Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Parallel Gaussian Processes for Optimal Sensor Placement

Author:
Tolga Hasan Dur

Supervisor:
Rossella Arcucci

Submitted in partial fulfillment of the requirements for the MSc degree in MSc. Computing Science of Imperial College London

August 2019

Abstract

Optimal sensor placement is of vital importance to many complex systems. However, despite its importance, existing algorithms that optimize sensor placements have scalability issues and therefore do not work with the big data generated by computational simulations. Instead, they use small-scale data collected by existing sensors, which is expensive and leads to imprecise results for large domains. This thesis attempts to provide a solution by parallelizing computations of a Gaussian process (GP) based sensor placement algorithm. By doing so, it provides unprecedented use of GP's with big data to solve this important problem in spatial statistics. As an example, sensor placements are optimized for part of the test-side around the London South Bank University.

Instead of running the algorithm for the whole domain, our approach only considers relevant sub-domains of the test-side that were obtained by Fluidity's native domain decomposition. For computations on these, a mixture of multi-processing and multi-threading is used to fully utilize all available resources. The results are validated by comparison of the pollution levels predicted by the posterior GP to the real pollution levels, and measurement of performance in data assimilation. According to both measures, our algorithm leads to near-perfect sensor placements. For instance, for one sub-domain, our mean estimation is off by only $6.15\mathrm{e}{-3}$ while random placement is off by over 1.93.

Acknowledgments

I would like to thank my supervisor Dr. Rossella Arcucci. During this thesis, I was often confused with various mathematical concepts that she helped me understand. Her foresight was of utmost importance for the development of this thesis.

Furthermore, I would like to thank the Data Learning Group with which I was fortunate enough to discuss various topics in the broad field of machine learning and statistics. The topics we covered not only provided great ideas for this thesis, but also lead to motivation and deeper insight in general.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The reliability and correctness of complex computational systems depends on input data that is often measured by expensive sensors. Hence, there is a high cost associated with unnecessary or wrong sensor placement. Specifically, sensors placed without a sound methodology can lead to wrong generalizations that can often go unnoticed. Therefore, optimal sensor placement is a fundamentally important task for many important projects in research and industry that is often neglected.

In the MAGIC (Managing Air for Green Inner Cities) project that this thesis is working under, for instance, optimal sensor placement is an unsolved task with previous placements being decided upon in a seemingly arbitrary manner. The ESPRC-funded project attempts to solve environmental problems, such as heat-islands and air pollution, in urban environments. In particular, it attempts to combat the heavy usage of heating, ventilation, and cooling (HVAC) systems. These produce CO_2 and other pollutants that lead to temperature increases and therefore the need for even more usage of these systems. To break this vicious cycle, HVAC systems need to be replaced with environmentally friendly alternatives. For instance, air-conditioning needs to be replaced with natural ventilation systems wherever possible. However, due to reduced wind speeds, varying air pollution and noise, identifying areas in which an investment in natural ventilation makes sense, is a challenging task. The MAGIC project, therefore, builds an advanced computational system that can identify areas by accurately predicting airflow and air quality. For the collection of necessary input data, it developed a monitor with six sensors and the capacity to simultaneously measure carbon dioxide CO_2, carbon monoxide CO, nitrogen dioxide NO_2, temperature, relative humidity RH and barometric pressure. However, placement of these sensors, for instance at the initial test-side around the London South Bank University (LSBU) that is illustrated in figure 1.1, were decided upon seemingly arbitrarily [4].

The lack of scalable sensor placement algorithms is part of the reason why many projects do not optimize sensor placement. Existing algorithms for optimal sensor placement do not work with big data due to scalability issues. Instead, they work

Figure 1.1: LSBU test-side in which initial experiments were run.

with small-scale data that is collected by already placed sensors. For instance, the Gaussian process based sensor placement approach, which is widely regarded as the most precise methodology, does not scale due to the $O(N^3)$ time-complexity of Gaussian processes [5]. However, relying on small-scale data that is collected by already placed sensors is not only expensive, but also imprecise for large domains. Particularly in the MAGIC project, this is a big issue as sensors are ought to be placed in large, previously untouched domains without access to previous sensor measurements. For example, besides the LSBU test-side, experiments were done in a 23-story tower in Hangzhou, China. Here, there was no access to previous measurements that could have been used to optimize sensor placements [6].

1.2 Aims

This thesis aims to use large-scale simulation data to optimally place sensors such that we have the most accurate representation of pollution across the entire space. The work is based on the Gaussian process based sensor placement algorithms suggested by Krause, Singh, and Guestrin [5]. As mentioned, the defining challenge is the previously undone application of Gaussian processes with big data to solve this important problem in spatial statistics.

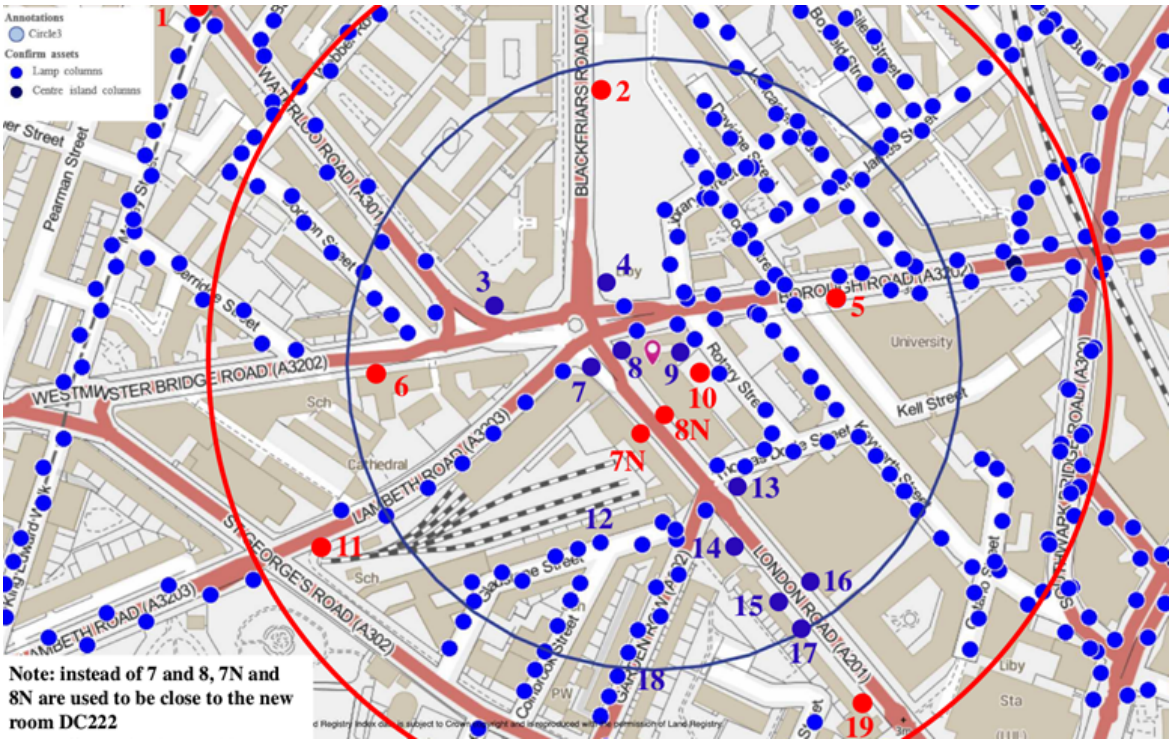


Figure 1.2: Location of existing sensors in the LSBU test-side. Blue: all available sensors. Red: sensors chosen for past work [1].

The code is ought to be used in two different, equally important scenarios: First, in certain domains such as the LSBU test-side, an abundance of sensors were placed as can be seen in figure 1.2. The final code should be able to select the best k of these existing sensors. Second, in new domains, the code should give the best k positions for sensor placement.

As an example, sensor placements were optimized for part of the LSBU test-side. For it, computational simulations were done to model pollution and other environmental factors in the fluid dynamics software Fluidity. Here, pollution measurements were split into seven different parts based on geographical location. The big data gathered from these simulations was used to optimize sensor placements for one of these parts.

1.3 Outcomes

We successfully implemented and optimized all three algorithms suggested by Krause, Singh, and Guestrin [5]. Furthermore, a much faster fourth algorithm was developed. The code was optimized for easy reusability and performance. The former is archived through an intuitive class structure that provides an easy understandable API. The latter through several methods such as the use of the best performing libraries for our computational resources confirmed through extensive tests. The defining challenge of scaling was solved by careful selection of relevant data and

parallel execution. Specifically, the LSBU test-side was decomposed into 32 sub-domains in Fluidity. Then, relevant sub-domains were identified and a separate sensor placement algorithm ran on each parallelly. Within each sub-domain, multi-threading was used to further exploit available resources.

During this thesis, several challenges were faced. Most notably, due to the sparse nature of the data, we faced numerical stability issues that were leading to negative conditional variances. We computed sensor placements with two alternative solutions to this: the first one is adding a small jitter to the covariance matrix and the second is taking absolute values of conditional variances. Our analysis shows that the former leads to much better results. Further, due to the large scale, monotonicity has to be ensured for mutual information. Fortunately, our analysis confirms that this is the case for our data.

We validated the results with two different methodologies: First, we used a Gaussian process fitted with the sensor placements to predict the mean pollution value of all nodes in the domain. By comparing them to the real mean pollution value and the mean pollution value predicted with random placement, we confirmed that indeed our sensor placements are close to optimal. Second, we calculated the performance of our sensor placements in data assimilation. While this is not what we have been optimizing for, it is heavily used within the MAGIC project and is therefore important. For this, sensor placements suggested by our algorithms are performing much better than random placement.

The sensor placement optimization in the LSBU test-side was very successful with near-perfect results according to our validation methodologies. We found that the fourth algorithm we developed is up to nine times faster than the algorithms suggested by Krause, Singh, and Guestrin, while not altering results significantly. Hence, we suggest using it unless precision is of utmost importance.

The rest of the thesis is structured as follows: Chapter 2 provides a background to Gaussian processes and previous work in sensor placement optimization. Chapter 3 describes the unique nature of the data while chapter 4 discusses the unique challenges stemming from it. Chapter 5 describes the unique design of the code that optimizes for performance and easy usability. Results and validation are discussed in chapter 6. Finally, chapter 7 concludes with a summary of achievements and a brief discussion about possible future work.

Chapter 2

Background

Optimal sensor placement can be separated into geometric and model-based approaches. The prior assume geometric properties and optimizes for area coverage while the latter take a probabilistic approach to optimize for predictive quality [5]. In the following subsections, both are briefly described.

2.1 Geometric Approaches

Unlike their model-based counterparts, geometric approaches are not probability-based. Instead, they assume fixed sensing radii and perfect observation within these radii, and then place the sensors such that the entire space is covered [7] [8]. Gonzales-Banos, for example, solves the problem as an instance of the Art-Gallery algorithm in logarithmic time complexity. Other geometric approaches include that of Bai et al. [9] and Chakrabarty and Iyengar [10].

However, as Krause, Singh, and Guestrin showed, the geometric assumption is too strong as sensors make noisy measurements that cannot be described by spheres or any other fixed-size representation [5]. Rather, their sensing field depends on the correlation between sensors which can be negative or positive. This means that the combined sensing radius can be smaller or larger than just the sum of fixed-size spherical sensing radii of two sensors. Therefore, a single sensor can often not held responsible for sensing one location. This fundamentally wrong assumption leads to sub-optimal results.

2.2 Model-Based Approaches

Model-based approaches take a model of the world and then place sensors to optimize an objective function of that model based on some design criteria. This model is some instance of the regression problem described in equation 2.1. Here, y is what the sensors measure (e.g. CO_2) throughout the space and x the chosen sensor placements. To optimize, classical and Bayesian experimental design focus on maxi-

mizing the quality of the parameter estimations, while information-theoretic criteria focus on minimizing uncertainty after the observations are made.

$$y \approx f(\boldsymbol{x}, \theta) \tag{2.1}$$

2.2.1 Design Criteria

Classical Design Criteria These criteria come from classical statistics literature where the problem in equation 2.1 is described as one of linear regression. The optimization criteria are derived from the variances of this linear regression as described in equation 2.2 and equation 2.3.

$$Var(\theta) = \sigma^2 (X^T X)^{-1} \tag{2.2}$$

$$Var(y_i) = \sigma^2 \boldsymbol{x}_i^T (X^T X)^{-1} \boldsymbol{x}_i \tag{2.3}$$

As can be seen, both dependent on the matrix $X^T X$. If this matrix is small, predictions y and parameter estimate θ will be accurate. The different design criteria (e.g. A-optimal, D-optimal and E-optimal criteria) describe the smallness of this matrix in different ways [5].

Bayesian Design Criteria The classical design criteria essentially tries to minimize the estimation error with the maximum likelihood methodology. In contrast, Bayesian design criteria uses Bayesian statistics to solve the regression in equation 2.1. Here, a utility function $U(p(\theta), \theta^*)$ is defined where $p(\theta)$ describes the prior distribution modelling the states of the world (i.e. the parameters) and θ^* the true state of the world. Collecting observations x_A change the prior distribution $p(\theta)$ to the posterior distribution $p(\theta | x_A = \boldsymbol{x}_A)$ due to new information. The Bayesian design criteria are defined as the change in expected utility between prior and posterior distribution [5].

Information-Theoretic Criteria These are special cases of the Bayesian experimental criteria where an information-theoretic function such as entropy or mutual information is defined as the utility function [5].

Here, entropy tries to quantify uncertainty by finding a function whose output continuously increases with uncertainty in a random variable. According to Shannon, such a function should be additive, transitive and maximal when $P_X(x)$ is uniform. As proven by him, the only function that satisfies these conditions is the one seen in equation 2.4, which is the formal definition of entropy.

$$H(X) = -\sum_x P_X(x) \log P_X(x) = -E_{P_X} \log P_X \tag{2.4}$$

Intuitively, entropy is the minimum number of yes/no questions it takes to guess a random variable given perfect knowledge of the underlying distribution. Naturally, conditional entropy is the uncertainty of the random variable X given knowledge about another random variable Y as seen in equation 2.5.

$$H(X|Y) = \sum_y P_Y(y) \left[-\sum_x P_{X|Y}(x|y) \log \left(P_{X|Y}(x|y) \right) \right] = E_{P_Y} \left[-E_{P_{X|Y}} \log P_{X|Y} \right] \quad (2.5)$$

Mutual information, on the other hand, is the reduction in uncertainty after observing random variable Y [11]. It is defined in equation 2.6 and is the one taken by this thesis.

$$I(X;Y) = H(X) - H(X|Y) \quad (2.6)$$

2.2.2 Optimization

Optimizing for any of the described design criteria yields a combinatorial optimization problem where one must search for the best design of sensors among a very large number of candidate solutions. Instead, as the number of candidate solutions is too large for an exhaustive search approach, commonly some heuristic is taken. However, there are alternative ways to optimize these uncertainty criteria that do not come from the field of combinatorial search. These include continuous relaxation and Machine Learning approaches. For instance, Worden and Burrows solve an instance of the sensor placement optimization by training neural networks with genetic algorithms: an optimization technique that works by randomly generating solutions (i.e. a neural network mapping sensor placements to some uncertainty criteria), evaluating their performance and then inheriting the best solutions properties to the next ”generation” of solution [12]. Other approaches include active learning which describes the instance in which the data is unlabelled and the algorithm requests labeling of individual examples when optimizing [5]. In this thesis, a Gaussian process based greedy approach is taken.

