels. Each of these voxels must be annotated with one of the following parameters, as described in Sec. 3.4:

1. Obstacle
2. User specified Region of Interest
3. 1-coverage (Standard) volume

The Region of Interest (ROI) procedure allows the user to specify volumes of special interest, which can be annotated as one of the following:

- Non-interest volume
- 2-coverage volume
- 3-coverage volume

The ROI groups are added to expand the problem to also include k-covered volumes, i.e. specific zones in the environment of either high or no importance. This expands the number of possible annotations for each cube. The procedure of generating a user-defined ROI will now be presented. A flowchart is made for better visualization of the process and can be seen in Fig. 4.5.

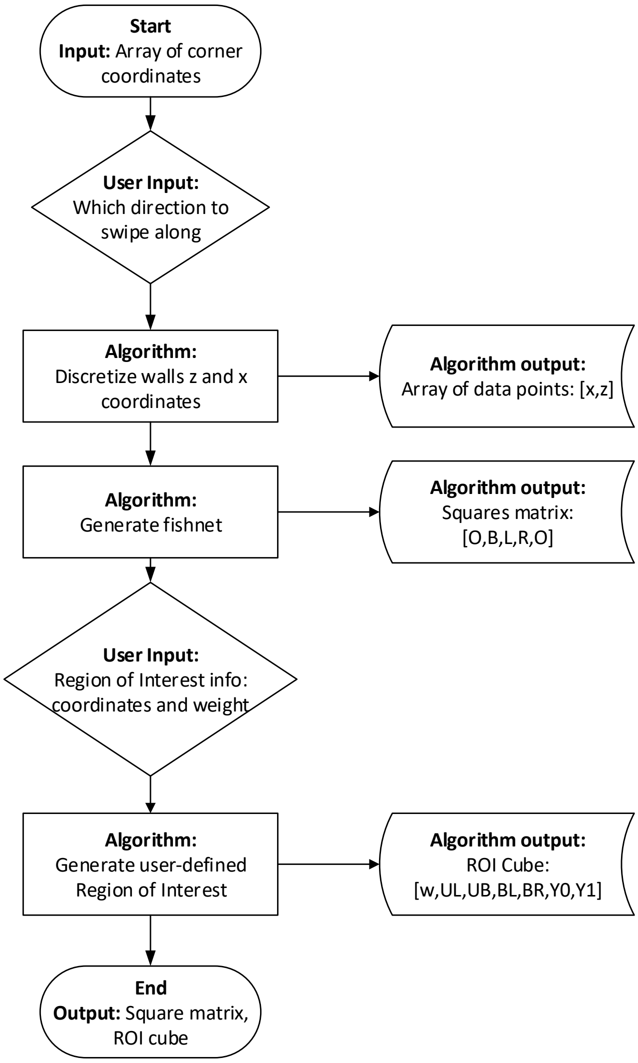


Figure 4.5: Flowchart of the Region of Interest Generation

As can be seen in Fig. 4.5, the ROI procedure consists mainly of three separate algorithms, which will be described in this section. The first two algorithms are used to discretize the floor into a fishnet, and the third algorithm generates the ROI based on user-specified inputs. Firstly, the user has to specify in which direction the fishnet generation algorithm should swipe along, either the horizontal or vertical axis. The algorithm should swipe along the horizontal axis if there are re-occurring values in the z-axis coordinates of the walls. If there are re-occurring values in the x-axis coordinates, the algorithm should swipe along the vertical axis. If neither, or both, of the conditions above are met, it generally does not matter which direction the algorithm is chosen to swipe along. This function is included to limit the number of generated voxels outside the environment.

Discretize walls z and x coordinates

The first algorithm makes data points along the edges of the polygon made out of the corners in the room. It is assumed that the corners can form a closed polygon. The algorithm is shown in Pseudocode in Alg. 6.

Algorithm 6: Discretize Walls

```
Input:
x: Array of corner x-coordinates
z: Array of corner z-coordinates
nlin: Number of data points along a line
Result:
xlin : Array of data points, x-coordinates
zlin : Array of data points, z-coordinates
Data:
xlin = [];
zlin = [];
for i = 1 : length(x) do
    if i == length(x) then
        if x(i) == x(1) then
            xl(i) = x(1);
            zl(i) = linspace(z(i),z(1),nlin);
        else
            m = (z(i) - z(1))/(x(i) - x(1));
            xl(i) = linspace(x(i),x(1),nlin);
            zl(i) = m*xl(i) - m*x(i) + z(i);
        end
    else
        m = (z(i) - z(i+1))/(x(i) - x(i+1));
        xl(i) = linspace(x(i),x(i+1),nlin);
        zl(i) = m*xl(i) - m*x(i) + z(i);
    end
    xlin = [xlin xl(i)];
    zlin = [zlin zl(i)];
end
```

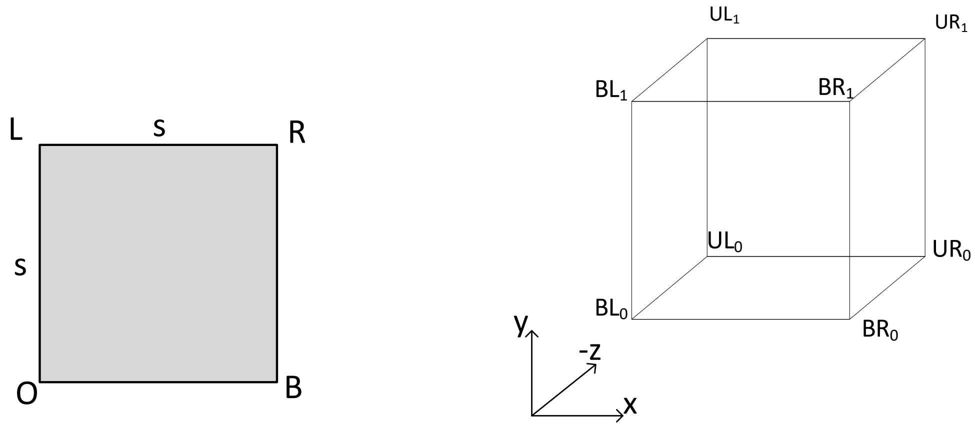
Generate fishnet

When the direction is defined, and the (x,z) data points are provided by Alg. 6, the next step in the procedure is to generate the fishnet. In Alg. 7, the algorithm for swiping along the vertical axis is shown in Pseudocode. The algorithm for generating the fishnet along the horizontal axis has minor differences but uses the same procedure. The size of each side of the voxel is set to 0.5 [length unit] as a standard. This is not available to change from the UI itself since it can produce errors that result in program failure. However, if it is desirable to change the size of the voxel, it can easily be done by altering the source code.

Algorithm 7: Fishnet Generation along Vertical Axis

```
Input:
xl - Array of linearized x-coordinates for walls
zl - Array of linearized z-coordinates for walls
s - length of square side
Result:
cubes : array of square data (x,z)
Data:
xlim = max(xl);
O = [min(xl), min(zl)];
bOK = true;
i = 0;
xd = round(xl,1);
while bOK do
    L = [O(1) ; O(2) + s];
    B = [O(1) + s; O(2)];
    R = [O(1) + s; O(2) + s];
    if R(1)>xlim then
        break
    end
    xc = find(xd==round(R(1),1));
    zv = zl(xc);
    zlim = max(zv);
    if L(1) < xlim then
        if L(2) < zlim then
            O(i) = [L(1); L(2)];
            cubes(i) = [O,L,B,R,O];
            i = i+1;
        else
            O(i) = [B(1); 0];
            cubes(i) = [O,L,B,R,O];
            i = i+1;
        end
    else
        bOK = false;
    end
end
```

The squares are defined with four coordinates: L,B,R and O as Alg. 7 shows. Each of these coordinates contains x and z components, and are shown for visualization in Fig. 4.6a.



(a) Fishnet Cube Coordinates (b) Region of Interest Example

Figure 4.6: Definition of a Region of Interest

Generate user-defined region of interest

The last algorithm utilizes the presented theory on faces and vertices in Sec. 3.1.4 to make a patch object based on the user-defined coordinates of the ROI volume. The user defines the volume using the following: BL, BR, UL, UR, Y0, Y1 and w. The first six variables define the cube, and w indicates the weight of the cube, recalling the three ROI groups described at the beginning of this section.

In Fig. 4.6b, a ROI volume is presented for visualization. The subscript of either 0 or 1 indicates the (x,z) coordinates at either Y0 or Y1, accordingly. Fig. 4.7 shows how to add a ROI in the UI which is done by choosing the *Add Region of Interest* menu item.

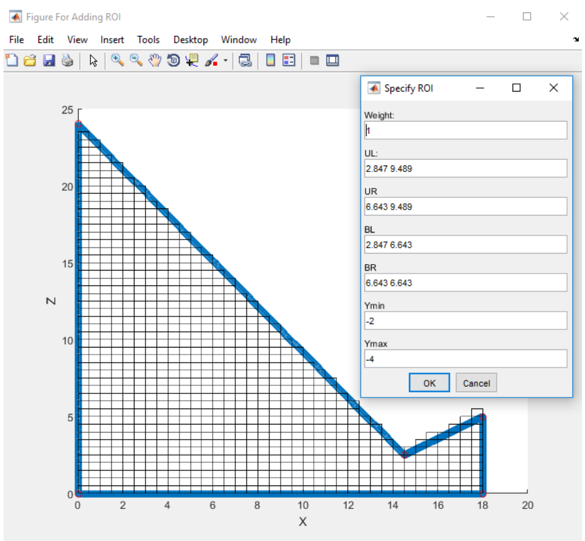
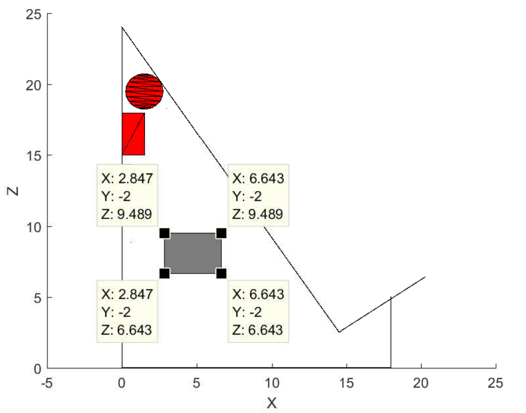
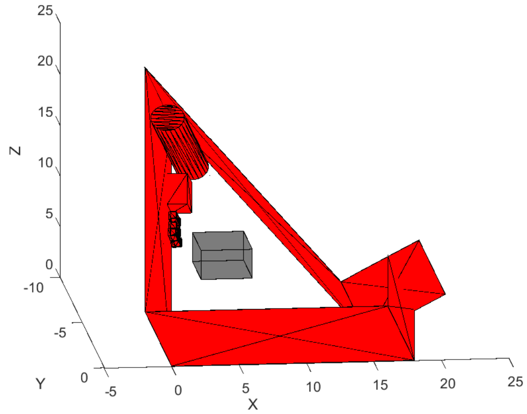


Figure 4.7: Adding a Region of Interest in the User Interface

The ROI defined in Fig. 4.7 can be seen in Fig. 4.8, which shows the result of the *Visualize Region of Interest* menu item. Fig. 4.8a shows that the ROI is added according to the coordinates specified in Fig. 4.7, and Fig. 4.8b shows the 3D representation of the ROI in the environment.



(a) Added Region of Interest 2D View in the User Interface with Coordinates



(b) Added Region of Interest 3D View in the User Interface

Figure 4.8: Region of Interest Visualization in the User Interface

4.1.4 Optimization Parameters and JSON Generation

To generate the final grid of data points, the optimization parameters need to be specified. First, the roof and floor height are defined along with the number of data points in the vertical direction. The voxel height is calculated from Eq. 4.1.

$$h_v = \frac{|h_t - h_f|}{n_v} \tag{4.1}$$

where:

h_v	-	Voxel height
h_f	-	Height at floor level
h_t	-	Height at top level
n_v	-	Number of vertical data points

Further, the user should also specify the distance between the placement points along the defined placement lines. For each line, a linearly spaced vector is computed from the start point to the end point. The distance between each point in the x-direction is determined by Eq. 4.2.

$$d_{pl} = \frac{|x_e - x_s|}{a_{pl} \cdot l_l} \tag{4.2}$$

where:

d_{pl}	-	Distance between each discrete placement point
x_e	-	X-coordinate at the start of the line
x_s	-	X-coordinate at the end of the line
a_{pl}	-	Placement line accuracy (discrete points per length unit)
l_l	-	Length of line

In Fig. 4.9, the UI window for specifying the floor height, top height, number of vertical data points and placement line accuracy.

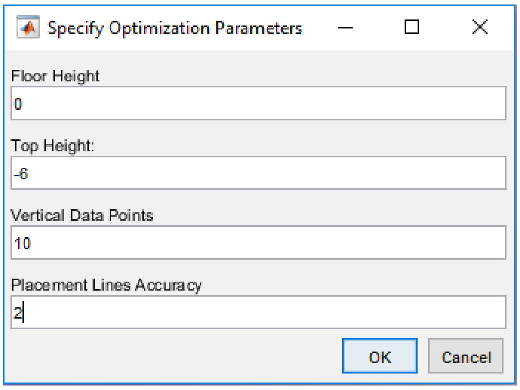


Figure 4.9: Problem Accuracy

To determine the array of annotations, every data point is evaluated with respect to whether or not it lies within an obstacle or a region of interest. It is then annotated corresponding to either the obstacle annotation or the weight of the ROI. This is done using a user-made function, *inpolyhedron* [69] which determines if a data point is inside a 3D object made of faces and vertices and *unifyMeshNormals* [70] which ensures that all face normals point in the same direction, which is a requirement to determine if a data point is inside an object.

The JSON file is then generated when the user selects *Generate JSON*. The JSON file consists of arrays of:

- X-coordinates of grid points
- Y-coordinates of grid points
- Z-coordinates of grid points
- X-coordinates of placement points
- Y-coordinates of placement points
- Z-coordinates of placement points
- Array of annotations
- Camera parameters for all specified cameras

Alternatively, the UI also stores this data in a **.mat* file if the optimization program is written in Matlab code or the data should be viewed in Matlab for another purpose.

4.2 Sensor Model Verification

To test and verify the sensor model developed in Sec. 3.2, a test case was made. This test is also used to verify that the output of the developed UI was correct. It should be noted that the length unit is not included for either the room size or the sensor range since it is not of importance as long as the length unit for the sensing range equals the length unit for the environment. The length unit could be understood, e.g. as meters, for this test.

In this case, two sensors are to be placed with fixed poses. A Brute Force Algorithm was developed specifically for this task. The environment in which maximum coverage is desired can be seen in Fig. 4.10. The camera parameters can be seen in Tab. 4.1

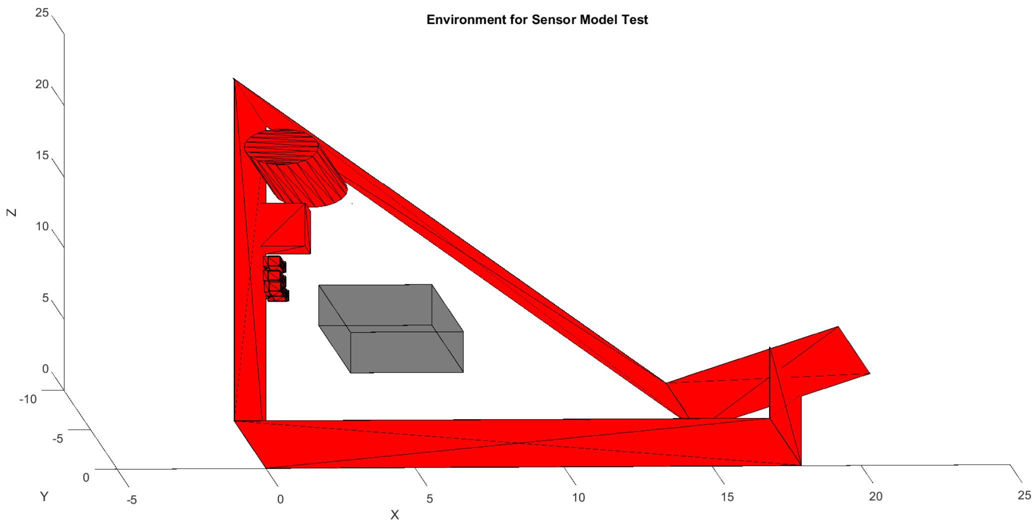


Figure 4.10: Environment Model for Sensor Model Verification

An obstacle is added, as can be seen in the middle of the room. This is added as a Region of Interest in the UI but defined as an obstacle in the program to test the sensor model. The C++ code for finding the camera positions can be seen in App. A.8.1. Since two cameras are to be placed, the optimization program utilizes two nested *for*-loops outside the main coverage function to ensure that all possible combinations are evaluated.

Table 4.1: Camera Parameters : Sensor Model Test

Camera Parameters	**Camera 1**	**Camera 2**
Field of View	60°	60°
Range	8	8
Pan	$-135°$	0
Tilt	0	0

The output of the Matlab UI can be seen in Fig. 4.11. The output is only viewed in the XZ-plane at $Y = -6$, hence the obstacles which can be seen in Fig. 4.10 are not displayed since they are beneath the plane in view.

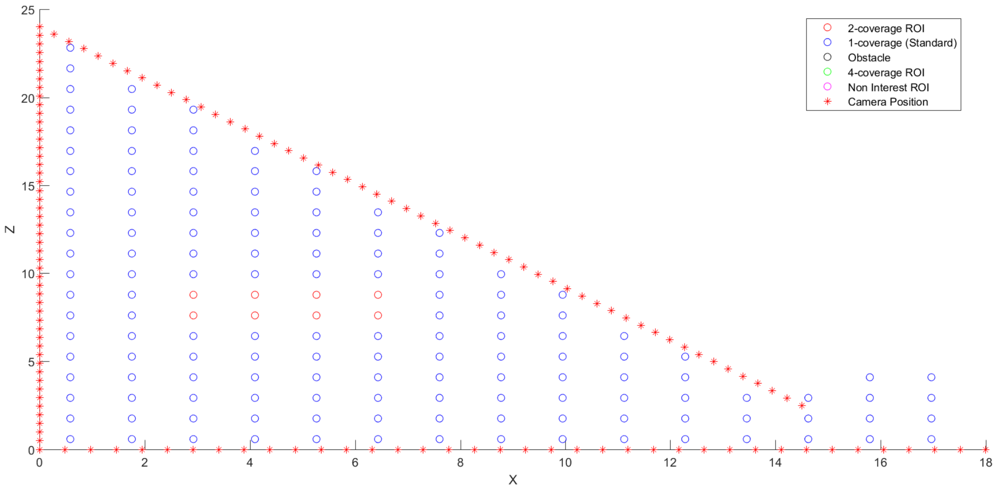


Figure 4.11: User Interface Output : Sensor Test Model

The result of the optimization algorithm can be seen in Fig. 4.12, with the camera positions being:

$$\begin{aligned}\text{Camera 1 (x,y,z)} &: (5.57692, -6, 15.7308)\\ \text{Camera 2 (x,y,z)} &: (0, -6, 4.40816)\end{aligned}$$

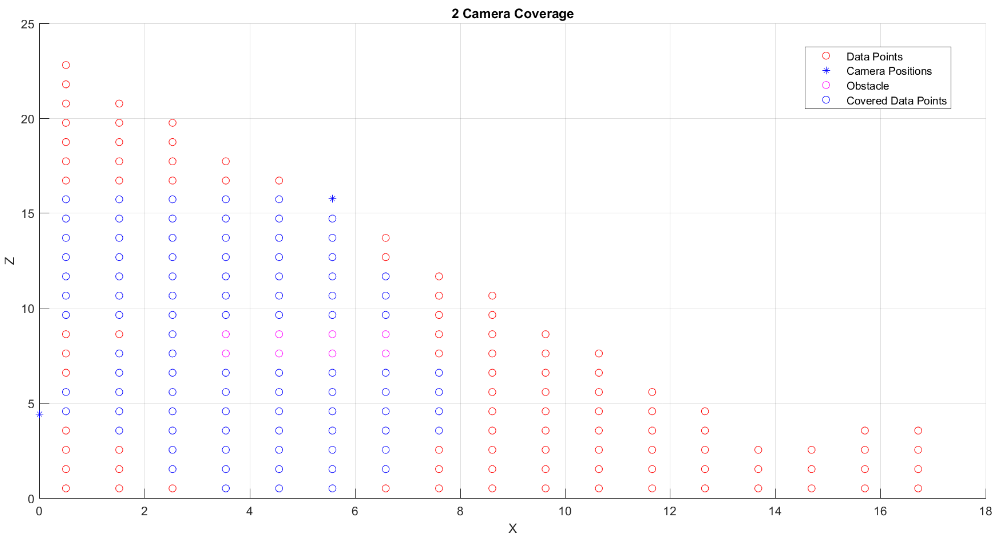


Figure 4.12: Optimization Result Visualization for Sensor Model Verification

With this setup, the cameras cover just over 50% of the data points. In this test, there were 1930 data points, 141 possible camera locations, and 70 obstacle points. In total, the script evaluated 269 million data points in 38 seconds on an Intel i7-6700K 4.00 GHz CPU. This case study verified the sensor model, which corresponded with the desired camera model. It also validated the UI output, ensuring that the JSON file was correctly exported from Matlab, imported into C++ and used successfully in the optimization program.

4.3 Camera Placement Algorithms

This case study aims to develop and compare two optimization algorithms. The first algorithm is a Greedy Algorithm designed to be fast, but not necessarily converge to the global optimum. The second algorithm computes all possible combinations of placing k cameras with n possible camera locations and evaluates every combination to find the combination which yields the highest coverage. The problem used to compare the algorithms will be limited to a relatively small problem in this section since the aim to benchmark the two algorithms.

4.3.1 Greedy Algorithm

As mentioned in Sec. 3.7.1, the Greedy Algorithm is a popular heuristic method for solving optimization problems. It converges to a solution extremely fast, since it only considers the best solution at a given iteration. The drawback of this algorithm is that the solution may not be the global optimum.

In this thesis, A Greedy Algorithm is developed for several reasons. Firstly, it can act as a reference for other algorithms concerning optimality. If the solution of the Greedy Algorithm is better than the solution of another algorithm, it can be concluded that this algorithm can not guarantee a global minimum for the given problem. Another function of the Greedy Algorithm is that it can provide an initial guess for other optimization algorithms, such as a suggested number of cameras needed to cover a given volume.

The suggested Greedy Algorithm is presented in Pseudo code in Alg. 8. In each iteration, the total coverage of the previously placed cameras is calculated. The objective of each iteration is then to find the best position and pose of the next camera, and re-calculate the coverage. The problem statement can be formulated in several ways with only minor changes to the algorithm, such as

$$max \sum_{i}^{n_s} \mathcal{C}_i \qquad s.t. \qquad n_s = n_{def} \tag{4.3}$$

$$min(n_s) \qquad s.t. \qquad \sum_{i}^{n_s} \mathcal{C}_i \geq \mathcal{C}_{def} \tag{4.4}$$

where:

n_{def}	-	Defined number of sensors to be placed
$\mathcal{C}_{def}$	-	Defined number of data points to be covered
$\mathcal{C}_i$	-	Coverage array for camera i
n_s	-	Number of cameras

It could also be extended to handle variable tilt angle. The variable tilt angle can be implemented in the same manner as variable pan angle by adding another *for*-loop. The C++ code of the Greedy Algorithm can be found in App. A.8.2.

Algorithm 8: Greedy Placement Algorithm

```
Input:
numcams - Number of cameras to be placed
campx, campy, campz - Arrays of placement points
datax, datay, dataz - Arrays of data points
pans - Array of pan configurations
obstacles - Array of obstacle points
fov,range,tilt - Camera parameters
Result:
x_out, y_out, z_out : Arrays of best camera position
pan_out - Array of pan angles
Data:
lcamp = length( campx ); ldata = length( datax );
out_coverage (1:ldata ) = 0; f_coverage (1:ldata ) = 0;
for i = 1 : numcams do
    if i > 0 then
        for covcount = 1:ldata do
            if f_coverage(covcount) == 1 then
                out_coverage(covcount) = 1 ;
            end
        end
    end
    max = 0 ;
    for j = 1:lcamp do
        x = campx(j) ; y = campy(j) ; z = campz(j) ;
        for k = 1:length( pans ) do
            pan = pans(k) ; sum = 0 ;
            for m = 1:ldata do
                if out_coverage(m) != 1 then
                    Compute coverage b : b = 1 if covered and visible
                    if b == 1 then
                        coverage(m) = 1 ;
                    else
                        coverage(m) = 0 ;
                    end
                    sum = sum +b ;
                end
            end
            if sum > max then
                max = sum ; num = j ;
                for covc = 1:ldata do
                    f_coverage(covc) = coverage(covc) ;
                end
                pan_out(i) = pan ; x_out(i) = x ; y_out(i) = y; z_out(i) = z;
            end
        end
    end
end
```

4.3.2 Brute Force Algorithm

The Brute Force Algorithm (BFA) aims to compute all combinations of possible camera placements. Given n possible placement points and k different cameras to be placed, the total number of combinations can be determined by the binomial coefficient, $\binom{n}{k}$. The formula for the binomial coefficient can be seen in Eq. 4.5.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{4.5}$$

When k and n increase to be large numbers, the binomial coefficient evaluates to be extremely large, making this algorithm computationally heavy. The rapid increase of the binomial coefficient is visualized in Fig. 4.13. The advantage of the BFA is that it is deemed to find the global optimum.

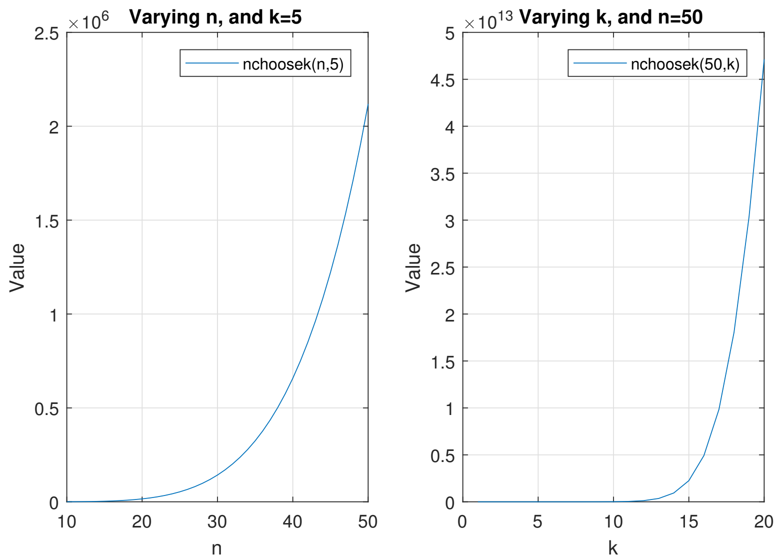


Figure 4.13: Visualization of the Rapid Increase of the Binomial Coefficient

The Brute Force Algorithm is formulated using nested *for*-loops to iterate through all possible solutions, and then calculating the best positions and poses for a given number of cameras.

The algorithm can be presented in five steps:

1. Load files from JSON and define the number of cameras and possible pan angles
2. Compute coverage from all poses at all positions
3. Make a matrix of all possible combinations for placing k cameras in the given positions and poses
4. Evaluate total coverage by looping through all possible combinations
5. Find best combination and print results

The Pseudo code of the algorithm can be seen in Alg. 9, where steps 2-4 are covered. The function for generation of the combinations matrix is inspired by the Rosetta Combinations code [71].

The BFA is expandable to k-coverage and tilt angle variation without altering too much of the code. K-coverage is an important aspect, and will be covered in Sec. 4.6. The C++ code of the BFA can be found in App. A.8.3.

Algorithm 9: Brute Force Placement Algorithm

```
Input:
numcams - Number of cameras to be placed
campx, campy, campz - Arrays of placement points
datax, datay, dataz - Arrays of data points
pans - Array of pan configurations
obstacles - Array of obstacle points
fov,range,tilt - Camera parameters
Result:
pannum - Pan angle indices for best combination
camnum - Camera indices for best combination
Data:
ldata = length(datax);
lcamp = length(campx);
lpans = length(pans);
for i = 1:lcamp do
    Set current camera position x,y,z ;
    for k = 1:lpans do
        Set current pan angle pan ;
        iter = iter + 1;
        for j = 1:ldata do
            Determine coverage and visibility ;
            and store in b(iter,j);
        end
    end
end
combarr = nchoosek(1:1:(lcamp*lpans),ncams);
for m = 1:length(combarr) do
    bcombs = false(1,ldata);
    for n=1:numcams do
        ind = combarr(m,n);
        bcombs = bcombs OR b(ind,1:end);
    end
    sumtot(m) = sum(bcombs);
end
valmax = max(sumtot);
paramind = find(sumtot == valmax);
outdata = combarr(paramind,1:end);
for l = 1:numcams do
    ind = outdata(j) - 1;
    cou = floor(ic/lpans);
    pannum(j) = outdata(j) - cou*lpans;
    camnum(j) = cou + 1;
end
```

4.3.3 Comparison

The algorithms are compared for $1 \dots 4$ cameras for a test case with the following parameters:

- Number of cameras : $1 \dots 4$
- Number of possible camera positions : 70
- Possible pan angles : $(-3 \cdot \pi/4, 0, \pi/2)[rad]$
- Number of data points to be covered : 1490
- Number of obstacle points : 98

The optimization task is then to maximize coverage given N_{def} number of cameras:

$$max \sum_{i}^{n_s} \mathcal{C}_i \tag{4.6}$$

With the following constraints

$$n_s = n_{def} \tag{4.7}$$
$$x_c, y_c, z_c \in \mathcal{P} \tag{4.8}$$
$$\alpha \in \alpha_{def} \tag{4.9}$$
$$r = 8 \tag{4.10}$$
$$\beta = 0 \ rad \tag{4.11}$$
$$f = \pi/3 \ rad \tag{4.12}$$

where:

n_{def}	-	Defined number of sensors to be placed
$\mathcal{C}_i$	-	Coverage array for camera i with length equal to the number of data points to be covered
N	-	Number of cameras
x_c, y_c, z_c	-	Camera positions
$\mathcal{P}$	-	Possible camera positions
α	-	Pan angles
α_{def}	-	Possible pan angles
r	-	Camera range
f	-	Camera field of view
β	-	Camera tilt angle

The algorithms were compiled in C++ using the following command.

g++ code.cpp -O3 -o outexec

The *-O3* flag specifies that the executable *outexec* should be compiled for optimal speed when running the executable. The executable is run using the following command.

cat jsonfile.json | ./outexec --stdin

The tests are performed on an Intel i7-6700K 4.00 GHz CPU.