2.3 Gaussian Processes

This thesis places sensors with the Gaussian process based approach that is suggested by Krause, Singh, and Guestrin. As shown by them, this approach is far superior to any other sensor placement algorithm [5]. Here, the regression in equation 2.1 is described by a Gaussian Process. Gaussian Processes are a way to learn functions from observations that solve issues regular linear regression methods such as maximum likelihood have. Namely, unlike the maximum likelihood methodology, they do not over-fit, do not require a previous definition of the functional form to be learned, and quantify uncertainty in their predictions.

2.3.1 General Background

The goal of any regression setting is to find a function f that best possibly explains observations y such that $y = f(x) + \epsilon$. When doing this, traditional regression methodologies assume a fixed functional form (i.e. the polynomial of the function) and, in the case of linear regression, linearity in the parameters. This means that only the parameters are "learned" by the data. Further, these have to be linear, meaning that, for example, θ^2 cannot occur. This very limiting frame creates the biggest problem that parametric regression methodologies have: over-fitting. In the case of Maximum likelihood estimation (MLE), for instance, parameters are chosen such that the observations are most likely to occur. That is, the likelihood function $L(\theta|x)$ is maximized. Naturally, it easy to overfit as the data might not be representative or the wrong functional form might have been chosen.

The maximum a posteriori (MAP) estimation mitigates this by adding a normally distributed prior $p(\theta)$ to the likelihood function before optimizing. The prior acts as a regularizer: due to the shape of the normal distribution, it penalizes extreme values when optimizing for parameters. The prior distribution is chosen such that all known information about the parameters are encoded in it. For instance, this can the interval that they are in or how they are correlated with each other. If no information are known about the parameters, the standard normal distribution is chosen as a prior. However, this penalization only postpones over-fitting to a higher degree polynomial. Usually, this is only a postponement by one or two polynomials. Therefore, over-fitting is not solved.

Bayesian linear regression remedies this problem by providing confidence intervals to the MAP estimates. The idea is to focus on the distribution of the parameters instead of parameter estimates. Essentially, parameters are eliminated by integration as illustrated in equation 2.7 [13]. In effect, this process finds the best estimate of the distribution of the parameters given our observations. Instead of a learned function, we have a distribution that we can sample parameters from. This leads to a range of possible functions with the mean being the same as the maximum a posteriori estimate. We can, therefore, quantify uncertainty when making decisions with this data. For instance, in the example seen in figure 2.1, we can see that we are very unsure for predictions regarding $x = -0.3$ but fairly sure for predictions regarding $x = 0.2$. However, we still have to define the functional form: i.e we have to choose the degree of the polynomial we want to learn [14]. This decision is based on data that might not be representative and can, therefore, lead to bad generalizations.

$$p(y|\theta) = \int p(y|x, \theta)p(\theta)d\theta \tag{2.7}$$

Gaussian processes eliminate the choice of the functional form by learning the distribution over possible functions instead of the distribution over parameter values. Intuitively, this is done by treating the function as an infinitely long vector of points

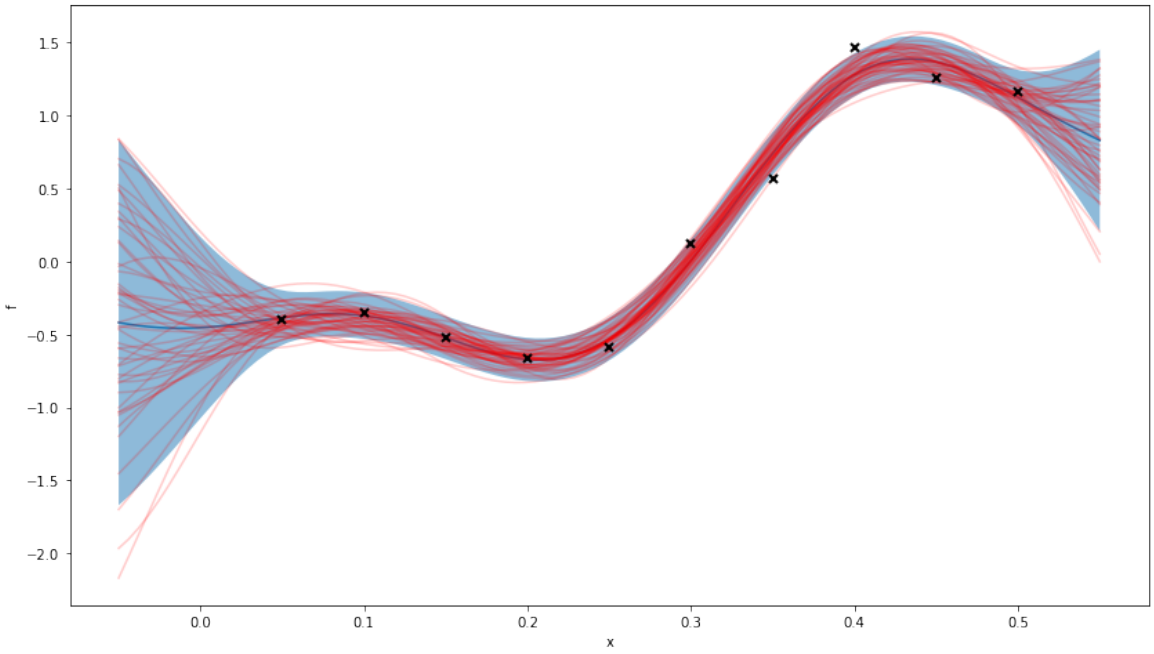


Figure 2.1: Hundred samples from a learned function.

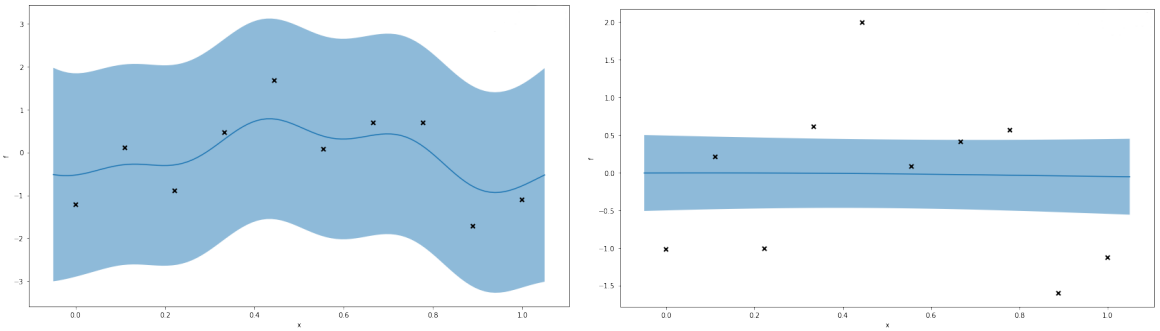
$\tilde{\boldsymbol{f}}$. The distribution of this infinitely long vector is learned in the following way: a finite set of points $\boldsymbol{f} = x_1, ..., x_N, \boldsymbol{f} \subseteq \tilde{\boldsymbol{f}}$ is taken as function values and our prior distribution defined as seen in equation 2.8. If we then integrate all points $\tilde{\boldsymbol{f}}$ out, as done in equation 2.9, we only have to learn the distribution over the finite set of points $\boldsymbol{f}$ to know the distribution over possible functions [13] [15]. The assumption made is that $p(f(x1), ...f(x_N))$ is jointly Gaussian with some mean $\boldsymbol{\mu}$ and some covariance $\boldsymbol{\Sigma}$. Gaussian processes try to learn the best estimate of this $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ given the data available [16]. After doing so, we can sample functions instead of parameters from it and are therefore, in effect, able to learn functions of all polynomials.

$$\mathcal{N}(\begin{bmatrix}\boldsymbol{\mu}_f \\ \boldsymbol{\mu}_{\tilde{f}}\end{bmatrix}, \begin{bmatrix}\boldsymbol{\Sigma}_{ff} & \boldsymbol{\Sigma}f\tilde{f} \\ \boldsymbol{\Sigma}_{\tilde{f}f} & \boldsymbol{\Sigma}\tilde{f}\tilde{f}\end{bmatrix}) \tag{2.8}$$

$$p(\boldsymbol{f}) = \int p(\boldsymbol{f}, \tilde{\boldsymbol{f}}) d\tilde{\boldsymbol{f}} = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{f}}, \boldsymbol{\Sigma}_{\boldsymbol{f}}) \tag{2.9}$$

The prior distribution $p(\boldsymbol{f})$ of a Gaussian process is defined by a mean function $m(x)$ and a kernel $k(x_i, x_j)$. The mean function $m(x)$ specifies the value of the function on average. That is, what do we expect the output value to be if no observations are available. In most cases, without, for instance, physical laws underlying the observations, this is set to $m(x) = 0$. The kernel $k(x_i, x_j)$ specifies the covariance Σ_{ij} of x_i and x_j. The idea is that if x_i and x_j are similar according to the kernel, then we expect the output to be similar too. Therefore, the choice of the kernel and the hyper-parameters of it determine how the Gaussian process and the predictions are going to look like. For instance, in the commonly used Gaussian kernel, the lengthscale hyper-parameter determines how strong values correlate with each other and hence

how ”wiggly” the Gaussian process is. The standard deviation hyper-parameter determines the amplitude, and the noise hyper-parameter the confidence interval of the Gaussian process. The Gaussian process in figure 2.2, for example, has a rather low lengthscale visible by its heavy wiggles, high noise visible by the light gray area and a high standard deviation visible by the amplitudes [17] [13].



(a) Variance:1, Length-scale:0.1, Noise:1 (b) Variance:0.1, Length-scale:0.8, Noise:0

Figure 2.2: Effect of hyper-parameters in a RBF-kernel to the final GP.

For a set of observations, the prior distribution $p(\boldsymbol{f})$ is converted into a posterior distribution $p(\boldsymbol{f}|\boldsymbol{X}, \boldsymbol{y})$ that explains the data. This conversion stems from Bayes rule as seen in equation 2.10. Here, $p(\boldsymbol{y}|\boldsymbol{X})$ is the marginal likelihood of our training outputs $\boldsymbol{y}$ given our training inputs $\boldsymbol{X}$. This is the ”evidence” that we are incorporating into our model. Further, $p(\boldsymbol{y}|\boldsymbol{f}, \boldsymbol{X}) = \mathcal{N}(f(x), \sigma^2\boldsymbol{I})$ is the Gaussian likelihood that encodes the noise in our model.

$$p(\boldsymbol{f}|\boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{f}, \boldsymbol{X})p(\boldsymbol{f})}{p(\boldsymbol{y}|\boldsymbol{X})} \tag{2.10}$$

Given a prior $p(\boldsymbol{f}|\boldsymbol{X}) = \mathcal{N}(m(x)|\boldsymbol{K})$ at training inputs and a set of training points $\boldsymbol{f}_*$, we can use the Bayes formula to learn the new mean and covariance of our model. Analogously to equation 2.9, we integrate the infinite set of points out of our model as seen in equation 2.11. To find the new posterior mean and covariance, we have to find the right-hand side of equation 2.11. As seen in equation 2.12, we can substitute the values with our mean and kernel, and, in the case of Σ_{ff} with the prior covariance $\boldsymbol{K}$ plus the noise estimation $\sigma^2\boldsymbol{I}$ that we gained from the Gaussian likelihood.

$$p(\boldsymbol{f}, \boldsymbol{f}_*) = \int p(\boldsymbol{f}, \boldsymbol{f}_*, \tilde{\boldsymbol{f}}) d\tilde{\boldsymbol{f}} = \mathcal{N}(\begin{bmatrix} \boldsymbol{\mu}_f \\ \boldsymbol{\mu}_{f_*} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{ff} & \boldsymbol{\Sigma}_{ff_*} \\ \boldsymbol{\Sigma}_{f_*f} & \boldsymbol{\Sigma}_{f_*f_*} \end{bmatrix}) \tag{2.11}$$

$$\mathcal{N}(\begin{bmatrix} \boldsymbol{\mu}_f \\ \boldsymbol{\mu}_{f_*} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{ff} & \boldsymbol{\Sigma}_{ff_*} \\ \boldsymbol{\Sigma}_{f_*f} & \boldsymbol{\Sigma}_{f_*f_*} \end{bmatrix}) = \mathcal{N}(\begin{bmatrix} m(\boldsymbol{X}) \\ m(\boldsymbol{X}_*) \end{bmatrix}, \begin{bmatrix} \boldsymbol{K} + \sigma^2\boldsymbol{I} & k(\boldsymbol{X}, \boldsymbol{X}_*) \\ k(\boldsymbol{X}_*, \boldsymbol{X}) & k(\boldsymbol{X}_*, \boldsymbol{X}_*) \end{bmatrix}) \tag{2.12}$$

Given this information, we can derive the new posterior mean and covariance, seen in equation 2.13 and 2.14. The necessary inversion of a matrix of all observations is the bottleneck of Gaussian processes. It it is the reason why they have $O(N^3)$ time complexity and why the practical limit of Gaussian processes is said to be approximately $10,000$ observations [13]. However, due to the confidence intervals and the non-parametricity, we can be very confident in our model and our predictions.

$$m_{post}(\boldsymbol{X}_*) = m(\boldsymbol{X}_*) + k(\boldsymbol{X}_*, \boldsymbol{X})(\boldsymbol{K} + \sigma^2\boldsymbol{I})^{-1})(y - m(\boldsymbol{X}) \tag{2.13}$$

$$k_{post}(\boldsymbol{X}_*, \boldsymbol{X}_*) = k(\boldsymbol{X}_*, \boldsymbol{X}_*) - k(\boldsymbol{X}_*, \boldsymbol{X})(\boldsymbol{K} + \sigma^2\boldsymbol{I})^{-1})k(\boldsymbol{X}, \boldsymbol{X}_*) \tag{2.14}$$

2.3.2 Bayesian Optimization

Bayesian optimization is an optimization methodology for black-box functions that are hard to evaluate. There are many similarities between Bayesian optimization and the sensor placement algorithm that uses Gaussian processes. Hence, to enable deeper insight into the algorithm whose details are discussed in the following subsection, a brief background is given to Bayesian optimization.

Bayesian optimization builds a probabilistic proxy model for the objective that is cheaper to evaluate than the original model. This proxy model is build using the outcomes of past experiments. It is optimized to evaluate where to evaluate the true model next. In this way, the objective, such as the maximum, can be found in a few selected evaluations of the original model. In a sense, this is a greedy approach to optimizing functions, though it is not referred to it in this way [15].

The proxy model is often build using Gaussian processes. It is an approximation of the real model and provides the information when deciding where to evaluate the model next. Once the point of the next evaluation is decided, the real model is evaluated at that point and the outcome is incorporated into the proxy model. In the case of a Gaussian process based proxy model, this means updating mean and covariance as shown in equation 2.13 and equation 2.14. Given good decisions regarding the evaluation points, this process leads to a very fast determination about the objective [15].

Acquisition or utility functions decide the points of evaluation by trading off exploration and exploitation. Exploration seeks places of high variance and uncertainty. These are marked by the areas in a Gaussian process in which the confidence intervals are the largest. Exploitation, in the case of minimization, seeks places of low mean. The acquisition function trades off these two in some defined way. The point of evaluation is chosen to be the maximum of the acquisition function. The one seen in figure 2.3 and figure 2.4, for instance, samples many functions from the proxy, and then simply counts how many sample functions are below zero at every given point. The maximum, and therefore the point of the next evaluation is, therefore, the point at which most samples are below zero [15].

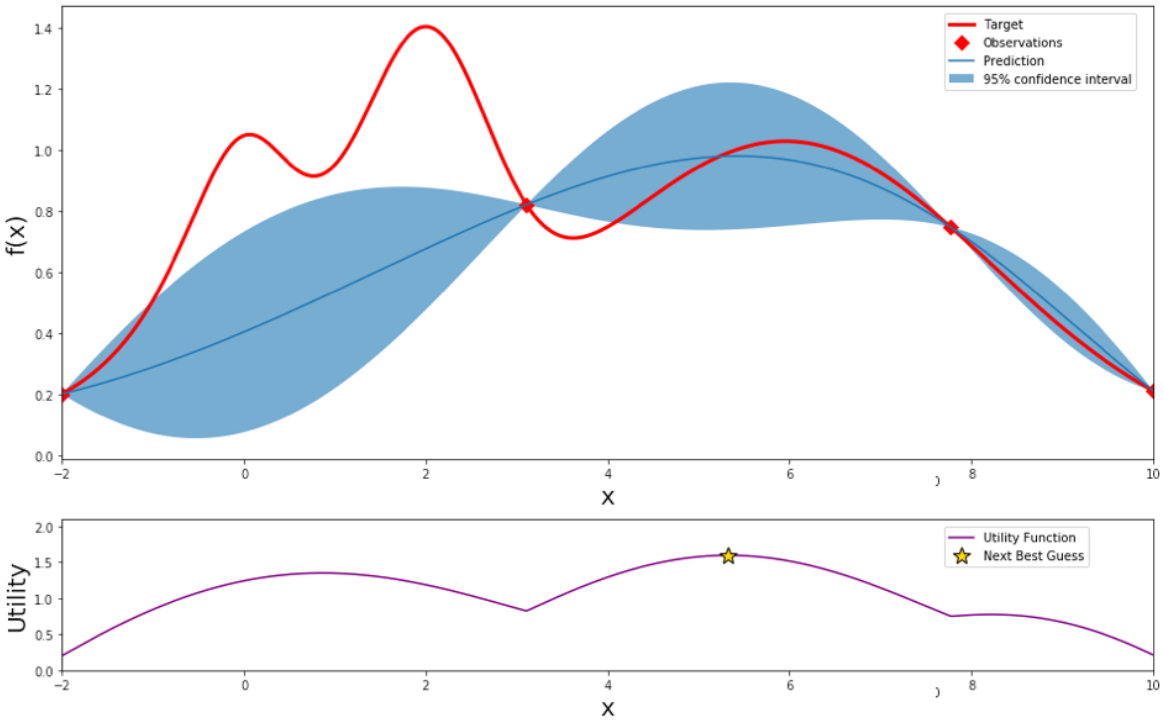


Figure 2.3: Example of a Bayesian optimization after five evaluation steps.

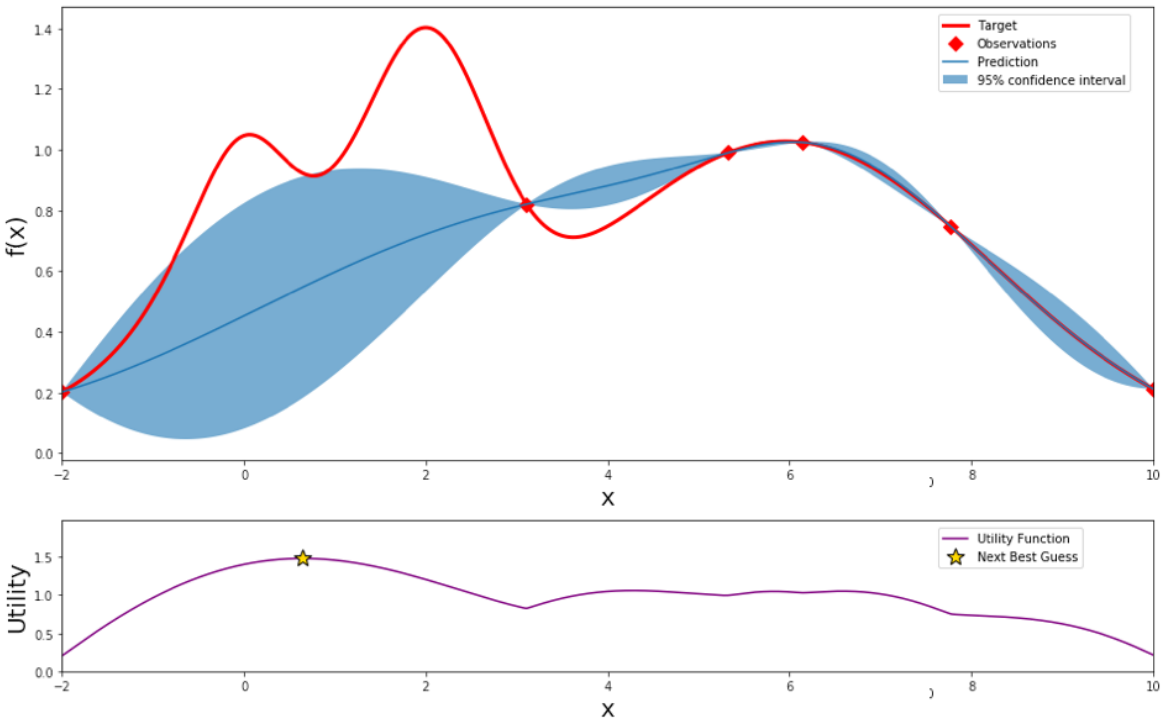


Figure 2.4: Example of a Bayesian optimization after six evaluation steps.

2.3.3 Gaussian Processes for Sensor Placement

For optimal sensor placement, we model the equation 2.1 with a Gaussian Process. Then, we want to choose the input sensor placements X such that uncertainty across space is reduced and the Gaussian Process optimized.

Mathematically, this means that the conditional entropy $H(X_{V\setminus A}|X_A)$ is minimized. Here, V are all nodes (e.g. points of interest) and A the chosen sensor placements. As $H(X_{V\setminus A}|X_A)$ is equivalent to to $H(X_V) - H(X_A)$, this means that minimizing $H(X_{V\setminus A}|X_A)$ is equivalent to maximizing entropy $H(X_A)$. This is an NP-hard combinatorial optimization problem and the basic approach means trying every possible combination of the sensor placements X_A until the best one is found, which is computationally not feasible [18].

Therefore, the typical approach suggests to greedily place the sensors at the point of the highest entropy [19][20]. That is, the locations we are most uncertain about given the already placed sensors akin to the exploitative objective of the acquisition function in Bayesian optimization. This approach, however, places sensors at the border of the area we are interested in and therefore wastes sensing information as can be seen in figure 2.5 [2] [21].

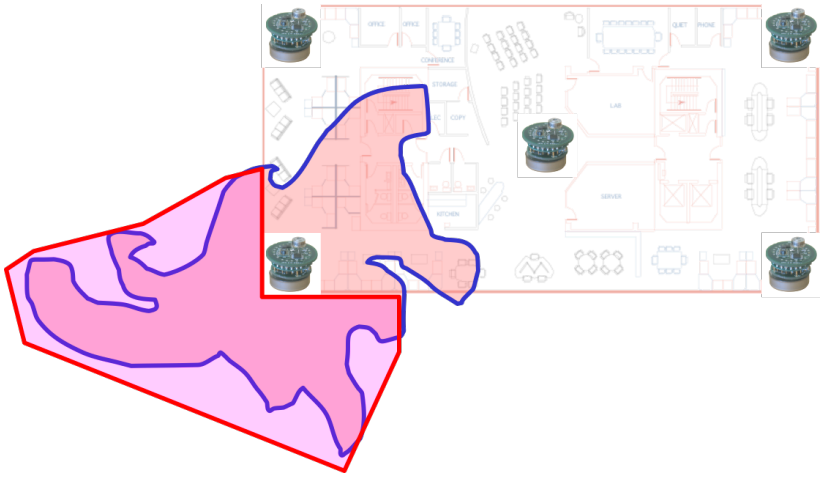


Figure 2.5: "Wasted" information observed by O'Hagan [2][3].

Mutual information, which is described in equation 2.6, on the other hand, does not have this problem as it measures the effect of sensor placement on the posterior uncertainty of the Gaussian Process directly. Our goal is, therefore, to find the best sensor locations A such that $I(A, V\setminus A)$ is maximized. As this is not computationally feasible for the same reasons as above, we use a greedy approach in which we maximize $I(A \cup y) - I(A)$ instead. This is equivalent to maximizing $H(y|A) - H(y|\widehat{A})$ with $\widehat{A} = V\setminus(A \cup y)$as described in section 2.2. Intuitively, this does the following: maximizing $H(y|A)$ means searching for a position y with high uncertainty given

the already placed sensor A (i.e a position that is different). Maximizing $-H(y|\widehat{A})$ means searching for a position y with low uncertainty given the rest of the field that we are interested in $V\backslash A$ (i.e. y is informative). This means that greedily selecting placements with the highest mutual information is finding positions are different and informative at the same, which is a little bit like the exploration and exploitation objective of the acquisition function in Bayesian optimization.

$$H(X|Y) = \frac{1}{2}\log(2\pi \exp \sigma^2_{X|Y}) = \frac{1}{2}\sigma^2_{X|Y} + \frac{1}{2}(\log(2\pi) + 1) \tag{2.15}$$

Assuming that the random variables are normally distributed, as they have to be for Gaussian processes to work, the conditional entropys $H(y|A)$ and $H(y|\widehat{A})$ are equivalent to the definition in equation 2.15. Following the transformation below, it can be seen that maximizing $H(y|A) - H(y|\widehat{A})$ is equivalent to maximizing $\sigma^2_{y|A}/\sigma^2_{y|\widehat{A}}$ [22].

$$\begin{aligned} max.(H(y|A) - H(y|\widehat{A})) &= max.(\frac{1}{2}\sigma^2_{y|A} + \frac{1}{2}(\log(2\pi)+1) - (\frac{1}{2}\sigma^2_{y|\widehat{A}} + \frac{1}{2}(\log(2\pi)+1)) \\ &= max.(\frac{1}{2}\sigma^2_{y|A} - \frac{1}{2}\sigma^2_{y|A}) \\ &= max.(\sigma^2_{y|A}/\sigma^2_{y|\widehat{A}}) \end{aligned}$$

$\sigma^2_{y|A}$ is equivalent to the variance update rule of Gaussian processes which is described in equation 2.14. Essentially, the greedy algorithm computes two Gaussian processes for each y in $V\backslash A$. The sensor placement is then chosen to be the one with the highest $\sigma^2_{y|A}/\sigma^2_{y|\widehat{A}}$.

It should be noted that mutual information in this context heavily depends on an accurate, non-stationary model for $P(X_V)$ as seen in equation 2.6. With the Gaussian process representation, this means that the kernel used to update the variance as seen in equation 2.14 has to be non-stationary.

Nemhauser et. al's theorem guarantees that this greedy approach is a (1-1/e) approximation of the optimal non-greedy solution if the function is monotonic submodular [23]. This would mean that in the case of mutual information, the greedy approach is always guaranteed to be 63% and mostly even 95% of the optimal solution. However, while mutual information is indeed sub-modular, it is not always monotonic. As Krause, Singh, and Guestrin showed, this does not matter for spaces with less than 2000 nodes and the approximation holds even if mutual information is not monotonic. For spaces with more than 2000 nodes, nonetheless, mutual information has to be monotonic for Nemhauser et. al's approximation guarantee to hold [5].

The pseudo-code for this methodology can be seen in algorithm 1 with k, S and U being the number of sensors to be placed, the possible sensor placements and the impossible sensor placements respectively. To calculate one sensor selection, mutual information is calculated for each node $S\backslash A$. The node with the highest mutual information value is taken as the next sensor placement. This process is done k times with respect to information gathered from previous sensor selections. The high time-complexity of $O(kn^4)$ comes from the number of matrix inversions necessary: for each sensor selection, we have to inverse $S\backslash A$ matrices the size of $\widehat{A}$.

Algorithm 1 Greedy approximation for maximizing mutual information. Time complexity: $O(kn^4)$ [5].

Input: Covariance matrix $\Sigma_{VV}, k, V = S \cup U$
Output: Sensor selection $A \subseteq S$

$A \leftarrow \emptyset$;
for $j = 1$ **to** k **do**
  for $y \in S\backslash A$ **do**
    $\delta_y \leftarrow H(y|A) - H(y|\widehat{A})$;
  end for
  $y^* \leftarrow argmax_{y \in S\backslash A} \delta_y$;
  $A \leftarrow A \cup y^*$;
end for

Algorithm 2 uses a priority queue to minimize time-complexity from $O(kn^4)$ to $O(kn^3)$. In the first iteration, mutual information is evaluated for all nodes in S and the one with the largest value is selected as the first sensor placement. For all other selections, mutual information is recalculated in the order of the priority queue. Only after a second pick, the node is selected as the next sensor placement. This process significantly lowers the number of mutual information evaluation which is the bottleneck of the algorithm as described earlier. To work, the function we are learning needs to be sub-modular and monotonic. If this is not ensured, a node can be picked twice even though another node, which has not been evaluated at all, has larger mutual information value. If ensured, however, Algorithm 2 is a lot faster than Algorithm 1 while always having the same output.

Algorithm 3 uses the local kernel approach to further lower time-complexity down to $O(kn)$. This works by only considering sensor placements for which the covariance is greater than ϵ when calculating entropy. Further, instead of evaluating mutual information for all nodes in $S\backslash A$ as done by Algorithm 1, we are only evaluating it for nodes $x \in S$ for which $|K(y^*, x)| > \epsilon$. As the covariance matrix in the case of sensor placement is usually very sparse, this lowers the size of the matrix that we are inversing and the number of evaluations significantly. Depending on the value of ϵ and the covariance matrix, however, this improvement in time-complexity might decrease accuracy.

Algorithm 2 Lazy evaluation of the greedy approximation algorithm using a priority queue. Time complexity: $O(kn^3)$ [5].

Input: Covariance matrix $\Sigma_{VV}, k, V = S \cup U$
Output: Sensor selection $A \subseteq S$

$A \leftarrow \emptyset$;
foreach $y \in S$ **do** $\delta_y \leftarrow +\infty$;
for $j = 1$ **to** k **do**
    foreach $y \in S \backslash A$ **do** $current_y \leftarrow False$;
    while True **do**
        $y^* \leftarrow argmax_{y \in S \backslash A} \delta_y$;
        if $current_{y^*}$ **then break;**
        $\delta_{y^*} \leftarrow H(y|A) - H(y|\widehat{A})$;
        $current_{y^*} \leftarrow True$
    end while
    $A \leftarrow A \cup y^*$;
end for

Algorithm 3 Local kernel approach of the greedy approximation algorithm. Time complexity: $O(kn)$ [5].

Input: Covariance matrix $\Sigma_{VV}, k, V = S \cup U$
Output: Sensor selection $A \subseteq S$

$A \leftarrow \emptyset$;
for $y \in S$ **do**
    $\delta_{y^*} \leftarrow H(y) - \tilde{H}_\epsilon(y|V \backslash y)$;
end for
for $j = 1$ **to** k **do**
    $y^* \leftarrow argmax_y \delta_y$;
    $A \leftarrow A \cup y^*$;
    for $y \in N(y^*, \epsilon)$ **do**
        $\delta_{y^*} \leftarrow \tilde{H}_\epsilon(y|A) - \tilde{H}_\epsilon(y|\widehat{A})$;
    end for
end for

Chapter 3

Data

The data that is used for sensor placement in the LSBU test-side is generated in the fluid dynamics software Fluidity. Fluidity uses the visualization toolkit (VTK) to represent and store data. The computational representation that is seen in 3.1 is an instance of such a VTK representation. The data for the LSBU test-side we are interested in has 861,959 nodes and measures velocity, pressure and pollution for 537 time-steps. Pollution measurements, which is what we are basing our sensor placements algorithms on, were split into seven parts based on location in the test-side. In this thesis, we optimized sensor placements for one part of the measurements.