The results of the BFA can be seen in Tab. 4.2. For all number of cameras, the program was executed five times to get the mean computational time and verify that the results were consistent.

Table 4.2: Brute Force Algorithm Test Case Results

No. Cameras	**1**	**2**	**3**	**4**
No. combinations	210	21945	1521520	78738660
X positions	$[0]$	$[0, 7]$	$[0, 0, 12]$	$[0, 0, 12, 7.54]$
Z positions	$[7]$	$[12, 0]$	$[5, 14, 0]$	$[3, 16, 0, 12.82]$
Pan angles	$[0]$	$[0, \pi/2]$	$[0, 0, \pi/2]$	$[0, 0, \pi/2, -3 \cdot \pi/4]$
Sum coverered datapoints	400	745	947	1088
Computational time [s]	0.1466	0.3263	18.395	1170.77

The Greedy Algorithm was also tested for the same problem for $1 \dots 4$ cameras. The program was executed five times for this algorithm also. The results of the Greedy Algorithm can be seen in Tab. 4.3.

Table 4.3: Greedy Algorithm Test Case Results

No. Cameras	**1**	**2**	**3**	**4**
X positions	$[0]$	$[0, 7]$	$[0, 0, 12]$	$[0, 0, 12, 0]$
Z positions	$[7]$	$[12, 0]$	$[7, 14, 0]$	$[7, 14, 0, 1]$
Pan angles	$[0]$	$[0, \pi/2]$	$[0, 0, \pi/2]$	$[0, 0, \pi/2, 0]$
Sum coverered datapoints	400	745	921	1044
Computational time [s]	0.1514	0.2534	0.3200	0.3654

As can be seen from the tables, the Greedy Algorithm computes the same positions as the Brute Force Algorithm for one and two camera(s). For more than two cameras, the Greedy Algorithm calculates sub-optimal solutions. When comparing the total computational time the Greedy Algorithm is seen to have the superior computational time. Where the computational time of the Brute Force Algorithm increases rapidly when more cameras are introduced, the Greedy Algorithm only uses slightly more time due to the decreasing search space for each iteration.

4.4 Developing a Genetic Algorithm

Based on the presented theory on algorithms in Sec. 3.7, a Genetic Algorithm (GA) is designed for the sensor placement problem. The intention of developing a GA is to make an approximation algorithm that has a higher probability of finding the optimal solution than the Greedy Algorithm while converging to the solution faster than the Brute Force Algorithm.

The Genetic Algorithm is not as prone to getting stuck in local optima as the Greedy Algorithm due to the randomness introduced in several parts of the algorithm. This is an important aspect in making good approximation algorithms. The GA developed in this thesis has rather conservative schemes for selection of parents and children, where *survival of the fittest* is used as a basis for the schemes. Also, the presented crossover method favors the parent with the highest fitness although the method introduces a random variable for children also to inherit genes from the weakest parent. In this section, the design of the algorithm will be presented and compared to the Greedy Algorithm and the Brute Force Algorithm using the same tests as presented in Sec. 4.3.

4.4.1 Algorithm Design

The algorithm is designed to solve the decision version of the sensor placement problem, formulated as:

Given the number of sensors, their possible positions and poses, how many data points can be covered?

Each solution can be scored using a fitness function:

$$F = \sum_{i=1}^{n_s} \mathcal{C}_i \tag{4.13}$$

The coverage for sensor i, $\mathcal{C}_i$, is calculated using the same methods as described for the Brute Force Algorithm.

Initialization

The population is initialized with user-determined population size. The first population consists of randomly chosen feasible solutions based on the possible sensor locations and poses. Each individual consists of n_s features, where each feature is an integer which corresponds to the sensor index. This index can be converted to sensor position and pan angle. The first population is then evaluated using the fitness function.

Parent Selection Scheme

The parent selection scheme is based on the roulette selection principle [72]. For each individual in the population, a probability is assigned based on its fitness. $I = (I_1, I_2, \ldots, I_n)$ is a population with the corresponding fitness scores $F = (F_1, F_2, \ldots, F_n)$. A probability is then given to each individual in the population according to the following equation

$$p_k = \frac{F_k}{\sum\limits_{i=1}^{n} F_i} \tag{4.14}$$

The parents are then chosen by calculating a random number $r \sim U(0,1)$ and selecting the appropriate parent given the random number. The probabilities are sorted in a list where the sum of all elements is equal to one. Then, starting from the top of the list, the fitness of the individual is subtracted from the random number until the sum of this is equal to, or below, zero. In this way, the probability is highest for choosing the fittest individuals as parents. There are several other methods of selecting parent individuals such as the tournament selection principle [73]. The roulette selection method presented in this thesis is conservative and has a significant selection pressure, meaning that it is biased towards selecting the fitter solutions. If the selection pressure is too high, the algorithm may be stuck in local optima, and if the selection pressure is too low, the algorithm becomes too random, which makes the design of the selection scheme an essential part of the GA design.

Crossover

Similarly to the roulette selection principle, the presented crossover method has a large selection pressure to favor the fittest parent when combining the features of two parents. The most popular crossover method is the single point crossover which is shown in Fig. 4.14, where two children are produced from the two parents. By defining a cut, both children inherit features from both parents.

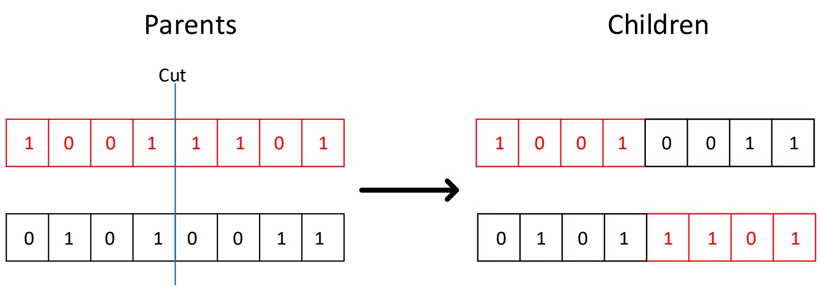


Figure 4.14: Single Point Crossover Method

In this thesis, another method is used. Unlike the single point method, only one child is produced from two parents; hence the children pool is half the size of the parent pool. The method is inspired by the fusion operator suggested by [61]. Suppose two parents are chosen for crossover, P_1 and P_2, with the fitness scores f_1 and f_2. Then, a probability can be assigned to both parents in the same way as for the parent selection. Also, say that the parents have a length of k features. For each feature of the child $C(n), n \in k$ the probability of choosing feature n from P_1 is $\dfrac{f_1}{f_1 + f_2}$ unless $P_1(n) = P_2(n)$, in which case the child will inherit $P_1(n)$ regardless.

Mutation

Where both the selection and crossover methods have a large selection pressure, the mutation provides random changes of the child to allow for diversity. By generating a random number $r_m \sim U(0,1)$, one random feature of the child will be mutated if $r_m > m_f$, where m_f is the mutation factor.

Children Selection and New Population

The fittest half of the children are selected, without any randomized selection method, to replace the lowest ranked individual in the current population to form a new population. The full algorithm can be seen in Pseudocode in Alg. 10, and as a Matlab script in App. A.7.1.

Algorithm 10: Genetic Algorithm for the Sensor Placement Problem

Input:
s_p - Size of population
n_{par} - Number of parents per generation
n_g - Number of generations
m_f - Mutation factor
n_s - Number of cameras to be placed
Result:
Positions and angles for best sensor positions
Data:
Initialize s_p randomly chosen individuals in a population
Evaluate fitness of all individuals in the initial population
while $n < n_g$ **do**
  Select n_{par} parents for reproduction based on the roulette selection scheme
  while $i < s_{ch}$ **do**
    Choose two parents from parent pool based on the parent selection scheme
    Apply probabilistic fitness based crossover to create one child
    Mutate the child with a probability of m_f ($m_f \in (0, 1)$)
    $i = i + 1$;
  end
  Evaluate the fitness of all children
  Select the best half of the children pool to replace the worst individuals the existing population
  Evaluate fitness of the new population
  $n = n + 1$;
end
Convert result to camera coordinates and camera angles

4.4.2 Tuning the Variables in the Genetic Algorithm

A test was made to investigate the effect of the population size, the number of generations and mutation factor in the Genetic Algorithm. Four tests were made, where, for each of the four tests, three sub-tests (1-3) were conducted with varying population size and number of generations. The main tests (A-D) were done with the setup seen in Tab. 4.4.

Table 4.4: Test Setup for Mutation Factor Tests

Test no.	n_{dp}	n_s	n_p	n_{cp}	n_{co}
A	1490	2	3	70	21945
B	1490	3	3	70	1 521 520
C	1490	4	3	70	78 738 660
D	1490	5	3	70	3.2440e+09

where:

n_{dp}	-	Number of data points in the universe to be covered
n_s	-	Number of cameras to be placed
n_p	-	Number of possible pan angles
n_{cp}	-	Number of possible camera placement points
n_{co}	-	Number of possible placement combinations given by the binomial coefficient

Since the test setup is the same as for the Greedy Algorithm and the Brute Force Algorithm, the global optimum has been found which was desirable to ensure proper verification methods for the Genetic Algorithm. The tests were executed in Matlab on an i7-490K CPU, which should suggest slower execution time than it would have been in C++ on an i7-6800K CPU, which was used for the Brute Force Algorithm and Greedy Algorithm tests previously presented.

The population size, number of generations and average computational time is shown in Tab. 4.5. It can be seen that, for smaller problems, the Genetic Algorithm is slower than the Brute Force Algorithm. For larger problems, however, the Genetic Algorithm is considerably faster than the Brute Force Algorithm. For each test (A1-D3), 15 tests are done for each of the seven different mutation factors: $(1/2, 1/4, 1/3, 1/5, 2/3, 1/6, 3/4)$. This means that the algorithm is executed 1260 times in total. Mutation factors higher than 3/4 did not yield good results because then the algorithm was more or less a random search algorithm. With mutation factors lower than 1/6, the algorithm became too similar to a hill-climbing algorithm and got stuck in local minima. The fitness score of the best individual for all tests can be seen in App. A.3. In the table below, s_p indicates the population size, n_g is the number of generations and t_c is the average computational time in seconds.

Table 4.5: Variables and Timing for Mutation Factor Tests

Test No:	**A1**	**A2**	**A3**	**B1**	**B2**	**B3**	**C1**	**C2**	**C3**	**D1**	**D2**	**D3**
s_p	500	300	900	1000	1500	1200	2500	1500	2000	2000	2800	3200
n_g	50	70	150	150	150	200	300	200	250	250	300	320
t_c	1.27	2.33	7.87	11.86	23.62	14.59	33.86	12.6	21.51	25.69	44.80	54.3

Fig. 4.15 to 4.18 show a visual representation of the mean value of all tests along with the global optimal solution. The figures clearly show that the higher mutation rates yield the best results. When looking at the figures, it should be noted that the optimal value (black line) is only reached if all 15

tests yielded the optimal result. For test A-C, the optimal results were reached in all 15 tests for mutation rate equal to 3/4. For test D, however, only 14 out of 15 tests with mutation factor 3/4 reached the global optimum which results in a mean value below the global optimum.

Figure 4.15: Mean Fitness Scores for Test A with Varying Mutation Factors

Figure 4.16: Mean Fitness Scores for Test B with Varying Mutation Factors

Figure 4.17: Mean Fitness Scores for Test C with Varying Mutation Factors

Figure 4.18: Mean Fitness Scores for Test D with Varying Mutation Factors

It has been shown in this case study that the Genetic Algorithm can find very good, and more often than not, the optimal solution of the sensor placement problem in a reasonable time. By having a high selection pressure on both the selection and crossover phases of the algorithm, the best solution was found by keeping the mutation rate high. For all tests, the number of parents in the parent pool, and children generated is equal to

$$n_{par} = \frac{3 \cdot s_p}{10} \tag{4.15}$$

$$n_{ch} = \frac{n_{par}}{4} \tag{4.16}$$

where:

n_{par}	-	Number of parents chosen for crossover
s_p	-	Population size
n_{ch}	-	Number of children chosen to replace the least fit individuals in the population

It must be emphasized that the Genetic Algorithm can never guarantee to find the global optimum of the presented problem because it is random to a certain degree. However, contrary to, e.g., the Monte-Carlo search which is strictly random it should be heavily influenced by the fittest solution in the population, increasing the probability of finding the optimal solution.

To visualize the difference in scalability for the GA, BFA and Greedy Algorithm, a comparison can be seen in Fig. 4.19.

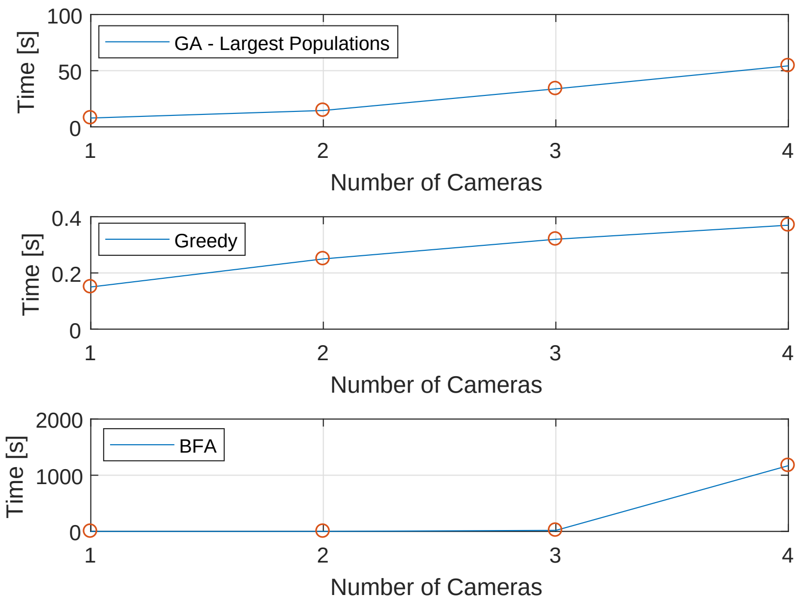


Figure 4.19: Comparison of Computational Times for the Developed Algorithms

As can be seen from the figure, the Genetic Algorithm provides better scalability compared to the BFA since the BFA proliferates for increasing input sizes, e.g. with a factor of ≈ 60 from three to four cameras.

4.5 Speeding Up a Brute Force Algorithm

In Sec. 4.3.2 the Brute Force Algorithm (BFA) is presented and tested on a CPU. It can be seen clearly that for larger problems, the computational time of this algorithm would be too large for practical use since the binomial coefficient proliferates. Each combination is independent of the others, and for each combination, the calculations are the same. These factors indicate that parts of the algorithm can be converted to a kernel function. For each combination, simple instructions are performed such as *for*-loops, *or*- and *addition*-operators.

4.5.1 Converting to CUDA

The C++ code is converted to CUDA code using the principles presented in Sec. 3.10. Since the code is compiled with *nvcc* and *clang-3.8* instead of *g++*, some libraries can no longer be used. The json11 library is not supported by CUDA which caused issues since it is essential in converting the code from JSON to readable arrays in C++. The proposed solution is to divide the problem into two different programs. The first program is responsible for importing the JSON file and doing the calculations for determining the coverage matrix and the combinations matrix. These matrices are exported to two separate **.txt* files which can be loaded into the second program. The **.txt* files are imported into the second program which is responsible for the optimization part. In the second program, the user has to define some parameters for the given problem:

- Number of data points
- Number of pan angles available
- Number of possible sensor positions
- Number of sensors to be placed

The **.txt* files are converted to *int* and *bool* 1D arrays. Since these arrays can become quite large, they need to be pre-allocated in the heap instead of allocating them on the stack which can cause a segmentation fault. The **.txt* file is imported using a *while* loop which utilizes the shift operator to collect the items in the file and store them in the correct locations in the arrays. These arrays are then copied to pre-allocated memory in the GPU. This limits the number of arrays to be copied onto the GPU memory to three arrays; the combination array, the subsets array and an array which stores the fitness of each combination. The arrays are then used in the kernel function which is being executed on the grid of threads and blocks. Finally, the fitness array is copied back to device memory for post-processing, before the allocated memory is freed.

4.5.2 Sizing the Problem for a NVIDIA GTX 1080 GPU

In Tab. 4.6, the hardware restrictions for the NVIDIA GTX 1080 can be seen. With 2560 CUDA cores distributed in 20 multiprocessors it can handle large grid sizes. The maximum restriction of blocks in a 1D grid is 2 147 483 647, and the maximum number of threads can be computed by multiplying the warp size with the number of threads per warp which evaluates to $32 \cdot 32 = 1024$ threads per blocks. Finally, the global memory is just above 8 GB, which is important to keep in mind when programming CUDA.

Table 4.6: NVIDIA GTX 1080 Device Query

Parameter	**Value**
Compute Capability	6.1
Multiprocessors	20
Max Blocks in the Grid (1D)	2 147 483 647
Warp Size	32
Number of CUDA Cores	2560
Global Memory [MB]	8113

The number of blocks in the grid is determined by the problem size given by Eq. 4.17.

$$n_b = \frac{n_c + n_{th} - 1}{n_{th}} \tag{4.17}$$

where:

n_c	-	Number of combinations given by $\dfrac{n_{cp}!}{n_s! \cdot (n_p \cdot n_{cp} - n_s)!}$
n_b	-	Number of blocks in the grid
n_p	-	Number of available pan angles
n_{cp}	-	Number of possible sensor locations
n_s	-	Number of sensors to be placed
n_{th}	-	Number of threads per block

If the block size calculated by Eq. 4.17 exceeds the maximum capacity given in Tab. 4.6, then the problem must be chopped up and executed in several kernel calls.

Memory Considerations

As previously mentioned, the maximum global memory for the GTX 1080 is just above 8 GB. If the arrays that should be copied onto the GPU exceeds this limit, the problem must be chopped up similarly as for the block size. The total required memory in MB can be determined by Eq. 4.18.

$$M = \frac{(n_c \cdot M_I) + (n_{dp} \cdot n_{cp} \cdot n_p \cdot M_B) + (n_c \cdot n_s \cdot M_I)}{1024^2} \quad [MB] \tag{4.18}$$

where:

M_I, M_B	-	Bytes allocated by integer and boolean values, respectively
n_{dp}	-	Number of datapoints to be observed

4.5.3 Kernel Function

The kernel function is distributed to the different multiprocessors of the GPU which are responsible for thread execution. The kernel function for the GPU is made from the most computationally heavy calculations done in the BFA, which can be seen in lines **28-34** in Alg. 9. The kernel function can be seen as a CUDA code snippet in Fig. 4.20.

```
__global__ void mykernel(int* devarr, bool* subs, int* sum,
                        unsigned long len, unsigned long nsens, unsigned long usize)
{
    // Kernel function to run on GPU
    // Defining variables (stored in each kernel)
    unsigned long th_id = blockIdx.x * blockDim.x + threadIdx.x;
    bool barr[1490] = {0}; //Array for storing coverage
    int totsum = 0; // Sum of covered points

    if(th_id < len){
        for(unsigned long i = 0; i < nsens; i++)
        {
            int ind = devarr[th_id*nsens + i];
            for(unsigned long j = 0; j < usize; j++)
            {
                if(barr[j] == 0)
                {
                    if(subs[ind*usize + j] == 1){
                        barr[j] = 1;
                        totsum +=1;
                    }
                }
            }
        }
        sum[th_id] = totsum;
    }else sum[th_id] = 0;
}
```

Figure 4.20: CUDA Code Snippet for the Brute Force Kernel Function

When comparing the mentioned instructions in the original BFA to the presented kernel function, a considerable resemblance can be seen. The main difference is that the outer *for* loop is removed in the kernel function. Instead of sequentially looping through all combinations in the CPU, the combinations are distributed onto the grid in the GPU, enabling each thread to handle its unique combination given by the thread id. Each thread stores the variables *th_id*, *barr*,*totsum* and *ind* inside local memory which can only be accessed by the thread. This makes each kernel execution unique, and by combining the result from each thread into an array (*sum*) the kernel represents the outer *for* loop of the BFA.

4.5.4 Test Results

The full CUDA program and the program for converting the information from the JSON file to **.txt* files can be seen in App. A.9.1 and App. A.9.2. The code was compiled in Linux with *nvcc V8.0.61* and *clang V3.8*. The command for compiling the code can be seen below:

nvcc -ccbin clang-3.8 -lstdc++ -arch=sm_61 main.cpp kernel.cu -o out_cuda

It is essential to include the architecture flag to specify that the code is compiled for the GTX 1080 GPU which has compute capability 6.1. This allows for more than 65535 blocks per grid, which is the maximum limit for previous architectures.

To compare the results of the CUDA program with the CPU BFA, four tests are done. These tests are the same as was done in Sec. 4.3. Tab. 4.7 shows the results of the test concerning memory usage and utilization of the GPU. As can be seen from the table, both the required power usage and the temperature increases when the number of blocks increases. For smaller tests, the power usage and temperature stay below $50W$ and $50°C$ which indicates that the GPU is not fully utilized. For the larger tests, however, both the temperature and power usage is higher, showing more significant workload on the GPU.

Table 4.7: GPU Test Results

Number of Sensors	**1**	**2**	**3**	**4**
Required Number of Blocks	1	22	1486	76894
Total Memory Used [Mb]	0.32	0.58	24.66	1575.1
Average Peak Temperature [C]	47	48	52	55
Average Peak Power Usage [W]	44	45	115	139

In Tab. 4.8, the comparison in runtime between the kernel call execution time and the CPU equivalent is shown. Each test was done ten times to ensure that the results were correct and consistent.

Table 4.8: GPU vs CPU Computational Results

Number of Sensors	**1**	**2**	**3**	**4**
Avg. GPU Calculation Time [s]	0.00309	0.00597	0.145	7.601
Avg. CPU Calculation Time [s]	0.146	0.3260	18.3950	1170.8
Achieved Speedup	x 47.25	x 54.60	x 126.86	x 154.03

As can be seen in Tab. 4.8, the speedup is considerable, especially when the problem size increases. Considering the power usage and temperature, the speedup becomes larger for more complicated problems since more of the GPU is utilized.

4.6 Adding K-Coverage Functionality

For many practical problems, it is desirable, or even necessary, to be able to define regions in the environment which need to be covered by multiple sensors. The standard coverage formulation is *flat coverage*, which means that the optimization goal is to have as many data points as possible visible and covered by one sensor. K-coverage implies that at least k sensors should cover every/some data points. It is common to set k-coverage in certain Regions of Interest (ROI).

In the literature, there are several methods of implementing k-coverage to the sensor placement problem. For omnidirectional sensors, the k-UC (k-Unit-disk Coverage) problem and the k-NC (k-Non-unit-disk Coverage) problem [74] are often used. The k-UC problem assumes that all sensors have the same sensing length, whereas the k-NC problem allows for different sensing ranges.

The authors of [75] presents two methods of solving the *k-coverage* problem. The first is a naive, but intuitive, approach to merely find good, or even optimal, placement to ensure 1-coverage of the region of interest and then just duplicating k sensors in these positions. This may result in an excessive number of sensors, and many data points that have higher than k-coverage.

The second method improves the duplication approach by determining how much of the environment that do not satisfy the k-coverage constraint after placing sensors to ensure 1-coverage.

The similarity between the methods proposed above is that they formulate the k-coverage as a hard constraint. A hard constraint, in optimization problems, must be satisfied for the solution to be feasible. Constraints can also be defined as soft constraints, meaning that the solution can be feasible without the constraint being fulfilled, although the solution will most likely be better if the constraint is fully satisfied.

In this thesis, the k-coverage constraint is defined as a soft constraint, although if it is not satisfied, the solution is heavily penalized. Thus, the optimal solution will most likely satisfy the constraint fully. The general idea is that if every data point in the ROI is not covered by k sensors, the fitness function will be penalized. The penalty will be proportional to the coverage percentage of the ROI. A proportionality constant η_k determines how heavily the solution is penalized according to Eq. 4.19.

$$P = \eta_k \cdot \frac{n_{roi}}{\sum\limits_{i=0}^{n_{roi}} c_i} \tag{4.19}$$

where:

η_k	-	Penalization constant
c_i	-	ROI coverage of point i. $c_i = 1$ if point i is covered by the required k sensors.
n_{roi}	-	Number of data points which requires k-coverage

4.6.1 Implementation

For the Brute Force Algorithm, the penalty is introduced by firstly defining an annotation array $\mathcal{R}$ which is defined as

$$\forall i \in (0, n_{dp}), \;\; \mathcal{R}_i = \begin{cases} 1 & \text{if point } i \text{ requires coverage from one sensor} \\ 2 & \text{if point } i \text{ requires coverage from two sensors} \\ 3 & \text{if point } i \text{ requires coverage from three sensors} \end{cases} \tag{4.20}$$

There are several methods of ensuring k-coverage. The annotations array is required for determining if a given solution is satisfactory k-covered. For the GPU-based BFA, the result of this is another required **.txt* file for storing the annotations. Recalling the output from the UI, the annotations array is already stored in the JSON file, meaning that implementing this requires little effort. By introducing the annotations array, the equation for calculating the required memory size is slightly altered. Previously, Eq. 4.18 described the memory requirement. The updated equation which accounts for the annotations array can be seen in Eq. 4.21.

$$M = \frac{(n_c \cdot M_I) + (n_{dp} \cdot n_{cp} \cdot n_p \cdot M_B) + (n_c \cdot n_s \cdot M_I) + (n_{dp} \cdot M_I)}{1024^2} \quad [MB] \tag{4.21}$$

Although the added memory is minimal, it is important to consider, since the memory limitations on the GPU cannot be exceeded. The annotations array is allocated and copied to the GPU memory using the same method as for the other arrays. Regarding the Genetic Algorithm, k-coverage is included in the same manner as in the kernel function for the BFA, described in Alg. 11. K-coverage will not be included in the Greedy Algorithm since it will mostly be used for initial evaluation purposes due to its sub-optimal performance compared to the other algorithms. However, introducing k-coverage in the Greedy Algorithm could be done by weighting the ROI data points higher than normal data points until satisfactory k-coverage is achieved, then placing the rest of the cameras using a Greedy method to cover as much of the remaining scene as possible.

Algorithm 11: K-Coverage Algorithm

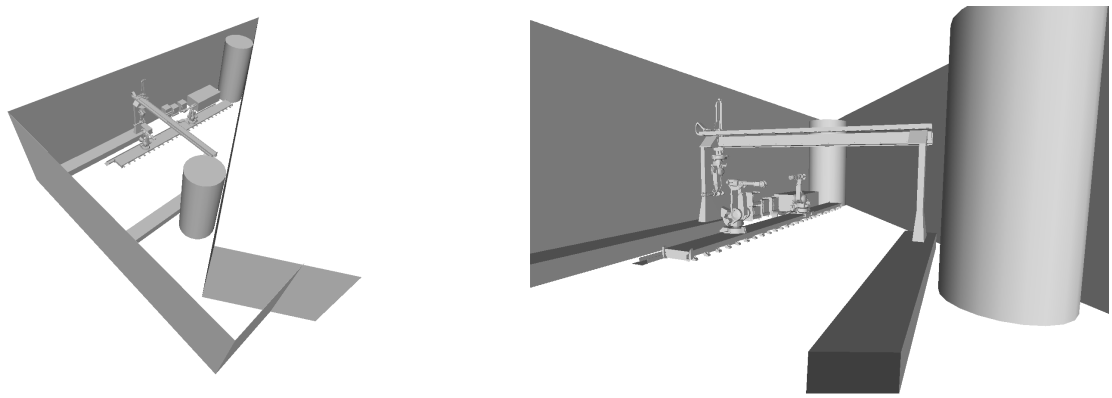
Given a combination of n_s sensors
Input:
$s_c = 0$ - Sum variable
$P = 0$ - Penalty
$covk$ - K-coverage array
$sumarr$ - Array for storing total coverage
Result:
$fitness$ - Fitness of current combination
Data:
for $i = 1 : n_s$ **do**
    for $j = 1 : n_{dp}$ **do**
        if *point j is covered by sensor i* **then**
            $covk(j) = covk(j) + 1$;
        end
    end
end
for $k = 1 : n_{dp}$ **do**
    if *point k is satisfactory covered* **then**
        $sumarr(k) = 1$;
        if *point k is a ROI point* **then**
            $sc = sc + 1$;
        end
    end
end
Set penalty according to ROI coverage if k-coverage is not achieved;
$fitness = sum(sumarr) - P$

4.7 High Accuracy Final Test

This case study aims to collect all the previous work and combine it into a real case with higher accuracy, i.e. more complexity, than previously presented. This problem concerns a robotic test laboratory located at the Mechatronics Innovation Lab (MIL) in Grimstad. The 3D model of the environment can be seen in Fig. 4.21. Multiple sensors are to be placed along the walls to maximize the coverage of the environment. The area in the bottom right corner is not considered in this task since coverage of this area is not necessary. It should, therefore, be excluded in the User Interface. The problem requirement is to cover $> 90\%$ of the environment with as few sensors as possible given the sensor parameters. If the coverage is improved drastically by adding one sensor even though the coverage requirement is fulfilled, it may be desirable to do so. The sensor parameters are given as:

- Range: $9[m]$
- Field of View (horizontally and vertically): $45°$
- Tilt : $30°$ downwards

The desired voxel size is $0.5x0.5x0.5[m^3]$ (x,y,z). The area of the room is approximately $192[m^2]$ and the height is $6[m]$. The expected number of voxels is therefore $\frac{192 \cdot 6}{0.5^3} = 9216$. Most likely, the number of voxels will be slightly lower than this prediction due to the quadratic approximations in the discretization of the environment.