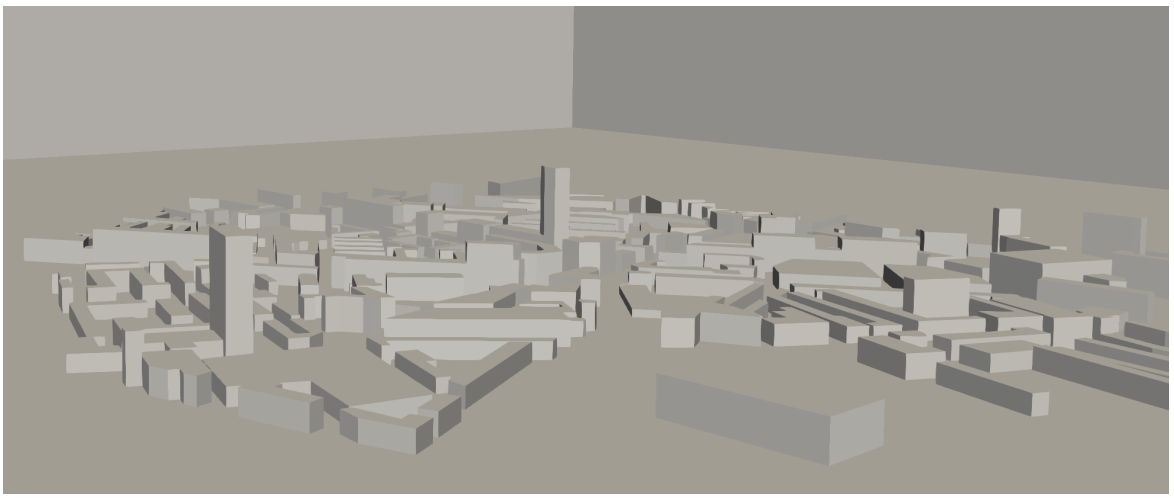


Figure 3.1: Computational representation of the LSBU test-side in VTK.

For easier computations, researchers in the MAGIC project used a domain decomposition method to decompose the data into 10 and 32 sub-domains as visualized in figures 3.3 and 3.2 respectively. This was done in Fluidity to ensure that the LSBU test-side is decomposed optimally for parallel execution. In the former, the test-side was decomposed into sub-domains of approximately $70,000 - 80,000$ nodes, while in the latter, the test-side was decomposed into sub-domains of approximately $25,000 - 30,000$ nodes. For the following of the thesis, the former will be referred to as LSBU10 and the latter as LSBU32 akin to its names within the MAGIC project. It

should be noted that the representation of the sub-domains is optimized for an accurate position similar to most geographical maps. Hence, a size comparison between the sub-domain cannot be done on the bases of computational representations.

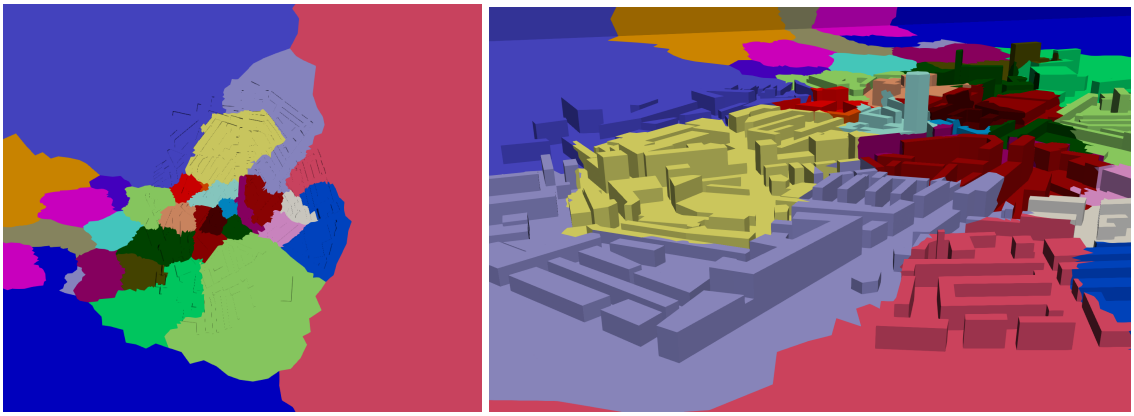


Figure 3.2: 32 sub-domains in the LSBU test-side.

Due to computational restrictions, we decided to use LSBU32 for our computations. Our algorithm is using Gaussian processes to calculate optimal sensor placement. In doing so, it inverses many large matrices. For this, $80,000$ nodes per sub-domain are way too large for a feasible computation with any architecture, which is why we decided to do calculations on LSBU32. The details of these computational restrictions are discussed in chapter 4.5 and 5.

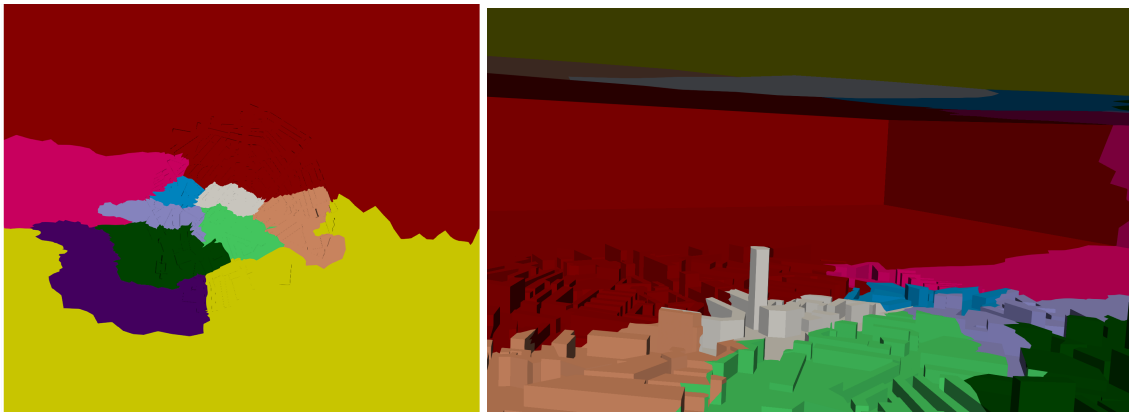


Figure 3.3: 10 sub-domains in the LSBU test-side.

Nodes are unevenly distributed with more nodes on ground levels and more observations in the central of the test-side, which can be seen in figure 3.4. Naturally, this means that areas with many nodes are heavier weighted than areas with fewer nodes in the sensor placement. However, this is less of an issue with the parallel execution method described in chapter 4.4, as the nodes in the sub-domain of interest are roughly evenly distributed.

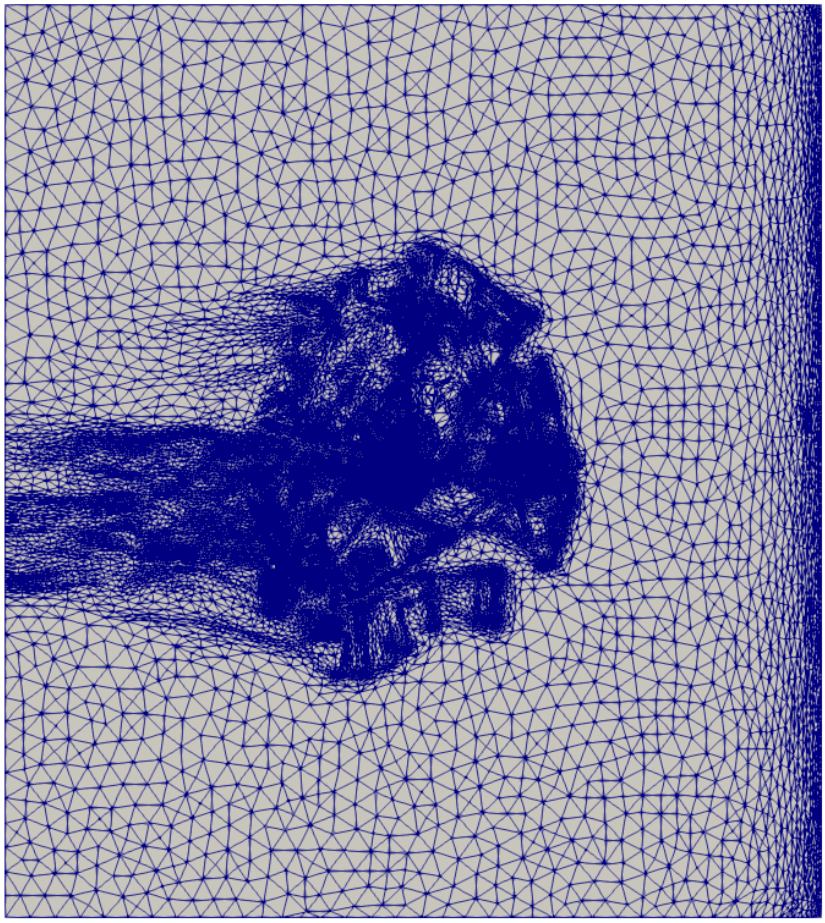


Figure 3.4: 2D visualization of nodes in the LSBU test-side

As described in chapter 2, nodes are separated into possible and impossible sensor placements in the sensor placement algorithm. Due to practical reasons, we decided that all nodes under 30 meters of height, which is the height of the tallest building in the test-side, are possible sensor placements, and nodes over 30 meters of height impossible sensor placements.

3.1 Distribution of Pollution

Pollution measurements for almost all nodes are zero or very close to zero. Normalization and standardization do not change this situation due to outliers in a considerable amount: namely, areas in which pollution is concentrated in. Furthermore, his even occurs, albeit to a much lesser extent, in the relevant sub-domains. This again is due to most nodes in the relevant sub-domain having little or no sig-

nificant observations. This sparse nature of the data is leading to numerical stability problems that are discussed in chapter 4.3.

For the Gaussian process based sensor placement algorithm to work, the residuals have to be normally distributed. We did some tests regarding the distribution of the residuals by plotting histograms of the residuals for random nodes in the test-side. One such histogram can be seen in figure 3.5. However, these tests were not extensive enough to confirm that pollution is normally definitely distributed. Hence, this thesis assumes that it is.

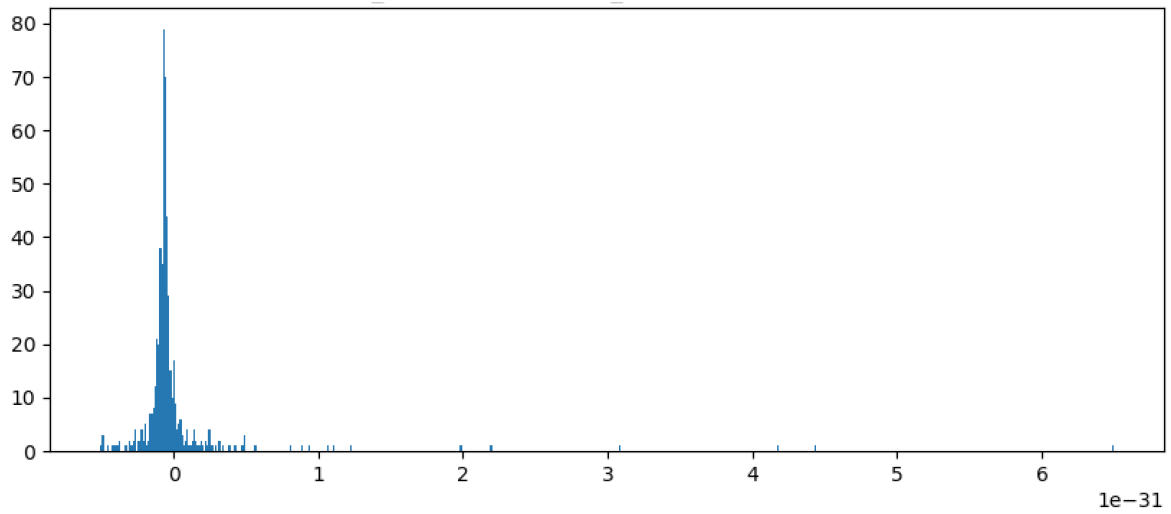


Figure 3.5: Histogram of residuals for a random node in sub-domain eight.

3.2 Covariance Matrix Estimation

The greedy algorithm used in this thesis has better results for non-stationary and non-isotropic covariance matrices as shown by Krause, Singh, and Guestrin [5]. Finding a sound methodology to estimate such a covariance matrix at this large scale was one of the greatest challenges we faced in this thesis.

Unfortunately, the easily implementable radial base kernel is isotropic and stationary. Nonetheless, initially, we used the RBF-kernel function, calculated optimal hyper-parameters based on the average tracer values at each point. The prior covariance matrix was then used as the input covariance matrix for the algorithm. However, apart from the matrix being stationary and isotropic, there are a few issues with this approach. First, the small number of hyper-parameters in the RBF-kernel is not sufficient for an accurate estimation of the covariance matrix. The only part that is differentiating the estimated covariance matrix from a random one is the three hyper-parameters lengthscale, signal, and noise. These are not sufficient for a precise estimation of a covariance matrix of this size. Second, a non-stationary covariance matrix based on average pollution values does not differ variances into

account. Pollution values change over time at differing speeds: some nodes have a large variance while others have a small variance in pollution values. Taking the average over time to estimate a non-stational covariance matrix does not take this variance into account. A particular node might have a comparatively high average pollution value due to outliers. Placing sensors based on such outliers does naturally not lead to the optimal solution.

However, as we were using simulation data and therefore had access to the values of every node, we had access to pollution values for enough time-steps to be able to calculate a sample covariance matrix that is very close to the real covariance matrix. This covariance matrix is not isotropic or stationary and does not have the problems mentioned in the previous paragraph. It is, however, very centered around zero due to the sparse nature of the data discussed in the previous section. As mentioned, this leads to numerical stability problems that are discussed in chapter 4.3.

Chapter 4

Challenges

The example sensor placement that Krause, Singh, and Guestrin propose in their paper, was done in the 54 node sensor network in the Intel Berkeley Lab. This means that the authors had access to data gathered from 54 evenly spaced sensors and had to choose the best k sensors out of these 54 sensors [5]. In the MAGIC project case, however, we are dealing with a much larger scale as the data is generated through simulations rather than actual measurements. Furthermore, we want to ensure that we have the best sensor placements at all times. This leads to a few challenges that are discussed in the following.

4.1 Data Assimilation

We continuously need to run the sensor placement algorithm to ensure that we have the best sensor placements at all times. However, as correlations in the space change over time, the error in our final model will add up and create a large discrepancy after a while. Data assimilation solves this by, instead of taking the output of the model as the input of the next time step, optimizing between the new observations and model forecasts to get the best estimate between the two. In effect, it continuously improves the model by incorporating the observed data. To do this in a computationally cheap way, Arcucci et al. described a method to reduce space with truncated singular value decomposition and thus make data assimilation computationally feasible to use [24]. When using the model developed by this thesis, Arcucci et al.'s described data assimilation methodology will be used to continuously improve the model and prohibit the error to get larger over time. However, depending on the sensor placements, performance varies greatly. Hence, in chapter 6, we show that sensor placements suggested by our algorithms are very well suited for data assimilation.

4.2 Monotonicity

Dealing with 861,595 nodes means that we have to ensure that monotonicity holds in our dataset as described in chapter 2. If it does not than the placements are not

ensured to be at least 63% as good as the optimal solution and might very well be worse than alternative solutions such as the standard geometric approach. Furthermore, the priority queue algorithm 2 relies on the monotonicity assumption to work.

Monotonicity in the case of mutual information holds if $H(y|A) - H(y|\widehat{A}) >= 0$. If it is not, Krause, Singh, and Guestrin suggests to maximize for $H(y|A) - H(y|Z \cup \widehat{A})$ instead with Z being a grid of unobservable nodes. Adding a large enough Z ensures that $H(y|Z \cup \widehat{A})$ is small enough for mutual information to be approximately monotonic [3]. Fortunately, we do not have to worry about this as our analysis shows that $H(y|A) - H(y|\widehat{A}) >= 0$ holds for the LSBU test-side.

4.3 Numerical Stability

The sparse nature of the data and covariance matrix is leading to numerical stability problems when calculating conditional variance's. Even using the double-precision floating-point format does not solve this problem as covariances can be smaller than $1\mathrm{e}{-20}$ for some nodes. This sometimes leads to the strange case of negative conditional variances as $\Sigma yA\Sigma_{AA}^{-1}\Sigma Ay$ can be larger than $K(y, y)$ in equation 2.14. Naturally, negative conditional variances, which should never occur, lead to undefined entropys due to undefined negative logarithms. This problem can also be seen when computing the determinant of the covariance matrix: although inverting the matrix works without issues, the determinant is said to be zero by NumPy and other libraries. The problem, again, is that the determinant is too small to represent in Python. Commonly, this leads to wrong sensor placements.

There are two ways to solve this issue: First, we could take absolute values of the conditional variance, which would ensure that the anomaly of negative conditional variances is eliminated. However, this alters sensor placements as conditional variances are in the heart of the sensor placement algorithms. Second, as often done in Gaussian processes, we could add a jitter $\mathbf{I} * x$ to the covariance matrix with x being very small number in the range of [$1\mathrm{e}{-3}$, $1\mathrm{e}{-10}$]. However, as some of our variance values are smaller than $1\mathrm{e}{-20}$, adding a number at this range alters the sensor placements as well. Interestingly, we found that it does not matter which of the two we use when not using the local kernel approach. If we do, however, we found that the latter solution leads to much better results than the former solution. We show this in chapter 6, where we compare validation results for sensor placements calculated with both alternatives.

4.4 Parallel Execution

The computational costs of using the algorithm proposed by Krause, Singh, and Guestrin on an area with 861,959 nodes are way too high to compute. This is by far the greatest challenge as we would have to inverse matrices of $861,959 * 861,959$

for each iteration, with $k * 861,959$ iterations in the naive case. However, as we discussed in chapter 3, the test-side is decomposed in 32 sub-domains. Our solution therefore only considers relevant sub-domains. For them, it computes optimal sensor placements parallelly.

We are using a mix of multi-processing and multi-threading to archive efficient parallelization. There are several ways with which parallelization can be archived in Python. Almost all of these methods can be separated into multi-threading and multi-processing. In multi-threading, the program is separated into blocks of threads that share memory. For instance, an email client might have a thread responsible for continuously fetching data, and another one responsible for I/O operation. These threads can then be concurrently executed to archive parallelization. In multi-processing, on the other hand, the program is separated into smaller processes, each of which has its memory. These sub-processes are then run concurrently. The disadvantage of multi-processing over multi-threading is the high memory footprint: processes are expensive to create and delete, context switches are expensive and it is very expensive to exchange data as processes, unlike threads, not sharing memory. Multi-threading, however, has the disadvantage that it can easily result in deadlocks or livelocks due to the necessary locking mechanism that prevents race conditions. Furthermore, threads of the same process are usually run on one core unless a clever architecture is developed. In the multi-threading library of Python, for instance, the global interpreter lock is preventing threads from the same library to run on multiple cores. multi-processing, on the other hand, does not have this issue: sub-processes can run on different cores out of the box without any issues.

We run a separate process for each sub-domain of interest and multiple threads within each of these processes. Due to the overlapping nature of sub-domains, correlations between sub-domains are taken into account without further communication. Hence, we run a separate sensor placement algorithm for each sub-domain of interest. However, this only utilizes CPU's to the number of sub-domains in consideration and is therefore still too slow. For instance, in our case, we were placing sensors in two sub-domains but had over 50 CPU's available. Hence, we utilize multi-threading within each process to fully utilize all computational resources.

Chapter 5

Code Design

We optimized the architecture for maximum performance and easy reusability. The prior is especially important due to the large scale and the high time complexity we are working with. This means ensuring, for example, that a good balance in the trade-off between accuracy and speed is found. The latter is important as the MAGIC project is going to run experiments and eventually deploy the finished system in many cities around the world. Particularly for mathematicians and physicists who are developing the system, sensor placement is an auxiliary consideration. Hence, to ease this process, we developed an intuitive API that can be used akin to a library in Python.

5.1 Performance Optimization

Maximum performance is ensured in two ways. First, parallel execution is done to significantly lower matrix sizes and ensure that all available resources are exhausted. The details of this are discussed in chapter 4.4. Second, the code is designed to maximize computational efficiency while maintaining good accuracy. These design choices are discussed in the following.

Every second node is discarded to further lower matrix sizes. The practical limit of a Gaussian process is said to be approximately $10,000$ observations [13]. However, even after parallelization, the individual sub-domains have up to $30,000$ nodes. Furthermore, the sensor placement algorithm works by fitting many Gaussian processes for all nodes available in order to calculate the conditional entropy $H(y|\hat{A})$ as described in chapter 2. As such, our tests show that $15,000$ nodes are the limit for our algorithms. To overcome this, we decided to only consider every second node in each sub-domain. The associated precision loss is not too tragic as the nodes are placed very closely together.

Instead of an evenly spaced sample matrix, only nodes with observations are considered for sensor placement. Initially, we wanted to compute a sample matrix to increase accuracy and eliminate the bias towards areas with many nodes that are described in chapter 3. However, we are working with three dimensions which means

that even for a small area the size of $100m^3$, we have $1,000,000,000$ nodes with decimeter-level precision. Even after removing buildings and other areas that are not of interest, this number very quickly gets too large to compute. Fortunately, the nodes with observations are placed closely together, often with less than a quarter meter separation between them. This trade-off enables us to ensure scalability which is of vital importance for the MAGIC project.

The computationally best performing libraries are used for expensive matrix operations. Specifically, our tests have shown that given the available resources, NumPy is much faster than all tested alternatives. Unfortunately, we did not have access to any GPU's which naturally are very well suited for matrix operations. Hence, the goal was to find the best performing library for matrix operations on multiple CPU's. NumPy proved much faster than all the alternatives we have tested. Furthermore, several libraries similarly designed as NumPy but work with GPUs instead of CPU's. Hence, using NumPy eases the process of adapting the code on servers with GPU's.

We replaced the for-loop initialization seen in line 1 of Algorithm 2 with Pythons native list comprehension. Instead of re-initializing an array with False values for every node considered, we define a list of 3-tuples containing $\inf$, the index of the node and -1. When evaluating mutual information we then replace -1 with the current $j \in k$ and check for this to ensure that a node has been picked twice before being selected as an optimal sensor placement. This lowers time taken by Algorithm 2 significantly.

In the local kernel approach, the value of epsilon was optimized. The higher it is, the faster our algorithm as described in chapter 2. However, accuracy also depreciates as some of the nodes we are not considered to have high mutual information and are thus perhaps optimal sensor placements. Therefore we ran several tests which suggested a value of $1\mathrm{e}{-10}$ to be sufficiently small to preserve accuracy while significantly lowering the number of nodes in consideration. For instance, in sub-domain six and eight, which are the sub-domains we place optimize sensor placements in, the number of nodes considered decreases by approximately 92% and 98% respectively.

The option to specify already placed sensors was added to all algorithms. While this does not speed up computations, it enables users to stop and continue computations at any point at minimal costs. Therefore, the overall total length is lowered significantly in some cases, for example if servers crash. Furthermore, this eases the process for additional placement in spaces in which an there already exists a set of sensors.

Furthermore, we developed a fourth greedy algorithm that combines the priority queue approach of Algorithm 2 with the local kernel approach of Algorithm 3. Albeit the time-complexity is the same, our test shows that this algorithm is even faster than

Algorithm 3. Further, as long as monotonicity is ensured, the output is equal to the output of Algorithm 3. The pseudo-code for this fourth algorithm can be seen in the following.

Algorithm 4 Greedy sensor placement algorithm that uses priority queues and the local kernel approach.

Input: Covariance matrix $\Sigma_{VV}, k, V = S \cup U, A$
Output: Sensor selection $A \subseteq S$

$A \leftarrow A$;
$\epsilon \leftarrow 1\text{e}{-10}$
foreach $y \in S$ **do** $\delta_y \leftarrow +\infty$;
for $j = 1$ **to** k **do**
    foreach $y \in S \backslash A$ **do** $current_y \leftarrow False$;
    while True **do**
        $y^* \leftarrow argmax_{y \in S \backslash A} \delta_y$;
        if $current_{y^*}$ **then break;**
        $\delta_{y^*} \leftarrow \tilde{H}_\epsilon(y|A) - \tilde{H}_\epsilon(y|\widehat{A})$;
        $current_{y^*} \leftarrow True$
    end while
    $A \leftarrow A \cup y^*$;
end for

5.2 Class Structure

The solution is implemented with two static classes that provide intuitive API for easy sensor placement in and outside the MAGIC project. The first class, SensorPlacement, contains general algorithms for optimal sensor placement. The second class, MagicProject, contains the specific code for parallel sensor placement within the MAGIC project. It provides an API that can be used to place sensors in the LSBU test-side by simply specifying the sub-domain of interest.

SensorPlacement was designed such that it can be used for sensor placement in any area, even if this area is not part of the MAGIC project. Provided with the necessary inputs for the sensor greedy algorithms, SensorPlacement will calculate optimal sensor placements with the specified algorithm. The downside is, however, that inputs, such as the covariance matrix, are not calculated by MagicPlacement but have to be prepared by the user. Specifically, in the case of the covariance matrix, this can be difficult as discussed in chapter 3. Furthermore, it should be noted, that the epsilon value for the local kernel approach algorithm is optimized for the MAGIC project. When using SensorPlacement for another purpose, this value should be adjusted accordingly.

To ease the process for the MAGIC project, VTU-files were converted to CSV-files. In particular, pollution values of all time-steps are saved as *tracer.csv* in the respective sub-domain folder in the folder structure seen in figure 5.1. This ensures easy access to the data and effectively enables an intuitive design of the API, where instead of having to specify file-paths, one can simply indicate the sub-domain for which sensors ought to be placed in. Furthermore, using CSV-files instead of VTU-files means that the algorithms can be used in project where the data does not stem from Fluidity.

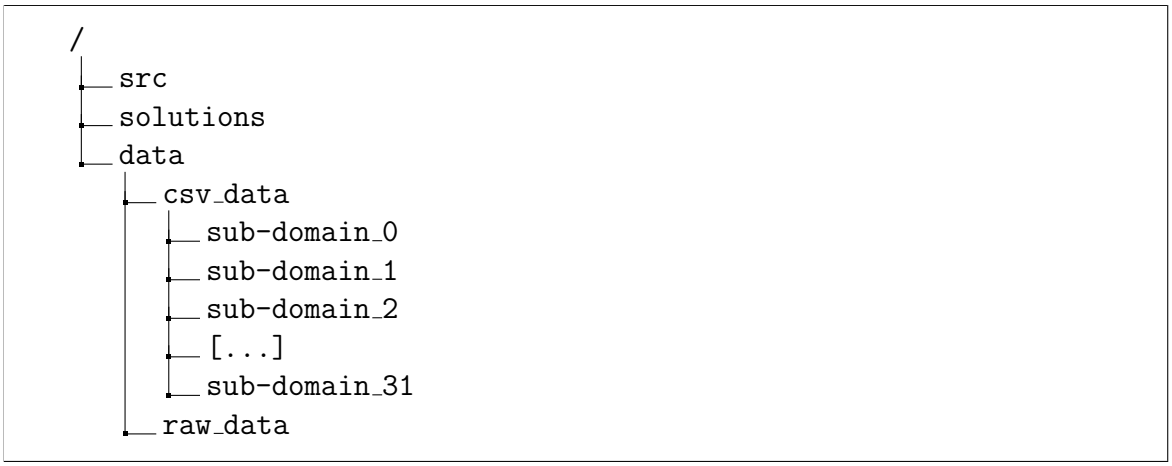


Figure 5.1: File structure that MagicProject requires for its API.

MagicProject provides an intuitive API that can be used to analyze data and compute optimal sensor placements for the MAGIC project by simply specifying which of the 32 sub-domains is of interest. The API relies on the file structure seen in figure 5.1 and the mentioned prior conversion of the VTU-files into CSV files. Other than that, no prior computations, file paths or other parameters are necessary. The API can simply be called with a few parameters such as an integer specifying the sub-domain of interest. MagicProject then does all necessary calculations such as the computation of the covariance matrix, the separation of all positions into placeable and unplaceable positions, and the setting up of the parallel execution structure. Then, SensorPlacement is called for the actual computations. Figure 5.3 illustrates this process with a simple UML sequence diagram.

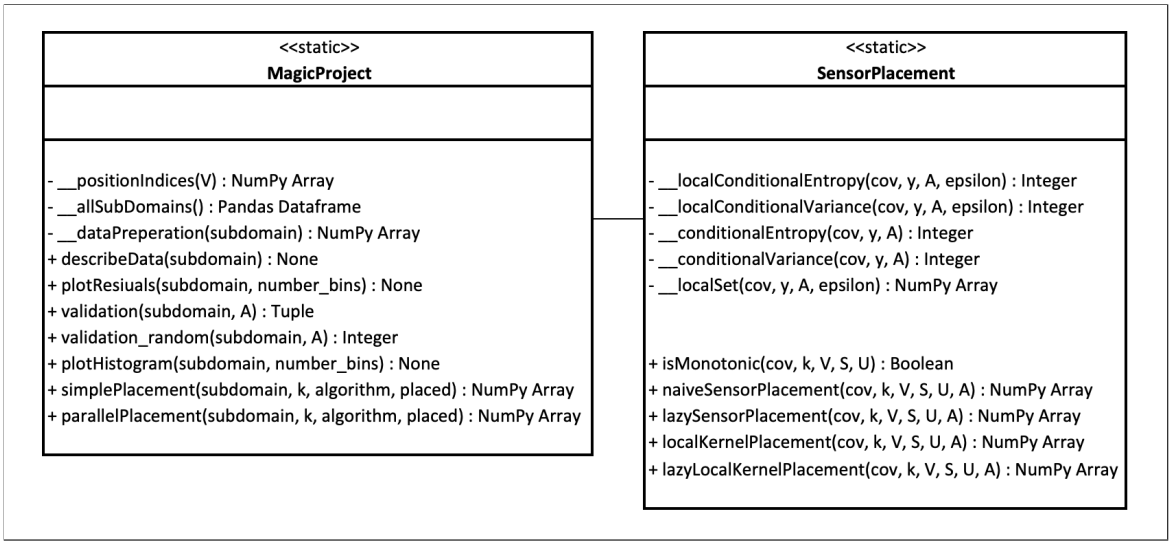


Figure 5.2: UML class diagram for MagicProject and SensorPlacement.