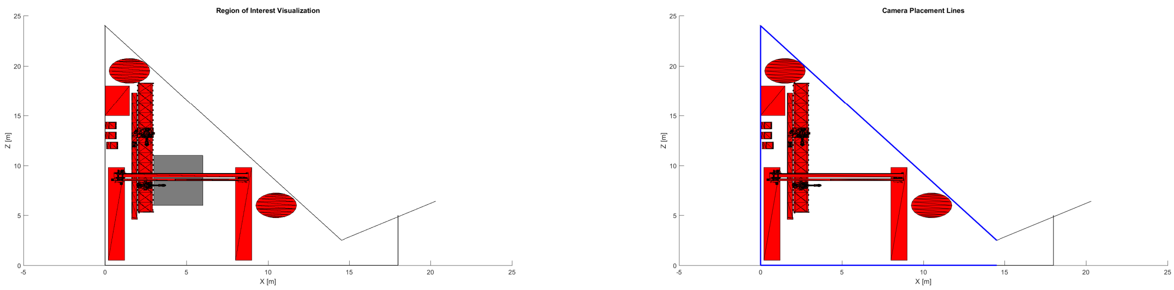
(a) Full 3D Scene

(b) 3D Scene Details

Figure 4.21: 3D Environment for the High Accuracy Test

The cameras should be placed at roof height, $6[m]$, along specified walls shown in Fig. 4.22b . The placement accuracy is set to $0.5[m]$, resulting a total of 134 possible camera locations.

A region of interest is added in an area where the gantry robot can interact with one of the other robots, as can be seen in the figure. This volume is especially important and therefore 100% of it must be covered by at least two cameras. The ROI can be seen in Fig. 4.22a. The region of interest is defined by the corners (x,z): $(3, 11), (6, 11), (3, 6)$ and $(6, 6)$, at height (y) 0 to -3.

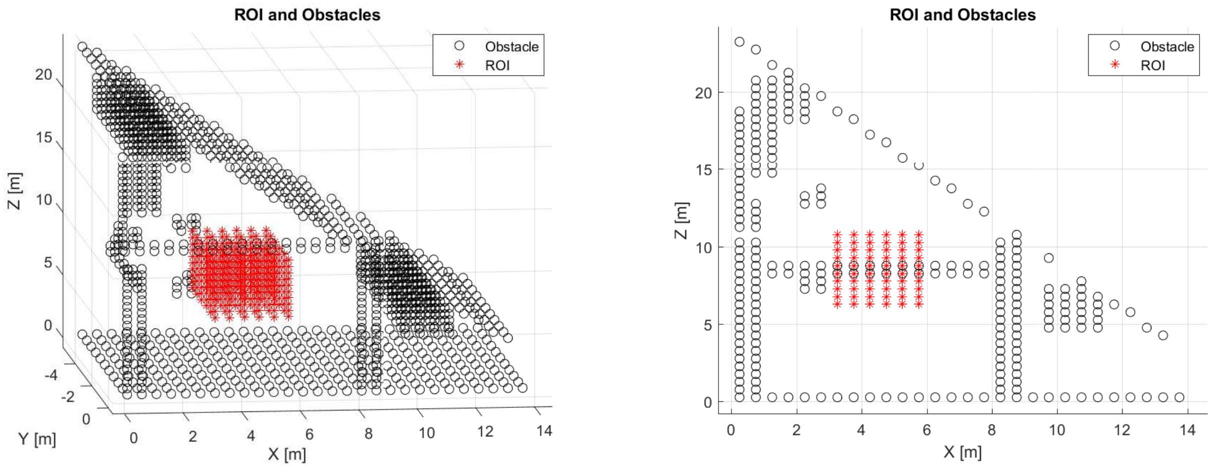


(a) Region of Interest Final Test

(b) Placement Lines in the Final Test

Figure 4.22: Defining the Region of Interest and Placement Lines for the Final test

The output of the User Interface can be seen in Fig. 4.23, which shows the point cloud of the ROI and the obstacles. As can be seen from the figure, the scene is accurately reproduced.



(a) Obstacles and ROI 3D View

(b) Obstacles and ROI XZ View

Figure 4.23: JSON Output Region of Interest and Obstacles Final Test

In total, the environment is represented by a total of 9084 data points. 1317 of these are considered obstacles, and 360 points are considered as a ROI. All tests are executed on an Intel i7-6700K running on Linux. A combination of Matlab and C++/CUDA code is used.

4.7.1 Case Analysis Using the Greedy Algorithm

The Greedy Algorithm is used to investigate how many cameras are needed and which pan angles to choose from for this problem. Therefore, the Greedy Algorithm is executed for $1 \ldots 10$ cameras, with the following available pan angles: $[0, \pi/4, \pi/2, 3 \cdot \pi/4, -\pi/4, -\pi/2, -3 \cdot \pi/4]$. The ROI is not considered in this analysis, as the aim is to merely determine the increase in coverage as the number of cameras increases.

The Greedy Algorithm is chosen due to the decreasing problem-size which results in slow convergence time, whereas the BFA computational time, on the other hand, increases rapidly for increasing problems. This makes the Greedy Algorithm very useful in the initial analysis of the problem. The computational time of the different tests with the Greedy Algorithm can be seen in Fig. 4.24.

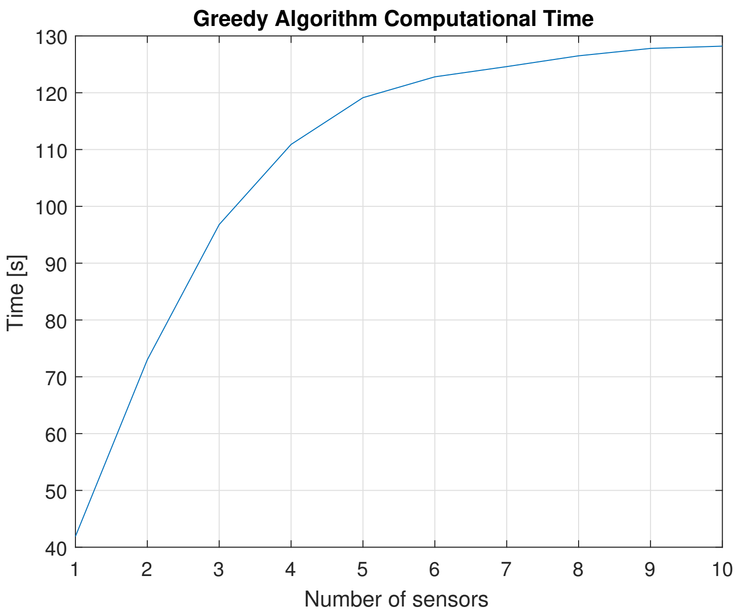


Figure 4.24: Computational Time for the Greedy Algorithm

As previously proven, the Greedy Algorithm will, presumably not produce the optimal result. Thus, the coverages listed in Tab. 4.9 will most likely be improved by both the BFA and the GA. In the table, $\boldsymbol{\mathcal{S}_a}$ and $\boldsymbol{\mathcal{S}_{r,a}}$ indicate the total accumulated coverage and the accumulated ROI coverage, respectively, for the associated sensor number and the previous sensors.

Table 4.9: Greedy Algorithm Analysis Results

Sensor No.	1	2	3	4	5	6	7	8	9	10
X Position	8.81	0	13.5	0	0	9.38	0	13.93	4.83	1.5
Z position	10.93	11	0	0	16.5	10.01	4.5	3.34	16.83	0
Pan Angle	$-3 \cdot \pi/4$	$\pi/4$	$3 \cdot \pi/4$	$\pi/4$	$\pi/4$	$-\pi/2$	$\pi/4$	$-3 \cdot \pi/4$	$-\pi/2$	$\pi/4$
$\boldsymbol{\mathcal{S}_a}$**(%)**	39.2	61.1	76.2	90.6	92.6	94.1	98.4	99.0	99.7	99.9
$\boldsymbol{\mathcal{S}_{r,a}}$**(%)**	0	0	0	7.8	7.8	9.4	86.7	86.7	96.1	96.7

From the above analysis, it can be seen that the Greedy algorithm produces acceptable results concerning coverage, but the ROI is not properly covered for any number of cameras. However, the results suggest that the number of cameras should be somewhere between 3 and 6 cameras to satisfy the requirements for this task.

4.7.2 Genetic Algorithm

In the previous analysis, the number of cameras were limited to $1\ldots10$ cameras. In this section, the Genetic Algorithm (GA) will be used to obtain more information regarding the minimum number of cameras to satisfy the requirements. Contrary to the Greedy Algorithm, the GA will maximize the coverage with respect to the ROI coverage requirement. The penalty function is defined as:

$$P = \begin{cases} \eta \cdot \dfrac{n_r}{c_r} & \text{if } c_r > 0 \quad \&\& \quad c_r < p_c \cdot n_r \\ \eta \cdot n_r & \text{if } c_r == 0 \\ 0 & \text{if } c_r \geq p_c \cdot n_r \end{cases} \tag{4.22}$$

where:

η	-	Penalization constant
c_r	-	Sum of covered ROI data points
n_r	-	Number of data points in the ROI
p_c	-	Required coverage of ROI $\left(\dfrac{\%}{100}\right)$

The following parameters are used in the GA:

Table 4.10: Genetic Algorithm Parameters Final Test

No. Cameras	**Req. ROI Coverage [%]**	η	**Population Size**	**Generations**
3	100	3500	3000	300
4	100	4000	4000	400
5	100	4000	5000	500
6	100	4000	6000	600

The available pan angles for the GA are: $(0, \pi/4, -3 \cdot \pi/4)$ [rad].For each number of cameras, three tests are done to ensure consistent solutions. These tests are performed using Matlab, the Matlab code can be seen in App. A.10.1. The timing can be seen in Fig. 4.25.

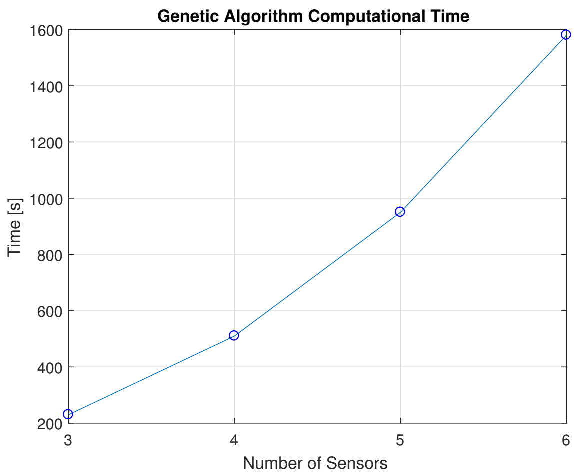


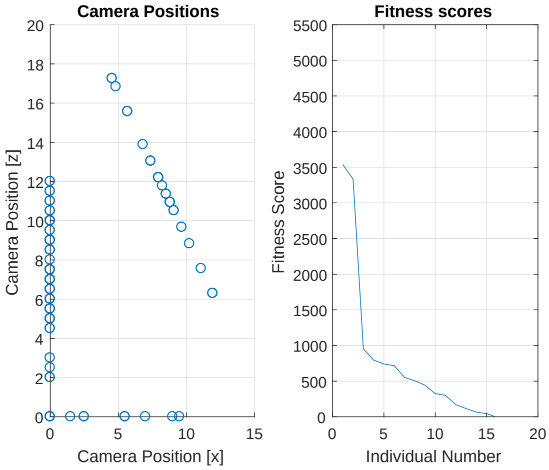
Figure 4.25: Computational Time for the Genetic Algorithm

The results are shown in Tab. 4.11.

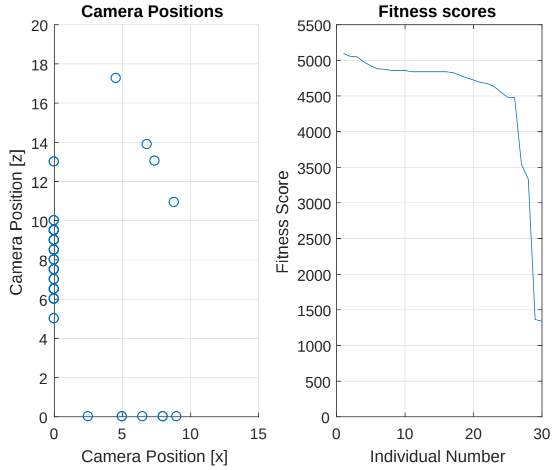
Table 4.11: Genetic Algorithm Final Test Results

Test	**X Positions**	**Z Positions**	**Pan Angles**	**Cov [%]**	**ROI Cov [%]**
3A	$[0, 0, 6.5]$	$[7.5, 9.5, 0]$	$[0, 0, \frac{\pi}{4}]$	59.1	100
3B	$[0, 0, 6.5]$	$[7.5, 9.5, 0]$	$[0, 0, \frac{\pi}{4}]$	59.1	100
3C	$[0, 0, 6.5]$	$[7.5, 9.5, 0]$	$[0, 0, \frac{\pi}{4}]$	59.1	100
4A	$[0, 0, 0, 7]$	$[5.5, 7.5, 9, 0]$	$[0, 0, \frac{\pi}{4}, \frac{\pi}{4}]$	77.3	100
4B	$[0, 0, 0, 7]$	$[5.5, 7.5, 9, 0]$	$[0, 0, \frac{\pi}{4}, \frac{\pi}{4}]$	77.3	100
4C	$[0, 0, 0, 7]$	$[5.5, 8.5, 9, 0]$	$[0, 0, \frac{\pi}{4}, \frac{\pi}{4}]$	77.2	100
5A	$[0, 0, 0, 0, 7.5]$	$[7.5, 9.5, 12, 0, 0]$	$[0, 0, \frac{\pi}{4}, \frac{\pi}{4}, \frac{\pi}{4}]$	91.5	100
5B	$[0, 0, 0, 0, 7.5]$	$[7.5, 9.5, 12, 0, 0]$	$[0, 0, \frac{\pi}{4}, \frac{\pi}{4}, \frac{\pi}{4}]$	91.5	100
5C	$[0, 0, 0, 0, 7.5]$	$[7.5, 9.5, 12, 0, 0]$	$[0, 0, \frac{\pi}{4}, \frac{\pi}{4}, \frac{\pi}{4}]$	91.5	100
6A	$[0, 0, 0, 0, 8, 8.81]$	$[0, 6, 9.5, 13.5, 0, 10.93]$	$[\frac{\pi}{4}, \frac{\pi}{4}, 0, \frac{\pi}{4}, \frac{\pi}{4}, \frac{-3\cdot\pi}{4}]$	96.1	100
6B	$[0, 0, 0, 0, 8, 8.81]$	$[0, 6, 9.5, 13.5, 0, 10.93]$	$[\frac{\pi}{4}, \frac{\pi}{4}, 0, \frac{\pi}{4}, \frac{\pi}{4}, \frac{-3\cdot\pi}{4}]$	96.1	100
6C	$[0, 0, 0, 0, 7.5, 8.81]$	$[0, 6, 9.5, 13, 0, 10.93]$	$[\frac{\pi}{4}, \frac{\pi}{4}, 0, \frac{\pi}{4}, \frac{\pi}{4}, \frac{-3\cdot\pi}{4}]$	95.8	100

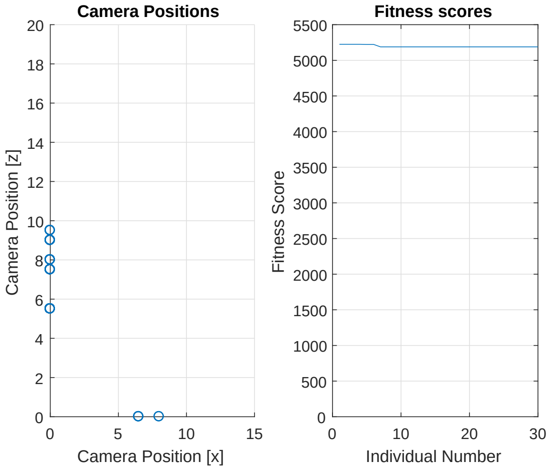
As can be seen in Tab. 4.11, the results are highly consistent. The largest deviation can be seen from test 6A to 6C (0.3%). This shows that the population size and number of generations are sufficient to produce consistent results, which indicates that result has been found which is close to the optimum. It is believed that the optimal result has been found for $3 - 5$ cameras and that the best results for 6 cameras may be optimal. To visualize the evolution of the population, the position and fitness of the 30 fittest individuals at generation 1,50,100,125 and 200 are shown for three cameras in Fig. 4.26.



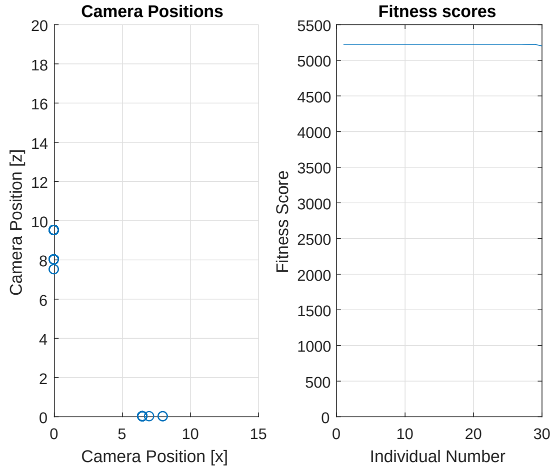
(a) Generation Number 1



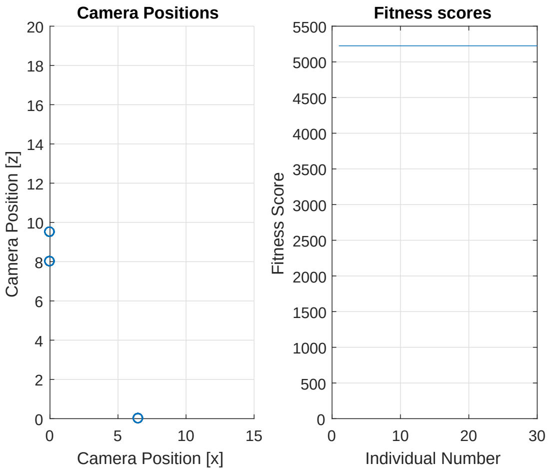
(b) Generation Number 50



(c) Generation Number 100



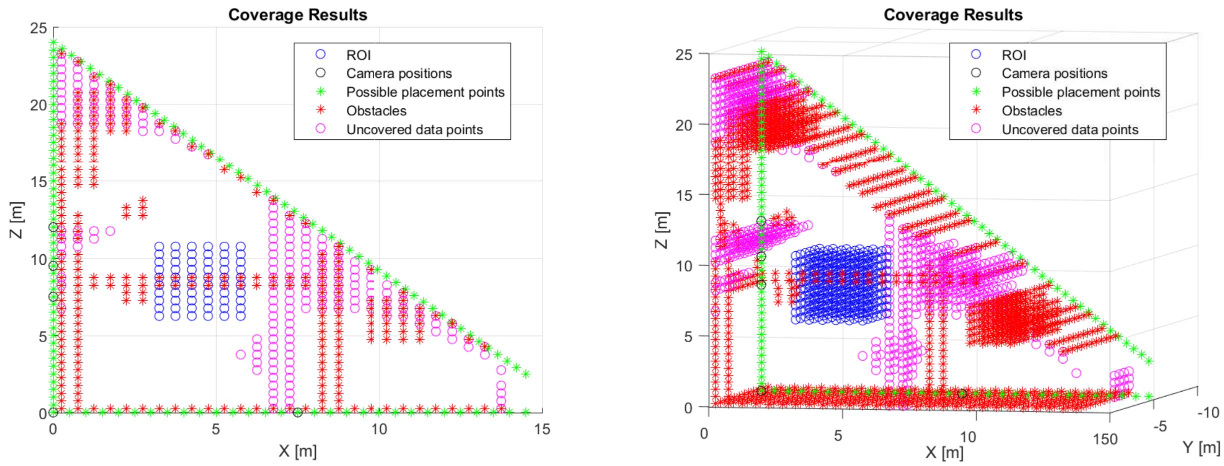
(d) Generation Number 125



(e) Generation Number 200

Figure 4.26: The Position and Fitness of the 30 Best Individuals for Various Generations

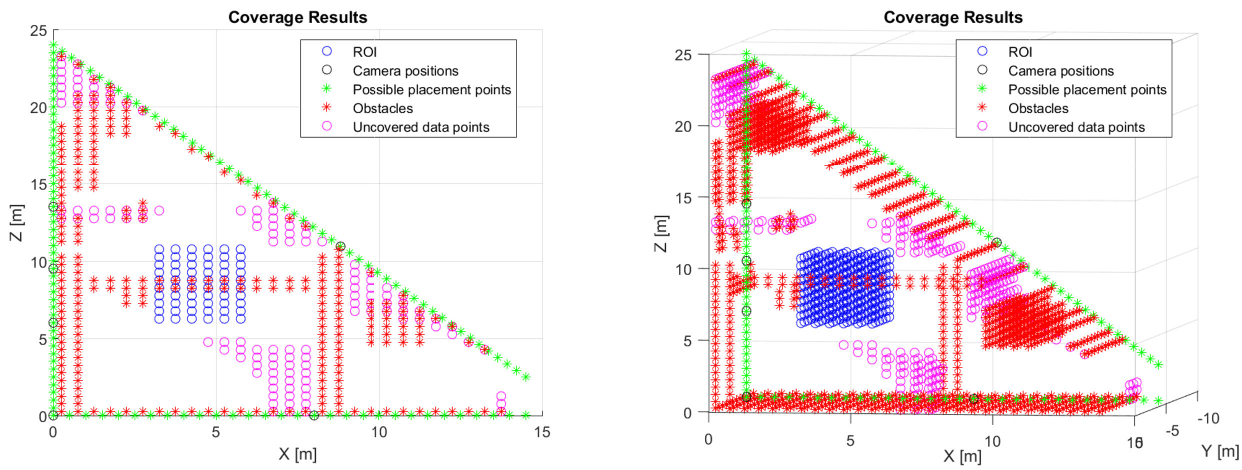
As seen in Fig. 4.26, the diversity decreases drastically as the algorithm evolves. In generation number 200, it can be seen that all the 30 best-ranked individuals have the same features, contrary to e.g. the first generation where the diversity is high, resulting in varying fitness scores. The coverage results can be seen for five cameras in Fig. 4.27, and for six cameras in Fig. 4.28. The program for evaluating and visualization of the optimization results can be seen in App. A.10.4.



(a) 2D Coverage for 5 Cameras

(b) 3D Coverage for 5 Cameras

Figure 4.27: GA Coverage Results for 5 Cameras



(a) 2D Coverage for 6 Cameras

(b) 3D Coverage for 6 Cameras

Figure 4.28: GA Coverage Results for 6 Cameras

The results show that the GA finds a feasible solution for both five and six cameras where the ROI is fully covered, and the overall coverage is 91.5% and 96.1%, respectively. Since the GA can not guarantee the global optimum, it may be desirable to verify the result by doing a Brute Force Algorithm analysis of the problem.

4.7.3 Brute Force Algorithm

Since it can not be said with absolute certainty that the GA produces the optimal result, the Brute Force Algorithm (BFA) can be used to possibly improve the solution. Based on the information from the previous algorithms, the problem can be further limited regarding possible camera locations and pan angles. The possible pan angles for this algorithm is $(0, \pi/4, -3 \cdot \pi/4)$ [rad] since these are the most commonly used in the Greedy algorithm, and the only pan angles used in the GA. Further, the possible camera locations can also be limited, since the best positions are assumed to be in a certain range for each wall, not at the ends of the wall. These assumptions are supported by the results from the GA. The new placement lines can be seen in Fig. 4.29. In total, there are 83 possible sensor locations, compared to the previous number of 134.

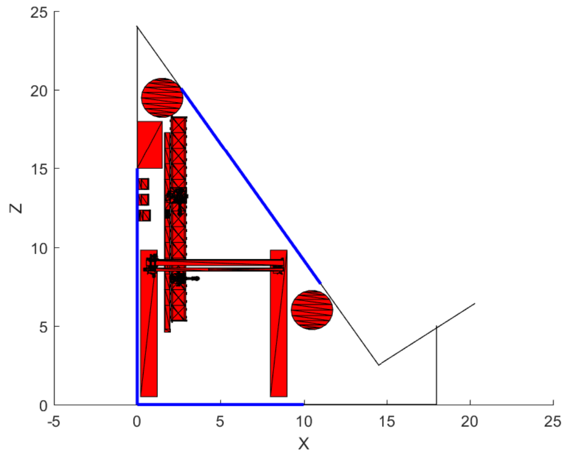


Figure 4.29: New Placement Lines for the BFA Final Test

By implementing these changes, the problem is reduced to a minimum size which is essential for the BFA due to its increasing problem size for a larger number of cameras. Tab. 4.12 shows the total number of possibilities for $4 \ldots 6$ cameras compared to the number of possibilities before the problem was minimized.

Table 4.12: Number of Combinations for the Final Test

No. Cameras	**No. Combinations (New)**	**No. Combinations (Old)**	**Reduction Factor**
4	156 340 626	$3.4008e + 09$	21.7522
5	$7.6607e + 09$	$3.6184e + 11$	47.2334
6	$3.1153e + 11$	$3.2023e + 13$	102.7908

It is clear that the problem needs to be chopped up due to the large array size of the combinations array. The number of chops is determined by Eq. 4.23.

$$n_{ch} = \frac{n_c \cdot n_s \cdot M_I}{M_{lim}} \tag{4.23}$$

where:

n_{ch}	-	Required number of chops
n_c	-	Number of combinations
n_s	-	Number of sensors
M_I	-	Integer size in bytes
M_{lim}	-	Maximum matrix size in bytes

In Sec. 4.5, a C++ program for making **.txt* files from the **.json* file was described. This program, however, does not support any functionality for chopping up the combinations matrix into several files. A solution was found using a user-created Matlab class [76], which includes the desired functionality. With this class, a function can be made which can be executed several times in series because the previous iteration is available. If the workspace is not cleared between function calls, the class enables the desired chopping by defining a maximum number of combinations to be stored at each function call.

Matlab includes the *dlmwrite* function for writing variables to **.txt* files. This function tends to be very slow for larger files. Therefore, a user-created *MEX* function is used [77] which is $30-40$ times faster than *dlmwrite*. This provides a major speedup but comes at the cost of requiring more memory than necessary since the MEX function requires a double matrix as its input. Knowing that a double is stored in Matlab as an 8-byte variable, this means that the matrix gets twice as large as it needs to be. The program for generating the chopped combinations matrix can be seen in A.10.2.

The computer with an i7-6800K processor has a RAM size of 32 GB. The GTX 1080 has a global memory size of 8 GB. The Matlab version installed on this computer is 2018a. Here, the latest supported GCC compiler is 6.3.9. Since the MEX script must be compiled in Linux, this compiler must be installed. Due to access restrictions at the end of the project, it was not possible to install this GCC compiler since this requires administrator access. Hence, the combinations matrix needed to be computed and stored on a Windows computer with a $i7-4790K$ processor and $8GB$ RAM. This limits the size of the computed matrix to $M_{lim} = 7.5e8$. Adding to this, the required computational time for one matrix generation is ≈ 45 minutes on the $i7-4790K$ compared to ≈ 25 minutes on the i7-6800K. Also, 20 minutes are required for file transfer from the computational computer to the computer with the GTX 1080 GPU which is responsible for evaluating the combinations.

For $M_{lim} = 7.5e8$, the number of required chops are 51. For each chop, a 7.65GB matrix is stored as a $*.txt$ file on the disk. Each file contains a matrix of $1.5e8$ combinations of 5 cameras. This file is then transferred to the GPU computer for evaluation of the combinations. The CUDA program can be found in App. A.10.3. During the tests, the peak GPU usage and C++ timings were recorded and can be seen in Tab. 4.13.