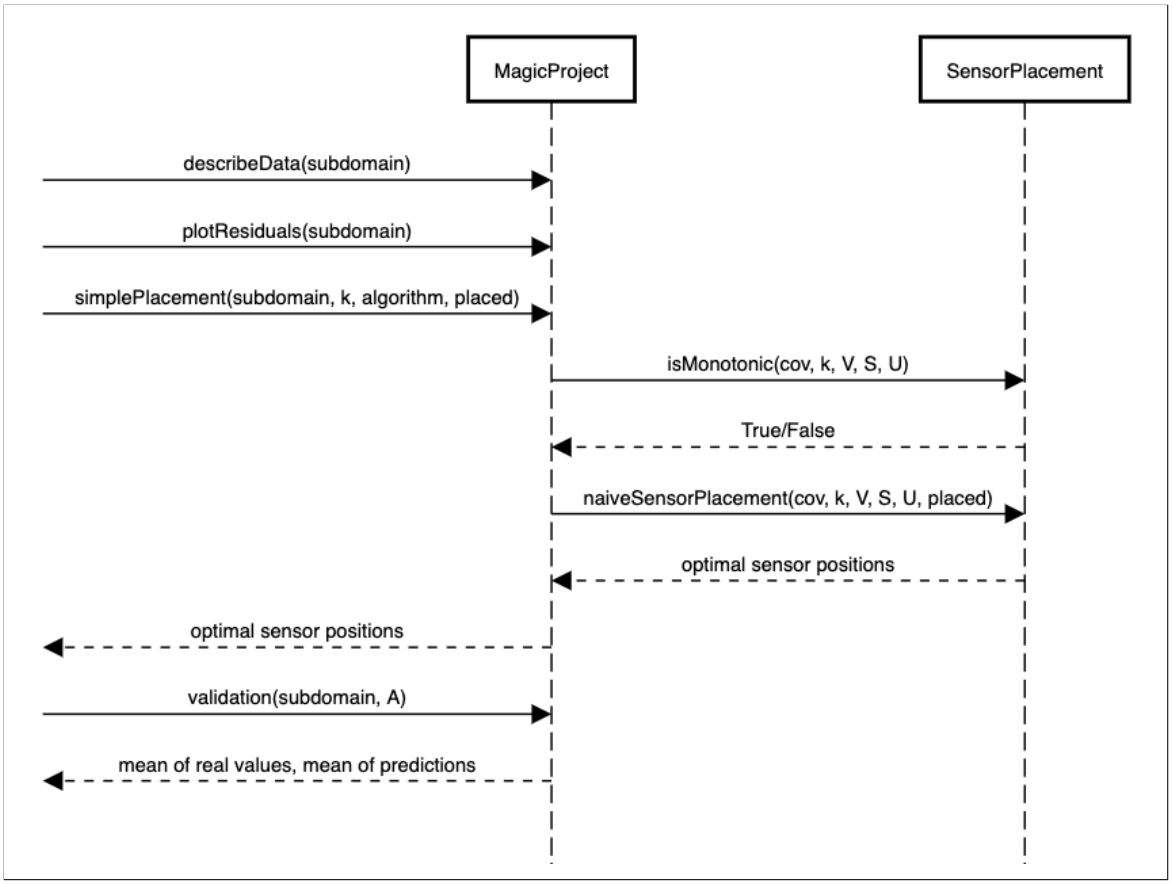


Figure 5.3: Simplified UML sequence diagram illustrating possible usage sequence.

Chapter 6

Results

Several decisions needed to be made to identify the best methodology for our requirements. For some, the answer was obvious. For instance, priority queues were always used, as time complexity is lowered from $O(kn^4)$ to $O(kn^3)$ without any loss in precision. For others, however, the answer was unclear. In particular, the local kernel approach further lowers time complexity to $O(kn)$, but, depending on the domain and value of epsilon, can lead to heavy losses in precision. Furthermore, the issue of numerical stability can be solved in two ways: by adding a jitter of $\mathbf{I} * x, x \in [1\mathrm{e}{-3}, 1\mathrm{e}{-10}]$ to the covariance matrix or by taking absolute values of the conditional variances as described in chapter 4.3. Interestingly, as mentioned in chapter 4.3, it does not matter for algorithm 1 and 2 which of the two alternatives is used to resolve numerical stability issues.

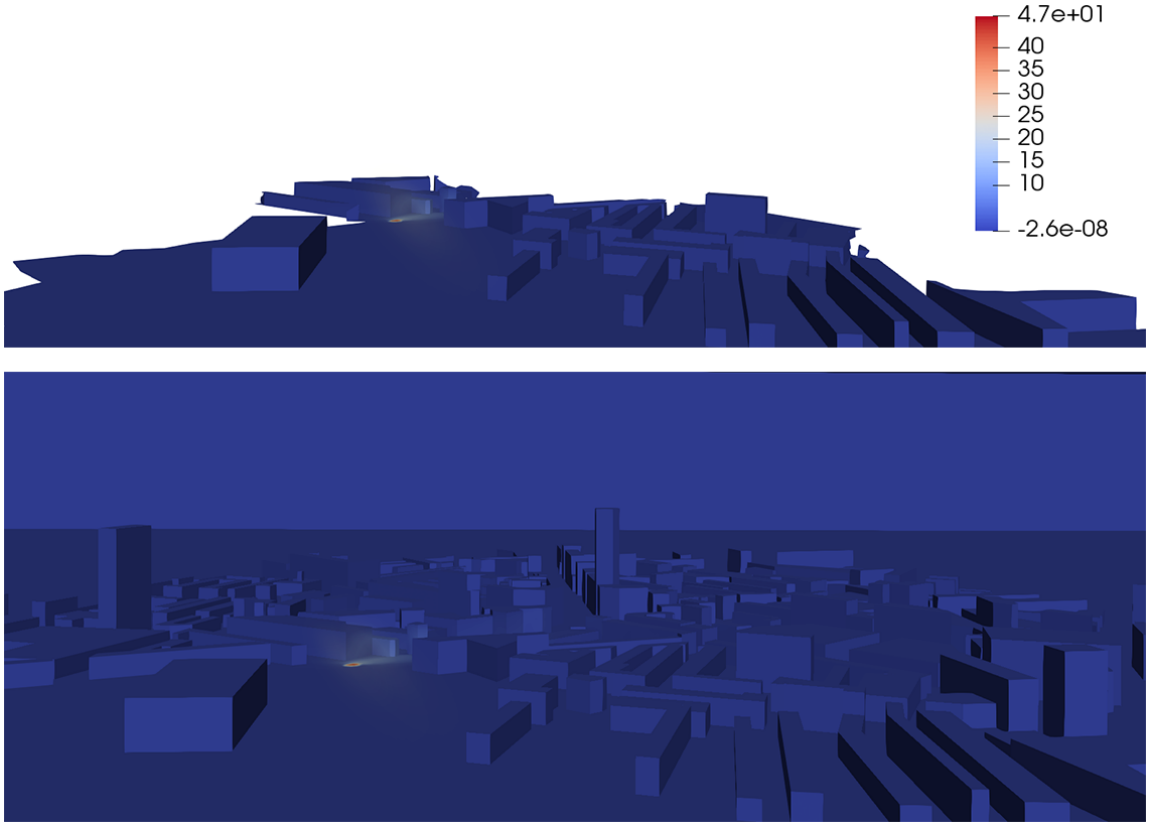
Hence, there are three main alternatives to choose from: using Algorithm 2, which does not use the local kernel approach, Algorithm 4 with an added jitter and Algorithm 4 with absolute values for conditional variances. In the remainder of this chapter, we refer to these as Algorithm 2, Algorithm 4 (Jitter) and Algorithm 4 (Abs.) respectively.

In the following, we discuss identification of relevant sub-domains and sensor placements calculated with each alternative. We validate results by comparing real pollution values to the predictions of the posterior GP, and, moreover, measuring performance in data assimilation. Based on the results we conclude that Algorithm 2 should be used in cases in which precision is of utmost importance, and Algorithm 4 (Jitter) in all other cases.

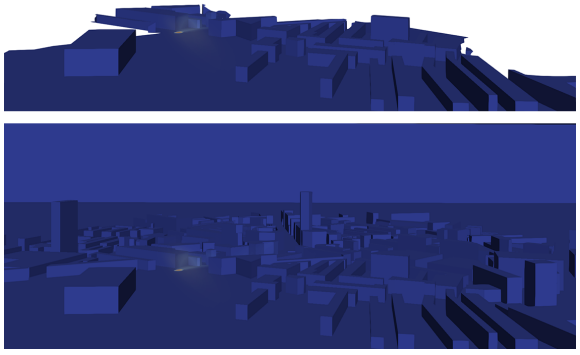
6.1 Relevant Sub-Domains

As described in chapter 4.4, we parallelly compute optimal sensor placement on relevant sub-domains of the LSBU test-side. To identify which sub-domains are of relevance, we analyzed the data numerically and visually in VTK. This analysis was done for all time-steps to ensure that we capture all sub-domains of relevance across

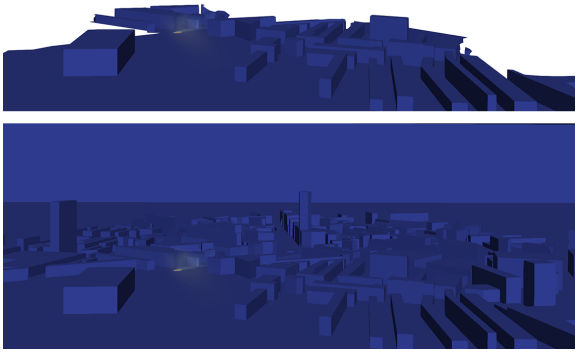
time. In our case, for sensor placements optimized for the pollution measurements of interest, only sub-domain six and eight were identified as relevant. In the following, this analysis is briefly presented.



(a) Time-step 0. Below: all sub-domains. Above: sub-domain six and eight.



(b) Time-step 239. Below: all sub-domains. Above: sub-domain six and eight.

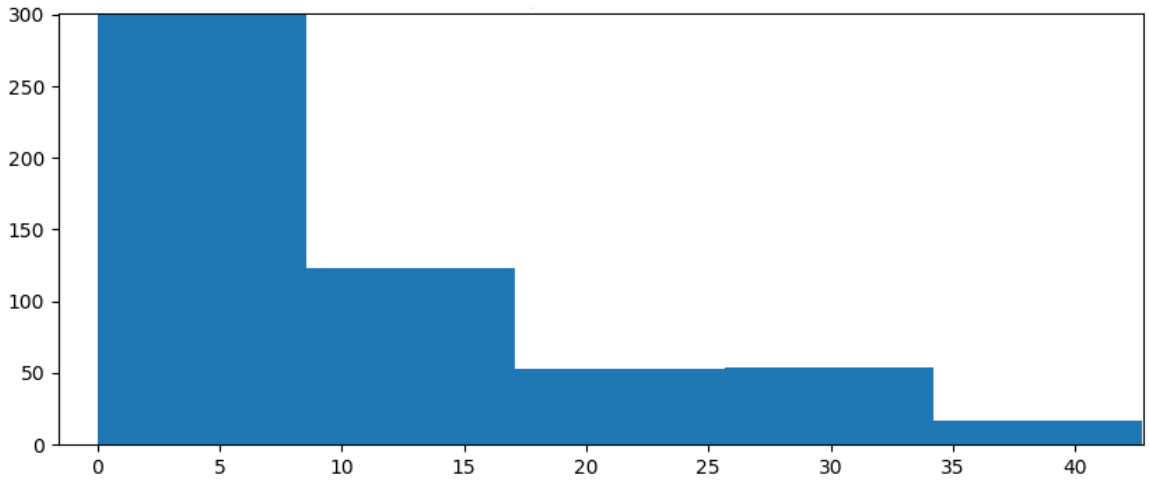


(c) Time-step 521. Below: all sub-domains. Above: sub-domain six and eight.

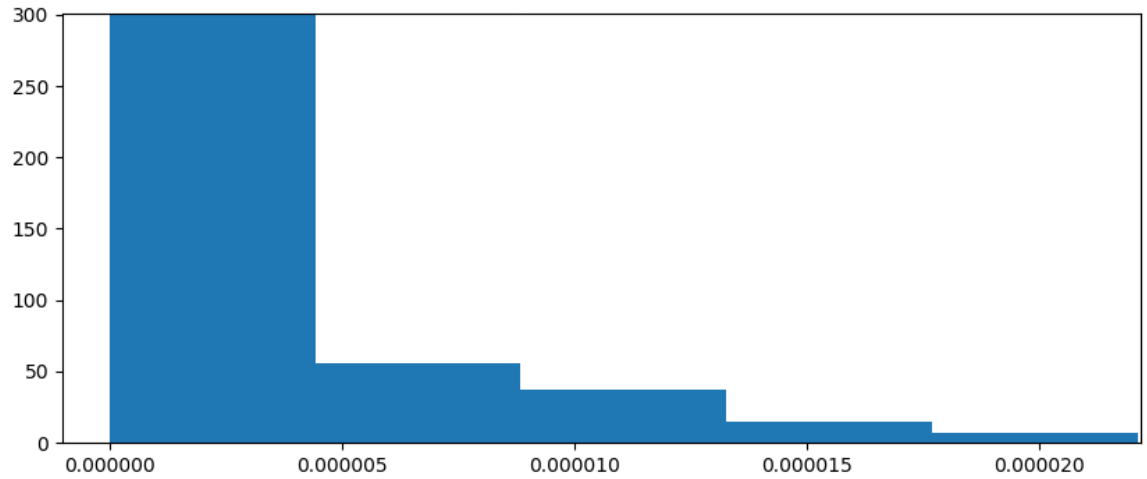
Figure 6.1: VTK visualizations for pollution measurements of interest.

Visually, the only non-zero observations are in the intersection of sub-domain six and eight, which can be seen in figure 6.1. For the pollution measurements of in-

terest, observation of almost all of the 861,959 nodes are approximately zero. This is seen by the dark blue coloring in figure 6.1 which represents observations in the range of $[1e{-}8, 0]$. Only a small part in the intersection of sub-domain six and eight have significantly large observations marked by the illuminated red-point in figure 6.1. Here, the largest observations are in sub-domain eight with many smaller observations in sub-domain six. This is true for all 537 time-steps, as illustrated in figure 6.1 for time-steps 0, 239 and 521.



(a) Sub-domain eight. Maximum value: 4.8516e01.



(b) Sub-domain fifteen. Maximum value: 6.8379e-05.

Figure 6.2: Enlarged histograms for the pollution measurements of interest for sub-domain six and fifteen.

Our numerical analysis confirms that only sub-domains six and eight are of relevance. To ensure that the results of the visual analysis of the computational representations are correct, we analyzed the data numerically. For instance, maximum and minimum values, means and standard deviations were analyzed and compared. An example of such a comparison can be seen in figure 6.2. Here, the histogram

above was calculated for sub-domain eight and the histogram below for sub-domain fifteen. Visibly, the former is much less centered around zero and covers a range from 0 to approximately 49 compared to 0 to approximately $6.8\text{e}{-5}$. Comparisons with all other sub-domains yielded similar results. Hence, it is confirmed that only sub-domains six and eight are of relevance.

6.2 Output and Validation

Sensor placements were computed with all three alternatives: Algorithm 2, Algorithm 4 (Jitter) and Algorithm 4 (Abs.). The algorithms respectively took 99.44 hours, 11.59 hours and 18.43 hours to complete on the servers of the data science institute. The results can be seen in table 6.4, table 6.1, and table 6.5. Furthermore, visualizations of these placements can be seen in figure 6.3.

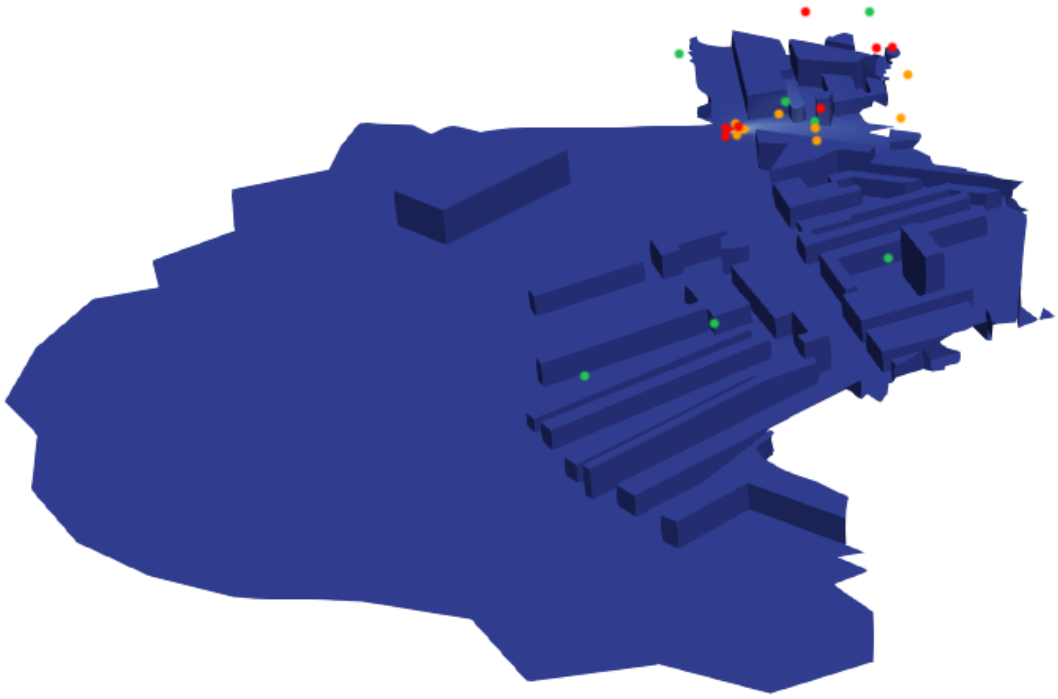


Figure 6.3: Optimal sensor placements in sub-domain six and eight.
Red: Algorithm 2 Orange: Algorithm 4 (Jitter) Green: Algorithm 4 (Abs.)

Validation happens in two ways: First, we use a Gaussian process fitted with our sensors to predict pollution values at each node. We then compare these to the real pollution values to see how far the predictions are from the real values. Second, we check how well our sensor placements are performing when used for data assimilation. This is important due to its heavy use in the MAGIC project and the proven

	Sub-Domain Six				Sub-Domain Eight		
	X	**Y**	**Z**		**X**	**Y**	**Z**
(6.1)	-132.8552	-122.2364	7.8948	**(8.1)**	50.5747	-370.8344	7.7107
(6.2)	-165.7590	-4.0554	26.6328	**(8.2)**	-110.7530	-118.9487	0.0002
(6.3)	-213.0900	-143.3059	29.9016	**(8.3)**	29.3658	-451.7570	0.2000
(6.4)	-141.7301	-8.5550	1.9237	**(8.4)**	66.9609	-243.4015	4.0323

Table 6.1: Sensor placements in sub-domain six and eight calculated with Algorithm 4 (Abs). The results are rounded to four decimal places.

link to the performance in Fluidity. In both instances, performance is compared to the mean performance of 100 sets of randomly placed sensors.

6.2.1 Gaussian Process Validation

This is the main validation methodology that is directly measuring what our algorithms are optimizing for: selecting the most informative k sensor such that pollution in the entire space can most accurately be sensed.

It works by fitting observations at our sensor placements to the prior covariance matrix estimated in the manner described in chapter 3.2. The new, posterior distribution is therefore archived as described in equation 2.13 and equation 2.14 in chapter 2.3. Here, the posterior mean function $m_{post}(X)$ describes the average prediction of pollution at every node given our sensor placements. We compare these estimations to the real values to see how good these predictions are. Furthermore, we compare this performance indication to that of random placement for further insight.

	Sub-Domain Six		Sub-Domain Eight	
	Real Mean	**Estimated Mean**	**Real Mean**	**Estimated Mean**
Algorithm 2.	8.5413e-03	2.3832e-03	2.4662e-01	1.9598e-01
Algorithm 4. (Jitter)	8.5413e-03	1.2502e-02	2.4662e-01	2.2771e-01
Algorithm 4. (Abs.)	8.5413e-03	-2.5045e02	2.4662e-01	-2.6403e02
Random	8.5413e-03	-1.9248e00	2.4662e-01	2.3900e02

Table 6.2: The means of predicted pollution values are compared to the real means for validation purposes. For 'Random' the average result of 100 random placements was taken.

The results of the Gaussian process based validation methodology can be seen in table 6.2. *Real Mean* values are the mean value of the real pollution observations and *Estimated Mean* values the mean of the Gaussian process predictions in the respective sub-domain

The predictions of the average random placement are very bad: For sub-domain six it estimates, on average, a mean value of approximately -1.9248 compared to the real mean of 0.0085. For sub-domain eight, it even estimates a mean of approximately 239.0000 compared to the real mean of 0.2466. Hence, there is a lot of room for improvement.

The estimations of Algorithm 2 and Algorithm 4 (Jitter) are very close to real pollution values. For Algorithm 2, the algorithm is only slightly off for both sub-domains. For Algorithm 4 (Jitter), the algorithm is off by only one polynomial for sub-domain six but very close for sub-domain eight. Hence, both, Algorithm 2 and Algorithm 4 (Jitter), are much better than random placement. This means that the Gaussian processes with only four observations at the positions suggested by the respective algorithm are almost perfect in predicting pollution for the entire sub-domain with over $25,000$ nodes.

However, Algorithm 4 (Abs.) performs much worse. For sub-domain eight, with a three polynomial difference to the real mean, its estimate is approximately equally apart from the real mean as the average random placement. For sub-domain eight, its estimates are even worse than that of random placement. Here, the estimated mean is off by five polynomials.

6.2.2 Data Assimilation

Performance in data assimilation is checked by locally comparing error reductions of the recommended sensor placements to error reductions of random placement. As described in chapter 4.2, the performance of data assimilation depends on sensor placements. Error reductions quantify this differing performance and are therefore a good metric for validation. By comparing the error reductions of our sensor placements to the error reductions of random sensor placements we can see how much more information our sensor placements retrain compared to a random placement.

Error reductions are calculated by taking the difference between the background error and the data assimilation error. The background error is calculated in the following way: a random time-step between 0 and 300 is chosen as the background. This represents the base pollution measurements. Another time-step much later than our background is chosen as the observations file. This represents the real measures in the future that we normally have no access to. The difference in pollution observations between the observation file and the background is the background error. This error is what we would have had if we did not use data assimilation. The data assimilation error is obtained by assimilating the background to the time-step of the observation file. The difference between this value and our background represents is the data assimilation error. Finally, the error reduction is obtained by subtracting this value from the background error. In other words, we calculate how much closer we get to the real observations by using data assimilation at these specific sensor placements.

The performance of the sensor placements in data assimilation is evaluated in table 6.3. Error reductions of the respective algorithm can be calculated by subtracting *DA Error* from the *Background Error*. Hence, the lower *DA Error* is in comparison to *Background error*, the higher the error reduction and the better our sensors are performing for this specific application.

	Sub-Domain Six		**Sub-Domain Eight**	
	Background Error	**DA Error**	**Background Error**	**DA Error**
Algorithm 2.	7.5643e-01	1.6574e-05	2.2551e-01	6.8829e-02
Algorithm 4. (Jitter)	1.4444e-01	7.9082e-04	1.7325e-01	1.1862e-01
Algorithm 4. (Abs.)	1.6891e-05	1.9384e-06	1.2648e-03	2.4650e-06
Random	8.4021e01	2.0700e00	7.5054e01	1.0173e01

Table 6.3: Error reductions for sensor placements calculated with the labelled algorithm. For 'Random' the average result of 100 random placements was taken.

The average random placement has almost negligible error reductions. For sub-domain six, it data assimilation lowers errors by one polynomial and for sub-domain eight it does not even do that. Just like with the Gaussian process validation methodology, this means that there is plenty of room for improvement.

As can be seen, Algorithm 2 and Algorithm 4 (Jitter) show pretty good results. The former has error reductions of four and one polynomials for sub-domain six and eight respectively. The latter has error reductions of three polynomials for sub-domain six but almost no error reduction for sub-domain eight. Hence, both algorithms perform strictly better than random placement.

Algorithm 4 (Abs.), on the other hand, shows mixed results. For sub-domain six, error reductions are approximately equal to that of random placement. Hence, in sub-domain six, Algorithm 4 (Abs.) performs much worse than Algorithm 2 and Algorithm 4 (Jitter). However, for sub-domain eight, it is by far the best performing algorithm with error reductions of three polynomials.

6.2.3 Recommendations

Based on these results, we recommend using either Algorithm 2 or Algorithm 4 (Jitter) depending on whether accuracy or computational cost is more important. Their recommended sensor placements are illustrated in figure 6.4 and can be seen in table 6.4 and table 6.5.

Due to bad validation results, we believe that Algorithm 4 (Abs.) should not be used under any circumstances. Only in data assimilation it performed better in sub-domain eight than the other two alternatives. This application, however, is not what our algorithm has been optimizing for and is merely checked because of its

Figure 6.4: Final recommendations for sensor placements in the LSBU test-side.
Red: Algorithm 2 Orange: Algorithm 4 (Jitter)

importance within the MAGIC project. Furthermore, as data assimilation errors are computed locally and as Algorithm 4 (Abs.) does not have the same performance in sub-domain six, this performance could be due to chance. The fact that it is performing much worse than random placement in the GP validation methodology and the fact that no sensors were placed in the height of the pollution as visible in figure 6.3, suggests to us that taking absolute values when computing conditional variance's is fundamentally breaking the greedy algorithm suggested by Krause, Singh, and Guestrin [5].

	Sub-Domain Six				**Sub-Domain Eight**		
	X	**Y**	**Z**		**X**	**Y**	**Z**
(6.1)	-90.8344	-133.3382	17.8474	**(8.1)**	-213.5999	-297.7235	11.8956
(6.2)	-199.4697	-34.5250	28.3916	**(8.2)**	-144.5949	-166.6230	0.7492
(6.3)	-115.5043	-37.6516	24.0135	**(8.3)**	-138.6805	-163.3081	2.9427
(6.4)	-144.5582	-21.3671	10.9458	**(8.4)**	-144.1859	-164.2906	0.2000

Table 6.4: Sensor placements in sub-domain six and eight calculated with Algorithm 2. The results are rounded to four decimal places.

	Sub-Domain Six				Sub-Domain Eight		
	X	**Y**	**Z**		**X**	**Y**	**Z**
(6.1)	-102.7970	-9.6625	45.4371	**(8.1)**	-139.2968	-165.1826	1.7112
(6.2)	-132.1179	-134.8961	6.2715	**(8.2)**	-77.5144	-141.0195	0.2000
(6.3)	-96.3141	-137.5103	4.5760	**(8.3)**	-145.0544	-164.5704	0.2000
(6.4)	-55.3314	-79.1452	0.2000	**(8.4)**	-140.7513	-159.8805	0.2000

Table 6.5: Sensor placements in sub-domain six and eight calculated with Algorithm 4 (Jitter). The results are rounded to four decimal places.

Algorithm 4 (Jitter) should be used when time or computational costs is an important factor. Computing sensor placements with it only took 11.59 hours compared to the 99.44 hours that Algorithm 2 took to complete. This means that Algorithm 4 (Jitter) is almost nine times faster than its alternative that does not use the local kernel approach. Furthermore, surprisingly, adding jitter and only considering covariances larger than $\epsilon = 1\mathrm{e}{-10}$ does not lower precision by much: for both validation methodologies, Algorithm 2 is at most better by one polynomial. Hence, we suggest using Algorithm 4 (Jitter) in almost all cases in which precision is not of utmost importance. For instance, this algorithm should be used when continuously running the algorithm to always pick the best k out of many sensors.

Algorithm 2 should be used when precision is very important. It consistently had better validation results than Algorithm 4 (Jitter) but took much longer to compute. While these validation results are admittedly not a lot better, this small precision advantage might be worth the computational costs for some use-cases. For example, when physically placing new sensors, Algorithm 2 should be used as these cannot be changed afterward.

Finally, it might be advisable to drop one of the sensors at the intersection of the sub-domains. As stated in chapter 4.4, sub-domains in the LSBU test-side are overlapping in the intersections. Hence, due to the penalization of nodes close to already placed sensors described in chapter 2, it is unlikely that our algorithm would have placed sensors so close together. Still, this is only a qualitative judgment that cannot be concluded with absolute certainty.

Chapter 7

Conclusion and Future Work

This thesis successfully developed and implemented a methodology and code to optimally place sensors using simulation data. Considering the requirements of the MAGIC project, it can be used in two ways: First, it can be used to place new sensors, for instance during deployment of the final computational system developed by the MAGIC project. Second, in test-sides in which sensors were already placed, it can be used to continuously select the best k sensors amongst them. parallelization and other techniques were used to reduce computational costs to the extent at which sensor placements can be done very large domains. As an example, the sensor placement in the LSBU test-side that has $861,959$ nodes was solved with very good validation results.

In the following, we will briefly summarize the achievements of this thesis. Then, we will conclude with a few ideas of future work that might further improve sensor placement or extend the work done to other applications.

7.1 Summary of Achievements

The issue of scale was solved by efficiently selecting relevant data and parallelizing computations. Our aim was to optimize sensor placements according to pollution values in the LSBU test-side that has $861,959$ nodes. Considering that our algorithm uses Gaussian processes whose practical limit is said to be around $10,000$ nodes, this is a seemingly impossible task [13]. We solved this by only considering relevant sub-domains and parallelizing computations on them.

We wrote a method to ensure monotonicity in the LSBU test-side. As mentioned in chapter 2, monotonicity is a necessary condition when using the algorithms suggested by Krause, Singh, and Guestrin in spaces with over 2000 nodes [5]. Hence, we wrote a method in the MagicProject class that checks whether this is ensured. Fortunately, we do not have any issues in the LSBU test-side. This is especially important considering the possible future usage of the sensor placement algorithm in different test-sides.

The non-stationary and non-isotropic prior covariance matrix necessary for optimal results was obtained by taking the sample covariance matrix across all time-steps in the simulation data. Normally, estimating such a covariance matrix is very difficult. However, as we were using simulation data, we were able to take the sample covariance matrix across the 537 time-steps to get a very good estimate of the real covariance matrix.

We found that solving numerical stability issues by adding a jitter to the covariance matrix is much better than taking absolute values of conditional variances. Due to the sparse nature of the data, our covariance matrix was very centered around zero. This was leading to leading to negative conditional variances that for obvious reasons should not occur. We tested two alternatives solutions and found that adding a jitter is far better than taking absolute values of the conditional variances.

In addition to effectively implementing the three sensor placements algorithms suggested by Krause, Singh, and Guestrin, we developed a fourth, much more efficient algorithm [5]. This fourth algorithm combines the priority queue approach of Algorithm 2 with the local kernel approach of Algorithm 3. Hence, it is just as precise as Algorithm 3, but much faster. Further, the three sensor placement algorithms were optimized with Python's native list comprehension and NumPy's efficient matrix operations.