Table 4.13: C++ and CUDA Requirements and Results

Parameter	**Value**
Maximum Temperature	69° C
Maximum Power Usage	145 W
Threads per Block	1024
Number of Blocks in Grid	146 485
GPU Memory Usage	4.9 GB
GPU Computational Time per Test	134.8 s
Total C++ Computational Time per Test	270 s

As the table shows, the GPU is not fully utilized. Both the temperature and power usage are below the listed maximum of 90° C and 198 W, respectively. Larger arrays could be imported, and there could be more blocks in the grid. If there were more RAM available on the computer which calculates the combinations matrices, the number of chops could be reduced, reducing the total computational time. The test results for all 51 tests can be seen in App. A.3. Fig. 4.30 shows the best solution with the highest coverage for each of the 51 chops. The total computational time for this test was 36 hours which includes a substantial amount of manual work between each combination.

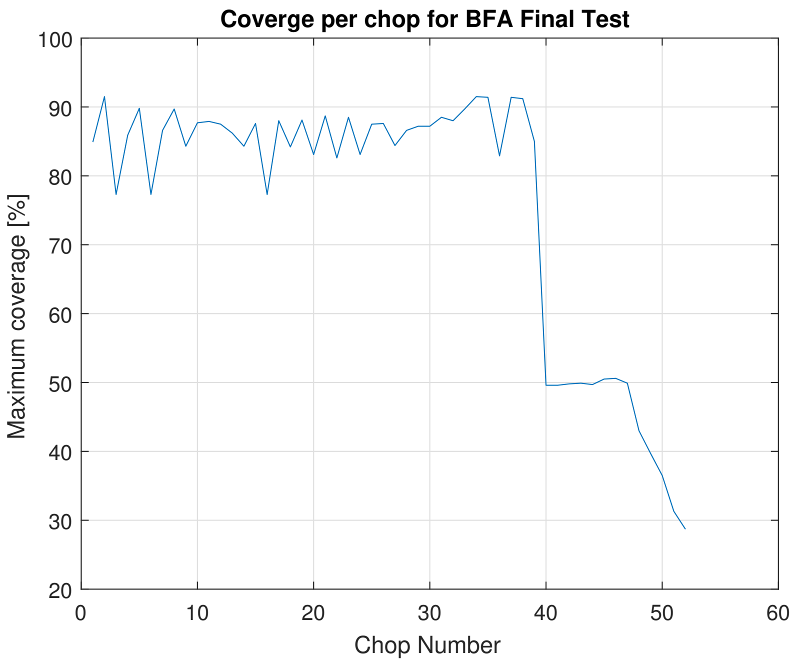


Figure 4.30: Coverage per Chop Final Test

From the figure, it can be seen that the maximum value occurs at two chops: 2 and 34. In the table in App. A.3 this can be seen to represent the following combination indices: $[45, 57, 73, 94, 139]$ and $[1, 45, 57, 73, 138]$. Due to the way the placement lines are defined, the point $(0, 0)$ appears twice. Both combinations are two representations of the same positions, which can be seen in Tab. 4.14.

Table 4.14: Brute Force Algorithm Result for 5 Cameras

X Postions	**Z Positions**	**Pan Angles**	**Coverage %**	**ROI Coverage %**
[0 0 0 0 7.5]	[0 7.5 9.5 12 0]	[π/4 0 0 pi/4 π/4]	91.5	100

This result is the same as from the Genetic Algorithm. The difference is that, with the BFA, the result is guaranteed to be the global optimum.

4.7.4 Continuous Neighborhood Optimization

The solution found by the GA and verified by the BFA meets the requirements. However, since the camera placement is a discrete optimization task, the optimal solution is limited by the discretization accuracy. Therefore, by making a continuous optimization problem in the neighborhood of the previous solution, an improved solution may be found. The continuous problem formulation is written in Matlab and solved by using built-in Matlab functions. The optimization problem is formulated as a minimization problem:

$$min \quad \frac{K}{\sum\limits_{i=1}^{n_s} \mathcal{C}_i} \tag{4.24}$$

s.t

$$P^* \in P_0 \pm \mathcal{N}_p \tag{4.25}$$
$$\Phi^* \in \Phi_0 \pm \mathcal{N}_\phi \tag{4.26}$$

where:

K	-	Large constant ($>> n_{dp}$)
n_s	-	Number of sensors
$\mathcal{C}_i$	-	Coverage array for sensor i
P^*, Φ^*	-	The decision variables, position and pan angle, respectively
P_0, Φ_0	-	The initial values found by discrete optimization
$\mathcal{N}_p, \mathcal{N}_\Phi$	-	Neighborhood size for position and pan angle, respectively

The coverage is calculated in the same manner as previously described, including visibility and k-coverage. The following Matlab optimization methods have been considered:

- **Fmincon** [78]:

  Among the most popular optimization methods in Matlab is *fmincon*. There are several available algorithms in *fmincon*, all of which will not be described here. Some methods require the gradient of the objective functions, which, naturally, is not obtainable in this optimization problem. The most common algorithm is the *interior-point algorithm*.

- **Global Search** [79]:

  The global search can be seen as an extension of the *fmincon* method. Here, there are options for repeated runs of local solvers, e.g., *fmincon*. By combining the results of the local solvers, it is more plausible to find the optimal solution rather than by only using one single solver, since more trial points are evaluated.

- **Simulated Annealing** [80]:

  Previously described in Sec. 3.7.2, the Simulated Annealing algorithm first produces a trial point at random inside the defined bounds, and determines whether or not it is better than the current point. It is accepted with a probability even though it is not better. When the temperature is lowered, the probability of choosing worse points decreases.

- **Particle Swarm** [81]:

  The particle swarm algorithm is a population-based algorithm, same as the Genetic Algorithm used previously. It is inspired by flock animals, e.g., birds, that swarms. Each particle is assigned some attraction force from both itself best-found solution, and the best solution found by the entire population. The population then gathers at either one or multiple minima after some iterations, which is likely to be either an optimal or very good solution to the problem.

The neighborhood size is determined to be $\mathcal{N}_p = 1[m], \mathcal{N}_\Phi = 45°$. The constant K is set to be 100000. The objective function of the optimization problem can be found in App. A.11.1 and the script for defining the problem and solving using the mentioned methods can be found in App. A.11.2. Tab. 4.15 shows the main results of the optimization algorithms.

Table 4.15: Results of Continuous Neighborhood Search

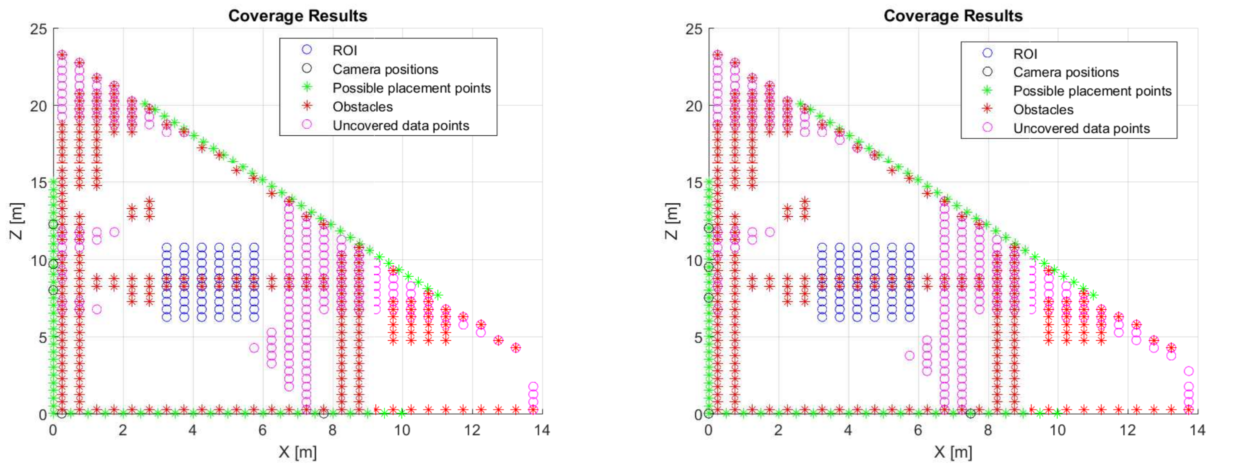
Optimization Algorithm	Objective Value	Computational Time [s]
Discrete BFA (Initial)	12.0322	-
Fmincon	12.0322	124.40
Global Search	12.0322	6645.33
Simulated Annealing	12.0322	15983.27
Particle Swarm	11.9717	19251.73

As can be seen from the table, the Particle Swarm Algorithm finds the best solution, at the cost of a much larger computational time ($\approx 5\,[hrs], 20\,[mins]$). Contrary to the other algorithms, the particle swarm algorithm does not need any initial guess. The position and pose found by the Particle Swarm Algorithm can be seen in Tab. 4.16.

Table 4.16: Result after Continuous Neighborhood Search

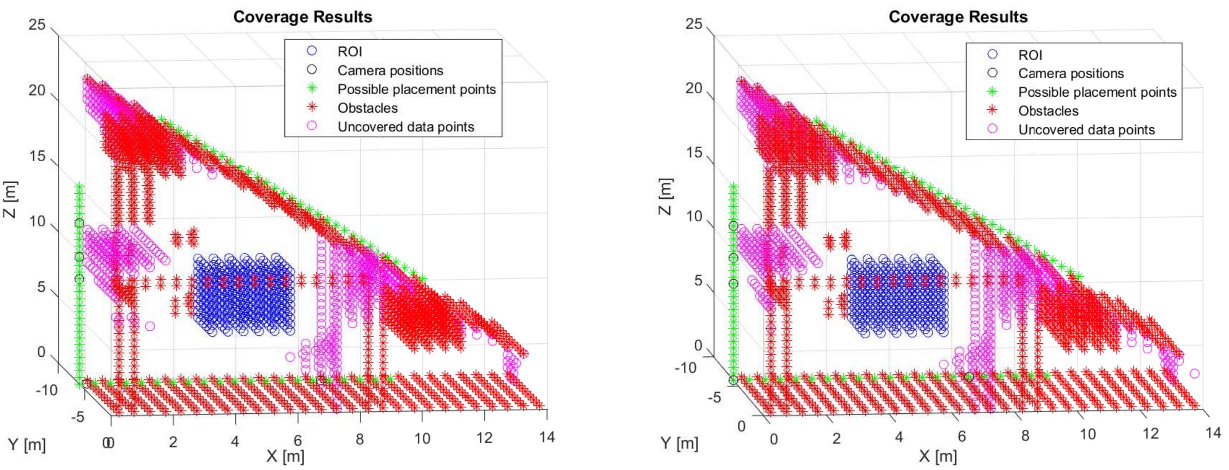
Algorithm	Brute Force Algorithm	Particle Swarm Algorithm
X Positions	[0 0 0 0 7.5]	[0 0 0.24 7.75]
Z Positions	[0 7.5 9.5 12 0]	[7.99 9.67 12.25 0 0]
Pan Angles [rad]	[$\pi/4$ 0 0 $\pi/4$ $\pi/4$]	[0.03 -0.03 0.77 0.80 0.81]
Coverage %	91.5	92
ROI Coverage %	100	100

As can be seen from the table, the Particle Swarm Algorithm improves the solution by 0.5%, which translates to covering 42 voxels more than the discrete BFA and GA covered. Fig. 4.31 and 4.32 shows a comparison between the BFA coverage and continuous coverage in 2D and 3D, respectively. As can be seen from the figures, the improvement in coverage for the continuous optimization method is not visually significant although the coverage is increased by 42 voxels.



(a) 2D Coverage for the Continuous Optimization (b) 2D Coverage for the Brute Force Optimization

Figure 4.31: 2D Comparison of BFA and Continous Optimization Results



(a) 3D Coverage for the Continuous Optimization (b) 3D Coverage for the Brute Force Optimization

Figure 4.32: 3D Comparison of BFA and Continous Optimization Results

4.7.5 Case Study Results

Using the presented methods in this section, a case study has been conducted where all aspects of this thesis have been used to perform a Sensor Placement task in an environment with several obstacles and a Region of Interest which requires coverage from 2 cameras. The User Interface was used to define the problem according to the problem description.

By first using the Greedy Algorithm the problem was analyzed, and information regarding the pan angles and the number of cameras could be extracted from the results. Based on this, the Genetic Algorithm was used to reduce the problem further. The results from the GA were highly consistent which indicates that optimal, or at least close to optimal, solutions have been found. The Genetic Algorithm found a feasible solution for minimum five cameras. As mentioned in the problem description for the case study, it was also desirable to investigate the increase in coverage for more cameras than required, if the increase in coverage is considerable. By increasing the number of cameras from five to six, the GA found an increase in coverage of 4.6%.

The GPU-Based Brute Force Algorithm was used to verify that the solution from the GA was indeed the global optimum. To conduct a Brute Force analysis, it is vital to reduce the problem to a minimum due to the complexity and required computational time of this method. The Brute Force analysis is a comprehensive task, but the results were highly successful, showing that the global optimum was found in the GA for the given problem description.

Since the above methods are conducted using a discrete optimization formulation, it was desirable also to conduct a neighborhood search using one of Matlab's optimization methods. A continuous problem statement was formulated and solved for five cameras. It was found an improved solution which covers 0.5% more voxels than the previous solution.

Tab. 4.17 shows the sensor position for both the discrete and continuous optimization results for five cameras and the GA result for six cameras.

Table 4.17: Results of the Final Case Study

Algorithm	**BFA**	**PSO**	**GA**
Number of Cameras	5	5	6
X Positions	[0 0 0 7.5]	[0 0 0.24 7.75]	[0,0,0,0,8,8.81]
Z Positions	[0 7.5 9.5 12 0]	[7.99 9.67 12.25 0 0]	[0,6,9.5,13.5,0,10.93]
Pan Angles [rad]	$[\frac{\pi}{4}, 0, 0, \frac{\pi}{4}, \frac{\pi}{4}]$	[0.03 -0.03 0.77 0.80 0.81]	$[\frac{\pi}{4}, \frac{\pi}{4}, 0, \frac{\pi}{4}, \frac{\pi}{4}, \frac{-3\cdot\pi}{4}]$
Coverage %	91.5	92	96.1
ROI Coverage %	100	100	100

Depending on the cost of adding another sensor, it could be decided to add another camera, opting for the GA solution for six cameras. However, since the problem stated a requirement of $> 90\%$ coverage, the solution for five cameras is feasible and most likely cheaper than the solution for six cameras. This section has demonstrated the work of this thesis and utilized both the User Interface and developed algorithms.

5 | Discussion and Further Work

In this chapter, the most significant results and methods are presented and discussed. Based on the problem description, the main topics of this thesis is the development of a User Interface and development and comparison of different optimization strategies. Finally, suggestions are made for further work in the field of Sensor Placement Problems.

5.1 User Interface

There have been several User Interfaces designed for various aspects of the sensor placement problem. However, there does not seem to be any open-source User Interface for the 3D Sensor Placement Problem with the versatility presented in this thesis. The UI can be used to produce a platform for optimization algorithms, aiming to provide a consensus for methods to solve Sensor Placement Problems.

In the developed User Interface, it is possible to define the environment and its constraints based on a 3D model stored as a VRML file. The UI is designed in Matlab and is available open-source in the GitHub repository. Matlab was chosen as the programming software due to its broad library of functions for both 3D models and development of UIs. Although the software is not open-source, it is commonly used in both academia and industry.

Development of the UI was a tremendous task since several different functions are needed for the UI to be useful in a variety of practical applications. The User Interface is robust and straightforward to use, especially considering the appended user manual which describes the required steps to define the environment. The code is structured to make bug fixes and functionality additions as simple as possible. All scripts and functions are also adequately commented for easier understanding.

The UI exports a JSON file which is a popular file type due to its compatibility with all popular programming languages. It is readable by humans due to the use of JavaScript notation. The UI output can also be exported as a MAT-file to be used in Matlab. Therefore, optimization algorithms can be developed using any language compatible with JSON. Although Matlab is easy to use regarding syntax and documentation, it can be seen that other programming languages are superior concerning computational speed and versatility.

5.2 GPU-Based Brute Force Algorithm

Several authors have previously claimed that a Brute Force search for the global optimum is practically impossible due to the complexity of the problem. This thesis has shown that by converting the Brute Force Algorithm to CUDA, a full search has been made possible in many sensor placement problems.

The developed GPU-based method is at least 100 times as fast as a CPU method for complex problems. This is significant and increases the threshold for which problem can be solved using the developed BFA. It has been scaled for the GTX1080, which is currently one of the best GPUs available from NVIDIA. One of the main limitations when scaling problems for a GPU is the accessible memory, which is only ≈ 8[GB]. In the final case study, the BFA was conducted for five cameras, where 51 chops were required. This number could have been decreased slightly since the available RAM in the computer developing the combinations matrix was 8 [GB]. If a computer was available with Matlab and RAM size of, e.g., 16 [GB], the number could have been reduced. Since CUDA is not compatible with the json11 library, a separate program was needed to generate the arrays containing information about the problem: the coverage matrix and the annotations array. Also, a program was needed to produce the chopped combinations matrices, which currently is conducted in Matlab. Since the annotations array and the coverage matrix only need to be computed once for a given problem, it is not seen as a problem to use the presented method. However, it would be beneficial to develop a method for generating the combinations matrices in CUDA, since a considerable speedup is most likely achievable by converting the code from Matlab to C++, in addition to eliminating the need for a secondary computer. Using a GPU to accelerate algorithms has given successful results, and should be considered if many parallel operations are to be done independently.

5.3 Genetic Algorithm

Some research exists on the development of Genetic Algorithms for the Sensor Placement Problem, or other closely related problem, however much of this research is limited to case-specific applications or weakly documented. The algorithm details have often been excluded nor has the algorithm been verified to produce the optimal result.

The Genetic Algorithm developed in this thesis has been shown to produce the optimal result in almost all performed tests. The algorithm uses conservative selection methods, focusing strongly on fitness when selecting parents and in the crossover phase. Too conservative parameters and schemes can often make the GA act too much like a hill-climbing algorithm, deeming it to end up in local minima. The proposed algorithm avoids this with the use of a high mutation factor, i.e., a high percentage of the children will have one of their features mutated randomly. Considering a sufficiently sized population, parent pool, children pool and number of children per generation this will, in most cases, introduce the required randomness to avoid the algorithm getting stuck in local minima.

Compared to the GPU-Based BFA, the GA is much faster, e.g., in the final test where the BFA needed ≈ 36 hours the GA produced the same result in ≈ 15 minutes. The computational time can most likely be reduced further by converting the problem to C++. The results show that the developed GA is very successful in finding the optimal placement, verified by the BFA.

If the GA is to be used alone, without verification by the BFA, there are several methods to be used to get the best possible results. Some suggested methods are:

- **Run the algorithm several times:**

  This is perhaps the most efficient method for ensuring a good solution. If the result is consistent with several algorithm executions, it is most likely the optimal result. It is essential that the positions must also be consistent, not only the overall coverage since, for complex problems, several configurations can result in the same overall coverage.

- **Select a sufficiently large population size and number of generations:**

  The population size and number of generations are the two most important variables for ensuring the best possible results. This has been shown in Sec. 4.4.2 where parameter tuning for the GA was investigated. For the final test, the number of individuals and number of generations was determined to be

$$n_i = n_s \cdot 1000 \tag{5.1}$$
$$n_g = n_s \cdot 100 \tag{5.2}$$

  where:

n_i	-	Number of individuals in the population
n_g	-	Number of generations
n_s	-	Number of sensor to be placed

  It should be noted that these numbers were applicable 5 sensors, 3 available pan angles and 134 possible placement points in an environment of 9084 grid points. Considering that the number of children generated per generations is equal to $\frac{3 \cdot n_i}{20} = 750$ children, the algorithm created 380 000 unique individuals, in total. Compared to the number of possible combinations, $8.53e10$, the number of individuals is very small.

Although the results of the Genetic Algorithm are highly satisfactory, it must be emphasized that it can not guarantee to find the global optimum every time for any parameter combination since it has several random aspects. However, for all tests in this thesis, the GA has found the best solution more often than not, indicating that the developed algorithm works very good for the sensor placement problem.

5.4 Further Work

Throughout this thesis, some of the most critical aspects of the Sensor Placement Problem has been researched; however, there are several fields where further work is proposed.

GPU-Based Genetic Algorithm

The Genetic Algorithm has been developed in Matlab due to its easy syntax and vast library compared to C++. Also, since the computational time is low compared to, e.g., the BFA, it was not required to convert the algorithm to C++ in this thesis. Suggested further work would be to convert the Genetic Algorithm to C++ to enable CUDA programming and GPU utilization.

Converting algorithms from C++ to CUDA has been shown to be highly beneficial in this thesis. The Genetic Algorithm has several parallel aspects which can be transformed to CUDA, which can result in massive reductions in computational time. Firstly, the initialization of the population is independent for each individual, meaning that it could be transformed into a kernel function to be executed on the GPU. Next, the evaluation of each individual could also be done in a kernel call, much like the kernel function used in the BFA. Selection of parents is also a parallel process which could be beneficial to parallelize using CUDA. There are several aspects of the developed GA algorithm that can be converted to kernel functions to achieve a considerable speedup after the code has been converted to C++.

Multiple Sensor Models

Often, multiple sensor models are available with different parameters and price. By extending the problem to handle multiple sensor models at once, the algorithms need slight modifications. For the Genetic Algorithm, there are no major problems related to this, other than developing a new objective function. By determining coverage for all sensor types at all possible locations, the algorithm could be used similarly by adding a feature to each individual where the sensor cost is stored. The new objective function to be maximized could be:

$$\mathcal{O} = \zeta \cdot \mathcal{C} - (1 - \zeta) \sum_{i=1}^{n_s} \mathcal{P}_i \tag{5.3}$$

where:

$\mathcal{O}$	-	Objective function
ζ	-	Weighting variable ($\in (0, 1)$)
$\mathcal{C}$	-	Coverage value
$\mathcal{P}$	-	Price per sensor i

By varying ζ, either coverage or total price can be weighted. Implementing multiple sensor models in the BFA provides more difficulties since the total number of combinations increases massively. Therefore, a proposed solution would be to use the Genetic Algorithm to determine the sensor type for each sensor and use the BFA to optimize sensor placement.

Dynamic Scene

In many situations, the environment to be observed is not static. Obstacles are considered to be static in this thesis, but many obstacles, e.g., robots, can move inside defined areas in practice. By including moving obstacles, the aspect of time is included in the Sensor Placement Problem. This could be done in several ways, depending on how the problem is formulated. The most probable formulation is for specific objects to move in pre-defined paths. If the position of every obstacle is known at every point in time, it should be possible to include this in the placement algorithm. The suggested approach is first to extend the User Interface to handle the dynamic scene by inputting multiple files at given time steps and comparing the files to determine which voxels are moving and which are static. Then, by restricting the movement pattern to be linear between each input file, the estimated position can be found for each voxel given the difference in time between two files. The number of input files provides the accuracy of this linearization.

The optimization problem should then be formulated to cover a given percentage of the voxels in the scene at any time step between the first and last input file, sorted chronologically dependent on the time step. One way to do this could be to define a large matrix where the position of each obstacle is stored at each time step. Then, the visible voxels can be determined from each possible camera location given the pose of the sensor at all time steps.

The next step would be to modify the algorithm to take the time into account and solve the optimization problem. There are probably other methods of solving the problem of dynamic scenes, but it is of the author's belief that the proposed method could at least act as an inspiration for further work in this topic.

6 | Conclusion

The literature study showed that the Sensor Placement Problem is a complex problem with several aspects which need to be considered. There seems to be little consensus regarding how to best solve the problem, and several algorithms have been developed and tested in the literature. In this thesis, a method of solving the problem is presented, by first designing a User Interface, which determines the environment and sets the constraints for the optimization algorithm which is responsible for determining the best sensor positions and poses given the constraints.

The User Interface is developed in Matlab, where the user can specify the constraints and define the environment of the optimization problem. The user can specify sensor parameters, available sensor positions, and regions of interest where redundancy is required. The output is a JSON file where all necessary information is stored in arrays.

To optimally place sensors in the defined environment, several methods are considered. A traditional approach is to use a Greedy Algorithm to get approximate solutions. The Greedy Algorithm has been proven to be sub-optimal; therefore other algorithms were desired. Two algorithms are developed in this thesis, a Genetic Algorithm, and a GPU-Based Brute Force Algorithm.

The Genetic Algorithm is inspired by the natural process of evolution. The appropriate genetic operators are defined for the Sensor Placement Problem and have provided excellent results. The selection pressure is high for both the crossover and selection schemes, but the mutation rate should be kept high to ensure the necessary random aspect to avoid the algorithm being stuck in local minima.

Since the problem is NC-complete, the Genetic Algorithm can never fully guarantee the global optimum solution. The only method of being able to guarantee this is by doing a Brute Force search of all feasible solutions in the search space. In this thesis, a Brute Force Algorithm is developed and adopted on a NVIDIA GTX 1080 GPU using CUDA programming. By incorporating the parallel nature of vital parts of the algorithm to a kernel function, a considerable speedup has been achieved. Comparing the Genetic Algorithm and the GPU-Based Brute Force Algorithm, the former is seen to have a much lower computational time whereas the latter always guarantees the optimal result. Therefore, for problems where the Brute Force method is applicable within a reasonable amount of time, and a NVIDIA GPU is available, the suggested method is the GPU-Based Brute Force Algorithm. However, since this is not always the case, the developed Genetic Algorithm has also been proven to find the global optimum with correct parameter tuning for all verifiable tests in this thesis. Since the algorithm converges much faster than the BFA, it can be executed several times to ensure consistent results for the given test.

Bibliography

[1] R. L. Rivest, C. E. Leiserson, and T. H. Corman, *Introduction to Algorithms*. New York, NY, USA: McGraw-Hill, Inc., 1990.

[2] V. Chvátal, "A combinatorial theorem in plane geometry," *Journal of Combinatorial Theory, Series B*, vol. 18, no. 1, pp. 39 – 41, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0095895675900611

[3] N. Kirchhof, "Optimal placement of multiple sensors for localization applications," in *International Conference on Indoor Positioning and Indoor Navigation*, Oct 2013, pp. 1–10.

[4] K. A. Tarabanis, P. K. Allen, and R. Y. Tsai, "A survey of sensor planning in computer vision," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 1, pp. 86–104, Feb 1995.

[5] E. Hörster and R. Lienhart, "On the optimal placement of multiple visual sensors," in *Proceedings of the 4th ACM International Workshop on Video Surveillance and Sensor Networks*, ser. VSSN '06. New York, NY, USA: ACM, 2006, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/1178782.1178800

[6] Bianco and Tisato, "Sensor placement optimization in buildings," pp. 8300 – 8300 – 13, 2012. [Online]. Available: http://dx.doi.org/10.1117/12.911021

[7] U. M. Erdem and S. Sclaroff, "Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements," *Computer Vision and Image Understanding*, vol. 103, no. 3, pp. 156 – 169, 2006, special issue on Omnidirectional Vision and Camera Networks. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314206000671

[8] . K. Altınel, N. Aras, E. Güney, and C. Ersoy, "Binary integer programming formulation and heuristics for differentiated coverage in heterogeneous sensor networks," *Computer Networks*, vol. 52, no. 12, pp. 2419 – 2431, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128608001436

[9] J. Dybedal and G. Hovland, "Optimal placement of 3d sensors considering range and field of view," in *2017 IEEE International Conference on Advanced Intelligent Mechatronics (AIM)*, July 2017, pp. 1588–1593.

[10] A. Mittal and L. S. Davis, "A general method for sensor planning in multi-sensor systems: Extension to random occlusion," *International Journal of Computer Vision*, vol. 76, no. 1, pp. 31–52, Jan. 2008. [Online]. Available: https://doi.org/10.1007/s11263-007-0057-9

[11] V. Akbarzadeh, C. Gagné, M. Parizeau, and M. A. Mostafavi, "Black-box optimization of sensor placement with elevation maps and probabilistic sensing models," in *2011 IEEE International Symposium on Robotic and Sensors Environments (ROSE)*, Sept 2011, pp. 89–94.

[12] V. Akbarzadeh, J.-C. Lévesque, C. Gagné, and M. Parizeau, "Efficient sensor placement optimization using gradient descent and probabilistic coverage," *Sensors*, vol. 14, no. 8, pp. 15 525–15 552, 2014. [Online]. Available: http://www.mdpi.com/1424-8220/14/8/15525

[13] T. T. Nguyen, H. D. Thanh, L. H. Son, and V. T. Le, "Optimization for the sensor placement problem in 3d environments," in *2015 IEEE 12th International Conference on Networking, Sensing and Control*, April 2015, pp. 327–333.

[14] H. Topcuoglu, M. Ermis, I. Bekmezci, and M. Sifyan, “A new three-dimensional wireless multimedia sensor network simulation environment for connected coverage problems,” *Simulation*, vol. 88, no. 1, pp. 110–122, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1177/0037549710383291

[15] S. Fleishman, D. Cohen-Or, and D. Lischinski, “Automatic camera placement for image-based modeling,” in *Proceedings. Seventh Pacific Conference on Computer Graphics and Applications (Cat. No.PR00293)*, 1999, pp. 12–20, 315.

[16] R. Chabra, A. Ilie, N. Rewkowski, Y. W. Cha, and H. Fuchs, “Optimizing placement of commodity depth cameras for known 3d dynamic scene capture,” in *2017 IEEE Virtual Reality (VR)*, March 2017, pp. 157–166.

[17] *An Interactive Camera Placement and Visibility Simulator for Image-Based VR Applications*, 2006.