Besides optimizing performance, we developed an intuitive API that can be used for easy sensor placement. Instead of a confusing script that requires lengthy data preparation of the raw simulation data, our API can be important akin to a library and simply called with a few parameters specifying things such as the sub-domain of interest and the number of sensors to be placed. It is ensured that, or sensor placements outside of the MAGIC project, the sensor placement algorithms can be called directly.

We developed a comprehensive validation methodology that measures the performance of sensor placements by calculating the mean prediction error of the posterior GP. It works by fitting sensor placements to the prior covariance matrix to obtain a posterior mean and covariance function. These can then be used to predict pollution values at every node in the domain of interest. The mean of this prediction is compared to the real mean of pollution values to see how far predictions of the Gaussian process are off.

The sensor placement optimization in the LSBU test-side was archived with very good validation results. Here, we found that Algorithm 2 is slightly more precise than Algorithm 4 (Jitter) but takes approximately nine times longer to complete. Hence, we conclude that unless the precision is of utmost importance, Algorithm 4 (Jitter), which only took 11.59 hours to complete and is therefore exceptionally fast, should be used.

7.2 Future Work

Future work building on this thesis can be separated in two broad categories: First, there is still work to be done to further improve the sensor placement methodology and apply it to other sub-domains in the LSBU test-side. Second, there are several ways the methodology can be used for other applications unrelated to sensor placement. In the following, we will briefly elaborate on both.

7.2.1 Sensor Placement

In this thesis, sensor placement was done for only one part of the pollution measurement in the LSBU test-side. Naturally, if pollution is to be observed within the whole test-side, the algorithm has to be run for the other six parts as well. However, the API written by us takes care of most of the work. For all other things such as the identification of relevant sub-domains, we have written scripts and developed sound methodology as described in this thesis.

There might be a better epsilon value in the local kernel approach. As stated in chapter 5, we used an epsilon value of $1e{-}10$ when running Algorithm 3 and Algorithm 4. We took this value due to the sparse nature of the covariance matrix. Our tests confirmed that with this value, the covariance matrix decreases by 92% with no significant loss in precision. However, this value is based on estimations and has no sound methodology behind it. Hence, we believe that Algorithm 4 (Jitter) can yield even better results if the ϵ value is optimally chosen, for instance with Bayesian optimization.

7.2.2 Other Applications

Besides sensor placement, the algorithms and overall methodology of this thesis can be used in several other applications where selecting the most expressive data of a very large set is important. For instance, in evolutionary algorithms, generating the first initial population is generally done randomly. The sensor selection algorithm could be used instead to ensure the best possible variety in optimal solutions.

Bibliography

[1] "Sensor placements in the lsbu test-side," 2019, unpublished work done within the MAGIC project. Available upon request. pages vii, 3

[2] A. O'Hagan, "Curve fitting and optimal design for prediction," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 40, no. 1, pp. 1–24, 1978. pages vii, 13

[3] A. Krause, A. Singh, and C. Guestrin, "Near-optimal sensor placements in gaussian processes presentation slides," 2008. [Online]. Available: https://slideplayer.com/slide/4626206/ pages vii, 13, 23

[4] M. I. Mead, O. A. M. Popoola, G. B. Stewart, P. Landshoff, M. Calleja, M. Hayes, J. J. Baldovi, M. W. McLeod, T. F. Hodgson, J. Dicks, A. Lewis, J. Cohen, R. Baron, J. R. Saffell, and R. L. Jones, "The use of electrochemical sensors for monitoring urban air quality in low-cost, high-density networks," *Atmospheric Environment*, vol. 70, pp. 186–203, MAY 2013, pT: J; NR: 25; TC: 212; J9: ATMOS ENVIRON; PG: 18; GA: 120GK; UT: WOS:000317158600019. pages 1

[5] A. Krause, A. Singh, and C. Guestrin, "Near-optimal sensor placements in gaussian processes: Theory, efficient algorithms and empirical studies," *Journal of Machine Learning Research*, vol. 9, no. Feb, pp. 235–284, 2008. pages 2, 3, 5, 6, 7, 14, 15, 16, 20, 22, 37, 39, 40

[6] C. A. Short, J. Song, L. Mottet, S. Chen, J. Wu, and J. Ge, "Challenges in the low-carbon adaptation of china's apartment towers," *Building Research and Information*, vol. 46, no. 8, pp. 899–930, 2018, pT: J; NR: 109; TC: 0; J9: BUILD RES INF; SI: SI; PG: 32; GA: GS6YP; UT: WOS:000443846300007. pages 2

[7] D. S. Hochbaum and W. Maass, "Approximation schemes for covering and packing problems in image processing and vlsi," *Journal of the ACM (JACM)*, vol. 32, no. 1, pp. 130–136, 1985. pages 5

[8] H. González-Banos, "A randomized art-gallery algorithm for sensor placement," in *Proceedings of the seventeenth annual symposium on Computational geometry*. ACM, 2001, pp. 232–240. pages 5

[9] X. Bai, S. Kumar, D. Xuan, Z. Yun, and T. H. Lai, "Deploying wireless sensors to achieve both coverage and connectivity," in *Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*. ACM, 2006, pp. 131–142. pages 5

[10] K. Chakrabarty and S. S. Iyengar, "Sensor placement in distributed sensor networks using a coding theory framework," in *Proceedings. 2001 IEEE International Symposium on Information Theory (IEEE Cat. No. 01CH37252)*. IEEE, 2001, p. 157. pages 5

[11] P. E. Latham and Y. Roudi, "Mutual information," 2009. [Online]. Available: http://www.scholarpedia.org/article/Mutual_information pages 7

[12] K. Worden and A. P. Burrows, "Optimal sensor placement for fault detection," *Engineering Structures*, vol. 23, no. 8, pp. 885–901, 2001. pages 7

[13] M. P. Deisenroth, A. A. Faisal, and C. S. Ong, *Mathematics for Machine Learning*. Cambridge University Press, 2019, [Online] Available: https://mml-book.com. pages 8, 9, 10, 11, 25, 39

[14] M. P. Deisenroth, "Linear regression," 2018. [Online]. Available: https://tinyurl.com/yyxwt5t9 pages 8

[15] ——, "Bayesian optimization," 2019, lecture slides from "Probabilistic Inference (CO-493)". Available upon request. pages 9, 11

[16] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012. pages 9

[17] M. P. Deisenroth, Y. Luo, and M. van der Wilk, "A practical guide to gaussian processes." [Online]. Available: https://drafts.distill.pub/gp/ pages 10

[18] C.-W. Ko, J. Lee, and M. Queyranne, "An exact algorithm for maximum entropy sampling," *Operations research*, vol. 43, no. 4, pp. 684–691, 1995. pages 13

[19] N. Cressie, *Statistics for spatial data*. Wiley-Blackwell, 1992, vol. 4, no. 5. pages 13

[20] M. C. Shewry and H. P. Wynn, "Maximum entropy sampling," *Journal of applied statistics*, vol. 14, no. 2, pp. 165–170, 1987. pages 13

[21] N. Ramakrishnan, C. Bailey-Kellogg, S. Tadepalli, and V. N. Pandey, "Gaussian processes for active data mining of spatial aggregates," in *Proceedings of the 2005 SIAM International Conference on Data Mining*. SIAM, 2005, pp. 427–438. pages 13

[22] W. F. Caselton and J. V. Zidek, "Optimal monitoring network designs," *Statistics & Probability Letters*, vol. 2, no. 4, pp. 223–227, 1984. pages 14

[23] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—i," *Mathematical programming*, vol. 14, no. 1, pp. 265–294, 1978. pages 14

[24] R. Arcucci, L. Mottet, C. Pain, and Y.-K. Guo, "Optimal reduced space for variational data assimilation," *Journal of Computational Physics*, vol. 379, pp. 51–69, 2019. pages 22

Glossary

Algorithm 2 Sensor placement algorithm that uses uses priority queues for faster computations.

Algorithm 4 (Jitter) Sensor placement algorithm that uses both: priority queues and the local kernel approach for faster computations. Numerical stability issues are solved by adding a jitter $I * x$ to the covariance matrix.

Algorithm 4 (Abs.) Sensor placement algorithm that uses both: priority queues and the local kernel approach for faster computations. Numerical stability issues are solved by taking absolute values when calculating conditional variances.

SensorPlacement Class that contains all sensor placement algorithms and necessary helper methods such as ones computing conditonal variances.

MagicProject Class that provides intuitive API for sensor placement in the MAGIC project. All necessary preliminary calculations for SensorPlacement, such as the estimation of covariance matrices, are taken care by it.

List of Abbreviations

BO Bayesian Optimization

DA Data Assimilation

DSI Data Science Institute

GP Gaussian Process

HVAC Heating, Ventilation and Cooling

LSBU10 Test-side at the London South Bank University split in 10 sub-domains

LSBU32 Test-side at the London South Bank University split in 32 sub-domains

LSBU London South Bank University

MAGIC Managing Air for Green Inner Cities

RBF Radial Basis Function

TSVD Truncated Singular Value Decomposition

UML Unified Modeling Language

VTK Visualization Toolkit

Appendix A

Legal and Ethical Considerations

The project is done under the umbrella of the MAGIC project which aims to fight global warming by introducing more natural ventilation to cities. As such, we are not aware of possible negative impacts that either the MAGIC project as a whole or the sensor placement code developed by this thesis could have. The data used is purely generated from simulations in the physics simulation software Fluidity and hence does not violate any GDPR or other regulations for personal or otherwise private data.

The work done in this thesis is solving the basic task of sensor placement. Hence, technically the results could be used by the military or another organization for sensor placement or a variety of other purposes. However, a possible argumentative chain leading to a negative outcome can be generated for any research done at this basic level. In other words: the probability that the work is done with this research will lead to anything significantly negative through dual or misuse is low enough for us to consider it non-existent.

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?		✓
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from "Human Embryos/Foetuses" i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		✓
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		✓
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora/protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?		✓
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics – e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		✓
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		✓
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		✓
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		✓
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		✓
SECTION 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		✓
Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓
SECTION 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		✓

Table A.1: Ethical and legal considerations checklist

Appendix B

Optimal Number of Sensors

Initially, we tried using our validation methodologies to estimate the optimal number of sensors for a sub-domain in LSBU32 and specifically for sub-domain six and eight. The idea was to keep placing sensors in each sub-domain to see whether there is a pattern in the GP validation errors or DA error reductions. Hence, Algorithm 4 (Jitter) was used to place six additional sensors on top of the four that were already placed. The coordinates of these can be seen in table B.1.

Unfortunately, our analysis does not show any pattern from which such a decision could be made. Hence, we suggest deciding on the number of sensors based on external factors such as budget constraints. However, judging from the recommended sensor placements, it seems that four sensors might be close to optimal for a sub-domain in LSBU32.

	Sub-Domain Six				**Sub-Domain Eight**		
	X	**Y**	**Z**		**X**	**Y**	**Z**
(6.1)	-102.7970	-9.6625	45.4371	**(8.1)**	-139.2968	-165.1826	1.7112
(6.2)	-132.1179	-134.8961	6.2715	**(8.2)**	-77.5144	-141.0195	0.2000
(6.3)	-96.3141	-137.5103	4.5760	**(8.3)**	-145.0544	-164.5704	0.2000
(6.4)	-55.3314	-79.1452	0.2000	**(8.4)**	-140.7513	-159.8805	0.2000
(6.5)	-107.1744	-134.5433	6.2976	**(8.5)**	-141.3328	-164.8661	1.9346
(6.6)	-138.3212	-148.1351	0.2000	**(8.6)**	-47.2486	-92.5293	0.2000
(6.7)	-110.7530	-118.9487	19.5000	**(8.7)**	-134.7249	-164.0222	0.6720
(6.8)	-103.7245	-124.2502	0.2000	**(8.8)**	-33.1216	-93.7006	8.4223
(6.9)	-94.2940	-138.7546	0.2000	**(8.9)**	-138.0821	-158.8279	0.2000
(6.10)	-70.4442	-97.5981	10.7281	**(8.10)**	-105.1915	-130.5747	17.2595

Table B.1: Extra sensor placements in sub-domain six and eight calculated with Algorithm 4 (Jitter). The results are rounded to four decimal places.

Appendix C

Software

During the scope of this project, a few software beyond common ones such as Python were used. Specifically, the data used in this project stems from Fluidity, a fluid dynamics simulation software. Furthermore, the visual analysis was done in ParaView.

C.1 Fluidity

Fluidity is an open-source fluid dynamics software that is used for various simulation purposes. While it was not directly used within this project, all data used in this thesis was gathered from simulations done in it. Hence, in a sense Fluidity represents the heart of our project. Furthermore, the Python interface of Fluidity, including for instance *vtktools*, was used heavily to process and transform the data.

C.2 ParaView

ParaView is an open-source software used for scientific visualizations with VTK. In the course of this thesis, ParaView was routinely used for visual analysis, for instance, the one described in chapter 6. Furthermore, all plots illustrating the LSBU test-side were done in ParaView.

Appendix D

Hardware

As mentioned, our sensor placement algorithms are relatively computationally expensive. Hence, to identify the correct methodology, fine-tune our code, compute the final sensor placements and validate all placements, many computationally expensive processes were run. For these, we relied on the servers of the DSI and Google's cloud computing service.

D.1 Data Science Institute Computing Servers

Most computations including the time estimations were done on the servers of the DSI. With over 50 readily available CPU's these powerful resources were very helpful for our thesis. Unfortunately, no GPU's were available in the DSI servers we had access to. These are especially well suited for expensive matrix operations and would have likely considerably lowered the time our algorithms took.

D.2 Google Cloud

We used Google Cloud during times were the DSI servers were unavailable due to relocation. Surprisingly, it was fairly cheap to rent similarly powerful machines as the DSI servers and even GPU's were readily available. Hence, we did not exceed the $\$300$ trial credit for our computations.

Appendix E

User Guide

For sensor placement outside the MAGIC project, the API of the SensorPlacement class can simply be called with the necessary parameters, such as the prior covariance matrix and the number of sensors to be placed. For sensor placement within the MAGIC project, however, an API was written to further ease this process. Furthermore, with a few alternations in the MagicProject class, this can also be used for sensor placement outside the MAGIC project. The process of using this API is outlined in the following.

The simulation data has to be converted into CSV-files and placed correctly in the file structure that is described in chapter 5. For VTU-files, the conversion is taken care of by the *data_preparation.py* script, which merely needs to be called with the file path. Optionally, this script also normalizes and standardizes observations. After conversion, the CSV-Files have to be placed into the relevant folder specifying the sub-domain that is of interest. For work outside the LSBU test-side, this can be placed into the folder called *subdomain_None*.

After these preparations, the MagicProject API can be called for description, validation and sensor placements. For sensor placement, for example, the API only needs a few parameters specifying, for example, the sub-domains of interest. If the placement is outside the LSBU test-side, this can be substituted with a $None$. Furthermore, there is the option of specifying already placed sensors with an array of indices. An example of how the API can be used can be seen in the following:

```
""" Description """
MagicProject.plotHistogram(subdomain=8, number_bins=5)
MagicProject.describeData(subdomain=8)

""" Validation """
indLoc = np.loadtxt('./solutions/subdomain_8/validation_format/num_sens/7sens.txt', dtype=int)
print(MagicProject.validation(subdomain=8, A=indLoc))

""" Calling functions to optimize sensor placement """
t0 = time()
A = MagicProject.parallelPlacement(subdomains=[1, 2], k=4, algorithm=4)
t1 = time()
print('The parallel placement algorithm takes ', (t1-t0), 'seconds')
```