[18] P. Rahimian and J. K. Kearney, “Optimal camera placement for motion capture systems,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 3, pp. 1209–1221, March 2017.

[19] L. Zhang, J. Tang, and W. Zhang, “Strong barrier coverage with directional sensors,” in *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE.* IEEE, 2009, pp. 1–6.

[20] J. Cortes, S. Martinez, T. Karatas, and F. Bullo, “Coverage control for mobile sensing networks,” *IEEE Transactions on Robotics and Automation*, vol. 20, no. 2, pp. 243–255, April 2004.

[21] M. Schwager, B. J. Julian, M. Angermann, and D. Rus, “Eyes in the sky: Decentralized control for the deployment of robotic camera networks,” *Proceedings of the IEEE*, vol. 99, no. 9, pp. 1541–1561, Sept 2011.

[22] A.R.Forrest, “Ii. current developments in the design and production of three- dimensional curved objects - computational geometry,” *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 321, no. 1545, pp. 187–195, 1971. [Online]. Available: http://rspa.royalsocietypublishing.org/content/321/1545/187

[23] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer, 2008. [Online]. Available: https://www.amazon.com/Computational-Geometry-Applications-Mark-Berg-ebook/dp/B014P9HOKU?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B014P9HOKU

[24] G. B. R. Carray and C. Marrin, “The virtual reality modeling language,” 1997.

[25] MeshLab, “Meshlab,” http://www.meshlab.net/#.

[26] MathWorks, “Virtual reality modeling language,” https://se.mathworks.com/help/sl3d/vrml.html.

[27] ——, “Introduction to patch objects,” https://se.mathworks.com/help/matlab/visualize/introduction-to-patch-objects.html#f2-17145, retrieved 13.02.2018.

[28] Mathworks, “vrifs2patch,” https://se.mathworks.com/help/sl3d/vrifs2patch.html, retrieved 13.02.2018.

[29] MathWorks, “vrpatch2ifs,” https://se.mathworks.com/help/sl3d/vrpatch2ifs.html, retrieved 13.02.2018.

[30] ——, “Multifaceted patches,” https://se.mathworks.com/help/matlab/visualize/multifaceted-patches.html0, retrieved 13.02.2018.

[31] J. O'rourke, *Art gallery theorems and algorithms.* Oxford University Press Oxford, 1987, vol. 57.

[32] S. Fisk, "A short proof of chvátal's watchman theorem," *Journal of Combinatorial Theory, Series B*, vol. 24, no. 3, p. 374, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/009589567890059X

[33] H. Ma and Y. Liu, "On coverage problems of directional sensor networks," in *Mobile Ad-hoc and Sensor Networks*, X. Jia, J. Wu, and Y. He, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 721–731.

[34] J. Peng, J. Jingqi, W. Chengdong, and H. Nan, "A coverage detection and re-deployment algorithm in 3d directional sensor networks," in *The 27th Chinese Control and Decision Conference (2015 CCDC)*, May 2015, pp. 1137–1142.

[35] V. Akbarzadeh, C. Gagné, M. Parizeau, M. Argany, and M. A. Mostafavi, "Probabilistic sensing model for sensor placement optimization based on line-of-sight coverage," *IEEE Transactions on Instrumentation and Measurement*, vol. 62, no. 2, pp. 293–303, 2013.

[36] Z. Zhou, S. Das, and H. Gupta, "Connected k-coverage problem in sensor networks," in *Proceedings. 13th International Conference on Computer Communications and Networks (IEEE Cat. No.04EX969)*, Oct 2004, pp. 373–378.

[37] C. Zhu, C. Zheng, L. Shu, and G. Han, "A survey on coverage and connectivity issues in wireless sensor networks," *Journal of Network and Computer Applications*, vol. 35, no. 2, pp. 619 – 632, 2012, simulation and Testbeds. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804511002323

[38] D. Stone, C. Jarrett, M. Woodroffe, and S. Minocha, *User interface design and evaluation.* Morgan Kaufmann, 2005.

[39] W. O. Galitz, *The essential guide to user interface design: an introduction to GUI design principles and techniques.* John Wiley & Sons, 2007.

[40] J. Nocedal and S. Wright, *Numerical Optimization (Springer Series in Operations Research and Financial Engineering).* Springer, 2006. [Online]. Available: https://www.amazon.com/Numerical-Optimization-Operations-Financial-Engineering/dp/0387303030?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0387303030

[41] R. Baldick, *Applied Optimization: Formulation and Algorithms for Engineering Systems.* Cambridge University Press, 2006. [Online]. Available: https://www.amazon.com/Applied-Optimization-Formulation-Algorithms-Engineering-ebook/dp/B00E3UR0YW?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B00E3UR0YW

[42] R. Fletcher, *Practical Methods of Optimization.* Wiley, 2000. [Online]. Available: https://www.amazon.com/Practical-Methods-Optimization-R-Fletcher/dp/0471494631?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0471494631

[43] A. Antoniou and W.-S. Lu, *Practical Optimization: Algorithms and Engineering Applications.* Springer, 2007. [Online]. Available: https://www.amazon.com/Practical-Optimization-Algorithms-Engineering-Applications/dp/0387711066?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0387711066

[44] H. Rosenbrock, "An automatic method for finding the greatest or least value of a function," *The Computer Journal*, vol. 3, no. 3, pp. 175–184, 1960.

[45] D. A. Simovici, *Linear Algebra Tools For Data Mining.* World Scientific Publishing Company, 2012. [Online]. Available: https://www.amazon.com/Linear-Algebra-Tools-Data-Mining/dp/981438349X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=981438349X

[46] D. L. Kreher and D. R. Stinson, *Combinatorial algorithms: generation, enumeration, and search.* CRC press, 1998, vol. 7.

[47] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms (Algorithms and Combinatorics).* Springer, 2005. [Online]. Available: https://www.amazon.com/Combinatorial-Optimization-Theory-Algorithms-Combinatorics/dp/3540256849?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=3540256849

[48] K.-L. Du and M. N. S. Swamy, *Search and Optimization by Metaheuristics: Techniques and Algorithms Inspired by Nature.* Birkhäuser, 2016. [Online]. Available: https://www.amazon.com/Search-Optimization-Metaheuristics-Techniques-Algorithms/dp/3319411918?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=3319411918

[49] J. Potvin and M. Gendreau, *Handbook of Metaheuristics (International Series in Operations Research & Management Science).* Springer, 2010. [Online]. Available: https://www.amazon.com/Handbook-Metaheuristics-International-Operations-Management/dp/1441916636?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1441916636

[50] M. Fleischer, "Simulated annealing: past, present, and future," in *Winter Simulation Conference Proceedings, 1995.*, Dec 1995, pp. 155–161.

[51] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, Nov 1995, pp. 1942–1948 vol.4.

[52] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[53] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: a tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, Mar 1996.

[54] F. Glover, "Tabu search—part i," *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

[55] G. Dieter and D. Bacon, *Mechanical Metallurgy (Materials Science & Engineering)*, 1988. [Online]. Available: https://www.amazon.com/Mechanical-Metallurgy-Materials-Science-Engineering/dp/0071004068?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0071004068

[56] Y. Nourani and B. Andresen, "A comparison of simulated annealing cooling strategies," *Journal of Physics A: Mathematical and General*, vol. 31, no. 41, p. 8373, 1998.

[57] E. Balas *et al.*, "A class of location, distribution and scheduling problems: Modeling and solution methods," 1982.

[58] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations.* Springer, 1972, pp. 85–103.

[59] U. Feige, "A threshold of ln n for approximating set cover," *J. ACM*, vol. 45, no. 4, pp. 634–652, Jul. 1998. [Online]. Available: http://doi.acm.org/10.1145/285055.285059

[60] R. Hassin and A. Levin, "A better-than-greedy approximation algorithm for the minimum set cover problem," *SIAM Journal on Computing*, vol. 35, no. 1, pp. 189–200, 2005.

[61] J. Beasley and P. Chu, "A genetic algorithm for the set covering problem," *European Journal of Operational Research*, vol. 94, no. 2, pp. 392 – 404, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/037722179500159X

[62] MathWorks, "Json format," https://se.mathworks.com/help/matlab/json-format.html, retrieved 04.05.2018.

[63] Dropbox, "json11," https://github.com/dropbox/json11, retrieved 04.05.2018.

[64] Nvidia, "What is gpu-accelerated computing?" http://www.nvidia.com/object/what-is-gpu-computing.html, retrieved 06.04.2018.

[65] ——, "Cuda c programming guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Mar. 2018, retrieved 06.04.2018.

[66] ——, "Cuda c best practices guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, Mar. 2018, retrieved 06.04.2018.

[67] ——, "Nvidia launches the world's first graphics processing unit: Geforce 256," http://www.nvidia.com/object/IO_20020111_5424.html, Aug. 1999, retrieved 06.04.2018.

[68] A. Fox, "Mte explains: The difference between a cpu and a gpu," https://www.maketecheasier.com/difference-between-cpu-and-gpu/, Jan. 2017, retrieved 05.04.2018.

[69] S. Holcombe, "inpolyhedron - are points inside a triangulated volume?" https://se.mathworks.com/matlabcentral/fileexchange/37856-inpolyhedron-are-points-inside-a-triangulated-volume-, Nov. 2015, retrieved 04.02.2018.

[70] ——, "unifymeshnormals," https://se.mathworks.com/matlabcentral/fileexchange/43013-unifymeshnormals, Sep. 2013, retrieved 04.02.2018.

[71] R. Code, "Combinations," https://rosettacode.org/wiki/Combinations, May 2018, retrieved 22.05.2018.

[72] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989. [Online]. Available: https://www.amazon.com/Genetic-Algorithms-Optimization-Machine-Learning/dp/0201157675?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201157675

[73] B. L. Miller, D. E. Goldberg *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[74] C.-F. Huang and Y.-C. Tseng, "The coverage problem in a wireless sensor network," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 519–528, Aug 2005. [Online]. Available: https://doi.org/10.1007/s11036-005-1564-y

[75] Y. C. Wang and Y. C. Tseng, "Distributed deployment schemes for mobile wireless sensor networks to ensure multilevel coverage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 9, pp. 1280–1294, Sept 2008.

[76] Stackoverflow, "Fastest solution for all possible combinations, taking k elements out of n possible with k>2 and n large," https://stackoverflow.com/questions/29136239/fastest-solution-for-all-possible-combinations-taking-k-elements-out-of-n-possi/, retrieved 08.05.2018.

[77] A. Nazarovsky, "Mex-writematrix," https://github.com/nazarovsky/mex-writematrix, Mar. 2015, retrieved 08.05.2018.

[78] MathWorks, "Fmincon," https://se.mathworks.com/help/optim/ug/fmincon.html, retrieved 16.05.2018.

[79] ——, "Globalsearch," https://se.mathworks.com/help/gads/globalsearch.html, retrieved 16.05.2018.

[80] ——, "Simulannealbnd," https://se.mathworks.com/help/gads/simulannealbnd.html, retrieved 16.05.2018.

[81] ——, "Particleswarm," https://se.mathworks.com/help/gads/particleswarm.html, retrieved 16.05.2018.

A | Appendix

A.1 Project Proposal

Prosjektforslag 35: Master Mekatronikk, Vår 2018

Forslag til Masteroppgave, vår 2018
Kontaktperson: Geir Hovland (UiA) og Jan-Einar Gravdal (IRIS)

Tittel: Optimalisering av 3D Sensor Plassering i en Rigg

I denne masteroppgaven er målet å utvikle en optimeringsmetode for å finne en optimal plassering av 3D sensorer i et område, typisk en rigg, men løsningen som skal utvikles bør også kunne benyttes i andre anvendelsesområder.

Figur 1: Eksempel på 3D Simulering fra Virtual Arena ved IRIS.

Oppgaven vil være en videreutvikling / forbedring av arbeidet presentert i artikkelen: Joacim Dybedal and Geir Hovland, «Optimal placement of 3D sensors considering range and field of view», http://ieeexplore.ieee.org/document/8014245

Oppgaven vil inneholde følgende elementer:

- Lage et grafisk CAD-Basert brukergrensesnitt hvor optimeringsproblemet kan defineres. Parametere som er aktuelle er: størrelsen på det aktuelle området, type 3D sensorer, antall 3D sensorer, pris per sensor, rekkevidde og synsfelt per sensor, aktuelle monteringspunkter for sensorene, redundans (dvs. områder som må være innenfor synsfeltet til mer enn en sensor).
- Det grafiske grensesnittet skal utvikles i Matlab / Simulink 3D
- Basert på randbetingelsene spesifisert i det grafiske brukergrensesittet, utvikle og programmere en optimeringsalgoritme som finner det beste valg av sensorer og den beste sensorplasseringen.
- I første del av oppgaven skal det gjennomføres et litteratursøk for å finne ut av state-of-the-art for denne type problemstillinger. Deretter skal en optimeringsmetodikk velges ut og implementeres.

Figure A.1: Problem Proposal

A.2 GUI: User Manual

This document is written as a guide for the Matlab GUI: Sensor Placement Optimization.

A.2.1 Intended Use

The UI is made to simplify the problem formulation of the Sensor Placement Problem in 3D for sensors defined by the field of view, range, and price. The UI supports **.wrl* (VRML) files with nodes defined as Indexed Face Set as its input. The process for defining the problem follows the procedure shown below, with further explanations throughout this document.

1. Import VRML model
2. Add floor
3. Add cameras
4. Define placement lines
5. Define regions of interest
6. Define optimization parameters
7. Generate optimization code
8. Visualize results

A.2.2 About

The GUI is made as part of a master thesis at the University of Agder 2018 by Vegard Tveit.

A.2.3 Requirements

The program is made using Matlab 2017a. It is assumed that the user has some programming experience with Matlab, but most of the code is properly commented for easier understanding and bug fixes. It is recommended to use separate software for interpreting, changing or converting the 3D VRML files. A proposed software is Meshlab.

In the VRML file, the geometry nodes must be defined as indexed face set. Also, the *transform* node must be defined as *layout*. In the VRML file, this should look as following: *DEF layout Transform {*.

The user should have some knowledge regarding computational geometry and optimization for best use of the software, but this is not required. The software is set up for an example program where the user is only required to follow the necessary steps without doing any modifications. By being able to see the intended options for a given problem, the user can get a better understanding of how to use the UI.

A.2.4 File

As a safety feature, in case the program crashes, or errors are made, the UI stores information for each significant step in **.mat* files. If the program shuts down and is started again, the **.mat* files can be used as a method of restoring previous data. This method replaces the commonly used *save as* and *load* functions, but the result is the same.

A.2.5 Edit

The *Edit* menu contains the function for defining the problem.

Edit VRML

The *Edit VRML* menu opens Matlab's VRML editor. The full documentation for this program can be seen in `https://se.mathworks.com/help/sl3d/the-3d-world-editor.html` (retrieved 23.01.2018), and will not be described in this document.

Floor

To determine the shape of the polygon enclosed by the walls, a function for adding a floor to the problem space is included. The "Add Floor" menu opens a figure window and a dialog box where the corner coordinates are specified. There are no limits on the number of defined corners. The coordinates should be separated by spaces, and it should be checked that there are equally many y- and z-coordinates as x-coordinates. To get the corner coordinates, the data cursor in the figure toolbar can be used. Another tip is when in "Rotate 3D" mode, the view can be changed by right-clicking in the figure window. By choosing XZ view, it is easier to get the correct x- and z-coordinates. When the dialog box is closed, the floor is added to the figure for visualization and saved to the Matlab workspace as well as added to the Indexed Face Set in the VRML model.

Camera

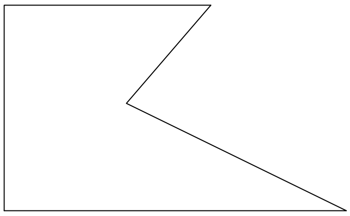
The camera menu is the most extensive sub-menu, where the sensors are defined as well as the placement lines for the sensors. Firstly, the camera parameters are specified in the "Add Camera" menu. Additionally, the price can be specified for each sensor in any desired unit in the dialog box.

After the cameras are defined, the user needs to specify where the sensors can be placed. This is done in the "Add Lines" menu, which opens a dialog box and a figure. The lines should be straight and defined by the start and end point of the line. Similarly to the procedure of adding the floor, the coordinates can be found using the data cursor tool. When the close button is pushed, the dialog box closes, and the lines are saved to the Matlab workspace. The lines that should be used for sensor placement must have the *Accepted* choice checked. In this way, the program understands which lines to use, and which to ignore. When the lines are specified, *lines.mat* is saved with the information from the dialog box. This mat file must not be deleted before the JSON file is generated since it will be used later in the program.

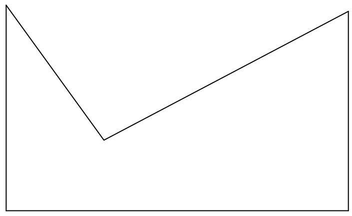
The lines can be visualized using the "Visualize Lines" menu. A new dialog box is opened, where the accepted lines are displayed. By pressing the *Visualize* button, a figure window opens with the 3D model (shown in red) as well as the placement lines (shown in blue).

Region of Interest

A region of Interest enables the user to have k-coverage possibilities. First, the user should specify whether the grid generation should be along the vertical or horizontal axis. To get a better understanding of why this is relevant, two figures are shown in Fig. A.2. Fig. A.2b shows an example of a polygonal floor plan, where it would be easiest to generate the fishnet along the vertical axis. The reason for this is that for this polygon, every point along the vertical axis represents a unique point at the polygonal edge. Contrarily, in Fig. A.2a, every point along the horizontal axis represents a unique point along the polygonal edge. It should be noted that the grid generation starts at the bottom left corner of the polygon.



(a) Figure for Horizontal Grid Generation



(b) Figure for Vertical Grid Generation

Figure A.2: Directions of Grid Generation

The next step is to add the Region of Interest. This is done by clicking the "Add Region of Interest" option, which opens an input window and a figure of the generated grid of the polygon representing the floor. The user has to specify the coordinates for the Region of Interest according to Fig. A.3.

In Fig. A.3, a ROI volume is presented for visualization. The subscript of either 0 or 1 indicates the corner coordinates at either Y0 or Y1, respectively.

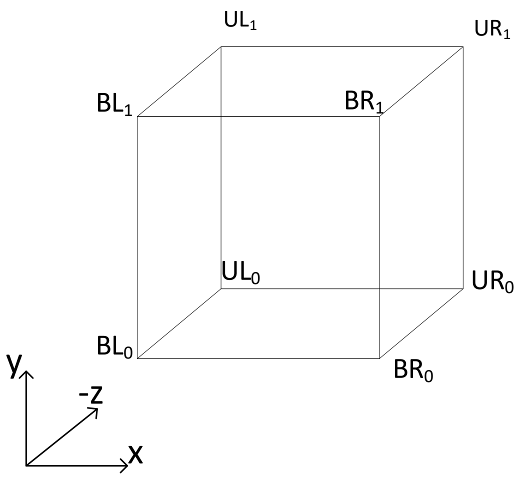


Figure A.3: Region of Interest Cube Coordinates and Indices

Additionally, the user has to specify the weight option, either 1, 2 or 3. A weight of 1 corresponds to a 2-covered region of interest, which is the standard option. 2 is a 3-covered region of interest, and 3 in a zero-covered region of interest, which means that the region is of no interest concerning coverage.

The user can add multiple ROIs, and display them in a list using the "Visualize Region of Interest" function, which also displays the region of interest in the 3D scene.

A.2.6 Optimization

Setup Parameters

Here, the main parameters for the optimization accuracy are defined. Firstly, the room height at floor and roof level must be established. Next, the height of the voxels is specified by determining the total number of voxels in the vertical direction. Finally, the placement lines accuracy is defined by specifying the number of placement points per length unit along the defined lines.

Generate JSON

When the parameters are defined, as well as all other necessary parameters are set up to describe the problem, the JSON file can be generated. Generating the JSON file may take some time. When the file is created, a figure is shown that graphically shows the output regarding obstacles, placement points, and regions of interest.

A.2.7 Visualization

In the *Visualization* menu, the result from the optimization algorithm can be seen.

Load Optimization Results

When *Load Optimization Results* is chosen, the user must specify the position *(x,y,z)* of all sensor to be placed along with the number of sensors and their respective pan angles. For now, the sensor parameters are considered to be fixed, but this can easily be changed in the callback functions. The user must also specify the filename of the **.mat* file where the *optim* data is stored from the UI output.

A.3 Test Data for the Genetic Algorithm

Table A.1: Mutation Factor Test: A1

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	749	749	749	749	732	749	749	749	736	749	749	749	749	749	749	747	5.33
1/4	731	749	736	731	731	749	749	736	749	732	749	736	749	749	749	741.67	8.29
1/3	749	749	749	749	749	732	749	749	749	732	749	736	731	749	749	744.67	7.51
1/5	749	732	731	749	749	749	720	749	736	749	731	749	736	749	749	741.80	9.79
2/3	736	749	749	749	749	749	749	749	749	749	749	749	749	749	749	748.13	3.36
1/6	731	749	731	749	749	749	749	736	749	749	749	749	749	732	749	744.60	7.63
3/4	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	0

Table A.2: Mutation Factor Test: A2

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	736	731	749	731	749	749	736	749	732	732	736	749	719	731	731	737.33	9.41
1/4	725	749	731	732	732	732	732	749	749	731	731	736	707	732	749	734.47	11.20
1/3	749	719	720	732	749	731	731	749	749	725	731	749	749	736	749	737.87	11.61
1/5	719	736	731	732	736	717	717	731	736	719	749	749	720	732	731	730.33	10.42
2/3	749	749	749	731	749	732	736	732	736	749	749	749	736	736	720	740.13	9.41
1/6	749	731	736	732	749	749	731	732	725	749	736	736	731	736	732	736.93	8.05
3/4	749	749	749	731	749	749	749	749	731	749	749	736	749	749	731	744.53	7.75

Table A.3: Mutation Factor Test: A3

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	0
1/4	749	749	749	749	749	749	749	749	749	749	736	749	749	749	749	748.13	3.36
1/3	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	0
1/5	749	749	749	749	749	749	749	749	732	749	749	749	749	749	749	747.87	4.39
2/3	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	0
1/6	749	749	749	749	749	736	749	749	732	749	749	749	749	749	749	747	5.33
3/4	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	749	0

Table A.4: Mutation Factor Test: B1

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	945	945	945	945	945	945	945	945	945	945	945	945	937	945	945	944.47	2.07
1/4	937	924	937	937	924	945	945	936	937	945	928	945	937	945	945	937.80	7.53
1/3	945	936	945	936	945	945	945	945	937	945	937	936	937	945	945	941.60	4.32
1/5	945	937	936	937	937	945	931	945	937	937	937	937	937	937	928	937.53	4.67
2/3	937	945	945	945	945	945	945	945	945	945	937	945	945	945	945	943.93	2.81
1/6	930	930	937	930	937	945	928	936	937	926	926	937	945	937	936	934.47	6.00
3/4	945	937	945	945	945	945	945	945	945	945	945	945	945	945	945	944.47	2.07

Table A.5: Mutation Factor Test: B2

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	945	937	945	945	945	945	945	937	937	945	945	945	945	933	945	942.60	4.22
1/4	945	945	945	945	945	937	945	937	931	937	937	937	945	937	945	940.87	4.81
1/3	945	937	945	937	945	945	945	945	945	945	932	945	945	945	945	943.07	4.15
1/5	945	937	931	945	932	937	937	945	937	945	937	933	937	945	945	939.20	5.27
2/3	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	0
1/6	937	937	945	945	928	937	945	924	931	945	928	937	937	945	945	937.73	7.28
3/4	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	0

Table A.6: Mutation Factor Test: B3

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	0
1/4	945	937	933	945	945	945	945	945	945	936	937	945	945	936	937	941.40	4.66
1/3	945	945	945	945	937	937	945	936	937	936	937	937	945	937	945	940.60	4.27
1/5	937	945	945	937	937	936	928	937	937	937	945	931	945	932	945	938.27	5.60
2/3	945	945	945	945	945	945	945	945	937	945	945	945	945	945	945	944.47	2.07
1/6	928	937	945	926	937	945	937	933	945	922	945	922	936	925	937	934.67	8.40
3/4	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	945	0

Table A.7: Mutation Factor Test: C1

m_f	**Test no.**															**Avg.**	**Std.**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	0
1/4	1070	1085	1086	1086	1068	1068	1086	1068	1068	1086	1086	1086	1086	1082	1075	1079.07	8.32
1/3	1086	1073	1077	1086	1072	1075	1068	1086	1086	1071	1086	1086	1068	1086	1086	1079.47	7.56
1/5	1086	1086	1086	1069	1076	1085	1068	1076	1086	1085	1085	1086	1077	1086	1071	1080.53	6.94
2/3	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1082	1082	1086	1086	1085.47	1.41
1/6	1076	1068	1072	1086	1086	1077	1085	1068	1069	1068	1086	1077	1085	1082	1086	1078.07	7.53
3/4	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	0

Table A.8: Mutation Factor Test: C2

m_f	**Test no.**															**Avg.**	**Std.**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	1070	1073	1082	1085	1085	1086	1086	1086	1082	1086	1085	1086	1085	1086	1067	1082	6.45
1/4	1082	1061	1086	1065	1086	1077	1065	1085	1067	1086	1070	1085	1073	1076	1055	1074.60	10.36
1/3	1068	1086	1086	1086	1086	1077	1067	1068	1082	1086	1086	1068	1085	1085	1085	1080.07	8.04
1/5	1072	1068	1082	1077	1059	1056	1071	1068	1072	1064	1082	1082	1077	1068	1082	1072	8.40
2/3	1086	1086	1086	1085	1086	1086	1086	1086	1077	1086	1086	1086	1086	1086	1085	1085.27	2.31
1/6	1069	1064	1067	1085	1067	1071	1063	1082	1070	1086	1068	1064	1071	1065	1070	1070.80	7.49
3/4	1086	1086	1086	1086	1086	1086	1086	1086	1085	1086	1086	1086	1086	1085	1086	1085.87	0.35

Table A.9: Mutation Factor Test: C3

m_f	**Test no.**															**Avg.**	**Std.**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	1086	1082	1082	1086	1086	1085	1086	1086	1086	1086	1086	1070	1086	1082	1086	1084.07	4.22
1/4	1085	1086	1068	1086	1086	1085	1086	1072	1070	1082	1071	1086	1068	1082	1086	1079.93	7.59
1/3	1085	1082	1068	1086	1082	1086	1085	1069	1068	1082	1086	1071	1086	1072	1086	1079.60	7.53
1/5	1086	1077	1067	1072	1071	1086	1068	1086	1054	1068	1069	1067	1068	1058	1068	1071	9.42
2/3	1086	1086	1086	1086	1086	1086	1086	1085	1086	1086	1086	1086	1086	1086	1086	1085.93	0.26
1/6	1073	1086	1067	1077	1082	1065	1070	1069	1063	1086	1086	1069	1064	1061	1085	1073.53	9.30
3/4	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	1086	0

Table A.10: Mutation Factor Test: D1

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	1195	1186	1195	1195	1194	1184	1186	1186	1186	1186	1195	1180	1195	1181	1195	1189.27	5.70
1/4	1180	1184	1186	1195	1174	1178	1179	1190	1179	1195	1187	1195	1180	1163	1177	1182.80	8.86
1/3	1183	1195	1176	1193	1182	1195	1195	1181	1187	1190	1183	1195	1184	1179	1176	1186.27	7.08
1/5	1180	1180	1190	1180	1186	1190	1179	1181	1189	1184	1179	1183	1187	1182	1181	1183.40	4.01
2/3	1195	1186	1195	1195	1195	1194	1181	1189	1195	1191	1195	1195	1195	1194	1186	1192.07	4.50
1/6	1182	1171	1185	1175	1184	1182	1179	1184	1186	1182	1174	1187	1162	1161	1182	1178.40	8.23
3/4	1195	1194	1195	1195	1195	1194	1195	1195	1194	1195	1191	1195	1194	1194	1189	1194	1.73

Table A.11: Mutation Factor Test: D2

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	1195	1190	1195	1195	1194	1195	1195	1195	1195	1186	1195	1195	1190	1182	1194	1192.73	4.01
1/4	1164	1194	1189	1185	1185	1170	1181	1182	1183	1194	1177	1184	1180	1195	1195	1183.87	9.04
1/3	1194	1195	1195	1181	1173	1185	1195	1185	1195	1195	1179	1181	1184	1187	1187	1187.40	7.15
1/5	1195	1186	1183	1194	1192	1195	1180	1191	1195	1182	1190	1188	1182	1187	1166	1187.07	7.79
2/3	1186	1195	1184	1194	1194	1195	1195	1195	1187	1193	1195	1195	1195	1195	1195	1192.87	3.81
1/6	1177	1165	1186	1194	1191	1186	1179	1183	1186	1182	1185	1186	1194	1179	1181	1183.60	7.26
3/4	1194	1195	1195	1195	1195	1195	1195	1194	1195	1195	1193	1195	1195	1195	1195	1194.73	0.59

Table A.12: Mutation Factor Test: D3

m_f	Test no.															Avg.	Std.
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**		
1/2	1195	1195	1181	1195	1195	1195	1194	1195	1195	1194	1194	1195	1195	1194	1186	1193.20	4.07
1/4	1168	1190	1181	1194	1195	1195	1180	1186	1181	1181	1195	1191	1194	1180	1195	1187.07	8.22
1/3	1183	1185	1194	1190	1164	1189	1195	1186	1195	1195	1195	1183	1195	1195	1186	1188.67	8.33
1/5	1185	1180	1192	1190	1194	1184	1186	1184	1176	1186	1189	1194	1183	1188	1178	1185.93	5.42
2/3	1194	1195	1194	1195	1195	1195	1195	1194	1195	1195	1195	1195	1194	1195	1195	1194.73	0.46
1/6	1182	1179	1187	1187	1179	1184	1194	1178	1182	1192	1179	1176	1173	1184	1194	1183.33	6.43
3/4	1195	1195	1195	1195	1190	1195	1195	1195	1195	1195	1195	1195	1195	1195	1195	1194.67	1.30

A.4 Test Data for the Final Test Brute Force Algorithm

Tab. A.13 and A.14 shows the result of each chop in the BFA analysis with 5 cameras. For each chop, the time used in Matlab to generate the combinations matrix is provided along with the time required to write the $*.txt$ file. Also, the last combination is listed to show the range of each chop. The best combination is given along with the coverage result for this combination.

Table A.13: BFA Best Coverage Result Chop 1-25 Final Test

Chop	**Generate [s]**	**Write [s]**	**Last**	**Best**	**Coverage [%]**
1	2354.2	305.8	0 149 175 176 246	0 45 57 73 139	84.9
2	2316.2	327.3	1 141 201 223 229	1 45 57 73 139	91.5
3	2317.9	330.1	2 148 150 207 211	2 33 45 55 136	77.3
4	2333.8	329.8	3 187 192 215 229	3 45 57 73 139	85.9
5	2313.8	330.1	5 8 37 167 211	4 45 57 73 136	89.8
6	2295.43	329.4	6 13 23 60 175	5 33 45 55 136	77.3
7	2311.0	330.7	7 19 53 122 209	6 45 57 73 139	86.6
8	2305.4	328.9	8 27 70 201 221	7 57 73 136 224	89.7
9	2320.7	337.3	9 38 41 103 164	9 36 48 58 139	84.3
10	2360.5	332.5	10 52 70 219 232	10 51 73 136 218	87.7
11	2326.3	342.0	11 72 171 201 202	10 57 73 136 224	87.9
12	2351.2	340.7	12 110 136 144 183	12 45 57 73 139	87.5
13	2334.2	335.1	14 20 91 113 162	13 57 73 136 224	86.2
14	2342.5	333.6	15 38 53 101 169	15 36 48 58 139	84.3
15	2346.9	339.8	16 63 121 138 184	15 45 57 73 139	87.6
16	2422.4	351.4	17 114 123 156 177	17 33 45 55 136	77.3
17	2447.1	333.0	19 32 151 153 210	18 45 60 76 139	88.0
18	2363.6	341.1	20 63 81 108 206	19 45 79 133 209	84.2
19	2352.2	342.7	21 139 145 146 165	21 45 70 139 221	88.1
20	2345.9	336.4	23 48 75 101 240	22 45 79 133 209	83.1
21	2358.3	340.6	24 104 105 184 221	24 46 54 85 139	88.7
22	2345.3	338.5	26 87 109 152 168	25 45 85 133 203	82.6
23	2380.4	342.8	27 114 124 126 191	27 48 85 139 203	88.5
24	2355.9	341.8	29 60 68 154 161	28 49 57 94 136	83.1
25	2384.5	348.4	31 36 68 221 232	30 45 52 88 136	87.6

Table A.14: BFA Best Coverage Result Chop 1-25 Final Test

Chop	**Generate [s]**	**Write [s]**	**Last**	**Best**	**Coverage [%]**
25	2384.5	348.4	31 36 68 221 232	30 45 52 88 136	87.6
26	2367.8	340.5	32 97 151 184 238	31 52 57 94 136	84.4
27	2405.6	349.4	34 68 143 187 237	33 45 55 91 136	86.6
28	2389.9	340.7	36 54 108 237 244	36 54 58 94 139	87.2
29	2400.3	341.4	38 47 89 182 248	37 45 58 94 139	87.2
30	2419.6	350.4	40 45 67 222 244	39 54 61 94 139	88.5
31	2411.8	353.6	42 47 77 95 133	42 45 64 94 139	88.0
32	2396.6	354.8	44 54 58 158 206	42 57 64 94 139	89.7
33	2528.0	374.0	46 66 155 178 197	45 57 73 94 139	91.5
34	2732.2	362.2	48 90 181 212 227	48 57 73 94 136	91.4
35	2448.7	345.3	50 187 193 208 212	49 51 57 94 136	82.9
36	2470.5	351.3	53 84 114 160 186	51 57 73 94 136	91.4
37	2470.7	353.1	56 63 123 160 246	54 57 73 94 136	91.2
38	2467.4	352.4	58 149 156 231 241	57 60 76 100 127	85.0
39	2474.3	353.6	61 142 150 178 215	60 85 94 136 203	49.6
40	2505.1	347.7	65 69 85 164 177	63 79 94 142 224	49.6
41	2471.7	347.1	68 96 106 187 210	66 79 94 142 224	49.8
42	2481.8	353.0	72 81 100 189 237	69 82 94 142 224	49.9
43	2498.1	355.4	76 86 188 240 248	72 82 91 142 224	49.7
44	2477.7	351.0	80 121 154 189 245	79 94 142 209 224	50.5
45	2495.6	351.5	85 124 198 229 232	85 94 142 203 224	50.6
46	2494.8	352.5	91 109 113 179 237	86 94 142 200 224	49.9
47	2572.9	367.6	98 101 112 117 145	94 42 167 203 224	43.0
48	2508.5	359.8	106 115 149 159 169	100 142 167 203 224	39.7
49	2520.4	355.7	116 171 200 209 230	106 145 167 203 224	36.5
50	2527.7	349.5	132 195 211 215 234	118 148 167 203 224	31.3
51	2571.3	351.5	180 197 201 220 245	145 167 203 224 248	28.7
52	191.6	215.3	0 0 0 0 0	182 203 221 236 248	19.5

A.5 Matlab Scripts for the Set-Cover Problem

A.5.1 Set Cover Greedy Algorithm

```
% Greedy Heuristic Algorithm for
% the set cover example
% Author: Vegard Tveit
% Date: 03.04.2018


% note that this algorithm
% is written specifically
% for the set cover
% example problem

clear;
clc;

% Initialize universe
U = ones(12,1);
Usize = length(U);

% Define subsets
S = [1 0 1 0 0 0;...
    1 0 0 1 0 0;...
    1 0 0 0 1 0;...
    1 0 1 0 0 0;...
    1 1 0 1 0 0;...
    1 1 0 0 1 0;...
    0 0 1 1 0 0;...
    0 1 0 1 0 0;...
    0 1 0 0 1 0;...
    0 0 1 0 0 1;...
    0 0 0 1 0 1;...
    0 0 0 0 1 0];

S = S';
bOK = true;
covered = zeros(Usize,1);

while(bOK)
    mat = zeros(6,1);


    mat = zeros(6,1);
    for i = 1:Usize
        if covered(i) == 0
            for j = 1:6
                if S(j,i) == 1
                    mat(j) = mat(j) + 1;
                end
             end
        end
    end

    maxi = max(mat);
    sol = find(mat == maxi);
    sol = sol(1);
```

```

    for k = 1:Usize
        if S(sol,k) == 1
            covered(k) = 1;
        end
    end
    disp('New sol found')
    disp(sol)
    disp('Coverage')
    disp(covered)

    if(sum(covered) == Usize)
        bOK = false;
    end
end
```

A.5.2 Set Cover Exact Algorithm

```
% Exact algorithm for
% the set cover example
% Author: Vegard Tveit
% Date: 03.04.2018


% note that this algorithm
% is written specifically
% for the set cover
% example problem

clear;
clc;

% Initialize universe
U = ones(12,1);
Usize = length(U);

% Define subsets
S = [1 0 1 0 0 0;...
    1 0 0 1 0 0;...
    1 0 0 0 1 0;...
    1 0 1 0 0 0;...
    1 1 0 1 0 0;...
    1 1 0 0 1 0;...
    0 0 1 1 0 0;...
    0 1 0 1 0 0;...
    0 1 0 0 1 0;...
    0 0 1 0 0 1;...
    0 0 0 1 0 1;...
    0 0 0 0 1 0];

Sub_vec = 1:1:6;
S = S';
bOK = true;
num_subs = 1;
maxim = 0;
while(bOK)
    comb_vec = nchoosek(Sub_vec,num_subs);
    for i = 1:nchoosek(length(Sub_vec),num_subs)
        curr_comb = zeros(1,Usize);
```

```
        for j = 1:num_subs
            index = comb_vec(i,j);
             curr_comb = curr_comb | S(index,1:end);
        end
        if sum(curr_comb) > maxim
            maxim = sum(curr_comb);
            indout = [i,j];
        end
    end
    if maxim == Usize
        bOK = false;
        fprintf('Best solution found with %i subsets \n',num_subs)
        fprintf('Subsets at indexes ')
        fprintf('%i ',comb_vec(indout(1),1:end))
        fprintf('fully covers the universe \n')
    else
        fprintf('Increasing the number of subsets \n')
        num_subs = num_subs + 1;
    end
end
```

A.6 Matlab Functions for the Sensor Placement Problem

A.6.1 Coverage Function

```
function b = covered1(i,x,y,z,datax,datay,dataz,obsta,pan)
% Function to determine the  coverage of point i
% in the data points array from sensor placed at (x,y,z)
% given the pan angle (pan).

% Camera parameters (valid for all cameras)
range = 9;
tilt = pi/6;
fov = 45 * pi/180;

% Point to locate
xp = datax(i);
yp = dataz(i);
zp = datay(i);
% Norm of (x,y,z),(xp,yp,zp)
L = sqrt((xp-x)^2+(zp-z)^2+(yp-y)^2);
% XY angle
xya = atan2(zp-z,xp-x);
% XZ angle
xza = atan2(yp-y,L);
% Determine if point is covered and visible from
% current sensor config. b = 1 if covered and visible
if (L < range)
    if ((pan-fov) <= xya) && (xya <= (pan+fov))
        if ((tilt-fov) <= xza) && (xza <= (tilt+fov))
          % Evaluate all obstacle points to ensure that no obstacle points block
          % the sensor coverage of point i. b = true if covered and visible
          for jj = 1:length(obsta)
            xoi = obsta(jj,1);
              yoi = obsta(jj,3);
              zoi = obsta(jj,2);
              oi_L = sqrt((xoi-x)^2+(zoi-z)^2+(yoi-y)^2);
```

```
                oi_xya = atan2(zoi-z,xoi-x);
                db = 0.25;
                oi_xza = atan2(yoi-y,oi_L);
                if ((pan-fov) <= oi_xya) && (oi_xya <= (pan+fov))
                  if(abs(xya-oi_xya) < db)
                      if((tilt-fov) <= oi_xza) && (oi_xza <= (tilt+fov))
                          if(abs(xza-oi_xza) < db)
                              if(L > oi_L)
                                  b = false;
                              else
                                  b = true;
                              end
                          else
                              b = true;
                          end
                      else
                          b = true;
                      end
                  else
                      b = true;
                  end
                else
                    b = true;
                end

              end

        else
            b = false;
        end
    else
        b = false;
    end
else
    b = false;
end
```

A.6.2 Environmental Function

```
function env = environment_generate(my_matfile)
% Function to produce a struct (env) of the UI output
% from the generated *.mat file

inp = load(my_matfile);
env.datax = inp.optim{4};
env.datay = inp.optim{5};
env.dataz = inp.optim{6};
env.campx = inp.optim{2};
env.campz = inp.optim{3};
% Note: Defined as -6 for this specific problem
env.campy(1:length(env.campx)) = -6;
env.annot = inp.optim{1};

oc = 0;
% Determine obstacles
for ka = 1:length(env.annot)
    if env.annot(ka) == 1
        oc = oc + 1;
        env.obsta(oc,1:3) = [env.datax(ka),env.datay(ka),env.dataz(ka)];
```

```
    end
end
```

A.6.3 Function for Finding the Camera Position and Pan Angle

```
function [panx,camx] = EvalNum(npans,i)
% function to translate the combinations indices
% to the indices of the camera position and pan arrays

ic = i - 1;
pc = floor(ic/npans);
panx = i - pc*npans;
camx = pc + 1;
```

A.7 Matlab Scripts for the Genetic Algorithm

A.7.1 Genetic Algorithm for the Sensor Placement Problem

```
%% Genetic Algorithm for the Sensor Placement Problem
% Author: Vegard Tveit
% Date: 20.04.2018
% Comment: Genetic algorithm with crossover and mutation operators

clear;
clc;
close all;

env = environment_generate('optim2.mat');
datax = env.datax;                          % Discrete data points (x)
datay = env.datay;                          % Discrete data points (y)
dataz = env.dataz;                          % Discrete data points (z)
campx = env.campx;                          % Discrete placement points (x)
campy = env.campy;                          % Discrete placement points (y)
campz = env.campz;                          % Discrete placement points (z)
obsta = env.obsta;

% Define number of pan angles
pans = [0,pi/2,-3*pi/4];

% Preallocate variables and initialize
ldata = length(datax);                      % Number of data points
lpans = length(pans);                       % Number of pan angles
lcamp = length(campx);                      % Number of placement points

x = zeros(lcamp,1);                         % Placement point array (x)
y = zeros(lcamp,1);                         % Placement point array (y)
z = zeros(lcamp,1);                         % Placement point array (x)
b = false(lpans,ldata);                     % Coverage matrix - Combs
iter = 0;                                   % Indexation counter

% Compute coverage for all possible camera poses and positions
for i = 1:lcamp
    x(i) = campx(i);                        % Get camera position (x)
    y(i) = campy(i);                        % Get camera position (y)
    z(i) = campz(i);                        % Get camera position (z)
```

```

    % Loop through all pan angles
    for k = 1:lpans
        pan = pans(k);                         % Get pan angle
        iter = iter + 1;                       % Update indexation

        % Compute coverage of all data points
        for n = 1:ldata
            b(iter,n) = covered1(n,x(i),y(i),z(i),datax,datay,dataz,...
                                    obsta,pan);
        end
    end
end

% Initalize pool of chromosomes randomly
numchromo = 900;
numsubs = iter;
usize = ldata;
numsens = 2;
chromo = zeros(numchromo,numsens);
for i = 1:numchromo
    for j = 1:numsens
        ind = randi(numsubs,1);
        chromo(i,j) = ind;
    end
end

b = b';
% Evaluate fitness of chromosomes
fitmat = zeros(numchromo,1);
for m = 1:numchromo
    bch = false(usize,1);
    for n = 1:numsens
        inde = chromo(m,n);
        bch(1:end,1) = bch(1:end,1) | b(1:end,inde);
    end
    fitmat(m) = sum(bch);
end
avgsum_init = sum(fitmat)/length(fitmat);

% Select best parents for next generation
[sortarr,indarr] = sort(fitmat,'descend');

% Make pool of parent solutions
% MUST BE AN EVEN NUMBER
numpar_gen = 3*numchromo/5;
max_generations = 90;
generations = 1;

childreninto = zeros(numpar_gen/4-1,numsens);
while(generations < max_generations)
% Probability of being chosen
prob_par = zeros(numpar_gen,1);
for pk = 1:numpar_gen
    prob_par(pk,1)=fitmat(indarr(pk))/sum((fitmat(indarr(1:numpar_gen))));
end

% Select half of the pool for reproduction
repIter = 1;
bRep = true;
par_out = zeros(numpar_gen/2,1);
```

```
for i = 1:numpar_gen/2
    randRep = rand();
    bRep = true;
    repIter = 1;
    while(bRep)
        randRep = randRep - prob_par(repIter);
        if randRep < 0
            bRep = false;
            indeRep = repIter;
        end
        repIter = repIter + 1;
    end
    par_out(i) = indeRep;
end
% par_out now contains the indices of the individuals
% in the mating pool. These chormosomes should undergo
% crossover and mutation

par = indarr(1:numpar_gen,1);

children_out = zeros(numpar_gen/2-1,numsens);
for i = 1:numpar_gen/2-1
    par1 = par_out(i);
    par2 = par_out(i+1);

    % Mutation Rate
    p_m = 3/4;

    % Crossover function
    CroP1 = sort(chromo(par1,1:end));
    CroP2 = sort(chromo(par2,1:end));

    scP1 = fitmat(par1);
    scP2 = fitmat(par2);

    ProbP1 = scP1/(scP1+scP2);
    ProbP2 = 1-ProbP1;


    % Cut
    rCro = rand();
    ChiCro = zeros(numsens,1);
    for Ci = 1:numsens
        if(CroP1(Ci) == CroP2(Ci))
            ChiCro(Ci) = CroP1(Ci);
        end
        if(CroP1(Ci) ~= CroP2(Ci))
            if rCro <= ProbP1
                ChiCro(Ci) = CroP1(Ci);
            else
                ChiCro(Ci) = CroP2(Ci);
            end
        end
    end

     randMutProb = rand();
     if(randMutProb < p_m)

        bMut = true;
        arrMut = zeros(numsubs,1);

```

```
        rMut = randi(numsens,1);
        for Mi = 1:numsubs
            for Ni = 1:numsens
                if(ChiCro(Ni) == Mi)
                    arrMut(Mi) = 1;
                end
            end
        end
        hMut = find(arrMut == 0);
        riMut = randi(length(hMut),1);
        ChiCro(rMut) = hMut(riMut);
     end

children_out(i,1:numsens) = ChiCro;
end

% Select best half+1 of the children to bring into the population
fitchildren = zeros(numpar_gen/2-1,1);
for m = 1:numpar_gen/2-1
    bch = false(usize,1);
    for n = 1:numsens
        inde = children_out(m,n);
        bch(1:end,1) = bch(1:end,1) | b(1:end,inde);
    end
    fitchildren(m) = sum(bch);
end

[chilsorted,indchild] = sort(fitchildren,'descend');
% Into population : childreninto ->
childreninto(1:numpar_gen/4-1,1:end) = ...
    children_out(indchild(1:numpar_gen/4-1,1:end),1:end);

% New population
for i = 1:length(childreninto)
   chromo(indarr(end-i),1:end) = childreninto(i,1:end);
end

% Fitness of new population
fitmat = zeros(numchromo,1);
for m = 1:numchromo
    bch = false(usize,1);
    for n = 1:numsens
        inde = chromo(m,n);
        bch(1:end,1) = bch(1:end,1) | b(1:end,inde);
    end
    fitmat(m) = sum(bch);
end

% Select best parents for next generation
[sortarr,indarr] = sort(fitmat,'descend');
generations = generations + 1;
end
outVal = chromo(indarr(1),1:end);

% Post process
[panx,camx] = EvalNum(lpans,outVal);
% Display Results
disp('Camera Positions x : ')
campx(camx)
disp('Camera Positions z : ')
campz(camx)
```

```
disp('Pan Angles [rad] : ')
pans(panx)
```

A.8 C++ Scripts

A.8.1 Sensor Model Test Case

```
// C++ Code for placement of two pre-defined cameras for a given JSON file with visibility model
// Author : Vegard Tveit
// Date : 22.02.2018
// Comment : This code uses the json11 library found at :
// https://github.com/dropbox/json11

// Initial Setup
#include <iostream>
#include "json11.hpp"
#include <string>
#include <fstream>
#include <vector>
#include <cmath>

using namespace json11;
using std::string;
typedef std::vector<Json> array;
static void parse_from_stdin() {
// Initialize strings, ints etc
string buf;
string line;
int i = 0;
double mat[2];
double val = 0;
double val_annot = 0;
double outval = 0;
int j = 0;
long long int ii; //gpu thread index (to be used later)
string err,mystr;
std::string mystrings;
std::string teststring;
std::string annotations;

while (std::getline(std::cin, line)) {
    buf += line + "\n";
}
// Pass data from json file
auto json = Json::parse(buf, err);

/*
Input Json Array Index :
    0 : Annotations
        1 : Camera Points (x)
        2 : Camera Points (z)
    3 : Data Points (x)
    4 : Data Points (y)
    5 : Data Points (z)
    6 : Camera Parameters
*/
/* Store JSON data into arrays
    annot : Double array of annotations
    datax : Double array of data points (x)
    datay : Double array of data points (y)
    dataz : Double array of data points (z)
```

```

    Note: Camera poisions - Fixed in y -direction at -6

    camx : Camera positions (x)
    camz : Camera poisitons (z)
*/
// Store annotations to annot
double annot[json[0].array_items().size()];
for(auto &value :json[0].array_items()){

    annotations = value.dump();
    std::string::size_type sz;
    val_annot = stod(annotations,&sz);
    annot[j] = val_annot;
    j = j + 1;
}
// Store data points (x) to datax
double datax[json[3].array_items().size()];
int kx = 0;
for(auto &valuex :json[3].array_items()){
    std::string dataxs;
    double valx;
        dataxs = valuex.dump();
        std::string::size_type sz;
        valx = stod(dataxs,&sz);
        datax[kx] = valx;
        kx = kx + 1;
}

// Store data points (y) to datay
double datay[json[4].array_items().size()];
int ky = 0;
for(auto &valuey :json[4].array_items()){
        std::string datays;
        double valy;
        datays = valuey.dump();
        std::string::size_type sz;
        valy = stod(datays,&sz);
        datay[ky] = valy;
        ky = ky + 1;
}
// Store data points (z) to dataz
double dataz[json[5].array_items().size()];
int kz = 0;
for(auto &valuez :json[5].array_items()){
    std::string datazs;
    double valz;
    datazs = valuez.dump();
    std::string::size_type sz;
    valz = stod(datazs,&sz);
    dataz[kz] = valz;
    kz = kz + 1;
}

// Store camera positions (x) to camposx
double camposx[json[1].array_items().size()];
int kcx = 0;
for(auto &value_cx :json[1].array_items()){
        std::string camposxs;
        double camx;
        camposxs = value_cx.dump();
```

```
        std::string::size_type sz;
        camx = stod(camposxs,&sz);
        camposx[kcx] = camx;
        kcx = kcx + 1;
}
// Store camera positions (z) to camposz
double camposz[json[2].array_items().size()];
int kcz = 0;
for(auto &value_cz :json[2].array_items()){
        std::string camposzs;
        double camz;
        camposzs = value_cz.dump();
        std::string::size_type sz;
        camz = stod(camposzs,&sz);
        camposz[kcz] = camz;
        kcz = kcz + 1;
}

// Determine number of obstacle points
int cot0 = 0;
for(int k0 = 0 ; k0 < json[0].array_items().size() ; k0++){
        if(annot[k0] != 0){
                cot0 = cot0 + 1;
        }
}

// Determine all obstacle coordinates and collect into double arrays
int coto = 0;
double obstx[cot0];
double obsty[cot0];
double obstz[cot0];

for(int ko = 0 ; ko < j ; ko++){
    if(annot[ko] != 0){
        double obstxx = datax[coto];
        double obstyy = datay[coto];
        double obstzz = dataz[coto];

        obstx[coto] = obstxx;
        obsty[coto] = obstyy;
        obstz[coto] = obstzz;
        coto = coto + 1;
    }
}

// Test sensor placement
// Initialize camera parameters for both cameras
int numkk = 0;
double xcam1, zcam1, xcam2, zcam2;
// Pan and tilt angles should be defined in [-pi,pi]
const double pi = 3.1415926535897;
double pan =-3* pi/4;           // Pan (Rot X of the camera)
double fov = 60*(pi/180);       // Half of the field of view
double tilt = 0;            // Tilt (Rot Z of the camera)
double range = 8;           // The range of the camera
double x,y,z,x2,y2,z2;
double pan2 = 0;
double fov2 = 60 * (pi/180);
double tilt2 = 0;
double range2 = 8;

```

```
// Initialize loop help variables
double max = 0;
double maxk = 0;
int numk = 0;
int num = 0;
bool b;
int iter = 0;
// Loop through all camera positions for camera 1
for(int j_k = 0 ; j_k < kcz; j_k++) {
    // Set current camera 1 position
    x = camposx[j_k];
    y = -4;
    z = camposz[j_k];
    // Initialize sum variable
    int sum = 0;

    // Loop through all camera positions for camera 2
    for(int k = 0 ; k < kcz ; k++){
        // Set current camera 2 position
        x2 = camposx[k];
        y2 = -4;
        z2 = camposz[k];

        // Initialize sum variable
        int sumk = 0;
        // Loop through all data points for the current camera position
        for(int i = 0; i < ky ; i++){
            // Set current data point set
            double xp = datax[i];
            double yp = dataz[i];
            double zp = datay[i];
            // Determine distance from sensor to data point
            double L = std::sqrt(std::pow(xp-x,2) + std::pow(zp-z,2) + std::pow(yp-y,2));
            double La = std::sqrt(std::pow(xp-x2,2) + std::pow(zp-z2,2) + std::pow(yp-y2,2));
            // Determine angle between sensor and data point (XY plane)
            double xya = std::atan2((zp-z),(xp-x));
            double xya2 = std::atan2((zp-z2),(xp-x2));
            // Determine angle between sensor and data point (XZ plane)
            double xza = std::atan2((yp-y),L);
            double xza2 = std::atan2((yp-y2),La);

            // Determine if point is seen by camera 1
            if(L < range){
                if( (pan - fov) <= xya && xya <= (pan+ fov) ){
                    if( ((tilt - fov) <= xza) && (xza <= (tilt + fov)) ){
                        // Determine visibility
                        for(int vk = 0 ;vk <  coto ; vk++){
                            //Set GPU thread index (to be used later)
                            ii = vk + coto*i + coto*ky*k + coto*ky*kcz*j_k;
                            //std::cout << ii << std::endl;

                            double xobs = obstx[vk];
                            double yobs = obsty[vk];
                            double zobs = obstz[vk];
                            double o_L = std::sqrt(std::pow(xobs-x,2) +
                            std::pow(zobs-z,2) + std::pow(yobs-y,2));
                            double o_xya = std::atan2((zobs-z),(xobs-x));
                            double thr = 0.25;
                            double o_xza = std::atan2((yobs-y),o_L);

                            // Is obstacle point within sensing range
```

```
                        b=1;
                        if((pan - fov) <= o_xya && o_xya <= (pan + fov) ){
                            if(std::abs(xya - o_xya) <  thr){
                                if( ((tilt - fov) <= o_xza) && (o_xza <= (tilt + fov)) ){
                                    if(std::abs(xza - o_xza) < thr){
                                        if(L > o_L){
                                            b = 0;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
                else{
                    b = 0;
                }
            }
            else{
                b = 0;
            }
        }
        // Determine if point is seen by camera 2
        else if(La < range2){
             if( (pan2 - fov2) <= xya2 && xya2 <= (pan2+ fov2) ){
                 if( (tilt2 - fov2) <= xza2 && xza2 <= (tilt2 + fov2) ){
                     // Determine visibility
                    for(int vk1 = 0 ;vk1 <  coto ; vk1++){
                        double xobs1 = obstx[vk1];
                        double yobs1 = obsty[vk1];
                        double zobs1 = obstz[vk1];
                        double o_L1 = std::sqrt(std::pow(xobs1-x,2) +
                        std::pow(zobs1-z,2) + std::pow(yobs1-y,2));
                        double o_xya1 = std::atan2((zobs1-z),(xobs1-x));
                        double thr1 = 0.25;
                        double o_xza1 = std::atan2((yobs1-y),o_L1);

                        // Is obstacle point within sensing range
                        b = 1;
                        if((pan2 - fov2) <= o_xya1 && o_xya1 <= (pan2 + fov2) ){
                            if(std::abs(xya2 - o_xya1) <  thr1){
                                if( ((tilt2 - fov2) <= o_xza1) &&
                                (o_xza1 <= (tilt2 + fov2)) ){
                                    if(std::abs(xza2 - o_xza1) < thr1){
                                        if(La > o_L1){
                                            b = 0;
                                        }
                                    }
                                }

                            }
                        }
                    }
                    }
                    else{
                        b = 0;
                    }
                }
                else{
                    b = 0;
                }
```

```
            } else{
                b = 0;

            }
// Sum for the current camera position increases by one if the above conditions are fullfilled
            sumk = sumk +  b;
    }
// End of inner loop (data points)
    // If the sum of the current camera position is better than the previous best
    // set current camera 1 and 2  position to best position for the given  camera 1 position
    if(sumk > maxk){
        maxk = sumk;
        numk = k;
        }
    }
    // If the sum for the current camera 1 and camera 2 combination is better
    // than the previous best, set the current positions to best
    if(maxk > max){
        max = maxk;
        num = j_k;
        numkk = numk;
        xcam1 = x;
        zcam1 = z;
        xcam2 = x2;
        zcam2 = z2;
    }
}

// Print results
std::cout << std::endl << std::endl << std::endl;

std::cout <<  "The best camera positions is : X : " << camposx[num] << " Y : " << camposz[num]
<<
 " and " << " X : " << camposx[numkk] << " Y : " << camposz[numkk] <<  std::endl;
}
// Main execution
int main(int argc, char **argv) {
    if (argc == 2 && argv[1] == string("--stdin")) {
        parse_from_stdin();
        return 0;
    }
}
```

A.8.2 Heuristic Greedy Algorithm

```
// C++ Code for iterative placement of n pre-defined cameras for a given
//JSON file with visibility model with variable pan angle
// Author : Vegard Tveit
// Date : 25.02.2018
// Comment : This code uses the json11 library found at :
// https://github.com/dropbox/json11

// Initial Setup
#include <iostream>
#include "json11.hpp"
#include <string>
#include <fstream>
#include <vector>
#include <cmath>
```

```
using namespace json11;
using std::string;
typedef std::vector<Json> array;
static void parse_from_stdin() {
// Initialize strings, ints etc
string buf;
string line;
int i = 0;

//double val = 0;
double val_annot = 0;
//double outval = 0;
int j = 0;
long long int ii; //gpu thread index (to be used later)
string err,mystr;
std::string mystrings;
std::string teststring;
std::string annotations;

while (std::getline(std::cin, line)) {
    buf += line + "\n";
}
// Pass data from json file
auto json = Json::parse(buf, err);

/*
Input Json Array Index :
    0 : Annotations
        1 : Camera Points (x)
        2 : Camera Points (z)
    3 : Data Points (x)
    4 : Data Points (y)
    5 : Data Points (z)
    6 : Camera Parameters
*/
/* Store JSON data into arrays
    annot : Double array of annotations
    datax : Double array of data points (x)
    datay : Double array of data points (y)
    dataz : Double array of data points (z)

    Note: Camera poisions - Fixed in y -direction at -6

    camx : Camera positions (x)
    camz : Camera poisitons (z)
*/
// Store annotations to annot
double annot[json[0].array_items().size()];
for(auto &value :json[0].array_items()){

    annotations = value.dump();
        std::string::size_type sz;
        val_annot = stod(annotations,&sz);
        annot[j] = val_annot;
    j = j + 1;
}
// Store data points (x) to datax
double datax[json[3].array_items().size()];
int kx = 0;
for(auto &valuex :json[3].array_items()){
```

```
    std::string dataxs;
    double valx;
        dataxs = valuex.dump();
        std::string::size_type sz;
        valx = stod(dataxs,&sz);
        datax[kx] = valx;
        kx = kx + 1;
}

// Store data points (y) to datay
double datay[json[4].array_items().size()];
int ky = 0;
for(auto &valuey :json[4].array_items()){
        std::string datays;
        double valy;
        datays = valuey.dump();
        std::string::size_type sz;
        valy = stod(datays,&sz);
        datay[ky] = valy;
        ky = ky + 1;
}
// Store data points (z) to dataz
double dataz[json[5].array_items().size()];
int kz = 0;
for(auto &valuez :json[5].array_items()){
    std::string datazs;
    double valz;
    datazs = valuez.dump();
    std::string::size_type sz;
    valz = stod(datazs,&sz);
    dataz[kz] = valz;
    kz = kz + 1;
}

// Store camera positions (x) to camposx
double camposx[json[1].array_items().size()];
int kcx = 0;
for(auto &value_cx :json[1].array_items()){
        std::string camposxs;
        double camx;
        camposxs = value_cx.dump();
        std::string::size_type sz;
        camx = stod(camposxs,&sz);
        camposx[kcx] = camx;
        kcx = kcx + 1;
}
// Store camera positions (z) to camposz
double camposz[json[2].array_items().size()];
int kcz = 0;
for(auto &value_cz :json[2].array_items()){
        std::string camposzs;
        double camz;
        camposzs = value_cz.dump();
        std::string::size_type sz;
        camz = stod(camposzs,&sz);
        camposz[kcz] = camz;
        kcz = kcz + 1;
}

// Determine number of obstacle points
int cot0 = 0;
```

```
for(int k0 = 0 ; k0 < json[0].array_items().size() ; k0++){
        if(annot[k0] != 0){
                cot0 = cot0 + 1;
        }
}



// Determine all obstacle coordinates and collect into double arrays
int coto = 0;
double obstx[cot0];
double obsty[cot0];
double obstz[cot0];

for(int ko = 0 ; ko < json[0].array_items().size() ; ko++){
    if(annot[ko] != 0){
        double obstxx = datax[coto];
        double obstyy = datay[coto];
        double obstzz = dataz[coto];

        obstx[coto] = obstxx;
        obsty[coto] = obstyy;
        obstz[coto] = obstzz;
        coto = coto + 1;
    }
}

// Test sensor placement
// Initialize camera parameters
// ------------------------------------------
// The following needs to be user specified :
// numcams : Number of cameras
// numpans : Number of pan options
// panarr : Array of pan options
// -------------------------------------------
int numcams = 4;
// Pan and tilt angles should be defined in [-pi,pi]
const double pi = 3.1415926535897;
double fov = 60*(pi/180);    // Half of the field of view
double tilt = 0;             // Tilt (Rot Z of the camera)
double range = 8;            // The range of the camera
double x,y,z;

double myxval,myzval;
// Initialize loop help variables
double max = 0;
int num = 0;
bool b;


// Initialize arrays and loop help variables
double pan;
int numpan = 3;
double mypval[numcams];
double camxpos[numcams];
double camzpos[numcams];
int coverage[kz];
int final_coverage[kz];
int ff_coverage[kz];
double panarr[numpan] = {0,pi/2,-3*pi/4};

```

```


// Loop through number of cameras
for(int camcount = 0 ; camcount < numcams ; camcount++){
    if(camcount > 0){
        for(int covcount = 0 ; covcount < ky ; covcount++){
            if(final_coverage[covcount] == 1){
                ff_coverage[covcount] = 1;
            }
        }
    }
    max = 0; // Reset max variable for current camer
    for(int j_k = 0 ; j_k < kcz; j_k++) {

        // Set current camera 1 position
        x = camposx[j_k];
        y = -6;
        z = camposz[j_k];
        // Initialize sum variable

        for(int pancount = 0 ; pancount < numpan ; pancount++){

            int sum = 0;    // Reset sum variable for current pan angle
            pan = panarr[pancount]; // Set pan angle

            for(int i = 0; i < ky ; i++){

                // Only compute visibility and coverage is point is not yet covered
                if(ff_coverage[i] != 1){

                    // Set current data point set
                    double xp = datax[i];
                    double yp = dataz[i];
                    double zp = datay[i];
                    // Determine distance from sensor to data point
                    double L = std::sqrt(std::pow(xp-x,2) + std::pow(zp-z,2) + std::pow(yp-y,2));
                    // Determine angle between sensor and data point (XY plane)
                    double xya = std::atan2((zp-z),(xp-x));
                    // Determine angle between sensor and data point (XZ plane)
                    double xza = std::atan2((yp-y),L);
                    // Determine if point is seen by camera
                    if(L < range){
                        if( (pan - fov) <= xya && xya <= (pan+ fov) ){
                            if( ((tilt - fov) <= xza) && (xza <= (tilt + fov)) ){
                                // Determine visibility
                                for(int vk = 0 ;vk <  coto ; vk++){
                                    double xobs = obstx[vk];
                                    double yobs = obsty[vk];
                                    double zobs = obstz[vk];
                                    double o_L = std::sqrt(std::pow(xobs-x,2) + std::pow(zobs-z,2)
                                    + std::pow(yobs-y,2));
                                    double o_xya = std::atan2((zobs-z),(xobs-x));
                                    double thr = 0.25;
                                    double o_xza = std::atan2((yobs-y),o_L);

                                    // Is obstacle point within sensing range
                                    b=1;
                                    if((pan - fov) <= o_xya && o_xya <= (pan + fov) ){
                                        if(std::abs(xya - o_xya) <  thr){
                                            if( ((tilt - fov) <= o_xza) && (o_xza<=(tilt + fov))){
                                                if(std::abs(xza - o_xza) < thr){
```

```
                                                    if(L > o_L){
                                                        b = 0;
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                            else{
                            b = 0;
                            }
                        }
                        else{
                            b = 0;
                        }
                    }
                    else{
                        b = 0;
                    }
                    if(b==1){
                        coverage[i] = 1;
                    }
                    else{
                        coverage[i] = 0;
                    }
                    // Sum for the current camera position increases by one
                    // if the above conditions are fulfilled
                    sum = sum +  b;
                }
            }

            // If current pan config is better than previously for the current camera
            // Save pan config as best
            if(sum > max){
                max = sum;
                num = j_k;

                // Loop through coverage array
                for(int covi = 0 ; covi < ky ; covi++){
                    final_coverage[covi] = coverage[covi];
                }
                myxval = camposx[num];
                myzval = camposz[num];
                mypval[camcount] = pan;
                camzpos[camcount] = myzval;
                camxpos[camcount] = myxval;
            }

        }
    }
}

// Output results
for(int ijj = 0; ijj < numcams ; ijj++){
    double myvalue = camxpos[ijj];
    double myzvalue = camzpos[ijj];
    double mypanvalue = mypval[ijj];
    std::cout << std::endl;
    std::cout << "Camera x : " << myvalue <<   std::endl;
    std::cout << "Camera z : " << myzvalue << std::endl;
```

```
    std::cout << "Pan : " << mypanvalue << std::endl;
}
double fsum = 0;
for(int ickk = 0; ickk < ky ; ickk++){
    if(ff_coverage[ickk] == 1){
    fsum = fsum + 1;
    }else if(final_coverage[ickk] == 1){
    fsum = fsum + 1;
    }
}
double fperc = (fsum/ky) * 100;

std::cout << "Percentage coverage " << fperc  <<  std::endl;
}
// Main execution loop
int main(int argc, char **argv) {
    if (argc == 2 && argv[1] == string("--stdin")) {
        parse_from_stdin();
        return 0;
    }
}
```

A.8.3 Combinatorial Brute Force Algorithm

```
// C++ Code for combinatorial placement of n pre-defined cameras for a given
// JSON file with visibility model with variable pan angle
// Author : Vegard Tveit
// Date : 27.03.2018
// Comment : This code uses the json11 library found at :
// https://github.com/dropbox/json11

// User need to define:
// "numcams" in line 158,
// "numpans" in line 162
// "panarray" in line 167

// Initial Setup
#include <iostream>
#include "json11.hpp"
#include <string>
#include <fstream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <chrono>
#include <numeric>
#include <functional>

// Typedefs and namespace
using namespace json11;
using std::string;
typedef std::vector<Json> array;
typedef std::vector< std::vector<int> > matrix_int;
typedef std::vector<int> array_int;
typedef std::vector<bool> array_bool;
typedef std::vector< std::vector<bool> > matrix_bool;
typedef std::vector< std::vector<double> > matrix_double;
typedef std::chrono::high_resolution_clock Clock;

```

```
// Initialize function callers
array_int evalu(int lpans, int valueo);
long long int nchoosek(int N, int K);
matrix_int comb(int  N, int  K);

// Function to run in main loop
static void parse_from_stdin() {
// Initialize strings, ints etc
string buf;
string line;
int i = 0;
double val_annot = 0;
int j = 0;
string err,mystr;
std::string mystrings;
std::string teststring;
std::string annotations;

while (std::getline(std::cin, line)) {
    buf += line + "\n";
}
// Pass data from json file
auto json = Json::parse(buf, err);

// Store annotations to annot
double annot[json[0].array_items().size()];
for(auto &value :json[0].array_items()){

    annotations = value.dump();
    std::string::size_type sz;
    val_annot = stod(annotations,&sz);
    annot[j] = val_annot;
    j = j + 1;
}
// Store data points (x) to datax
double datax[json[3].array_items().size()];
int kx = 0;
for(auto &valuex :json[3].array_items()){
    std::string dataxs;
    double valx;
    dataxs = valuex.dump();
    std::string::size_type sz;
    valx = stod(dataxs,&sz);
    datax[kx] = valx;
    kx = kx + 1;
}

// Store data points (y) to datay
double datay[json[4].array_items().size()];
int ky = 0;
for(auto &valuey :json[4].array_items()){
    std::string datays;
    double valy;
    datays = valuey.dump();
    std::string::size_type sz;
    valy = stod(datays,&sz);
    datay[ky] = valy;
    ky = ky + 1;
}
// Store data points (z) to dataz
double dataz[json[5].array_items().size()];
```

```
int kz = 0;
for(auto &valuez :json[5].array_items()){
    std::string datazs;
    double valz;
    datazs = valuez.dump();
    std::string::size_type sz;
    valz = stod(datazs,&sz);
    dataz[kz] = valz;
    kz = kz + 1;
}

// Store camera positions (x) to camposx
double camposx[json[1].array_items().size()];
int kcx = 0;
for(auto &value_cx :json[1].array_items()){
    std::string camposxs;
    double camx;
    camposxs = value_cx.dump();
    std::string::size_type sz;
    camx = stod(camposxs,&sz);
    camposx[kcx] = camx;
    kcx = kcx + 1;
}
// Store camera positions (z) to camposz
double camposz[json[2].array_items().size()];
int kcz = 0;
for(auto &value_cz :json[2].array_items()){
    std::string camposzs;
    double camz;
    camposzs = value_cz.dump();
    std::string::size_type sz;
    camz = stod(camposzs,&sz);
    camposz[kcz] = camz;
    kcz = kcz + 1;
}

// Determine number of obstacle points
int cot0 = 0;
for(int k0 = 0 ; k0 < json[0].array_items().size() ; k0++){
        if(annot[k0] != 0){
            cot0 = cot0 + 1;
        }
}

// Determine all obstacle coordinates and collect into double arrays
int coto = 0;
double obstx[cot0];
double obsty[cot0];
double obstz[cot0];

for(int ko = 0 ; ko < json[0].array_items().size() ; ko++){
    if(annot[ko] != 0){
        double obstxx = datax[coto];
        double obstyy = datay[coto];
        double obstzz = dataz[coto];

        obstx[coto] = obstxx;
        obsty[coto] = obstyy;
        obstz[coto] = obstzz;
        coto = coto + 1;
    }
```

```
}

// Initialize constants
const double pi = 3.1415926535897;
const int numcams = 2;
const int numpan = 3;
double pan;
bool b;

// USER DEFINED array of possible pan angles
const double  panarray[numpan] = {0,pi/2,-3*pi/4};
// Pan and tilt angles should be defined in [-pi,pi]

// Sensor parameters
double fov = 60*(pi/180);        // Half of the field of view
double tilt = 0;             // Tilt (Rot Z of the camera)
double range = 8;            // The range of the camera
double x,y,z;

// Determine coverage for all positions and all poses
int combco = numpan*kcz;
auto t1 = Clock::now();
matrix_bool outmat(combco,std::vector<bool>(ky));
int iteration = 0;

for(int j_k = 0; j_k < kcz ; j_k++){

    // Set current camera 1 position
    x = camposx[j_k];
    y = -6;
    z = camposz[j_k];

    for(int pancount = 0 ; pancount < numpan ; pancount++){

        pan = panarray[pancount]; // Set pan angle

        for(int i = 0; i < ky ; i++){

            // Set current data point set
            double xp = datax[i];
            double yp = dataz[i];
            double zp = datay[i];
            // Determine distance from sensor to data point
            double L = std::sqrt(std::pow(xp-x,2) + std::pow(zp-z,2) + std::pow(yp-y,2));
            // Determine angle between sensor and data point (XY plane)
            double xya = std::atan2((zp-z),(xp-x));
            // Determine angle between sensor and data point (XZ plane)
            double xza = std::atan2((yp-y),L);
            // Determine if point is seen by camera
            if(L < range){
                if( (pan - fov) <= xya && xya <= (pan+ fov) ){
                    if( ((tilt - fov) <= xza) && (xza <= (tilt + fov)) ){
                        // Determine visibility
                        for(int vk = 0 ;vk <  coto ; vk++){
                            double xobs = obstx[vk];
                            double yobs = obsty[vk];
                            double zobs = obstz[vk];
                            double o_L = std::sqrt(std::pow(xobs-x,2) + std::pow(zobs-z,2)
                            + std::pow(yobs-y,2));
                            double o_xya = std::atan2((zobs-z),(xobs-x));
                            double thr = 0.25;
```

```
                            double o_xza = std::atan2((yobs-y),o_L);

                            // Is obstacle point within sensing range
                            b=true;
                            if((pan - fov) <= o_xya && o_xya <= (pan + fov) ){
                                if(std::abs(xya - o_xya) <  thr){
                                    if( ((tilt - fov) <= o_xza) && (o_xza <= (tilt + fov)) ){
                                        if(std::abs(xza - o_xza) < thr){
                                            if(L > o_L){
                                                b = false;
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                    else{
                    b = false;
                    }
                }
                else{
                    b = false;
                }
            }
            else{
                b = false;
            }

        outmat[iteration][i] = b;
        }
        iteration = iteration + 1;
    }
}

// Determine number of combinations
long int comnum = nchoosek(combco,numcams);
// Array to store combinations
matrix_int combarr(comnum , std::vector<int>(numcams));
combarr = comb((int)combco,(int)numcams);
int sumof;

// Print the number of combinations and number of cameras
std::cout << "No. of combinations: " << comnum << " " << "No. of cameras: "
<< numcams << std::endl;
array_bool testbool(ky);
array_bool bout(ky);
auto tt = Clock::now();
array_int sum(comnum);
for(int m = 0; m < comnum ; m++){

    std::fill(bout.begin(),bout.end(),false);
    for(int n = 0; n < numcams ; n++){
        int ind = combarr[m][n];
        // bout = bout | helparr
        std::transform(bout.begin(),bout.end(),outmat[ind].begin(),bout.begin(),std::plus<bool>());

    }
    sum[m] = std::accumulate(bout.begin(),bout.end(),0);

}
```

```

auto ttt = Clock::now();
auto t2 = Clock::now();
int max = 0;

int ind;
for(int j = 0; j < comnum ; j++)
{
    if (sum[j]> max)
    {
        max = sum[j];
        ind = j;
    }
}

std::cout << "Elapsed Time: "
<< std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count()
<< " milliseconds" << std::endl;

std::cout << "Max Covered : " << max << " at index " << ind << std::endl;

// Post process
array_int camout(numcams);
for(int i = 0 ; i < numcams ; i++)
{
    camout[i] = combarr[ind][i];
}

matrix_int indarr(numcams,std::vector<int>(2));

for(int i = 0; i < numcams ; i++)
{
    array_int arrhelp(numcams);
    arrhelp = evalu(numpan, camout[i]);
    indarr[i][0] = arrhelp[0];
    indarr[i][1] = arrhelp[1];

    double cameraxpar = camposx[indarr[i][1]];
    double cameraypar = camposz[indarr[i][1]];
    double camerapan = panarray[indarr[i][0]];
    std::cout << std::endl << std::endl;
    std::cout << "Camera x coordinate : " << cameraxpar << std::endl;
    std::cout << "Camera y coordinate : " << cameraypar << std::endl;
    std::cout << "Pan angle : " << camerapan << std::endl;
}
}

// Main execution loop
int main(int argc, char **argv) {
    if (argc == 2 && argv[1] == string("--stdin")) {
        parse_from_stdin();
        return 0;
    }
}
//----------------------------------------------------------------------------------------------------
// FUNCTIONS

// Function for determining matrix of all combinations
matrix_int comb(int  N, int  K)
{
    long long int fi = nchoosek(N,K);
```

```
    matrix_int  out((int)fi + 1,std::vector<int>(K));
    int c2;
    std::string bitmask(K, 1); // K leading 1's
    bitmask.resize(N, 0); // N-K trailing 0's
   // int testi = 1;
    int count = 0;
    // print integers and permute bitmask
    do {
        c2 = 0;
        for (int i = 0; i < N; ++i) // [0..N-1] integers
        {
            if (bitmask[i]){
         out[count][c2] = i;
         c2 = c2 + 1;
        }
        }
    count = count + 1;
    } while (std::prev_permutation(bitmask.begin(), bitmask.end()));
    return out;
}

// Function for determining the binomial coefficient
long long int nchoosek(int N, int K)
{
std::string bitmask(K, 1); // K leading 1's
    bitmask.resize(N, 0); // N-K trailing 0's
    long long int counter = 0;
    do {
        counter = counter + 1;
    } while (std::prev_permutation(bitmask.begin(), bitmask.end()));

    return counter;
}
// Function for evaluating the camera and pan indexes
array_int evalu(int lpans, int valueo)
{
    array_int outarray(2);              //Initialize array
    double ic = (double)valueo;
    double pc = floor(ic/lpans);        // Determine the current camera index
    int panx = valueo - (int)pc*lpans;  // Determine the current pan angle index
    int camx = (int)pc;

    // Return results
    outarray[0] = panx;
    outarray[1] = camx;
    return outarray;
}
```

A.9 CUDA C++ Scripts

A.9.1 C++ Program for Importing JSON and Exporting Text File Matrices for CUDA

```cpp
// C++ Code for reading a JSON file and converting to txt files
// for CUDA usage
// Author : Vegard Tveit
// Date : 10.04.2018
// Comment : This code uses the json11 library found at :
// https://github.com/dropbox/json11
// Initial Setup
#include <iostream>
#include "json11.hpp"
#include <string>
#include <fstream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <chrono>
#include <numeric>
#include <functional>
#include <fstream>


// Typedefs and namespace
using namespace json11;
using std::string;
typedef std::vector<Json> array;
typedef std::vector< std::vector<int> > matrix_int;
typedef std::vector<int> array_int;
typedef std::vector<bool> array_bool;
typedef std::vector< std::vector<bool> > matrix_bool;
typedef std::vector< std::vector<double> > matrix_double;
typedef std::chrono::high_resolution_clock Clock;

// Initialize function callers
array_int evalu(int lpans, int valueo);
//double factorial(double n);
long long int nchoosek(int N, int K);
matrix_int comb(int  N, int  K);

// Function to run in main loop
static void parse_from_stdin() {
// Initialize strings, ints etc
string buf;
string line;
int i = 0;
double val_annot = 0;
int j = 0;
long long int ii; //gpu thread index (to be used later)
string err,mystr;
std::string mystrings;
std::string teststring;
std::string annotations;

while (std::getline(std::cin, line)) {
    buf += line + "\n";
```

```
}
// Pass data from json file
auto json = Json::parse(buf, err);

/*
Input Json Array Index :
    0 : Annotations
        1 : Camera Points (x)
        2 : Camera Points (z)
    3 : Data Points (x)
    4 : Data Points (y)
    5 : Data Points (z)
    6 : Camera Parameters
*/
/* Store JSON data into arrays
    annot : Double array of annotations
    datax : Double array of data points (x)
    datay : Double array of data points (y)
    dataz : Double array of data points (z)

    Note: Camera poisions - Fixed in y -direction at -6

    camx : Camera positions (x)
    camz : Camera poisitons (z)
*/
// Store annotations to annot
double annot[json[0].array_items().size()];
for(auto &value :json[0].array_items()){

    annotations = value.dump();
        std::string::size_type sz;
        val_annot = stod(annotations,&sz);
        annot[j] = val_annot;
    j = j + 1;
}
// Store data points (x) to datax
double datax[json[3].array_items().size()];
int kx = 0;
for(auto &valuex :json[3].array_items()){
    std::string dataxs;
    double valx;
        dataxs = valuex.dump();
        std::string::size_type sz;
        valx = stod(dataxs,&sz);
        datax[kx] = valx;
        kx = kx + 1;
}

// Store data points (y) to datay
double datay[json[4].array_items().size()];
int ky = 0;
for(auto &valuey :json[4].array_items()){
        std::string datays;
        double valy;
        datays = valuey.dump();
        std::string::size_type sz;
        valy = stod(datays,&sz);
        datay[ky] = valy;
        ky = ky + 1;
}
// Store data points (z) to dataz
```

```
double dataz[json[5].array_items().size()];
int kz = 0;
for(auto &valuez :json[5].array_items()){
    std::string datazs;
    double valz;
    datazs = valuez.dump();
    std::string::size_type sz;
    valz = stod(datazs,&sz);
    dataz[kz] = valz;
    kz = kz + 1;
}

// Store camera positions (x) to camposx
double camposx[json[1].array_items().size()];
int kcx = 0;
for(auto &value_cx :json[1].array_items()){
        std::string camposxs;
        double camx;
        camposxs = value_cx.dump();
        std::string::size_type sz;
        camx = stod(camposxs,&sz);
        camposx[kcx] = camx;
        kcx = kcx + 1;
}
// Store camera positions (z) to camposz
double camposz[json[2].array_items().size()];
int kcz = 0;
for(auto &value_cz :json[2].array_items()){
        std::string camposzs;
        double camz;
        camposzs = value_cz.dump();
        std::string::size_type sz;
        camz = stod(camposzs,&sz);
        camposz[kcz] = camz;
        kcz = kcz + 1;
}

// Determine number of obstacle points
int cot0 = 0;
for(int k0 = 0 ; k0 < json[0].array_items().size() ; k0++){
        if(annot[k0] != 0){
                cot0 = cot0 + 1;
        }
}

// Determine all obstacle coordinates and collect into double arrays
int coto = 0;
double obstx[cot0];
double obsty[cot0];
double obstz[cot0];

for(int ko = 0 ; ko < json[0].array_items().size() ; ko++){
    if(annot[ko] != 0){
        double obstxx = datax[coto];
        double obstyy = datay[coto];
        double obstzz = dataz[coto];

        obstx[coto] = obstxx;
        obsty[coto] = obstyy;
        obstz[coto] = obstzz;
        coto = coto + 1;
```

```
    }
}

// Initialize constants
const double pi = 3.1415926535897;
const int numcams = 2;
const int numpan = 3;
double pan;
bool b;



// USER DEFINED array of possible pan angles
const double  panarray[numpan] = {0,pi/2,-3*pi/4};
// Pan and tilt angles should be defined in [-pi,pi]


// Sensor parameters
double fov = 60*(pi/180);       // Half of the field of view
double tilt = 0;             // Tilt (Rot Z of the camera)
double range = 8;            // The range of the camera
double x,y,z;


// Determine coverage for all positions and all poses
int combco = numpan*kcz;
auto t1 = Clock::now();
matrix_bool outmat(combco,std::vector<bool>(ky));
int iteration = 0;
for(int j_k = 0; j_k < kcz ; j_k++){

    // Set current camera 1 position
    x = camposx[j_k];
    y = -6;
    z = camposz[j_k];

    for(int pancount = 0 ; pancount < numpan ; pancount++){

        pan = panarray[pancount]; // Set pan angle

        for(int i = 0; i < ky ; i++){

            // Set current data point set
            double xp = datax[i];
            double yp = dataz[i];
            double zp = datay[i];
            // Determine distance from sensor to data point
            double L = std::sqrt(std::pow(xp-x,2) + std::pow(zp-z,2) +
            std::pow(yp-y,2));
            // Determine angle between sensor and data point (XY plane)
            double xya = std::atan2((zp-z),(xp-x));
            // Determine angle between sensor and data point (XZ plane)
            double xza = std::atan2((yp-y),L);
            // Determine if point is seen by camera
            if(L < range){
                if( (pan - fov) <= xya && xya <= (pan+ fov) ){
                    if( ((tilt - fov) <= xza) && (xza <= (tilt + fov)) ){
                        // Determine visibility
                        for(int vk = 0 ;vk <  coto ; vk++){
                            double xobs = obstx[vk];
                            double yobs = obsty[vk];
```

```
                                double zobs = obstz[vk];
                                double o_L = std::sqrt(std::pow(xobs-x,2) + std::pow(zobs-z,2)
                                + std::pow(yobs-y,2));
                                double o_xya = std::atan2((zobs-z),(xobs-x));
                                double thr = 0.25;
                                double o_xza = std::atan2((yobs-y),o_L);

                                // Is obstacle point within sensing range
                                b=true;
                                if((pan - fov) <= o_xya && o_xya <= (pan + fov) ){
                                    if(std::abs(xya - o_xya) <  thr){
                                        if( ((tilt - fov) <= o_xza) && (o_xza <= (tilt + fov)) ){
                                            if(std::abs(xza - o_xza) < thr){
                                                if(L > o_L){
                                                    b = false;
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                        else{
                        b = false;
                        }
                    }
                    else{
                        b = false;
                    }
                }
                else{
                    b = false;
                }

        outmat[iteration][i] = b;
        }
        iteration = iteration + 1;
    }
}

    // Determine number of combinations
    long int comnum = nchoosek(combco,numcams);
    // Array to store combinations
    matrix_int combarr(comnum , std::vector<int>(numcams));
    combarr = comb((int)combco,(int)numcams);
    std::cout << combarr[1][1] << std::endl;

    // Write all binary subsets to "subsets.txt"
    std::ofstream subsets;
    subsets.open("Subsets_1.txt");
    for(int i = 0; i < iteration; i++){
        for(int j = 0; j < ky ; j++){
            subsets << outmat[i][j] << " ";
        }
        subsets << "\n";
    }

    // Write all combinations to "combinations.txt"
    std::ofstream myfile;
    myfile.open("Combinations_2.txt");
    for(int j = 0; j < comnum ; j++){
```

```
        for(int i = 0; i < numcams ; i++){

            myfile << combarr[j][i] << " ";
        }
        myfile << "\n";
    }
    myfile.close();
}

// Main execution loop
int main(int argc, char **argv) {
    if (argc == 2 && argv[1] == string("--stdin")) {
        parse_from_stdin();
        return 0;
    }
}
//----------------------------------------------------------------------------------------------------
// FUNCTIONS

// Function for determining matrix of all combinations
matrix_int comb(int  N, int  K)
{
    long long int fi = nchoosek(N,K);
    matrix_int  out((int)fi + 1,std::vector<int>(K));
    int c2;
    std::string bitmask(K, 1); // K leading 1's
    bitmask.resize(N, 0); // N-K trailing 0's
   // int testi = 1;
    int count = 0;
    // print integers and permute bitmask
    do {
        c2 = 0;
        for (int i = 0; i < N; ++i) // [0..N-1] integers
        {
            if (bitmask[i]){
         out[count][c2] = i;
         c2 = c2 + 1;
        }
        }
    count = count + 1;
    } while (std::prev_permutation(bitmask.begin(), bitmask.end()));
    return out;
}

// Function for determining the binomial coefficient
long long int nchoosek(int N, int K)
{
std::string bitmask(K, 1); // K leading 1's
    bitmask.resize(N, 0); // N-K trailing 0's
    long long int counter = 0;
    do {
        counter = counter + 1;
    } while (std::prev_permutation(bitmask.begin(), bitmask.end()));

    return counter;
}
// Function for evaluating the camera and pan indexes
array_int evalu(int lpans, int valueo)
{
    array_int outarray(2);               //Initialize array
    double ic = (double)valueo;
```

```
    double pc = floor(ic/lpans);        // Determine the current camera index
    int panx = valueo - (int)pc*lpans;  // Determine the current pan angle index
    int camx = (int)pc;

    // Return results
    outarray[0] = panx;
    outarray[1] = camx;
    return outarray;
}
```

A.9.2 CUDA Program for the Brute Force Algorithm

```
/*
CUDA code for GPU optimization of camera placement problem
Author : Vegard Tveit
Date : 17.04.2018
Comment : The user has to specify:

    - Number of sensors to be placed
    - Number of possible combinations(nchoosek)
    - Modify UNISIZE
    - Number of datapoints
    - Number of possible placement points
    - Number of possible pan angles
    - "subsets.txt" and "combinations.txt"

*/
// Initial Setup
#include <iostream>
#include <string>
#include <fstream>
#include <vector>

#include <new>
#define UNISIZE 1490
#include <cmath>
#include <algorithm>
#include <numeric>
#include <functional>
#include <fstream>
__global__ void mykernel(int* devarr, bool* subs, int* sum, unsigned long len,
 unsigned long nsubs, unsigned long usize)
{
    // Kernel function to run on GPU
    // Defining variables (stored in each kernel)
    // The id of the current thread
    unsigned long th_id = blockIdx.x * blockDim.x + threadIdx.x;
    bool barr[1490] = {0}; //Array for storing coverage
    int totsum = 0; // Sum of covered points
    if(th_id < len){
        for(unsigned long i = 0; i < nsubs; i++)
        {
            int ind = devarr[th_id*nsubs + i];
            for(unsigned long j = 0; j < usize; j++)
            {
                // Only do calculations if point is uncovered by current combination

                if(barr[j] == 0)
                {
```

```

                    if(subs[ind*usize + j] == 1){
                        barr[j] = 1;
                        totsum +=1;
                    }


                }
            }
        }
        sum[th_id] = totsum;
    }else sum[th_id] = 0;
}

void readfromtxt(){

    unsigned long num_sensors = 3;
    unsigned long ncombs = 1521520;

    unsigned long ndp = 1490;
    unsigned long campos = 70;
    unsigned long numpans = 3;

    // Dynamically allocate arrays
    int* array = (int*)malloc(ncombs*num_sensors*sizeof(int));
    bool* subs_array = (bool*)malloc(ndp*campos*numpans*sizeof(bool*));

    //Load subsets from txt file and store in 1D array
    std::ifstream subsfile("Subsets.txt");
    bool b;
    unsigned long col_s = 0;
    while (subsfile >> b)
    {
        subs_array[col_s] = b;
        col_s +=1;
    }

    std::cout << " " <<std::endl;
    std::cout << " " <<std::endl;
    std::cout << " " <<std::endl;
    // Store combinations array in a 1D array
    std::ifstream myfile("Combinations_1.txt");
    int a;
    unsigned long col = 0;

    while (myfile >> a)
    {
        array[col] = a;
        col += 1;
    }
    std::cout << "Col subs : " << col_s << " and col arr: " << col << std::endl;
    //GPU variables
    unsigned long n_threads_per_block = 1024; //Threads per block
    unsigned long n_blocks = (ncombs + n_threads_per_block - 1)/n_threads_per_block;

    std::cout << "Number of blocks :" << n_blocks << std::endl;
    unsigned long data_n = n_blocks*n_threads_per_block; // Total number of available threads

    //Vectorize array for GPU calculations
    unsigned long chop_combs;
    //unsigned long ncrit = data_n;
```

```
    chop_combs = ncombs;

    std::cout << "No. of available threads: " <<  data_n << std::endl;
    std::cout << "Number of used threads : " << chop_combs << std::endl;
    size_t i_datasize = chop_combs*sizeof(int);
    std::cout << "i_datasize [bytes] : " << i_datasize <<  std::endl;
    // Allocate CPU Memory
    int* sum_host = new int[chop_combs];
    //std::cout << "bool array datasize [bytes] : " << b_datasize << std::endl;
    size_t array_datas = chop_combs*num_sensors*sizeof(int);
    size_t bool_subs_size = ndp*numpans*campos*sizeof(bool);

    std:: cout << "Array size : " << array_datas <<
    " and subs size " << bool_subs_size << std::endl;

    // Allocate GPU Memory
    bool* subs_dev;
    int* sum_dev;
    int* array_dev;
    cudaMalloc(&subs_dev,bool_subs_size);
    cudaMalloc(&array_dev, array_datas);
    cudaMalloc(&sum_dev,i_datasize);

    // Copy host (CPU) arrays to device (GPU) arrays
    cudaMemcpy(subs_dev, subs_array, bool_subs_size, cudaMemcpyHostToDevice);
    cudaMemcpy(sum_dev, sum_host, i_datasize, cudaMemcpyHostToDevice);
    cudaMemcpy(array_dev, array, array_datas, cudaMemcpyHostToDevice);

    // Run "mykernel" function on GPU threads with gpu timing
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    mykernel <<< n_blocks,n_threads_per_block >>>
    (array_dev,subs_dev,sum_dev,chop_combs,num_sensors,ndp);

    cudaDeviceSynchronize();
    cudaEventRecord(stop);

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("The elapsed time for kernel execution was %.2f ms\n", milliseconds);
    // Copy results back to cpu memory
    cudaMemcpy(sum_host, sum_dev, i_datasize, cudaMemcpyDeviceToHost);

    // Post process
    int max = 0;
    unsigned long  ind = 0;
    for (unsigned long  i = 0; i < chop_combs ; i++){
        if(sum_host[i] > max){
            max = sum_host[i];
            ind = i;
        }
    }

    printf("Highest coverage value: %i, at index %lu. \n",max,ind);
    std::cout << "The index represents camera index: ";
    for(int m = 0; m < num_sensors ; m++){
```

```

        printf("%i ", array[ind*num_sensors + m]);
    }
    std::cout << std::endl;
    //Free allocated memory on CPU and GPU
    cudaFree(subs_dev);
    cudaFree(sum_dev);
    cudaFree(array_dev);
    delete[] sum_host;
    free(array);
    free(subs_array);
}
```

A.10 Scripts for the Final Test

A.10.1 Matlab Program for Genetic Algorithm with K-Cover

```matlab
%% Genetic Algorithm with K-Coverage
% Author: Vegard Tveit
% Date: 22.05.2018
clear;
clc;
close all;

% Specify the input *.mat file from the UI to set up the environment
env = environment_generate('Final_0705.mat');
datax = env.datax;                          % Discrete data points (x)
datay = env.datay;                          % Discrete data points (y)
dataz = env.dataz;                          % Discrete data points (z)
campx = env.campx;                          % Discrete placement points (x)
campy = env.campy;                          % Discrete placement points (y)
campz = env.campz;                          % Discrete placement points (z)
obsta = env.obsta;
annot = env.annot;
% Define number of pan angles
pans = [pi/4,pi/2,-3*pi/4,0];

% Preallocate variables and initialize
ldata = length(datax);                      % Number of data points
lpans = length(pans);                       % Number of pan angles
lcamp = length(campx);                      % Number of placement points

x = zeros(lcamp,1);                         % Placement point array (x)
y = zeros(lcamp,1);                         % Placement point array (y)
z = zeros(lcamp,1);                         % Placement point array (x)
b = false(lpans,ldata);                     % Coverage matrix - Combs
iter = 0;

% Indexation counter
c = 0;
for i = 1:ldata
    if(annot(i) == 2)
        init_cov(i) = 2;
        c = c + 1;
    else
        init_cov(i) = 1;
    end
end

% Compute coverage for all possible camera poses and positions
for i = 1:lcamp
    x(i) = campx(i);                        % Get camera position (x)
    y(i) = campy(i);                        % Get camera position (y)
    z(i) = campz(i);                        % Get camera position (z)

    % Loop through all pan angles
    for k = 1:lpans
        pan = pans(k);                      % Get pan angle
        iter = iter + 1;                    % Update indexation

        % Compute coverage of all data points
```

```
        for n = 1:ldata
            b(iter,n) = covered1(n,x(i),y(i),z(i),datax,datay,dataz,...
                                obsta,pan);
        end
    end
end

% Initalize pool of chromosomes randomly
numchromo = 3000;
numsubs = iter;
usize = ldata;
numsens = 3;
chromo = zeros(numchromo,numsens);
alpha = 4000;
for i = 1:numchromo
    for j = 1:numsens
        ind = randi(numsubs,1);
        chromo(i,j) = ind;
    end
end

b = b';
% Evaluate fitness of chromosomes
fitmat = zeros(numchromo,1);
cove = zeros(1,ldata);
cove(1:end) = init_cov(1:end);
cf = 0;
for m = 1:numchromo
    bch = false(usize,1);
    cove = zeros(1,ldata);
    cove(1:end) = init_cov(1:end);
    cf = 0;
    penalty = 0;
     barr = zeros(1,ldata);
    for n = 1:numsens
        inde = chromo(m,n);
        for i = 1:ldata
            if( b(i,inde))
                barr(i) = barr(i) + 1;
            end
        end
    end
    for j = 1:ldata
        if(barr(j) >= init_cov(j))
            bch(j) = true;
            if(annot(j) == 2)
                cf = cf + 1;
            end
        end
    end
    if cf > 0 && cf < c
        penalty = alpha*(c/cf);
    end
    if cf == 0
        penalty = alpha*c;
    end
    fitmat(m) = sum(bch) - penalty;
end
avgsum_init = sum(fitmat)/length(fitmat)
% Select best parents for next generation
[sortarr,indarr] = sort(fitmat,'descend');
```

```

% Make pool of parent solutions
% MUST BE AN EVEN NUMBER
numpar_gen = 3*numchromo/5;
max_generations = 300;
generations = 1;

childreninto = zeros(numpar_gen/4-1,numsens);

tic
% Main loop iterating through the generation
while(generations < max_generations)

% Probability of being chosen
for pk = 1:numpar_gen
    prob_par(pk,1) = fitmat(indarr(pk))/sum((fitmat(indarr(1:numpar_gen))));
end

% Select half of the pool for reproduction
repIter = 1;
bRep = true;
par_out = zeros(numpar_gen/2,1);
for i = 1:numpar_gen/2
    randRep = rand();
    bRep = true;
    repIter = 1;
    while(bRep)
        randRep = randRep - prob_par(repIter);
        if randRep < 0
            bRep = false;
            indeRep = repIter;
        end
        repIter = repIter + 1;
    end
    par_out(i) = indeRep;
end
% par_out now contains the indices of the individuals
% in the mating pool. These chormosomes should undergo
% crossover and mutation

par = indarr(1:numpar_gen,1);

children_out = zeros(numpar_gen/2-1,numsens);
for i = 1:numpar_gen/2-1
    par1 = par_out(i);
    par2 = par_out(i+1);

    % Mutation Rate
    p_m = 3/4;

    % Crossover function
    CroP1 = sort(chromo(par1,1:end));
    CroP2 = sort(chromo(par2,1:end));

    scP1 = fitmat(par1);
    scP2 = fitmat(par2);

    ProbP1 = scP1/(scP1+scP2);
    ProbP2 = 1-ProbP1;
    rCro = rand();
    for Ci = 1:numsens
```

```
        if(CroP1(Ci) == CroP2(Ci))
            ChiCro(Ci) = CroP1(Ci);
        end
        if(CroP1(Ci) ~= CroP2(Ci))
            if rCro <= ProbP1
                ChiCro(Ci) = CroP1(Ci);
            else
                ChiCro(Ci) = CroP2(Ci);
            end
        end
    end
    randMutProb = rand();
    if(randMutProb < p_m)

        bMut = true;
        arrMut = zeros(numsubs,1);

        rMut = randi(numsens,1);
        for Mi = 1:numsubs
            for Ni = 1:numsens
                if(ChiCro(Ni) == Mi)
                    arrMut(Mi) = 1;
                end
            end
        end
        hMut = find(arrMut == 0);
        riMut = randi(length(hMut),1);
        ChiCro(rMut) = hMut(riMut);
     end

children_out(i,1:numsens) = ChiCro;
end

% Select best half+1 of the children to bring into the population
fitchildren = zeros(numpar_gen/2-1,1);
for m = 1:numpar_gen/2-1
    bch = false(usize,1);
    for n = 1:numsens
        inde = children_out(m,n);
        bch(1:end,1) = bch(1:end,1) | b(1:end,inde);
    end
    fitchildren(m) = sum(bch);
end

[chilsorted,indchild] = sort(fitchildren,'descend');
children_out(indchild(1:6),1:end);
% Into population : childreninto ->
childreninto(1:numpar_gen/4-1,1:end) = children_out(indchild(1:numpar_gen/4-1,1:end),1:end);
% New population
for i = 1:length(childreninto)
   chromo(indarr(end-i),1:end) = childreninto(i,1:end);
end
chromo(450:end,1:end);
% Fitness of new population
fitmat = zeros(numchromo,1);
for m = 1:numchromo
    bch = false(usize,1);
    cove = zeros(1,ldata);
    cove(1:end) = init_cov(1:end);
    cf = 0;
    penalty = 0;
```

```
    barr = zeros(1,ldata);
    for n = 1:numsens
        inde = chromo(m,n);
        for i = 1:ldata
            if( b(i,inde))
                barr(i) = barr(i) + 1;
            end
        end
    end
    for j = 1:ldata
        if(barr(j) >= init_cov(j))
            bch(j) = true;
            if(annot(j) == 2)
                cf = cf + 1;
            end
        end
    end
    if cf > 0 && cf < c
        penalty = alpha*(c/cf);
    end
    if cf == 0
        penalty = alpha*c;
    end
    fitmat(m) = sum(bch) - penalty;
end

% Select best parents for next generation
[sortarr,indarr] = sort(fitmat,'descend');
best(generations) = sortarr(1);

generations = generations + 1;
end
toc
sortarr(1:10);
outVal = chromo(indarr(1),1:end);

% Post process
[panx,camx] = EvalNum(lpans,outVal);

% Display Results
disp('Camera Positions x : ')
campx(camx)

disp('Camera Positions z : ')
campz(camx)

disp('Pan Angles [rad] : ')
pans(panx)
```

A.10.2 Matlab Program for Generation of Combinations Matrix

```
%% Generate Combinations
clc; clear
% User Need to Specify
%    - k : Number of cameras
%    - n : Number of possible placement points
n = 249;
k = 5;
e = CombinaisonEnumerator(k, 249);

%% This part of the script needs to be executed several times
% Ctrl + Left Click for execution of this section only
clear A
limiter = 1;
maxlim = 1.5e8;
ncams = 5;
A = zeros(maxlim,ncams);
tic
while(e.MoveNext() && limiter < maxlim + 1)

    A(limiter,1:ncams) = e.Current + 45;
    limiter = limiter + 1;
end
toc
tic
mex_WriteMatrix('combtests_1.txt',A,'%f',' ', 'w+');
toc
A(end,:)
```

A.10.3 CUDA Program for Brute Force Algorithm with K-Cover

```
/*
CUDA code for GPU optimization of camera placement problem
With support of 2-coverage Region of Interest
Author : Vegard Tveit
Date : 17.04.2018
Comment : The user has to specify:

    - Number of sensors to be placed
    - Number of possible combinations(nchoosek)
    - Modify UNISIZE
    - Number of datapoints
    - Number of possible placement points
    - Number of possible pan angles
    - "subsets.txt", "annotations.txt" and "combinations.txt"

*/
// Initial Setup
#include <iostream>
#include <string>
#include <fstream>
#include <vector>

#include <new>
#define UNISIZE 9084
#include <cmath>
```

```
#include <algorithm>
#include <numeric>
#include <functional>
#include <fstream>
__global__ void mykernel(int* annotations, int* devarr, bool* subs,
 int* sum, unsigned long len, unsigned long nsubs, unsigned long usize, int roisum)
{
    // Kernel function to run on GPU
    // Defining variables (stored in each kernel)
    unsigned long th_id = blockIdx.x * blockDim.x + threadIdx.x;
    int barr[9084] = {0}; //Array for storing coverage
    int totsum = 0; // Sum of covered points
    int count_roi = 0;
    int penalty = 0;
    int alpha = 4000;

    int ct = 0;
    if(th_id < len){
        for(unsigned long i = 0; i < nsubs; i++)
        {
            int ind = devarr[th_id*nsubs + i];
            for(unsigned long j = 0; j < usize; j++)
            {
                if(subs[ind*usize + j]){
                    barr[j] += 1;
                }
            }

        }
        for(int i = 0 ; i < usize ; i++){
            if(barr[i] >= annotations[i]){
                totsum += 1;
                if(annotations[i] == 2){
                    count_roi += 1;
                }
            }
        }
        if(count_roi > 0 && count_roi < roisum){
            penalty = alpha*(roisum/count_roi);
        }
        if(count_roi == 0){
            penalty = alpha*roisum;
        }

        sum[th_id] = totsum - penalty;
    }else sum[th_id] = 0;

}

void readfromtxt(){

    //Specify the inputs
    int num_sensors = 5;
    int ncombs = 1.5e8; // Size of chopped Combinations Matrix
    unsigned long ndp = 9084;
    unsigned long campos = 83;
    unsigned long numpans = 3;

    std::cout << "num combs : " << ncombs << std::endl;

    // Dynamically allocate arrays on CPU
```

```
    int* array = (int*)malloc(ncombs*num_sensors*sizeof(int));
    bool* subs_array = (bool*)malloc(ndp*campos*numpans*sizeof(bool*));
    int* annot_array = (int*)malloc(ndp*sizeof(int));

    //Load subsets from txt file and store in 1D array
    std::ifstream subsfile("Subsets.txt");
    double b;
    unsigned long col_s = 0;
    while (subsfile >> b)
    {
        subs_array[col_s] = (bool) b;

        col_s +=1;

    }
    for(int i = 0; i < 15 ; i++){
        std::cout << subs_array[i] << std::endl;
    }
    std::cout << std::endl << std::endl  <<std::endl;
    std::ifstream myfile("combtinations.txt");
    double bb;
    unsigned long col = 0;
    while (myfile >> bb)
     {
        array[col] = (int) bb;
        if(col < 10) std::cout << bb << std::endl;
        col += 1;
     }
    // Store annotations in a 1D array
    // The annotation of a point describes whether it is
    // a ROI, obstacle or normal data point
    std::ifstream annotfile("Annotations.txt");
    double an;
    unsigned long col2 = 0;
    while (annotfile >> an)
    {
        annot_array[col2] =(int) an;
        col2 += 1;
    }
    // Make annotation array (to be used inside kernel)
    int* init_cov = (int*)malloc(ndp*sizeof(int));
    //int init_cov[ndp];
    int c = 0;
    for(int i = 0 ; i < ndp ; i++){
        if(annot_array[i] == 2){
            c += 1;
            init_cov[i] = 2;
        }else{
            init_cov[i] = 1;
        }
    }
    //GPU variables
    unsigned long n_threads_per_block = 1024;
    unsigned long n_blocks = (ncombs + n_threads_per_block - 1)/n_threads_per_block;

    std::cout << "Number of blocks :" << n_blocks << std::endl;
    unsigned long data_n = n_blocks*n_threads_per_block; // Total number of available threads

    //Vectorize array for GPU calculations
    unsigned long chop_combs;
    chop_combs = ncombs;
```

```
    std::cout << "No. of available threads: " <<  data_n << std::endl;
    std::cout << "Number of used threads : " << chop_combs << std::endl;

    size_t i_datasize = chop_combs*sizeof(int);
    size_t array_datas = chop_combs*num_sensors*sizeof(int);
    size_t bool_subs_size = ndp*numpans*campos*sizeof(bool);
    size_t annot_size = ndp*sizeof(int);

    std::cout << "i_datasize [bytes] : " << i_datasize <<  std::endl;

    // Allocate CPU Memory
    int* sum_host = new int[chop_combs];

    std:: cout << "Array size : " << array_datas <<" and subs size "
    << bool_subs_size << std::endl;

    // Allocate GPU Memory
    int* annot_dev;
    bool* subs_dev;
    int* sum_dev;
    int* array_dev;

    cudaMalloc(&subs_dev,bool_subs_size);
    cudaMalloc(&array_dev, array_datas);
    cudaMalloc(&sum_dev,i_datasize);
    cudaMalloc(&annot_dev,annot_size);

    // Copy host (CPU) arrays to device (GPU) arrays
    cudaMemcpy(subs_dev, subs_array, bool_subs_size, cudaMemcpyHostToDevice);
    cudaMemcpy(sum_dev, sum_host, i_datasize, cudaMemcpyHostToDevice);
    cudaMemcpy(array_dev, array, array_datas, cudaMemcpyHostToDevice);
    cudaMemcpy(annot_dev,init_cov,annot_size,cudaMemcpyHostToDevice);

    // Run "mykernel" function on GPU threads with gpu timing
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    mykernel <<< n_blocks,n_threads_per_block >>>
    (annot_dev,array_dev,subs_dev,sum_dev,chop_combs,num_sensors,ndp,c);

    cudaDeviceSynchronize();
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("The elapsed time for kernel execution was %.2f ms\n", milliseconds);
    // Copy results back to cpu memory
    cudaMemcpy(sum_host, sum_dev, i_datasize, cudaMemcpyDeviceToHost);

    // Post process
    int max = 0;
    unsigned long  ind = 0;
    for (unsigned long  i = 0; i < chop_combs ; i++){
        if(sum_host[i] > max){
            max = sum_host[i];
            ind = i;
        }
```

```
    }
    std::cout << "Max val : " << max << std::endl;
    printf("Highest coverage value at index %lu. \n",ind);
    std::cout << "The index represents camera index: ";
    for(int m = 0; m < num_sensors ; m++){

        printf("%i ", array[ind*num_sensors + m]);
    }
    std::cout << std::endl;

    //Free allocated memory on CPU and GPU
    cudaFree(subs_dev);
    cudaFree(sum_dev);
    cudaFree(array_dev);
    delete[] sum_host;
    free(array);
    free(subs_array);

}
```

A.10.4 Matlab Program for Visualization

```
%% Program to Evaluate Fitness of a Sensor Combination
clear;
clc;
close all;
env = environment_generate('mini_0705.mat');
datax = env.datax;                          % Discrete data points (x)
datay = env.datay;                          % Discrete data points (y)
dataz = env.dataz;                          % Discrete data points (z)
campx = env.campx;                          % Discrete placement points (x)
campy = env.campy;                          % Discrete placement points (y)
campz = env.campz;                          % Discrete placement points (z)
obsta = env.obsta;
annot = env.annot;

% Preallocate variables and initialize
ldata = length(datax);                      % Number of data points
ncams = 5;
c = 0;
% Determine initial coverage array
for i = 1:ldata
    if(annot(i) == 2)
        init_cov(i) = 2;
        c = c + 1;
    else
        init_cov(i) = 1;
    end
end
b = false(ncams,ldata);                     % Coverage matrix - Combs
iter = 0;
p = [0.0264   -0.0296    0.7689    0.8033    0.8050];    %Pan
x = [0          0          0    0.2426    7.7448];       % X Pos
z = [7.9870    9.6687   12.2536 0 0];                    % Z Pos

y(1:ncams) = -6;

% Compute coverage for all possible camera poses and positions
for i = 1:ncams
```

```
    pan = p(i);
        % Compute coverage of all data points
        for n = 1:ldata
            b(i,n) = covered1(n,x(i),y(i),z(i),datax,datay,dataz,...
                                obsta,pan);
        end
end
cove = zeros(1,ldata);
out = zeros(1,ldata);
cf = 0;
totsum = 0;
% Evaluate visibility and coverage from all sensors
for j = 1:ncams
    for i = 1:ldata
        if( b(j,i) )
            cove(i) = cove(i) + 1;
        end
    end
end
% Evaluation k-coverage
for k = 1:ldata
    if(cove(k) >= init_cov(k))
        totsum = totsum + 1;
        out(k) = 1;
        if(init_cov(k) == 2)
            cf = cf + 1;
        end
    end
end
tot = totsum;
% Figure for visualization
figure()
ccc = 0;
for i = 1:ldata
    if(annot(i) == 2 && out(i) == 1)
        scatter3(datax(i),dataz(i),datay(i),'b');
        hold on
    end
    if(annot(i) == 2 && out(i) == 0)
        scatter3(datax(i),dataz(i),datay(i),'r');
        hold on
    end
    if out(i) == 0
        ccc = ccc + 1;
        scatter3(datax(i), dataz(i), datay(i),'m')
        hold on
    end
end
for(k = 1:length(campx))
    scatter3(campx(k),campy(k),campz(k),'g*')
    hold on
end
for j = 1:ncams
    scatter3(x(j), y(j), z(j),'k')
    hold on

end
for i = 1:length(obsta)
    scatter3(obsta(i,1), obsta(i,3), obsta(i,2),'r*')
    hold on
end
```

```
h = zeros(5, 1);
h(1) = plot(NaN,NaN,'ob');
h(2) = plot(NaN,NaN,'ok');
h(3) = plot(NaN,NaN,'*g');
h(4) = plot(NaN,NaN,'*r');
h(5) = plot(NaN,NaN,'om');
legend(h, 'ROI','Camera positions','Possible placement points',...
        'Obstacles','Uncovered data points');
title('Coverage Results');
xlabel('X [m]')
ylabel('Y [m]')
zlabel('Z [m]')

fprintf('The sum is %i \n',tot)
if cf < c
    fprintf('The ROI is not fully covered\n')
else
    fprintf('The ROI is fully covered\n')
end

disp(tot/9084)
```

A.11 Matlab Scripts for the Continuous Neighborhood Optimization

A.11.1 Objective Funciton

```
function val = objfunc(p,yi,ncams)
%Import problem structure
env = environment_generate('mini_0705.mat');
datax = env.datax;
datay = env.datay;
dataz = env.dataz;
obsta = env.obsta;
annot = env.annot;

ldata = length(datax);
b = zeros(ncams,ldata);
alpha = 4000;
% Compute coverage of all data points
for i = 1:ncams
    x = p(5 + i);
    y = yi(i);
    z = p(10 + i);
    for n = 1:ldata
        b(i,n) = covered1(n,x,y,z,datax,datay,dataz,...
                        obsta,p(i));
    end
end
c = 0;
% Initialize array of required coverage
init_cov = zeros(ldata,1);
for i = 1:ldata
    if(annot(i) == 2)
        init_cov(i) = 2;
        c = c + 1;
    else
        init_cov(i) = 1;
```

```
    end
end
cove = zeros(ldata,1);
cf = 0;
tot = 0;
penalty = 0;
bch = false(ldata,1);
% Evaluate coverage
for j = 1:ncams
    for i = 1:ldata
        if( b(j,i) )
            cove(i) = cove(i) + 1;
        end
    end
end
for k = 1:ldata
    if(cove(k) >= init_cov(k))
        bch(k) = true;

        if(init_cov(k) == 2)
            cf = cf + 1;
        end
    end
end

% Penalize solutions that does not fully cover the ROI
if cf > 0 && cf < c
    penalty = alpha*(c/cf);
end
if cf == 0
    penalty = alpha*c;
end

sumt = sum(bch) - penalty;
if sumt <= 0
    sumt = 1;
end
val = 100000/sumt; % Output value
end
```

A.11.2 Main Script

```
clc; clear

% Initialize problem

ncams = 5;
yi(1:ncams) = -6;
func = @(p)objfunc(p,yi,ncams);
% p is array of decision variables

% Determine bounds for each decision variable
lb(1:5) = [0 -pi/4 -pi/4 0 0];
ub(1:5) = [pi/2 pi/4 pi/4 pi/2 pi/2];
lb(6:15) = [0 0 0 0 6.5 6.5 8.5 11 0 0];
ub(6:15) = [0 0 0 1 8.5 8.5 10.5 13 0 0];

%Initial guess
x0 = [
```

```
    0 0 pi/4 pi/4 pi/4 ...
    0 0 0 0 7.5 ...
    7.5 9.5 12 0 0
    ];
%% Fmincon
tic
[x,fval,output] = fmincon(func,x0,[],[],[],[],lb,ub);
toc

%% Simulated Annealing
tic
options = saoptimset('PlotFcns',{@saplotbestx,...
          @saplotbestf,@saplotx,@saplotf},'Display','iter');
[x,fval,output] = simulannealbnd(func,x0,lb,ub,options)
toc
%% Global Search
tic
gs = GlobalSearch('Display','iter','StartPointsToRun','bounds');
problem = createOptimProblem('fmincon','x0',x0,'objective',func,'lb',lb,...
    'ub',ub);

x = run(gs,problem);
toc
%% Particle Swarm Optimization
tic
nvars = 15;
options = optimoptions('particleswarm','Display','iter')
[x,fval,exitflag,output] = particleswarm(func,nvars,lb,ub,options)
toc
```